//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

/\*

\* RUNTIME\_ARGUMENTS is a simple wrapper around the variadic calling convention

\* used by JavaScript functions. It is a low level macro that does not try to

\* differentiate between script usable Vars and runtime data structures.

\* To be able to access only script usable args use the ARGUMENTS macro instead.

\*/

#define RUNTIME\_ARGUMENTS(n, s) \

va\_list argptr; \

va\_start(argptr, s); \

Js::Arguments n(s, (Js::Var \*)argptr);

#define ARGUMENTS(n, s) \

va\_list argptr; \

va\_start(argptr, s); \

Js::ArgumentReader n(&s, (Js::Var \*)argptr);

namespace Js

{

struct Arguments

{

public:

Arguments(CallInfo callInfo, Var \*values) : Info(callInfo), Values(values) {}

Arguments(VirtualTableInfoCtorEnum v) : Info(v) {}

Var operator [](int idxArg) { return const\_cast<Var>(static\_cast<const Arguments&>(\*this)[idxArg]); }

const Var operator [](int idxArg) const

{

AssertMsg((idxArg < (int)Info.Count) && (idxArg >= 0), "Ensure a valid argument index");

return Values[idxArg];

}

CallInfo Info;

Var\* Values;

static uint32 GetCallInfoOffset() { return offsetof(Arguments, Info); }

static uint32 GetValuesOffset() { return offsetof(Arguments, Values); }

};

struct ArgumentReader : public Arguments

{

ArgumentReader(CallInfo \*callInfo, Var \*values)

: Arguments(\*callInfo, values)

{

AssertMsg(!(Info.Flags & Js::CallFlags\_NewTarget) || (Info.Flags & Js::CallFlags\_ExtraArg), "NewTarget flag must be used together with ExtraArg.");

if (Info.Flags & Js::CallFlags\_ExtraArg)

{

// If "calling eval" is set, then the last param is the frame display, which only

// the eval built-in should see.

Assert(Info.Count > 0);

// The local version should be consistent. On the other hand, lots of code throughout

// jscript uses the callInfo from stack to get argument list etc. We'll need

// to change all the caller to be aware of the id or somehow make sure they don't use

// the stack version. Both seem risky. It would be safer and more robust to just

// change the stack version.

Info.Flags = (CallFlags)(Info.Flags & ~Js::CallFlags\_ExtraArg);

Info.Count--;

callInfo->Flags = (CallFlags)(callInfo->Flags & ~Js::CallFlags\_ExtraArg);

callInfo->Count--;

}

}

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

// Portions of this file are copyright 2014 Mozilla Foundation, available under the Apache 2.0 license.

//-------------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------------

// Copyright 2014 Mozilla Foundation

//

// Licensed under the Apache License, Version 2.0 (the "License");

// you may not use this file except in compliance with the License.

// You may obtain a copy of the License at

//

// http ://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing, software

// distributed under the License is distributed on an "AS IS" BASIS,

// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

// See the License for the specific language governing permissions and

// limitations under the License.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

#include "ByteCode\Symbol.h"

#include "ByteCode\FuncInfo.h"

#include "ByteCode\ByteCodeWriter.h"

#include "ByteCode\ByteCodeGenerator.h"

namespace Js

{

bool

AsmJSCompiler::CheckIdentifier(AsmJsModuleCompiler &m, ParseNode \*usepn, PropertyName name)

{

if (name == m.GetParser()->names()->arguments || name == m.GetParser()->names()->eval)

{

return m.FailName(usepn, L"'%s' is not an allowed identifier", name);

}

return true;

}

bool

AsmJSCompiler::CheckModuleLevelName(AsmJsModuleCompiler &m, ParseNode \*usepn, PropertyName name)

{

if (!CheckIdentifier(m, usepn, name))

{

return false;

}

if (name == m.GetModuleFunctionName())

{

return m.FailName(usepn, L"duplicate name '%s' not allowed", name);

}

//Check for all the duplicates here.

return true;

}

bool

AsmJSCompiler::CheckFunctionHead(AsmJsModuleCompiler &m, ParseNode \*fn, bool isGlobal /\*= true\*/)

{

PnFnc fnc = fn->sxFnc;

if (!fnc.IsSimpleParameterList())

{

return m.Fail(fn, L"default & rest args not allowed");

}

if (fnc.IsStaticMember())

{

return m.Fail(fn, L"static functions are not allowed");

}

if (fnc.IsGenerator())

{

return m.Fail(fn, L"generator functions are not allowed");

}

if (fnc.IsLambda())

{

return m.Fail(fn, L"lambda functions are not allowed");

}

if (!isGlobal && fnc.nestedCount != 0)

{

return m.Fail(fn, L"closure functions are not allowed");

}

if (fnc.HasDefaultArguments())

{

return m.Fail(fn, L"default arguments not allowed");

}

return true;

}

bool AsmJSCompiler::CheckTypeAnnotation( AsmJsModuleCompiler &m, ParseNode \*coercionNode, AsmJSCoercion \*coercion,

ParseNode \*\*coercedExpr /\*= nullptr \*/)

{

switch( coercionNode->nop )

{

case knopRsh:

case knopLsh:

case knopXor:

case knopAnd:

case knopOr: {

ParseNode \*rhs = ParserWrapper::GetBinaryRight( coercionNode );

\*coercion = AsmJS\_ToInt32;

if( coercedExpr )

{

if( rhs->nop == knopInt && rhs->sxInt.lw == 0 )

{

if( rhs->nop == knopAnd )

{

// X & 0 == 0;

\*coercedExpr = rhs;

}

else

{

// (X|0) == (X^0) == (X<<0) == (X>>0) == X

\*coercedExpr = ParserWrapper::GetBinaryLeft( coercionNode );

}

}

else

{

\*coercedExpr = coercionNode;

}

}

return true;

}

case knopPos: {

\*coercion = AsmJS\_ToNumber;

if( coercedExpr )

{

\*coercedExpr = ParserWrapper::GetUnaryNode( coercionNode );

}

return true;

}

case knopCall: {

ParseNode\* target;

AsmJsFunctionDeclaration\* sym;

AsmJsMathFunction\* mathSym;

AsmJsSIMDFunction\* simdSym;

target = coercionNode->sxCall.pnodeTarget;

if (!target || target->nop != knopName)

{

return m.Fail(coercionNode, L"Call must be of the form id(...)");

}

simdSym = m.LookupSimdTypeCheck(target->name());

// var x = f4(ffi.field)

if (simdSym)

{

if (coercionNode->sxCall.argCount == simdSym->GetArgCount())

{

switch (simdSym->GetSimdBuiltInFunction())

{

case AsmJsSIMDBuiltin\_int32x4\_check:

\*coercion = AsmJS\_Int32x4;

break;

case AsmJsSIMDBuiltin\_float32x4\_check:

\*coercion = AsmJS\_Float32x4;

break;

case AsmJsSIMDBuiltin\_float64x2\_check:

\*coercion = AsmJS\_Float64x2;

break;

default:

Assert(UNREACHED);

}

if (coercedExpr)

{

\*coercedExpr = coercionNode->sxCall.pnodeArgs;

}

return true;

}

else

{

return m.Fail(coercionNode, L"Invalid SIMD coercion");

}

}

// not a SIMD coercion, fall through

\*coercion = AsmJS\_FRound;

sym = m.LookupFunction(target->name());

mathSym = (AsmJsMathFunction\*)sym;

if (!(mathSym && mathSym->GetMathBuiltInFunction() == AsmJSMathBuiltin\_fround))

{

return m.Fail( coercionNode, L"call must be to fround coercion" );

}

if( coercedExpr )

{

\*coercedExpr = coercionNode->sxCall.pnodeArgs;

}

return true;

}

case knopInt:{

\*coercion = AsmJS\_ToInt32;

if( coercedExpr )

{

\*coercedExpr = coercionNode;

}

return true;

}

case knopFlt:{

if (ParserWrapper::IsMinInt(coercionNode))

{

\*coercion = AsmJS\_ToInt32;

}

else if (coercionNode->sxFlt.maybeInt)

{

return m.Fail(coercionNode, L"Integer literal in return must be in range [-2^31, 2^31)");

}

else

{

\*coercion = AsmJS\_ToNumber;

}

if( coercedExpr )

{

\*coercedExpr = coercionNode ;

}

return true;

}

case knopName:{

// in this case we are returning a constant var from the global scope

AsmJsSymbol \* constSymSource = m.LookupIdentifier(coercionNode->name());

if (!constSymSource)

{

return m.Fail(coercionNode, L"Identifier not globally declared");

}

AsmJsVar \* constSrc = constSymSource->Cast<AsmJsVar>();

if (constSymSource->GetSymbolType() != AsmJsSymbol::Variable || constSrc->isMutable())

{

return m.Fail(coercionNode, L"Unannotated variables must be constant");

}

if (constSrc->GetType().isSigned())

{

\*coercion = AsmJS\_ToInt32;

}

else if (constSrc->GetType().isDouble())

{

\*coercion = AsmJS\_ToNumber;

}

else

{

Assert(constSrc->GetType().isFloat());

\*coercion = AsmJS\_FRound;

}

if (coercedExpr)

{

\*coercedExpr = coercionNode;

}

return true;

}

default:;

}

return m.Fail( coercionNode, L"must be of the form +x, fround(x) or x|0" );

}

bool

AsmJSCompiler::CheckModuleArgument(AsmJsModuleCompiler &m, ParseNode \*arg, PropertyName \*name, AsmJsModuleArg::ArgType type)

{

if (!ParserWrapper::IsDefinition(arg))

{

return m.Fail(arg, L"duplicate argument name not allowed");

}

if (!CheckIdentifier(m, arg, arg->name()))

{

return false;

}

\*name = arg->name();

m.GetByteCodeGenerator()->AssignPropertyId(\*name);

AsmJsModuleArg \* moduleArg = Anew(m.GetAllocator(), AsmJsModuleArg, \*name, type);

if (!m.DefineIdentifier(\*name, moduleArg))

{

return m.Fail(arg, L"duplicate argument name not allowed");

}

if (!CheckModuleLevelName(m, arg, \*name))

{

return false;

}

return true;

}

bool

AsmJSCompiler::CheckModuleArguments(AsmJsModuleCompiler &m, ParseNode \*fn)

{

ArgSlot numFormals = 0;

ParseNode \*arg1 = ParserWrapper::FunctionArgsList( fn, numFormals );

ParseNode \*arg2 = arg1 ? ParserWrapper::NextVar( arg1 ) : nullptr;

ParseNode \*arg3 = arg2 ? ParserWrapper::NextVar( arg2 ) : nullptr;

if (numFormals > 3)

{

return m.Fail(fn, L"asm.js modules takes at most 3 argument");

}

PropertyName arg1Name = nullptr;

if (numFormals >= 1 && !CheckModuleArgument(m, arg1, &arg1Name, AsmJsModuleArg::ArgType::StdLib))

{

return false;

}

m.InitStdLibArgName(arg1Name);

PropertyName arg2Name = nullptr;

if (numFormals >= 2 && !CheckModuleArgument(m, arg2, &arg2Name, AsmJsModuleArg::ArgType::Import))

{

return false;

}

m.InitForeignArgName(arg2Name);

PropertyName arg3Name = nullptr;

if (numFormals >= 3 && !CheckModuleArgument(m, arg3, &arg3Name, AsmJsModuleArg::ArgType::Heap))

{

return false;

}

m.InitBufferArgName(arg3Name);

return true;

}

bool AsmJSCompiler::CheckGlobalVariableImportExpr( AsmJsModuleCompiler &m, PropertyName varName, AsmJSCoercion coercion, ParseNode \*coercedExpr )

{

if( !ParserWrapper::IsDotMember(coercedExpr) )

{

return m.FailName( coercedExpr, L"invalid import expression for global '%s'", varName );

}

ParseNode \*base = ParserWrapper::DotBase(coercedExpr);

PropertyName field = ParserWrapper::DotMember(coercedExpr);

PropertyName importName = m.GetForeignArgName();

if (!importName || !field)

{

return m.Fail(coercedExpr, L"cannot import without an asm.js foreign parameter");

}

m.GetByteCodeGenerator()->AssignPropertyId(field);

if ((base->name() != importName))

{

return m.FailName(coercedExpr, L"base of import expression must be '%s'", importName);

}

return m.AddGlobalVarImport(varName, field, coercion);

}

bool AsmJSCompiler::CheckGlobalVariableInitImport( AsmJsModuleCompiler &m, PropertyName varName, ParseNode \*initNode, bool isMutable /\*= true\*/)

{

AsmJSCoercion coercion;

ParseNode \*coercedExpr;

if( !CheckTypeAnnotation( m, initNode, &coercion, &coercedExpr ) )

{

return false;

}

if ((ParserWrapper::IsFroundNumericLiteral(coercedExpr)) && coercion == AsmJSCoercion::AsmJS\_FRound)

{

return m.AddNumericVar(varName, coercedExpr, true, isMutable);

}

return CheckGlobalVariableImportExpr( m, varName, coercion, coercedExpr );

}

bool AsmJSCompiler::CheckNewArrayView( AsmJsModuleCompiler &m, PropertyName varName, ParseNode \*newExpr )

{

Assert( newExpr->nop == knopNew );

ParseNode \*ctorExpr = newExpr->sxCall.pnodeTarget;

ArrayBufferView::ViewType type;

if( ParserWrapper::IsDotMember(ctorExpr) )

{

ParseNode \*base = ParserWrapper::DotBase(ctorExpr);

PropertyName globalName = m.GetStdLibArgName();

if (!globalName)

{

return m.Fail(base, L"cannot create array view without an asm.js global parameter");

}

if (!ParserWrapper::IsNameDeclaration(base) || base->name() != globalName)

{

return m.FailName(base, L"expecting '%s.\*Array", globalName);

}

PropertyName fieldName = ParserWrapper::DotMember(ctorExpr);

if (!fieldName)

{

return m.FailName(ctorExpr, L"Failed to define array view to var %s", varName);

}

PropertyId field = fieldName->GetPropertyId();

switch (field)

{

case PropertyIds::Int8Array:

type = ArrayBufferView::TYPE\_INT8;

m.AddArrayBuiltinUse(AsmJSTypedArrayBuiltin\_Int8Array);

break;

case PropertyIds::Uint8Array:

type = ArrayBufferView::TYPE\_UINT8;

m.AddArrayBuiltinUse(AsmJSTypedArrayBuiltin\_Uint8Array);

break;

case PropertyIds::Int16Array:

type = ArrayBufferView::TYPE\_INT16;

m.AddArrayBuiltinUse(AsmJSTypedArrayBuiltin\_Int16Array);

break;

case PropertyIds::Uint16Array:

type = ArrayBufferView::TYPE\_UINT16;

m.AddArrayBuiltinUse(AsmJSTypedArrayBuiltin\_Uint16Array);

break;

case PropertyIds::Int32Array:

type = ArrayBufferView::TYPE\_INT32;

m.AddArrayBuiltinUse(AsmJSTypedArrayBuiltin\_Int32Array);

break;

case PropertyIds::Uint32Array:

type = ArrayBufferView::TYPE\_UINT32;

m.AddArrayBuiltinUse(AsmJSTypedArrayBuiltin\_Uint32Array);

break;

case PropertyIds::Float32Array:

type = ArrayBufferView::TYPE\_FLOAT32;

m.AddArrayBuiltinUse(AsmJSTypedArrayBuiltin\_Float32Array);

break;

case PropertyIds::Float64Array:

type = ArrayBufferView::TYPE\_FLOAT64;

m.AddArrayBuiltinUse(AsmJSTypedArrayBuiltin\_Float64Array);

break;

default:

return m.Fail(ctorExpr, L"could not match typed array name");

break;

}

}

else if (ctorExpr->nop == knopName)

{

AsmJsSymbol \* buffFunc = m.LookupIdentifier(ctorExpr->name());

if (!buffFunc || buffFunc->GetSymbolType() != AsmJsSymbol::TypedArrayBuiltinFunction)

{

return m.Fail(ctorExpr, L"invalid 'new' import");

}

type = buffFunc->Cast<AsmJsTypedArrayFunction>()->GetViewType();

if (type == ArrayBufferView::TYPE\_INVALID)

{

return m.Fail(ctorExpr, L"could not match typed array name");

}

}

else

{

return m.Fail(newExpr, L"invalid 'new' import");

}

ParseNode \*bufArg = newExpr->sxCall.pnodeArgs;

if( !bufArg || !ParserWrapper::IsNameDeclaration( bufArg ) )

{

return m.Fail( ctorExpr, L"array view constructor takes exactly one argument" );

}

PropertyName bufferName = m.GetBufferArgName();

if( !bufferName )

{

return m.Fail( bufArg, L"cannot create array view without an asm.js heap parameter" );

}

if( bufferName != bufArg->name() )

{

return m.FailName( bufArg, L"argument to array view constructor must be '%s'", bufferName );

}

if( !m.AddArrayView( varName, type ) )

{

return m.FailName( ctorExpr, L"Failed to define array view to var %s", varName );

}

return true;

}

bool

AsmJSCompiler::CheckGlobalDotImport(AsmJsModuleCompiler &m, PropertyName varName, ParseNode \*initNode)

{

ParseNode \*base = ParserWrapper::DotBase(initNode);

PropertyName field = ParserWrapper::DotMember(initNode);

if( !field )

{

return m.Fail( initNode, L"Global import must be in the form c.x where c is stdlib or foreign and x is a string literal" );

}

m.GetByteCodeGenerator()->AssignPropertyId(field);

PropertyName lib = nullptr;

if (ParserWrapper::IsDotMember(base))

{

lib = ParserWrapper::DotMember(base);

base = ParserWrapper::DotBase(base);

if (m.GetScriptContext()->GetConfig()->IsSimdjsEnabled())

{

if (!lib || (lib->GetPropertyId() != PropertyIds::Math && lib->GetPropertyId() != PropertyIds::SIMD))

{

return m.FailName(initNode, L"'%s' should be Math or SIMD, as in global.Math.xxxx", field);

}

}

else

{

if (!lib || lib->GetPropertyId() != PropertyIds::Math)

{

return m.FailName(initNode, L"'%s' should be Math, as in global.Math.xxxx", field);

}

}

}

if( ParserWrapper::IsNameDeclaration(base) && base->name() == m.GetStdLibArgName() )

{

if (m.GetScriptContext()->GetConfig()->IsSimdjsEnabled())

{

if (lib && lib->GetPropertyId() == PropertyIds::SIMD)

{

// global.SIMD.xxx

AsmJsSIMDFunction \*simdFunc;

if (!m.LookupStdLibSIMDName(field->GetPropertyId(), field, &simdFunc))

{

return m.FailName(initNode, L"'%s' is not standard SIMD builtin", varName);

}

if (simdFunc->GetName() != nullptr)

{

OutputMessage(m.GetScriptContext(), DEIT\_ASMJS\_FAILED, L"Warning: SIMD Builtin already defined for var %s", simdFunc->GetName()->Psz());

}

simdFunc->SetName(varName);

if (!m.DefineIdentifier(varName, simdFunc))

{

return m.FailName(initNode, L"Failed to define SIMD builtin function to var %s", varName);

}

m.AddSimdBuiltinUse(simdFunc->GetSimdBuiltInFunction());

return true;

}

}

// global.Math.xxx

MathBuiltin mathBuiltin;

if (m.LookupStandardLibraryMathName(field, &mathBuiltin))

{

switch (mathBuiltin.kind)

{

case MathBuiltin::Function:{

auto func = mathBuiltin.u.func;

if (func->GetName() != nullptr)

{

OutputMessage(m.GetScriptContext(), DEIT\_ASMJS\_FAILED, L"Warning: Math Builtin already defined for var %s", func->GetName()->Psz());

}

func->SetName(varName);

if (!m.DefineIdentifier(varName, func))

{

return m.FailName(initNode, L"Failed to define math builtin function to var %s", varName);

}

m.AddMathBuiltinUse(func->GetMathBuiltInFunction());

}

break;

case MathBuiltin::Constant:

if (!m.AddNumericConst(varName, mathBuiltin.u.cst))

{

return m.FailName(initNode, L"Failed to define math constant to var %s", varName);

}

m.AddMathBuiltinUse(mathBuiltin.mathLibFunctionName);

break;

default:

Assume(UNREACHED);

}

return true;

}

TypedArrayBuiltin arrayBuiltin;

if (m.LookupStandardLibraryArrayName(field, &arrayBuiltin))

{

if (arrayBuiltin.mFunc->GetName() != nullptr)

{

OutputMessage(m.GetScriptContext(), DEIT\_ASMJS\_FAILED, L"Warning: Typed array builtin already defined for var %s", arrayBuiltin.mFunc->GetName()->Psz());

}

arrayBuiltin.mFunc->SetName(varName);

if (!m.DefineIdentifier(varName, arrayBuiltin.mFunc))

{

return m.FailName(initNode, L"Failed to define typed array builtin function to var %s", varName);

}

m.AddArrayBuiltinUse(arrayBuiltin.mFunc->GetArrayBuiltInFunction());

return true;

}

return m.FailName(initNode, L"'%s' is not a standard Math builtin", field);

}

else if( ParserWrapper::IsNameDeclaration(base) && base->name() == m.GetForeignArgName() )

{

// foreign import

return m.AddModuleFunctionImport( varName, field );

}

else if (ParserWrapper::IsNameDeclaration(base))

{

// Check if SIMD function import

// e.g. var x = f4.add

AsmJsSIMDFunction \*simdFunc, \*operation;

simdFunc = m.LookupSimdConstructor(base->name());

if (simdFunc == nullptr || !m.LookupStdLibSIMDName(simdFunc->GetSimdBuiltInFunction(), field, &operation))

{

return m.FailName(initNode, L"Invalid dot expression import. %s is not a standard SIMD operation", varName);

}

if (operation->GetName() != nullptr)

{

OutputMessage(m.GetScriptContext(), DEIT\_ASMJS\_FAILED, L"Warning: SIMD Builtin already defined for var %s", operation->GetName()->Psz());

}

// bind operation to var

operation->SetName(varName);

if (!m.DefineIdentifier(varName, operation))

{

return m.FailName(initNode, L"Failed to define SIMD builtin function to var %s", varName);

}

m.AddSimdBuiltinUse(operation->GetSimdBuiltInFunction());

return true;

}

return m.Fail(initNode, L"expecting c.y where c is either the global or foreign parameter");

}

bool

AsmJSCompiler::CheckModuleGlobal(AsmJsModuleCompiler &m, ParseNode \*var)

{

Assert(var->nop == knopVarDecl || var->nop == knopConstDecl);

bool isMutable = var->nop != knopConstDecl;

PropertyName name = var->name();

m.GetByteCodeGenerator()->AssignPropertyId(name);

if (m.LookupIdentifier(name))

{

return m.FailName(var, L"import variable %s names must be unique", name);

}

if (!CheckModuleLevelName(m, var, name))

{

return false;

}

if (!var->sxVar.pnodeInit)

{

return m.Fail(var, L"module import needs initializer");

}

ParseNode \*initNode = var->sxVar.pnodeInit;

if( ParserWrapper::IsNumericLiteral( initNode ) )

{

if (m.AddNumericVar(name, initNode, false, isMutable))

{

return true;

}

else

{

return m.FailName(var, L"Failed to declare numeric var %s", name);

}

}

if (initNode->nop == knopOr || initNode->nop == knopPos || initNode->nop == knopCall)

{

// SIMD\_JS

// e.g. var x = f4(1.0, 2.0, 3.0, 4.0)

if (initNode->nop == knopCall)

{

AsmJsSIMDFunction\* simdSym;

// also checks if simd constructor

simdSym = m.LookupSimdConstructor(initNode->sxCall.pnodeTarget->name());

// call to simd constructor

if (simdSym)

{

// validate args and define a SIMD symbol

return m.AddSimdValueVar(name, initNode, simdSym);

}

// else it is FFI import: var x = f4check(FFI.field), handled in CheckGlobalVariableInitImport

}

return CheckGlobalVariableInitImport(m, name, initNode, isMutable );

}

if( initNode->nop == knopNew )

{

return CheckNewArrayView(m, var->name(), initNode);

}

if (ParserWrapper::IsDotMember(initNode))

{

return CheckGlobalDotImport(m, name, initNode);

}

return m.Fail( initNode, L"Failed to recognize global variable" );

}

bool

AsmJSCompiler::CheckModuleGlobals(AsmJsModuleCompiler &m)

{

ParseNode \*varStmts;

if( !ParserWrapper::ParseVarOrConstStatement( m.GetCurrentParserNode(), &varStmts ) )

{

return false;

}

if (!varStmts)

{

return true;

}

while (varStmts->nop == knopList)

{

ParseNode \* pnode = ParserWrapper::GetBinaryLeft(varStmts);

while (pnode && pnode->nop != knopEndCode)

{

ParseNode \* decl;

if (pnode->nop == knopList)

{

decl = ParserWrapper::GetBinaryLeft(pnode);

pnode = ParserWrapper::GetBinaryRight(pnode);

}

else

{

decl = pnode;

pnode = nullptr;

}

if (decl->nop == knopFncDecl)

{

goto varDeclEnd;

}

else if (decl->nop != knopConstDecl && decl->nop != knopVarDecl)

{

break;

}

if (decl->sxVar.pnodeInit && decl->sxVar.pnodeInit->nop == knopArray)

{

// Assume we reached func tables

goto varDeclEnd;

}

if (!CheckModuleGlobal(m, decl))

{

return false;

}

}

if (ParserWrapper::GetBinaryRight(varStmts)->nop == knopEndCode)

{

// this is an error condition, but CheckFunctionsSequential will figure it out

goto varDeclEnd;

}

varStmts = ParserWrapper::GetBinaryRight(varStmts);

}

varDeclEnd:

// we will collect information on the function tables now and come back to the functions themselves afterwards,

// because function table information is used when processing function bodies

ParseNode \* fnNodes = varStmts;

while (fnNodes->nop != knopEndCode && ParserWrapper::GetBinaryLeft(fnNodes)->nop == knopFncDecl)

{

fnNodes = ParserWrapper::GetBinaryRight(fnNodes);

}

if (fnNodes->nop == knopEndCode)

{

// if there are no function tables, we can just initialize count to 0

m.SetFuncPtrTableCount(0);

}

else

{

m.SetCurrentParseNode(fnNodes);

if (!CheckFunctionTables(m))

{

return false;

}

}

// this will move us back to the beginning of the function declarations

m.SetCurrentParseNode(varStmts);

return true;

}

bool AsmJSCompiler::CheckFunction( AsmJsModuleCompiler &m, ParseNode\* fncNode )

{

Assert( fncNode->nop == knopFncDecl );

if( PHASE\_TRACE1( Js::ByteCodePhase ) )

{

Output::Print( L" Checking Asm function: %s\n", fncNode->sxFnc.funcInfo->name);

}

if( !CheckFunctionHead( m, fncNode, false ) )

{

return false;

}

AsmJsFunc\* func = m.CreateNewFunctionEntry(fncNode);

if (!func)

{

return m.Fail(fncNode, L" Error creating function entry");

}

return true;

}

bool AsmJSCompiler::CheckFunctionsSequential( AsmJsModuleCompiler &m )

{

AsmJSParser& list = m.GetCurrentParserNode();

Assert( list->nop == knopList );

ParseNode\* pnode = ParserWrapper::GetBinaryLeft(list);

while (pnode->nop == knopFncDecl)

{

if( !CheckFunction( m, pnode ) )

{

return false;

}

if(ParserWrapper::GetBinaryRight(list)->nop == knopEndCode)

{

break;

}

list = ParserWrapper::GetBinaryRight(list);

pnode = ParserWrapper::GetBinaryLeft(list);

}

m.SetCurrentParseNode( list );

return true;

}

bool AsmJSCompiler::CheckFunctionTables(AsmJsModuleCompiler &m)

{

AsmJSParser& list = m.GetCurrentParserNode();

Assert(list->nop == knopList);

int32 funcPtrTableCount = 0;

while (list->nop != knopEndCode)

{

ParseNode \* varStmt = ParserWrapper::GetBinaryLeft(list);

if (varStmt->nop != knopConstDecl && varStmt->nop != knopVarDecl)

{

break;

}

if (!varStmt->sxVar.pnodeInit || varStmt->sxVar.pnodeInit->nop != knopArray)

{

break;

}

const uint tableSize = varStmt->sxVar.pnodeInit->sxArrLit.count;

if (!::Math::IsPow2(tableSize))

{

return m.FailName(varStmt, L"Function table [%s] size must be a power of 2", varStmt->name());

}

if (!m.AddFunctionTable(varStmt->name(), tableSize))

{

return m.FailName(varStmt, L"Unable to create new function table %s", varStmt->name());

}

AsmJsFunctionTable\* ftable = (AsmJsFunctionTable\*)m.LookupIdentifier(varStmt->name());

Assert(ftable);

ParseNode\* pnode = varStmt->sxVar.pnodeInit->sxArrLit.pnode1;

if (pnode->nop == knopList)

{

pnode = ParserWrapper::GetBinaryLeft(pnode);

}

if (!ParserWrapper::IsNameDeclaration(pnode))

{

return m.FailName(pnode, L"Invalid element in function table %s", varStmt->name());

}

++funcPtrTableCount;

list = ParserWrapper::GetBinaryRight(list);

}

m.SetFuncPtrTableCount(funcPtrTableCount);

m.SetCurrentParseNode(list);

return true;

}

bool AsmJSCompiler::CheckModuleReturn( AsmJsModuleCompiler& m )

{

ParseNode\* endStmt = m.GetCurrentParserNode();

Assert( endStmt->nop == knopList );

ParseNode\* node = ParserWrapper::GetBinaryLeft( endStmt );

ParseNode\* endNode = ParserWrapper::GetBinaryRight( endStmt );

if( node->nop != knopReturn || endNode->nop != knopEndCode )

{

return m.Fail( node, L"Only expression after table functions must be a return" );

}

ParseNode\* objNode = node->sxReturn.pnodeExpr;

if( objNode->nop != knopObject )

{

if( ParserWrapper::IsNameDeclaration( objNode ) )

{

PropertyName name = objNode->name();

AsmJsSymbol\* sym = m.LookupIdentifier( name );

if( !sym )

{

return m.FailName( node, L"Symbol %s not recognized inside module", name );

}

if( sym->GetSymbolType() != AsmJsSymbol::ModuleFunction )

{

return m.FailName( node, L"Symbol %s can only be a function of the module", name );

}

AsmJsFunc\* func = sym->Cast<AsmJsFunc>();

if( !m.SetExportFunc( func ) )

{

return m.FailName( node, L"Error adding return Symbol %s", name );

}

return true;

}

return m.Fail( node, L"Module return must be an object or 1 function" );

}

ParseNode\* objectElement = ParserWrapper::GetUnaryNode(objNode);

while( objectElement )

{

ParseNode\* member = nullptr;

if( objectElement->nop == knopList )

{

member = ParserWrapper::GetBinaryLeft( objectElement );

objectElement = ParserWrapper::GetBinaryRight( objectElement );

}

else if( objectElement->nop == knopMember )

{

member = objectElement;

objectElement = nullptr;

}

else

{

return m.Fail( node, L"Return object must only contain members" );

}

if( member )

{

ParseNode\* field = ParserWrapper::GetBinaryLeft( member );

ParseNode\* value = ParserWrapper::GetBinaryRight( member );

if( !ParserWrapper::IsNameDeclaration( field ) || !ParserWrapper::IsNameDeclaration( value ) )

{

return m.Fail( node, L"Return object member must be fields" );

}

AsmJsSymbol\* sym = m.LookupIdentifier( value->name() );

if( !sym )

{

return m.FailName( node, L"Symbol %s not recognized inside module", value->name() );

}

if( sym->GetSymbolType() != AsmJsSymbol::ModuleFunction )

{

return m.FailName( node, L"Symbol %s can only be a function of the module", value->name() );

}

AsmJsFunc\* func = sym->Cast<AsmJsFunc>();

if( !m.AddExport( field->name(), func->GetFunctionIndex() ) )

{

return m.FailName( node, L"Error adding return Symbol %s", value->name() );

}

}

}

return true;

}

bool AsmJSCompiler::CheckFuncPtrTables( AsmJsModuleCompiler &m )

{

ParseNode \*list = m.GetCurrentParserNode();

if (!list)

{

return true;

}

while (list->nop != knopEndCode)

{

ParseNode \* varStmt = ParserWrapper::GetBinaryLeft(list);

if (varStmt->nop != knopConstDecl && varStmt->nop != knopVarDecl)

{

break;

}

ParseNode\* nodeInit = varStmt->sxVar.pnodeInit;

if( !nodeInit || nodeInit->nop != knopArray )

{

return m.Fail( varStmt, L"Invalid variable after function declaration" );

}

PropertyName tableName = varStmt->name();

AsmJsSymbol\* sym = m.LookupIdentifier(tableName);

if( !sym )

{

// func table not used in functions disregard it

}

else

{

//Check name

if( sym->GetSymbolType() != AsmJsSymbol::FuncPtrTable )

{

return m.FailName( varStmt, L"Variable %s is already defined", tableName );

}

AsmJsFunctionTable\* table = sym->Cast<AsmJsFunctionTable>();

if( table->IsDefined() )

{

return m.FailName( varStmt, L"Multiple declaration of function table %s", tableName );

}

// Check content of the array

uint count = nodeInit->sxArrLit.count;

if( table->GetSize() != count )

{

return m.FailName( varStmt, L"Invalid size of function table %s", tableName );

}

// Set the content of the array in the table

ParseNode\* node = nodeInit->sxArrLit.pnode1;

uint i = 0;

while( node )

{

ParseNode\* funcNameNode = nullptr;

if( node->nop == knopList )

{

funcNameNode = ParserWrapper::GetBinaryLeft( node );

node = ParserWrapper::GetBinaryRight( node );

}

else

{

Assert( i + 1 == count );

funcNameNode = node;

node = nullptr;

}

if( ParserWrapper::IsNameDeclaration( funcNameNode ) )

{

AsmJsSymbol\* sym = m.LookupIdentifier( funcNameNode->name() );

if( !sym || sym->GetSymbolType() != AsmJsSymbol::ModuleFunction )

{

return m.FailName( varStmt, L"Element in function table %s is not a function", tableName );

}

AsmJsFunc\* func = sym->Cast<AsmJsFunc>();

AsmJsRetType retType;

if (!table->SupportsArgCall(func->GetArgCount(), func->GetArgTypeArray(), retType))

{

return m.FailName(funcNameNode, L"Function signatures in table %s do not match", tableName);

}

if (!table->CheckAndSetReturnType(func->GetReturnType()))

{

return m.FailName(funcNameNode, L"Function return types in table %s do not match", tableName);

}

table->SetModuleFunctionIndex( func->GetFunctionIndex(), i );

++i;

}

else

{

return m.FailName(funcNameNode, L"Element in function table %s is not a function name", tableName);

}

}

table->Define();

}

list = ParserWrapper::GetBinaryRight(list);

}

if( !m.AreAllFuncTableDefined() )

{

return m.Fail(list, L"Some function table were used but not defined");

}

m.SetCurrentParseNode(list);

return true;

}

bool AsmJSCompiler::CheckModule( ExclusiveContext \*cx, AsmJSParser &parser, ParseNode \*stmtList )

{

AsmJsModuleCompiler m( cx, parser );

if( !m.Init() )

{

return false;

}

if( PropertyName moduleFunctionName = ParserWrapper::FunctionName( m.GetModuleFunctionNode() ) )

{

if( !CheckModuleLevelName( m, m.GetModuleFunctionNode(), moduleFunctionName ) )

{

return false;

}

m.InitModuleName( moduleFunctionName );

if( PHASE\_TRACE1( Js::ByteCodePhase ) )

{

Output::Print( L"Asm.Js Module [%s] detected, trying to compile\n", moduleFunctionName->Psz() );

}

}

m.AccumulateCompileTime(AsmJsCompilation::Module);

if( !CheckFunctionHead( m, m.GetModuleFunctionNode() ) )

{

goto AsmJsCompilationError;

}

if (!CheckModuleArguments(m, m.GetModuleFunctionNode()))

{

goto AsmJsCompilationError;

}

if (!CheckModuleGlobals(m))

{

goto AsmJsCompilationError;

}

m.AccumulateCompileTime(AsmJsCompilation::Module);

if (!CheckFunctionsSequential(m))

{

goto AsmJsCompilationError;

}

m.AccumulateCompileTime();

m.InitMemoryOffsets();

if( !m.CompileAllFunctions() )

{

return false;

}

m.AccumulateCompileTime(AsmJsCompilation::ByteCode);

if (!CheckFuncPtrTables(m))

{

m.RevertAllFunctions();

return false;

}

m.AccumulateCompileTime();

if (!CheckModuleReturn(m))

{

m.RevertAllFunctions();

return false;

}

m.CommitFunctions();

m.CommitModule();

m.AccumulateCompileTime(AsmJsCompilation::Module);

m.PrintCompileTrace();

return true;

AsmJsCompilationError:

ParseNode \* moduleNode = m.GetModuleFunctionNode();

if( moduleNode )

{

FunctionBody\* body = moduleNode->sxFnc.funcInfo->GetParsedFunctionBody();

body->ResetByteCodeGenState();

}

cx->byteCodeGenerator->Writer()->Reset();

return false;

}

bool AsmJSCompiler::Compile(ExclusiveContext \*cx, AsmJSParser parser, ParseNode \*stmtList)

{

if (!CheckModule(cx, parser, stmtList))

{

OutputError(cx->scriptContext, L"Asm.js compilation failed.");

return false;

}

return true;

}

void AsmJSCompiler::OutputError(ScriptContext \* scriptContext, const wchar \* message, ...)

{

va\_list argptr;

va\_start(argptr, message);

VOutputMessage(scriptContext, DEIT\_ASMJS\_FAILED, message, argptr);

}

void AsmJSCompiler::OutputMessage(ScriptContext \* scriptContext, const DEBUG\_EVENT\_INFO\_TYPE messageType, const wchar \* message, ...)

{

va\_list argptr;

va\_start(argptr, message);

VOutputMessage(scriptContext, messageType, message, argptr);

}

void AsmJSCompiler::VOutputMessage(ScriptContext \* scriptContext, const DEBUG\_EVENT\_INFO\_TYPE messageType, const wchar \* message, va\_list argptr)

{

wchar\_t buf[2048];

size\_t size;

size = \_vsnwprintf\_s(buf, \_countof(buf), \_TRUNCATE, message, argptr);

if (size == -1)

{

size = 2048;

}

scriptContext->RaiseMessageToDebugger(messageType, buf, scriptContext->GetUrl());

if (PHASE\_TRACE1(AsmjsPhase) || PHASE\_TESTTRACE1(AsmjsPhase))

{

Output::PrintBuffer(buf, size);

Output::Print(L"\n");

Output::Flush();

}

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

// Portions of this file are copyright 2014 Mozilla Foundation, available under the Apache 2.0 license.

//-------------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------------

// Copyright 2014 Mozilla Foundation

//

// Licensed under the Apache License, Version 2.0 (the "License");

// you may not use this file except in compliance with the License.

// You may obtain a copy of the License at

//

// http ://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing, software

// distributed under the License is distributed on an "AS IS" BASIS,

// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

// See the License for the specific language governing permissions and

// limitations under the License.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

struct ExclusiveContext

{

ByteCodeGenerator\* byteCodeGenerator;

ScriptContext \*scriptContext;

ExclusiveContext( ByteCodeGenerator \*\_byteCodeGenerator, ScriptContext \* \_scriptContext ) :byteCodeGenerator( \_byteCodeGenerator ), scriptContext( \_scriptContext ){};

};

class AsmJSCompiler

{

public:

static bool CheckModule( ExclusiveContext \*cx, AsmJSParser &parser, ParseNode \*stmtList );

static bool CheckIdentifier( AsmJsModuleCompiler &m, ParseNode \*usepn, PropertyName name );

static bool CheckModuleLevelName( AsmJsModuleCompiler &m, ParseNode \*usepn, PropertyName name );

static bool CheckFunctionHead( AsmJsModuleCompiler &m, ParseNode \*fn, bool isGlobal = true );

static bool CheckTypeAnnotation( AsmJsModuleCompiler &m, ParseNode \*coercionNode, AsmJSCoercion \*coercion, ParseNode \*\*coercedExpr = nullptr);

static bool CheckModuleArgument( AsmJsModuleCompiler &m, ParseNode \*arg, PropertyName \*name, AsmJsModuleArg::ArgType type);

static bool CheckModuleArguments( AsmJsModuleCompiler &m, ParseNode \*fn );

static bool CheckModuleGlobals( AsmJsModuleCompiler &m );

static bool CheckModuleGlobal( AsmJsModuleCompiler &m, ParseNode \*var );

static bool CheckGlobalDotImport( AsmJsModuleCompiler &m, PropertyName varName, ParseNode \*initNode );

static bool CheckNewArrayView( AsmJsModuleCompiler &m, PropertyName varName, ParseNode \*newExpr );

static bool CheckFunction( AsmJsModuleCompiler &m, ParseNode\* fncNode );

static bool CheckFunctionsSequential(AsmJsModuleCompiler &m);

static bool CheckChangeHeap(AsmJsModuleCompiler &m);

static bool CheckByteLengthCall(AsmJsModuleCompiler &m, ParseNode \* node, ParseNode \* newBufferDecl);

static bool CheckGlobalVariableInitImport( AsmJsModuleCompiler &m, PropertyName varName, ParseNode \*initNode, bool isMutable = true );

static bool CheckGlobalVariableImportExpr(AsmJsModuleCompiler &m, PropertyName varName, AsmJSCoercion coercion, ParseNode \*coercedExpr);

static bool CheckFunctionTables(AsmJsModuleCompiler& m);

static bool CheckModuleReturn( AsmJsModuleCompiler& m );

static bool CheckFuncPtrTables( AsmJsModuleCompiler &m );

static void OutputError(ScriptContext \* scriptContext, const wchar \* message, ...);

static void OutputMessage(ScriptContext \* scriptContext, const DEBUG\_EVENT\_INFO\_TYPE messageType, const wchar \* message, ...);

static void VOutputMessage(ScriptContext \* scriptContext, const DEBUG\_EVENT\_INFO\_TYPE messageType, const wchar \* message, va\_list argptr);

public:

bool static Compile(ExclusiveContext \*cx, AsmJSParser parser, ParseNode \*stmtList);

};

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

// Default all macros to nothing

#ifndef ASMJS\_MATH\_FUNC\_NAMES

#define ASMJS\_MATH\_FUNC\_NAMES(name, propertyName)

#endif

#ifndef ASMJS\_MATH\_CONST\_NAMES

#define ASMJS\_MATH\_CONST\_NAMES(name, propertyName)

#endif

#ifndef ASMJS\_ARRAY\_NAMES

#define ASMJS\_ARRAY\_NAMES(name, propertyName)

#endif

#ifndef ASMJS\_SIMD\_NAMES

#define ASMJS\_SIMD\_NAMES(name, propertyName)

#endif

ASMJS\_MATH\_FUNC\_NAMES(sin, sin)

ASMJS\_MATH\_FUNC\_NAMES(cos, cos)

ASMJS\_MATH\_FUNC\_NAMES(tan, tan)

ASMJS\_MATH\_FUNC\_NAMES(asin, asin)

ASMJS\_MATH\_FUNC\_NAMES(acos, acos)

ASMJS\_MATH\_FUNC\_NAMES(atan, atan)

ASMJS\_MATH\_FUNC\_NAMES(ceil, ceil)

ASMJS\_MATH\_FUNC\_NAMES(floor, floor)

ASMJS\_MATH\_FUNC\_NAMES(exp, exp)

ASMJS\_MATH\_FUNC\_NAMES(log, log)

ASMJS\_MATH\_FUNC\_NAMES(pow, pow)

ASMJS\_MATH\_FUNC\_NAMES(sqrt, sqrt)

ASMJS\_MATH\_FUNC\_NAMES(abs, abs)

ASMJS\_MATH\_FUNC\_NAMES(atan2, atan2)

ASMJS\_MATH\_FUNC\_NAMES(imul, imul)

ASMJS\_MATH\_FUNC\_NAMES(fround, fround)

ASMJS\_MATH\_FUNC\_NAMES(min, min)

ASMJS\_MATH\_FUNC\_NAMES(max, max)

ASMJS\_MATH\_FUNC\_NAMES(clz32, clz32)

ASMJS\_MATH\_CONST\_NAMES(e, E)

ASMJS\_MATH\_CONST\_NAMES(ln10, LN10)

ASMJS\_MATH\_CONST\_NAMES(ln2, LN2)

ASMJS\_MATH\_CONST\_NAMES(log2e, LOG2E)

ASMJS\_MATH\_CONST\_NAMES(log10e, LOG10E)

ASMJS\_MATH\_CONST\_NAMES(pi, PI)

ASMJS\_MATH\_CONST\_NAMES(sqrt1\_2, SQRT1\_2)

ASMJS\_MATH\_CONST\_NAMES(sqrt2, SQRT2)

ASMJS\_MATH\_CONST\_NAMES(infinity, Infinity)

ASMJS\_MATH\_CONST\_NAMES(nan, NaN)

ASMJS\_ARRAY\_NAMES(Uint8Array, Uint8Array)

ASMJS\_ARRAY\_NAMES(Int8Array, Int8Array)

ASMJS\_ARRAY\_NAMES(Uint16Array, Uint16Array)

ASMJS\_ARRAY\_NAMES(Int16Array, Int16Array)

ASMJS\_ARRAY\_NAMES(Uint32Array, Uint32Array)

ASMJS\_ARRAY\_NAMES(Int32Array, Int32Array)

ASMJS\_ARRAY\_NAMES(Float32Array, Float32Array)

ASMJS\_ARRAY\_NAMES(Float64Array, Float64Array)

ASMJS\_ARRAY\_NAMES(byteLength, byteLength)

ASMJS\_SIMD\_NAMES(Int32x4, Int32x4)

ASMJS\_SIMD\_NAMES(int32x4\_check, check)

ASMJS\_SIMD\_NAMES(int32x4\_splat, splat)

ASMJS\_SIMD\_NAMES(int32x4\_fromFloat64x2, fromFloat64x2)

ASMJS\_SIMD\_NAMES(int32x4\_fromFloat64x2Bits, fromFloat64x2Bits)

ASMJS\_SIMD\_NAMES(int32x4\_fromFloat32x4, fromFloat32x4)

ASMJS\_SIMD\_NAMES(int32x4\_fromFloat32x4Bits, fromFloat32x4Bits)

ASMJS\_SIMD\_NAMES(int32x4\_neg, neg)

ASMJS\_SIMD\_NAMES(int32x4\_add, add)

ASMJS\_SIMD\_NAMES(int32x4\_sub, sub)

ASMJS\_SIMD\_NAMES(int32x4\_mul, mul)

ASMJS\_SIMD\_NAMES(int32x4\_extractLane, extractLane)

ASMJS\_SIMD\_NAMES(int32x4\_replaceLane, replaceLane)

ASMJS\_SIMD\_NAMES(int32x4\_swizzle, swizzle)

ASMJS\_SIMD\_NAMES(int32x4\_shuffle, shuffle)

ASMJS\_SIMD\_NAMES(int32x4\_lessThan, lessThan)

ASMJS\_SIMD\_NAMES(int32x4\_equal, equal)

ASMJS\_SIMD\_NAMES(int32x4\_greaterThan, greaterThan)

ASMJS\_SIMD\_NAMES(int32x4\_select, select)

ASMJS\_SIMD\_NAMES(int32x4\_and, and)

ASMJS\_SIMD\_NAMES(int32x4\_or, or)

ASMJS\_SIMD\_NAMES(int32x4\_xor, xor)

ASMJS\_SIMD\_NAMES(int32x4\_not, not)

// ToDo: Enable after fix in lib

//ASMJS\_SIMD\_NAMES(int32x4\_shiftLeftByScalar, shiftLeftByScalar)

//ASMJS\_SIMD\_NAMES(int32x4\_shiftRightLogicalByScalar, shiftRightLogicalByScalar)

//ASMJS\_SIMD\_NAMES(int32x4\_shiftRightArithmeticByScalar, shiftRightArithmeticByScalar)

ASMJS\_SIMD\_NAMES(int32x4\_load, load)

ASMJS\_SIMD\_NAMES(int32x4\_load1, load1)

ASMJS\_SIMD\_NAMES(int32x4\_load2, load2)

ASMJS\_SIMD\_NAMES(int32x4\_load3, load3)

ASMJS\_SIMD\_NAMES(int32x4\_store, store)

ASMJS\_SIMD\_NAMES(int32x4\_store1, store1)

ASMJS\_SIMD\_NAMES(int32x4\_store2, store2)

ASMJS\_SIMD\_NAMES(int32x4\_store3, store3)

ASMJS\_SIMD\_NAMES(Float32x4, Float32x4)

ASMJS\_SIMD\_NAMES(float32x4\_check, check)

ASMJS\_SIMD\_NAMES(float32x4\_splat, splat)

ASMJS\_SIMD\_NAMES(float32x4\_fromFloat64x2, fromFloat64x2)

ASMJS\_SIMD\_NAMES(float32x4\_fromFloat64x2Bits, fromFloat64x2Bits)

ASMJS\_SIMD\_NAMES(float32x4\_fromInt32x4, fromInt32x4)

ASMJS\_SIMD\_NAMES(float32x4\_fromInt32x4Bits, fromInt32x4Bits)

ASMJS\_SIMD\_NAMES(float32x4\_abs, abs)

ASMJS\_SIMD\_NAMES(float32x4\_neg, neg)

ASMJS\_SIMD\_NAMES(float32x4\_add, add)

ASMJS\_SIMD\_NAMES(float32x4\_sub, sub)

ASMJS\_SIMD\_NAMES(float32x4\_mul, mul)

ASMJS\_SIMD\_NAMES(float32x4\_div, div)

ASMJS\_SIMD\_NAMES(float32x4\_clamp, clamp)

ASMJS\_SIMD\_NAMES(float32x4\_min, min)

ASMJS\_SIMD\_NAMES(float32x4\_max, max)

ASMJS\_SIMD\_NAMES(float32x4\_reciprocal, reciprocal)

ASMJS\_SIMD\_NAMES(float32x4\_reciprocalSqrt, reciprocalSqrt)

ASMJS\_SIMD\_NAMES(float32x4\_sqrt, sqrt)

ASMJS\_SIMD\_NAMES(float32x4\_swizzle, swizzle)

ASMJS\_SIMD\_NAMES(float32x4\_shuffle, shuffle)

ASMJS\_SIMD\_NAMES(float32x4\_extractLane, extractLane)

ASMJS\_SIMD\_NAMES(float32x4\_replaceLane, replaceLane)

ASMJS\_SIMD\_NAMES(float32x4\_lessThan, lessThan)

ASMJS\_SIMD\_NAMES(float32x4\_lessThanOrEqual, lessThanOrEqual)

ASMJS\_SIMD\_NAMES(float32x4\_equal, equal)

ASMJS\_SIMD\_NAMES(float32x4\_notEqual, notEqual)

ASMJS\_SIMD\_NAMES(float32x4\_greaterThan, greaterThan)

ASMJS\_SIMD\_NAMES(float32x4\_greaterThanOrEqual, greaterThanOrEqual)

ASMJS\_SIMD\_NAMES(float32x4\_select, select)

ASMJS\_SIMD\_NAMES(float32x4\_and, and)

ASMJS\_SIMD\_NAMES(float32x4\_or, or)

ASMJS\_SIMD\_NAMES(float32x4\_xor, xor)

ASMJS\_SIMD\_NAMES(float32x4\_not, not)

ASMJS\_SIMD\_NAMES(float32x4\_load, load)

ASMJS\_SIMD\_NAMES(float32x4\_load1, load1)

ASMJS\_SIMD\_NAMES(float32x4\_load2, load2)

ASMJS\_SIMD\_NAMES(float32x4\_load3, load3)

ASMJS\_SIMD\_NAMES(float32x4\_store, store)

ASMJS\_SIMD\_NAMES(float32x4\_store1, store1)

ASMJS\_SIMD\_NAMES(float32x4\_store2, store2)

ASMJS\_SIMD\_NAMES(float32x4\_store3, store3)

ASMJS\_SIMD\_NAMES(Float64x2, Float64x2)

ASMJS\_SIMD\_NAMES(float64x2\_check, check)

ASMJS\_SIMD\_NAMES(float64x2\_splat, splat)

ASMJS\_SIMD\_NAMES(float64x2\_fromFloat32x4, fromFloat32x4)

ASMJS\_SIMD\_NAMES(float64x2\_fromFloat32x4Bits, fromFloat32x4Bits)

ASMJS\_SIMD\_NAMES(float64x2\_fromInt32x4, fromInt32x4)

ASMJS\_SIMD\_NAMES(float64x2\_fromInt32x4Bits, fromInt32x4Bits)

ASMJS\_SIMD\_NAMES(float64x2\_abs, abs)

ASMJS\_SIMD\_NAMES(float64x2\_neg, neg)

ASMJS\_SIMD\_NAMES(float64x2\_add, add)

ASMJS\_SIMD\_NAMES(float64x2\_sub, sub)

ASMJS\_SIMD\_NAMES(float64x2\_mul, mul)

ASMJS\_SIMD\_NAMES(float64x2\_div, div)

ASMJS\_SIMD\_NAMES(float64x2\_clamp, clamp)

ASMJS\_SIMD\_NAMES(float64x2\_min, min)

ASMJS\_SIMD\_NAMES(float64x2\_max, max)

ASMJS\_SIMD\_NAMES(float64x2\_reciprocal, reciprocal)

ASMJS\_SIMD\_NAMES(float64x2\_reciprocalSqrt, reciprocalSqrt)

ASMJS\_SIMD\_NAMES(float64x2\_sqrt, sqrt)

ASMJS\_SIMD\_NAMES(float64x2\_swizzle, swizzle)

ASMJS\_SIMD\_NAMES(float64x2\_shuffle, shuffle)

ASMJS\_SIMD\_NAMES(float64x2\_lessThan, lessThan)

ASMJS\_SIMD\_NAMES(float64x2\_lessThanOrEqual, lessThanOrEqual)

ASMJS\_SIMD\_NAMES(float64x2\_equal, equal)

ASMJS\_SIMD\_NAMES(float64x2\_notEqual, notEqual)

ASMJS\_SIMD\_NAMES(float64x2\_greaterThan, greaterThan)

ASMJS\_SIMD\_NAMES(float64x2\_greaterThanOrEqual, greaterThanOrEqual)

ASMJS\_SIMD\_NAMES(float64x2\_select, select)

ASMJS\_SIMD\_NAMES(float64x2\_load, load)

ASMJS\_SIMD\_NAMES(float64x2\_load1, load1)

ASMJS\_SIMD\_NAMES(float64x2\_store, store)

ASMJS\_SIMD\_NAMES(float64x2\_store1, store1)

// help the caller to undefine all the macros

#undef ASMJS\_MATH\_FUNC\_NAMES

#undef ASMJS\_MATH\_CONST\_NAMES

#undef ASMJS\_ARRAY\_NAMES

#undef ASMJS\_SIMD\_NAMES

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

#include "ByteCode\Symbol.h"

#include "ByteCode\FuncInfo.h"

#ifdef DBG\_DUMP

#include "ByteCode\ByteCodeDumper.h"

#include "ByteCode\AsmJSByteCodeDumper.h"

#endif

#include "ByteCode\ByteCodeWriter.h"

#include "ByteCode\ByteCodeGenerator.h"

#include "ByteCode\AsmJsByteCodeWriter.h"

#include "Language\AsmJsByteCodeGenerator.h"

namespace Js

{

enum EBinaryMathOpCodes

{

BMO\_ADD,

BMO\_SUB,

BMO\_MUL,

BMO\_DIV,

BMO\_REM,

BMO\_MAX,

};

enum EBinaryMathOpCodesTypes

{

BMOT\_Int,

BMOT\_UInt,

BMOT\_Float,

BMOT\_Double,

BMOT\_MAX

};

const OpCodeAsmJs BinaryMathOpCodes[BMO\_MAX][BMOT\_MAX] = {

/\*BMO\_ADD\*/{ OpCodeAsmJs::Add\_Int, OpCodeAsmJs::Add\_Int, OpCodeAsmJs::Add\_Flt, OpCodeAsmJs::Add\_Db },

/\*BMO\_SUB\*/{ OpCodeAsmJs::Sub\_Int, OpCodeAsmJs::Sub\_Int, OpCodeAsmJs::Sub\_Flt, OpCodeAsmJs::Sub\_Db },

/\*BMO\_MUL\*/{ OpCodeAsmJs::Mul\_Int, OpCodeAsmJs::Mul\_Int, OpCodeAsmJs::Mul\_Flt, OpCodeAsmJs::Mul\_Db },

/\*BMO\_DIV\*/{ OpCodeAsmJs::Div\_Int, OpCodeAsmJs::Div\_UInt,OpCodeAsmJs::Div\_Flt, OpCodeAsmJs::Div\_Db },

/\*BMO\_REM\*/{ OpCodeAsmJs::Rem\_Int, OpCodeAsmJs::Rem\_UInt,OpCodeAsmJs::Nop, OpCodeAsmJs::Rem\_Db }

};

enum EBinaryComparatorOpCodes

{

/\*<, <=, >, >=, ==, !=\*/

BCO\_LT,

BCO\_LE,

BCO\_GT,

BCO\_GE,

BCO\_EQ,

BCO\_NE,

BCO\_MAX,

};

enum EBinaryComparatorOpCodesTypes

{

BCOT\_Int,

BCOT\_UInt,

BCOT\_Float,

BCOT\_Double,

BCOT\_MAX

};

const OpCodeAsmJs BinaryComparatorOpCodes[BCO\_MAX][BCOT\_MAX] = {

// int unsigned int double

/\*BCO\_LT\*/{ OpCodeAsmJs::CmLt\_Int, OpCodeAsmJs::CmLt\_UnInt, OpCodeAsmJs::CmLt\_Flt, OpCodeAsmJs::CmLt\_Db },

/\*BCO\_LE\*/{ OpCodeAsmJs::CmLe\_Int, OpCodeAsmJs::CmLe\_UnInt, OpCodeAsmJs::CmLe\_Flt, OpCodeAsmJs::CmLe\_Db },

/\*BCO\_GT\*/{ OpCodeAsmJs::CmGt\_Int, OpCodeAsmJs::CmGt\_UnInt, OpCodeAsmJs::CmGt\_Flt, OpCodeAsmJs::CmGt\_Db },

/\*BCO\_GE\*/{ OpCodeAsmJs::CmGe\_Int, OpCodeAsmJs::CmGe\_UnInt, OpCodeAsmJs::CmGe\_Flt, OpCodeAsmJs::CmGe\_Db },

/\*BCO\_EQ\*/{ OpCodeAsmJs::CmEq\_Int, OpCodeAsmJs::CmEq\_Int, OpCodeAsmJs::CmEq\_Flt, OpCodeAsmJs::CmEq\_Db },

/\*BCO\_NE\*/{ OpCodeAsmJs::CmNe\_Int, OpCodeAsmJs::CmNe\_Int, OpCodeAsmJs::CmNe\_Flt, OpCodeAsmJs::CmNe\_Db },

};

#define CheckNodeLocation(info,type) if(!mFunction->IsValidLocation<type>(&info)){\

throw AsmJsCompilationException( L"Invalid Node location[%d] ", info.location ); }

AsmJSByteCodeGenerator::AsmJSByteCodeGenerator( AsmJsFunc\* func, AsmJsModuleCompiler\* compiler ) :

mFunction( func )

, mAllocator(L"AsmjsByteCode", compiler->GetScriptContext()->GetThreadContext()->GetPageAllocator(), Throw::OutOfMemory)

, mInfo( mFunction->GetFuncInfo() )

, mCompiler( compiler )

, mByteCodeGenerator(mCompiler->GetByteCodeGenerator())

, mNestedCallCount(0)

, mIsCallLegal(true)

{

mWriter.Create();

const long astSize = func->GetFncNode()->sxFnc.astSize/AstBytecodeRatioEstimate;

// Use the temp allocator in bytecode write temp buffer.

mWriter.InitData(&mAllocator, astSize);

#ifdef LOG\_BYTECODE\_AST\_RATIO

// log the max Ast size

Output::Print(L"Max Ast size: %d", astSize);

#endif

}

bool AsmJSByteCodeGenerator::BlockHasOwnScope( ParseNode\* pnodeBlock )

{

Assert( pnodeBlock->nop == knopBlock );

return pnodeBlock->sxBlock.scope != nullptr && ( !( pnodeBlock->grfpn & fpnSyntheticNode ) );

}

// copy all constants from reg spaces to function body.

void AsmJSByteCodeGenerator::LoadAllConstants()

{

FunctionBody \*funcBody = mFunction->GetFuncBody();

funcBody->CreateConstantTable();

Var\* table = (Var\*)funcBody->GetConstTable();

table += AsmJsFunctionMemory::RequiredVarConstants - 1; // we do -1 here as the VarConstant count is erobased calculation

int\* intTable = (int\*)table;

// int Return Register

\*intTable = 0;

intTable++;

JsUtil::BaseDictionary<int, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer> intMap = mFunction->GetRegisterSpace<int>().GetConstMap();

for (auto it = intMap.GetIterator(); it.IsValid(); it.MoveNext())

{

JsUtil::BaseDictionary<int, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer>::EntryType &entry = it.Current();

\*intTable = entry.Key();

intTable++;

}

float\* floatTable = (float\*)intTable;

// float Return Register

\*floatTable = 0;

floatTable++;

JsUtil::BaseDictionary<float, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer> floatMap = mFunction->GetRegisterSpace<float>().GetConstMap();

for (auto it = floatMap.GetIterator(); it.IsValid(); it.MoveNext())

{

JsUtil::BaseDictionary<float, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer>::EntryType &entry = it.Current();

\*floatTable = entry.Key();

floatTable++;

}

double\* doubleTable = (double\*)floatTable;

// double Return Register

\*doubleTable = 0;

doubleTable++;

JsUtil::BaseDictionary<double, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer> doubleMap = mFunction->GetRegisterSpace<double>().GetConstMap();

for (auto it = doubleMap.GetIterator(); it.IsValid(); it.MoveNext())

{

JsUtil::BaseDictionary<double, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer>::EntryType &entry = it.Current();

\*doubleTable = entry.Key();

doubleTable++;

}

// SIMD\_JS

if (IsSimdjsEnabled())

{

AsmJsSIMDValue\* simdTable = (AsmJsSIMDValue\*)doubleTable;

// SIMD return register

simdTable->f64[0] = 0; simdTable->f64[1] = 0;

JsUtil::BaseDictionary<AsmJsSIMDValue, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer> simdMap = mFunction->GetRegisterSpace<AsmJsSIMDValue>().GetConstMap();

for (auto it = simdMap.GetIterator(); it.IsValid(); it.MoveNext())

{

JsUtil::BaseDictionary<AsmJsSIMDValue, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer>::EntryType &entry = it.Current();

RegSlot regSlot = entry.Value();

Assert((Var\*)simdTable + regSlot < (Var\*)funcBody->GetConstTable() + funcBody->GetConstantCount());

// we cannot do sequential copy since registers are assigned to constants in the order they appear in the code, not per dictionary order.

simdTable[entry.Value()] = entry.Key();

}

}

}

void AsmJSByteCodeGenerator::FinalizeRegisters( FunctionBody\* byteCodeFunction )

{

// this value is the number of Var slots needed to allocate all the const

int nbConst =

((mFunction->GetRegisterSpace<double>().GetConstCount() + 1) \* DOUBLE\_SLOTS\_SPACE) // space required for all double constants + 1 return register reserved

+ (int)((mFunction->GetRegisterSpace<float>().GetConstCount() + 1)\* FLOAT\_SLOTS\_SPACE + 0.5 /\*ceil\*/) // space required for all float constants + 1 return register reserved

+ (int)((mFunction->GetRegisterSpace<int>().GetConstCount() + 1) \* INT\_SLOTS\_SPACE + 0.5/\*ceil\*/) // space required for all int constants + 1 return register reserved

+ AsmJsFunctionMemory::RequiredVarConstants;

if (IsSimdjsEnabled())

{

nbConst += (int)((mFunction->GetRegisterSpace<AsmJsSIMDValue>().GetConstCount() + 1) \* SIMD\_SLOTS\_SPACE); // Return register is already reserved in the register space.

}

byteCodeFunction->SetConstantCount(nbConst);

// add 3 for each of I0, F0, and D0

RegSlot regCount = mInfo->RegCount() + 3 + AsmJsFunctionMemory::RequiredVarConstants;

if (IsSimdjsEnabled())

{

// 1 return reg for SIMD

regCount++;

}

byteCodeFunction->SetFirstTmpReg(regCount);

}

bool AsmJSByteCodeGenerator::EmitOneFunction()

{

Assert(mFunction->GetFncNode());

Assert(mFunction->GetBodyNode());

AsmJsFunctionCompilation autoCleanup( this );

try

{

ParseNode\* pnode = mFunction->GetFncNode();

Assert( pnode && pnode->nop == knopFncDecl );

Assert( mInfo != nullptr );

ByteCodeGenerator\* byteCodeGen = GetOldByteCodeGenerator();

MaybeTodo( mInfo->IsFakeGlobalFunction( byteCodeGen->GetFlags() ) );

// Support default arguments ?

MaybeTodo( pnode->sxFnc.HasDefaultArguments() );

FunctionBody\* functionBody = mFunction->GetFuncBody();

functionBody->SetStackNestedFunc( false );

FinalizeRegisters(functionBody);

ArenaAllocator\* alloc = byteCodeGen->GetAllocator();

mInfo->inlineCacheMap = Anew( alloc, FuncInfo::InlineCacheMap,

alloc,

mInfo->RegCount() // Pass the actual register count. TODO: Check if we can reduce this count

);

mInfo->rootObjectLoadInlineCacheMap = Anew( alloc, FuncInfo::RootObjectInlineCacheIdMap,

alloc,

10 );

mInfo->rootObjectStoreInlineCacheMap = Anew( alloc, FuncInfo::RootObjectInlineCacheIdMap,

alloc,

10 );

mInfo->referencedPropertyIdToMapIndex = Anew( alloc, FuncInfo::RootObjectInlineCacheIdMap,

alloc,

10 );

functionBody->AllocateLiteralRegexArray();

mWriter.Begin(byteCodeGen, functionBody, alloc, true /\* byteCodeGen->DoJitLoopBodies( funcInfo )\*/, mInfo->hasLoop);

// for now, emit all constant loads at top of function (should instead put in

// closest dominator of uses)

LoadAllConstants();

DefineLabels( );

EmitAsmJsFunctionBody();

// Set that the function is asmjsFuntion in functionBody here so that Initialize ExecutionMode call later will check for that and not profile in asmjsMode

functionBody->SetIsAsmJsFunction(true);

functionBody->SetIsAsmjsMode(true);

// Do a uint32 add just to verify that we haven't overflowed the reg slot type.

UInt32Math::Add( mFunction->GetRegisterSpace<int>().GetTotalVarCount(), mFunction->GetRegisterSpace<int>().GetConstCount());

UInt32Math::Add( mFunction->GetRegisterSpace<double>().GetTotalVarCount(), mFunction->GetRegisterSpace<double>().GetConstCount());

UInt32Math::Add( mFunction->GetRegisterSpace<float>().GetTotalVarCount(), mFunction->GetRegisterSpace<float>().GetConstCount());

byteCodeGen->MapCacheIdsToPropertyIds( mInfo );

byteCodeGen->MapReferencedPropertyIds( mInfo );

mWriter.End();

autoCleanup.FinishCompilation();

functionBody->SetInitialDefaultEntryPoint();

functionBody->SetIsByteCodeDebugMode( byteCodeGen->IsInDebugMode() );

#if DBG\_DUMP

if( PHASE\_DUMP( ByteCodePhase, mInfo->byteCodeFunction ) && Configuration::Global.flags.Verbose )

{

pnode->Dump();

}

if( byteCodeGen->Trace() || PHASE\_DUMP( ByteCodePhase, mInfo->byteCodeFunction ) )

{

AsmJsByteCodeDumper::Dump( mFunction, functionBody );

}

#endif

}

catch( AsmJsCompilationException& e )

{

PrintAsmJsCompilationError( e.msg() );

return false;

}

return true;

}

void AsmJSByteCodeGenerator::PrintAsmJsCompilationError(\_\_out\_ecount(256) wchar\_t\* msg)

{

uint offset = mWriter.GetCurrentOffset();

ULONG line = 0;

LONG col = 0;

if (!mFunction->GetFuncBody()->GetLineCharOffset(offset, &line, &col))

{

line = 0;

col = 0;

}

wchar\_t filename[\_MAX\_FNAME];

wchar\_t ext[\_MAX\_EXT];

\_wsplitpath\_s( Configuration::Global.flags.Filename, NULL, 0, NULL, 0, filename, \_MAX\_FNAME, ext, \_MAX\_EXT );

LPCOLESTR NoneName = L"None";

LPCOLESTR moduleName = NoneName;

if(mCompiler->GetModuleFunctionName())

{

moduleName = mCompiler->GetModuleFunctionName()->Psz();

}

AsmJSCompiler::OutputError(mCompiler->GetScriptContext(),

L"\n%s%s(%d, %d)\n\tAsm.js Compilation Error function : %s::%s\n\t%s\n",

filename, ext, line + 1, col + 1, moduleName, mFunction->GetName()->Psz(), msg);

}

void AsmJSByteCodeGenerator::DefineLabels()

{

mInfo->singleExit=mWriter.DefineLabel();

SList<ParseNode \*>::Iterator iter(&mInfo->targetStatements);

while (iter.Next())

{

ParseNode \* node = iter.Data();

node->sxStmt.breakLabel=mWriter.DefineLabel();

node->sxStmt.continueLabel=mWriter.DefineLabel();

node->emitLabels=true;

}

}

void AsmJSByteCodeGenerator::EmitAsmJsFunctionBody()

{

ParseNode \*pnodeBody = mFunction->GetBodyNode();

ParseNode \*varStmts = pnodeBody;

// Emit local var declarations: Load of constants to variables.

while (varStmts->nop == knopList)

{

ParseNode \* pnode = ParserWrapper::GetBinaryLeft(varStmts);

while (pnode && pnode->nop != knopEndCode)

{

ParseNode \* decl;

if (pnode->nop == knopList)

{

decl = ParserWrapper::GetBinaryLeft(pnode);

pnode = ParserWrapper::GetBinaryRight(pnode);

}

else

{

decl = pnode;

pnode = nullptr;

}

if (decl->nop != knopVarDecl)

{

goto varDeclEnd;

}

Assert(decl->nop == knopVarDecl);

// since we are parsing the same way we created variables the same time, it is safe to assume these are AsmJsVar\*

AsmJsVar\* var = (AsmJsVar\*)mFunction->FindVar(ParserWrapper::VariableName(decl));

AnalysisAssert(var);

if (var->GetType().isInt())

{

mWriter.AsmInt1Const1(Js::OpCodeAsmJs::Ld\_IntConst, var->GetLocation(), var->GetIntInitialiser());

}

else

{

AsmJsVar \* initSource = nullptr;

if (decl->sxVar.pnodeInit->nop == knopName)

{

AsmJsSymbol \* initSym = mCompiler->LookupIdentifier(decl->sxVar.pnodeInit->name(), mFunction);

if (initSym->GetSymbolType() == AsmJsSymbol::Variable)

{

// in this case we are initializing with value of a constant var

initSource = initSym->Cast<AsmJsVar>();

}

else

{

Assert(initSym->GetSymbolType() == AsmJsSymbol::MathConstant);

Assert(initSym->GetType() == AsmJsType::Double);

AsmJsMathConst\* initConst = initSym->Cast<AsmJsMathConst>();

mWriter.AsmReg2(Js::OpCodeAsmJs::Ld\_Db, var->GetLocation(), mFunction->GetConstRegister<double>(\*initConst->GetVal()));

}

}

else

{

initSource = var;

}

if (initSource)

{

if (var->GetType().isDouble())

{

mWriter.AsmReg2(Js::OpCodeAsmJs::Ld\_Db, var->GetLocation(), mFunction->GetConstRegister<double>(initSource->GetDoubleInitialiser()));

}

else if (var->GetType().isFloat())

{

mWriter.AsmReg2(Js::OpCodeAsmJs::Ld\_Flt, var->GetLocation(), mFunction->GetConstRegister<float>(initSource->GetFloatInitialiser()));

}

else

{

// SIMD\_JS

Assert(var->GetType().isSIMDType());

switch (var->GetType().GetWhich())

{

case AsmJsType::Float32x4:

mWriter.AsmReg2(Js::OpCodeAsmJs::Simd128\_Ld\_F4, var->GetLocation(), mFunction->GetConstRegister<AsmJsSIMDValue>(var->GetSimdConstInitialiser()));

break;

case AsmJsType::Float64x2:

mWriter.AsmReg2(Js::OpCodeAsmJs::Simd128\_Ld\_D2, var->GetLocation(), mFunction->GetConstRegister<AsmJsSIMDValue>(var->GetSimdConstInitialiser()));

break;

case AsmJsType::Int32x4:

mWriter.AsmReg2(Js::OpCodeAsmJs::Simd128\_Ld\_I4, var->GetLocation(), mFunction->GetConstRegister<AsmJsSIMDValue>(var->GetSimdConstInitialiser()));

break;

default:

Assert(UNREACHED);

}

}

}

}

}

varStmts = ParserWrapper::GetBinaryRight(varStmts);

}

varDeclEnd:

// Emit a function body. Only explicit returns and the implicit "undef" at the bottom

// get copied to the return register.

while (varStmts->nop == knopList)

{

ParseNode \*stmt = ParserWrapper::GetBinaryLeft(varStmts);

EmitTopLevelStatement( stmt );

varStmts = ParserWrapper::GetBinaryRight(varStmts);

}

Assert(!varStmts->CapturesSyms());

EmitTopLevelStatement(varStmts);

}

void AsmJSByteCodeGenerator::EmitTopLevelStatement( ParseNode \*stmt )

{

if( stmt->nop == knopFncDecl && stmt->sxFnc.IsDeclaration() )

{

throw AsmJsCompilationException( L"Cannot declare functions inside asm.js functions" );

}

const EmitExpressionInfo& info = Emit( stmt );

// free tmp register here

mFunction->ReleaseLocationGeneric( &info );

}

EmitExpressionInfo AsmJSByteCodeGenerator::Emit( ParseNode \*pnode )

{

if( !pnode )

{

return EmitExpressionInfo( AsmJsType::Void );

}

switch( pnode->nop )

{

case knopReturn:

return EmitReturn( pnode );

case knopList:{

while( pnode && pnode->nop == knopList )

{

const EmitExpressionInfo& info = Emit( ParserWrapper::GetBinaryLeft( pnode ) );

mFunction->ReleaseLocationGeneric( &info );

pnode = ParserWrapper::GetBinaryRight( pnode );

}

return Emit( pnode );

}

case knopComma:{

const EmitExpressionInfo& info = Emit( ParserWrapper::GetBinaryLeft( pnode ) );

mFunction->ReleaseLocationGeneric( &info );

return Emit( ParserWrapper::GetBinaryRight( pnode ) );

}

case knopBlock:

{

EmitExpressionInfo info = Emit(pnode->sxBlock.pnodeStmt);

if (pnode->emitLabels)

{

mWriter.MarkAsmJsLabel(pnode->sxStmt.breakLabel);

}

return info;

}

case knopCall:

return EmitCall( pnode );

case knopPos:

return EmitUnaryPos( pnode );

case knopNeg:

return EmitUnaryNeg( pnode );

case knopNot:

return EmitUnaryNot( pnode );

case knopLogNot:

return EmitUnaryLogNot( pnode );

case knopEq:

return EmitBinaryComparator( pnode, BCO\_EQ );

case knopNe:

return EmitBinaryComparator( pnode, BCO\_NE );

case knopLt:

return EmitBinaryComparator( pnode, BCO\_LT );

case knopLe:

return EmitBinaryComparator( pnode, BCO\_LE );

case knopGe:

return EmitBinaryComparator( pnode, BCO\_GE );

case knopGt:

return EmitBinaryComparator( pnode, BCO\_GT );

case knopOr:

return EmitBinaryInt( pnode, OpCodeAsmJs::Or\_Int );

case knopXor:

return EmitBinaryInt( pnode, OpCodeAsmJs::Xor\_Int );

case knopAnd:

return EmitBinaryInt( pnode, OpCodeAsmJs::And\_Int );

case knopLsh:

return EmitBinaryInt( pnode, OpCodeAsmJs::Shl\_Int );

case knopRsh:

return EmitBinaryInt( pnode, OpCodeAsmJs::Shr\_Int );

case knopRs2:

return EmitBinaryInt( pnode, OpCodeAsmJs::ShrU\_Int );

case knopMod:

return EmitBinaryMultiType( pnode, BMO\_REM );

case knopDiv:

return EmitBinaryMultiType( pnode, BMO\_DIV );

case knopMul:

return EmitBinaryMultiType( pnode, BMO\_MUL );

case knopSub:

return EmitBinaryMultiType( pnode, BMO\_SUB );

case knopAdd:

return EmitBinaryMultiType( pnode, BMO\_ADD );

case knopName:

case knopStr:

return EmitIdentifier( pnode );

case knopIndex:

return EmitLdArrayBuffer( pnode );

case knopEndCode:

StartStatement(pnode);

if( mFunction->GetReturnType() == AsmJsRetType::Void )

{

mWriter.AsmReg1( Js::OpCodeAsmJs::LdUndef, AsmJsFunctionMemory::ReturnRegister );

}

mWriter.MarkAsmJsLabel( mFunction->GetFuncInfo()->singleExit );

mWriter.EmptyAsm( OpCodeAsmJs::Ret );

EndStatement(pnode);

break;

case knopAsg:

return EmitAssignment( pnode );

case knopFlt:

if (ParserWrapper::IsMinInt(pnode))

{

return EmitExpressionInfo(mFunction->GetConstRegister<int>(MININT32), AsmJsType::Signed);

}

else if (ParserWrapper::IsUnsigned(pnode))

{

return EmitExpressionInfo(mFunction->GetConstRegister<int>((uint32)pnode->sxFlt.dbl), AsmJsType::Unsigned);

}

else

{

return EmitExpressionInfo(mFunction->GetConstRegister<double>(pnode->sxFlt.dbl), AsmJsType::DoubleLit);

}

case knopInt:

if (pnode->sxInt.lw < 0)

{

return EmitExpressionInfo(mFunction->GetConstRegister<int>(pnode->sxInt.lw), AsmJsType::Signed);

}

else

{

return EmitExpressionInfo(mFunction->GetConstRegister<int>(pnode->sxInt.lw), AsmJsType::Fixnum);

}

case knopIf:

return EmitIf( pnode );

case knopQmark:

return EmitQMark( pnode );

case knopSwitch:

return EmitSwitch( pnode );

case knopFor:

MaybeTodo( pnode->sxFor.pnodeInverted != NULL );

{

const EmitExpressionInfo& initInfo = Emit( pnode->sxFor.pnodeInit );

mFunction->ReleaseLocationGeneric( &initInfo );

return EmitLoop( pnode,

pnode->sxFor.pnodeCond,

pnode->sxFor.pnodeBody,

pnode->sxFor.pnodeIncr);

}

break;

case knopWhile:

return EmitLoop( pnode,

pnode->sxWhile.pnodeCond,

pnode->sxWhile.pnodeBody,

nullptr);

case knopDoWhile:

return EmitLoop( pnode,

pnode->sxWhile.pnodeCond,

pnode->sxWhile.pnodeBody,

NULL,

true );

case knopBreak:

Assert( pnode->sxJump.pnodeTarget->emitLabels );

StartStatement(pnode);

mWriter.AsmBr( pnode->sxJump.pnodeTarget->sxStmt.breakLabel );

if( pnode->emitLabels )

{

mWriter.MarkAsmJsLabel( pnode->sxStmt.breakLabel );

}

EndStatement(pnode);

break;

case knopContinue:

Assert( pnode->sxJump.pnodeTarget->emitLabels );

StartStatement(pnode);

mWriter.AsmBr( pnode->sxJump.pnodeTarget->sxStmt.continueLabel );

EndStatement(pnode);

break;

case knopLabel:

break;

case knopVarDecl:

throw AsmJsCompilationException( L"Variable declaration must happen at the top of the function" );

break;

case knopDot:

// To handle expr.signMask for now, until Bools are suppored.

return EmitDotExpr(pnode);

default:

throw AsmJsCompilationException( L"Unhandled parse opcode for asm.js" );

break;

}

return EmitExpressionInfo(AsmJsType::Void);

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitBinaryMultiType( ParseNode \* pnode, EBinaryMathOpCodes op )

{

ParseNode\* lhs = ParserWrapper::GetBinaryLeft(pnode);

ParseNode\* rhs = ParserWrapper::GetBinaryRight(pnode);

EmitExpressionInfo lhsEmit = Emit( lhs );

EmitExpressionInfo rhsEmit = Emit( rhs );

AsmJsType& lType = lhsEmit.type;

AsmJsType& rType = rhsEmit.type;

// don't need coercion inside an a+b+c type expression

if (op == BMO\_ADD || op == BMO\_SUB)

{

if (lType.GetWhich() == AsmJsType::Intish && (lhs->nop == knopAdd || lhs->nop == knopSub))

{

lType = AsmJsType::Int;

}

if (rType.GetWhich() == AsmJsType::Intish && (rhs->nop == knopAdd || rhs->nop == knopSub))

{

rType = AsmJsType::Int;

}

}

EmitExpressionInfo emitInfo( AsmJsType::Double );

StartStatement(pnode);

if( lType.isInt() && rType.isInt() )

{

CheckNodeLocation( lhsEmit, int );

CheckNodeLocation( rhsEmit, int );

auto opType = lType.isUnsigned() ? BMOT\_UInt : BMOT\_Int;

if (op == BMO\_REM || op == BMO\_DIV)

{

// div and rem must have explicit sign

if (!(lType.isSigned() && rType.isSigned()) && !(lType.isUnsigned() && rType.isUnsigned()))

{

throw AsmJsCompilationException(L"arguments to / or %% must both be double?, float?, signed, or unsigned; %s and %s given", lType.toChars(), rType.toChars());

}

}

// try to reuse tmp register

RegSlot intReg = GetAndReleaseBinaryLocations<int>( &lhsEmit, &rhsEmit );

mWriter.AsmReg3(BinaryMathOpCodes[op][opType], intReg, lhsEmit.location, rhsEmit.location );

emitInfo.location = intReg;

emitInfo.type = AsmJsType::Intish;

}

else if (lType.isMaybeDouble() && rType.isMaybeDouble())

{

CheckNodeLocation( lhsEmit, double );

CheckNodeLocation( rhsEmit, double );

RegSlot dbReg = GetAndReleaseBinaryLocations<double>( &lhsEmit, &rhsEmit );

mWriter.AsmReg3( BinaryMathOpCodes[op][BMOT\_Double], dbReg, lhsEmit.location, rhsEmit.location );

emitInfo.location = dbReg;

}

else if (lType.isMaybeFloat() && rType.isMaybeFloat())

{

if (BinaryMathOpCodes[op][BMOT\_Float] == OpCodeAsmJs::Nop)

{

throw AsmJsCompilationException(L"invalid Binary float operation");

}

CheckNodeLocation(lhsEmit, float);

CheckNodeLocation(rhsEmit, float);

RegSlot floatReg = GetAndReleaseBinaryLocations<float>(&lhsEmit, &rhsEmit);

mWriter.AsmReg3(BinaryMathOpCodes[op][BMOT\_Float], floatReg, lhsEmit.location, rhsEmit.location);

emitInfo.location = floatReg;

emitInfo.type = AsmJsType::Floatish;

}

else

{

throw AsmJsCompilationException( L"Unsupported math operation" );

}

EndStatement(pnode);

return emitInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitBinaryInt( ParseNode \* pnode, OpCodeAsmJs op )

{

ParseNode\* lhs = ParserWrapper::GetBinaryLeft( pnode );

ParseNode\* rhs = ParserWrapper::GetBinaryRight( pnode );

const bool isRhs0 = rhs->nop == knopInt && rhs->sxInt.lw == 0;

const bool isOr0Operation = op == OpCodeAsmJs::Or\_Int && isRhs0;

if( isOr0Operation && lhs->nop == knopCall )

{

EmitExpressionInfo info = EmitCall(lhs, AsmJsRetType::Signed);

if (!info.type.isIntish())

{

throw AsmJsCompilationException(L"Invalid type for [| & ^ >> << >>>] left and right operand must be of type intish");

}

info.type = AsmJsType::Signed;

return info;

}

const EmitExpressionInfo& lhsEmit = Emit( lhs );

const EmitExpressionInfo& rhsEmit = Emit( rhs );

const AsmJsType& lType = lhsEmit.type;

const AsmJsType& rType = rhsEmit.type;

if( !lType.isIntish() || !rType.isIntish() )

{

throw AsmJsCompilationException( L"Invalid type for [| & ^ >> << >>>] left and right operand must be of type intish" );

}

CheckNodeLocation( lhsEmit, int );

CheckNodeLocation( rhsEmit, int );

StartStatement(pnode);

EmitExpressionInfo emitInfo( AsmJsType::Signed );

if( op == OpCodeAsmJs::ShrU\_Int )

{

emitInfo.type = AsmJsType::Unsigned;

}

// ignore this specific operation, useful for non asm.js

if( !isRhs0 || op == OpCodeAsmJs::And\_Int )

{

RegSlot dstReg = GetAndReleaseBinaryLocations<int>( &lhsEmit, &rhsEmit );

mWriter.AsmReg3( op, dstReg, lhsEmit.location, rhsEmit.location );

emitInfo.location = dstReg;

}

else

{

mFunction->ReleaseLocation<int>( &rhsEmit );

emitInfo.location = lhsEmit.location;

}

EndStatement(pnode);

return emitInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitReturn( ParseNode \* pnode )

{

ParseNode\* expr = pnode->sxReturn.pnodeExpr;

// return is always the beginning of a statement

AsmJsRetType retType;

EmitExpressionInfo emitInfo( Constants::NoRegister, AsmJsType::Void );

if( !expr )

{

if( !mFunction->CheckAndSetReturnType( AsmJsRetType::Void ) )

{

throw AsmJsCompilationException( L"Different return type for the function" );

}

retType = AsmJsRetType::Void;

// Make sure we return something

mWriter.AsmReg1(Js::OpCodeAsmJs::LdUndef, AsmJsFunctionMemory::ReturnRegister);

}

else

{

EmitExpressionInfo info = Emit(expr);

StartStatement(pnode);

if (info.type.isSubType(AsmJsType::Double))

{

CheckNodeLocation(info, double);

// get return value from tmp register

mWriter.Conv(OpCodeAsmJs::Return\_Db, 0, info.location);

mFunction->ReleaseLocation<double>(&info);

emitInfo.type = AsmJsType::Double;

retType = AsmJsRetType::Double;

}

else if (info.type.isSubType(AsmJsType::Signed))

{

CheckNodeLocation(info, int);

// get return value from tmp register

mWriter.Conv(OpCodeAsmJs::Return\_Int, 0, info.location);

mFunction->ReleaseLocation<int>(&info);

emitInfo.type = AsmJsType::Signed;

retType = AsmJsRetType::Signed;

}

else if (info.type.isSubType(AsmJsType::Float))

{

CheckNodeLocation(info, float);

// get return value from tmp register

mWriter.Conv(OpCodeAsmJs::Return\_Flt, 0, info.location);

mFunction->ReleaseLocation<float>(&info);

emitInfo.type = AsmJsType::Float;

retType = AsmJsRetType::Float;

}

else if (info.type.isSubType(AsmJsType::Float32x4))

{

CheckNodeLocation(info, AsmJsSIMDValue);

mWriter.Conv(OpCodeAsmJs::Simd128\_Return\_F4, 0, info.location);

mFunction->ReleaseLocation<AsmJsSIMDValue>(&info);

emitInfo.type = AsmJsType::Float32x4;

retType = AsmJsRetType::Float32x4;

}

else if (info.type.isSubType(AsmJsType::Int32x4))

{

CheckNodeLocation(info, AsmJsSIMDValue);

mWriter.Conv(OpCodeAsmJs::Simd128\_Return\_I4, 0, info.location);

mFunction->ReleaseLocation<AsmJsSIMDValue>(&info);

emitInfo.type = AsmJsType::Int32x4;

retType = AsmJsRetType::Int32x4;

}

else if (info.type.isSubType(AsmJsType::Float64x2))

{

CheckNodeLocation(info, AsmJsSIMDValue);

mWriter.Conv(OpCodeAsmJs::Simd128\_Return\_D2, 0, info.location);

mFunction->ReleaseLocation<AsmJsSIMDValue>(&info);

emitInfo.type = AsmJsType::Float64x2;

retType = AsmJsRetType::Float64x2;

}

else

{

throw AsmJsCompilationException(L"Expression for return must be subtype of Signed, Double, or Float");

}

EndStatement(pnode);

}

// check if we saw another return already with a different type

if (!mFunction->CheckAndSetReturnType(retType))

{

throw AsmJsCompilationException(L"Different return type for the function %s", mFunction->GetName()->Psz());

}

mWriter.AsmBr( mFunction->GetFuncInfo()->singleExit );

return emitInfo;

}

bool AsmJSByteCodeGenerator::IsFRound(AsmJsMathFunction\* sym)

{

return (sym && sym->GetMathBuiltInFunction() == AsmJSMathBuiltin\_fround);

}

// First set of opcode are for External calls, second set is for internal calls

static const OpCodeAsmJs callOpCode[2][7] =

{

{

OpCodeAsmJs::StartCall

, OpCodeAsmJs::Call

, OpCodeAsmJs::ArgOut\_Db

, OpCodeAsmJs::ArgOut\_Int

, OpCodeAsmJs::Conv\_VTD

, OpCodeAsmJs::Conv\_VTI

, OpCodeAsmJs::Conv\_VTF

},

{

OpCodeAsmJs::I\_StartCall

, OpCodeAsmJs::I\_Call

, OpCodeAsmJs::I\_ArgOut\_Db

, OpCodeAsmJs::I\_ArgOut\_Int

, OpCodeAsmJs::I\_Conv\_VTD

, OpCodeAsmJs::I\_Conv\_VTI

, OpCodeAsmJs::I\_Conv\_VTF

}

};

Js::EmitExpressionInfo AsmJSByteCodeGenerator::EmitCall(ParseNode \* pnode, AsmJsRetType expectedType /\*= AsmJsType::Void\*/)

{

Assert( pnode->nop == knopCall );

ParseNode\* identifierNode = pnode->sxCall.pnodeTarget;

RegSlot funcTableIndexRegister = Constants::NoRegister;

// Function table

if( pnode->sxCall.pnodeTarget->nop == knopIndex )

{

identifierNode = ParserWrapper::GetBinaryLeft( pnode->sxCall.pnodeTarget );

ParseNode\* indexNode = ParserWrapper::GetBinaryRight( pnode->sxCall.pnodeTarget );

// check for table size annotation

if( indexNode->nop != knopAnd )

{

throw AsmJsCompilationException( L"Function table call must be of format identifier[expr & NumericLiteral](...)" );

}

ParseNode\* tableSizeNode = ParserWrapper::GetBinaryRight( indexNode );

if( tableSizeNode->nop != knopInt )

{

throw AsmJsCompilationException( L"Function table call must be of format identifier[expr & NumericLiteral](...)" );

}

if (tableSizeNode->sxInt.lw < 0)

{

throw AsmJsCompilationException(L"Function table size must be positive");

}

const uint tableSize = tableSizeNode->sxInt.lw+1;

if( !::Math::IsPow2(tableSize) )

{

throw AsmJsCompilationException( L"Function table size must be a power of 2" );

}

// Check for function table identifier

if( !ParserWrapper::IsNameDeclaration( identifierNode ) )

{

throw AsmJsCompilationException( L"Function call must be of format identifier(...) or identifier[expr & size](...)" );

}

PropertyName funcName = identifierNode->name();

AsmJsFunctionDeclaration\* sym = mCompiler->LookupFunction( funcName );

if( !sym )

{

throw AsmJsCompilationException( L"Unable to find function table %s", funcName->Psz() );

}

else

{

if( sym->GetSymbolType() != AsmJsSymbol::FuncPtrTable )

{

throw AsmJsCompilationException( L"Identifier %s is not a function table", funcName->Psz() );

}

AsmJsFunctionTable\* funcTable = sym->Cast<AsmJsFunctionTable>();

if( funcTable->GetSize() != tableSize )

{

throw AsmJsCompilationException( L"Trying to load from Function table %s of size [%d] with size [%d]", funcName->Psz(), funcTable->GetSize(), tableSize );

}

}

const EmitExpressionInfo& indexInfo = Emit( indexNode );

if( !indexInfo.type.isInt() )

{

throw AsmJsCompilationException( L"Array Buffer View index must be type int" );

}

CheckNodeLocation( indexInfo, int );

funcTableIndexRegister = indexInfo.location;

}

if( !ParserWrapper::IsNameDeclaration( identifierNode ) )

{

throw AsmJsCompilationException( L"Function call must be of format identifier(...) or identifier[expr & size](...)" );

}

PropertyName funcName = identifierNode->name();

AsmJsFunctionDeclaration\* sym = mCompiler->LookupFunction(funcName);

if( !sym )

{

throw AsmJsCompilationException( L"Undefined function %s", funcName );

}

if (sym->GetSymbolType() == AsmJsSymbol::SIMDBuiltinFunction)

{

// Special handling for .load\*/.store\* operations

AsmJsSIMDFunction \*simdFun = sym->Cast<AsmJsSIMDFunction>();

if (simdFun->IsSimdLoadFunc() || simdFun->IsSimdStoreFunc())

{

return EmitSimdLoadStoreBuiltin(pnode, sym->Cast<AsmJsSIMDFunction>(), expectedType);

}

else

{

return EmitSimdBuiltin(pnode, sym->Cast<AsmJsSIMDFunction>(), expectedType);

}

}

if (IsFRound((AsmJsMathFunction\*)sym))

{

expectedType = AsmJsRetType::Float;

}

const bool isFFI = sym->GetSymbolType() == AsmJsSymbol::ImportFunction;

const bool isMathBuiltin = sym->GetSymbolType() == AsmJsSymbol::MathBuiltinFunction;

if( isMathBuiltin )

{

return EmitMathBuiltin( pnode, sym->Cast<AsmJsMathFunction>(), expectedType );

}

// math builtins have different requirements for call-site coercion

if (!sym->CheckAndSetReturnType(expectedType))

{

throw AsmJsCompilationException(L"Different return type found for function %s", funcName->Psz());

}

if (!mIsCallLegal)

{

Assert(!isMathBuiltin); // math builtins cannot change heap, so they are specifically excluded from this rule

throw AsmJsCompilationException(L"Call is not legal at this location");

}

const int StartCallIndex = 0;

const int CallIndex = 1;

const int ArgOut\_DbIndex = 2;

const int ArgOut\_IntIndex = 3;

const int Conv\_VTDIndex = 4;

const int Conv\_VTIIndex =5;

const int Conv\_VTFIndex = 6;

const int funcOpCode = isFFI ? 0 : 1;

// StartCall

const ArgSlot argCount = pnode->sxCall.argCount;

StartStatement(pnode);

++mNestedCallCount;

uint startCallOffset = mWriter.GetCurrentOffset();

auto startCallChunk = mWriter.GetCurrentChunk();

uint startCallChunkOffset = startCallChunk->GetCurrentOffset();

bool patchStartCall = sym->GetArgCount() == Constants::InvalidArgSlot;

if (patchStartCall)

{

// we will not know the types of the arguments for the first call to a deferred function,

// so we put a placeholder instr in the bytecode and then patch it with correct arg size

// once we evaluate the arguments

mWriter.AsmStartCall(callOpCode[funcOpCode][StartCallIndex], Constants::InvalidArgSlot);

}

else

{

// args size + 1 pointer

const ArgSlot argByteSize = UInt16Math::Add(sym->GetArgByteSize(argCount), sizeof(Var));

mWriter.AsmStartCall(callOpCode[funcOpCode][StartCallIndex], argByteSize);

}

AutoArrayPtr<AsmJsType> types(nullptr, 0);

int maxDepthForLevel = mFunction->GetArgOutDepth();

if( argCount > 0 )

{

ParseNode\* argNode = pnode->sxCall.pnodeArgs;

uint16 regSlotLocation = 1;

types.Set(HeapNewArray( AsmJsType, argCount ), argCount);

for(ArgSlot i = 0; i < argCount; i++)

{

// Get i arg node

ParseNode\* arg = argNode;

if( argNode->nop == knopList )

{

arg = ParserWrapper::GetBinaryLeft( argNode );

argNode = ParserWrapper::GetBinaryRight( argNode );

}

// Emit argument

const EmitExpressionInfo& argInfo = Emit( arg );

types[i] = argInfo.type;

// OutParams i

if( argInfo.type.isDouble() )

{

CheckNodeLocation( argInfo, double );

if (callOpCode[funcOpCode][ArgOut\_DbIndex] == OpCodeAsmJs::ArgOut\_Db)

{

mWriter.AsmReg2(callOpCode[funcOpCode][ArgOut\_DbIndex], regSlotLocation, argInfo.location);

regSlotLocation++; // in case of external calls this is boxed and converted to a Var

}

else

{

mWriter.AsmReg2(callOpCode[funcOpCode][ArgOut\_DbIndex], regSlotLocation, argInfo.location);

regSlotLocation += sizeof(double) / sizeof(Var);// in case of internal calls we will pass this arg as double

}

mFunction->ReleaseLocation<double>( &argInfo );

}

else if (argInfo.type.isFloat())

{

CheckNodeLocation(argInfo, float);

if (isFFI)

{

throw AsmJsCompilationException(L"FFI function %s doesn't support float arguments", funcName->Psz());

}

mWriter.AsmReg2(OpCodeAsmJs::I\_ArgOut\_Flt, regSlotLocation, argInfo.location);

regSlotLocation++;

mFunction->ReleaseLocation<float>(&argInfo);

}

else if (argInfo.type.isInt())

{

CheckNodeLocation( argInfo, int );

mWriter.AsmReg2(callOpCode[funcOpCode][ArgOut\_IntIndex], regSlotLocation, argInfo.location);

regSlotLocation++;

mFunction->ReleaseLocation<int>( &argInfo );

}

else if (argInfo.type.isSIMDType())

{

if (isFFI)

{

throw AsmJsCompilationException(L"FFI function %s doesn't support SIMD arguments", funcName->Psz());

}

CheckNodeLocation(argInfo, AsmJsSIMDValue);

switch (argInfo.type.GetWhich())

{

case AsmJsType::Int32x4:

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_I\_ArgOut\_I4, regSlotLocation, argInfo.location);

break;

case AsmJsType::Float32x4:

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_I\_ArgOut\_F4, regSlotLocation, argInfo.location);

break;

case AsmJsType::Float64x2:

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_I\_ArgOut\_D2, regSlotLocation, argInfo.location);

break;

}

regSlotLocation += sizeof(AsmJsSIMDValue) / sizeof(Var);

mFunction->ReleaseLocation<AsmJsSIMDValue>(&argInfo);

}

else

{

throw AsmJsCompilationException(L"Function %s doesn't support argument of type %s", funcName->Psz(), argInfo.type.toChars());

}

// if there are nested calls, track whichever is the deepest

if (maxDepthForLevel < mFunction->GetArgOutDepth())

{

maxDepthForLevel = mFunction->GetArgOutDepth();

}

}

}

// Check if this function supports the type of these arguments

AsmJsRetType retType;

const bool supported = sym->SupportsArgCall( argCount, types, retType );

if( !supported )

{

throw AsmJsCompilationException( L"Function %s doesn't support arguments", funcName->Psz() );

}

// need to validate return type again because function might support arguments,

// but return a different type, i.e.: abs(int) -> int, but expecting double

// don't validate the return type for foreign import functions

if( !isFFI && retType != expectedType )

{

throw AsmJsCompilationException( L"Function %s returns different type", funcName->Psz() );

}

const ArgSlot argByteSize = UInt16Math::Add(sym->GetArgByteSize(argCount), sizeof(Var));

// +1 is for function object

ArgSlot runtimeArg = UInt16Math::Add(argCount, 1);

if (funcOpCode == 1) // for non import functions runtimeArg is calculated from argByteSize

{

runtimeArg = (ArgSlot)(::ceil((double)(argByteSize / sizeof(Var)))) + 1;

}

// +1 is for return address

maxDepthForLevel += UInt16Math::Add(runtimeArg, 1);

// Make sure we have enough memory allocated for OutParameters

if (mNestedCallCount > 1)

{

mFunction->SetArgOutDepth(maxDepthForLevel);

}

else

{

mFunction->SetArgOutDepth(0);

}

mFunction->UpdateMaxArgOutDepth(maxDepthForLevel);

if (patchStartCall)

{

uint latestOffset = mWriter.GetCurrentOffset();

auto latestChunk = mWriter.GetCurrentChunk();

uint latestChunkOffset = latestChunk->GetCurrentOffset();

// now that we know the types, we can patch the StartCall instr

startCallChunk->SetCurrentOffset(startCallChunkOffset);

mWriter.SetCurrent(startCallOffset, startCallChunk);

// args size + 1 pointer

mWriter.AsmStartCall(callOpCode[funcOpCode][StartCallIndex], argByteSize, true /\* isPatching \*/);

// ... and return to where we left off in the buffer like nothing ever happened

latestChunk->SetCurrentOffset(latestChunkOffset);

mWriter.SetCurrent(latestOffset, latestChunk);

}

// Load function from env

switch( sym->GetSymbolType() )

{

case AsmJsSymbol::ModuleFunction:

LoadModuleFunction( AsmJsFunctionMemory::FunctionRegister, sym->GetFunctionIndex() );

break;

case AsmJsSymbol::ImportFunction:

LoadModuleFFI( AsmJsFunctionMemory::FunctionRegister, sym->GetFunctionIndex() );

break;

case AsmJsSymbol::FuncPtrTable:

LoadModuleFunctionTable( AsmJsFunctionMemory::FunctionRegister, sym->GetFunctionIndex(), funcTableIndexRegister );

mFunction->ReleaseTmpRegister<int>( funcTableIndexRegister );

break;

default:

Assert( false );

}

// Call

mWriter.AsmCall( callOpCode[funcOpCode][CallIndex], AsmJsFunctionMemory::CallReturnRegister, AsmJsFunctionMemory::FunctionRegister, runtimeArg, expectedType );

// use expected type because return type could be invalid if the function is a FFI

EmitExpressionInfo info( expectedType.toType() );

switch( expectedType.which() )

{

case AsmJsRetType::Void:

// do nothing

break;

case AsmJsRetType::Signed:

{

RegSlot intReg = mFunction->AcquireTmpRegister<int>();

mWriter.AsmReg2( callOpCode[funcOpCode][Conv\_VTIIndex], intReg, AsmJsFunctionMemory::CallReturnRegister );

info.location = intReg;

break;

}

case AsmJsRetType::Double:

{

RegSlot dbReg = mFunction->AcquireTmpRegister<double>();

mWriter.AsmReg2( callOpCode[funcOpCode][Conv\_VTDIndex], dbReg, AsmJsFunctionMemory::CallReturnRegister );

info.location = dbReg;

break;

}

case AsmJsRetType::Float:

{

Assert(!isFFI); //check spec

RegSlot fltReg = mFunction->AcquireTmpRegister<float>();

mWriter.AsmReg2(callOpCode[funcOpCode][Conv\_VTFIndex], fltReg, AsmJsFunctionMemory::CallReturnRegister);

info.location = fltReg;

break;

}

case AsmJsRetType::Float32x4:

{

Assert(!isFFI);

RegSlot simdReg = mFunction->AcquireTmpRegister<AsmJsSIMDValue>();

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_I\_Conv\_VTF4, simdReg, AsmJsFunctionMemory::CallReturnRegister);

info.location = simdReg;

break;

}

case AsmJsRetType::Int32x4:

{

Assert(!isFFI);

RegSlot simdReg = mFunction->AcquireTmpRegister<AsmJsSIMDValue>();

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_I\_Conv\_VTI4, simdReg, AsmJsFunctionMemory::CallReturnRegister);

info.location = simdReg;

break;

}

case AsmJsRetType::Float64x2:

{

Assert(!isFFI);

RegSlot simdReg = mFunction->AcquireTmpRegister<AsmJsSIMDValue>();

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_I\_Conv\_VTD2, simdReg, AsmJsFunctionMemory::CallReturnRegister);

info.location = simdReg;

break;

}

default:

break;

}

EndStatement(pnode);

--mNestedCallCount;

Assert(mNestedCallCount >= 0);

return info;

}

EmitExpressionInfo\* AsmJSByteCodeGenerator::EmitSimdBuiltinArguments(ParseNode\* pnode, AsmJsFunctionDeclaration\* func, \_\_out\_ecount(pnode->sxCall.argCount) AsmJsType \*argsTypes, EmitExpressionInfo \*argsInfo)

{

const uint16 argCount = pnode->sxCall.argCount;

Assert(argsTypes);

Assert(argsInfo);

if (argCount > 0)

{

ParseNode\* argNode = pnode->sxCall.pnodeArgs;

for (ArgSlot i = 0; i < argCount; i++)

{

// Get i arg node

ParseNode\* arg = argNode;

if (argNode->nop == knopList)

{

arg = ParserWrapper::GetBinaryLeft(argNode);

argNode = ParserWrapper::GetBinaryRight(argNode);

}

if (func->GetSymbolType() == AsmJsSymbol::SIMDBuiltinFunction)

{

AsmJsSIMDFunction \*simdFunc = func->Cast<AsmJsSIMDFunction>();

if (arg->nop == knopCall)

{

// REVIEW: Is this exactly according to spec ?

// This enforces Asm.js rule that all arg calls to user-functions have to be coerced.

// Generic calls have to be coerced unless used in a SIMD coercion.

// For example, we cannot do f4add(foo(), bar()), but we can do f4add(f4check(foo()), f4check(bar()))

//

// We are only allowed calls as args in similar cases:

// Float32x4:

// f4check(foo()); call coercion, any call is allowed

// f4(fround(), fround(), ...); constructor, only fround is allowed

// f4add(f4\*(..),f4\*(..)); operation, only other SIMD functions are allowed (including coercion)

//

// Int32x4:

// i4check(foo()); call coercion, any call is allowed

// i4add(i4\*(), i4\*()); operation, only other SIMD functions are allowed (including coercion)

//

// Float64x2:

// similar to Int32x4

PropertyName argCallTarget = ParserWrapper::VariableName(arg->sxCall.pnodeTarget);

AsmJsFunctionDeclaration\* argCall = mCompiler->LookupFunction(argCallTarget);

if (!argCall)

{

throw AsmJsCompilationException(L"Undefined function %s.", argCallTarget->Psz());

}

EmitExpressionInfo argInfo;

if (simdFunc->IsTypeCheck())

{

// type check. Any call is allowed as argument.

argInfo = EmitCall(arg, simdFunc->GetReturnType());

}

// special case for fround inside some float32x4 operations

// f4(fround(), ...) , f4splat(fround()), f4.replaceLane(..,..,fround())

else if ((simdFunc->IsConstructor() && simdFunc->GetSimdBuiltInFunction() == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_Float32x4) || /\*float32x4 all args\*/

simdFunc->GetSimdBuiltInFunction() == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_splat || /\*splat all args\*/

(i == 2 && simdFunc->GetSimdBuiltInFunction() == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_replaceLane))

{

if (argCall && argCall->GetSymbolType() == AsmJsSymbol::MathBuiltinFunction && IsFRound(argCall->Cast<AsmJsMathFunction>()))

{

argInfo = EmitCall(arg, AsmJsRetType::Float);

}

else

{

throw AsmJsCompilationException(L"Invalid call as SIMD argument. Expecting fround.");

}

}

else if (argCall->GetSymbolType() == AsmJsSymbol::SIMDBuiltinFunction && AsmJsSIMDFunction::SameTypeOperations(simdFunc, argCall->Cast<AsmJsSIMDFunction>()))

{

// any other simd operation. call arguments have to be SIMD operations of same type.

argInfo = EmitCall(arg, simdFunc->GetArgType(i).toRetType());

}

else

{

throw AsmJsCompilationException(L"Invalid call as SIMD argument");

}

argsTypes[i] = argInfo.type;

argsInfo[i].type = argInfo.type;

argsInfo[i].location = argInfo.location;

// arg already emitted

continue;

}

else if (simdFunc->IsFloat32x4Func() && arg->nop == knopFlt)

{

// Any floating point constant as float32x4 op arg is considered DoubleLit

// For all float32x4 operations, if the arg type is DoubleLit, regSlot should be in Float reg space.

argsTypes[i] = AsmJsType::DoubleLit;

argsInfo[i].type = AsmJsType::DoubleLit;

argsInfo[i].location = mFunction->GetConstRegister<float>((float)arg->sxFlt.dbl);

// no need to emit constant

continue;

}

else if (simdFunc->IsLaneAccessFunc())

{

if (i == 0 && !simdFunc->GetArgType(i).isSIMDType())

{

throw AsmJsCompilationException(L"Invalid arguments to ExtractLane/ReplaceLane, SIMD type expected for first argument.");

}

if (i == 1) //lane index

{

Assert(simdFunc->GetArgType(i) == AsmJsType::Int);

int lane = (int)arg->sxInt.lw;

if (arg->nop == knopInt)

{

if (lane < 0 || lane > 3)

{

throw AsmJsCompilationException(L"Invalid arguments to ExtractLane/ReplaceLane, out of range lane indices.");

}

}

else

{

throw AsmJsCompilationException(L"Invalid arguments to extractLane/replaceLane, expecting literals for lane indices.");

}

Assert(argCount == 2 || argCount == 3);

argsTypes[i] = AsmJsType::Int;

argsInfo[i].type = AsmJsType::Int;

argsInfo[i].location = mFunction->GetConstRegister<int>((int)lane);

continue;

}

}

else if ((simdFunc->IsShuffleFunc() || simdFunc->IsSwizzleFunc()) && simdFunc->GetArgType(i) == AsmJsType::Int)

{

/\* Int args to shuffle/swizzle should be literals and in-range to match MD instruction\*/

if (arg->nop == knopInt)

{

// E.g.

// f4shuffle(v1, v2, [0-3], [0-3], [4-7], [4-7])

// f4swizzle(v1, [0-3], [0-3], [0-3], [0-3])

bool valid = true;

long laneValue = (int) arg->sxInt.lw;

int argPos = i;

switch (simdFunc->GetSimdBuiltInFunction())

{

case AsmJsSIMDBuiltin\_float32x4\_shuffle:

case AsmJsSIMDBuiltin\_int32x4\_shuffle:

valid = ((argPos == 2 || argPos == 3) && (laneValue >= 0 && laneValue <= 3)) || ((argPos == 4 || argPos == 5) && (laneValue >= 4 && laneValue <= 7));

break;

case AsmJsSIMDBuiltin\_float64x2\_shuffle:

valid = (argPos == 2 && (laneValue >= 0 && laneValue <= 1)) || (argPos == 3 && (laneValue >= 2 && laneValue <= 3));

break;

case AsmJsSIMDBuiltin\_float32x4\_swizzle:

case AsmJsSIMDBuiltin\_int32x4\_swizzle:

valid = (argPos >=1 && argPos <= 4) && (laneValue >= 0 && laneValue <= 3);

break;

case AsmJsSIMDBuiltin\_float64x2\_swizzle:

valid = (argPos >= 1 && argPos <= 2) && (laneValue >= 0 && laneValue <= 1);

break;

default:

Assert(UNREACHED);

}

if (!valid)

{

throw AsmJsCompilationException(L"Invalid arguments to shuffle, out of range lane indices.");

}

argsTypes[i] = AsmJsType::Int;

argsInfo[i].type = AsmJsType::Int;

argsInfo[i].location = mFunction->GetConstRegister<int>((int)laneValue);

// no need to emit constant

continue;

}

else

{

throw AsmJsCompilationException(L"Invalid arguments to swizzle/shuffle, expecting literals for lane indices.");

}

}

}

// Emit argument

const EmitExpressionInfo& argInfo = Emit(arg);

argsTypes[i] = argInfo.type;

argsInfo[i].type = argInfo.type;

argsInfo[i].location = argInfo.location;

}

}

return argsInfo;

}

bool AsmJSByteCodeGenerator::ValidateSimdFieldAccess(PropertyName field, const AsmJsType& receiverType, OpCodeAsmJs &op)

{

PropertyId fieldId = field->GetPropertyId();

// Bind propertyId if not already.

if (fieldId == Js::Constants::NoProperty)

{

mByteCodeGenerator->AssignPropertyId(field);

fieldId = field->GetPropertyId();

}

if (receiverType.isSIMDType())

{

if (fieldId == PropertyIds::signMask)

{

switch (receiverType.GetWhich())

{

case AsmJsType::Int32x4:

op = OpCodeAsmJs::Simd128\_LdSignMask\_I4;

break;

case AsmJsType::Float32x4:

op = OpCodeAsmJs::Simd128\_LdSignMask\_F4;

break;

case AsmJsType::Float64x2:

op = OpCodeAsmJs::Simd128\_LdSignMask\_D2;

break;

default:

Assert(UNREACHED);

}

return true;

}

}

return false;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitDotExpr(ParseNode\* pnode)

{

Assert(ParserWrapper::IsDotMember(pnode));

EmitExpressionInfo exprInfo(Constants::NoRegister, AsmJsType::Void);

OpCodeAsmJs opcode;

RegSlot dst = Constants::NoRegister;

ParseNode\* base = ParserWrapper::DotBase(pnode);

PropertyName field = ParserWrapper::DotMember(pnode);

EmitExpressionInfo baseInfo = Emit(base);

if (!ValidateSimdFieldAccess(field, baseInfo.type, opcode))

{

throw AsmJsCompilationException(L"Expression does not support field access or invalid field name");

}

AssertMsg(baseInfo.type.isSIMDType(), "Expecting SIMD value");

mFunction->ReleaseLocation<AsmJsSIMDValue>(&baseInfo);

// sign mask

dst = mFunction->AcquireTmpRegister<int>();

mWriter.AsmReg2(opcode, dst, baseInfo.location);

exprInfo.type = AsmJsType::Signed;

exprInfo.location = dst;

return exprInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitSimdBuiltin(ParseNode\* pnode, AsmJsSIMDFunction\* simdFunction, AsmJsRetType expectedType)

{

Assert(pnode->nop == knopCall);

// StartCall

const uint16 argCount = pnode->sxCall.argCount;

AutoArrayPtr<AsmJsType> types(nullptr, 0);

AutoArrayPtr<EmitExpressionInfo> argsInfo(nullptr, 0);

if (argCount > 0)

{

types.Set(HeapNewArray(AsmJsType, argCount), argCount);

argsInfo.Set(HeapNewArray(EmitExpressionInfo, argCount), argCount);

EmitSimdBuiltinArguments(pnode, simdFunction, types, argsInfo);

}

AsmJsRetType retType;

OpCodeAsmJs op;

const bool supported = simdFunction->SupportsSIMDCall(argCount, types, op, retType);

if (!supported)

{

throw AsmJsCompilationException(L"SIMD builtin function doesn't support arguments");

}

// If a simd built-in is used without coercion, then expectedType is Void

// e.g. x = f4add(a, b);

if (expectedType != AsmJsRetType::Void && retType != expectedType)

{

throw AsmJsCompilationException(L"SIMD builtin function returns wrong type");

}

// Release all used location before acquiring a new tmp register

for (int i = argCount - 1; i >= 0; i--)

{

mFunction->ReleaseLocationGeneric(&argsInfo[i]);

}

RegSlot dst = Constants::NoRegister;

AsmJsType dstType = AsmJsType::Void;

switch (retType.which())

{

case AsmJsType::Signed:

dst = mFunction->AcquireTmpRegister<int>();

dstType = AsmJsType::Signed;

break;

case AsmJsType::Float:

dst = mFunction->AcquireTmpRegister<float>();

dstType = AsmJsType::Float;

break;

default:

Assert(retType.toVarType().isSIMD());

dst = mFunction->AcquireTmpRegister<AsmJsSIMDValue>();

}

EmitExpressionInfo emitInfo(dst, retType.toType());

if (dstType != AsmJsType::Void)

{

emitInfo.type = dstType;

}

switch (argCount){

case 1:

mWriter.AsmReg2(op, dst, argsInfo[0].location);

break;

case 2:

mWriter.AsmReg3(op, dst, argsInfo[0].location, argsInfo[1].location);

break;

case 3:

mWriter.AsmReg4(op, dst, argsInfo[0].location, argsInfo[1].location, argsInfo[2].location);

break;

case 4:

mWriter.AsmReg5(op, dst, argsInfo[0].location, argsInfo[1].location, argsInfo[2].location, argsInfo[3].location);

break;

case 5:

mWriter.AsmReg6(op, dst, argsInfo[0].location, argsInfo[1].location, argsInfo[2].location, argsInfo[3].location, argsInfo[4].location);

break;

case 6:

mWriter.AsmReg7(op, dst, argsInfo[0].location, argsInfo[1].location, argsInfo[2].location, argsInfo[3].location, argsInfo[4].location, argsInfo[5].location);

break;

default:

AssertMsg(UNREACHED, "Wrong argument count to SIMD function");

}

return emitInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitSimdLoadStoreBuiltin(ParseNode\* pnode, AsmJsSIMDFunction\* simdFunction, AsmJsRetType expectedType)

{

Assert(pnode->nop == knopCall);

Assert(simdFunction->IsSimdLoadFunc() || simdFunction->IsSimdStoreFunc());

const uint16 argCount = pnode->sxCall.argCount;

// Check number of arguments

if ( argCount != simdFunction->GetArgCount())

{

throw AsmJsCompilationException(L"SIMD builtin function doesn't support arguments");

}

ParseNode \*argNode = pnode->sxCall.pnodeArgs;

// Arg1 - tarray

ParseNode\* arrayNameNode = ParserWrapper::GetBinaryLeft(argNode);

argNode = ParserWrapper::GetBinaryRight(argNode);

if (!ParserWrapper::IsNameDeclaration(arrayNameNode))

{

throw AsmJsCompilationException(L"Invalid symbol ");

}

PropertyName name = arrayNameNode->name();

AsmJsSymbol\* sym = mCompiler->LookupIdentifier(name, mFunction);

if (!sym || sym->GetSymbolType() != AsmJsSymbol::ArrayView)

{

throw AsmJsCompilationException(L"Invalid identifier %s", name->Psz());

}

AsmJsArrayView\* arrayView = sym->Cast<AsmJsArrayView>();

ArrayBufferView::ViewType viewType = arrayView->GetViewType();

// Arg2 - index

ParseNode\* indexNode = argNode;

ParseNode\* valueNode = nullptr;

if (simdFunction->IsSimdStoreFunc())

{

indexNode = ParserWrapper::GetBinaryLeft(argNode);

valueNode = ParserWrapper::GetBinaryRight(argNode);

}

OpCodeAsmJs op;

uint32 indexSlot = 0;

TypedArrayEmitType emitType = simdFunction->IsSimdLoadFunc() ? TypedArrayEmitType::LoadTypedArray : TypedArrayEmitType::StoreTypedArray;

// if changeHeap is implemented, calls are illegal in index expression

bool wasCallLegal = mIsCallLegal;

mIsCallLegal = !mCompiler->UsesChangeHeap();

EmitExpressionInfo indexInfo = EmitTypedArrayIndex(indexNode, op, indexSlot, viewType, emitType);

mIsCallLegal = wasCallLegal;

EmitExpressionInfo valueInfo = { 0, AsmJsType::Void };

// convert opcode to const if needed

OpCodeAsmJs opcode = simdFunction->GetOpcode();

if (op == OpCodeAsmJs::LdArrConst || op == OpCodeAsmJs::StArrConst)

{

switch (opcode)

{

case OpCodeAsmJs::Simd128\_LdArr\_I4:

opcode = OpCodeAsmJs::Simd128\_LdArrConst\_I4;

break;

case OpCodeAsmJs::Simd128\_LdArr\_F4:

opcode = OpCodeAsmJs::Simd128\_LdArrConst\_F4;

break;

case OpCodeAsmJs::Simd128\_LdArr\_D2:

opcode = OpCodeAsmJs::Simd128\_LdArrConst\_D2;

break;

case OpCodeAsmJs::Simd128\_StArr\_I4:

opcode = OpCodeAsmJs::Simd128\_StArrConst\_I4;

break;

case OpCodeAsmJs::Simd128\_StArr\_F4:

opcode = OpCodeAsmJs::Simd128\_StArrConst\_F4;

break;

case OpCodeAsmJs::Simd128\_StArr\_D2:

opcode = OpCodeAsmJs::Simd128\_StArrConst\_D2;

break;

default:

Assert(UNREACHED);

}

}

// Adjust dataWidth

int8 dataWidth = 0;

switch (simdFunction->GetSimdBuiltInFunction())

{

case AsmJsSIMDBuiltin\_float32x4\_load1:

case AsmJsSIMDBuiltin\_float32x4\_store1:

case AsmJsSIMDBuiltin\_int32x4\_load1:

case AsmJsSIMDBuiltin\_int32x4\_store1:

dataWidth = 4;

break;

case AsmJsSIMDBuiltin\_float64x2\_load1:

case AsmJsSIMDBuiltin\_float64x2\_store1:

case AsmJsSIMDBuiltin\_float32x4\_load2:

case AsmJsSIMDBuiltin\_float32x4\_store2:

case AsmJsSIMDBuiltin\_int32x4\_load2:

case AsmJsSIMDBuiltin\_int32x4\_store2:

dataWidth = 8;

break;

case AsmJsSIMDBuiltin\_float32x4\_load3:

case AsmJsSIMDBuiltin\_float32x4\_store3:

case AsmJsSIMDBuiltin\_int32x4\_load3:

case AsmJsSIMDBuiltin\_int32x4\_store3:

dataWidth = 12;

break;

case AsmJsSIMDBuiltin\_int32x4\_load:

case AsmJsSIMDBuiltin\_int32x4\_store:

case AsmJsSIMDBuiltin\_float32x4\_load:

case AsmJsSIMDBuiltin\_float32x4\_store:

case AsmJsSIMDBuiltin\_float64x2\_load:

case AsmJsSIMDBuiltin\_float64x2\_store:

dataWidth = 16;

break;

default:

Assert(UNREACHED);

}

EmitExpressionInfo emitInfo;

// Arg3 - Value to Store

if (simdFunction->IsSimdStoreFunc())

{

Assert(valueNode);

// Emit 3rd argument

valueInfo = Emit(valueNode);

if (valueInfo.type != simdFunction->GetArgType(2))

{

throw AsmJsCompilationException(L"Invalid value to SIMD store ");

}

// write opcode

mWriter.AsmSimdTypedArr(opcode, valueInfo.location, indexSlot, dataWidth, viewType);

mFunction->ReleaseLocation<AsmJsSIMDValue>(&valueInfo);

emitInfo.location = 0;

emitInfo.type = AsmJsType::Void;

}

else

{

// load

emitInfo.location = mFunction->AcquireTmpRegister<AsmJsSIMDValue>();

mWriter.AsmSimdTypedArr(opcode, emitInfo.location, indexSlot, dataWidth, viewType);

emitInfo.type = simdFunction->GetReturnType().toType();

}

mFunction->ReleaseLocationGeneric(&indexInfo);

return emitInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitMathBuiltin(ParseNode\* pnode, AsmJsMathFunction\* mathFunction, AsmJsRetType expectedType)

{

if (mathFunction->GetMathBuiltInFunction() == AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_max || mathFunction->GetMathBuiltInFunction() == AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_min)

{

return EmitMinMax(pnode, mathFunction, expectedType);

}

++mNestedCallCount;

const ArgSlot argCount = pnode->sxCall.argCount;

ParseNode\* argNode = pnode->sxCall.pnodeArgs;

// for fround, if we have a fround(NumericLiteral), we want to just emit Ld\_Flt NumericLiteral

if (argCount == 1 && IsFRound(mathFunction) && ParserWrapper::IsFroundNumericLiteral(argNode))

{

Assert(expectedType == AsmJsRetType::Float);

StartStatement(pnode);

RegSlot dst = mFunction->AcquireTmpRegister<float>();

EmitExpressionInfo emitInfo(dst, expectedType.toType());

if (argNode->nop == knopFlt)

{

mWriter.AsmReg2(OpCodeAsmJs::Ld\_Flt, dst, mFunction->GetConstRegister<float>((float)argNode->sxFlt.dbl));

}

else if (argNode->nop == knopInt)

{

mWriter.AsmReg2(OpCodeAsmJs::Ld\_Flt, dst, mFunction->GetConstRegister<float>((float)argNode->sxInt.lw));

}

else

{

Assert(ParserWrapper::IsNegativeZero(argNode));

mWriter.AsmReg2(OpCodeAsmJs::Ld\_Flt, dst, mFunction->GetConstRegister<float>(-0.0f));

}

EndStatement(pnode);

return emitInfo;

}

// The logic here is similar to EmitSimdBuiltinArguments()

// TODO: Maybe outline this to EmitArguments() after RI. Currently it is causing frequent conflicts upon FI.

AutoArrayPtr<AsmJsType> types(nullptr, 0);

AutoArrayPtr<EmitExpressionInfo> argsInfo(nullptr, 0);

int maxDepthForLevel = mFunction->GetArgOutDepth();

if( argCount > 0 )

{

types.Set(HeapNewArray(AsmJsType, argCount), argCount);

argsInfo.Set(HeapNewArray(EmitExpressionInfo, argCount), argCount);

for(ArgSlot i = 0; i < argCount; i++)

{

// Get i arg node

ParseNode\* arg = argNode;

// Special case for fround(abs()) call

if (argNode->nop == knopCall && mathFunction->GetMathBuiltInFunction() == AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_fround)

{

// Emit argument

const EmitExpressionInfo& argInfo = EmitCall(arg, AsmJsRetType::Float);

types[i] = argInfo.type;

argsInfo[i].type = argInfo.type;

argsInfo[i].location = argInfo.location;

}

else

{

if (argNode->nop == knopList)

{

arg = ParserWrapper::GetBinaryLeft(argNode);

argNode = ParserWrapper::GetBinaryRight(argNode);

}

// Emit argument

const EmitExpressionInfo& argInfo = Emit(arg);

types[i] = argInfo.type;

argsInfo[i].type = argInfo.type;

argsInfo[i].location = argInfo.location;

}

// if there are nested calls, track whichever is the deepest

if (maxDepthForLevel < mFunction->GetArgOutDepth())

{

maxDepthForLevel = mFunction->GetArgOutDepth();

}

}

}

StartStatement(pnode);

// Check if this function supports the type of these arguments

AsmJsRetType retType;

OpCodeAsmJs op;

const bool supported = mathFunction->SupportsMathCall( argCount, types, op, retType );

if( !supported )

{

throw AsmJsCompilationException( L"Math builtin function doesn't support arguments" );

}

// Release all used location before acquiring a new tmp register

for (int i = argCount - 1; i >= 0 ; i--)

{

mFunction->ReleaseLocationGeneric( &argsInfo[i] );

}

const int argByteSize = mathFunction->GetArgByteSize(argCount) + sizeof(Var);

// + 1 is for function object

int runtimeArg = (int)(::ceil((double)(argByteSize / sizeof(Var)))) + 1;

// + 1 for return address

maxDepthForLevel += runtimeArg + 1;

// Make sure we have enough memory allocated for OutParameters

if (mNestedCallCount > 1)

{

mFunction->SetArgOutDepth(maxDepthForLevel);

}

else

{

mFunction->SetArgOutDepth(0);

}

mFunction->UpdateMaxArgOutDepth(maxDepthForLevel);

const bool isInt = retType.toType().isInt();

const bool isFloatish = retType.toType().isFloatish();

Assert(isInt || isFloatish || retType.toType().isDouble());

RegSlot dst;

if( isInt )

{

dst = mFunction->AcquireTmpRegister<int>();

}

else if (isFloatish)

{

dst = mFunction->AcquireTmpRegister<float>();

}

else

{

dst = mFunction->AcquireTmpRegister<double>();

}

EmitExpressionInfo emitInfo(dst, retType.toType());

switch( argCount )

{

case 1:

mWriter.AsmReg2( op, dst, argsInfo[0].location );

break;

case 2:

mWriter.AsmReg3( op, dst, argsInfo[0].location, argsInfo[1].location );

break;

default:

Assume(UNREACHED);

}

#if DBG

for (int i = 0; i < argCount; i++)

{

if (argsInfo[i].type.isSubType(AsmJsType::Floatish))

{

CheckNodeLocation(argsInfo[i], float);

}

else if (argsInfo[i].type.isSubType(AsmJsType::MaybeDouble))

{

CheckNodeLocation(argsInfo[i], double);

}

else if (argsInfo[i].type.isSubType(AsmJsType::Intish))

{

CheckNodeLocation(argsInfo[i], int);

}

}

#endif

EndStatement(pnode);

--mNestedCallCount;

return emitInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitMinMax(ParseNode\* pnode, AsmJsMathFunction\* mathFunction, AsmJsRetType expectedType)

{

Assert(mathFunction->GetArgCount() == 2);

++mNestedCallCount;

uint16 argCount = pnode->sxCall.argCount;

ParseNode\* argNode = pnode->sxCall.pnodeArgs;

if (argCount < 2)

{

throw AsmJsCompilationException(L"Math builtin function doesn't support arguments");

}

AutoArrayPtr<AsmJsType> types(nullptr, 0);

AutoArrayPtr<EmitExpressionInfo> argsInfo(nullptr, 0);

types.Set(HeapNewArray(AsmJsType, mathFunction->GetArgCount()), mathFunction->GetArgCount());

argsInfo.Set(HeapNewArray(EmitExpressionInfo, mathFunction->GetArgCount()), mathFunction->GetArgCount());

ParseNode \* arg = ParserWrapper::GetBinaryLeft(argNode);

argNode = ParserWrapper::GetBinaryRight(argNode);

// Emit first arg as arg0

argsInfo[0] = Emit(arg);

int maxDepthForLevel = mFunction->GetArgOutDepth();

types[0] = argsInfo[0].type;

EmitExpressionInfo dstInfo;

for (int i = 1; i < argCount; i++)

{

if (argNode->nop == knopList)

{

arg = ParserWrapper::GetBinaryLeft(argNode);

argNode = ParserWrapper::GetBinaryRight(argNode);

}

else

{

arg = argNode;

}

// arg1 will always be the next arg in the argList

argsInfo[1] = Emit(arg);

types[1] = argsInfo[1].type;

// if there are nested calls, track whichever is the deepest

if (maxDepthForLevel < mFunction->GetArgOutDepth())

{

maxDepthForLevel = mFunction->GetArgOutDepth();

}

// Check if this function supports the type of these arguments

AsmJsRetType retType;

OpCodeAsmJs op;

const bool supported = mathFunction->SupportsMathCall(mathFunction->GetArgCount(), types, op, retType);

if (!supported)

{

throw AsmJsCompilationException(L"Math builtin function doesn't support arguments");

}

const int argByteSize = mathFunction->GetArgByteSize(argCount) + sizeof(Var);

// +1 is for function object

int runtimeArg = (int)(::ceil((double)(argByteSize / sizeof(Var)))) + 1;

// +1 is for return address

maxDepthForLevel += runtimeArg + 1;

// Make sure we have enough memory allocated for OutParameters

if (mNestedCallCount > 1)

{

mFunction->SetArgOutDepth(maxDepthForLevel);

}

else

{

mFunction->SetArgOutDepth(0);

}

mFunction->UpdateMaxArgOutDepth(maxDepthForLevel);

maxDepthForLevel = 0;

mFunction->ReleaseLocationGeneric(&argsInfo[1]);

mFunction->ReleaseLocationGeneric(&argsInfo[0]);

dstInfo.type = retType.toType();

if (retType.toType().isSigned())

{

dstInfo.location = mFunction->AcquireTmpRegister<int>();

}

else

{

Assert(retType.toType().isDouble());

dstInfo.location = mFunction->AcquireTmpRegister<double>();

}

mWriter.AsmReg3(op, dstInfo.location, argsInfo[0].location, argsInfo[1].location);

// for max/min calls with more than 2 arguments, we use the result of previous call for arg0

argsInfo[0] = dstInfo;

#if DBG

for (uint j = 0; j < mathFunction->GetArgCount(); j++)

{

if (argsInfo[j].type.isSubType(AsmJsType::MaybeDouble))

{

CheckNodeLocation(argsInfo[j], double);

}

else if (argsInfo[j].type.isSubType(AsmJsType::Intish))

{

CheckNodeLocation(argsInfo[j], int);

}

else

{

Assert(UNREACHED);

}

}

#endif

}

--mNestedCallCount;

return dstInfo;

}

Js::EmitExpressionInfo AsmJSByteCodeGenerator::EmitIdentifier( ParseNode \* pnode )

{

Assert( ParserWrapper::IsNameDeclaration( pnode ) );

PropertyName name = pnode->name();

AsmJsLookupSource::Source source;

AsmJsSymbol\* sym = mCompiler->LookupIdentifier( name, mFunction, &source );

if( !sym )

{

throw AsmJsCompilationException( L"Undefined identifier %s", name->Psz() );

}

switch( sym->GetSymbolType() )

{

case AsmJsSymbol::Variable:

{

AsmJsVar \* var = sym->Cast<AsmJsVar>();

if (!var->isMutable())

{

// currently const is only allowed for variables at module scope

Assert(source == AsmJsLookupSource::AsmJsModule);

EmitExpressionInfo emitInfo(var->GetType());

if (var->GetVarType().isInt())

{

emitInfo.location = mFunction->AcquireTmpRegister<int>();

mWriter.AsmInt1Const1(Js::OpCodeAsmJs::Ld\_IntConst, emitInfo.location, var->GetIntInitialiser());

}

else if (var->GetVarType().isFloat())

{

emitInfo.location = mFunction->AcquireTmpRegister<float>();

mWriter.AsmReg2(Js::OpCodeAsmJs::Ld\_Flt, emitInfo.location, mFunction->GetConstRegister<float>(var->GetFloatInitialiser()));

}

else

{

Assert(var->GetVarType().isDouble());

emitInfo.location = mFunction->AcquireTmpRegister<double>();

mWriter.AsmReg2(Js::OpCodeAsmJs::Ld\_Db, emitInfo.location, mFunction->GetConstRegister<double>(var->GetDoubleInitialiser()));

}

return emitInfo;

}

// else fall through

}

case AsmJsSymbol::Argument:

case AsmJsSymbol::ConstantImport:

{

AsmJsVarBase\* var = sym->Cast<AsmJsVarBase>();

if( source == AsmJsLookupSource::AsmJsFunction )

{

return EmitExpressionInfo( var->GetLocation(), var->GetType() );

}

else

{

Assert( source == AsmJsLookupSource::AsmJsModule );

EmitExpressionInfo emitInfo(var->GetType());

if (var->GetVarType().isInt())

{

emitInfo.location = mFunction->AcquireTmpRegister<int>();

LoadModuleInt(emitInfo.location, var->GetLocation());

}

else if (var->GetVarType().isFloat())

{

emitInfo.location = mFunction->AcquireTmpRegister<float>();

LoadModuleFloat(emitInfo.location, var->GetLocation());

}

else if (var->GetVarType().isDouble())

{

emitInfo.location = mFunction->AcquireTmpRegister<double>();

LoadModuleDouble(emitInfo.location, var->GetLocation());

}

else if (var->GetVarType().isSIMD())

{

emitInfo.location = mFunction->AcquireTmpRegister<AsmJsSIMDValue>();

LoadModuleSimd( emitInfo.location, var->GetLocation(), var->GetVarType());

}

else

{

Assert(UNREACHED);

}

return emitInfo;

}

break;

}

case AsmJsSymbol::MathConstant:

{

AsmJsMathConst\* mathConst = sym->Cast<AsmJsMathConst>();

Assert(mathConst->GetType().isDouble());

RegSlot loc = mFunction->AcquireTmpRegister<double>();

mWriter.AsmReg2( OpCodeAsmJs::Ld\_Db, loc, mFunction->GetConstRegister<double>(\*mathConst->GetVal()) );

return EmitExpressionInfo(loc, AsmJsType::Double);

}

case AsmJsSymbol::SIMDBuiltinFunction:

case AsmJsSymbol::ImportFunction:

case AsmJsSymbol::FuncPtrTable:

case AsmJsSymbol::ModuleFunction:

case AsmJsSymbol::ArrayView:

case AsmJsSymbol::MathBuiltinFunction:

default:

throw AsmJsCompilationException( L"Cannot use identifier %s in this context", name->Psz() );

}

}

static const OpCodeAsmJs typedArrayOp[2][2] =

{

{ OpCodeAsmJs::LdArrConst, OpCodeAsmJs::LdArr },//LoadTypedArray

{ OpCodeAsmJs::StArrConst, OpCodeAsmJs::StArr },//StoreTypedArray

};

EmitExpressionInfo AsmJSByteCodeGenerator::EmitTypedArrayIndex(ParseNode\* indexNode, OpCodeAsmJs &op, uint32 &indexSlot, ArrayBufferView::ViewType viewType, TypedArrayEmitType emitType)

{

mCompiler->SetUsesHeapBuffer(true);

bool isConst = false;

uint32 slot = 0;

if(indexNode->nop == knopName)

{

AsmJsSymbol \* declSym = mCompiler->LookupIdentifier(indexNode->name(), mFunction);

if (declSym && !declSym->isMutable() && declSym->GetSymbolType() == AsmJsSymbol::Variable)

{

AsmJsVar \* definition = declSym->Cast<AsmJsVar>();

if(definition->GetVarType().isInt())

{

slot = (uint32)definition->GetIntInitialiser();

isConst = true;

}

}

}

if (indexNode->nop == knopInt || indexNode->nop == knopFlt || isConst)

{

// Emit a different opcode for numerical literal

if (!isConst)

{

if (indexNode->nop == knopInt)

{

slot = (uint32)indexNode->sxInt.lw;

}

else if (ParserWrapper::IsMinInt(indexNode))

{

// this is going to be an error, but we can do this to allow it to get same error message as invalid int

slot = (uint32)MININT32;

}

else if (ParserWrapper::IsUnsigned(indexNode))

{

slot = (uint32)indexNode->sxFlt.dbl;

}

else

{

EmitExpressionInfo indexInfo = Emit(indexNode);

throw AsmJsCompilationException(L"Array Index must be intish; %s given", indexInfo.type.toChars());

}

}

// do the right shift now

switch( viewType )

{

case Js::ArrayBufferView::TYPE\_INT16:

case Js::ArrayBufferView::TYPE\_UINT16:

if (slot & 0x80000000)

{

throw AsmJsCompilationException(L"Numeric literal for heap16 must be within 0 <= n < 2^31; %d given", slot);

}

slot <<= 1;

break;

case Js::ArrayBufferView::TYPE\_INT32:

case Js::ArrayBufferView::TYPE\_UINT32:

case Js::ArrayBufferView::TYPE\_FLOAT32:

if (slot & 0xC0000000)

{

throw AsmJsCompilationException(L"Numeric literal for heap32 must be within 0 <= n < 2^30; %d given", slot);

}

slot <<= 2;

break;

case Js::ArrayBufferView::TYPE\_FLOAT64:

if (slot & 0xE0000000)

{

throw AsmJsCompilationException(L"Numeric literal for heap64 must be within 0 <= n < 2^29; %d given", slot);

}

slot <<= 3;

break;

default:

break;

}

mCompiler->UpdateMaxHeapAccess(slot);

op = typedArrayOp[emitType][0];

}

else

{

EmitExpressionInfo indexInfo;

if (indexNode->nop != knopRsh && viewType != Js::ArrayBufferView::TYPE\_INT8 && viewType != Js::ArrayBufferView::TYPE\_UINT8)

{

throw AsmJsCompilationException( L"index expression isn't shifted; must be an Int8/Uint8 access" );

}

int val = 0;

uint32 mask = (uint32)~0;

ParseNode\* index;

if (indexNode->nop == knopRsh)

{

ParseNode\* rhsNode = ParserWrapper::GetBinaryRight(indexNode);

if (!rhsNode || rhsNode->nop != knopInt)

{

throw AsmJsCompilationException(L"shift amount must be constant");

}

switch (viewType)

{

case Js::ArrayBufferView::TYPE\_INT8:

case Js::ArrayBufferView::TYPE\_UINT8:

val = 0;

mask = (uint32)~0;

break;

case Js::ArrayBufferView::TYPE\_INT16:

case Js::ArrayBufferView::TYPE\_UINT16:

val = 1;

mask = (uint32)~1;

break;

case Js::ArrayBufferView::TYPE\_INT32:

case Js::ArrayBufferView::TYPE\_UINT32:

case Js::ArrayBufferView::TYPE\_FLOAT32:

val = 2;

mask = (uint32)~3;

break;

case Js::ArrayBufferView::TYPE\_FLOAT64:

val = 3;

mask = (uint32)~7;

break;

default:

Assume(UNREACHED);

}

if (rhsNode->sxInt.lw != val)

{

throw AsmJsCompilationException(L"shift amount must be %d", val);

}

index = ParserWrapper::GetBinaryLeft(indexNode);

}

else

{

index = indexNode;

}

bool isConst = false;

if (index->nop == knopName)

{

AsmJsSymbol \* declSym = mCompiler->LookupIdentifier(index->name(), mFunction);

if (declSym && !declSym->isMutable() && declSym->GetSymbolType() == AsmJsSymbol::Variable)

{

AsmJsVar \* definition = declSym->Cast<AsmJsVar>();

if (definition->GetVarType().isInt())

{

slot = (uint32)definition->GetIntInitialiser();

slot &= mask;

op = typedArrayOp[emitType][0];

isConst = true;

mCompiler->UpdateMaxHeapAccess(slot);

}

}

}

if( ParserWrapper::IsUInt( index) )

{

slot = ParserWrapper::GetUInt(index);

slot &= mask;

op = typedArrayOp[emitType][0];

mCompiler->UpdateMaxHeapAccess(slot);

}

else if (!isConst)

{

indexInfo = Emit( index );

if( !indexInfo.type.isIntish() )

{

throw AsmJsCompilationException( L"Left operand of >> must be intish; %s given", indexInfo.type.toChars() );

}

indexSlot = indexInfo.location;

op = typedArrayOp[emitType][1];

return indexInfo;

}

}

indexSlot = slot;

return EmitExpressionInfo();

}

Js::EmitExpressionInfo AsmJSByteCodeGenerator::EmitLdArrayBuffer( ParseNode \* pnode )

{

ParseNode\* arrayNameNode = ParserWrapper::GetBinaryLeft( pnode );

ParseNode\* indexNode = ParserWrapper::GetBinaryRight( pnode );

if( !ParserWrapper::IsNameDeclaration( arrayNameNode ) )

{

throw AsmJsCompilationException( L"Invalid symbol " );

}

PropertyName name = arrayNameNode->name();

AsmJsSymbol\* sym = mCompiler->LookupIdentifier(name, mFunction);

if( !sym || sym->GetSymbolType() != AsmJsSymbol::ArrayView )

{

throw AsmJsCompilationException( L"Invalid identifier %s", name->Psz() );

}

AsmJsArrayView\* arrayView = sym->Cast<AsmJsArrayView>();

ArrayBufferView::ViewType viewType = arrayView->GetViewType();

OpCodeAsmJs op;

uint32 indexSlot = 0;

// if changeHeap is implemented, calls are illegal in index expression

bool wasCallLegal = mIsCallLegal;

mIsCallLegal = !mCompiler->UsesChangeHeap();

EmitExpressionInfo indexInfo = EmitTypedArrayIndex(indexNode, op, indexSlot, viewType, LoadTypedArray);

mIsCallLegal = wasCallLegal;

mFunction->ReleaseLocationGeneric(&indexInfo);

EmitExpressionInfo info( arrayView->GetType() );

if( info.type.isIntish() )

{

info.location = mFunction->AcquireTmpRegister<int>();

}

else if (info.type.isMaybeFloat())

{

info.location = mFunction->AcquireTmpRegister<float>();

}

else

{

Assert(info.type.isMaybeDouble());

info.location = mFunction->AcquireTmpRegister<double>();

}

mWriter.AsmTypedArr( op, info.location, indexSlot, viewType );

return info;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitAssignment( ParseNode \* pnode )

{

StartStatement(pnode);

ParseNode\* lhs = ParserWrapper::GetBinaryLeft( pnode );

ParseNode\* rhs = ParserWrapper::GetBinaryRight(pnode);

EmitExpressionInfo rhsEmit;

if( ParserWrapper::IsNameDeclaration( lhs ) )

{

rhsEmit = Emit(rhs);

const AsmJsType& rType = rhsEmit.type;

PropertyName name = lhs->name();

AsmJsLookupSource::Source source;

AsmJsSymbol\* sym = mCompiler->LookupIdentifier( name, mFunction, &source );

if( !sym )

{

throw AsmJsCompilationException( L"Undefined identifier %s", name->Psz() );

}

if( !sym->isMutable() )

{

throw AsmJsCompilationException( L"Cannot assign to identifier %s", name->Psz() );

}

AsmJsVarBase\* var = sym->Cast<AsmJsVarBase>();

if( !var->GetType().isSuperType( rType ) )

{

throw AsmJsCompilationException( L"Cannot assign this type to identifier %s", name->Psz() );

}

switch( source )

{

case Js::AsmJsLookupSource::AsmJsModule:

if( var->GetVarType().isInt() )

{

CheckNodeLocation( rhsEmit, int );

SetModuleInt( var->GetLocation(), rhsEmit.location );

}

else if (var->GetVarType().isFloat())

{

CheckNodeLocation(rhsEmit, float);

SetModuleFloat(var->GetLocation(), rhsEmit.location);

}

else if (var->GetVarType().isDouble())

{

CheckNodeLocation( rhsEmit, double );

SetModuleDouble( var->GetLocation(), rhsEmit.location );

}

else if (var->GetVarType().isSIMD())

{

CheckNodeLocation(rhsEmit, AsmJsSIMDValue);

SetModuleSimd(var->GetLocation(), rhsEmit.location, var->GetVarType());

}

else

{

Assert(UNREACHED);

}

break;

case Js::AsmJsLookupSource::AsmJsFunction:

if( var->GetVarType().isInt() )

{

CheckNodeLocation( rhsEmit, int );

mWriter.AsmReg2( Js::OpCodeAsmJs::Ld\_Int, var->GetLocation(), rhsEmit.location );

}

else if (var->GetVarType().isFloat())

{

CheckNodeLocation(rhsEmit, float);

mWriter.AsmReg2(Js::OpCodeAsmJs::Ld\_Flt, var->GetLocation(), rhsEmit.location);

}

else if (var->GetVarType().isDouble())

{

CheckNodeLocation( rhsEmit, double );

mWriter.AsmReg2( Js::OpCodeAsmJs::Ld\_Db, var->GetLocation(), rhsEmit.location );

}

else if (var->GetVarType().isSIMD())

{

CheckNodeLocation(rhsEmit, AsmJsSIMDValue);

LoadSimd(var->GetLocation(), rhsEmit.location, var->GetVarType());

}

else

{

Assert(UNREACHED);

}

break;

default:

break;

}

}

else if( lhs->nop == knopIndex )

{

ParseNode\* arrayNameNode = ParserWrapper::GetBinaryLeft( lhs );

ParseNode\* indexNode = ParserWrapper::GetBinaryRight( lhs );

if( !ParserWrapper::IsNameDeclaration( arrayNameNode ) )

{

throw AsmJsCompilationException( L"Invalid symbol " );

}

PropertyName name = arrayNameNode->name();

AsmJsSymbol\* sym = mCompiler->LookupIdentifier(name, mFunction);

if( !sym || sym->GetSymbolType() != AsmJsSymbol::ArrayView )

{

throw AsmJsCompilationException( L"Invalid identifier %s", name->Psz() );

}

// must emit index expr first in case it has side effects

AsmJsArrayView\* arrayView = sym->Cast<AsmJsArrayView>();

ArrayBufferView::ViewType viewType = arrayView->GetViewType();

OpCodeAsmJs op;

uint32 indexSlot = 0;

// if changeHeap is implemented, calls are illegal in index expression and on right hand side of assignments

bool wasCallLegal = mIsCallLegal;

mIsCallLegal = !mCompiler->UsesChangeHeap();

EmitExpressionInfo indexInfo = EmitTypedArrayIndex(indexNode, op, indexSlot, viewType, StoreTypedArray);

rhsEmit = Emit(rhs);

mIsCallLegal = wasCallLegal;

if (viewType == ArrayBufferView::TYPE\_FLOAT32)

{

if (!rhsEmit.type.isFloatish() && !rhsEmit.type.isMaybeDouble())

{

throw AsmJsCompilationException(L"Cannot assign value to TYPE\_FLOAT32 ArrayBuffer");

}

// do the conversion to float only for double

if (rhsEmit.type.isMaybeDouble())

{

CheckNodeLocation(rhsEmit, double);

RegSlot dst = mFunction->AcquireTmpRegister<float>();

mWriter.AsmReg2(OpCodeAsmJs::Fround\_Db, dst, rhsEmit.location);

mFunction->ReleaseLocation<double>(&rhsEmit);

rhsEmit.location = dst;

rhsEmit.type = AsmJsType::Float;

}

}

else if (viewType == ArrayBufferView::TYPE\_FLOAT64)

{

if (!rhsEmit.type.isMaybeFloat() && !rhsEmit.type.isMaybeDouble())

{

throw AsmJsCompilationException(L"Cannot assign value to TYPE\_FLOAT64 ArrayBuffer");

}

// do the conversion to double only for float

if (rhsEmit.type.isMaybeFloat())

{

CheckNodeLocation(rhsEmit, float);

RegSlot dst = mFunction->AcquireTmpRegister<double>();

mWriter.AsmReg2(OpCodeAsmJs::Conv\_FTD, dst, rhsEmit.location);

mFunction->ReleaseLocation<float>(&rhsEmit);

rhsEmit.location = dst;

rhsEmit.type = AsmJsType::Double;

}

}

else if (!rhsEmit.type.isSubType(arrayView->GetType()))

{

throw AsmJsCompilationException( L"Cannot assign value ArrayBuffer" );

}

// to keep tmp registers in order, I need to release rhsEmit.local before indexInfo.location

mWriter.AsmTypedArr(op, rhsEmit.location, indexSlot, viewType);

RegSlot rhsReg = rhsEmit.location;

mFunction->ReleaseLocationGeneric(&rhsEmit);

mFunction->ReleaseLocationGeneric(&indexInfo);

RegSlot newRhsReg;

if (rhsEmit.type.isMaybeDouble())

{

newRhsReg = mFunction->AcquireTmpRegister<double>();

mWriter.AsmReg2(OpCodeAsmJs::Ld\_Db, newRhsReg, rhsReg);

}

else if (rhsEmit.type.isFloatish())

{

newRhsReg = mFunction->AcquireTmpRegister<float>();

mWriter.AsmReg2(OpCodeAsmJs::Ld\_Flt, newRhsReg, rhsReg);

}

else

{

newRhsReg = mFunction->AcquireTmpRegister<int>();

mWriter.AsmReg2(OpCodeAsmJs::Ld\_Int, newRhsReg, rhsReg);

}

rhsEmit.location = newRhsReg;

}

else

{

throw AsmJsCompilationException( L"Can only assign to an identifier or an ArrayBufferView" );

}

EndStatement(pnode);

return rhsEmit;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitBinaryComparator( ParseNode \* pnode, EBinaryComparatorOpCodes op )

{

ParseNode\* lhs = ParserWrapper::GetBinaryLeft( pnode );

ParseNode\* rhs = ParserWrapper::GetBinaryRight( pnode );

const EmitExpressionInfo& lhsEmit = Emit( lhs );

EmitExpressionInfo rhsEmit = Emit( rhs );

const AsmJsType& lType = lhsEmit.type;

const AsmJsType& rType = rhsEmit.type;

StartStatement(pnode);

EmitExpressionInfo emitInfo(AsmJsType::Int);

OpCodeAsmJs compOp;

if (lType.isUnsigned() && rType.isUnsigned())

{

CheckNodeLocation(lhsEmit, int);

CheckNodeLocation(rhsEmit, int);

emitInfo.location = GetAndReleaseBinaryLocations<int>(&lhsEmit, &rhsEmit);

compOp = BinaryComparatorOpCodes[op][BCOT\_UInt];

}

else if( lType.isSigned() && rType.isSigned() )

{

CheckNodeLocation( lhsEmit, int );

CheckNodeLocation( rhsEmit, int );

emitInfo.location = GetAndReleaseBinaryLocations<int>( &lhsEmit, &rhsEmit );

compOp = BinaryComparatorOpCodes[op][BCOT\_Int];

}

else if( lType.isDouble() && rType.isDouble() )

{

CheckNodeLocation( lhsEmit, double );

CheckNodeLocation( rhsEmit, double );

emitInfo.location = mFunction->AcquireTmpRegister<int>();

mFunction->ReleaseLocation<double>( &rhsEmit );

mFunction->ReleaseLocation<double>( &lhsEmit );

compOp = BinaryComparatorOpCodes[op][BCOT\_Double];

}

else if (lType.isFloat() && rType.isFloat())

{

CheckNodeLocation(lhsEmit, float);

CheckNodeLocation(rhsEmit, float);

emitInfo.location = mFunction->AcquireTmpRegister<int>();

mFunction->ReleaseLocation<float>(&rhsEmit);

mFunction->ReleaseLocation<float>(&lhsEmit);

compOp = BinaryComparatorOpCodes[op][BCOT\_Float];

}

else

{

throw AsmJsCompilationException( L"Type not supported for comparison" );

}

mWriter.AsmReg3( compOp, emitInfo.location, lhsEmit.location, rhsEmit.location );

EndStatement(pnode);

return emitInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitUnaryPos( ParseNode \* pnode )

{

ParseNode\* rhs = ParserWrapper::GetUnaryNode( pnode );

EmitExpressionInfo rhsEmit ;

if (rhs->nop == knopCall)

{

rhsEmit = EmitCall(rhs, AsmJsRetType::Double);

}

else

{

rhsEmit = Emit(rhs);

}

const AsmJsType& rType = rhsEmit.type;

StartStatement(pnode);

EmitExpressionInfo emitInfo( AsmJsType::Double );

RegSlot dst;

if( rType.isUnsigned() )

{

CheckNodeLocation( rhsEmit, int );

dst = mFunction->AcquireTmpRegister<double>();

mWriter.AsmReg2( OpCodeAsmJs::Conv\_UTD, dst, rhsEmit.location );

mFunction->ReleaseLocation<int>( &rhsEmit );

}

else if( rType.isSigned() )

{

CheckNodeLocation( rhsEmit, int );

dst = mFunction->AcquireTmpRegister<double>();

mWriter.AsmReg2( OpCodeAsmJs::Conv\_ITD, dst, rhsEmit.location );

mFunction->ReleaseLocation<int>( &rhsEmit );

}

else if (rType.isMaybeDouble())

{

CheckNodeLocation( rhsEmit, double );

dst = rhsEmit.location;

}

else if (rType.isMaybeFloat())

{

CheckNodeLocation(rhsEmit, float);

dst = mFunction->AcquireTmpRegister<double>();

mWriter.AsmReg2(OpCodeAsmJs::Conv\_FTD, dst, rhsEmit.location);

mFunction->ReleaseLocation<float>(&rhsEmit);

}

else

{

throw AsmJsCompilationException( L"Type not supported for unary +" );

}

emitInfo.location = dst;

EndStatement(pnode);

return emitInfo;

}

Js::EmitExpressionInfo AsmJSByteCodeGenerator::EmitUnaryNeg( ParseNode \* pnode )

{

ParseNode\* rhs = ParserWrapper::GetUnaryNode( pnode );

const EmitExpressionInfo& rhsEmit = Emit( rhs );

const AsmJsType& rType = rhsEmit.type;

StartStatement(pnode);

EmitExpressionInfo emitInfo;

if( rType.isInt() )

{

CheckNodeLocation( rhsEmit, int );

RegSlot dst = GetAndReleaseUnaryLocations<int>( &rhsEmit );

emitInfo.type = AsmJsType::Intish;

mWriter.AsmReg2( OpCodeAsmJs::Neg\_Int, dst, rhsEmit.location );

emitInfo.location = dst;

}

else if (rType.isMaybeDouble())

{

CheckNodeLocation( rhsEmit, double );

RegSlot dst = GetAndReleaseUnaryLocations<double>( &rhsEmit );

emitInfo.type = AsmJsType::Double;

mWriter.AsmReg2( OpCodeAsmJs::Neg\_Db, dst, rhsEmit.location );

emitInfo.location = dst;

}

else if (rType.isMaybeFloat())

{

CheckNodeLocation(rhsEmit, float);

RegSlot dst = GetAndReleaseUnaryLocations<float>(&rhsEmit);

emitInfo.type = AsmJsType::Floatish;

mWriter.AsmReg2(OpCodeAsmJs::Neg\_Flt, dst, rhsEmit.location);

emitInfo.location = dst;

}

else

{

throw AsmJsCompilationException( L"Type not supported for unary -" );

}

EndStatement(pnode);

return emitInfo;

}

Js::EmitExpressionInfo AsmJSByteCodeGenerator::EmitUnaryNot( ParseNode \* pnode )

{

ParseNode\* rhs = ParserWrapper::GetUnaryNode( pnode );

int count = 1;

while( rhs->nop == knopNot )

{

++count;

rhs = ParserWrapper::GetUnaryNode( rhs );

}

EmitExpressionInfo rhsEmit = Emit( rhs );

AsmJsType rType = rhsEmit.type;

StartStatement(pnode);

if( count >= 2 && rType.isMaybeDouble() )

{

CheckNodeLocation( rhsEmit, double );

count -= 2;

RegSlot dst = mFunction->AcquireTmpRegister<int>();

mWriter.AsmReg2( OpCodeAsmJs::Conv\_DTI, dst, rhsEmit.location );

mFunction->ReleaseLocation<double>( &rhsEmit );

// allow the converted value to be negated (useful for ~(~~(+x)) )

rType = AsmJsType::Signed;

rhsEmit.location = dst;

}

if (count >= 2 && rType.isMaybeFloat())

{

CheckNodeLocation(rhsEmit, float);

count -= 2;

RegSlot dst = mFunction->AcquireTmpRegister<int>();

mWriter.AsmReg2(OpCodeAsmJs::Conv\_FTI, dst, rhsEmit.location);

mFunction->ReleaseLocation<float>(&rhsEmit);

// allow the converted value to be negated (useful for ~(~~(fround(x))) )

rType = AsmJsType::Signed;

rhsEmit.location = dst;

}

if( rType.isIntish() )

{

if( count & 1 )

{

CheckNodeLocation( rhsEmit, int );

RegSlot dst = GetAndReleaseUnaryLocations<int>( &rhsEmit );

// do the conversion only if we have an odd number of the operator

mWriter.AsmReg2( OpCodeAsmJs::Not\_Int, dst, rhsEmit.location );

rhsEmit.location = dst;

}

rhsEmit.type = AsmJsType::Signed;

}

else

{

throw AsmJsCompilationException( L"Type not supported for unary ~" );

}

EndStatement(pnode);

return rhsEmit;

}

Js::EmitExpressionInfo AsmJSByteCodeGenerator::EmitUnaryLogNot( ParseNode \* pnode )

{

ParseNode\* rhs = ParserWrapper::GetUnaryNode( pnode );

int count = 1;

while( rhs->nop == knopLogNot )

{

++count;

rhs = ParserWrapper::GetUnaryNode( rhs );

}

const EmitExpressionInfo& rhsEmit = Emit( rhs );

const AsmJsType& rType = rhsEmit.type;

StartStatement(pnode);

EmitExpressionInfo emitInfo( AsmJsType::Signed );

if( rType.isInt() )

{

CheckNodeLocation( rhsEmit, int );

RegSlot dst = GetAndReleaseUnaryLocations<int>( &rhsEmit );

if( count & 1 )

{

// do the conversion only if we have an odd number of the operator

mWriter.AsmReg2( OpCodeAsmJs::LogNot\_Int, dst, rhsEmit.location );

}

else

{

// otherwise, make sure the result is 0|1

mWriter.AsmReg2( OpCodeAsmJs::Conv\_ITB, dst, rhsEmit.location );

}

emitInfo.location = dst;

}

else

{

throw AsmJsCompilationException( L"Type not supported for unary !" );

}

EndStatement(pnode);

return emitInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitBooleanExpression( ParseNode\* expr, Js::ByteCodeLabel trueLabel, Js::ByteCodeLabel falseLabel )

{

switch( expr->nop )

{

case knopLogNot:{

const EmitExpressionInfo& info = EmitBooleanExpression( expr->sxUni.pnode1, falseLabel, trueLabel );

return info;

break;

}

// case knopEq:

// case knopNe:

// case knopLt:

// case knopLe:

// case knopGe:

// case knopGt:

// byteCodeGenerator->StartStatement( expr );

// EmitBinaryOpnds( expr->sxBin.pnode1, expr->sxBin.pnode2, byteCodeGenerator, funcInfo );

// funcInfo->ReleaseLoc( expr->sxBin.pnode2 );

// funcInfo->ReleaseLoc( expr->sxBin.pnode1 );

// mWriter.BrReg2( nopToOp[expr->nop], trueLabel, expr->sxBin.pnode1->location,

// expr->sxBin.pnode2->location );

// mWriter.AsmBr( falseLabel );

// byteCodeGenerator->EndStatement( expr );

// break;

// case knopName:

// byteCodeGenerator->StartStatement( expr );

// Emit( expr, byteCodeGenerator, funcInfo, false );

// mWriter.BrReg1( Js::OpCode::BrTrue\_A, trueLabel, expr->location );

// mWriter.AsmBr( falseLabel );

// byteCodeGenerator->EndStatement( expr );

// break;

default:{

const EmitExpressionInfo& info = Emit( expr );

if( !info.type.isInt() )

{

throw AsmJsCompilationException( L"Comparison expressions must be type signed" );

}

mWriter.AsmBrReg1( Js::OpCodeAsmJs::BrTrue\_Int, trueLabel, info.location );

mWriter.AsmBr( falseLabel );

return info;

break;

}

}

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitIf( ParseNode \* pnode )

{

Js::ByteCodeLabel trueLabel = mWriter.DefineLabel();

Js::ByteCodeLabel falseLabel = mWriter.DefineLabel();

const EmitExpressionInfo& boolInfo = EmitBooleanExpression( pnode->sxIf.pnodeCond, trueLabel, falseLabel );

mFunction->ReleaseLocation<int>( &boolInfo );

mWriter.MarkAsmJsLabel( trueLabel );

const EmitExpressionInfo& trueInfo = Emit( pnode->sxIf.pnodeTrue );

mFunction->ReleaseLocationGeneric( &trueInfo );

if( pnode->sxIf.pnodeFalse != nullptr )

{

// has else clause

Js::ByteCodeLabel skipLabel = mWriter.DefineLabel();

// Record the branch bytecode offset

mWriter.RecordStatementAdjustment( Js::FunctionBody::SAT\_FromCurrentToNext );

// then clause skips else clause

mWriter.AsmBr( skipLabel );

// generate code for else clause

mWriter.MarkAsmJsLabel( falseLabel );

const EmitExpressionInfo& falseInfo = Emit( pnode->sxIf.pnodeFalse );

mFunction->ReleaseLocationGeneric( &falseInfo );

mWriter.MarkAsmJsLabel( skipLabel );

}

else

{

mWriter.MarkAsmJsLabel( falseLabel );

}

if( pnode->emitLabels )

{

mWriter.MarkAsmJsLabel( pnode->sxStmt.breakLabel );

}

return EmitExpressionInfo( AsmJsType::Void );

}

Js::EmitExpressionInfo AsmJSByteCodeGenerator::EmitLoop( ParseNode \*loopNode, ParseNode \*cond, ParseNode \*body, ParseNode \*incr, BOOL doWhile /\*= false \*/ )

{

// Need to increment loop count whether we are going to profile or not for HasLoop()

StartStatement(loopNode);

Js::ByteCodeLabel loopEntrance = mWriter.DefineLabel();

Js::ByteCodeLabel continuePastLoop = mWriter.DefineLabel();

uint loopId = mWriter.EnterLoop( loopEntrance );

loopNode->sxLoop.loopId = loopId;

EndStatement(loopNode);

if( doWhile )

{

const EmitExpressionInfo& bodyInfo = Emit( body );

mFunction->ReleaseLocationGeneric( &bodyInfo );

if( loopNode->emitLabels )

{

mWriter.MarkAsmJsLabel( loopNode->sxStmt.continueLabel );

}

if( !ByteCodeGenerator::IsFalse( cond ) )

{

const EmitExpressionInfo& condInfo = EmitBooleanExpression( cond, loopEntrance, continuePastLoop );

mFunction->ReleaseLocationGeneric( &condInfo );

}

}

else

{

if( cond )

{

Js::ByteCodeLabel trueLabel = mWriter.DefineLabel();

const EmitExpressionInfo& condInfo = EmitBooleanExpression( cond, trueLabel, continuePastLoop );

mFunction->ReleaseLocationGeneric( &condInfo );

mWriter.MarkAsmJsLabel( trueLabel );

}

const EmitExpressionInfo& bodyInfo = Emit( body );

mFunction->ReleaseLocationGeneric( &bodyInfo );

if( loopNode->emitLabels )

{

mWriter.MarkAsmJsLabel( loopNode->sxStmt.continueLabel );

}

if( incr != NULL )

{

const EmitExpressionInfo& incrInfo = Emit( incr );

mFunction->ReleaseLocationGeneric( &incrInfo );

}

mWriter.AsmBr( loopEntrance );

}

mWriter.MarkAsmJsLabel( continuePastLoop );

if( loopNode->emitLabels )

{

mWriter.MarkAsmJsLabel( loopNode->sxStmt.breakLabel );

}

mWriter.ExitLoop( loopId );

return EmitExpressionInfo( AsmJsType::Void );

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitQMark( ParseNode \* pnode )

{

StartStatement(pnode->sxTri.pnode1);

Js::ByteCodeLabel trueLabel = mWriter.DefineLabel();

Js::ByteCodeLabel falseLabel = mWriter.DefineLabel();

Js::ByteCodeLabel skipLabel = mWriter.DefineLabel();

EndStatement(pnode->sxTri.pnode1);

const EmitExpressionInfo& boolInfo = EmitBooleanExpression( pnode->sxTri.pnode1, trueLabel, falseLabel );

mFunction->ReleaseLocationGeneric( &boolInfo );

RegSlot intReg = mFunction->AcquireTmpRegister<int>();

RegSlot doubleReg = mFunction->AcquireTmpRegister<double>();

RegSlot floatReg = mFunction->AcquireTmpRegister<float>();

EmitExpressionInfo emitInfo( AsmJsType::Void );

mWriter.MarkAsmJsLabel( trueLabel );

const EmitExpressionInfo& trueInfo = Emit( pnode->sxTri.pnode2 );

StartStatement(pnode->sxTri.pnode2);

if( trueInfo.type.isInt() )

{

mWriter.AsmReg2( Js::OpCodeAsmJs::Ld\_Int, intReg, trueInfo.location );

mFunction->ReleaseLocation<int>( &trueInfo );

mFunction->ReleaseTmpRegister<double>(doubleReg);

mFunction->ReleaseTmpRegister<float>(floatReg);

emitInfo.location = intReg;

emitInfo.type = AsmJsType::Int;

}

else if( trueInfo.type.isDouble() )

{

mWriter.AsmReg2( Js::OpCodeAsmJs::Ld\_Db, doubleReg, trueInfo.location );

mFunction->ReleaseLocation<double>( &trueInfo );

mFunction->ReleaseTmpRegister<int>( intReg );

mFunction->ReleaseTmpRegister<float>(floatReg);

emitInfo.location = doubleReg;

emitInfo.type = AsmJsType::Double;

}

else if (trueInfo.type.isFloat())

{

mWriter.AsmReg2(Js::OpCodeAsmJs::Ld\_Flt, floatReg, trueInfo.location);

mFunction->ReleaseLocation<float>(&trueInfo);

mFunction->ReleaseTmpRegister<int>(intReg);

mFunction->ReleaseTmpRegister<double>(doubleReg);

emitInfo.location = floatReg;

emitInfo.type = AsmJsType::Float;

}

else

{

throw AsmJsCompilationException(L"Conditional expressions must be of type int, double, or float");

}

mWriter.AsmBr( skipLabel );

EndStatement(pnode->sxTri.pnode2);

mWriter.MarkAsmJsLabel( falseLabel );

const EmitExpressionInfo& falseInfo = Emit( pnode->sxTri.pnode3 );

StartStatement(pnode->sxTri.pnode3);

if( falseInfo.type.isInt() )

{

if( !trueInfo.type.isInt() )

{

throw AsmJsCompilationException( L"Conditional expressions results must be the same type" );

}

mWriter.AsmReg2( Js::OpCodeAsmJs::Ld\_Int, intReg, falseInfo.location );

mFunction->ReleaseLocation<int>( &falseInfo );

}

else if( falseInfo.type.isDouble() )

{

if( !trueInfo.type.isDouble() )

{

throw AsmJsCompilationException( L"Conditional expressions results must be the same type" );

}

mWriter.AsmReg2( Js::OpCodeAsmJs::Ld\_Db, doubleReg, falseInfo.location );

mFunction->ReleaseLocation<double>( &falseInfo );

}

else if(falseInfo.type.isFloat())

{

if (!trueInfo.type.isFloat())

{

throw AsmJsCompilationException(L"Conditional expressions results must be the same type");

}

mWriter.AsmReg2(Js::OpCodeAsmJs::Ld\_Flt, floatReg, falseInfo.location);

mFunction->ReleaseLocation<float>(&falseInfo);

}

else

{

throw AsmJsCompilationException(L"Conditional expressions must be of type int, double, or float");

}

mWriter.MarkAsmJsLabel( skipLabel );

EndStatement(pnode->sxTri.pnode3);

return emitInfo;

}

EmitExpressionInfo AsmJSByteCodeGenerator::EmitSwitch( ParseNode \* pnode )

{

BOOL fHasDefault = false;

Assert( pnode->sxSwitch.pnodeVal != NULL );

const EmitExpressionInfo& valInfo = Emit( pnode->sxSwitch.pnodeVal );

if( !valInfo.type.isSigned() )

{

throw AsmJsCompilationException( L"Switch value must be type Signed, FixNum" );

}

RegSlot regVal = GetAndReleaseUnaryLocations<int>( &valInfo );

StartStatement(pnode);

mWriter.AsmReg2(OpCodeAsmJs::BeginSwitch\_Int, regVal, valInfo.location);

EndStatement(pnode);

// TODO: if all cases are compile-time constants, emit a switch statement in the byte

// code so the BE can optimize it.

ParseNode \*pnodeCase;

for( pnodeCase = pnode->sxSwitch.pnodeCases; pnodeCase; pnodeCase = pnodeCase->sxCase.pnodeNext )

{

// Jump to the first case body if this one doesn't match. Make sure any side-effects of the case

// expression take place regardless.

pnodeCase->sxCase.labelCase = mWriter.DefineLabel();

if( pnodeCase == pnode->sxSwitch.pnodeDefault )

{

fHasDefault = true;

continue;

}

ParseNode\* caseExpr = pnodeCase->sxCase.pnodeExpr;

if ((caseExpr->nop != knopInt || (caseExpr->sxInt.lw >> 31) > 1) && !ParserWrapper::IsMinInt(caseExpr))

{

throw AsmJsCompilationException( L"Switch case value must be int in the range [-2^31, 2^31)" );

}

const EmitExpressionInfo& caseExprInfo = Emit( pnodeCase->sxCase.pnodeExpr );

mWriter.AsmBrReg2( OpCodeAsmJs::Case\_Int, pnodeCase->sxCase.labelCase, regVal, caseExprInfo.location );

// do not need to release location because int constants cannot be released

}

// No explicit case value matches. Jump to the default arm (if any) or break out altogether.

if( fHasDefault )

{

mWriter.AsmBr( pnode->sxSwitch.pnodeDefault->sxCase.labelCase, OpCodeAsmJs::EndSwitch\_Int );

}

else

{

if( !pnode->emitLabels )

{

pnode->sxStmt.breakLabel = mWriter.DefineLabel();

}

mWriter.AsmBr( pnode->sxStmt.breakLabel, OpCodeAsmJs::EndSwitch\_Int );

}

// Now emit the case arms to which we jump on matching a case value.

for( pnodeCase = pnode->sxSwitch.pnodeCases; pnodeCase; pnodeCase = pnodeCase->sxCase.pnodeNext )

{

mWriter.MarkAsmJsLabel( pnodeCase->sxCase.labelCase );

const EmitExpressionInfo& caseBodyInfo = Emit( pnodeCase->sxCase.pnodeBody );

mFunction->ReleaseLocationGeneric( &caseBodyInfo );

}

mFunction->ReleaseTmpRegister<int>( regVal );

if( !fHasDefault || pnode->emitLabels )

{

mWriter.MarkAsmJsLabel( pnode->sxStmt.breakLabel );

}

return EmitExpressionInfo( AsmJsType::Void );

}

void AsmJSByteCodeGenerator::EmitEmptyByteCode(FuncInfo \* funcInfo, ByteCodeGenerator \* byteCodeGen, ParseNode \* functionNode)

{

funcInfo->byteCodeFunction->SetGrfscr(byteCodeGen->GetFlags());

funcInfo->byteCodeFunction->SetSourceInfo(byteCodeGen->GetCurrentSourceIndex(),

funcInfo->root,

!!(byteCodeGen->GetFlags() & fscrEvalCode),

((byteCodeGen->GetFlags() & fscrDynamicCode) && !(byteCodeGen->GetFlags() & fscrEvalCode)));

FunctionBody \* functionBody = funcInfo->byteCodeFunction->GetFunctionBody();

class AutoCleanup

{

private:

FunctionBody \* mFunctionBody;

ByteCodeGenerator \* mByteCodeGen;

public:

AutoCleanup(FunctionBody \* functionBody, ByteCodeGenerator \* byteCodeGen) : mFunctionBody(functionBody), mByteCodeGen(byteCodeGen)

{

}

void Done()

{

mFunctionBody = nullptr;

}

~AutoCleanup()

{

if (mFunctionBody)

{

mFunctionBody->ResetByteCodeGenState();

mByteCodeGen->Writer()->Reset();

}

}

} autoCleanup(functionBody, byteCodeGen);

byteCodeGen->Writer()->Begin(byteCodeGen, functionBody, byteCodeGen->GetAllocator(), false, false);

byteCodeGen->Writer()->StartStatement(functionNode, 0);

byteCodeGen->Writer()->Empty(OpCode::Nop);

byteCodeGen->Writer()->EndStatement(functionNode);

byteCodeGen->Writer()->End();

autoCleanup.Done();

}

void AsmJSByteCodeGenerator::StartStatement(ParseNode\* pnode)

{

mWriter.StartStatement(pnode, 0);

// Output::Print( L"%\*s+%d\n",tab, " ", pnode->ichMin );

// ++tab;

}

void AsmJSByteCodeGenerator::EndStatement(ParseNode\* pnode)

{

mWriter.EndStatement(pnode);

// Output::Print( L"%\*s-%d\n",tab, " ", pnode->ichMin );

// --tab;

}

// int tab = 0;

void AsmJSByteCodeGenerator::LoadModuleInt( RegSlot dst, RegSlot index )

{

mWriter.AsmSlot(OpCodeAsmJs::LdSlot\_Int, dst, AsmJsFunctionMemory::ModuleEnvRegister, index + (int32)(mCompiler->GetIntOffset() / INT\_SLOTS\_SPACE + 0.5));

}

void AsmJSByteCodeGenerator::LoadModuleFloat(RegSlot dst, RegSlot index)

{

mWriter.AsmSlot(OpCodeAsmJs::LdSlot\_Flt, dst, AsmJsFunctionMemory::ModuleEnvRegister, index + (int32)(mCompiler->GetFloatOffset() / FLOAT\_SLOTS\_SPACE + 0.5));

}

void AsmJSByteCodeGenerator::LoadModuleDouble( RegSlot dst, RegSlot index )

{

mWriter.AsmSlot(OpCodeAsmJs::LdSlot\_Db, dst, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetDoubleOffset() / DOUBLE\_SLOTS\_SPACE);

}

void AsmJSByteCodeGenerator::LoadModuleFFI( RegSlot dst, RegSlot index )

{

mWriter.AsmSlot(OpCodeAsmJs::LdSlot, dst, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetFFIOffset());

}

void AsmJSByteCodeGenerator::LoadModuleFunction( RegSlot dst, RegSlot index )

{

mWriter.AsmSlot(OpCodeAsmJs::LdSlot, dst, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetFuncOffset());

}

void AsmJSByteCodeGenerator::LoadModuleFunctionTable( RegSlot dst, RegSlot FuncTableIndex, RegSlot FuncIndexLocation )

{

mWriter.AsmSlot( OpCodeAsmJs::LdSlotArr, AsmJsFunctionMemory::ModuleSlotRegister, AsmJsFunctionMemory::ModuleEnvRegister, FuncTableIndex+mCompiler->GetFuncPtrOffset() );

mWriter.AsmSlot( OpCodeAsmJs::LdArr\_Func, dst, AsmJsFunctionMemory::ModuleSlotRegister, FuncIndexLocation );

}

void AsmJSByteCodeGenerator::SetModuleInt( Js::RegSlot dst, RegSlot src )

{

mWriter.AsmSlot(OpCodeAsmJs::StSlot\_Int, src, AsmJsFunctionMemory::ModuleEnvRegister, dst + (int32)(mCompiler->GetIntOffset() / INT\_SLOTS\_SPACE + 0.5));

}

void AsmJSByteCodeGenerator::SetModuleFloat(Js::RegSlot dst, RegSlot src)

{

mWriter.AsmSlot(OpCodeAsmJs::StSlot\_Flt, src, AsmJsFunctionMemory::ModuleEnvRegister, dst + (int32)(mCompiler->GetFloatOffset() / FLOAT\_SLOTS\_SPACE + 0.5));

}

void AsmJSByteCodeGenerator::SetModuleDouble( Js::RegSlot dst, RegSlot src )

{

mWriter.AsmSlot(OpCodeAsmJs::StSlot\_Db, src, AsmJsFunctionMemory::ModuleEnvRegister, dst + mCompiler->GetDoubleOffset() / DOUBLE\_SLOTS\_SPACE);

}

void AsmJSByteCodeGenerator::LoadModuleSimd(RegSlot dst, RegSlot index, AsmJsVarType type)

{

switch (type.which())

{

case AsmJsVarType::Int32x4:

mWriter.AsmSlot(OpCodeAsmJs::Simd128\_LdSlot\_I4, dst, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetSimdOffset());

break;

case AsmJsVarType::Float32x4:

mWriter.AsmSlot(OpCodeAsmJs::Simd128\_LdSlot\_F4, dst, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetSimdOffset());

break;

case AsmJsVarType::Float64x2:

mWriter.AsmSlot(OpCodeAsmJs::Simd128\_LdSlot\_D2, dst, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetSimdOffset());

break;

default:

Assert(UNREACHED);

}

}

void AsmJSByteCodeGenerator::SetModuleSimd(RegSlot index, RegSlot src, AsmJsVarType type)

{

switch (type.which())

{

case AsmJsVarType::Int32x4:

mWriter.AsmSlot(OpCodeAsmJs::Simd128\_StSlot\_I4, src, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetSimdOffset());

break;

case AsmJsVarType::Float32x4:

mWriter.AsmSlot(OpCodeAsmJs::Simd128\_StSlot\_F4, src, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetSimdOffset());

break;

case AsmJsVarType::Float64x2:

mWriter.AsmSlot(OpCodeAsmJs::Simd128\_StSlot\_D2, src, AsmJsFunctionMemory::ModuleEnvRegister, index + mCompiler->GetSimdOffset());

break;

default:

Assert(UNREACHED);

}

}

void AsmJSByteCodeGenerator::LoadSimd(RegSlot dst, RegSlot src, AsmJsVarType type)

{

switch (type.which())

{

case AsmJsVarType::Int32x4:

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_Ld\_I4, dst, src);

break;

case AsmJsVarType::Float32x4:

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_Ld\_F4, dst, src);

break;

case AsmJsVarType::Float64x2:

mWriter.AsmReg2(OpCodeAsmJs::Simd128\_Ld\_D2, dst, src);

break;

default:

Assert(UNREACHED);

}

}

void AsmJsFunctionCompilation::CleanUp()

{

if( mGenerator && mGenerator->mInfo )

{

FunctionBody\* body = mGenerator->mFunction->GetFuncBody();

if( body )

{

body->ResetByteCodeGenState();

}

mGenerator->mWriter.Reset();

}

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

// Information about the expression that has been emitted

struct EmitExpressionInfo

{

EmitExpressionInfo( RegSlot location\_, const AsmJsType& type\_ ) :

location( location\_ ), type( type\_ )

{

}

EmitExpressionInfo( const AsmJsType& type\_ ) :

location( Constants::NoRegister ), type( type\_ )

{

}

EmitExpressionInfo():

location( Constants::NoRegister ), type( AsmJsType::Void )

{

}

RegSlot location;

AsmJsType type;

};

/// AutoPtr cleanup for asmjs bytecode compilation

class AsmJsFunctionCompilation

{

AsmJSByteCodeGenerator\* mGenerator;

public:

AsmJsFunctionCompilation( AsmJSByteCodeGenerator\* gen ) :

mGenerator( gen )

{

}

~AsmJsFunctionCompilation()

{

CleanUp();

}

void CleanUp();

void FinishCompilation()

{

mGenerator = nullptr;

}

};

class AsmJSByteCodeGenerator

{

friend AsmJsFunctionCompilation;

AsmJsFunc\* mFunction;

FuncInfo\* mInfo;

AsmJsModuleCompiler\* mCompiler;

// Reference to non-asmjs bytecode gen. Needed to bind fields for SIMD.js code

ByteCodeGenerator\* mByteCodeGenerator;

AsmJsByteCodeWriter mWriter;

int mNestedCallCount;

bool mIsCallLegal;

public:

AsmJSByteCodeGenerator(AsmJsFunc\* func, AsmJsModuleCompiler\* compiler);

static void EmitEmptyByteCode(FuncInfo\* funcInfo, ByteCodeGenerator\* byteCodeGen, ParseNode\* funcNode);

bool EmitOneFunction();

private:

ArenaAllocator mAllocator;

bool BlockHasOwnScope( ParseNode\* pnodeBlock );

void PrintAsmJsCompilationError(\_\_out\_ecount(256) wchar\_t\* msg);

void DefineLabels();

void EmitAsmJsFunctionBody();

void EmitTopLevelStatement( ParseNode \*stmt );

EmitExpressionInfo Emit( ParseNode \*pnode );

EmitExpressionInfo EmitIdentifier( ParseNode \* pnode );

EmitExpressionInfo EmitLdArrayBuffer( ParseNode \* pnode );

enum TypedArrayEmitType{

LoadTypedArray,

StoreTypedArray,

};

EmitExpressionInfo EmitTypedArrayIndex(ParseNode\* indexNode, OpCodeAsmJs &op, uint32 &indexSlot, ArrayBufferView::ViewType viewType, TypedArrayEmitType emitType);

EmitExpressionInfo EmitAssignment( ParseNode \* pnode );

EmitExpressionInfo EmitReturn( ParseNode \* pnode );

EmitExpressionInfo EmitCall( ParseNode \* pnode, AsmJsRetType expectedType = AsmJsRetType::Void );

EmitExpressionInfo EmitMathBuiltin( ParseNode\* pnode, AsmJsMathFunction\* mathFunction, AsmJsRetType expectedType );

EmitExpressionInfo EmitMinMax(ParseNode\* pnode, AsmJsMathFunction\* mathFunction, AsmJsRetType expectedType);

EmitExpressionInfo EmitUnaryPos( ParseNode \* pnode );

EmitExpressionInfo EmitUnaryNeg( ParseNode \* pnode );

EmitExpressionInfo EmitUnaryNot( ParseNode \* pnode );

EmitExpressionInfo EmitUnaryLogNot( ParseNode \* pnode );

EmitExpressionInfo EmitBinaryMultiType( ParseNode \* pnode, enum EBinaryMathOpCodes op );

EmitExpressionInfo EmitBinaryInt( ParseNode \* pnode, OpCodeAsmJs op );

EmitExpressionInfo EmitQMark( ParseNode \* pnode );

EmitExpressionInfo EmitSwitch( ParseNode \* pnode );

EmitExpressionInfo EmitBinaryComparator( ParseNode \* pnode, enum EBinaryComparatorOpCodes op);

EmitExpressionInfo EmitLoop( ParseNode \*loopNode, ParseNode \*cond, ParseNode \*body, ParseNode \*incr, BOOL doWhile = false );

EmitExpressionInfo EmitIf( ParseNode \* pnode );

EmitExpressionInfo EmitBooleanExpression( ParseNode\* pnodeCond, Js::ByteCodeLabel trueLabel, Js::ByteCodeLabel falseLabel );

EmitExpressionInfo\* EmitSimdBuiltinArguments(ParseNode\* pnode, AsmJsFunctionDeclaration\* func, \_\_out\_ecount(pnode->sxCall.argCount) AsmJsType \*argsTypes, EmitExpressionInfo \*argsInfo);

bool ValidateSimdFieldAccess(PropertyName field, const AsmJsType& receiverType, OpCodeAsmJs &op);

EmitExpressionInfo EmitDotExpr(ParseNode\* pnode);

EmitExpressionInfo EmitSimdBuiltin(ParseNode\* pnode, AsmJsSIMDFunction\* simdFunction, AsmJsRetType expectedType);

EmitExpressionInfo EmitSimdLoadStoreBuiltin(ParseNode\* pnode, AsmJsSIMDFunction\* simdFunction, AsmJsRetType expectedType);

void FinalizeRegisters( FunctionBody\* byteCodeFunction );

void LoadAllConstants();

void StartStatement(ParseNode\* pnode);

void EndStatement(ParseNode\* pnode);

// Emits the bytecode to load from the module

// dst is the location of the variable in the function

// index is the location of the target in the module's table

void LoadModuleInt(RegSlot dst, RegSlot index); // dst points to the IntRegisterSpace

void LoadModuleFloat(RegSlot dst, RegSlot index); // dst points to the FloatRegisterSpace

void LoadModuleDouble( RegSlot dst, RegSlot index ); // dst points to the DoubleRegisterSpace

void LoadModuleFFI( RegSlot dst, RegSlot index ); // dst points to a Var

void LoadModuleFunction( RegSlot dst, RegSlot index ); // dst points to a Var

void LoadModuleFunctionTable( RegSlot dst, RegSlot FuncTableIndex, RegSlot FuncIndexLocation ); // dst points to a Var

// Emits the bytecode to set a variable int the module

// dst is the location of the variable in the module's table

// src is the location of the variable in the function

void SetModuleInt(Js::RegSlot dst, RegSlot src);

void SetModuleFloat(Js::RegSlot dst, RegSlot src);

void SetModuleDouble( Js::RegSlot dst, RegSlot src );

void LoadModuleSimd(RegSlot dst, RegSlot index, AsmJsVarType type);

void SetModuleSimd(RegSlot dst, RegSlot src, AsmJsVarType type);

void LoadSimd(RegSlot dst, RegSlot src, AsmJsVarType type);

bool IsFRound(AsmJsMathFunction\* sym);

/// TODO:: Finish removing references to old bytecode generator

ByteCodeGenerator\* GetOldByteCodeGenerator() const

{

return mByteCodeGenerator;

}

bool IsSimdjsEnabled()

{

return mFunction->GetFuncBody()->GetScriptContext()->GetConfig()->IsSimdjsEnabled();

}

// try to reuse a tmp register or acquire a new one

// also takes care of releasing tmp register

template<typename T>

RegSlot GetAndReleaseBinaryLocations( const EmitExpressionInfo\* lhs, const EmitExpressionInfo\* rhs )

{

RegSlot tmpRegToUse;

if( mFunction->IsTmpLocation<T>( lhs ) )

{

tmpRegToUse = lhs->location;

mFunction->ReleaseLocation<T>( rhs );

}

else if( mFunction->IsTmpLocation<T>( rhs ) )

{

tmpRegToUse = rhs->location;

mFunction->ReleaseLocation<T>( lhs );

}

else

{

tmpRegToUse = mFunction->AcquireTmpRegister<T>();

mFunction->ReleaseLocation<T>( rhs );

mFunction->ReleaseLocation<T>( lhs );

}

return tmpRegToUse;

}

template<typename T>

RegSlot GetAndReleaseUnaryLocations( const EmitExpressionInfo\* rhs )

{

RegSlot tmpRegToUse;

if( mFunction->IsTmpLocation<T>( rhs ) )

{

tmpRegToUse = rhs->location;

}

else

{

tmpRegToUse = mFunction->AcquireTmpRegister<T>();

mFunction->ReleaseLocation<T>( rhs );

}

return tmpRegToUse;

}

};

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

#include "CodeGenAllocators.h"

namespace Js

{

AsmJsCodeGenerator::AsmJsCodeGenerator( ScriptContext\* scriptContext ) :

mScriptContext( scriptContext )

,mPageAllocator(scriptContext->GetThreadContext()->GetPageAllocator())

{

//use the same foreground allocator as NativeCodeGen

mForegroundAllocators = GetForegroundAllocator(scriptContext->GetNativeCodeGenerator(),mPageAllocator);

mEncoder.SetPageAllocator( mPageAllocator );

mEncoder.SetCodeGenAllocator( mForegroundAllocators );

}

void AsmJsCodeGenerator::CodeGen( FunctionBody\* functionBody )

{

AsmJsFunctionInfo\* asmInfo = functionBody->GetAsmJsFunctionInfo();

Assert( asmInfo );

void\* address = mEncoder.Encode( functionBody );

if( address )

{

FunctionEntryPointInfo\* funcEntrypointInfo = (FunctionEntryPointInfo\*)functionBody->GetDefaultEntryPointInfo();

EntryPointInfo\* entrypointInfo = (EntryPointInfo\*)funcEntrypointInfo;

Assert(entrypointInfo->GetIsAsmJSFunction());

//set entrypointinfo address and nativeAddress with TJ address

entrypointInfo->address = address;

entrypointInfo->SetNativeAddress((void\*)address);

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

funcEntrypointInfo->SetIsTJMode(true);

#endif

}

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

class ScriptContext;

class AsmJsCodeGenerator

{

ScriptContext\* mScriptContext;

CodeGenAllocators\* mForegroundAllocators;

PageAllocator \* mPageAllocator;

AsmJsEncoder mEncoder;

public:

AsmJsCodeGenerator( ScriptContext\* scriptContext );

void CodeGen( FunctionBody\* functionBody );

};

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

#include "..\Backend\CodeGenAllocators.h"

#ifdef DBG\_DUMP

#include "ByteCode\ByteCodeDumper.h"

#include "ByteCode\AsmJSByteCodeDumper.h"

#endif

#include "AsmJSEncoder.inl"

#if DBG\_DUMP

#include "ByteCode\OpCodeUtilAsmJs.h"

#endif

namespace Js

{

template<> int AsmJsEncoder::GetOffset<int>() const{return mIntOffset;}

template<> int AsmJsEncoder::GetOffset<Var>() const{return AsmJsJitTemplate::Globals::StackVarCount \* sizeof( Var );}

template<> int AsmJsEncoder::GetOffset<double>() const{ return mDoubleOffset; }

template<> int AsmJsEncoder::GetOffset<float>() const{ return mFloatOffset; }

template<> int AsmJsEncoder::GetOffset<AsmJsSIMDValue>() const{ return mSimdOffset; }

template<>

void AsmJsEncoder::ReadOpTemplate<Js::SmallLayout>( OpCodeAsmJs op )

{

switch( op )

{

#define DEF2(x, op, func) PROCESS\_ENCODE\_##x(op, func)

#define DEF3(x, op, func, y) PROCESS\_ENCODE\_##x(op, func, y)

#define DEF2\_WMS(x, op, func) PROCESS\_ENCODE\_##x##\_COMMON(op, func, \_Small)

#define DEF3\_WMS(x, op, func, y) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Small)

#define DEF4\_WMS(x, op, func, y, t) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Small, t)

#define EXDEF2(x, op, func) PROCESS\_ENCODE\_##x(op, func)

#define EXDEF3(x, op, func, y) PROCESS\_ENCODE\_##x(op, func, y)

#define EXDEF2\_WMS(x, op, func) PROCESS\_ENCODE\_##x##\_COMMON(op, func, \_Small)

#define EXDEF3\_WMS(x, op, func, y) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Small)

#define EXDEF4\_WMS(x, op, func, y, t) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Small, t)

#include "AsmJSEncoderHandler.inl"

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

#if DBG\_DUMP

Output::Print( L"Dispatch to bad opcode : %s\n", OpCodeUtilAsmJs::GetOpCodeName(op));

Output::Flush();

#endif

Assert( false );

\_\_assume( false );

};

}

template<>

void AsmJsEncoder::ReadOpTemplate<Js::MediumLayout>( OpCodeAsmJs op )

{

switch( op )

{

#define DEF2\_WMS(x, op, func) PROCESS\_ENCODE\_##x##\_COMMON(op, func, \_Medium)

#define DEF3\_WMS(x, op, func, y) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Medium)

#define DEF4\_WMS(x, op, func, y, t) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Medium, t)

#define EXDEF2\_WMS(x, op, func) PROCESS\_ENCODE\_##x##\_COMMON(op, func, \_Medium)

#define EXDEF3\_WMS(x, op, func, y) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Medium)

#define EXDEF4\_WMS(x, op, func, y, t) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Medium, t)

#include "AsmJSEncoderHandler.inl"

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

#if DBG\_DUMP

Output::Print( L"Dispatch to bad opcode : %s\n", OpCodeUtilAsmJs::GetOpCodeName(op));

Output::Flush();

#endif

Assert( false );

\_\_assume( false );

};

}

template<>

void AsmJsEncoder::ReadOpTemplate<Js::LargeLayout>( OpCodeAsmJs op )

{

switch( op )

{

#define DEF2\_WMS(x, op, func) PROCESS\_ENCODE\_##x##\_COMMON(op, func, \_Large)

#define DEF3\_WMS(x, op, func, y) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Large)

#define DEF4\_WMS(x, op, func, y, t) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Large, t)

#define EXDEF2\_WMS(x, op, func) PROCESS\_ENCODE\_##x##\_COMMON(op, func, \_Large)

#define EXDEF3\_WMS(x, op, func, y) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Large)

#define EXDEF4\_WMS(x, op, func, y, t) PROCESS\_ENCODE\_##x##\_COMMON(op, func, y, \_Large, t)

#include "AsmJSEncoderHandler.inl"

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

#if DBG\_DUMP

Output::Print( L"Dispatch to bad opcode : %s\n", OpCodeUtilAsmJs::GetOpCodeName(op));

Output::Flush();

#endif

Assert( false );

\_\_assume( false );

};

}

bool AsmJsEncoder::ReadOp()

{

#if DBG\_DUMP

int bytecodeoffset = mReader.GetCurrentOffset();

#endif

LayoutSize layoutSize;

OpCodeAsmJs op = (OpCodeAsmJs)mReader.ReadOp(layoutSize);

ip = mReader.GetIP();

#if DBG\_DUMP

if (PHASE\_TRACE(Js::AsmjsEncoderPhase, mFunctionBody))

{

Output::Print(L"%d.%d:Encoding ",

this->mFunctionBody->GetSourceContextId(),

this->mFunctionBody->GetLocalFunctionId());

AsmJsByteCodeDumper::DumpOp( op, layoutSize, mReader, mFunctionBody );

if( ip != mReader.GetIP() )

{

mReader.SetIP( ip );

}

Output::Print(L" at offset 0x%X (buffer size = 0x%X)\n",

bytecodeoffset, (int)(mPc-mEncodeBuffer));

Output::Flush();

}

#endif

if( op == OpCodeAsmJs::EndOfBlock )

{

Assert(mReader.GetCurrentOffset() == mFunctionBody->GetByteCode()->GetLength());

// last bytecode

return false;

}

switch( layoutSize )

{

case Js::SmallLayout:

ReadOpTemplate<Js::SmallLayout>( op );

break;

case Js::MediumLayout:

ReadOpTemplate<Js::MediumLayout>( op );

break;

case Js::LargeLayout:

ReadOpTemplate<Js::LargeLayout>( op );

break;

default:

break;

}

return true;

}

uint32 AsmJsEncoder::GetEncodeBufferSize(FunctionBody\* functionBody)

{

// TODO: Make a good heuristic; this is completely arbitrary. As we emit each bytecode we can calculate the max instruction size.

return UInt32Math::Add(

UInt32Math::Mul(functionBody->GetByteCodeCount(), 30),

49 /\*prolog\*/ + 11 /\*epilog\*/

);

}

void\* AsmJsEncoder::Encode( FunctionBody\* functionBody )

{

Assert( functionBody );

mFunctionBody = functionBody;

#if DBG\_DUMP

AsmJsJitTemplate::Globals::CurrentEncodingFunction = mFunctionBody;

#endif

AsmJsFunctionInfo\* asmInfo = functionBody->GetAsmJsFunctionInfo();

FunctionEntryPointInfo\* entryPointInfo = ((FunctionEntryPointInfo\*)(functionBody->GetDefaultEntryPointInfo()));

// number of var on the stack + ebp + eip

mIntOffset = asmInfo->GetIntByteOffset() + GetOffset<Var>();

mDoubleOffset = asmInfo->GetDoubleByteOffset() + GetOffset<Var>();

mFloatOffset = asmInfo->GetFloatByteOffset() + GetOffset<Var>();

mSimdOffset = asmInfo->GetSimdByteOffset() + GetOffset<Var>();

NoRecoverMemoryArenaAllocator localAlloc(L"BE-AsmJsEncoder", GetPageAllocator(), Js::Throw::OutOfMemory);

mLocalAlloc = &localAlloc;

mRelocLabelMap = Anew( mLocalAlloc, RelocLabelMap, mLocalAlloc );

mTemplateData = AsmJsJitTemplate::InitTemplateData();

mEncodeBufferSize = GetEncodeBufferSize(functionBody);

mEncodeBuffer = AnewArray((&localAlloc), BYTE, mEncodeBufferSize);

mPc = mEncodeBuffer;

mReader.Create( functionBody );

ip = mReader.GetIP();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if( PHASE\_TRACE( Js::AsmjsEncoderPhase, mFunctionBody ) )

{

Output::Print( L"\n\n" );

functionBody->DumpFullFunctionName();

Output::Print( L"\n StackSize = %d , Offsets: Var = %d, Int = %d, Double = %d\n", mFunctionBody->GetAsmJsFunctionInfo()->GetTotalSizeinBytes(), GetOffset<Var>(), GetOffset<int>(), GetOffset<double>() );

}

#endif

AsmJsRetType retType = asmInfo->GetReturnType();

AsmJsJitTemplate::FunctionEntry::ApplyTemplate( this, mPc );

while( ReadOp() ){}

AsmJsJitTemplate::FunctionExit::ApplyTemplate( this, mPc );

AsmJsJitTemplate::FreeTemplateData( mTemplateData );

#if DBG\_DUMP

AsmJsJitTemplate::Globals::CurrentEncodingFunction = nullptr;

#endif

ApplyRelocs();

ptrdiff\_t codeSize = mPc - mEncodeBuffer;

if( codeSize > 0 )

{

Assert( ::Math::FitsInDWord( codeSize ) );

BYTE \*buffer;

EmitBufferAllocation \*allocation = GetCodeGenAllocator()->emitBufferManager.AllocateBuffer( codeSize, &buffer, false, 0, 0 );

functionBody->GetAsmJsFunctionInfo()->mTJBeginAddress = buffer;

Assert( allocation != nullptr );

if( buffer == nullptr )

Js::Throw::OutOfMemory();

if (!GetCodeGenAllocator()->emitBufferManager.CommitBuffer(allocation, buffer, codeSize, mEncodeBuffer))

{

Js::Throw::OutOfMemory();

}

functionBody->GetScriptContext()->GetThreadContext()->SetValidCallTargetForCFG(buffer);

// TODO: improve this once EntryPoint cleanup work is complete!

#if 0

const wchar\_t \*const functionName = functionBody->GetDisplayName();

const wchar\_t \*const suffix = L"TJ";

wchar\_t functionNameArray[256];

const size\_t functionNameCharLength = functionBody->GetDisplayNameLength();

wcscpy\_s(functionNameArray, 256, functionName);

wcscpy\_s(&functionNameArray[functionNameCharLength], 256 - functionNameCharLength, suffix);

#endif

JS\_ETW(EventWriteMethodLoad(functionBody->GetScriptContext(),

(void \*)buffer,

codeSize,

EtwTrace::GetFunctionId(functionBody),

0 /\* methodFlags - for future use\*/,

MethodType\_Jit,

EtwTrace::GetSourceId(functionBody),

functionBody->GetLineNumber(),

functionBody->GetColumnNumber(),

functionBody->GetDisplayName()));

entryPointInfo->SetTJCodeGenDone(); // set the codegen to done state for TJ

entryPointInfo->SetCodeSize(codeSize);

return buffer;

}

return nullptr;

}

void Js::AsmJsEncoder::AddReloc( const int labelOffset, BYTE\* patchAddr )

{

EncoderRelocLabel\* label = nullptr;

if( mRelocLabelMap->TryGetReference( labelOffset, &label ) )

{

EncoderReloc::New( label, patchAddr, mPc, mLocalAlloc );

}

else

{

EncoderRelocLabel newLabel;

EncoderReloc::New( &newLabel, patchAddr, mPc, mLocalAlloc );

mRelocLabelMap->AddNew( labelOffset, newLabel );

}

}

void AsmJsEncoder::ApplyRelocs()

{

const int size = mRelocLabelMap->Count();

for (int i = 0; i < size ; i++)

{

EncoderRelocLabel\* label = mRelocLabelMap->GetReferenceAt( i );

#if DBG\_DUMP

if( !label->labelSeen )

{

Output::Print( L"Label expected at bytecode offset 0x%x\n", mRelocLabelMap->GetKeyAt( i ) );

Output::Flush();

}

#endif

Assert( label->labelSeen );

EncoderReloc\* reloc = label->relocList;

ptrdiff\_t offset1 = label->pc - mEncodeBuffer;

this->GetAsmJsFunctionInfo()->mbyteCodeTJMap->AddNew(mRelocLabelMap->GetKeyAt(i), offset1);

while( reloc )

{

ptrdiff\_t offset = label->pc - reloc->pc;

\*(ptrdiff\_t\*)reloc->patchAddr = offset;

reloc = reloc->next;

}

}

}

void AsmJsEncoder::EncoderReloc::New( EncoderRelocLabel\* label, BYTE\* \_patchAddr, BYTE\* \_pc, ArenaAllocator\* allocator )

{

AsmJsEncoder::EncoderReloc\* reloc = AnewStruct( allocator, AsmJsEncoder::EncoderReloc );

reloc->next = label->relocList;

label->relocList = reloc;

reloc->patchAddr = \_patchAddr;

reloc->pc = \_pc;

}

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

class AsmJsEncoder

{

struct EncoderRelocLabel;

struct EncoderReloc

{

static void New( EncoderRelocLabel\* label, BYTE\* \_patchAddr, BYTE\* \_pc, ArenaAllocator\* allocator );

BYTE\* patchAddr;

BYTE\* pc;

EncoderReloc\* next;

};

struct EncoderRelocLabel

{

EncoderRelocLabel() :labelSeen( false ), relocList( nullptr ){}

EncoderRelocLabel(BYTE\* \_pc) :labelSeen( true ), pc(\_pc), relocList( nullptr ){}

bool labelSeen : 1;

BYTE\* pc;

EncoderReloc\* relocList;

};

// the key is the bytecode address

typedef JsUtil::BaseDictionary<int, EncoderRelocLabel, ArenaAllocator> RelocLabelMap;

const byte\* ip;

ByteCodeReader mReader;

uint32 mEncodeBufferSize;

BYTE\* mEncodeBuffer;

BYTE\* mPc;

PageAllocator\* mPageAllocator;

CodeGenAllocators\* mForegroundAllocators;

FunctionBody\* mFunctionBody;

RelocLabelMap\* mRelocLabelMap;

ArenaAllocator\* mLocalAlloc;

// Byte offset of first int and double

int mIntOffset, mDoubleOffset, mFloatOffset;

int mSimdOffset;

// architecture dependant data to build templatized JIT

void\* mTemplateData;

public:

void\* Encode( FunctionBody\* functionBody );

void\* GetTemplateData() { return mTemplateData; }

inline PageAllocator\* GetPageAllocator() const{return mPageAllocator;}

inline void SetPageAllocator( PageAllocator\* val ){mPageAllocator = val;}

inline CodeGenAllocators\* GetCodeGenAllocator() const{return mForegroundAllocators;}

inline void SetCodeGenAllocator( CodeGenAllocators\* val ){mForegroundAllocators = val;}

FunctionBody\* GetFunctionBody() { return mFunctionBody; }

private:

void ApplyRelocs();

void AddReloc( const int labelOffset, BYTE\* patchAddr );

uint32 GetEncodeBufferSize(FunctionBody\* functionBody);

AsmJsFunctionInfo\* GetAsmJsFunctionInfo(){ return mFunctionBody->GetAsmJsFunctionInfo(); }

bool ReadOp();

template<LayoutSize T> void ReadOpTemplate( OpCodeAsmJs op );

template<typename T> int GetOffset() const;

template<typename T> int CalculateOffset(int stackLocation) { return stackLocation\*sizeof(T)+GetOffset<T>(); }

void OP\_Label( const unaligned OpLayoutEmpty\* playout );

template <class T> void OP\_LdUndef( const unaligned T\* playout );

template <class T> void OP\_Br( const unaligned T\* playout );

template <class T> void OP\_BrEq( const unaligned T\* playout );

template <class T> void OP\_BrTrue( const unaligned T\* playout );

template <class T> void OP\_Empty( const unaligned T\* playout );

template <class T> void Op\_LdSlot\_Db( const unaligned T\* playout );

template <class T> void Op\_LdSlot\_Int(const unaligned T\* playout);

template <class T> void Op\_LdSlot\_Flt(const unaligned T\* playout);

template <class T> void Op\_StSlot\_Db( const unaligned T\* playout );

template <class T> void Op\_StSlot\_Int(const unaligned T\* playout);

template <class T> void Op\_StSlot\_Flt(const unaligned T\* playout);

template <class T> void Op\_LdConst\_Int(const unaligned T\* playout);

template <class T> void Op\_LdArr ( const unaligned T\* playout );

template <class T> void Op\_LdArrConst( const unaligned T\* playout );

template <class T> void Op\_StArr ( const unaligned T\* playout );

template <class T> void Op\_StArrConst( const unaligned T\* playout );

template <class T> void OP\_SetReturnInt( const unaligned T\* playout );

template <class T> void OP\_SetReturnDouble(const unaligned T\* playout);

template <class T> void OP\_SetReturnFloat(const unaligned T\* playout);

template <class T> void OP\_SetFroundInt(const unaligned T\* playout);

template <class T> void OP\_SetFroundDb(const unaligned T\* playout);

template <class T> void OP\_SetFroundFlt(const unaligned T\* playout);

template <class T> void Op\_Float\_To\_Int(const unaligned T\* playout);

template <class T> void Op\_Float\_To\_Db(const unaligned T\* playout);

template <class T> void Op\_UInt\_To\_Db(const unaligned T\* playout);

template <class T> void Op\_Int\_To\_Db( const unaligned T\* playout );

template <class T> void Op\_Db\_To\_Int( const unaligned T\* playout );

template <class T> void Op\_LdAddr\_Db( const unaligned T\* playout );

template <class T> void OP\_LdSlot( const unaligned T\* playout );

template <class T> void OP\_StartCall( const unaligned T\* playout );

template <class T> void OP\_Call( const unaligned T\* playout );

template <class T> void OP\_ArgOut\_Db( const unaligned T\* playout );

template <class T> void OP\_ArgOut\_Int(const unaligned T\* playout);

template <class T> void OP\_Conv\_VTD( const unaligned T\* playout );

template <class T> void OP\_Conv\_VTI( const unaligned T\* playout );

template <class T> void OP\_Conv\_VTF( const unaligned T\* playout );

template <class T> void OP\_I\_StartCall( const unaligned T\* playout );

template <class T> void OP\_I\_Call( const unaligned T\* playout );

template <class T> void OP\_I\_ArgOut\_Db( const unaligned T\* playout );

template <class T> void OP\_I\_ArgOut\_Int(const unaligned T\* playout);

template <class T> void OP\_I\_ArgOut\_Flt(const unaligned T\* playout);

template <class T> void OP\_I\_Conv\_VTD( const unaligned T\* playout );

template <class T> void OP\_I\_Conv\_VTI( const unaligned T\* playout );

template <class T> void OP\_I\_Conv\_VTF( const unaligned T\* playout );

template <class T> void OP\_AsmJsLoopBody(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdSlotF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdSlotI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdSlotD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_StSlotF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_StSlotI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_StSlotD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FloatsToF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_IntsToI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_DoublesToD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_ReturnF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_ReturnI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_ReturnD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SplatF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SplatI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SplatD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromFloat64x2F4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromInt32x4F4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromFloat32x4I4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromFloat64x2I4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromFloat32x4D2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromInt32x4D2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromFloat32x4BitsD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromInt32x4BitsD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromFloat32x4BitsI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromFloat64x2BitsI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromFloat64x2BitsF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_FromInt32x4BitsF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_AbsF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_AbsD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_NegF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_NegI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_NegD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_RcpF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_RcpD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_RcpSqrtF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_RcpSqrtD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SqrtF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SqrtD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_NotF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_NotI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_AddF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_AddI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_AddD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SubF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SubI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SubD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_MulF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_MulI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_MulD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_DivF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_DivD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_MinF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_MinD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_MaxF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_MaxD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LtF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LtI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LtD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_GtF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_GtI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_GtD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LtEqF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LtEqD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_GtEqF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_GtEqD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_EqF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_EqI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_EqD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_NeqF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_NeqD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_AndF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_AndI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_OrF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_OrI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_XorF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_XorI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SelectF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SelectI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_SelectD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdSignMaskF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdSignMaskI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_LdSignMaskD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_ExtractLaneI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_ExtractLaneF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_ReplaceLaneI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_ReplaceLaneF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_I\_ArgOutF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_I\_ArgOutI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_I\_ArgOutD2(const unaligned T\* playout);

template <class T> void OP\_Simd128\_I\_Conv\_VTF4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_I\_Conv\_VTI4(const unaligned T\* playout);

template <class T> void OP\_Simd128\_I\_Conv\_VTD2(const unaligned T\* playout);

};

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#define PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix) \

CompileAssert(OpCodeInfoAsmJs<OpCodeAsmJs::name>::Layout == OpLayoutTypeAsmJs::layout); \

const unaligned OpLayout##layout##suffix \* playout = mReader.layout##suffix(ip);

#define PROCESS\_ENCODE\_CUSTOM\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

func(playout); \

break; \

}

#define PROCESS\_ENCODE\_CUSTOM(name,func,layout) PROCESS\_ENCODE\_CUSTOM\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_INT2\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<int>(playout->I1) );\

break; \

}

#define PROCESS\_ENCODE\_INT2(name,func,layout) PROCESS\_ENCODE\_INT2\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_INT3\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<int>(playout->I1), CalculateOffset<int>(playout->I2) );\

break; \

}

#define PROCESS\_ENCODE\_INT3(name,func,layout) PROCESS\_ENCODE\_INT3\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_DOUBLE2\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->D0), CalculateOffset<double>(playout->D1) );\

break; \

}

#define PROCESS\_ENCODE\_DOUBLE2(name,func,layout) PROCESS\_ENCODE\_DOUBLE2\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_DOUBLE3\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->D0), CalculateOffset<double>(playout->D1), CalculateOffset<double>(playout->D2) );\

break; \

}

#define PROCESS\_ENCODE\_DOUBLE3(name,func,layout) PROCESS\_ENCODE\_DOUBLE3\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_INT1DOUBLE2\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<double>(playout->D1), CalculateOffset<double>(playout->D2) );\

break; \

}

#define PROCESS\_ENCODE\_INT1DOUBLE2(name,func,layout) PROCESS\_ENCODE\_INT1DOUBLE2\_COMMON(name,func,layout,)

typedef double( \*UnaryDoubleFunc )( double );

#define PROCESS\_ENCODE\_CALLDOUBLE2\_COMMON(name, func, addEsp, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, Double2, suffix); \

int offsets[2] = {CalculateOffset<double>(playout->D0),CalculateOffset<double>(playout->D1)};\

AsmJsJitTemplate::Call\_Db::ApplyTemplate( this, mPc, 2, offsets, ((UnaryDoubleFunc)(func)),addEsp );\

break; \

}

#define PROCESS\_ENCODE\_CALLDOUBLE2(name,func,layout) PROCESS\_ENCODE\_CALLDOUBLE2\_COMMON(name,func,layout,)

typedef double( \*BinaryDoubleFunc )( double, double );

#define PROCESS\_ENCODE\_CALLDOUBLE3\_COMMON(name, func, addEsp, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, Double3, suffix); \

int offsets[3] = {CalculateOffset<double>(playout->D0),CalculateOffset<double>(playout->D1),CalculateOffset<double>(playout->D2)};\

AsmJsJitTemplate::Call\_Db::ApplyTemplate( this, mPc, 3, offsets, ((BinaryDoubleFunc)(func)),addEsp );\

break; \

}

#define PROCESS\_ENCODE\_CALLDOUBLE3(name,func,addEsp) PROCESS\_ENCODE\_CALLDOUBLE3\_COMMON(name,func,addEsp,)

//Floats

#define PROCESS\_ENCODE\_FLOAT2\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F0), CalculateOffset<float>(playout->F1)); \

break; \

}

#define PROCESS\_ENCODE\_FLOAT2(name,func,layout) PROCESS\_ENCODE\_FLOAT2\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_FLOAT3\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F0), CalculateOffset<float>(playout->F1), CalculateOffset<float>(playout->F2)); \

break; \

}

#define PROCESS\_ENCODE\_FLOAT3(name,func,layout) PROCESS\_ENCODE\_FLOAT3\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_INT1FLOAT2\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate(this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<float>(playout->F1), CalculateOffset<float>(playout->F2)); \

break; \

}

#define PROCESS\_ENCODE\_INT1FLOAT2(name,func,layout) PROCESS\_ENCODE\_INT1FLOAT2\_COMMON(name,func,layout,)

typedef float(\*UnaryFloatFunc)(float);

#define PROCESS\_ENCODE\_CALLFLOAT2\_COMMON(name, func, addEsp, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, Float2, suffix); \

int offsets[2] = { CalculateOffset<float>(playout->F0), CalculateOffset<float>(playout->F1) }; \

AsmJsJitTemplate::Call\_Flt::ApplyTemplate(this, mPc, 2, offsets, ((UnaryFloatFunc)(func)), addEsp); \

break; \

}

#define PROCESS\_ENCODE\_CALLFLOAT2(name,func,layout) PROCESS\_ENCODE\_CALLFLOAT2\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_ELEMENTSLOT\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

AsmJsJitTemplate::func::ApplyTemplate( this, mPc, CalculateOffset<Var>(playout->Value), CalculateOffset<Var>(playout->Instance), CalculateOffset<int>(playout->SlotIndex) );\

break; \

}

#define PROCESS\_ENCODE\_ELEMENTSLOT(name,func,layout) PROCESS\_ENCODE\_ELEMENTSLOT\_COMMON(name,func,layout,)

#define PROCESS\_ENCODE\_TYPED\_ARR\_COMMON(name, func, viewType, suffix, type ) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_ENCODE\_READ\_LAYOUT\_ASMJS(name, ElementSlot, suffix); \

AsmJsJitTemplate::func::ApplyTemplate( this, mPc, CalculateOffset<type>(playout->Value), CalculateOffset<int>(playout->SlotIndex), ArrayBufferView::viewType );\

break; \

}

#define PROCESS\_ENCODE\_TYPED\_ARR(name,viewType, func,layout) PROCESS\_ENCODE\_TYPED\_ARR\_COMMON(name,viewType, func,layout,)

namespace Js

{

template <class T>

void AsmJsEncoder::OP\_Empty( const unaligned T\* playout )

{

}

void AsmJsEncoder::OP\_Label( const unaligned OpLayoutEmpty\* playout )

{

const int labelOffset = mReader.GetCurrentOffset() - 1;

AsmJsJitTemplate::Label::ApplyTemplate( this, mPc );

EncoderRelocLabel\* label = nullptr;

if( mRelocLabelMap->TryGetReference( labelOffset,&label ) )

{

label->labelSeen = true;

label->pc = mPc;

}

else

{

EncoderRelocLabel newLabel( mPc );

mRelocLabelMap->AddNew( labelOffset, newLabel );

}

// Check - this should not be needed as we add to the map in Relocs , but the bytecodeoffset is off by 1 in relocs , see if we can work around that

ptrdiff\_t offset = mPc - mEncodeBuffer;

this->GetAsmJsFunctionInfo()->mbyteCodeTJMap->AddNew(mReader.GetCurrentOffset(), offset);

}

template <class T>

void AsmJsEncoder::OP\_LdUndef( const unaligned T\* playout )

{

AsmJsJitTemplate::LdUndef::ApplyTemplate( this, mPc, CalculateOffset<Var>(playout->R0) );

}

template <class T>

void AsmJsEncoder::OP\_Br( const unaligned T\* playout )

{

if( playout->RelativeJumpOffset )

{

const int labelOffset = mReader.GetCurrentOffset() + playout->RelativeJumpOffset;

Assert( playout->RelativeJumpOffset > 0 || mRelocLabelMap->ContainsKey( labelOffset ) );

bool isBackEdge = false;

if (playout->RelativeJumpOffset < 0)

{

isBackEdge = true;

}

BYTE\* relocAddr = nullptr;

AsmJsJitTemplate::Br::ApplyTemplate(this, mPc, &relocAddr, isBackEdge);

Assert( relocAddr );

AddReloc( labelOffset, relocAddr );

}

}

template <class T>

void AsmJsEncoder::OP\_BrTrue( const unaligned T\* playout )

{

if( playout->RelativeJumpOffset )

{

const int labelOffset = mReader.GetCurrentOffset() + playout->RelativeJumpOffset;

Assert( playout->RelativeJumpOffset > 0 || mRelocLabelMap->ContainsKey( labelOffset ) );

bool isBackEdge = false;

if (playout->RelativeJumpOffset < 0)

isBackEdge = true;

BYTE\* relocAddr = nullptr;

AsmJsJitTemplate::BrTrue::ApplyTemplate( this, mPc, CalculateOffset<int>( playout->I1 ), &relocAddr, isBackEdge );

Assert( relocAddr );

AddReloc( labelOffset, relocAddr );

}

}

template <class T>

void AsmJsEncoder::OP\_BrEq( const unaligned T\* playout )

{

if( playout->RelativeJumpOffset )

{

const int labelOffset = mReader.GetCurrentOffset() + playout->RelativeJumpOffset;

Assert( playout->RelativeJumpOffset > 0 || mRelocLabelMap->ContainsKey( labelOffset ) );

bool isBackEdge = false;

if (playout->RelativeJumpOffset < 0)

isBackEdge = true;

BYTE\* relocAddr = nullptr;

AsmJsJitTemplate::BrEq::ApplyTemplate(this, mPc, CalculateOffset<int>(playout->I1), CalculateOffset<int>(playout->I2), &relocAddr, isBackEdge);

Assert( relocAddr );

AddReloc( labelOffset, relocAddr );

}

}

template <class T>

void Js::AsmJsEncoder::Op\_LdConst\_Int( const unaligned T\* playout )

{

AsmJsJitTemplate::LdConst\_Int::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->I0), playout->C1 );

}

template <class T>

void Js::AsmJsEncoder::OP\_SetReturnInt( const unaligned T\* playout )

{

AsmJsJitTemplate::SetReturn\_Int::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->I1) );

}

template <class T>

void Js::AsmJsEncoder::OP\_SetReturnDouble( const unaligned T\* playout )

{

AsmJsJitTemplate::SetReturn\_Db::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->D1) );

}

template <class T>

void Js::AsmJsEncoder::OP\_SetReturnFloat(const unaligned T\* playout)

{

AsmJsJitTemplate::SetReturn\_Flt::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F1));

}

template <class T>

void Js::AsmJsEncoder::OP\_SetFroundInt(const unaligned T\* playout)

{

AsmJsJitTemplate::SetFround\_Int::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F0), CalculateOffset<int>(playout->I1));

}

template <class T>

void Js::AsmJsEncoder::OP\_SetFroundDb(const unaligned T\* playout)

{

AsmJsJitTemplate::SetFround\_Db::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F0), CalculateOffset<double>(playout->D1));

}

template <class T>

void Js::AsmJsEncoder::OP\_SetFroundFlt(const unaligned T\* playout)

{

AsmJsJitTemplate::SetFround\_Flt::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F0), CalculateOffset<float>(playout->F1));

}

template <class T>

void Js::AsmJsEncoder::OP\_AsmJsLoopBody(const unaligned T\* playout)

{

AsmJsJitTemplate::AsmJsLoopBody::ApplyTemplate(this, mPc, (int)playout->C1);

}

template <class T>

void Js::AsmJsEncoder::Op\_Float\_To\_Int(const unaligned T\* playout)

{

AsmJsJitTemplate::Float\_To\_Int::ApplyTemplate(this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<float>(playout->F1));

}

template <class T>

void Js::AsmJsEncoder::Op\_Float\_To\_Db(const unaligned T\* playout)

{

AsmJsJitTemplate::Float\_To\_Db::ApplyTemplate(this, mPc, CalculateOffset<double>(playout->D0), CalculateOffset<float>(playout->F1));

}

template <class T>

void Js::AsmJsEncoder::Op\_UInt\_To\_Db( const unaligned T\* playout )

{

AsmJsJitTemplate::UInt\_To\_Db::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->D0), CalculateOffset<int>(playout->I1) );

}

template <class T>

void Js::AsmJsEncoder::Op\_Int\_To\_Db( const unaligned T\* playout )

{

AsmJsJitTemplate::Int\_To\_Db::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->D0), CalculateOffset<int>(playout->I1) );

}

template <class T>

void Js::AsmJsEncoder::Op\_Db\_To\_Int( const unaligned T\* playout )

{

AsmJsJitTemplate::Db\_To\_Int::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<double>(playout->D1) );

}

template <class T>

void Js::AsmJsEncoder::Op\_StSlot\_Int( const unaligned T\* playout )

{

AsmJsJitTemplate::StSlot\_Int::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->Value), playout->SlotIndex );

}

template <class T>

void Js::AsmJsEncoder::Op\_StSlot\_Db( const unaligned T\* playout )

{

AsmJsJitTemplate::StSlot\_Db::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->Value), playout->SlotIndex );

}

template <class T>

void Js::AsmJsEncoder::Op\_StSlot\_Flt(const unaligned T\* playout)

{

AsmJsJitTemplate::StSlot\_Flt::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->Value), playout->SlotIndex);

}

template <class T>

void Js::AsmJsEncoder::Op\_LdSlot\_Int( const unaligned T\* playout )

{

AsmJsJitTemplate::LdSlot\_Int::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->Value), playout->SlotIndex );

}

template <class T>

void Js::AsmJsEncoder::Op\_LdSlot\_Db( const unaligned T\* playout )

{

AsmJsJitTemplate::LdSlot\_Db::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->Value), playout->SlotIndex );

}

template <class T>

void Js::AsmJsEncoder::Op\_LdSlot\_Flt(const unaligned T\* playout)

{

AsmJsJitTemplate::LdSlot\_Flt::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->Value), playout->SlotIndex);

}

template <class T>

void Js::AsmJsEncoder::Op\_LdAddr\_Db( const unaligned T\* playout )

{

AsmJsJitTemplate::LdAddr\_Db::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->D0), playout->A1 );

}

template <class T> void Js::AsmJsEncoder::OP\_LdSlot( const unaligned T\* playout )

{

AsmJsJitTemplate::LdSlot::ApplyTemplate( this, mPc, CalculateOffset<Var>(playout->Value), CalculateOffset<Var>(playout->Instance), playout->SlotIndex );

}

template <class T> void Js::AsmJsEncoder::OP\_StartCall( const unaligned T\* playout )

{

AsmJsJitTemplate::StartCall::ApplyTemplate( this, mPc, playout->ArgCount);

}

template <class T> void Js::AsmJsEncoder::OP\_Call( const unaligned T\* playout )

{

AsmJsJitTemplate::Call::ApplyTemplate( this, mPc, CalculateOffset<Var>(playout->Return), CalculateOffset<Var>(playout->Function), playout->ArgCount );

}

template <class T> void Js::AsmJsEncoder::OP\_ArgOut\_Db( const unaligned T\* playout )

{

AsmJsJitTemplate::ArgOut\_Db::ApplyTemplate( this, mPc, playout->R0, CalculateOffset<double>(playout->D1));

}

template <class T> void Js::AsmJsEncoder::OP\_ArgOut\_Int( const unaligned T\* playout )

{

AsmJsJitTemplate::ArgOut\_Int::ApplyTemplate( this, mPc, playout->R0, CalculateOffset<int>(playout->I1));

}

template <class T> void Js::AsmJsEncoder::OP\_Conv\_VTD( const unaligned T\* playout )

{

AsmJsJitTemplate::Conv\_VTD::ApplyTemplate( this, mPc, CalculateOffset<double>(playout->D0), CalculateOffset<Var>(playout->R1));

}

template <class T> void Js::AsmJsEncoder::OP\_Conv\_VTF(const unaligned T\* playout)

{

AsmJsJitTemplate::Conv\_VTF::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F0), CalculateOffset<Var>(playout->R1));

}

template <class T> void Js::AsmJsEncoder::OP\_Conv\_VTI( const unaligned T\* playout )

{

AsmJsJitTemplate::Conv\_VTI::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<Var>(playout->R1));

}

template <class T> void Js::AsmJsEncoder::OP\_I\_StartCall( const unaligned T\* playout )

{

AsmJsJitTemplate::I\_StartCall::ApplyTemplate( this, mPc, playout->ArgCount);

}

template <class T> void Js::AsmJsEncoder::OP\_I\_Call( const unaligned T\* playout )

{

AsmJsJitTemplate::I\_Call::ApplyTemplate( this, mPc, CalculateOffset<Var>(playout->Return), CalculateOffset<Var>(playout->Function), playout->ArgCount, AsmJsRetType((AsmJsRetType::Which)playout->ReturnType) );

}

template <class T> void Js::AsmJsEncoder::OP\_I\_ArgOut\_Db( const unaligned T\* playout )

{

AsmJsJitTemplate::I\_ArgOut\_Db::ApplyTemplate( this, mPc, playout->R0, CalculateOffset<double>(playout->D1));

}

template <class T> void Js::AsmJsEncoder::OP\_I\_ArgOut\_Flt(const unaligned T\* playout)

{

AsmJsJitTemplate::I\_ArgOut\_Flt::ApplyTemplate(this, mPc, playout->R0, CalculateOffset<float>(playout->F1));

}

template <class T> void Js::AsmJsEncoder::OP\_I\_ArgOut\_Int( const unaligned T\* playout )

{

AsmJsJitTemplate::I\_ArgOut\_Int::ApplyTemplate( this, mPc, playout->R0, CalculateOffset<int>(playout->I1));

}

template <class T> void Js::AsmJsEncoder::OP\_I\_Conv\_VTD( const unaligned T\* playout )

{

AsmJsJitTemplate::I\_Conv\_VTD::ApplyTemplate(this, mPc, CalculateOffset<double>(playout->D0), CalculateOffset<double>(playout->D1));

}

template <class T> void Js::AsmJsEncoder::OP\_I\_Conv\_VTF(const unaligned T\* playout)

{

AsmJsJitTemplate::I\_Conv\_VTF::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F0), CalculateOffset<float>(playout->F1));

}

template <class T> void Js::AsmJsEncoder::OP\_I\_Conv\_VTI( const unaligned T\* playout )

{

AsmJsJitTemplate::I\_Conv\_VTI::ApplyTemplate( this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<int>(playout->I1));

}

template <class T>

void Js::AsmJsEncoder::Op\_LdArr( const unaligned T\* playout )

{

if (playout->ViewType == ArrayBufferView::TYPE\_FLOAT32)

{

AsmJsJitTemplate::LdArrFlt::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->Value), CalculateOffset<int>(playout->SlotIndex), (ArrayBufferView::ViewType)playout->ViewType);

}

else if(playout->ViewType == ArrayBufferView::TYPE\_FLOAT64)

{

AsmJsJitTemplate::LdArrDb::ApplyTemplate( this, mPc, CalculateOffset<double>( playout->Value ), CalculateOffset<int>( playout->SlotIndex ), (ArrayBufferView::ViewType)playout->ViewType );

}

else

{

AsmJsJitTemplate::LdArr::ApplyTemplate( this, mPc, CalculateOffset<int>( playout->Value ), CalculateOffset<int>( playout->SlotIndex ), (ArrayBufferView::ViewType)playout->ViewType );

}

}

template <class T>

void Js::AsmJsEncoder::Op\_LdArrConst( const unaligned T\* playout )

{

if (playout->ViewType == ArrayBufferView::TYPE\_FLOAT32)

{

AsmJsJitTemplate::ConstLdArrFlt::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->Value), playout->SlotIndex, (ArrayBufferView::ViewType)playout->ViewType);

}

else if(playout->ViewType == ArrayBufferView::TYPE\_FLOAT64 )

{

AsmJsJitTemplate::ConstLdArrDb::ApplyTemplate( this, mPc, CalculateOffset<double>( playout->Value ), playout->SlotIndex, (ArrayBufferView::ViewType)playout->ViewType );

}

else

{

AsmJsJitTemplate::ConstLdArr::ApplyTemplate( this, mPc, CalculateOffset<int>( playout->Value ), playout->SlotIndex, (ArrayBufferView::ViewType)playout->ViewType );

}

}

template <class T>

void Js::AsmJsEncoder::Op\_StArr( const unaligned T\* playout )

{

if (playout->ViewType == ArrayBufferView::TYPE\_FLOAT32 )

{

//Value can be double

AsmJsJitTemplate::StArrFlt::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->Value), CalculateOffset<int>(playout->SlotIndex), (ArrayBufferView::ViewType)playout->ViewType);

}

else if( playout->ViewType == ArrayBufferView::TYPE\_FLOAT64 )

{

AsmJsJitTemplate::StArrDb::ApplyTemplate( this, mPc, CalculateOffset<double>( playout->Value ), CalculateOffset<int>( playout->SlotIndex ), (ArrayBufferView::ViewType)playout->ViewType );

}

else

{

AsmJsJitTemplate::StArr::ApplyTemplate( this, mPc, CalculateOffset<int>( playout->Value ), CalculateOffset<int>( playout->SlotIndex ), (ArrayBufferView::ViewType)playout->ViewType );

}

}

template <class T>

void Js::AsmJsEncoder::Op\_StArrConst( const unaligned T\* playout )

{

if (playout->ViewType == ArrayBufferView::TYPE\_FLOAT32 )

{

AsmJsJitTemplate::ConstStArrFlt::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->Value), playout->SlotIndex, (ArrayBufferView::ViewType)playout->ViewType);

}

else if( playout->ViewType == ArrayBufferView::TYPE\_FLOAT64 )

{

AsmJsJitTemplate::ConstStArrDb::ApplyTemplate( this, mPc, CalculateOffset<double>( playout->Value ), playout->SlotIndex , (ArrayBufferView::ViewType)playout->ViewType );

}

else

{

AsmJsJitTemplate::ConstStArr::ApplyTemplate( this, mPc, CalculateOffset<int>( playout->Value ), playout->SlotIndex, (ArrayBufferView::ViewType)playout->ViewType );

}

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Ld\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Ld\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Ld\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdSlotF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_LdSlot\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->Value), playout->SlotIndex);

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdSlotI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_LdSlot\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->Value), playout->SlotIndex);

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdSlotD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_LdSlot\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->Value), playout->SlotIndex);

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_StSlotF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_StSlot\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->Value), playout->SlotIndex);

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_StSlotI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_StSlot\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->Value), playout->SlotIndex);

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_StSlotD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_StSlot\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->Value), playout->SlotIndex);

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FloatsToF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FloatsToF4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), \

CalculateOffset<float>(playout->F1), CalculateOffset<float>(playout->F2), CalculateOffset<float>(playout->F3), CalculateOffset<float>(playout->F4));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_IntsToI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_IntsToI4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), \

CalculateOffset<int>(playout->I1), CalculateOffset<int>(playout->I2), CalculateOffset<int>(playout->I3), CalculateOffset<int>(playout->I4));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_DoublesToD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_DoublesToD2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), \

CalculateOffset<double>(playout->D1), CalculateOffset<double>(playout->D2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_ReturnF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Return\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_ReturnI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Return\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_ReturnD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Return\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SplatF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Splat\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<float>(playout->F1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SplatI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Splat\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<int>(playout->I1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SplatD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Splat\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<double>(playout->D1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromFloat64x2F4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromFloat64x2\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromInt32x4F4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromInt32x4\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromFloat32x4I4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromFloat32x4\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromFloat64x2I4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromFloat64x2\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromFloat32x4D2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromFloat32x4\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromInt32x4D2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromInt32x4\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromFloat64x2BitsF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromFloat64x2Bits\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromInt32x4BitsF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromInt32x4Bits\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromFloat32x4BitsI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromFloat32x4Bits\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromFloat64x2BitsI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromFloat64x2Bits\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromFloat32x4BitsD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromFloat32x4Bits\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_FromInt32x4BitsD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_FromInt32x4Bits\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_AbsF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Abs\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_AbsD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Abs\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_NegF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Neg\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_NegI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Neg\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_NegD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Neg\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_RcpF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Rcp\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_RcpD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Rcp\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_RcpSqrtF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_RcpSqrt\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_RcpSqrtD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_RcpSqrt\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SqrtF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Sqrt\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SqrtD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Sqrt\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_NotF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Not\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_NotI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Not\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_AddF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Add\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_AddI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Add\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_AddD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Add\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SubF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Sub\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SubI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Sub\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SubD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Sub\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_MulF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Mul\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_MulI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Mul\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_MulD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Mul\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_DivF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Div\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_DivD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Div\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_MinF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Min\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_MinD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Min\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_MaxF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Max\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_MaxD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Max\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LtF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Lt\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LtI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Lt\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LtD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Lt\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_GtF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Gt\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_GtI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Gt\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_GtD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Gt\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LtEqF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_LtEq\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LtEqD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_LtEq\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_GtEqF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_GtEq\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_GtEqD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_GtEq\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_EqF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Eq\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_EqI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Eq\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_EqD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Eq\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_NeqF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Neq\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_NeqD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Neq\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_AndF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_And\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_AndI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_And\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_OrF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Or\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_OrI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Or\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_XorF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Xor\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_XorI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Xor\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SelectF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Select\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->F4\_2), CalculateOffset<AsmJsSIMDValue>(playout->F4\_3));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SelectI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Select\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->I4\_2), CalculateOffset<AsmJsSIMDValue>(playout->I4\_3));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_SelectD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_Select\_D2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<AsmJsSIMDValue>(playout->D2\_2), CalculateOffset<AsmJsSIMDValue>(playout->D2\_3));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_ExtractLaneI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_ExtractLane\_I4::ApplyTemplate(this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<int>(playout->I2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_ExtractLaneF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_ExtractLane\_F4::ApplyTemplate(this, mPc, CalculateOffset<float>(playout->F0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<int>(playout->I2));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_ReplaceLaneI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_ReplaceLane\_I4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1), CalculateOffset<int>(playout->I2), CalculateOffset<int>(playout->I3));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_ReplaceLaneF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_ReplaceLane\_F4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1), CalculateOffset<int>(playout->I2), CalculateOffset<float>(playout->F3));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdSignMaskF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_LdSignMask\_F4::ApplyTemplate(this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdSignMaskI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_LdSignMask\_I4::ApplyTemplate(this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_LdSignMaskD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_LdSignMask\_D2::ApplyTemplate(this, mPc, CalculateOffset<int>(playout->I0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_I\_ArgOutF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_I\_ArgOut\_F4::ApplyTemplate(this, mPc, playout->R0, CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_I\_ArgOutI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_I\_ArgOut\_I4::ApplyTemplate(this, mPc, playout->R0, CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T>

void Js::AsmJsEncoder::OP\_Simd128\_I\_ArgOutD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_I\_ArgOut\_D2::ApplyTemplate(this, mPc, playout->R0, CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

template <class T> void Js::AsmJsEncoder::OP\_Simd128\_I\_Conv\_VTF4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_I\_Conv\_VTF4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->F4\_0), CalculateOffset<AsmJsSIMDValue>(playout->F4\_1));

}

template <class T> void Js::AsmJsEncoder::OP\_Simd128\_I\_Conv\_VTI4(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_I\_Conv\_VTI4::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->I4\_0), CalculateOffset<AsmJsSIMDValue>(playout->I4\_1));

}

template <class T> void Js::AsmJsEncoder::OP\_Simd128\_I\_Conv\_VTD2(const unaligned T\* playout)

{

AsmJsJitTemplate::Simd128\_I\_Conv\_VTD2::ApplyTemplate(this, mPc, CalculateOffset<AsmJsSIMDValue>(playout->D2\_0), CalculateOffset<AsmJsSIMDValue>(playout->D2\_1));

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

// Default all macro to nothing

#ifndef DEF2

#define DEF2(process, op, func)

#endif

#ifndef DEF3

#define DEF3(process, op, func, y)

#endif

#ifndef DEF2\_WMS

#define DEF2\_WMS(process, op, func)

#endif

#ifndef DEF3\_WMS

#define DEF3\_WMS(process, op, func, y)

#endif

#ifndef DEF4\_WMS

#define DEF4\_WMS(process, op, func, y, t)

#endif

#ifndef EXDEF2

#define EXDEF2(process, op, func)

#endif

#ifndef EXDEF3

#define EXDEF3(process, op, func, y)

#endif

#ifndef EXDEF2\_WMS

#define EXDEF2\_WMS(process, op, func)

#endif

#ifndef EXDEF3\_WMS

#define EXDEF3\_WMS(process, op, func, y)

#endif

#ifndef EXDEF4\_WMS

#define EXDEF4\_WMS(process, op, func, y, t)

#endif

#ifdef NTBUILD

// NT header is missing \_\_cdecl on these API

#define \_CRT\_HAS\_CDECL false

#else

#define \_CRT\_HAS\_CDECL true

#endif

DEF3 ( CUSTOM , Nop , OP\_Empty , Empty )

EXDEF3 ( CUSTOM , NopEx , OP\_Empty , Empty )

DEF3 ( CUSTOM , Label , OP\_Label , Empty )

DEF3 ( CUSTOM , Ret , OP\_Empty , Empty )

// External Calls

DEF3 ( CUSTOM , StartCall , OP\_StartCall , StartCall )

DEF3\_WMS( CUSTOM , Call , OP\_Call , AsmCall )

DEF3\_WMS( CUSTOM , ArgOut\_Db , OP\_ArgOut\_Db , Reg1Double1 )

DEF3\_WMS( CUSTOM , ArgOut\_Int , OP\_ArgOut\_Int , Reg1Int1 )

DEF3\_WMS( CUSTOM , Conv\_VTD , OP\_Conv\_VTD , Double1Reg1 )

DEF3\_WMS( CUSTOM , Conv\_VTF , OP\_Conv\_VTF , Float1Reg1 )

DEF3\_WMS( CUSTOM , Conv\_VTI , OP\_Conv\_VTI , Int1Reg1 )

// Internal Calls

DEF3 ( CUSTOM , I\_StartCall , OP\_I\_StartCall , StartCall )

DEF3\_WMS( CUSTOM , I\_Call , OP\_I\_Call , AsmCall )

DEF3\_WMS( CUSTOM , I\_ArgOut\_Db , OP\_I\_ArgOut\_Db , Reg1Double1 )

DEF3\_WMS( CUSTOM , I\_ArgOut\_Flt , OP\_I\_ArgOut\_Flt , Reg1Float1 )

DEF3\_WMS( CUSTOM , I\_ArgOut\_Int , OP\_I\_ArgOut\_Int , Reg1Int1 )

DEF3\_WMS( CUSTOM , I\_Conv\_VTD , OP\_I\_Conv\_VTD , Double2 )

DEF3\_WMS( CUSTOM , I\_Conv\_VTI , OP\_I\_Conv\_VTI , Int2 )

DEF3\_WMS( CUSTOM , I\_Conv\_VTF , OP\_I\_Conv\_VTF , Float2 )

DEF3 ( CUSTOM , AsmBr , OP\_Br , AsmBr )

DEF3\_WMS( CUSTOM , BrTrue\_Int , OP\_BrTrue , BrInt1 )

DEF3\_WMS( CUSTOM , BrEq\_Int , OP\_BrEq , BrInt2 )

// Switching

DEF3\_WMS( INT2 , BeginSwitch\_Int , Ld\_Int , Int2 )

DEF3 ( CUSTOM , EndSwitch\_Int , OP\_Br , AsmBr )

DEF3\_WMS( CUSTOM , Case\_Int , OP\_BrEq , BrInt2 )

DEF3\_WMS( CUSTOM , Conv\_DTI , Op\_Db\_To\_Int , Int1Double1 )

DEF3\_WMS( CUSTOM , Conv\_ITD , Op\_Int\_To\_Db , Double1Int1 )

DEF3\_WMS( CUSTOM , Conv\_UTD , Op\_UInt\_To\_Db , Double1Int1 )

DEF3\_WMS( CUSTOM , Conv\_FTD , Op\_Float\_To\_Db , Double1Float1 )

DEF3\_WMS( CUSTOM , Conv\_FTI , Op\_Float\_To\_Int , Int1Float1 )

DEF3\_WMS( CUSTOM , Return\_Db , OP\_SetReturnDouble , Double2 )

DEF3\_WMS( CUSTOM , Return\_Flt , OP\_SetReturnFloat , Float2 )

DEF3\_WMS( CUSTOM , Return\_Int , OP\_SetReturnInt , Int2 )

DEF3\_WMS( CUSTOM , LdUndef , OP\_LdUndef , AsmReg1 )

DEF3\_WMS( CUSTOM , LdSlotArr , OP\_LdSlot , ElementSlot )

DEF3\_WMS( CUSTOM , LdSlot , OP\_LdSlot , ElementSlot )

DEF3\_WMS( CUSTOM , LdSlot\_Db , Op\_LdSlot\_Db , ElementSlot )

DEF3\_WMS( CUSTOM , LdSlot\_Int , Op\_LdSlot\_Int , ElementSlot )

DEF3\_WMS( CUSTOM , LdSlot\_Flt , Op\_LdSlot\_Flt , ElementSlot )

DEF3\_WMS( CUSTOM , StSlot\_Db , Op\_StSlot\_Db , ElementSlot )

DEF3\_WMS( CUSTOM , StSlot\_Int , Op\_StSlot\_Int , ElementSlot )

DEF3\_WMS( CUSTOM , StSlot\_Flt , Op\_StSlot\_Flt , ElementSlot )

DEF3\_WMS( ELEMENTSLOT, LdArr\_Func , LdArr\_Func , ElementSlot )

DEF3\_WMS( CUSTOM , LdArr , Op\_LdArr , AsmTypedArr )

DEF3\_WMS( CUSTOM , LdArrConst , Op\_LdArrConst , AsmTypedArr )

DEF3\_WMS( CUSTOM , StArr , Op\_StArr , AsmTypedArr )

DEF3\_WMS( CUSTOM , StArrConst , Op\_StArrConst , AsmTypedArr )

DEF3\_WMS( CUSTOM , Ld\_IntConst , Op\_LdConst\_Int , Int1Const1 )

DEF3\_WMS( INT2 , Ld\_Int , Ld\_Int , Int2 )

DEF3\_WMS( INT2 , Neg\_Int , Neg\_Int , Int2 )

DEF3\_WMS( INT2 , Not\_Int , Not\_Int , Int2 )

DEF3\_WMS( INT2 , LogNot\_Int , LogNot\_Int , Int2 )

DEF3\_WMS( INT2 , Conv\_ITB , Int\_To\_Bool , Int2 )

DEF3\_WMS( INT3 , Add\_Int , Add\_Int , Int3 )

DEF3\_WMS( INT3 , Sub\_Int , Sub\_Int , Int3 )

DEF3\_WMS( INT3 , Mul\_Int , Mul\_Int , Int3 )

DEF3\_WMS( INT3 , Div\_Int , Div\_Int , Int3 )

DEF3\_WMS( INT3 , Rem\_Int , Rem\_Int , Int3 )

DEF3\_WMS( INT3 , And\_Int , And\_Int , Int3 )

DEF3\_WMS( INT3 , Or\_Int , Or\_Int , Int3 )

DEF3\_WMS( INT3 , Xor\_Int , Xor\_Int , Int3 )

DEF3\_WMS( INT3 , Shl\_Int , Shl\_Int , Int3 )

DEF3\_WMS( INT3 , Shr\_Int , Shr\_Int , Int3 )

DEF3\_WMS( INT3 , ShrU\_Int , ShrU\_Int , Int3 )

DEF3\_WMS( INT3 , Mul\_UInt , Mul\_UInt , Int3 )

DEF3\_WMS( INT3 , Div\_UInt , Div\_UInt , Int3 )

DEF3\_WMS( INT3 , Rem\_UInt , Rem\_UInt , Int3 )

DEF3\_WMS( DOUBLE2 , Ld\_Db , Ld\_Db , Double2 )

DEF3\_WMS( FLOAT2 , Ld\_Flt , Ld\_Flt , Float2 )

DEF3\_WMS( DOUBLE2 , Neg\_Db , Neg\_Db , Double2 )

DEF3\_WMS( DOUBLE3 , Add\_Db , Add\_Db , Double3 )

DEF3\_WMS( DOUBLE3 , Sub\_Db , Sub\_Db , Double3 )

DEF3\_WMS( DOUBLE3 , Mul\_Db , Mul\_Db , Double3 )

DEF3\_WMS( DOUBLE3 , Div\_Db , Div\_Db , Double3 )

DEF3\_WMS( DOUBLE3 , Rem\_Db , Rem\_Db , Double3 )

//float math

DEF3\_WMS( FLOAT2 , Neg\_Flt , Neg\_Flt , Float2 )

DEF3\_WMS( FLOAT3 , Add\_Flt , Add\_Flt , Float3 )

DEF3\_WMS( FLOAT3 , Sub\_Flt , Sub\_Flt , Float3 )

DEF3\_WMS( FLOAT3 , Mul\_Flt , Mul\_Flt , Float3 )

DEF3\_WMS (FLOAT3 , Div\_Flt , Div\_Flt , Float3 )

DEF3\_WMS( INT3 , CmLt\_Int , Lt\_Int , Int3 )

DEF3\_WMS( INT3 , CmLe\_Int , Le\_Int , Int3 )

DEF3\_WMS( INT3 , CmGt\_Int , Gt\_Int , Int3 )

DEF3\_WMS( INT3 , CmGe\_Int , Ge\_Int , Int3 )

DEF3\_WMS( INT3 , CmEq\_Int , Eq\_Int , Int3 )

DEF3\_WMS( INT3 , CmNe\_Int , Ne\_Int , Int3 )

DEF3\_WMS( INT3 , CmLt\_UnInt , Lt\_UInt , Int3 )

DEF3\_WMS( INT3 , CmLe\_UnInt , Le\_UInt , Int3 )

DEF3\_WMS( INT3 , CmGt\_UnInt , Gt\_UInt , Int3 )

DEF3\_WMS( INT3 , CmGe\_UnInt , Ge\_UInt , Int3 )

DEF3\_WMS( INT1DOUBLE2, CmLt\_Db , CmpLt\_Db , Int1Double2 )

DEF3\_WMS( INT1DOUBLE2, CmLe\_Db , CmpLe\_Db , Int1Double2 )

DEF3\_WMS( INT1DOUBLE2, CmGt\_Db , CmpGt\_Db , Int1Double2 )

DEF3\_WMS( INT1DOUBLE2, CmGe\_Db , CmpGe\_Db , Int1Double2 )

DEF3\_WMS( INT1DOUBLE2, CmEq\_Db , CmpEq\_Db , Int1Double2 )

DEF3\_WMS( INT1DOUBLE2, CmNe\_Db , CmpNe\_Db , Int1Double2 )

DEF3\_WMS( INT1FLOAT2 , CmLt\_Flt , CmpLt\_Flt , Int1Float2 )

DEF3\_WMS( INT1FLOAT2 , CmLe\_Flt , CmpLe\_Flt , Int1Float2 )

DEF3\_WMS( INT1FLOAT2 , CmGt\_Flt , CmpGt\_Flt , Int1Float2 )

DEF3\_WMS( INT1FLOAT2 , CmGe\_Flt , CmpGe\_Flt , Int1Float2 )

DEF3\_WMS( INT1FLOAT2 , CmEq\_Flt , CmpEq\_Flt , Int1Float2 )

DEF3\_WMS( INT1FLOAT2 , CmNe\_Flt , CmpNe\_Flt , Int1Float2 )

DEF3\_WMS( INT2 , Abs\_Int , Abs\_Int , Int2 )

DEF3\_WMS( INT3 , Min\_Int , Min\_Int , Int3 )

DEF3\_WMS( INT3 , Max\_Int , Max\_Int , Int3 )

DEF3\_WMS( INT3 , Imul\_Int , Mul\_Int , Int3 )

DEF3\_WMS( INT2 , Clz32\_Int , Clz32\_Int , Int2 )

DEF3\_WMS( CALLDOUBLE2, Sin\_Db , Math::Sin , false )

DEF3\_WMS( CALLDOUBLE2, Cos\_Db , Math::Cos , false )

DEF3\_WMS( CALLDOUBLE2, Tan\_Db , Math::Tan , false )

DEF3\_WMS( CALLDOUBLE2, Asin\_Db , Math::Asin , false )

DEF3\_WMS( CALLDOUBLE2, Acos\_Db , Math::Acos , false )

DEF3\_WMS( CALLDOUBLE2, Atan\_Db , Math::Atan , false )

DEF3\_WMS( CALLDOUBLE2, Ceil\_Db , ::ceil , true )

DEF3\_WMS( CALLFLOAT2 , Ceil\_Flt , ::ceilf , \_CRT\_HAS\_CDECL)

DEF3\_WMS( CALLDOUBLE2, Floor\_Db , ::floor , true )

DEF3\_WMS( CALLFLOAT2 , Floor\_Flt , ::floorf , \_CRT\_HAS\_CDECL)

DEF3\_WMS( CALLDOUBLE2, Exp\_Db , Math::Exp , false )

DEF3\_WMS( CALLDOUBLE2, Log\_Db , Math::Log , false )

DEF3\_WMS( CALLDOUBLE3, Pow\_Db , Math::Pow , false )

DEF3\_WMS( CALLDOUBLE2, Sqrt\_Db , ::sqrt , true )

DEF3\_WMS( CALLFLOAT2, Sqrt\_Flt , ::sqrtf , \_CRT\_HAS\_CDECL)

DEF3\_WMS( CALLDOUBLE2, Abs\_Db , Math::Abs , false )

DEF3\_WMS( CALLFLOAT2 , Abs\_Flt , ::fabsf , \_CRT\_HAS\_CDECL)

DEF3\_WMS( CALLDOUBLE3, Atan2\_Db , Math::Atan2 , false )

DEF3\_WMS( CALLDOUBLE3, Min\_Db , AsmJsMath::Min<double> , false )

DEF3\_WMS( CALLDOUBLE3, Max\_Db , AsmJsMath::Max<double> , false )

DEF3\_WMS( CUSTOM , Fround\_Flt , OP\_SetFroundFlt , Float2 )

DEF3\_WMS( CUSTOM , Fround\_Db , OP\_SetFroundDb , Float1Double1 )

DEF3\_WMS( CUSTOM , Fround\_Int , OP\_SetFroundInt , Float1Int1 )

DEF3\_WMS( CUSTOM , AsmJsLoopBodyStart,OP\_AsmJsLoopBody , AsmUnsigned1 )

DEF3\_WMS( CUSTOM , Simd128\_Ld\_F4 , OP\_Simd128\_LdF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Ld\_I4 , OP\_Simd128\_LdI4 , Int32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Ld\_D2 , OP\_Simd128\_LdD2 , Float64x2\_2 )

DEF3\_WMS( CUSTOM , Simd128\_LdSlot\_F4 , OP\_Simd128\_LdSlotF4 , ElementSlot )

DEF3\_WMS( CUSTOM , Simd128\_LdSlot\_I4 , OP\_Simd128\_LdSlotI4 , ElementSlot )

DEF3\_WMS( CUSTOM , Simd128\_LdSlot\_D2 , OP\_Simd128\_LdSlotD2 , ElementSlot )

DEF3\_WMS( CUSTOM , Simd128\_StSlot\_F4 , OP\_Simd128\_StSlotF4 , ElementSlot )

DEF3\_WMS( CUSTOM , Simd128\_StSlot\_I4 , OP\_Simd128\_StSlotI4 , ElementSlot )

DEF3\_WMS( CUSTOM , Simd128\_StSlot\_D2 , OP\_Simd128\_StSlotD2 , ElementSlot )

DEF3\_WMS( CUSTOM , Simd128\_FloatsToF4 , OP\_Simd128\_FloatsToF4 , Float32x4\_1Float4 )

DEF3\_WMS( CUSTOM , Simd128\_IntsToI4 , OP\_Simd128\_IntsToI4 , Int32x4\_1Int4 )

DEF3\_WMS( CUSTOM , Simd128\_DoublesToD2 , OP\_Simd128\_DoublesToD2 , Float64x2\_1Double2 )

DEF3\_WMS( CUSTOM , Simd128\_Return\_F4 , OP\_Simd128\_ReturnF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Return\_I4 , OP\_Simd128\_ReturnI4 , Int32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Return\_D2 , OP\_Simd128\_ReturnD2 , Float64x2\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Splat\_F4 , OP\_Simd128\_SplatF4 , Float32x4\_1Float1 )

DEF3\_WMS( CUSTOM , Simd128\_Splat\_I4 , OP\_Simd128\_SplatI4 , Int32x4\_1Int1 )

DEF3\_WMS( CUSTOM , Simd128\_Splat\_D2 , OP\_Simd128\_SplatD2 , Float64x2\_1Double1 )

DEF3\_WMS( CUSTOM , Simd128\_FromFloat64x2\_F4 , OP\_Simd128\_FromFloat64x2F4 , Float32x4\_1Float64x2\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromInt32x4\_F4 , OP\_Simd128\_FromInt32x4F4 , Float32x4\_1Int32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromFloat64x2Bits\_F4 , OP\_Simd128\_FromFloat64x2BitsF4 , Float32x4\_1Float64x2\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromInt32x4Bits\_F4 , OP\_Simd128\_FromInt32x4BitsF4 , Float32x4\_1Int32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromFloat32x4\_D2 , OP\_Simd128\_FromFloat32x4D2 , Float64x2\_1Float32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromInt32x4\_D2 , OP\_Simd128\_FromInt32x4D2 , Float64x2\_1Int32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromFloat32x4Bits\_D2 , OP\_Simd128\_FromFloat32x4BitsD2 , Float64x2\_1Float32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromInt32x4Bits\_D2 , OP\_Simd128\_FromInt32x4BitsD2 , Float64x2\_1Int32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromFloat32x4\_I4 , OP\_Simd128\_FromFloat32x4I4 , Int32x4\_1Float32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromFloat64x2\_I4 , OP\_Simd128\_FromFloat64x2I4 , Int32x4\_1Float64x2\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromFloat32x4Bits\_I4 , OP\_Simd128\_FromFloat32x4BitsI4 , Int32x4\_1Float32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_FromFloat64x2Bits\_I4 , OP\_Simd128\_FromFloat64x2BitsI4 , Int32x4\_1Float64x2\_1 )

DEF3\_WMS( CUSTOM , Simd128\_Abs\_F4 , OP\_Simd128\_AbsF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Abs\_D2 , OP\_Simd128\_AbsD2 , Float64x2\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Neg\_F4 , OP\_Simd128\_NegF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Neg\_I4 , OP\_Simd128\_NegI4 , Int32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Neg\_D2 , OP\_Simd128\_NegD2 , Float64x2\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Rcp\_F4 , OP\_Simd128\_RcpF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Rcp\_D2 , OP\_Simd128\_RcpD2 , Float64x2\_2 )

DEF3\_WMS( CUSTOM , Simd128\_RcpSqrt\_F4 , OP\_Simd128\_RcpSqrtF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_RcpSqrt\_D2 , OP\_Simd128\_RcpSqrtD2 , Float64x2\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Sqrt\_F4 , OP\_Simd128\_SqrtF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Sqrt\_D2 , OP\_Simd128\_SqrtD2 , Float64x2\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Not\_F4 , OP\_Simd128\_NotF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Not\_I4 , OP\_Simd128\_NotI4 , Int32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_Add\_F4 , OP\_Simd128\_AddF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Add\_I4 , OP\_Simd128\_AddI4 , Int32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Add\_D2 , OP\_Simd128\_AddD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Sub\_F4 , OP\_Simd128\_SubF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Sub\_I4 , OP\_Simd128\_SubI4 , Int32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Sub\_D2 , OP\_Simd128\_SubD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Mul\_F4 , OP\_Simd128\_MulF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Mul\_I4 , OP\_Simd128\_MulI4 , Int32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Mul\_D2 , OP\_Simd128\_MulD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Div\_F4 , OP\_Simd128\_DivF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Div\_D2 , OP\_Simd128\_DivD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Min\_F4 , OP\_Simd128\_MinF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Min\_D2 , OP\_Simd128\_MinD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Max\_F4 , OP\_Simd128\_MaxF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Max\_D2 , OP\_Simd128\_MaxD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Lt\_F4 , OP\_Simd128\_LtF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Lt\_I4 , OP\_Simd128\_LtI4 , Int32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Lt\_D2 , OP\_Simd128\_LtD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Gt\_F4 , OP\_Simd128\_GtF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Gt\_I4 , OP\_Simd128\_GtI4 , Int32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Gt\_D2 , OP\_Simd128\_GtD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_LtEq\_F4 , OP\_Simd128\_LtEqF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_LtEq\_D2 , OP\_Simd128\_LtEqD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_GtEq\_F4 , OP\_Simd128\_GtEqF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_GtEq\_D2 , OP\_Simd128\_GtEqD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Eq\_F4 , OP\_Simd128\_EqF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Eq\_I4 , OP\_Simd128\_EqI4 , Int32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Eq\_D2 , OP\_Simd128\_EqD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Neq\_F4 , OP\_Simd128\_NeqF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Neq\_D2 , OP\_Simd128\_NeqD2 , Float64x2\_3 )

DEF3\_WMS( CUSTOM , Simd128\_And\_F4 , OP\_Simd128\_AndF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_And\_I4 , OP\_Simd128\_AndI4 , Int32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Or\_F4 , OP\_Simd128\_OrF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Or\_I4 , OP\_Simd128\_OrI4 , Int32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Xor\_F4 , OP\_Simd128\_XorF4 , Float32x4\_3 )

DEF3\_WMS( CUSTOM , Simd128\_Xor\_I4 , OP\_Simd128\_XorI4 , Int32x4\_3 )

// ToDO: Spec change: Change to BitSelect

DEF3\_WMS( CUSTOM , Simd128\_Select\_F4 , OP\_Simd128\_SelectF4 , Float32x4\_1Int32x4\_1Float32x4\_2)

DEF3\_WMS( CUSTOM , Simd128\_Select\_I4 , OP\_Simd128\_SelectI4 , Int32x4\_4 )

DEF3\_WMS( CUSTOM , Simd128\_Select\_D2 , OP\_Simd128\_SelectD2 , Float64x2\_1Int32x4\_1Float64x2\_2)

DEF3\_WMS( CUSTOM , Simd128\_LdSignMask\_F4 , OP\_Simd128\_LdSignMaskF4 , Int1Float32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_LdSignMask\_I4 , OP\_Simd128\_LdSignMaskI4 , Int1Int32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_LdSignMask\_D2 , OP\_Simd128\_LdSignMaskD2 , Int1Float64x2\_1 )

DEF3\_WMS( CUSTOM , Simd128\_I\_ArgOut\_F4 , OP\_Simd128\_I\_ArgOutF4 , Reg1Float32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_I\_ArgOut\_I4 , OP\_Simd128\_I\_ArgOutI4 , Reg1Int32x4\_1 )

DEF3\_WMS( CUSTOM , Simd128\_I\_ArgOut\_D2 , OP\_Simd128\_I\_ArgOutD2 , Reg1Float64x2\_1 )

//Lane acess

DEF3\_WMS( CUSTOM , Simd128\_ExtractLane\_I4 , OP\_Simd128\_ExtractLaneI4 , Int1Int32x4\_1Int1)

DEF3\_WMS( CUSTOM , Simd128\_ExtractLane\_F4 , OP\_Simd128\_ExtractLaneF4 , Float1Float32x4\_1Int1)

DEF3\_WMS( CUSTOM , Simd128\_ReplaceLane\_I4 , OP\_Simd128\_ReplaceLaneI4 , Int32x4\_2Int2)

DEF3\_WMS( CUSTOM , Simd128\_ReplaceLane\_F4 , OP\_Simd128\_ReplaceLaneF4 , Float32x4\_2Int1Float1)

DEF3\_WMS( CUSTOM , Simd128\_I\_Conv\_VTF4 , OP\_Simd128\_I\_Conv\_VTF4 , Float32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_I\_Conv\_VTI4 , OP\_Simd128\_I\_Conv\_VTI4 , Int32x4\_2 )

DEF3\_WMS( CUSTOM , Simd128\_I\_Conv\_VTD2 , OP\_Simd128\_I\_Conv\_VTD2 , Float64x2\_2 )

// help the caller to undefine all the macros

#undef DEF2

#undef DEF3

#undef DEF2\_WMS

#undef DEF3\_WMS

#undef DEF4\_WMS

#undef EXDEF2

#undef EXDEF3

#undef EXDEF2\_WMS

#undef EXDEF3\_WMS

#undef EXDEF4\_WMS

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

void AsmJsCommonEntryPoint(Js::ScriptFunction\* func, void\* savedEbp);

namespace AsmJsJitTemplate

{

const int PAGESIZE = 0x1000;

typedef AsmJsEncoder\* TemplateContext;

// Initialise template data for architecture specific

void\* InitTemplateData();

// Free template data for architecture specific

void FreeTemplateData( void\* userData );

struct Globals

{

#if DBG\_DUMP

static FunctionBody\* CurrentEncodingFunction;

#endif

// Number of Vars on the stack before the first variable

static const int StackVarCount = 2;

static const int ModuleSlotOffset ;

static const int ModuleEnvOffset ;

static const int ArrayBufferOffset ;

static const int ArraySizeOffset ;

static const int ScriptContextOffset;

};

#ifdef \_M\_IX86

#define CreateTemplate(name,...) \

struct name\

{\

static int ApplyTemplate( TemplateContext context, BYTE\*& buffer,##\_\_VA\_ARGS\_\_ );\

}

#else

#define CreateTemplate(name,...) \

struct name\

{\

static int ApplyTemplate( TemplateContext context, BYTE\*& buffer,##\_\_VA\_ARGS\_\_ ) { \_\_debugbreak(); return 0; }\

}

#endif

CreateTemplate( FunctionEntry );

CreateTemplate( FunctionExit );

CreateTemplate( Br, BYTE\*\* relocAddr, bool isBackEdge);

CreateTemplate( BrTrue, int offset, BYTE\*\* relocAddr, bool isBackEdge);

CreateTemplate( BrEq, int leftOffset, int rightOffset, BYTE\*\* relocAddr, bool isBackEdge);

CreateTemplate( Label );

CreateTemplate( LdUndef, int targetOffset );

CreateTemplate( LdSlot, int targetOffset, int arrOffset, int slotIndex );

CreateTemplate( LdArr\_Func, int targetOffset, int arrOffset, int slotVarIndex );

// int operations

CreateTemplate( Ld\_Int, int targetOffset, int rightOffset );

CreateTemplate( LdSlot\_Int, int targetOffset, int slotIndex);

CreateTemplate( LdSlot\_Flt, int targetOffset, int slotIndex);

CreateTemplate( StSlot\_Flt, int srcOffset, int slotIndex);

CreateTemplate( StSlot\_Int, int srcOffset, int slotIndex);

CreateTemplate( LdConst\_Int, int offset, int value );

CreateTemplate( SetReturn\_Int, int offset );

CreateTemplate( Db\_To\_Int, int targetOffset, int rightOffset );

CreateTemplate( Int\_To\_Bool, int targetOffset, int rightOffset );

CreateTemplate( LogNot\_Int, int targetOffset, int rightOffset );

CreateTemplate( Neg\_Int, int targetOffset, int rightOffset );

CreateTemplate( Not\_Int, int targetOffset, int rightOffset );

CreateTemplate( Or\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( And\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Xor\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Shl\_Int , int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Shr\_Int , int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( ShrU\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Add\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Sub\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Mul\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Div\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Rem\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Lt\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Le\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Gt\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Ge\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Eq\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Ne\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Min\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Max\_Int, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Abs\_Int, int targetOffset, int rightOffset );

CreateTemplate( Clz32\_Int,int targetOffset,int rightOffset );

// uint operations

CreateTemplate( Div\_UInt, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Mul\_UInt, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Rem\_UInt, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Lt\_UInt, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Le\_UInt, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Gt\_UInt, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Ge\_UInt, int targetOffset, int leftOffset, int rightOffset );

//Float Operations

CreateTemplate( Add\_Flt, int targetOffset, int leftOffset, int rightOffset);

CreateTemplate( Sub\_Flt, int targetOffset, int leftOffset, int rightOffset);

CreateTemplate( Mul\_Flt, int targetOffset, int leftOffset, int rightOffset);

CreateTemplate( Div\_Flt, int targetOffset, int leftOffset, int rightOffset);

// Double operations

CreateTemplate( Ld\_Db, int targetOffset, int rightOffset);

CreateTemplate( Ld\_Flt, int targetOffset, int rightOffset);

CreateTemplate( LdAddr\_Db, int targetOffset, const double\* dbAddr);

CreateTemplate( LdSlot\_Db, int targetOffset, int slotIndex );

CreateTemplate( StSlot\_Db, int srcOffset, int slotIndex );

CreateTemplate( SetReturn\_Db, int offset);

CreateTemplate( SetReturn\_Flt, int offset);

CreateTemplate( SetFround\_Db, int targetOffset, int rightOffset);

CreateTemplate( SetFround\_Flt, int targetOffset, int rightOffset);

CreateTemplate( SetFround\_Int, int targetOffset, int rightOffset);

CreateTemplate( Int\_To\_Db, int targetOffset, int rightOffset );

CreateTemplate( Float\_To\_Db, int targetOffset, int rightOffset);

CreateTemplate( Float\_To\_Int, int targetOffset, int rightOffset);

CreateTemplate( UInt\_To\_Db, int targetOffset, int rightOffset);

CreateTemplate( Neg\_Db, int targetOffset, int rightOffset);

CreateTemplate( Neg\_Flt, int targetOffset, int rightOffset);

CreateTemplate( Add\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Sub\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Mul\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Div\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( Rem\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( CmpLt\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( CmpLe\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( CmpGt\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( CmpGe\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( CmpEq\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( CmpNe\_Db, int targetOffset, int leftOffset, int rightOffset );

CreateTemplate( CmpLt\_Flt, int targetOffset, int leftOffset, int rightOffset);

CreateTemplate( CmpLe\_Flt, int targetOffset, int leftOffset, int rightOffset);

CreateTemplate( CmpGt\_Flt, int targetOffset, int leftOffset, int rightOffset);

CreateTemplate( CmpGe\_Flt, int targetOffset, int leftOffset, int rightOffset);

CreateTemplate( CmpEq\_Flt, int targetOffset, int leftOffset, int rightOffset);

CreateTemplate( CmpNe\_Flt, int targetOffset, int leftOffset, int rightOffset);

// offsets : array of offset for double variables, the first one is where the result should be put in

// D0 = func(D1,D2,D3); => offsets = [D0,D1,D2,D3]

CreateTemplate(Call\_Db, int nbOffsets, int\* offsets, void\* addr, bool addEsp);

CreateTemplate(Call\_Flt, int nbOffsets, int\* offsets, void\* addr, bool addEsp);

//external calls

CreateTemplate( StartCall, int argBytesSize );

CreateTemplate( ArgOut\_Int, int argIndex, int offset );

CreateTemplate( ArgOut\_Db, int argIndex, int offset);

CreateTemplate( Call, int targetOffset, int funcOffset, int nbArgs);

CreateTemplate( Conv\_VTI, int targetOffset, int srcOffset);

CreateTemplate( Conv\_VTD, int targetOffset, int srcOffset);

CreateTemplate( Conv\_VTF, int targetOffset, int srcOffset);

//internal calls

CreateTemplate( I\_StartCall, int argBytesSize );

CreateTemplate( I\_ArgOut\_Int, int argIndex, int offset );

CreateTemplate( I\_ArgOut\_Db, int argIndex, int offset);

CreateTemplate( I\_ArgOut\_Flt, int argIndex, int offset);

CreateTemplate( I\_Call, int targetOffset, int funcOffset, int nbArgs, AsmJsRetType retType);

CreateTemplate( I\_Conv\_VTI, int targetOffset, int srcOffset);

CreateTemplate( I\_Conv\_VTD, int targetOffset, int srcOffset);

CreateTemplate( I\_Conv\_VTF, int targetOffset, int srcOffset);

CreateTemplate( LdArr, int targetOffset, int slotVarIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( LdArrDb, int targetOffset, int slotVarIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( LdArrFlt, int targetOffset, int slotVarIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( StArr, int srcOffset, int slotVarIndex, ArrayBufferView::ViewType viewType );

CreateTemplate( StArrDb, int srcOffset, int slotVarIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( StArrFlt, int srcOffset, int slotVarIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( ConstLdArr, int targetOffset, int constIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( ConstLdArrDb, int targetOffset, int constIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( ConstLdArrFlt, int targetOffset, int constIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( ConstStArr, int srcOffset, int constIndex, ArrayBufferView::ViewType viewType );

CreateTemplate( ConstStArrDb, int srcOffset, int constIndex, ArrayBufferView::ViewType viewType);

CreateTemplate( ConstStArrFlt, int srcOffset, int constIndex, ArrayBufferView::ViewType viewType);

CreateTemplate(AsmJsLoopBody, int offset);

CreateTemplate(Simd128\_Ld\_F4, int targetOffsetF4, int srcOffsetF4);

CreateTemplate(Simd128\_Ld\_I4, int targetOffsetI4, int srcOffsetI4);

CreateTemplate(Simd128\_Ld\_D2, int targetOffsetD2, int srcOffsetD2);

CreateTemplate(Simd128\_LdSlot\_F4, int targetOffset, int slotIndex);

CreateTemplate(Simd128\_LdSlot\_I4, int targetOffset, int slotIndex);

CreateTemplate(Simd128\_LdSlot\_D2, int targetOffset, int slotIndex);

CreateTemplate(Simd128\_StSlot\_F4, int srcOffset, int slotIndex);

CreateTemplate(Simd128\_StSlot\_I4, int srcOffset, int slotIndex);

CreateTemplate(Simd128\_StSlot\_D2, int srcOffset, int slotIndex);

CreateTemplate(Simd128\_FloatsToF4, int targetOffsetF4\_0, int srcOffsetF1, int srcOffsetF2, int srcOffsetF3, int srcOffsetF4);

CreateTemplate(Simd128\_IntsToI4, int targetOffsetI4\_0, int srcOffsetI1, int srcOffsetI2, int srcOffsetI3, int srcOffsetI4);

CreateTemplate(Simd128\_DoublesToD2, int targetOffsetD2\_0, int srcOffsetD0, int srcOffsetD1);

CreateTemplate(Simd128\_Return\_F4, int srcOffsetF4);

CreateTemplate(Simd128\_Return\_I4, int srcOffsetI4);

CreateTemplate(Simd128\_Return\_D2, int srcOffsetD2);

CreateTemplate(Simd128\_Splat\_F4, int targetOffsetF4\_0, int srcOffsetF1);

CreateTemplate(Simd128\_Splat\_I4, int targetOffsetI4\_0, int srcOffsetI1);

CreateTemplate(Simd128\_Splat\_D2, int targetOffsetD2\_0, int srcOffsetD1);

CreateTemplate(Simd128\_FromFloat64x2\_F4, int targetOffsetF4\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_FromInt32x4\_F4, int targetOffsetF4\_0, int srcOffsetI4\_1);

CreateTemplate(Simd128\_FromFloat32x4\_I4, int targetOffsetI4\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_FromFloat64x2\_I4, int targetOffsetI4\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_FromFloat32x4\_D2, int targetOffsetD2\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_FromInt32x4\_D2, int targetOffsetD2\_0, int srcOffsetI4\_1);

CreateTemplate(Simd128\_FromFloat32x4Bits\_D2, int targetOffsetD2\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_FromInt32x4Bits\_D2, int targetOffsetD2\_0, int srcOffsetI4\_1);

CreateTemplate(Simd128\_FromFloat32x4Bits\_I4, int targetOffsetI4\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_FromFloat64x2Bits\_I4, int targetOffsetI4\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_FromFloat64x2Bits\_F4, int targetOffsetF4\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_FromInt32x4Bits\_F4, int targetOffsetF4\_0, int srcOffsetI4\_1);

CreateTemplate(Simd128\_Abs\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_Abs\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_Neg\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_Neg\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1);

CreateTemplate(Simd128\_Neg\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_Rcp\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_Rcp\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_RcpSqrt\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_RcpSqrt\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_Sqrt\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_Sqrt\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_Not\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_Not\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1);

CreateTemplate(Simd128\_Add\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Add\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Add\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Sub\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Sub\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Sub\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Mul\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Mul\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Mul\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Div\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Div\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Min\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Min\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Max\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Max\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Lt\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Lt\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Lt\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Gt\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Gt\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Gt\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_LtEq\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_LtEq\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_GtEq\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_GtEq\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Eq\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Eq\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Eq\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_Neq\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Neq\_D2, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2);

CreateTemplate(Simd128\_And\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_And\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Or\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Or\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Xor\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2);

CreateTemplate(Simd128\_Xor\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2);

CreateTemplate(Simd128\_Select\_F4, int targetOffsetF4\_0, int srcOffsetI4\_1, int srcOffsetF4\_2, int srcOffsetF4\_3);

CreateTemplate(Simd128\_Select\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2, int srcOffsetI4\_3);

CreateTemplate(Simd128\_Select\_D2, int targetOffsetD2\_0, int srcOffsetI4\_1, int srcOffsetD2\_2, int srcOffsetD2\_3);

CreateTemplate(Simd128\_ExtractLane\_I4, int targetOffsetI0, int srcOffsetI4\_1, int index);

CreateTemplate(Simd128\_ExtractLane\_F4, int targetOffsetF0, int srcOffsetF4\_1, int index);

CreateTemplate(Simd128\_ReplaceLane\_I4, int targetOffsetI4\_0, int srcOffsetI4\_1, int index, int srcOffsetI3);

CreateTemplate(Simd128\_ReplaceLane\_F4, int targetOffsetF4\_0, int srcOffsetF4\_1, int index, int srcOffsetF3);

CreateTemplate(Simd128\_LdSignMask\_F4, int targetOffsetI0, int srcOffsetF4\_1);

CreateTemplate(Simd128\_LdSignMask\_I4, int targetOffsetI0, int srcOffsetI4\_1);

CreateTemplate(Simd128\_LdSignMask\_D2, int targetOffsetI0, int srcOffsetD2\_1);

CreateTemplate(Simd128\_I\_ArgOut\_F4, int argIndex, int offset );

CreateTemplate(Simd128\_I\_ArgOut\_I4, int argIndex, int offset);

CreateTemplate(Simd128\_I\_ArgOut\_D2, int argIndex, int offset);

CreateTemplate(Simd128\_I\_Conv\_VTF4, int targetOffset, int srcOffset);

CreateTemplate(Simd128\_I\_Conv\_VTI4, int targetOffset, int srcOffset);

CreateTemplate(Simd128\_I\_Conv\_VTD2, int targetOffset, int srcOffset);

};

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Library\BoundFunction.h"

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js{

bool ASMLink::CheckArrayBuffer(ScriptContext\* scriptContext, Var bufferView, const AsmJsModuleInfo \* info)

{

if (!bufferView)

{

return true;

}

if (!JavascriptArrayBuffer::Is(bufferView))

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : Buffer parameter is not an Array buffer");

return false;

}

JavascriptArrayBuffer\* buffer = (JavascriptArrayBuffer\*)bufferView;

if (buffer->GetByteLength() <= info->GetMaxHeapAccess())

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : Buffer bytelength is smaller than constant accesses");

return false;

}

if (info->GetUsesChangeHeap())

{

if (buffer->GetByteLength() < 0x1000000)

{

Output::Print(L"Asm.js Runtime Error : Buffer bytelength is not a valid size for asm.js\n");

return false;

}

if (info->GetMaxHeapAccess() >= 0x1000000)

{

Output::Print(L"Asm.js Runtime Error : Cannot have such large constant accesses\n");

return false;

}

}

if (!buffer->IsValidAsmJsBufferLength(buffer->GetByteLength(), true))

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : Buffer bytelength is not a valid size for asm.js");

return false;

}

return true;

}

bool ASMLink::CheckFFI(ScriptContext\* scriptContext, AsmJsModuleInfo\* info, const Var foreign)

{

if (info->GetFunctionImportCount() == 0 && info->GetVarImportCount() == 0)

{

return true;

}

Assert(foreign);

if (!RecyclableObject::Is(foreign))

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : FFI is not an object");

return false;

}

TypeId foreignObjType = RecyclableObject::FromVar(foreign)->GetTypeId();

if (StaticType::Is(foreignObjType) || TypeIds\_Proxy == foreignObjType)

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : FFI is not an object");

return false;

}

return true;

}

bool ASMLink::CheckStdLib(ScriptContext\* scriptContext, const AsmJsModuleInfo\* info, const Var stdlib)

{

BVStatic<ASMMATH\_BUILTIN\_SIZE> mathBuiltinUsed = info->GetAsmMathBuiltinUsed();

BVStatic<ASMARRAY\_BUILTIN\_SIZE> arrayBuiltinUsed = info->GetAsmArrayBuiltinUsed();

if (mathBuiltinUsed.IsAllClear() && arrayBuiltinUsed.IsAllClear())

{

return true;

}

Assert(stdlib);

if (!RecyclableObject::Is(stdlib))

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : StdLib is not an object");

return false;

}

TypeId stdLibObjType = RecyclableObject::FromVar(stdlib)->GetTypeId();

if (StaticType::Is(stdLibObjType) || TypeIds\_Proxy == stdLibObjType)

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : StdLib is not an object");

return false;

}

Js::JavascriptLibrary\* library = scriptContext->GetLibrary();

if (mathBuiltinUsed.Test(AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_infinity))

{

Var asmInfinityObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Infinity, scriptContext);

if (!JavascriptConversion::SameValue(asmInfinityObj, library->GetPositiveInfinite()))

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : Math constant Infinity is invalid");

return false;

}

}

if (mathBuiltinUsed.Test(AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_nan))

{

Var asmNaNObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::NaN, scriptContext);

if (!JavascriptConversion::SameValue(asmNaNObj, library->GetNaN()))

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : Math constant NaN is invalid");

return false;

}

}

Var asmMathObject = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Math, scriptContext);

for (int i = 0; i < AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_COUNT; i++)

{

//check if bit is set

if (!mathBuiltinUsed.Test(i) || i == AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_infinity || i == AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_nan)

{

continue;

}

AsmJSMathBuiltinFunction mathBuiltinFunc = (AsmJSMathBuiltinFunction)i;

if (!CheckMathLibraryMethod(scriptContext, asmMathObject, mathBuiltinFunc))

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : Math builtin function is invalid");

return false;

}

}

for (int i = 0; i < AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_COUNT; i++)

{

//check if bit is set

if (!arrayBuiltinUsed.Test(i))

{

continue;

}

AsmJSTypedArrayBuiltinFunction arrayBuiltinFunc = (AsmJSTypedArrayBuiltinFunction)i;

if (!CheckArrayLibraryMethod(scriptContext, stdlib, arrayBuiltinFunc))

{

AsmJSCompiler::OutputError(scriptContext, L"Asm.js Runtime Error : Array builtin function is invalid");

return false;

}

}

return true;

}

bool ASMLink::CheckArrayLibraryMethod(ScriptContext\* scriptContext, const Var stdlib, const AsmJSTypedArrayBuiltinFunction arrayLibMethod)

{

Var arrayFuncObj;

switch (arrayLibMethod)

{

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_byteLength:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::byteLength, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

if (arrayLibFunc->IsBoundFunction())

{

BoundFunction\* boundFunc = (BoundFunction\*)arrayLibFunc;

RecyclableObject\* thisObj = boundFunc->GetBoundThis();

if (JavascriptFunction::Is(thisObj))

{

JavascriptFunction \* thisFunc = (JavascriptFunction\*)thisObj;

if (thisFunc->GetFunctionInfo()->GetOriginalEntryPoint() != (&ArrayBuffer::EntryInfo::GetterByteLength)->GetOriginalEntryPoint())

{

return false;

}

}

JavascriptFunction\* targetFunc = boundFunc->GetTargetFunction();

return targetFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&JavascriptFunction::EntryInfo::Call)->GetOriginalEntryPoint();

}

}

break;

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_Int8Array:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Int8Array, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

return arrayLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Int8Array::EntryInfo::NewInstance)->GetOriginalEntryPoint();

}

break;

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_Uint8Array:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Uint8Array, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

return arrayLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Uint8Array::EntryInfo::NewInstance)->GetOriginalEntryPoint();

}

break;

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_Int16Array:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Int16Array, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

return arrayLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Int16Array::EntryInfo::NewInstance)->GetOriginalEntryPoint();

}

break;

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_Uint16Array:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Uint16Array, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

return arrayLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Uint16Array::EntryInfo::NewInstance)->GetOriginalEntryPoint();

}

break;

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_Int32Array:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Int32Array, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

return arrayLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Int32Array::EntryInfo::NewInstance)->GetOriginalEntryPoint();

}

break;

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_Uint32Array:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Uint32Array, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

return arrayLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Uint32Array::EntryInfo::NewInstance)->GetOriginalEntryPoint();

}

break;

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_Float32Array:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Float32Array, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

return arrayLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Float32Array::EntryInfo::NewInstance)->GetOriginalEntryPoint();

}

break;

case AsmJSTypedArrayBuiltinFunction::AsmJSTypedArrayBuiltin\_Float64Array:

arrayFuncObj = JavascriptOperators::OP\_GetProperty(stdlib, PropertyIds::Float64Array, scriptContext);

if (JavascriptFunction::Is(arrayFuncObj))

{

JavascriptFunction\* arrayLibFunc = (JavascriptFunction\*)arrayFuncObj;

return arrayLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Float64Array::EntryInfo::NewInstance)->GetOriginalEntryPoint();

}

break;

default:

Assume(UNREACHED);

}

return false;

}

bool ASMLink::CheckMathLibraryMethod(ScriptContext\* scriptContext, const Var asmMathObject, const AsmJSMathBuiltinFunction mathLibMethod)

{

Var mathFuncObj;

switch (mathLibMethod)

{

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_sin:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::sin, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Sin)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_cos:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::cos, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Cos)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_tan:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::tan, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Tan)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_asin:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::asin, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Asin)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_acos:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::acos, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Acos)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_atan:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::atan, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Atan)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_ceil:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::ceil, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Ceil)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_floor:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::floor, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Floor)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_exp:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::exp, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Exp)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_log:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::log, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Log)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_pow:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::pow, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Pow)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_sqrt:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::sqrt, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Sqrt)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_abs:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::abs, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Abs)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_atan2:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::atan2, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Atan2)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_imul:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::imul, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Imul)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_clz32:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::clz32, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Clz32)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_min:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::min, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Min)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_max:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::max, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Max)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_fround:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::fround, scriptContext);

if (JavascriptFunction::Is(mathFuncObj))

{

JavascriptFunction\* mathLibFunc = (JavascriptFunction\*)mathFuncObj;

if (mathLibFunc->GetFunctionInfo()->GetOriginalEntryPoint() == (&Math::EntryInfo::Fround)->GetOriginalEntryPoint())

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_e:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::E, scriptContext);

if (JavascriptNumber::Is(mathFuncObj))

{

JavascriptNumber\* mathConstNumber = (JavascriptNumber\*)mathFuncObj;

if (JavascriptNumber::GetValue(mathConstNumber) == (Math::E))

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_ln10:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::LN10, scriptContext);

if (JavascriptNumber::Is(mathFuncObj))

{

JavascriptNumber\* mathConstNumber = (JavascriptNumber\*)mathFuncObj;

if (JavascriptNumber::GetValue(mathConstNumber) == (Math::LN10))

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_ln2:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::LN2, scriptContext);

if (JavascriptNumber::Is(mathFuncObj))

{

JavascriptNumber\* mathConstNumber = (JavascriptNumber\*)mathFuncObj;

if (JavascriptNumber::GetValue(mathConstNumber) == (Math::LN2))

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_log2e:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::LOG2E, scriptContext);

if (JavascriptNumber::Is(mathFuncObj))

{

JavascriptNumber\* mathConstNumber = (JavascriptNumber\*)mathFuncObj;

if (JavascriptNumber::GetValue(mathConstNumber) == (Math::LOG2E))

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_log10e:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::LOG10E, scriptContext);

if (JavascriptNumber::Is(mathFuncObj))

{

JavascriptNumber\* mathConstNumber = (JavascriptNumber\*)mathFuncObj;

if (JavascriptNumber::GetValue(mathConstNumber) == (Math::LOG10E))

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_pi:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::PI, scriptContext);

if (JavascriptNumber::Is(mathFuncObj))

{

JavascriptNumber\* mathConstNumber = (JavascriptNumber\*)mathFuncObj;

if (JavascriptNumber::GetValue(mathConstNumber) == (Math::PI))

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_sqrt1\_2:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::SQRT1\_2, scriptContext);

if (JavascriptNumber::Is(mathFuncObj))

{

JavascriptNumber\* mathConstNumber = (JavascriptNumber\*)mathFuncObj;

if (JavascriptNumber::GetValue(mathConstNumber) == (Math::SQRT1\_2))

{

return true;

}

}

break;

case AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_sqrt2:

mathFuncObj = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::SQRT2, scriptContext);

if (JavascriptNumber::Is(mathFuncObj))

{

JavascriptNumber\* mathConstNumber = (JavascriptNumber\*)mathFuncObj;

if (JavascriptNumber::GetValue(mathConstNumber) == (Math::SQRT2))

{

return true;

}

}

break;

default:

Assume(UNREACHED);

}

return false;

}

bool ASMLink::CheckParams(ScriptContext\* scriptContext, AsmJsModuleInfo\* info, const Var stdlib, const Var foreign, const Var bufferView)

{

if (CheckStdLib(scriptContext, info, stdlib) && CheckArrayBuffer(scriptContext, bufferView, info) && CheckFFI(scriptContext, info, stdlib))

{

return true;

}

Output::Flush();

return false;

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js{

class ASMLink{

public:

static bool CheckParams(ScriptContext\* scriptContext, AsmJsModuleInfo\* info , const Var stdlib, const Var foreigh, Var bufferView);

private:

static bool CheckArrayBuffer(ScriptContext\* scriptContext, const Var bufferView, const AsmJsModuleInfo\* info);

static bool CheckStdLib(ScriptContext\* scriptContext, const AsmJsModuleInfo\* info, const Var stdlib);

static bool CheckFFI(ScriptContext\* scriptContext, AsmJsModuleInfo\* info, const Var foreign);

static bool CheckArrayLibraryMethod(ScriptContext\* scriptContext, const Var stdlib, const AsmJSTypedArrayBuiltinFunction arrayBuiltin);

static bool CheckMathLibraryMethod(ScriptContext\* scriptContext, const Var asmMathObject, const AsmJSMathBuiltinFunction mathBuiltin);

};

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

#include "ByteCode\Symbol.h"

#include "ByteCode\FuncInfo.h"

#include "ByteCode\ByteCodeAPI.h"

#include "ByteCode\ByteCodeWriter.h"

#include "ByteCode\ByteCodeGenerator.h"

#include "ByteCode\AsmJsByteCodeWriter.h"

#include "Language\AsmJsByteCodeGenerator.h"

namespace Js

{

bool AsmJsModuleCompiler::CompileAllFunctions()

{

const int size = mFunctionArray.Count();

for (int i = 0; i < size; i++)

{

AsmJsFunc\* func = mFunctionArray.Item(i);

if (!CompileFunction(func, i))

{

// an error occurred in the function, revert state on all asm.js functions

for (int j = 0; j <= i; j++)

{

RevertFunction(j);

}

return false;

}

func->Finish();

}

return true;

}

void AsmJsModuleCompiler::RevertFunction(int funcIndex)

{

AsmJsFunc\* func = mFunctionArray.Item(funcIndex);

FunctionBody \* funcBody = func->GetFuncBody();

funcBody->ResetByteCodeGenState();

funcBody->AddDeferParseAttribute();

funcBody->SetFunctionParsed(false);

funcBody->ResetEntryPoint();

funcBody->SetEntryPoint(funcBody->GetDefaultEntryPointInfo(), GetScriptContext()->DeferredParsingThunk);

funcBody->SetIsAsmjsMode(false);

funcBody->SetIsAsmJsFunction(false);

func->GetFncNode()->sxFnc.funcInfo->byteCodeFunction = func->GetFuncBody();

}

void AsmJsModuleCompiler::RevertAllFunctions()

{

for (int i = 0; i < mFunctionArray.Count(); i++)

{

RevertFunction(i);

}

}

bool AsmJsModuleCompiler::CommitFunctions()

{

const int size = mFunctionArray.Count();

// if changeHeap is defined, it must be first function, so we should skip it

for (int i = 0; i < size; i++)

{

AsmJsFunc\* func = mFunctionArray.Item(i);

FunctionBody\* functionBody = func->GetFuncBody();

AsmJsFunctionInfo\* asmInfo = functionBody->AllocateAsmJsFunctionInfo();

if (i == 0 && mUsesChangeHeap)

{

continue;

}

const auto& intRegisterSpace = func->GetRegisterSpace<int>();

const auto& doubleRegisterSpace = func->GetRegisterSpace<double>();

const auto& floatRegisterSpace = func->GetRegisterSpace<float>();

if (!asmInfo->Init(func))

{

return false;

}

asmInfo->SetIsHeapBufferConst(!mUsesChangeHeap);

asmInfo->SetUsesHeapBuffer(mUsesHeapBuffer);

int varCount = 0;

varCount += (int)((intRegisterSpace.GetTotalVarCount() \* INT\_SLOTS\_SPACE) + 0.5);

varCount += (int)(floatRegisterSpace.GetTotalVarCount() \* FLOAT\_SLOTS\_SPACE + 0.5);

varCount += doubleRegisterSpace.GetTotalVarCount() \* DOUBLE\_SLOTS\_SPACE;

if (IsSimdjsEnabled())

{

const auto& simdRegisterSpace = func->GetRegisterSpace<AsmJsSIMDValue>();

varCount += (int)(simdRegisterSpace.GetTotalVarCount() \* SIMD\_SLOTS\_SPACE);

}

functionBody->SetOutParamDepth(func->GetMaxArgOutDepth());

functionBody->SetVarCount(varCount);

// should be set in EmitOneFunction

Assert(functionBody->GetIsAsmjsMode());

Assert(functionBody->GetIsAsmJsFunction());

((EntryPointInfo\*)functionBody->GetDefaultEntryPointInfo())->SetIsAsmJSFunction(true);

#if \_M\_IX86

if (PHASE\_ON1(AsmJsJITTemplatePhase) && !Configuration::Global.flags.NoNative)

{

AsmJsCodeGenerator\* generator = GetScriptContext()->GetAsmJsCodeGenerator();

AccumulateCompileTime();

if (!generator)

{

generator = GetScriptContext()->InitAsmJsCodeGenerator();

}

Assert( generator );

generator->CodeGen(functionBody);

AccumulateCompileTime(AsmJsCompilation::TemplateJIT);

}

#endif

}

return true;

}

bool AsmJsModuleCompiler::CommitModule()

{

FuncInfo\* funcInfo = GetModuleFunctionNode()->sxFnc.funcInfo;

FunctionBody\* functionBody = funcInfo->GetParsedFunctionBody();

AsmJsModuleInfo\* asmInfo = functionBody->AllocateAsmJsModuleInfo();

int argCount = 0;

if (mBufferArgName)

{

argCount = 3;

}

else if (mForeignArgName)

{

argCount = 2;

}

else if (mStdLibArgName)

{

argCount = 1;

}

const int functionCount = mFunctionArray.Count();

const int functionTableCount = mFunctionTableArray.Count();

const int importFunctionCount = mImportFunctions.GetTotalVarCount();

asmInfo->SetFunctionCount(functionCount);

asmInfo->SetFunctionTableCount(functionTableCount);

asmInfo->SetFunctionImportCount(importFunctionCount);

asmInfo->SetVarCount(mVarCount);

asmInfo->SetVarImportCount(mVarImportCount);

asmInfo->SetArgInCount(argCount);

asmInfo->SetModuleMemory(mModuleMemory);

asmInfo->SetAsmMathBuiltinUsed(mAsmMathBuiltinUsedBV);

asmInfo->SetAsmArrayBuiltinUsed(mAsmArrayBuiltinUsedBV);

asmInfo->SetUsesChangeHeap(mUsesChangeHeap);

asmInfo->SetMaxHeapAccess(mMaxHeapAccess);

if (IsSimdjsEnabled())

{

asmInfo->SetAsmSimdBuiltinUsed(mAsmSimdBuiltinUsedBV);

asmInfo->SetSimdRegCount(mSimdVarSpace.GetTotalVarCount());

}

int varCount = 3; // 3 possible arguments

functionBody->SetInParamsCount(4); // Always set 4 inParams so the memory space is the same (globalEnv,stdlib,foreign,buffer)

functionBody->SetReportedInParamsCount(4);

functionBody->SetConstantCount(2); // Return register + Root

functionBody->CreateConstantTable();

functionBody->SetVarCount(varCount);

functionBody->SetIsAsmjsMode(true);

functionBody->NewObjectLiteral(); // allocate one object literal for the export object

AsmJSByteCodeGenerator::EmitEmptyByteCode(funcInfo, GetByteCodeGenerator(), GetModuleFunctionNode());

// Create export module proxy

asmInfo->SetExportFunctionIndex(mExportFuncIndex);

asmInfo->SetExportsCount(mExports.Count());

for (int i = 0; i < mExports.Count(); i++)

{

AsmJsModuleExport& exMod = mExports.Item(i);

auto ex = asmInfo->GetExport(i);

\*ex.id = exMod.id;

\*ex.location = exMod.location;

}

int iVar = 0, iVarImp = 0, iFunc = 0, iFuncImp = 0;

const int size = mModuleEnvironment.Count();

asmInfo->InitializeSlotMap(size);

auto slotMap = asmInfo->GetAsmJsSlotMap();

for (int i = 0; i < size; i++)

{

AsmJsSymbol\* sym = mModuleEnvironment.GetValueAt(i);

if (sym)

{

AsmJsSlot \* slot = RecyclerNewLeaf(GetScriptContext()->GetRecycler(), AsmJsSlot);

slot->symType = sym->GetSymbolType();

slotMap->AddNew(sym->GetName()->GetPropertyId(), slot);

switch (sym->GetSymbolType())

{

case AsmJsSymbol::Variable:{

AsmJsVar\* var = sym->Cast<AsmJsVar>();

auto& modVar = asmInfo->GetVar(iVar++);

modVar.location = var->GetLocation();

modVar.type = var->GetVarType().which();

if (var->GetVarType().isInt())

{

modVar.initialiser.intInit = var->GetIntInitialiser();

}

else if (var->GetVarType().isFloat())

{

modVar.initialiser.floatInit = var->GetFloatInitialiser();

}

else if (var->GetVarType().isDouble())

{

modVar.initialiser.doubleInit = var->GetDoubleInitialiser();

}

else if (IsSimdjsEnabled() && var->GetVarType().isSIMD())

{

modVar.initialiser.simdInit = var->GetSimdConstInitialiser();

}

else

{

Assert(UNREACHED);

}

modVar.isMutable = var->isMutable();

slot->location = modVar.location;

slot->varType = var->GetVarType().which();

slot->isConstVar = !modVar.isMutable;

break;

}

case AsmJsSymbol::ConstantImport:{

AsmJsConstantImport\* var = sym->Cast<AsmJsConstantImport>();

auto& modVar = asmInfo->GetVarImport(iVarImp++);

modVar.location = var->GetLocation();

modVar.field = var->GetField()->GetPropertyId();

modVar.type = var->GetVarType().which();

slot->location = modVar.location;

slot->varType = modVar.type;

break;

}

case AsmJsSymbol::ImportFunction:{

AsmJsImportFunction\* func = sym->Cast<AsmJsImportFunction>();

auto& modVar = asmInfo->GetFunctionImport(iFuncImp++);

modVar.location = func->GetFunctionIndex();

modVar.field = func->GetField()->GetPropertyId();

slot->location = modVar.location;

break;

}

case AsmJsSymbol::FuncPtrTable:{

AsmJsFunctionTable\* funcTable = sym->Cast<AsmJsFunctionTable>();

const uint size = funcTable->GetSize();

const RegSlot index = funcTable->GetFunctionIndex();

asmInfo->SetFunctionTableSize(index, size);

auto& modTable = asmInfo->GetFunctionTable(index);

for (uint j = 0; j < size; j++)

{

modTable.moduleFunctionIndex[j] = funcTable->GetModuleFunctionIndex(j);

}

slot->funcTableSize = size;

slot->location = index;

break;

}

case AsmJsSymbol::ModuleFunction:{

AsmJsFunc\* func = sym->Cast<AsmJsFunc>();

auto& modVar = asmInfo->GetFunction(iFunc++);

modVar.location = func->GetFunctionIndex();

slot->location = modVar.location;

break;

}

case AsmJsSymbol::ArrayView:

{

AsmJsArrayView \* var = sym->Cast<AsmJsArrayView>();

slot->viewType = var->GetViewType();

break;

}

case AsmJsSymbol::ModuleArgument:

{

AsmJsModuleArg \* arg = sym->Cast<AsmJsModuleArg>();

slot->argType = arg->GetArgType();

break;

}

// used only for module validation

case AsmJsSymbol::MathConstant:

{

AsmJsMathConst \* constVar = sym->Cast<AsmJsMathConst>();

slot->mathConstVal = \*constVar->GetVal();

break;

}

case AsmJsSymbol::MathBuiltinFunction:

{

AsmJsMathFunction \* mathFunc = sym->Cast<AsmJsMathFunction>();

slot->builtinMathFunc = mathFunc->GetMathBuiltInFunction();

break;

}

case AsmJsSymbol::TypedArrayBuiltinFunction:

{

AsmJsTypedArrayFunction \* mathFunc = sym->Cast<AsmJsTypedArrayFunction>();

slot->builtinArrayFunc = mathFunc->GetArrayBuiltInFunction();

break;

}

case AsmJsSymbol::SIMDBuiltinFunction:

{

AsmJsSIMDFunction \* mathFunc = sym->Cast<AsmJsSIMDFunction>();

slot->builtinSIMDFunc = mathFunc->GetSimdBuiltInFunction();

break;

}

default:

Assume(UNREACHED);

}

}

}

return true;

}

void AsmJsModuleCompiler::ASTPrepass(ParseNodePtr pnode, AsmJsFunc \* func)

{

ThreadContext::ProbeCurrentStackNoDispose(Js::Constants::MinStackByteCodeVisitor, GetByteCodeGenerator()->GetScriptContext());

if (pnode == NULL)

{

return;

}

switch (pnode->nop) {

// these first cases do the interesting work

case knopBreak:

case knopContinue:

GetByteCodeGenerator()->AddTargetStmt(pnode->sxJump.pnodeTarget);

break;

case knopInt:

func->AddConst<int>(pnode->sxInt.lw);

break;

case knopFlt:

{

const double d = pnode->sxFlt.dbl;

if (ParserWrapper::IsMinInt(pnode))

{

func->AddConst<int>((int)d);

}

else if (ParserWrapper::IsUnsigned(pnode))

{

func->AddConst<int>((int)(uint32)d);

}

else

{

func->AddConst<double>(d);

}

break;

}

case knopName:

{

GetByteCodeGenerator()->AssignPropertyId(pnode->name());

AsmJsSymbol \* declSym = LookupIdentifier(pnode->name());

if (declSym)

{

if (declSym->GetSymbolType() == AsmJsSymbol::MathConstant)

{

AsmJsMathConst \* definition = declSym->Cast<AsmJsMathConst>();

Assert(definition->GetType().isDouble());

func->AddConst<double>(\*definition->GetVal());

}

else if (declSym->GetSymbolType() == AsmJsSymbol::Variable && !declSym->isMutable())

{

AsmJsVar \* definition = declSym->Cast<AsmJsVar>();

switch (definition->GetVarType().which())

{

case AsmJsVarType::Double:

func->AddConst<double>(definition->GetDoubleInitialiser());

break;

case AsmJsVarType::Float:

func->AddConst<float>(definition->GetFloatInitialiser());

break;

case AsmJsVarType::Int:

func->AddConst<int>(definition->GetIntInitialiser());

break;

default:

Assume(UNREACHED);

}

}

}

break;

}

case knopCall:

{

ASTPrepass(pnode->sxCall.pnodeTarget, func);

bool evalArgs = true;

if (pnode->sxCall.pnodeTarget->nop == knopName)

{

AsmJsFunctionDeclaration\* funcDecl = this->LookupFunction(pnode->sxCall.pnodeTarget->name());

if (funcDecl && funcDecl->GetSymbolType() == AsmJsSymbol::MathBuiltinFunction)

{

AsmJsMathFunction\* mathFunc = funcDecl->Cast<AsmJsMathFunction>();

if (mathFunc->GetMathBuiltInFunction() == AsmJSMathBuiltin\_fround)

{

switch (pnode->sxCall.pnodeArgs->nop)

{

case knopFlt:

func->AddConst<float>((float)pnode->sxCall.pnodeArgs->sxFlt.dbl);

evalArgs = false;

break;

case knopInt:

func->AddConst<float>((float)pnode->sxCall.pnodeArgs->sxInt.lw);

evalArgs = false;

break;

case knopNeg:

if (pnode->sxCall.pnodeArgs->sxUni.pnode1->nop == knopInt && pnode->sxCall.pnodeArgs->sxUni.pnode1->sxInt.lw == 0)

{

func->AddConst<float>(-0.0f);

evalArgs = false;

break;

}

}

}

}

else if (IsSimdjsEnabled())

{

/\*

Float32x4 operations work on Float reg space.

If any of the args is a literal (DoubleLit), we need to have a copy of it in the Float reg space.

Note that we may end up with redundant copies in the Double reg space, since we ASTPrepass the args (Fix later ?)

\*/

if (funcDecl && funcDecl->GetSymbolType() == AsmJsSymbol::SIMDBuiltinFunction)

{

AsmJsSIMDFunction\* simdFunc = funcDecl->Cast<AsmJsSIMDFunction>();

if (simdFunc->IsFloat32x4Func())

{

ParseNode \*argNode, \*arg;

argNode = arg = pnode->sxCall.pnodeArgs;

do

{

if (argNode->nop == knopList)

{

arg = ParserWrapper::GetBinaryLeft(argNode);

argNode = ParserWrapper::GetBinaryRight(argNode);

}

if (arg->nop == knopFlt)

{

func->AddConst<float>((float)arg->sxFlt.dbl);

}

if (argNode != arg && argNode->nop == knopFlt)

{ // last arg

func->AddConst<float>((float)argNode->sxFlt.dbl);

}

} while (argNode->nop == knopList);

}

}

}

}

if (evalArgs)

{

ASTPrepass(pnode->sxCall.pnodeArgs, func);

}

break;

}

case knopVarDecl:

GetByteCodeGenerator()->AssignPropertyId(pnode->name());

ASTPrepass(pnode->sxVar.pnodeInit, func);

break;

// all the rest of the cases simply walk the AST

case knopQmark:

ASTPrepass(pnode->sxTri.pnode1, func);

ASTPrepass(pnode->sxTri.pnode2, func);

ASTPrepass(pnode->sxTri.pnode3, func);

break;

case knopList:

do

{

ParseNode \* pnode1 = pnode->sxBin.pnode1;

ASTPrepass(pnode1, func);

pnode = pnode->sxBin.pnode2;

} while (pnode->nop == knopList);

ASTPrepass(pnode, func);

break;

case knopFor:

ASTPrepass(pnode->sxFor.pnodeInit, func);

ASTPrepass(pnode->sxFor.pnodeCond, func);

ASTPrepass(pnode->sxFor.pnodeIncr, func);

ASTPrepass(pnode->sxFor.pnodeBody, func);

break;

case knopIf:

ASTPrepass(pnode->sxIf.pnodeCond, func);

ASTPrepass(pnode->sxIf.pnodeTrue, func);

ASTPrepass(pnode->sxIf.pnodeFalse, func);

break;

case knopDoWhile:

case knopWhile:

ASTPrepass(pnode->sxWhile.pnodeCond, func);

ASTPrepass(pnode->sxWhile.pnodeBody, func);

break;

case knopReturn:

ASTPrepass(pnode->sxReturn.pnodeExpr, func);

break;

case knopBlock:

ASTPrepass(pnode->sxBlock.pnodeStmt, func);

break;

case knopSwitch:

ASTPrepass(pnode->sxSwitch.pnodeVal, func);

for (ParseNode \*pnodeT = pnode->sxSwitch.pnodeCases; NULL != pnodeT; pnodeT = pnodeT->sxCase.pnodeNext)

{

ASTPrepass(pnodeT, func);

}

ASTPrepass(pnode->sxSwitch.pnodeBlock, func);

break;

case knopCase:

ASTPrepass(pnode->sxCase.pnodeExpr, func);

ASTPrepass(pnode->sxCase.pnodeBody, func);

break;

case knopComma:

{

ParseNode \*pnode1 = pnode->sxBin.pnode1;

if (pnode1->nop == knopComma)

{

// avoid recursion on very large comma expressions.

ArenaAllocator \*alloc = GetByteCodeGenerator()->GetAllocator();

SList<ParseNode\*> \*rhsStack = Anew(alloc, SList<ParseNode\*>, alloc);

do {

rhsStack->Push(pnode1->sxBin.pnode2);

pnode1 = pnode1->sxBin.pnode1;

} while (pnode1->nop == knopComma);

ASTPrepass(pnode1, func);

while (!rhsStack->Empty())

{

ParseNode \*pnodeRhs = rhsStack->Pop();

ASTPrepass(pnodeRhs, func);

}

Adelete(alloc, rhsStack);

}

else

{

ASTPrepass(pnode1, func);

}

ASTPrepass(pnode->sxBin.pnode2, func);

break;

}

default:

{

uint flags = ParseNode::Grfnop(pnode->nop);

if (flags&fnopUni)

{

ASTPrepass(pnode->sxUni.pnode1, func);

}

else if (flags&fnopBin)

{

ASTPrepass(pnode->sxBin.pnode1, func);

ASTPrepass(pnode->sxBin.pnode2, func);

}

break;

}

}

}

void AsmJsModuleCompiler::BindArguments(ParseNode\* argList)

{

for (ParseNode\* pnode = argList; pnode; pnode = pnode->sxVar.pnodeNext)

{

GetByteCodeGenerator()->AssignPropertyId(pnode->name());

}

}

bool AsmJsModuleCompiler::CompileFunction(AsmJsFunc \* func, int funcIndex)

{

ParseNodePtr fncNode = func->GetFncNode();

ParseNodePtr pnodeBody = nullptr;

Assert(fncNode->nop == knopFncDecl && fncNode->sxFnc.funcInfo && fncNode->sxFnc.funcInfo->IsDeferred() && fncNode->sxFnc.pnodeBody == NULL);

Js::ParseableFunctionInfo\* deferParseFunction = fncNode->sxFnc.funcInfo->byteCodeFunction;

Utf8SourceInfo \* utf8SourceInfo = deferParseFunction->GetUtf8SourceInfo();

ULONG grfscr = utf8SourceInfo->GetParseFlags();

grfscr = grfscr & (~fscrGlobalCode);

func->SetOrigParseFlags(grfscr);

deferParseFunction->SetGrfscr(grfscr | (grfscr & ~fscrDeferredFncExpression));

deferParseFunction->SetSourceInfo(GetByteCodeGenerator()->GetCurrentSourceIndex(),

fncNode,

!!(grfscr & fscrEvalCode),

((grfscr & fscrDynamicCode) && !(grfscr & fscrEvalCode)));

deferParseFunction->SetInParamsCount(fncNode->sxFnc.funcInfo->inArgsCount);

deferParseFunction->SetReportedInParamsCount(fncNode->sxFnc.funcInfo->inArgsCount);

if (fncNode->sxFnc.pnodeBody == NULL)

{

if (GetScriptContext()->GetConfig()->BindDeferredPidRefs() &&

!PHASE\_OFF1(Js::SkipNestedDeferredPhase))

{

deferParseFunction->BuildDeferredStubs(fncNode);

}

}

deferParseFunction->SetIsAsmjsMode(true);

PageAllocator tempPageAlloc(NULL, Js::Configuration::Global.flags);

Parser ps(GetScriptContext(), FALSE, &tempPageAlloc);

FunctionBody \* funcBody;

ParseNodePtr parseTree;

Assert(!deferParseFunction->GetIsStrictMode());

CompileScriptException se;

funcBody = deferParseFunction->ParseAsmJs(&ps, &se, &parseTree);

TRACE\_BYTECODE(L"\nDeferred parse %s\n", funcBody->GetDisplayName());

if (parseTree && parseTree->nop == knopProg)

{

auto body = parseTree->sxProg.pnodeBody;

if (body && body->nop == knopList)

{

auto fncDecl = body->sxBin.pnode1;

if (fncDecl && fncDecl->nop == knopFncDecl)

{

pnodeBody = fncDecl->sxFnc.pnodeBody;

func->SetFuncBody(funcBody);

}

}

}

GetByteCodeGenerator()->PushFuncInfo(L"Start asm.js AST prepass", fncNode->sxFnc.funcInfo);

fncNode->sxFnc.funcInfo->byteCodeFunction->SetBoundPropertyRecords(GetByteCodeGenerator()->EnsurePropertyRecordList());

BindArguments(fncNode->sxFnc.pnodeArgs);

ASTPrepass(pnodeBody, func);

GetByteCodeGenerator()->PopFuncInfo(L"End asm.js AST prepass");

fncNode->sxFnc.pnodeBody = pnodeBody;

if (!pnodeBody)

{

// body should never be null if parsing succeeded

Assert(UNREACHED);

return Fail(fncNode, L"Function should always have parse nodes");

}

// Check if this function requires a bigger Ast

UpdateMaxAstSize(fncNode->sxFnc.astSize);

if (funcIndex == 0 && CheckChangeHeap(func))

{

fncNode->sxFnc.pnodeBody = NULL;

return true;

}

if (!SetupFunctionArguments(func, pnodeBody))

{

// failure message will be printed by SetupFunctionArguments

fncNode->sxFnc.pnodeBody = NULL;

return false;

}

if (!SetupLocalVariables(func))

{

// failure message will be printed by SetupLocalVariables

fncNode->sxFnc.pnodeBody = NULL;

return false;

}

// now that we have setup the function, we can generate bytecode for it

AsmJSByteCodeGenerator gen(func, this);

bool wasEmit = gen.EmitOneFunction();

fncNode->sxFnc.pnodeBody = NULL;

return wasEmit;

}

bool AsmJsModuleCompiler::SetupFunctionArguments(AsmJsFunc \* func, ParseNodePtr pnode)

{

// Check arguments

ArgSlot numArguments = 0;

ParseNode \* fncNode = func->GetFncNode();

ParseNode\* argNode = ParserWrapper::FunctionArgsList(fncNode, numArguments);

if (!func->EnsureArgCount(numArguments))

{

return Fail(argNode, L"Cannot have variable number of arguments");

}

ArgSlot index = 0;

while (argNode)

{

if (pnode->nop != knopList)

{

return Fail(pnode, L"Missing assignment statement for argument");

}

if (!ParserWrapper::IsDefinition(argNode))

{

return Fail(argNode, L"duplicate argument name not allowed");

}

PropertyName argName = argNode->name();

if (!AsmJSCompiler::CheckIdentifier(\*this, argNode, argName))

{

return false;

}

// creates the variable

AsmJsVarBase\* var = func->DefineVar(argName, true);

if (!var)

{

return Fail(argNode, L"Failed to define var");

}

ParseNode\* argDefinition = ParserWrapper::GetBinaryLeft(pnode);

if (argDefinition->nop != knopAsg)

{

return Fail(argDefinition, L"Expecting an assignment");

}

ParseNode\* lhs = ParserWrapper::GetBinaryLeft(argDefinition);

ParseNode\* rhs = ParserWrapper::GetBinaryRight(argDefinition);

#define NodeDefineThisArgument(n,var) (n->nop == knopName && ParserWrapper::VariableName(n)->GetPropertyId() == var->GetName()->GetPropertyId())

if (!NodeDefineThisArgument(lhs, var))

{

return Fail(lhs, L"Defining wrong argument");

}

if (rhs->nop == knopPos)

{

// unary + => double

var->SetVarType(AsmJsVarType::Double);

var->SetLocation(func->AcquireRegister<double>());

// validate stmt

ParseNode\* argSym = ParserWrapper::GetUnaryNode(rhs);

if (!NodeDefineThisArgument(argSym, var))

{

return Fail(lhs, L"Defining wrong argument");

}

}

else if (rhs->nop == knopOr)

{

var->SetVarType(AsmJsVarType::Int);

var->SetLocation(func->AcquireRegister<int>());

ParseNode\* argSym = ParserWrapper::GetBinaryLeft(rhs);

ParseNode\* intSym = ParserWrapper::GetBinaryRight(rhs);

// validate stmt

if (!NodeDefineThisArgument(argSym, var))

{

return Fail(lhs, L"Defining wrong argument");

}

if (intSym->nop != knopInt || intSym->sxInt.lw != 0)

{

return Fail(lhs, L"Or value must be 0 when defining arguments");

}

}

else if (rhs->nop == knopCall)

{

if (rhs->sxCall.pnodeTarget->nop != knopName)

{

return Fail(rhs, L"call should be for fround");

}

AsmJsFunctionDeclaration\* funcDecl = this->LookupFunction(rhs->sxCall.pnodeTarget->name());

if (!funcDecl)

return Fail(rhs, L"Cannot resolve function for argument definition, or wrong function");

if (funcDecl->GetSymbolType() == AsmJsSymbol::MathBuiltinFunction)

{

AsmJsMathFunction\* mathFunc = funcDecl->Cast<AsmJsMathFunction>();

if (!(mathFunc && mathFunc->GetMathBuiltInFunction() == AsmJSMathBuiltin\_fround))

{

return Fail(rhs, L"call should be for fround");

}

var->SetVarType(AsmJsVarType::Float);

var->SetLocation(func->AcquireRegister<float>());

}

else if (IsSimdjsEnabled() && funcDecl->GetSymbolType() == AsmJsSymbol::SIMDBuiltinFunction)

{

AsmJsSIMDFunction\* simdFunc = funcDecl->Cast<AsmJsSIMDFunction>();

// x = f4check(x)

if (!simdFunc->IsTypeCheck())

{

return Fail(rhs, L"Invalid SIMD argument type check. E.g. expected x = f4check(x)");

}

var->SetVarType(simdFunc->GetTypeCheckVarType());

// We don't set SIMD args reg location here. We defer that after all function locals are processed.

// This allows us to capture all SIMD constants from locals initializations, add them to the register space before we assign registers to args and locals.

func->GetSimdVarsList().Add(var);

}

else

{

return Fail(rhs, L"Wrong function used for argument definition");

}

if (!NodeDefineThisArgument(rhs->sxCall.pnodeArgs, var))

{

return Fail(lhs, L"Defining wrong argument");

}

}

else

{

return Fail(rhs, L"arguments are not casted as valid Asm.js type");

}

if (PHASE\_TRACE1(ByteCodePhase))

{

Output::Print(L" Argument [%s] Valid", argName->Psz());

}

if (!func->EnsureArgType(var, index++))

{

return Fail(rhs, L"Unexpected argument type");

}

argNode = ParserWrapper::NextVar(argNode);

pnode = ParserWrapper::GetBinaryRight(pnode);

}

func->SetBodyNode(pnode);

return true;

}

bool AsmJsModuleCompiler::SetupLocalVariables(AsmJsFunc \* func)

{

ParseNodePtr pnode = func->GetBodyNode();

MathBuiltin mathBuiltin;

AsmJsMathFunction\* mathFunc = nullptr;

AsmJsSIMDFunction\* simdFunc = nullptr;

AsmJsSIMDValue simdValue;

simdValue.Zero();

// define all variables

while (pnode->nop == knopList)

{

ParseNode \* varNode = ParserWrapper::GetBinaryLeft(pnode);

while (varNode && varNode->nop != knopEndCode)

{

ParseNode \* decl;

if (varNode->nop == knopList)

{

decl = ParserWrapper::GetBinaryLeft(varNode);

varNode = ParserWrapper::GetBinaryRight(varNode);

}

else

{

decl = varNode;

varNode = nullptr;

}

// if we have hit a non-declaration, we are done processing the function header

if (decl->nop != knopVarDecl)

{

goto varDeclEnd;

}

ParseNode\* pnodeInit = decl->sxVar.pnodeInit;

AsmJsSymbol \* declSym = nullptr;

mathFunc = nullptr;

simdFunc = nullptr;

if (pnodeInit->nop == knopName)

{

declSym = LookupIdentifier(pnodeInit->name(), func);

if (!declSym || declSym->isMutable() || (declSym->GetSymbolType() != AsmJsSymbol::Variable && declSym->GetSymbolType() != AsmJsSymbol::MathConstant))

{

return Fail(decl, L"Var declaration with non-constant");

}

}

else if (pnodeInit->nop == knopCall)

{

if (pnodeInit->sxCall.pnodeTarget->nop != knopName)

{

return Fail(decl, L"Var declaration with something else than a literal value|fround call");

}

AsmJsFunctionDeclaration\* funcDecl = this->LookupFunction(pnodeInit->sxCall.pnodeTarget->name());

if (!funcDecl)

return Fail(pnodeInit, L"Cannot resolve function name");

if (funcDecl->GetSymbolType() == AsmJsSymbol::MathBuiltinFunction)

{

mathFunc = funcDecl->Cast<AsmJsMathFunction>();

if (!(mathFunc && mathFunc->GetMathBuiltInFunction() == AsmJSMathBuiltin\_fround))

{

return Fail(decl, L"Var declaration with something else than a literal value|fround call");

}

if (!ParserWrapper::IsFroundNumericLiteral(pnodeInit->sxCall.pnodeArgs))

{

return Fail(decl, L"Var declaration with something else than a literal value|fround call");

}

}

else if (IsSimdjsEnabled() && funcDecl->GetSymbolType() == AsmJsSymbol::SIMDBuiltinFunction)

{

// var x = f4(1.0, 2.0, 3.0, 4.0);

simdFunc = funcDecl->Cast<AsmJsSIMDFunction>();

if (!ValidateSimdConstructor(pnodeInit, simdFunc, simdValue))

{

return Fail(varNode, L"Invalid SIMD local declaration");

}

}

}

else if (pnodeInit->nop != knopInt && pnodeInit->nop != knopFlt)

{

return Fail(decl, L"Var declaration with something else than a literal value|fround call");

}

if (!AsmJSCompiler::CheckIdentifier(\*this, decl, decl->name()))

{

// CheckIdentifier will print failure message

return false;

}

AsmJsVar\* var = (AsmJsVar\*)func->DefineVar(decl->name(), false);

if (!var)

{

return Fail(decl, L"Failed to define var");

}

RegSlot loc = Constants::NoRegister;

if (pnodeInit->nop == knopInt)

{

var->SetVarType(AsmJsVarType::Int);

var->SetLocation(func->AcquireRegister<int>());

var->SetConstInitialiser(pnodeInit->sxInt.lw);

loc = func->GetConstRegister<int>(pnodeInit->sxInt.lw);

}

else if (ParserWrapper::IsMinInt(pnodeInit))

{

var->SetVarType(AsmJsVarType::Int);

var->SetLocation(func->AcquireRegister<int>());

var->SetConstInitialiser(MININT);

loc = func->GetConstRegister<int>(MININT);

}

else if (ParserWrapper::IsUnsigned(pnodeInit))

{

var->SetVarType(AsmJsVarType::Int);

var->SetLocation(func->AcquireRegister<int>());

var->SetConstInitialiser((int)((uint32)pnodeInit->sxFlt.dbl));

loc = func->GetConstRegister<int>((uint32)pnodeInit->sxFlt.dbl);

}

else if (pnodeInit->nop == knopFlt)

{

if (pnodeInit->sxFlt.maybeInt)

{

return Fail(decl, L"Var declaration with integer literal outside range [-2^31, 2^32)");

}

var->SetVarType(AsmJsVarType::Double);

var->SetLocation(func->AcquireRegister<double>());

loc = func->GetConstRegister<double>(pnodeInit->sxFlt.dbl);

var->SetConstInitialiser(pnodeInit->sxFlt.dbl);

}

else if (pnodeInit->nop == knopName)

{

if (declSym->GetSymbolType() == AsmJsSymbol::Variable)

{

AsmJsVar \* definition = declSym->Cast<AsmJsVar>();

switch (definition->GetVarType().which())

{

case AsmJsVarType::Double:

var->SetVarType(AsmJsVarType::Double);

var->SetLocation(func->AcquireRegister<double>());

var->SetConstInitialiser(definition->GetDoubleInitialiser());

break;

case AsmJsVarType::Float:

var->SetVarType(AsmJsVarType::Float);

var->SetLocation(func->AcquireRegister<float>());

var->SetConstInitialiser(definition->GetFloatInitialiser());

break;

case AsmJsVarType::Int:

var->SetVarType(AsmJsVarType::Int);

var->SetLocation(func->AcquireRegister<int>());

var->SetConstInitialiser(definition->GetIntInitialiser());

break;

default:

Assume(UNREACHED);

}

}

else

{

Assert(declSym->GetSymbolType() == AsmJsSymbol::MathConstant);

Assert(declSym->GetType() == AsmJsType::Double);

AsmJsMathConst \* definition = declSym->Cast<AsmJsMathConst>();

var->SetVarType(AsmJsVarType::Double);

var->SetLocation(func->AcquireRegister<double>());

var->SetConstInitialiser(\*definition->GetVal());

}

}

else if (pnodeInit->nop == knopCall)

{

if (mathFunc)

{

var->SetVarType(AsmJsVarType::Float);

var->SetLocation(func->AcquireRegister<float>());

if (pnodeInit->sxCall.pnodeArgs->nop == knopInt)

{

int iVal = pnodeInit->sxCall.pnodeArgs->sxInt.lw;

var->SetConstInitialiser((float)iVal);

loc = func->GetConstRegister<float>((float)iVal);

}

else if (ParserWrapper::IsNegativeZero(pnodeInit->sxCall.pnodeArgs))

{

var->SetConstInitialiser(-0.0f);

loc = func->GetConstRegister<float>(-0.0f);

}

else

{

// note: fround((-)NumericLiteral) is explicitly allowed for any range, so we do not need to check for maybeInt

Assert(pnodeInit->sxCall.pnodeArgs->nop == knopFlt);

float fVal = (float)pnodeInit->sxCall.pnodeArgs->sxFlt.dbl;

var->SetConstInitialiser((float)fVal);

loc = func->GetConstRegister<float>(fVal);

}

}

else if (IsSimdjsEnabled() && simdFunc)

{

// simd constructor call

// en-register the simdvalue constant first

func->AddConst<AsmJsSIMDValue>(simdValue);

loc = func->GetConstRegister<AsmJsSIMDValue>(simdValue);

var->SetConstInitialiser(simdValue);

var->SetVarType(simdFunc->GetConstructorVarType());

// add to list. assign register after all constants.

func->GetSimdVarsList().Add(var);

}

else

{

Assert(UNREACHED);

}

}

if (loc == Constants::NoRegister && pnodeInit->nop != knopName)

{

return Fail(decl, L"Cannot find Register constant for var");

}

}

if (ParserWrapper::GetBinaryRight(pnode)->nop == knopEndCode)

{

break;

}

pnode = ParserWrapper::GetBinaryRight(pnode);

}

varDeclEnd:

// this code has to be on all exit-path from the function

if (IsSimdjsEnabled())

{

// Now, assign registers to all SIMD vars after all constants are en-registered.

for (int i = 0; i < func->GetSimdVarsList().Count(); i++)

{

AsmJsVarBase \*var = func->GetSimdVarsList().Item(i);

var->SetLocation(func->AcquireRegister<AsmJsSIMDValue>());

}

func->GetSimdVarsList().Reset(); // list not needed anymore

}

return true;

}

AsmJsFunc\* AsmJsModuleCompiler::CreateNewFunctionEntry( ParseNode\* pnodeFnc )

{

PropertyName name = ParserWrapper::FunctionName( pnodeFnc );

GetByteCodeGenerator()->AssignPropertyId(name);

AsmJsFunc\* func = Anew( &mAllocator, AsmJsFunc, name, pnodeFnc, &mAllocator );

if( func )

{

if( DefineIdentifier( name, func ) )

{

func->SetFunctionIndex( pnodeFnc->sxFnc.nestedIndex );

// Add extra check to make sure all the slots between 0 - Count are filled with func;

mFunctionArray.SetItem( func->GetFunctionIndex(), func );

return func;

}

// Error adding function

mAllocator.Free( func, sizeof( AsmJsFunc ) );

}

// Error allocating a new function

return nullptr;

}

bool AsmJsModuleCompiler::CheckChangeHeap(AsmJsFunc \* func)

{

ParseNode \* fncNode = func->GetFncNode();

ParseNode \* pnodeBody = fncNode->sxFnc.pnodeBody;

ParseNode \* pnodeArgs = fncNode->sxFnc.pnodeArgs;

// match AST for changeHeap function.

// it must be defined in the following format (names/whitespace can differ):

//function changeHeap(newBuffer)

//{

// if (byteLength(newBuffer) & 0xffffff ||

// byteLength(newBuffer) <= 0xffffff ||

// byteLength(newBuffer) > 0x80000000)

// return false;

// heap32 = new Int32Array(newBuffer);

// ...

// buffer = newBuffer;

// return true;

//}

// ensure function

if (pnodeBody->nop != knopList || !pnodeArgs || pnodeArgs->nop != knopVarDecl)

{

return false;

}

// ensure if expression

ParseNode \* ifNode = pnodeBody->sxBin.pnode1;

if (ifNode->nop != knopIf || ifNode->sxIf.pnodeFalse)

{

return false;

}

// validate "byteLength(newBuffer) > 0x80000000"

ParseNode \* orNode = ifNode->sxIf.pnodeCond;

if (orNode->nop != knopLogOr || orNode->sxBin.pnode1->nop != knopLogOr)

{

return false;

}

ParseNode \* cond = orNode->sxBin.pnode2;

if (cond->nop != knopGt || !CheckByteLengthCall(cond->sxBin.pnode1, pnodeArgs) || cond->sxBin.pnode2->nop != knopFlt || cond->sxBin.pnode2->sxFlt.dbl != 2147483648.0 || !cond->sxBin.pnode2->sxFlt.maybeInt)

{

return false;

}

// validate "byteLength(newBuffer) <= 0xffffff"

orNode = orNode->sxBin.pnode1;

cond = orNode->sxBin.pnode2;

if (cond->nop != knopLe || !CheckByteLengthCall(cond->sxBin.pnode1, pnodeArgs) || cond->sxBin.pnode2->nop != knopInt || cond->sxBin.pnode2->sxInt.lw != 0x00ffffff)

{

return false;

}

// validate "byteLength(newBuffer) & 0xffffff"

cond = orNode->sxBin.pnode1;

if (cond->nop != knopAnd || !CheckByteLengthCall(cond->sxBin.pnode1, pnodeArgs) || cond->sxBin.pnode2->nop != knopInt || cond->sxBin.pnode2->sxInt.lw != 0x00ffffff)

{

return false;

}

// validate "return false;"

cond = ifNode->sxIf.pnodeTrue;

if (!cond || cond->nop != knopReturn || cond->sxReturn.pnodeExpr->nop != knopFalse)

{

return false;

}

// validate heap32 = new Int32Array(newBuffer); etc.

while (!mArrayViews.Empty())

{

// all views that were instantiated must be replaced in the order which they were instantiated

AsmJsArrayView \* requiredArrayView = mArrayViews.Dequeue();

pnodeBody = pnodeBody->sxBin.pnode2;

if (pnodeBody->nop != knopList)

{

return false;

}

ParseNode \* assignNode = pnodeBody->sxBin.pnode1;

if (assignNode->nop != knopAsg || assignNode->sxBin.pnode1->nop != knopName)

{

return false;

}

// validate left hand side

AsmJsSymbol \* actualArraySym = LookupIdentifier(assignNode->sxBin.pnode1->name());

if (requiredArrayView != actualArraySym)

{

return false;

}

ParseNode \* callNode = assignNode->sxBin.pnode2;

// validate correct argument is passed

if (callNode->nop != knopNew || !callNode->sxCall.pnodeArgs || callNode->sxCall.pnodeArgs->nop != knopName || callNode->sxCall.pnodeArgs->name()->GetPropertyId() != pnodeArgs->name()->GetPropertyId() || callNode->sxCall.pnodeTarget->nop != knopName)

{

return false;

}

// validate correct function is being called

AsmJsSymbol \* callTargetSym = LookupIdentifier(callNode->sxCall.pnodeTarget->name());

if (!callTargetSym || callTargetSym->GetSymbolType() != AsmJsSymbol::TypedArrayBuiltinFunction)

{

return false;

}

if (requiredArrayView->GetViewType() != callTargetSym->Cast<AsmJsTypedArrayFunction>()->GetViewType())

{

return false;

}

}

pnodeBody = pnodeBody->sxBin.pnode2;

if (pnodeBody->nop != knopList)

{

return false;

}

// validate buffer = newBuffer;

ParseNode \* assign = pnodeBody->sxBin.pnode1;

if (assign->nop != knopAsg || assign->sxBin.pnode1->nop != knopName || !mBufferArgName || mBufferArgName->GetPropertyId() != assign->sxBin.pnode1->name()->GetPropertyId() ||

assign->sxBin.pnode2->nop != knopName || pnodeArgs->name()->GetPropertyId() != assign->sxBin.pnode2->name()->GetPropertyId())

{

return false;

}

// validate return true;

pnodeBody = pnodeBody->sxBin.pnode2;

if (pnodeBody->nop != knopList || pnodeBody->sxBin.pnode2->nop != knopEndCode ||

pnodeBody->sxBin.pnode1->nop != knopReturn || !pnodeBody->sxBin.pnode1->sxReturn.pnodeExpr || pnodeBody->sxBin.pnode1->sxReturn.pnodeExpr->nop != knopTrue)

{

return false;

}

// now we should flag this module as containing changeHeap method

mUsesChangeHeap = true;

AsmJSByteCodeGenerator::EmitEmptyByteCode(func->GetFuncInfo(), GetByteCodeGenerator(), fncNode);

return true;

}

bool AsmJsModuleCompiler::CheckByteLengthCall(ParseNode \* callNode, ParseNode \* bufferDecl)

{

if (callNode->nop != knopCall || callNode->sxCall.pnodeTarget->nop != knopName)

{

return false;

}

AsmJsSymbol\* funcDecl = LookupIdentifier(callNode->sxCall.pnodeTarget->name());

if (!funcDecl || funcDecl->GetSymbolType() != AsmJsSymbol::TypedArrayBuiltinFunction)

{

return false;

}

AsmJsTypedArrayFunction\* arrayFunc = funcDecl->Cast<AsmJsTypedArrayFunction>();

return callNode->sxCall.argCount == 1 &&

!callNode->sxCall.isApplyCall &&

!callNode->sxCall.isEvalCall &&

callNode->sxCall.spreadArgCount == 0 &&

arrayFunc->GetArrayBuiltInFunction() == AsmJSTypedArrayBuiltin\_byteLength &&

callNode->sxCall.pnodeArgs->nop == knopName &&

callNode->sxCall.pnodeArgs->name()->GetPropertyId() == bufferDecl->name()->GetPropertyId();

}

bool AsmJsModuleCompiler::Fail( ParseNode\* usepn, const wchar \*error )

{

AsmJSCompiler::OutputError(GetScriptContext(), error);

return false;

}

bool AsmJsModuleCompiler::FailName( ParseNode \*usepn, const wchar \*fmt, PropertyName name )

{

AsmJSCompiler::OutputError(GetScriptContext(), fmt, name->Psz());

return false;

}

bool AsmJsModuleCompiler::LookupStandardLibraryMathName( PropertyName name, MathBuiltin \*mathBuiltin ) const

{

return mStandardLibraryMathNames.TryGetValue( name->GetPropertyId(), mathBuiltin );

}

bool AsmJsModuleCompiler::LookupStandardLibraryArrayName(PropertyName name, TypedArrayBuiltin \*builtin) const

{

return mStandardLibraryArrayNames.TryGetValue(name->GetPropertyId(), builtin);

}

void AsmJsModuleCompiler::InitBufferArgName( PropertyName n )

{

#if DBG

Assert( !mBufferArgNameInit );

mBufferArgNameInit = true;

#endif

mBufferArgName = n;

}

void AsmJsModuleCompiler::InitForeignArgName( PropertyName n )

{

#if DBG

Assert( !mForeignArgNameInit );

mForeignArgNameInit = true;

#endif

mForeignArgName = n;

}

void AsmJsModuleCompiler::InitStdLibArgName( PropertyName n )

{

#if DBG

Assert( !mStdLibArgNameInit );

mStdLibArgNameInit = true;

#endif

mStdLibArgName = n;

}

Js::PropertyName AsmJsModuleCompiler::GetStdLibArgName() const

{

#if DBG

Assert( mBufferArgNameInit );

#endif

return mStdLibArgName;

}

Js::PropertyName AsmJsModuleCompiler::GetForeignArgName() const

{

#if DBG

Assert( mForeignArgNameInit );

#endif

return mForeignArgName;

}

Js::PropertyName AsmJsModuleCompiler::GetBufferArgName() const

{

#if DBG

Assert( mStdLibArgNameInit );

#endif

return mBufferArgName;

}

bool AsmJsModuleCompiler::Init()

{

if( mInitialised )

{

return false;

}

mInitialised = true;

struct MathFunc

{

MathFunc( PropertyId id\_ = 0, AsmJsMathFunction\* val\_ = nullptr ) :

id( id\_ ), val( val\_ )

{

}

PropertyId id;

AsmJsMathFunction\* val;

};

MathFunc mathFunctions[AsmJSMathBuiltinFunction\_COUNT];

// we could move the mathBuiltinFuncname to MathFunc struct

mathFunctions[AsmJSMathBuiltin\_sin ] = MathFunc(PropertyIds::sin , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_sin , OpCodeAsmJs::Sin\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_cos ] = MathFunc(PropertyIds::cos , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_cos , OpCodeAsmJs::Cos\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_tan ] = MathFunc(PropertyIds::tan , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_tan , OpCodeAsmJs::Tan\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_asin ] = MathFunc(PropertyIds::asin , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_asin , OpCodeAsmJs::Asin\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_acos ] = MathFunc(PropertyIds::acos , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_acos , OpCodeAsmJs::Acos\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_atan ] = MathFunc(PropertyIds::atan , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_atan , OpCodeAsmJs::Atan\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_ceil ] = MathFunc(PropertyIds::ceil , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_ceil , OpCodeAsmJs::Ceil\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_floor ] = MathFunc(PropertyIds::floor , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_floor , OpCodeAsmJs::Floor\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_exp ] = MathFunc(PropertyIds::exp , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_exp , OpCodeAsmJs::Exp\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_log ] = MathFunc(PropertyIds::log , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_log , OpCodeAsmJs::Log\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_pow ] = MathFunc(PropertyIds::pow , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 2, AsmJSMathBuiltin\_pow , OpCodeAsmJs::Pow\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_sqrt ] = MathFunc(PropertyIds::sqrt , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_sqrt , OpCodeAsmJs::Sqrt\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_abs ] = MathFunc(PropertyIds::abs , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_abs , OpCodeAsmJs::Abs\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_atan2 ] = MathFunc(PropertyIds::atan2 , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 2, AsmJSMathBuiltin\_atan2 , OpCodeAsmJs::Atan2\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble, AsmJsType::MaybeDouble ));

mathFunctions[AsmJSMathBuiltin\_imul ] = MathFunc(PropertyIds::imul , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 2, AsmJSMathBuiltin\_imul , OpCodeAsmJs::Imul\_Int , AsmJsRetType::Signed, AsmJsType::Intish , AsmJsType::Intish ));

mathFunctions[AsmJSMathBuiltin\_fround] = MathFunc(PropertyIds::fround, Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_fround, OpCodeAsmJs::Fround\_Flt,AsmJsRetType::Float , AsmJsType::Floatish ));

mathFunctions[AsmJSMathBuiltin\_min ] = MathFunc(PropertyIds::min , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 2, AsmJSMathBuiltin\_min , OpCodeAsmJs::Min\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble, AsmJsType::MaybeDouble));

mathFunctions[AsmJSMathBuiltin\_max ] = MathFunc(PropertyIds::max , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 2, AsmJSMathBuiltin\_max , OpCodeAsmJs::Max\_Db , AsmJsRetType::Double, AsmJsType::MaybeDouble, AsmJsType::MaybeDouble));

mathFunctions[AsmJSMathBuiltin\_clz32 ] = MathFunc(PropertyIds::clz32 , Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_clz32 , OpCodeAsmJs::Clz32\_Int, AsmJsRetType::Fixnum, AsmJsType::Intish));

mathFunctions[AsmJSMathBuiltin\_abs].val->SetOverload(Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_abs, OpCodeAsmJs::Abs\_Int, AsmJsRetType::Unsigned, AsmJsType::Signed));

mathFunctions[AsmJSMathBuiltin\_min].val->SetOverload(Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 2, AsmJSMathBuiltin\_min, OpCodeAsmJs::Min\_Int, AsmJsRetType::Signed, AsmJsType::Signed, AsmJsType::Signed));

mathFunctions[AsmJSMathBuiltin\_max].val->SetOverload(Anew( &mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 2, AsmJSMathBuiltin\_max, OpCodeAsmJs::Max\_Int, AsmJsRetType::Signed, AsmJsType::Signed, AsmJsType::Signed));

//Float Overloads

mathFunctions[AsmJSMathBuiltin\_fround].val->SetOverload(Anew(&mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_fround, OpCodeAsmJs::Fround\_Db, AsmJsRetType::Float, AsmJsType::MaybeDouble));

mathFunctions[AsmJSMathBuiltin\_fround].val->SetOverload(Anew(&mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_fround, OpCodeAsmJs::Fround\_Int, AsmJsRetType::Float, AsmJsType::Int));// should we split this into signed and unsigned?

mathFunctions[AsmJSMathBuiltin\_abs].val->SetOverload( Anew(&mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_abs, OpCodeAsmJs::Abs\_Flt, AsmJsRetType::Floatish, AsmJsType::MaybeFloat));

mathFunctions[AsmJSMathBuiltin\_ceil].val->SetOverload( Anew(&mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_ceil, OpCodeAsmJs::Ceil\_Flt, AsmJsRetType::Floatish, AsmJsType::MaybeFloat));

mathFunctions[AsmJSMathBuiltin\_floor].val->SetOverload( Anew(&mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_floor, OpCodeAsmJs::Floor\_Flt, AsmJsRetType::Floatish, AsmJsType::MaybeFloat));

mathFunctions[AsmJSMathBuiltin\_sqrt].val->SetOverload( Anew(&mAllocator, AsmJsMathFunction, nullptr, &mAllocator, 1, AsmJSMathBuiltin\_sqrt, OpCodeAsmJs::Sqrt\_Flt, AsmJsRetType::Floatish, AsmJsType::MaybeFloat));

for (int i = 0; i < AsmJSMathBuiltinFunction\_COUNT ; i++)

{

if( !AddStandardLibraryMathName( (PropertyId)mathFunctions[i].id, mathFunctions[i].val, mathFunctions[i].val->GetMathBuiltInFunction() ) )

{

return false;

}

}

struct ConstMath

{

ConstMath( PropertyId id\_, const double\* val\_, AsmJSMathBuiltinFunction mathLibConstName\_):

id(id\_), val(val\_), mathLibConstName(mathLibConstName\_) { }

PropertyId id;

AsmJSMathBuiltinFunction mathLibConstName;

const double\* val;

};

ConstMath constMath[] = {

ConstMath( PropertyIds::E , &Math::E , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_e ),

ConstMath(PropertyIds::LN10 , &Math::LN10 , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_ln10),

ConstMath(PropertyIds::LN2 , &Math::LN2 , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_ln2),

ConstMath(PropertyIds::LOG2E , &Math::LOG2E , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_log2e),

ConstMath(PropertyIds::LOG10E , &Math::LOG10E , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_log10e),

ConstMath(PropertyIds::PI , &Math::PI , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_pi),

ConstMath(PropertyIds::SQRT1\_2 , &Math::SQRT1\_2 , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_sqrt1\_2),

ConstMath(PropertyIds::SQRT2 , &Math::SQRT2 , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_sqrt2),

ConstMath(PropertyIds::Infinity , &NumberConstants::POSITIVE\_INFINITY, AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_infinity),

ConstMath(PropertyIds::NaN , &NumberConstants::NaN , AsmJSMathBuiltinFunction::AsmJSMathBuiltin\_nan),

};

const int size = sizeof( constMath ) / sizeof( ConstMath );

for (int i = 0; i < size ; i++)

{

if( !AddStandardLibraryMathName( constMath[i].id, constMath[i].val, constMath[i].mathLibConstName ) )

{

return false;

}

}

struct ArrayFunc

{

ArrayFunc(PropertyId id\_ = 0, AsmJsTypedArrayFunction\* val\_ = nullptr) :

id(id\_), val(val\_)

{

}

PropertyId id;

AsmJsTypedArrayFunction\* val;

};

ArrayFunc arrayFunctions[AsmJSMathBuiltinFunction\_COUNT];

arrayFunctions[AsmJSTypedArrayBuiltin\_Int8Array ] = ArrayFunc(PropertyIds::Int8Array, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_Int8Array, ArrayBufferView::TYPE\_INT8));

arrayFunctions[AsmJSTypedArrayBuiltin\_Uint8Array ] = ArrayFunc(PropertyIds::Uint8Array, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_Uint8Array, ArrayBufferView::TYPE\_UINT8));

arrayFunctions[AsmJSTypedArrayBuiltin\_Int16Array ] = ArrayFunc(PropertyIds::Int16Array, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_Int16Array, ArrayBufferView::TYPE\_INT16));

arrayFunctions[AsmJSTypedArrayBuiltin\_Uint16Array ] = ArrayFunc(PropertyIds::Uint16Array, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_Uint16Array, ArrayBufferView::TYPE\_UINT16));

arrayFunctions[AsmJSTypedArrayBuiltin\_Int32Array ] = ArrayFunc(PropertyIds::Int32Array, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_Int32Array, ArrayBufferView::TYPE\_INT32));

arrayFunctions[AsmJSTypedArrayBuiltin\_Uint32Array ] = ArrayFunc(PropertyIds::Uint32Array, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_Uint32Array, ArrayBufferView::TYPE\_UINT32));

arrayFunctions[AsmJSTypedArrayBuiltin\_Float32Array] = ArrayFunc(PropertyIds::Float32Array, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_Float32Array, ArrayBufferView::TYPE\_FLOAT32));

arrayFunctions[AsmJSTypedArrayBuiltin\_Float64Array] = ArrayFunc(PropertyIds::Float64Array, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_Float64Array, ArrayBufferView::TYPE\_FLOAT64));

arrayFunctions[AsmJSTypedArrayBuiltin\_byteLength ] = ArrayFunc(PropertyIds::byteLength, Anew(&mAllocator, AsmJsTypedArrayFunction, nullptr, &mAllocator, AsmJSTypedArrayBuiltin\_byteLength, ArrayBufferView::TYPE\_INVALID));

for (int i = 0; i < AsmJSTypedArrayBuiltin\_COUNT; i++)

{

if (!AddStandardLibraryArrayName((PropertyId)arrayFunctions[i].id, arrayFunctions[i].val, arrayFunctions[i].val->GetArrayBuiltInFunction()))

{

return false;

}

}

// similar to math functions maps initialization.

if (IsSimdjsEnabled())

{

if (!InitSIMDBuiltins())

{

return false;

}

}

return true;

}

bool AsmJsModuleCompiler::InitSIMDBuiltins()

{

struct SIMDFunc

{

SIMDFunc(PropertyId id\_ = 0, AsmJsSIMDFunction\* val\_ = nullptr) :

id(id\_), val(val\_)

{

}

PropertyId id;

AsmJsSIMDFunction\* val;

};

SIMDFunc simdFunctions[AsmJsSIMDBuiltin\_COUNT];

// !! NOTE: Keep these grouped by SIMD type

/\* Int32x4 builtins\*/

//-------------------

simdFunctions[AsmJsSIMDBuiltin\_Int32x4] = SIMDFunc(PropertyIds::Int32x4, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 4, AsmJsSIMDBuiltin\_Int32x4, OpCodeAsmJs::Simd128\_IntsToI4, AsmJsRetType::Int32x4, AsmJsType::Intish, AsmJsType::Intish, AsmJsType::Intish, AsmJsType::Intish));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_check] = SIMDFunc(PropertyIds::check, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_int32x4\_check, OpCodeAsmJs::Simd128\_Ld\_I4 /\*no dynamic checks\*/, AsmJsRetType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_splat] = SIMDFunc(PropertyIds::splat, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_int32x4\_splat, OpCodeAsmJs::Simd128\_Splat\_I4, AsmJsRetType::Int32x4, AsmJsType::Int));

// Q: Is this operation supported in ASMJS ? We don't have bool type.

//simdFunctions[AsmJsSIMDBuiltin\_int32x4\_bool] = SIMDFunc(PropertyIds::bool\_, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 4, AsmJsSIMDBuiltin\_int32x4\_bool, OpCodeAsmJs::Simd128\_Bool\_I4, AsmJsRetType::Int32x4, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_fromFloat64x2] = SIMDFunc(PropertyIds::fromFloat64x2, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_int32x4\_fromFloat64x2, OpCodeAsmJs::Simd128\_FromFloat64x2\_I4, AsmJsRetType::Int32x4, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_fromFloat64x2Bits] = SIMDFunc(PropertyIds::fromFloat64x2Bits, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_int32x4\_fromFloat64x2Bits, OpCodeAsmJs::Simd128\_FromFloat64x2Bits\_I4, AsmJsRetType::Int32x4, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_fromFloat32x4] = SIMDFunc(PropertyIds::fromFloat32x4, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_int32x4\_fromFloat32x4, OpCodeAsmJs::Simd128\_FromFloat32x4\_I4, AsmJsRetType::Int32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_fromFloat32x4Bits] = SIMDFunc(PropertyIds::fromFloat32x4Bits, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_int32x4\_fromFloat32x4Bits, OpCodeAsmJs::Simd128\_FromFloat32x4Bits\_I4, AsmJsRetType::Int32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_neg] = SIMDFunc(PropertyIds::neg, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_int32x4\_neg, OpCodeAsmJs::Simd128\_Neg\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_add] = SIMDFunc(PropertyIds::add, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_add, OpCodeAsmJs::Simd128\_Add\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_sub] = SIMDFunc(PropertyIds::sub, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_sub, OpCodeAsmJs::Simd128\_Sub\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_mul] = SIMDFunc(PropertyIds::mul, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_mul, OpCodeAsmJs::Simd128\_Mul\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

// TODO: Enable after fix in lib

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_swizzle] = SIMDFunc(PropertyIds::swizzle, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 5, AsmJsSIMDBuiltin\_int32x4\_swizzle, OpCodeAsmJs::Simd128\_Swizzle\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_shuffle] = SIMDFunc(PropertyIds::shuffle, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 6, AsmJsSIMDBuiltin\_int32x4\_shuffle, OpCodeAsmJs::Simd128\_Shuffle\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_extractLane] = SIMDFunc(PropertyIds::extractLane, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_extractLane, OpCodeAsmJs::Simd128\_ExtractLane\_I4, AsmJsRetType::Signed, AsmJsType::Int32x4, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_replaceLane] = SIMDFunc(PropertyIds::replaceLane, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_int32x4\_replaceLane, OpCodeAsmJs::Simd128\_ReplaceLane\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_lessThan] = SIMDFunc(PropertyIds::lessThan, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_lessThan, OpCodeAsmJs::Simd128\_Lt\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_equal] = SIMDFunc(PropertyIds::equal, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_equal, OpCodeAsmJs::Simd128\_Eq\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_greaterThan] = SIMDFunc(PropertyIds::greaterThan, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_greaterThan, OpCodeAsmJs::Simd128\_Gt\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_select] = SIMDFunc(PropertyIds::select, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_int32x4\_select, OpCodeAsmJs::Simd128\_Select\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_and] = SIMDFunc(PropertyIds::and, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_and, OpCodeAsmJs::Simd128\_And\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_or] = SIMDFunc(PropertyIds::or, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_or, OpCodeAsmJs::Simd128\_Or\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_xor] = SIMDFunc(PropertyIds::xor, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_xor, OpCodeAsmJs::Simd128\_Xor\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_not] = SIMDFunc(PropertyIds::not, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_int32x4\_not, OpCodeAsmJs::Simd128\_Not\_I4, AsmJsRetType::Int32x4, AsmJsType::Int32x4));

// Loads and Stores

// We fill Void for the tarray type. This is ok since we special handle these ops.

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_load] = SIMDFunc(PropertyIds::load, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_load, OpCodeAsmJs::Simd128\_LdArr\_I4, AsmJsRetType::Int32x4, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_load1] = SIMDFunc(PropertyIds::load1, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_load1, OpCodeAsmJs::Simd128\_LdArr\_I4, AsmJsRetType::Int32x4, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_load2] = SIMDFunc(PropertyIds::load2, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_load2, OpCodeAsmJs::Simd128\_LdArr\_I4, AsmJsRetType::Int32x4, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_load3] = SIMDFunc(PropertyIds::load3, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_int32x4\_load3, OpCodeAsmJs::Simd128\_LdArr\_I4, AsmJsRetType::Int32x4, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_store] = SIMDFunc(PropertyIds::store, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_int32x4\_store, OpCodeAsmJs::Simd128\_StArr\_I4, AsmJsRetType::Int32x4, AsmJsType::Void, AsmJsType::Int, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_store1] = SIMDFunc(PropertyIds::store1, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_int32x4\_store1, OpCodeAsmJs::Simd128\_StArr\_I4, AsmJsRetType::Int32x4, AsmJsType::Void, AsmJsType::Int, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_store2] = SIMDFunc(PropertyIds::store2, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_int32x4\_store2, OpCodeAsmJs::Simd128\_StArr\_I4, AsmJsRetType::Int32x4, AsmJsType::Void, AsmJsType::Int, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_int32x4\_store3] = SIMDFunc(PropertyIds::store3, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_int32x4\_store3, OpCodeAsmJs::Simd128\_StArr\_I4, AsmJsRetType::Int32x4, AsmJsType::Void, AsmJsType::Int, AsmJsType::Int32x4));

/\* Float32x4 builtins\*/

//-------------------

simdFunctions[AsmJsSIMDBuiltin\_Float32x4] = SIMDFunc(PropertyIds::Float32x4, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 4, AsmJsSIMDBuiltin\_Float32x4, OpCodeAsmJs::Simd128\_FloatsToF4, AsmJsRetType::Float32x4, AsmJsType::FloatishDoubleLit, AsmJsType::FloatishDoubleLit, AsmJsType::FloatishDoubleLit, AsmJsType::FloatishDoubleLit));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_check] = SIMDFunc(PropertyIds::check, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_check, OpCodeAsmJs::Simd128\_Ld\_F4 /\*no dynamic checks\*/, AsmJsRetType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_splat] = SIMDFunc(PropertyIds::splat, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_splat, OpCodeAsmJs::Simd128\_Splat\_F4, AsmJsRetType::Float32x4, AsmJsType::FloatishDoubleLit));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_fromFloat64x2] = SIMDFunc(PropertyIds::fromFloat64x2, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_fromFloat64x2, OpCodeAsmJs::Simd128\_FromFloat64x2\_F4, AsmJsRetType::Float32x4, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_fromFloat64x2Bits] = SIMDFunc(PropertyIds::fromFloat64x2Bits, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_fromFloat64x2Bits, OpCodeAsmJs::Simd128\_FromFloat64x2Bits\_F4, AsmJsRetType::Float32x4, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_fromInt32x4] = SIMDFunc(PropertyIds::fromInt32x4, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_fromInt32x4, OpCodeAsmJs::Simd128\_FromInt32x4\_F4, AsmJsRetType::Float32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_fromInt32x4Bits] = SIMDFunc(PropertyIds::fromInt32x4Bits, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_fromInt32x4Bits, OpCodeAsmJs::Simd128\_FromInt32x4Bits\_F4, AsmJsRetType::Float32x4, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_abs] = SIMDFunc(PropertyIds::abs, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_abs, OpCodeAsmJs::Simd128\_Abs\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_neg] = SIMDFunc(PropertyIds::neg, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_neg, OpCodeAsmJs::Simd128\_Neg\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_add] = SIMDFunc(PropertyIds::add, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_add, OpCodeAsmJs::Simd128\_Add\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_sub] = SIMDFunc(PropertyIds::sub, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_sub, OpCodeAsmJs::Simd128\_Sub\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_mul] = SIMDFunc(PropertyIds::mul, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_mul, OpCodeAsmJs::Simd128\_Mul\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_div] = SIMDFunc(PropertyIds::div, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_div, OpCodeAsmJs::Simd128\_Div\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_clamp] = SIMDFunc(PropertyIds::clamp, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float32x4\_clamp, OpCodeAsmJs::Simd128\_Clamp\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_min] = SIMDFunc(PropertyIds::min, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_min, OpCodeAsmJs::Simd128\_Min\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_max] = SIMDFunc(PropertyIds::max, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_max, OpCodeAsmJs::Simd128\_Max\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_reciprocal] = SIMDFunc(PropertyIds::reciprocal, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_reciprocal, OpCodeAsmJs::Simd128\_Rcp\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_reciprocalSqrt] = SIMDFunc(PropertyIds::reciprocalSqrt, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_reciprocalSqrt, OpCodeAsmJs::Simd128\_RcpSqrt\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_sqrt] = SIMDFunc(PropertyIds::sqrt, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_sqrt, OpCodeAsmJs::Simd128\_Sqrt\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_swizzle] = SIMDFunc(PropertyIds::swizzle, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 5, AsmJsSIMDBuiltin\_float32x4\_swizzle, OpCodeAsmJs::Simd128\_Swizzle\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_shuffle] = SIMDFunc(PropertyIds::shuffle, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 6, AsmJsSIMDBuiltin\_float32x4\_shuffle, OpCodeAsmJs::Simd128\_Shuffle\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_extractLane] = SIMDFunc(PropertyIds::extractLane, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_extractLane, OpCodeAsmJs::Simd128\_ExtractLane\_F4, AsmJsRetType::Float, AsmJsType::Float32x4, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_replaceLane] = SIMDFunc(PropertyIds::replaceLane, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float32x4\_replaceLane, OpCodeAsmJs::Simd128\_ReplaceLane\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Int, AsmJsType::FloatishDoubleLit));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_lessThan] = SIMDFunc(PropertyIds::lessThan, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_lessThan, OpCodeAsmJs::Simd128\_Lt\_F4, AsmJsRetType::Int32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_lessThanOrEqual] = SIMDFunc(PropertyIds::lessThanOrEqual, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_lessThanOrEqual, OpCodeAsmJs::Simd128\_LtEq\_F4, AsmJsRetType::Int32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_equal] = SIMDFunc(PropertyIds::equal, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_equal, OpCodeAsmJs::Simd128\_Eq\_F4, AsmJsRetType::Int32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_notEqual] = SIMDFunc(PropertyIds::notEqual, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_notEqual, OpCodeAsmJs::Simd128\_Neq\_F4, AsmJsRetType::Int32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_greaterThan] = SIMDFunc(PropertyIds::greaterThan, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_greaterThan, OpCodeAsmJs::Simd128\_Gt\_F4, AsmJsRetType::Int32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_greaterThanOrEqual]= SIMDFunc(PropertyIds::greaterThanOrEqual, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_greaterThanOrEqual, OpCodeAsmJs::Simd128\_GtEq\_F4, AsmJsRetType::Int32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_select] = SIMDFunc(PropertyIds::select, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float32x4\_select, OpCodeAsmJs::Simd128\_Select\_F4, AsmJsRetType::Float32x4, AsmJsType::Int32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_and] = SIMDFunc(PropertyIds::and, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_and, OpCodeAsmJs::Simd128\_And\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_or] = SIMDFunc(PropertyIds::or, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_or, OpCodeAsmJs::Simd128\_Or\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_xor] = SIMDFunc(PropertyIds::xor, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_xor, OpCodeAsmJs::Simd128\_Xor\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_not] = SIMDFunc(PropertyIds::not, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float32x4\_not, OpCodeAsmJs::Simd128\_Not\_F4, AsmJsRetType::Float32x4, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_load] = SIMDFunc(PropertyIds::load, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_load, OpCodeAsmJs::Simd128\_LdArr\_F4, AsmJsRetType::Float32x4, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_load1] = SIMDFunc(PropertyIds::load1, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_load1, OpCodeAsmJs::Simd128\_LdArr\_F4, AsmJsRetType::Float32x4, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_load2] = SIMDFunc(PropertyIds::load2, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_load2, OpCodeAsmJs::Simd128\_LdArr\_F4, AsmJsRetType::Float32x4, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_load3] = SIMDFunc(PropertyIds::load3, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float32x4\_load3, OpCodeAsmJs::Simd128\_LdArr\_F4, AsmJsRetType::Float32x4, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_store] = SIMDFunc(PropertyIds::store, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float32x4\_store, OpCodeAsmJs::Simd128\_StArr\_F4, AsmJsRetType::Float32x4, AsmJsType::Void, AsmJsType::Int, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_store1] = SIMDFunc(PropertyIds::store1, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float32x4\_store1, OpCodeAsmJs::Simd128\_StArr\_F4, AsmJsRetType::Float32x4, AsmJsType::Void, AsmJsType::Int, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_store2] = SIMDFunc(PropertyIds::store2, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float32x4\_store2, OpCodeAsmJs::Simd128\_StArr\_F4, AsmJsRetType::Float32x4, AsmJsType::Void, AsmJsType::Int, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float32x4\_store3] = SIMDFunc(PropertyIds::store3, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float32x4\_store3, OpCodeAsmJs::Simd128\_StArr\_F4, AsmJsRetType::Float32x4, AsmJsType::Void, AsmJsType::Int, AsmJsType::Float32x4));

/\* Float64x2 builtins\*/

//-------------------

simdFunctions[AsmJsSIMDBuiltin\_Float64x2] = SIMDFunc(PropertyIds::Float64x2, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_Float64x2, OpCodeAsmJs::Simd128\_DoublesToD2, AsmJsRetType::Float64x2, AsmJsType::MaybeDouble, AsmJsType::MaybeDouble));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_check] = SIMDFunc(PropertyIds::check, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_check, OpCodeAsmJs::Simd128\_Ld\_D2 /\*no dynamic checks\*/, AsmJsRetType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_splat] = SIMDFunc(PropertyIds::splat, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_splat, OpCodeAsmJs::Simd128\_Splat\_D2, AsmJsRetType::Float64x2, AsmJsType::Double));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_fromFloat32x4] = SIMDFunc(PropertyIds::fromFloat32x4, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_fromFloat32x4, OpCodeAsmJs::Simd128\_FromFloat32x4\_D2, AsmJsRetType::Float64x2, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_fromFloat32x4Bits] = SIMDFunc(PropertyIds::fromFloat32x4Bits, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_fromFloat32x4Bits, OpCodeAsmJs::Simd128\_FromFloat32x4Bits\_D2, AsmJsRetType::Float64x2, AsmJsType::Float32x4));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_fromInt32x4] = SIMDFunc(PropertyIds::fromInt32x4, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_fromInt32x4, OpCodeAsmJs::Simd128\_FromInt32x4\_D2, AsmJsRetType::Float64x2, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_fromInt32x4Bits] = SIMDFunc(PropertyIds::fromInt32x4Bits, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_fromInt32x4Bits, OpCodeAsmJs::Simd128\_FromInt32x4Bits\_D2, AsmJsRetType::Float64x2, AsmJsType::Int32x4));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_abs] = SIMDFunc(PropertyIds::abs, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_abs, OpCodeAsmJs::Simd128\_Abs\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_neg] = SIMDFunc(PropertyIds::neg, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_neg, OpCodeAsmJs::Simd128\_Neg\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_add] = SIMDFunc(PropertyIds::add, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_add, OpCodeAsmJs::Simd128\_Add\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_sub] = SIMDFunc(PropertyIds::sub, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_sub, OpCodeAsmJs::Simd128\_Sub\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_mul] = SIMDFunc(PropertyIds::mul, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_mul, OpCodeAsmJs::Simd128\_Mul\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_div] = SIMDFunc(PropertyIds::div, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_div, OpCodeAsmJs::Simd128\_Div\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_clamp] = SIMDFunc(PropertyIds::clamp, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float64x2\_clamp, OpCodeAsmJs::Simd128\_Clamp\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_min] = SIMDFunc(PropertyIds::min, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_min, OpCodeAsmJs::Simd128\_Min\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_max] = SIMDFunc(PropertyIds::max, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_max, OpCodeAsmJs::Simd128\_Max\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_reciprocal] = SIMDFunc(PropertyIds::reciprocal, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_reciprocal, OpCodeAsmJs::Simd128\_Rcp\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_reciprocalSqrt] = SIMDFunc(PropertyIds::reciprocalSqrt, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_reciprocalSqrt, OpCodeAsmJs::Simd128\_RcpSqrt\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_sqrt] = SIMDFunc(PropertyIds::sqrt, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 1, AsmJsSIMDBuiltin\_float64x2\_sqrt, OpCodeAsmJs::Simd128\_Sqrt\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_swizzle] = SIMDFunc(PropertyIds::swizzle, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float64x2\_swizzle, OpCodeAsmJs::Simd128\_Swizzle\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Int, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_shuffle] = SIMDFunc(PropertyIds::shuffle, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 4, AsmJsSIMDBuiltin\_float64x2\_shuffle, OpCodeAsmJs::Simd128\_Shuffle\_D2, AsmJsRetType::Float64x2, AsmJsType::Float64x2, AsmJsType::Float64x2, AsmJsType::Int, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_lessThan] = SIMDFunc(PropertyIds::lessThan, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_lessThan, OpCodeAsmJs::Simd128\_Lt\_D2, AsmJsRetType::Int32x4, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_lessThanOrEqual] = SIMDFunc(PropertyIds::lessThanOrEqual, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_lessThanOrEqual, OpCodeAsmJs::Simd128\_LtEq\_D2, AsmJsRetType::Int32x4, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_equal] = SIMDFunc(PropertyIds::equal, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_equal, OpCodeAsmJs::Simd128\_Eq\_D2, AsmJsRetType::Int32x4, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_notEqual] = SIMDFunc(PropertyIds::notEqual, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_notEqual, OpCodeAsmJs::Simd128\_Neq\_D2, AsmJsRetType::Int32x4, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_greaterThan] = SIMDFunc(PropertyIds::greaterThan, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_greaterThan, OpCodeAsmJs::Simd128\_Gt\_D2, AsmJsRetType::Int32x4, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_greaterThanOrEqual]= SIMDFunc(PropertyIds::greaterThanOrEqual, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_greaterThanOrEqual, OpCodeAsmJs::Simd128\_GtEq\_D2, AsmJsRetType::Int32x4, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_select] = SIMDFunc(PropertyIds::select, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float64x2\_select, OpCodeAsmJs::Simd128\_Select\_D2, AsmJsRetType::Float64x2, AsmJsType::Int32x4, AsmJsType::Float64x2, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_load] = SIMDFunc(PropertyIds::load, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_load, OpCodeAsmJs::Simd128\_LdArr\_D2, AsmJsRetType::Float64x2, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_load1] = SIMDFunc(PropertyIds::load1, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 2, AsmJsSIMDBuiltin\_float64x2\_load1, OpCodeAsmJs::Simd128\_LdArr\_D2, AsmJsRetType::Float64x2, AsmJsType::Void, AsmJsType::Int));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_store] = SIMDFunc(PropertyIds::store, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float64x2\_store, OpCodeAsmJs::Simd128\_StArr\_D2, AsmJsRetType::Float64x2, AsmJsType::Void, AsmJsType::Int, AsmJsType::Float64x2));

simdFunctions[AsmJsSIMDBuiltin\_float64x2\_store1] = SIMDFunc(PropertyIds::store1, Anew(&mAllocator, AsmJsSIMDFunction, nullptr, &mAllocator, 3, AsmJsSIMDBuiltin\_float64x2\_store1, OpCodeAsmJs::Simd128\_StArr\_D2, AsmJsRetType::Float64x2, AsmJsType::Void, AsmJsType::Int, AsmJsType::Float64x2));

{

SIMDNameMap \*map = &mStdLibSIMDInt32x4Map;

for (int i = 0; i < AsmJsSIMDBuiltin\_COUNT; i++)

{

if (i == AsmJsSIMDBuiltin\_Float32x4)

{

map = &mStdLibSIMDFloat32x4Map;

}

if (i == AsmJsSIMDBuiltin\_Float64x2)

{

map = &mStdLibSIMDFloat64x2Map;

}

if (simdFunctions[i].id && simdFunctions[i].val)

{

if (!AddStandardLibrarySIMDNameInMap(simdFunctions[i].id, simdFunctions[i].val, map))

{

AsmJSCompiler::OutputError(GetScriptContext(), L"Cannot initialize SIMD library");

return false;

}

}

}

}

return true;

}

AsmJsModuleCompiler::AsmJsModuleCompiler( ExclusiveContext \*cx, AsmJSParser &parser ) :

mCx( cx )

, mCurrentParserNode( parser )

, mAllocator( L"Asmjs", cx->scriptContext->GetThreadContext()->GetPageAllocator(), Throw::OutOfMemory )

, mModuleFunctionName( nullptr )

, mStandardLibraryMathNames(&mAllocator)

, mStandardLibraryArrayNames(&mAllocator)

, mFunctionArray( &mAllocator )

, mModuleEnvironment( &mAllocator )

, mFunctionTableArray( &mAllocator )

, mInitialised(false)

, mIntVarSpace( )

, mDoubleVarSpace( )

, mExports(&mAllocator)

, mExportFuncIndex(Js::Constants::NoRegister)

, mVarImportCount(0)

, mVarCount(0)

, mFuncPtrTableCount(0)

, mCompileTime()

, mCompileTimeLastTick(GetTick())

, mMaxAstSize(0)

, mArrayViews(&mAllocator)

, mUsesChangeHeap(false)

, mUsesHeapBuffer(false)

, mMaxHeapAccess(0)

#if DBG

, mStdLibArgNameInit(false)

, mForeignArgNameInit(false)

, mBufferArgNameInit(false)

#endif

, mStdLibSIMDInt32x4Map(&mAllocator)

, mStdLibSIMDFloat32x4Map(&mAllocator)

, mStdLibSIMDFloat64x2Map(&mAllocator)

{

InitModuleNode( parser );

}

bool AsmJsModuleCompiler::AddStandardLibraryMathName( PropertyId id, const double\* cstAddr, AsmJSMathBuiltinFunction mathLibFunctionName )

{

// make sure this name is unique

if( mStandardLibraryMathNames.ContainsKey( id ) )

{

return false;

}

MathBuiltin mathBuiltin(mathLibFunctionName, cstAddr);

int addResult = mStandardLibraryMathNames.AddNew( id, mathBuiltin );

if( addResult == -1 )

{

// Error adding the function

return false;

}

return true;

}

bool AsmJsModuleCompiler::AddStandardLibraryMathName(PropertyId id, AsmJsMathFunction\* func, AsmJSMathBuiltinFunction mathLibFunctionName)

{

// make sure this name is unique

if( mStandardLibraryMathNames.ContainsKey( id ) )

{

return false;

}

MathBuiltin mathBuiltin(mathLibFunctionName, func);

int addResult = mStandardLibraryMathNames.AddNew( id, mathBuiltin );

if( addResult == -1 )

{

// Error adding the function

return false;

}

return true;

}

bool AsmJsModuleCompiler::AddStandardLibraryArrayName(PropertyId id, AsmJsTypedArrayFunction\* func, AsmJSTypedArrayBuiltinFunction arrayLibFunctionName)

{

// make sure this name is unique

if (mStandardLibraryArrayNames.ContainsKey(id))

{

return false;

}

TypedArrayBuiltin arrayBuiltin(arrayLibFunctionName, func);

int addResult = mStandardLibraryArrayNames.AddNew(id, arrayBuiltin);

if (addResult == -1)

{

// Error adding the function

return false;

}

return true;

}

Parser \* AsmJsModuleCompiler::GetParser() const

{

return mCx->byteCodeGenerator->GetParser();

}

ByteCodeGenerator\* AsmJsModuleCompiler::GetByteCodeGenerator() const

{

return mCx->byteCodeGenerator;

}

ScriptContext \* AsmJsModuleCompiler::GetScriptContext() const

{

return mCx->scriptContext;

}

AsmJsSymbol\* AsmJsModuleCompiler::LookupIdentifier( PropertyName name, AsmJsFunc\* func /\*= nullptr \*/, AsmJsLookupSource::Source\* lookupSource /\*= nullptr\*/ )

{

AsmJsSymbol\* lookupResult = nullptr;

if (name)

{

if (func)

{

lookupResult = func->LookupIdentifier(name, lookupSource);

if (lookupResult)

{

return lookupResult;

}

}

lookupResult = mModuleEnvironment.LookupWithKey(name->GetPropertyId(), nullptr);

if (lookupSource)

{

\*lookupSource = AsmJsLookupSource::AsmJsModule;

}

}

return lookupResult;

}

bool AsmJsModuleCompiler::DefineIdentifier( PropertyName name, AsmJsSymbol\* symbol )

{

Assert( symbol );

if( symbol )

{

// make sure this identifier is unique

if(!LookupIdentifier( name ))

{

int addResult = mModuleEnvironment.AddNew(name->GetPropertyId(), symbol);

return addResult != -1;

}

}

return false;

}

bool AsmJsModuleCompiler::AddNumericVar( PropertyName name, ParseNode\* pnode, bool isFloat, bool isMutable /\*= true\*/ )

{

Assert(ParserWrapper::IsNumericLiteral(pnode) || (isFloat && ParserWrapper::IsFroundNumericLiteral(pnode)));

AsmJsVar\* var = Anew( &mAllocator, AsmJsVar, name, isMutable );

if( !var )

{

return false;

}

if( !DefineIdentifier( name, var ) )

{

return false;

}

++mVarCount;

if (isFloat)

{

var->SetVarType(AsmJsVarType::Float);

var->SetLocation(mFloatVarSpace.AcquireRegister());

if (pnode->nop == knopInt)

{

var->SetConstInitialiser((float)pnode->sxInt.lw);

}

else if (ParserWrapper::IsNegativeZero(pnode))

{

var->SetConstInitialiser(-0.0f);

}

else

{

var->SetConstInitialiser((float)pnode->sxFlt.dbl);

}

}

else if (pnode->nop == knopInt)

{

var->SetVarType(AsmJsVarType::Int);

var->SetLocation(mIntVarSpace.AcquireRegister());

var->SetConstInitialiser(pnode->sxInt.lw);

}

else

{

if (ParserWrapper::IsMinInt(pnode))

{

var->SetVarType(AsmJsVarType::Int);

var->SetLocation(mIntVarSpace.AcquireRegister());

var->SetConstInitialiser(MININT);

}

else if (ParserWrapper::IsUnsigned(pnode))

{

var->SetVarType(AsmJsVarType::Int);

var->SetLocation(mIntVarSpace.AcquireRegister());

var->SetConstInitialiser((int)((uint32)pnode->sxFlt.dbl));

}

else if (pnode->sxFlt.maybeInt)

{

// this means there was an int literal not in range [-2^31,3^32)

return false;

}

else

{

var->SetVarType(AsmJsVarType::Double);

var->SetLocation(mDoubleVarSpace.AcquireRegister());

var->SetConstInitialiser(pnode->sxFlt.dbl);

}

}

return true;

}

bool AsmJsModuleCompiler::AddGlobalVarImport( PropertyName name, PropertyName field, AsmJSCoercion coercion )

{

AsmJsConstantImport\* var = Anew( &mAllocator, AsmJsConstantImport, name, field );

if( !var )

{

return false;

}

if( !DefineIdentifier( name, var ) )

{

return false;

}

++mVarImportCount;

switch( coercion )

{

case Js::AsmJS\_ToInt32:

var->SetVarType( AsmJsVarType::Int );

var->SetLocation( mIntVarSpace.AcquireRegister() );

break;

case Js::AsmJS\_ToNumber:

var->SetVarType( AsmJsVarType::Double );

var->SetLocation( mDoubleVarSpace.AcquireRegister() );

break;

case Js::AsmJS\_FRound:

var->SetVarType( AsmJsVarType::Float );

var->SetLocation(mFloatVarSpace.AcquireRegister());

break;

case Js::AsmJS\_Int32x4:

if (IsSimdjsEnabled())

{

var->SetVarType(AsmJsVarType::Int32x4);

var->SetLocation(mSimdVarSpace.AcquireRegister());

break;

}

Assert(UNREACHED);

case AsmJS\_Float32x4:

if (IsSimdjsEnabled())

{

var->SetVarType(AsmJsVarType::Float32x4);

var->SetLocation(mSimdVarSpace.AcquireRegister());

break;

}

Assert(UNREACHED);

case AsmJS\_Float64x2:

if (IsSimdjsEnabled())

{

var->SetVarType(AsmJsVarType::Float64x2);

var->SetLocation(mSimdVarSpace.AcquireRegister());

break;

}

Assert(UNREACHED);

default:

break;

}

return true;

}

bool AsmJsModuleCompiler::AddModuleFunctionImport( PropertyName name, PropertyName field )

{

AsmJsImportFunction\* var = Anew( &mAllocator, AsmJsImportFunction, name, field, &mAllocator );

if( !var )

{

return false;

}

if( !DefineIdentifier( name, var ) )

{

return false;

}

var->SetFunctionIndex( mImportFunctions.AcquireRegister() );

return true;

}

bool AsmJsModuleCompiler::AddNumericConst( PropertyName name, const double\* cst )

{

AsmJsMathConst\* var = Anew( &mAllocator, AsmJsMathConst, name, cst );

if( !var )

{

return false;

}

if( !DefineIdentifier( name, var ) )

{

return false;

}

return true;

}

bool AsmJsModuleCompiler::AddArrayView( PropertyName name, ArrayBufferView::ViewType type )

{

AsmJsArrayView\* view = Anew( &mAllocator, AsmJsArrayView, name, type );

if( !view )

{

return false;

}

if( !DefineIdentifier( name, view ) )

{

return false;

}

mArrayViews.Enqueue(view);

return true;

}

bool AsmJsModuleCompiler::AddFunctionTable( PropertyName name, const int size )

{

GetByteCodeGenerator()->AssignPropertyId(name);

AsmJsFunctionTable\* funcTable = Anew( &mAllocator, AsmJsFunctionTable, name, &mAllocator );

if( !funcTable )

{

return false;

}

if( !DefineIdentifier( name, funcTable ) )

{

return false;

}

funcTable->SetSize( size );

int pos = mFunctionTableArray.Add( funcTable );

funcTable->SetFunctionIndex( pos );

return true;

}

bool AsmJsModuleCompiler::AddExport( PropertyName name, RegSlot location )

{

AsmJsModuleExport ex;

ex.id = name->GetPropertyId();

ex.location = location;

// return is < 0 if count overflowed 31bits

return mExports.Add( ex ) >= 0;

}

bool AsmJsModuleCompiler::SetExportFunc( AsmJsFunc\* func )

{

Assert( mExports.Count() == 0 && func);

mExportFuncIndex = func->GetFunctionIndex();

return mExports.Count() == 0 && (uint32)mExportFuncIndex < (uint32)mFunctionArray.Count();

}

AsmJsFunctionDeclaration\* AsmJsModuleCompiler::LookupFunction( PropertyName name )

{

if (name)

{

AsmJsSymbol\* sym = LookupIdentifier(name);

if (sym)

{

switch (sym->GetSymbolType())

{

case AsmJsSymbol::SIMDBuiltinFunction:

case AsmJsSymbol::MathBuiltinFunction:

case AsmJsSymbol::ModuleFunction:

case AsmJsSymbol::ImportFunction:

case AsmJsSymbol::FuncPtrTable:

return sym->Cast<AsmJsFunctionDeclaration>();

default:

break;

}

}

}

return nullptr;

}

bool AsmJsModuleCompiler::AreAllFuncTableDefined()

{

const int size = mFunctionTableArray.Count();

for (int i = 0; i < size ; i++)

{

AsmJsFunctionTable\* funcTable = mFunctionTableArray.Item( i );

if( !funcTable->IsDefined() )

{

AsmJSCompiler::OutputError(GetScriptContext(), L"Function table %s was used in a function but does not appear in the module", funcTable->GetName()->Psz());

return false;

}

}

return true;

}

void AsmJsModuleCompiler::UpdateMaxHeapAccess(uint index)

{

if (mMaxHeapAccess < index)

{

mMaxHeapAccess = index;

}

}

void AsmJsModuleCompiler::InitMemoryOffsets()

{

mModuleMemory.mArrayBufferOffset = AsmJsModuleMemory::MemoryTableBeginOffset;

mModuleMemory.mStdLibOffset = mModuleMemory.mArrayBufferOffset + 1;

mModuleMemory.mDoubleOffset = mModuleMemory.mStdLibOffset + 1;

mModuleMemory.mFuncOffset = mModuleMemory.mDoubleOffset + (mDoubleVarSpace.GetTotalVarCount() \* DOUBLE\_SLOTS\_SPACE);

mModuleMemory.mFFIOffset = mModuleMemory.mFuncOffset + mFunctionArray.Count();

mModuleMemory.mFuncPtrOffset = mModuleMemory.mFFIOffset + mImportFunctions.GetTotalVarCount();

mModuleMemory.mFloatOffset = mModuleMemory.mFuncPtrOffset + GetFuncPtrTableCount();

mModuleMemory.mIntOffset = mModuleMemory.mFloatOffset + (int32)(mFloatVarSpace.GetTotalVarCount() \* FLOAT\_SLOTS\_SPACE + 0.5);

mModuleMemory.mMemorySize = mModuleMemory.mIntOffset + (int32)(mIntVarSpace.GetTotalVarCount() \* INT\_SLOTS\_SPACE + 0.5);

if (IsSimdjsEnabled())

{

mModuleMemory.mSimdOffset = (int) ::ceil(mModuleMemory.mMemorySize / SIMD\_SLOTS\_SPACE);

if (mSimdVarSpace.GetTotalVarCount())

{

mModuleMemory.mMemorySize = (int)((mModuleMemory.mSimdOffset + mSimdVarSpace.GetTotalVarCount()) \* SIMD\_SLOTS\_SPACE);

// no alignment

// mModuleMemory.mMemorySize += (int)SIMD\_SLOTS\_SPACE;

}

}

}

void AsmJsModuleCompiler::AccumulateCompileTime()

{

Js::TickDelta td;

AsmJsCompileTime curTime = GetTick();

td = curTime - mCompileTimeLastTick;

mCompileTime = mCompileTime+td;

mCompileTimeLastTick = curTime;

}

void AsmJsModuleCompiler::AccumulateCompileTime(AsmJsCompilation::Phases phase)

{

Js::TickDelta td;

AsmJsCompileTime curTime = GetTick();

td = curTime - mCompileTimeLastTick;

mCompileTime = mCompileTime+td;

mCompileTimeLastTick = curTime;

mPhaseCompileTime[phase] = mPhaseCompileTime[phase] + td;

}

Js::AsmJsCompileTime AsmJsModuleCompiler::GetTick()

{

return Js::Tick::Now();

}

uint64 AsmJsModuleCompiler::GetCompileTime() const

{

return mCompileTime.ToMicroseconds();

}

static const wchar\_t\* AsmPhaseNames[AsmJsCompilation::Phases\_COUNT] = {

L"Module",

L"ByteCode",

L"TemplateJIT",

};

void AsmJsModuleCompiler::PrintCompileTrace() const

{

// for testtrace, don't print time so that it can be used for baselines

if (PHASE\_TESTTRACE1(AsmjsPhase))

{

AsmJSCompiler::OutputMessage(GetScriptContext(), DEIT\_ASMJS\_SUCCEEDED, L"Successfully compiled asm.js code");

}

else

{

uint64 us = GetCompileTime();

uint64 ms = us / 1000;

us = us % 1000;

AsmJSCompiler::OutputMessage(GetScriptContext(), DEIT\_ASMJS\_SUCCEEDED, L"Successfully compiled asm.js code (total compilation time %llu.%llums)", ms, us);

}

if (PHASE\_TRACE1(AsmjsPhase))

{

for (int i = 0; i < AsmJsCompilation::Phases\_COUNT; i++)

{

uint64 us = mPhaseCompileTime[i].ToMicroseconds();

uint64 ms = us / 1000;

us = us % 1000;

Output::Print(L"%20s : %llu.%llums\n", AsmPhaseNames[i], ms, us);

}

Output::Flush();

}

}

BVStatic<ASMMATH\_BUILTIN\_SIZE> AsmJsModuleCompiler::GetAsmMathBuiltinUsedBV()

{

return mAsmMathBuiltinUsedBV;

}

BVStatic<ASMARRAY\_BUILTIN\_SIZE> AsmJsModuleCompiler::GetAsmArrayBuiltinUsedBV()

{

return mAsmArrayBuiltinUsedBV;

}

void AsmJsModuleInfo::SetFunctionCount( int val )

{

Assert( mFunctions == nullptr );

mFunctionCount = val;

mFunctions = RecyclerNewArray( mRecycler, ModuleFunction, val );

}

void AsmJsModuleInfo::SetFunctionTableCount( int val )

{

Assert( mFunctionTables == nullptr );

mFunctionTableCount = val;

mFunctionTables = RecyclerNewArray( mRecycler, ModuleFunctionTable, val );

}

void AsmJsModuleInfo::SetFunctionImportCount( int val )

{

Assert( mFunctionImports == nullptr );

mFunctionImportCount = val;

mFunctionImports = RecyclerNewArray( mRecycler, ModuleFunctionImport, val );

}

void AsmJsModuleInfo::SetVarCount( int val )

{

Assert( mVars == nullptr );

mVarCount = val;

mVars = RecyclerNewArray( mRecycler, ModuleVar, val );

}

void AsmJsModuleInfo::SetVarImportCount( int val )

{

Assert( mVarImports == nullptr );

mVarImportCount = val;

mVarImports = RecyclerNewArray( mRecycler, ModuleVarImport, val );

}

void AsmJsModuleInfo::SetExportsCount( int count )

{

if( count )

{

mExports = RecyclerNewPlus( mRecycler, count \* sizeof( PropertyId ), PropertyIdArray, count );

mExportsFunctionLocation = RecyclerNewArray( mRecycler, RegSlot, count );

}

mExportsCount = count;

}

void AsmJsModuleInfo::InitializeSlotMap(int val)

{

Assert(mSlotMap == nullptr);

mSlotsCount = val;

mSlotMap = RecyclerNew(mRecycler, AsmJsSlotMap, mRecycler);

}

void AsmJsModuleInfo::SetFunctionTableSize( int index, uint size )

{

Assert( mFunctionTables != nullptr );

Assert( index < mFunctionTableCount );

ModuleFunctionTable& table = mFunctionTables[index];

table.size = size;

table.moduleFunctionIndex = RecyclerNewArray( mRecycler, RegSlot, size );

}

void AsmJsModuleInfo::EnsureHeapAttached(ScriptFunction \* func)

{

FrameDisplay\* frame = func->GetEnvironment();

ArrayBuffer\* moduleArrayBuffer = \*(ArrayBuffer\*\*)((Var\*)frame->GetItem(0) + AsmJsModuleMemory::MemoryTableBeginOffset);

if (moduleArrayBuffer && moduleArrayBuffer->IsDetached())

{

Throw::OutOfMemory();

}

}

void \* AsmJsModuleInfo::ConvertFrameForJavascript(void \* asmMemory, ScriptFunction\* func)

{

FunctionBody \* body = func->GetFunctionBody();

AsmJsFunctionInfo \* asmFuncInfo = body->GetAsmJsFunctionInfo();

FunctionBody \* moduleBody = asmFuncInfo->GetModuleFunctionBody();

AsmJsModuleInfo \* asmModuleInfo = moduleBody->GetAsmJsModuleInfo();

Assert(asmModuleInfo);

ScriptContext \* scriptContext = func->GetScriptContext();

// AsmJsModuleEnvironment is all laid out here

Var \* asmJsEnvironment = static\_cast<Var\*>(func->GetEnvironment()->GetItem(0));

Var \* asmBufferPtr = asmJsEnvironment + asmModuleInfo->GetModuleMemory().mArrayBufferOffset;

ArrayBuffer \* asmBuffer = \*asmBufferPtr ? ArrayBuffer::FromVar(\*asmBufferPtr) : nullptr;

Var stdLibObj = \*(asmJsEnvironment + asmModuleInfo->GetModuleMemory().mStdLibOffset);

Var asmMathObject = stdLibObj ? JavascriptOperators::OP\_GetProperty(stdLibObj, PropertyIds::Math, scriptContext) : nullptr;

Var \* asmFFIs = asmJsEnvironment + asmModuleInfo->GetModuleMemory().mFFIOffset;

Var \* asmFuncs = asmJsEnvironment + asmModuleInfo->GetModuleMemory().mFuncOffset;

Var \*\* asmFuncPtrs = reinterpret\_cast<Var\*\*>(asmJsEnvironment + asmModuleInfo->GetModuleMemory().mFuncPtrOffset);

double \* asmDoubleVars = reinterpret\_cast<double\*>(asmJsEnvironment + asmModuleInfo->GetModuleMemory().mDoubleOffset);

int \* asmIntVars = reinterpret\_cast<int\*>(asmJsEnvironment + asmModuleInfo->GetModuleMemory().mIntOffset);

float \* asmFloatVars = reinterpret\_cast<float\*>(asmJsEnvironment + asmModuleInfo->GetModuleMemory().mFloatOffset);

AsmJsSIMDValue \* asmSIMDVars = reinterpret\_cast<AsmJsSIMDValue\*>(asmJsEnvironment + asmModuleInfo->GetModuleMemory().mSimdOffset);

#if DEBUG

Var \* slotArray = RecyclerNewArrayZ(scriptContext->GetRecycler(), Var, moduleBody->scopeSlotArraySize + ScopeSlots::FirstSlotIndex);

#else

Var \* slotArray = RecyclerNewArray(scriptContext->GetRecycler(), Var, moduleBody->scopeSlotArraySize + ScopeSlots::FirstSlotIndex);

#endif

ScopeSlots scopeSlots(slotArray);

scopeSlots.SetCount(moduleBody->scopeSlotArraySize);

scopeSlots.SetScopeMetadata(moduleBody);

auto asmSlotMap = asmModuleInfo->GetAsmJsSlotMap();

Assert((uint)asmModuleInfo->GetSlotsCount() == moduleBody->scopeSlotArraySize);

Js::ActivationObject\* activeScopeObject = nullptr;

if (moduleBody->GetObjectRegister() != 0)

{

activeScopeObject = static\_cast<ActivationObject\*>(scriptContext->GetLibrary()->CreateActivationObject());

}

PropertyId\* propertyIdArray = moduleBody->GetPropertyIdsForScopeSlotArray();

for (int i = 0; i < asmModuleInfo->GetSlotsCount(); ++i)

{

AsmJsSlot \* asmSlot;

bool found = asmSlotMap->TryGetValue(propertyIdArray[i], &asmSlot);

// we should have everything we need in the map

Assert(found);

Var value = nullptr;

switch (asmSlot->symType)

{

case AsmJsSymbol::ConstantImport:

case AsmJsSymbol::Variable:

{

switch (asmSlot->varType)

{

case AsmJsVarType::Double:

value = JavascriptNumber::New(asmDoubleVars[asmSlot->location], scriptContext);

break;

case AsmJsVarType::Float:

value = JavascriptNumber::New(asmFloatVars[asmSlot->location], scriptContext);

break;

case AsmJsVarType::Int:

value = JavascriptNumber::ToVar(asmIntVars[asmSlot->location], scriptContext);

break;

case AsmJsVarType::Float32x4:

value = JavascriptSIMDFloat32x4::New(&asmSIMDVars[asmSlot->location], scriptContext);

break;

case AsmJsVarType::Float64x2:

value = JavascriptSIMDFloat64x2::New(&asmSIMDVars[asmSlot->location], scriptContext);

break;

case AsmJsVarType::Int32x4:

value = JavascriptSIMDInt32x4::New(&asmSIMDVars[asmSlot->location], scriptContext);

break;

default:

Assume(UNREACHED);

}

break;

}

case AsmJsSymbol::ModuleArgument:

{

switch (asmSlot->argType)

{

case AsmJsModuleArg::ArgType::StdLib:

value = stdLibObj;

break;

case AsmJsModuleArg::ArgType::Import:

// we can't reference this inside functions (and don't hold onto it), but must set to something, so set it to be undefined

value = scriptContext->GetLibrary()->GetUndefined();

break;

case AsmJsModuleArg::ArgType::Heap:

value = asmBuffer;

break;

default:

Assume(UNREACHED);

}

break;

}

case AsmJsSymbol::ImportFunction:

value = asmFFIs[asmSlot->location];

break;

case AsmJsSymbol::FuncPtrTable:

value = JavascriptArray::OP\_NewScArrayWithElements(asmSlot->funcTableSize, asmFuncPtrs[asmSlot->location], scriptContext);

break;

case AsmJsSymbol::ModuleFunction:

value = asmFuncs[asmSlot->location];

break;

case AsmJsSymbol::MathConstant:

value = JavascriptNumber::New(asmSlot->mathConstVal, scriptContext);

break;

case AsmJsSymbol::ArrayView:

{

AnalysisAssert(asmBuffer);

#ifdef \_M\_X64

const bool isOptimizedBuffer = true;

#elif \_M\_IX86

const bool isOptimizedBuffer = false;

#else

Assert(UNREACHED);

const bool isOptimizedBuffer = false;

#endif

Assert(isOptimizedBuffer == asmBuffer->IsValidVirtualBufferLength(asmBuffer->GetByteLength()));

switch (asmSlot->viewType)

{

case ArrayBufferView::TYPE\_FLOAT32:

value = TypedArray<float, false, isOptimizedBuffer>::Create(asmBuffer, 0, asmBuffer->GetByteLength() >> 2, scriptContext->GetLibrary());

break;

case ArrayBufferView::TYPE\_FLOAT64:

value = TypedArray<double, false, isOptimizedBuffer>::Create(asmBuffer, 0, asmBuffer->GetByteLength() >> 3, scriptContext->GetLibrary());

break;

case ArrayBufferView::TYPE\_INT8:

value = TypedArray<int8, false, isOptimizedBuffer>::Create(asmBuffer, 0, asmBuffer->GetByteLength(), scriptContext->GetLibrary());

break;

case ArrayBufferView::TYPE\_INT16:

value = TypedArray<int16, false, isOptimizedBuffer>::Create(asmBuffer, 0, asmBuffer->GetByteLength() >> 1, scriptContext->GetLibrary());

break;

case ArrayBufferView::TYPE\_INT32:

value = TypedArray<int32, false, isOptimizedBuffer>::Create(asmBuffer, 0, asmBuffer->GetByteLength() >> 2, scriptContext->GetLibrary());

break;

case ArrayBufferView::TYPE\_UINT8:

value = TypedArray<uint8, false, isOptimizedBuffer>::Create(asmBuffer, 0, asmBuffer->GetByteLength(), scriptContext->GetLibrary());

break;

case ArrayBufferView::TYPE\_UINT16:

value = TypedArray<uint16, false, isOptimizedBuffer>::Create(asmBuffer, 0, asmBuffer->GetByteLength() >> 1, scriptContext->GetLibrary());

break;

case ArrayBufferView::TYPE\_UINT32:

value = TypedArray<uint32, false, isOptimizedBuffer>::Create(asmBuffer, 0, asmBuffer->GetByteLength() >> 2, scriptContext->GetLibrary());

break;

default:

Assume(UNREACHED);

}

break;

}

case AsmJsSymbol::MathBuiltinFunction:

{

switch (asmSlot->builtinMathFunc)

{

#define ASMJS\_MATH\_FUNC\_NAMES(name, propertyName) \

case AsmJSMathBuiltin\_##name: \

value = JavascriptOperators::OP\_GetProperty(asmMathObject, PropertyIds::##propertyName, scriptContext); \

break;

#include "AsmJsBuiltinNames.h"

default:

Assume(UNREACHED);

}

break;

}

case AsmJsSymbol::TypedArrayBuiltinFunction:

switch (asmSlot->builtinArrayFunc)

{

#define ASMJS\_ARRAY\_NAMES(name, propertyName) \

case AsmJSTypedArrayBuiltin\_##name: \

value = JavascriptOperators::OP\_GetProperty(stdLibObj, PropertyIds::##propertyName, scriptContext); \

break;

#include "AsmJsBuiltinNames.h"

default:

Assume(UNREACHED);

}

break;

case AsmJsSymbol::SIMDBuiltinFunction:

switch (asmSlot->builtinSIMDFunc)

{

#define ASMJS\_SIMD\_NAMES(name, propertyName) \

case AsmJsSIMDBuiltin\_##name: \

value = JavascriptOperators::OP\_GetProperty(stdLibObj, PropertyIds::##propertyName, scriptContext); \

break;

#include "AsmJsBuiltinNames.h"

default:

Assume(UNREACHED);

}

break;

default:

Assume(UNREACHED);

}

if (activeScopeObject != nullptr)

{

activeScopeObject->SetPropertyWithAttributes(

propertyIdArray[i],

value,

asmSlot->isConstVar ? PropertyConstDefaults : PropertyDynamicTypeDefaults,

nullptr);

}

else

{

// ensure we aren't multiply writing to a slot

Assert(scopeSlots.Get(i) == nullptr);

scopeSlots.Set(i, value);

}

}

if (activeScopeObject != nullptr)

{

return (void\*)activeScopeObject;

}

else

{

return (void\*)slotArray;

}

}

bool AsmJsModuleCompiler::LookupStdLibSIMDNameInMap(PropertyName name, AsmJsSIMDFunction \*\*simdFunc, SIMDNameMap\* map) const

{

return map->TryGetValue(name->GetPropertyId(), simdFunc);

}

bool AsmJsModuleCompiler::AddStandardLibrarySIMDNameInMap(PropertyId id, AsmJsSIMDFunction \*simdFunc, SIMDNameMap\* map)

{

//SimdBuiltin simdBuiltin(simdFunc->GetSimdBuiltInFunction(), simdFunc);

if (map->ContainsKey(id))

{

return nullptr;

}

return map->AddNew(id, simdFunc) == -1 ? false : true;

}

bool AsmJsModuleCompiler::LookupStdLibSIMDName(PropertyId baseId, PropertyName fieldName, AsmJsSIMDFunction \*\*simdFunc)

{

switch (baseId)

{

case PropertyIds::Int32x4:

return LookupStdLibSIMDNameInMap(fieldName, simdFunc, &mStdLibSIMDInt32x4Map);

case PropertyIds::Float32x4:

return LookupStdLibSIMDNameInMap(fieldName, simdFunc, &mStdLibSIMDFloat32x4Map);

case PropertyIds::Float64x2:

return LookupStdLibSIMDNameInMap(fieldName, simdFunc, &mStdLibSIMDFloat64x2Map);

default:

AssertMsg(false, "Invalid SIMD type");

return false;

}

}

bool AsmJsModuleCompiler::LookupStdLibSIMDName(AsmJsSIMDBuiltinFunction baseId, PropertyName fieldName, AsmJsSIMDFunction \*\*simdFunc)

{

switch (baseId)

{

case AsmJsSIMDBuiltin\_Int32x4:

return LookupStdLibSIMDNameInMap(fieldName, simdFunc, &mStdLibSIMDInt32x4Map);

case AsmJsSIMDBuiltin\_Float32x4:

return LookupStdLibSIMDNameInMap(fieldName, simdFunc, &mStdLibSIMDFloat32x4Map);

case AsmJsSIMDBuiltin\_Float64x2:

return LookupStdLibSIMDNameInMap(fieldName, simdFunc, &mStdLibSIMDFloat64x2Map);

default:

AssertMsg(false, "Invalid SIMD type");

return false;

}

}

AsmJsSIMDFunction\* AsmJsModuleCompiler::LookupSimdConstructor(PropertyName name)

{

AsmJsFunctionDeclaration \*func = LookupFunction(name);

if (func == nullptr || func->GetSymbolType() != AsmJsSymbol::SIMDBuiltinFunction)

{

return nullptr;

}

AsmJsSIMDFunction \*simdFunc = func->Cast<AsmJsSIMDFunction>();

if (simdFunc->IsConstructor())

{

return simdFunc;

}

return nullptr;

}

AsmJsSIMDFunction\* AsmJsModuleCompiler::LookupSimdTypeCheck(PropertyName name)

{

AsmJsFunctionDeclaration \*func = LookupFunction(name);

if (func == nullptr || func->GetSymbolType() != AsmJsSymbol::SIMDBuiltinFunction)

{

return nullptr;

}

AsmJsSIMDFunction \*simdFunc = func->Cast<AsmJsSIMDFunction>();

if (simdFunc->IsTypeCheck())

{

return simdFunc;

}

return nullptr;

}

AsmJsSIMDFunction\* AsmJsModuleCompiler::LookupSimdOperation(PropertyName name)

{

AsmJsFunctionDeclaration \*func = LookupFunction(name);

if (func == nullptr || func->GetSymbolType() != AsmJsSymbol::SIMDBuiltinFunction)

{

return nullptr;

}

AsmJsSIMDFunction \*simdFunc = func->Cast<AsmJsSIMDFunction>();

if (simdFunc->GetSimdBuiltInFunction() != AsmJsSIMDBuiltin\_Int32x4 &&

simdFunc->GetSimdBuiltInFunction() != AsmJsSIMDBuiltin\_Float32x4 &&

simdFunc->GetSimdBuiltInFunction() != AsmJsSIMDBuiltin\_Float64x2)

{

return simdFunc;

}

return nullptr;

}

bool AsmJsModuleCompiler::AddSimdValueVar(PropertyName name, ParseNode\* pnode, AsmJsSIMDFunction\* simdFunc)

{

AssertMsg(simdFunc->GetSymbolType() == AsmJsSymbol::SIMDBuiltinFunction, "Expecting SIMD builtin");

AssertMsg(simdFunc->IsConstructor(), "Expecting constructor function");

AsmJsSIMDValue value;

AsmJsVarType type = simdFunc->GetConstructorVarType();

// e.g. var g1 = f4(1.0, 2.0, 3.0, 4.0);

if (!ValidateSimdConstructor(pnode, simdFunc, value))

{

return false;

}

AsmJsVar\* var = Anew(&mAllocator, AsmJsVar, name);

if (!var || !DefineIdentifier(name, var))

{

return false;

}

++mVarCount;

var->SetVarType(type);

var->SetConstInitialiser(value);

// acquire register

var->SetLocation(mSimdVarSpace.AcquireRegister());

return true;

}

bool AsmJsModuleCompiler::ValidateSimdConstructor(ParseNode\* pnode, AsmJsSIMDFunction\* simdFunc, AsmJsSIMDValue& value)

{

Assert(pnode->nop == knopCall);

uint argCount = pnode->sxCall.argCount;

ParseNode\* argNode = pnode->sxCall.pnodeArgs;

ParseNode \*arg = argNode;

uint nop = 0;

AsmJsSIMDBuiltinFunction simdBuiltin = simdFunc->GetSimdBuiltInFunction();

if (!simdFunc->IsConstructor(argCount))

{

return Fail(pnode, L"Invalid SIMD constructor or wrong number of arguments.");

}

switch (simdBuiltin)

{

case AsmJsSIMDBuiltin\_Float64x2:

case AsmJsSIMDBuiltin\_Float32x4:

nop = (uint)knopFlt;

break;

case AsmJsSIMDBuiltin\_Int32x4:

nop = (uint)knopInt;

break;

default:

Assert(UNREACHED);

}

Assert(simdBuiltin == AsmJsSIMDBuiltin\_Float64x2 || simdBuiltin == AsmJsSIMDBuiltin\_Float32x4 || simdBuiltin == AsmJsSIMDBuiltin\_Int32x4);

if (simdFunc->GetArgCount() != argCount)

{

return Fail(pnode, L"Invalid number of arguments to SIMD constructor.");

}

for (uint i = 0; i < argCount; i++)

{

arg = argNode;

if (argNode->nop == knopList)

{

arg = ParserWrapper::GetBinaryLeft(argNode);

argNode = ParserWrapper::GetBinaryRight(argNode);

}

Assert(arg);

// store to SIMD Value

if (arg->nop == nop)

{

if (nop == (uint)knopInt)

{

value.i32[i] = arg->sxInt.lw;

}

else if (nop == (uint)knopFlt)

{

if (simdBuiltin == AsmJsSIMDBuiltin\_Float32x4)

{

value.f32[i] = (float)arg->sxFlt.dbl;

}

else // float64x2

{

value.f64[i] = arg->sxFlt.dbl;

}

}

}

else

{

return Fail(pnode, L"Invalid argument type to SIMD constructor.");

}

}

return true;

}

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

#define ASMMATH\_BUILTIN\_SIZE (32)

#define ASMARRAY\_BUILTIN\_SIZE (16)

#define ASMSIMD\_BUILTIN\_SIZE (128)

namespace Js {

// ModuleCompiler encapsulates the compilation of an entire asm.js module. Over

// the course of an ModuleCompiler object's lifetime, many FunctionCompiler

// objects will be created and destroyed in sequence, one for each function in

// the module.

//

// \*\*\* asm.js FFI calls \*\*\*

//

// asm.js allows calling out to non-asm.js via "FFI calls". The asm.js type

// system does not place any constraints on the FFI call. In particular:

// - an FFI call's target is not known or speculated at module-compile time;

// - a single external function can be called with different signatures.

//

// If performance didn't matter, all FFI calls could simply box their arguments

// and call js::Invoke. However, we'd like to be able to specialize FFI calls

// to be more efficient in several cases:

//

// - for calls to JS functions which have been JITed, we'd like to call

// directly into JIT code without going through C++.

//

// - for calls to certain builtins, we'd like to be call directly into the C++

// code for the builtin without going through the general call path.

//

// All of this requires dynamic specialization techniques which must happen

// after module compilation. To support this, at module-compilation time, each

// FFI call generates a call signature according to the system ABI, as if the

// callee was a C++ function taking/returning the same types as the caller was

// passing/expecting. The callee is loaded from a fixed offset in the global

// data array which allows the callee to change at runtime. Initially, the

// callee is stub which boxes its arguments and calls js::Invoke.

//

// To do this, we need to generate a callee stub for each pairing of FFI callee

// and signature. We call this pairing an "exit". For example, this code has

// two external functions and three exits:

//

// function f(global, imports) {

// "use asm";

// var foo = imports.foo;

// var bar = imports.bar;

// function g() {

// foo(1); // Exit #1: (int) -> void

// foo(1.5); // Exit #2: (double) -> void

// bar(1)|0; // Exit #3: (int) -> int

// bar(2)|0; // Exit #3: (int) -> int

// }

// }

//

// The ModuleCompiler maintains a hash table (ExitMap) which allows a call site

// to add a new exit or reuse an existing one. The key is an ExitDescriptor

// (which holds the exit pairing) and the value is an index into the

// Vector<Exit> stored in the AsmJSModule.

//

// Rooting note: ModuleCompiler is a stack class that contains un-rooted

// PropertyName (JSAtom) pointers. This is safe because it cannot be

// constructed without a TokenStream reference. TokenStream is itself a stack

// class that cannot be constructed without an AutoKeepAtoms being live on the

// stack, which prevents collection of atoms.

//

// ModuleCompiler is marked as rooted in the rooting analysis. Don't add

// non-JSAtom pointers, or this will break!

typedef Js::Tick AsmJsCompileTime;

namespace AsmJsLookupSource

{

enum Source

{

AsmJsModule, AsmJsFunction

};

}

struct AsmJsModuleMemory

{

static const int32 MemoryTableBeginOffset = 0;

// Memory is allocated in this order

int32 mArrayBufferOffset

, mStdLibOffset

, mDoubleOffset

, mFuncOffset

, mFFIOffset

, mFuncPtrOffset

, mIntOffset

, mFloatOffset

, mSimdOffset // in SIMDValues

;

int32 mMemorySize;

};

struct AsmJsFunctionMemory

{

// Register where module slots are loaded

static const RegSlot ModuleSlotRegister = 0;

static const RegSlot ReturnRegister = 0;

static const RegSlot FunctionRegister = 0;

static const RegSlot CallReturnRegister = 0;

static const RegSlot ModuleEnvRegister = 1;

static const RegSlot ArrayBufferRegister = 2;

static const RegSlot ArraySizeRegister = 3;

static const RegSlot ScriptContextBufferRegister = 4;

//Var Return register and Module Environment and Array Buffer

static const int32 RequiredVarConstants = 5;

};

namespace AsmJsCompilation

{

enum Phases

{

Module,

ByteCode,

TemplateJIT,

Phases\_COUNT

};

};

class AsmJsModuleCompiler

{

struct AsmJsModuleExport

{

PropertyId id;

RegSlot location;

};

private:

typedef JsUtil::BaseDictionary<PropertyId, MathBuiltin, ArenaAllocator> MathNameMap;

typedef JsUtil::BaseDictionary<PropertyId, TypedArrayBuiltin, ArenaAllocator> ArrayNameMap;

typedef JsUtil::BaseDictionary<PropertyId, AsmJsSymbol\*, ArenaAllocator> ModuleEnvironment;

typedef JsUtil::List<AsmJsFunc\*, ArenaAllocator> ModuleFunctionArray;

typedef JsUtil::List<AsmJsFunctionTable\*, ArenaAllocator> ModuleFunctionTableArray;

typedef JsUtil::List<AsmJsModuleExport, ArenaAllocator> ModuleExportArray;

typedef JsUtil::Queue<AsmJsArrayView \*, ArenaAllocator> ModuleArrayViewList;

typedef AsmJsRegisterSpaceGeneric<int, 0> ModuleIntVars;

typedef AsmJsRegisterSpaceGeneric<double, 0> ModuleDoubleVars;

typedef AsmJsRegisterSpaceGeneric<float, 0> ModuleFloatVars;

typedef AsmJsRegisterSpaceGeneric<AsmJsImportFunction, 0> ModuleImportFunctions;

typedef AsmJsRegisterSpaceGeneric<AsmJsSIMDValue, 0> ModuleSIMDVars;

typedef JsUtil::BaseDictionary<PropertyId, AsmJsSIMDFunction\*, ArenaAllocator> SIMDNameMap;

inline bool LookupStdLibSIMDNameInMap (PropertyName name, AsmJsSIMDFunction \*\*simdFunc, SIMDNameMap\* map) const;

bool AddStandardLibrarySIMDNameInMap (PropertyId id, AsmJsSIMDFunction\* simdFunc, SIMDNameMap\* map);

// Keep allocator first to free Dictionary before deleting the allocator

ArenaAllocator mAllocator;

ExclusiveContext \* mCx;

AsmJSParser & mCurrentParserNode;

PropertyName mModuleFunctionName;

ParseNode \* mModuleFunctionNode;

MathNameMap mStandardLibraryMathNames;

ArrayNameMap mStandardLibraryArrayNames;

ModuleEnvironment mModuleEnvironment;

PropertyName mStdLibArgName, mForeignArgName, mBufferArgName;

ModuleFunctionArray mFunctionArray;

ModuleIntVars mIntVarSpace;

ModuleDoubleVars mDoubleVarSpace;

ModuleFloatVars mFloatVarSpace;

ModuleImportFunctions mImportFunctions;

// Maps functions names to func symbols. Three maps since names are not unique across SIMD types (e.g. SIMD.{float32x4|int32x4}.add)

// Also used to find if an operation is supported on a SIMD type.

SIMDNameMap mStdLibSIMDInt32x4Map;

SIMDNameMap mStdLibSIMDFloat32x4Map;

SIMDNameMap mStdLibSIMDFloat64x2Map;

// global SIMD values space.

ModuleSIMDVars mSimdVarSpace;

BVStatic<ASMSIMD\_BUILTIN\_SIZE> mAsmSimdBuiltinUsedBV;

ModuleExportArray mExports;

RegSlot mExportFuncIndex; // valid only if export object is empty

ModuleFunctionTableArray mFunctionTableArray;

int mVarImportCount;

int mVarCount;

int32 mFuncPtrTableCount;

AsmJsModuleMemory mModuleMemory;

AsmJsCompileTime mCompileTime;

AsmJsCompileTime mCompileTimeLastTick;

long mMaxAstSize;

BVStatic<ASMMATH\_BUILTIN\_SIZE> mAsmMathBuiltinUsedBV;

BVStatic<ASMARRAY\_BUILTIN\_SIZE> mAsmArrayBuiltinUsedBV;

AsmJsCompileTime mPhaseCompileTime[AsmJsCompilation::Phases\_COUNT];

ModuleArrayViewList mArrayViews;

uint mMaxHeapAccess;

#if DBG

bool mStdLibArgNameInit : 1;

bool mForeignArgNameInit : 1;

bool mBufferArgNameInit : 1;

#endif

bool mInitialised : 1;

bool mUsesChangeHeap : 1;

bool mUsesHeapBuffer : 1;

public:

AsmJsModuleCompiler( ExclusiveContext \*cx, AsmJSParser &parser );

bool Init();

bool InitSIMDBuiltins();

// Resolves a SIMD function name to its symbol

bool LookupStdLibSIMDName(PropertyId baseId, PropertyName fieldName, AsmJsSIMDFunction \*\*simdFunc);

bool LookupStdLibSIMDName(AsmJsSIMDBuiltinFunction baseId, PropertyName fieldName, AsmJsSIMDFunction \*\*simdFunc);

// Resolves a symbol name to SIMD constructor/operation and perform checks

AsmJsSIMDFunction \*LookupSimdConstructor(PropertyName name);

AsmJsSIMDFunction \*LookupSimdTypeCheck(PropertyName name);

AsmJsSIMDFunction \*LookupSimdOperation(PropertyName name);

void AddSimdBuiltinUse(int index){ mAsmSimdBuiltinUsedBV.Set(index); }

// adds SIMD constant var to module

bool AddSimdValueVar(PropertyName name, ParseNode\* pnode, AsmJsSIMDFunction\* simdFunc);

AsmJsCompileTime GetTick();

void AccumulateCompileTime();

void AccumulateCompileTime(AsmJsCompilation::Phases phase);

// Return compile time in ms

uint64 GetCompileTime() const;

void PrintCompileTrace() const;

// A valid module may have a NULL name

inline PropertyName GetModuleFunctionName() const{return mModuleFunctionName;}

inline ParseNode \*GetModuleFunctionNode() const{return mModuleFunctionNode;}

inline ArenaAllocator\* GetAllocator() {return &mAllocator;}

inline long GetMaxAstSize() const{return mMaxAstSize;}

inline void UpdateMaxAstSize( long val ){mMaxAstSize = val>mMaxAstSize?val:mMaxAstSize;}

//Mutable interface

inline void InitModuleName( PropertyName name ){mModuleFunctionName = name;}

inline void InitModuleNode( AsmJSParser &parser ){mModuleFunctionNode = parser;}

inline AsmJSParser& GetCurrentParserNode(){return mCurrentParserNode;}

inline void SetCurrentParseNode( AsmJSParser & val ){mCurrentParserNode = val;}

void InitStdLibArgName( PropertyName n );

void InitForeignArgName( PropertyName n );

void InitBufferArgName( PropertyName n );

PropertyName GetBufferArgName() const;

PropertyName GetForeignArgName() const;

PropertyName GetStdLibArgName() const;

BVStatic<ASMMATH\_BUILTIN\_SIZE> GetAsmMathBuiltinUsedBV();

void AddMathBuiltinUse(int index){ mAsmMathBuiltinUsedBV.Set(index); }

BVStatic<ASMARRAY\_BUILTIN\_SIZE> GetAsmArrayBuiltinUsedBV();

void AddArrayBuiltinUse(int index){ mAsmArrayBuiltinUsedBV.Set(index); }

bool LookupStandardLibraryMathName(PropertyName name, MathBuiltin \*mathBuiltin) const;

bool LookupStandardLibraryArrayName(PropertyName name, TypedArrayBuiltin \*builtin) const;

// Lookup the name in the function environment if provided, then the module environment

// indicate the origin of the symbol if specified

AsmJsSymbol\* LookupIdentifier( PropertyName name, AsmJsFunc\* func = nullptr, AsmJsLookupSource::Source\* lookupSource = nullptr );

AsmJsFunctionDeclaration\* LookupFunction( PropertyName name );

bool DefineIdentifier( PropertyName name, AsmJsSymbol\* symbol );

bool AddNumericVar( PropertyName name, ParseNode\* pnode, bool isFloat, bool isMutable = true);

bool AddGlobalVarImport( PropertyName name, PropertyName field, AsmJSCoercion coercion );

bool AddModuleFunctionImport( PropertyName name, PropertyName field );

bool AddNumericConst( PropertyName name, const double\* cst );

bool AddArrayView( PropertyName name, ArrayBufferView::ViewType type );

bool AddExport( PropertyName name, RegSlot location );

bool SetExportFunc( AsmJsFunc\* func );

bool AddFunctionTable( PropertyName name, const int size );

void AddMathLibName(PropertyId pid);

//Immutable interface

Parser \*GetParser() const;

ByteCodeGenerator\* GetByteCodeGenerator() const;

ScriptContext \*GetScriptContext() const;

bool FailName( ParseNode \*usepn, const wchar \*fmt, PropertyName name );

bool Fail( ParseNode\* usepn, const wchar \*error );

bool AreAllFuncTableDefined();

bool UsesChangeHeap() { return mUsesChangeHeap; }

bool UsesHeapBuffer() { return mUsesHeapBuffer; }

void SetUsesHeapBuffer(bool val) { mUsesHeapBuffer = val; }

void UpdateMaxHeapAccess(uint index);

uint GetMaxHeapAccess() { return mMaxHeapAccess; }

// Compile/Validate function name and arguments (define their types)

bool CompileFunction(AsmJsFunc \* func, int funcIndex);

bool CompileAllFunctions();

void RevertAllFunctions();

bool CommitFunctions();

bool CommitModule();

bool FinalizeModule();

AsmJsFunc\* CreateNewFunctionEntry(ParseNode\* pnodeFnc);

bool CheckChangeHeap(AsmJsFunc \* func);

void InitMemoryOffsets ();

inline int32 GetIntOffset () const{return mModuleMemory.mIntOffset;}

inline int32 GetFloatOffset () const{return mModuleMemory.mFloatOffset;}

inline int32 GetFuncPtrOffset () const{return mModuleMemory.mFuncPtrOffset;}

inline int32 GetFFIOffset () const{return mModuleMemory.mFFIOffset;}

inline int32 GetFuncOffset () const{return mModuleMemory.mFuncOffset;}

inline int32 GetDoubleOffset () const{return mModuleMemory.mDoubleOffset; }

inline int32 GetSimdOffset () const{ return mModuleMemory.mSimdOffset; }

inline int32 GetFuncPtrTableCount() const{return mFuncPtrTableCount;}

inline void SetFuncPtrTableCount ( int32 val ){mFuncPtrTableCount = val;}

private:

void RevertFunction(int funcIndex);

bool SetupFunctionArguments(AsmJsFunc \* func, ParseNodePtr pnode);

bool SetupLocalVariables(AsmJsFunc \* func);

void ASTPrepass(ParseNodePtr pnode, AsmJsFunc \* func);

void BindArguments(ParseNode\* argList);

bool AddStandardLibraryMathName(PropertyId id, AsmJsMathFunction\* func, AsmJSMathBuiltinFunction mathLibFunctionName);

bool AddStandardLibraryMathName(PropertyId id, const double\* cstAddr, AsmJSMathBuiltinFunction mathLibFunctionName);

bool AddStandardLibraryArrayName(PropertyId id, AsmJsTypedArrayFunction \* func, AsmJSTypedArrayBuiltinFunction mathLibFunctionName);

bool CheckByteLengthCall(ParseNode \* node, ParseNode \* newBufferDecl);

bool ValidateSimdConstructor(ParseNode\* pnode, AsmJsSIMDFunction\* simdFunc, AsmJsSIMDValue& value);

bool IsSimdjsEnabled() { return GetScriptContext()->GetConfig()->IsSimdjsEnabled(); }

};

struct AsmJsSlot

{

RegSlot location;

AsmJsSymbol::SymbolType symType;

union

{

AsmJsVarType::Which varType;

ArrayBufferView::ViewType viewType;

double mathConstVal;

uint funcTableSize;

AsmJsModuleArg::ArgType argType;

AsmJSMathBuiltinFunction builtinMathFunc;

AsmJSTypedArrayBuiltinFunction builtinArrayFunc;

AsmJsSIMDBuiltinFunction builtinSIMDFunc;

};

bool isConstVar = false;

};

class AsmJsModuleInfo

{

public:

/// proxy of asmjs module

struct ModuleVar

{

RegSlot location;

AsmJsVarType::Which type;

union

{

int intInit;

float floatInit;

double doubleInit;

AsmJsSIMDValue simdInit;

} initialiser;

bool isMutable;

};

struct ModuleVarImport

{

RegSlot location;

AsmJsVarType::Which type;

PropertyId field;

};

struct ModuleFunctionImport

{

RegSlot location;

PropertyId field;

};

struct ModuleFunction

{

RegSlot location;

};

struct ModuleExport

{

PropertyId\* id;

RegSlot\* location;

};

struct ModuleFunctionTable

{

uint size;

RegSlot\* moduleFunctionIndex;

};

typedef JsUtil::BaseDictionary<PropertyId, AsmJsSlot\*, Memory::Recycler> AsmJsSlotMap;

private:

Recycler\* mRecycler;

int mArgInCount; // for runtime validation of arguments in

int mVarCount, mVarImportCount, mFunctionImportCount, mFunctionCount, mFunctionTableCount, mExportsCount, mSlotsCount;

int mSimdRegCount; // part of mVarCount

PropertyIdArray\* mExports;

RegSlot\* mExportsFunctionLocation;

RegSlot mExportFunctionIndex; // valid only if export object is empty

ModuleVar\* mVars;

ModuleVarImport\* mVarImports;

ModuleFunctionImport\* mFunctionImports;

ModuleFunction\* mFunctions;

ModuleFunctionTable\* mFunctionTables;

AsmJsModuleMemory mModuleMemory;

AsmJsSlotMap\* mSlotMap;

BVStatic<ASMMATH\_BUILTIN\_SIZE> mAsmMathBuiltinUsed;

BVStatic<ASMARRAY\_BUILTIN\_SIZE> mAsmArrayBuiltinUsed;

BVStatic<ASMSIMD\_BUILTIN\_SIZE> mAsmSimdBuiltinUsed;

uint mMaxHeapAccess;

bool mUsesChangeHeap;

bool mIsProcessed;

public:

AsmJsModuleInfo( Recycler\* recycler ) :

mRecycler( recycler )

, mArgInCount( 0 )

, mVarCount( 0 )

, mVarImportCount( 0 )

, mFunctionImportCount( 0 )

, mFunctionCount( 0 )

, mFunctionTableCount( 0 )

, mSimdRegCount(0)

, mVars( nullptr )

, mVarImports( nullptr )

, mFunctionImports( nullptr )

, mFunctions( nullptr )

, mMaxHeapAccess(0)

, mUsesChangeHeap(false)

, mIsProcessed(false)

, mSlotMap(nullptr)

{

}

ModuleVar& GetVar( int i )

{

Assert( i < mVarCount );

return mVars[i];

}

void SetVar(int i, ModuleVar var)

{

Assert(i < mVarCount);

mVars[i] = var;

}

ModuleVarImport& GetVarImport( int i )

{

Assert( i < mVarImportCount );

return mVarImports[i];

}

void SetVarImport(int i, ModuleVarImport var)

{

Assert(i < mVarImportCount);

mVarImports[i] = var;

}

ModuleFunctionImport& GetFunctionImport( int i )

{

Assert( i < mFunctionImportCount );

return mFunctionImports[i];

}

void SetFunctionImport(int i, ModuleFunctionImport var)

{

Assert(i < mFunctionImportCount);

mFunctionImports[i] = var;

}

ModuleFunction& GetFunction( int i )

{

Assert( i < mFunctionCount );

return mFunctions[i];

}

void SetFunction(int i, ModuleFunction var)

{

Assert(i < mFunctionCount);

mFunctions[i] = var;

}

ModuleFunctionTable& GetFunctionTable( int i )

{

Assert( i < mFunctionTableCount );

return mFunctionTables[i];

}

void SetFunctionTable(int i, ModuleFunctionTable var)

{

Assert(i < mFunctionTableCount);

mFunctionTables[i] = var;

}

void SetFunctionTableSize( int index, uint size );

ModuleExport GetExport( int i )

{

ModuleExport ex;

ex.id = &mExports->elements[i];

ex.location = &mExportsFunctionLocation[i];

return ex;

}

RegSlot\* GetExportsFunctionLocation() const

{

return mExportsFunctionLocation;

}

PropertyIdArray\* GetExportsIdArray() const

{

return mExports;

}

AsmJsSlotMap\* GetAsmJsSlotMap()

{

return mSlotMap;

}

// Accessors

public:

inline Js::RegSlot GetExportFunctionIndex() const{return mExportFunctionIndex;}

inline void SetExportFunctionIndex( Js::RegSlot val ){mExportFunctionIndex = val;}

void SetExportsCount(int count);

inline int GetExportsCount()const

{

return mExportsCount;

}

inline int GetArgInCount() const

{

return mArgInCount;

}

inline void SetArgInCount( int val )

{

mArgInCount = val;

}

inline int GetFunctionCount() const

{

return mFunctionCount;

}

void SetFunctionCount( int val );

inline int GetFunctionTableCount() const

{

return mFunctionTableCount;

}

void SetFunctionTableCount( int val );

inline int GetFunctionImportCount() const

{

return mFunctionImportCount;

}

void SetFunctionImportCount( int val );

inline int GetVarImportCount() const

{

return mVarImportCount;

}

void SetVarImportCount( int val );

inline int GetVarCount() const

{

return mVarCount;

}

void SetVarCount( int val );

inline int GetSlotsCount() const

{

return mSlotsCount;

}

void InitializeSlotMap(int val);

inline bool IsRuntimeProcessed() const

{

return mIsProcessed;

}

void SetIsRuntimeProcessed(bool val)

{

mIsProcessed = val;

}

inline AsmJsModuleMemory& GetModuleMemory()

{

return mModuleMemory;

}

inline void SetModuleMemory( const AsmJsModuleMemory& val )

{

mModuleMemory = val;

}

inline void SetAsmMathBuiltinUsed(const BVStatic<ASMMATH\_BUILTIN\_SIZE> val)

{

mAsmMathBuiltinUsed = val;

}

inline BVStatic<ASMMATH\_BUILTIN\_SIZE> GetAsmMathBuiltinUsed()const

{

return mAsmMathBuiltinUsed;

}

inline void SetAsmArrayBuiltinUsed(const BVStatic<ASMARRAY\_BUILTIN\_SIZE> val)

{

mAsmArrayBuiltinUsed = val;

}

inline BVStatic<ASMARRAY\_BUILTIN\_SIZE> GetAsmArrayBuiltinUsed()const

{

return mAsmArrayBuiltinUsed;

}

void SetUsesChangeHeap(bool val)

{

mUsesChangeHeap = val;

}

inline bool GetUsesChangeHeap() const

{

return mUsesChangeHeap;

}

void SetMaxHeapAccess(uint val)

{

mMaxHeapAccess = val;

}

inline uint GetMaxHeapAccess() const

{

return mMaxHeapAccess;

}

inline void SetSimdRegCount(int val) { mSimdRegCount = val; }

inline int GetSimdRegCount() const { return mSimdRegCount; }

inline void SetAsmSimdBuiltinUsed(const BVStatic<ASMSIMD\_BUILTIN\_SIZE> val)

{

mAsmSimdBuiltinUsed = val;

}

inline BVStatic<ASMSIMD\_BUILTIN\_SIZE> GetAsmSimdBuiltinUsed()const

{

return mAsmSimdBuiltinUsed;

}

static void EnsureHeapAttached(ScriptFunction \* func);

static void \* ConvertFrameForJavascript(void\* asmJsMemory, ScriptFunction \* func);

};

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

// Portions of this file are copyright 2014 Mozilla Foundation, available under the Apache 2.0 license.

//-------------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------------

// Copyright 2014 Mozilla Foundation

//

// Licensed under the Apache License, Version 2.0 (the "License");

// you may not use this file except in compliance with the License.

// You may obtain a copy of the License at

//

// http ://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing, software

// distributed under the License is distributed on an "AS IS" BASIS,

// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

// See the License for the specific language governing permissions and

// limitations under the License.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

#include "ByteCode\ByteCodeWriter.h"

#include "ByteCode\AsmJsByteCodeWriter.h"

#include "Language\AsmJSByteCodeGenerator.h"

namespace Js

{

#if DBG\_DUMP

template<> void PrintTmpRegisterAllocation<double>(RegSlot loc)

{

if (PHASE\_ON1(AsmjsTmpRegisterAllocationPhase))

Output::Print(L"+D%d\n", loc);

}

template<> void PrintTmpRegisterDeAllocation<double>(RegSlot loc)

{

if (PHASE\_ON1(AsmjsTmpRegisterAllocationPhase))

Output::Print(L"-D%d\n", loc);

}

template<> void PrintTmpRegisterAllocation<float>(RegSlot loc)

{

if (PHASE\_ON1(AsmjsTmpRegisterAllocationPhase))

Output::Print(L"+F%d\n", loc);

}

template<> void PrintTmpRegisterDeAllocation<float>(RegSlot loc)

{

if (PHASE\_ON1(AsmjsTmpRegisterAllocationPhase))

Output::Print(L"-F%d\n", loc);

}

template<> void PrintTmpRegisterAllocation<int>(RegSlot loc)

{

if (PHASE\_ON1(AsmjsTmpRegisterAllocationPhase))

Output::Print(L"+I%d\n", loc);

}

template<> void PrintTmpRegisterDeAllocation<int>(RegSlot loc)

{

if (PHASE\_ON1(AsmjsTmpRegisterAllocationPhase))

Output::Print(L"-I%d\n", loc);

}

template<> void PrintTmpRegisterAllocation<AsmJsSIMDValue>(RegSlot loc)

{

if (PHASE\_ON1(AsmjsTmpRegisterAllocationPhase))

Output::Print(L"+SIMD%d\n", loc);

}

template<> void PrintTmpRegisterDeAllocation<AsmJsSIMDValue>(RegSlot loc)

{

if (PHASE\_ON1(AsmjsTmpRegisterAllocationPhase))

Output::Print(L"-SIMD%d\n", loc);

}

template<typename T> void PrintTmpRegisterAllocation(RegSlot loc) {}

template<typename T> void PrintTmpRegisterDeAllocation(RegSlot loc) {}

#endif

const wchar\_t \* AsmJsType::toChars() const

{

switch (which\_)

{

case Double: return L"double";

case MaybeDouble: return L"double?";

case DoubleLit: return L"doublelit";

case Float: return L"float";

case Floatish: return L"floatish";

case FloatishDoubleLit: return L"FloatishDoubleLit";

case MaybeFloat: return L"float?";

case Fixnum: return L"fixnum";

case Int: return L"int";

case Signed: return L"signed";

case Unsigned: return L"unsigned";

case Intish: return L"intish";

case Void: return L"void";

case Int32x4: return L"SIMD.Int32x4";

case Float32x4: return L"SIMD.Float32x4";

case Float64x2: return L"SIMD.Float64x2";

}

Assert(false);

return L"none";

}

bool AsmJsType::isSIMDType() const

{

return isSIMDInt32x4() || isSIMDFloat32x4() || isSIMDFloat64x2();

}

bool AsmJsType::isSIMDInt32x4() const

{

return which\_ == Int32x4;

}

bool AsmJsType::isSIMDFloat32x4() const

{

return which\_ == Float32x4;

}

bool AsmJsType::isSIMDFloat64x2() const

{

return which\_ == Float64x2;

}

bool AsmJsType::isVarAsmJsType() const

{

return isInt() || isMaybeDouble() || isMaybeFloat();

}

bool AsmJsType::isExtern() const

{

return isDouble() || isSigned();

}

bool AsmJsType::isVoid() const

{

return which\_ == Void;

}

bool AsmJsType::isFloatish() const

{

return isMaybeFloat() || which\_ == Floatish;

}

bool AsmJsType::isFloatishDoubleLit() const

{

return isFloatish() || isDoubleLit();

}

bool AsmJsType::isMaybeFloat() const

{

return isFloat() || which\_ == MaybeFloat;

}

bool AsmJsType::isFloat() const

{

return which\_ == Float;

}

bool AsmJsType::isMaybeDouble() const

{

return isDouble() || which\_ == MaybeDouble;

}

bool AsmJsType::isDouble() const

{

return isDoubleLit() || which\_ == Double;

}

bool AsmJsType::isDoubleLit() const

{

return which\_ == DoubleLit;

}

bool AsmJsType::isIntish() const

{

return isInt() || which\_ == Intish;

}

bool AsmJsType::isInt() const

{

return isSigned() || isUnsigned() || which\_ == Int;

}

bool AsmJsType::isUnsigned() const

{

return which\_ == Unsigned || which\_ == Fixnum;

}

bool AsmJsType::isSigned() const

{

return which\_ == Signed || which\_ == Fixnum;

}

bool AsmJsType::operator!=(AsmJsType rhs) const

{

return which\_ != rhs.which\_;

}

bool AsmJsType::operator==(AsmJsType rhs) const

{

return which\_ == rhs.which\_;

}

bool AsmJsType::isSubType(AsmJsType type) const

{

switch (type.which\_)

{

case Js::AsmJsType::Double:

return isDouble();

break;

case Js::AsmJsType::MaybeDouble:

return isMaybeDouble();

break;

case Js::AsmJsType::DoubleLit:

return isDoubleLit();

break;

case Js::AsmJsType::Float:

return isFloat();

break;

case Js::AsmJsType::MaybeFloat:

return isMaybeFloat();

break;

case Js::AsmJsType::Floatish:

return isFloatish();

break;

case Js::AsmJsType::FloatishDoubleLit:

return isFloatishDoubleLit();

break;

case Js::AsmJsType::Fixnum:

return which\_ == Fixnum;

break;

case Js::AsmJsType::Int:

return isInt();

break;

case Js::AsmJsType::Signed:

return isSigned();

break;

case Js::AsmJsType::Unsigned:

return isUnsigned();

break;

case Js::AsmJsType::Intish:

return isIntish();

break;

case Js::AsmJsType::Void:

return isVoid();

break;

case AsmJsType::Int32x4:

return isSIMDInt32x4();

break;

case AsmJsType::Float32x4:

return isSIMDFloat32x4();

break;

case AsmJsType::Float64x2:

return isSIMDFloat64x2();

break;

default:

break;

}

return false;

}

bool AsmJsType::isSuperType(AsmJsType type) const

{

return type.isSubType(which\_);

}

Js::AsmJsRetType AsmJsType::toRetType() const

{

Which w = which\_;

// DoubleLit is for expressions only.

if (w == DoubleLit)

{

w = Double;

}

return AsmJsRetType::Which(w);

}

/// RetType

bool AsmJsRetType::operator!=(AsmJsRetType rhs) const

{

return which\_ != rhs.which\_;

}

bool AsmJsRetType::operator==(AsmJsRetType rhs) const

{

return which\_ == rhs.which\_;

}

Js::AsmJsType AsmJsRetType::toType() const

{

return AsmJsType::Which(which\_);

}

Js::AsmJsVarType AsmJsRetType::toVarType() const

{

return AsmJsVarType::Which(which\_);

}

Js::AsmJsRetType::Which AsmJsRetType::which() const

{

return which\_;

}

AsmJsRetType::AsmJsRetType(AsmJSCoercion coercion)

{

switch (coercion)

{

case AsmJS\_ToInt32: which\_ = Signed; break;

case AsmJS\_ToNumber: which\_ = Double; break;

case AsmJS\_FRound: which\_ = Float; break;

case AsmJS\_Int32x4: which\_ = Int32x4; break;

case AsmJS\_Float32x4: which\_ = Float32x4; break;

case AsmJS\_Float64x2: which\_ = Float64x2; break;

}

}

AsmJsRetType::AsmJsRetType(Which w) : which\_(w)

{

}

AsmJsRetType::AsmJsRetType() : which\_(Which(-1))

{

}

/// VarType

bool AsmJsVarType::operator!=(AsmJsVarType rhs) const

{

return which\_ != rhs.which\_;

}

bool AsmJsVarType::operator==(AsmJsVarType rhs) const

{

return which\_ == rhs.which\_;

}

Js::AsmJsVarType AsmJsVarType::FromCheckedType(AsmJsType type)

{

Assert( type.isInt() || type.isMaybeDouble() || type.isFloatish() || type.isSIMDType());

if (type.isMaybeDouble())

return Double;

else if (type.isFloatish())

return Float;

else if (type.isInt())

return Int;

else

{

// SIMD type

return AsmJsVarType::Which(type.GetWhich());

}

}

Js::AsmJSCoercion AsmJsVarType::toCoercion() const

{

switch (which\_)

{

case Int: return AsmJS\_ToInt32;

case Double: return AsmJS\_ToNumber;

case Float: return AsmJS\_FRound;

case Int32x4: return AsmJS\_Int32x4;

case Float32x4: return AsmJS\_Float32x4;

case Float64x2: return AsmJS\_Float64x2;

}

Assert(false);

return AsmJS\_ToInt32;

}

Js::AsmJsType AsmJsVarType::toType() const

{

return AsmJsType::Which(which\_);

}

Js::AsmJsVarType::Which AsmJsVarType::which() const

{

return which\_;

}

AsmJsVarType::AsmJsVarType(AsmJSCoercion coercion)

{

switch (coercion)

{

case AsmJS\_ToInt32: which\_ = Int; break;

case AsmJS\_ToNumber: which\_ = Double; break;

case AsmJS\_FRound: which\_ = Float; break;

case AsmJS\_Int32x4: which\_ = Int32x4; break;

case AsmJS\_Float32x4: which\_ = Float32x4; break;

case AsmJS\_Float64x2: which\_ = Float64x2; break;

}

}

AsmJsVarType::AsmJsVarType(Which w) : which\_(w)

{

}

AsmJsVarType::AsmJsVarType() : which\_(Which(-1))

{

}

template<>

AsmJsMathConst\* Js::AsmJsSymbol::Cast()

{

Assert(mType == MathConstant);

return (AsmJsMathConst\*)this;

}

template<>

AsmJsVar\* Js::AsmJsSymbol::Cast()

{

Assert(mType == Variable);

return (AsmJsVar\*)this;

}

template<>

AsmJsVarBase\* Js::AsmJsSymbol::Cast()

{

Assert( mType == Argument || mType == Variable || mType == ConstantImport);

return ( AsmJsVarBase\* )this;

}

template<>

AsmJsFunctionDeclaration\* Js::AsmJsSymbol::Cast()

{

Assert(mType == ModuleFunction || mType == ImportFunction || mType == MathBuiltinFunction || mType == SIMDBuiltinFunction || mType == FuncPtrTable);

return (AsmJsFunctionDeclaration\*)this;

}

template<>

AsmJsFunc\* Js::AsmJsSymbol::Cast()

{

Assert(mType == ModuleFunction);

return (AsmJsFunc\*)this;

}

template<>

AsmJsImportFunction\* Js::AsmJsSymbol::Cast()

{

Assert(mType == ImportFunction);

return (AsmJsImportFunction\*)this;

}

template<>

AsmJsMathFunction\* Js::AsmJsSymbol::Cast()

{

Assert(mType == MathBuiltinFunction);

return (AsmJsMathFunction\*)this;

}

template<>

AsmJsSIMDFunction\* Js::AsmJsSymbol::Cast()

{

Assert(mType == SIMDBuiltinFunction);

return (AsmJsSIMDFunction\*) this;

}

template<>

AsmJsArrayView\* Js::AsmJsSymbol::Cast()

{

Assert(mType == ArrayView);

return (AsmJsArrayView\*)this;

}

template<>

AsmJsConstantImport\* Js::AsmJsSymbol::Cast()

{

Assert(mType == ConstantImport);

return (AsmJsConstantImport\*)this;

}

template<>

AsmJsFunctionTable\* Js::AsmJsSymbol::Cast()

{

Assert(mType == FuncPtrTable);

return (AsmJsFunctionTable\*)this;

}

template<>

AsmJsTypedArrayFunction\* Js::AsmJsSymbol::Cast()

{

Assert(mType == TypedArrayBuiltinFunction);

return (AsmJsTypedArrayFunction\*)this;

}

template<>

AsmJsModuleArg\* Js::AsmJsSymbol::Cast()

{

Assert(mType == ModuleArgument);

return (AsmJsModuleArg\*)this;

}

Js::AsmJsType AsmJsModuleArg::GetType() const

{

Assert(UNREACHED);

return AsmJsType::Void;

}

bool AsmJsModuleArg::isMutable() const

{

Assert(UNREACHED);

return true;

}

Js::AsmJsType AsmJsMathConst::GetType() const

{

return AsmJsType::Double;

}

bool AsmJsMathConst::isMutable() const

{

return false;

}

bool AsmJsFunctionDeclaration::EnsureArgCount(ArgSlot count)

{

if (mArgCount == Constants::InvalidArgSlot)

{

SetArgCount(count);

return true;

}

else

{

return mArgCount == count;

}

}

void AsmJsFunctionDeclaration::SetArgCount(ArgSlot count )

{

Assert( mArgumentsType == nullptr );

Assert(mArgCount == Constants::InvalidArgSlot);

Assert(count != Constants::InvalidArgSlot);

mArgCount = count;

if( count > 0 )

{

mArgumentsType = AnewArrayZ( mAllocator, AsmJsType, count );

}

}

AsmJsType\* AsmJsFunctionDeclaration::GetArgTypeArray()

{

return mArgumentsType;

}

bool AsmJsFunctionDeclaration::CheckAndSetReturnType(Js::AsmJsRetType val)

{

Assert((val != AsmJsRetType::Fixnum && val != AsmJsRetType::Unsigned && val != AsmJsRetType::Floatish) || GetSymbolType() == AsmJsSymbol::MathBuiltinFunction);

if (mReturnTypeKnown)

{

Assert((mReturnType != AsmJsRetType::Fixnum && mReturnType != AsmJsRetType::Unsigned && mReturnType != AsmJsRetType::Floatish) || GetSymbolType() == AsmJsSymbol::MathBuiltinFunction);

return mReturnType.toType().isSubType(val.toType());

}

mReturnType = val;

mReturnTypeKnown = true;

return true;

}

Js::AsmJsType AsmJsFunctionDeclaration::GetType() const

{

return mReturnType.toType();

}

bool AsmJsFunctionDeclaration::isMutable() const

{

return false;

}

bool AsmJsFunctionDeclaration::EnsureArgType(AsmJsVarBase\* arg, ArgSlot index)

{

if (mArgumentsType[index].GetWhich() == -1)

{

SetArgType(arg, index);

return true;

}

else

{

return mArgumentsType[index] == arg->GetType();

}

}

bool AsmJsFunctionDeclaration::SupportsArgCall( ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType )

{

// we will assume the first reference to the function is correct, until proven wrong

if (GetArgCount() == Constants::InvalidArgSlot)

{

SetArgCount(argCount);

for (ArgSlot i = 0; i < argCount; i++)

{

if (args[i].isSubType(AsmJsType::Double))

{

mArgumentsType[i] = AsmJsType::Double;

}

else if (args[i].isSubType(AsmJsType::Float))

{

mArgumentsType[i] = AsmJsType::Float;

}

else if (args[i].isSubType(AsmJsType::Int))

{

mArgumentsType[i] = AsmJsType::Int;

}

else if (args[i].isSIMDType())

{

mArgumentsType[i] = args[i];

}

else

{

// call did not have valid argument type

return false;

}

}

retType = mReturnType;

return true;

}

else if( argCount == GetArgCount() )

{

for(ArgSlot i = 0; i < argCount; i++ )

{

if (!args[i].isSubType(mArgumentsType[i]))

{

return false;

}

}

retType = mReturnType;

return true;

}

return false;

}

ArgSlot AsmJsFunctionDeclaration::GetArgByteSize(ArgSlot inArgCount) const

{

uint argSize = 0;

if (GetSymbolType() == AsmJsSymbol::ImportFunction)

{

Assert(inArgCount != Constants::InvalidArgSlot);

argSize = inArgCount \* MachPtr;

}

#if \_M\_IX86

else

{

for (ArgSlot i = 0; i < GetArgCount(); i++)

{

if( GetArgType(i).isMaybeDouble() )

{

argSize += sizeof(double);

}

else if (GetArgType(i).isIntish())

{

argSize += sizeof(int);

}

else if (GetArgType(i).isFloatish())

{

argSize += sizeof(float);

}

else if (GetArgType(i).isSIMDType())

{

argSize += sizeof(AsmJsSIMDValue);

}

else

{

Assume(UNREACHED);

}

}

}

#elif \_M\_X64

else

{

for (ArgSlot i = 0; i < GetArgCount(); i++)

{

if (GetArgType(i).isSIMDType())

{

argSize += sizeof(AsmJsSIMDValue);

}

else

{

argSize += MachPtr;

}

}

}

#else

Assert(UNREACHED);

#endif

if (argSize >= (1 << 16))

{

// throw OOM on overflow

Throw::OutOfMemory();

}

return static\_cast<ArgSlot>(argSize);

}

AsmJsMathFunction::AsmJsMathFunction( PropertyName name, ArenaAllocator\* allocator, ArgSlot argCount, AsmJSMathBuiltinFunction builtIn, OpCodeAsmJs op, AsmJsRetType retType, ... ) :

AsmJsFunctionDeclaration( name, AsmJsSymbol::MathBuiltinFunction, allocator )

, mBuiltIn( builtIn )

, mOverload( nullptr )

, mOpCode(op)

{

bool ret = CheckAndSetReturnType(retType);

Assert(ret);

va\_list arguments;

SetArgCount( argCount );

va\_start( arguments, retType );

for(ArgSlot iArg = 0; iArg < argCount; iArg++)

{

SetArgType(va\_arg(arguments, AsmJsType), iArg);

}

va\_end(arguments);

}

void AsmJsMathFunction::SetOverload(AsmJsMathFunction\* val)

{

#if DBG

AsmJsMathFunction\* over = val->mOverload;

while (over)

{

if (over == this)

{

Assert(false);

break;

}

over = over->mOverload;

}

#endif

Assert(val->GetSymbolType() == GetSymbolType());

if (this->mOverload)

{

this->mOverload->SetOverload(val);

}

else

{

mOverload = val;

}

}

bool AsmJsMathFunction::CheckAndSetReturnType(Js::AsmJsRetType val)

{

return AsmJsFunctionDeclaration::CheckAndSetReturnType(val) || (mOverload && mOverload->CheckAndSetReturnType(val));

}

bool AsmJsMathFunction::SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType )

{

return AsmJsFunctionDeclaration::SupportsArgCall(argCount, args, retType) || (mOverload && mOverload->SupportsArgCall(argCount, args, retType));

}

bool AsmJsMathFunction::SupportsMathCall(ArgSlot argCount, AsmJsType\* args, OpCodeAsmJs& op, AsmJsRetType& retType )

{

if (AsmJsFunctionDeclaration::SupportsArgCall(argCount, args, retType))

{

op = mOpCode;

return true;

}

return mOverload && mOverload->SupportsMathCall(argCount, args, op, retType);

}

AsmJsFunc::AsmJsFunc(PropertyName name, ParseNode\* pnodeFnc, ArenaAllocator\* allocator) :

AsmJsFunctionDeclaration(name, AsmJsSymbol::ModuleFunction, allocator)

, mCompileTime(0)

, mVarMap(allocator)

, mBodyNode(nullptr)

, mFncNode(pnodeFnc)

, mIntRegisterSpace(allocator)

, mFloatRegisterSpace(allocator)

, mDoubleRegisterSpace( allocator )

, mFuncInfo(pnodeFnc->sxFnc.funcInfo)

, mFuncBody(nullptr)

, mSimdRegisterSpace(allocator)

, mSimdVarsList(allocator)

, mArgOutDepth(0)

, mMaxArgOutDepth(0)

, mDefined( false )

{

}

/// AsmJsFunc

AsmJsVarBase\* AsmJsFunc::DefineVar( PropertyName name, bool isArg /\*= false\*/, bool isMutable /\*= true\*/ )

{

AsmJsVarBase\* var = FindVar(name);

if (var)

{

Output::Print(L"Variable redefinition: %s\n", name->Psz());

return nullptr;

}

if (isArg)

{

// arg cannot be const

Assert(isMutable);

var = Anew( mAllocator, AsmJsArgument, name );

}

else

{

var = Anew(mAllocator, AsmJsVar, name, isMutable);

}

int addResult = mVarMap.AddNew(name->GetPropertyId(), var);

if( addResult == -1 )

{

mAllocator->Free(var, isArg ? sizeof(AsmJsArgument) : sizeof(AsmJsVar));

return nullptr;

}

return var;

}

AsmJsVarBase\* AsmJsFunc::FindVar(const PropertyName name) const

{

return mVarMap.LookupWithKey(name->GetPropertyId(), nullptr);

}

void AsmJsFunc::ReleaseLocationGeneric(const EmitExpressionInfo\* pnode)

{

if (pnode)

{

if (pnode->type.isIntish())

{

ReleaseLocation<int>(pnode);

}

else if (pnode->type.isMaybeDouble())

{

ReleaseLocation<double>(pnode);

}

else if (pnode->type.isFloatish())

{

ReleaseLocation<float>(pnode);

}

else if (pnode->type.isSIMDType())

{

ReleaseLocation<AsmJsSIMDValue>(pnode);

}

}

}

AsmJsSymbol\* AsmJsFunc::LookupIdentifier(const PropertyName name, AsmJsLookupSource::Source\* lookupSource /\*= nullptr \*/) const

{

auto var = FindVar(name);

if (var && lookupSource)

{

\*lookupSource = AsmJsLookupSource::AsmJsFunction;

}

return var;

}

void AsmJsFunc::SetArgOutDepth( int outParamsCount )

{

mArgOutDepth = outParamsCount;

}

void AsmJsFunc::UpdateMaxArgOutDepth(int outParamsCount)

{

if (mMaxArgOutDepth < outParamsCount)

{

mMaxArgOutDepth = outParamsCount;

}

}

bool AsmJsFunctionInfo::Init(AsmJsFunc\* func)

{

const auto& intRegisterSpace = func->GetRegisterSpace<int>();

const auto& doubleRegisterSpace = func->GetRegisterSpace<double>();

const auto& floatRegisterSpace = func->GetRegisterSpace<float>();

const auto& simdRegisterSpace = func->GetRegisterSpace<AsmJsSIMDValue>();

mIntConstCount = (intRegisterSpace.GetConstCount());

mDoubleConstCount = (doubleRegisterSpace.GetConstCount());

mFloatConstCount = (floatRegisterSpace.GetConstCount());

bool isSimdjsEnabled = func->GetFuncBody()->GetScriptContext()->GetConfig()->IsSimdjsEnabled();

if (isSimdjsEnabled)

{

mSimdConstCount = (simdRegisterSpace.GetConstCount());

}

Recycler\* recycler = func->GetFuncBody()->GetScriptContext()->GetRecycler();

mArgCount = func->GetArgCount();

if (mArgCount > 0)

{

mArgType = RecyclerNewArrayLeaf(recycler, AsmJsVarType::Which, mArgCount);

}

// on x64, AsmJsExternalEntryPoint reads first 3 elements to figure out how to shadow args on stack

// always alloc space for these such that we need to do less work in the entrypoint

mArgSizesLength = max(mArgCount, 3ui16);

mArgSizes = RecyclerNewArrayLeafZ(recycler, uint, mArgSizesLength);

mbyteCodeTJMap = RecyclerNew(recycler, ByteCodeToTJMap,recycler);

for(ArgSlot i = 0; i < GetArgCount(); i++)

{

AsmJsType varType = func->GetArgType(i);

SetArgType(AsmJsVarType::FromCheckedType(varType), i);

}

mIntVarCount = intRegisterSpace.GetVarCount();

mDoubleVarCount = doubleRegisterSpace.GetVarCount();

mFloatVarCount = floatRegisterSpace.GetVarCount();

if (isSimdjsEnabled)

{

mSimdVarCount = simdRegisterSpace.GetVarCount();

}

mIntTmpCount = intRegisterSpace.GetTmpCount();

mDoubleTmpCount = doubleRegisterSpace.GetTmpCount();

mFloatTmpCount = floatRegisterSpace.GetTmpCount();

if (isSimdjsEnabled)

{

mSimdTmpCount = simdRegisterSpace.GetTmpCount();

}

mReturnType = func->GetReturnType();

mIntByteOffset = AsmJsFunctionMemory::RequiredVarConstants \* sizeof(Var);

const int totalIntCount = mIntConstCount + mIntVarCount + mIntTmpCount;

const int count32bitsVarConst = (AsmJsFunctionMemory::RequiredVarConstants\*sizeof(Var)) / sizeof(int);

const int total32bitsBeforeFloat = count32bitsVarConst + totalIntCount;

const int floatOffset32bitsFix = total32bitsBeforeFloat;

// Offset of floats from (float\*)m\_localSlot

mFloatByteOffset = (floatOffset32bitsFix \*sizeof(int));

const int totalFloatCount = mFloatConstCount + mFloatVarCount + mFloatTmpCount;

const int total32bitsBeforeDoubles = total32bitsBeforeFloat + totalFloatCount;

// if its an odd number, add 1

const int doubleOffset32bitsFix = total32bitsBeforeDoubles + (total32bitsBeforeDoubles & 1);

// Offset of doubles from (double\*)m\_localSlot

mDoubleByteOffset = (doubleOffset32bitsFix \*sizeof(int));

if (isSimdjsEnabled)

{

const int totalDoubleCount = mDoubleConstCount + mDoubleVarCount + mDoubleTmpCount;

mSimdByteOffset = mDoubleByteOffset + totalDoubleCount \* sizeof(double);

}

if (PHASE\_TRACE1(AsmjsInterpreterStackPhase))

{

Output::Print(L"ASMFunctionInfo Stack Data\n");

Output::Print(L"==========================\n");

Output::Print(L"RequiredVarConstants:%d\n", AsmJsFunctionMemory::RequiredVarConstants);

Output::Print(L"IntOffset:%d IntConstCount:%d IntVarCount:%d IntTmpCount:%d\n", mIntByteOffset, mIntConstCount, mIntVarCount, mIntTmpCount);

Output::Print(L"FloatOffset:%d FloatConstCount:%d FloatVarCount:%d FloatTmpCount:%d\n", mFloatByteOffset, mFloatConstCount, mFloatVarCount, mFloatTmpCount);

Output::Print(L"DoubleOffset:%d DoubleConstCount:%d DoubleVarCount:%d DoubleTmpCount:%d\n", mDoubleByteOffset, mDoubleConstCount, mDoubleVarCount, mDoubleTmpCount);

if (isSimdjsEnabled)

{

Output::Print(L"SimdOffset:%d SimdConstCount:%d SimdVarCount:%d SimdTmpCount:%d\n", mSimdByteOffset, mSimdConstCount, mSimdVarCount, mSimdTmpCount);

}

Output::Print(L"\n");

}

return true;

}

int AsmJsFunctionInfo::GetTotalSizeinBytes() const

{

int size = mDoubleByteOffset + (mDoubleConstCount + mDoubleVarCount + mDoubleTmpCount) \* sizeof(double);

// SimdJs values

size += GetSimdAllCount()\* sizeof(AsmJsSIMDValue);

return size;

}

void AsmJsFunctionInfo::SetArgType(AsmJsVarType type, ArgSlot index)

{

Assert(mArgCount != Constants::InvalidArgSlot);

AnalysisAssert(index < mArgCount);

Assert(type.which() == AsmJsVarType::Int || type.which() == AsmJsVarType::Float || type.which() == AsmJsVarType::Double || type.isSIMD());

mArgType[index] = type.which();

mArgSizes[index] = 0;

// add 4 if int, 8 if double

if (type.isDouble())

{

mArgByteSize = UInt16Math::Add(mArgByteSize, sizeof(double));

mArgSizes[index] = sizeof(double);

}

else if (type.isSIMD())

{

mArgByteSize = UInt16Math::Add(mArgByteSize, sizeof(AsmJsSIMDValue));

mArgSizes[index] = sizeof(AsmJsSIMDValue);

}

else

{

mArgByteSize = UInt16Math::Add(mArgByteSize, sizeof(Var));

mArgSizes[index] = MachPtr;

}

}

Js::AsmJsType AsmJsArrayView::GetType() const

{

switch (mViewType)

{

case ArrayBufferView::TYPE\_INT8:

case ArrayBufferView::TYPE\_INT16:

case ArrayBufferView::TYPE\_INT32:

case ArrayBufferView::TYPE\_UINT8:

case ArrayBufferView::TYPE\_UINT16:

case ArrayBufferView::TYPE\_UINT32:

return AsmJsType::Intish;

case ArrayBufferView::TYPE\_FLOAT32:

return AsmJsType::MaybeFloat;

case ArrayBufferView::TYPE\_FLOAT64:

return AsmJsType::MaybeDouble;

default:;

}

AssertMsg(false, "Unexpected array type");

return AsmJsType::Intish;

}

bool AsmJsArrayView::isMutable() const

{

return false;

}

bool AsmJsImportFunction::SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType )

{

for (ArgSlot i = 0; i < argCount ; i++)

{

if (!args[i].isExtern())

{

return false;

}

}

return true;

}

AsmJsImportFunction::AsmJsImportFunction(PropertyName name, PropertyName field, ArenaAllocator\* allocator) :

AsmJsFunctionDeclaration(name, AsmJsSymbol::ImportFunction, allocator)

, mField(field)

{

CheckAndSetReturnType(AsmJsRetType::Void);

}

bool AsmJsFunctionTable::SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType )

{

if (mAreArgumentsKnown)

{

return AsmJsFunctionDeclaration::SupportsArgCall(argCount, args, retType);

}

Assert(GetArgCount() == Constants::InvalidArgSlot);

SetArgCount( argCount );

retType = this->GetReturnType();

for (ArgSlot i = 0; i < argCount ; i++)

{

if (args[i].isInt())

{

this->SetArgType(AsmJsType::Int, i);

}

else if (args[i].isDouble())

{

this->SetArgType(AsmJsType::Double, i);

}

else if (args[i].isFloat())

{

this->SetArgType(AsmJsType::Float, i);

}

else

{

// Function tables can only have int, double or float as arguments

return false;

}

}

mAreArgumentsKnown = true;

return true;

}

AsmJsSIMDFunction::AsmJsSIMDFunction(PropertyName name, ArenaAllocator\* allocator, ArgSlot argCount, AsmJsSIMDBuiltinFunction builtIn, OpCodeAsmJs op, AsmJsRetType retType, ...) :

AsmJsFunctionDeclaration(name, AsmJsSymbol::SIMDBuiltinFunction, allocator)

, mBuiltIn(builtIn)

, mOverload(nullptr)

, mOpCode(op)

{

bool ret = CheckAndSetReturnType(retType);

Assert(ret);

va\_list arguments;

SetArgCount(argCount);

va\_start(arguments, retType);

for (ArgSlot iArg = 0; iArg < argCount; iArg++)

{

SetArgType(va\_arg(arguments, AsmJsType), iArg);

}

va\_end(arguments);

}

bool AsmJsSIMDFunction::SupportsSIMDCall(ArgSlot argCount, AsmJsType\* args, OpCodeAsmJs& op, AsmJsRetType& retType)

{

if (AsmJsFunctionDeclaration::SupportsArgCall(argCount, args, retType))

{

op = mOpCode;

return true;

}

return mOverload && mOverload->SupportsSIMDCall(argCount, args, op, retType);

}

bool AsmJsSIMDFunction::SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType)

{

return AsmJsFunctionDeclaration::SupportsArgCall(argCount, args, retType) || (mOverload && mOverload->SupportsArgCall(argCount, args, retType));

}

bool AsmJsSIMDFunction::CheckAndSetReturnType(Js::AsmJsRetType val)

{

return AsmJsFunctionDeclaration::CheckAndSetReturnType(val) || (mOverload && mOverload->CheckAndSetReturnType(val));

}

void AsmJsSIMDFunction::SetOverload(AsmJsSIMDFunction\* val)

{

#if DBG

AsmJsSIMDFunction\* over = val->mOverload;

while (over)

{

if (over == this)

{

Assert(false);

break;

}

over = over->mOverload;

}

#endif

Assert(val->GetSymbolType() == GetSymbolType());

if (this->mOverload)

{

this->mOverload->SetOverload(val);

}

else

{

mOverload = val;

}

}

bool AsmJsSIMDFunction::IsTypeCheck()

{

return mBuiltIn == AsmJsSIMDBuiltin\_int32x4\_check || mBuiltIn == AsmJsSIMDBuiltin\_float32x4\_check || mBuiltIn == AsmJsSIMDBuiltin\_float64x2\_check;

}

AsmJsVarType AsmJsSIMDFunction::GetTypeCheckVarType()

{

Assert(this->IsTypeCheck());

return GetReturnType().toVarType();

}

bool AsmJsSIMDFunction::IsConstructor()

{

return mBuiltIn == AsmJsSIMDBuiltin\_Int32x4 || mBuiltIn == AsmJsSIMDBuiltin\_Float32x4 || mBuiltIn == AsmJsSIMDBuiltin\_Float64x2;

}

// Is a constructor with the correct argCount ?

bool AsmJsSIMDFunction::IsConstructor(uint argCount)

{

if (!IsConstructor())

{

return false;

}

switch (mBuiltIn)

{

case AsmJsSIMDBuiltin\_Float64x2:

return argCount == 2;

case AsmJsSIMDBuiltin\_Float32x4:

case AsmJsSIMDBuiltin\_Int32x4:

return argCount == 4;

};

return false;

}

AsmJsVarType AsmJsSIMDFunction::GetConstructorVarType()

{

Assert(this->IsConstructor());

return GetReturnType().toVarType();

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

// Portions of this file are copyright 2014 Mozilla Foundation, available under the Apache 2.0 license.

//-------------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------------

// Copyright 2014 Mozilla Foundation

//

// Licensed under the Apache License, Version 2.0 (the "License");

// you may not use this file except in compliance with the License.

// You may obtain a copy of the License at

//

// http ://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing, software

// distributed under the License is distributed on an "AS IS" BASIS,

// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

// See the License for the specific language governing permissions and

// limitations under the License.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

typedef uint32 uint32\_t;

typedef IdentPtr PropertyName;

typedef ParseNode\* AsmJSParser;

// These EcmaScript-defined coercions form the basis of the asm.js type system.

enum AsmJSCoercion

{

AsmJS\_ToInt32,

AsmJS\_ToNumber,

AsmJS\_FRound,

AsmJS\_Int32x4,

AsmJS\_Float32x4,

AsmJS\_Float64x2,

};

namespace ArrayBufferView

{

enum ViewType

{

TYPE\_INT8 = 0,

TYPE\_UINT8,

TYPE\_INT16,

TYPE\_UINT16,

TYPE\_INT32,

TYPE\_UINT32,

TYPE\_FLOAT32,

TYPE\_FLOAT64,

TYPE\_INVALID

};

} /\* namespace ArrayBufferView \*/

// The asm.js spec recognizes this set of builtin Math functions.

enum AsmJSMathBuiltinFunction

{

#define ASMJS\_MATH\_FUNC\_NAMES(name, propertyName) AsmJSMathBuiltin\_##name,

#include "AsmJsBuiltinNames.h"

AsmJSMathBuiltinFunction\_COUNT,

#define ASMJS\_MATH\_CONST\_NAMES(name, propertyName) AsmJSMathBuiltin\_##name,

#include "AsmJsBuiltinNames.h"

AsmJSMathBuiltin\_COUNT

};

enum AsmJSTypedArrayBuiltinFunction

{

#define ASMJS\_ARRAY\_NAMES(name, propertyName) AsmJSTypedArrayBuiltin\_##name,

#include "AsmJsBuiltinNames.h"

AsmJSTypedArrayBuiltin\_COUNT

};

// Represents the type of a general asm.js expression.

class AsmJsType

{

public:

enum Which

{

Int,

Double,

Float,

MaybeDouble,

DoubleLit, // Double literal. Needed for SIMD.js. Sub-type of Double

MaybeFloat,

Floatish,

FloatishDoubleLit, // A sum-type for Floatish and DoubleLit. Needed for float32x4(..) arg types.

Fixnum,

Signed,

Unsigned,

Intish,

Void,

Int32x4,

Float32x4,

Float64x2

};

private:

Which which\_;

public:

AsmJsType() : which\_( Which( -1 ) ){}

AsmJsType( Which w ) : which\_( w ){}

bool operator==( AsmJsType rhs ) const;

bool operator!=( AsmJsType rhs ) const;

inline Js::AsmJsType::Which GetWhich() const{return which\_;}

bool isSigned() const;

bool isUnsigned() const;

bool isInt() const;

bool isIntish() const;

bool isDouble() const;

bool isMaybeDouble() const;

bool isDoubleLit() const;

bool isFloat() const;

bool isMaybeFloat() const;

bool isFloatish() const;

bool isFloatishDoubleLit() const;

bool isVoid() const;

bool isExtern() const;

bool isVarAsmJsType() const;

bool isSubType( AsmJsType type ) const;

bool isSuperType( AsmJsType type ) const;

const wchar\_t \*toChars() const;

bool isSIMDType() const;

bool isSIMDInt32x4() const;

bool isSIMDFloat32x4() const;

bool isSIMDFloat64x2() const;

AsmJsRetType toRetType() const;

};

// Represents the subset of AsmJsType that can be used as the return AsmJsType of a

// function.

class AsmJsRetType

{

public:

enum Which

{

Void = AsmJsType::Void,

Signed = AsmJsType::Signed,

Double = AsmJsType::Double,

Float = AsmJsType::Float,

Fixnum = AsmJsType::Fixnum,

Unsigned = AsmJsType::Unsigned,

Floatish = AsmJsType::Floatish,

Int32x4 = AsmJsType::Int32x4,

Float32x4 = AsmJsType::Float32x4,

Float64x2 = AsmJsType::Float64x2

};

private:

Which which\_;

public:

AsmJsRetType();

AsmJsRetType( Which w );

AsmJsRetType( AsmJSCoercion coercion );

Which which() const;

AsmJsType toType() const;

AsmJsVarType toVarType() const;

bool operator==( AsmJsRetType rhs ) const;

bool operator!=( AsmJsRetType rhs ) const;

};

// Represents the subset of Type that can be used as a variable or

// argument's type. Note: AsmJSCoercion and VarType are kept separate to

// make very clear the signed/int distinction: a coercion may explicitly sign

// an \*expression\* but, when stored as a variable, this signedness information

// is explicitly thrown away by the asm.js type system. E.g., in

//

// function f(i) {

// i = i | 0; (1)

// if (...)

// i = foo() >>> 0;

// else

// i = bar() | 0;

// return i | 0; (2)

// }

//

// the AsmJSCoercion of (1) is Signed (since | performs ToInt32) but, when

// translated to an VarType, the result is a plain Int since, as shown, it

// is legal to assign both Signed and Unsigned (or some other Int) values to

// it. For (2), the AsmJSCoercion is also Signed but, when translated to an

// RetType, the result is Signed since callers (asm.js and non-asm.js) can

// rely on the return value being Signed.

class AsmJsVarType

{

public:

enum Which : byte

{

Int = AsmJsType::Int,

Double = AsmJsType::Double,

Float = AsmJsType::Float,

Int32x4 = AsmJsType::Int32x4,

Float32x4 = AsmJsType::Float32x4,

Float64x2 = AsmJsType::Float64x2

};

private:

Which which\_;

public:

AsmJsVarType();

AsmJsVarType( Which w );

AsmJsVarType( AsmJSCoercion coercion );

Which which() const;

AsmJsType toType() const;

AsmJSCoercion toCoercion() const;

static AsmJsVarType FromCheckedType( AsmJsType type );

inline bool isInt()const {return which\_ == Int; }

inline bool isDouble()const {return which\_ == Double; }

inline bool isFloat()const {return which\_ == Float; }

inline bool isInt32x4()const { return which\_ == Int32x4; }

inline bool isFloat32x4()const { return which\_ == Float32x4; }

inline bool isFloat64x2()const { return which\_ == Float64x2; }

inline bool isSIMD() const { return isInt32x4() || isFloat32x4() || isFloat64x2(); }

bool operator==( AsmJsVarType rhs ) const;

bool operator!=( AsmJsVarType rhs ) const;

};

// Implements <: (subtype) operator when the RHS is an VarType

static inline bool

operator<=( AsmJsType lhs, AsmJsVarType rhs )

{

switch( rhs.which() )

{

case AsmJsVarType::Int: return lhs.isInt();

case AsmJsVarType::Double: return lhs.isDouble();

case AsmJsVarType::Float: return lhs.isFloat();

}

AssertMsg( false, "Unexpected RHS type" );

}

// Base class for all the symbol in Asm.Js during compilation

// Defined by a type and a name

class AsmJsSymbol

{

public:

enum SymbolType

{

Variable,

Argument,

MathConstant,

ConstantImport,

ImportFunction,

FuncPtrTable,

ModuleFunction,

ArrayView,

MathBuiltinFunction,

TypedArrayBuiltinFunction,

/\*SIMDVariable,\*/

SIMDBuiltinFunction,

ModuleArgument

};

private:

// name of the symbol, all symbols must have unique names

PropertyName mName;

// Type of the symbol, used for casting

SymbolType mType;

public:

// Constructor

AsmJsSymbol(PropertyName name, SymbolType type) : mName(name), mType(type) { }

// Accessor for the name

inline PropertyName GetName() const{return mName;}

// Sets the name of the symbol

inline void SetName(PropertyName name) {mName = name;}

// Returns the type of the symbol

inline SymbolType GetSymbolType()const { return mType; }

// Casts the symbol to a derived class, additional test done to make sure is it the right type

template<typename T>

T\* Cast();

// AsmJsSymbol interface

public:

// retrieve the type of the symbol when it is use in an expression

virtual AsmJsType GetType() const = 0;

// if the symbol is mutable, it can be on the LHS of an assignment operation

virtual bool isMutable() const = 0;

};

// Symbol representing a module argument

class AsmJsModuleArg : public AsmJsSymbol

{

public:

enum ArgType: int8

{

StdLib,

Import,

Heap

};

private:

ArgType mArgType;

public:

// Constructor

AsmJsModuleArg(PropertyName name, ArgType type) : AsmJsSymbol(name, AsmJsSymbol::ModuleArgument), mArgType(type) { }

// Accessor

inline const ArgType GetArgType()const { return mArgType; }

// AsmJsSymbol interface

public:

virtual AsmJsType GetType() const override;

virtual bool isMutable() const override;

};

// Symbol representing a double constant from the standard library

class AsmJsMathConst : public AsmJsSymbol

{

// address of the constant, lifetime of this address must be for the whole execution of the program (global var)

const double\* mVal;

public:

// Constructor

AsmJsMathConst(PropertyName name, const double\* val) : AsmJsSymbol(name, AsmJsSymbol::MathConstant), mVal(val) { }

// Accessor

inline const double\* GetVal()const { return mVal; }

// AsmJsSymbol interface

public:

virtual AsmJsType GetType() const override;

virtual bool isMutable() const override;

};

// Base class defining Variables in asm.js, can be a variable of the module or a function argument

class AsmJsVarBase : public AsmJsSymbol

{

// type of the variable, isDouble => double registerSpace, isInt => int registerSpace

AsmJsVarType mType;

// register where the value of this variable resides

RegSlot mLocation;

bool mIsMutable;

public:

// Constructor

AsmJsVarBase(PropertyName name, AsmJsSymbol::SymbolType type, bool isMutable = true) :

AsmJsSymbol(name, type)

, mType(AsmJsVarType::Double)

, mLocation(Js::Constants::NoRegister)

, mIsMutable(isMutable)

{

}

// Accessors

inline Js::RegSlot GetLocation() const { return mLocation; }

inline void SetLocation( Js::RegSlot val ) { mLocation = val; }

inline AsmJsVarType GetVarType() const { return mType; }

inline void SetVarType( const AsmJsVarType& type ){ mType = type; }

// AsmJsSymbol interface

public:

virtual AsmJsType GetType() const override

{

return GetVarType().toType();

}

virtual bool isMutable() const override

{

return mIsMutable;

}

};

// Defines a Variable, a variable can be changed and has a default value used to initialize the variable.

// Function and the module can have variables

class AsmJsVar : public AsmJsVarBase

{

// register of the const value that initialize this variable, NoRegister for Args

union

{

double doubleVal;

float floatVal;

int intVal;

AsmJsSIMDValue simdVal;

}mConstInitialiser;

public:

// Constructors

AsmJsVar( PropertyName name, bool isMutable = true) :

AsmJsVarBase(name, AsmJsSymbol::Variable, isMutable)

{

mConstInitialiser.doubleVal = 0;

}

// Accessors

inline void SetConstInitialiser ( double val ){ mConstInitialiser.doubleVal = val; }

inline double GetDoubleInitialiser() const { return mConstInitialiser.doubleVal; }

inline void SetConstInitialiser(float val) { mConstInitialiser.floatVal = val; }

inline float GetFloatInitialiser() const { return mConstInitialiser.floatVal; }

inline void SetConstInitialiser ( int val ) { mConstInitialiser.intVal = val; }

inline int GetIntInitialiser () const { return mConstInitialiser.intVal; }

inline void SetConstInitialiser(AsmJsSIMDValue val) { mConstInitialiser.simdVal = val; }

inline AsmJsSIMDValue GetSimdConstInitialiser() { return mConstInitialiser.simdVal; }

};

// AsmJsArgument defines the arguments of a function

class AsmJsArgument : public AsmJsVarBase

{

public:

// Constructor

AsmJsArgument( PropertyName name ) :

AsmJsVarBase( name, AsmJsSymbol::Argument )

{

}

};

// AsmJsConstantImport defines a variable that is initialized by an import from the foreign object

class AsmJsConstantImport : public AsmJsVarBase

{

// name of the field used to initialize the variable, i.e.: var i1 = foreign.mField;

PropertyName mField;

public:

// Constructor

AsmJsConstantImport( PropertyName name, PropertyName field ) :

AsmJsVarBase( name, AsmJsSymbol::ConstantImport ),

mField( field )

{

}

// Accessor

inline Js::PropertyName GetField() const { return mField; }

};

#if DBG\_DUMP

// Function used to debug Temporary register allocation in the bytecode generator

template<typename T> void PrintTmpRegisterAllocation( RegSlot loc );

template<typename T> void PrintTmpRegisterDeAllocation( RegSlot loc );

#endif

/// Register space for const, parameters, variables and tmp values

/// --------------------------------------------------------

/// | 0 (Reserved) | Consts | Parameters | Variables | Tmp

/// --------------------------------------------------------

/// Cannot allocate in any different order

template<typename T, RegSlot Reserved\_Slots\_Count>

class AsmJsRegisterSpaceGeneric

{

// Total number of register allocated

RegSlot mRegisterCount;

// location of the first temporary register and last variable + 1

RegSlot mFirstTmpReg;

// Location of the next register to be allocated

RegSlot mNextLocation;

// number of const, includes the reserved slots

RegSlot mNbConst;

public:

// Constructor

AsmJsRegisterSpaceGeneric() :

mRegisterCount( Reserved\_Slots\_Count )

, mFirstTmpReg( Reserved\_Slots\_Count )

, mNextLocation( Reserved\_Slots\_Count )

, mNbConst( Reserved\_Slots\_Count )

{

CompileAssert( Reserved\_Slots\_Count >= 0 );

}

// Get the number of const allocated

inline RegSlot GetConstCount() const { return mNbConst; }

// Get the location of the first temporary register

inline RegSlot GetFirstTmpRegister() const{ return mFirstTmpReg; }

// Get the total number of temporary register allocated

inline RegSlot GetTmpCount() const { return mRegisterCount-mFirstTmpReg; }

// Get number of local variables

inline RegSlot GetVarCount() const { return mFirstTmpReg - mNbConst; }

// Get the total number of variable allocated ( including temporaries )

inline RegSlot GetTotalVarCount() const { return mRegisterCount - mNbConst; }

inline RegSlot GetRegisterCount()const { return mRegisterCount; }

// Acquire a location for a register. Use only for arguments and Variables

inline RegSlot AcquireRegister()

{

// Makes sure no temporary register have been allocated yet

Assert( mFirstTmpReg == mRegisterCount && mNextLocation == mFirstTmpReg );

++mFirstTmpReg;

++mRegisterCount;

return mNextLocation++;

}

// Acquire a location for a constant

inline RegSlot AcquireConstRegister()

{

++mNbConst;

return AcquireRegister();

}

// Acquire a location for a temporary register

RegSlot AcquireTmpRegister()

{

// Make sure this function is called correctly

Assert( this->mNextLocation <= this->mRegisterCount && this->mNextLocation >= this->mFirstTmpReg );

// Allocate a new temp pseudo-register, increasing the locals count if necessary.

if( this->mNextLocation == this->mRegisterCount )

{

++this->mRegisterCount;

}

#if DBG\_DUMP

PrintTmpRegisterAllocation<T>( mNextLocation );

#endif

return mNextLocation++;

}

// Release a location for a temporary register, must be the last location acquired

void ReleaseTmpRegister( RegSlot tmpReg )

{

// make sure the location released is valid

Assert( tmpReg != Constants::NoRegister );

// Put this reg back on top of the temp stack (if it's a temp).

if( this->IsTmpReg( tmpReg ) )

{

Assert( tmpReg == this->mNextLocation - 1 );

#if DBG\_DUMP

PrintTmpRegisterDeAllocation<T>( mNextLocation-1 );

#endif

this->mNextLocation--;

}

}

// Checks if the register is a temporary register

bool IsTmpReg( RegSlot tmpReg )

{

Assert( this->mFirstTmpReg != Js::Constants::NoRegister );

return !IsConstReg( tmpReg ) && tmpReg >= mFirstTmpReg;

}

// Checks if the register is a const register

bool IsConstReg( RegSlot reg )

{

// a register is const if it is between the first register and the end of consts

return reg < mNbConst && reg != 0;

}

// Checks if the register is a variable register

bool IsVarReg( RegSlot reg )

{

// a register is a var if it is between the last const and the end

// equivalent to reg>=mNbConst && reg<mRegisterCount

// forcing unsigned, if reg < mNbConst then reg-mNbConst = 0xFFFFF..

return (uint32\_t)( reg - mNbConst ) < (uint32\_t)( mRegisterCount - mNbConst );

}

// Releases a location if its a temporary, safe to call with any expression

void ReleaseLocation( const EmitExpressionInfo \*pnode )

{

// Release the temp assigned to this expression so it can be re-used.

if( pnode && pnode->location != Js::Constants::NoRegister )

{

this->ReleaseTmpRegister( pnode->location );

}

}

// Checks if the location points to a temporary register

bool IsTmpLocation( const EmitExpressionInfo\* pnode )

{

if( pnode && pnode->location != Js::Constants::NoRegister )

{

return IsTmpReg( pnode->location );

}

return false;

}

// Checks if the location points to a constant register

bool IsConstLocation( const EmitExpressionInfo\* pnode )

{

if( pnode && pnode->location != Js::Constants::NoRegister )

{

return IsConstReg( pnode->location );

}

return false;

}

// Checks if the location points to a variable register

bool IsVarLocation( const EmitExpressionInfo\* pnode )

{

if( pnode && pnode->location != Js::Constants::NoRegister )

{

return IsVarReg( pnode->location );

}

return false;

}

// Checks if the location is valid ( within bounds of already allocated registers )

bool IsValidLocation( const EmitExpressionInfo\* pnode )

{

if( pnode && pnode->location != Js::Constants::NoRegister )

{

return pnode->location < mRegisterCount;

}

return false;

}

};

template <typename T>

struct AsmJsComparer : public DefaultComparer<T> {};

template <>

struct AsmJsComparer<float>

{

\_\_inline static bool Equals(float x, float y)

{

int32 i32x = \*(int32\*)&x;

int32 i32y = \*(int32\*)&y;

return i32x == i32y;

}

\_\_inline static hash\_t GetHashCode(float i)

{

return (hash\_t)i;

}

};

template <>

struct AsmJsComparer<double>

{

\_\_inline static bool Equals(double x, double y)

{

int64 i64x = \*(int64\*)&x;

int64 i64y = \*(int64\*)&y;

return i64x == i64y;

}

\_\_inline static hash\_t GetHashCode(double d)

{

\_\_int64 i64 = \*(\_\_int64\*)&d;

return (uint)((i64 >> 32) ^ (uint)i64);

}

};

// Register space use by the function, include a map to quickly find the location assigned to constants

template<typename T>

class AsmJsRegisterSpace : public AsmJsRegisterSpaceGeneric < T, 1 > // reserves 1 location for return

{

typedef JsUtil::BaseDictionary<T, RegSlot, ArenaAllocator, PowerOf2SizePolicy, AsmJsComparer> ConstMap;

// Map for constant and their location

ConstMap mConstMap;

public:

// Constructor

AsmJsRegisterSpace( ArenaAllocator\* allocator ) :

mConstMap( allocator )

{

}

inline void AddConst( T val )

{

if( !mConstMap.ContainsKey( val ) )

{

mConstMap.Add( val, AcquireConstRegister() );

}

}

inline RegSlot GetConstRegister( T val ) const

{

return mConstMap.LookupWithKey( val, Constants::NoRegister );

}

inline const ConstMap GetConstMap()

{

return mConstMap;

}

};

class AsmJsFunctionDeclaration : public AsmJsSymbol

{

AsmJsRetType mReturnType;

ArgSlot mArgCount;

RegSlot mLocation;

AsmJsType\* mArgumentsType;

bool mReturnTypeKnown : 1;

protected:

ArenaAllocator\* mAllocator;

public:

AsmJsFunctionDeclaration( PropertyName name, AsmJsSymbol::SymbolType type, ArenaAllocator\* allocator):

AsmJsSymbol( name, type )

, mAllocator(allocator)

, mReturnType( AsmJsRetType::Void )

, mArgCount(Constants::InvalidArgSlot)

, mLocation( 0 )

, mReturnTypeKnown( false )

, mArgumentsType(nullptr)

{ }

// returns false if the current return type is known and different

virtual bool CheckAndSetReturnType( Js::AsmJsRetType val );

inline Js::AsmJsRetType GetReturnType() const{return mReturnType;}

bool EnsureArgCount(ArgSlot count);

void SetArgCount(ArgSlot count );

ArgSlot GetArgCount() const

{

return mArgCount;

}

AsmJsType\* GetArgTypeArray();

const AsmJsType& GetArgType( ArgSlot index ) const

{

Assert( mArgumentsType && index < GetArgCount() );

return mArgumentsType[index];

}

void SetArgType(const AsmJsType& arg, ArgSlot index)

{

Assert( index < GetArgCount() ); mArgumentsType[index] = arg;

}

void SetArgType(AsmJsVarBase\* arg, ArgSlot index)

{

Assert( mArgumentsType != nullptr && index < GetArgCount() );

SetArgType( arg->GetType(), index );

}

bool EnsureArgType(AsmJsVarBase\* arg, ArgSlot index);

inline Js::RegSlot GetFunctionIndex() const{return mLocation;}

inline void SetFunctionIndex( Js::RegSlot val ){mLocation = val;}

// argCount : number of arguments to check

// args : dynamic array with the argument type

// retType : returnType associated with this function signature

virtual bool SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType);

// Return the size in bytes of the arguments, inArgCount is the number of argument in the call ( can be different than mArgCount for FFI )

ArgSlot GetArgByteSize(ArgSlot inArgCount) const;

//AsmJsSymbol interface

virtual AsmJsType GetType() const;

virtual bool isMutable() const;

};

class AsmJsMathFunction : public AsmJsFunctionDeclaration

{

AsmJSMathBuiltinFunction mBuiltIn;

// chain list of supported signature (t1,t2,...) -> retType

// careful not to create a cycle in the chain

AsmJsMathFunction\* mOverload;

OpCodeAsmJs mOpCode;

public:

AsmJsMathFunction(PropertyName name, ArenaAllocator\* allocator, ArgSlot argCount, AsmJSMathBuiltinFunction builtIn, OpCodeAsmJs op, AsmJsRetType retType, ...);

void SetOverload( AsmJsMathFunction\* val );

AsmJSMathBuiltinFunction GetMathBuiltInFunction(){ return mBuiltIn; };

virtual bool CheckAndSetReturnType( Js::AsmJsRetType val ) override;

bool SupportsMathCall(ArgSlot argCount, AsmJsType\* args, OpCodeAsmJs& op, AsmJsRetType& retType);

private:

virtual bool SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType ) override;

};

class AsmJsTypedArrayFunction : public AsmJsFunctionDeclaration

{

AsmJSTypedArrayBuiltinFunction mBuiltIn;

ArrayBufferView::ViewType mType;

public:

AsmJsTypedArrayFunction(PropertyName name, ArenaAllocator\* allocator, AsmJSTypedArrayBuiltinFunction builtIn, ArrayBufferView::ViewType type) :

AsmJsFunctionDeclaration(name, AsmJsSymbol::TypedArrayBuiltinFunction, allocator), mBuiltIn(builtIn), mType(type) { }

AsmJSTypedArrayBuiltinFunction GetArrayBuiltInFunction(){ return mBuiltIn; };

ArrayBufferView::ViewType GetViewType(){ return mType; };

};

class AsmJsImportFunction : public AsmJsFunctionDeclaration

{

PropertyName mField;

public:

AsmJsImportFunction( PropertyName name, PropertyName field, ArenaAllocator\* allocator );

inline Js::PropertyName GetField() const

{

return mField;

}

// We cannot know the return type of an Import Function so always think its return type is correct

virtual bool CheckAndSetReturnType( Js::AsmJsRetType val ) override{return true;}

virtual bool SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType ) override;

};

class AsmJsFunctionTable : public AsmJsFunctionDeclaration

{

typedef JsUtil::List<RegSlot, ArenaAllocator> FuncIndexTable;

FuncIndexTable mTable;

unsigned int mSize;

bool mIsDefined : 1;

bool mAreArgumentsKnown : 1;

public:

AsmJsFunctionTable( PropertyName name, ArenaAllocator\* allocator ) :

AsmJsFunctionDeclaration( name, AsmJsSymbol::FuncPtrTable, allocator )

, mTable(allocator)

, mSize( 0 )

, mIsDefined( false )

, mAreArgumentsKnown( false )

{

}

inline bool IsDefined() const{return mIsDefined;}

inline void Define(){mIsDefined = true;}

inline uint GetSize() const{return mSize;}

inline void SetSize( unsigned int val )

{

mSize = val;

mTable.EnsureArray( mSize );

}

inline void SetModuleFunctionIndex( RegSlot funcIndex, unsigned int index )

{

Assert( index < mSize );

mTable.SetItem( index, funcIndex );

}

inline RegSlot GetModuleFunctionIndex( unsigned int index )

{

Assert( index < mSize );

return mTable.Item( index );

}

virtual bool SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType );

};

class AsmJsFunc : public AsmJsFunctionDeclaration

{

typedef JsUtil::BaseDictionary<PropertyId, AsmJsVarBase\*, ArenaAllocator> VarNameMap;

unsigned mCompileTime; //unused

VarNameMap mVarMap;

ParseNode\* mBodyNode;

ParseNode\* mFncNode;

AsmJsRegisterSpace<int> mIntRegisterSpace;

AsmJsRegisterSpace<float> mFloatRegisterSpace;

AsmJsRegisterSpace<double> mDoubleRegisterSpace;

typedef JsUtil::List<AsmJsVarBase\*, ArenaAllocator> SIMDVarsList;

AsmJsRegisterSpace<AsmJsSIMDValue> mSimdRegisterSpace;

SIMDVarsList mSimdVarsList;

FuncInfo\* mFuncInfo;

FunctionBody\* mFuncBody;

int mArgOutDepth;

int mMaxArgOutDepth;

ULONG mOrigParseFlags;

bool mDeferred;

bool mDefined : 1; // true when compiled completely without any errors

public:

AsmJsFunc( PropertyName name, ParseNode\* pnodeFnc, ArenaAllocator\* allocator );

unsigned GetCompileTime() const { return mCompileTime; }

void AccumulateCompileTime(unsigned ms) { mCompileTime += ms; }

inline ParseNode\* GetFncNode() const{ return mFncNode; }

inline void SetFncNode(ParseNode\* fncNode) { mFncNode = fncNode; }

inline FuncInfo\* GetFuncInfo() const{ return mFuncInfo; }

inline void SetFuncInfo(FuncInfo\* fncInfo) { mFuncInfo = fncInfo; }

inline FunctionBody\*GetFuncBody() const{ return mFuncBody; }

inline void SetFuncBody(FunctionBody\* fncBody) { mFuncBody = fncBody; }

inline ULONG GetOrigParseFlags() const{ return mOrigParseFlags; }

inline void SetOrigParseFlags(ULONG parseFlags) { mOrigParseFlags = parseFlags; }

inline ParseNode\* GetBodyNode() const{return mBodyNode;}

inline void SetBodyNode( ParseNode\* val ){mBodyNode = val;}

inline void Finish() { mDefined = true; }

inline bool IsDefined()const { return mDefined; }

inline void SetDeferred() { mDeferred = true; }

inline bool IsDeferred()const { return mDeferred; }

template<typename T> inline AsmJsRegisterSpace<T>& GetRegisterSpace() {return \*(AsmJsRegisterSpace<T>\*)&mIntRegisterSpace;}

template<> inline AsmJsRegisterSpace<int>& GetRegisterSpace(){return mIntRegisterSpace;}

template<> inline AsmJsRegisterSpace<double>& GetRegisterSpace(){return mDoubleRegisterSpace;}

template<> inline AsmJsRegisterSpace<float>& GetRegisterSpace(){ return mFloatRegisterSpace; }

template<> inline AsmJsRegisterSpace<AsmJsSIMDValue>& GetRegisterSpace() { return mSimdRegisterSpace; }

inline SIMDVarsList& GetSimdVarsList() { return mSimdVarsList; }

/// Wrapper for RegisterSpace methods

template<typename T> inline RegSlot AcquireRegister (){return GetRegisterSpace<T>().AcquireRegister();}

template<typename T> inline void AddConst ( T val ){GetRegisterSpace<T>().AddConst( val );}

template<typename T> inline RegSlot GetConstRegister ( T val ){return GetRegisterSpace<T>().GetConstRegister( val );}

template<typename T> inline RegSlot AcquireTmpRegister(){return GetRegisterSpace<T>().AcquireTmpRegister();}

template<typename T> inline void ReleaseTmpRegister ( Js::RegSlot tmpReg ){GetRegisterSpace<T>().ReleaseTmpRegister( tmpReg );}

template<typename T> inline void ReleaseLocation ( const EmitExpressionInfo\* pnode ){GetRegisterSpace<T>().ReleaseLocation( pnode );}

template<typename T> inline bool IsTmpLocation ( const EmitExpressionInfo\* pnode ){return GetRegisterSpace<T>().IsTmpLocation( pnode );}

template<typename T> inline bool IsConstLocation ( const EmitExpressionInfo\* pnode ){return GetRegisterSpace<T>().IsConstLocation( pnode );}

template<typename T> inline bool IsVarLocation ( const EmitExpressionInfo\* pnode ){return GetRegisterSpace<T>().IsVarLocation( pnode );}

template<typename T> inline bool IsValidLocation ( const EmitExpressionInfo\* pnode ){return GetRegisterSpace<T>().IsValidLocation( pnode );}

void ReleaseLocationGeneric( const EmitExpressionInfo\* pnode );

// Search for a var in the varMap of the function, return nullptr if not found

AsmJsVarBase\* FindVar( const PropertyName name ) const;

// Defines a new variable int the function, return nullptr if already exists or theres an error

AsmJsVarBase\* DefineVar(PropertyName name, bool isArg = false, bool isMutable = true);

AsmJsSymbol\* LookupIdentifier( const PropertyName name, AsmJsLookupSource::Source\* lookupSource = nullptr ) const;

void SetArgOutDepth(int outParamsCount);

void UpdateMaxArgOutDepth(int outParamsCount);

inline int GetArgOutDepth() const{ return mArgOutDepth; }

inline int GetMaxArgOutDepth() const{ return mMaxArgOutDepth; }

};

struct MathBuiltin

{

enum Kind

{

Function, Constant

};

Kind kind;

AsmJSMathBuiltinFunction mathLibFunctionName;

union

{

const double\* cst;

AsmJsMathFunction\* func;

} u;

MathBuiltin() : kind( Kind( -1 ) )

{

}

MathBuiltin(AsmJSMathBuiltinFunction mathLibFunctionName, const double\* cst) : kind(Constant), mathLibFunctionName(mathLibFunctionName)

{

u.cst = cst;

}

MathBuiltin(AsmJSMathBuiltinFunction mathLibFunctionName, AsmJsMathFunction\* func) : kind(Function), mathLibFunctionName(mathLibFunctionName)

{

u.func = func;

}

};

struct TypedArrayBuiltin

{

AsmJSTypedArrayBuiltinFunction mArrayLibFunctionName;

AsmJsTypedArrayFunction\* mFunc;

TypedArrayBuiltin() { }

TypedArrayBuiltin(AsmJSTypedArrayBuiltinFunction arrayLibFunctionName, AsmJsTypedArrayFunction\* func) :

mArrayLibFunctionName(arrayLibFunctionName),

mFunc(func)

{ }

};

class AsmJsArrayView : public AsmJsSymbol

{

ArrayBufferView::ViewType mViewType;

public:

AsmJsArrayView( PropertyName name, ArrayBufferView::ViewType viewType ) :

AsmJsSymbol( name, AsmJsSymbol::ArrayView )

, mViewType( viewType )

{

}

virtual AsmJsType GetType() const;

virtual bool isMutable() const;

inline ArrayBufferView::ViewType GetViewType() const

{

return mViewType;

}

};

class AsmJsFunctionInfo

{

int mIntConstCount, mDoubleConstCount, mFloatConstCount;

ArgSlot mArgCount;

int mIntVarCount, mDoubleVarCount, mFloatVarCount, mIntTmpCount, mDoubleTmpCount, mFloatTmpCount;

AsmJsVarType::Which \* mArgType;

ArgSlot mArgSizesLength;

uint \* mArgSizes;

ArgSlot mArgByteSize;

// offset in Byte from the beggining of the stack aka R0

int mIntByteOffset, mDoubleByteOffset, mFloatByteOffset;

AsmJsRetType mReturnType;

bool mIsHeapBufferConst;

bool mUsesHeapBuffer;

int mSimdConstCount, mSimdVarCount, mSimdTmpCount, mSimdByteOffset;

FunctionBody\* asmJsModuleFunctionBody;

public:

AsmJsFunctionInfo() : mArgCount(0),

mIntConstCount(0),

mFloatConstCount(0),

mDoubleConstCount(0),

mIntVarCount(0),

mDoubleVarCount(0),

mFloatVarCount(0),

mIntTmpCount(0),

mDoubleTmpCount(0),

mFloatTmpCount(0),

mArgSizesLength(0),

mIntByteOffset(0),

mDoubleByteOffset(0),

mFloatByteOffset(0),

mReturnType(AsmJsRetType::Void),

mArgByteSize(0),

mSimdConstCount(0),

mSimdVarCount(0),

mSimdTmpCount(0),

mSimdByteOffset(0),

asmJsModuleFunctionBody(nullptr),

mTJBeginAddress(nullptr),

mUsesHeapBuffer(false),

mIsHeapBufferConst(false),

mArgType(nullptr),

mArgSizes(nullptr) {}

// the key is the bytecode address

typedef JsUtil::BaseDictionary<int, ptrdiff\_t, Recycler> ByteCodeToTJMap;

ByteCodeToTJMap\* mbyteCodeTJMap;

BYTE\* mTJBeginAddress;

inline int GetDoubleConstCount() const{ return mDoubleConstCount; }

inline void SetDoubleConstCount(int val) { mDoubleConstCount = val; }

inline int GetFloatConstCount() const{ return mFloatConstCount; }

inline void SetFloatConstCount(int val) { mFloatConstCount = val; }

inline int GetIntConstCount() const{return mIntConstCount;}

inline void SetIntConstCount(int val) { mIntConstCount = val; }

inline int GetIntVarCount()const { return mIntVarCount; }

inline void SetIntVarCount(int val) { mIntVarCount = val; }

inline int GetFloatVarCount()const { return mFloatVarCount; }

inline void SetFloatVarCount(int val) { mFloatVarCount = val; }

inline int GetDoubleVarCount()const { return mDoubleVarCount; }

inline void SetDoubleVarCount(int val) { mDoubleVarCount = val; }

inline int GetIntTmpCount()const { return mIntTmpCount; }

inline void SetIntTmpCount(int val) { mIntTmpCount = val; }

inline int GetFloatTmpCount()const { return mFloatTmpCount; }

inline void SetFloatTmpCount(int val) { mFloatTmpCount = val; }

inline int GetDoubleTmpCount()const { return mDoubleTmpCount; }

inline void SetDoubleTmpCount(int val) { mDoubleTmpCount = val; }

inline ArgSlot GetArgCount() const{ return mArgCount; }

inline void SetArgCount(ArgSlot val) { mArgCount = val; }

inline AsmJsRetType GetReturnType() const{return mReturnType;}

inline void SetReturnType(AsmJsRetType val) { mReturnType = val; }

inline ArgSlot GetArgByteSize() const{return mArgByteSize;}

inline void SetArgByteSize(ArgSlot val) { mArgByteSize = val; }

inline int GetDoubleByteOffset() const{ return mDoubleByteOffset; }

inline void SetDoubleByteOffset(int val) { mDoubleByteOffset = val; }

inline int GetFloatByteOffset() const{ return mFloatByteOffset; }

inline void SetFloatByteOffset(int val) { mFloatByteOffset = val; }

inline int GetIntByteOffset() const{ return mIntByteOffset; }

inline void SetIntByteOffset(int val) { mIntByteOffset = val; }

inline void SetIsHeapBufferConst(bool val) { mIsHeapBufferConst = val; }

inline bool IsHeapBufferConst() const{ return mIsHeapBufferConst; }

inline void SetUsesHeapBuffer(bool val) { mUsesHeapBuffer = val; }

inline bool UsesHeapBuffer() const{ return mUsesHeapBuffer; }

inline int GetSimdConstCount() const { return mSimdConstCount; }

inline void SetSimdConstCount(int val) { mSimdConstCount = val; }

inline int GetSimdVarCount() const { return mSimdVarCount; }

inline void SetSimdVarCount(int val) { mSimdVarCount = val; }

inline int GetSimdTmpCount() const { return mSimdTmpCount; }

inline void SetSimdTmpCount(int val) { mSimdTmpCount = val; }

inline int GetSimdByteOffset() const { return mSimdByteOffset; }

inline void SetSimdByteOffset(int val) { mSimdByteOffset = val; }

inline int GetSimdAllCount() const { return GetSimdConstCount() + GetSimdVarCount() + GetSimdTmpCount(); }

int GetTotalSizeinBytes()const;

void SetArgType(AsmJsVarType type, ArgSlot index);

inline AsmJsVarType GetArgType(ArgSlot index ) const

{

Assert(mArgCount != Constants::InvalidArgSlot);

AnalysisAssert( index < mArgCount);

return mArgType[index];

}

bool Init( AsmJsFunc\* func );

void SetModuleFunctionBody(FunctionBody\* body){ asmJsModuleFunctionBody = body; };

FunctionBody\* GetModuleFunctionBody()const{ return asmJsModuleFunctionBody; };

ArgSlot GetArgSizeArrayLength()

{

return mArgSizesLength;

}

void SetArgSizeArrayLength(ArgSlot val)

{

mArgSizesLength = val;

}

uint\* GetArgsSizesArray()

{

return mArgSizes;

}

void SetArgsSizesArray(uint\* val)

{

mArgSizes = val;

}

AsmJsVarType::Which \* GetArgTypeArray()

{

return mArgType;

}

void SetArgTypeArray(AsmJsVarType::Which\* val)

{

mArgType = val;

}

};

// The asm.js spec recognizes this set of builtin SIMD functions.

// !! Note: keep these grouped by SIMD type

enum AsmJsSIMDBuiltinFunction

{

#define ASMJS\_SIMD\_NAMES(name, propertyName) AsmJsSIMDBuiltin\_##name,

#include "AsmJsBuiltinNames.h"

AsmJsSIMDBuiltin\_COUNT

};

// SIMD built-in function symbol

// Do we have overloads for any SIMD function ?

class AsmJsSIMDFunction : public AsmJsFunctionDeclaration

{

AsmJsSIMDBuiltinFunction mBuiltIn;

AsmJsSIMDFunction\* mOverload;

OpCodeAsmJs mOpCode;

public:

AsmJsSIMDFunction(PropertyName name, ArenaAllocator\* allocator, ArgSlot argCount, AsmJsSIMDBuiltinFunction builtIn, OpCodeAsmJs op, AsmJsRetType retType, ...);

PropertyId GetBuiltinPropertyId();

void SetOverload(AsmJsSIMDFunction\* val);

AsmJsSIMDBuiltinFunction GetSimdBuiltInFunction(){ return mBuiltIn; };

virtual bool CheckAndSetReturnType(Js::AsmJsRetType val) override;

bool SupportsSIMDCall(ArgSlot argCount, AsmJsType\* args, OpCodeAsmJs& op, AsmJsRetType& retType);

bool IsConstructor();

bool IsConstructor(uint argCount);

bool IsTypeCheck(); // e.g. float32x4(x)

bool IsInt32x4Func() { return mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_Int32x4 && mBuiltIn < AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_Float32x4; }

bool IsFloat32x4Func() { return mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_Float32x4 && mBuiltIn < AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_Float64x2; }

bool IsFloat64x2Func() { return mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_Float64x2 && mBuiltIn < AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_COUNT; }

bool IsSimdLoadFunc()

{

return (mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_int32x4\_load && mBuiltIn <= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_int32x4\_load3) ||

(mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_load && mBuiltIn <= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_load3) ||

(mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float64x2\_load && mBuiltIn <= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float64x2\_load1);

}

bool IsSimdStoreFunc()

{

return (mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_int32x4\_store && mBuiltIn <= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_int32x4\_store3) ||

(mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_store && mBuiltIn <= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_store3) ||

(mBuiltIn >= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float64x2\_store && mBuiltIn <= AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float64x2\_store1);

}

bool IsExtractLaneFunc()

{

return (

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_int32x4\_extractLane ||

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_extractLane

);

}

bool IsReplaceLaneFunc()

{

return (

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_int32x4\_replaceLane ||

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_replaceLane

);

}

bool IsLaneAccessFunc()

{

return (

IsExtractLaneFunc() || IsReplaceLaneFunc()

);

}

bool IsShuffleFunc()

{

return (

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_int32x4\_shuffle ||

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_shuffle ||

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float64x2\_shuffle

);

}

bool IsSwizzleFunc()

{

return (

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_int32x4\_swizzle ||

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float32x4\_swizzle ||

mBuiltIn == AsmJsSIMDBuiltinFunction::AsmJsSIMDBuiltin\_float64x2\_swizzle

);

}

static bool SameTypeOperations(AsmJsSIMDFunction \*func1, AsmJsSIMDFunction \*func2)

{

bool result = func1->IsFloat32x4Func() && func2->IsFloat32x4Func();

result = result || (func1->IsFloat64x2Func() && func2->IsFloat64x2Func());

result = result || (func1->IsInt32x4Func() && func2->IsInt32x4Func());

return result;

}

AsmJsVarType GetTypeCheckVarType();

AsmJsVarType GetConstructorVarType();

OpCodeAsmJs GetOpcode() { return mOpCode; }

private:

virtual bool SupportsArgCall(ArgSlot argCount, AsmJsType\* args, AsmJsRetType& retType) override;

};

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

// Portions of this file are copyright 2014 Mozilla Foundation, available under the Apache 2.0 license.

//-------------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------------

// Copyright 2014 Mozilla Foundation

//

// Licensed under the Apache License, Version 2.0 (the "License");

// you may not use this file except in compliance with the License.

// You may obtain a copy of the License at

//

// http ://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing, software

// distributed under the License is distributed on an "AS IS" BASIS,

// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

// See the License for the specific language governing permissions and

// limitations under the License.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

bool ParserWrapper::ParseVarOrConstStatement( AsmJSParser &parser, ParseNode \*\*var )

{

Assert( parser );

\*var = nullptr;

ParseNode \*body = parser->sxFnc.pnodeBody;

if( body )

{

ParseNode\* lhs = GetBinaryLeft( body );

ParseNode\* rhs = GetBinaryRight( body );

if( rhs && rhs->nop == knopList )

{

AssertMsg( lhs->nop == knopStr, "this should be use asm" );

\*var = rhs;

return true;

}

}

return false;

}

bool ParserWrapper::IsDefinition( ParseNode \*arg )

{

//TODO, eliminate duplicates

return true;

}

ParseNode\* ParserWrapper::NextInList( ParseNode \*node )

{

Assert( node->nop == knopList );

return node->sxBin.pnode2;

}

ParseNode\* ParserWrapper::NextVar( ParseNode \*node )

{

return node->sxVar.pnodeNext;

}

ParseNode\* ParserWrapper::FunctionArgsList( ParseNode \*node, ArgSlot &numformals )

{

Assert( node->nop == knopFncDecl );

PnFnc func = node->sxFnc;

ParseNode\* first = func.pnodeArgs;

// throws OOM on uint16 overflow

for( ParseNode\* pnode = first; pnode; pnode = pnode->sxVar.pnodeNext, UInt16Math::Inc(numformals));

return first;

}

PropertyName ParserWrapper::VariableName( ParseNode \*node )

{

return node->name();

}

PropertyName ParserWrapper::FunctionName( ParseNode \*node )

{

if( node->nop == knopFncDecl )

{

PnFnc function = node->sxFnc;

if( function.pnodeName && function.pnodeName->nop == knopVarDecl )

{

return function.pnodeName->sxVar.pid;

}

}

return nullptr;

}

ParseNode \* ParserWrapper::GetVarDeclList( ParseNode \* pnode )

{

ParseNode\* varNode = pnode;

while (varNode->nop == knopList)

{

ParseNode \* var = GetBinaryLeft(varNode);

if (var->nop == knopVarDecl)

{

return var;

}

else if (var->nop == knopList)

{

var = GetBinaryLeft(var);

if (var->nop == knopVarDecl)

{

return var;

}

}

varNode = GetBinaryRight(varNode);

}

return nullptr;

}

void ParserWrapper::ReachEndVarDeclList( ParseNode\*\* outNode )

{

ParseNode\* pnode = \*outNode;

// moving down to the last var declaration

while( pnode->nop == knopList )

{

ParseNode\* var = GetBinaryLeft( pnode );

if (var->nop == knopVarDecl)

{

pnode = GetBinaryRight( pnode );

continue;

}

else if (var->nop == knopList)

{

var = GetBinaryLeft( var );

if (var->nop == knopVarDecl)

{

pnode = GetBinaryRight( pnode );

continue;

}

}

break;

}

\*outNode = pnode;

}

AsmJsCompilationException::AsmJsCompilationException( const wchar\_t\* \_msg, ... )

{

va\_list arglist;

va\_start( arglist, \_msg );

vswprintf\_s( msg\_, \_msg, arglist );

}

Var AsmJsChangeHeapBuffer(RecyclableObject \* function, CallInfo callInfo, ...)

{

PROBE\_STACK(function->GetScriptContext(), Js::Constants::MinStackDefault);

ARGUMENTS(args, callInfo);

ScriptContext\* scriptContext = function->GetScriptContext();

Assert(!(callInfo.Flags & CallFlags\_New));

if (args.Info.Count < 1 || !ArrayBuffer::Is(args[1]))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedArrayBufferObject);

}

ArrayBuffer\* newArrayBuffer = ArrayBuffer::FromVar(args[1]);

if (newArrayBuffer->IsDetached() || newArrayBuffer->GetByteLength() & 0xffffff || newArrayBuffer->GetByteLength() <= 0xffffff || newArrayBuffer->GetByteLength() > 0x80000000)

{

return JavascriptBoolean::ToVar(FALSE, scriptContext);

}

FrameDisplay\* frame = ((ScriptFunction\*)function)->GetEnvironment();

Var\* moduleArrayBuffer = (Var\*)frame->GetItem(0) + AsmJsModuleMemory::MemoryTableBeginOffset;

\*moduleArrayBuffer = newArrayBuffer;

return JavascriptBoolean::ToVar(TRUE, scriptContext);

}

#if \_M\_X64

// returns an array containing the size of each argument

uint \*GetArgsSizesArray(ScriptFunction\* func)

{

AsmJsFunctionInfo\* info = func->GetFunctionBody()->GetAsmJsFunctionInfo();

return info->GetArgsSizesArray();

}

int GetStackSizeForAsmJsUnboxing(ScriptFunction\* func)

{

AsmJsFunctionInfo\* info = func->GetFunctionBody()->GetAsmJsFunctionInfo();

int argSize = MachPtr;

for (ArgSlot i = 0; i < info->GetArgCount(); i++)

{

if (info->GetArgType(i).isSIMD())

{

argSize += sizeof(AsmJsSIMDValue);

}

else

{

argSize += MachPtr;

}

}

argSize = ::Math::Align<int32>(argSize, 16);

if (argSize < 32)

{

argSize = 32; // convention is to always allocate spill space for rcx,rdx,r8,r9

}

PROBE\_STACK\_CALL(func->GetScriptContext(), func, argSize);

return argSize;

}

void \* UnboxAsmJsArguments(ScriptFunction\* func, Var \* origArgs, char \* argDst, CallInfo callInfo)

{

void \* address = func->GetEntryPointInfo()->address;

Assert(address);

AsmJsFunctionInfo\* info = func->GetFunctionBody()->GetAsmJsFunctionInfo();

ScriptContext\* scriptContext = func->GetScriptContext();

AsmJsModuleInfo::EnsureHeapAttached(func);

uint actualArgCount = callInfo.Count - 1; // -1 for ScriptFunction

argDst = argDst + MachPtr; // add one first so as to skip the ScriptFunction argument

for (ArgSlot i = 0; i < info->GetArgCount(); i++)

{

if (info->GetArgType(i).isInt())

{

int32 intVal;

if (i < actualArgCount)

{

intVal = JavascriptMath::ToInt32(\*origArgs, scriptContext);

}

else

{

intVal = 0;

}

\*(int64\*)(argDst) = 0;

\*(int32\*)argDst = intVal;

argDst = argDst + MachPtr;

}

else if (info->GetArgType(i).isFloat())

{

float floatVal;

if (i < actualArgCount)

{

floatVal = (float)(JavascriptConversion::ToNumber(\*origArgs, scriptContext));

}

else

{

floatVal = (float)(JavascriptNumber::NaN);

}

\*(int64\*)(argDst) = 0;

\*(float\*)argDst = floatVal;

argDst = argDst + MachPtr;

}

else if (info->GetArgType(i).isDouble())

{

double doubleVal;

if (i < actualArgCount)

{

doubleVal = JavascriptConversion::ToNumber(\*origArgs, scriptContext);

}

else

{

doubleVal = JavascriptNumber::NaN;

}

\*(int64\*)(argDst) = 0;

\*(double\*)argDst = doubleVal;

argDst = argDst + MachPtr;

}

else if (info->GetArgType(i).isSIMD())

{

AsmJsVarType argType = info->GetArgType(i);

AsmJsSIMDValue simdVal = { 0, 0, 0, 0 };

// SIMD values are copied unaligned.

// SIMD values cannot be implicitly coerced from/to other types. If the SIMD parameter is missing (i.e. Undefined), we throw type error since there is not equivalent SIMD value to coerce to.

switch (argType.which())

{

case AsmJsType::Int32x4:

if (!JavascriptSIMDInt32x4::Is(\*origArgs))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_SimdInt32x4TypeMismatch, L"Int32x4");

}

simdVal = ((JavascriptSIMDInt32x4\*)(\*origArgs))->GetValue();

break;

case AsmJsType::Float32x4:

if (!JavascriptSIMDFloat32x4::Is(\*origArgs))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_SimdFloat32x4TypeMismatch, L"Float32x4");

}

simdVal = ((JavascriptSIMDFloat32x4\*)(\*origArgs))->GetValue();

break;

case AsmJsType::Float64x2:

if (!JavascriptSIMDFloat64x2::Is(\*origArgs))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_SimdFloat64x2TypeMismatch, L"Float64x2");

}

simdVal = ((JavascriptSIMDFloat64x2\*)(\*origArgs))->GetValue();

break;

default:

Assert(UNREACHED);

}

\*(AsmJsSIMDValue\*)argDst = simdVal;

argDst = argDst + sizeof(AsmJsSIMDValue);

}

++origArgs;

}

// for convenience, lets take the opportunity to return the asm.js entrypoint address

return address;

}

Var BoxAsmJsReturnValue(ScriptFunction\* func, int intRetVal, double doubleRetVal, float floatRetVal, \_\_m128 simdRetVal)

{

// ExternalEntryPoint doesn't know the return value, so it will send garbage for everything except actual return type

Var returnValue = nullptr;

// make call and convert primitive type back to Var

AsmJsFunctionInfo\* info = func->GetFunctionBody()->GetAsmJsFunctionInfo();

switch (info->GetReturnType().which())

{

case AsmJsRetType::Void:

returnValue = JavascriptOperators::OP\_LdUndef(func->GetScriptContext());

break;

case AsmJsRetType::Signed:{

returnValue = JavascriptNumber::ToVar(intRetVal, func->GetScriptContext());

break;

}

case AsmJsRetType::Double:{

returnValue = JavascriptNumber::New(doubleRetVal, func->GetScriptContext());

break;

}

case AsmJsRetType::Float:{

returnValue = JavascriptNumber::New(floatRetVal, func->GetScriptContext());

break;

}

case AsmJsRetType::Float32x4:

{

X86SIMDValue simdVal;

simdVal.m128\_value = simdRetVal;

returnValue = JavascriptSIMDFloat32x4::New(&X86SIMDValue::ToSIMDValue(simdVal), func->GetScriptContext());

break;

}

case AsmJsRetType::Int32x4:

{

X86SIMDValue simdVal;

simdVal.m128\_value = simdRetVal;

returnValue = JavascriptSIMDInt32x4::New(&X86SIMDValue::ToSIMDValue(simdVal), func->GetScriptContext());

break;

}

case AsmJsRetType::Float64x2:

{

X86SIMDValue simdVal;

simdVal.m128\_value = simdRetVal;

returnValue = JavascriptSIMDFloat64x2::New(&X86SIMDValue::ToSIMDValue(simdVal), func->GetScriptContext());

break;

}

default:

Assume(UNREACHED);

}

return returnValue;

}

#elif \_M\_IX86

Var AsmJsExternalEntryPoint(RecyclableObject\* entryObject, CallInfo callInfo, ...)

{

ARGUMENTS(args, callInfo);

ScriptFunction\* func = (ScriptFunction\*)entryObject;

FunctionBody\* body = func->GetFunctionBody();

AsmJsFunctionInfo\* info = body->GetAsmJsFunctionInfo();

ScriptContext\* scriptContext = func->GetScriptContext();

const uint argInCount = callInfo.Count - 1;

int argSize = info->GetArgByteSize();

char\* dst;

Var returnValue = 0;

AsmJsModuleInfo::EnsureHeapAttached(func);

argSize = ::Math::Align<int32>(argSize, 8);

// Allocate stack space for args

\_\_asm

{

sub esp, argSize

mov dst, esp

};

// Unbox Var to primitive type

{

int32 intVal; double doubleVal; float floatVal;

for (ArgSlot i = 0; i < info->GetArgCount(); i++)

{

if (info->GetArgType(i).isInt())

{

if (i < argInCount)

{

intVal = JavascriptMath::ToInt32(args.Values[i + 1], scriptContext);

}

else

{

intVal = 0;

}

\*(int32\*)dst = intVal;

dst += sizeof(int32);

}

else if (info->GetArgType(i).isFloat())

{

if (i < argInCount)

{

floatVal = (float)(JavascriptConversion::ToNumber(args.Values[i + 1], scriptContext));

}

else

{

floatVal = (float)(JavascriptNumber::NaN);

}

\*(float\*)dst = floatVal;

dst += sizeof(float);

}

else if (info->GetArgType(i).isDouble())

{

if (i < argInCount)

{

doubleVal = JavascriptConversion::ToNumber(args.Values[i + 1], scriptContext);

}

else

{

doubleVal = JavascriptNumber::NaN;

}

\*(double\*)dst = doubleVal;

dst += sizeof(double);

}

else if (info->GetArgType(i).isSIMD())

{

AsmJsVarType argType = info->GetArgType(i);

AsmJsSIMDValue simdVal;

// SIMD values are copied unaligned.

// SIMD values cannot be implicitly coerced from/to other types. If the SIMD parameter is missing (i.e. Undefined), we throw type error since there is not equivalent SIMD value to coerce to.

switch (argType.which())

{

case AsmJsType::Int32x4:

if (i >= argInCount || !JavascriptSIMDInt32x4::Is(args.Values[i + 1]))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_SimdInt32x4TypeMismatch, L"Int32x4");

}

simdVal = ((JavascriptSIMDInt32x4\*)(args.Values[i + 1]))->GetValue();

break;

case AsmJsType::Float32x4:

if (i >= argInCount || !JavascriptSIMDFloat32x4::Is(args.Values[i + 1]))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_SimdFloat32x4TypeMismatch, L"Float32x4");

}

simdVal = ((JavascriptSIMDFloat32x4\*)(args.Values[i + 1]))->GetValue();

break;

case AsmJsType::Float64x2:

if (i >= argInCount || !JavascriptSIMDFloat64x2::Is(args.Values[i + 1]))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_SimdFloat64x2TypeMismatch, L"Float64x2");

}

simdVal = ((JavascriptSIMDFloat64x2\*)(args.Values[i + 1]))->GetValue();

break;

default:

Assert(UNREACHED);

}

\*(AsmJsSIMDValue\*)dst = simdVal;

dst += sizeof(AsmJsSIMDValue);

}

else

{

AssertMsg(UNREACHED, "Invalid function arg type.");

}

}

}

const void \* asmJSEntryPoint = func->GetEntryPointInfo()->address;

// make call and convert primitive type back to Var

switch (info->GetReturnType().which())

{

case AsmJsRetType::Void:

\_\_asm

{

mov ecx, asmJSEntryPoint

#ifdef \_CONTROL\_FLOW\_GUARD

call[\_\_guard\_check\_icall\_fptr]

#endif

push func

call ecx

}

returnValue = JavascriptOperators::OP\_LdUndef(func->GetScriptContext());

break;

case AsmJsRetType::Signed:{

int32 ival = 0;

\_\_asm

{

mov ecx, asmJSEntryPoint

#ifdef \_CONTROL\_FLOW\_GUARD

call[\_\_guard\_check\_icall\_fptr]

#endif

push func

call ecx

mov ival, eax

}

returnValue = JavascriptNumber::ToVar(ival, func->GetScriptContext());

break;

}

case AsmJsRetType::Double:{

double dval = 0;

\_\_asm

{

mov ecx, asmJSEntryPoint

#ifdef \_CONTROL\_FLOW\_GUARD

call[\_\_guard\_check\_icall\_fptr]

#endif

push func

call ecx

movsd dval, xmm0

}

returnValue = JavascriptNumber::New(dval, func->GetScriptContext());

break;

}

case AsmJsRetType::Float:{

float fval = 0;

\_\_asm

{

mov ecx, asmJSEntryPoint

#ifdef \_CONTROL\_FLOW\_GUARD

call[\_\_guard\_check\_icall\_fptr]

#endif

push func

call ecx

movss fval, xmm0

}

returnValue = JavascriptNumber::New((double)fval, func->GetScriptContext());

break;

}

case AsmJsRetType::Int32x4:

AsmJsSIMDValue simdVal;

simdVal.Zero();

\_\_asm

{

mov ecx, asmJSEntryPoint

#ifdef \_CONTROL\_FLOW\_GUARD

call[\_\_guard\_check\_icall\_fptr]

#endif

push func

call ecx

movups simdVal, xmm0

}

returnValue = JavascriptSIMDInt32x4::New(&simdVal, func->GetScriptContext());

break;

case AsmJsRetType::Float32x4:

simdVal.Zero();

\_\_asm

{

mov ecx, asmJSEntryPoint

#ifdef \_CONTROL\_FLOW\_GUARD

call[\_\_guard\_check\_icall\_fptr]

#endif

push func

call ecx

movups simdVal, xmm0

}

returnValue = JavascriptSIMDFloat32x4::New(&simdVal, func->GetScriptContext());

break;

case AsmJsRetType::Float64x2:

simdVal.Zero();

\_\_asm

{

mov ecx, asmJSEntryPoint

#ifdef \_CONTROL\_FLOW\_GUARD

call[\_\_guard\_check\_icall\_fptr]

#endif

push func

call ecx

movups simdVal, xmm0

}

returnValue = JavascriptSIMDFloat64x2::New(&simdVal, func->GetScriptContext());

break;

default:

Assume(UNREACHED);

}

return returnValue;

}

#endif

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

// Portions of this file are copyright 2014 Mozilla Foundation, available under the Apache 2.0 license.

//-------------------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------------------

// Copyright 2014 Mozilla Foundation

//

// Licensed under the Apache License, Version 2.0 (the "License");

// you may not use this file except in compliance with the License.

// You may obtain a copy of the License at

//

// http ://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing, software

// distributed under the License is distributed on an "AS IS" BASIS,

// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

// See the License for the specific language governing permissions and

// limitations under the License.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifndef TEMP\_DISABLE\_ASMJS

// Removed code from original location, if the expression is true, check if extra code needed

#define MaybeTodo( expr ) AssertMsg( !(expr), "Unhandled scenario in asm.js" )

namespace Js {

static const int DOUBLE\_SLOTS\_SPACE = (sizeof(double) / sizeof(Var)); // 2 in x86 and 1 in x64

static const double FLOAT\_SLOTS\_SPACE = (sizeof(float) / (double)sizeof(Var)); // 1 in x86 and 0.5 in x64

static const double INT\_SLOTS\_SPACE = ( sizeof( int ) / (double)sizeof( Var ) ); // 1 in x86 and 0.5 in x64

static const double SIMD\_SLOTS\_SPACE = (sizeof(SIMDValue) / sizeof(Var)); // 4 in x86 and 2 in x64

Var AsmJsChangeHeapBuffer(RecyclableObject \* function, CallInfo callInfo, ...);

Var AsmJsExternalEntryPoint(Js::RecyclableObject\* entryObject, Js::CallInfo callInfo, ...);

#if \_M\_X64

int GetStackSizeForAsmJsUnboxing(ScriptFunction\* func);

void \* UnboxAsmJsArguments(ScriptFunction\* func, Var \* origArgs, char \* argDst, CallInfo callInfo);

Var BoxAsmJsReturnValue(ScriptFunction\* func, int intRetVal, double doubleRetVal, float floatRetVal);

#endif

class AsmJsCompilationException

{

wchar\_t msg\_[256];

public:

AsmJsCompilationException( const wchar\_t\* \_msg, ... );

inline wchar\_t\* msg() { return msg\_; }

};

class ParserWrapper

{

public:

static PropertyName FunctionName( ParseNode \*node );

static PropertyName VariableName( ParseNode \*node );

static ParseNode\* FunctionArgsList( ParseNode \*node, ArgSlot &numformals );

static ParseNode\* NextVar( ParseNode \*node );

static ParseNode\* NextInList( ParseNode \*node );

static inline ParseNode \*GetListHead( ParseNode \*node );

static inline bool IsNameDeclaration(ParseNode \*node);

static inline bool IsUInt(ParseNode \*node);

static inline uint GetUInt(ParseNode \*node);

static inline bool IsNegativeZero(ParseNode\* node);

static inline bool IsMinInt(ParseNode \*node){ return node && node->nop == knopFlt && node->sxFlt.maybeInt && node->sxFlt.dbl == -2147483648.0; };

static inline bool IsUnsigned(ParseNode \*node){ return node && node->nop == knopFlt && node->sxFlt.maybeInt && (((uint32)node->sxFlt.dbl) >> 31); };

static bool IsDefinition( ParseNode \*arg );

static bool ParseVarOrConstStatement( AsmJSParser &parser, ParseNode \*\*var );

static inline bool IsNumericLiteral(ParseNode\* node) { return node && (node->nop == knopInt || node->nop == knopFlt); }

static inline bool IsFroundNumericLiteral(ParseNode\* node) { return node && (IsNumericLiteral(node) || IsNegativeZero(node)); }

static inline ParseNode\* GetUnaryNode( ParseNode\* node ){Assert(IsNodeUnary(node));return node->sxUni.pnode1;}

static inline ParseNode\* GetBinaryLeft( ParseNode\* node ){Assert(IsNodeBinary(node));return node->sxBin.pnode1;}

static inline ParseNode\* GetBinaryRight( ParseNode\* node ){Assert(IsNodeBinary(node));return node->sxBin.pnode2;}

static inline ParseNode\* DotBase( ParseNode \*node );

static inline bool IsDotMember( ParseNode \*node );

static inline PropertyName DotMember( ParseNode \*node );

// Get the VarDecl from the node or nullptr if unable to find

static ParseNode\* GetVarDeclList(ParseNode\* node);

// Goes through the nodes until the end of the list of VarDecl

static void ReachEndVarDeclList( ParseNode\*\* node );

// nop utils

static inline bool IsNodeBinary (ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & (fnopBin|fnopBinList)); }

static inline bool IsNodeUnary (ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopUni ); }

static inline bool IsNodeConst (ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopConst ); }

static inline bool IsNodeLeaf (ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopLeaf ); }

static inline bool IsNodeRelational(ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopRel ); }

static inline bool IsNodeAssignment(ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopAsg ); }

static inline bool IsNodeBreak (ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopBreak ); }

static inline bool IsNodeContinue (ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopContinue ); }

static inline bool IsNodeCleanUp (ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopCleanup ); }

static inline bool IsNodeJump (ParseNode\* pnode){ return pnode && !!(ParseNode::Grfnop(pnode->nop) & fnopJump ); }

static inline bool IsNodeExpression(ParseNode\* pnode){ return pnode && !(ParseNode::Grfnop(pnode->nop) & fnopNotExprStmt); }

};

bool ParserWrapper::IsNameDeclaration( ParseNode \*node )

{

return node->nop == knopName || node->nop == knopStr;

}

bool ParserWrapper::IsNegativeZero(ParseNode \*node)

{

return node && ((node->nop == knopFlt && JavascriptNumber::IsNegZero(node->sxFlt.dbl)) ||

(node->nop == knopNeg && node->sxUni.pnode1->nop == knopInt && node->sxUni.pnode1->sxInt.lw == 0));

}

bool ParserWrapper::IsUInt( ParseNode \*node )

{

return node->nop == knopInt || IsUnsigned(node);

}

uint ParserWrapper::GetUInt( ParseNode \*node )

{

Assert( IsUInt( node ) );

if( node->nop == knopInt )

{

return (uint)node->sxInt.lw;

}

Assert( node->nop == knopFlt );

return (uint)node->sxFlt.dbl;

}

bool ParserWrapper::IsDotMember( ParseNode \*node )

{

return node && (node->nop == knopDot || node->nop == knopIndex);

}

PropertyName ParserWrapper::DotMember( ParseNode \*node )

{

Assert( IsDotMember(node) );

if( IsNameDeclaration( GetBinaryRight( node ) ) )

{

return GetBinaryRight( node )->name();

}

return nullptr;

}

ParseNode\* ParserWrapper::DotBase( ParseNode \*node )

{

Assert( IsDotMember( node ) );

return GetBinaryLeft( node );

}

ParseNode \* ParserWrapper::GetListHead( ParseNode \*node )

{

Assert( node->nop == knopList );

return node->sxBin.pnode1;

}

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if DBG

#include "Types\PathTypeHandler.h"

#endif

namespace Js

{

void CacheOperators::CachePropertyRead(

Var startingObject,

RecyclableObject \* objectWithProperty,

const bool isRoot,

PropertyId propertyId,

const bool isMissing,

PropertyValueInfo\* info,

ScriptContext \* requestContext)

{

if (!CanCachePropertyRead(info, objectWithProperty, requestContext))

{

return;

}

AssertMsg(!(info->GetFlags() & InlineCacheGetterFlag), "We cache getters only in DictionaryTypeHandler::GetProperty() before they were executed");

AssertMsg((info->GetFlags() & InlineCacheSetterFlag) == 0, "invalid setter flag in CachePropertyRead");

if (info->GetInstance() != objectWithProperty) // We can't cache if slot owner is not the object

{

AssertMsg(info->IsNoCache() || objectWithProperty->GetPropertyIndex(propertyId) == Constants::NoSlot, "Missed updating PropertyValueInfo?");

return;

}

PropertyIndex propertyIndex = info->GetPropertyIndex();

Assert(propertyIndex == objectWithProperty->GetPropertyIndex(propertyId) ||

(RootObjectBase::Is(objectWithProperty) && propertyIndex == RootObjectBase::FromVar(objectWithProperty)->GetRootPropertyIndex(propertyId)));

Assert(DynamicType::Is(objectWithProperty->GetTypeId()));

// We are populating a cache guarded by the instance's type (not the type of the object with property somewhere in the prototype chain),

// so we only care if the instance's property (if any) is fixed.

Assert(info->IsNoCache() || !info->IsStoreFieldCacheEnabled() || info->GetInstance() != objectWithProperty || !objectWithProperty->IsFixedProperty(propertyId));

PropertyIndex slotIndex;

bool isInlineSlot;

DynamicObject::FromVar(objectWithProperty)->GetDynamicType()->GetTypeHandler()->PropertyIndexToInlineOrAuxSlotIndex(propertyIndex, &slotIndex, &isInlineSlot);

const bool isProto = objectWithProperty != startingObject;

if(!isProto)

{

Assert(info->IsWritable() == (objectWithProperty->IsWritable(propertyId) ? true : false));

// Because StFld and LdFld caches aren't shared, we can safely cache for LdFld operations even if the property isn't writable.

}

else if(

PropertyValueInfo::PrototypeCacheDisabled((PropertyValueInfo\*)info) ||

!RecyclableObject::Is(startingObject) ||

RecyclableObject::FromVar(startingObject)->GetScriptContext() != requestContext)

{

// Don't need to cache if the beginning property is number etc.

return;

}

#ifdef TELEMETRY\_AddToCache

// For performance reasons, only execute this code in interpreted mode, not JIT.

// This method only returns true in interpreted mode and can be used to detect interpreted mode.

if (info->AllowResizingPolymorphicInlineCache())

{

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

requestContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(objectWithProperty, propertyId, nullptr, !isMissing);

}

}

#endif

Cache<false, true, true>(

isProto,

DynamicObject::FromVar(objectWithProperty),

isRoot,

RecyclableObject::FromVar(startingObject)->GetType(),

nullptr,

propertyId,

slotIndex,

isInlineSlot,

isMissing,

0,

info,

requestContext);

}

void CacheOperators::CachePropertyReadForGetter(

PropertyValueInfo \*info,

Var originalInstance,

JsUtil::CharacterBuffer<WCHAR> const& propertyName,

ScriptContext\* requestContext)

{

PropertyRecord const\* propertyRecord;

requestContext->GetOrAddPropertyRecord(propertyName, &propertyRecord);

CachePropertyReadForGetter(info, originalInstance, propertyRecord->GetPropertyId(), requestContext);

}

void CacheOperators::CachePropertyReadForGetter(

PropertyValueInfo \*info,

Var originalInstance,

PropertyId propertyId,

ScriptContext\* requestContext)

{

if (!info || !CacheOperators::CanCachePropertyRead(info, info->GetInstance(), requestContext))

{

return;

}

Assert(RecyclableObject::Is(originalInstance));

Assert(DynamicType::Is(info->GetInstance()->GetTypeId()));

PropertyIndex slotIndex;

bool isInlineSlot;

DynamicObject::FromVar(info->GetInstance())->GetDynamicType()->GetTypeHandler()->PropertyIndexToInlineOrAuxSlotIndex(info->GetPropertyIndex(), &slotIndex, &isInlineSlot);

const bool isProto = info->GetInstance() != originalInstance;

if(isProto &&

(

!RecyclableObject::Is(originalInstance) ||

RecyclableObject::FromVar(originalInstance)->GetScriptContext() != requestContext

))

{

// Don't need to cache if the beginning property is number etc.

return;

}

#ifdef TELEMETRY\_AddToCache

if (info->AllowResizingPolymorphicInlineCache()) // If in interpreted mode, not JIT.

{

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

requestContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(info->GetInstance(), propertyId, nullptr, true /\* true, because if a getter is being evaluated then the property does exist. \*/);

}

}

#endif

Cache<true, true, false>(

isProto,

DynamicObject::FromVar(info->GetInstance()),

false,

RecyclableObject::FromVar(originalInstance)->GetType(),

nullptr,

propertyId,

slotIndex,

isInlineSlot,

false,

0,

info,

requestContext);

}

void CacheOperators::CachePropertyWrite(

RecyclableObject \* object,

const bool isRoot,

Type\* typeWithoutProperty,

PropertyId propertyId,

PropertyValueInfo\* info,

ScriptContext \* requestContext)

{

Assert(typeWithoutProperty != nullptr);

if (!CacheOperators::CanCachePropertyWrite(info, object, requestContext))

{

return;

}

BOOL isSetter = (info->GetFlags() == InlineCacheSetterFlag);

if (info->GetInstance() != object) // We can't cache if slot owner is not the object

{

if (!isSetter)

{

AssertMsg(info->IsNoCache() || object->GetPropertyIndex(propertyId) == Constants::NoSlot, "Missed updating PropertyValueInfo?");

return;

}

}

PropertyIndex propertyIndex = info->GetPropertyIndex();

if (isSetter)

{

if (propertyIndex & 0xf000)

{

return;

}

}

else

{

if (propertyIndex == Constants::NoSlot)

{

return;

}

}

Assert((!isRoot && propertyIndex == object->GetPropertyIndex(propertyId)) || isSetter ||

(isRoot && propertyIndex == RootObjectBase::FromVar(object)->GetRootPropertyIndex(propertyId)));

Assert(DynamicType::Is(object->GetTypeId()));

AssertMsg((info->GetFlags() & InlineCacheGetterFlag) == 0, "invalid getter for CachePropertyWrite");

RecyclableObject\* instance = info->GetInstance();

PropertyIndex slotIndex;

bool isInlineSlot;

DynamicObject::FromVar(instance)->GetDynamicType()->GetTypeHandler()->PropertyIndexToInlineOrAuxSlotIndex(propertyIndex, &slotIndex, &isInlineSlot);

if (!isSetter)

{

AssertMsg(instance == object, "invalid instance for non setter");

Assert(DynamicType::Is(typeWithoutProperty->GetTypeId()));

Assert(info->IsNoCache() || !info->IsStoreFieldCacheEnabled() || object->CanStorePropertyValueDirectly(propertyId, isRoot));

Assert(info->IsWritable());

DynamicType\* newType = (DynamicType\*)object->GetType();

DynamicType\* oldType = (DynamicType\*)typeWithoutProperty;

Assert(newType);

int requiredAuxSlotCapacity;

// Don't cache property adds for types that aren't (yet) shared. We must go down the slow path to force type sharing

// and invalidate any potential fixed fields this type may have.

// Don't cache property adds to prototypes, so we don't have to check if the object is a prototype on the fast path.

if (newType != oldType && newType->GetIsShared() && newType->GetTypeHandler()->IsPathTypeHandler() && (!oldType->GetTypeHandler()->GetIsPrototype()))

{

DynamicTypeHandler\* oldTypeHandler = oldType->GetTypeHandler();

DynamicTypeHandler\* newTypeHandler = newType->GetTypeHandler();

// the newType is a path-type so the old one should be too:

Assert(oldTypeHandler->IsPathTypeHandler());

int oldCapacity = oldTypeHandler->GetSlotCapacity();

int newCapacity = newTypeHandler->GetSlotCapacity();

int newInlineCapacity = newTypeHandler->GetInlineSlotCapacity();

// We are adding only one property here. If some other properties were added as a side effect on the slow path

// we should never cache the type transition, as the other property slots will not be populated by the fast path.

AssertMsg(((PathTypeHandlerBase \*)oldTypeHandler)->GetPropertyCount() + 1 == ((PathTypeHandlerBase \*)newTypeHandler)->GetPropertyCount(),

"Don't cache type transitions that add multiple properties.");

// InlineCache::TrySetProperty assumes the following invariants to decide if and how to adjust auxiliary slot capacity.

AssertMsg(DynamicObject::IsTypeHandlerCompatibleForObjectHeaderInlining(oldTypeHandler, newTypeHandler),

"TypeHandler should be compatible for transition.");

// If the slot is inlined then we should never need to adjust auxiliary slot capacity.

Assert(!isInlineSlot || oldCapacity == newCapacity || newCapacity <= newInlineCapacity);

// If the slot is not inlined then the new type must have some auxiliary slots.

Assert(isInlineSlot || newCapacity > newInlineCapacity);

// If the slot is not inlined and the property being added exceeds the old type's slot capacity, then slotIndex corresponds

// to the number of occupied auxiliary slots (i.e. the old type's auxiliary slot capacity).

// If the object is optimized for <=2 properties, then slotIndex should be same as the oldCapacity(as there is no inlineSlots in the new typeHandler).

Assert(

isInlineSlot ||

oldCapacity == newCapacity ||

slotIndex == oldCapacity - oldTypeHandler->GetInlineSlotCapacity() ||

(

oldTypeHandler->IsObjectHeaderInlinedTypeHandler() &&

newInlineCapacity ==

oldTypeHandler->GetInlineSlotCapacity() -

DynamicTypeHandler::GetObjectHeaderInlinableSlotCapacity() &&

slotIndex == DynamicTypeHandler::GetObjectHeaderInlinableSlotCapacity()

));

requiredAuxSlotCapacity = (!isInlineSlot && oldCapacity < newCapacity) ? newCapacity - newInlineCapacity : 0;

// Required auxiliary slot capacity must fit in the available inline cache bits.

Assert(requiredAuxSlotCapacity < (0x01 << InlineCache::RequiredAuxSlotCapacityBitCount));

}

else

{

typeWithoutProperty = nullptr;

requiredAuxSlotCapacity = 0;

}

Cache<false, false, true>(

false,

DynamicObject::FromVar(object),

isRoot,

object->GetType(),

typeWithoutProperty,

propertyId,

slotIndex,

isInlineSlot,

false,

requiredAuxSlotCapacity,

info,

requestContext);

return;

}

const bool isProto = instance != object;

if(isProto && instance->GetScriptContext() != requestContext)

{

// Don't cache if object and prototype are from different context

return;

}

Cache<true, false, false>(

isProto,

DynamicObject::FromVar(instance),

false,

object->GetType(),

nullptr,

propertyId,

slotIndex,

isInlineSlot,

false,

0,

info,

requestContext);

}

bool CacheOperators::CanCachePropertyRead(const PropertyValueInfo \*info, RecyclableObject \* object, ScriptContext \* requestContext)

{

return

info &&

info->GetPropertyIndex() != Constants::NoSlot &&

(info->GetInlineCache() || info->GetPolymorphicInlineCache() || info->GetFunctionBody()) &&

CanCachePropertyRead(object, requestContext);

}

bool CacheOperators::CanCachePropertyRead(RecyclableObject \* object, ScriptContext \* requestContext)

{

return object->GetScriptContext() == requestContext && !PHASE\_OFF1(InlineCachePhase);

}

bool CacheOperators::CanCachePropertyWrite(const PropertyValueInfo \*info, RecyclableObject \* object, ScriptContext \* requestContext)

{

return

info &&

(info->GetInlineCache() || info->GetPolymorphicInlineCache() || info->GetFunctionBody()) &&

CanCachePropertyWrite(object, requestContext);

}

bool CacheOperators::CanCachePropertyWrite(RecyclableObject \* object, ScriptContext \* requestContext)

{

return object->GetScriptContext() == requestContext && DynamicType::Is(object->GetTypeId()) && !PHASE\_OFF1(InlineCachePhase);

}

#if DBG\_DUMP

void CacheOperators::TraceCache(InlineCache \* inlineCache, const wchar\_t \* methodName, PropertyId propertyId, ScriptContext \* requestContext, RecyclableObject \* object)

{

TraceCacheCommon(methodName, propertyId, requestContext, object);

if(inlineCache)

{

Output::Print(L"Inline Cache: \n ");

inlineCache->Dump();

}

Output::Print(L"\n");

Output::Flush();

}

void CacheOperators::TraceCache(PolymorphicInlineCache \* polymorphicInlineCache, const wchar\_t \* methodName, PropertyId propertyId, ScriptContext \* requestContext, RecyclableObject \* object)

{

TraceCacheCommon(methodName, propertyId, requestContext, object);

Output::Print(L"Polymorphic Inline Cache, size = %d :\n", polymorphicInlineCache->GetSize());

polymorphicInlineCache->Dump();

Output::Flush();

}

void CacheOperators::TraceCacheCommon(const wchar\_t \* methodName, PropertyId propertyId, ScriptContext \* requestContext, RecyclableObject \* object)

{

if(object)

{

JavascriptFunction\* caller;

const WCHAR\* callerName = NULL;

uint lineNumber = 0;

uint columnNumber = 0;

if(JavascriptStackWalker::GetCaller(&caller, requestContext))

{

FunctionBody \*functionBody = caller->GetFunctionBody();

callerName = functionBody->GetExternalDisplayName();

lineNumber = functionBody->GetLineNumber();

columnNumber = functionBody->GetColumnNumber();

}

Output::Print(L"%s, %s, %s(%d:%d), InType: 0x%X ",

methodName,

requestContext->GetPropertyName(propertyId)->GetBuffer(),

callerName,

lineNumber,

columnNumber,

object->GetType());

}

}

#endif

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

class CacheOperators

{

public:

static void CachePropertyRead(Var startingObject, RecyclableObject \* objectWithProperty, const bool isRoot, PropertyId propertyId, const bool isMissing, PropertyValueInfo\* info, ScriptContext \* requestContext);

static void CachePropertyReadForGetter(PropertyValueInfo \*info, Var originalInstance, JsUtil::CharacterBuffer<WCHAR> const& propertyName, ScriptContext\* requestContext);

static void CachePropertyReadForGetter(PropertyValueInfo \*info, Var originalInstance, PropertyId propertyId, ScriptContext\* requestContext);

static void CachePropertyWrite(RecyclableObject \* object, const bool isRoot, Type\* typeWithoutProperty, PropertyId propertyId, PropertyValueInfo\* info, ScriptContext \* requestContext);

template<

bool IsAccessor,

bool IsRead,

bool IncludeTypePropertyCache>

static void Cache(const bool isProto, DynamicObject \*const objectWithProperty, const bool isRoot, Type \*const type, Type \*const typeWithoutProperty, const PropertyId propertyId, const PropertyIndex propertyIndex, const bool isInlineSlot, const bool isMissing, const int requiredAuxSlotCapacity, const PropertyValueInfo \*const info, ScriptContext \*const requestContext);

template<

bool CheckLocal,

bool CheckProto,

bool CheckAccessor,

bool CheckMissing,

bool CheckPolymorphicInlineCache,

bool CheckTypePropertyCache,

bool IsInlineCacheAvailable,

bool IsPolymorphicInlineCacheAvailable,

bool ReturnOperationInfo>

static bool TryGetProperty(Var const instance, const bool isRoot, RecyclableObject \*const object, const PropertyId propertyId, Var \*const propertyValue, ScriptContext \*const requestContext, PropertyCacheOperationInfo \* operationInfo, PropertyValueInfo \*const propertyValueInfo);

template<

bool CheckLocal,

bool CheckLocalTypeWithoutProperty,

bool CheckAccessor,

bool CheckPolymorphicInlineCache,

bool CheckTypePropertyCache,

bool IsInlineCacheAvailable,

bool IsPolymorphicInlineCacheAvailable,

bool ReturnOperationInfo>

static bool TrySetProperty(RecyclableObject \*const object, const bool isRoot, const PropertyId propertyId, Var propertyValue, ScriptContext \*const requestContext, const PropertyOperationFlags propertyOperationFlags, PropertyCacheOperationInfo \* operationInfo, PropertyValueInfo \*const propertyValueInfo);

template<

bool IsInlineCacheAvailable,

bool IsPolymorphicInlineCacheAvailable>

static void PretendTryGetProperty(Type \*const type, PropertyCacheOperationInfo \* operationInfo, PropertyValueInfo \*const propertyValueInfo);

template<

bool IsInlineCacheAvailable,

bool IsPolymorphicInlineCacheAvailable>

static void PretendTrySetProperty(Type \*const type, Type \*const oldType, PropertyCacheOperationInfo \* operationInfo, PropertyValueInfo \*const propertyValueInfo);

#if DBG\_DUMP

static void TraceCache(InlineCache \* inlineCache, const wchar\_t \* methodName, PropertyId propertyId, ScriptContext \* requestContext, RecyclableObject \* object);

static void TraceCache(PolymorphicInlineCache \* polymorphicInlineCache, const wchar\_t \* methodName, PropertyId propertyId, ScriptContext \* requestContext, RecyclableObject \* object);

#endif

private:

static bool CanCachePropertyRead(const PropertyValueInfo \*info, RecyclableObject \* object, ScriptContext \* requestContext);

static bool CanCachePropertyRead(RecyclableObject \* object, ScriptContext \* requestContext);

static bool CanCachePropertyWrite(const PropertyValueInfo \*info, RecyclableObject \* object, ScriptContext \* requestContext);

static bool CanCachePropertyWrite(RecyclableObject \* object, ScriptContext \* requestContext);

#if DBG\_DUMP

static void TraceCacheCommon(const wchar\_t \* methodName, PropertyId propertyId, ScriptContext \* requestContext, RecyclableObject \* object);

#endif

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

template<

bool CheckLocal,

bool CheckProto,

bool CheckAccessor,

bool CheckMissing,

bool CheckPolymorphicInlineCache,

bool CheckTypePropertyCache,

bool IsInlineCacheAvailable,

bool IsPolymorphicInlineCacheAvailable,

bool ReturnOperationInfo>

\_\_inline bool CacheOperators::TryGetProperty(

Var const instance,

const bool isRoot,

RecyclableObject \*const object,

const PropertyId propertyId,

Var \*const propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \* operationInfo,

PropertyValueInfo \*const propertyValueInfo)

{

CompileAssert(IsInlineCacheAvailable || IsPolymorphicInlineCacheAvailable);

Assert(!CheckTypePropertyCache || !isRoot);

Assert(propertyValueInfo);

Assert(IsInlineCacheAvailable == !!propertyValueInfo->GetInlineCache());

Assert(IsPolymorphicInlineCacheAvailable == !!propertyValueInfo->GetPolymorphicInlineCache());

Assert(!ReturnOperationInfo || operationInfo);

if(CheckLocal || CheckProto || CheckAccessor)

{

InlineCache \*const inlineCache = IsInlineCacheAvailable ? propertyValueInfo->GetInlineCache() : nullptr;

if(IsInlineCacheAvailable)

{

if (inlineCache->TryGetProperty<CheckLocal, CheckProto, CheckAccessor, CheckMissing, ReturnOperationInfo>(

instance,

object,

propertyId,

propertyValue,

requestContext,

operationInfo))

{

return true;

}

if(ReturnOperationInfo)

{

operationInfo->isPolymorphic = inlineCache->HasDifferentType(object->GetType());

}

}

else if(ReturnOperationInfo)

{

operationInfo->isPolymorphic = true;

}

if(CheckPolymorphicInlineCache)

{

Assert(IsPolymorphicInlineCacheAvailable || propertyValueInfo->GetFunctionBody());

PolymorphicInlineCache \*const polymorphicInlineCache =

IsPolymorphicInlineCacheAvailable

? propertyValueInfo->GetPolymorphicInlineCache()

: propertyValueInfo->GetFunctionBody()->GetPolymorphicInlineCache(

propertyValueInfo->GetInlineCacheIndex());

if ((IsPolymorphicInlineCacheAvailable || polymorphicInlineCache) &&

polymorphicInlineCache->TryGetProperty<

CheckLocal,

CheckProto,

CheckAccessor,

CheckMissing,

IsInlineCacheAvailable,

ReturnOperationInfo

>(

instance,

object,

propertyId,

propertyValue,

requestContext,

operationInfo,

inlineCache

))

{

return true;

}

}

}

if(!CheckTypePropertyCache)

{

return false;

}

TypePropertyCache \*const typePropertyCache = object->GetType()->GetPropertyCache();

if(!typePropertyCache ||

!typePropertyCache->TryGetProperty(

CheckMissing,

object,

propertyId,

propertyValue,

requestContext,

ReturnOperationInfo ? operationInfo : nullptr,

propertyValueInfo))

{

return false;

}

if(!ReturnOperationInfo || operationInfo->cacheType == CacheType\_TypeProperty)

{

return true;

}

// The property access was cached in an inline cache. Get the proper property operation info.

PretendTryGetProperty<IsInlineCacheAvailable, IsPolymorphicInlineCacheAvailable>(

object->GetType(),

operationInfo,

propertyValueInfo);

return true;

}

template<

bool CheckLocal,

bool CheckLocalTypeWithoutProperty,

bool CheckAccessor,

bool CheckPolymorphicInlineCache,

bool CheckTypePropertyCache,

bool IsInlineCacheAvailable,

bool IsPolymorphicInlineCacheAvailable,

bool ReturnOperationInfo>

\_\_inline bool CacheOperators::TrySetProperty(

RecyclableObject \*const object,

const bool isRoot,

const PropertyId propertyId,

Var propertyValue,

ScriptContext \*const requestContext,

const PropertyOperationFlags propertyOperationFlags,

PropertyCacheOperationInfo \* operationInfo,

PropertyValueInfo \*const propertyValueInfo)

{

CompileAssert(IsInlineCacheAvailable || IsPolymorphicInlineCacheAvailable);

Assert(!CheckTypePropertyCache || !isRoot);

Assert(propertyValueInfo);

Assert(IsInlineCacheAvailable == !!propertyValueInfo->GetInlineCache());

Assert(IsPolymorphicInlineCacheAvailable == !!propertyValueInfo->GetPolymorphicInlineCache());

Assert(!ReturnOperationInfo || operationInfo);

if(CheckLocal || CheckLocalTypeWithoutProperty || CheckAccessor)

{

InlineCache \*const inlineCache = IsInlineCacheAvailable ? propertyValueInfo->GetInlineCache() : nullptr;

if(IsInlineCacheAvailable)

{

if (inlineCache->TrySetProperty<CheckLocal, CheckLocalTypeWithoutProperty, CheckAccessor, ReturnOperationInfo>(

object,

propertyId,

propertyValue,

requestContext,

operationInfo,

propertyOperationFlags))

{

return true;

}

if(ReturnOperationInfo)

{

operationInfo->isPolymorphic = inlineCache->HasDifferentType(object->GetType());

}

}

else if(ReturnOperationInfo)

{

operationInfo->isPolymorphic = true;

}

if(CheckPolymorphicInlineCache)

{

Assert(IsPolymorphicInlineCacheAvailable || propertyValueInfo->GetFunctionBody());

PolymorphicInlineCache \*const polymorphicInlineCache =

IsPolymorphicInlineCacheAvailable

? propertyValueInfo->GetPolymorphicInlineCache()

: propertyValueInfo->GetFunctionBody()->GetPolymorphicInlineCache(

propertyValueInfo->GetInlineCacheIndex());

if ((IsPolymorphicInlineCacheAvailable || polymorphicInlineCache) &&

polymorphicInlineCache->TrySetProperty<

CheckLocal,

CheckLocalTypeWithoutProperty,

CheckAccessor,

IsInlineCacheAvailable,

ReturnOperationInfo

>(

object,

propertyId,

propertyValue,

requestContext,

operationInfo,

inlineCache,

propertyOperationFlags

))

{

return true;

}

}

}

if(!CheckTypePropertyCache)

{

return false;

}

TypePropertyCache \*const typePropertyCache = object->GetType()->GetPropertyCache();

if(!typePropertyCache ||

!typePropertyCache->TrySetProperty(

object,

propertyId,

propertyValue,

requestContext,

ReturnOperationInfo ? operationInfo : nullptr,

propertyValueInfo))

{

return false;

}

if(!ReturnOperationInfo || operationInfo->cacheType == CacheType\_TypeProperty)

{

return true;

}

// The property access was cached in an inline cache. Get the proper property operation info.

PretendTrySetProperty<IsInlineCacheAvailable, IsPolymorphicInlineCacheAvailable>(

object->GetType(),

object->GetType(),

operationInfo,

propertyValueInfo);

return true;

}

template<

bool IsInlineCacheAvailable,

bool IsPolymorphicInlineCacheAvailable>

\_\_inline void CacheOperators::PretendTryGetProperty(

Type \*const type,

PropertyCacheOperationInfo \*operationInfo,

PropertyValueInfo \*const propertyValueInfo)

{

CompileAssert(IsInlineCacheAvailable || IsPolymorphicInlineCacheAvailable);

Assert(propertyValueInfo);

Assert(IsInlineCacheAvailable == !!propertyValueInfo->GetInlineCache());

Assert(!IsPolymorphicInlineCacheAvailable || propertyValueInfo->GetPolymorphicInlineCache());

Assert(operationInfo);

if (IsInlineCacheAvailable && propertyValueInfo->GetInlineCache()->PretendTryGetProperty(type, operationInfo))

{

return;

}

Assert(IsPolymorphicInlineCacheAvailable || propertyValueInfo->GetFunctionBody());

PolymorphicInlineCache \*const polymorphicInlineCache =

IsPolymorphicInlineCacheAvailable

? propertyValueInfo->GetPolymorphicInlineCache()

: propertyValueInfo->GetFunctionBody()->GetPolymorphicInlineCache(propertyValueInfo->GetInlineCacheIndex());

if (IsPolymorphicInlineCacheAvailable || polymorphicInlineCache)

{

polymorphicInlineCache->PretendTryGetProperty(type, operationInfo);

}

}

template<

bool IsInlineCacheAvailable,

bool IsPolymorphicInlineCacheAvailable>

\_\_inline void CacheOperators::PretendTrySetProperty(

Type \*const type,

Type \*const oldType,

PropertyCacheOperationInfo \* operationInfo,

PropertyValueInfo \*const propertyValueInfo)

{

CompileAssert(IsInlineCacheAvailable || IsPolymorphicInlineCacheAvailable);

Assert(propertyValueInfo);

Assert(IsInlineCacheAvailable == !!propertyValueInfo->GetInlineCache());

Assert(!IsPolymorphicInlineCacheAvailable || propertyValueInfo->GetPolymorphicInlineCache());

Assert(operationInfo);

if (IsInlineCacheAvailable && propertyValueInfo->GetInlineCache()->PretendTrySetProperty(type, oldType, operationInfo))

{

return;

}

Assert(IsPolymorphicInlineCacheAvailable || propertyValueInfo->GetFunctionBody());

PolymorphicInlineCache \*const polymorphicInlineCache =

IsPolymorphicInlineCacheAvailable

? propertyValueInfo->GetPolymorphicInlineCache()

: propertyValueInfo->GetFunctionBody()->GetPolymorphicInlineCache(propertyValueInfo->GetInlineCacheIndex());

if (IsPolymorphicInlineCacheAvailable || polymorphicInlineCache)

{

polymorphicInlineCache->PretendTrySetProperty(type, oldType, operationInfo);

}

}

template<

bool IsAccessor,

bool IsRead,

bool IncludeTypePropertyCache>

\_\_inline void CacheOperators::Cache(

const bool isProto,

DynamicObject \*const objectWithProperty,

const bool isRoot,

Type \*const type,

Type \*const typeWithoutProperty,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

const bool isMissing,

const int requiredAuxSlotCapacity,

const PropertyValueInfo \*const info,

ScriptContext \*const requestContext)

{

CompileAssert(!IsAccessor || !IncludeTypePropertyCache);

Assert(info);

Assert(objectWithProperty);

if(!IsAccessor)

{

if(!isProto)

{

Assert(type == objectWithProperty->GetType());

}

else

{

Assert(IsRead);

Assert(type != objectWithProperty->GetType());

}

}

else

{

Assert(!isRoot); // could still be root object, but the parameter will be false and shouldn't be used for accessors

Assert(!typeWithoutProperty);

Assert(requiredAuxSlotCapacity == 0);

}

if(IsRead)

{

Assert(!typeWithoutProperty);

Assert(requiredAuxSlotCapacity == 0);

Assert(CanCachePropertyRead(objectWithProperty, requestContext));

if(!IsAccessor && isProto && PropertyValueInfo::PrototypeCacheDisabled(info))

{

return;

}

}

else

{

Assert(CanCachePropertyWrite(objectWithProperty, requestContext));

// TODO(ianhall): the following assert would let global const properties slip through when they shadow

// a global property. Reason being DictionaryTypeHandler::IsWritable cannot tell if it should check

// the global property or the global let/const. Fix this by updating IsWritable to recognize isRoot.

// Built-in Function.prototype properties 'length', 'arguments', and 'caller' are special cases.

Assert(

objectWithProperty->IsWritable(propertyId) ||

(isRoot && RootObjectBase::FromVar(objectWithProperty)->IsLetConstGlobal(propertyId)) ||

JavascriptFunction::IsBuiltinProperty(objectWithProperty, propertyId));

}

const bool includeTypePropertyCache = IncludeTypePropertyCache && !isRoot;

bool createTypePropertyCache = false;

PolymorphicInlineCache \*polymorphicInlineCache = info->GetPolymorphicInlineCache();

if(!polymorphicInlineCache && info->GetFunctionBody())

{

polymorphicInlineCache = info->GetFunctionBody()->GetPolymorphicInlineCache(info->GetInlineCacheIndex());

}

InlineCache \*const inlineCache = info->GetInlineCache();

if(inlineCache)

{

const bool tryCreatePolymorphicInlineCache = !polymorphicInlineCache && info->GetFunctionBody();

if((includeTypePropertyCache || tryCreatePolymorphicInlineCache) &&

inlineCache->HasDifferentType<IsAccessor>(isProto, type, typeWithoutProperty))

{

if(tryCreatePolymorphicInlineCache)

{

polymorphicInlineCache =

info->GetFunctionBody()->CreateNewPolymorphicInlineCache(

info->GetInlineCacheIndex(),

propertyId,

inlineCache);

}

if(includeTypePropertyCache)

{

createTypePropertyCache = true;

}

}

if(!IsAccessor)

{

if(!isProto)

{

inlineCache->CacheLocal(

type,

propertyId,

propertyIndex,

isInlineSlot,

typeWithoutProperty,

requiredAuxSlotCapacity,

requestContext);

}

else

{

inlineCache->CacheProto(

objectWithProperty,

propertyId,

propertyIndex,

isInlineSlot,

isMissing,

type,

requestContext);

}

}

else

{

inlineCache->CacheAccessor(

IsRead,

propertyId,

propertyIndex,

isInlineSlot,

type,

objectWithProperty,

isProto,

requestContext);

}

}

if(polymorphicInlineCache)

{

// Don't resize a polymorphic inline cache from full JIT because it currently doesn't rejit to use the new

// polymorphic inline cache. Once resized, bailouts would populate only the new set of caches and full JIT would

// continue to use to old set of caches.

Assert(!info->AllowResizingPolymorphicInlineCache() || info->GetFunctionBody());

if((includeTypePropertyCache && !createTypePropertyCache || info->AllowResizingPolymorphicInlineCache()) &&

polymorphicInlineCache->HasDifferentType<IsAccessor>(isProto, type, typeWithoutProperty))

{

if(info->AllowResizingPolymorphicInlineCache() && polymorphicInlineCache->CanAllocateBigger())

{

polymorphicInlineCache =

info->GetFunctionBody()->CreateBiggerPolymorphicInlineCache(

info->GetInlineCacheIndex(),

propertyId);

}

if(includeTypePropertyCache)

{

createTypePropertyCache = true;

}

}

if(!IsAccessor)

{

if(!isProto)

{

polymorphicInlineCache->CacheLocal(

type,

propertyId,

propertyIndex,

isInlineSlot,

typeWithoutProperty,

requiredAuxSlotCapacity,

requestContext);

}

else

{

polymorphicInlineCache->CacheProto(

objectWithProperty,

propertyId,

propertyIndex,

isInlineSlot,

isMissing,

type,

requestContext);

}

}

else

{

polymorphicInlineCache->CacheAccessor(

IsRead,

propertyId,

propertyIndex,

isInlineSlot,

type,

objectWithProperty,

isProto,

requestContext);

}

}

if(!includeTypePropertyCache)

{

return;

}

Assert(!IsAccessor);

TypePropertyCache \*typePropertyCache = type->GetPropertyCache();

if(!typePropertyCache)

{

if(!createTypePropertyCache)

{

return;

}

typePropertyCache = type->CreatePropertyCache();

}

if(isProto)

{

typePropertyCache->Cache(

propertyId,

propertyIndex,

isInlineSlot,

info->IsWritable() && info->IsStoreFieldCacheEnabled(),

isMissing,

objectWithProperty,

type);

typePropertyCache = objectWithProperty->GetType()->GetPropertyCache();

if(!typePropertyCache)

{

return;

}

}

typePropertyCache->Cache(

propertyId,

propertyIndex,

isInlineSlot,

info->IsWritable() && info->IsStoreFieldCacheEnabled());

}

}

<?xml version="1.0" encoding="utf-8"?>

<Project DefaultTargets="Build" ToolsVersion="12.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Import Condition="'$(ChakraBuildPathImported)'!='true'" Project="$(SolutionDir)Chakra.Build.Paths.props" />

<Import Project="$(BuildConfigPropsPath)Chakra.Build.ProjectConfiguration.props" />

<PropertyGroup Label="Globals">

<TargetName>Chakra.Runtime.Language</TargetName>

<ProjectGuid>{706083F7-6AA4-4558-A153-6352EF9110F8}</ProjectGuid>

<RootNamespace>JS</RootNamespace>

<Keyword>Win32Proj</Keyword>

</PropertyGroup>

<PropertyGroup Label="Configuration">

<ConfigurationType>StaticLibrary</ConfigurationType>

</PropertyGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />

<ImportGroup Label="ExtensionSettings">

<Import Project="$(VCTargetsPath)\BuildCustomizations\masm.props" />

<Import Project="$(BuildConfig\_ARMASM\_Path)armasm.props" />

</ImportGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.props" />

<PropertyGroup>

<\_ProjectFileVersion>10.0.30319.1</\_ProjectFileVersion>

</PropertyGroup>

<ItemDefinitionGroup>

<ClCompile>

<AdditionalIncludeDirectories>

$(MSBuildThisFileDirectory)..;

$(MSBuildThisFileDirectory)..\..\Common;

$(MSBuildThisFileDirectory)..\..\Parser;

$(MSBuildThisFileDirectory)..\..\Backend;

%(AdditionalIncludeDirectories)

</AdditionalIncludeDirectories>

<PrecompiledHeader>Use</PrecompiledHeader>

<PrecompiledHeaderFile>RuntimeLanguagePch.h</PrecompiledHeaderFile>

</ClCompile>

</ItemDefinitionGroup>

<ItemGroup>

<ClCompile Include="$(MSBuildThisFileDirectory)i386\stackframe.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32'">true</ExcludedFromBuild>

<ObjectFileName Condition="'$(Platform)'!='Win32'">$(IntDir)\i386</ObjectFileName>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)amd64\stackframe.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='x64'">true</ExcludedFromBuild>

<ObjectFileName Condition="'$(Platform)'!='x64'">$(IntDir)\amd64</ObjectFileName>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)arm\stackframe.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='ARM'">true</ExcludedFromBuild>

<ObjectFileName Condition="'$(Platform)'!='ARM'">$(IntDir)\arm</ObjectFileName>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)arm64\stackframe.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Arm64'">true</ExcludedFromBuild>

<ObjectFileName Condition="'$(Platform)'!='Arm64'">$(IntDir)\arm64</ObjectFileName>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)i386\AsmJSJitTemplate.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32'">true</ExcludedFromBuild>

<ObjectFileName Condition="'$(Platform)'!='Win32'">$(IntDir)\i386</ObjectFileName>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)amd64\AsmJSJitTemplate.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='x64'">true</ExcludedFromBuild>

<ObjectFileName Condition="'$(Platform)'!='x64'">$(IntDir)\amd64</ObjectFileName>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJSCodeGenerator.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32' AND '$(Platform)'!='x64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJSEncoder.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32' AND '$(Platform)'!='x64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJS.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJSBytecodeGenerator.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJSLink.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJSModule.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJSTypes.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJSUtils.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)CacheOperators.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)CodeGenRecyclableData.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)DynamicProfileInfo.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)DynamicProfileMutator.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)DynamicProfileStorage.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ExecutionMode.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)FunctionCodeGenJitTimeData.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)FunctionCodeGenRuntimeData.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)InlineCache.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)JavascriptExceptionObject.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)JavascriptExceptionOperators.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)JavascriptMathOperators.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ProfilingHelpers.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ReadOnlyDynamicProfileInfo.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDFloat32x4Operation.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='ARM' AND '$(Platform)'!='Arm64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDfloat32x4OperationX86X64.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32' AND '$(Platform)'!='x64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDInt32x4Operation.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='ARM' AND '$(Platform)'!='Arm64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDInt32x4OperationX86X64.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32' AND '$(Platform)'!='x64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDInt8x16Operation.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='ARM' AND '$(Platform)'!='Arm64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDInt8x16OperationX86X64.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32' AND '$(Platform)'!='x64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDFloat64x2Operation.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='ARM' AND '$(Platform)'!='Arm64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDfloat64x2OperationX86X64.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32' AND '$(Platform)'!='x64'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)SIMDUtils.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)SourceDynamicProfileManager.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)StackTraceArguments.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)TaggedInt.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ValueType.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)InterpreterStackFrame.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)JavascriptConversion.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)JavascriptOperators.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)JavascriptStackWalker.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)RuntimeLanguagePch.cpp">

<PrecompiledHeader>Create</PrecompiledHeader>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)JavascriptNativeOperators.cpp" />

</ItemGroup>

<ItemGroup>

<ClInclude Include="amd64\stackframe.h">

<ExcludedFromBuild Condition="'$(Platform)'!='x64'">true</ExcludedFromBuild>

</ClInclude>

<ClInclude Include="arm\stackframe.h">

<ExcludedFromBuild Condition="'$(Platform)'!='ARM'">true</ExcludedFromBuild>

</ClInclude>

<ClInclude Include="AsmJS.h" />

<ClInclude Include="AsmJSBytecodeGenerator.h" />

<ClInclude Include="AsmJSCodeGenerator.h" />

<ClInclude Include="AsmJSEncoder.h">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32' AND '$(Platform)'!='x64'">true</ExcludedFromBuild>

</ClInclude>

<ClInclude Include="AsmJSJitTemplate.h" />

<ClInclude Include="AsmJSLink.h" />

<ClInclude Include="AsmJSModule.h" />

<ClInclude Include="AsmJsBuiltinNames.h" />

<ClInclude Include="AsmJSTypes.h" />

<ClInclude Include="AsmJSUtils.h" />

<ClInclude Include="CacheOperators.h" />

<ClInclude Include="i386\AsmJsInstructionTemplate.h">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32'">true</ExcludedFromBuild>

</ClInclude>

<ClInclude Include="InterpreterProcessOpCodeAsmJs.h" />

<ClInclude Include="CodeGenRecyclableData.h" />

<ClInclude Include="DynamicProfileInfo.h" />

<ClInclude Include="DynamicProfileMutator.h" />

<ClInclude Include="DynamicProfileStorage.h" />

<ClInclude Include="EvalMapRecord.h" />

<ClInclude Include="ExecutionMode.h" />

<ClInclude Include="ExecutionModes.h" />

<ClInclude Include="FunctionCodeGenJitTimeData.h" />

<ClInclude Include="FunctionCodeGenRuntimeData.h" />

<ClInclude Include="i386\stackframe.h">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32'">true</ExcludedFromBuild>

</ClInclude>

<ClInclude Include="InlineCache.h" />

<ClInclude Include="InlineCachePointerArray.h" />

<ClInclude Include="JavascriptExceptionOperators.h" />

<ClInclude Include="JavascriptMathOperators.h" />

<ClInclude Include="JavascriptNativeOperators.h" />

<ClInclude Include="ProfilingHelpers.h" />

<ClInclude Include="ReadOnlyDynamicProfileInfo.h" />

<ClInclude Include="SourceDynamicProfileManager.h" />

<ClInclude Include="StackTraceArguments.h" />

<ClInclude Include="ValueType.h" />

<ClInclude Include="Arguments.h" />

<ClInclude Include="InterpreterStackFrame.h" />

<ClInclude Include="JavascriptConversion.h" />

<ClInclude Include="JavascriptExceptionContext.h" />

<ClInclude Include="JavascriptExceptionObject.h" />

<ClInclude Include="JavascriptOperators.h" />

<ClInclude Include="JavascriptFunctionArgIndex.h" />

<ClInclude Include="JavascriptStackWalker.h" />

<ClInclude Include="SIMDFloat32x4Operation.h" />

<ClInclude Include="SIMDFloat64x2Operation.h" />

<ClInclude Include="SIMDInt32x4Operation.h" />

<ClInclude Include="SIMDInt8x16Operation.h" />

<ClInclude Include="SimdOps.h" />

<ClInclude Include="SIMDUtils.h" />

<ClInclude Include="TaggedInt.h" />

<ClInclude Include="RuntimeLanguagePch.h" />

</ItemGroup>

<ItemGroup>

<MASM Include="$(MSBuildThisFileDirectory)amd64\JavascriptOperatorsA.asm">

<ExcludedFromBuild Condition="'$(Platform)'!='x64'">true</ExcludedFromBuild>

</MASM>

<MASM Include="$(MSBuildThisFileDirectory)amd64\amd64\_Thunks.asm">

<PreprocessorDefinitions Condition="'$(BuildJIT)'!='false'">%(PreprocessorDefinitions);\_ENABLE\_DYNAMIC\_THUNKS=1</PreprocessorDefinitions>

<ExcludedFromBuild Condition="'$(Platform)'!='x64'">true</ExcludedFromBuild>

</MASM>

<ARMASM Include="$(MSBuildThisFileDirectory)arm\arm\_Thunks.asm">

<ExcludedFromBuild Condition="'$(Platform)'!='ARM'">true</ExcludedFromBuild>

</ARMASM>

<ARMASM Include="$(MSBuildThisFileDirectory)arm\arm\_CallEhFrame.asm">

<ExcludedFromBuild Condition="'$(Platform)'!='ARM'">true</ExcludedFromBuild>

</ARMASM>

<ARMASM Include="$(MSBuildThisFileDirectory)arm64\arm64\_Thunks.asm">

<ExcludedFromBuild Condition="'$(Platform)'!='Arm64'">true</ExcludedFromBuild>

</ARMASM>

<ARMASM Include="$(MSBuildThisFileDirectory)arm64\arm64\_CallEhFrame.asm">

<ExcludedFromBuild Condition="'$(Platform)'!='Arm64'">true</ExcludedFromBuild>

</ARMASM>

<None Include="amd64\stackframe.inl" />

<None Include="AsmJSEncoder.inl" />

<None Include="AsmJSEncoderHandler.inl" />

<None Include="CacheOperators.inl" />

<None Include="i386\AsmJsInstructionTemplate.inl" />

<None Include="InlineCache.inl" />

<None Include="InlineCachePointerArray.inl" />

<None Include="InterpreterHandler.inl" />

<None Include="InterpreterHandlerAsmJs.inl" />

<None Include="InterpreterLoop.inl" />

<None Include="JavascriptMathOperators.inl" />

<None Include="JavascriptConversion.inl" />

<None Include="JavascriptOperators.inl" />

<None Include="TaggedInt.inl" />

</ItemGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.targets" Condition="exists('$(BuildConfigPropsPath)Chakra.Build.targets')"/>

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />

<ImportGroup Label="ExtensionTargets">

<Import Project="$(VCTargetsPath)\BuildCustomizations\masm.targets" />

<Import Project="$(BuildConfig\_ARMASM\_Path)armasm.targets" />

</ImportGroup>

</Project>

<?xml version="1.0" encoding="utf-8"?>

<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<ItemGroup>

<ClCompile Include="$(MsBuildThisFileDirectory)..\StdAfx.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)AsmJS.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)AsmJSBytecodeGenerator.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)AsmJSCodeGenerator.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)AsmJSEncoder.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)AsmJSLink.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)AsmJSModule.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)AsmJSTypes.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)AsmJSUtils.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)ByteCodeSerializer.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)CacheOperators.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)CallInfo.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)CodeGenRecyclableData.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)DynamicProfileInfo.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)DynamicProfileMutator.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)DynamicProfileStorage.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)ExecutionMode.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)FunctionCodeGenJitTimeData.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)FunctionCodeGenRuntimeData.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)InlineCache.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)JavascriptExceptionOperators.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)JavascriptMathOperators.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)ProfilingHelpers.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)PropertyRecord.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)ReadOnlyDynamicProfileInfo.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)SourceDynamicProfileManager.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)SourceHolder.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)StackTraceArguments.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)Utf8SourceInfo.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)ValueType.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)ScriptContextOptimizationOverrideInfo.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)DiagHelperMethodWrapper.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)InterpreterStackFrame.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)JavascriptConversion.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)JavascriptOperators.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)JavascriptStackWalker.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)ScriptContext.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)ScriptContextProfiler.cpp" />

<ClCompile Include="$(MsBuildThisFileDirectory)amd64\AsmJSJitTemplate.cpp">

<Filter>amd64</Filter>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)i386\AsmJSJitTemplate.cpp">

<Filter>i386</Filter>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)amd64\stackframe.cpp">

<Filter>amd64</Filter>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)i386\stackframe.cpp">

<Filter>i386</Filter>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)arm64\stackframe.cpp">

<Filter>arm64</Filter>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)arm\stackframe.cpp">

<Filter>arm</Filter>

</ClCompile>

<ClCompile Include="JavascriptNativeOperators.cpp" />

</ItemGroup>

<ItemGroup>

<ClInclude Include="AsmJS.h" />

<ClInclude Include="AsmJSBytecodeGenerator.h" />

<ClInclude Include="AsmJSCodeGenerator.h" />

<ClInclude Include="AsmJSEncoder.h" />

<ClInclude Include="AsmJSJitTemplate.h" />

<ClInclude Include="AsmJSLink.h" />

<ClInclude Include="AsmJSModule.h" />

<ClInclude Include="AsmJSUtils.h" />

<ClInclude Include="byteCodeCacheReleaseFileVersion.h" />

<ClInclude Include="CacheOperators.h" />

<ClInclude Include="InterpreterProcessOpCodeAsmJs.h" />

<ClInclude Include="CodeGenRecyclableData.h" />

<ClInclude Include="DynamicProfileInfo.h" />

<ClInclude Include="DynamicProfileMutator.h" />

<ClInclude Include="DynamicProfileStorage.h" />

<ClInclude Include="EvalMapRecord.h" />

<ClInclude Include="ExecutionMode.h" />

<ClInclude Include="ExecutionModes.h" />

<ClInclude Include="FunctionCodeGenJitTimeData.h" />

<ClInclude Include="FunctionCodeGenRuntimeData.h" />

<ClInclude Include="InlineCache.h" />

<ClInclude Include="InlineCachePointerArray.h" />

<ClInclude Include="JavascriptExceptionOperators.h" />

<ClInclude Include="JavascriptMathOperators.h" />

<ClInclude Include="LeaveScriptObject.h" />

<ClInclude Include="ProfilingHelpers.h" />

<ClInclude Include="PropertyRecord.h" />

<ClInclude Include="ReadOnlyDynamicProfileInfo.h" />

<ClInclude Include="SourceDynamicProfileManager.h" />

<ClInclude Include="SourceHolder.h" />

<ClInclude Include="StackTraceArguments.h" />

<ClInclude Include="Utf8SourceInfo.h" />

<ClInclude Include="ValueType.h" />

<ClInclude Include="ScriptContextOptimizationOverrideInfo.h" />

<ClInclude Include="Arguments.h" />

<ClInclude Include="CallInfo.h" />

<ClInclude Include="DiagHelperMethodWrapper.h" />

<ClInclude Include="InterpreterStackFrame.h" />

<ClInclude Include="JavascriptConversion.h" />

<ClInclude Include="JavascriptOperators.h" />

<ClInclude Include="JavascriptFunctionArgIndex.h" />

<ClInclude Include="JavascriptStackWalker.h" />

<ClInclude Include="ScriptContext.h" />

<ClInclude Include="ScriptContextProfiler.h" />

<ClInclude Include="i386\AsmJsInstructionTemplate.h">

<Filter>i386</Filter>

</ClInclude>

<ClInclude Include="amd64\stackframe.h">

<Filter>amd64</Filter>

</ClInclude>

<ClInclude Include="i386\stackframe.h">

<Filter>i386</Filter>

</ClInclude>

<ClInclude Include="arm\stackframe.h">

<Filter>arm</Filter>

</ClInclude>

<ClInclude Include="ByteCodeSerializeFlags.h" />

<ClInclude Include="ScriptContextBase.h" />

<ClInclude Include="AsmJSTypes.h" />

<ClInclude Include="AsmJsBuiltinNames.h" />

<ClInclude Include="JavascriptNativeOperators.h" />

</ItemGroup>

<ItemGroup>

<MASM Include="$(MSBuildThisFileDirectory)amd64\amd64\_Thunks.asm">

<Filter>amd64</Filter>

</MASM>

<MASM Include="$(MSBuildThisFileDirectory)amd64\JavascriptOperatorsA.asm">

<Filter>amd64</Filter>

</MASM>

</ItemGroup>

<ItemGroup>

<None Include="AsmJSEncoder.inl" />

<None Include="AsmJSEncoderHandler.inl" />

<None Include="CacheOperators.inl" />

<None Include="InlineCache.inl" />

<None Include="InlineCachePointerArray.inl" />

<None Include="InterpreterHandler.inl" />

<None Include="InterpreterHandlerAsmJs.inl" />

<None Include="InterpreterLoop.inl" />

<None Include="JavascriptExceptionOperators.inl" />

<None Include="JavascriptMathOperators.inl" />

<None Include="JavascriptObject.inl" />

<None Include="InterpreterStackFrame.inl" />

<None Include="JavascriptConversion.inl" />

<None Include="JavascriptOperators.inl" />

<None Include="ScriptContext.inl" />

<None Include="DiagHelperMethodWrapper.inl" />

<None Include="amd64\stackframe.inl">

<Filter>amd64</Filter>

</None>

<None Include="i386\AsmJsInstructionTemplate.inl">

<Filter>i386</Filter>

</None>

</ItemGroup>

<ItemGroup>

<Filter Include="amd64">

<UniqueIdentifier>{4e93923e-1e63-4f8e-93ab-4015a8f4dfe2}</UniqueIdentifier>

</Filter>

<Filter Include="arm">

<UniqueIdentifier>{f9327b60-3e20-4b93-b8ff-835fab180090}</UniqueIdentifier>

</Filter>

<Filter Include="arm64">

<UniqueIdentifier>{88887430-4252-4438-a157-ff7b193085a7}</UniqueIdentifier>

</Filter>

<Filter Include="i386">

<UniqueIdentifier>{dd87d172-424d-422f-8c0c-23f324878ed6}</UniqueIdentifier>

</Filter>

</ItemGroup>

<ItemGroup>

<ARMASM Include="$(MSBuildThisFileDirectory)arm\arm\_Thunks.asm">

<Filter>arm</Filter>

</ARMASM>

<ARMASM Include="$(MSBuildThisFileDirectory)arm\arm\_CallEhFrame.asm">

<Filter>arm</Filter>

</ARMASM>

<ARMASM Include="$(MSBuildThisFileDirectory)arm64\arm64\_Thunks.asm">

<Filter>arm64</Filter>

</ARMASM>

<ARMASM Include="$(MSBuildThisFileDirectory)arm64\arm64\_CallEhFrame.asm">

<Filter>arm64</Filter>

</ARMASM>

</ItemGroup>

</Project>

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Language\CodeGenRecyclableData.h"

namespace Js

{

CodeGenRecyclableData::CodeGenRecyclableData(const FunctionCodeGenJitTimeData \*const jitTimeData) : jitTimeData(jitTimeData)

{

Assert(jitTimeData);

}

const FunctionCodeGenJitTimeData \*CodeGenRecyclableData::JitTimeData() const

{

return jitTimeData;

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

class FunctionCodeGenJitTimeData;

// Keeps data relevant to a function body that is needed for jitting the function, alive until jitting is complete

class CodeGenRecyclableData sealed : public JsUtil::DoublyLinkedListElement<CodeGenRecyclableData>

{

private:

const FunctionCodeGenJitTimeData \*const jitTimeData;

public:

CodeGenRecyclableData(const FunctionCodeGenJitTimeData \*const jitTimeData);

public:

const FunctionCodeGenJitTimeData \*JitTimeData() const;

PREVENT\_COPY(CodeGenRecyclableData);

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if ENABLE\_NATIVE\_CODEGEN

namespace Js

{

#ifdef DYNAMIC\_PROFILE\_STORAGE

DynamicProfileInfo::DynamicProfileInfo()

{

hasFunctionBody = false;

}

#endif

struct Allocation

{

uint offset;

size\_t size;

};

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

bool DynamicProfileInfo::NeedProfileInfoList()

{

#pragma prefast(suppress: 6235 6286, "(<non-zero constant> || <expression>) is always a non-zero constant. - This is wrong, DBG\_DUMP is not set in some build variants")

return DBG\_DUMP

#ifdef DYNAMIC\_PROFILE\_STORAGE

|| DynamicProfileStorage::IsEnabled()

#endif

#ifdef RUNTIME\_DATA\_COLLECTION

|| (Configuration::Global.flags.RuntimeDataOutputFile != nullptr)

#endif

;

}

#endif

void ArrayCallSiteInfo::SetIsNotNativeIntArray()

{

OUTPUT\_TRACE\_WITH\_STACK(Js::NativeArrayConversionPhase, L"SetIsNotNativeIntArray \n");

bits |= NotNativeIntBit;

}

void ArrayCallSiteInfo::SetIsNotNativeFloatArray()

{

OUTPUT\_TRACE\_WITH\_STACK(Js::NativeArrayConversionPhase, L"SetIsNotNativeFloatArray \n");

bits |= NotNativeFloatBit;

}

void ArrayCallSiteInfo::SetIsNotNativeArray()

{

OUTPUT\_TRACE\_WITH\_STACK(Js::NativeArrayConversionPhase, L"SetIsNotNativeArray \n");

bits = NotNativeIntBit | NotNativeFloatBit;

}

DynamicProfileInfo\* DynamicProfileInfo::New(Recycler\* recycler, FunctionBody\* functionBody, bool persistsAcrossScriptContexts)

{

size\_t totalAlloc = 0;

Allocation batch[] =

{

{ (uint)offsetof(DynamicProfileInfo, callSiteInfo), functionBody->GetProfiledCallSiteCount() \* sizeof(CallSiteInfo) },

{ (uint)offsetof(DynamicProfileInfo, ldElemInfo), functionBody->GetProfiledLdElemCount() \* sizeof(LdElemInfo) },

{ (uint)offsetof(DynamicProfileInfo, stElemInfo), functionBody->GetProfiledStElemCount() \* sizeof(StElemInfo) },

{ (uint)offsetof(DynamicProfileInfo, arrayCallSiteInfo), functionBody->GetProfiledArrayCallSiteCount() \* sizeof(ArrayCallSiteInfo) },

{ (uint)offsetof(DynamicProfileInfo, fldInfo), functionBody->GetProfiledFldCount() \* sizeof(FldInfo) },

{ (uint)offsetof(DynamicProfileInfo, divideTypeInfo), functionBody->GetProfiledDivOrRemCount() \* sizeof(ValueType) },

{ (uint)offsetof(DynamicProfileInfo, switchTypeInfo), functionBody->GetProfiledSwitchCount() \* sizeof(ValueType)},

{ (uint)offsetof(DynamicProfileInfo, slotInfo), functionBody->GetProfiledSlotCount() \* sizeof(ValueType) },

{ (uint)offsetof(DynamicProfileInfo, parameterInfo), functionBody->GetProfiledInParamsCount() \* sizeof(ValueType) },

{ (uint)offsetof(DynamicProfileInfo, returnTypeInfo), functionBody->GetProfiledReturnTypeCount() \* sizeof(ValueType) },

{ (uint)offsetof(DynamicProfileInfo, loopImplicitCallFlags), (EnableImplicitCallFlags(functionBody) ? (functionBody->GetLoopCount() \* sizeof(ImplicitCallFlags)) : 0) },

{ (uint)offsetof(DynamicProfileInfo, loopFlags), functionBody->GetLoopCount() ? BVFixed::GetAllocSize(functionBody->GetLoopCount() \* LoopFlags::COUNT) : 0 }

};

for (uint i = 0; i < \_countof(batch); i++)

{

totalAlloc += batch[i].size;

}

DynamicProfileInfo\* info = nullptr;

// In the profile storage case (-only), always allocate a non-leaf profile

// In the regular profile case, we need to allocate it as non-leaf only if it's

// a profile being used in the in-memory cache. This is because in that case, the profile

// also allocates dynamicProfileFunctionInfo, which it uses to match functions across

// script contexts. In the normal case, since we don't allocate that structure, we

// can be a leaf allocation.

if (persistsAcrossScriptContexts)

{

info = RecyclerNewPlusZ(recycler, totalAlloc, DynamicProfileInfo, functionBody);

#if DBG

info->persistsAcrossScriptContexts = true;

#endif

}

else

{

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

if (DynamicProfileInfo::NeedProfileInfoList())

{

info = RecyclerNewPlusZ(recycler, totalAlloc, DynamicProfileInfo, functionBody);

}

else

#endif

{

info = RecyclerNewPlusLeafZ(recycler, totalAlloc, DynamicProfileInfo, functionBody);

}

}

BYTE\* current = (BYTE\*)info + sizeof(DynamicProfileInfo);

for (uint i = 0; i < \_countof(batch); i++)

{

if (batch[i].size > 0)

{

BYTE\*\* field = (BYTE\*\*)(((BYTE\*)info + batch[i].offset));

\*field = current;

current += batch[i].size;

}

}

Assert(current - reinterpret\_cast<BYTE\*>(info) - sizeof(DynamicProfileInfo) == totalAlloc);

info->Initialize(functionBody);

return info;

}

DynamicProfileInfo::DynamicProfileInfo(FunctionBody \* functionBody)

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

: functionBody(DynamicProfileInfo::NeedProfileInfoList() ? functionBody : nullptr)

#endif

{

hasFunctionBody = true;

#if DBG

persistsAcrossScriptContexts = true;

#endif

}

void DynamicProfileInfo::Initialize(FunctionBody \*const functionBody)

{

// Need to make value types uninitialized, which is not equivalent to zero

thisInfo.valueType = ValueType::Uninitialized;

const BVIndex loopFlagsCount = functionBody->GetLoopCount() \* LoopFlags::COUNT;

if (loopFlagsCount)

{

this->loopFlags->Init(loopFlagsCount);

LoopFlags defaultValues;

for (uint i = 0; i < functionBody->GetLoopCount(); ++i)

{

this->loopFlags->SetRange(&defaultValues, i \* LoopFlags::COUNT, LoopFlags::COUNT);

}

}

for (ProfileId i = 0; i < functionBody->GetProfiledCallSiteCount(); ++i)

{

callSiteInfo[i].returnType = ValueType::Uninitialized;

callSiteInfo[i].u.functionData.sourceId = NoSourceId;

}

for (ProfileId i = 0; i < functionBody->GetProfiledLdElemCount(); ++i)

{

ldElemInfo[i].arrayType = ValueType::Uninitialized;

ldElemInfo[i].elemType = ValueType::Uninitialized;

}

for (ProfileId i = 0; i < functionBody->GetProfiledStElemCount(); ++i)

{

stElemInfo[i].arrayType = ValueType::Uninitialized;

}

for (uint i = 0; i < functionBody->GetProfiledFldCount(); ++i)

{

fldInfo[i].flags = FldInfo\_NoInfo;

fldInfo[i].valueType = ValueType::Uninitialized;

fldInfo[i].polymorphicInlineCacheUtilization = PolymorphicInlineCacheUtilizationThreshold;

}

for (ProfileId i = 0; i < functionBody->GetProfiledDivOrRemCount(); ++i)

{

divideTypeInfo[i] = ValueType::Uninitialized;

}

for (ProfileId i = 0; i < functionBody->GetProfiledSwitchCount(); ++i)

{

switchTypeInfo[i] = ValueType::Uninitialized;

}

for (ProfileId i = 0; i < functionBody->GetProfiledSlotCount(); ++i)

{

slotInfo[i] = ValueType::Uninitialized;

}

for (ArgSlot i = 0; i < functionBody->GetProfiledInParamsCount(); ++i)

{

parameterInfo[i] = ValueType::Uninitialized;

}

for (ProfileId i = 0; i < functionBody->GetProfiledReturnTypeCount(); ++i)

{

returnTypeInfo[i] = ValueType::Uninitialized;

}

#if DBG

for (ProfileId i = 0; i < functionBody->GetProfiledArrayCallSiteCount(); ++i)

{

arrayCallSiteInfo[i].functionNumber = functionBody->GetFunctionNumber();

arrayCallSiteInfo[i].callSiteNumber = i;

}

#endif

}

bool DynamicProfileInfo::IsEnabledForAtLeastOneFunction(const ScriptContext \*const scriptContext)

{

return IsEnabled\_OptionalFunctionBody(nullptr, scriptContext);

}

bool DynamicProfileInfo::IsEnabled(const FunctionBody \*const functionBody)

{

Assert(functionBody);

return IsEnabled\_OptionalFunctionBody(functionBody, functionBody->GetScriptContext());

}

bool DynamicProfileInfo::IsEnabled\_OptionalFunctionBody(const FunctionBody \*const functionBody, const ScriptContext \*const scriptContext)

{

Assert(scriptContext);

return

!PHASE\_OFF\_OPTFUNC(DynamicProfilePhase, functionBody) &&

(

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

PHASE\_FORCE\_OPTFUNC(DynamicProfilePhase, functionBody) ||

#else

Js::Configuration::Global.flags.ForceDynamicProfile ||

#endif

!scriptContext->GetConfig()->IsNoNative() ||

scriptContext->IsInDebugMode()

#ifdef DYNAMIC\_PROFILE\_STORAGE

|| DynamicProfileStorage::DoCollectInfo()

#endif

);

}

bool DynamicProfileInfo::IsEnabledForAtLeastOneFunction(const Js::Phase phase, const ScriptContext \*const scriptContext)

{

return IsEnabled\_OptionalFunctionBody(phase, nullptr, scriptContext);

}

bool DynamicProfileInfo::IsEnabled(const Js::Phase phase, const FunctionBody \*const functionBody)

{

Assert(functionBody);

return IsEnabled\_OptionalFunctionBody(phase, functionBody, functionBody->GetScriptContext());

}

bool DynamicProfileInfo::IsEnabled\_OptionalFunctionBody(

const Js::Phase phase,

const FunctionBody \*const functionBody,

const ScriptContext \*const scriptContext)

{

if (!DynamicProfileInfo::IsEnabled\_OptionalFunctionBody(functionBody, scriptContext))

{

return false;

}

switch (phase)

{

case Phase::TypedArrayPhase:

case Phase::AggressiveIntTypeSpecPhase:

case Phase::CheckThisPhase:

case Phase::ProfileBasedFldFastPathPhase:

case Phase::ObjTypeSpecPhase:

case Phase::ArrayCheckHoistPhase:

case Phase::SwitchOptPhase:

case Phase::FixedNewObjPhase:

return !PHASE\_OFF\_PROFILED\_BYTE\_CODE\_OPTFUNC(phase, functionBody);

case Phase::NativeArrayPhase:

case Phase::FloatTypeSpecPhase:

return !PHASE\_OFF\_PROFILED\_BYTE\_CODE\_OPTFUNC(phase, functionBody)

#ifdef \_M\_IX86

&& AutoSystemInfo::Data.SSE2Available()

#endif

;

case Phase::InlinePhase:

return !PHASE\_OFF\_PROFILED\_BYTE\_CODE\_OPTFUNC(Phase::InlinePhase, functionBody);

}

return false;

}

bool DynamicProfileInfo::EnableImplicitCallFlags(const FunctionBody \*const functionBody)

{

return DynamicProfileInfo::IsEnabled(functionBody);

}

#ifdef \_M\_IX86

\_\_declspec(naked)

Var

DynamicProfileInfo::EnsureDynamicProfileInfoThunk(RecyclableObject\* function, CallInfo callInfo, ...)

{

\_\_asm

{

push ebp

mov ebp, esp

push[esp + 8] // push function object

call DynamicProfileInfo::EnsureDynamicProfileInfo;

#ifdef \_CONTROL\_FLOW\_GUARD

// verify that the call target is valid

mov ecx, eax

call[\_\_guard\_check\_icall\_fptr]

mov eax, ecx

#endif

pop ebp

jmp eax

}

}

#endif

JavascriptMethod DynamicProfileInfo::EnsureDynamicProfileInfo(ScriptFunction \* function)

{

// If we're creating a dynamic profile, make sure that the function

// has an entry point and this entry point is the "default" entrypoint

// created when a function body is created.

Assert(function->GetEntryPointInfo() != nullptr);

Assert(function->GetFunctionEntryPointInfo()->entryPointIndex == 0);

FunctionBody \* functionBody = function->GetFunctionBody();

// This is used only if the first entry point codegen completes.

// So there is no concurrency concern with background code gen thread modifying the entry point.

EntryPointInfo \* entryPoint = functionBody->GetEntryPointInfo(0);

Assert(entryPoint == function->GetEntryPointInfo());

Assert(entryPoint->IsCodeGenDone());

JavascriptMethod directEntryPoint = (JavascriptMethod)entryPoint->address;

// Check if it has changed already

if (directEntryPoint == DynamicProfileInfo::EnsureDynamicProfileInfoThunk)

{

functionBody->EnsureDynamicProfileInfo();

if (functionBody->GetScriptContext()->CurrentThunk == ProfileEntryThunk)

{

directEntryPoint = ProfileEntryThunk;

}

else

{

directEntryPoint = (JavascriptMethod)entryPoint->GetNativeAddress();

}

entryPoint->address = directEntryPoint;

}

else

{

Assert(directEntryPoint == ProfileEntryThunk || IsNativeFunctionAddr(functionBody->GetScriptContext(), directEntryPoint));

Assert(functionBody->HasExecutionDynamicProfileInfo());

}

return function->UpdateThunkEntryPoint(static\_cast<FunctionEntryPointInfo\*>(entryPoint), directEntryPoint);

}

bool DynamicProfileInfo::hasLdFldCallSiteInfo()

{

return bits.hasLdFldCallSite;

}

bool DynamicProfileInfo::RecordLdFldCallSiteInfo(FunctionBody\* functionBody, RecyclableObject\* callee, bool callApplyTarget)

{

auto SetBits = [&]() -> bool

{

this->bits.hasLdFldCallSite = true;

this->currentInlinerVersion++; // we don't mind if this overflows

return true;

};

FunctionInfo\* calleeFunctionInfo = callee->GetTypeId() == TypeIds\_Function ? JavascriptFunction::FromVar(callee)->GetFunctionInfo() : nullptr;

if (calleeFunctionInfo == nullptr)

{

return false;

}

else if (!calleeFunctionInfo->HasBody())

{

// We can inline fastDOM getter/setter.

// We can directly call Math.max/min as apply targets.

if ((calleeFunctionInfo->GetAttributes() & Js::FunctionInfo::Attributes::NeedCrossSiteSecurityCheck) ||

(callApplyTarget && (calleeFunctionInfo->GetAttributes() & Js::FunctionInfo::Attributes::BuiltInInlinableAsLdFldInlinee)))

{

if (functionBody->GetScriptContext() == callee->GetScriptContext())

{

return SetBits();

}

}

return false;

}

else if (functionBody->CheckCalleeContextForInlining(calleeFunctionInfo->GetFunctionProxy()))

{

// If functionInfo !HasBody(), the previous 'else if' branch is executed; otherwise it has a body and therefore it has a proxy

return SetBits();

}

return false;

}

void DynamicProfileInfo::RecordConstParameterAtCallSite(ProfileId callSiteId, int argNum)

{

Assert(argNum < Js::InlineeCallInfo::MaxInlineeArgoutCount);

Assert(callSiteId < functionBody->GetProfiledCallSiteCount());

callSiteInfo[callSiteId].isArgConstant = callSiteInfo[callSiteId].isArgConstant | (1 << argNum);

}

uint16 DynamicProfileInfo::GetConstantArgInfo(ProfileId callSiteId)

{

return callSiteInfo[callSiteId].isArgConstant;

}

void DynamicProfileInfo::RecordCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, FunctionInfo\* calleeFunctionInfo, JavascriptFunction\* calleeFunction, ArgSlot actualArgCount, bool isConstructorCall, InlineCacheIndex ldFldInlineCacheId)

{

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

// If we persistsAcrossScriptContext, the dynamic profile info may be referred to by multiple function body from

// different script context

Assert(!DynamicProfileInfo::NeedProfileInfoList() || this->persistsAcrossScriptContexts || this->functionBody == functionBody);

#endif

bool doInline = true;

// This is a hard limit as we only use 4 bits to encode the actual count in the InlineeCallInfo

if (actualArgCount > Js::InlineeCallInfo::MaxInlineeArgoutCount)

{

doInline = false;

}

// Mark the callsite bit where caller and callee is same function

if (functionBody == calleeFunctionInfo && callSiteId < 32)

{

this->m\_recursiveInlineInfo = this->m\_recursiveInlineInfo | (1 << callSiteId);

}

if (!callSiteInfo[callSiteId].isPolymorphic)

{

Js::SourceId oldSourceId = callSiteInfo[callSiteId].u.functionData.sourceId;

if (oldSourceId == InvalidSourceId)

{

return;

}

Js::LocalFunctionId oldFunctionId = callSiteInfo[callSiteId].u.functionData.functionId;

Js::SourceId sourceId = InvalidSourceId;

Js::LocalFunctionId functionId;

if (calleeFunctionInfo == nullptr)

{

functionId = CallSiteNonFunction;

}

else if (!calleeFunctionInfo->HasBody())

{

Assert(calleeFunction); // calleeFunction can only be passed as null if the calleeFunctionInfo was null (which is checked above)

if (functionBody->GetScriptContext() == calleeFunction->GetScriptContext())

{

sourceId = BuiltInSourceId;

functionId = calleeFunctionInfo->GetLocalFunctionId();

}

else

{

functionId = CallSiteCrossContext;

}

}

else

{

// We can only inline function that are from the same script context. So only record that data

// We're about to call this function so deserialize it right now

FunctionProxy\* calleeFunctionProxy = calleeFunctionInfo->GetFunctionProxy();

if (functionBody->GetScriptContext() == calleeFunctionProxy->GetScriptContext())

{

if (functionBody->GetSecondaryHostSourceContext() == calleeFunctionProxy->GetSecondaryHostSourceContext())

{

if (functionBody->GetHostSourceContext() == calleeFunctionProxy->GetHostSourceContext())

{

sourceId = CurrentSourceId; // Caller and callee in same file

}

else

{

sourceId = (Js::SourceId)calleeFunctionProxy->GetHostSourceContext(); // Caller and callee in different files

}

functionId = calleeFunctionProxy->GetLocalFunctionId();

}

else

{

// Pretend that we are cross context when call is crossing script file.

functionId = CallSiteCrossContext;

}

}

else

{

functionId = CallSiteCrossContext;

}

}

if (oldSourceId == NoSourceId)

{

callSiteInfo[callSiteId].u.functionData.sourceId = sourceId;

callSiteInfo[callSiteId].u.functionData.functionId = functionId;

this->currentInlinerVersion++; // we don't mind if this overflows

}

else if (oldSourceId != sourceId || oldFunctionId != functionId)

{

if (oldFunctionId != CallSiteMixed)

{

this->currentInlinerVersion++; // we don't mind if this overflows

}

if (doInline && IsPolymorphicCallSite(functionId, sourceId, oldFunctionId, oldSourceId))

{

CreatePolymorphicDynamicProfileCallSiteInfo(functionBody, callSiteId, functionId, oldFunctionId, sourceId, oldSourceId);

}

else

{

callSiteInfo[callSiteId].u.functionData.functionId = CallSiteMixed;

}

}

callSiteInfo[callSiteId].isConstructorCall = isConstructorCall;

callSiteInfo[callSiteId].dontInline = !doInline;

callSiteInfo[callSiteId].ldFldInlineCacheId = ldFldInlineCacheId;

}

else

{

Assert(doInline);

Assert(callSiteInfo[callSiteId].isConstructorCall == isConstructorCall);

RecordPolymorphicCallSiteInfo(functionBody, callSiteId, calleeFunctionInfo);

}

return;

}

bool DynamicProfileInfo::IsPolymorphicCallSite(Js::LocalFunctionId curFunctionId, Js::SourceId curSourceId, Js::LocalFunctionId oldFunctionId, Js::SourceId oldSourceId)

{

AssertMsg(oldSourceId != NoSourceId, "There is no previous call in this callsite, we shouldn't be checking for polymorphic");

if (oldSourceId == NoSourceId || oldSourceId == InvalidSourceId || oldSourceId == BuiltInSourceId)

{

return false;

}

if (curFunctionId == CallSiteCrossContext || curFunctionId == CallSiteNonFunction || oldFunctionId == CallSiteMixed || oldFunctionId == CallSiteCrossContext)

{

return false;

}

Assert(oldFunctionId != CallSiteNonFunction);

Assert(curFunctionId != oldFunctionId || curSourceId != oldSourceId);

return true;

}

void DynamicProfileInfo::CreatePolymorphicDynamicProfileCallSiteInfo(FunctionBody \*funcBody, ProfileId callSiteId, Js::LocalFunctionId functionId, Js::LocalFunctionId oldFunctionId, Js::SourceId sourceId, Js::SourceId oldSourceId)

{

PolymorphicCallSiteInfo \*localPolyCallSiteInfo = RecyclerNewStructZ(funcBody->GetScriptContext()->GetRecycler(), PolymorphicCallSiteInfo);

Assert(maxPolymorphicInliningSize >= 2);

localPolyCallSiteInfo->functionIds[0] = oldFunctionId;

localPolyCallSiteInfo->functionIds[1] = functionId;

localPolyCallSiteInfo->sourceIds[0] = oldSourceId;

localPolyCallSiteInfo->sourceIds[1] = sourceId;

localPolyCallSiteInfo->next = funcBody->GetPolymorphicCallSiteInfoHead();

for (int i = 2; i < maxPolymorphicInliningSize; i++)

{

localPolyCallSiteInfo->functionIds[i] = CallSiteNoInfo;

}

callSiteInfo[callSiteId].isPolymorphic = true;

callSiteInfo[callSiteId].u.polymorphicCallSiteInfo = localPolyCallSiteInfo;

funcBody->SetPolymorphicCallSiteInfoHead(localPolyCallSiteInfo);

}

void DynamicProfileInfo::ResetAllPolymorphicCallSiteInfo()

{

if (dynamicProfileFunctionInfo)

{

for (ProfileId i = 0; i < dynamicProfileFunctionInfo->callSiteInfoCount; i++)

{

if (callSiteInfo[i].isPolymorphic)

{

ResetPolymorphicCallSiteInfo(i, CallSiteMixed);

}

}

}

}

void DynamicProfileInfo::ResetPolymorphicCallSiteInfo(ProfileId callSiteId, Js::LocalFunctionId functionId)

{

callSiteInfo[callSiteId].isPolymorphic = false;

callSiteInfo[callSiteId].u.functionData.sourceId = CurrentSourceId;

callSiteInfo[callSiteId].u.functionData.functionId = functionId;

this->currentInlinerVersion++;

}

void DynamicProfileInfo::SetFunctionIdSlotForNewPolymorphicCall(ProfileId callSiteId, Js::LocalFunctionId curFunctionId, Js::SourceId curSourceId, Js::FunctionBody \*inliner)

{

for (int i = 0; i < maxPolymorphicInliningSize; i++)

{

if (callSiteInfo[callSiteId].u.polymorphicCallSiteInfo->functionIds[i] == curFunctionId &&

callSiteInfo[callSiteId].u.polymorphicCallSiteInfo->sourceIds[i] == curSourceId)

{

// we have it already

return;

}

else if (callSiteInfo[callSiteId].u.polymorphicCallSiteInfo->functionIds[i] == CallSiteNoInfo)

{

callSiteInfo[callSiteId].u.polymorphicCallSiteInfo->functionIds[i] = curFunctionId;

callSiteInfo[callSiteId].u.polymorphicCallSiteInfo->sourceIds[i] = curSourceId;

this->currentInlinerVersion++;

return;

}

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.TestTrace.IsEnabled(Js::PolymorphicInlinePhase))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"INLINING (Polymorphic): More than 4 functions at this call site \t callSiteId: %d\t calleeFunctionId: %d TopFunc %s (%s)\n",

callSiteId,

curFunctionId,

inliner->GetDisplayName(),

inliner->GetDebugNumberSet(debugStringBuffer)

);

Output::Flush();

}

#endif

#ifdef PERF\_HINT

if (PHASE\_TRACE1(Js::PerfHintPhase))

{

WritePerfHint(PerfHints::PolymorphicInilineCap, inliner);

}

#endif

// We reached the max allowed to inline, no point in continuing collecting the information. Reset and move on.

ResetPolymorphicCallSiteInfo(callSiteId, CallSiteMixed);

}

void DynamicProfileInfo::RecordPolymorphicCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, FunctionInfo \* calleeFunctionInfo)

{

Js::LocalFunctionId functionId;

if (calleeFunctionInfo == nullptr || !calleeFunctionInfo->HasBody())

{

return ResetPolymorphicCallSiteInfo(callSiteId, CallSiteMixed);

}

// We can only inline function that are from the same script context. So only record that data

// We're about to call this function so deserialize it right now.

FunctionProxy\* calleeFunctionProxy = calleeFunctionInfo->GetFunctionProxy();

if (functionBody->GetScriptContext() == calleeFunctionProxy->GetScriptContext())

{

if (functionBody->GetSecondaryHostSourceContext() == calleeFunctionProxy->GetSecondaryHostSourceContext())

{

Js::SourceId sourceId = (Js::SourceId)calleeFunctionProxy->GetHostSourceContext();

if (functionBody->GetHostSourceContext() == sourceId) // if caller and callee in same file

{

sourceId = CurrentSourceId;

}

functionId = calleeFunctionProxy->GetLocalFunctionId();

SetFunctionIdSlotForNewPolymorphicCall(callSiteId, functionId, sourceId, functionBody);

return;

}

}

// Pretend that we are cross context when call is crossing script file.

ResetPolymorphicCallSiteInfo(callSiteId, CallSiteCrossContext);

}

bool DynamicProfileInfo::HasCallSiteInfo(FunctionBody\* functionBody)

{

SourceContextInfo \*sourceContextInfo = functionBody->GetSourceContextInfo();

return !functionBody->GetScriptContext()->IsNoContextSourceContextInfo(sourceContextInfo);

}

bool DynamicProfileInfo::GetPolymorphicCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, bool \*isConstructorCall, \_\_inout\_ecount(functionBodyArrayLength) FunctionBody\*\* functionBodyArray, uint functionBodyArrayLength)

{

Assert(functionBody);

const auto callSiteCount = functionBody->GetProfiledCallSiteCount();

Assert(callSiteId < callSiteCount);

Assert(HasCallSiteInfo(functionBody));

Assert(functionBodyArray);

Assert(functionBodyArrayLength == DynamicProfileInfo::maxPolymorphicInliningSize);

\*isConstructorCall = callSiteInfo[callSiteId].isConstructorCall;

if (callSiteInfo[callSiteId].dontInline)

{

return false;

}

if (callSiteInfo[callSiteId].isPolymorphic)

{

PolymorphicCallSiteInfo \*polymorphicCallSiteInfo = callSiteInfo[callSiteId].u.polymorphicCallSiteInfo;

for (uint i = 0; i < functionBodyArrayLength; i++)

{

Js::LocalFunctionId localFunctionId;

Js::SourceId localSourceId;

if (!polymorphicCallSiteInfo->GetFunction(i, &localFunctionId, &localSourceId))

{

AssertMsg(i >= 2, "We found at least two function Body");

return true;

}

FunctionBody\* matchedFunctionBody;

if (localSourceId == CurrentSourceId) // caller and callee in same file

{

matchedFunctionBody = functionBody->GetUtf8SourceInfo()->FindFunction(localFunctionId);

if (!matchedFunctionBody)

{

return false;

}

functionBodyArray[i] = matchedFunctionBody;

}

else if (localSourceId == NoSourceId || localSourceId == InvalidSourceId)

{

return false;

}

else

{

// For call across files find the function from the right source

typedef JsUtil::List<RecyclerWeakReference<Utf8SourceInfo>\*, Recycler, false, Js::FreeListedRemovePolicy> SourceList;

SourceList \* sourceList = functionBody->GetScriptContext()->GetSourceList();

bool found = false;

for (int j = 0; j < sourceList->Count() && !found; j++)

{

if (sourceList->IsItemValid(j))

{

Utf8SourceInfo \*srcInfo = sourceList->Item(j)->Get();

if (srcInfo && srcInfo->GetHostSourceContext() == localSourceId)

{

matchedFunctionBody = srcInfo->FindFunction(localFunctionId);

if (!matchedFunctionBody)

{

return false;

}

functionBodyArray[i] = matchedFunctionBody;

found = true;

}

}

}

if (!found)

{

return false;

}

}

}

return true;

}

return false;

}

bool DynamicProfileInfo::HasCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId)

{

Assert(functionBody);

const auto callSiteCount = functionBody->GetProfiledCallSiteCount();

Assert(callSiteId < callSiteCount);

Assert(HasCallSiteInfo(functionBody));

if (callSiteInfo[callSiteId].isPolymorphic)

{

return true;

}

return callSiteInfo[callSiteId].u.functionData.sourceId != NoSourceId;

}

FunctionInfo \* DynamicProfileInfo::GetCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, bool \*isConstructorCall, bool \*isPolymorphicCall)

{

Assert(functionBody);

const auto callSiteCount = functionBody->GetProfiledCallSiteCount();

Assert(callSiteId < callSiteCount);

Assert(HasCallSiteInfo(functionBody));

\*isConstructorCall = callSiteInfo[callSiteId].isConstructorCall;

if (callSiteInfo[callSiteId].dontInline)

{

return nullptr;

}

if (!callSiteInfo[callSiteId].isPolymorphic)

{

Js::SourceId sourceId = callSiteInfo[callSiteId].u.functionData.sourceId;

Js::LocalFunctionId functionId = callSiteInfo[callSiteId].u.functionData.functionId;

if (sourceId == BuiltInSourceId)

{

return JavascriptBuiltInFunction::GetFunctionInfo(functionId);

}

if (sourceId == CurrentSourceId) // caller and callee in same file

{

return functionBody->GetUtf8SourceInfo()->FindFunction(functionId);

}

if (sourceId != NoSourceId && sourceId != InvalidSourceId)

{

// For call across files find the function from the right source

JsUtil::List<RecyclerWeakReference<Utf8SourceInfo>\*, Recycler, false, Js::FreeListedRemovePolicy> \* sourceList = functionBody->GetScriptContext()->GetSourceList();

for (int i = 0; i < sourceList->Count(); i++)

{

if (sourceList->IsItemValid(i))

{

Utf8SourceInfo \*srcInfo = sourceList->Item(i)->Get();

if (srcInfo && srcInfo->GetHostSourceContext() == sourceId)

{

return srcInfo->FindFunction(functionId);

}

}

}

}

}

else

{

\*isPolymorphicCall = true;

}

return nullptr;

}

uint DynamicProfileInfo::GetLdFldCacheIndexFromCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId)

{

Assert(functionBody);

const auto callSiteCount = functionBody->GetProfiledCallSiteCount();

Assert(callSiteId < callSiteCount);

Assert(HasCallSiteInfo(functionBody));

return callSiteInfo[callSiteId].ldFldInlineCacheId;

}

void DynamicProfileInfo::RecordElementLoad(FunctionBody\* functionBody, ProfileId ldElemId, const LdElemInfo& info)

{

Assert(ldElemId < functionBody->GetProfiledLdElemCount());

Assert(info.WasProfiled());

ldElemInfo[ldElemId].Merge(info);

}

void DynamicProfileInfo::RecordElementLoadAsProfiled(FunctionBody \*const functionBody, const ProfileId ldElemId)

{

Assert(ldElemId < functionBody->GetProfiledLdElemCount());

ldElemInfo[ldElemId].wasProfiled = true;

}

void DynamicProfileInfo::RecordElementStore(FunctionBody\* functionBody, ProfileId stElemId, const StElemInfo& info)

{

Assert(stElemId < functionBody->GetProfiledStElemCount());

Assert(info.WasProfiled());

stElemInfo[stElemId].Merge(info);

}

void DynamicProfileInfo::RecordElementStoreAsProfiled(FunctionBody \*const functionBody, const ProfileId stElemId)

{

Assert(stElemId < functionBody->GetProfiledStElemCount());

stElemInfo[stElemId].wasProfiled = true;

}

ArrayCallSiteInfo \* DynamicProfileInfo::GetArrayCallSiteInfo(FunctionBody \*functionBody, ProfileId index) const

{

Assert(index < functionBody->GetProfiledArrayCallSiteCount());

return &arrayCallSiteInfo[index];

}

\_\_inline void DynamicProfileInfo::RecordFieldAccess(FunctionBody\* functionBody, uint fieldAccessId, Var object, FldInfoFlags flags)

{

Assert(fieldAccessId < functionBody->GetProfiledFldCount());

FldInfoFlags oldFlags = fldInfo[fieldAccessId].flags;

if (object) // if not provided, the saved value type is not changed

{

fldInfo[fieldAccessId].valueType = fldInfo[fieldAccessId].valueType.Merge(object);

}

const auto mergedFlags = MergeFldInfoFlags(oldFlags, flags);

fldInfo[fieldAccessId].flags = mergedFlags;

if (flags & FldInfo\_Polymorphic)

{

bits.hasPolymorphicFldAccess = true;

if (!(oldFlags & FldInfo\_Polymorphic))

{

this->SetHasNewPolyFieldAccess(functionBody);

}

if (fldInfo[fieldAccessId].polymorphicInlineCacheUtilization < (PolymorphicInlineCacheUtilizationMaxValue - PolymorphicInlineCacheUtilizationIncrement))

{

fldInfo[fieldAccessId].polymorphicInlineCacheUtilization += PolymorphicInlineCacheUtilizationIncrement;

}

else

{

fldInfo[fieldAccessId].polymorphicInlineCacheUtilization = PolymorphicInlineCacheUtilizationMaxValue;

}

}

else if (flags != FldInfo\_NoInfo &&

fldInfo[fieldAccessId].polymorphicInlineCacheUtilization != PolymorphicInlineCacheUtilizationMaxValue)

{

if (fldInfo[fieldAccessId].polymorphicInlineCacheUtilization > (PolymorphicInlineCacheUtilizationMinValue + PolymorphicInlineCacheUtilizationDecrement))

{

fldInfo[fieldAccessId].polymorphicInlineCacheUtilization -= PolymorphicInlineCacheUtilizationDecrement;

}

else

{

fldInfo[fieldAccessId].polymorphicInlineCacheUtilization = PolymorphicInlineCacheUtilizationMinValue;

}

}

}

\_\_inline void DynamicProfileInfo::RecordDivideResultType(FunctionBody\* body, ProfileId divideId, Var object)

{

Assert(divideId < body->GetProfiledDivOrRemCount());

divideTypeInfo[divideId] = divideTypeInfo[divideId].Merge(object);

}

// We are overloading the value types to store whether it is a mod by power of 2.

// TaggedInt:

\_\_inline void DynamicProfileInfo::RecordModulusOpType(FunctionBody\* body, ProfileId profileId, bool isModByPowerOf2)

{

Assert(profileId < body->GetProfiledDivOrRemCount());

// allow one op of the modulus to be optimized - anyway

if (divideTypeInfo[profileId].IsUninitialized())

{

divideTypeInfo[profileId] = ValueType::GetInt(true);

}

else

{

if (isModByPowerOf2)

{

divideTypeInfo[profileId] = divideTypeInfo[profileId].Merge(ValueType::GetInt(true));

}

else

{

divideTypeInfo[profileId] = divideTypeInfo[profileId].Merge(ValueType::Float);

}

}

}

bool DynamicProfileInfo::IsModulusOpByPowerOf2(FunctionBody\* body, ProfileId profileId) const

{

Assert(profileId < body->GetProfiledDivOrRemCount());

return divideTypeInfo[profileId].IsLikelyTaggedInt();

}

ValueType DynamicProfileInfo::GetDivideResultType(FunctionBody\* body, ProfileId divideId) const

{

Assert(divideId < body->GetProfiledDivOrRemCount());

return divideTypeInfo[divideId];

}

\_\_inline void DynamicProfileInfo::RecordSwitchType(FunctionBody\* body, ProfileId switchId, Var object)

{

Assert(switchId < body->GetProfiledSwitchCount());

switchTypeInfo[switchId] = switchTypeInfo[switchId].Merge(object);

}

ValueType DynamicProfileInfo::GetSwitchType(FunctionBody\* body, ProfileId switchId) const

{

Assert(switchId < body->GetProfiledSwitchCount());

return switchTypeInfo[switchId];

}

void DynamicProfileInfo::SetHasNewPolyFieldAccess(FunctionBody \*functionBody)

{

this->polymorphicCacheState = functionBody->GetScriptContext()->GetThreadContext()->GetNextPolymorphicCacheState();

PHASE\_PRINT\_TRACE(

Js::ObjTypeSpecPhase, functionBody,

L"New profile cache state: %d\n", this->polymorphicCacheState);

}

\_\_inline void DynamicProfileInfo::RecordPolymorphicFieldAccess(FunctionBody\* functionBody, uint fieldAccessId)

{

this->RecordFieldAccess(functionBody, fieldAccessId, nullptr, FldInfo\_Polymorphic);

}

\_\_inline void DynamicProfileInfo::RecordSlotLoad(FunctionBody\* functionBody, ProfileId slotLoadId, Var object)

{

Assert(slotLoadId < functionBody->GetProfiledSlotCount());

slotInfo[slotLoadId] = slotInfo[slotLoadId].Merge(object);

}

FldInfoFlags DynamicProfileInfo::MergeFldInfoFlags(FldInfoFlags oldFlags, FldInfoFlags newFlags)

{

return static\_cast<FldInfoFlags>(oldFlags | newFlags);

}

\_\_inline void DynamicProfileInfo::RecordParameterInfo(FunctionBody \*functionBody, ArgSlot index, Var object)

{

Assert(this->parameterInfo != nullptr);

Assert(index < functionBody->GetProfiledInParamsCount());

parameterInfo[index] = parameterInfo[index].Merge(object);

}

ValueType DynamicProfileInfo::GetParameterInfo(FunctionBody\* functionBody, ArgSlot index) const

{

Assert(this->parameterInfo != nullptr);

Assert(index < functionBody->GetProfiledInParamsCount());

return parameterInfo[index];

}

\_\_inline void DynamicProfileInfo::RecordReturnTypeOnCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, Var object)

{

Assert(callSiteId < functionBody->GetProfiledCallSiteCount());

this->callSiteInfo[callSiteId].returnType = this->callSiteInfo[callSiteId].returnType.Merge(object);

}

\_\_inline void DynamicProfileInfo::RecordReturnType(FunctionBody\* functionBody, ProfileId callSiteId, Var object)

{

Assert(callSiteId < functionBody->GetProfiledReturnTypeCount());

this->returnTypeInfo[callSiteId] = this->returnTypeInfo[callSiteId].Merge(object);

}

ValueType DynamicProfileInfo::GetReturnType(FunctionBody\* functionBody, Js::OpCode opcode, ProfileId callSiteId) const

{

if (opcode < Js::OpCode::ProfiledReturnTypeCallI)

{

Assert(IsProfiledCallOp(opcode));

Assert(callSiteId < functionBody->GetProfiledCallSiteCount());

return this->callSiteInfo[callSiteId].returnType;

}

Assert(IsProfiledReturnTypeOp(opcode));

Assert(callSiteId < functionBody->GetProfiledReturnTypeCount());

return this->returnTypeInfo[callSiteId];

}

\_\_inline void DynamicProfileInfo::RecordThisInfo(Var object, ThisType thisType)

{

this->thisInfo.valueType = this->thisInfo.valueType.Merge(object);

this->thisInfo.thisType = max(this->thisInfo.thisType, thisType);

}

ThisInfo DynamicProfileInfo::GetThisInfo() const

{

return this->thisInfo;

}

void DynamicProfileInfo::RecordLoopImplicitCallFlags(FunctionBody\* functionBody, uint loopNum, ImplicitCallFlags flags)

{

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(functionBody));

Assert(loopNum < functionBody->GetLoopCount());

this->loopImplicitCallFlags[loopNum] = (ImplicitCallFlags)(this->loopImplicitCallFlags[loopNum] | flags);

}

ImplicitCallFlags DynamicProfileInfo::GetLoopImplicitCallFlags(FunctionBody\* functionBody, uint loopNum) const

{

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(functionBody));

Assert(loopNum < functionBody->GetLoopCount());

// Mask out the dispose implicit call. We would bailout on reentrant dispose,

// but it shouldn't affect optimization.

return (ImplicitCallFlags)(this->loopImplicitCallFlags[loopNum] & ImplicitCall\_All);

}

void DynamicProfileInfo::RecordImplicitCallFlags(ImplicitCallFlags flags)

{

this->implicitCallFlags = (ImplicitCallFlags)(this->implicitCallFlags | flags);

}

ImplicitCallFlags DynamicProfileInfo::GetImplicitCallFlags() const

{

// Mask out the dispose implicit call. We would bailout on reentrant dispose,

// but it shouldn't affect optimization.

return (ImplicitCallFlags)(this->implicitCallFlags & ImplicitCall\_All);

}

void DynamicProfileInfo::UpdateFunctionInfo(FunctionBody\* functionBody, Recycler\* recycler)

{

Assert(this->persistsAcrossScriptContexts);

if (!this->dynamicProfileFunctionInfo)

{

this->dynamicProfileFunctionInfo = RecyclerNewStructLeaf(recycler, DynamicProfileFunctionInfo);

}

this->dynamicProfileFunctionInfo->callSiteInfoCount = functionBody->GetProfiledCallSiteCount();

this->dynamicProfileFunctionInfo->paramInfoCount = functionBody->GetProfiledInParamsCount();

this->dynamicProfileFunctionInfo->divCount = functionBody->GetProfiledDivOrRemCount();

this->dynamicProfileFunctionInfo->switchCount = functionBody->GetProfiledSwitchCount();

this->dynamicProfileFunctionInfo->returnTypeInfoCount = functionBody->GetProfiledReturnTypeCount();

this->dynamicProfileFunctionInfo->loopCount = functionBody->GetLoopCount();

this->dynamicProfileFunctionInfo->ldElemInfoCount = functionBody->GetProfiledLdElemCount();

this->dynamicProfileFunctionInfo->stElemInfoCount = functionBody->GetProfiledStElemCount();

this->dynamicProfileFunctionInfo->arrayCallSiteCount = functionBody->GetProfiledArrayCallSiteCount();

this->dynamicProfileFunctionInfo->fldInfoCount = functionBody->GetProfiledFldCount();

this->dynamicProfileFunctionInfo->slotInfoCount = functionBody->GetProfiledSlotCount();

}

void DynamicProfileInfo::Save(ScriptContext \* scriptContext)

{

// For now, we only support our local storage

#ifdef DYNAMIC\_PROFILE\_STORAGE

if (!DynamicProfileStorage::IsEnabled())

{

return;

}

if (scriptContext->GetSourceContextInfoMap() == nullptr)

{

// We don't have savable code

Assert(!scriptContext->GetProfileInfoList() || scriptContext->GetProfileInfoList()->Empty() || scriptContext->GetNoContextSourceContextInfo()->nextLocalFunctionId != 0);

return;

}

DynamicProfileInfo::UpdateSourceDynamicProfileManagers(scriptContext);

scriptContext->GetSourceContextInfoMap()->Map([&](DWORD\_PTR dwHostSourceContext, SourceContextInfo \* sourceContextInfo)

{

if (sourceContextInfo->sourceDynamicProfileManager != nullptr && sourceContextInfo->url != nullptr

&& !sourceContextInfo->IsDynamic())

{

sourceContextInfo->sourceDynamicProfileManager->SaveToDynamicProfileStorage(sourceContextInfo->url);

}

});

#endif

}

bool DynamicProfileInfo::MatchFunctionBody(FunctionBody \* functionBody)

{

// This function is called to set a function body to the dynamic profile loaded from cache.

// Need to verify that the function body matches with the profile info

Assert(this->dynamicProfileFunctionInfo);

if (this->dynamicProfileFunctionInfo->paramInfoCount != functionBody->GetProfiledInParamsCount()

|| this->dynamicProfileFunctionInfo->ldElemInfoCount != functionBody->GetProfiledLdElemCount()

|| this->dynamicProfileFunctionInfo->stElemInfoCount != functionBody->GetProfiledStElemCount()

|| this->dynamicProfileFunctionInfo->arrayCallSiteCount != functionBody->GetProfiledArrayCallSiteCount()

|| this->dynamicProfileFunctionInfo->fldInfoCount != functionBody->GetProfiledFldCount()

|| this->dynamicProfileFunctionInfo->slotInfoCount != functionBody->GetProfiledSlotCount()

|| this->dynamicProfileFunctionInfo->callSiteInfoCount != functionBody->GetProfiledCallSiteCount()

|| this->dynamicProfileFunctionInfo->returnTypeInfoCount != functionBody->GetProfiledReturnTypeCount()

|| this->dynamicProfileFunctionInfo->loopCount != functionBody->GetLoopCount()

|| this->dynamicProfileFunctionInfo->switchCount != functionBody->GetProfiledSwitchCount()

|| this->dynamicProfileFunctionInfo->divCount != functionBody->GetProfiledDivOrRemCount())

{

// Reject, the dynamic profile information doesn't match the function body

return false;

}

#ifdef DYNAMIC\_PROFILE\_STORAGE

this->functionBody = functionBody;

#endif

this->hasFunctionBody = true;

return true;

}

FldInfo \* DynamicProfileInfo::GetFldInfo(FunctionBody\* functionBody, uint fieldAccessId) const

{

Assert(fieldAccessId < functionBody->GetProfiledFldCount());

return &fldInfo[fieldAccessId];

}

ValueType DynamicProfileInfo::GetSlotLoad(FunctionBody\* functionBody, ProfileId slotLoadId) const

{

Assert(slotLoadId < functionBody->GetProfiledSlotCount());

return slotInfo[slotLoadId];

}

FldInfoFlags DynamicProfileInfo::FldInfoFlagsFromCacheType(CacheType cacheType)

{

switch (cacheType)

{

case CacheType\_Local:

return FldInfo\_FromLocal;

case CacheType\_Proto:

return FldInfo\_FromProto;

case CacheType\_LocalWithoutProperty:

return FldInfo\_FromLocalWithoutProperty;

case CacheType\_Getter:

case CacheType\_Setter:

return FldInfo\_FromAccessor;

default:

return FldInfo\_NoInfo;

}

}

FldInfoFlags DynamicProfileInfo::FldInfoFlagsFromSlotType(SlotType slotType)

{

switch (slotType)

{

case SlotType\_Inline:

return FldInfo\_FromInlineSlots;

case SlotType\_Aux:

return FldInfo\_FromAuxSlots;

default:

return FldInfo\_NoInfo;

}

}

#if DBG\_DUMP

void DynamicProfileInfo::DumpProfiledValue(wchar\_t const \* name, CallSiteInfo \* callSiteInfo, uint count)

{

if (count != 0)

{

Output::Print(L" %-16s(%2d):", name, count);

for (uint i = 0; i < count; i++)

{

Output::Print(i != 0 && (i % 10) == 0 ? L"\n " : L" ");

Output::Print(L"%2d:", i);

if (!callSiteInfo[i].isPolymorphic)

{

switch (callSiteInfo[i].u.functionData.sourceId)

{

case NoSourceId:

Output::Print(L" ????");

break;

case BuiltInSourceId:

Output::Print(L" b%03d", callSiteInfo[i].u.functionData.functionId);

break;

case InvalidSourceId:

if (callSiteInfo[i].u.functionData.functionId == CallSiteMixed)

{

Output::Print(L" mix");

}

else if (callSiteInfo[i].u.functionData.functionId == CallSiteCrossContext)

{

Output::Print(L" x");

}

else if (callSiteInfo[i].u.functionData.functionId == CallSiteNonFunction)

{

Output::Print(L" !fn");

}

else

{

Assert(false);

}

break;

default:

Output::Print(L" %4d:%4d", callSiteInfo[i].u.functionData.sourceId, callSiteInfo[i].u.functionData.functionId);

break;

};

}

else

{

Output::Print(L" poly");

for (int j = 0; j < DynamicProfileInfo::maxPolymorphicInliningSize; j++)

{

if (callSiteInfo[i].u.polymorphicCallSiteInfo->functionIds[j] != CallSiteNoInfo)

{

Output::Print(L" %4d:%4d", callSiteInfo[i].u.polymorphicCallSiteInfo->sourceIds[j], callSiteInfo[i].u.polymorphicCallSiteInfo->functionIds[j]);

}

}

}

}

Output::Print(L"\n");

Output::Print(L" %-16s(%2d):", L"Callsite RetType", count);

for (uint i = 0; i < count; i++)

{

Output::Print(i != 0 && (i % 10) == 0 ? L"\n " : L" ");

Output::Print(L"%2d:", i);

char returnTypeStr[VALUE\_TYPE\_MAX\_STRING\_SIZE];

callSiteInfo[i].returnType.ToString(returnTypeStr);

Output::Print(L" %S", returnTypeStr);

}

Output::Print(L"\n");

}

}

void DynamicProfileInfo::DumpProfiledValue(wchar\_t const \* name, ArrayCallSiteInfo \* arrayCallSiteInfo, uint count)

{

if (count != 0)

{

Output::Print(L" %-16s(%2d):", name, count);

Output::Print(L"\n");

for (uint i = 0; i < count; i++)

{

Output::Print(i != 0 && (i % 10) == 0 ? L"\n " : L" ");

Output::Print(L"%4d:", i);

Output::Print(L" Function Number: %2d, CallSite Number: %2d, IsNativeIntArray: %2d, IsNativeFloatArray: %2d",

arrayCallSiteInfo[i].functionNumber, arrayCallSiteInfo[i].callSiteNumber, !arrayCallSiteInfo[i].isNotNativeInt, !arrayCallSiteInfo[i].isNotNativeFloat);

Output::Print(L"\n");

}

Output::Print(L"\n");

}

}

void DynamicProfileInfo::DumpProfiledValue(wchar\_t const \* name, ValueType \* value, uint count)

{

if (count != 0)

{

Output::Print(L" %-16s(%2d):", name, count);

for (uint i = 0; i < count; i++)

{

Output::Print(i != 0 && (i % 10) == 0 ? L"\n " : L" ");

Output::Print(L"%2d:", i);

char valueStr[VALUE\_TYPE\_MAX\_STRING\_SIZE];

value[i].ToString(valueStr);

Output::Print(L" %S", valueStr);

}

Output::Print(L"\n");

}

}

void DynamicProfileInfo::DumpProfiledValue(wchar\_t const \* name, uint \* value, uint count)

{

if (count != 0)

{

Output::Print(L" %-16s(%2d):", name, count);

for (uint i = 0; i < count; i++)

{

Output::Print(i != 0 && (i % 10) == 0 ? L"\n " : L" ");

Output::Print(L"%2d:%-4d", i, value[i]);

}

Output::Print(L"\n");

}

}

wchar\_t const \* DynamicProfileInfo::GetImplicitCallFlagsString(ImplicitCallFlags flags)

{

// Mask out the dispose implicit call. We would bailout on reentrant dispose,

// but it shouldn't affect optimization

flags = (ImplicitCallFlags)(flags & ImplicitCall\_All);

return flags == ImplicitCall\_HasNoInfo ? L"???" : flags == ImplicitCall\_None ? L"no" : L"yes";

}

void DynamicProfileInfo::DumpProfiledValue(wchar\_t const \* name, ImplicitCallFlags \* loopImplicitCallFlags, uint count)

{

if (count != 0)

{

Output::Print(L" %-16s(%2d):", name, count);

for (uint i = 0; i < count; i++)

{

Output::Print(i != 0 && (i % 10) == 0 ? L"\n " : L" ");

Output::Print(L"%2d:%-4s", i, GetImplicitCallFlagsString(loopImplicitCallFlags[i]));

}

Output::Print(L"\n");

}

}

bool DynamicProfileInfo::IsProfiledCallOp(OpCode op)

{

return Js::OpCodeUtil::IsProfiledCallOp(op) || Js::OpCodeUtil::IsProfiledCallOpWithICIndex(op);

}

bool DynamicProfileInfo::IsProfiledReturnTypeOp(OpCode op)

{

return Js::OpCodeUtil::IsProfiledReturnTypeCallOp(op);

}

template<class TData, class FGetValueType>

void DynamicProfileInfo::DumpProfiledValuesGroupedByValue(

const wchar\_t \*const name,

const TData \*const data,

const uint count,

const FGetValueType GetValueType,

ArenaAllocator \*const dynamicProfileInfoAllocator)

{

JsUtil::BaseDictionary<ValueType, bool, ArenaAllocator> uniqueValueTypes(dynamicProfileInfoAllocator);

for (uint i = 0; i < count; i++)

{

const ValueType valueType(GetValueType(data, i));

if (!valueType.IsUninitialized())

{

uniqueValueTypes.Item(valueType, false);

}

}

uniqueValueTypes.Map([&](const ValueType groupValueType, const bool)

{

bool header = true;

uint lastTempFld = (uint)-1;

for (uint i = 0; i < count; i++)

{

const ValueType valueType(GetValueType(data, i));

if (valueType == groupValueType)

{

if (lastTempFld == (uint)-1)

{

if (header)

{

char valueTypeStr[VALUE\_TYPE\_MAX\_STRING\_SIZE];

valueType.ToString(valueTypeStr);

Output::Print(L" %s %S", name, valueTypeStr);

Output::SkipToColumn(24);

Output::Print(L": %d", i);

}

else

{

Output::Print(L", %d", i);

}

header = false;

lastTempFld = i;

}

}

else

{

if (lastTempFld != (uint)-1)

{

if (lastTempFld != i - 1)

{

Output::Print(L"-%d", i - 1);

}

lastTempFld = (uint)-1;

}

}

}

if (lastTempFld != (uint)-1 && lastTempFld != count - 1)

{

Output::Print(L"-%d\n", count - 1);

}

else if (!header)

{

Output::Print(L"\n");

}

});

}

void DynamicProfileInfo::DumpFldInfoFlags(wchar\_t const \* name, FldInfo \* fldInfo, uint count, FldInfoFlags value, wchar\_t const \* valueName)

{

bool header = true;

uint lastTempFld = (uint)-1;

for (uint i = 0; i < count; i++)

{

if (fldInfo[i].flags & value)

{

if (lastTempFld == (uint)-1)

{

if (header)

{

Output::Print(L" %s %s", name, valueName);

Output::SkipToColumn(24);

Output::Print(L": %d", i);

}

else

{

Output::Print(L", %d", i);

}

header = false;

lastTempFld = i;

}

}

else

{

if (lastTempFld != (uint)-1)

{

if (lastTempFld != i - 1)

{

Output::Print(L"-%d", i - 1);

}

lastTempFld = (uint)-1;

}

}

}

if (lastTempFld != (uint)-1 && lastTempFld != count - 1)

{

Output::Print(L"-%d\n", count - 1);

}

else if (!header)

{

Output::Print(L"\n");

}

}

void DynamicProfileInfo::DumpLoopInfo(FunctionBody \*fbody)

{

if (fbody->DoJITLoopBody())

{

uint count = fbody->GetLoopCount();

Output::Print(L" %-16s(%2d):", L"Loops", count);

for (uint i = 0; i < count; i++)

{

Output::Print(i != 0 && (i % 10) == 0 ? L"\n " : L" ");

Output::Print(L"%2d:%-4d", i, fbody->GetLoopHeader(i)->interpretCount);

}

Output::Print(L"\n");

Output::Print(L" %-16s(%2d):", L"Loops JIT", count);

for (uint i = 0; i < count; i++)

{

Output::Print(i != 0 && (i % 10) == 0 ? L"\n " : L" ");

Output::Print(L"%2d:%-4d", i, fbody->GetLoopHeader(i)->nativeCount);

}

Output::Print(L"\n");

}

}

void DynamicProfileInfo::Dump(FunctionBody\* functionBody, ArenaAllocator \* dynamicProfileInfoAllocator)

{

functionBody->DumpFunctionId(true);

Js::ArgSlot paramcount = functionBody->GetProfiledInParamsCount();

Output::Print(L": %-20s Interpreted:%6d, Param:%2d, ImpCall:%s, Callsite:%3d, ReturnType:%3d, LdElem:%3d, StElem:%3d, Fld%3d\n",

functionBody->GetDisplayName(), functionBody->interpretedCount, paramcount, DynamicProfileInfo::GetImplicitCallFlagsString(this->GetImplicitCallFlags()),

functionBody->GetProfiledCallSiteCount(),

functionBody->GetProfiledReturnTypeCount(),

functionBody->GetProfiledLdElemCount(),

functionBody->GetProfiledStElemCount(),

functionBody->GetProfiledFldCount());

if (Configuration::Global.flags.Verbose)

{

DumpProfiledValue(L"Div result type", this->divideTypeInfo, functionBody->GetProfiledDivOrRemCount());

DumpProfiledValue(L"Switch opt type", this->switchTypeInfo, functionBody->GetProfiledSwitchCount());

DumpProfiledValue(L"Param type", this->parameterInfo, paramcount);

DumpProfiledValue(L"Callsite", this->callSiteInfo, functionBody->GetProfiledCallSiteCount());

DumpProfiledValue(L"ArrayCallSite", this->arrayCallSiteInfo, functionBody->GetProfiledArrayCallSiteCount());

DumpProfiledValue(L"Return type", this->returnTypeInfo, functionBody->GetProfiledReturnTypeCount());

if (dynamicProfileInfoAllocator)

{

DumpProfiledValuesGroupedByValue(

L"Element load",

this->ldElemInfo,

this->functionBody->GetProfiledLdElemCount(),

[](const LdElemInfo \*const ldElemInfo, const uint i) -> ValueType

{

return ldElemInfo[i].GetElementType();

},

dynamicProfileInfoAllocator);

DumpProfiledValuesGroupedByValue(

L"Fld",

this->fldInfo,

functionBody->GetProfiledFldCount(),

[](const FldInfo \*const fldInfos, const uint i) -> ValueType

{

return fldInfos[i].valueType;

},

dynamicProfileInfoAllocator);

}

DumpFldInfoFlags(L"Fld", this->fldInfo, functionBody->GetProfiledFldCount(), FldInfo\_FromLocal, L"FldInfo\_FromLocal");

DumpFldInfoFlags(L"Fld", this->fldInfo, functionBody->GetProfiledFldCount(), FldInfo\_FromProto, L"FldInfo\_FromProto");

DumpFldInfoFlags(L"Fld", this->fldInfo, functionBody->GetProfiledFldCount(), FldInfo\_FromLocalWithoutProperty, L"FldInfo\_FromLocalWithoutProperty");

DumpFldInfoFlags(L"Fld", this->fldInfo, functionBody->GetProfiledFldCount(), FldInfo\_FromAccessor, L"FldInfo\_FromAccessor");

DumpFldInfoFlags(L"Fld", this->fldInfo, functionBody->GetProfiledFldCount(), FldInfo\_Polymorphic, L"FldInfo\_Polymorphic");

DumpFldInfoFlags(L"Fld", this->fldInfo, functionBody->GetProfiledFldCount(), FldInfo\_FromInlineSlots, L"FldInfo\_FromInlineSlots");

DumpFldInfoFlags(L"Fld", this->fldInfo, functionBody->GetProfiledFldCount(), FldInfo\_FromAuxSlots, L"FldInfo\_FromAuxSlots");

DumpLoopInfo(functionBody);

if (DynamicProfileInfo::EnableImplicitCallFlags(functionBody))

{

DumpProfiledValue(L"Loop Imp Call", this->loopImplicitCallFlags, functionBody->GetLoopCount());

}

if (functionBody->GetLoopCount())

{

Output::Print(L" Loop Flags:\n");

for (uint i = 0; i < functionBody->GetLoopCount(); ++i)

{

Output::Print(L" Loop %d:\n", i);

LoopFlags lf = this->GetLoopFlags(i);

Output::Print(

L" isInterpreted : %s\n"

L" memopMinCountReached : %s\n",

IsTrueOrFalse(lf.isInterpreted),

IsTrueOrFalse(lf.memopMinCountReached)

);

}

}

Output::Print(

L" Settings:"

L" disableAggressiveIntTypeSpec : %s"

L" disableAggressiveIntTypeSpec\_jitLoopBody : %s"

L" disableAggressiveMulIntTypeSpec : %s"

L" disableAggressiveMulIntTypeSpec\_jitLoopBody : %s"

L" disableDivIntTypeSpec : %s"

L" disableDivIntTypeSpec\_jitLoopBody : %s"

L" disableLossyIntTypeSpec : %s"

L" disableMemOp : %s"

L" disableTrackIntOverflow : %s"

L" disableFloatTypeSpec : %s"

L" disableCheckThis : %s"

L" disableArrayCheckHoist : %s"

L" disableArrayCheckHoist\_jitLoopBody : %s"

L" disableArrayMissingValueCheckHoist : %s"

L" disableArrayMissingValueCheckHoist\_jitLoopBody : %s"

L" disableJsArraySegmentHoist : %s"

L" disableJsArraySegmentHoist\_jitLoopBody : %s"

L" disableArrayLengthHoist : %s"

L" disableArrayLengthHoist\_jitLoopBody : %s"

L" disableTypedArrayTypeSpec: %s"

L" disableTypedArrayTypeSpec\_jitLoopBody: %s"

L" disableLdLenIntSpec: %s"

L" disableBoundCheckHoist : %s"

L" disableBoundCheckHoist\_jitLoopBody : %s"

L" disableLoopCountBasedBoundCheckHoist : %s"

L" disableLoopCountBasedBoundCheckHoist\_jitLoopBody : %s"

L" hasPolymorphicFldAccess : %s"

L" hasLdFldCallSite: %s"

L" disableFloorInlining: %s"

L" disableNoProfileBailouts: %s"

L" disableSwitchOpt : %s"

L" disableEquivalentObjTypeSpec : %s\n"

L" disableObjTypeSpec\_jitLoopBody : %s\n",

IsTrueOrFalse(this->bits.disableAggressiveIntTypeSpec),

IsTrueOrFalse(this->bits.disableAggressiveIntTypeSpec\_jitLoopBody),

IsTrueOrFalse(this->bits.disableAggressiveMulIntTypeSpec),

IsTrueOrFalse(this->bits.disableAggressiveMulIntTypeSpec\_jitLoopBody),

IsTrueOrFalse(this->bits.disableDivIntTypeSpec),

IsTrueOrFalse(this->bits.disableDivIntTypeSpec\_jitLoopBody),

IsTrueOrFalse(this->bits.disableLossyIntTypeSpec),

IsTrueOrFalse(this->bits.disableMemOp),

IsTrueOrFalse(this->bits.disableTrackCompoundedIntOverflow),

IsTrueOrFalse(this->bits.disableFloatTypeSpec),

IsTrueOrFalse(this->bits.disableCheckThis),

IsTrueOrFalse(this->bits.disableArrayCheckHoist),

IsTrueOrFalse(this->bits.disableArrayCheckHoist\_jitLoopBody),

IsTrueOrFalse(this->bits.disableArrayMissingValueCheckHoist),

IsTrueOrFalse(this->bits.disableArrayMissingValueCheckHoist\_jitLoopBody),

IsTrueOrFalse(this->bits.disableJsArraySegmentHoist),

IsTrueOrFalse(this->bits.disableJsArraySegmentHoist\_jitLoopBody),

IsTrueOrFalse(this->bits.disableArrayLengthHoist),

IsTrueOrFalse(this->bits.disableArrayLengthHoist\_jitLoopBody),

IsTrueOrFalse(this->bits.disableTypedArrayTypeSpec),

IsTrueOrFalse(this->bits.disableTypedArrayTypeSpec\_jitLoopBody),

IsTrueOrFalse(this->bits.disableLdLenIntSpec),

IsTrueOrFalse(this->bits.disableBoundCheckHoist),

IsTrueOrFalse(this->bits.disableBoundCheckHoist\_jitLoopBody),

IsTrueOrFalse(this->bits.disableLoopCountBasedBoundCheckHoist),

IsTrueOrFalse(this->bits.disableLoopCountBasedBoundCheckHoist\_jitLoopBody),

IsTrueOrFalse(this->bits.hasPolymorphicFldAccess),

IsTrueOrFalse(this->bits.hasLdFldCallSite),

IsTrueOrFalse(this->bits.disableFloorInlining),

IsTrueOrFalse(this->bits.disableNoProfileBailouts),

IsTrueOrFalse(this->bits.disableSwitchOpt),

IsTrueOrFalse(this->bits.disableEquivalentObjTypeSpec),

IsTrueOrFalse(this->bits.disableObjTypeSpec\_jitLoopBody));

}

}

void DynamicProfileInfo::DumpList(SListBase<DynamicProfileInfo \*> \* profileInfoList, ArenaAllocator \* dynamicProfileInfoAllocator)

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_DisableCheck);

if (Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase))

{

FOREACH\_SLISTBASE\_ENTRY(DynamicProfileInfo \*, info, profileInfoList)

{

if (Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase, info->GetFunctionBody()->GetSourceContextId(), info->GetFunctionBody()->GetLocalFunctionId()))

{

info->Dump(info->GetFunctionBody(), dynamicProfileInfoAllocator);

}

}

NEXT\_SLISTBASE\_ENTRY;

}

if (Configuration::Global.flags.Dump.IsEnabled(JITLoopBodyPhase) && !Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase))

{

FOREACH\_SLISTBASE\_ENTRY(DynamicProfileInfo \*, info, profileInfoList)

{

if (info->functionBody->GetLoopCount() > 0)

{

info->functionBody->DumpFunctionId(true);

Output::Print(L": %-20s\n", info->functionBody->GetDisplayName());

DumpLoopInfo(info->functionBody);

}

}

NEXT\_SLISTBASE\_ENTRY;

}

if (PHASE\_STATS1(DynamicProfilePhase))

{

uint estimatedSavedBytes = sizeof(uint); // count of functions

uint functionSaved = 0;

uint loopSaved = 0;

uint callSiteSaved = 0;

uint elementAccessSaved = 0;

uint fldAccessSaved = 0;

FOREACH\_SLISTBASE\_ENTRY(DynamicProfileInfo \*, info, profileInfoList)

{

bool hasHotLoop = false;

if (info->functionBody->DoJITLoopBody())

{

for (uint i = 0; i < info->functionBody->GetLoopCount(); i++)

{

if (info->functionBody->GetLoopHeader(i)->interpretCount >= 10)

{

hasHotLoop = true;

break;

}

}

}

if (hasHotLoop || info->functionBody->interpretedCount >= 10)

{

functionSaved++;

loopSaved += info->functionBody->GetLoopCount();

estimatedSavedBytes += sizeof(uint) \* 5; // function number, loop count, call site count, local array, temp array

estimatedSavedBytes += (info->functionBody->GetLoopCount() + 7) / 8; // hot loop bit vector

estimatedSavedBytes += (info->functionBody->GetProfiledCallSiteCount() + 7) / 8; // call site bit vector

// call site function number

for (ProfileId i = 0; i < info->functionBody->GetProfiledCallSiteCount(); i++)

{

// TODO poly

if ((info->callSiteInfo[i].u.functionData.sourceId != NoSourceId) && (info->callSiteInfo[i].u.functionData.sourceId != InvalidSourceId))

{

estimatedSavedBytes += sizeof(CallSiteInfo);

callSiteSaved++;

}

}

elementAccessSaved += info->functionBody->GetProfiledLdElemCount() + info->functionBody->GetProfiledStElemCount();

fldAccessSaved += info->functionBody->GetProfiledFldCount();

estimatedSavedBytes += (info->functionBody->GetProfiledLdElemCount() + info->functionBody->GetProfiledStElemCount() + 7) / 8; // temp array access

}

}

NEXT\_SLISTBASE\_ENTRY;

if (estimatedSavedBytes != sizeof(uint))

{

Output::Print(L"Estimated save size (Memory used): %6d (%6d): %3d %3d %4d %4d %3d\n",

estimatedSavedBytes, dynamicProfileInfoAllocator->Size(), functionSaved, loopSaved, callSiteSaved,

elementAccessSaved, fldAccessSaved);

}

}

}

void DynamicProfileInfo::DumpScriptContext(ScriptContext \* scriptContext)

{

if (Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase))

{

Output::Print(L"Sources:\n");

if (scriptContext->GetSourceContextInfoMap() != nullptr)

{

scriptContext->GetSourceContextInfoMap()->Map([&](DWORD\_PTR dwHostSourceContext, SourceContextInfo \* sourceContextInfo)

{

if (sourceContextInfo->sourceContextId != Js::Constants::NoSourceContext)

{

Output::Print(L"%2d: %s (Function count: %d)\n", sourceContextInfo->sourceContextId, sourceContextInfo->url, sourceContextInfo->nextLocalFunctionId);

}

});

}

if (scriptContext->GetDynamicSourceContextInfoMap() != nullptr)

{

scriptContext->GetDynamicSourceContextInfoMap()->Map([&](DWORD\_PTR dwHostSourceContext, SourceContextInfo \* sourceContextInfo)

{

Output::Print(L"%2d: %d (Dynamic) (Function count: %d)\n", sourceContextInfo->sourceContextId, sourceContextInfo->hash, sourceContextInfo->nextLocalFunctionId);

});

}

}

DynamicProfileInfo::DumpList(scriptContext->GetProfileInfoList(), scriptContext->DynamicProfileInfoAllocator());

Output::Flush();

}

#endif

#ifdef DYNAMIC\_PROFILE\_STORAGE

#if DBG\_DUMP

void BufferWriter::Log(DynamicProfileInfo\* info)

{

if (Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase, info->GetFunctionBody()->GetSourceContextId(), info->GetFunctionBody()->GetLocalFunctionId()))

{

Output::Print(L"Saving:");

info->Dump(info->GetFunctionBody());

}

}

#endif

template <typename T>

bool DynamicProfileInfo::Serialize(T \* writer)

{

#if DBG\_DUMP

writer->Log(this);

#endif

FunctionBody \* functionBody = this->GetFunctionBody();

Js::ArgSlot paramInfoCount = functionBody->GetProfiledInParamsCount();

if (!writer->Write(functionBody->GetLocalFunctionId())

|| !writer->Write(paramInfoCount)

|| !writer->WriteArray(this->parameterInfo, paramInfoCount)

|| !writer->Write(functionBody->GetProfiledLdElemCount())

|| !writer->WriteArray(this->ldElemInfo, functionBody->GetProfiledLdElemCount())

|| !writer->Write(functionBody->GetProfiledStElemCount())

|| !writer->WriteArray(this->stElemInfo, functionBody->GetProfiledStElemCount())

|| !writer->Write(functionBody->GetProfiledArrayCallSiteCount())

|| !writer->WriteArray(this->arrayCallSiteInfo, functionBody->GetProfiledArrayCallSiteCount())

|| !writer->Write(functionBody->GetProfiledFldCount())

|| !writer->WriteArray(this->fldInfo, functionBody->GetProfiledFldCount())

|| !writer->Write(functionBody->GetProfiledSlotCount())

|| !writer->WriteArray(this->slotInfo, functionBody->GetProfiledSlotCount())

|| !writer->Write(functionBody->GetProfiledCallSiteCount())

|| !writer->WriteArray(this->callSiteInfo, functionBody->GetProfiledCallSiteCount())

|| !writer->Write(functionBody->GetProfiledDivOrRemCount())

|| !writer->WriteArray(this->divideTypeInfo, functionBody->GetProfiledDivOrRemCount())

|| !writer->Write(functionBody->GetProfiledSwitchCount())

|| !writer->WriteArray(this->switchTypeInfo, functionBody->GetProfiledSwitchCount())

|| !writer->Write(functionBody->GetProfiledReturnTypeCount())

|| !writer->WriteArray(this->returnTypeInfo, functionBody->GetProfiledReturnTypeCount())

|| !writer->Write(functionBody->GetLoopCount())

|| !writer->WriteArray(this->loopImplicitCallFlags, functionBody->GetLoopCount())

|| !writer->Write(this->implicitCallFlags)

|| !writer->Write(this->thisInfo)

|| !writer->Write(this->bits)

|| !writer->Write(this->m\_recursiveInlineInfo)

|| (this->loopFlags && !writer->WriteArray(this->loopFlags->GetData(), this->loopFlags->WordCount())))

{

return false;

}

return true;

}

template <typename T>

DynamicProfileInfo \* DynamicProfileInfo::Deserialize(T \* reader, Recycler\* recycler, Js::LocalFunctionId \* functionId)

{

Js::ArgSlot paramInfoCount = 0;

ProfileId ldElemInfoCount = 0;

ProfileId stElemInfoCount = 0;

ProfileId arrayCallSiteCount = 0;

ProfileId slotInfoCount = 0;

ProfileId callSiteInfoCount = 0;

ProfileId returnTypeInfoCount = 0;

ProfileId divCount = 0;

ProfileId switchCount = 0;

uint fldInfoCount = 0;

uint loopCount = 0;

ValueType \* paramInfo = nullptr;

LdElemInfo \* ldElemInfo = nullptr;

StElemInfo \* stElemInfo = nullptr;

ArrayCallSiteInfo \* arrayCallSiteInfo = nullptr;

FldInfo \* fldInfo = nullptr;

ValueType \* slotInfo = nullptr;

CallSiteInfo \* callSiteInfo = nullptr;

ValueType \* divTypeInfo = nullptr;

ValueType \* switchTypeInfo = nullptr;

ValueType \* returnTypeInfo = nullptr;

ImplicitCallFlags \* loopImplicitCallFlags = nullptr;

BVFixed \* loopFlags = nullptr;

ImplicitCallFlags implicitCallFlags;

ThisInfo thisInfo;

Bits bits;

uint32 recursiveInlineInfo = 0;

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_OutOfMemory);

if (!reader->Read(functionId))

{

return nullptr;

}

if (!reader->Read(&paramInfoCount))

{

return nullptr;

}

if (paramInfoCount != 0)

{

paramInfo = RecyclerNewArrayLeaf(recycler, ValueType, paramInfoCount);

if (!reader->ReadArray(paramInfo, paramInfoCount))

{

goto Error;

}

}

if (!reader->Read(&ldElemInfoCount))

{

goto Error;

}

if (ldElemInfoCount != 0)

{

ldElemInfo = RecyclerNewArrayLeaf(recycler, LdElemInfo, ldElemInfoCount);

if (!reader->ReadArray(ldElemInfo, ldElemInfoCount))

{

goto Error;

}

}

if (!reader->Read(&stElemInfoCount))

{

goto Error;

}

if (stElemInfoCount != 0)

{

stElemInfo = RecyclerNewArrayLeaf(recycler, StElemInfo, stElemInfoCount);

if (!reader->ReadArray(stElemInfo, stElemInfoCount))

{

goto Error;

}

}

if (!reader->Read(&arrayCallSiteCount))

{

goto Error;

}

if (arrayCallSiteCount != 0)

{

arrayCallSiteInfo = RecyclerNewArrayLeaf(recycler, ArrayCallSiteInfo, arrayCallSiteCount);

if (!reader->ReadArray(arrayCallSiteInfo, arrayCallSiteCount))

{

goto Error;

}

}

if (!reader->Read(&fldInfoCount))

{

goto Error;

}

if (fldInfoCount != 0)

{

fldInfo = RecyclerNewArrayLeaf(recycler, FldInfo, fldInfoCount);

if (!reader->ReadArray(fldInfo, fldInfoCount))

{

goto Error;

}

}

if (!reader->Read(&slotInfoCount))

{

goto Error;

}

if (slotInfoCount != 0)

{

slotInfo = RecyclerNewArrayLeaf(recycler, ValueType, slotInfoCount);

if (!reader->ReadArray(slotInfo, slotInfoCount))

{

goto Error;

}

}

if (!reader->Read(&callSiteInfoCount))

{

goto Error;

}

if (callSiteInfoCount != 0)

{

callSiteInfo = RecyclerNewArrayLeaf(recycler, CallSiteInfo, callSiteInfoCount);

if (!reader->ReadArray(callSiteInfo, callSiteInfoCount))

{

goto Error;

}

}

if (!reader->Read(&divCount))

{

goto Error;

}

if (divCount != 0)

{

divTypeInfo = RecyclerNewArrayLeaf(recycler, ValueType, divCount);

if (!reader->ReadArray(divTypeInfo, divCount))

{

goto Error;

}

}

if (!reader->Read(&switchCount))

{

goto Error;

}

if (switchCount != 0)

{

switchTypeInfo = RecyclerNewArrayLeaf(recycler, ValueType, switchCount);

if (!reader->ReadArray(switchTypeInfo, switchCount))

{

goto Error;

}

}

if (!reader->Read(&returnTypeInfoCount))

{

goto Error;

}

if (returnTypeInfoCount != 0)

{

returnTypeInfo = RecyclerNewArrayLeaf(recycler, ValueType, returnTypeInfoCount);

if (!reader->ReadArray(returnTypeInfo, returnTypeInfoCount))

{

goto Error;

}

}

if (!reader->Read(&loopCount))

{

goto Error;

}

if (loopCount != 0)

{

loopImplicitCallFlags = RecyclerNewArrayLeaf(recycler, ImplicitCallFlags, loopCount);

if (!reader->ReadArray(loopImplicitCallFlags, loopCount))

{

goto Error;

}

}

if (!reader->Read(&implicitCallFlags) ||

!reader->Read(&thisInfo) ||

!reader->Read(&bits) ||

!reader->Read(&recursiveInlineInfo))

{

goto Error;

}

if (loopCount != 0)

{

loopFlags = BVFixed::New(loopCount \* LoopFlags::COUNT, recycler);

if (!reader->ReadArray(loopFlags->GetData(), loopFlags->WordCount()))

{

goto Error;

}

}

DynamicProfileFunctionInfo \* dynamicProfileFunctionInfo = RecyclerNewStructLeaf(recycler, DynamicProfileFunctionInfo);

dynamicProfileFunctionInfo->paramInfoCount = paramInfoCount;

dynamicProfileFunctionInfo->ldElemInfoCount = ldElemInfoCount;

dynamicProfileFunctionInfo->stElemInfoCount = stElemInfoCount;

dynamicProfileFunctionInfo->arrayCallSiteCount = arrayCallSiteCount;

dynamicProfileFunctionInfo->fldInfoCount = fldInfoCount;

dynamicProfileFunctionInfo->slotInfoCount = slotInfoCount;

dynamicProfileFunctionInfo->callSiteInfoCount = callSiteInfoCount;

dynamicProfileFunctionInfo->divCount = divCount;

dynamicProfileFunctionInfo->switchCount = switchCount;

dynamicProfileFunctionInfo->returnTypeInfoCount = returnTypeInfoCount;

dynamicProfileFunctionInfo->loopCount = loopCount;

DynamicProfileInfo \* dynamicProfileInfo = RecyclerNew(recycler, DynamicProfileInfo);

dynamicProfileInfo->dynamicProfileFunctionInfo = dynamicProfileFunctionInfo;

dynamicProfileInfo->parameterInfo = paramInfo;

dynamicProfileInfo->ldElemInfo = ldElemInfo;

dynamicProfileInfo->stElemInfo = stElemInfo;

dynamicProfileInfo->arrayCallSiteInfo = arrayCallSiteInfo;

dynamicProfileInfo->fldInfo = fldInfo;

dynamicProfileInfo->slotInfo = slotInfo;

dynamicProfileInfo->callSiteInfo = callSiteInfo;

dynamicProfileInfo->divideTypeInfo = divTypeInfo;

dynamicProfileInfo->switchTypeInfo = switchTypeInfo;

dynamicProfileInfo->returnTypeInfo = returnTypeInfo;

dynamicProfileInfo->loopImplicitCallFlags = loopImplicitCallFlags;

dynamicProfileInfo->implicitCallFlags = implicitCallFlags;

dynamicProfileInfo->loopFlags = loopFlags;

dynamicProfileInfo->thisInfo = thisInfo;

dynamicProfileInfo->bits = bits;

dynamicProfileInfo->m\_recursiveInlineInfo = recursiveInlineInfo;

// Fixed functions and object type data is not serialized. There is no point in trying to serialize polymorphic call site info.

dynamicProfileInfo->ResetAllPolymorphicCallSiteInfo();

return dynamicProfileInfo;

}

catch (OutOfMemoryException)

{

}

Error:

return nullptr;

}

// Explicit instantiations - to force the compiler to generate these - so they can be referenced from other compilation units.

template DynamicProfileInfo \* DynamicProfileInfo::Deserialize<BufferReader>(BufferReader\*, Recycler\*, Js::LocalFunctionId \*);

template bool DynamicProfileInfo::Serialize<BufferSizeCounter>(BufferSizeCounter\*);

template bool DynamicProfileInfo::Serialize<BufferWriter>(BufferWriter\*);

void DynamicProfileInfo::UpdateSourceDynamicProfileManagers(ScriptContext \* scriptContext)

{

// We don't clear old dynamic data here, because if a function is inlined, it will never go through the

// EnsureDynamicProfileThunk and thus not appear in the list. We would want to keep those data as well.

// Just save/update the data from function that has execute.

// That means that the data will never go away, probably not a good policy if this is cached for web page in WININET.

SListBase<DynamicProfileInfo \*> \* profileInfoList = scriptContext->GetProfileInfoList();

FOREACH\_SLISTBASE\_ENTRY(DynamicProfileInfo \*, info, profileInfoList)

{

FunctionBody \* functionBody = info->GetFunctionBody();

SourceDynamicProfileManager \* sourceDynamicProfileManager = functionBody->GetSourceContextInfo()->sourceDynamicProfileManager;

sourceDynamicProfileManager->SaveDynamicProfileInfo(functionBody->GetLocalFunctionId(), info);

}

NEXT\_SLISTBASE\_ENTRY

}

#endif

#ifdef RUNTIME\_DATA\_COLLECTION

CriticalSection DynamicProfileInfo::s\_csOutput;

template <typename T>

void DynamicProfileInfo::WriteData(T data, FILE \* file)

{

fwrite(&data, sizeof(T), 1, file);

}

template <>

void DynamicProfileInfo::WriteData<wchar\_t const \*>(wchar\_t const \* sz, FILE \* file)

{

if (sz)

{

charcount\_t len = static\_cast<charcount\_t>(wcslen(sz));

utf8char\_t \* tempBuffer = HeapNewArray(utf8char\_t, len \* 3);

size\_t cbNeeded = utf8::EncodeInto(tempBuffer, sz, len);

fwrite(&cbNeeded, sizeof(cbNeeded), 1, file);

fwrite(tempBuffer, sizeof(utf8char\_t), cbNeeded, file);

HeapDeleteArray(len \* 3, tempBuffer);

}

else

{

charcount\_t len = 0;

fwrite(&len, sizeof(len), 1, file);

}

}

template <typename T>

void DynamicProfileInfo::WriteArray(uint count, T \* arr, FILE \* file)

{

WriteData(count, file);

for (uint i = 0; i < count; i++)

{

WriteData(arr[i], file);

}

}

template <>

void DynamicProfileInfo::WriteData<FunctionBody \*>(FunctionBody \* functionBody, FILE \* file)

{

WriteData(functionBody->GetSourceContextInfo()->sourceContextId, file);

WriteData(functionBody->GetLocalFunctionId(), file);

}

void DynamicProfileInfo::DumpScriptContextToFile(ScriptContext \* scriptContext)

{

if (Configuration::Global.flags.RuntimeDataOutputFile == nullptr)

{

return;

}

AutoCriticalSection autocs(&s\_csOutput);

FILE \* file;

if (\_wfopen\_s(&file, Configuration::Global.flags.RuntimeDataOutputFile, L"ab+") != 0 || file == nullptr)

{

return;

}

WriteData(scriptContext->GetAllocId(), file);

WriteData(scriptContext->GetCreateTime(), file);

WriteData(scriptContext->GetUrl(), file);

WriteData(scriptContext->GetSourceContextInfoMap() != nullptr ? scriptContext->GetSourceContextInfoMap()->Count() : 0, file);

if (scriptContext->GetSourceContextInfoMap())

{

scriptContext->GetSourceContextInfoMap()->Map([&](DWORD\_PTR dwHostSourceContext, SourceContextInfo \* sourceContextInfo)

{

WriteData(sourceContextInfo->sourceContextId, file);

WriteData(sourceContextInfo->nextLocalFunctionId, file);

WriteData(sourceContextInfo->url, file);

});

}

FOREACH\_SLISTBASE\_ENTRY(DynamicProfileInfo \*, info, scriptContext->GetProfileInfoList())

{

WriteData((byte)1, file);

WriteData(info->functionBody, file);

WriteData(info->functionBody->GetDisplayName(), file);

WriteData(info->functionBody->interpretedCount, file);

uint loopCount = info->functionBody->GetLoopCount();

WriteData(loopCount, file);

for (uint i = 0; i < loopCount; i++)

{

if (info->functionBody->DoJITLoopBody())

{

WriteData(info->functionBody->GetLoopHeader(i)->interpretCount, file);

}

else

{

WriteData(-1, file);

}

}

WriteArray(info->functionBody->GetProfiledLdElemCount(), info->ldElemInfo, file);

WriteArray(info->functionBody->GetProfiledStElemCount(), info->stElemInfo, file);

WriteArray(info->functionBody->GetProfiledArrayCallSiteCount(), info->arrayCallSiteInfo, file);

WriteArray(info->functionBody->GetProfiledCallSiteCount(), info->callSiteInfo, file);

}

NEXT\_SLISTBASE\_ENTRY;

WriteData((byte)0, file);

fflush(file);

fclose(file);

}

#endif

void DynamicProfileInfo::InstantiateForceInlinedMembers()

{

// Force-inlined functions defined in a translation unit need a reference from an extern non-force-inlined function in the

// same translation unit to force an instantiation of the force-inlined function. Otherwise, if the force-inlined function

// is not referenced in the same translation unit, it will not be generated and the linker is not able to find the

// definition to inline the function in other translation units.

Assert(false);

FunctionBody \*const functionBody = nullptr;

const Js::Var var = nullptr;

DynamicProfileInfo \*const p = nullptr;

p->RecordFieldAccess(functionBody, 0, var, FldInfo\_NoInfo);

p->RecordDivideResultType(functionBody, 0, var);

p->RecordModulusOpType(functionBody, 0, false);

p->RecordSwitchType(functionBody, 0, var);

p->RecordPolymorphicFieldAccess(functionBody, 0);

p->RecordSlotLoad(functionBody, 0, var);

p->RecordParameterInfo(functionBody, 0, var);

p->RecordReturnTypeOnCallSiteInfo(functionBody, 0, var);

p->RecordReturnType(functionBody, 0, var);

p->RecordThisInfo(var, ThisType\_Unknown);

}

};

bool IR::IsTypeCheckBailOutKind(IR::BailOutKind kind)

{

IR::BailOutKind kindWithoutBits = kind & ~IR::BailOutKindBits;

return

kindWithoutBits == IR::BailOutFailedTypeCheck ||

kindWithoutBits == IR::BailOutFailedFixedFieldTypeCheck ||

kindWithoutBits == IR::BailOutFailedEquivalentTypeCheck ||

kindWithoutBits == IR::BailOutFailedEquivalentFixedFieldTypeCheck;

}

bool IR::IsEquivalentTypeCheckBailOutKind(IR::BailOutKind kind)

{

IR::BailOutKind kindWithoutBits = kind & ~IR::BailOutKindBits;

return

kindWithoutBits == IR::BailOutFailedEquivalentTypeCheck ||

kindWithoutBits == IR::BailOutFailedEquivalentFixedFieldTypeCheck;

}

IR::BailOutKind IR::EquivalentToMonoTypeCheckBailOutKind(IR::BailOutKind kind)

{

switch (kind & ~IR::BailOutKindBits)

{

case IR::BailOutFailedEquivalentTypeCheck:

return IR::BailOutFailedTypeCheck | (kind & IR::BailOutKindBits);

case IR::BailOutFailedEquivalentFixedFieldTypeCheck:

return IR::BailOutFailedFixedFieldTypeCheck | (kind & IR::BailOutKindBits);

default:

Assert(0);

return IR::BailOutInvalid;

}

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

const char \*const BailOutKindNames[] =

{

#define BAIL\_OUT\_KIND\_LAST(n) "" STRINGIZE(n) ""

#define BAIL\_OUT\_KIND(n, ...) BAIL\_OUT\_KIND\_LAST(n),

#define BAIL\_OUT\_KIND\_VALUE\_LAST(n, v) BAIL\_OUT\_KIND\_LAST(n)

#define BAIL\_OUT\_KIND\_VALUE(n, v) BAIL\_OUT\_KIND(n)

#include "BailOutKind.h"

};

IR::BailOutKind const BailOutKindValidBits[] =

{

#define BAIL\_OUT\_KIND(n, bits) (IR::BailOutKind)bits,

#define BAIL\_OUT\_KIND\_VALUE\_LAST(n, v)

#define BAIL\_OUT\_KIND\_VALUE(n, v)

#define BAIL\_OUT\_KIND\_LAST(n)

#include "BailOutKind.h"

};

bool IsValidBailOutKindAndBits(IR::BailOutKind bailOutKind)

{

IR::BailOutKind kindNoBits = bailOutKind & ~IR::BailOutKindBits;

if (kindNoBits >= IR::BailOutKindBitsStart)

{

return false;

}

return ((bailOutKind & IR::BailOutKindBits) & ~BailOutKindValidBits[kindNoBits]) == 0;

}

// Concats into the buffer, specified by the name parameter, the name of 'bit' bailout kind, specified by the enumEntryOffsetFromBitsStart parameter.

// Returns the number of bytes printed to the buffer.

size\_t ConcatBailOutKindBits(\_Out\_writes\_bytes\_(dstSizeBytes) char\* dst, \_In\_ size\_t dstSizeBytes, \_In\_ size\_t position, \_In\_ uint enumEntryOffsetFromBitsStart)

{

const char\* kindName = BailOutKindNames[IR::BailOutKindBitsStart + static\_cast<IR::BailOutKind>(enumEntryOffsetFromBitsStart)];

int printedBytes =

sprintf\_s(

&dst[position],

dstSizeBytes - position \* sizeof(dst[0]),

position == 0 ? "%s" : " | %s",

kindName);

return printedBytes;

}

const char\* GetBailOutKindName(IR::BailOutKind kind)

{

using namespace IR;

if (!(kind & BailOutKindBits))

{

return BailOutKindNames[kind];

}

static char name[512];

size\_t position = 0;

const auto normalKind = kind & ~BailOutKindBits;

if (normalKind != 0)

{

kind -= normalKind;

position +=

sprintf\_s(

&name[position],

sizeof(name) / sizeof(name[0]) - position \* sizeof(name[0]),

position == 0 ? "%s" : " | %s",

BailOutKindNames[normalKind]);

}

uint offset = 1;

if (kind & BailOutOnOverflow)

{

kind ^= BailOutOnOverflow;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutOnMulOverflow)

{

kind ^= BailOutOnMulOverflow;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutOnNegativeZero)

{

kind ^= BailOutOnNegativeZero;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

// BailOutOnResultConditions

++offset;

if (kind & BailOutOnMissingValue)

{

kind ^= BailOutOnMissingValue;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutConventionalNativeArrayAccessOnly)

{

kind ^= BailOutConventionalNativeArrayAccessOnly;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutConvertedNativeArray)

{

kind ^= BailOutConvertedNativeArray;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutOnArrayAccessHelperCall)

{

kind ^= BailOutOnArrayAccessHelperCall;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutOnInvalidatedArrayHeadSegment)

{

kind ^= BailOutOnInvalidatedArrayHeadSegment;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutOnInvalidatedArrayLength)

{

kind ^= BailOutOnInvalidatedArrayLength;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

// BailOutForArrayBits

++offset;

if (kind & BailOutForceByFlag)

{

kind ^= BailOutForceByFlag;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutBreakPointInFunction)

{

kind ^= BailOutBreakPointInFunction;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutStackFrameBase)

{

kind ^= BailOutStackFrameBase;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutLocalValueChanged)

{

kind ^= BailOutLocalValueChanged;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutExplicit)

{

kind ^= BailOutExplicit;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutStep)

{

kind ^= BailOutStep;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutIgnoreException)

{

kind ^= BailOutIgnoreException;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

// BailOutForDebuggerBits

++offset;

if (kind & BailOutOnDivByZero)

{

kind ^= BailOutOnDivByZero;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

if (kind & BailOutOnDivOfMinInt)

{

kind ^= BailOutOnDivOfMinInt;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

// BailOutOnDivSrcConditions

++offset;

if (kind & BailOutMarkTempObject)

{

kind ^= BailOutMarkTempObject;

position += ConcatBailOutKindBits(name, sizeof(name), position, offset);

}

++offset;

// BailOutKindBits

Assert(position != 0);

Assert(!kind);

return name;

}

#endif

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

// DisableJit-TODO

#if ENABLE\_PROFILE\_INFO

#ifdef DYNAMIC\_PROFILE\_MUTATOR

class DynamicProfileMutatorImpl;

#endif

#define PolymorphicInlineCacheUtilizationMinValue 0

#define PolymorphicInlineCacheUtilizationMaxValue 0xFF

#define PolymorphicInlineCacheUtilizationThreshold 0x80

#define PolymorphicInlineCacheUtilizationIncrement 10

#define PolymorphicInlineCacheUtilizationDecrement 1

namespace IR

{

enum BailOutKind : uint

{

#define BAIL\_OUT\_KIND\_LAST(n) n

#define BAIL\_OUT\_KIND(n, ...) BAIL\_OUT\_KIND\_LAST(n),

#define BAIL\_OUT\_KIND\_VALUE\_LAST(n, v) n = v

#define BAIL\_OUT\_KIND\_VALUE(n, v) BAIL\_OUT\_KIND\_VALUE\_LAST(n, v),

#include "BailOutKind.h"

};

ENUM\_CLASS\_HELPERS(BailOutKind, uint);

CompileAssert(BailOutKind::BailOutKindEnd < BailOutKind::BailOutKindBitsStart);

bool IsTypeCheckBailOutKind(BailOutKind kind);

bool IsEquivalentTypeCheckBailOutKind(BailOutKind kind);

BailOutKind EquivalentToMonoTypeCheckBailOutKind(BailOutKind kind);

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

const char \*GetBailOutKindName(IR::BailOutKind kind);

bool IsValidBailOutKindAndBits(IR::BailOutKind bailOutKind);

#endif

namespace Js

{

enum CacheType : byte;

enum SlotType : byte;

struct PolymorphicCallSiteInfo;

// Information about dynamic profile information loaded from cache.

// Used to verify whether the loaded information matches the paired function body.

class DynamicProfileFunctionInfo

{

public:

Js::ArgSlot paramInfoCount;

ProfileId ldElemInfoCount;

ProfileId stElemInfoCount;

ProfileId arrayCallSiteCount;

ProfileId slotInfoCount;

ProfileId callSiteInfoCount;

ProfileId returnTypeInfoCount;

ProfileId divCount;

ProfileId switchCount;

uint loopCount;

uint fldInfoCount;

};

enum ThisType : BYTE

{

ThisType\_Unknown = 0,

ThisType\_Simple,

ThisType\_Mapped

};

struct ThisInfo

{

ValueType valueType;

ThisType thisType;

ThisInfo() : thisType(ThisType\_Unknown)

{

}

};

// TODO: include ImplicitCallFlags in this structure

struct LoopFlags

{

// maintain the bits and the enum at the same time, it must match

bool isInterpreted : 1;

bool memopMinCountReached : 1;

enum

{

INTERPRETED,

MEMOP\_MIN\_COUNT\_FOUND,

COUNT

};

LoopFlags() :

isInterpreted(false),

memopMinCountReached(false)

{

CompileAssert((sizeof(LoopFlags) \* 8) >= LoopFlags::COUNT);

}

// Right now supports up to 8 bits.

typedef byte LoopFlags\_t;

LoopFlags(uint64 flags)

{

Assert(flags >> LoopFlags::COUNT == 0);

LoopFlags\_t\* thisFlags = (LoopFlags\_t \*)this;

CompileAssert(sizeof(LoopFlags\_t) == sizeof(LoopFlags));

\*thisFlags = (LoopFlags\_t)flags;

}

};

enum FldInfoFlags : BYTE

{

FldInfo\_NoInfo = 0x00,

FldInfo\_FromLocal = 0x01,

FldInfo\_FromProto = 0x02,

FldInfo\_FromLocalWithoutProperty = 0x04,

FldInfo\_FromAccessor = 0x08,

FldInfo\_Polymorphic = 0x10,

FldInfo\_FromInlineSlots = 0x20,

FldInfo\_FromAuxSlots = 0x40,

FldInfo\_InlineCandidate = 0x80

};

struct FldInfo

{

typedef struct { ValueType::TSize f1; byte f2; byte f3; } TSize;

ValueType valueType;

FldInfoFlags flags;

byte polymorphicInlineCacheUtilization;

bool ShouldUsePolymorphicInlineCache()

{

#if DBG

if (PHASE\_FORCE1(PolymorphicInlineCachePhase))

{

return true;

}

#endif

return polymorphicInlineCacheUtilization > PolymorphicInlineCacheUtilizationThreshold;

}

bool WasLdFldProfiled() const

{

return !valueType.IsUninitialized();

}

uint32 GetOffsetOfFlags() { return offsetof(FldInfo, flags); }

};

CompileAssert(sizeof(FldInfo::TSize) == sizeof(FldInfo));

struct LdElemInfo

{

ValueType arrayType;

ValueType elemType;

union

{

struct

{

bool wasProfiled : 1;

bool neededHelperCall : 1;

};

byte bits;

};

LdElemInfo() : bits(0)

{

wasProfiled = true;

}

void Merge(const LdElemInfo &other)

{

arrayType = arrayType.Merge(other.arrayType);

elemType = elemType.Merge(other.elemType);

bits |= other.bits;

}

ValueType GetArrayType() const

{

return arrayType;

}

ValueType GetElementType() const

{

return elemType;

}

bool WasProfiled() const

{

return wasProfiled;

}

bool LikelyNeedsHelperCall() const

{

return neededHelperCall;

}

};

struct StElemInfo

{

ValueType arrayType;

union

{

struct

{

bool wasProfiled : 1;

bool createdMissingValue : 1;

bool filledMissingValue : 1;

bool neededHelperCall : 1;

bool storedOutsideHeadSegmentBounds : 1;

bool storedOutsideArrayBounds : 1;

};

byte bits;

};

StElemInfo() : bits(0)

{

wasProfiled = true;

}

void Merge(const StElemInfo &other)

{

arrayType = arrayType.Merge(other.arrayType);

bits |= other.bits;

}

ValueType GetArrayType() const

{

return arrayType;

}

bool WasProfiled() const

{

return wasProfiled;

}

bool LikelyCreatesMissingValue() const

{

return createdMissingValue;

}

bool LikelyFillsMissingValue() const

{

return filledMissingValue;

}

bool LikelyNeedsHelperCall() const

{

return createdMissingValue || filledMissingValue || neededHelperCall || storedOutsideHeadSegmentBounds;

}

bool LikelyStoresOutsideHeadSegmentBounds() const

{

return createdMissingValue || storedOutsideHeadSegmentBounds;

}

bool LikelyStoresOutsideArrayBounds() const

{

return storedOutsideArrayBounds;

}

};

struct ArrayCallSiteInfo

{

union {

struct {

byte isNotNativeInt : 1;

byte isNotNativeFloat : 1;

#if ENABLE\_COPYONACCESS\_ARRAY

byte isNotCopyOnAccessArray : 1;

byte copyOnAccessArrayCacheIndex : 5;

#endif

};

byte bits;

};

#if DBG

uint functionNumber;

ProfileId callSiteNumber;

#endif

bool IsNativeIntArray() const { return !(bits & NotNativeIntBit) && !PHASE\_OFF1(NativeArrayPhase); }

bool IsNativeFloatArray() const { return !(bits & NotNativeFloatBit) && !PHASE\_OFF1(NativeArrayPhase); }

bool IsNativeArray() const { return IsNativeFloatArray(); }

void SetIsNotNativeIntArray();

void SetIsNotNativeFloatArray();

void SetIsNotNativeArray();

static uint32 GetOffsetOfBits() { return offsetof(ArrayCallSiteInfo, bits); }

static byte const NotNativeIntBit = 1;

static byte const NotNativeFloatBit = 2;

};

class DynamicProfileInfo

{

public:

static DynamicProfileInfo\* New(Recycler\* recycler, FunctionBody\* functionBody, bool persistsAcrossScriptContexts = false);

void Initialize(FunctionBody \*const functionBody);

public:

static bool IsEnabledForAtLeastOneFunction(const ScriptContext \*const scriptContext);

static bool IsEnabled(const FunctionBody \*const functionBody);

private:

static bool IsEnabled\_OptionalFunctionBody(const FunctionBody \*const functionBody, const ScriptContext \*const scriptContext);

public:

static bool IsEnabledForAtLeastOneFunction(const Js::Phase phase, const ScriptContext \*const scriptContext);

static bool IsEnabled(const Js::Phase phase, const FunctionBody \*const functionBody);

private:

static bool IsEnabled\_OptionalFunctionBody(const Js::Phase phase, const FunctionBody \*const functionBody, const ScriptContext \*const scriptContext);

public:

static bool EnableImplicitCallFlags(const FunctionBody \*const functionBody);

static Var EnsureDynamicProfileInfoThunk(RecyclableObject \* function, CallInfo callInfo, ...);

#ifdef DYNAMIC\_PROFILE\_STORAGE

bool HasFunctionBody() const { return hasFunctionBody; }

FunctionBody \* GetFunctionBody() const { Assert(hasFunctionBody); return functionBody; }

#endif

void RecordElementLoad(FunctionBody\* functionBody, ProfileId ldElemId, const LdElemInfo& info);

void RecordElementLoadAsProfiled(FunctionBody \*const functionBody, const ProfileId ldElemId);

const LdElemInfo \*GetLdElemInfo() const { return ldElemInfo; }

void RecordElementStore(FunctionBody\* functionBody, ProfileId stElemId, const StElemInfo& info);

void RecordElementStoreAsProfiled(FunctionBody \*const functionBody, const ProfileId stElemId);

const StElemInfo \*GetStElemInfo() const { return stElemInfo; }

ArrayCallSiteInfo \*GetArrayCallSiteInfo(FunctionBody \*functionBody, ProfileId index) const;

void RecordFieldAccess(FunctionBody\* functionBody, uint fieldAccessId, Var object, FldInfoFlags flags);

void RecordPolymorphicFieldAccess(FunctionBody \*functionBody, uint fieldAccessid);

bool HasPolymorphicFldAccess() const { return bits.hasPolymorphicFldAccess; }

FldInfo \* GetFldInfo(FunctionBody\* functionBody, uint fieldAccessId) const;

void RecordSlotLoad(FunctionBody\* functionBody, ProfileId slotLoadId, Var object);

ValueType GetSlotLoad(FunctionBody\* functionBody, ProfileId slotLoadId) const;

void RecordThisInfo(Var object, ThisType thisType);

ThisInfo GetThisInfo() const;

void RecordDivideResultType(FunctionBody\* body, ProfileId divideId, Var object);

ValueType GetDivideResultType(FunctionBody\* body, ProfileId divideId) const;

void RecordModulusOpType(FunctionBody\* body, ProfileId profileId, bool isModByPowerOf2);

bool IsModulusOpByPowerOf2(FunctionBody\* body, ProfileId profileId) const;

void RecordSwitchType(FunctionBody\* body, ProfileId switchId, Var object);

ValueType GetSwitchType(FunctionBody\* body, ProfileId switchId) const;

void RecordCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, FunctionInfo \* calleeFunctionInfo, JavascriptFunction\* calleeFunction, ArgSlot actualArgCount, bool isConstructorCall, InlineCacheIndex ldFldInlnlineCacheId = Js::Constants::NoInlineCacheIndex);

void RecordConstParameterAtCallSite(ProfileId callSiteId, int argNum);

bool HasCallSiteInfo(FunctionBody\* functionBody);

bool HasCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId); // Does a particular callsite have ProfileInfo?

FunctionInfo \* GetCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, bool \*isConstructorCall, bool \*isPolymorphicCall);

uint16 GetConstantArgInfo(ProfileId callSiteId);

uint GetLdFldCacheIndexFromCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId);

bool GetPolymorphicCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, bool \*isConstructorCall, \_\_inout\_ecount(functionBodyArrayLength) FunctionBody\*\* functionBodyArray, uint functionBodyArrayLength);

bool RecordLdFldCallSiteInfo(FunctionBody\* functionBody, RecyclableObject\* callee, bool callApplyTarget);

bool hasLdFldCallSiteInfo();

void RecordReturnTypeOnCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, Var object);

void RecordReturnType(FunctionBody\* functionBody, ProfileId callSiteId, Var object);

ValueType GetReturnType(FunctionBody\* functionBody, Js::OpCode opcode, ProfileId callSiteId) const;

void RecordParameterInfo(FunctionBody\* functionBody, ArgSlot index, Var object);

ValueType GetParameterInfo(FunctionBody\* functionBody, ArgSlot index) const;

void RecordLoopImplicitCallFlags(FunctionBody\* functionBody, uint loopNum, ImplicitCallFlags flags);

ImplicitCallFlags GetLoopImplicitCallFlags(FunctionBody\* functionBody, uint loopNum) const;

void RecordImplicitCallFlags(ImplicitCallFlags flags);

ImplicitCallFlags GetImplicitCallFlags() const;

static void Save(ScriptContext \* scriptContext);

void UpdateFunctionInfo(FunctionBody\* functionBody, Recycler\* allocator);

void ResetAllPolymorphicCallSiteInfo();

bool CallSiteHasProfileData(ProfileId callSiteId)

{

return this->callSiteInfo[callSiteId].isPolymorphic

|| this->callSiteInfo[callSiteId].u.functionData.sourceId != NoSourceId

|| this->callSiteInfo[callSiteId].dontInline;

}

static bool IsProfiledCallOp(OpCode op);

static bool IsProfiledReturnTypeOp(OpCode op);

static FldInfoFlags FldInfoFlagsFromCacheType(CacheType cacheType);

static FldInfoFlags FldInfoFlagsFromSlotType(SlotType slotType);

static FldInfoFlags MergeFldInfoFlags(FldInfoFlags oldFlags, FldInfoFlags newFlags);

const static uint maxPolymorphicInliningSize = 4;

#if DBG\_DUMP

static void DumpScriptContext(ScriptContext \* scriptContext);

static wchar\_t const \* GetImplicitCallFlagsString(ImplicitCallFlags flags);

#endif

#ifdef RUNTIME\_DATA\_COLLECTION

static void DumpScriptContextToFile(ScriptContext \* scriptContext);

#endif

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

static bool NeedProfileInfoList();

#endif

#ifdef DYNAMIC\_PROFILE\_MUTATOR

friend class DynamicProfileMutatorImpl;

#endif

#if JS\_PROFILE\_DATA\_INTERFACE

friend class ProfileDataObject;

#endif

private:

// Have the dynamicProfileFunctionInfo after loaded from cache.

// Replaced with the function body it is verified and matched (See DynamicProfileInfo::MatchFunctionBody)

DynamicProfileFunctionInfo \* dynamicProfileFunctionInfo;

struct CallSiteInfo

{

uint16 isArgConstant : 13;

uint16 isConstructorCall : 1;

uint16 dontInline : 1;

uint16 isPolymorphic : 1;

ValueType returnType;

InlineCacheIndex ldFldInlineCacheId;

union

{

struct

{

Js::SourceId sourceId;

Js::LocalFunctionId functionId;

} functionData;

// As of now polymorphic info is allocated only if the source Id is current

PolymorphicCallSiteInfo\* polymorphicCallSiteInfo;

} u;

} \*callSiteInfo;

ValueType \* returnTypeInfo; // return type of calls for non inline call sites

ValueType \* divideTypeInfo;

ValueType \* switchTypeInfo;

LdElemInfo \* ldElemInfo;

StElemInfo \* stElemInfo;

ArrayCallSiteInfo \*arrayCallSiteInfo;

ValueType \* parameterInfo;

FldInfo \* fldInfo;

ValueType \* slotInfo;

ImplicitCallFlags \* loopImplicitCallFlags;

ImplicitCallFlags implicitCallFlags;

BVFixed\* loopFlags;

ThisInfo thisInfo;

// TODO (jedmiad): Consider storing a pair of property ID bit vectors indicating which properties are

// known to be non-fixed or non-equivalent. We could turn these on if we bailed out of fixed field type

// checks and equivalent type checks in a way that indicates one of these failures as opposed to type

// mismatch.

struct Bits

{

bool disableAggressiveIntTypeSpec : 1;

bool disableAggressiveIntTypeSpec\_jitLoopBody : 1;

bool disableAggressiveMulIntTypeSpec : 1;

bool disableAggressiveMulIntTypeSpec\_jitLoopBody : 1;

bool disableDivIntTypeSpec : 1;

bool disableDivIntTypeSpec\_jitLoopBody : 1;

bool disableLossyIntTypeSpec : 1;

// TODO: put this flag in LoopFlags if we can find a reliable way to determine the loopNumber in bailout for a hoisted instr

bool disableMemOp : 1;

bool disableTrackCompoundedIntOverflow : 1;

bool disableFloatTypeSpec : 1;

bool disableCheckThis : 1;

bool disableArrayCheckHoist : 1;

bool disableArrayCheckHoist\_jitLoopBody : 1;

bool disableArrayMissingValueCheckHoist : 1;

bool disableArrayMissingValueCheckHoist\_jitLoopBody : 1;

bool disableJsArraySegmentHoist : 1;

bool disableJsArraySegmentHoist\_jitLoopBody : 1;

bool disableArrayLengthHoist : 1;

bool disableArrayLengthHoist\_jitLoopBody : 1;

bool disableTypedArrayTypeSpec : 1;

bool disableTypedArrayTypeSpec\_jitLoopBody : 1;

bool disableLdLenIntSpec : 1;

bool disableBoundCheckHoist : 1;

bool disableBoundCheckHoist\_jitLoopBody : 1;

bool disableLoopCountBasedBoundCheckHoist : 1;

bool disableLoopCountBasedBoundCheckHoist\_jitLoopBody : 1;

bool hasPolymorphicFldAccess : 1;

bool hasLdFldCallSite : 1; // getters, setters, .apply (possibly .call too in future)

bool disableFloorInlining : 1;

bool disableNoProfileBailouts : 1;

bool disableSwitchOpt : 1;

bool disableEquivalentObjTypeSpec : 1;

bool disableObjTypeSpec\_jitLoopBody : 1;

} bits;

uint32 m\_recursiveInlineInfo; // Bit is set for each callsites where the function is called recursively

BYTE currentInlinerVersion; // Used to detect when inlining profile changes

uint32 polymorphicCacheState;

bool hasFunctionBody;

#if DBG

bool persistsAcrossScriptContexts;

#endif

static JavascriptMethod EnsureDynamicProfileInfo(Js::ScriptFunction \* function);

#if DBG\_DUMP

static void DumpList(SListBase<DynamicProfileInfo \*> \* profileInfoList, ArenaAllocator \* dynamicProfileInfoAllocator);

static void DumpProfiledValue(wchar\_t const \* name, uint \* value, uint count);

static void DumpProfiledValue(wchar\_t const \* name, ValueType \* value, uint count);

static void DumpProfiledValue(wchar\_t const \* name, CallSiteInfo \* callSiteInfo, uint count);

static void DumpProfiledValue(wchar\_t const \* name, ArrayCallSiteInfo \* arrayCallSiteInfo, uint count);

static void DumpProfiledValue(wchar\_t const \* name, ImplicitCallFlags \* loopImplicitCallFlags, uint count);

template<class TData, class FGetValueType>

static void DumpProfiledValuesGroupedByValue(const wchar\_t \*const name, const TData \*const data, const uint count, const FGetValueType GetValueType, ArenaAllocator \*const dynamicProfileInfoAllocator);

static void DumpFldInfoFlags(wchar\_t const \* name, FldInfo \* fldInfo, uint count, FldInfoFlags value, wchar\_t const \* valueName);

static void DumpLoopInfo(FunctionBody \*fbody);

#endif

bool IsPolymorphicCallSite(Js::LocalFunctionId curFunctionId, Js::SourceId curSourceId, Js::LocalFunctionId oldFunctionId, Js::SourceId oldSourceId);

void CreatePolymorphicDynamicProfileCallSiteInfo(FunctionBody \* funcBody, ProfileId callSiteId, Js::LocalFunctionId functionId, Js::LocalFunctionId oldFunctionId, Js::SourceId sourceId, Js::SourceId oldSourceId);

void ResetPolymorphicCallSiteInfo(ProfileId callSiteId, Js::LocalFunctionId functionId);

void SetFunctionIdSlotForNewPolymorphicCall(ProfileId callSiteId, Js::LocalFunctionId curFunctionId, Js::SourceId curSourceId, Js::FunctionBody \*inliner);

void RecordPolymorphicCallSiteInfo(FunctionBody\* functionBody, ProfileId callSiteId, FunctionInfo \* calleeFunctionInfo);

#ifdef RUNTIME\_DATA\_COLLECTION

static CriticalSection s\_csOutput;

template <typename T>

static void WriteData(T data, FILE \* file);

template <>

static void WriteData<wchar\_t const \*>(wchar\_t const \* sz, FILE \* file);

template <>

static void WriteData<FunctionInfo \*>(FunctionInfo \* functionInfo, FILE \* file); // Not defined, to prevent accidentally writing function info

template <>

static void WriteData<FunctionBody \*>(FunctionBody \* functionInfo, FILE \* file);

template <typename T>

static void WriteArray(uint count, T \* arr, FILE \* file);

#endif

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

FunctionBody \* functionBody; // This will only be populated if NeedProfileInfoList is true

#endif

#ifdef DYNAMIC\_PROFILE\_STORAGE

// Used by de-serialize

DynamicProfileInfo();

template <typename T>

static DynamicProfileInfo \* Deserialize(T \* reader, Recycler\* allocator, Js::LocalFunctionId \* functionId);

template <typename T>

bool Serialize(T \* writer);

static void UpdateSourceDynamicProfileManagers(ScriptContext \* scriptContext);

#endif

static Js::LocalFunctionId const CallSiteMixed = (Js::LocalFunctionId)-1;

static Js::LocalFunctionId const CallSiteCrossContext = (Js::LocalFunctionId)-2;

static Js::LocalFunctionId const CallSiteNonFunction = (Js::LocalFunctionId)-3;

static Js::LocalFunctionId const CallSiteNoInfo = (Js::LocalFunctionId)-4;

static Js::LocalFunctionId const StartInvalidFunction = (Js::LocalFunctionId)-4;

static Js::SourceId const NoSourceId = (SourceId)-1;

static Js::SourceId const BuiltInSourceId = (SourceId)-2;

static Js::SourceId const CurrentSourceId = (SourceId)-3; // caller and callee in the same file

static Js::SourceId const InvalidSourceId = (SourceId)-4;

bool MatchFunctionBody(FunctionBody \* functionBody);

DynamicProfileInfo(FunctionBody \* functionBody);

friend class SourceDynamicProfileManager;

public:

bool IsAggressiveIntTypeSpecDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableAggressiveIntTypeSpec\_jitLoopBody

: this->bits.disableAggressiveIntTypeSpec;

}

void DisableAggressiveIntTypeSpec(const bool isJitLoopBody)

{

this->bits.disableAggressiveIntTypeSpec\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableAggressiveIntTypeSpec = true;

}

}

bool IsAggressiveMulIntTypeSpecDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableAggressiveMulIntTypeSpec\_jitLoopBody

: this->bits.disableAggressiveMulIntTypeSpec;

}

void DisableAggressiveMulIntTypeSpec(const bool isJitLoopBody)

{

this->bits.disableAggressiveMulIntTypeSpec\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableAggressiveMulIntTypeSpec = true;

}

}

bool IsDivIntTypeSpecDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableDivIntTypeSpec\_jitLoopBody

: this->bits.disableDivIntTypeSpec;

}

void DisableDivIntTypeSpec(const bool isJitLoopBody)

{

this->bits.disableDivIntTypeSpec\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableDivIntTypeSpec = true;

}

}

bool IsLossyIntTypeSpecDisabled() const { return bits.disableLossyIntTypeSpec; }

void DisableLossyIntTypeSpec() { this->bits.disableLossyIntTypeSpec = true; }

LoopFlags GetLoopFlags(int loopNumber) const

{

Assert(loopFlags);

return loopFlags->GetRange<LoopFlags>(loopNumber \* LoopFlags::COUNT, LoopFlags::COUNT);

}

void SetLoopInterpreted(int loopNumber) { loopFlags->Set(loopNumber \* LoopFlags::COUNT + LoopFlags::INTERPRETED); }

void SetMemOpMinReached(int loopNumber) { loopFlags->Set(loopNumber \* LoopFlags::COUNT + LoopFlags::MEMOP\_MIN\_COUNT\_FOUND); }

bool IsMemOpDisabled() const { return this->bits.disableMemOp; }

void DisableMemOp() { this->bits.disableMemOp = true; }

bool IsTrackCompoundedIntOverflowDisabled() const { return this->bits.disableTrackCompoundedIntOverflow; }

void DisableTrackCompoundedIntOverflow() { this->bits.disableTrackCompoundedIntOverflow = true; }

bool IsFloatTypeSpecDisabled() const { return this->bits.disableFloatTypeSpec; }

void DisableFloatTypeSpec() { this->bits.disableFloatTypeSpec = true; }

bool IsCheckThisDisabled() const { return this->bits.disableCheckThis; }

void DisableCheckThis() { this->bits.disableCheckThis = true; }

bool IsArrayCheckHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableArrayCheckHoist\_jitLoopBody

: this->bits.disableArrayCheckHoist;

}

void DisableArrayCheckHoist(const bool isJitLoopBody)

{

this->bits.disableArrayCheckHoist\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableArrayCheckHoist = true;

}

}

bool IsArrayMissingValueCheckHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableArrayMissingValueCheckHoist\_jitLoopBody

: this->bits.disableArrayMissingValueCheckHoist;

}

void DisableArrayMissingValueCheckHoist(const bool isJitLoopBody)

{

this->bits.disableArrayMissingValueCheckHoist\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableArrayMissingValueCheckHoist = true;

}

}

bool IsJsArraySegmentHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableJsArraySegmentHoist\_jitLoopBody

: this->bits.disableJsArraySegmentHoist;

}

void DisableJsArraySegmentHoist(const bool isJitLoopBody)

{

this->bits.disableJsArraySegmentHoist\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableJsArraySegmentHoist = true;

}

}

bool IsArrayLengthHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableArrayLengthHoist\_jitLoopBody

: this->bits.disableArrayLengthHoist;

}

void DisableArrayLengthHoist(const bool isJitLoopBody)

{

this->bits.disableArrayLengthHoist\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableArrayLengthHoist = true;

}

}

bool IsTypedArrayTypeSpecDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableTypedArrayTypeSpec\_jitLoopBody

: this->bits.disableTypedArrayTypeSpec;

}

void DisableTypedArrayTypeSpec(const bool isJitLoopBody)

{

this->bits.disableTypedArrayTypeSpec\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableTypedArrayTypeSpec = true;

}

}

bool IsLdLenIntSpecDisabled() const { return this->bits.disableLdLenIntSpec; }

void DisableLdLenIntSpec() { this->bits.disableLdLenIntSpec = true; }

bool IsBoundCheckHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableBoundCheckHoist\_jitLoopBody

: this->bits.disableBoundCheckHoist;

}

void DisableBoundCheckHoist(const bool isJitLoopBody)

{

this->bits.disableBoundCheckHoist\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableBoundCheckHoist = true;

}

}

bool IsLoopCountBasedBoundCheckHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->bits.disableLoopCountBasedBoundCheckHoist\_jitLoopBody

: this->bits.disableLoopCountBasedBoundCheckHoist;

}

void DisableLoopCountBasedBoundCheckHoist(const bool isJitLoopBody)

{

this->bits.disableLoopCountBasedBoundCheckHoist\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->bits.disableLoopCountBasedBoundCheckHoist = true;

}

}

BYTE GetInlinerVersion() { return this->currentInlinerVersion; }

uint32 GetPolymorphicCacheState() const { return this->polymorphicCacheState; }

uint32 GetRecursiveInlineInfo() const { return this->m\_recursiveInlineInfo; }

void SetHasNewPolyFieldAccess(FunctionBody \*functionBody);

bool IsFloorInliningDisabled() const { return this->bits.disableFloorInlining; }

void DisableFloorInlining() { this->bits.disableFloorInlining = true; }

bool IsNoProfileBailoutsDisabled() const { return this->bits.disableNoProfileBailouts; }

void DisableNoProfileBailouts() { this->bits.disableNoProfileBailouts = true; }

bool IsSwitchOptDisabled() const { return this->bits.disableSwitchOpt; }

void DisableSwitchOpt() { this->bits.disableSwitchOpt = true; }

bool IsEquivalentObjTypeSpecDisabled() const { return this->bits.disableEquivalentObjTypeSpec; }

void DisableEquivalentObjTypeSpec() { this->bits.disableEquivalentObjTypeSpec = true; }

bool IsObjTypeSpecDisabledInJitLoopBody() const { return this->bits.disableObjTypeSpec\_jitLoopBody; }

void DisableObjTypeSpecInJitLoopBody() { this->bits.disableObjTypeSpec\_jitLoopBody = true; }

static bool IsCallSiteNoInfo(Js::LocalFunctionId functionId) { return functionId == CallSiteNoInfo; }

#if DBG\_DUMP

void Dump(FunctionBody\* functionBody, ArenaAllocator \* dynamicProfileInfoAllocator = nullptr);

#endif

private:

static void InstantiateForceInlinedMembers();

};

struct PolymorphicCallSiteInfo

{

Js::LocalFunctionId functionIds[DynamicProfileInfo::maxPolymorphicInliningSize];

Js::SourceId sourceIds[DynamicProfileInfo::maxPolymorphicInliningSize];

PolymorphicCallSiteInfo \*next;

bool GetFunction(uint index, Js::LocalFunctionId \*functionId, Js::SourceId \*sourceId)

{

Assert(index < DynamicProfileInfo::maxPolymorphicInliningSize);

Assert(functionId);

Assert(sourceId);

if (DynamicProfileInfo::IsCallSiteNoInfo(functionIds[index]))

{

return false;

}

\*functionId = functionIds[index];

\*sourceId = sourceIds[index];

return true;

}

};

#ifdef DYNAMIC\_PROFILE\_STORAGE

class BufferReader

{

public:

BufferReader(\_\_in\_ecount(length) char const \* buffer, size\_t length) : current(buffer), lengthLeft(length) {}

template <typename T>

bool Read(T \* data)

{

if (lengthLeft < sizeof(T))

{

return false;

}

\*data = \*(T \*)current;

current += sizeof(T);

lengthLeft -= sizeof(T);

return true;

}

template <typename T>

bool Peek(T \* data)

{

if (lengthLeft < sizeof(T))

{

return false;

}

\*data = \*(T \*)current;

return true;

}

template <typename T>

bool ReadArray(\_\_inout\_ecount(len) T \* data, size\_t len)

{

size\_t size = sizeof(T) \* len;

if (lengthLeft < size)

{

return false;

}

memcpy\_s(data, size, current, size);

current += size;

lengthLeft -= size;

return true;

}

private:

char const \* current;

size\_t lengthLeft;

};

class BufferSizeCounter

{

public:

BufferSizeCounter() : count(0) {}

size\_t GetByteCount() const { return count; }

template <typename T>

bool Write(T const& data)

{

return WriteArray(&data, 1);

}

#if DBG\_DUMP

void Log(DynamicProfileInfo\* info) {}

#endif

template <typename T>

bool WriteArray(\_\_in\_ecount(len) T \* data, size\_t len)

{

count += sizeof(T) \* len;

return true;

}

private:

size\_t count;

};

class BufferWriter

{

public:

BufferWriter(\_\_in\_ecount(length) char \* buffer, size\_t length) : current(buffer), lengthLeft(length) {}

template <typename T>

bool Write(T const& data)

{

return WriteArray(&data, 1);

}

#if DBG\_DUMP

void Log(DynamicProfileInfo\* info);

#endif

template <typename T>

bool WriteArray(\_\_in\_ecount(len) T \* data, size\_t len)

{

size\_t size = sizeof(T) \* len;

if (lengthLeft < size)

{

return false;

}

memcpy\_s(current, size, data, size);

current += size;

lengthLeft -= size;

return true;

}

private:

char \* current;

size\_t lengthLeft;

};

#endif

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if ENABLE\_PROFILE\_INFO

#ifdef DYNAMIC\_PROFILE\_MUTATOR

#include "DynamicProfileMutator.h"

char const \* const DynamicProfileMutator::CreateMutatorProcName = STRINGIZE(CREATE\_MUTATOR\_PROC\_NAME);

void

DynamicProfileMutator::Mutate(Js::FunctionBody \* functionBody)

{

Js::ScriptContext \* scriptContext = functionBody->GetScriptContext();

DynamicProfileMutator \* dynamicProfileMutator = scriptContext->GetThreadContext()->dynamicProfileMutator;

if (dynamicProfileMutator != nullptr)

{

if (functionBody->dynamicProfileInfo == nullptr)

{

functionBody->dynamicProfileInfo = Js::DynamicProfileInfo::New(scriptContext->GetRecycler(), functionBody);

}

dynamicProfileMutator->Mutate(functionBody->dynamicProfileInfo);

// Save the profile information, it will help in case of Crash/Failure

Js::DynamicProfileInfo::Save(scriptContext);

}

}

DynamicProfileMutator \*

DynamicProfileMutator::GetMutator()

{

if (!Js::Configuration::Global.flags.IsEnabled(Js::DynamicProfileMutatorFlag))

{

return nullptr;

}

wchar\_t const \* dllname = Js::Configuration::Global.flags.DynamicProfileMutatorDll;

HMODULE hModule = ::LoadLibraryW(dllname);

if (hModule == nullptr)

{

Output::Print(L"ERROR: Unable to load dynamic profile mutator dll %s\n", dllname);

Js::Throw::FatalInternalError();

}

CreateMutatorFunc procAddress = (CreateMutatorFunc)::GetProcAddress(hModule, CreateMutatorProcName);

if (procAddress == nullptr)

{

Output::Print(L"ERROR: Unable to get function %S from dll %s\n", CreateMutatorProcName, dllname);

Js::Throw::FatalInternalError();

}

DynamicProfileMutator \* mutator = procAddress();

if (mutator == nullptr)

{

Output::Print(L"ERROR: Failed to create mutator from dll %s\n", dllname);

Js::Throw::FatalInternalError();

}

mutator->Initialize(Js::Configuration::Global.flags.DynamicProfileMutator);

return mutator;

}

#endif

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#if ENABLE\_PROFILE\_INFO

#ifdef DYNAMIC\_PROFILE\_MUTATOR

#define CREATE\_MUTATOR\_PROC\_NAME CreateDynamicProfileMutator

class DynamicProfileMutator

{

public:

virtual void Mutate(Js::DynamicProfileInfo \* info) = 0;

virtual void Delete() = 0;

virtual void Initialize(const wchar\_t \* options) = 0;

static void Mutate(Js::FunctionBody \* functionBody);

static DynamicProfileMutator \* GetMutator();

static char const \* const CreateMutatorProcName;

typedef DynamicProfileMutator \* (\*CreateMutatorFunc)();

};

#endif

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifdef DYNAMIC\_PROFILE\_STORAGE

bool DynamicProfileStorage::initialized = false;

bool DynamicProfileStorage::uninitialized = false;

bool DynamicProfileStorage::enabled = false;

bool DynamicProfileStorage::useCacheDir = false;

bool DynamicProfileStorage::collectInfo = false;

HANDLE DynamicProfileStorage::mutex = nullptr;

wchar\_t DynamicProfileStorage::cacheDrive[\_MAX\_DRIVE];

wchar\_t DynamicProfileStorage::cacheDir[\_MAX\_DIR];

wchar\_t DynamicProfileStorage::catalogFilename[\_MAX\_PATH];

CriticalSection DynamicProfileStorage::cs;

DynamicProfileStorage::InfoMap DynamicProfileStorage::infoMap(&NoCheckHeapAllocator::Instance);

DWORD DynamicProfileStorage::creationTime = 0;

long DynamicProfileStorage::lastOffset = 0;

DWORD const DynamicProfileStorage::MagicNumber = 20100526;

DWORD const DynamicProfileStorage::FileFormatVersion = 2;

DWORD DynamicProfileStorage::nextFileId = 0;

#if DBG

bool DynamicProfileStorage::locked = false;

#endif

class DynamicProfileStorageReaderWriter

{

public:

DynamicProfileStorageReaderWriter() : filename(nullptr), file(nullptr) {}

~DynamicProfileStorageReaderWriter();

bool Init(wchar\_t const \* filename, wchar\_t const \* mode, bool deleteNonClosed, errno\_t \* err);

template <typename T>

bool Read(T \* t);

template <typename T>

bool ReadArray(T \* t, size\_t len);

\_Success\_(return) bool ReadUtf8String(\_\_deref\_out\_z wchar\_t \*\* str, \_\_out DWORD \* len);

template <typename T>

bool Write(T const& t);

template <typename T>

bool WriteArray(T \* t, size\_t len);

bool WriteUtf8String(wchar\_t const \* str);

bool Seek(long offset);

bool SeekToEnd();

long Size();

void Close(bool deleteFile = false);

private:

wchar\_t const \* filename;

FILE \* file;

bool deleteNonClosed;

};

DynamicProfileStorageReaderWriter::~DynamicProfileStorageReaderWriter()

{

if (file)

{

Close(deleteNonClosed);

}

}

bool DynamicProfileStorageReaderWriter::Init(wchar\_t const \* filename, wchar\_t const \* mode, bool deleteNonClosed, errno\_t \* err = nullptr)

{

Assert(file == nullptr);

errno\_t e = \_wfopen\_s(&file, filename, mode);

if (e != 0)

{

if (err)

{

\*err = e;

}

return false;

}

this->filename = filename;

this->deleteNonClosed = deleteNonClosed;

return true;

}

template <typename T>

bool DynamicProfileStorageReaderWriter::Read(T \* t)

{

return ReadArray(t, 1);

}

template <typename T>

bool DynamicProfileStorageReaderWriter::ReadArray(T \* t, size\_t len)

{

Assert(file);

long pos = ftell(file);

if (fread(t, sizeof(T), len, file) != len)

{

Output::Print(L"ERROR: DynamicProfileStorage: '%s': File corrupted at %d\n", filename, pos);

Output::Flush();

return false;

}

return true;

}

\_Success\_(return) bool DynamicProfileStorageReaderWriter::ReadUtf8String(\_\_deref\_out\_z wchar\_t \*\* str, \_\_out DWORD \* len)

{

DWORD urllen;

if (!Read(&urllen))

{

return false;

}

utf8char\_t \* tempBuffer = NoCheckHeapNewArray(utf8char\_t, urllen);

if (tempBuffer == nullptr)

{

Output::Print(L"ERROR: DynamicProfileStorage: Out of memory reading '%s'\n", filename);

Output::Flush();

return false;

}

if (!ReadArray(tempBuffer, urllen))

{

HeapDeleteArray(urllen, tempBuffer);

return false;

}

charcount\_t length = utf8::ByteIndexIntoCharacterIndex(tempBuffer, urllen);

wchar\_t \* name = NoCheckHeapNewArray(wchar\_t, length + 1);

if (name == nullptr)

{

Output::Print(L"ERROR: DynamicProfileStorage: Out of memory reading '%s'\n", filename);

Output::Flush();

HeapDeleteArray(urllen, tempBuffer);

return false;

}

utf8::DecodeIntoAndNullTerminate(name, tempBuffer, length);

NoCheckHeapDeleteArray(urllen, tempBuffer);

\*str = name;

\*len = length;

return true;

}

template <typename T>

bool DynamicProfileStorageReaderWriter::Write(T const& t)

{

return WriteArray(&t, 1);

}

template <typename T>

bool DynamicProfileStorageReaderWriter::WriteArray(T \* t, size\_t len)

{

Assert(file);

if (fwrite(t, sizeof(T), len, file) != len)

{

Output::Print(L"ERROR: DynamicProfileStorage: Unable to write to file '%s'\n", filename);

Output::Flush();

return false;

}

return true;

}

bool DynamicProfileStorageReaderWriter::WriteUtf8String(wchar\_t const \* str)

{

charcount\_t len = static\_cast<charcount\_t>(wcslen(str));

utf8char\_t \* tempBuffer = NoCheckHeapNewArray(utf8char\_t, len \* 3);

if (tempBuffer == nullptr)

{

Output::Print(L"ERROR: DynamicProfileStorage: Out of memory writing to file '%s'\n", filename);

Output::Flush();

return false;

}

DWORD cbNeeded = (DWORD)utf8::EncodeInto(tempBuffer, str, len);

bool success = Write(cbNeeded) && WriteArray(tempBuffer, cbNeeded);

NoCheckHeapDeleteArray(len \* 3, tempBuffer);

return success;

}

bool DynamicProfileStorageReaderWriter::Seek(long offset)

{

Assert(file);

return fseek(file, offset, SEEK\_SET) == 0;

}

bool DynamicProfileStorageReaderWriter::SeekToEnd()

{

Assert(file);

return fseek(file, 0, SEEK\_END) == 0;

}

long DynamicProfileStorageReaderWriter::Size()

{

Assert(file);

long current = ftell(file);

SeekToEnd();

long end = ftell(file);

fseek(file, current, SEEK\_SET);

return end;

}

void DynamicProfileStorageReaderWriter::Close(bool deleteFile)

{

Assert(file);

fflush(file);

fclose(file);

file = nullptr;

if (deleteFile)

{

\_wunlink(filename);

}

filename = nullptr;

}

void DynamicProfileStorage::StorageInfo::GetFilename(\_Out\_writes\_z\_(\_MAX\_PATH) wchar\_t filename[\_MAX\_PATH]) const

{

wchar\_t tempFile[\_MAX\_PATH];

wcscpy\_s(tempFile, L"jsdpcache\_file");

\_itow\_s(this->fileId, tempFile + \_countof(L"jsdpcache\_file") - 1, \_countof(tempFile) - \_countof(L"jsdpcache\_file") + 1, 10);

\_wmakepath\_s(filename, \_MAX\_PATH, cacheDrive, cacheDir, tempFile, L".dpd");

}

char const \* DynamicProfileStorage::StorageInfo::ReadRecord() const

{

wchar\_t cacheFilename[\_MAX\_PATH];

this->GetFilename(cacheFilename);

DynamicProfileStorageReaderWriter reader;

if (!reader.Init(cacheFilename, L"rb", false))

{

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Unable to open cache dir file '%s'", cacheFilename);

Output::Flush();

}

#endif

return nullptr;

}

long size = reader.Size();

char \* record = AllocRecord(size);

if (record == nullptr)

{

Output::Print(L"ERROR: DynamicProfileStorage: Out of memory reading '%s'", cacheFilename);

Output::Flush();

return nullptr;

}

if (!reader.ReadArray(GetRecordBuffer(record), size))

{

DeleteRecord(record);

return nullptr;

}

return record;

}

bool DynamicProfileStorage::StorageInfo::WriteRecord(\_\_in\_ecount(sizeof(DWORD) + \*record)char const \* record) const

{

wchar\_t cacheFilename[\_MAX\_PATH];

this->GetFilename(cacheFilename);

DynamicProfileStorageReaderWriter writer;

if (!writer.Init(cacheFilename, L"wcb", true))

{

Output::Print(L"ERROR: DynamicProfileStorage: Unable open record file '%s'", cacheFilename);

Output::Flush();

return false;

}

if (!writer.WriteArray(GetRecordBuffer(record), GetRecordSize(record)))

{

return false;

}

// Success

writer.Close();

return true;

}

#if DBG\_DUMP

bool DynamicProfileStorage::DoTrace()

{

return Js::Configuration::Global.flags.Trace.IsEnabled(Js::DynamicProfileStoragePhase);

}

#endif

wchar\_t const \* DynamicProfileStorage::GetMessageType()

{

if (!DynamicProfileStorage::DoCollectInfo())

{

return L"WARNING";

}

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

return L"TRACE";

}

#endif

return nullptr;

}

bool DynamicProfileStorage::Initialize()

{

AssertMsg(!initialized, "Initialize called multiple times");

if (initialized)

{

return true;

}

bool success = true;

initialized = true;

#ifdef FORCE\_DYNAMIC\_PROFILE\_STORAGE

enabled = true;

collectInfo = true;

if (!SetupCacheDir(nullptr))

{

success = false;

}

#else

if (Js::Configuration::Global.flags.IsEnabled(Js::DynamicProfileCacheDirFlag))

{

enabled = true;

collectInfo = true;

if (!SetupCacheDir(Js::Configuration::Global.flags.DynamicProfileCacheDir))

{

success = false;

}

}

#endif

// If -DynamicProfileInput is specified, the file specified in -DynamicProfileCache

// will not be imported and will be overwritten

if (Js::Configuration::Global.flags.IsEnabled(Js::DynamicProfileInputFlag))

{

enabled = true;

ClearCacheCatalog();

// -DynamicProfileInput

// Without other -DynamicProfile flags - enable in memory profile cache without exporting

// With -DyanmicProfileCache - override the dynamic profile cache file

// With -DyanmicProfileCacheDir - clear the dynamic profile cache directory

if (Js::Configuration::Global.flags.DynamicProfileInput != nullptr)

{

// Error if we can't in the profile info if we are not using a cache file or directory.

collectInfo = collectInfo || Js::Configuration::Global.flags.IsEnabled(Js::DynamicProfileCacheFlag);

// Try to open the DynamicProfileInput.

// If failure to open, retry at 100 ms intervals until a timeout.

const uint32 MAX\_DELAY = 2000; // delay at most 2 seconds

const uint32 DELAY\_INTERVAL = 100;

const uint32 MAX\_TRIES = MAX\_DELAY / DELAY\_INTERVAL;

bool readSuccessful = false;

for (uint32 i = 0; i < MAX\_TRIES; i++)

{

readSuccessful = ImportFile(Js::Configuration::Global.flags.DynamicProfileInput, false);

if (readSuccessful)

{

break;

}

Sleep(DELAY\_INTERVAL);

if (Js::Configuration::Global.flags.Verbose)

{

Output::Print(L" Retrying load of dynamic profile from '%s' (attempt %d)...\n",

Js::Configuration::Global.flags.DynamicProfileInput, i + 1);

Output::Flush();

}

}

if (!readSuccessful)

{

// If file cannot be read, behave as if DynamicProfileInput == null.

collectInfo = true;

}

}

else

{

// Don't error if we can't find the profile info

collectInfo = true;

}

}

else if (Js::Configuration::Global.flags.IsEnabled(Js::DynamicProfileCacheFlag))

{

enabled = true;

collectInfo = true;

if (Js::Configuration::Global.flags.DynamicProfileCache)

{

if (!ImportFile(Js::Configuration::Global.flags.DynamicProfileCache, true))

{

success = false;

}

}

}

return success;

}

// We used to have problem with dynamic profile being corrupt and this is to verify it.

// We don't see this any more so we will just disable it to speed up unittest

#if 0

#if DBG && defined(\_M\_AMD64)

#define DYNAMIC\_PROFILE\_EXPORT\_FILE\_CHECK

#endif

#endif

bool DynamicProfileStorage::Uninitialize()

{

AssertMsg(!uninitialized, "Uninitialize called multiple times");

if (!initialized || uninitialized)

{

return true;

}

#ifdef DYNAMIC\_PROFILE\_EXPORT\_FILE\_CHECK

bool exportFile = false;

#endif

uninitialized = true;

bool success = true;

if (Js::Configuration::Global.flags.DynamicProfileCache != nullptr)

{

Assert(enabled);

if (!ExportFile(Js::Configuration::Global.flags.DynamicProfileCache))

{

success = false;

}

#ifdef DYNAMIC\_PROFILE\_EXPORT\_FILE\_CHECK

exportFile = true;

#endif

}

if (mutex != nullptr)

{

CloseHandle(mutex);

}

#ifdef DYNAMIC\_PROFILE\_EXPORT\_FILE\_CHECK

ulong oldCount = infoMap.Count();

#endif

ClearInfoMap(false);

#ifdef DYNAMIC\_PROFILE\_EXPORT\_FILE\_CHECK

if (exportFile)

{

HRESULT hr;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT

{

if (!ImportFile(Js::Configuration::Global.flags.DynamicProfileCache, false))

{

success = false;

}

Assert(oldCount == infoMap.Count());

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT(hr)

ClearInfoMap(false);

}

#endif

return success;

}

void DynamicProfileStorage::ClearInfoMap(bool deleteFileStorage)

{

uint recordCount = infoMap.Count();

uint recordFreed = 0;

for (uint i = 0; recordFreed < recordCount; i++)

{

wchar\_t const \* name = infoMap.GetKeyAt(i);

if (name == nullptr)

{

continue;

}

NoCheckHeapDeleteArray(wcslen(name) + 1, name);

StorageInfo const& info = infoMap.GetValueAt(i);

if (info.isFileStorage)

{

Assert(useCacheDir);

if (deleteFileStorage)

{

wchar\_t filename[\_MAX\_PATH];

info.GetFilename(filename);

\_wunlink(filename);

}

}

else

{

DeleteRecord(info.record);

}

recordFreed++;

}

infoMap.Clear();

}

bool DynamicProfileStorage::ImportFile(\_\_in\_z wchar\_t const \* filename, bool allowNonExistingFile)

{

Assert(enabled);

DynamicProfileStorageReaderWriter reader;

errno\_t e;

if (!reader.Init(filename, L"rb", false, &e))

{

if (allowNonExistingFile)

{

return true;

}

else

{

if (Js::Configuration::Global.flags.Verbose)

{

Output::Print(L"ERROR: DynamicProfileStorage: Unable to open file '%s' to import (%d)\n", filename, e);

wchar\_t error\_string[256];

\_wcserror\_s(error\_string, e);

Output::Print(L"ERROR: For file '%s': %s (%d)\n", filename, error\_string, e);

Output::Flush();

}

return false;

}

}

DWORD magic;

DWORD version;

DWORD recordCount;

if (!reader.Read(&magic)

|| !reader.Read(&version)

|| !reader.Read(&recordCount))

{

return false;

}

if (magic != MagicNumber)

{

Output::Print(L"ERROR: DynamicProfileStorage: '%s' is not a dynamic profile data file", filename);

Output::Flush();

return false;

}

if (version != FileFormatVersion)

{

if (allowNonExistingFile)

{

// Treat version mismatch as non-existent file

return true;

}

Output::Print(L"ERROR: DynamicProfileStorage: '%s' has format version %d; version %d expected", filename,

version, FileFormatVersion);

Output::Flush();

return false;

}

for (uint i = 0; i < recordCount; i++)

{

DWORD len;

wchar\_t \* name;

if (!reader.ReadUtf8String(&name, &len))

{

Assert(false);

return false;

}

DWORD recordLen;

if (!reader.Read(&recordLen))

{

Assert(false);

return false;

}

char \* record = AllocRecord(recordLen);

if (record == nullptr)

{

Output::Print(L"ERROR: DynamicProfileStorage: Out of memory importing '%s'\n", filename);

Output::Flush();

NoCheckHeapDeleteArray(len + 1, name);

return false;

}

if (!reader.ReadArray(GetRecordBuffer(record), recordLen))

{

NoCheckHeapDeleteArray(len + 1, name);

DeleteRecord(record);

Assert(false);

return false;

}

SaveRecord(name, record);

// Save record will make a copy of the name

NoCheckHeapDeleteArray(len + 1, name);

}

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Imported file: '%s'\n", filename);

Output::Flush();

}

#endif

AssertMsg(recordCount == (uint)infoMap.Count(), "failed to read all the records");

return true;

}

bool DynamicProfileStorage::ExportFile(\_\_in\_z wchar\_t const \* filename)

{

Assert(enabled);

if (useCacheDir && AcquireLock())

{

if (!LoadCacheCatalog()) // refresh the cache catalog

{

ReleaseLock();

Assert(FALSE);

return false;

}

}

DynamicProfileStorageReaderWriter writer;

if (!writer.Init(filename, L"wcb", true))

{

Output::Print(L"ERROR: DynamicProfileStorage: Unable to open file '%s' to export\n", filename);

Output::Flush();

return false;

}

DWORD recordCount = infoMap.Count();

if (!writer.Write(MagicNumber)

|| !writer.Write(FileFormatVersion)

|| !writer.Write(recordCount))

{

Assert(FALSE);

return false;

}

uint recordWritten = 0;

for (uint i = 0; recordWritten < recordCount; i++)

{

wchar\_t const \* url = infoMap.GetKeyAt(i);

if (url == nullptr)

{

Assert(false);

continue;

}

StorageInfo const& info = infoMap.GetValueAt(i);

char const \* record;

if (info.isFileStorage)

{

Assert(useCacheDir);

record = info.ReadRecord();

if (record == nullptr)

{

ReleaseLock();

Assert(FALSE);

return false;

}

}

else

{

Assert(!useCacheDir);

Assert(!locked);

record = info.record;

}

DWORD recordSize = GetRecordSize(record);

bool failed = (!writer.WriteUtf8String(url)

|| !writer.Write(recordSize)

|| !writer.WriteArray(GetRecordBuffer(record), recordSize));

if (useCacheDir)

{

DeleteRecord(record);

}

if (failed)

{

if (useCacheDir)

{

ReleaseLock();

}

Assert(FALSE);

return false;

}

recordWritten++;

}

writer.Close();

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Exported file: '%s'\n", filename);

Output::Flush();

}

#endif

return true;

}

void DynamicProfileStorage::DisableCacheDir()

{

Assert(useCacheDir);

ClearInfoMap(false);

useCacheDir = false;

#ifdef FORCE\_DYNAMIC\_PROFILE\_STORAGE

Js::Throw::FatalInternalError();

#endif

}

bool DynamicProfileStorage::AcquireLock()

{

Assert(mutex != nullptr);

Assert(!locked);

DWORD ret = WaitForSingleObject(mutex, INFINITE);

if (ret == WAIT\_OBJECT\_0 || ret == WAIT\_ABANDONED)

{

#if DBG

locked = true;

#endif

return true;

}

Output::Print(L"ERROR: DynamicProfileStorage: Unable to acquire mutex %d\n", ret);

Output::Flush();

DisableCacheDir();

return false;

}

bool DynamicProfileStorage::ReleaseLock()

{

Assert(locked);

Assert(mutex != nullptr);

#if DBG

locked = false;

#endif

if (ReleaseMutex(mutex))

{

return true;

}

DisableCacheDir();

Output::Print(L"ERROR: DynamicProfileStorage: Unable to release mutex");

Output::Flush();

return false;

}

bool DynamicProfileStorage::SetupCacheDir(\_\_in\_z wchar\_t const \* dirname)

{

Assert(enabled);

mutex = CreateMutex(NULL, FALSE, L"JSDPCACHE");

if (mutex == nullptr)

{

Output::Print(L"ERROR: DynamicProfileStorage: Unable to create mutex");

Output::Flush();

return false;

}

useCacheDir = true;

if (!AcquireLock())

{

return false;

}

wchar\_t tempPath[\_MAX\_PATH];

if (dirname == nullptr)

{

ulong len = GetTempPath(\_MAX\_PATH, tempPath);

if (len >= \_MAX\_PATH || wcscat\_s(tempPath, L"jsdpcache") != 0)

{

DisableCacheDir();

Output::Print(L"ERROR: DynamicProfileStorage: Can't setup cache directory: Unable to create directory\n");

Output::Flush();

ReleaseLock();

return false;

}

if (!CreateDirectory(tempPath, NULL) && GetLastError() != ERROR\_ALREADY\_EXISTS)

{

DisableCacheDir();

Output::Print(L"ERROR: DynamicProfileStorage: Can't setup cache directory: Unable to create directory\n");

Output::Flush();

ReleaseLock();

return false;

}

dirname = tempPath;

}

wchar\_t cacheFile[\_MAX\_FNAME];

wchar\_t cacheExt[\_MAX\_EXT];

\_wsplitpath\_s(dirname, cacheDrive, cacheDir, cacheFile, cacheExt);

wcscat\_s(cacheDir, cacheFile);

wcscat\_s(cacheDir, cacheExt);

\_wmakepath\_s(catalogFilename, cacheDrive, cacheDir, L"jsdpcache\_master", L".dpc");

bool succeed = LoadCacheCatalog();

ReleaseLock();

return succeed;

}

bool DynamicProfileStorage::CreateCacheCatalog()

{

Assert(enabled);

Assert(useCacheDir);

Assert(locked);

nextFileId = 0;

creationTime = \_time32(NULL);

DynamicProfileStorageReaderWriter catalogFile;

if (!catalogFile.Init(catalogFilename, L"wb", true)

|| !catalogFile.Write(MagicNumber)

|| !catalogFile.Write(FileFormatVersion)

|| !catalogFile.Write(creationTime)

|| !catalogFile.Write(0)) // count

{

DisableCacheDir();

Output::Print(L"ERROR: DynamicProfileStorage: Unable to create cache catalog\n");

Output::Flush();

return false;

}

lastOffset = catalogFile.Size();

ClearInfoMap(true);

catalogFile.Close();

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Cache directory catalog created: '%s'\n", catalogFilename);

Output::Flush();

}

#endif

return true;

}

bool DynamicProfileStorage::AppendCacheCatalog(\_\_in\_z wchar\_t const \* url)

{

Assert(enabled);

Assert(useCacheDir);

Assert(locked);

DWORD magic;

DWORD version;

DWORD count;

DWORD time;

DynamicProfileStorageReaderWriter catalogFile;

if (!catalogFile.Init(catalogFilename, L"rcb+", false))

{

return CreateCacheCatalog();

}

if (!catalogFile.Seek(0)

|| !catalogFile.Read(&magic)

|| !catalogFile.Read(&version)

|| !catalogFile.Read(&time)

|| !catalogFile.Read(&count)

|| magic != MagicNumber

|| version < FileFormatVersion)

{

catalogFile.Close();

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Overwriting file for cache directory catalog: '%s'\n", catalogFilename);

Output::Flush();

}

#endif

return CreateCacheCatalog();

}

if (version > FileFormatVersion)

{

DisableCacheDir();

Output::Print(L"ERROR: DynamicProfileStorage: Existing cache catalog has a newer format\n");

Output::Flush();

return false;

}

if (time != creationTime || count + 1 != nextFileId)

{

// This should not happen, as we are under lock from the LoadCacheCatalog

DisableCacheDir();

Output::Print(L"ERROR: DynamicProfileStorage: Internal error, file modified under lock\n");

Output::Flush();

return false;

}

if (!catalogFile.SeekToEnd() ||

!catalogFile.WriteUtf8String(url) ||

!catalogFile.Seek(3 \* sizeof(DWORD)) ||

!catalogFile.Write(nextFileId))

{

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Write failure. Cache directory catalog corrupted: '%s'\n", catalogFilename);

Output::Flush();

}

#endif

catalogFile.Close();

return CreateCacheCatalog();

}

lastOffset = catalogFile.Size();

return true;

}

bool DynamicProfileStorage::LoadCacheCatalog()

{

Assert(enabled);

Assert(useCacheDir);

Assert(locked);

DynamicProfileStorageReaderWriter catalogFile;

DWORD magic;

DWORD version;

DWORD count;

DWORD time;

if (!catalogFile.Init(catalogFilename, L"rb", false))

{

return CreateCacheCatalog();

}

if (!catalogFile.Read(&magic)

|| !catalogFile.Read(&version)

|| !catalogFile.Read(&time)

|| !catalogFile.Read(&count)

|| magic != MagicNumber

|| version < FileFormatVersion)

{

catalogFile.Close();

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Overwriting file for cache directory catalog: '%s'\n", catalogFilename);

Output::Flush();

}

#endif

return CreateCacheCatalog();

}

if (version > FileFormatVersion)

{

DisableCacheDir();

Output::Print(L"ERROR: DynamicProfileStorage: Existing cache catalog has a newer format.\n");

Output::Flush();

return false;

}

DWORD start = 0;

Assert(useCacheDir);

if (time == creationTime)

{

// We can reuse existing data

start = infoMap.Count();

Assert(count >= start);

Assert(catalogFile.Size() >= lastOffset);

if (count == nextFileId)

{

Assert(catalogFile.Size() == lastOffset);

return true;

}

if (!catalogFile.Seek(lastOffset))

{

catalogFile.Close();

Output::Print(L"ERROR: DynamicProfileStorage: Unable to seek to last known offset\n");

Output::Flush();

return CreateCacheCatalog();

}

}

else if (creationTime != 0)

{

Output::Print(L"WARNING: DynamicProfileStorage: Reloading full catalog\n");

Output::Flush();

}

for (DWORD i = start; i < count; i++)

{

DWORD len;

wchar\_t \* url;

if (!catalogFile.ReadUtf8String(&url, &len))

{

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Cache dir catalog file corrupted: '%s'\n", catalogFilename);

Output::Flush();

}

#endif

// REVIEW: the file is corrupted, should we not flush the cache totally?

catalogFile.Close();

return CreateCacheCatalog();

}

StorageInfo \* oldInfo;

if (infoMap.TryGetReference(url, &oldInfo))

{

Assert(oldInfo->isFileStorage);

oldInfo->fileId = i;

}

else

{

StorageInfo newInfo;

newInfo.isFileStorage = true;

newInfo.fileId = i;

infoMap.Add(url, newInfo);

}

}

#if DBG\_DUMP

if (creationTime == 0 && DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Cache directory catalog loaded: '%s'\n", catalogFilename);

Output::Flush();

}

#endif

nextFileId = count;

creationTime = time;

lastOffset = catalogFile.Size();

return true;

}

void DynamicProfileStorage::ClearCacheCatalog()

{

Assert(enabled);

if (useCacheDir)

{

if (!AcquireLock())

{

return;

}

bool success = CreateCacheCatalog();

ReleaseLock();

if (success)

{

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Cache dir clears\n");

Output::Flush();

}

#endif

return;

}

}

else

{

ClearInfoMap(false);

}

}

void DynamicProfileStorage::SaveRecord(\_\_in\_z wchar\_t const \* filename, \_\_in\_ecount(sizeof(DWORD) + \*record) char const \* record)

{

Assert(enabled);

AutoCriticalSection autocs(&cs);

StorageInfo \* info;

if (useCacheDir && AcquireLock())

{

if (!LoadCacheCatalog()) // refresh the cache catalog

{

ReleaseLock();

}

}

if (infoMap.TryGetReference(filename, &info))

{

if (!info->isFileStorage)

{

Assert(!useCacheDir);

if (info->record != nullptr)

{

DeleteRecord(info->record);

}

info->record = record;

return;

}

Assert(useCacheDir);

wchar\_t cacheFilename[\_MAX\_PATH];

info->GetFilename(cacheFilename);

DynamicProfileStorageReaderWriter writer;

if (info->WriteRecord(record))

{

// Success

ReleaseLock();

return;

}

// Fail, try to add it again

info->fileId = nextFileId++;

if (info->WriteRecord(record))

{

if (AppendCacheCatalog(filename))

{

ReleaseLock();

return;

}

}

else

{

DisableCacheDir();

}

// Can't add a new file. Disable and use memory mode

Assert(!useCacheDir);

ReleaseLock();

}

size\_t len = wcslen(filename) + 1;

wchar\_t \* newFilename = NoCheckHeapNewArray(wchar\_t, len);

if (newFilename == nullptr)

{

// out of memory, don't save anything

AssertMsg(false, "OOM");

DeleteRecord(record);

if (useCacheDir)

{

ReleaseLock();

}

return;

}

wmemcpy\_s(newFilename, len, filename, len);

StorageInfo newInfo;

if (useCacheDir)

{

newInfo.isFileStorage = true;

newInfo.fileId = nextFileId++;

if (newInfo.WriteRecord(record))

{

infoMap.Add(newFilename, newInfo);

if (AppendCacheCatalog(newFilename))

{

ReleaseLock();

return;

}

}

// Can't even add a record. Disable and use memory mode

DisableCacheDir();

ReleaseLock();

}

Assert(!useCacheDir);

Assert(!locked);

newInfo.isFileStorage = false;

newInfo.record = record;

infoMap.Add(newFilename, newInfo);

}

char \* DynamicProfileStorage::AllocRecord(DWORD bufferSize)

{

Assert(enabled);

char \* buffer = NoCheckHeapNewArray(char, bufferSize + sizeof(DWORD));

if (buffer != nullptr)

{

\*(DWORD \*)buffer = bufferSize;

}

return buffer;

}

DWORD DynamicProfileStorage::GetRecordSize(\_\_in\_ecount(sizeof(DWORD) + \*buffer) char const \* buffer)

{

Assert(enabled);

return \*(DWORD \*)buffer;

}

char const \* DynamicProfileStorage::GetRecordBuffer(\_\_in\_ecount(sizeof(DWORD) + \*buffer) char const \* buffer)

{

Assert(enabled);

return buffer + sizeof(DWORD);

}

char \* DynamicProfileStorage::GetRecordBuffer(\_\_in\_ecount(sizeof(DWORD) + \*buffer) char \* buffer)

{

Assert(enabled);

return buffer + sizeof(DWORD);

}

void DynamicProfileStorage::DeleteRecord(\_\_in\_ecount(sizeof(DWORD) + \*buffer) char const \* buffer)

{

Assert(enabled);

NoCheckHeapDeleteArray(GetRecordSize(buffer) + sizeof(DWORD), buffer);

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifdef DYNAMIC\_PROFILE\_STORAGE

class DynamicProfileStorage

{

public:

static bool Initialize();

static bool Uninitialize();

static bool IsEnabled() { return enabled; }

static bool DoCollectInfo() { return collectInfo; }

template <typename Fn>

static Js::SourceDynamicProfileManager \* Load(\_\_in\_z wchar\_t const \* filename, Fn loadFn);

static void SaveRecord(\_\_in\_z wchar\_t const \* filename, \_\_in\_ecount(sizeof(DWORD) + \*record) char const \* record);

static char \* AllocRecord(DWORD bufferSize);

static void DeleteRecord(\_\_in\_ecount(sizeof(DWORD) + \*record) char const \* record);

static char const \* GetRecordBuffer(\_\_in\_ecount(sizeof(DWORD) + \*record) char const \* record);

static char \* GetRecordBuffer(\_\_in\_ecount(sizeof(DWORD) + \*record) char \* record);

static DWORD GetRecordSize(\_\_in\_ecount(sizeof(DWORD) + \*record) char const \* record);

private:

static wchar\_t const \* GetMessageType();

static void ClearInfoMap(bool deleteFileStorage);

static bool ImportFile(\_\_in\_z wchar\_t const \* filename, bool allowNonExistingFile);

static bool ExportFile(\_\_in\_z wchar\_t const \* filename);

static bool SetupCacheDir(\_\_in\_z wchar\_t const \* dirname);

static void DisableCacheDir();

static bool CreateCacheCatalog();

static void ClearCacheCatalog();

static bool LoadCacheCatalog();

static bool AppendCacheCatalog(\_\_in\_z wchar\_t const \* url);

static bool AcquireLock();

static bool ReleaseLock();

static bool VerifyHeader();

static bool initialized;

static bool uninitialized;

static bool enabled;

static bool collectInfo;

static bool useCacheDir;

static wchar\_t cacheDrive[\_MAX\_DRIVE];

static wchar\_t cacheDir[\_MAX\_DIR];

static wchar\_t catalogFilename[\_MAX\_PATH];

static DWORD const MagicNumber;

static DWORD const FileFormatVersion;

static DWORD creationTime;

static long lastOffset;

static HANDLE mutex;

static CriticalSection cs;

static DWORD nextFileId;

#if DBG

static bool locked;

#endif

#if DBG\_DUMP

static bool DoTrace();

#endif

class StorageInfo

{

public:

void GetFilename(\_Out\_writes\_z\_(\_MAX\_PATH) wchar\_t filename[\_MAX\_PATH]) const;

char const \* ReadRecord() const;

bool WriteRecord(\_\_in\_ecount(sizeof(DWORD) + \*record) char const \* record) const;

bool isFileStorage;

union

{

DWORD fileId;

char const \* record;

};

};

typedef JsUtil::BaseDictionary<wchar\_t const \*, StorageInfo, NoCheckHeapAllocator, PrimeSizePolicy, DefaultComparer, JsUtil::DictionaryEntry> InfoMap;

static InfoMap infoMap;

};

template <class Fn>

Js::SourceDynamicProfileManager \*

DynamicProfileStorage::Load(wchar\_t const \* filename, Fn loadFn)

{

Assert(DynamicProfileStorage::IsEnabled());

AutoCriticalSection autocs(&cs);

if (useCacheDir && AcquireLock())

{

LoadCacheCatalog(); // refresh the cache catalog

}

StorageInfo \* info;

if (!infoMap.TryGetReference(filename, &info))

{

if (useCacheDir)

{

ReleaseLock();

}

#if !DBG || !defined(\_M\_AMD64)

wchar\_t const \* messageType = GetMessageType();

if (messageType)

{

Output::Print(L"%s: DynamicProfileStorage: Dynamic Profile Data not found for '%s'\n", messageType, filename);

Output::Flush();

}

#endif

return nullptr;

}

char const \* record;

if (info->isFileStorage)

{

Assert(useCacheDir);

Assert(locked);

record = info->ReadRecord();

ReleaseLock();

if (record == nullptr)

{

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace())

{

Output::Print(L"TRACE: DynamicProfileStorage: Faile to load from cache dir for '%s'", filename);

Output::Flush();

}

#endif

return nullptr;

}

}

else

{

record = info->record;

}

Js::SourceDynamicProfileManager \* sourceDynamicProfileManager = loadFn(GetRecordBuffer(record), GetRecordSize(record));

if (info->isFileStorage)

{

// The data is backed by a file, we can delete the memory

Assert(useCacheDir);

DeleteRecord(record);

}

#if DBG\_DUMP

if (DynamicProfileStorage::DoTrace() && sourceDynamicProfileManager)

{

Output::Print(L"TRACE: DynamicProfileStorage: Dynamic Profile Data Loaded: '%s'\n", filename);

}

#endif

if (sourceDynamicProfileManager == nullptr)

{

wchar\_t const \* messageType = GetMessageType();

if (messageType)

{

Output::Print(L"%s: DynamicProfileStorage: Dynamic Profile Data corrupted: '%s'\n", messageType, filename);

Output::Flush();

}

}

return sourceDynamicProfileManager;

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

class EHBailoutData

{

public:

int32 nestingDepth;

int32 catchOffset;

EHBailoutData \* parent;

EHBailoutData \* child;

public:

EHBailoutData() : nestingDepth(-1), catchOffset(0), parent(nullptr), child(nullptr) {}

EHBailoutData(int32 nestingDepth, int32 catchOffset, EHBailoutData \* parent)

{

this->nestingDepth = nestingDepth;

this->catchOffset = catchOffset;

this->parent = parent;

this->child = nullptr;

}

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

// Use as the top level comparer for two level dictionary. Note that two

// values are equal as long as their fastHash is the same (and moduleID/isStrict is the same).

// This comparer is used for the top level dictionary in two level evalmap dictionary.

template <class T>

struct FastEvalMapStringComparer

{

static bool Equals(T left, T right)

{

return (left.hash == right.hash) &&

(left.moduleID == right.moduleID) &&

(left.IsStrict() == right.IsStrict());

}

static hash\_t GetHashCode(T t)

{

return (hash\_t)t;

}

};

// The value in top level of two level dictionary. It might contain only the single value

// (TValue), or a second level dictionary.

template <class TKey, class TValue, class SecondaryDictionary, class NestedKey>

class TwoLevelHashRecord

{

public:

TwoLevelHashRecord(TValue newValue) :

singleValue(true), value(newValue) {}

TwoLevelHashRecord() :

singleValue(true), value(nullptr) {}

SecondaryDictionary\* GetDictionary()

{

Assert(!singleValue);

return nestedMap;

}

bool TryGetValue(TKey& key, TValue\* value)

{

if (IsValue())

{

\*value = GetValue();

return true;

}

return GetDictionary()->TryGetValue(key, value);

}

void Add(const TKey& key, TValue& newValue)

{

Assert(!singleValue);

NestedKey nestedKey;

ConvertKey(key, nestedKey);

nestedMap->Item(nestedKey, newValue);

#ifdef PROFILE\_EVALMAP

if (Configuration::Global.flags.ProfileEvalMap)

{

Output::Print(L"EvalMap fastcache collision:\t key = %d count = %d\n", (hash\_t)key, nestedMap->Count());

}

#endif

}

void Remove(const TKey& key)

{

Assert(!singleValue);

NestedKey nestedKey;

ConvertKey(key, nestedKey);

nestedMap->Remove(nestedKey);

}

void ConvertToDictionary(TKey& key, Recycler\* recycler)

{

Assert(singleValue);

SecondaryDictionary\* dictionary = RecyclerNew(recycler, SecondaryDictionary, recycler);

auto newValue = value;

nestedMap = dictionary;

singleValue = false;

Add(key, newValue);

}

bool IsValue() const { return singleValue; }

TValue GetValue() const { Assert(singleValue); return value; }

bool IsDictionaryEntry() const { return !singleValue; }

private:

bool singleValue;

union

{

TValue value;

SecondaryDictionary\* nestedMap;

};

};

// The two level dictionary. top level needs to be either simple hash value, or

// key needs to be equals for all nested values.

template <class Key, class Value, class EntryRecord, class TopLevelDictionary, class NestedKey>

class TwoLevelHashDictionary

{

template <class T, class Value>

class AutoRestoreSetInAdd

{

public:

AutoRestoreSetInAdd(T\* instance, Value value) :

instance(instance), value(value)

{

instance->SetIsInAdd(value);

}

~AutoRestoreSetInAdd()

{

instance->SetIsInAdd(!value);

}

private:

T\* instance;

Value value;

};

public:

TwoLevelHashDictionary(TopLevelDictionary\* cache, Recycler\* recycler) :

dictionary(cache),

recycler(recycler)

{

}

bool TryGetValue(const Key& key, Value\* value)

{

EntryRecord\*\* entryRecord;

Key cachedKey;

int index;

bool success = dictionary->TryGetReference(key, &entryRecord, &index);

if (success && ((\*entryRecord) != nullptr))

{

cachedKey = dictionary->GetKeyAt(index);

if ((\*entryRecord)->IsValue())

{

success = (cachedKey == key);

if (success)

{

\*value = (\*entryRecord)->GetValue();

}

}

else

{

NestedKey nestedKey;

ConvertKey(key, nestedKey);

success = (\*entryRecord)->GetDictionary()->TryGetValue(nestedKey, value);

}

}

else

{

success = false;

}

return success;

}

TopLevelDictionary\* GetDictionary() const { return dictionary; }

void NotifyAdd(const Key& key)

{

dictionary->NotifyAdd(key);

}

void Add(const Key& key, Value value)

{

EntryRecord\*\* entryRecord;

int index;

bool success = dictionary->TryGetReference(key, &entryRecord, &index);

if (success && ((\*entryRecord) != nullptr))

{

AutoRestoreSetInAdd<TopLevelDictionary, bool> autoRestoreSetInAdd(this->dictionary, true);

if ((\*entryRecord)->IsValue())

{

Key oldKey = dictionary->GetKeyAt(index);

(\*entryRecord)->ConvertToDictionary(oldKey, recycler);

}

(\*entryRecord)->Add(key, value);

}

else

{

EntryRecord\* newRecord = RecyclerNew(recycler, EntryRecord, value);

dictionary->Add(key, newRecord);

#ifdef PROFILE\_EVALMAP

if (Configuration::Global.flags.ProfileEvalMap)

{

Output::Print(L"EvalMap fastcache set:\t key = %d \n", (hash\_t)key);

}

#endif

}

}

private:

TopLevelDictionary\* dictionary;

Recycler\* recycler;

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

static const char \*const ExecutionModeNames[] =

{

#define EXECUTION\_MODE(name) "" STRINGIZE(name) "",

#include "ExecutionModes.h"

#undef EXECUTION\_MODE

};

const char \*ExecutionModeName(const ExecutionMode executionMode)

{

Assert(executionMode < ExecutionMode::Count);

return ExecutionModeNames[static\_cast<size\_t>(executionMode)];

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

enum class ExecutionMode : uint8

{

#define EXECUTION\_MODE(name) name,

#include "ExecutionModes.h"

#undef EXECUTION\_MODE

};

ENUM\_CLASS\_HELPERS(ExecutionMode, uint8);

extern const char \*ExecutionModeName(const ExecutionMode executionMode);

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

// Non-profiling interpreter

// - For instance, it is used for NoNative mode

// - Does not transition to other execution modes

EXECUTION\_MODE(Interpreter)

// Auto-profiling interpreter

// - Starts in min-profiling mode

// - Switches to profiling mode for loops based on iteration count

// - Switches back to min-profiling mode upon leaving a loop

EXECUTION\_MODE(AutoProfilingInterpreter)

// Profiling interpreter (does full profiling)

EXECUTION\_MODE(ProfilingInterpreter)

// Simple JIT

// - Behavior is determined based on the NewSimpleJit flag

// - Off: Behave as old simple JIT (does full profiling)

// - On: Behave as new simple JIT (does not profile, includes fast paths)

EXECUTION\_MODE(SimpleJit)

// Full JIT (no profiling, self-explanatory)

EXECUTION\_MODE(FullJit)

EXECUTION\_MODE(Count)

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if ENABLE\_NATIVE\_CODEGEN

namespace Js

{

ObjTypeSpecFldInfo\* ObjTypeSpecFldInfo::CreateFrom(uint id, InlineCache\* cache, uint cacheId, EntryPointInfo \*entryPoint,

FunctionBody\* const topFunctionBody, FunctionBody \*const functionBody, FieldAccessStatsPtr inlineCacheStats)

{

if (cache->IsEmpty())

{

return nullptr;

}

InlineCache localCache(\*cache);

// Need to keep a reference to the types before memory allocation in case they are tagged

Type \* type = nullptr;

Type \* typeWithoutProperty = nullptr;

Js::Type \* propertyOwnerType;

bool isLocal = localCache.IsLocal();

bool isProto = localCache.IsProto();

bool isAccessor = localCache.IsAccessor();

bool isGetter = localCache.IsGetterAccessor();

if (isLocal)

{

type = TypeWithoutAuxSlotTag(localCache.u.local.type);

if (localCache.u.local.typeWithoutProperty)

{

typeWithoutProperty = TypeWithoutAuxSlotTag(localCache.u.local.typeWithoutProperty);

}

propertyOwnerType = type;

}

else if (isProto)

{

type = TypeWithoutAuxSlotTag(localCache.u.proto.type);

propertyOwnerType = localCache.u.proto.prototypeObject->GetType();

}

else

{

if (PHASE\_OFF(Js::FixAccessorPropsPhase, functionBody))

{

return nullptr;

}

type = TypeWithoutAuxSlotTag(localCache.u.accessor.type);

propertyOwnerType = localCache.u.accessor.object->GetType();

}

ScriptContext\* scriptContext = functionBody->GetScriptContext();

Recycler \*const recycler = scriptContext->GetRecycler();

Js::PropertyId propertyId = functionBody->GetPropertyIdFromCacheId(cacheId);

uint16 slotIndex = Constants::NoSlot;

bool usesAuxSlot = false;

DynamicObject\* prototypeObject = nullptr;

PropertyGuard\* propertyGuard = nullptr;

JitTimeConstructorCache\* ctorCache = nullptr;

Var fieldValue = nullptr;

bool keepFieldValue = false;

bool isFieldValueFixed = false;

bool isMissing = false;

bool isBuiltIn = false;

// Save untagged type pointers, remembering whether the original type was tagged.

// The type pointers must be untagged so that the types cannot be collected during JIT.

if (isLocal)

{

slotIndex = localCache.u.local.slotIndex;

if (type != localCache.u.local.type)

{

usesAuxSlot = true;

}

if (typeWithoutProperty)

{

Assert(entryPoint->GetJitTransferData() != nullptr);

entryPoint->GetJitTransferData()->AddJitTimeTypeRef(typeWithoutProperty, recycler);

// These shared property guards are registered on the main thread and checked during entry point installation

// (see NativeCodeGenerator::CheckCodeGenDone) to ensure that no property became read-only while we were

// JIT-ing on the background thread.

propertyGuard = entryPoint->RegisterSharedPropertyGuard(propertyId, scriptContext);

}

}

else if (isProto)

{

prototypeObject = localCache.u.proto.prototypeObject;

slotIndex = localCache.u.proto.slotIndex;

if (type != localCache.u.proto.type)

{

usesAuxSlot = true;

fieldValue = prototypeObject->GetAuxSlot(slotIndex);

}

else

{

fieldValue = prototypeObject->GetInlineSlot(slotIndex);

}

isMissing = localCache.u.proto.isMissing;

propertyGuard = entryPoint->RegisterSharedPropertyGuard(propertyId, scriptContext);

}

else

{

slotIndex = localCache.u.accessor.slotIndex;

if (type != localCache.u.accessor.type)

{

usesAuxSlot = true;

fieldValue = localCache.u.accessor.object->GetAuxSlot(slotIndex);

}

else

{

fieldValue = localCache.u.accessor.object->GetInlineSlot(slotIndex);

}

}

// Keep the type alive until the entry point is installed. Note that this is longer than just during JIT, for which references

// from the JitTimeData would have been enough.

Assert(entryPoint->GetJitTransferData() != nullptr);

entryPoint->GetJitTransferData()->AddJitTimeTypeRef(type, recycler);

bool allFixedPhaseOFF = PHASE\_OFF(Js::FixedMethodsPhase, topFunctionBody) & PHASE\_OFF(Js::UseFixedDataPropsPhase, topFunctionBody);

if (!allFixedPhaseOFF)

{

Assert(propertyOwnerType != nullptr);

if (Js::DynamicType::Is(propertyOwnerType->GetTypeId()))

{

Js::DynamicTypeHandler\* propertyOwnerTypeHandler = ((Js::DynamicType\*)propertyOwnerType)->GetTypeHandler();

Js::PropertyId propertyId = functionBody->GetPropertyIdFromCacheId(cacheId);

Js::PropertyRecord const \* const fixedPropertyRecord = functionBody->GetScriptContext()->GetPropertyName(propertyId);

Var fixedProperty = nullptr;

Js::JavascriptFunction\* functionObject = nullptr;

if (isLocal || isProto)

{

if (typeWithoutProperty == nullptr)

{

// Since we don't know if this cache is used for LdMethodFld, we may fix up (use as fixed) too

// aggressively here, if we load a function that we don't actually call. This happens in the case

// of constructors (e.g. Object.defineProperty or Object.create).

// TODO (InlineCacheCleanup): if we don't zero some slot(s) in the inline cache, we could store a bit there

// indicating if this cache is used by LdMethodFld, and only ask then. We could even store a bit in the cache

// indicating that the value loaded is a fixed function (or at least may still be). That last bit could work

// even if we zero inline caches, since we would always set it when populating the cache.

propertyOwnerTypeHandler->TryUseFixedProperty(fixedPropertyRecord, &fixedProperty, (Js::FixedPropertyKind)(Js::FixedPropertyKind::FixedMethodProperty | Js::FixedPropertyKind::FixedDataProperty), scriptContext);

}

}

else

{

propertyOwnerTypeHandler->TryUseFixedAccessor(fixedPropertyRecord, &fixedProperty, (Js::FixedPropertyKind)(Js::FixedPropertyKind::FixedAccessorProperty), isGetter, scriptContext);

}

if (fixedProperty != nullptr && propertyGuard == nullptr)

{

propertyGuard = entryPoint->RegisterSharedPropertyGuard(propertyId, scriptContext);

}

if (fixedProperty != nullptr && Js::JavascriptFunction::Is(fixedProperty))

{

functionObject = (Js::JavascriptFunction \*)fixedProperty;

if (PHASE\_VERBOSE\_TRACE(Js::FixedMethodsPhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

wchar\_t debugStringBuffer2[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Js::DynamicObject\* protoObject = isProto ? prototypeObject : nullptr;

Output::Print(L"FixedFields: function %s (%s) cloning cache with fixed method: %s (%s), function: 0x%p, body: 0x%p (cache id: %d, layout: %s, type: 0x%p, proto: 0x%p, proto type: 0x%p)\n",

functionBody->GetDisplayName(), functionBody->GetDebugNumberSet(debugStringBuffer),

fixedPropertyRecord->GetBuffer(), functionObject->GetFunctionInfo()->GetFunctionProxy() ?

functionObject->GetFunctionInfo()->GetFunctionProxy()->GetDebugNumberSet(debugStringBuffer2) : L"(null)", functionObject, functionObject->GetFunctionInfo(),

cacheId, isProto ? L"proto" : L"local", type, protoObject, protoObject != nullptr ? protoObject->GetType() : nullptr);

Output::Flush();

}

if (PHASE\_VERBOSE\_TESTTRACE(Js::FixedMethodsPhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

wchar\_t debugStringBuffer2[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"FixedFields: function %s (%s) cloning cache with fixed method: %s (%s) (cache id: %d, layout: %s)\n",

functionBody->GetDisplayName(), functionBody->GetDebugNumberSet(debugStringBuffer), fixedPropertyRecord->GetBuffer(), functionObject->GetFunctionInfo()->GetFunctionProxy() ?

functionObject->GetFunctionInfo()->GetFunctionProxy()->GetDebugNumberSet(debugStringBuffer2) : L"(null)", functionObject, functionObject->GetFunctionInfo(),

cacheId, isProto ? L"proto" : L"local");

Output::Flush();

}

// We don't need to check for a singleton here. We checked that the singleton still existed

// when we obtained the fixed field value inside TryUseFixedProperty. Since we don't need the

// singleton instance later in this function, we don't care if the instance got collected.

// We get the type handler from the propertyOwnerType, which we got from the cache. At runtime,

// if the object is collected, it is by definition unreachable and thus the code in the function

// we're about to emit cannot reach the object to try to access any of this object's properties.

ConstructorCache\* runtimeConstructorCache = functionObject->GetConstructorCache();

if (runtimeConstructorCache->IsSetUpForJit() && runtimeConstructorCache->GetScriptContext() == scriptContext)

{

FunctionInfo\* functionInfo = functionObject->GetFunctionInfo();

Assert(functionInfo != nullptr);

Assert((functionInfo->GetAttributes() & FunctionInfo::ErrorOnNew) == 0);

Assert(!runtimeConstructorCache->IsDefault());

if (runtimeConstructorCache->IsNormal())

{

Assert(runtimeConstructorCache->GetType()->GetIsShared());

// If we populated the cache with initial type before calling the constructor, but then didn't end up updating the cache

// after the constructor and shrinking (and locking) the inline slot capacity, we must lock it now. In that case it is

// also possible that the inline slot capacity was shrunk since we originally cached it, so we must update it also.

#if DBG\_DUMP

if (!runtimeConstructorCache->GetType()->GetTypeHandler()->GetIsInlineSlotCapacityLocked())

{

if (PHASE\_TRACE(Js::FixedNewObjPhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

wchar\_t debugStringBuffer2[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"FixedNewObj: function %s (%s) ctor cache for %s (%s) about to be cloned has unlocked inline slot count: guard value = 0x%p, type = 0x%p, slots = %d, inline slots = %d\n",

functionBody->GetDisplayName(), functionBody->GetDebugNumberSet(debugStringBuffer), fixedPropertyRecord->GetBuffer(), functionObject->GetFunctionInfo()->GetFunctionBody() ?

functionObject->GetFunctionInfo()->GetFunctionBody()->GetDebugNumberSet(debugStringBuffer2) : L"(null)",

runtimeConstructorCache->GetRawGuardValue(), runtimeConstructorCache->GetType(),

runtimeConstructorCache->GetSlotCount(), runtimeConstructorCache->GetInlineSlotCount());

Output::Flush();

}

}

#endif

runtimeConstructorCache->GetType()->GetTypeHandler()->EnsureInlineSlotCapacityIsLocked();

runtimeConstructorCache->UpdateInlineSlotCount();

}

// We must keep the runtime cache alive as long as this entry point exists and may try to dereference it.

entryPoint->RegisterConstructorCache(runtimeConstructorCache, recycler);

ctorCache = RecyclerNew(recycler, JitTimeConstructorCache, functionObject, runtimeConstructorCache);

if (PHASE\_TRACE(Js::FixedNewObjPhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

wchar\_t debugStringBuffer2[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"FixedNewObj: function %s (%s) cloning ctor cache for %s (%s): guard value = 0x%p, type = 0x%p, slots = %d, inline slots = %d\n",

functionBody->GetDisplayName(), functionBody->GetDebugNumberSet(debugStringBuffer), fixedPropertyRecord->GetBuffer(), functionObject->GetFunctionInfo()->GetFunctionBody() ?

functionObject->GetFunctionInfo()->GetFunctionBody()->GetDebugNumberSet(debugStringBuffer2) : L"(null)", functionObject, functionObject->GetFunctionInfo(),

runtimeConstructorCache->GetRawGuardValue(), runtimeConstructorCache->IsNormal() ? runtimeConstructorCache->GetType() : nullptr,

runtimeConstructorCache->GetSlotCount(), runtimeConstructorCache->GetInlineSlotCount());

Output::Flush();

}

}

else

{

if (!runtimeConstructorCache->IsDefault())

{

if (PHASE\_TRACE(Js::FixedNewObjPhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

wchar\_t debugStringBuffer2[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"FixedNewObj: function %s (%s) skipping ctor cache for %s (%s), because %s (guard value = 0x%p, script context = %p).\n",

functionBody->GetDisplayName(), functionBody->GetDebugNumberSet(debugStringBuffer), fixedPropertyRecord->GetBuffer(), functionObject->GetFunctionInfo()->GetFunctionBody() ?

functionObject->GetFunctionInfo()->GetFunctionBody()->GetDebugNumberSet(debugStringBuffer2) : L"(null)", functionObject, functionObject->GetFunctionInfo(),

runtimeConstructorCache->IsEmpty() ? L"cache is empty (or has been cleared)" :

runtimeConstructorCache->IsInvalidated() ? L"cache is invalidated" :

runtimeConstructorCache->SkipDefaultNewObject() ? L"default new object isn't needed" :

runtimeConstructorCache->NeedsTypeUpdate() ? L"cache needs to be updated" :

runtimeConstructorCache->NeedsUpdateAfterCtor() ? L"cache needs update after ctor" :

runtimeConstructorCache->IsPolymorphic() ? L"cache is polymorphic" :

runtimeConstructorCache->GetScriptContext() != functionBody->GetScriptContext() ? L"script context mismatch" : L"of an unexpected situation",

runtimeConstructorCache->GetRawGuardValue(), runtimeConstructorCache->GetScriptContext());

Output::Flush();

}

}

}

isBuiltIn = Js::JavascriptLibrary::GetBuiltinFunctionForPropId(propertyId) != Js::BuiltinFunction::None;

}

if (fixedProperty != nullptr)

{

Assert(fieldValue == nullptr || fieldValue == fixedProperty);

fieldValue = fixedProperty;

isFieldValueFixed = true;

keepFieldValue = true;

}

}

}

FixedFieldInfo\* fixedFieldInfoArray = RecyclerNewArrayZ(recycler, FixedFieldInfo, 1);

fixedFieldInfoArray[0].fieldValue = fieldValue;

fixedFieldInfoArray[0].type = type;

fixedFieldInfoArray[0].nextHasSameFixedField = false;

ObjTypeSpecFldInfo\* info;

// If we stress equivalent object type spec, let's pretend that every inline cache was polymorphic and equivalent.

bool forcePoly = false;

if ((!PHASE\_OFF(Js::EquivObjTypeSpecByDefaultPhase, topFunctionBody) ||

PHASE\_STRESS(Js::EquivObjTypeSpecPhase, topFunctionBody))

&& !PHASE\_OFF(Js::EquivObjTypeSpecPhase, topFunctionBody))

{

Assert(topFunctionBody->HasDynamicProfileInfo());

auto profileData = topFunctionBody->GetAnyDynamicProfileInfo();

// We only support equivalent fixed fields on loads from prototype, and no equivalence on missing properties

forcePoly |= !profileData->IsEquivalentObjTypeSpecDisabled() && (!isFieldValueFixed || isProto) && !isMissing;

}

if (isFieldValueFixed)

{

// Fixed field checks allow us to assume a specific type ID, but the assumption is only

// valid if we lock the type. Otherwise, the type ID may change out from under us without

// evolving the type.

if (DynamicType::Is(type->GetTypeId()))

{

DynamicType \*dynamicType = static\_cast<DynamicType\*>(type);

if (!dynamicType->GetIsLocked())

{

dynamicType->LockType();

}

}

}

if (forcePoly)

{

uint16 typeCount = 1;

Js::Type\*\* types = RecyclerNewArray(recycler, Js::Type\*, typeCount);

types[0] = type;

EquivalentTypeSet\* typeSet = RecyclerNew(recycler, EquivalentTypeSet, types, typeCount);

info = RecyclerNew(recycler, ObjTypeSpecFldInfo,

id, type->GetTypeId(), typeWithoutProperty, typeSet, usesAuxSlot, isProto, isAccessor, isFieldValueFixed, keepFieldValue, false/\*doesntHaveEquivalence\*/, false, slotIndex, propertyId,

prototypeObject, propertyGuard, ctorCache, fixedFieldInfoArray, 1);

if (PHASE\_TRACE(Js::ObjTypeSpecPhase, topFunctionBody) || PHASE\_TRACE(Js::EquivObjTypeSpecPhase, topFunctionBody))

{

const PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(propertyId);

Output::Print(L"Created ObjTypeSpecFldInfo: id %u, property %s(#%u), slot %u, type set: 0x%p\n",

id, propertyRecord->GetBuffer(), propertyId, slotIndex, type);

Output::Flush();

}

}

else

{

info = RecyclerNew(recycler, ObjTypeSpecFldInfo,

id, type->GetTypeId(), typeWithoutProperty, usesAuxSlot, isProto, isAccessor, isFieldValueFixed, keepFieldValue, isBuiltIn, slotIndex, propertyId,

prototypeObject, propertyGuard, ctorCache, fixedFieldInfoArray);

if (PHASE\_TRACE(Js::ObjTypeSpecPhase, topFunctionBody) || PHASE\_TRACE(Js::EquivObjTypeSpecPhase, topFunctionBody))

{

const PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(propertyId);

Output::Print(L"Created ObjTypeSpecFldInfo: id %u, property %s(#%u), slot %u, type: 0x%p\n",

id, propertyRecord->GetBuffer(), propertyId, slotIndex, type);

Output::Flush();

}

}

return info;

}

ObjTypeSpecFldInfo\* ObjTypeSpecFldInfo::CreateFrom(uint id, PolymorphicInlineCache\* cache, uint cacheId, EntryPointInfo \*entryPoint,

FunctionBody\* const topFunctionBody, FunctionBody \*const functionBody, FieldAccessStatsPtr inlineCacheStats)

{

#ifdef FIELD\_ACCESS\_STATS

#define IncInlineCacheCount(counter) if (inlineCacheStats) { inlineCacheStats->counter++; }

#else

#define IncInlineCacheCount(counter)

#endif

Assert(topFunctionBody->HasDynamicProfileInfo());

auto profileData = topFunctionBody->GetAnyDynamicProfileInfo();

bool gatherDataForInlining = cache->GetCloneForJitTimeUse() && functionBody->PolyInliningUsingFixedMethodsAllowedByConfigFlags(topFunctionBody);

if (PHASE\_OFF(Js::EquivObjTypeSpecPhase, topFunctionBody) || profileData->IsEquivalentObjTypeSpecDisabled())

{

if (!gatherDataForInlining)

{

return nullptr;

}

}

Assert(cache->GetSize() < MAXUINT16);

Js::InlineCache\* inlineCaches = cache->GetInlineCaches();

Js::DynamicObject\* prototypeObject = nullptr;

Js::DynamicObject\* accessorOwnerObject = nullptr;

Js::TypeId typeId = TypeIds\_Limit;

uint16 polyCacheSize = (uint16)cache->GetSize();

uint16 firstNonEmptyCacheIndex = MAXUINT16;

uint16 slotIndex = 0;

bool areEquivalent = true;

bool usesAuxSlot = false;

bool isProto = false;

bool isAccessor = false;

bool isGetterAccessor = false;

bool isAccessorOnProto = false;

bool stress = PHASE\_STRESS(Js::EquivObjTypeSpecPhase, topFunctionBody);

bool areStressEquivalent = stress;

uint16 typeCount = 0;

for (uint16 i = 0; (areEquivalent || stress || gatherDataForInlining) && i < polyCacheSize; i++)

{

InlineCache& inlineCache = inlineCaches[i];

if (inlineCache.IsEmpty()) continue;

if (firstNonEmptyCacheIndex == MAXUINT16)

{

if (inlineCache.IsLocal())

{

typeId = TypeWithoutAuxSlotTag(inlineCache.u.local.type)->GetTypeId();

usesAuxSlot = TypeHasAuxSlotTag(inlineCache.u.local.type);

slotIndex = inlineCache.u.local.slotIndex;

// We don't support equivalent object type spec for adding properties.

areEquivalent = inlineCache.u.local.typeWithoutProperty == nullptr;

gatherDataForInlining = false;

}

// Missing properties cannot be treated as equivalent, because for objects with SDTH or DTH, we don't change the object's type

// when we add a property. We also don't invalidate proto inline caches (and guards) unless the property being added exists on the proto chain.

// Missing properties by definition do not exist on the proto chain, so in the end we could have an EquivalentObjTypeSpec cache hit on a

// property that once was missing, but has since been added. (See OS Bugs 280582).

else if (inlineCache.IsProto() && !inlineCache.u.proto.isMissing)

{

isProto = true;

typeId = TypeWithoutAuxSlotTag(inlineCache.u.proto.type)->GetTypeId();

usesAuxSlot = TypeHasAuxSlotTag(inlineCache.u.proto.type);

slotIndex = inlineCache.u.proto.slotIndex;

prototypeObject = inlineCache.u.proto.prototypeObject;

}

else

{

if (!PHASE\_OFF(Js::FixAccessorPropsPhase, functionBody))

{

isAccessor = true;

isGetterAccessor = inlineCache.IsGetterAccessor();

isAccessorOnProto = inlineCache.u.accessor.isOnProto;

accessorOwnerObject = inlineCache.u.accessor.object;

typeId = TypeWithoutAuxSlotTag(inlineCache.u.accessor.type)->GetTypeId();

usesAuxSlot = TypeHasAuxSlotTag(inlineCache.u.accessor.type);

slotIndex = inlineCache.u.accessor.slotIndex;

}

else

{

areEquivalent = false;

areStressEquivalent = false;

}

gatherDataForInlining = false;

}

// If we're stressing equivalent object type spec then let's keep trying to find a cache that we could use.

if (!stress || areStressEquivalent)

{

firstNonEmptyCacheIndex = i;

}

}

else

{

if (inlineCache.IsLocal())

{

if (isProto || isAccessor || inlineCache.u.local.typeWithoutProperty != nullptr || slotIndex != inlineCache.u.local.slotIndex ||

typeId != TypeWithoutAuxSlotTag(inlineCache.u.local.type)->GetTypeId() || usesAuxSlot != TypeHasAuxSlotTag(inlineCache.u.local.type))

{

areEquivalent = false;

}

gatherDataForInlining = false;

}

else if (inlineCache.IsProto())

{

if (!isProto || isAccessor || prototypeObject != inlineCache.u.proto.prototypeObject || slotIndex != inlineCache.u.proto.slotIndex ||

typeId != TypeWithoutAuxSlotTag(inlineCache.u.proto.type)->GetTypeId() || usesAuxSlot != TypeHasAuxSlotTag(inlineCache.u.proto.type))

{

areEquivalent = false;

}

}

else

{

// Supporting equivalent obj type spec only for those polymorphic accessor property operations for which

// 1. the property is on the same prototype, and

// 2. the types are equivalent.

//

// This is done to keep the equivalence check helper as-is

if (!isAccessor || isGetterAccessor != inlineCache.IsGetterAccessor() || !isAccessorOnProto || !inlineCache.u.accessor.isOnProto || accessorOwnerObject != inlineCache.u.accessor.object ||

slotIndex != inlineCache.u.accessor.slotIndex || typeId != TypeWithoutAuxSlotTag(inlineCache.u.accessor.type)->GetTypeId() || usesAuxSlot != TypeHasAuxSlotTag(inlineCache.u.accessor.type))

{

areEquivalent = false;

}

gatherDataForInlining = false;

}

}

typeCount++;

}

if (firstNonEmptyCacheIndex == MAXUINT16)

{

IncInlineCacheCount(emptyPolyInlineCacheCount);

return nullptr;

}

if (cache->GetIgnoreForEquivalentObjTypeSpec())

{

areEquivalent = areStressEquivalent = false;

}

gatherDataForInlining = gatherDataForInlining && (typeCount <= 4); // Only support 4-way (max) polymorphic inlining

if (!areEquivalent && !areStressEquivalent)

{

IncInlineCacheCount(nonEquivPolyInlineCacheCount);

cache->SetIgnoreForEquivalentObjTypeSpec(true);

if (!gatherDataForInlining)

{

return nullptr;

}

}

Assert(firstNonEmptyCacheIndex < polyCacheSize);

Assert(typeId != TypeIds\_Limit);

IncInlineCacheCount(equivPolyInlineCacheCount);

// If we're stressing equivalent object type spec and the types are not equivalent, let's grab the first one only.

if (stress && (areEquivalent != areStressEquivalent))

{

polyCacheSize = firstNonEmptyCacheIndex + 1;

}

ScriptContext\* scriptContext = functionBody->GetScriptContext();

Recycler\* recycler = scriptContext->GetRecycler();

uint16 fixedFunctionCount = 0;

// Need to create a local array here and not allocate one from the recycler,

// as the allocation may trigger a GC which can clear the inline caches.

FixedFieldInfo localFixedFieldInfoArray[Js::DynamicProfileInfo::maxPolymorphicInliningSize] = { 0 };

// For polymorphic field loads we only support fixed functions on prototypes. This helps keep the equivalence check helper simple.

// Since all types in the polymorphic cache share the same prototype, it's enough to grab the fixed function from the prototype object.

Var fixedProperty = nullptr;

if ((isProto || isAccessorOnProto) && (areEquivalent || areStressEquivalent))

{

const Js::PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(functionBody->GetPropertyIdFromCacheId(cacheId));

if (isProto)

{

prototypeObject->GetDynamicType()->GetTypeHandler()->TryUseFixedProperty(propertyRecord, &fixedProperty, (Js::FixedPropertyKind)(Js::FixedPropertyKind::FixedMethodProperty | Js::FixedPropertyKind::FixedDataProperty), scriptContext);

}

else if (isAccessorOnProto)

{

accessorOwnerObject->GetDynamicType()->GetTypeHandler()->TryUseFixedAccessor(propertyRecord, &fixedProperty, Js::FixedPropertyKind::FixedAccessorProperty, isGetterAccessor, scriptContext);

}

localFixedFieldInfoArray[0].fieldValue = fixedProperty;

localFixedFieldInfoArray[0].type = nullptr;

localFixedFieldInfoArray[0].nextHasSameFixedField = false;

// TODO (ObjTypeSpec): Enable constructor caches on equivalent polymorphic field loads with fixed functions.

}

// Let's get the types.

Js::Type\* localTypes[MaxPolymorphicInlineCacheSize];

uint16 typeNumber = 0;

Js::JavascriptFunction\* fixedFunctionObject = nullptr;

for (uint16 i = firstNonEmptyCacheIndex; i < polyCacheSize; i++)

{

InlineCache& inlineCache = inlineCaches[i];

if (inlineCache.IsEmpty()) continue;

localTypes[typeNumber] = inlineCache.IsLocal() ? TypeWithoutAuxSlotTag(inlineCache.u.local.type) :

inlineCache.IsProto() ? TypeWithoutAuxSlotTag(inlineCache.u.proto.type) :

TypeWithoutAuxSlotTag(inlineCache.u.accessor.type);

if (gatherDataForInlining)

{

inlineCache.TryGetFixedMethodFromCache(functionBody, cacheId, &fixedFunctionObject);

if (!fixedFunctionObject || !fixedFunctionObject->GetFunctionInfo()->HasBody())

{

if (!(areEquivalent || areStressEquivalent))

{

// If we reach here only because we are gathering data for inlining, and one of the Inline Caches doesn't have a fixedfunction object, return.

return nullptr;

}

else

{

// If one of the inline caches doesn't have a fixed function object, abort gathering inlining data.

gatherDataForInlining = false;

typeNumber++;

continue;

}

}

// We got a fixed function object from the cache.

localFixedFieldInfoArray[typeNumber].type = localTypes[typeNumber];

localFixedFieldInfoArray[typeNumber].fieldValue = fixedFunctionObject;

localFixedFieldInfoArray[typeNumber].nextHasSameFixedField = false;

fixedFunctionCount++;

}

typeNumber++;

}

if (isAccessor && gatherDataForInlining)

{

Assert(fixedFunctionCount <= 1);

}

if (stress && (areEquivalent != areStressEquivalent))

{

typeCount = 1;

}

AnalysisAssert(typeNumber == typeCount);

// Now that we've copied all material info into local variables, we can start allocating without fear

// that a garbage collection will clear any of the live inline caches.

FixedFieldInfo\* fixedFieldInfoArray;

if (gatherDataForInlining)

{

fixedFieldInfoArray = RecyclerNewArrayZ(recycler, FixedFieldInfo, fixedFunctionCount);

memcpy(fixedFieldInfoArray, localFixedFieldInfoArray, fixedFunctionCount \* sizeof(FixedFieldInfo));

}

else

{

fixedFieldInfoArray = RecyclerNewArrayZ(recycler, FixedFieldInfo, 1);

memcpy(fixedFieldInfoArray, localFixedFieldInfoArray, 1 \* sizeof(FixedFieldInfo));

}

Js::PropertyId propertyId = functionBody->GetPropertyIdFromCacheId(cacheId);

Js::PropertyGuard\* propertyGuard = entryPoint->RegisterSharedPropertyGuard(propertyId, scriptContext);

// For polymorphic, non-equivalent objTypeSpecFldInfo's, hasFixedValue is true only if each of the inline caches has a fixed function for the given cacheId, or

// in the case of an accessor cache, only if the there is only one version of the accessor.

bool hasFixedValue = gatherDataForInlining ||

((isProto || isAccessorOnProto) && (areEquivalent || areStressEquivalent) && localFixedFieldInfoArray[0].fieldValue);

bool doesntHaveEquivalence = !(areEquivalent || areStressEquivalent);

EquivalentTypeSet\* typeSet = nullptr;

auto jitTransferData = entryPoint->GetJitTransferData();

Assert(jitTransferData != nullptr);

if (areEquivalent || areStressEquivalent)

{

for (uint16 i = 0; i < typeCount; i++)

{

jitTransferData->AddJitTimeTypeRef(localTypes[i], recycler);

if (hasFixedValue)

{

// Fixed field checks allow us to assume a specific type ID, but the assumption is only

// valid if we lock the type. Otherwise, the type ID may change out from under us without

// evolving the type.

if (DynamicType::Is(localTypes[i]->GetTypeId()))

{

DynamicType \*dynamicType = static\_cast<DynamicType\*>(localTypes[i]);

if (!dynamicType->GetIsLocked())

{

dynamicType->LockType();

}

}

}

}

Js::Type\*\* types = RecyclerNewArray(recycler, Js::Type\*, typeCount);

memcpy(types, localTypes, typeCount \* sizeof(Js::Type\*));

typeSet = RecyclerNew(recycler, EquivalentTypeSet, types, typeCount);

}

ObjTypeSpecFldInfo\* info = RecyclerNew(recycler, ObjTypeSpecFldInfo,

id, typeId, nullptr, typeSet, usesAuxSlot, isProto, isAccessor, hasFixedValue, hasFixedValue, doesntHaveEquivalence, true, slotIndex, propertyId,

prototypeObject, propertyGuard, nullptr, fixedFieldInfoArray, fixedFunctionCount/\*, nullptr, nullptr, nullptr\*/);

if (PHASE\_TRACE(Js::ObjTypeSpecPhase, topFunctionBody) || PHASE\_TRACE(Js::EquivObjTypeSpecPhase, topFunctionBody))

{

if (PHASE\_TRACE(Js::ObjTypeSpecPhase, topFunctionBody) || PHASE\_TRACE(Js::EquivObjTypeSpecPhase, topFunctionBody))

{

if (typeSet)

{

const PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(propertyId);

Output::Print(L"Created ObjTypeSpecFldInfo: id %u, property %s(#%u), slot %u, type set: ",

id, propertyRecord->GetBuffer(), propertyId, slotIndex);

for (uint16 ti = 0; ti < typeCount - 1; ti++)

{

Output::Print(L"0x%p, ", typeSet->GetType(ti));

}

Output::Print(L"0x%p\n", typeSet->GetType(typeCount - 1));

Output::Flush();

}

}

}

return info;

#undef IncInlineCacheCount

}

Js::JavascriptFunction\* ObjTypeSpecFldInfo::GetFieldValueAsFixedFunction() const

{

Assert(HasFixedValue());

Assert(IsMono() || (IsPoly() && !DoesntHaveEquivalence()));

Assert(this->fixedFieldInfoArray[0].fieldValue != nullptr && Js::JavascriptFunction::Is(this->fixedFieldInfoArray[0].fieldValue));

return Js::JavascriptFunction::FromVar(this->fixedFieldInfoArray[0].fieldValue);

}

Js::JavascriptFunction\* ObjTypeSpecFldInfo::GetFieldValueAsFixedFunction(uint i) const

{

Assert(HasFixedValue());

Assert(IsPoly());

Assert(this->fixedFieldInfoArray[i].fieldValue != nullptr && Js::JavascriptFunction::Is(this->fixedFieldInfoArray[i].fieldValue));

return Js::JavascriptFunction::FromVar(this->fixedFieldInfoArray[i].fieldValue);

}

Js::JavascriptFunction\* ObjTypeSpecFldInfo::GetFieldValueAsFunction() const

{

Assert(IsMono() || (IsPoly() && !DoesntHaveEquivalence()));

Assert(this->fixedFieldInfoArray[0].fieldValue != nullptr && JavascriptFunction::Is(this->fixedFieldInfoArray[0].fieldValue));

return JavascriptFunction::FromVar(this->fixedFieldInfoArray[0].fieldValue);

}

Js::JavascriptFunction\* ObjTypeSpecFldInfo::GetFieldValueAsFunctionIfAvailable() const

{

Assert(IsMono() || (IsPoly() && !DoesntHaveEquivalence()));

return this->fixedFieldInfoArray[0].fieldValue != nullptr && JavascriptFunction::Is(this->fixedFieldInfoArray[0].fieldValue) ?

JavascriptFunction::FromVar(this->fixedFieldInfoArray[0].fieldValue) : nullptr;

}

Js::JavascriptFunction\* ObjTypeSpecFldInfo::GetFieldValueAsFixedFunctionIfAvailable() const

{

Assert(HasFixedValue());

Assert(IsMono() || (IsPoly() && !DoesntHaveEquivalence()));

return GetFieldValueAsFunctionIfAvailable();

}

Js::JavascriptFunction\* ObjTypeSpecFldInfo::GetFieldValueAsFixedFunctionIfAvailable(uint i) const

{

Assert(HasFixedValue());

Assert(IsPoly());

return this->fixedFieldInfoArray[i].fieldValue != nullptr && JavascriptFunction::Is(this->fixedFieldInfoArray[i].fieldValue) ?

JavascriptFunction::FromVar(this->fixedFieldInfoArray[i].fieldValue) : nullptr;

}

Js::Var ObjTypeSpecFldInfo::GetFieldValueAsFixedDataIfAvailable() const

{

Assert(HasFixedValue() && this->fixedFieldCount == 1);

return this->fixedFieldInfoArray[0].fieldValue;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

const wchar\_t\* ObjTypeSpecFldInfo::GetCacheLayoutString() const

{

return IsLoadedFromProto() ? L"proto" : UsesAccessor() ? L"flags" : L"local";

}

#endif

ObjTypeSpecFldInfoArray::ObjTypeSpecFldInfoArray()

: infoArray(nullptr)

#if DBG

, infoCount(0)

#endif

{

}

void ObjTypeSpecFldInfoArray::EnsureArray(Recycler \*const recycler, FunctionBody \*const functionBody)

{

Assert(recycler != nullptr);

Assert(functionBody != nullptr);

Assert(functionBody->GetInlineCacheCount() != 0);

if (this->infoArray)

{

Assert(functionBody->GetInlineCacheCount() == this->infoCount);

return;

}

this->infoArray = RecyclerNewArrayZ(recycler, ObjTypeSpecFldInfo\*, functionBody->GetInlineCacheCount());

#if DBG

this->infoCount = functionBody->GetInlineCacheCount();

#endif

}

ObjTypeSpecFldInfo\* ObjTypeSpecFldInfoArray::GetInfo(FunctionBody \*const functionBody, const uint index) const

{

Assert(functionBody);

Assert(this->infoArray == nullptr || functionBody->GetInlineCacheCount() == this->infoCount);

Assert(index < functionBody->GetInlineCacheCount());

return this->infoArray ? this->infoArray[index] : nullptr;

}

void ObjTypeSpecFldInfoArray::SetInfo(Recycler \*const recycler, FunctionBody \*const functionBody,

const uint index, ObjTypeSpecFldInfo\* info)

{

Assert(recycler);

Assert(functionBody);

Assert(this->infoArray == nullptr || functionBody->GetInlineCacheCount() == this->infoCount);

Assert(index < functionBody->GetInlineCacheCount());

Assert(info);

EnsureArray(recycler, functionBody);

this->infoArray[index] = info;

}

void ObjTypeSpecFldInfoArray::Reset()

{

this->infoArray = nullptr;

#if DBG

this->infoCount = 0;

#endif

}

FunctionCodeGenJitTimeData::FunctionCodeGenJitTimeData(FunctionInfo \*const functionInfo, EntryPointInfo \*const entryPoint, bool isInlined) :

functionInfo(functionInfo), entryPointInfo(entryPoint), globalObjTypeSpecFldInfoCount(0), globalObjTypeSpecFldInfoArray(nullptr),

weakFuncRef(nullptr), inlinees(nullptr), inlineeCount(0), ldFldInlineeCount(0), isInlined(isInlined), isAggressiveInliningEnabled(false),

#ifdef FIELD\_ACCESS\_STATS

inlineCacheStats(nullptr),

#endif

profiledIterations(GetFunctionBody() ? GetFunctionBody()->GetProfiledIterations() : 0),

next(0)

{

}

FunctionInfo \*FunctionCodeGenJitTimeData::GetFunctionInfo() const

{

return this->functionInfo;

}

FunctionBody \*FunctionCodeGenJitTimeData::GetFunctionBody() const

{

return this->functionInfo->GetFunctionBody();

}

bool FunctionCodeGenJitTimeData::IsPolymorphicCallSite(const ProfileId profiledCallSiteId) const

{

Assert(GetFunctionBody());

Assert(profiledCallSiteId < GetFunctionBody()->GetProfiledCallSiteCount());

return inlinees ? inlinees[profiledCallSiteId]->next != nullptr : false;

}

const FunctionCodeGenJitTimeData \*FunctionCodeGenJitTimeData::GetInlinee(const ProfileId profiledCallSiteId) const

{

Assert(GetFunctionBody());

Assert(profiledCallSiteId < GetFunctionBody()->GetProfiledCallSiteCount());

return inlinees ? inlinees[profiledCallSiteId] : nullptr;

}

const FunctionCodeGenJitTimeData \*FunctionCodeGenJitTimeData::GetJitTimeDataFromFunctionInfo(FunctionInfo \*polyFunctionInfo) const

{

const FunctionCodeGenJitTimeData \*next = this;

while (next && next->functionInfo != polyFunctionInfo)

{

next = next->next;

}

return next;

}

const FunctionCodeGenJitTimeData \*FunctionCodeGenJitTimeData::GetLdFldInlinee(const InlineCacheIndex inlineCacheIndex) const

{

Assert(GetFunctionBody());

Assert(inlineCacheIndex < GetFunctionBody()->GetInlineCacheCount());

return ldFldInlinees ? ldFldInlinees[inlineCacheIndex] : nullptr;

}

FunctionCodeGenJitTimeData \*FunctionCodeGenJitTimeData::AddInlinee(

Recycler \*const recycler,

const ProfileId profiledCallSiteId,

FunctionInfo \*const inlinee,

bool isInlined)

{

Assert(recycler);

const auto functionBody = GetFunctionBody();

Assert(functionBody);

Assert(profiledCallSiteId < functionBody->GetProfiledCallSiteCount());

Assert(inlinee);

if (!inlinees)

{

inlinees = RecyclerNewArrayZ(recycler, FunctionCodeGenJitTimeData \*, functionBody->GetProfiledCallSiteCount());

}

FunctionCodeGenJitTimeData \*inlineeData = nullptr;

if (!inlinees[profiledCallSiteId])

{

inlineeData = RecyclerNew(recycler, FunctionCodeGenJitTimeData, inlinee, nullptr /\* entryPoint \*/, isInlined);

inlinees[profiledCallSiteId] = inlineeData;

if (++inlineeCount == 0)

{

Js::Throw::OutOfMemory();

}

}

else

{

inlineeData = RecyclerNew(recycler, FunctionCodeGenJitTimeData, inlinee, nullptr /\* entryPoint \*/, isInlined);

// This is polymorphic, chain the data.

inlineeData->next = inlinees[profiledCallSiteId];

inlinees[profiledCallSiteId] = inlineeData;

}

return inlineeData;

}

FunctionCodeGenJitTimeData \*FunctionCodeGenJitTimeData::AddLdFldInlinee(

Recycler \*const recycler,

const InlineCacheIndex inlineCacheIndex,

FunctionInfo \*const inlinee)

{

Assert(recycler);

const auto functionBody = GetFunctionBody();

Assert(functionBody);

Assert(inlineCacheIndex < GetFunctionBody()->GetInlineCacheCount());

Assert(inlinee);

if (!ldFldInlinees)

{

ldFldInlinees = RecyclerNewArrayZ(recycler, FunctionCodeGenJitTimeData \*, GetFunctionBody()->GetInlineCacheCount());

}

const auto inlineeData = RecyclerNew(recycler, FunctionCodeGenJitTimeData, inlinee, nullptr);

Assert(!ldFldInlinees[inlineCacheIndex]);

ldFldInlinees[inlineCacheIndex] = inlineeData;

if (++ldFldInlineeCount == 0)

{

Js::Throw::OutOfMemory();

}

return inlineeData;

}

uint FunctionCodeGenJitTimeData::InlineeCount() const

{

return inlineeCount;

}

#ifdef FIELD\_ACCESS\_STATS

void FunctionCodeGenJitTimeData::EnsureInlineCacheStats(Recycler\* recycler)

{

this->inlineCacheStats = RecyclerNew(recycler, FieldAccessStats);

}

void FunctionCodeGenJitTimeData::AddInlineeInlineCacheStats(FunctionCodeGenJitTimeData\* inlineeJitTimeData)

{

Assert(this->inlineCacheStats != nullptr);

Assert(inlineeJitTimeData != nullptr && inlineeJitTimeData->inlineCacheStats != nullptr);

this->inlineCacheStats->Add(inlineeJitTimeData->inlineCacheStats);

}

#endif

uint16 FunctionCodeGenJitTimeData::GetProfiledIterations() const

{

return profiledIterations;

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#if ENABLE\_NATIVE\_CODEGEN

namespace Js

{

struct JitTimeConstructorCache

{

// TODO (FixedNewObj): Consider making these private and provide getters to ensure the values are not changed.

const JavascriptFunction\* constructor;

ConstructorCache\* runtimeCache;

const ScriptContext\* scriptContext;

const DynamicType\* type;

BVSparse<JitArenaAllocator>\* guardedPropOps;

int slotCount;

int16 inlineSlotCount;

bool skipNewScObject;

bool ctorHasNoExplicitReturnValue;

bool typeIsFinal;

bool isUsed;

public:

JitTimeConstructorCache(const JavascriptFunction\* constructor, ConstructorCache\* runtimeCache)

{

Assert(constructor != nullptr);

Assert(runtimeCache != nullptr);

this->constructor = constructor;

this->runtimeCache = runtimeCache;

this->type = runtimeCache->content.type;

this->guardedPropOps = nullptr;

this->scriptContext = runtimeCache->content.scriptContext;

this->slotCount = runtimeCache->content.slotCount;

this->inlineSlotCount = runtimeCache->content.inlineSlotCount;

this->skipNewScObject = runtimeCache->content.skipDefaultNewObject;

this->ctorHasNoExplicitReturnValue = runtimeCache->content.ctorHasNoExplicitReturnValue;

this->typeIsFinal = runtimeCache->content.typeIsFinal;

this->isUsed = false;

}

JitTimeConstructorCache(const JitTimeConstructorCache\* other)

{

Assert(other != nullptr);

Assert(other->constructor != nullptr);

Assert(other->runtimeCache != nullptr);

this->constructor = other->constructor;

this->runtimeCache = other->runtimeCache;

this->type = other->type;

this->guardedPropOps = nullptr;

this->scriptContext = other->scriptContext;

this->slotCount = other->slotCount;

this->inlineSlotCount = other->inlineSlotCount;

this->skipNewScObject = other->skipNewScObject;

this->ctorHasNoExplicitReturnValue = other->ctorHasNoExplicitReturnValue;

this->typeIsFinal = other->typeIsFinal;

this->isUsed = false;

}

JitTimeConstructorCache\* Clone(JitArenaAllocator\* allocator)

{

JitTimeConstructorCache\* clone = Anew(allocator, JitTimeConstructorCache, this);

return clone;

}

const BVSparse<JitArenaAllocator>\* GetGuardedPropOps() const

{

return this->guardedPropOps;

}

void EnsureGuardedPropOps(JitArenaAllocator\* allocator)

{

if (this->guardedPropOps == nullptr)

{

this->guardedPropOps = Anew(allocator, BVSparse<JitArenaAllocator>, allocator);

}

}

void SetGuardedPropOp(uint propOpId)

{

Assert(this->guardedPropOps != nullptr);

this->guardedPropOps->Set(propOpId);

}

void AddGuardedPropOps(const BVSparse<JitArenaAllocator>\* propOps)

{

Assert(this->guardedPropOps != nullptr);

this->guardedPropOps->Or(propOps);

}

};

#define InitialObjTypeSpecFldInfoFlagValue 0x01

struct FixedFieldInfo

{

Var fieldValue;

Type\* type;

bool nextHasSameFixedField; // set to true if the next entry in the FixedFieldInfo array on ObjTypeSpecFldInfo has the same type

};

class ObjTypeSpecFldInfo

{

private:

DynamicObject\* protoObject;

PropertyGuard\* propertyGuard;

EquivalentTypeSet\* typeSet;

Type\* initialType;

JitTimeConstructorCache\* ctorCache;

FixedFieldInfo\* fixedFieldInfoArray;

PropertyId propertyId;

Js::TypeId typeId;

uint id;

// Union with uint16 flags for fast default initialization

union

{

struct

{

bool falseReferencePreventionBit : 1;

bool isPolymorphic : 1;

bool isRootObjectNonConfigurableField : 1;

bool isRootObjectNonConfigurableFieldLoad : 1;

bool usesAuxSlot : 1;

bool isLocal : 1;

bool isLoadedFromProto : 1;

bool usesAccessor : 1;

bool hasFixedValue : 1;

bool keepFieldValue : 1;

bool isBeingStored : 1;

bool isBeingAdded : 1;

bool doesntHaveEquivalence : 1;

bool isBuiltIn : 1;

};

struct

{

uint16 flags;

};

};

uint16 slotIndex;

uint16 fixedFieldCount; // currently used only for fields that are functions

public:

ObjTypeSpecFldInfo() :

id(0), typeId(TypeIds\_Limit), typeSet(nullptr), initialType(nullptr), flags(InitialObjTypeSpecFldInfoFlagValue),

slotIndex(Constants::NoSlot), propertyId(Constants::NoProperty), protoObject(nullptr), propertyGuard(nullptr),

ctorCache(nullptr), fixedFieldInfoArray(nullptr) {}

ObjTypeSpecFldInfo(uint id, TypeId typeId, Type\* initialType,

bool usesAuxSlot, bool isLoadedFromProto, bool usesAccessor, bool isFieldValueFixed, bool keepFieldValue, bool isBuiltIn,

uint16 slotIndex, PropertyId propertyId, DynamicObject\* protoObject, PropertyGuard\* propertyGuard,

JitTimeConstructorCache\* ctorCache, FixedFieldInfo\* fixedFieldInfoArray) :

id(id), typeId(typeId), typeSet(nullptr), initialType(initialType), flags(InitialObjTypeSpecFldInfoFlagValue),

slotIndex(slotIndex), propertyId(propertyId), protoObject(protoObject), propertyGuard(propertyGuard),

ctorCache(ctorCache), fixedFieldInfoArray(fixedFieldInfoArray)

{

this->isPolymorphic = false;

this->usesAuxSlot = usesAuxSlot;

this->isLocal = !isLoadedFromProto && !usesAccessor;

this->isLoadedFromProto = isLoadedFromProto;

this->usesAccessor = usesAccessor;

this->hasFixedValue = isFieldValueFixed;

this->keepFieldValue = keepFieldValue;

this->isBeingAdded = initialType != nullptr;

this->doesntHaveEquivalence = true; // doesn't mean anything for data from a monomorphic cache

this->isBuiltIn = isBuiltIn;

this->fixedFieldCount = 1;

}

ObjTypeSpecFldInfo(uint id, TypeId typeId, Type\* initialType, EquivalentTypeSet\* typeSet,

bool usesAuxSlot, bool isLoadedFromProto, bool usesAccessor, bool isFieldValueFixed, bool keepFieldValue, bool doesntHaveEquivalence, bool isPolymorphic,

uint16 slotIndex, PropertyId propertyId, DynamicObject\* protoObject, PropertyGuard\* propertyGuard,

JitTimeConstructorCache\* ctorCache, FixedFieldInfo\* fixedFieldInfoArray, uint16 fixedFieldCount) :

id(id), typeId(typeId), typeSet(typeSet), initialType(initialType), flags(InitialObjTypeSpecFldInfoFlagValue),

slotIndex(slotIndex), propertyId(propertyId), protoObject(protoObject), propertyGuard(propertyGuard),

ctorCache(ctorCache), fixedFieldInfoArray(fixedFieldInfoArray)

{

this->isPolymorphic = isPolymorphic;

this->usesAuxSlot = usesAuxSlot;

this->isLocal = !isLoadedFromProto && !usesAccessor;

this->isLoadedFromProto = isLoadedFromProto;

this->usesAccessor = usesAccessor;

this->hasFixedValue = isFieldValueFixed;

this->keepFieldValue = keepFieldValue;

this->isBeingAdded = initialType != nullptr;

this->doesntHaveEquivalence = doesntHaveEquivalence;

this->isBuiltIn = false;

this->fixedFieldCount = fixedFieldCount;

}

static ObjTypeSpecFldInfo\* CreateFrom(uint id, InlineCache\* cache, uint cacheId,

EntryPointInfo \*entryPoint, FunctionBody\* const topFunctionBody, FunctionBody \*const functionBody, FieldAccessStatsPtr inlineCacheStats);

static ObjTypeSpecFldInfo\* CreateFrom(uint id, PolymorphicInlineCache\* cache, uint cacheId,

EntryPointInfo \*entryPoint, FunctionBody\* const topFunctionBody, FunctionBody \*const functionBody, FieldAccessStatsPtr inlineCacheStats);

uint GetObjTypeSpecFldId() const

{

return this->id;

}

bool IsMono() const

{

return !this->isPolymorphic;

}

bool IsPoly() const

{

return this->isPolymorphic;

}

bool UsesAuxSlot() const

{

return this->usesAuxSlot;

}

void SetUsesAuxSlot(bool value)

{

this->usesAuxSlot = value;

}

bool IsLoadedFromProto() const

{

return this->isLoadedFromProto;

}

bool IsLocal() const

{

return this->isLocal;

}

bool UsesAccessor() const

{

return this->usesAccessor;

}

bool HasFixedValue() const

{

return this->hasFixedValue;

}

void SetHasFixedValue(bool value)

{

this->hasFixedValue = value;

}

bool IsBeingStored() const

{

return this->isBeingStored;

}

void SetIsBeingStored(bool value)

{

this->isBeingStored = value;

}

bool IsBeingAdded() const

{

return this->isBeingAdded;

}

void SetIsBeingAdded(bool value)

{

this->isBeingAdded = value;

}

bool IsRootObjectNonConfigurableField() const

{

return this->isRootObjectNonConfigurableField;

}

bool IsRootObjectNonConfigurableFieldLoad() const

{

return this->isRootObjectNonConfigurableField && this->isRootObjectNonConfigurableFieldLoad;

}

void SetRootObjectNonConfigurableField(bool isLoad)

{

this->isRootObjectNonConfigurableField = true;

this->isRootObjectNonConfigurableFieldLoad = isLoad;

}

bool DoesntHaveEquivalence() const

{

return this->doesntHaveEquivalence;

}

void ClearFlags()

{

this->flags = 0;

}

void SetFlags(uint16 flags)

{

this->flags = flags | 0x01;

}

uint16 GetSlotIndex() const

{

return this->slotIndex;

}

void SetSlotIndex(uint16 index)

{

this->slotIndex = index;

}

PropertyId GetPropertyId() const

{

return this->propertyId;

}

Js::DynamicObject\* GetProtoObject() const

{

Assert(IsLoadedFromProto());

return this->protoObject;

}

Var GetFieldValue() const

{

Assert(IsMono() || (IsPoly() && !DoesntHaveEquivalence()));

return this->fixedFieldInfoArray[0].fieldValue;

}

Var GetFieldValue(uint i) const

{

Assert(IsPoly());

return this->fixedFieldInfoArray[i].fieldValue;

}

void SetFieldValue(Var value)

{

Assert(IsMono() || (IsPoly() && !DoesntHaveEquivalence()));

this->fixedFieldInfoArray[0].fieldValue = value;

}

Var GetFieldValueAsFixedDataIfAvailable() const;

Js::JavascriptFunction\* GetFieldValueAsFixedFunction() const;

Js::JavascriptFunction\* GetFieldValueAsFixedFunction(uint i) const;

Js::JavascriptFunction\* GetFieldValueAsFunction() const;

Js::JavascriptFunction\* GetFieldValueAsFunctionIfAvailable() const;

Js::JavascriptFunction\* GetFieldValueAsFixedFunctionIfAvailable() const;

Js::JavascriptFunction\* GetFieldValueAsFixedFunctionIfAvailable(uint i) const;

bool GetKeepFieldValue() const

{

return this->keepFieldValue;

}

Js::JitTimeConstructorCache\* GetCtorCache() const

{

return this->ctorCache;

}

Js::PropertyGuard\* GetPropertyGuard() const

{

return this->propertyGuard;

}

bool IsObjTypeSpecCandidate() const

{

return true;

}

bool IsMonoObjTypeSpecCandidate() const

{

return IsObjTypeSpecCandidate() && IsMono();

}

bool IsPolyObjTypeSpecCandidate() const

{

return IsObjTypeSpecCandidate() && IsPoly();

}

Js::TypeId GetTypeId() const

{

Assert(typeId != TypeIds\_Limit);

return this->typeId;

}

Js::TypeId GetTypeId(uint i) const

{

Assert(IsPoly());

return this->fixedFieldInfoArray[i].type->GetTypeId();

}

Js::Type \* GetType() const

{

Assert(IsObjTypeSpecCandidate() && IsMono());

return this->fixedFieldInfoArray[0].type;

}

Js::Type \* GetType(uint i) const

{

Assert(IsPoly());

return this->fixedFieldInfoArray[i].type;

}

bool HasInitialType() const

{

return IsObjTypeSpecCandidate() && IsMono() && !IsLoadedFromProto() && this->initialType != nullptr;

}

Js::Type \* GetInitialType() const

{

Assert(IsObjTypeSpecCandidate() && IsMono() && !IsLoadedFromProto());

return this->initialType;

}

bool HasEquivalentTypeSet() const

{

Assert(IsObjTypeSpecCandidate());

return this->typeSet != nullptr;

}

Js::EquivalentTypeSet \* GetEquivalentTypeSet() const

{

Assert(IsObjTypeSpecCandidate());

return this->typeSet;

}

Js::Type \* GetFirstEquivalentType() const

{

Assert(IsObjTypeSpecCandidate() && this->typeSet);

return this->typeSet->GetFirstType();

}

Js::FixedFieldInfo\* GetFixedFieldInfoArray()

{

return this->fixedFieldInfoArray;

}

uint16 GetFixedFieldCount()

{

return this->fixedFieldCount;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

const wchar\_t \*GetCacheLayoutString() const;

#endif

};

class ObjTypeSpecFldInfoArray

{

private:

ObjTypeSpecFldInfo\*\* infoArray;

#if DBG

uint infoCount;

#endif

public:

ObjTypeSpecFldInfoArray();

private:

void EnsureArray(Recycler \*const recycler, FunctionBody \*const functionBody);

public:

ObjTypeSpecFldInfo\* GetInfo(FunctionBody \*const functionBody, const uint index) const;

void SetInfo(Recycler \*const recycler, FunctionBody \*const functionBody,

const uint index, ObjTypeSpecFldInfo\* info);

void Reset();

template <class Fn>

void Map(Fn fn, uint count) const

{

if (this->infoArray != nullptr)

{

for (uint i = 0; i < count; i++)

{

ObjTypeSpecFldInfo\* info = this->infoArray[i];

if (info != nullptr)

{

fn(info);

}

}

}

};

PREVENT\_COPY(ObjTypeSpecFldInfoArray)

};

// - Data generated for jitting purposes

// - Recycler-allocated, lifetime is from when a code gen work item is added to the jit queue, to when jitting is complete

// - Also keeps the function body and inlinee function bodies alive while jitting.

class FunctionCodeGenJitTimeData

{

private:

FunctionInfo \*const functionInfo;

// Point's to an entry point if the work item needs the entry point alive- null for cases where the entry point isn't used

EntryPointInfo \*const entryPointInfo;

// These cloned inline caches are guaranteed to have stable data while jitting, but will be collectible after jitting

ObjTypeSpecFldInfoArray objTypeSpecFldInfoArray;

// Globally ordered list of all object type specialized property access information (monomorphic and polymorphic caches combined).

uint globalObjTypeSpecFldInfoCount;

ObjTypeSpecFldInfo\*\* globalObjTypeSpecFldInfoArray;

// There will be a non-null entry for each profiled call site where a function is to be inlined

FunctionCodeGenJitTimeData \*\*inlinees;

FunctionCodeGenJitTimeData \*\*ldFldInlinees;

RecyclerWeakReference<FunctionBody> \*weakFuncRef;

// Number of functions that are to be inlined (this is not the length of the 'inlinees' array above, includes getter setter inlinee count)

uint inlineeCount;

// Number of counts of getter setter to be inlined. This is not an exact count as inline caches are shared and we have no way of knowing

// accurate count.

uint ldFldInlineeCount;

// For polymorphic call site we will have linked list of FunctionCodeGenJitTimeData

// Each is differentiated by id starting from 0, 1

FunctionCodeGenJitTimeData \*next;

bool isInlined;

// This indicates the function is aggressively Inlined(see NativeCodeGenerator::TryAggressiveInlining) .

bool isAggressiveInliningEnabled;

// The profiled iterations need to be determined at the time of gathering code gen data on the main thread

const uint16 profiledIterations;

#ifdef FIELD\_ACCESS\_STATS

public:

FieldAccessStatsPtr inlineCacheStats;

void EnsureInlineCacheStats(Recycler\* recycler);

void AddInlineeInlineCacheStats(FunctionCodeGenJitTimeData\* inlineeJitTimeData);

#endif

public:

FunctionCodeGenJitTimeData(FunctionInfo \*const functionInfo, EntryPointInfo \*const entryPoint, bool isInlined = true);

public:

BVFixed \*inlineesBv;

FunctionInfo \*GetFunctionInfo() const;

FunctionBody \*GetFunctionBody() const;

FunctionCodeGenJitTimeData \*GetNext() const { return next; };

const ObjTypeSpecFldInfoArray\* GetObjTypeSpecFldInfoArray() const { return &this->objTypeSpecFldInfoArray; }

ObjTypeSpecFldInfoArray\* GetObjTypeSpecFldInfoArray() { return &this->objTypeSpecFldInfoArray; }

EntryPointInfo\* GetEntryPointInfo() const { return this->entryPointInfo; }

public:

const FunctionCodeGenJitTimeData \*GetInlinee(const ProfileId profiledCallSiteId) const;

const FunctionCodeGenJitTimeData \*GetLdFldInlinee(const InlineCacheIndex inlineCacheIndex) const;

FunctionCodeGenJitTimeData \*AddInlinee(

Recycler \*const recycler,

const ProfileId profiledCallSiteId,

FunctionInfo \*const inlinee,

bool isInlined = true);

uint InlineeCount() const;

bool IsLdFldInlineePresent() const { return ldFldInlineeCount != 0; }

RecyclerWeakReference<FunctionBody> \*GetWeakFuncRef() const { return this->weakFuncRef; }

void SetWeakFuncRef(RecyclerWeakReference<FunctionBody> \*weakFuncRef)

{

Assert(this->weakFuncRef == nullptr || weakFuncRef == nullptr || this->weakFuncRef == weakFuncRef);

this->weakFuncRef = weakFuncRef;

}

FunctionCodeGenJitTimeData \*AddLdFldInlinee(

Recycler \*const recycler,

const InlineCacheIndex inlineCacheIndex,

FunctionInfo \*const inlinee);

bool IsPolymorphicCallSite(const ProfileId profiledCallSiteId) const;

// This function walks all the chained jittimedata and returns the one which match the functionInfo.

// This can return null, if the functionInfo doesn't match.

const FunctionCodeGenJitTimeData \*GetJitTimeDataFromFunctionInfo(FunctionInfo \*polyFunctionInfo) const;

ObjTypeSpecFldInfo\* GetGlobalObjTypeSpecFldInfo(uint propertyInfoId) const

{

Assert(this->globalObjTypeSpecFldInfoArray != nullptr && propertyInfoId < this->globalObjTypeSpecFldInfoCount);

return this->globalObjTypeSpecFldInfoArray[propertyInfoId];

}

void SetGlobalObjTypeSpecFldInfo(uint propertyInfoId, ObjTypeSpecFldInfo\* info) const

{

Assert(this->globalObjTypeSpecFldInfoArray != nullptr && propertyInfoId < this->globalObjTypeSpecFldInfoCount);

this->globalObjTypeSpecFldInfoArray[propertyInfoId] = info;

}

void SetGlobalObjTypeSpecFldInfoArray(ObjTypeSpecFldInfo\*\* array, uint count)

{

Assert(array != nullptr);

this->globalObjTypeSpecFldInfoArray = array;

this->globalObjTypeSpecFldInfoCount = count;

}

bool GetIsInlined() const

{

return isInlined;

}

bool GetIsAggressiveInliningEnabled() const

{

return isAggressiveInliningEnabled;

}

void SetIsAggressiveInliningEnabled()

{

isAggressiveInliningEnabled = true;

}

void SetupRecursiveInlineeChain(

Recycler \*const recycler,

const ProfileId profiledCallSiteId)

{

if (!inlinees)

{

inlinees = RecyclerNewArrayZ(recycler, FunctionCodeGenJitTimeData \*, GetFunctionBody()->GetProfiledCallSiteCount());

}

inlinees[profiledCallSiteId] = this;

inlineeCount++;

this->isInlined = isInlined;

}

uint16 GetProfiledIterations() const;

PREVENT\_COPY(FunctionCodeGenJitTimeData)

};

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Language\FunctionCodeGenRuntimeData.h"

namespace Js

{

FunctionCodeGenRuntimeData::FunctionCodeGenRuntimeData(FunctionBody \*const functionBody)

: functionBody(functionBody), inlinees(nullptr), next(0)

{

}

FunctionBody \*FunctionCodeGenRuntimeData::GetFunctionBody() const

{

return functionBody;

}

const InlineCachePointerArray<InlineCache> \*FunctionCodeGenRuntimeData::ClonedInlineCaches() const

{

return &clonedInlineCaches;

}

InlineCachePointerArray<InlineCache> \*FunctionCodeGenRuntimeData::ClonedInlineCaches()

{

return &clonedInlineCaches;

}

const FunctionCodeGenRuntimeData \*FunctionCodeGenRuntimeData::GetInlinee(const ProfileId profiledCallSiteId) const

{

Assert(profiledCallSiteId < functionBody->GetProfiledCallSiteCount());

return inlinees ? inlinees[profiledCallSiteId] : nullptr;

}

const FunctionCodeGenRuntimeData \*FunctionCodeGenRuntimeData::GetInlineeForTargetInlinee(const ProfileId profiledCallSiteId, FunctionBody \*inlineeFuncBody) const

{

Assert(profiledCallSiteId < functionBody->GetProfiledCallSiteCount());

if (!inlinees)

{

return nullptr;

}

FunctionCodeGenRuntimeData \*runtimeData = inlinees[profiledCallSiteId];

while (runtimeData && runtimeData->GetFunctionBody() != inlineeFuncBody)

{

runtimeData = runtimeData->next;

}

return runtimeData;

}

void FunctionCodeGenRuntimeData::SetupRecursiveInlineeChain(

Recycler \*const recycler,

const ProfileId profiledCallSiteId,

FunctionBody \*const inlinee)

{

Assert(recycler);

Assert(profiledCallSiteId < functionBody->GetProfiledCallSiteCount());

Assert(inlinee == functionBody);

if (!inlinees)

{

inlinees = RecyclerNewArrayZ(recycler, FunctionCodeGenRuntimeData \*, functionBody->GetProfiledCallSiteCount());

}

inlinees[profiledCallSiteId] = this;

}

FunctionCodeGenRuntimeData \*FunctionCodeGenRuntimeData::EnsureInlinee(

Recycler \*const recycler,

const ProfileId profiledCallSiteId,

FunctionBody \*const inlinee)

{

Assert(recycler);

Assert(profiledCallSiteId < functionBody->GetProfiledCallSiteCount());

Assert(inlinee);

if (!inlinees)

{

inlinees = RecyclerNewArrayZ(recycler, FunctionCodeGenRuntimeData \*, functionBody->GetProfiledCallSiteCount());

}

const auto inlineeData = inlinees[profiledCallSiteId];

if (!inlineeData)

{

return inlinees[profiledCallSiteId] = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

}

// Find the right code gen runtime data

FunctionCodeGenRuntimeData \*next = inlineeData;

while (next && next->functionBody != inlinee)

{

next = next->next;

}

if (next)

{

return next;

}

FunctionCodeGenRuntimeData \*runtimeData = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

runtimeData->next = inlineeData;

return inlinees[profiledCallSiteId] = runtimeData;

}

const FunctionCodeGenRuntimeData \*FunctionCodeGenRuntimeData::GetLdFldInlinee(const InlineCacheIndex inlineCacheIndex) const

{

Assert(inlineCacheIndex < functionBody->GetInlineCacheCount());

return ldFldInlinees ? ldFldInlinees[inlineCacheIndex] : nullptr;

}

const FunctionCodeGenRuntimeData \*FunctionCodeGenRuntimeData::GetRuntimeDataFromFunctionInfo(FunctionInfo \*polyFunctionInfo) const

{

const FunctionCodeGenRuntimeData \*next = this;

while (next && next->functionBody != polyFunctionInfo)

{

next = next->next;

}

return next;

}

FunctionCodeGenRuntimeData \*FunctionCodeGenRuntimeData::EnsureLdFldInlinee(

Recycler \*const recycler,

const InlineCacheIndex inlineCacheIndex,

FunctionBody \*const inlinee)

{

Assert(recycler);

Assert(inlineCacheIndex < functionBody->GetInlineCacheCount());

Assert(inlinee);

if (!ldFldInlinees)

{

ldFldInlinees = RecyclerNewArrayZ(recycler, FunctionCodeGenRuntimeData \*, functionBody->GetInlineCacheCount());

}

const auto inlineeData = ldFldInlinees[inlineCacheIndex];

if (!inlineeData)

{

return ldFldInlinees[inlineCacheIndex] = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

}

return inlineeData;

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

// - Data generated for jitting and runtime purposes. Data grows as different versions of the function body are jitted.

// - Recycler-allocated, lifetime is from when a code gen work item is added to the jit queue, to when the function body

// becomes collectible

class FunctionCodeGenRuntimeData sealed

{

private:

FunctionBody \*const functionBody;

// These cloned inline caches are guaranteed to be kept alive for the life of the function body. They may be shared

// by different versions of the same function body that have been or will be jitted. Cached data is not guaranteed to be

// stable while jitting.

InlineCachePointerArray<InlineCache> clonedInlineCaches;

// There will be a non-null entry for each profiled call site where a function is to be inlined or has previously been

// inlined

FunctionCodeGenRuntimeData \*\*inlinees;

// There will be a non-null entry for each call site where a getter setter is to be inlined or has previously been

// inlined

FunctionCodeGenRuntimeData \*\*ldFldInlinees;

FunctionCodeGenRuntimeData \*next;

public:

FunctionCodeGenRuntimeData(FunctionBody \*const functionBody);

void SetupRuntimeDataChain(FunctionCodeGenRuntimeData \*nextRuntimeData) { this->next = nextRuntimeData; }

public:

FunctionBody \*GetFunctionBody() const;

FunctionCodeGenRuntimeData \*GetNext() const { return next; };

const InlineCachePointerArray<InlineCache> \*ClonedInlineCaches() const;

InlineCachePointerArray<InlineCache> \*ClonedInlineCaches();

const FunctionCodeGenRuntimeData \*GetInlinee(const ProfileId profiledCallSiteId) const;

const FunctionCodeGenRuntimeData \*GetInlineeForTargetInlinee(const ProfileId profiledCallSiteId, FunctionBody \*inlineeFuncBody) const;

FunctionCodeGenRuntimeData \*EnsureInlinee(

Recycler \*const recycler,

const ProfileId profiledCallSiteId,

FunctionBody \*const inlinee);

void SetupRecursiveInlineeChain(

Recycler \*const recycler,

const ProfileId profiledCallSiteId,

FunctionBody \*const inlinee);

const FunctionCodeGenRuntimeData \*GetLdFldInlinee(const InlineCacheIndex inlineCacheIndex) const;

FunctionCodeGenRuntimeData \*EnsureLdFldInlinee(

Recycler \*const recycler,

const InlineCacheIndex inlineCacheIndex,

FunctionBody \*const inlinee);

// This function walks all the chained jittimedata and returns the one which match the functionInfo.

// This can return null, if the functionInfo doesn't match.

const FunctionCodeGenRuntimeData \*GetRuntimeDataFromFunctionInfo(FunctionInfo \*polyFunctionInfo) const;

template<class Fn>

void MapInlineCaches(Fn fn) const

{

this->clonedInlineCaches.Map(fn, this->functionBody->GetInlineCacheCount());

for (ProfileId iInlinee = 0; iInlinee < this->functionBody->GetProfiledCallSiteCount(); iInlinee++)

{

const FunctionCodeGenRuntimeData\* runtimeData = this->GetInlinee(iInlinee);

while (runtimeData)

{

if (functionBody == runtimeData->GetFunctionBody())

{

break;

}

// Map for chained ones as well.

runtimeData->MapInlineCaches(fn);

runtimeData = runtimeData->next;

}

}

}

PREVENT\_COPY(FunctionCodeGenRuntimeData)

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

namespace Js

{

void InlineCache::CacheLocal(

Type \*const type,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const typeWithoutProperty,

int requiredAuxSlotCapacity,

ScriptContext \*const requestContext)

{

Assert(type);

Assert(propertyId != Constants::NoProperty);

Assert(propertyIndex != Constants::NoSlot);

Assert(requestContext);

Assert(type->GetScriptContext() == requestContext);

DebugOnly(VerifyRegistrationForInvalidation(this, requestContext, propertyId));

Assert(requiredAuxSlotCapacity >= 0 && requiredAuxSlotCapacity < 0x01 << RequiredAuxSlotCapacityBitCount);

// Store field and load field caches are never shared so we should never have a prototype cache morphing into an add property cache.

// We may, however, have a flags cache (setter) change to add property cache.

Assert(typeWithoutProperty == nullptr || !IsProto());

requestContext->RegisterAsScriptContextWithInlineCaches();

// Add cache into a store field cache list if required, but not there yet.

if (typeWithoutProperty != nullptr && invalidationListSlotPtr == nullptr)

{

// Note, this can throw due to OOM, so we need to do it before the inline cache is set below.

requestContext->RegisterStoreFieldInlineCache(this, propertyId);

}

if (isInlineSlot)

{

u.local.type = type;

u.local.typeWithoutProperty = typeWithoutProperty;

}

else

{

u.local.type = TypeWithAuxSlotTag(type);

u.local.typeWithoutProperty = typeWithoutProperty ? TypeWithAuxSlotTag(typeWithoutProperty) : nullptr;

}

u.local.isLocal = true;

u.local.slotIndex = propertyIndex;

u.local.requiredAuxSlotCapacity = requiredAuxSlotCapacity;

DebugOnly(VerifyRegistrationForInvalidation(this, requestContext, propertyId));

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

Output::Print(L"IC::CacheLocal, %s: ", requestContext->GetPropertyName(propertyId)->GetBuffer());

Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

void InlineCache::CacheProto(

DynamicObject \*const prototypeObjectWithProperty,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

const bool isMissing,

Type \*const type,

ScriptContext \*const requestContext)

{

Assert(prototypeObjectWithProperty);

Assert(propertyId != Constants::NoProperty);

Assert(propertyIndex != Constants::NoSlot);

Assert(type);

Assert(requestContext);

Assert(prototypeObjectWithProperty->GetScriptContext() == requestContext);

DebugOnly(VerifyRegistrationForInvalidation(this, requestContext, propertyId));

// This is an interesting quirk. In the browser Chakra's global object cannot be used directly as a prototype, because

// the global object (referenced either as window or as this) always points to the host object. Thus, when we retrieve

// a property from Chakra's global object the prototypeObjectWithProperty != info->GetInstance() and we will never cache

// such property loads (see CacheOperators::CachePropertyRead). However, in jc.exe or jshost.exe the only global object

// is Chakra's global object, and so prototypeObjectWithProperty == info->GetInstance() and we can cache. Hence, the

// assert below is only correct when running in the browser.

// Assert(prototypeObjectWithProperty != prototypeObjectWithProperty->type->GetLibrary()->GetGlobalObject());

// Store field and load field caches are never shared so we should never have an add property cache morphing into a prototype cache.

Assert(!IsLocal() || u.local.typeWithoutProperty == nullptr);

requestContext->RegisterAsScriptContextWithInlineCaches();

// Add cache into a proto cache list if not there yet.

if (invalidationListSlotPtr == nullptr)

{

// Note, this can throw due to OOM, so we need to do it before the inline cache is set below.

requestContext->RegisterProtoInlineCache(this, propertyId);

}

u.proto.prototypeObject = prototypeObjectWithProperty;

u.proto.isProto = true;

u.proto.isMissing = isMissing;

u.proto.slotIndex = propertyIndex;

if (isInlineSlot)

{

u.proto.type = type;

}

else

{

u.proto.type = TypeWithAuxSlotTag(type);

}

DebugOnly(VerifyRegistrationForInvalidation(this, requestContext, propertyId));

Assert(u.proto.isMissing == (uint16)(u.proto.prototypeObject == requestContext->GetLibrary()->GetMissingPropertyHolder()));

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

Output::Print(L"IC::CacheProto, %s: ", requestContext->GetPropertyName(propertyId)->GetBuffer());

Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

// TODO (InlineCacheCleanup): When simplifying inline caches due to not sharing between loads and stores, create two

// separate methods CacheSetter and CacheGetter.

void InlineCache::CacheAccessor(

const bool isGetter,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const type,

DynamicObject \*const object,

const bool isOnProto,

ScriptContext \*const requestContext)

{

Assert(propertyId != Constants::NoProperty);

Assert(propertyIndex != Constants::NoSlot);

Assert(type);

Assert(object);

Assert(requestContext);

DebugOnly(VerifyRegistrationForInvalidation(this, requestContext, propertyId));

// It is possible that prototype is from a different scriptContext than the original instance. We don't want to cache

// in this case.

Assert(type->GetScriptContext() == requestContext);

requestContext->RegisterAsScriptContextWithInlineCaches();

if (isOnProto && invalidationListSlotPtr == nullptr)

{

// Note, this can throw due to OOM, so we need to do it before the inline cache is set below.

if (!isGetter)

{

// If the setter is on a prototype, this cache must be invalidated whenever proto

// caches are invalidated, so we must register it here. Note that store field inline

// caches are invalidated any time proto caches are invalidated.

requestContext->RegisterStoreFieldInlineCache(this, propertyId);

}

else

{

requestContext->RegisterProtoInlineCache(this, propertyId);

}

}

u.accessor.isAccessor = true;

// TODO (PersistentInlineCaches): Consider removing the flag altogether and just have a bit indicating

// whether the cache itself is a store field cache (isStore?).

u.accessor.flags = isGetter ? InlineCacheGetterFlag : InlineCacheSetterFlag;

u.accessor.isOnProto = isOnProto;

u.accessor.type = isInlineSlot ? type : TypeWithAuxSlotTag(type);

u.accessor.slotIndex = propertyIndex;

u.accessor.object = object;

DebugOnly(VerifyRegistrationForInvalidation(this, requestContext, propertyId));

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

Output::Print(L"IC::CacheAccessor, %s: ", requestContext->GetPropertyName(propertyId)->GetBuffer());

Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

bool InlineCache::PretendTryGetProperty(Type \*const type, PropertyCacheOperationInfo \* operationInfo)

{

if (type == u.local.type)

{

operationInfo->cacheType = CacheType\_Local;

operationInfo->slotType = SlotType\_Inline;

return true;

}

if (TypeWithAuxSlotTag(type) == u.local.type)

{

operationInfo->cacheType = CacheType\_Local;

operationInfo->slotType = SlotType\_Aux;

return true;

}

if (type == u.proto.type)

{

operationInfo->cacheType = CacheType\_Proto;

operationInfo->slotType = SlotType\_Inline;

return true;

}

if (TypeWithAuxSlotTag(type) == u.proto.type)

{

operationInfo->cacheType = CacheType\_Proto;

operationInfo->slotType = SlotType\_Aux;

return true;

}

if (type == u.accessor.type)

{

Assert(u.accessor.flags & InlineCacheGetterFlag);

operationInfo->cacheType = CacheType\_Getter;

operationInfo->slotType = SlotType\_Inline;

return true;

}

if (TypeWithAuxSlotTag(type) == u.accessor.type)

{

Assert(u.accessor.flags & InlineCacheGetterFlag);

operationInfo->cacheType = CacheType\_Getter;

operationInfo->slotType = SlotType\_Aux;

return true;

}

return false;

}

bool InlineCache::PretendTrySetProperty(Type \*const type, Type \*const oldType, PropertyCacheOperationInfo \* operationInfo)

{

if (oldType == u.local.typeWithoutProperty)

{

operationInfo->cacheType = CacheType\_LocalWithoutProperty;

operationInfo->slotType = SlotType\_Inline;

return true;

}

if (TypeWithAuxSlotTag(oldType) == u.local.typeWithoutProperty)

{

operationInfo->cacheType = CacheType\_LocalWithoutProperty;

operationInfo->slotType = SlotType\_Aux;

return true;

}

if (type == u.local.type)

{

operationInfo->cacheType = CacheType\_Local;

operationInfo->slotType = SlotType\_Inline;

return true;

}

if (TypeWithAuxSlotTag(type) == u.local.type)

{

operationInfo->cacheType = CacheType\_Local;

operationInfo->slotType = SlotType\_Aux;

return true;

}

if (type == u.accessor.type)

{

if (u.accessor.flags & InlineCacheSetterFlag)

{

operationInfo->cacheType = CacheType\_Setter;

operationInfo->slotType = SlotType\_Inline;

return true;

}

}

if (TypeWithAuxSlotTag(type) == u.accessor.type)

{

if (u.accessor.flags & InlineCacheSetterFlag)

{

operationInfo->cacheType = CacheType\_Setter;

operationInfo->slotType = SlotType\_Aux;

return true;

}

}

return false;

}

bool InlineCache::GetGetterSetter(Type \*const type, RecyclableObject \*\*callee)

{

Type \*const taggedType = TypeWithAuxSlotTag(type);

\*callee = nullptr;

if (u.accessor.flags & (InlineCacheGetterFlag | InlineCacheSetterFlag))

{

if (type == u.accessor.type)

{

\*callee = RecyclableObject::FromVar(u.accessor.object->GetInlineSlot(u.accessor.slotIndex));

return true;

}

else if (taggedType == u.accessor.type)

{

\*callee = RecyclableObject::FromVar(u.accessor.object->GetAuxSlot(u.accessor.slotIndex));

return true;

}

}

return false;

}

bool InlineCache::GetCallApplyTarget(RecyclableObject\* obj, RecyclableObject \*\*callee)

{

Type \*const type = obj->GetType();

Type \*const taggedType = TypeWithAuxSlotTag(type);

\*callee = nullptr;

if (IsLocal())

{

if (type == u.local.type)

{

const Var objectAtInlineSlot = DynamicObject::FromVar(obj)->GetInlineSlot(u.local.slotIndex);

if (!Js::TaggedNumber::Is(objectAtInlineSlot))

{

\*callee = RecyclableObject::FromVar(objectAtInlineSlot);

return true;

}

}

else if (taggedType == u.local.type)

{

const Var objectAtAuxSlot = DynamicObject::FromVar(obj)->GetAuxSlot(u.local.slotIndex);

if (!Js::TaggedNumber::Is(objectAtAuxSlot))

{

\*callee = RecyclableObject::FromVar(DynamicObject::FromVar(obj)->GetAuxSlot(u.local.slotIndex));

return true;

}

}

return false;

}

else if (IsProto())

{

if (type == u.proto.type)

{

const Var objectAtInlineSlot = u.proto.prototypeObject->GetInlineSlot(u.proto.slotIndex);

if (!Js::TaggedNumber::Is(objectAtInlineSlot))

{

\*callee = RecyclableObject::FromVar(objectAtInlineSlot);

return true;

}

}

else if (taggedType == u.proto.type)

{

const Var objectAtAuxSlot = u.proto.prototypeObject->GetAuxSlot(u.proto.slotIndex);

if (!Js::TaggedNumber::Is(objectAtAuxSlot))

{

\*callee = RecyclableObject::FromVar(objectAtAuxSlot);

return true;

}

}

return false;

}

return false;

}

void InlineCache::Clear()

{

// IsEmpty() is a quick check to see that the cache is not populated, it only checks u.local.type, which does not

// guarantee that the proto or flags cache would not hit. So Clear() must still clear everything.

u.local.type = nullptr;

u.local.isLocal = true;

u.local.typeWithoutProperty = nullptr;

}

InlineCache \*InlineCache::Clone(Js::PropertyId propertyId, ScriptContext\* scriptContext)

{

Assert(scriptContext);

InlineCacheAllocator\* allocator = scriptContext->GetInlineCacheAllocator();

// Important to zero the allocated cache to be sure CopyTo doesn't see garbage

// when it uses the next pointer.

InlineCache\* clone = AllocatorNewZ(InlineCacheAllocator, allocator, InlineCache);

CopyTo(propertyId, scriptContext, clone);

return clone;

}

bool InlineCache::TryGetFixedMethodFromCache(Js::FunctionBody\* functionBody, uint cacheId, Js::JavascriptFunction\*\* pFixedMethod)

{

Assert(pFixedMethod);

if (IsEmpty())

{

return false;

}

Js::Type \* propertyOwnerType = nullptr;

bool isLocal = IsLocal();

bool isProto = IsProto();

if (isLocal)

{

propertyOwnerType = TypeWithoutAuxSlotTag(this->u.local.type);

}

else if (isProto)

{

// TODO (InlineCacheCleanup): For loads from proto, we could at least grab the value from protoObject's slot

// (given by the cache) and see if its a function. Only then, does it make sense to check with the type handler.

propertyOwnerType = this->u.proto.prototypeObject->GetType();

}

else

{

propertyOwnerType = this->u.accessor.object->GetType();

}

Assert(propertyOwnerType != nullptr);

if (Js::DynamicType::Is(propertyOwnerType->GetTypeId()))

{

Js::DynamicTypeHandler\* propertyOwnerTypeHandler = ((Js::DynamicType\*)propertyOwnerType)->GetTypeHandler();

Js::PropertyId propertyId = functionBody->GetPropertyIdFromCacheId(cacheId);

Js::PropertyRecord const \* const methodPropertyRecord = functionBody->GetScriptContext()->GetPropertyName(propertyId);

Var fixedMethod = nullptr;

bool isUseFixedProperty;

if (isLocal || isProto)

{

isUseFixedProperty = propertyOwnerTypeHandler->TryUseFixedProperty(methodPropertyRecord, &fixedMethod, Js::FixedPropertyKind::FixedMethodProperty, functionBody->GetScriptContext());

}

else

{

isUseFixedProperty = propertyOwnerTypeHandler->TryUseFixedAccessor(methodPropertyRecord, &fixedMethod, Js::FixedPropertyKind::FixedAccessorProperty, this->IsGetterAccessor(), functionBody->GetScriptContext());

}

AssertMsg(fixedMethod == nullptr || Js::JavascriptFunction::Is(fixedMethod), "The fixed value should have been a Method !!!");

\*pFixedMethod = reinterpret\_cast<JavascriptFunction\*>(fixedMethod);

return isUseFixedProperty;

}

return false;

}

void InlineCache::CopyTo(PropertyId propertyId, ScriptContext \* scriptContext, InlineCache \* const clone)

{

DebugOnly(VerifyRegistrationForInvalidation(this, scriptContext, propertyId));

DebugOnly(VerifyRegistrationForInvalidation(clone, scriptContext, propertyId));

Assert(clone != nullptr);

// Note, the Register methods can throw due to OOM, so we need to do it before the inline cache is copied below.

if (this->invalidationListSlotPtr != nullptr && clone->invalidationListSlotPtr == nullptr)

{

if (this->NeedsToBeRegisteredForProtoInvalidation())

{

scriptContext->RegisterProtoInlineCache(clone, propertyId);

}

else if (this->NeedsToBeRegisteredForStoreFieldInvalidation())

{

scriptContext->RegisterStoreFieldInlineCache(clone, propertyId);

}

}

clone->u = this->u;

DebugOnly(VerifyRegistrationForInvalidation(clone, scriptContext, propertyId));

}

template <bool isAccessor>

bool InlineCache::HasDifferentType(const bool isProto, const Type \* type, const Type \* typeWithoutProperty) const

{

Assert(!isAccessor && !isProto || !typeWithoutProperty);

if (isAccessor)

{

return !IsEmpty() && u.accessor.type != type && u.accessor.type != TypeWithAuxSlotTag(type);

}

if (isProto)

{

return !IsEmpty() && u.proto.type != type && u.proto.type != TypeWithAuxSlotTag(type);

}

// If the new type matches the cached type, the types without property must also match (unless one of them is null).

Assert((u.local.typeWithoutProperty == nullptr || typeWithoutProperty == nullptr) ||

((u.local.type != type || u.local.typeWithoutProperty == typeWithoutProperty) &&

(u.local.type != TypeWithAuxSlotTag(type) || u.local.typeWithoutProperty == TypeWithAuxSlotTag(typeWithoutProperty))));

// Don't consider a cache polymorphic, if it differs only by the typeWithoutProperty. We can handle this case with

// the monomorphic cache.

return !IsEmpty() && (u.local.type != type && u.local.type != TypeWithAuxSlotTag(type));

}

// explicit instantiation

template bool InlineCache::HasDifferentType<true>(const bool isProto, const Type \* type, const Type \* typeWithoutProperty) const;

template bool InlineCache::HasDifferentType<false>(const bool isProto, const Type \* type, const Type \* typeWithoutProperty) const;

bool InlineCache::NeedsToBeRegisteredForProtoInvalidation() const

{

return (IsProto() || IsGetterAccessorOnProto());

}

bool InlineCache::NeedsToBeRegisteredForStoreFieldInvalidation() const

{

return (IsLocal() && this->u.local.typeWithoutProperty != nullptr) || IsSetterAccessorOnProto();

}

#if DEBUG

bool InlineCache::NeedsToBeRegisteredForInvalidation() const

{

int howManyInvalidationsNeeded =

(int)NeedsToBeRegisteredForProtoInvalidation() +

(int)NeedsToBeRegisteredForStoreFieldInvalidation();

Assert(howManyInvalidationsNeeded <= 1);

return howManyInvalidationsNeeded > 0;

}

void InlineCache::VerifyRegistrationForInvalidation(const InlineCache\* cache, ScriptContext\* scriptContext, Js::PropertyId propertyId)

{

bool needsProtoInvalidation = cache->NeedsToBeRegisteredForProtoInvalidation();

bool needsStoreFieldInvalidation = cache->NeedsToBeRegisteredForStoreFieldInvalidation();

int howManyInvalidationsNeeded = (int)needsProtoInvalidation + (int)needsStoreFieldInvalidation;

bool hasListSlotPtr = cache->invalidationListSlotPtr != nullptr;

bool isProtoRegistered = hasListSlotPtr ? scriptContext->GetThreadContext()->IsProtoInlineCacheRegistered(cache, propertyId) : false;

bool isStoreFieldRegistered = hasListSlotPtr ? scriptContext->GetThreadContext()->IsStoreFieldInlineCacheRegistered(cache, propertyId) : false;

int howManyRegistrations = (int)isProtoRegistered + (int)isStoreFieldRegistered;

Assert(howManyInvalidationsNeeded <= 1);

Assert((howManyInvalidationsNeeded == 0) || hasListSlotPtr);

Assert(!needsProtoInvalidation || isProtoRegistered);

Assert(!needsStoreFieldInvalidation || isStoreFieldRegistered);

Assert(!hasListSlotPtr || howManyRegistrations > 0);

Assert(!hasListSlotPtr || (\*cache->invalidationListSlotPtr) == cache);

}

// Confirm inline cache miss against instance property lookup info.

bool InlineCache::ConfirmCacheMiss(const Type \* oldType, const PropertyValueInfo\* info) const

{

return u.local.type != oldType

&& u.proto.type != oldType

&& (u.accessor.type != oldType || info == NULL || u.accessor.flags != info->GetFlags());

}

#endif

#if DBG\_DUMP

void InlineCache::Dump()

{

if (this->u.local.isLocal)

{

Output::Print(L"LOCAL { types: 0x%X -> 0x%X, slot = %d, list slot ptr = 0x%X }",

this->u.local.typeWithoutProperty,

this->u.local.type,

this->u.local.slotIndex,

this->invalidationListSlotPtr

);

}

else if (this->u.proto.isProto)

{

Output::Print(L"PROTO { type = 0x%X, prototype = 0x%X, slot = %d, list slot ptr = 0x%X }",

this->u.proto.type,

this->u.proto.prototypeObject,

this->u.proto.slotIndex,

this->invalidationListSlotPtr

);

}

else if (this->u.accessor.isAccessor)

{

Output::Print(L"FLAGS { type = 0x%X, object = 0x%X, flag = 0x%X, slot = %d, list slot ptr = 0x%X }",

this->u.accessor.type,

this->u.accessor.object,

this->u.accessor.slotIndex,

this->u.accessor.flags,

this->invalidationListSlotPtr

);

}

else

{

Assert(this->u.accessor.type == 0);

Assert(this->u.accessor.slotIndex == 0);

Output::Print(L"uninitialized");

}

}

#endif

PolymorphicInlineCache \* PolymorphicInlineCache::New(uint16 size, FunctionBody \* functionBody)

{

ScriptContext \* scriptContext = functionBody->GetScriptContext();

InlineCache \* inlineCaches = AllocatorNewArrayZ(InlineCacheAllocator, scriptContext->GetInlineCacheAllocator(), InlineCache, size);

#ifdef POLY\_INLINE\_CACHE\_SIZE\_STATS

scriptContext->GetInlineCacheAllocator()->LogPolyCacheAlloc(size \* sizeof(InlineCache));

#endif

PolymorphicInlineCache \* polymorphicInlineCache = RecyclerNewFinalizedLeaf(scriptContext->GetRecycler(), PolymorphicInlineCache, inlineCaches, size, functionBody);

// Insert the cache into finalization list. We maintain this linked list of polymorphic caches because when we allocate

// a larger cache, the old one might still be used by some code on the stack. Consequently, we can't release

// the inline cache array back to the arena allocator. The list is leaf-allocated and so does not keep the

// old caches alive. As soon as they are collectible, their finalizer releases the inline cache array to the arena.

polymorphicInlineCache->prev = nullptr;

polymorphicInlineCache->next = functionBody->GetPolymorphicInlineCachesHead();

if (polymorphicInlineCache->next)

{

polymorphicInlineCache->next->prev = polymorphicInlineCache;

}

functionBody->SetPolymorphicInlineCachesHead(polymorphicInlineCache);

return polymorphicInlineCache;

}

template<bool isAccessor>

bool PolymorphicInlineCache::HasDifferentType(

const bool isProto,

const Type \* type,

const Type \* typeWithoutProperty) const

{

Assert(!isAccessor && !isProto || !typeWithoutProperty);

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

if (inlineCaches[inlineCacheIndex].HasDifferentType<isAccessor>(isProto, type, typeWithoutProperty))

{

return true;

}

if (!isAccessor && !isProto && typeWithoutProperty)

{

inlineCacheIndex = GetInlineCacheIndexForType(typeWithoutProperty);

return inlineCaches[inlineCacheIndex].HasDifferentType<isAccessor>(isProto, type, typeWithoutProperty);

}

return false;

}

// explicit instantiation

template bool PolymorphicInlineCache::HasDifferentType<true>(const bool isProto, const Type \* type, const Type \* typeWithoutProperty) const;

template bool PolymorphicInlineCache::HasDifferentType<false>(const bool isProto, const Type \* type, const Type \* typeWithoutProperty) const;

bool PolymorphicInlineCache::HasType\_Flags(const Type \* type) const

{

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

return inlineCaches[inlineCacheIndex].HasType\_Flags(type);

}

void PolymorphicInlineCache::UpdateInlineCachesFillInfo(uint index, bool set)

{

Assert(index < 0x20);

if (set)

{

this->inlineCachesFillInfo |= 1 << index;

}

else

{

this->inlineCachesFillInfo &= ~(1 << index);

}

}

bool PolymorphicInlineCache::IsFull()

{

Assert(this->size <= 0x20);

return this->inlineCachesFillInfo == ((1 << (this->size - 1)) << 1) - 1;

}

void PolymorphicInlineCache::CacheLocal(

Type \*const type,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const typeWithoutProperty,

int requiredAuxSlotCapacity,

ScriptContext \*const requestContext)

{

// Let's not waste polymorphic cache slots by caching both the type without property and type with property. If the

// cache is used for both adding a property and setting the existing property, then those instances will cause both

// types to be cached. Until then, caching both types proactively here can unnecessarily trash useful cached info

// because the types use different slots, unlike a monomorphic inline cache.

if (!typeWithoutProperty)

{

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

#if INTRUSIVE\_TESTTRACE\_PolymorphicInlineCache

bool collision = !inlineCaches[inlineCacheIndex].IsEmpty();

#endif

if (!PHASE\_OFF1(Js::CloneCacheInCollisionPhase))

{

if (!inlineCaches[inlineCacheIndex].IsEmpty() && !inlineCaches[inlineCacheIndex].NeedsToBeRegisteredForStoreFieldInvalidation())

{

if (inlineCaches[inlineCacheIndex].IsLocal())

{

CloneInlineCacheToEmptySlotInCollision<true, false, false>(type, inlineCacheIndex);

}

else if (inlineCaches[inlineCacheIndex].IsProto())

{

CloneInlineCacheToEmptySlotInCollision<false, true, false>(type, inlineCacheIndex);

}

else

{

CloneInlineCacheToEmptySlotInCollision<false, false, true>(type, inlineCacheIndex);

}

}

}

inlineCaches[inlineCacheIndex].CacheLocal(

type, propertyId, propertyIndex, isInlineSlot, nullptr, requiredAuxSlotCapacity, requestContext);

UpdateInlineCachesFillInfo(inlineCacheIndex, true /\*set\*/);

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::PolymorphicInlineCachePhase))

{

Output::Print(L"PIC::CacheLocal, %s, %d: ", requestContext->GetPropertyName(propertyId)->GetBuffer(), inlineCacheIndex);

inlineCaches[inlineCacheIndex].Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1(

Js::PolymorphicInlineCachePhase,

L"TestTrace PIC: CacheLocal, 0x%x, entryIndex = %d, collision = %s, entries = %d\n", this, inlineCacheIndex, collision ? L"true" : L"false", GetEntryCount());

}

else

{

uint inlineCacheIndex = GetInlineCacheIndexForType(typeWithoutProperty);

#if INTRUSIVE\_TESTTRACE\_PolymorphicInlineCache

bool collision = !inlineCaches[inlineCacheIndex].IsEmpty();

#endif

inlineCaches[inlineCacheIndex].CacheLocal(

type, propertyId, propertyIndex, isInlineSlot, typeWithoutProperty, requiredAuxSlotCapacity, requestContext);

UpdateInlineCachesFillInfo(inlineCacheIndex, true /\*set\*/);

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::PolymorphicInlineCachePhase))

{

Output::Print(L"PIC::CacheLocal, %s, %d: ", requestContext->GetPropertyName(propertyId)->GetBuffer(), inlineCacheIndex);

inlineCaches[inlineCacheIndex].Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1(

Js::PolymorphicInlineCachePhase,

L"TestTrace PIC: CacheLocal, 0x%x, entryIndex = %d, collision = %s, entries = %d\n", this, inlineCacheIndex, collision ? L"true" : L"false", GetEntryCount());

}

}

void PolymorphicInlineCache::CacheProto(

DynamicObject \*const prototypeObjectWithProperty,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

const bool isMissing,

Type \*const type,

ScriptContext \*const requestContext)

{

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

#if INTRUSIVE\_TESTTRACE\_PolymorphicInlineCache

bool collision = !inlineCaches[inlineCacheIndex].IsEmpty();

#endif

if (!PHASE\_OFF1(Js::CloneCacheInCollisionPhase))

{

if (!inlineCaches[inlineCacheIndex].IsEmpty() && !inlineCaches[inlineCacheIndex].NeedsToBeRegisteredForStoreFieldInvalidation())

{

if (inlineCaches[inlineCacheIndex].IsLocal())

{

CloneInlineCacheToEmptySlotInCollision<true, false, false>(type, inlineCacheIndex);

}

else if (inlineCaches[inlineCacheIndex].IsProto())

{

CloneInlineCacheToEmptySlotInCollision<false, true, false>(type, inlineCacheIndex);

}

else

{

CloneInlineCacheToEmptySlotInCollision<false, false, true>(type, inlineCacheIndex);

}

}

}

inlineCaches[inlineCacheIndex].CacheProto(

prototypeObjectWithProperty, propertyId, propertyIndex, isInlineSlot, isMissing, type, requestContext);

UpdateInlineCachesFillInfo(inlineCacheIndex, true /\*set\*/);

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::PolymorphicInlineCachePhase))

{

Output::Print(L"PIC::CacheProto, %s, %d: ", requestContext->GetPropertyName(propertyId)->GetBuffer(), inlineCacheIndex);

inlineCaches[inlineCacheIndex].Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1(

Js::PolymorphicInlineCachePhase,

L"TestTrace PIC: CacheProto, 0x%x, entryIndex = %d, collision = %s, entries = %d\n", this, inlineCacheIndex, collision ? L"true" : L"false", GetEntryCount());

}

void PolymorphicInlineCache::CacheAccessor(

const bool isGetter,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const type,

DynamicObject \*const object,

const bool isOnProto,

ScriptContext \*const requestContext)

{

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

#if INTRUSIVE\_TESTTRACE\_PolymorphicInlineCache

bool collision = !inlineCaches[inlineCacheIndex].IsEmpty();

#endif

if (!PHASE\_OFF1(Js::CloneCacheInCollisionPhase))

{

if (!inlineCaches[inlineCacheIndex].IsEmpty() && !inlineCaches[inlineCacheIndex].NeedsToBeRegisteredForStoreFieldInvalidation())

{

if (inlineCaches[inlineCacheIndex].IsLocal())

{

CloneInlineCacheToEmptySlotInCollision<true, false, false>(type, inlineCacheIndex);

}

else if (inlineCaches[inlineCacheIndex].IsProto())

{

CloneInlineCacheToEmptySlotInCollision<false, true, false>(type, inlineCacheIndex);

}

else

{

CloneInlineCacheToEmptySlotInCollision<false, false, true>(type, inlineCacheIndex);

}

}

}

inlineCaches[inlineCacheIndex].CacheAccessor(isGetter, propertyId, propertyIndex, isInlineSlot, type, object, isOnProto, requestContext);

UpdateInlineCachesFillInfo(inlineCacheIndex, true /\*set\*/);

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::PolymorphicInlineCachePhase))

{

Output::Print(L"PIC::CacheAccessor, %s, %d: ", requestContext->GetPropertyName(propertyId)->GetBuffer(), inlineCacheIndex);

inlineCaches[inlineCacheIndex].Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1(

Js::PolymorphicInlineCachePhase,

L"TestTrace PIC: CacheAccessor, 0x%x, entryIndex = %d, collision = %s, entries = %d\n", this, inlineCacheIndex, collision ? L"true" : L"false", GetEntryCount());

}

bool PolymorphicInlineCache::PretendTryGetProperty(

Type \*const type,

PropertyCacheOperationInfo \* operationInfo)

{

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

return inlineCaches[inlineCacheIndex].PretendTryGetProperty(type, operationInfo);

}

bool PolymorphicInlineCache::PretendTrySetProperty(

Type \*const type,

Type \*const oldType,

PropertyCacheOperationInfo \* operationInfo)

{

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

return inlineCaches[inlineCacheIndex].PretendTrySetProperty(type, oldType, operationInfo);

}

void PolymorphicInlineCache::CopyTo(PropertyId propertyId, ScriptContext\* scriptContext, PolymorphicInlineCache \*const clone)

{

Assert(clone);

clone->ignoreForEquivalentObjTypeSpec = this->ignoreForEquivalentObjTypeSpec;

clone->cloneForJitTimeUse = this->cloneForJitTimeUse;

for (uint i = 0; i < GetSize(); ++i)

{

Type \* type = inlineCaches[i].GetType();

if (type)

{

uint inlineCacheIndex = clone->GetInlineCacheIndexForType(type);

// When copying inline caches from one polymorphic cache to another, types are again hashed to get the corresponding indices in the new polymorphic cache.

// This might lead to collision in the new cache. We need to try to resolve that collision.

if (!PHASE\_OFF1(Js::CloneCacheInCollisionPhase))

{

if (!clone->inlineCaches[inlineCacheIndex].IsEmpty() && !clone->inlineCaches[inlineCacheIndex].NeedsToBeRegisteredForStoreFieldInvalidation())

{

if (clone->inlineCaches[inlineCacheIndex].IsLocal())

{

clone->CloneInlineCacheToEmptySlotInCollision<true, false, false>(type, inlineCacheIndex);

}

else if (clone->inlineCaches[inlineCacheIndex].IsProto())

{

clone->CloneInlineCacheToEmptySlotInCollision<false, true, false>(type, inlineCacheIndex);

}

else

{

clone->CloneInlineCacheToEmptySlotInCollision<false, false, true>(type, inlineCacheIndex);

}

}

}

inlineCaches[i].CopyTo(propertyId, scriptContext, &clone->inlineCaches[inlineCacheIndex]);

clone->UpdateInlineCachesFillInfo(inlineCacheIndex, true /\*set\*/);

}

}

}

#if DBG\_DUMP

void PolymorphicInlineCache::Dump()

{

for (uint i = 0; i < size; ++i)

{

if (!inlineCaches[i].IsEmpty())

{

Output::Print(L" %d: ", i);

inlineCaches[i].Dump();

Output::Print(L"\n");

}

}

}

#endif

bool EquivalentTypeSet::Contains(const Js::Type \* type, uint16\* pIndex) const

{

for (uint16 ti = 0; ti < this->count; ti++)

{

if (this->types[ti] == type)

{

if (pIndex)

{

\*pIndex = ti;

}

return true;

}

}

return false;

}

bool EquivalentTypeSet::AreIdentical(EquivalentTypeSet \* left, EquivalentTypeSet \* right)

{

if (!left->GetSortedAndDuplicatesRemoved())

{

left->SortAndRemoveDuplicates();

}

if (!right->GetSortedAndDuplicatesRemoved())

{

right->SortAndRemoveDuplicates();

}

Assert(left->GetSortedAndDuplicatesRemoved() && right->GetSortedAndDuplicatesRemoved());

if (left->count != right->count)

{

return false;

}

if (memcmp(left->types, right->types, left->count \* sizeof(Type\*)) == 0)

{

return true;

}

else

{

return false;

}

}

bool EquivalentTypeSet::IsSubsetOf(EquivalentTypeSet \* left, EquivalentTypeSet \* right)

{

if (!left->GetSortedAndDuplicatesRemoved())

{

left->SortAndRemoveDuplicates();

}

if (!right->GetSortedAndDuplicatesRemoved())

{

right->SortAndRemoveDuplicates();

}

if (left->count > right->count)

{

return false;

}

// Try to find each left type in the right set.

int j = 0;

for (int i = 0; i < left->count; i++)

{

bool found = false;

for (; j < right->count; j++)

{

if (left->types[i] < right->types[j])

{

// Didn't find the left type. Fail.

return false;

}

if (left->types[i] == right->types[j])

{

// Found the left type. Continue to the next left/right pair.

found = true;

j++;

break;

}

}

Assert(j <= right->count);

if (j == right->count && !found)

{

// Exhausted the right set without finding the current left type.

return false;

}

}

return true;

}

void EquivalentTypeSet::SortAndRemoveDuplicates()

{

uint16 oldCount = this->count;

uint16 i;

// sorting

for (i = 1; i < oldCount; i++)

{

uint16 j = i;

while (j > 0 && (this->types[j - 1] > this->types[j]))

{

Type\* tmp = this->types[j];

this->types[j] = this->types[j - 1];

this->types[j - 1] = tmp;

}

}

// removing duplicate types from the sorted set

i = 0;

for (uint16 j = 1; j < oldCount; j++)

{

if (this->types[i] != this->types[j])

{

this->types[++i] = this->types[j];

}

}

this->count = ++i;

for (i; i < oldCount; i++)

{

this->types[i] = nullptr;

}

this->sortedAndDuplicatesRemoved = true;

}

ConstructorCache ConstructorCache::DefaultInstance;

ConstructorCache\* ConstructorCache::EnsureValidInstance(ConstructorCache\* currentCache, ScriptContext\* scriptContext)

{

Assert(currentCache != nullptr);

ConstructorCache\* newCache = currentCache;

// If the old cache has been invalidated, we need to create a new one to avoid incorrectly re-validating

// caches that may have been hard-coded in the JIT-ed code with different prototype and type. However, if

// the cache is already polymorphic, it will not be hard-coded, and hence we don't need to allocate a new

// one - in case the prototype property changes frequently.

if (ConstructorCache::IsDefault(currentCache) || (currentCache->IsInvalidated() && !currentCache->IsPolymorphic()))

{

// Review (jedmiad): I don't think we need to zero the struct, since we initialize each field.

newCache = RecyclerNew(scriptContext->GetRecycler(), ConstructorCache);

// TODO: Consider marking the cache as polymorphic only if the prototype and type actually changed. In fact,

// if they didn't change we could reuse the same cache and simply mark it as valid. Not really true. The cache

// might have been invalidated due to a property becoming read-only. In that case we can't re-validate an old

// monomorphic cache. We must allocate a new one.

newCache->content.isPolymorphic = currentCache->content.isPopulated && currentCache->content.hasPrototypeChanged;

}

// If we kept the old invalidated cache, it better be marked as polymorphic.

Assert(!newCache->IsInvalidated() || newCache->IsPolymorphic());

// If the cache was polymorphic, we shouldn't have allocated a new one.

Assert(!currentCache->IsPolymorphic() || newCache == currentCache);

return newCache;

}

void ConstructorCache::InvalidateOnPrototypeChange()

{

if (IsDefault(this))

{

Assert(this->guard.value == CtorCacheGuardValues::Invalid);

Assert(!this->content.isPopulated);

}

else if (this->guard.value == CtorCacheGuardValues::Special && this->content.skipDefaultNewObject)

{

// Do nothing. If we skip the default object, changes to the prototype property don't affect

// what we'll do during object allocation.

// Can't assert the following because we set the prototype property during library initialization.

// AssertMsg(false, "Overriding a prototype on a built-in constructor should be illegal.");

}

else

{

this->guard.value = CtorCacheGuardValues::Invalid;

this->content.hasPrototypeChanged = true;

// Make sure we don't leak the old type.

Assert(this->content.type == nullptr);

this->content.pendingType = nullptr;

Assert(this->content.pendingType == nullptr);

Assert(IsInvalidated());

}

Assert(IsConsistent());

}

#if DBG\_DUMP

void ConstructorCache::Dump() const

{

Output::Print(L"guard value or type = 0x%p, script context = 0x%p, pending type = 0x%p, slots = %d, inline slots = %d, populated = %d, polymorphic = %d, update cache = %d, update type = %d, skip default = %d, no return = %d",

this->GetRawGuardValue(), this->GetScriptContext(), this->GetPendingType(), this->GetSlotCount(), this->GetInlineSlotCount(),

this->IsPopulated(), this->IsPolymorphic(), this->GetUpdateCacheAfterCtor(), this->GetTypeUpdatePending(),

this->GetSkipDefaultNewObject(), this->GetCtorHasNoExplicitReturnValue());

}

#endif

void IsInstInlineCache::Set(Type \* instanceType, JavascriptFunction \* function, JavascriptBoolean \* result)

{

this->type = instanceType;

this->function = function;

this->result = result;

}

void IsInstInlineCache::Clear()

{

this->type = NULL;

this->function = NULL;

this->result = NULL;

}

void IsInstInlineCache::Unregister(ScriptContext \* scriptContext)

{

scriptContext->GetThreadContext()->UnregisterIsInstInlineCache(this, this->function);

}

bool IsInstInlineCache::TryGetResult(Var instance, JavascriptFunction \* function, JavascriptBoolean \*\* result)

{

// In order to get the result from the cache we must have a function instance.

Assert(function != NULL);

if (this->function == function &&

this->type == RecyclableObject::FromVar(instance)->GetType())

{

if (result != nullptr)

{

(\*result = this->result);

}

return true;

}

else

{

return false;

}

}

void IsInstInlineCache::Cache(Type \* instanceType, JavascriptFunction \* function, JavascriptBoolean \* result, ScriptContext \* scriptContext)

{

// In order to populate the cache we must have a function instance.

Assert(function != nullptr);

// We assume the following invariant: (cache->function != nullptr) => script context is registered as having some populated instance-of inline caches and

// this cache is registered with thread context for invalidation.

if (this->function == function)

{

Assert(scriptContext->IsRegisteredAsScriptContextWithIsInstInlineCaches());

Assert(scriptContext->IsIsInstInlineCacheRegistered(this, function));

this->Set(instanceType, function, result);

}

else

{

if (this->function != nullptr)

{

Unregister(scriptContext);

Clear();

}

scriptContext->RegisterAsScriptContextWithIsInstInlineCaches();

// If the cache's function is not null, the cache must have been registered already. No need to register again.

// In fact, ThreadContext::RegisterIsInstInlineCache, would assert if we tried to re-register the same cache (to enforce the invariant above).

// Review (jedmiad): What happens if we run out of memory inside RegisterIsInstInlieCache?

scriptContext->RegisterIsInstInlineCache(this, function);

this->Set(instanceType, function, result);

}

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#define TypeWithAuxSlotTag(\_t) \

(reinterpret\_cast<Type\*>(reinterpret\_cast<size\_t>(\_t) | InlineCacheAuxSlotTypeTag))

#define TypeWithoutAuxSlotTag(\_t) \

(reinterpret\_cast<Js::Type\*>(reinterpret\_cast<size\_t>(\_t) & ~InlineCacheAuxSlotTypeTag))

#define TypeHasAuxSlotTag(\_t) \

(!!(reinterpret\_cast<size\_t>(\_t) & InlineCacheAuxSlotTypeTag))

#if defined(\_M\_IX86\_OR\_ARM32)

#define PolymorphicInlineCacheShift 5 // On 32 bit architectures, the least 5 significant bits of a DynamicTypePointer is 0

#else

#define PolymorphicInlineCacheShift 6 // On 64 bit architectures, the least 6 significant bits of a DynamicTypePointer is 0

#endif

namespace Js

{

enum CacheType : byte

{

CacheType\_None,

CacheType\_Local,

CacheType\_Proto,

CacheType\_LocalWithoutProperty,

CacheType\_Getter,

CacheType\_Setter,

CacheType\_TypeProperty,

};

enum SlotType : byte

{

SlotType\_None,

SlotType\_Inline,

SlotType\_Aux,

};

struct PropertyCacheOperationInfo

{

PropertyCacheOperationInfo()

: cacheType(CacheType\_None), slotType(SlotType\_None), isPolymorphic(false)

{

}

CacheType cacheType;

SlotType slotType;

bool isPolymorphic;

};

struct JitTimeInlineCache;

struct InlineCache

{

static const int CacheLayoutSelectorBitCount = 1;

static const int RequiredAuxSlotCapacityBitCount = 15;

static const bool IsPolymorphic = false;

InlineCache() {}

union

{

// Invariants:

// - Type\* fields do not overlap.

// - "next" field is non-null iff the cache is linked in a list of proto-caches

// (see ScriptContext::RegisterProtoInlineCache and ScriptContext::InvalidateProtoCaches).

struct s\_local

{

Type\* type;

// PatchPutValue caches here the type the object has before a new property is added.

// If this type is hit again we can immediately change the object's type to "type"

// and store the value into the slot "slotIndex".

Type\* typeWithoutProperty;

union

{

struct

{

uint16 isLocal : 1;

uint16 requiredAuxSlotCapacity : 15; // Maximum auxiliary slot capacity (for a path type) must be < 2^16

};

struct

{

uint16 rawUInt16; // Required for access from JIT-ed code

};

};

uint16 slotIndex;

} local;

struct s\_proto

{

uint16 isProto : 1;

uint16 isMissing : 1;

uint16 unused : 14;

uint16 slotIndex;

// It's OK for the type in proto layout to overlap with typeWithoutProperty in the local layout, because

// we only use typeWithoutProperty on field stores, which can never have a proto layout.

Type\* type;

DynamicObject\* prototypeObject;

} proto;

struct s\_accessor

{

DynamicObject \*object;

union

{

struct {

uint16 isAccessor : 1;

uint16 flags : 2;

uint16 isOnProto : 1;

uint16 unused : 12;

};

uint16 rawUInt16;

};

uint16 slotIndex;

Type \* type;

} accessor;

CompileAssert(sizeof(s\_local) == sizeof(s\_proto));

CompileAssert(sizeof(s\_local) == sizeof(s\_accessor));

} u;

InlineCache\*\* invalidationListSlotPtr;

bool IsEmpty() const

{

return u.local.type == nullptr;

}

bool IsLocal() const

{

return u.local.isLocal;

}

bool IsProto() const

{

return u.proto.isProto;

}

bool IsAccessor() const

{

return u.accessor.isAccessor;

}

bool IsAccessorOnProto() const

{

return IsAccessor() && u.accessor.isOnProto;

}

bool IsGetterAccessor() const

{

return IsAccessor() && !!(u.accessor.flags & InlineCacheGetterFlag);

}

bool IsGetterAccessorOnProto() const

{

return IsGetterAccessor() && u.accessor.isOnProto;

}

bool IsSetterAccessor() const

{

return IsAccessor() && !!(u.accessor.flags & InlineCacheSetterFlag);

}

bool IsSetterAccessorOnProto() const

{

return IsSetterAccessor() && u.accessor.isOnProto;

}

Type\* GetRawType() const

{

return IsLocal() ? u.local.type : (IsProto() ? u.proto.type : (IsAccessor() ? u.accessor.type : nullptr));

}

Type\* GetType() const

{

return TypeWithoutAuxSlotTag(GetRawType());

}

template<bool isAccessor>

bool HasDifferentType(const bool isProto, const Type \* type, const Type \* typeWithoutProperty) const;

bool HasType\_Flags(const Type \* type) const

{

return u.accessor.type == type || u.accessor.type == TypeWithAuxSlotTag(type);

}

bool HasDifferentType(const Type \* type) const

{

return !IsEmpty() && GetType() != type;

}

bool RemoveFromInvalidationList()

{

if (this->invalidationListSlotPtr == nullptr)

{

return false;

}

\*this->invalidationListSlotPtr = nullptr;

this->invalidationListSlotPtr = nullptr;

return true;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

const wchar\_t \*LayoutString() const

{

if (IsEmpty())

{

return L"Empty";

}

if (IsLocal())

{

return L"Local";

}

if (IsAccessor())

{

return L"Accessor";

}

return L"Proto";

}

#endif

public:

void CacheLocal(

Type \*const type,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const typeWithoutProperty,

int requiredAuxSlotCapacity,

ScriptContext \*const requestContext);

void CacheProto(

DynamicObject \*const prototypeObjectWithProperty,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

const bool isMissing,

Type \*const type,

ScriptContext \*const requestContext);

void CacheMissing(

DynamicObject \*const missingPropertyHolder,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const type,

ScriptContext \*const requestContext);

void CacheAccessor(

const bool isGetter,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const type,

DynamicObject \*const object,

const bool isOnProto,

ScriptContext \*const requestContext);

template<

bool CheckLocal,

bool CheckProto,

bool CheckAccessor,

bool CheckMissing,

bool ReturnOperationInfo>

bool TryGetProperty(

Var const instance,

RecyclableObject \*const propertyObject,

const PropertyId propertyId,

Var \*const propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo);

template<

bool CheckLocal,

bool CheckLocalTypeWithoutProperty,

bool CheckAccessor,

bool ReturnOperationInfo>

bool TrySetProperty(

RecyclableObject \*const object,

const PropertyId propertyId,

Var propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo,

const PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

bool PretendTryGetProperty(Type \*const type, PropertyCacheOperationInfo \* operationInfo);

bool PretendTrySetProperty(Type \*const type, Type \*const oldType, PropertyCacheOperationInfo \* operationInfo);

void Clear();

template <class TAllocator>

InlineCache \*Clone(TAllocator \*const allocator);

InlineCache \*Clone(Js::PropertyId propertyId, ScriptContext\* scriptContext);

void CopyTo(PropertyId propertyId, ScriptContext \* scriptContext, InlineCache \* const clone);

bool TryGetFixedMethodFromCache(Js::FunctionBody\* functionBody, uint cacheId, Js::JavascriptFunction\*\* pFixedMethod);

bool GetGetterSetter(Type \*const type, RecyclableObject \*\*callee);

bool GetCallApplyTarget(RecyclableObject\* obj, RecyclableObject \*\*callee);

static uint GetGetterFlagMask()

{

// First bit is marked for isAccessor in the accessor cache layout.

return InlineCacheGetterFlag << 1;

}

static uint GetSetterFlagMask()

{

// First bit is marked for isAccessor in the accessor cache layout.

return InlineCacheSetterFlag << 1;

}

static uint GetGetterSetterFlagMask()

{

// First bit is marked for isAccessor in the accessor cache layout.

return (InlineCacheGetterFlag | InlineCacheSetterFlag) << 1;

}

bool NeedsToBeRegisteredForProtoInvalidation() const;

bool NeedsToBeRegisteredForStoreFieldInvalidation() const;

#if DEBUG

bool ConfirmCacheMiss(const Type \* oldType, const PropertyValueInfo\* info) const;

bool NeedsToBeRegisteredForInvalidation() const;

static void VerifyRegistrationForInvalidation(const InlineCache\* cache, ScriptContext\* scriptContext, Js::PropertyId propertyId);

#endif

#if DBG\_DUMP

void Dump();

#endif

};

#if defined(\_M\_IX86\_OR\_ARM32)

CompileAssert(sizeof(InlineCache) == 0x10);

#else

CompileAssert(sizeof(InlineCache) == 0x20);

#endif

CompileAssert(sizeof(InlineCache) == sizeof(InlineCacheAllocator::CacheLayout));

CompileAssert(offsetof(InlineCache, invalidationListSlotPtr) == offsetof(InlineCacheAllocator::CacheLayout, strongRef));

struct JitTimePolymorphicInlineCache;

struct PolymorphicInlineCache sealed : public FinalizableObject

{

#ifdef INLINE\_CACHE\_STATS

friend class Js::ScriptContext;

#endif

public:

static const bool IsPolymorphic = true;

private:

InlineCache \* inlineCaches;

FunctionBody \* functionBody;

uint16 size;

bool ignoreForEquivalentObjTypeSpec;

bool cloneForJitTimeUse;

int32 inlineCachesFillInfo;

// DList chaining all polymorphic inline caches of a FunctionBody together.

// Since PolymorphicInlineCache is a leaf object, these references do not keep

// the polymorphic inline caches alive. When a PolymorphicInlineCache is finalized,

// it removes itself from the list and deletes its inline cache array.

PolymorphicInlineCache \* next;

PolymorphicInlineCache \* prev;

PolymorphicInlineCache(InlineCache \* inlineCaches, uint16 size, FunctionBody \* functionBody)

: inlineCaches(inlineCaches), functionBody(functionBody), size(size), ignoreForEquivalentObjTypeSpec(false), cloneForJitTimeUse(true), inlineCachesFillInfo(0), next(nullptr), prev(nullptr)

{

Assert((size == 0 && inlineCaches == nullptr) ||

(inlineCaches != nullptr && size >= MinPolymorphicInlineCacheSize && size <= MaxPolymorphicInlineCacheSize));

}

public:

static PolymorphicInlineCache \* New(uint16 size, FunctionBody \* functionBody);

static uint16 GetInitialSize() { return MinPolymorphicInlineCacheSize; }

bool CanAllocateBigger() { return GetSize() < MaxPolymorphicInlineCacheSize; }

static uint16 GetNextSize(uint16 currentSize)

{

if (currentSize == MaxPolymorphicInlineCacheSize)

{

return 0;

}

else

{

Assert(currentSize >= MinPolymorphicInlineCacheSize && currentSize <= (MaxPolymorphicInlineCacheSize / 2));

return currentSize \* 2;

}

}

template<bool isAccessor>

bool HasDifferentType(const bool isProto, const Type \* type, const Type \* typeWithoutProperty) const;

bool HasType\_Flags(const Type \* type) const;

InlineCache \* GetInlineCaches() const { return inlineCaches; }

uint16 GetSize() const { return size; }

PolymorphicInlineCache \* GetNext() { return next; }

bool GetIgnoreForEquivalentObjTypeSpec() const { return this->ignoreForEquivalentObjTypeSpec; }

void SetIgnoreForEquivalentObjTypeSpec(bool value) { this->ignoreForEquivalentObjTypeSpec = value; }

bool GetCloneForJitTimeUse() const { return this->cloneForJitTimeUse; }

void SetCloneForJitTimeUse(bool value) { this->cloneForJitTimeUse = value; }

uint32 GetInlineCachesFillInfo() { return this->inlineCachesFillInfo; }

void UpdateInlineCachesFillInfo(uint32 index, bool set);

bool IsFull();

virtual void Finalize(bool isShutdown) override;

virtual void Dispose(bool isShutdown) override { };

virtual void Mark(Recycler \*recycler) override { AssertMsg(false, "Mark called on object that isn't TrackableObject"); }

void CacheLocal(

Type \*const type,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const typeWithoutProperty,

int requiredAuxSlotCapacity,

ScriptContext \*const requestContext);

void CacheProto(

DynamicObject \*const prototypeObjectWithProperty,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

const bool isMissing,

Type \*const type,

ScriptContext \*const requestContext);

void CacheAccessor(

const bool isGetter,

const PropertyId propertyId,

const PropertyIndex propertyIndex,

const bool isInlineSlot,

Type \*const type,

DynamicObject \*const object,

const bool isOnProto,

ScriptContext \*const requestContext);

template<

bool CheckLocal,

bool CheckProto,

bool CheckAccessor,

bool CheckMissing,

bool IsInlineCacheAvailable,

bool ReturnOperationInfo>

bool TryGetProperty(

Var const instance,

RecyclableObject \*const propertyObject,

const PropertyId propertyId,

Var \*const propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo,

InlineCache \*const inlineCacheToPopulate);

template<

bool CheckLocal,

bool CheckLocalTypeWithoutProperty,

bool CheckAccessor,

bool ReturnOperationInfo,

bool PopulateInlineCache>

bool TrySetProperty(

RecyclableObject \*const object,

const PropertyId propertyId,

Var propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo,

InlineCache \*const inlineCacheToPopulate,

const PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

bool PretendTryGetProperty(Type \*const type, PropertyCacheOperationInfo \* operationInfo);

bool PretendTrySetProperty(Type \*const type, Type \*const oldType, PropertyCacheOperationInfo \* operationInfo);

void CopyTo(PropertyId propertyId, ScriptContext\* scriptContext, PolymorphicInlineCache \*const clone);

#if DBG\_DUMP

void Dump();

#endif

uint GetInlineCacheIndexForType(const Type \* type) const

{

return (((size\_t)type) >> PolymorphicInlineCacheShift) & (GetSize() - 1);

}

private:

uint GetNextInlineCacheIndex(uint index) const

{

if (++index == GetSize())

{

index = 0;

}

return index;

}

template<bool CheckLocal, bool CheckProto, bool CheckAccessor>

void CloneInlineCacheToEmptySlotInCollision(Type \*const type, uint index);

#ifdef CLONE\_INLINECACHE\_TO\_EMPTYSLOT

template <typename TDelegate>

bool CheckClonedInlineCache(uint inlineCacheIndex, TDelegate mapper);

#endif

#if INTRUSIVE\_TESTTRACE\_PolymorphicInlineCache

uint GetEntryCount()

{

uint count = 0;

for (uint i = 0; i < size; ++i)

{

if (!inlineCaches[i].IsEmpty())

{

count++;

}

}

return count;

}

#endif

};

class EquivalentTypeSet

{

private:

Type\*\* types;

uint16 count;

bool sortedAndDuplicatesRemoved;

public:

EquivalentTypeSet(Type\*\* types, uint16 count)

: types(types), count(count), sortedAndDuplicatesRemoved(false) {}

uint16 GetCount() const

{

return this->count;

}

Type\* GetFirstType() const

{

return GetType(0);

}

Type\* GetType(uint16 index) const

{

Assert(this->types != nullptr && this->count > 0 && index < this->count);

return this->types[index];

}

bool GetSortedAndDuplicatesRemoved() const

{

return this->sortedAndDuplicatesRemoved;

}

bool Contains(const Js::Type \* type, uint16 \* pIndex = nullptr) const;

static bool AreIdentical(EquivalentTypeSet \* left, EquivalentTypeSet \* right);

static bool IsSubsetOf(EquivalentTypeSet \* left, EquivalentTypeSet \* right);

void SortAndRemoveDuplicates();

};

enum class CtorCacheGuardValues : intptr\_t

{

TagFlag = 0x01,

Invalid = 0x00,

Special = TagFlag

};

ENUM\_CLASS\_HELPERS(CtorCacheGuardValues, intptr\_t);

#define MaxCachedSlotCount 65535

struct ConstructorCache

{

friend class JavascriptFunction;

typedef struct GuardStruct

{

CtorCacheGuardValues value;

};

typedef struct ContentStruct

{

DynamicType\* type;

ScriptContext\* scriptContext;

// In a pinch we could eliminate this and store type pending sharing in the type field as long

// as the guard value flags fit below the object alignment boundary. However, this wouldn't

// keep the type alive, so it would only work if we zeroed constructor caches before GC.

DynamicType\* pendingType;

// We cache only types whose slotCount < 64K to ensure the slotCount field doesn't look like a pointer to the recycler.

int slotCount;

// This layout (i.e. one-byte bit fields first, then the one-byte updateAfterCtor, and then the two byte inlineSlotCount) is

// chosen intentionally to make sure the whole four bytes never look like a pointer and create a false reference pinning something

// in recycler heap. The isPopulated bit is always set when the cache holds any data - even if it got invalidated.

bool isPopulated : 1;

bool isPolymorphic : 1;

bool typeUpdatePending : 1;

bool ctorHasNoExplicitReturnValue : 1;

bool skipDefaultNewObject : 1;

// This field indicates that the type stored in this cache is the final type after constructor.

bool typeIsFinal : 1;

// This field indicates that the constructor cache has been invalidated due to a constructor's prototype property change.

// We use this flag to determine if we should mark the cache as polymorphic and not attempt subsequent optimizations.

// The cache may also be invalidated due to a guard invalidation resulting from some property change (e.g. in proto chain),

// in which case we won't deem the cache polymorphic.

bool hasPrototypeChanged : 1;

uint8 callCount;

// Separate from the bit field below for convenient compare from the JIT-ed code. Doesn't currently increase the size.

// If size becomes an issue, we could merge back into the bit field and use a TEST instead of CMP.

bool updateAfterCtor;

int16 inlineSlotCount;

};

union

{

GuardStruct guard;

ContentStruct content;

};

CompileAssert(offsetof(GuardStruct, value) == offsetof(ContentStruct, type));

CompileAssert(sizeof(((GuardStruct\*)nullptr)->value) == sizeof(((ContentStruct\*)nullptr)->type));

CompileAssert(static\_cast<intptr\_t>(CtorCacheGuardValues::Invalid) == static\_cast<intptr\_t>(NULL));

static ConstructorCache DefaultInstance;

public:

ConstructorCache()

{

this->content.type = nullptr;

this->content.scriptContext = nullptr;

this->content.slotCount = 0;

this->content.inlineSlotCount = 0;

this->content.updateAfterCtor = false;

this->content.ctorHasNoExplicitReturnValue = false;

this->content.skipDefaultNewObject = false;

this->content.isPopulated = false;

this->content.isPolymorphic = false;

this->content.typeUpdatePending = false;

this->content.typeIsFinal = false;

this->content.hasPrototypeChanged = false;

this->content.callCount = 0;

Assert(IsConsistent());

}

ConstructorCache(ConstructorCache const \* other)

{

Assert(other != nullptr);

this->content.type = other->content.type;

this->content.scriptContext = other->content.scriptContext;

this->content.slotCount = other->content.slotCount;

this->content.inlineSlotCount = other->content.inlineSlotCount;

this->content.updateAfterCtor = other->content.updateAfterCtor;

this->content.ctorHasNoExplicitReturnValue = other->content.ctorHasNoExplicitReturnValue;

this->content.skipDefaultNewObject = other->content.skipDefaultNewObject;

this->content.isPopulated = other->content.isPopulated;

this->content.isPolymorphic = other->content.isPolymorphic;

this->content.typeUpdatePending = other->content.typeUpdatePending;

this->content.typeIsFinal = other->content.typeIsFinal;

this->content.hasPrototypeChanged = other->content.hasPrototypeChanged;

this->content.callCount = other->content.callCount;

Assert(IsConsistent());

}

static size\_t const GetOffsetOfGuardValue() { return offsetof(Js::ConstructorCache, guard.value); }

static size\_t const GetSizeOfGuardValue() { return sizeof(((Js::ConstructorCache\*)nullptr)->guard.value); }

void Populate(DynamicType\* type, ScriptContext\* scriptContext, bool ctorHasNoExplicitReturnValue, bool updateAfterCtor)

{

Assert(scriptContext == type->GetScriptContext());

Assert(type->GetIsShared());

Assert(IsConsistent());

Assert(!this->content.isPopulated || this->content.isPolymorphic);

Assert(type->GetTypeHandler()->GetSlotCapacity() <= MaxCachedSlotCount);

this->content.isPopulated = true;

this->content.type = type;

this->content.scriptContext = scriptContext;

this->content.slotCount = type->GetTypeHandler()->GetSlotCapacity();

this->content.inlineSlotCount = type->GetTypeHandler()->GetInlineSlotCapacity();

this->content.ctorHasNoExplicitReturnValue = ctorHasNoExplicitReturnValue;

this->content.updateAfterCtor = updateAfterCtor;

Assert(IsConsistent());

}

void PopulateForSkipDefaultNewObject(ScriptContext\* scriptContext)

{

Assert(IsConsistent());

Assert(!this->content.isPopulated);

this->content.isPopulated = true;

this->guard.value = CtorCacheGuardValues::Special;

this->content.scriptContext = scriptContext;

this->content.skipDefaultNewObject = true;

Assert(IsConsistent());

}

bool TryUpdateAfterConstructor(DynamicType\* type, ScriptContext\* scriptContext)

{

Assert(scriptContext == type->GetScriptContext());

Assert(type->GetTypeHandler()->GetMayBecomeShared());

Assert(IsConsistent());

Assert(this->content.isPopulated);

Assert(this->content.scriptContext == scriptContext);

Assert(!this->content.typeUpdatePending);

Assert(this->content.ctorHasNoExplicitReturnValue);

if (type->GetTypeHandler()->GetSlotCapacity() > MaxCachedSlotCount)

{

return false;

}

if (type->GetIsShared())

{

this->content.type = type;

this->content.typeIsFinal = true;

this->content.pendingType = nullptr;

}

else

{

AssertMsg(false, "No one calls this part of the code?");

this->guard.value = CtorCacheGuardValues::Special;

this->content.pendingType = type;

this->content.typeUpdatePending = true;

}

this->content.slotCount = type->GetTypeHandler()->GetSlotCapacity();

this->content.inlineSlotCount = type->GetTypeHandler()->GetInlineSlotCapacity();

Assert(IsConsistent());

return true;

}

void UpdateInlineSlotCount()

{

Assert(IsConsistent());

Assert(this->content.isPopulated);

Assert(IsEnabled() || NeedsTypeUpdate());

DynamicType\* type = this->content.typeUpdatePending ? this->content.pendingType : this->content.type;

DynamicTypeHandler\* typeHandler = type->GetTypeHandler();

// Inline slot capacity should never grow as a result of shrinking.

Assert(typeHandler->GetInlineSlotCapacity() <= this->content.inlineSlotCount);

// Slot capacity should never grow as a result of shrinking.

Assert(typeHandler->GetSlotCapacity() <= this->content.slotCount);

this->content.slotCount = typeHandler->GetSlotCapacity();

this->content.inlineSlotCount = typeHandler->GetInlineSlotCapacity();

Assert(IsConsistent());

}

void EnableAfterTypeUpdate()

{

Assert(IsConsistent());

Assert(this->content.isPopulated);

Assert(!IsEnabled());

Assert(this->guard.value == CtorCacheGuardValues::Special);

Assert(this->content.typeUpdatePending);

Assert(this->content.slotCount == this->content.pendingType->GetTypeHandler()->GetSlotCapacity());

Assert(this->content.inlineSlotCount == this->content.pendingType->GetTypeHandler()->GetInlineSlotCapacity());

Assert(this->content.pendingType->GetIsShared());

this->content.type = this->content.pendingType;

this->content.typeIsFinal = true;

this->content.pendingType = nullptr;

this->content.typeUpdatePending = false;

Assert(IsConsistent());

}

intptr\_t GetRawGuardValue() const

{

return static\_cast<intptr\_t>(this->guard.value);

}

DynamicType\* GetGuardValueAsType() const

{

return reinterpret\_cast<DynamicType\*>(this->guard.value & ~CtorCacheGuardValues::TagFlag);

}

DynamicType\* GetType() const

{

Assert(static\_cast<intptr\_t>(this->guard.value & CtorCacheGuardValues::TagFlag) == 0);

return this->content.type;

}

DynamicType\* GetPendingType() const

{

return this->content.pendingType;

}

ScriptContext\* GetScriptContext() const

{

return this->content.scriptContext;

}

int GetSlotCount() const

{

return this->content.slotCount;

}

int16 GetInlineSlotCount() const

{

return this->content.inlineSlotCount;

}

static bool IsDefault(const ConstructorCache\* constructorCache)

{

return constructorCache == &ConstructorCache::DefaultInstance;

}

bool IsDefault() const

{

return IsDefault(this);

}

bool IsPopulated() const

{

Assert(IsConsistent());

return this->content.isPopulated;

}

bool IsEmpty() const

{

Assert(IsConsistent());

return !this->content.isPopulated;

}

bool IsPolymorphic() const

{

Assert(IsConsistent());

return this->content.isPolymorphic;

}

bool GetSkipDefaultNewObject() const

{

return this->content.skipDefaultNewObject;

}

bool GetCtorHasNoExplicitReturnValue() const

{

return this->content.ctorHasNoExplicitReturnValue;

}

bool GetUpdateCacheAfterCtor() const

{

return this->content.updateAfterCtor;

}

bool GetTypeUpdatePending() const

{

return this->content.typeUpdatePending;

}

bool IsEnabled() const

{

return GetGuardValueAsType() != nullptr;

}

bool IsInvalidated() const

{

return this->guard.value == CtorCacheGuardValues::Invalid && this->content.isPopulated;

}

bool NeedsTypeUpdate() const

{

return this->guard.value == CtorCacheGuardValues::Special && this->content.typeUpdatePending;

}

uint8 CallCount() const

{

return content.callCount;

}

void IncCallCount()

{

++content.callCount;

Assert(content.callCount != 0);

}

bool NeedsUpdateAfterCtor() const

{

return this->content.updateAfterCtor;

}

bool IsNormal() const

{

return this->guard.value != CtorCacheGuardValues::Invalid && static\_cast<intptr\_t>(this->guard.value & CtorCacheGuardValues::TagFlag) == 0;

}

bool SkipDefaultNewObject() const

{

return this->guard.value == CtorCacheGuardValues::Special && this->content.skipDefaultNewObject;

}

bool IsSetUpForJit() const

{

return GetRawGuardValue() != NULL && !IsPolymorphic() && !NeedsUpdateAfterCtor() && (IsNormal() || SkipDefaultNewObject());

}

void ClearUpdateAfterCtor()

{

Assert(IsConsistent());

Assert(this->content.isPopulated);

Assert(this->content.updateAfterCtor);

this->content.updateAfterCtor = false;

Assert(IsConsistent());

}

static ConstructorCache\* EnsureValidInstance(ConstructorCache\* currentCache, ScriptContext\* scriptContext);

const void\* GetAddressOfGuardValue() const

{

return reinterpret\_cast<const void\*>(&this->guard.value);

}

static uint32 GetOffsetOfUpdateAfterCtor()

{

return offsetof(ConstructorCache, content.updateAfterCtor);

}

void InvalidateAsGuard()

{

Assert(!IsDefault(this));

this->guard.value = CtorCacheGuardValues::Invalid;

// Make sure we don't leak the types.

Assert(this->content.type == nullptr);

Assert(this->content.pendingType == nullptr);

Assert(IsInvalidated());

Assert(IsConsistent());

}

#if DBG

bool IsConsistent() const

{

return this->guard.value == CtorCacheGuardValues::Invalid ||

(this->content.isPopulated && (

(this->guard.value == CtorCacheGuardValues::Special && !this->content.updateAfterCtor && this->content.skipDefaultNewObject && !this->content.typeUpdatePending && this->content.slotCount == 0 && this->content.inlineSlotCount == 0 && this->content.pendingType == nullptr) ||

(this->guard.value == CtorCacheGuardValues::Special && !this->content.updateAfterCtor && this->content.typeUpdatePending && !this->content.skipDefaultNewObject && this->content.pendingType != nullptr) ||

((this->guard.value & CtorCacheGuardValues::TagFlag) == CtorCacheGuardValues::Invalid && !this->content.skipDefaultNewObject && !this->content.typeUpdatePending && this->content.pendingType == nullptr)));

}

#endif

#if DBG\_DUMP

void Dump() const;

#endif

private:

void InvalidateOnPrototypeChange();

};

// Caches the result of an instanceof operator over a type and a function

struct IsInstInlineCache

{

Type \* type; // The type of object operand an inline cache caches a result for

JavascriptFunction \* function; // The function operand an inline cache caches a result for

JavascriptBoolean \* result; // The result of doing (object instanceof function) where object->type == this->type

IsInstInlineCache \* next; // Used to link together caches that have the same function operand

public:

bool IsEmpty() const { return type == nullptr; }

bool TryGetResult(Var instance, JavascriptFunction \* function, JavascriptBoolean \*\* result);

void Cache(Type \* instanceType, JavascriptFunction \* function, JavascriptBoolean \* result, ScriptContext \* scriptContext);

private:

void Set(Type \* instanceType, JavascriptFunction \* function, JavascriptBoolean \* result);

void Clear();

void Unregister(ScriptContext \* scriptContext);

};

#if defined(\_M\_IX86\_OR\_ARM32)

CompileAssert(sizeof(IsInstInlineCache) == 0x10);

#else

CompileAssert(sizeof(IsInstInlineCache) == 0x20);

#endif

CompileAssert(sizeof(IsInstInlineCache) == sizeof(IsInstInlineCacheAllocator::CacheLayout));

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

template<

bool CheckLocal,

bool CheckProto,

bool CheckAccessor,

bool CheckMissing,

bool ReturnOperationInfo>

bool InlineCache::TryGetProperty(

Var const instance,

RecyclableObject \*const propertyObject,

const PropertyId propertyId,

Var \*const propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo)

{

CompileAssert(CheckLocal || CheckProto || CheckAccessor);

Assert(!ReturnOperationInfo || operationInfo);

CompileAssert(!ReturnOperationInfo || (CheckLocal && CheckProto && CheckAccessor));

Assert(instance);

Assert(propertyObject);

Assert(propertyId != Constants::NoProperty);

Assert(propertyValue);

Assert(requestContext);

DebugOnly(VerifyRegistrationForInvalidation(this, requestContext, propertyId));

Type \*const type = propertyObject->GetType();

if (CheckLocal && type == u.local.type)

{

Assert(propertyObject->GetScriptContext() == requestContext); // we never cache a type from another script context

\*propertyValue = DynamicObject::FromVar(propertyObject)->GetInlineSlot(u.local.slotIndex);

Assert(\*propertyValue == JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext) ||

\*propertyValue == JavascriptOperators::GetRootProperty(propertyObject, propertyId, requestContext));

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Local;

operationInfo->slotType = SlotType\_Inline;

}

return true;

}

if (CheckLocal && TypeWithAuxSlotTag(type) == u.local.type)

{

Assert(propertyObject->GetScriptContext() == requestContext); // we never cache a type from another script context

\*propertyValue = DynamicObject::FromVar(propertyObject)->GetAuxSlot(u.local.slotIndex);

Assert(\*propertyValue == JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext) ||

\*propertyValue == JavascriptOperators::GetRootProperty(propertyObject, propertyId, requestContext));

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Local;

operationInfo->slotType = SlotType\_Aux;

}

return true;

}

if (CheckProto && type == u.proto.type && !this->u.proto.isMissing)

{

Assert(u.proto.prototypeObject->GetScriptContext() == requestContext); // we never cache a type from another script context

\*propertyValue = u.proto.prototypeObject->GetInlineSlot(u.proto.slotIndex);

Assert(\*propertyValue == JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext) ||

\*propertyValue == JavascriptOperators::GetRootProperty(propertyObject, propertyId, requestContext));

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Proto;

operationInfo->slotType = SlotType\_Inline;

}

return true;

}

if (CheckProto && TypeWithAuxSlotTag(type) == u.proto.type && !this->u.proto.isMissing)

{

Assert(u.proto.prototypeObject->GetScriptContext() == requestContext); // we never cache a type from another script context

\*propertyValue = u.proto.prototypeObject->GetAuxSlot(u.proto.slotIndex);

// TODO: This assert often results in Assert(RootObjectBase::Is(object)) inside GetRootProperty, which is misleading

// when the problem is that GetProperty returned a different value than propertyValue. Consider reworking it here and elsewhere.

Assert(\*propertyValue == JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext) ||

\*propertyValue == JavascriptOperators::GetRootProperty(propertyObject, propertyId, requestContext));

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Proto;

operationInfo->slotType = SlotType\_Aux;

}

return true;

}

if (CheckAccessor && type == u.accessor.type)

{

Assert(propertyObject->GetScriptContext() == requestContext); // we never cache a type from another script context

Assert(u.accessor.flags & InlineCacheGetterFlag);

RecyclableObject \*const function = RecyclableObject::FromVar(u.accessor.object->GetInlineSlot(u.accessor.slotIndex));

\*propertyValue = JavascriptOperators::CallGetter(function, instance, requestContext);

// Can't assert because the getter could have a side effect

#ifdef CHKGETTER

Assert(JavascriptOperators::Equal(\*propertyValue, JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext), requestContext));

#endif

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Getter;

operationInfo->slotType = SlotType\_Inline;

}

return true;

}

if (CheckAccessor && TypeWithAuxSlotTag(type) == u.accessor.type)

{

Assert(propertyObject->GetScriptContext() == requestContext); // we never cache a type from another script context

Assert(u.accessor.flags & InlineCacheGetterFlag);

RecyclableObject \*const function = RecyclableObject::FromVar(u.accessor.object->GetAuxSlot(u.accessor.slotIndex));

\*propertyValue = JavascriptOperators::CallGetter(function, instance, requestContext);

// Can't assert because the getter could have a side effect

#ifdef CHKGETTER

Assert(JavascriptOperators::Equal(\*propertyValue, JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext), requestContext));

#endif

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Getter;

operationInfo->slotType = SlotType\_Aux;

}

return true;

}

if (CheckMissing && type == u.proto.type && this->u.proto.isMissing)

{

Assert(u.proto.prototypeObject->GetScriptContext() == requestContext); // we never cache a type from another script context

\*propertyValue = u.proto.prototypeObject->GetInlineSlot(u.proto.slotIndex);

// TODO: This assert often results in Assert(RootObjectBase::Is(object)) inside GetRootProperty, which is misleading

// when the problem is that GetProperty returned a different value than propertyValue. Consider reworking it here and elsewhere.

Assert(\*propertyValue == JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext) ||

\*propertyValue == JavascriptOperators::GetRootProperty(propertyObject, propertyId, requestContext));

#ifdef MISSING\_PROPERTY\_STATS

if (PHASE\_STATS1(MissingPropertyCachePhase))

{

requestContext->RecordMissingPropertyHit();

}

#endif

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Proto;

operationInfo->slotType = SlotType\_Inline;

}

return true;

}

if (CheckMissing && TypeWithAuxSlotTag(type) == u.proto.type && this->u.proto.isMissing)

{

Assert(u.proto.prototypeObject->GetScriptContext() == requestContext); // we never cache a type from another script context

\*propertyValue = u.proto.prototypeObject->GetAuxSlot(u.proto.slotIndex);

// TODO: This assert often results in Assert(RootObjectBase::Is(object)) inside GetRootProperty, which is misleading

// when the problem is that GetProperty returned a different value than propertyValue. Consider reworking it here and elsewhere.

Assert(\*propertyValue == JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext) ||

\*propertyValue == JavascriptOperators::GetRootProperty(propertyObject, propertyId, requestContext));

#ifdef MISSING\_PROPERTY\_STATS

if (PHASE\_STATS1(MissingPropertyCachePhase))

{

requestContext->RecordMissingPropertyHit();

}

#endif

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Proto;

operationInfo->slotType = SlotType\_Aux;

}

return true;

}

return false;

}

template<

bool CheckLocal,

bool CheckLocalTypeWithoutProperty,

bool CheckAccessor,

bool ReturnOperationInfo>

bool InlineCache::TrySetProperty(

RecyclableObject \*const object,

const PropertyId propertyId,

Var propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo,

const PropertyOperationFlags propertyOperationFlags)

{

CompileAssert(CheckLocal || CheckLocalTypeWithoutProperty || CheckAccessor);

Assert(!ReturnOperationInfo || operationInfo);

CompileAssert(!ReturnOperationInfo || (CheckLocal && CheckLocalTypeWithoutProperty && CheckAccessor));

Assert(object);

Assert(propertyId != Constants::NoProperty);

Assert(requestContext);

DebugOnly(VerifyRegistrationForInvalidation(this, requestContext, propertyId));

#if DBG

const bool isRoot = (propertyOperationFlags & PropertyOperation\_Root) != 0;

bool canSetField; // To verify if we can set a field on the object

Var setterValue = nullptr;

{

// We need to disable implicit call to ensure the check doesn't cause unwanted side effects in debug code

// Save old disableImplicitFlags and implicitCallFlags and disable implicit call and exception

ThreadContext \* threadContext = requestContext->GetThreadContext();

DisableImplicitFlags disableImplicitFlags = \*threadContext->GetAddressOfDisableImplicitFlags();

Js::ImplicitCallFlags implicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

\*threadContext->GetAddressOfDisableImplicitFlags() = DisableImplicitCallAndExceptionFlag;

DescriptorFlags flags = DescriptorFlags::None;

canSetField = !JavascriptOperators::CheckPrototypesForAccessorOrNonWritablePropertySlow(object, propertyId, &setterValue, &flags, isRoot, requestContext);

if (threadContext->GetImplicitCallFlags() != Js::ImplicitCall\_None)

{

canSetField = true; // If there was an implicit call, inconclusive. Disable debug check.

setterValue = nullptr;

}

else

if ((flags & Accessor) == Accessor)

{

Assert(setterValue != nullptr);

}

// Restore old disableImplicitFlags and implicitCallFlags

\*threadContext->GetAddressOfDisableImplicitFlags() = disableImplicitFlags;

threadContext->SetImplicitCallFlags(implicitCallFlags);

}

#endif

Type \*const type = object->GetType();

if (CheckLocal && type == u.local.type)

{

Assert(object->GetScriptContext() == requestContext); // we never cache a type from another script context

Assert(isRoot || object->GetPropertyIndex(propertyId) == DynamicObject::FromVar(object)->GetTypeHandler()->InlineOrAuxSlotIndexToPropertyIndex(u.local.slotIndex, true));

Assert(!isRoot || RootObjectBase::FromVar(object)->GetRootPropertyIndex(propertyId) == DynamicObject::FromVar(object)->GetTypeHandler()->InlineOrAuxSlotIndexToPropertyIndex(u.local.slotIndex, true));

Assert(object->CanStorePropertyValueDirectly(propertyId, isRoot));

DynamicObject::FromVar(object)->SetInlineSlot(SetSlotArgumentsRoot(propertyId, isRoot, u.local.slotIndex, propertyValue));

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Local;

operationInfo->slotType = SlotType\_Inline;

}

Assert(canSetField);

return true;

}

if (CheckLocal && TypeWithAuxSlotTag(type) == u.local.type)

{

Assert(object->GetScriptContext() == requestContext); // we never cache a type from another script context

Assert(isRoot || object->GetPropertyIndex(propertyId) == DynamicObject::FromVar(object)->GetTypeHandler()->InlineOrAuxSlotIndexToPropertyIndex(u.local.slotIndex, false));

Assert(!isRoot || RootObjectBase::FromVar(object)->GetRootPropertyIndex(propertyId) == DynamicObject::FromVar(object)->GetTypeHandler()->InlineOrAuxSlotIndexToPropertyIndex(u.local.slotIndex, false));

Assert(object->CanStorePropertyValueDirectly(propertyId, isRoot));

DynamicObject::FromVar(object)->SetAuxSlot(SetSlotArgumentsRoot(propertyId, isRoot, u.local.slotIndex, propertyValue));

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Local;

operationInfo->slotType = SlotType\_Aux;

}

Assert(canSetField);

return true;

}

if (CheckLocalTypeWithoutProperty && type == u.local.typeWithoutProperty)

{

// CAREFUL! CheckIfPrototypeChainHasOnlyWritableDataProperties may do allocation that triggers GC and

// clears this cache, so save any info that is needed from the cache before calling those functions.

Type \*const typeWithProperty = u.local.type;

const PropertyIndex propertyIndex = u.local.slotIndex;

#if DBG

uint16 newAuxSlotCapacity = u.local.requiredAuxSlotCapacity;

#endif

Assert(object->GetScriptContext() == requestContext); // we never cache a type from another script context

Assert(typeWithProperty);

Assert(DynamicType::Is(typeWithProperty->GetTypeId()));

Assert(((DynamicType\*)typeWithProperty)->GetIsShared());

Assert(((DynamicType\*)typeWithProperty)->GetTypeHandler()->IsPathTypeHandler());

AssertMsg(!((DynamicType\*)u.local.typeWithoutProperty)->GetTypeHandler()->GetIsPrototype(), "Why did we cache a property add for a prototype?");

Assert(((DynamicType\*)typeWithProperty)->GetTypeHandler()->CanStorePropertyValueDirectly((const DynamicObject\*)object, propertyId, isRoot));

DynamicObject \*const dynamicObject = DynamicObject::FromVar(object);

// If we're adding a property to an inlined slot, we should never need to adjust auxiliary slot array size.

Assert(newAuxSlotCapacity == 0);

dynamicObject->type = typeWithProperty;

Assert(isRoot || object->GetPropertyIndex(propertyId) == DynamicObject::FromVar(object)->GetTypeHandler()->InlineOrAuxSlotIndexToPropertyIndex(propertyIndex, true));

Assert(!isRoot || RootObjectBase::FromVar(object)->GetRootPropertyIndex(propertyId) == DynamicObject::FromVar(object)->GetTypeHandler()->InlineOrAuxSlotIndexToPropertyIndex(propertyIndex, true));

dynamicObject->SetInlineSlot(SetSlotArgumentsRoot(propertyId, isRoot, propertyIndex, propertyValue));

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_LocalWithoutProperty;

operationInfo->slotType = SlotType\_Inline;

}

Assert(canSetField);

return true;

}

if (CheckLocalTypeWithoutProperty && TypeWithAuxSlotTag(type) == u.local.typeWithoutProperty)

{

// CAREFUL! CheckIfPrototypeChainHasOnlyWritableDataProperties or AdjustSlots may do allocation that triggers GC and

// clears this cache, so save any info that is needed from the cache before calling those functions.

Type \*const typeWithProperty = TypeWithoutAuxSlotTag(u.local.type);

const PropertyIndex propertyIndex = u.local.slotIndex;

uint16 newAuxSlotCapacity = u.local.requiredAuxSlotCapacity;

Assert(object->GetScriptContext() == requestContext); // we never cache a type from another script context

Assert(typeWithProperty);

Assert(DynamicType::Is(typeWithProperty->GetTypeId()));

Assert(((DynamicType\*)typeWithProperty)->GetIsShared());

Assert(((DynamicType\*)typeWithProperty)->GetTypeHandler()->IsPathTypeHandler());

AssertMsg(!((DynamicType\*)TypeWithoutAuxSlotTag(u.local.typeWithoutProperty))->GetTypeHandler()->GetIsPrototype(), "Why did we cache a property add for a prototype?");

Assert(((DynamicType\*)typeWithProperty)->GetTypeHandler()->CanStorePropertyValueDirectly((const DynamicObject\*)object, propertyId, isRoot));

DynamicObject \*const dynamicObject = DynamicObject::FromVar(object);

if (newAuxSlotCapacity > 0)

{

DynamicTypeHandler::AdjustSlots(

dynamicObject,

static\_cast<DynamicType \*>(typeWithProperty)->GetTypeHandler()->GetInlineSlotCapacity(),

newAuxSlotCapacity);

}

dynamicObject->type = typeWithProperty;

Assert(isRoot || object->GetPropertyIndex(propertyId) == DynamicObject::FromVar(object)->GetTypeHandler()->InlineOrAuxSlotIndexToPropertyIndex(propertyIndex, false));

Assert(!isRoot || RootObjectBase::FromVar(object)->GetRootPropertyIndex(propertyId) == DynamicObject::FromVar(object)->GetTypeHandler()->InlineOrAuxSlotIndexToPropertyIndex(propertyIndex, false));

dynamicObject->SetAuxSlot(SetSlotArgumentsRoot(propertyId, isRoot, propertyIndex, propertyValue));

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_LocalWithoutProperty;

operationInfo->slotType = SlotType\_Aux;

}

Assert(canSetField);

return true;

}

if (CheckAccessor && type == u.accessor.type)

{

Assert(object->GetScriptContext() == requestContext); // we never cache a type from another script context

Assert(u.accessor.flags & InlineCacheSetterFlag);

RecyclableObject \*const function = RecyclableObject::FromVar(u.accessor.object->GetInlineSlot(u.accessor.slotIndex));

Assert(setterValue == nullptr || setterValue == function);

Js::JavascriptOperators::CallSetter(function, object, propertyValue, requestContext);

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Setter;

operationInfo->slotType = SlotType\_Inline;

}

return true;

}

if (CheckAccessor && TypeWithAuxSlotTag(type) == u.accessor.type)

{

Assert(object->GetScriptContext() == requestContext); // we never cache a type from another script context

Assert(u.accessor.flags & InlineCacheSetterFlag);

RecyclableObject \*const function = RecyclableObject::FromVar(u.accessor.object->GetAuxSlot(u.accessor.slotIndex));

Assert(setterValue == nullptr || setterValue == function);

Js::JavascriptOperators::CallSetter(function, object, propertyValue, requestContext);

if (ReturnOperationInfo)

{

operationInfo->cacheType = CacheType\_Setter;

operationInfo->slotType = SlotType\_Aux;

}

return true;

}

return false;

}

template<

bool CheckLocal,

bool CheckProto,

bool CheckAccessor>

void PolymorphicInlineCache::CloneInlineCacheToEmptySlotInCollision(Type \* const type, uint inlineCacheIndex)

{

if (CheckLocal && (inlineCaches[inlineCacheIndex].u.local.type == type || inlineCaches[inlineCacheIndex].u.local.type == TypeWithAuxSlotTag(type)))

{

return;

}

if (CheckProto && (inlineCaches[inlineCacheIndex].u.proto.type == type || inlineCaches[inlineCacheIndex].u.proto.type == TypeWithAuxSlotTag(type)))

{

return;

}

if (CheckAccessor && (inlineCaches[inlineCacheIndex].u.accessor.type == type || inlineCaches[inlineCacheIndex].u.accessor.type == TypeWithAuxSlotTag(type)))

{

return;

}

if (this->IsFull())

{

// If the cache is full, we won't find an empty slot to move the contents of the colliding inline cache to.

return;

}

// Collision is with a cache having a different type.

uint tryInlineCacheIndex = GetNextInlineCacheIndex(inlineCacheIndex);

// Iterate over the inline caches in the polymorphic cache, stop when:

// 1. an empty inline cache is found, or

// 2. a cache already populated with the incoming type is found, or

// 3. all the inline caches have been looked at.

while (!inlineCaches[tryInlineCacheIndex].IsEmpty() && tryInlineCacheIndex != inlineCacheIndex)

{

if (CheckLocal && (inlineCaches[tryInlineCacheIndex].u.local.type == type || inlineCaches[tryInlineCacheIndex].u.local.type == TypeWithAuxSlotTag(type)))

{

break;

}

if (CheckProto && (inlineCaches[tryInlineCacheIndex].u.proto.type == type || inlineCaches[tryInlineCacheIndex].u.proto.type == TypeWithAuxSlotTag(type)))

{

Assert(GetInlineCacheIndexForType(inlineCaches[tryInlineCacheIndex].u.proto.type) == inlineCacheIndex);

break;

}

if (CheckAccessor && (inlineCaches[tryInlineCacheIndex].u.accessor.type == type || inlineCaches[tryInlineCacheIndex].u.accessor.type == TypeWithAuxSlotTag(type)))

{

Assert(GetInlineCacheIndexForType(inlineCaches[tryInlineCacheIndex].u.accessor.type) == inlineCacheIndex);

break;

}

tryInlineCacheIndex = GetNextInlineCacheIndex(tryInlineCacheIndex);

}

if (tryInlineCacheIndex != inlineCacheIndex)

{

if (inlineCaches[inlineCacheIndex].invalidationListSlotPtr != nullptr)

{

Assert(\*(inlineCaches[inlineCacheIndex].invalidationListSlotPtr) == &inlineCaches[inlineCacheIndex]);

if (inlineCaches[tryInlineCacheIndex].invalidationListSlotPtr != nullptr)

{

Assert(\*(inlineCaches[tryInlineCacheIndex].invalidationListSlotPtr) == &inlineCaches[tryInlineCacheIndex]);

}

else

{

inlineCaches[tryInlineCacheIndex].invalidationListSlotPtr = inlineCaches[inlineCacheIndex].invalidationListSlotPtr;

\*(inlineCaches[tryInlineCacheIndex].invalidationListSlotPtr) = &inlineCaches[tryInlineCacheIndex];

inlineCaches[inlineCacheIndex].invalidationListSlotPtr = nullptr;

}

}

inlineCaches[tryInlineCacheIndex].u = inlineCaches[inlineCacheIndex].u;

UpdateInlineCachesFillInfo(tryInlineCacheIndex, true /\*set\*/);

// Let's clear the cache slot on which we had the collision. We might have stolen the invalidationListSlotPtr,

// so it may not pass VerifyRegistrationForInvalidation. Besides, it will be repopulated with the incoming data,

// and registered for invalidation, if necessary.

inlineCaches[inlineCacheIndex].Clear();

Assert((this->inlineCachesFillInfo & (1 << inlineCacheIndex)) != 0);

UpdateInlineCachesFillInfo(inlineCacheIndex, false /\*set\*/);

}

}

#ifdef CLONE\_INLINECACHE\_TO\_EMPTYSLOT

template <typename TDelegate>

bool PolymorphicInlineCache::CheckClonedInlineCache(uint inlineCacheIndex, TDelegate mapper)

{

bool success = false;

uint tryInlineCacheIndex = GetNextInlineCacheIndex(inlineCacheIndex);

do

{

if (inlineCaches[tryInlineCacheIndex].IsEmpty())

{

break;

}

success = mapper(tryInlineCacheIndex);

if (success)

{

Assert(inlineCaches[inlineCacheIndex].invalidationListSlotPtr == nullptr || \*inlineCaches[inlineCacheIndex].invalidationListSlotPtr == &inlineCaches[inlineCacheIndex]);

Assert(inlineCaches[tryInlineCacheIndex].invalidationListSlotPtr == nullptr || \*inlineCaches[tryInlineCacheIndex].invalidationListSlotPtr == &inlineCaches[tryInlineCacheIndex]);

// Swap inline caches, including their invalidationListSlotPtrs.

InlineCache temp = inlineCaches[tryInlineCacheIndex];

inlineCaches[tryInlineCacheIndex] = inlineCaches[inlineCacheIndex];

inlineCaches[inlineCacheIndex] = temp;

// Fix up invalidationListSlotPtrs to point to their owners.

if (inlineCaches[inlineCacheIndex].invalidationListSlotPtr != nullptr)

{

\*inlineCaches[inlineCacheIndex].invalidationListSlotPtr = &inlineCaches[inlineCacheIndex];

}

if (inlineCaches[tryInlineCacheIndex].invalidationListSlotPtr != nullptr)

{

\*inlineCaches[tryInlineCacheIndex].invalidationListSlotPtr = &inlineCaches[tryInlineCacheIndex];

}

break;

}

tryInlineCacheIndex = GetNextInlineCacheIndex(tryInlineCacheIndex);

} while (tryInlineCacheIndex != inlineCacheIndex);

return success;

}

#endif

template<

bool CheckLocal,

bool CheckProto,

bool CheckAccessor,

bool CheckMissing,

bool IsInlineCacheAvailable,

bool ReturnOperationInfo>

bool PolymorphicInlineCache::TryGetProperty(

Var const instance,

RecyclableObject \*const propertyObject,

const PropertyId propertyId,

Var \*const propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo,

InlineCache \*const inlineCacheToPopulate)

{

Assert(!IsInlineCacheAvailable || inlineCacheToPopulate);

Assert(!ReturnOperationInfo || operationInfo);

Type \* const type = propertyObject->GetType();

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

InlineCache \*cache = &inlineCaches[inlineCacheIndex];

#ifdef INLINE\_CACHE\_STATS

bool isEmpty = false;

if (PHASE\_STATS1(Js::PolymorphicInlineCachePhase))

{

isEmpty = cache->IsEmpty();

}

#endif

bool result = cache->TryGetProperty<CheckLocal, CheckProto, CheckAccessor, CheckMissing, ReturnOperationInfo>(

instance, propertyObject, propertyId, propertyValue, requestContext, operationInfo);

#ifdef CLONE\_INLINECACHE\_TO\_EMPTYSLOT

if (!result && !cache->IsEmpty())

{

result = CheckClonedInlineCache(inlineCacheIndex, [&](uint tryInlineCacheIndex) -> bool

{

cache = &inlineCaches[tryInlineCacheIndex];

return cache->TryGetProperty<CheckLocal, CheckProto, CheckAccessor, CheckMissing, ReturnOperationInfo>(

instance, propertyObject, propertyId, propertyValue, requestContext, operationInfo);

});

}

#endif

if (IsInlineCacheAvailable && result)

{

cache->CopyTo(propertyId, requestContext, inlineCacheToPopulate);

}

#ifdef INLINE\_CACHE\_STATS

if (PHASE\_STATS1(Js::PolymorphicInlineCachePhase))

{

bool collision = !result && !isEmpty;

this->functionBody->GetScriptContext()->LogCacheUsage(this, /\*isGet\*/ true, propertyId, result, collision);

}

#endif

return result;

}

template<

bool CheckLocal,

bool CheckLocalTypeWithoutProperty,

bool CheckAccessor,

bool IsInlineCacheAvailable,

bool ReturnOperationInfo>

bool PolymorphicInlineCache::TrySetProperty(

RecyclableObject \*const object,

const PropertyId propertyId,

Var propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo,

InlineCache \*const inlineCacheToPopulate,

const PropertyOperationFlags propertyOperationFlags)

{

Assert(!IsInlineCacheAvailable || inlineCacheToPopulate);

Assert(!ReturnOperationInfo || operationInfo);

Type \* const type = object->GetType();

uint inlineCacheIndex = GetInlineCacheIndexForType(type);

InlineCache \*cache = &inlineCaches[inlineCacheIndex];

#ifdef INLINE\_CACHE\_STATS

bool isEmpty = false;

if (PHASE\_STATS1(Js::PolymorphicInlineCachePhase))

{

isEmpty = cache->IsEmpty();

}

#endif

bool result = cache->TrySetProperty<CheckLocal, CheckLocalTypeWithoutProperty, CheckAccessor, ReturnOperationInfo>(

object, propertyId, propertyValue, requestContext, operationInfo, propertyOperationFlags);

#ifdef CLONE\_INLINECACHE\_TO\_EMPTYSLOT

if (!result && !cache->IsEmpty())

{

result = CheckClonedInlineCache(inlineCacheIndex, [&](uint tryInlineCacheIndex) -> bool

{

cache = &inlineCaches[tryInlineCacheIndex];

return cache->TrySetProperty<CheckLocal, CheckLocalTypeWithoutProperty, CheckAccessor, ReturnOperationInfo>(

object, propertyId, propertyValue, requestContext, operationInfo, propertyOperationFlags);

});

}

#endif

if (IsInlineCacheAvailable && result)

{

cache->CopyTo(propertyId, requestContext, inlineCacheToPopulate);

}

#ifdef INLINE\_CACHE\_STATS

if (PHASE\_STATS1(Js::PolymorphicInlineCachePhase))

{

bool collision = !result && !isEmpty;

this->functionBody->GetScriptContext()->LogCacheUsage(this, /\*isGet\*/ false, propertyId, result, collision);

}

#endif

return result;

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

template<class T>

class InlineCachePointerArray

{

private:

WriteBarrierPtr<T\*> inlineCaches;

#if DBG

uint inlineCacheCount;

#endif

public:

InlineCachePointerArray<T>();

private:

void EnsureInlineCaches(Recycler \*const recycler, FunctionBody \*const functionBody);

public:

T \*GetInlineCache(FunctionBody \*const functionBody, const uint index) const;

void SetInlineCache(

Recycler \*const recycler,

FunctionBody \*const functionBody,

const uint index,

T \*const inlineCache);

void Reset();

template <class Fn>

void Map(Fn fn, uint count) const

{

if (NULL != inlineCaches)

{

for (uint i =0; i < count; i++)

{

T\* inlineCache = inlineCaches[i];

if (inlineCache != NULL)

{

fn(inlineCache);

}

}

}

};

PREVENT\_COPY(InlineCachePointerArray)

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

namespace Js

{

template<class T>

InlineCachePointerArray<T>::InlineCachePointerArray()

: inlineCaches(nullptr)

#if DBG

, inlineCacheCount(0)

#endif

{

}

template<class T>

void InlineCachePointerArray<T>::EnsureInlineCaches(Recycler \*const recycler, FunctionBody \*const functionBody)

{

Assert(recycler);

Assert(functionBody);

Assert(functionBody->GetInlineCacheCount() != 0);

if(inlineCaches)

{

Assert(functionBody->GetInlineCacheCount() == inlineCacheCount);

return;

}

inlineCaches = RecyclerNewArrayZ(recycler, T \*, functionBody->GetInlineCacheCount());

#if DBG

inlineCacheCount = functionBody->GetInlineCacheCount();

#endif

}

template<class T>

T \*InlineCachePointerArray<T>::GetInlineCache(FunctionBody \*const functionBody, const uint index) const

{

Assert(functionBody);

Assert(!inlineCaches || functionBody->GetInlineCacheCount() == inlineCacheCount);

Assert(index < functionBody->GetInlineCacheCount());

return inlineCaches ? inlineCaches[index] : nullptr;

}

template<class T>

void InlineCachePointerArray<T>::SetInlineCache(

Recycler \*const recycler,

FunctionBody \*const functionBody,

const uint index,

T \*const inlineCache)

{

Assert(recycler);

Assert(functionBody);

Assert(!inlineCaches || functionBody->GetInlineCacheCount() == inlineCacheCount);

Assert(index < functionBody->GetInlineCacheCount());

Assert(inlineCache);

EnsureInlineCaches(recycler, functionBody);

inlineCaches[index] = inlineCache;

}

template<class T>

void InlineCachePointerArray<T>::Reset()

{

inlineCaches = nullptr;

#if DBG

inlineCacheCount = 0;

#endif

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

// Default all macro to nothing

#ifndef DEF2

#define DEF2(process, op, func)

#endif

#ifndef DEF3

#define DEF3(process, op, func, y)

#endif

#ifndef DEF2\_WMS

#define DEF2\_WMS(process, op, func)

#endif

#ifndef DEF3\_WMS

#define DEF3\_WMS(process, op, func, y)

#endif

#ifndef DEF4\_WMS

#define DEF4\_WMS(process, op, func, y, t)

#endif

#ifndef EXDEF2

#define EXDEF2(process, op, func)

#endif

#ifndef EXDEF3

#define EXDEF3(process, op, func, y)

#endif

#ifndef EXDEF2\_WMS

#define EXDEF2\_WMS(process, op, func)

#endif

#ifndef EXDEF3\_WMS

#define EXDEF3\_WMS(process, op, func, y)

#endif

#ifndef EXDEF4\_WMS

#define EXDEF4\_WMS(process, op, func, y, t)

#endif

#if defined(INTERPRETER\_ASMJS) && !defined(TEMP\_DISABLE\_ASMJS)

#include "InterpreterHandlerAsmJs.inl"

#else

DEF2 (FALLTHROUGH, EndSwitch, /\* Common case with Br \*/)

DEF2 (BR, Br, OP\_Br)

#ifdef BYTECODE\_BRANCH\_ISLAND

EXDEF2 (BRLONG, BrLong, OP\_Br)

#endif

DEF3 (CUSTOM, StartCall, OP\_StartCall, StartCall)

DEF2 (NOP, Nop, Empty)

DEF2\_WMS(NOP, Unused, Reg1)

DEF2\_WMS(IP\_TARG, ProfiledLoopStart, OP\_ProfiledLoopStart)

DEF2\_WMS(FALLTHROUGH, LoopBodyStart, /\* Common case with ProfiledLoopBodyStart \*/)

DEF2\_WMS(IP\_TARG, ProfiledLoopBodyStart, OP\_ProfiledLoopBodyStart)

DEF2\_WMS(IP\_TARG, ProfiledLoopEnd, OP\_ProfiledLoopEnd)

DEF2\_WMS(BRCMem, BrEq\_A, JavascriptOperators::Equal)

DEF2\_WMS(BRCMem, BrGt\_A, JavascriptOperators::Greater)

DEF2\_WMS(BRCMem, BrGe\_A, JavascriptOperators::GreaterEqual)

DEF2\_WMS(BRCMem, BrLt\_A, JavascriptOperators::Less)

DEF2\_WMS(BRCMem, BrLe\_A, JavascriptOperators::LessEqual)

DEF2\_WMS(BRCMem, BrNeq\_A, JavascriptOperators::NotEqual)

DEF2\_WMS(BRBMem\_ALLOW\_STACK, BrFalse\_A, OP\_BrFalse\_A)

DEF2\_WMS(BRBMem\_ALLOW\_STACK, BrTrue\_A, OP\_BrTrue\_A)

DEF2\_WMS(BRB\_ALLOW\_STACK, BrOnObject\_A, JavascriptOperators::IsObject)

DEF2\_WMS(BRB, BrNotNull\_A, OP\_BrNotNull\_A)

//Not emitted for byte code, keep it here for completeness

//EXDEF2\_WMS(BRB, BrUndecl\_A, OP\_BrUndecl\_A)

EXDEF2\_WMS(BRB, BrNotUndecl\_A, OP\_BrNotUndecl\_A)

DEF2\_WMS(FALLTHROUGH, Case, /\* Common case with BrSrEq\_A \*/)

DEF2\_WMS(BRCMem, BrSrEq\_A, JavascriptOperators::StrictEqual)

DEF2\_WMS(BRCMem, BrSrNeq\_A, JavascriptOperators::NotStrictEqual)

//Not emitted for byte code, keep it here for completeness

//DEF2 (BRS, BrHasSideEffects, JavascriptOperators::OP\_BrHasSideEffects)

DEF2 (BRS, BrNotHasSideEffects, JavascriptOperators::OP\_BrNotHasSideEffects)

EXDEF2 (BRPROP, BrOnHasProperty, OP\_BrOnHasProperty)

DEF2 (BRPROP, BrOnNoProperty, OP\_BrOnNoProperty)

DEF2 (BRLOCALPROP, BrOnNoLocalProperty, OP\_BrOnNoProperty)

DEF2 (BRENVPROP, BrOnNoEnvProperty, OP\_BrOnNoEnvProperty)

DEF2\_WMS(BRBS, BrFncNeqApply, JavascriptOperators::OP\_BrFncNeqApply)

//Not emitted for byte code, keep it here for completeness

//DEF2\_WMS(BRBS, BrFncEqApply, JavascriptOperators::OP\_BrFncEqApply)

DEF3\_WMS(CALL\_FLAGS\_None, CallI, OP\_CallI, CallI)

DEF3\_WMS(CALL\_FLAGS\_None, CallIExtended, OP\_CallIExtended, CallIExtended)

DEF3\_WMS(CALL\_FLAGS\_None, CallIExtendedFlags, OP\_CallIExtendedFlags, CallIExtendedFlags)

DEF3\_WMS(CALL\_FLAGS\_None, CallIFlags, OP\_CallIFlags, CallIFlags)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledCallI, PROFILEDOP(OP\_ProfiledCallI, OP\_CallI), ProfiledCallI)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledCallIExtended, PROFILEDOP(OP\_ProfiledCallIExtended, OP\_CallIExtended), ProfiledCallIExtended)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledCallIExtendedFlags, PROFILEDOP(OP\_ProfiledCallIExtendedFlags, OP\_CallIExtendedFlags), ProfiledCallIExtendedFlags)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledCallIWithICIndex, PROFILEDOP(OP\_ProfiledCallIWithICIndex, OP\_CallI), ProfiledCallIWithICIndex)

DEF3\_WMS(CALL\_FLAGS\_Value, ProfiledCallIExtendedWithICIndex, PROFILEDOP(OP\_ProfiledCallIExtendedWithICIndex, OP\_CallIExtended), ProfiledCallIExtendedWithICIndex)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledCallIExtendedFlagsWithICIndex, PROFILEDOP(OP\_ProfiledCallIExtendedFlagsWithICIndex, OP\_CallIExtendedFlags), ProfiledCallIExtendedFlagsWithICIndex)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledCallIFlagsWithICIndex, PROFILEDOP(OP\_ProfiledCallIFlags, OP\_CallIFlags), ProfiledCallIFlagsWithICIndex)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledCallIFlags, PROFILEDOP(OP\_ProfiledCallIFlags, OP\_CallIFlags), ProfiledCallIFlags)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledReturnTypeCallI, PROFILEDOP(OP\_ProfiledReturnTypeCallI, OP\_CallI), ProfiledCallI)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledReturnTypeCallIExtended, PROFILEDOP(OP\_ProfiledReturnTypeCallIExtended, OP\_CallIExtended), ProfiledCallIExtended)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledReturnTypeCallIExtendedFlags, PROFILEDOP(OP\_ProfiledReturnTypeCallIExtendedFlags, OP\_CallIExtendedFlags), ProfiledCallIExtendedFlags)

DEF3\_WMS(CALL\_FLAGS\_None, ProfiledReturnTypeCallIFlags, PROFILEDOP(OP\_ProfiledReturnTypeCallIFlags, OP\_CallIFlags), ProfiledCallIFlags)

EXDEF2\_WMS(A1toA1Mem, Conv\_Str, JavascriptConversion::ToString)

DEF2\_WMS(A1toA1Mem, Conv\_Obj, JavascriptOperators::ToObject)

EXDEF2\_WMS(A1toA1Mem, NewWithObject, JavascriptOperators::ToWithObject)

DEF2\_WMS(A1toA1Mem, Conv\_Num, JavascriptOperators::ToNumber)

DEF2\_WMS(A1toA1Mem, Incr\_A, JavascriptMath::Increment)

DEF2\_WMS(A1toA1Mem, Decr\_A, JavascriptMath::Decrement)

DEF2\_WMS(A1toA1Mem, Neg\_A, JavascriptMath::Negate)

DEF2\_WMS(A1toA1Mem, Not\_A, JavascriptMath::Not)

DEF2\_WMS(A1toA1Mem, Typeof, JavascriptOperators::Typeof)

DEF2\_WMS(A1toA1Mem, Delete\_A, JavascriptOperators::Delete)

DEF2\_WMS(GET\_ELEM\_IMem, TypeofElem, JavascriptOperators::TypeofElem)

DEF2\_WMS(A3toA1Mem, Concat3, JavascriptOperators::Concat3)

DEF2\_WMS(A2I1toA1Mem, NewConcatStrMulti, JavascriptOperators::NewConcatStrMulti)

DEF2\_WMS(A2I1toXXMem, SetConcatStrMultiItem, JavascriptOperators::SetConcatStrMultiItem)

DEF2\_WMS(A3I1toXXMem, SetConcatStrMultiItem2, JavascriptOperators::SetConcatStrMultiItem2)

DEF2\_WMS(A2toA1Mem, Add\_A, JavascriptMath::Add)

DEF2\_WMS(A2toA1Mem, Div\_A, JavascriptMath::Divide)

DEF2\_WMS(A2toA1MemProfiled, ProfiledDiv\_A, PROFILEDOP(ProfiledDivide<true>, ProfiledDivide<false>))

DEF2\_WMS(A2toA1Mem, Mul\_A, JavascriptMath::Multiply)

DEF2\_WMS(A2toA1Mem, Expo\_A, JavascriptMath::Exponentiation)

DEF2\_WMS(A2toA1Mem, Rem\_A, JavascriptMath::Modulus)

DEF2\_WMS(A2toA1MemProfiled, ProfiledRem\_A, PROFILEDOP(ProfileModulus<true>, ProfileModulus<false>))

DEF2\_WMS(A2toA1Mem, Sub\_A, JavascriptMath::Subtract)

DEF2\_WMS(A2toA1Mem, And\_A, JavascriptMath::And)

DEF2\_WMS(A2toA1Mem, Or\_A, JavascriptMath::Or)

DEF2\_WMS(A2toA1Mem, Xor\_A, JavascriptMath::Xor)

DEF2\_WMS(A2toA1Mem, Shl\_A, JavascriptMath::ShiftLeft)

DEF2\_WMS(A2toA1Mem, Shr\_A, JavascriptMath::ShiftRight)

DEF2\_WMS(A2toA1Mem, ShrU\_A, JavascriptMath::ShiftRightU)

DEF2\_WMS(CMMem, CmEq\_A, JavascriptOperators::Equal)

DEF2\_WMS(CMMem, CmGt\_A, JavascriptOperators::Greater)

DEF2\_WMS(CMMem, CmGe\_A, JavascriptOperators::GreaterEqual)

DEF2\_WMS(CMMem, CmLt\_A, JavascriptOperators::Less)

DEF2\_WMS(CMMem, CmLe\_A, JavascriptOperators::LessEqual)

DEF2\_WMS(CMMem, CmNeq\_A, JavascriptOperators::NotEqual)

DEF2\_WMS(CMMem, CmSrEq\_A, JavascriptOperators::StrictEqual)

DEF2\_WMS(CMMem, CmSrNeq\_A, JavascriptOperators::NotStrictEqual)

DEF2\_WMS(FALLTHROUGH, BeginSwitch, /\* Common case with Ld\_A \*/)

DEF2\_WMS(FALLTHROUGH, InitConst, /\* Common case with Ld\_A \*/)

DEF2\_WMS(A1toA1\_ALLOW\_STACK, Ld\_A, OP\_Ld\_A)

DEF2\_WMS(INNERtoA1, LdInnerScope, OP\_Ld\_A)

DEF2\_WMS(XXtoA1, LdLocalObj, OP\_LdLocalObj)

EXDEF2\_WMS(A1toA1\_ALLOW\_STACK, UnwrapWithObj, JavascriptOperators::OP\_UnwrapWithObj)

EXDEF2\_WMS(A2toXX, SetComputedNameVar, JavascriptOperators::OP\_SetComputedNameVar)

DEF2\_WMS(A1toXX\_ALLOW\_STACK, ChkUndecl, OP\_ChkUndecl)

DEF2\_WMS(XXtoA1, InitUndecl, OP\_InitUndecl)

DEF2\_WMS(ELEM\_RtU\_to\_XX, EnsureNoRootFld, OP\_EnsureNoRootProperty)

DEF2\_WMS(ELEM\_RtU\_to\_XX, EnsureNoRootRedeclFld, OP\_EnsureNoRootRedeclProperty)

DEF2\_WMS(ELEM\_C2\_to\_XX, ScopedEnsureNoRedeclFld, OP\_ScopedEnsureNoRedeclProperty)

DEF2\_WMS(A1toA1Profiled, ProfiledBeginSwitch, PROFILEDOP(ProfiledSwitch<true>, ProfiledSwitch<false>))

DEF2\_WMS(XXtoA1Mem, LdC\_A\_Null, JavascriptOperators::OP\_LdNull)

DEF2\_WMS(XXtoA1, ArgIn0, OP\_ArgIn0)

DEF2\_WMS(CUSTOM\_ArgNoSrc, ArgOut\_Env, OP\_ArgOut\_Env)

#if DBG

DEF2\_WMS(CUSTOM\_L\_Arg, ArgOut\_ANonVar, OP\_ArgOut\_ANonVar)

#else

DEF2\_WMS(FALLTHROUGH, ArgOut\_ANonVar, /\* Common case with ArgOUt\_A in fre build \*/)

#endif

DEF2\_WMS(CUSTOM\_L\_Arg, ArgOut\_A, OP\_ArgOut\_A)

DEF3\_WMS(CUSTOM\_L\_Arg2, ProfiledArgOut\_A, PROFILEDOP(OP\_ProfiledArgOut\_A, OP\_ArgOut\_A), ProfiledArg)

DEF3\_WMS(CUSTOM\_L\_Value, LdFld, OP\_GetProperty, ElementCP)

DEF3\_WMS(CUSTOM\_L\_Value, LdLocalFld, OP\_GetLocalProperty, ElementP)

EXDEF3\_WMS(CUSTOM\_L\_Value, LdSuperFld, OP\_GetSuperProperty, ElementC2)

DEF3\_WMS(CUSTOM\_L\_Value, LdFldForTypeOf, OP\_GetPropertyForTypeOf, ElementCP)

EXDEF3\_WMS(CUSTOM\_L\_Value, LdRootFldForTypeOf, OP\_GetRootPropertyForTypeOf, ElementRootCP)

DEF3\_WMS(CUSTOM\_L\_Value, LdFldForCallApplyTarget, OP\_GetProperty, ElementCP)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdFld, PROFILEDOP(OP\_ProfiledGetProperty, OP\_GetProperty), ElementCP)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdLocalFld, PROFILEDOP(OP\_ProfiledGetLocalProperty, OP\_GetLocalProperty), ElementP)

EXDEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdSuperFld, PROFILEDOP(OP\_ProfiledGetSuperProperty, OP\_GetSuperProperty), ElementC2)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdFldForTypeOf, PROFILEDOP(OP\_ProfiledGetPropertyForTypeOf, OP\_GetPropertyForTypeOf), ElementCP)

EXDEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdRootFldForTypeOf, PROFILEDOP(OP\_ProfiledGetRootPropertyForTypeOf, OP\_GetRootPropertyForTypeOf), ElementRootCP)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdFldForCallApplyTarget,PROFILEDOP(OP\_ProfiledGetPropertyCallApplyTarget, OP\_GetProperty), ElementCP)

DEF3\_WMS(CUSTOM\_L\_Value, LdRootFld, OP\_GetRootProperty, ElementRootCP)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdRootFld, PROFILEDOP(OP\_ProfiledGetRootProperty, OP\_GetRootProperty), ElementRootCP)

DEF3\_WMS(CUSTOM\_L\_Value, LdMethodFld, OP\_GetMethodProperty, ElementCP)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdMethodFld, PROFILEDOP(OP\_ProfiledGetMethodProperty, OP\_GetMethodProperty), ElementCP)

EXDEF3\_WMS(CUSTOM\_L\_Value, LdLocalMethodFld, OP\_GetLocalMethodProperty, ElementP)

EXDEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdLocalMethodFld, PROFILEDOP(OP\_ProfiledGetLocalMethodProperty, OP\_GetLocalMethodProperty), ElementP)

DEF3\_WMS(CUSTOM\_L\_Value, LdRootMethodFld, OP\_GetRootMethodProperty, ElementRootCP)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdRootMethodFld, PROFILEDOP(OP\_ProfiledGetRootMethodProperty, OP\_GetRootMethodProperty), ElementRootCP)

DEF3\_WMS(CUSTOM\_L\_Value, DeleteFld, OP\_DeleteFld, ElementC)

EXDEF3\_WMS(CUSTOM\_L\_Value, DeleteLocalFld, OP\_DeleteLocalFld, ElementU)

DEF3\_WMS(CUSTOM\_L\_Value, DeleteRootFld, OP\_DeleteRootFld, ElementC)

DEF3\_WMS(CUSTOM\_L\_Value, DeleteFldStrict, OP\_DeleteFldStrict, ElementC)

DEF3\_WMS(CUSTOM\_L\_Value, DeleteRootFldStrict, OP\_DeleteRootFldStrict, ElementC)

DEF3\_WMS(CUSTOM, StFld, OP\_SetProperty, ElementCP)

DEF3\_WMS(CUSTOM, StLocalFld, OP\_SetLocalProperty, ElementP)

EXDEF3\_WMS(CUSTOM\_L\_Value, StSuperFld, OP\_SetSuperProperty, ElementC2)

DEF3\_WMS(CUSTOM, ProfiledStFld, PROFILEDOP(OP\_ProfiledSetProperty, OP\_SetProperty), ElementCP)

DEF3\_WMS(CUSTOM, ProfiledStLocalFld, PROFILEDOP(OP\_ProfiledSetLocalProperty, OP\_SetLocalProperty), ElementP)

EXDEF3\_WMS(CUSTOM\_L\_Value, ProfiledStSuperFld, PROFILEDOP(OP\_ProfiledSetSuperProperty, OP\_SetSuperProperty), ElementC2)

DEF3\_WMS(CUSTOM, StRootFld, OP\_SetRootProperty, ElementRootCP)

DEF3\_WMS(CUSTOM, ProfiledStRootFld, PROFILEDOP(OP\_ProfiledSetRootProperty, OP\_SetRootProperty), ElementRootCP)

DEF3\_WMS(CUSTOM, StFldStrict, OP\_SetPropertyStrict, ElementCP)

DEF3\_WMS(CUSTOM, ProfiledStFldStrict, PROFILEDOP(OP\_ProfiledSetPropertyStrict, OP\_SetPropertyStrict), ElementCP)

DEF3\_WMS(CUSTOM, StRootFldStrict, OP\_SetRootPropertyStrict, ElementRootCP)

DEF3\_WMS(CUSTOM, ProfiledStRootFldStrict, PROFILEDOP(OP\_ProfiledSetRootPropertyStrict, OP\_SetRootPropertyStrict), ElementRootCP)

DEF3\_WMS(CUSTOM, InitFld, OP\_InitProperty, ElementCP)

DEF3\_WMS(CUSTOM, ProfiledInitFld, PROFILEDOP(OP\_ProfiledInitProperty, OP\_InitProperty), ElementCP)

DEF3\_WMS(CUSTOM, InitLocalFld, OP\_InitLocalProperty, ElementP)

DEF3\_WMS(CUSTOM, ProfiledInitLocalFld, PROFILEDOP(OP\_ProfiledInitLocalProperty, OP\_InitLocalProperty), ElementP)

DEF3\_WMS(CUSTOM, InitRootFld, OP\_InitRootProperty, ElementRootCP)

DEF3\_WMS(CUSTOM, ProfiledInitRootFld, PROFILEDOP(OP\_ProfiledInitRootProperty, OP\_InitRootProperty), ElementRootCP)

DEF3\_WMS(CUSTOM, InitUndeclLetFld, OP\_InitUndeclLetProperty, ElementPIndexed)

EXDEF3\_WMS(CUSTOM, InitUndeclLocalLetFld, OP\_InitUndeclLocalLetProperty, ElementP)

DEF3\_WMS(CUSTOM, InitUndeclConstFld, OP\_InitUndeclConstProperty, ElementPIndexed)

EXDEF3\_WMS(CUSTOM, InitUndeclLocalConstFld, OP\_InitUndeclLocalConstProperty, ElementP)

DEF3\_WMS(CUSTOM, InitLetFld, OP\_InitLetFld, ElementCP)

EXDEF3\_WMS(CUSTOM, InitLocalLetFld, OP\_InitLocalLetFld, ElementP)

EXDEF3\_WMS(CUSTOM, InitInnerFld, OP\_InitInnerFld, ElementPIndexed)

EXDEF3\_WMS(CUSTOM, InitInnerLetFld, OP\_InitInnerLetFld, ElementPIndexed)

DEF3\_WMS(CUSTOM, InitRootLetFld, OP\_InitRootLetFld, ElementRootCP)

DEF3\_WMS(CUSTOM, InitConstFld, OP\_InitConstFld, ElementCP)

DEF3\_WMS(CUSTOM, InitRootConstFld, OP\_InitRootConstFld, ElementRootCP)

DEF2\_WMS(ELEM\_RtU\_to\_XX, InitUndeclRootLetFld, OP\_InitUndeclRootLetProperty)

DEF2\_WMS(ELEM\_RtU\_to\_XX, InitUndeclRootConstFld, OP\_InitUndeclRootConstProperty)

EXDEF3\_WMS(CUSTOM, InitUndeclConsoleLetFld, OP\_InitUndeclConsoleLetProperty, ElementScopedU)

EXDEF3\_WMS(CUSTOM, InitUndeclConsoleConstFld, OP\_InitUndeclConsoleConstProperty, ElementScopedU)

DEF3\_WMS(CUSTOM, InitClassMember, OP\_InitClassMember, ElementCP)

EXDEF3\_WMS(CUSTOM, InitClassMemberComputedName,OP\_InitClassMemberComputedName, ElementI)

EXDEF3\_WMS(CUSTOM, InitClassMemberSet, OP\_InitClassMemberSet, ElementC)

EXDEF3\_WMS(CUSTOM, InitClassMemberGetComputedName, OP\_InitClassMemberGetComputedName, ElementI)

EXDEF3\_WMS(CUSTOM, InitClassMemberGet, OP\_InitClassMemberGet, ElementC)

EXDEF3\_WMS(CUSTOM, InitClassMemberSetComputedName, OP\_InitClassMemberSetComputedName, ElementI)

EXDEF2\_WMS(BRB, BrOnClassConstructor, OP\_BrOnClassConstructor)

DEF3\_WMS(GET\_ELEM\_LOCALSLOTNonVar,LdLocalSlot, OP\_LdSlot, ElementSlotI1)

DEF3\_WMS(GET\_ELEM\_INNERSLOTNonVar,LdInnerSlot, OP\_LdInnerSlot, ElementSlotI2)

EXDEF3\_WMS(GET\_ELEM\_INNERSLOTNonVar,LdInnerObjSlot, OP\_LdInnerObjSlot, ElementSlotI2)

DEF3\_WMS(GET\_ELEM\_ENVSLOTNonVar, LdEnvSlot, OP\_LdEnvSlot, ElementSlotI2)

DEF3\_WMS(GET\_ELEM\_ENVSLOTNonVar, LdEnvObj, OP\_LdEnvObj, ElementSlotI1)

EXDEF3\_WMS(GET\_ELEM\_ENVSLOTNonVar, LdEnvObjSlot, OP\_LdEnvObjSlot, ElementSlotI2)

DEF3\_WMS(GET\_ELEM\_SLOTNonVar, ProfiledLdSlot, PROFILEDOP(OP\_ProfiledLdSlot, OP\_LdSlot), ProfiledElementSlot)

DEF3\_WMS(GET\_ELEM\_INNERSLOTNonVar,ProfiledLdInnerSlot, PROFILEDOP(OP\_ProfiledLdInnerSlot, OP\_LdInnerSlot), ProfiledElementSlotI2)

EXDEF3\_WMS(GET\_ELEM\_INNERSLOTNonVar,ProfiledLdInnerObjSlot, PROFILEDOP(OP\_ProfiledLdInnerObjSlot, OP\_LdInnerObjSlot), ProfiledElementSlotI2)

DEF3\_WMS(GET\_ELEM\_LOCALSLOTNonVar,ProfiledLdLocalSlot, PROFILEDOP(OP\_ProfiledLdSlot, OP\_LdSlot), ProfiledElementSlotI1)

DEF3\_WMS(GET\_ELEM\_ENVSLOTNonVar, ProfiledLdEnvSlot, PROFILEDOP(OP\_ProfiledLdEnvSlot, OP\_LdEnvSlot), ProfiledElementSlotI2)

EXDEF3\_WMS(GET\_ELEM\_ENVSLOTNonVar, ProfiledLdEnvObjSlot, PROFILEDOP(OP\_ProfiledLdEnvObjSlot, OP\_LdEnvObjSlot), ProfiledElementSlotI2)

EXDEF3\_WMS(GET\_ELEM\_SLOTNonVar, LdObjSlot, OP\_LdObjSlot, ElementSlot)

EXDEF3\_WMS(GET\_ELEM\_SLOTNonVar, ProfiledLdObjSlot, PROFILEDOP(OP\_ProfiledLdObjSlot, OP\_LdObjSlot), ProfiledElementSlot)

EXDEF3\_WMS(GET\_ELEM\_LOCALSLOTNonVar,LdLocalObjSlot, OP\_LdObjSlot, ElementSlotI1)

EXDEF3\_WMS(GET\_ELEM\_LOCALSLOTNonVar,ProfiledLdLocalObjSlot, PROFILEDOP(OP\_ProfiledLdObjSlot, OP\_LdObjSlot), ProfiledElementSlotI1)

DEF2\_WMS(SET\_ELEM\_LOCALSLOTNonVar,StLocalSlot, OP\_StSlot)

EXDEF2\_WMS(SET\_ELEM\_LOCALSLOTNonVar,StLocalSlotChkUndecl, OP\_StSlotChkUndecl)

DEF2\_WMS(SET\_ELEM\_INNERSLOTNonVar,StInnerSlot, OP\_StSlot)

EXDEF2\_WMS(SET\_ELEM\_INNERSLOTNonVar,StInnerObjSlot, OP\_StObjSlot)

DEF2\_WMS(SET\_ELEM\_ENVSLOTNonVar, StEnvSlot, OP\_StEnvSlot)

EXDEF2\_WMS(SET\_ELEM\_ENVSLOTNonVar, StEnvSlotChkUndecl, OP\_StEnvSlotChkUndecl)

EXDEF2\_WMS(SET\_ELEM\_SLOTNonVar, StObjSlot, OP\_StObjSlot)

EXDEF2\_WMS(SET\_ELEM\_LOCALSLOTNonVar,StLocalObjSlot, OP\_StObjSlot)

EXDEF2\_WMS(SET\_ELEM\_LOCALSLOTNonVar,StLocalObjSlotChkUndecl, OP\_StObjSlotChkUndecl)

EXDEF2\_WMS(SET\_ELEM\_ENVSLOTNonVar, StEnvObjSlot, OP\_StEnvObjSlot)

EXDEF2\_WMS(SET\_ELEM\_SLOTNonVar, StObjSlotChkUndecl, OP\_StObjSlotChkUndecl)

EXDEF2\_WMS(SET\_ELEM\_ENVSLOTNonVar, StEnvObjSlotChkUndecl, OP\_StEnvObjSlotChkUndecl)

DEF3\_WMS(CUSTOM\_L\_Value, LdElemI\_A, OP\_GetElementI, ElementI)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledLdElemI\_A, PROFILEDOP(OP\_ProfiledGetElementI, OP\_GetElementI), ProfiledElementI)

DEF2\_WMS(GET\_ELEM\_IMem, LdMethodElem, JavascriptOperators::OP\_GetMethodElement)

DEF3\_WMS(CUSTOM, StElemI\_A, OP\_SetElementI, ElementI)

DEF3\_WMS(CUSTOM, StElemI\_A\_Strict, OP\_SetElementIStrict, ElementI)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledStElemI\_A, PROFILEDOP(OP\_ProfiledSetElementI, OP\_SetElementI), ProfiledElementI)

DEF3\_WMS(CUSTOM\_L\_Value, ProfiledStElemI\_A\_Strict, PROFILEDOP(OP\_ProfiledSetElementIStrict, OP\_SetElementIStrict), ProfiledElementI)

DEF3\_WMS(CUSTOM, StArrItemI\_CI4, OP\_SetArrayItemI\_CI4, ElementUnsigned1)

DEF2\_WMS(FALLTHROUGH, StArrInlineItem\_CI4, /\*Common case with StArrItemC\_CI4 \*/)

DEF3\_WMS(CUSTOM, StArrItemC\_CI4, OP\_SetArrayItemC\_CI4, ElementUnsigned1)

DEF3\_WMS(CUSTOM\_L\_R0, LdArrHead, OP\_LdArrayHeadSegment, Reg2)

DEF3\_WMS(CUSTOM, StArrSegItem\_CI4, OP\_SetArraySegmentItem\_CI4, ElementUnsigned1)

DEF3 (CUSTOM, StArrSegItem\_A, OP\_SetArraySegmentVars, Auxiliary)

DEF3\_WMS(CALL, NewScObject, OP\_NewScObject, CallI)

DEF3\_WMS(CUSTOM\_L\_R0, NewScObjectNoCtorFull, OP\_NewScObjectNoCtorFull, Reg2)

EXDEF2\_WMS(A1toA1Mem, LdCustomSpreadIteratorList, JavascriptOperators::OP\_LdCustomSpreadIteratorList)

EXDEF3\_WMS(CALL, NewScObjectSpread, OP\_NewScObjectSpread, CallIExtended)

DEF3\_WMS(CALL, NewScObjArray, OP\_NewScObjArray, CallI)

DEF3\_WMS(CALL, NewScObjArraySpread, OP\_NewScObjArraySpread, CallIExtended)

DEF3\_WMS(CALL, ProfiledNewScObject, PROFILEDOP(OP\_ProfiledNewScObject, OP\_NewScObject), ProfiledCallI)

EXDEF3\_WMS(CALL, ProfiledNewScObjectSpread, PROFILEDOP(OP\_ProfiledNewScObjectSpread, OP\_NewScObjectSpread), ProfiledCallIExtended)

DEF3\_WMS(CALL, ProfiledNewScObjectWithICIndex, PROFILEDOP(OP\_ProfiledNewScObjectWithICIndex, OP\_NewScObject), ProfiledCallIWithICIndex)

DEF3\_WMS(CALL, ProfiledNewScObjArray, PROFILEDOP(OP\_ProfiledNewScObjArray, OP\_ProfiledNewScObjArray\_NoProfile), Profiled2CallI)

DEF3\_WMS(CALL, ProfiledNewScObjArraySpread,PROFILEDOP(OP\_ProfiledNewScObjArraySpread, OP\_ProfiledNewScObjArraySpread\_NoProfile), Profiled2CallIExtended)

DEF2\_WMS(XXtoA1NonVar, LdArgCnt, OP\_LdArgCnt)

DEF3\_WMS(CUSTOM\_L\_R0, LdLen\_A, OP\_LdLen, Reg2)

DEF3\_WMS(CUSTOM\_L\_R0, ProfiledLdLen\_A, PROFILEDOP(OP\_ProfiledLdLen, OP\_LdLen), ProfiledReg2)

DEF2\_WMS(XXtoA1Mem, LdUndef, JavascriptOperators::OP\_LdUndef)

DEF2\_WMS(XXtoA1Mem, LdNaN, JavascriptOperators::OP\_LdNaN)

DEF2\_WMS(XXtoA1Mem, LdInfinity, JavascriptOperators::OP\_LdInfinity)

DEF2\_WMS(XXtoA1Mem, LdTrue, JavascriptBoolean::OP\_LdTrue)

DEF2\_WMS(XXtoA1Mem, LdFalse, JavascriptBoolean::OP\_LdFalse)

DEF2\_WMS(A1I1toA1Mem, LdThis, JavascriptOperators::OP\_GetThisNoFastPath)

EXDEF2\_WMS(XXtoA1Mem, LdSuper, OP\_LdSuper)

EXDEF2\_WMS(XXtoA1Mem, LdSuperCtor, OP\_LdSuperCtor)

EXDEF2\_WMS(XXtoA1Mem, ScopedLdSuper, OP\_ScopedLdSuper)

EXDEF2\_WMS(XXtoA1Mem, ScopedLdSuperCtor, OP\_ScopedLdSuperCtor)

EXDEF2\_WMS(A2toXX, SetHomeObj, JavascriptOperators::OP\_SetHomeObj)

DEF2\_WMS(A1toA1Mem, StrictLdThis, JavascriptOperators::OP\_StrictGetThis)

DEF2\_WMS(A1I1toA1Mem, ProfiledLdThis, PROFILEDOP(OP\_ProfiledLdThis, JavascriptOperators::OP\_GetThisNoFastPath))

DEF2\_WMS(A1toA1Mem, ProfiledStrictLdThis, PROFILEDOP(OP\_ProfiledStrictLdThis, JavascriptOperators::OP\_StrictGetThis))

DEF2\_WMS(XXtoA1Mem, LdHeapArgsCached, OP\_LdHeapArgsCached)

EXDEF2\_WMS(XXtoA1Mem, LdLetHeapArgsCached, OP\_LdLetHeapArgsCached)

EXDEF2\_WMS(XXtoA1NonVar, LdStackArgPtr, OP\_LdStackArgPtr)

EXDEF3\_WMS(CUSTOM, InitSetFld, OP\_InitSetFld, ElementC)

EXDEF3\_WMS(CUSTOM, InitGetFld, OP\_InitGetFld, ElementC)

EXDEF3\_WMS(CUSTOM, InitSetElemI, OP\_InitSetElemI, ElementI)

EXDEF3\_WMS(CUSTOM, InitGetElemI, OP\_InitGetElemI, ElementI)

EXDEF3\_WMS(CUSTOM, InitComputedProperty, OP\_InitComputedProperty, ElementI)

EXDEF3\_WMS(CUSTOM, InitProto, OP\_InitProto, ElementC)

DEF3\_WMS(CUSTOM, LdElemUndefScoped, OP\_LdElementUndefinedScoped, ElementScopedU)

DEF3\_WMS(CUSTOM\_L\_R0, LdFuncExpr, OP\_LdFunctionExpression, Reg1)

DEF3\_WMS(CUSTOM, StFuncExpr, OP\_StFunctionExpression, ElementC)

DEF3\_WMS(CUSTOM, StLocalFuncExpr, OP\_StLocalFunctionExpression, ElementU)

EXDEF3\_WMS(CUSTOM\_L\_R0, LdNewTarget, OP\_LdNewTarget, Reg1)

EXDEF2 (EMPTY, ChkNewCallFlag, OP\_ChkNewCallFlag)

DEF2\_WMS(U1toINNERMemNonVar, NewBlockScope, JavascriptOperators::OP\_NewBlockScope)

DEF3\_WMS(CUSTOM, CloneBlockScope, OP\_CloneBlockScope, Unsigned1)

DEF2\_WMS(U1toINNERMemNonVar, NewPseudoScope, JavascriptOperators::OP\_NewPseudoScope)

DEF3\_WMS(CUSTOM\_L\_Value, NewStackScFunc, OP\_NewStackScFunc, ElementSlotI1)

DEF2\_WMS(GET\_SLOT\_FB, NewScFunc, ScriptFunction::OP\_NewScFunc)

DEF2\_WMS(GET\_SLOT\_FB, NewScGenFunc, JavascriptGeneratorFunction::OP\_NewScGenFunc)

EXDEF3\_WMS(CUSTOM\_L\_Value, NewInnerStackScFunc, OP\_NewInnerStackScFunc, ElementSlot)

EXDEF2\_WMS(GET\_ELEM\_SLOT\_FB, NewInnerScFunc, ScriptFunction::OP\_NewScFunc)

EXDEF2\_WMS(GET\_ELEM\_SLOT\_FB, NewInnerScGenFunc, JavascriptGeneratorFunction::OP\_NewScGenFunc)

DEF2\_WMS(A1toA1MemNonVar, GetForInEnumerator, JavascriptOperators::OP\_GetForInEnumerator)

DEF3\_WMS(A1toXXMemNonVar, ReleaseForInEnumerator, JavascriptOperators::OP\_ReleaseForInEnumerator, ForInObjectEnumerator \*)

DEF2\_WMS(A1toXXMem, Throw, JavascriptExceptionOperators::OP\_Throw)

DEF2\_WMS(XXtoA1NonVar, LdArgumentsFromFrame, OP\_LdArgumentsFromFrame)

DEF2\_WMS(A1toA1MemNonVar, LdHeapArguments, OP\_LdHeapArguments)

DEF2\_WMS(A1toA1MemNonVar, LdLetHeapArguments, OP\_LdLetHeapArguments)

DEF2\_WMS(A2toA1MemNonVar, LdInnerFrameDisplay, OP\_LdInnerFrameDisplay)

DEF2\_WMS(A1toA1MemNonVar, LdInnerFrameDisplayNoParent,OP\_LdInnerFrameDisplayNoParent)

DEF2\_WMS(A1INNERtoA1MemNonVar, LdIndexedFrameDisplay, OP\_LdFrameDisplay)

DEF2\_WMS(XXINNERtoA1MemNonVar, LdIndexedFrameDisplayNoParent,OP\_LdFrameDisplayNoParent<true>)

DEF2\_WMS(A2toXXMemNonVar, LdFuncExprFrameDisplay, OP\_LdFuncExprFrameDisplaySetLocal)

DEF3\_WMS(CUSTOM\_L\_R0, IsInst, OP\_IsInst, Reg3C)

DEF2\_WMS(A2toA1Mem, IsIn, JavascriptOperators::IsIn)

DEF3\_WMS(CUSTOM\_L\_Value, ScopedLdFld, OP\_GetPropertyScoped, ElementP)

EXDEF3\_WMS(CUSTOM\_L\_Value, ScopedLdFldForTypeOf, OP\_GetPropertyForTypeOfScoped, ElementP)

DEF3\_WMS(CUSTOM\_L\_Value, ScopedLdMethodFld, OP\_GetMethodPropertyScoped, ElementCP)

DEF3\_WMS(CUSTOM, ScopedStFld, OP\_SetPropertyScoped, ElementP)

EXDEF3\_WMS(CUSTOM, ConsoleScopedStFld, OP\_ConsoleSetPropertyScoped, ElementP)

DEF3\_WMS(CUSTOM, ScopedStFldStrict, OP\_SetPropertyScopedStrict, ElementP)

DEF2\_WMS(GET\_ELEM\_IMem, DeleteElemI\_A, JavascriptOperators::OP\_DeleteElementI)

DEF2\_WMS(GET\_ELEM\_IMem\_Strict, DeleteElemIStrict\_A, JavascriptOperators::OP\_DeleteElementI)

DEF3\_WMS(CUSTOM\_L\_Value, ScopedLdInst, OP\_ScopedLdInst, ElementScopedC2)

DEF3\_WMS(CUSTOM, ScopedInitFunc, OP\_ScopedInitFunc, ElementScopedC)

DEF3\_WMS(CUSTOM\_L\_Value, ScopedDeleteFld, OP\_ScopedDeleteFld, ElementScopedC)

DEF3\_WMS(CUSTOM\_L\_Value, ScopedDeleteFldStrict, OP\_ScopedDeleteFldStrict, ElementScopedC)

DEF3\_WMS(CUSTOM, LdElemUndef, OP\_LdElementUndefined, ElementU)

EXDEF3\_WMS(CUSTOM, LdLocalElemUndef, OP\_LdLocalElementUndefined, ElementRootU)

DEF2\_WMS(XXtoA1, NewScObjectSimple, OP\_NewScObjectSimple)

DEF3 (CUSTOM, NewScObject\_A, OP\_NewScObject\_A, Auxiliary)

DEF3 (CUSTOM, NewScObjectLiteral, OP\_NewScObjectLiteral, Auxiliary)

DEF3 (CUSTOM\_L\_R0, LdPropIds, OP\_LdPropIds, Auxiliary)

DEF3 (CUSTOM, InitCachedFuncs, OP\_InitCachedFuncs, AuxNoReg)

DEF2\_WMS(LOCALI1toA1, GetCachedFunc, OP\_GetCachedFunc)

DEF2\_WMS(EnvU1toXX, InvalCachedScope, JavascriptOperators::OP\_InvalidateCachedScope)

DEF3 (CUSTOM, CommitScope, OP\_CommitScope, AuxNoReg)

DEF2\_WMS(A1I2toXXNonVar\_FuncBody, NewInnerScopeSlots, OP\_NewInnerScopeSlots)

DEF3\_WMS(CUSTOM, CloneInnerScopeSlots, OP\_CloneInnerScopeSlots, Unsigned1)

DEF3\_WMS(CUSTOM\_L\_R0, NewScArray, OP\_NewScArray, Reg1Unsigned1)

DEF2\_WMS(U1toA1, NewScArrayWithMissingValues,JavascriptArray::OP\_NewScArrayWithMissingValues)

DEF3 (CUSTOM\_L\_R0, NewScIntArray, OP\_NewScIntArray, Auxiliary)

DEF3 (CUSTOM\_L\_R0, NewScFltArray, OP\_NewScFltArray, Auxiliary)

DEF3\_WMS(CUSTOM\_L\_R0, ProfiledNewScArray, PROFILEDOP(OP\_ProfiledNewScArray, OP\_ProfiledNewScArray\_NoProfile), ProfiledReg1Unsigned1)

DEF3 (CUSTOM\_L\_R0, ProfiledNewScIntArray, PROFILEDOP(OP\_ProfiledNewScIntArray, OP\_NewScIntArray), ProfiledAuxiliary)

DEF3 (CUSTOM\_L\_R0, ProfiledNewScFltArray, PROFILEDOP(OP\_ProfiledNewScFltArray, OP\_NewScFltArray), ProfiledAuxiliary)

DEF2\_WMS(RegextoA1, NewRegEx, JavascriptRegExp::OP\_NewRegEx)

EXDEF3\_WMS(CUSTOM, InitClass, OP\_InitClass, Class)

DEF3\_WMS(BRBReturnP1toA1, BrOnEmpty, JavascriptOperators::OP\_BrOnEmpty, ForInObjectEnumerator \*)

DEF2 (TRY, TryCatch, OP\_TryCatch)

DEF2 (TRY, TryFinally, OP\_TryFinally)

EXDEF2\_WMS(TRYBR2, TryFinallyWithYield, OP\_TryFinallyWithYield)

EXDEF2 (EMPTY, ResumeCatch, OP\_ResumeCatch)

EXDEF2\_WMS(TRYBR2, ResumeFinally, OP\_ResumeFinally)

DEF2\_WMS(A1NonVarToA1, ResumeYield, OP\_ResumeYield)

DEF2\_WMS(A2NonVarToA1Reg, ResumeYieldStar, OP\_ResumeYield)

EXDEF2\_WMS(A2toA1Mem, AsyncSpawn, JavascriptOperators::OP\_AsyncSpawn)

EXDEF2 (W1, RuntimeTypeError, JavascriptExceptionOperators::OP\_RuntimeTypeError)

EXDEF2 (W1, RuntimeReferenceError, JavascriptExceptionOperators::OP\_RuntimeReferenceError)

DEF3 (CUSTOM\_L\_R0, SpreadArrayLiteral, OP\_SpreadArrayLiteral, Reg2Aux)

EXDEF2\_WMS(A1toXX, ObjectFreeze, JavascriptOperators::OP\_Freeze)

EXDEF3\_WMS(CUSTOM, ClearAttributes, OP\_ClearAttributes, ElementU)

DEF3\_WMS(CUSTOM, ApplyArgs, OP\_ApplyArgs, Reg5)

EXDEF3\_WMS(CUSTOM, EmitTmpRegCount, OP\_EmitTmpRegCount, Unsigned1)

#endif

// help the caller to undefine all the macros

#undef DEF2

#undef DEF3

#undef DEF2\_WMS

#undef DEF3\_WMS

#undef DEF4\_WMS

#undef EXDEF2

#undef EXDEF3

#undef EXDEF2\_WMS

#undef EXDEF3\_WMS

#undef EXDEF4\_WMS

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#ifndef TEMP\_DISABLE\_ASMJS

// See Lib\Runtime\Language\InterpreterProcessOpCodeAsmJs.h for Handler Process

// ( HandlerProcess , OpCodeAsmJs , HandlerFunction , LayoutAsmJs , Type )

// ( | , | , | , | , | )

// ( | , | , | , | , | )

// ( V , V , V , V , V )

DEF2 (NOPASMJS , Nop , Empty )

DEF2 (NOPASMJS , Label , Empty )

EXDEF2 (NOPASMJS , NopEx , Empty )

DEF2 (BR\_ASM , AsmBr , OP\_Br )

DEF2\_WMS(FALLTHROUGH\_ASM , LdSlotArr , /\* Common case with LdSlot \*/ )

DEF3\_WMS(GET\_ELEM\_SLOT\_ASM , LdSlot , OP\_LdAsmJsSlot , ElementSlot )

DEF2\_WMS(FUNCtoA1Mem , LdUndef , JavascriptOperators::OP\_LdUndef )

// Function Calls

DEF2(FALLTHROUGH\_ASM, I\_StartCall, /\* Common case with StartCall \*/ )

DEF3 ( CUSTOM\_ASMJS , StartCall , OP\_AsmStartCall , StartCall )

DEF3\_WMS(CUSTOM\_ASMJS , I\_Call , OP\_I\_AsmCall , AsmCall )

DEF3\_WMS( CUSTOM\_ASMJS , Call , OP\_AsmCall , AsmCall )

DEF2\_WMS(D1toR1Out , I\_ArgOut\_Db , OP\_I\_SetOutAsmDb )

DEF2\_WMS( D1toR1Out , ArgOut\_Db , OP\_SetOutAsmDb ) // convert double to var and set it as outparam

DEF2\_WMS(I1toR1Out , I\_ArgOut\_Int , OP\_I\_SetOutAsmInt )

DEF2\_WMS( I1toR1Out , ArgOut\_Int , OP\_SetOutAsmInt ) // convert int to var and set it as outparam

DEF2\_WMS( F1toR1Out , I\_ArgOut\_Flt , OP\_I\_SetOutAsmFlt ) // convert float to var and set it as outparam

DEF2\_WMS(D1toD1Mem , I\_Conv\_VTD , (double) )

DEF2\_WMS( R1toD1Mem , Conv\_VTD , JavascriptConversion::ToNumber ) // convert var to double

DEF2\_WMS(F1toF1Mem , I\_Conv\_VTF , (float) )

DEF2\_WMS(R1toF1Mem , Conv\_VTF , JavascriptConversion::ToNumber ) // convert var to float

DEF2\_WMS(I1toI1Mem , I\_Conv\_VTI , (int) )

DEF2\_WMS( R1toI1Mem , Conv\_VTI , JavascriptMath::ToInt32 ) // convert var to int

DEF3\_WMS( CUSTOM\_ASMJS , LdArr\_Func , OP\_LdArrFunc , ElementSlot )

DEF4\_WMS( TEMPLATE\_ASMJS , LdSlot\_Db , OP\_LdSlotPrimitive , ElementSlot, double )

DEF4\_WMS( TEMPLATE\_ASMJS , LdSlot\_Int , OP\_LdSlotPrimitive , ElementSlot, int )

DEF4\_WMS( TEMPLATE\_ASMJS , LdSlot\_Flt , OP\_LdSlotPrimitive , ElementSlot, float )

DEF4\_WMS( TEMPLATE\_ASMJS , StSlot\_Db , OP\_StSlotPrimitive , ElementSlot, double )

DEF4\_WMS( TEMPLATE\_ASMJS , StSlot\_Int , OP\_StSlotPrimitive , ElementSlot, int )

DEF4\_WMS( TEMPLATE\_ASMJS , StSlot\_Flt , OP\_StSlotPrimitive , ElementSlot, float )

DEF3\_WMS( CUSTOM\_ASMJS , LdArr , OP\_LdArrGeneric , AsmTypedArr )

DEF3\_WMS( CUSTOM\_ASMJS , LdArrConst , OP\_LdArrConstIndex , AsmTypedArr )

DEF3\_WMS( CUSTOM\_ASMJS , StArr , OP\_StArrGeneric , AsmTypedArr )

DEF3\_WMS( CUSTOM\_ASMJS , StArrConst , OP\_StArrConstIndex , AsmTypedArr )

DEF2\_WMS( C1toI1 , Ld\_IntConst , None )

DEF2\_WMS( BR\_ASM\_MemStack , BrTrue\_Int , None ) // Jumps to location if int reg is true

DEF2\_WMS( BR\_ASM\_Mem , BrEq\_Int , AsmJsMath::CmpEq<int> ) // Jumps to location if both int reg are equal

DEF2\_WMS( D1toI1Scr , Conv\_DTI , JavascriptConversion::ToInt32 ) // convert double to int

DEF2\_WMS( F1toI1Scr , Conv\_FTI , JavascriptConversion::ToInt32 ) // convert float to int

DEF2\_WMS( I1toD1Mem , Conv\_ITD , (double) ) // convert int to double

DEF2\_WMS( U1toD1Mem , Conv\_UTD , (double) ) // convert unsigned int to double

DEF2\_WMS( F1toD1Mem , Conv\_FTD , (double) ) // convert unsigned float to double

DEF2\_WMS( I1toI1Mem , Ld\_Int , (int) )

DEF2\_WMS( D1toD1Mem , Ld\_Db , (double) )

DEF2\_WMS( F1toF1Mem , Ld\_Flt , (float) )

DEF2\_WMS( D1toD1Mem , Return\_Db , (double) ) // convert double to var

DEF2\_WMS( F1toF1Mem , Return\_Flt , (float) ) // convert float to var

DEF2\_WMS( I1toI1Mem , Return\_Int , (int) ) // convert int to var

DEF2\_WMS( I1toI1Mem , BeginSwitch\_Int, (int) )

DEF2 ( BR\_ASM , EndSwitch\_Int, OP\_Br )

DEF2\_WMS( BR\_ASM\_Mem , Case\_Int , AsmJsMath::CmpEq<int> )

DEF2\_WMS( I1toI1Mem , Neg\_Int , AsmJsMath::Neg<int> ) // int unary '-'

DEF2\_WMS( I1toI1Mem , Not\_Int , AsmJsMath::Not ) // int unary '~'

DEF2\_WMS( I1toI1Mem , LogNot\_Int , AsmJsMath::LogNot ) // int unary '!'

DEF2\_WMS( I1toI1Mem , Conv\_ITB , AsmJsMath::ToBool ) // convert an int to a bool (0|1)

DEF2\_WMS( I2toI1Mem , Add\_Int , AsmJsMath::Add<int> )

DEF2\_WMS( I2toI1Mem , Sub\_Int , AsmJsMath::Sub<int> )

DEF2\_WMS( I2toI1Mem , Mul\_Int , AsmJsMath::Mul<int> )

DEF2\_WMS( I2toI1Mem , Div\_Int , AsmJsMath::Div<int> )

DEF2\_WMS( I2toI1Mem , Rem\_Int , AsmJsMath::Rem<int> )

DEF2\_WMS( I2toI1Mem , And\_Int , AsmJsMath::And )

DEF2\_WMS( I2toI1Mem , Or\_Int , AsmJsMath::Or )

DEF2\_WMS( I2toI1Mem , Xor\_Int , AsmJsMath::Xor )

DEF2\_WMS( I2toI1Mem , Shl\_Int , AsmJsMath::Shl )

DEF2\_WMS( I2toI1Mem , Shr\_Int , AsmJsMath::Shr )

DEF2\_WMS( I2toI1Mem , ShrU\_Int , AsmJsMath::ShrU )

DEF2\_WMS( I2toI1MemDConv , Mul\_UInt , AsmJsMath::Mul<double> )

DEF2\_WMS( I2toI1MemDConv , Div\_UInt , AsmJsMath::Div<double> )

DEF2\_WMS( I2toI1MemDConv , Rem\_UInt , AsmJsMath::Rem<uint> )

DEF2\_WMS( D1toD1Mem , Neg\_Db , AsmJsMath::Neg<double> ) // double unary '-'

DEF2\_WMS( D2toD1Mem , Add\_Db , AsmJsMath::Add<double> )

DEF2\_WMS( D2toD1Mem , Sub\_Db , AsmJsMath::Sub<double> )

DEF2\_WMS( D2toD1Mem , Mul\_Db , AsmJsMath::Mul<double> )

DEF2\_WMS( D2toD1Mem , Div\_Db , AsmJsMath::Div<double> )

DEF2\_WMS( D2toD1Mem , Rem\_Db , AsmJsMath::Rem<double> )

DEF2\_WMS( F1toF1Mem , Neg\_Flt , AsmJsMath::Neg<float> ) // float unary '-'

DEF2\_WMS( F2toF1Mem , Add\_Flt , AsmJsMath::Add<float> )

DEF2\_WMS( F2toF1Mem , Sub\_Flt , AsmJsMath::Sub<float> )

DEF2\_WMS( F2toF1Mem , Mul\_Flt , AsmJsMath::Mul<float> )

DEF2\_WMS( F2toF1Mem , Div\_Flt , AsmJsMath::Div<float> )

DEF2\_WMS( I2toI1Mem , CmLt\_Int , AsmJsMath::CmpLt<int> )

DEF2\_WMS( I2toI1Mem , CmLe\_Int , AsmJsMath::CmpLe<int> )

DEF2\_WMS( I2toI1Mem , CmGt\_Int , AsmJsMath::CmpGt<int> )

DEF2\_WMS( I2toI1Mem , CmGe\_Int , AsmJsMath::CmpGe<int> )

DEF2\_WMS( I2toI1Mem , CmEq\_Int , AsmJsMath::CmpEq<int> )

DEF2\_WMS( I2toI1Mem , CmNe\_Int , AsmJsMath::CmpNe<int> )

DEF2\_WMS( I2toI1Mem , CmLt\_UnInt , AsmJsMath::CmpLt<unsigned int> )

DEF2\_WMS( I2toI1Mem , CmLe\_UnInt , AsmJsMath::CmpLe<unsigned int> )

DEF2\_WMS( I2toI1Mem , CmGt\_UnInt , AsmJsMath::CmpGt<unsigned int> )

DEF2\_WMS( I2toI1Mem , CmGe\_UnInt , AsmJsMath::CmpGe<unsigned int> )

DEF2\_WMS( I1toI1Mem , Abs\_Int , ::abs )

DEF2\_WMS( I2toI1Mem , Min\_Int , min )

DEF2\_WMS( I2toI1Mem , Max\_Int , max )

DEF2\_WMS( I2toI1Mem , Imul\_Int , AsmJsMath::Mul<int> )

DEF2\_WMS( I1toI1Mem , Clz32\_Int , AsmJsMath::Clz32 )

DEF2\_WMS( D2toI1Mem , CmLt\_Db , AsmJsMath::CmpLt<double> )

DEF2\_WMS( D2toI1Mem , CmLe\_Db , AsmJsMath::CmpLe<double> )

DEF2\_WMS( D2toI1Mem , CmGt\_Db , AsmJsMath::CmpGt<double> )

DEF2\_WMS( D2toI1Mem , CmGe\_Db , AsmJsMath::CmpGe<double> )

DEF2\_WMS( D2toI1Mem , CmEq\_Db , AsmJsMath::CmpEq<double> )

DEF2\_WMS( D2toI1Mem , CmNe\_Db , AsmJsMath::CmpNe<double> )

DEF2\_WMS( F2toI1Mem , CmLt\_Flt , AsmJsMath::CmpLt<float> )

DEF2\_WMS( F2toI1Mem , CmLe\_Flt , AsmJsMath::CmpLe<float> )

DEF2\_WMS( F2toI1Mem , CmGt\_Flt , AsmJsMath::CmpGt<float> )

DEF2\_WMS( F2toI1Mem , CmGe\_Flt , AsmJsMath::CmpGe<float> )

DEF2\_WMS( F2toI1Mem , CmEq\_Flt , AsmJsMath::CmpEq<float> )

DEF2\_WMS( F2toI1Mem , CmNe\_Flt , AsmJsMath::CmpNe<float> )

DEF2\_WMS( D1toD1Mem , Sin\_Db , Math::Sin )

DEF2\_WMS( D1toD1Mem , Cos\_Db , Math::Cos )

DEF2\_WMS( D1toD1Mem , Tan\_Db , Math::Tan )

DEF2\_WMS( D1toD1Mem , Asin\_Db , Math::Asin )

DEF2\_WMS( D1toD1Mem , Acos\_Db , Math::Acos )

DEF2\_WMS( D1toD1Mem , Atan\_Db , Math::Atan )

DEF2\_WMS( D1toD1Mem , Ceil\_Db , ::ceil )

DEF2\_WMS( F1toF1Mem , Ceil\_Flt , ::ceilf )

DEF2\_WMS( D1toD1Mem , Floor\_Db , ::floor )

DEF2\_WMS( F1toF1Mem , Floor\_Flt , ::floorf )

DEF2\_WMS( D1toD1Mem , Exp\_Db , Math::Exp )

DEF2\_WMS( D1toD1Mem , Log\_Db , Math::Log )

DEF2\_WMS( D2toD1Mem , Pow\_Db , Math::Pow )

DEF2\_WMS( D1toD1Mem , Sqrt\_Db , ::sqrt )

DEF2\_WMS( F1toF1Mem , Sqrt\_Flt , ::sqrtf )

DEF2\_WMS( D1toD1Mem , Abs\_Db , Math::Abs )

DEF2\_WMS( F1toF1Mem , Abs\_Flt , ::fabsf )

DEF2\_WMS( D2toD1Mem , Atan2\_Db , Math::Atan2 )

DEF2\_WMS( D2toD1Mem , Min\_Db , AsmJsMath::Min<double> )

DEF2\_WMS( D2toD1Mem , Max\_Db , AsmJsMath::Max<double> )

DEF2\_WMS( F1toF1Mem , Fround\_Flt , (float) )

DEF2\_WMS( D1toF1Mem , Fround\_Db , (float) )

DEF2\_WMS( I1toF1Mem , Fround\_Int , (float) )

DEF2\_WMS( IP\_TARG\_ASM , AsmJsLoopBodyStart, OP\_ProfiledLoopBodyStart )

//unary ops

DEF2\_WMS( SIMD\_F4\_1toF4\_1 , Simd128\_Ld\_F4 , (AsmJsSIMDValue) )

DEF2\_WMS( SIMD\_I4\_1toI4\_1 , Simd128\_Ld\_I4 , (AsmJsSIMDValue) )

EXDEF2\_WMS( SIMD\_D2\_1toD2\_1 , Simd128\_Ld\_D2 , (AsmJsSIMDValue) )

DEF2\_WMS( SIMD\_F4toF4\_1 , Simd128\_FloatsToF4 , SIMDFloat32x4Operation::OpFloat32x4 )

DEF2\_WMS( SIMD\_I4toI4\_1 , Simd128\_IntsToI4 , SIMDInt32x4Operation::OpInt32x4 )

DEF2\_WMS( SIMD\_D2toD2\_1 , Simd128\_DoublesToD2 , SIMDFloat64x2Operation::OpFloat64x2 )

DEF4\_WMS( TEMPLATE\_ASMJS , Simd128\_LdSlot\_F4 , OP\_LdSlotPrimitive , ElementSlot, AsmJsSIMDValue)

DEF4\_WMS( TEMPLATE\_ASMJS , Simd128\_LdSlot\_I4 , OP\_LdSlotPrimitive , ElementSlot, AsmJsSIMDValue)

EXDEF4\_WMS( TEMPLATE\_ASMJS , Simd128\_LdSlot\_D2 , OP\_LdSlotPrimitive , ElementSlot, AsmJsSIMDValue)

DEF4\_WMS(TEMPLATE\_ASMJS , Simd128\_StSlot\_F4 , OP\_StSlotPrimitive , ElementSlot, AsmJsSIMDValue)

DEF4\_WMS(TEMPLATE\_ASMJS , Simd128\_StSlot\_I4 , OP\_StSlotPrimitive , ElementSlot, AsmJsSIMDValue)

EXDEF4\_WMS(TEMPLATE\_ASMJS , Simd128\_StSlot\_D2 , OP\_StSlotPrimitive , ElementSlot, AsmJsSIMDValue)

DEF2\_WMS( SIMD\_F4\_1toF4\_1 , Simd128\_Return\_F4 , (AsmJsSIMDValue) )

DEF2\_WMS( SIMD\_I4\_1toI4\_1 , Simd128\_Return\_I4 , (AsmJsSIMDValue) )

DEF2\_WMS( SIMD\_D2\_1toD2\_1 , Simd128\_Return\_D2 , (AsmJsSIMDValue) )

DEF2\_WMS( SIMD\_F1toF4\_1 , Simd128\_Splat\_F4 ,Js::SIMDFloat32x4Operation::OpSplat )

DEF2\_WMS( SIMD\_I1toI4\_1 , Simd128\_Splat\_I4 ,Js::SIMDInt32x4Operation::OpSplat )

DEF2\_WMS( SIMD\_D1toD2\_1 , Simd128\_Splat\_D2 ,Js::SIMDFloat64x2Operation::OpSplat )

DEF2\_WMS( SIMD\_D2\_1toF4\_1 , Simd128\_FromFloat64x2\_F4 ,SIMDFloat32x4Operation::OpFromFloat64x2 )

DEF2\_WMS( SIMD\_D2\_1toF4\_1 , Simd128\_FromFloat64x2Bits\_F4,SIMDFloat32x4Operation::OpFromFloat64x2Bits )

DEF2\_WMS( SIMD\_I4\_1toF4\_1 , Simd128\_FromInt32x4\_F4 ,SIMDFloat32x4Operation::OpFromInt32x4 )

DEF2\_WMS( SIMD\_I4\_1toF4\_1 , Simd128\_FromInt32x4Bits\_F4 ,SIMDFloat32x4Operation::OpFromInt32x4Bits )

DEF2\_WMS( SIMD\_D2\_1toI4\_1 , Simd128\_FromFloat64x2\_I4 ,SIMDInt32x4Operation::OpFromFloat64x2 )

DEF2\_WMS( SIMD\_D2\_1toI4\_1 , Simd128\_FromFloat64x2Bits\_I4,SIMDInt32x4Operation::OpFromFloat64x2Bits )

DEF2\_WMS( SIMD\_F4\_1toI4\_1 , Simd128\_FromFloat32x4\_I4 ,SIMDInt32x4Operation::OpFromFloat32x4 )

DEF2\_WMS( SIMD\_F4\_1toI4\_1 , Simd128\_FromFloat32x4Bits\_I4,SIMDInt32x4Operation::OpFromFloat32x4Bits )

DEF2\_WMS( SIMD\_F4\_1toD2\_1 , Simd128\_FromFloat32x4\_D2 ,SIMDFloat64x2Operation::OpFromFloat32x4 )

DEF2\_WMS( SIMD\_F4\_1toD2\_1 , Simd128\_FromFloat32x4Bits\_D2,SIMDFloat64x2Operation::OpFromFloat32x4Bits )

DEF2\_WMS( SIMD\_I4\_1toD2\_1 , Simd128\_FromInt32x4\_D2 ,SIMDFloat64x2Operation::OpFromInt32x4 )

DEF2\_WMS( SIMD\_I4\_1toD2\_1 , Simd128\_FromInt32x4Bits\_D2 ,SIMDFloat64x2Operation::OpFromInt32x4Bits )

DEF2\_WMS( SIMD\_F4\_1toF4\_1 , Simd128\_Abs\_F4 ,SIMDFloat32x4Operation::OpAbs )

DEF2\_WMS( SIMD\_D2\_1toD2\_1 , Simd128\_Abs\_D2 ,SIMDFloat64x2Operation::OpAbs )

DEF2\_WMS( SIMD\_F4\_1toF4\_1 , Simd128\_Neg\_F4 ,SIMDFloat32x4Operation::OpNeg )

DEF2\_WMS( SIMD\_I4\_1toI4\_1 , Simd128\_Neg\_I4 ,SIMDInt32x4Operation::OpNeg )

DEF2\_WMS( SIMD\_D2\_1toD2\_1 , Simd128\_Neg\_D2 ,SIMDFloat64x2Operation::OpNeg )

DEF2\_WMS( SIMD\_F4\_1toF4\_1 , Simd128\_Rcp\_F4 ,SIMDFloat32x4Operation::OpReciprocal )

DEF2\_WMS( SIMD\_D2\_1toD2\_1 , Simd128\_Rcp\_D2 ,SIMDFloat64x2Operation::OpReciprocal )

DEF2\_WMS( SIMD\_F4\_1toF4\_1 , Simd128\_RcpSqrt\_F4 ,SIMDFloat32x4Operation::OpReciprocalSqrt )

DEF2\_WMS( SIMD\_D2\_1toD2\_1 , Simd128\_RcpSqrt\_D2 ,SIMDFloat64x2Operation::OpReciprocalSqrt )

DEF2\_WMS( SIMD\_F4\_1toF4\_1 , Simd128\_Sqrt\_F4 ,SIMDFloat32x4Operation::OpSqrt )

DEF2\_WMS( SIMD\_D2\_1toD2\_1 , Simd128\_Sqrt\_D2 ,SIMDFloat64x2Operation::OpSqrt )

DEF2\_WMS( SIMD\_F4\_1toF4\_1 , Simd128\_Not\_F4 , Js::SIMDFloat32x4Operation::OpNot )

DEF2\_WMS( SIMD\_I4\_1toI4\_1 , Simd128\_Not\_I4 , Js::SIMDInt32x4Operation::OpNot )

// binary ops

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Add\_F4 , Js::SIMDFloat32x4Operation::OpAdd )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_Add\_I4 , Js::SIMDInt32x4Operation::OpAdd )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Add\_D2 , Js::SIMDFloat64x2Operation::OpAdd )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Sub\_F4 , Js::SIMDFloat32x4Operation::OpSub )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_Sub\_I4 , Js::SIMDInt32x4Operation::OpSub )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Sub\_D2 , Js::SIMDFloat64x2Operation::OpSub )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Mul\_F4 , Js::SIMDFloat32x4Operation::OpMul )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_Mul\_I4 , Js::SIMDInt32x4Operation::OpMul )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Mul\_D2 , Js::SIMDFloat64x2Operation::OpMul )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Div\_F4 , Js::SIMDFloat32x4Operation::OpDiv )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Div\_D2 , Js::SIMDFloat64x2Operation::OpDiv )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Min\_D2 , Js::SIMDFloat64x2Operation::OpMin )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Min\_F4 , Js::SIMDFloat32x4Operation::OpMin )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Max\_F4 , Js::SIMDFloat32x4Operation::OpMax )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Max\_D2 , Js::SIMDFloat64x2Operation::OpMax )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Lt\_F4 , Js::SIMDFloat32x4Operation::OpLessThan )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_Lt\_I4 , Js::SIMDInt32x4Operation::OpLessThan )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Lt\_D2 , Js::SIMDFloat64x2Operation::OpLessThan )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_LtEq\_F4 , Js::SIMDFloat32x4Operation::OpLessThanOrEqual)

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_LtEq\_D2 , Js::SIMDFloat64x2Operation::OpLessThanOrEqual)

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Eq\_F4 , Js::SIMDFloat32x4Operation::OpEqual )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_Eq\_I4 , Js::SIMDInt32x4Operation::OpEqual )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Eq\_D2 , Js::SIMDFloat64x2Operation::OpEqual )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Neq\_F4 , Js::SIMDFloat32x4Operation::OpNotEqual )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Neq\_D2 , Js::SIMDFloat64x2Operation::OpNotEqual )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_GtEq\_F4 , Js::SIMDFloat32x4Operation::OpGreaterThanOrEqual)

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_GtEq\_D2 , Js::SIMDFloat64x2Operation::OpGreaterThanOrEqual)

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Gt\_F4 , Js::SIMDFloat32x4Operation::OpGreaterThan )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_Gt\_I4 , Js::SIMDInt32x4Operation::OpGreaterThan )

DEF2\_WMS( SIMD\_D2\_2toD2\_1 , Simd128\_Gt\_D2 , Js::SIMDFloat64x2Operation::OpGreaterThan )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_And\_F4 , Js::SIMDFloat32x4Operation::OpAnd )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_And\_I4 , Js::SIMDInt32x4Operation::OpAnd )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Or\_F4 , Js::SIMDFloat32x4Operation::OpOr )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_Or\_I4 , Js::SIMDInt32x4Operation::OpOr )

DEF2\_WMS( SIMD\_F4\_2toF4\_1 , Simd128\_Xor\_F4 , Js::SIMDFloat32x4Operation::OpXor )

DEF2\_WMS( SIMD\_I4\_2toI4\_1 , Simd128\_Xor\_I4 , Js::SIMDInt32x4Operation::OpXor )

// ternary ops

DEF2\_WMS( SIMD\_F4\_3toF4\_1 , Simd128\_Clamp\_F4 , Js::SIMDFloat32x4Operation::OpClamp )

DEF2\_WMS( SIMD\_D2\_3toD2\_1 , Simd128\_Clamp\_D2 , Js::SIMDFloat64x2Operation::OpClamp )

DEF2\_WMS( SIMD\_I4\_1F4\_2toF4\_1 , Simd128\_Select\_F4 , Js::SIMDFloat32x4Operation::OpSelect )

DEF2\_WMS( SIMD\_I4\_3toI4\_1 , Simd128\_Select\_I4 , Js::SIMDInt32x4Operation::OpSelect )

DEF2\_WMS( SIMD\_I4\_1D2\_2toD2\_1 , Simd128\_Select\_D2 , Js::SIMDFloat64x2Operation::OpSelect )

DEF2\_WMS( SIMD\_F4\_1toI1 , Simd128\_LdSignMask\_F4 , Js::SIMDFloat32x4Operation::OpGetSignMask )

DEF2\_WMS( SIMD\_I4\_1toI1 , Simd128\_LdSignMask\_I4 , Js::SIMDInt32x4Operation::OpGetSignMask )

DEF2\_WMS( SIMD\_D2\_1toI1 , Simd128\_LdSignMask\_D2 , Js::SIMDFloat64x2Operation::OpGetSignMask )

// args out, copy value to outParams

DEF2\_WMS ( SIMD\_F4\_1toR1Mem , Simd128\_I\_ArgOut\_F4 , OP\_I\_SetOutAsmSimd ) // ArgOut Float64x2

DEF2\_WMS ( SIMD\_I4\_1toR1Mem , Simd128\_I\_ArgOut\_I4 , OP\_I\_SetOutAsmSimd ) // ArgOut Float64x2

DEF2\_WMS ( SIMD\_D2\_1toR1Mem , Simd128\_I\_ArgOut\_D2 , OP\_I\_SetOutAsmSimd ) // ArgOut Float64x2

DEF2\_WMS ( SIMD\_F4\_1toF4\_1 , Simd128\_I\_Conv\_VTF4 , (AsmJsSIMDValue) )

DEF2\_WMS ( SIMD\_I4\_1toI4\_1 , Simd128\_I\_Conv\_VTI4 , (AsmJsSIMDValue) )

DEF2\_WMS ( SIMD\_D2\_1toD2\_1 , Simd128\_I\_Conv\_VTD2 , (AsmJsSIMDValue) )

DEF2\_WMS ( SIMD\_F4\_1I4toF4\_1 , Simd128\_Swizzle\_F4 , SIMD128InnerShuffle )

DEF2\_WMS ( SIMD\_F4\_2I4toF4\_1 , Simd128\_Shuffle\_F4 , SIMD128InnerShuffle )

DEF2\_WMS ( SIMD\_I4\_1I4toI4\_1 , Simd128\_Swizzle\_I4 , SIMD128InnerShuffle )

DEF2\_WMS ( SIMD\_I4\_2I4toI4\_1 , Simd128\_Shuffle\_I4 , SIMD128InnerShuffle )

// Extended opcodes

EXDEF2\_WMS ( SIMD\_D2\_1I2toD2\_1 , Simd128\_Swizzle\_D2 , SIMD128InnerShuffle )

EXDEF2\_WMS ( SIMD\_D2\_2I2toD2\_1 , Simd128\_Shuffle\_D2 , SIMD128InnerShuffle )

//Lane Access

EXDEF2\_WMS ( SIMD\_I4\_1I1toI1 , Simd128\_ExtractLane\_I4 , SIMD128InnerExtractLaneI4 )

EXDEF2\_WMS ( SIMD\_I4\_1I2toI4\_1 , Simd128\_ReplaceLane\_I4 , SIMD128InnerReplaceLaneI4 )

EXDEF2\_WMS ( SIMD\_F4\_1I1toF1 , Simd128\_ExtractLane\_F4 , SIMD128InnerExtractLaneF4 )

EXDEF2\_WMS ( SIMD\_F4\_1I1F1toF4\_1, Simd128\_ReplaceLane\_F4 , SIMD128InnerReplaceLaneF4 )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_LdArr\_F4 , OP\_SimdLdArrGeneric , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_LdArr\_I4 , OP\_SimdLdArrGeneric , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_LdArr\_D2 , OP\_SimdLdArrGeneric , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_StArr\_F4 , OP\_SimdStArrGeneric , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_StArr\_I4 , OP\_SimdStArrGeneric , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_StArr\_D2 , OP\_SimdStArrGeneric , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_LdArrConst\_F4 , OP\_SimdLdArrConstIndex , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_LdArrConst\_I4 , OP\_SimdLdArrConstIndex , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_LdArrConst\_D2 , OP\_SimdLdArrConstIndex , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_StArrConst\_F4 , OP\_SimdStArrConstIndex , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_StArrConst\_I4 , OP\_SimdStArrConstIndex , AsmSimdTypedArr )

EXDEF3\_WMS ( CUSTOM\_ASMJS , Simd128\_StArrConst\_D2 , OP\_SimdStArrConstIndex , AsmSimdTypedArr )

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

// Shared intepreter loop

//

// This holds the single definition of the interpreter loop.

// It allows for configurable copies of the loop that do extra work without

// impacting the mainline performance. (for example the debug loop can simply

// check a bit without concern for impacting the nondebug mode.)

#if defined(INTERPRETER\_ASMJS) && !defined(TEMP\_DISABLE\_ASMJS)

#define INTERPRETER\_OPCODE OpCodeAsmJs

#else

#define INTERPRETER\_OPCODE OpCode

#endif

#ifdef PROVIDE\_DEBUGGING

#define DEBUGGING\_LOOP 1

#else

#define DEBUGGING\_LOOP 0

#endif

#ifdef PROVIDE\_INTERPRETERPROFILE

#define INTERPRETERPROFILE 1

#define PROFILEDOP(prof, unprof) prof

#else

#define INTERPRETERPROFILE 0

#define PROFILEDOP(prof, unprof) unprof

#endif

#if defined (DBG)

// Win8 516184: Huge switch with lots of labels each having a few locals on ARM.DBG causes each occurrence

// of this function (call of a javascript function in interpreter mode) to take 7+KB stack space

// (without optimizations the compiler accounts for ALL locals inside case labels when allocating space on stack

// for locals - SP does not change inside the function). On other platforms this is still huge but better than ARM.

// So, for DBG turn on optimizations to prevent this huge loss of stack.

#pragma optimize("g", on)

#endif

Var Js::InterpreterStackFrame::INTERPRETERLOOPNAME()

{

PROBE\_STACK(scriptContext, Js::Constants::MinStackInterpreter);

if (!this->closureInitDone)

{

// If this is the start of the function, then we've waited until after the stack probe above

// to set up the FD/SS pointers, so do it now.

Assert(this->m\_reader.GetCurrentOffset() == 0);

this->InitializeClosures();

}

Assert(this->returnAddress != nullptr);

AssertMsg(m\_arguments == NULL || Js::ArgumentsObject::Is(m\_arguments), "Bad arguments!");

// IP Passing in the interpreter:

// We keep a local copy of the bytecode's instruction pointer and

// pass it by reference to the bytecode reader.

// This allows the optimizer to recognize that the local (held in a register)

// dominates the copy in the reader.

// The effect is our dispatch loop is significantly smaller in the common case

// on optimized builds.

//

// For checked builds this does mean we are incrementing 2 different counters to

// track the ip.

const byte\* ip = m\_reader.GetIP();

while (true)

{

INTERPRETER\_OPCODE op = ReadByteOp<INTERPRETER\_OPCODE>(ip);

#ifdef ENABLE\_BASIC\_TELEMETRY

if( TELEMETRY\_OPCODE\_OFFSET\_ENABLED )

{

OpcodeTelemetry& opcodeTelemetry = this->scriptContext->GetTelemetry().GetOpcodeTelemetry();

opcodeTelemetry.ProgramLocationFunctionId ( this->function->GetFunctionInfo()->GetLocalFunctionId() );

opcodeTelemetry.ProgramLocationBytecodeOffset( this->m\_reader.GetCurrentOffset() );

}

#endif

#if DEBUGGING\_LOOP

if (this->scriptContext->GetThreadContext()->GetDebugManager()->stepController.IsActive() &&

this->scriptContext->GetThreadContext()->GetDebugManager()->stepController.IsStepComplete\_AllowingFalsePositives(this))

{

// BrLong is used for branch island, we don't want to break over there, as they don't belong to any statement. Just skip this.

if (!InterpreterStackFrame::IsBrLong(op, ip) && !this->m\_functionBody->GetUtf8SourceInfo()->GetIsLibraryCode())

{

uint prevOffset = m\_reader.GetCurrentOffset();

InterpreterHaltState haltState(STOP\_STEPCOMPLETE, m\_functionBody);

this->scriptContext->GetDebugContext()->GetProbeContainer()->DispatchStepHandler(&haltState, &op);

if (prevOffset != m\_reader.GetCurrentOffset())

{

// The location of the statement has been changed, setnextstatement was called.

// Reset m\_outParams and m\_outSp as before SetNext was called, we could be in the middle of StartCall.

// It's fine to do because SetNext can only be done to a statement -- function-level destination,

// and can't land to an expression inside call.

ResetOut();

ip = m\_reader.GetIP();

continue;

}

}

}

// The break opcode will be handled later in the switch block.

if (op != OpCode::Break && this->scriptContext->GetThreadContext()->GetDebugManager()->asyncBreakController.IsBreak())

{

if (!InterpreterStackFrame::IsBrLong(op, ip) && !this->m\_functionBody->GetUtf8SourceInfo()->GetIsLibraryCode())

{

uint prevOffset = m\_reader.GetCurrentOffset();

InterpreterHaltState haltState(STOP\_ASYNCBREAK, m\_functionBody);

this->scriptContext->GetDebugContext()->GetProbeContainer()->DispatchAsyncBreak(&haltState);

if (prevOffset != m\_reader.GetCurrentOffset())

{

// The location of the statement has been changed, setnextstatement was called.

ip = m\_reader.GetIP();

continue;

}

}

}

SWAP\_BP\_FOR\_OPCODE:

#endif

switch (op)

{

case INTERPRETER\_OPCODE::Ret:

{

//

// Return "Reg: 0" as the return-value.

// - JavaScript functions always return a value, and this value is always

// accessible to the caller. For an empty "return;" or exiting the end of the

// function's body, it is assumed that the byte-code author

// (ByteCodeGenerator) will load 'undefined' into R0.

// - If R0 has not explicitly been set, it will contain whatever garbage value

// was last set.

//

this->retOffset = m\_reader.GetCurrentOffset();

m\_reader.Empty(ip);

return GetReg((RegSlot)0);

}

case INTERPRETER\_OPCODE::Yield:

{

m\_reader.Reg2\_Small(ip);

return GetReg(GetFunctionBody()->GetYieldRegister());

}

#define DEF2(x, op, func) PROCESS\_##x(op, func)

#define DEF3(x, op, func, y) PROCESS\_##x(op, func, y)

#define DEF2\_WMS(x, op, func) PROCESS\_##x##\_COMMON(op, func, \_Small)

#define DEF3\_WMS(x, op, func, y) PROCESS\_##x##\_COMMON(op, func, y, \_Small)

#define DEF4\_WMS(x, op, func, y, t) PROCESS\_##x##\_COMMON(op, func, y, \_Small, t)

#include "InterpreterHandler.inl"

case INTERPRETER\_OPCODE::Leave:

// Return the continuation address to the helper.

// This tells the helper that control left the scope without completing the try/handler,

// which is particularly significant when executing a finally.

m\_reader.Empty(ip);

return (Var)this->m\_reader.GetCurrentOffset();

case INTERPRETER\_OPCODE::LeaveNull:

// Return to the helper without specifying a continuation address,

// indicating that the handler completed without jumping, so exception processing

// should continue.

m\_reader.Empty(ip);

return nullptr;

case INTERPRETER\_OPCODE::ExtendedOpcodePrefix:

{

ip = [this](const byte \* ip) -> const byte \*

{

INTERPRETER\_OPCODE op = (INTERPRETER\_OPCODE)(ReadByteOp<INTERPRETER\_OPCODE>(ip

#if DBG\_DUMP

, true

#endif

) + (INTERPRETER\_OPCODE::ExtendedOpcodePrefix << 8));

switch (op)

{

#define EXDEF2(x, op, func) PROCESS\_##x(op, func)

#define EXDEF3(x, op, func, y) PROCESS\_##x(op, func, y)

#define EXDEF2\_WMS(x, op, func) PROCESS\_##x##\_COMMON(op, func, \_Small)

#define EXDEF3\_WMS(x, op, func, y) PROCESS\_##x##\_COMMON(op, func, y, \_Small)

#define EXDEF4\_WMS(x, op, func, y, t) PROCESS\_##x##\_COMMON(op, func, y, \_Small, t)

#include "InterpreterHandler.inl"

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

AssertMsg(false, "dispatch to bad opcode");

\_\_assume(false);

};

return ip;

}(ip);

#if ENABLE\_PROFILE\_INFO

if (switchProfileMode)

{

// Aborting the current interpreter loop to switch the profile mode

return nullptr;

}

#endif

break;

}

case INTERPRETER\_OPCODE::MediumLayoutPrefix:

{

Var yieldValue = nullptr;

ip = [this, &yieldValue](const byte \* ip) -> const byte \*

{

INTERPRETER\_OPCODE op = ReadByteOp<INTERPRETER\_OPCODE>(ip);

switch (op)

{

case INTERPRETER\_OPCODE::Yield:

m\_reader.Reg2\_Medium(ip);

yieldValue = GetReg(GetFunctionBody()->GetYieldRegister());

break;

#define DEF2\_WMS(x, op, func) PROCESS\_##x##\_COMMON(op, func, \_Medium)

#define DEF3\_WMS(x, op, func, y) PROCESS\_##x##\_COMMON(op, func, y, \_Medium)

#define DEF4\_WMS(x, op, func, y, t) PROCESS\_##x##\_COMMON(op, func, y, \_Medium, t)

#include "InterpreterHandler.inl"

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

AssertMsg(false, "dispatch to bad opcode");

\_\_assume(false);

}

return ip;

}(ip);

if (yieldValue != nullptr)

{

return yieldValue;

}

#if ENABLE\_PROFILE\_INFO

if (switchProfileMode)

{

// Aborting the current interpreter loop to switch the profile mode

return nullptr;

}

#endif

break;

}

case INTERPRETER\_OPCODE::ExtendedMediumLayoutPrefix:

{

#ifndef INTERPRETER\_ASMJS // Asmjs doesn't have any extended opcodes for now, remove that case

ip = [this](const byte \* ip) -> const byte \*

{

INTERPRETER\_OPCODE op = (INTERPRETER\_OPCODE)(ReadByteOp<INTERPRETER\_OPCODE>(ip

#if DBG\_DUMP

, true

#endif

) + (INTERPRETER\_OPCODE::ExtendedOpcodePrefix << 8));

switch (op)

{

#define EXDEF2\_WMS(x, op, func) PROCESS\_##x##\_COMMON(op, func, \_Medium)

#define EXDEF3\_WMS(x, op, func, y) PROCESS\_##x##\_COMMON(op, func, y, \_Medium)

#define EXDEF4\_WMS(x, op, func, y, t) PROCESS\_##x##\_COMMON(op, func, y, \_Medium, t)

#include "InterpreterHandler.inl"

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

AssertMsg(false, "dispatch to bad opcode");

\_\_assume(false);

};

return ip;

}(ip);

#if ENABLE\_PROFILE\_INFO

if (switchProfileMode)

{

// Aborting the current interpreter loop to switch the profile mode

return nullptr;

}

#endif

#endif

break;

}

case INTERPRETER\_OPCODE::LargeLayoutPrefix:

{

Var yieldValue = nullptr;

ip = [this, &yieldValue](const byte \* ip) -> const byte \*

{

INTERPRETER\_OPCODE op = ReadByteOp<INTERPRETER\_OPCODE>(ip);

switch (op)

{

case INTERPRETER\_OPCODE::Yield:

m\_reader.Reg2\_Large(ip);

yieldValue = GetReg(GetFunctionBody()->GetYieldRegister());

break;

#define DEF2\_WMS(x, op, func) PROCESS\_##x##\_COMMON(op, func, \_Large)

#define DEF3\_WMS(x, op, func, y) PROCESS\_##x##\_COMMON(op, func, y, \_Large)

#define DEF4\_WMS(x, op, func, y, t) PROCESS\_##x##\_COMMON(op, func, y, \_Large, t)

#include "InterpreterHandler.inl"

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

AssertMsg(false, "dispatch to bad opcode");

\_\_assume(false);

}

return ip;

}(ip);

if (yieldValue != nullptr)

{

return yieldValue;

}

#if ENABLE\_PROFILE\_INFO

if(switchProfileMode)

{

// Aborting the current interpreter loop to switch the profile mode

return nullptr;

}

#endif

break;

}

case INTERPRETER\_OPCODE::ExtendedLargeLayoutPrefix:

{

#ifndef INTERPRETER\_ASMJS // Asmjs doesn't have any extended opcodes for now, remove that case

ip = [this](const byte \* ip) -> const byte \*

{

INTERPRETER\_OPCODE op = (INTERPRETER\_OPCODE)(ReadByteOp<INTERPRETER\_OPCODE>(ip

#if DBG\_DUMP

, true

#endif

) + (INTERPRETER\_OPCODE::ExtendedOpcodePrefix << 8));

switch (op)

{

#define EXDEF2\_WMS(x, op, func) PROCESS\_##x##\_COMMON(op, func, \_Large)

#define EXDEF3\_WMS(x, op, func, y) PROCESS\_##x##\_COMMON(op, func, y, \_Large)

#define EXDEF4\_WMS(x, op, func, y, t) PROCESS\_##x##\_COMMON(op, func, y, \_Large, t)

#include "InterpreterHandler.inl"

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

AssertMsg(false, "dispatch to bad opcode");

\_\_assume(false);

};

return ip;

}(ip);

#if ENABLE\_PROFILE\_INFO

if(switchProfileMode)

{

// Aborting the current interpreter loop to switch the profile mode

return nullptr;

}

#endif

#endif

break;

}

case INTERPRETER\_OPCODE::EndOfBlock:

{

// Note that at this time though ip was advanced by 'OpCode op = ReadByteOp<INTERPRETER\_OPCODE>(ip)',

// we haven't advanced m\_reader.m\_currentLocation yet, thus m\_reader.m\_currentLocation still points to EndOfBLock,

// and that +1 will point to 1st byte past the buffer.

Assert(m\_reader.GetCurrentOffset() + sizeof(byte) == m\_functionBody->GetByteCode()->GetLength());

//

// Reached an "OpCode::EndOfBlock" so need to exit this interpreter loop because

// there is no more byte-code to execute.

// - This prevents us from accessing random memory as byte-codes.

// - Functions should contain an "OpCode::Ret" instruction to organize an

// orderly return.

//

#if DEBUGGING\_LOOP

// However, during debugging an exception can be skipped which causes the

// statement that caused to exception to be skipped. If this statement is

// the statement that contains the OpCode::Ret then the EndOfBlock will

// be executed. In these cases it is sufficient to return undefined.

return this->scriptContext->GetLibrary()->GetUndefined();

#else

return nullptr;

#endif

}

case INTERPRETER\_OPCODE::Break:

{

#if DEBUGGING\_LOOP

// The reader has already advanced the IP:

if (this->m\_functionBody->ProbeAtOffset(m\_reader.GetCurrentOffset(), &op))

{

uint prevOffset = m\_reader.GetCurrentOffset();

InterpreterHaltState haltState(STOP\_BREAKPOINT, m\_functionBody);

this->scriptContext->GetDebugContext()->GetProbeContainer()->DispatchProbeHandlers(&haltState);

if (prevOffset != m\_reader.GetCurrentOffset())

{

// The location of the statement has been changed, setnextstatement was called.

ip = m\_reader.GetIP();

continue;

}

// Jump back to the start of the switch.

goto SWAP\_BP\_FOR\_OPCODE;

}

else

{

#if DEBUGGING\_LOOP

// an inline break statement rather than a probe

if (!this->scriptContext->GetThreadContext()->GetDebugManager()->stepController.ContinueFromInlineBreakpoint())

{

uint prevOffset = m\_reader.GetCurrentOffset();

InterpreterHaltState haltState(STOP\_INLINEBREAKPOINT, m\_functionBody);

this->scriptContext->GetDebugContext()->GetProbeContainer()->DispatchInlineBreakpoint(&haltState);

if (prevOffset != m\_reader.GetCurrentOffset())

{

// The location of the statement has been changed, setnextstatement was called.

ip = m\_reader.GetIP();

continue;

}

}

#endif

// Consume after dispatching

m\_reader.Empty(ip);

}

#else

m\_reader.Empty(ip);

#endif

break;

}

default:

// Help the C++ optimizer by declaring that the cases we

// have above are sufficient

AssertMsg(false, "dispatch to bad opcode");

\_\_assume(false);

}

}

}

#if defined (DBG)

// Restore optimizations to what's specified by the /O switch.

#pragma optimize("", on)

#endif

#undef DEBUGGING\_LOOP

#undef INTERPRETERPROFILE

#undef PROFILEDOP

#undef INTERPRETER\_OPCODE

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#ifndef TEMP\_DISABLE\_ASMJS

#define PROCESS\_FALLTHROUGH\_ASM(name, func) \

case OpCodeAsmJs::name:

#define PROCESS\_FALLTHROUGH\_ASM\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name:

#define PROCESS\_READ\_LAYOUT\_ASMJS(name, layout, suffix) \

CompileAssert(OpCodeInfoAsmJs<OpCodeAsmJs::name>::Layout == OpLayoutTypeAsmJs::layout); \

const unaligned OpLayout##layout##suffix \* playout = m\_reader.layout##suffix(ip); \

Assert((playout != nullptr) == (Js::OpLayoutTypeAsmJs::##layout != Js::OpLayoutTypeAsmJs::Empty)); // Make sure playout is used

#define PROCESS\_NOPASMJS\_COMMON(name, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

break; \

}

#define PROCESS\_NOPASMJS(name, func) PROCESS\_NOPASMJS\_COMMON(name, func,)

#define PROCESS\_BR\_ASM(name, func) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, AsmBr,); \

ip = func(playout); \

break; \

}

#define PROCESS\_GET\_ELEM\_SLOT\_ASM\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

SetNonVarReg(playout->Value, \

func(GetNonVarReg(playout->Instance), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_SLOT\_ASM(name, func, layout) PROCESS\_GET\_ELEM\_SLOT\_ASM\_COMMON(name, func, layout,)

#define PROCESS\_FUNCtoA1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, AsmReg1, suffix); \

SetReg(playout->R0, \

func(GetScriptContext())); \

break; \

}

#define PROCESS\_FUNCtoA1Mem(name, func) PROCESS\_FUNCtoA1Mem\_COMMON(name, func,)

#define PROCESS\_CUSTOM\_ASMJS\_COMMON(name, func, layout, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

func(playout); \

break; \

}

#define PROCESS\_CUSTOM\_ASMJS(name, func, layout) PROCESS\_CUSTOM\_ASMJS\_COMMON(name, func, layout,)

#define PROCESS\_I2toI1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int3, suffix); \

SetRegRawInt(playout->I0, \

func(GetRegRawInt(playout->I1), GetRegRawInt(playout->I2))); \

break; \

}

#define PROCESS\_I2toI1Mem(name, func) PROCESS\_I2toI1Mem\_COMMON(name, func,)

#define PROCESS\_I2toI1MemDConv\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int3, suffix); \

SetRegRawInt(playout->I0, \

JavascriptConversion::ToInt32(\

func((unsigned int)GetRegRawInt(playout->I1), (unsigned int)GetRegRawInt(playout->I2)))); \

break; \

}

#define PROCESS\_I2toI1MemDConv(name, func) PROCESS\_I2toI1MemDConv\_COMMON(name, func,)

#define PROCESS\_F2toF1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float3, suffix); \

SetRegRawFloat(playout->F0, \

func(GetRegRawFloat(playout->F1), GetRegRawFloat(playout->F2))); \

break; \

}

#define PROCESS\_D2toD1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Double3, suffix); \

SetRegRawDouble(playout->D0, \

func(GetRegRawDouble(playout->D1), GetRegRawDouble(playout->D2))); \

break; \

}

#define PROCESS\_D2toD1Mem(name, func) PROCESS\_D2toD1Mem\_COMMON(name, func,)

#define PROCESS\_F2toF1Mem(name, func) PROCESS\_F2toF1Mem\_COMMON(name, func,)

#define PROCESS\_I1toI1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int2, suffix); \

SetRegRawInt(playout->I0, \

func(GetRegRawInt(playout->I1))); \

break; \

}

#define PROCESS\_I1toI1Mem(name, func) PROCESS\_I1toI1Mem\_COMMON(name, func,)

#define PROCESS\_D1toD1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Double2, suffix); \

SetRegRawDouble(playout->D0, \

GetRegRawDouble(playout->D1)); \

break; \

}

#define PROCESS\_D1toD1(name, func) PROCESS\_D1toD1\_COMMON(name, func,)

#define PROCESS\_D1toD1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Double2, suffix); \

SetRegRawDouble(playout->D0, \

func(GetRegRawDouble(playout->D1))); \

break; \

}

#define PROCESS\_F1toF1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float2, suffix); \

SetRegRawFloat(playout->F0, \

func(GetRegRawFloat(playout->F1))); \

break; \

}

#define PROCESS\_D1toF1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float1Double1, suffix); \

SetRegRawFloat(playout->F0, \

func(GetRegRawDouble(playout->D1))); \

break; \

}

#define PROCESS\_I1toF1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float1Int1, suffix); \

SetRegRawFloat(playout->F0, \

func(GetRegRawInt(playout->I1))); \

break; \

}

#define PROCESS\_D1toD1Mem(name, func) PROCESS\_D1toD1Mem\_COMMON(name, func,)

#define PROCESS\_F1toF1Mem(name, func) PROCESS\_F1toF1Mem\_COMMON(name, func,)

#define PROCESS\_D1toF1Mem(name, func) PROCESS\_D1toF1Mem\_COMMON(name, func,)

#define PROCESS\_I1toF1Mem(name, func) PROCESS\_I1toF1Mem\_COMMON(name, func,)

#define PROCESS\_IP\_TARG\_ASM\_IMPL(name, func, layoutSize) \

case OpCodeAsmJs::name: \

{ \

Assert(!switchProfileMode); \

ip = func<layoutSize, INTERPRETERPROFILE>(ip); \

if (switchProfileMode) \

{ \

m\_reader.SetIP(ip); \

return nullptr; \

} \

break; \

}

#define PROCESS\_IP\_TARG\_ASM\_COMMON(name, func, suffix) PROCESS\_IP\_TARG\_ASM##suffix(name, func)

#define PROCESS\_IP\_TARG\_ASM\_Large(name, func) PROCESS\_IP\_TARG\_ASM\_IMPL(name, func, Js::LargeLayout)

#define PROCESS\_IP\_TARG\_ASM\_Medium(name, func) PROCESS\_IP\_TARG\_ASM\_IMPL(name, func, Js::MediumLayout)

#define PROCESS\_IP\_TARG\_ASM\_Small(name, func) PROCESS\_IP\_TARG\_ASM\_IMPL(name, func, Js::SmallLayout)

#define PROCESS\_D1toI1Scr\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Double1, suffix); \

SetRegRawInt(playout->I0, \

func(GetRegRawDouble(playout->D1))); \

break; \

}

#define PROCESS\_D1toI1Scr(name, func) PROCESS\_D1toI1Scr\_COMMON(name, func,)

#define PROCESS\_F1toI1Scr\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Float1, suffix); \

SetRegRawInt(playout->I0, \

func(GetRegRawFloat(playout->F1))); \

break; \

}

#define PROCESS\_F1toI1Scr(name, func) PROCESS\_F1toI1Scr\_COMMON(name, func,)

#define PROCESS\_I1toD1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Double1Int1, suffix); \

SetRegRawDouble(playout->D0, \

func(GetRegRawInt(playout->I1))); \

break; \

}

#define PROCESS\_I1toD1Mem(name, func) PROCESS\_I1toD1Mem\_COMMON(name, func,)

#define PROCESS\_U1toD1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Double1Int1, suffix); \

SetRegRawDouble(playout->D0, \

func((unsigned int)GetRegRawInt(playout->I1)) ); \

break; \

}

#define PROCESS\_F1toD1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Double1Float1, suffix); \

SetRegRawDouble(playout->D0, \

func((float)GetRegRawFloat(playout->F1))); \

break; \

}

#define PROCESS\_U1toD1Mem(name, func) PROCESS\_U1toD1Mem\_COMMON(name, func,)

#define PROCESS\_F1toD1Mem(name, func) PROCESS\_F1toD1Mem\_COMMON(name, func,)

#define PROCESS\_R1toD1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Double1Reg1, suffix); \

SetRegRawDouble(playout->D0, \

func(GetReg(playout->R1),scriptContext)); \

break; \

}

#define PROCESS\_R1toD1Mem(name, func) PROCESS\_R1toD1Mem\_COMMON(name, func,)

#define PROCESS\_R1toF1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float1Reg1, suffix); \

SetRegRawFloat(playout->F0, \

(float)func(GetReg(playout->R1), scriptContext)); \

break; \

}

#define PROCESS\_R1toF1Mem(name, func) PROCESS\_R1toF1Mem\_COMMON(name, func,)

#define PROCESS\_R1toI1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Reg1, suffix); \

SetRegRawInt(playout->I0, \

func(GetReg(playout->R1),scriptContext)); \

break; \

}

#define PROCESS\_R1toI1Mem(name, func) PROCESS\_R1toI1Mem\_COMMON(name, func,)

#define PROCESS\_D1toR1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Double1, suffix); \

SetRegRaw(playout->R0, \

func(GetRegRawDouble(playout->D1),scriptContext)); \

break; \

}

#define PROCESS\_F1toR1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Float1, suffix); \

SetRegRawFloat(playout->R0, \

GetRegRawFloat(playout->F1)); \

break; \

}

#define PROCESS\_D1toR1Mem(name, func) PROCESS\_D1toR1Mem\_COMMON(name, func,)

#define PROCESS\_F1toR1Mem(name, func) PROCESS\_F1toR1Mem\_COMMON(name, func,)

#define PROCESS\_C1toI1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Const1, suffix); \

SetRegRawInt( playout->I0, \

func( playout->C1 )); \

break; \

}

#define PROCESS\_C1toI1Mem(name, func) PROCESS\_C1toI1Mem\_COMMON(name, func,)

#define PROCESS\_C1toI1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Const1, suffix); \

SetRegRawInt( playout->I0, playout->C1 ); \

break; \

}

#define PROCESS\_C1toI1(name, func) PROCESS\_C1toI1\_COMMON(name, func,)

#define PROCESS\_I1toR1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Int1, suffix); \

SetRegRawInt(playout->R0, \

GetRegRawInt(playout->I1)); \

break; \

}

#define PROCESS\_I1toR1Mem(name, func) PROCESS\_I1toR1Mem\_COMMON(name, func,)

#define PROCESS\_I1toR1Out\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Int1, suffix); \

func(playout->R0, GetRegRawInt(playout->I1)); \

break; \

}

#define PROCESS\_F1toR1Out\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Float1, suffix); \

func(playout->R0, GetRegRawFloat(playout->F1)); \

break; \

}

#define PROCESS\_D1toR1Out\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Double1, suffix); \

func(playout->R0, GetRegRawDouble(playout->D1)); \

break; \

}

#define PROCESS\_D2toI1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Double2, suffix); \

SetRegRawInt(playout->I0, \

func(GetRegRawDouble(playout->D1),GetRegRawDouble(playout->D2))); \

break; \

}

#define PROCESS\_F2toI1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Float2, suffix); \

SetRegRawInt(playout->I0, \

func(GetRegRawFloat(playout->F1), GetRegRawFloat(playout->F2))); \

break; \

}

#define PROCESS\_D2toI1Mem(name, func) PROCESS\_D2toI1Mem\_COMMON(name, func,)

#define PROCESS\_F2toI1Mem(name, func) PROCESS\_F2toI1Mem\_COMMON(name, func,)

#define PROCESS\_BR\_ASM\_Mem\_COMMON(name, func,suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, BrInt2, suffix); \

if (func(GetRegRawInt(playout->I1), GetRegRawInt(playout->I2))) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BR\_ASM\_Mem(name, func) PROCESS\_BR\_ASM\_Mem\_COMMON(name, func,)

#define PROCESS\_BR\_ASM\_MemStack\_COMMON(name, func,suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, BrInt1, suffix); \

if (GetRegRawInt(playout->I1)) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BR\_ASM\_MemStack(name, func) PROCESS\_BR\_ASM\_MemStack\_COMMON(name, func,)

#define PROCESS\_TEMPLATE\_ASMJS\_COMMON(name, func, layout, suffix, type) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, layout, suffix); \

func<OpLayout##layout##suffix,type>(playout); \

break; \

}

// initializers

#define PROCESS\_SIMD\_F4\_1toR1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Float32x4\_1, suffix); \

func(playout->R0, GetRegRawSimd(playout->F4\_1)); \

break; \

}

#define PROCESS\_SIMD\_F4\_1toR1Mem(name, func, suffix) PROCESS\_F4\_1toR1Mem\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_1toR1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Int32x4\_1, suffix); \

func(playout->R0, GetRegRawSimd(playout->I4\_1)); \

break; \

}

#define PROCESS\_SIMD\_I4\_1toR1Mem(name, func, suffix) PROCESS\_I4\_1toR1Mem\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_D2\_1toR1Mem\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Reg1Float64x2\_1, suffix); \

func(playout->R0, GetRegRawSimd(playout->D2\_1)); \

break; \

}

#define PROCESS\_SIMD\_D2\_1toR1Mem(name, func, suffix) PROCESS\_D2\_1toR1Mem\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I1toI4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_1Int1, suffix); \

SetRegRawSimd(playout->I4\_0, func(GetRegRawInt(playout->I1))); \

break; \

}

#define PROCESS\_SIMD\_I1toI4\_1(name, func, suffix) PROCESS\_SIMD\_I1toI4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_F1toF4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_1Float1, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawFloat(playout->F1))); \

break; \

}

#define PROCESS\_SIMD\_F1toF4\_1(name, func, suffix) PROCESS\_SIMD\_F1toF4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_D1toD2\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_1Double1, suffix); \

SetRegRawSimd(playout->D2\_0, func(GetRegRawDouble(playout->D1))); \

break; \

}

#define PROCESS\_SIMD\_D1toD2\_1(name, func, suffix) PROCESS\_SIMD\_D1toD2\_1\_COMMON(name, func, suffix)

//// Value transfer

#define PROCESS\_SIMD\_F4toF4\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_1Float4, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawFloat(playout->F1), GetRegRawFloat(playout->F2), GetRegRawFloat(playout->F3), GetRegRawFloat(playout->F4))); \

break; \

}

#define PROCESS\_SIMD\_F4toF4\_1(name, func, suffix) PROCESS\_SIMD\_F4toF4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4toI4\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_1Int4, suffix); \

SetRegRawSimd(playout->I4\_0, func(GetRegRawInt(playout->I1), GetRegRawInt(playout->I2), GetRegRawInt(playout->I3), GetRegRawInt(playout->I4))); \

break; \

}

#define PROCESS\_SIMD\_I4toI4\_1(name, func, suffix) PROCESS\_SIMD\_I4toI4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_D2toD2\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_1Double2, suffix); \

SetRegRawSimd(playout->D2\_0, func(GetRegRawDouble(playout->D1), GetRegRawDouble(playout->D2))); \

break; \

}

#define PROCESS\_SIMD\_D2toD2\_1(name, func, suffix) PROCESS\_SIMD\_D2toD2\_1\_COMMON(name, func, suffix)

//// Conversions

#define PROCESS\_SIMD\_D2\_1toF4\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_1Float64x2\_1, suffix); \

SetRegRawSimd(playout->F4\_0, \

func(GetRegRawSimd(playout->D2\_1))); \

break; \

}

#define PROCESS\_SIMD\_D2\_1toF4\_1(name, func, suffix) PROCESS\_SIMD\_D2\_1toF4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_1toF4\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_1Int32x4\_1, suffix); \

SetRegRawSimd(playout->F4\_0, \

func(GetRegRawSimd(playout->I4\_1))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1toF4\_1\_1(name, func, suffix) PROCESS\_SIMD\_I4\_1toF4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_D2\_1toI4\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_1Float64x2\_1, suffix); \

SetRegRawSimd(playout->I4\_0, \

func(GetRegRawSimd(playout->D2\_1))); \

break; \

}

#define PROCESS\_SIMD\_D2\_1toI4\_1(name, func, suffix) PROCESS\_SIMD\_D2\_1toI4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_F4\_1toI4\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_1Float32x4\_1, suffix); \

SetRegRawSimd(playout->I4\_0, \

func(GetRegRawSimd(playout->F4\_1))); \

break; \

}

#define PROCESS\_SIMD\_F4\_1toI4\_1(name, func, suffix) PROCESS\_SIMD\_F4\_1toI4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_F4\_1toD2\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_1Float32x4\_1, suffix); \

SetRegRawSimd(playout->D2\_0, \

func(GetRegRawSimd(playout->F4\_1))); \

break; \

}

#define PROCESS\_SIMD\_F4\_1toD2\_1(name, func, suffix) PROCESS\_SIMD\_F4\_1toD2\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_1toD2\_1\_COMMON(name, func, suffix)\

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_1Int32x4\_1, suffix); \

SetRegRawSimd(playout->D2\_0, \

func(GetRegRawSimd(playout->I4\_1))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1toD2\_1(name, func, suffix) PROCESS\_SIMD\_I4\_1toD2\_1\_COMMON(name, func, suffix)

// unary ops

#define PROCESS\_SIMD\_F4\_1toF4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_2, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawSimd(playout->F4\_1))); \

break; \

}

#define PROCESS\_SIMD\_F4\_1toF4\_1 (name, func,) PROCESS\_F4\_1toF4\_1\_COMMON(name, func,)

#define PROCESS\_SIMD\_I4\_1toI4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_2, suffix); \

SetRegRawSimd(playout->I4\_0, func(GetRegRawSimd(playout->I4\_1))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1toI4\_1 (name, func,) PROCESS\_I4\_1toI4\_1\_COMMON(name, func,)

#define PROCESS\_SIMD\_D2\_1toD2\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_2, suffix); \

SetRegRawSimd(playout->D2\_0, func(GetRegRawSimd(playout->D2\_1))); \

break; \

}

#define PROCESS\_SIMD\_D2\_1toD2\_1 (name, func,) PROCESS\_D2\_1toD2\_1\_COMMON(name, func,)

// binary ops

#define PROCESS\_SIMD\_F4\_2toF4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_3, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawSimd(playout->F4\_1), GetRegRawSimd(playout->F4\_2))); \

break; \

}

#define PROCESS\_SIMD\_F4\_2toF4\_1(name, func, suffix) PROCESS\_SIMD\_F4\_2toF4\_1COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_2toI4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_3, suffix); \

SetRegRawSimd(playout->I4\_0, func(GetRegRawSimd(playout->I4\_1), GetRegRawSimd(playout->I4\_2))); \

break; \

}

#define PROCESS\_SIMD\_I4\_2toI4\_1(name, func, suffix) PROCESS\_SIMD\_I4\_2toI4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_D2\_2toD2\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_3, suffix); \

SetRegRawSimd(playout->D2\_0, func(GetRegRawSimd(playout->D2\_1), GetRegRawSimd(playout->D2\_2))); \

break; \

}

#define PROCESS\_SIMD\_D2\_2toD2\_1(name, func, suffix) PROCESS\_SIMD\_D2\_2toD2\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_SetLane\_F4\_COMMON(name, field, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_2Float1, suffix); \

AsmJsSIMDValue val = GetRegRawSimd(playout->F4\_1); \

val.f32[field] = GetRegRawFloat(playout->F2); \

SetRegRawSimd(playout->F4\_0, val); \

break; \

}

#define PROCESS\_SIMD\_SetLane\_F4(name, field, suffix) PROCESS\_SIMD\_SetLane\_F4\_COMMON(name, field, suffix)

#define PROCESS\_SIMD\_SetLane\_I4\_COMMON(name, field, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_2Int1, suffix); \

AsmJsSIMDValue val = GetRegRawSimd(playout->I4\_1); \

val.i32[field] = GetRegRawInt(playout->I2); \

SetRegRawSimd(playout->I4\_0, val); \

break; \

}

#define PROCESS\_SIMD\_SetLane\_I4(name, field, suffix) PROCESS\_SIMD\_SetLane\_I4\_COMMON(name, field, suffix)

#define PROCESS\_SIMD\_SetLane\_D2\_COMMON(name, field, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_2Double1, suffix); \

AsmJsSIMDValue val = GetRegRawSimd(playout->D2\_1); \

val.f64[field] = GetRegRawDouble(playout->D2); \

SetRegRawSimd(playout->D2\_0, val); \

break; \

}

#define PROCESS\_SIMD\_SetLane\_D2(name, field, suffix) PROCESS\_SIMD\_SetLane\_D2\_COMMON(name, field, suffix)

// ternary

#define PROCESS\_SIMD\_F4\_3toF4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_4, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawSimd(playout->F4\_1), GetRegRawSimd(playout->F4\_2), GetRegRawSimd(playout->F4\_3))); \

break; \

}

#define PROCESS\_SIMD\_F4\_3toF4\_1(name, func, suffix) PROCESS\_SIMD\_F4\_3toF4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_3toI4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_4, suffix); \

SetRegRawSimd(playout->I4\_0, func(GetRegRawSimd(playout->I4\_1), GetRegRawSimd(playout->I4\_2), GetRegRawSimd(playout->I4\_3))); \

break; \

}

#define PROCESS\_SIMD\_I4\_3toI4\_1(name, func, suffix) PROCESS\_SIMD\_I4\_3toI4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_D2\_3toD2\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_4, suffix); \

SetRegRawSimd(playout->D2\_0, func(GetRegRawSimd(playout->D2\_1), GetRegRawSimd(playout->D2\_2), GetRegRawSimd(playout->D2\_3))); \

break; \

}

#define PROCESS\_SIMD\_D2\_3toD2\_1(name, func, suffix) PROCESS\_SIMD\_D2\_3toD2\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_1F4\_2toF4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_1Int32x4\_1Float32x4\_2, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawSimd(playout->I4\_1), GetRegRawSimd(playout->F4\_2), GetRegRawSimd(playout->F4\_3))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1F4\_2toF4\_1(name, func, suffix) PROCESS\_SIMD\_I4\_1F4\_2toF4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_1D2\_2toD2\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_1Int32x4\_1Float64x2\_2, suffix); \

SetRegRawSimd(playout->D2\_0, func(GetRegRawSimd(playout->I4\_1), GetRegRawSimd(playout->D2\_2), GetRegRawSimd(playout->D2\_3))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1D2\_2toD2\_1(name, func, suffix) PROCESS\_SIMD\_I4\_1D2\_2toD2\_1\_COMMON(name, func, suffix)

// Extract Lane / Replace Lane

#define PROCESS\_SIMD\_I4\_1I1toI1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Int32x4\_1Int1, suffix); \

SetRegRawInt(playout->I0, func(GetRegRawSimd(playout->I4\_1), GetRegRawInt(playout->I2))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1I1toI1(name, func, suffix) PROCESS\_SIMD\_I4\_1I1toI1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_1I2toI4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_2Int2, suffix); \

SetRegRawSimd(playout->I4\_0, func(GetRegRawSimd(playout->I4\_1), GetRegRawInt(playout->I2), GetRegRawInt(playout->I3))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1I2toI4\_1(name, func, suffix) PROCESS\_SIMD\_I4\_1I2toI4\_1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_F4\_1I1toF1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float1Float32x4\_1Int1, suffix); \

SetRegRawFloat(playout->F0, func(GetRegRawSimd(playout->F4\_1), GetRegRawInt(playout->I2))); \

break; \

}

#define PROCESS\_SIMD\_F4\_1I1toF1(name, func, suffix) PROCESS\_SIMD\_F4\_1I1toF1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_F4\_1I1F1toF4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_2Int1Float1, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawSimd(playout->F4\_1), GetRegRawInt(playout->I2), GetRegRawFloat(playout->F3))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1I2toI4\_1(name, func, suffix) PROCESS\_SIMD\_I4\_1I2toI4\_1\_COMMON(name, func, suffix)

// signmask

#define PROCESS\_SIMD\_F4\_1toI1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Float32x4\_1, suffix); \

SetRegRawInt(playout->I0, func(GetRegRawSimd(playout->F4\_1))); \

break; \

}

#define PROCESS\_SIMD\_F4\_1toI1(name, func, suffix) PROCESS\_SIMD\_F4\_1toI1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_I4\_1toI1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Int32x4\_1, suffix); \

SetRegRawInt(playout->I0, func(GetRegRawSimd(playout->I4\_1))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1toI1(name, func, suffix) PROCESS\_SIMD\_I4\_1toI1\_COMMON(name, func, suffix)

#define PROCESS\_SIMD\_D2\_1toI1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int1Float64x2\_1, suffix); \

SetRegRawInt(playout->I0, func(GetRegRawSimd(playout->D2\_1))); \

break; \

}

#define PROCESS\_SIMD\_D2\_1toI1(name, func, suffix) PROCESS\_SIMD\_D2\_1toI1\_COMMON(name, func, suffix)

// f4swizzle

#define PROCESS\_SIMD\_F4\_1I4toF4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_2Int4, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawSimd(playout->F4\_1), GetRegRawSimd(playout->F4\_1), GetRegRawInt(playout->I2), GetRegRawInt(playout->I3), GetRegRawInt(playout->I4), GetRegRawInt(playout->I5))); \

break; \

}

#define PROCESS\_SIMD\_F4\_1I4toF4\_1(name, func, suffix) PROCESS\_SIMD\_F4\_1I4toF4\_1\_COMMON(name, func, suffix)

// f4shuffle

#define PROCESS\_SIMD\_F4\_2I4toF4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float32x4\_3Int4, suffix); \

SetRegRawSimd(playout->F4\_0, func(GetRegRawSimd(playout->F4\_1), GetRegRawSimd(playout->F4\_2), GetRegRawInt(playout->I3), GetRegRawInt(playout->I4), GetRegRawInt(playout->I5), GetRegRawInt(playout->I6))); \

break; \

}

#define PROCESS\_SIMD\_F4\_1I4toF4\_1(name, func, suffix) PROCESS\_SIMD\_F4\_1I4toF4\_1\_COMMON(name, func, suffix)

// i4swizzle

#define PROCESS\_SIMD\_I4\_1I4toI4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_2Int4, suffix); \

SetRegRawSimd(playout->I4\_0, func(GetRegRawSimd(playout->I4\_1), GetRegRawSimd(playout->I4\_1), GetRegRawInt(playout->I2), GetRegRawInt(playout->I3), GetRegRawInt(playout->I4), GetRegRawInt(playout->I5))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1I4toI4\_1(name, func, suffix) PROCESS\_SIMD\_I4\_1I4toI4\_1\_COMMON(name, func, suffix)

// i4shuffle

#define PROCESS\_SIMD\_I4\_2I4toI4\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Int32x4\_3Int4, suffix); \

SetRegRawSimd(playout->I4\_0, func(GetRegRawSimd(playout->I4\_1), GetRegRawSimd(playout->I4\_2), GetRegRawInt(playout->I3), GetRegRawInt(playout->I4), GetRegRawInt(playout->I5), GetRegRawInt(playout->I6))); \

break; \

}

#define PROCESS\_SIMD\_I4\_1I4toI4\_1(name, func, suffix) PROCESS\_SIMD\_I4\_1I4toI4\_1\_COMMON(name, func, suffix)

// d2swizzle

#define PROCESS\_SIMD\_D2\_1I2toD2\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_2Int2, suffix); \

SetRegRawSimd(playout->D2\_0, func<2>(GetRegRawSimd(playout->D2\_1), GetRegRawSimd(playout->D2\_1), GetRegRawInt(playout->I2), GetRegRawInt(playout->I3), 0, 0)); \

break; \

}

#define PROCESS\_SIMD\_D2\_1I2toD2\_1(name, func, suffix) PROCESS\_SIMD\_D2\_1I2toD2\_1\_COMMON(name, func, suffix)

// d2shuffle

#define PROCESS\_SIMD\_D2\_2I2toD2\_1\_COMMON(name, func, suffix) \

case OpCodeAsmJs::name: \

{ \

PROCESS\_READ\_LAYOUT\_ASMJS(name, Float64x2\_3Int2, suffix); \

SetRegRawSimd(playout->D2\_0, func<2>(GetRegRawSimd(playout->D2\_1), GetRegRawSimd(playout->D2\_2), GetRegRawInt(playout->I3), GetRegRawInt(playout->I4), 0, 0)); \

break; \

}

#define PROCESS\_SIMD\_D2\_2I4toD2\_1(name, func, suffix) PROCESS\_SIMD\_D2\_2I2toD2\_1COMMON(name, func, suffix)

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "EHBailoutData.h"

#include "Library\JavascriptRegularExpression.h"

#if DBG\_DUMP

#include "ByteCode\OpCodeUtilAsmJs.h"

#endif

#include "Language\InterpreterStackFrame.h"

#include "Library\JavascriptGeneratorFunction.h"

///----------------------------------------------------------------------------

///

/// macros PROCESS\_INtoOUT

///

/// This set of macros defines standard patterns for processing OpCodes in

/// RcInterpreter::Run(). Each macro is named for "in" - "out":

/// - A: Var

/// - I: Integer

/// - R: Double

/// - X: Nothing

///

/// Examples:

/// - "A2toA1" reads two registers, each storing an Var, and writes a single

/// register with a new Var.

/// - "A1I1toA2" reads two registers, first an Var and second an Int32, then

/// writes two Var registers.

///

/// Although these could use lookup tables to standard OpLayout types, this

/// additional indirection would slow the main interpreter loop further by

/// preventing the main 'switch' statement from using the OpCode to become a

/// direct local-function jump.

///----------------------------------------------------------------------------

#define PROCESS\_FALLTHROUGH(name, func) \

case OpCode::name:

#define PROCESS\_FALLTHROUGH\_COMMON(name, func, suffix) \

case OpCode::name:

#define PROCESS\_READ\_LAYOUT(name, layout, suffix) \

CompileAssert(OpCodeInfo<OpCode::name>::Layout == OpLayoutType::layout); \

const unaligned OpLayout##layout##suffix \* playout = m\_reader.layout##suffix(ip); \

Assert((playout != nullptr) == (Js::OpLayoutType::##layout != Js::OpLayoutType::Empty)); // Make sure playout is used

#define PROCESS\_NOP\_COMMON(name, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

break; \

}

#define PROCESS\_NOP(name, layout) PROCESS\_NOP\_COMMON(name, layout,)

#define PROCESS\_CUSTOM\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

func(playout); \

break; \

}

#define PROCESS\_CUSTOM(name, func, layout) PROCESS\_CUSTOM\_COMMON(name, func, layout,)

#define PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, regslot, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

func(playout); \

break; \

}

#define PROCESS\_CUSTOM\_L(name, func, layout, regslot) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, regslot,)

#define PROCESS\_CUSTOM\_L\_Arg\_COMMON(name, func, suffix) PROCESS\_CUSTOM\_L\_COMMON(name, func, Arg, Arg, suffix)

#define PROCESS\_CUSTOM\_L\_Arg2\_COMMON(name, func, layout, suffix) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, Arg, suffix)

#define PROCESS\_CUSTOM\_L\_Arg(name, func) PROCESS\_CUSTOM\_L\_COMMON(name, func, Arg, Arg,)

#define PROCESS\_CUSTOM\_ArgNoSrc\_COMMON(name, func, suffix) PROCESS\_CUSTOM\_COMMON(name, func, ArgNoSrc, suffix)

#define PROCESS\_CUSTOM\_ArgNoSrc(name, func) PROCESS\_CUSTOM\_COMMON(name, func, ArgNoSrc,)

#define PROCESS\_CUSTOM\_L\_R0\_COMMON(name, func, layout, suffix) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, R0, suffix)

#define PROCESS\_CUSTOM\_L\_R0(name, func, layout) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, R0,)

#define PROCESS\_CUSTOM\_L\_Value\_COMMON(name, func, layout, suffix) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, Value, suffix)

#define PROCESS\_CUSTOM\_L\_Value(name, func, layout) PROCESS\_CUSTOM\_L\_COMMON(name, func, layout, Value,)

#define PROCESS\_TRY(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Br,); \

func(playout); \

ip = m\_reader.GetIP(); \

break; \

}

#define PROCESS\_EMPTY(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Empty, ); \

func(); \

ip = m\_reader.GetIP(); \

break; \

}

#define PROCESS\_TRYBR2\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg2, suffix); \

func((const byte\*)(playout + 1), playout->RelativeJumpOffset, playout->R1, playout->R2); \

ip = m\_reader.GetIP(); \

break; \

}

#define PROCESS\_CALL\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

func(playout); \

break; \

}

#define PROCESS\_CALL(name, func, layout) PROCESS\_CALL\_COMMON(name, func, layout,)

#define PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, flags, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

func(playout, flags); \

break; \

}

#define PROCESS\_CALL\_FLAGS(name, func, layout, regslot) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, regslot,)

#define PROCESS\_CALL\_FLAGS\_None\_COMMON(name, func, layout, suffix) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_None, suffix)

#define PROCESS\_CALL\_FLAGS\_None(name, func, layout) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_None,)

#define PROCESS\_CALL\_FLAGS\_Value\_COMMON(name, func, layout, suffix) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_Value, suffix)

#define PROCESS\_CALL\_FLAGS\_Value(name, func, layout) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_Value,)

#define PROCESS\_CALL\_FLAGS\_CallEval\_COMMON(name, func, layout, suffix) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_ExtraArg, suffix)

#define PROCESS\_CALL\_FLAGS\_CallEval(name, func, layout) PROCESS\_CALL\_FLAGS\_COMMON(name, func, layout, CallFlags\_ExtraArg,)

#define PROCESS\_A1toXX\_ALLOW\_STACK\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

func(GetRegAllowStackVar(playout->R0)); \

break; \

}

#define PROCESS\_A1toXX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

func(GetReg(playout->R0)); \

break; \

}

#define PROCESS\_A1toXX(name, func) PROCESS\_A1toXX\_COMMON(name, func,)

#define PROCESS\_A1toXXMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

func(GetReg(playout->R0), GetScriptContext()); \

break; \

}

#define PROCESS\_A1toXXMem(name, func) PROCESS\_A1toXXMem\_COMMON(name, func,)

#define PROCESS\_A1toXXMemNonVar\_COMMON(name, func, type, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

func((type)GetNonVarReg(playout->R0), GetScriptContext()); \

break; \

}

#define PROCESS\_A1toXXMemNonVar(name, func, type) PROCESS\_A1toXXMemNonVar\_COMMON(name, func, type,)

#define PROCESS\_XXtoA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

SetReg(playout->R0, \

func()); \

break; \

}

#define PROCESS\_XXtoA1(name, func) PROCESS\_XXtoA1\_COMMON(name, func,)

#define PROCESS\_XXtoA1NonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

SetNonVarReg(playout->R0, \

func()); \

break; \

}

#define PROCESS\_XXtoA1NonVar(name, func) PROCESS\_XXtoA1NonVar\_COMMON(name, func,)

#define PROCESS\_XXtoA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1, suffix); \

SetReg(playout->R0, \

func(GetScriptContext())); \

break; \

}

#define PROCESS\_XXtoA1Mem(name, func) PROCESS\_XXtoA1Mem\_COMMON(name, func,)

#define PROCESS\_A1toA1\_ALLOW\_STACK\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetRegAllowStackVar(playout->R0, \

func(GetRegAllowStackVar(playout->R1))); \

break; \

}

#define PROCESS\_A1toA1\_ALLOW\_STACK(name, func) PROCESS\_A1toA1\_ALLOW\_STACK\_COMMON(name, func,)

#define PROCESS\_A1toA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1))); \

break; \

}

#define PROCESS\_A1toA1(name, func) PROCESS\_A1toA1\_COMMON(name, func,)

#define PROCESS\_A1toA1Profiled\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ProfiledReg2, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), playout->profileId)); \

break; \

}

#define PROCESS\_A1toA1Profiled(name, func) PROCESS\_A1toA1Profiled\_COMMON(name, func,)

#define PROCESS\_A1toA1CallNoArg\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetReg(playout->R0, \

func(playout)); \

break; \

}

#define PROCESS\_A1toA1CallNoArg(name, func, layout) PROCESS\_A1toA1CallNoArg\_COMMON(name, func, layout,)

#define PROCESS\_A1toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1),GetScriptContext())); \

break; \

}

#define PROCESS\_A1toA1Mem(name, func) PROCESS\_A1toA1Mem\_COMMON(name, func,)

#define PROCESS\_A1toA1NonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetNonVarReg(playout->R0, \

func(GetNonVarReg(playout->R1))); \

break; \

}

#define PROCESS\_A1toA1NonVar(name, func) PROCESS\_A1toA1NonVar\_COMMON(name, func,)

#define PROCESS\_A1toA1MemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetNonVarReg(playout->R0, \

func(GetNonVarReg(playout->R1),GetScriptContext())); \

break; \

}

#define PROCESS\_A1toA1MemNonVar(name, func) PROCESS\_A1toA1MemNonVar\_COMMON(name, func,)

#define PROCESS\_INNERtoA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetReg(playout->R0, InnerScopeFromIndex(playout->C1)); \

break; \

}

#define PROCESS\_INNERtoA1(name, fun) PROCESS\_INNERtoA1\_COMMON(name, func,)

#define PROCESS\_U1toINNERMemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Unsigned1, suffix); \

SetInnerScopeFromIndex(playout->C1, func(GetScriptContext())); \

break; \

}

#define PROCESS\_U1toINNERMemNonVar(name, func) PROCESS\_U1toINNERMemNonVar\_COMMON(name, func,)

#define PROCESS\_XXINNERtoA1MemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetNonVarReg(playout->R0, \

func(InnerScopeFromIndex(playout->C1), GetScriptContext())); \

break; \

}

#define PROCESS\_XXINNERtoA1MemNonVar(name, func) PROCESS\_XXINNERtoA1MemNonVar\_COMMON(name, func,)

#define PROCESS\_A1INNERtoA1MemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2Int1, suffix); \

SetNonVarReg(playout->R0, \

func(InnerScopeFromIndex(playout->C1), GetNonVarReg(playout->R1), GetScriptContext())); \

break; \

}

#define PROCESS\_A1LOCALtoA1MemNonVar(name, func) PROCESS\_A1LOCALtoA1MemNonVar\_COMMON(name, func,)

#define PROCESS\_LOCALI1toA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetReg(playout->R0, \

func(this->localClosure, playout->C1)); \

break; \

}

#define PROCESS\_LOCALI1toA1(name, func) PROCESS\_LOCALI1toA1\_COMMON(name, func,)

#define PROCESS\_A1I1toA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2Int1, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), playout->C1)); \

break; \

}

#define PROCESS\_A1I1toA1(name, func) PROCESS\_A1I1toA1\_COMMON(name, func,)

#define PROCESS\_A1I1toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2Int1, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), playout->C1, GetScriptContext())); \

break; \

}

#define PROCESS\_A1I1toA1Mem(name, func) PROCESS\_A1I1toA1Mem\_COMMON(name, func,)

#define PROCESS\_RegextoA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetReg(playout->R0, \

func(this->m\_functionBody->GetLiteralRegex(playout->C1), GetScriptContext())); \

break; \

}

#define PROCESS\_RegextoA1(name, func) PROCESS\_RegextoA1\_COMMON(name, func,)

#define PROCESS\_A2toXX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

func(GetReg(playout->R0), GetReg(playout->R1)); \

break; \

}

#define PROCESS\_A2toXX(name, func) PROCESS\_A2toXX\_COMMON(name, func,)

#define PROCESS\_A2toXXMemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

func(GetNonVarReg(playout->R0), GetNonVarReg(playout->R1), GetScriptContext()); \

break; \

}

#define PROCESS\_A2toXXMemNonVar(name, func) PROCESS\_A2toXXMemNonVar\_COMMON(name, func,)

#define PROCESS\_A1NonVarToA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2, suffix); \

SetReg(playout->R0, \

func(GetNonVarReg(playout->R1))); \

break; \

}

#define PROCESS\_A2NonVarToA1Reg\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetReg(playout->R0, \

func(GetNonVarReg(playout->R1), playout->R2)); \

break; \

}

#define PROCESS\_A2toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2),GetScriptContext())); \

break; \

}

#define PROCESS\_A2toA1Mem(name, func) PROCESS\_A2toA1Mem\_COMMON(name, func,)

#define PROCESS\_A2toA1MemProfiled\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ProfiledReg3, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2),GetScriptContext(), playout->profileId)); \

break; \

}

#define PROCESS\_A2toA1MemProfiled(name, func) PROCESS\_A2toA1MemProfiled\_COMMON(name, func,)

#define PROCESS\_A2toA1NonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetNonVarReg(playout->R0, \

func(GetNonVarReg(playout->R1), GetNonVarReg(playout->R2))); \

break; \

}

#define PROCESS\_A2toA1NonVar(name, func) PROCESS\_A2toA1NonVar\_COMMON(name, func,)

#define PROCESS\_A2toA1MemNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetNonVarReg(playout->R0, \

func(GetNonVarReg(playout->R1), GetNonVarReg(playout->R2),GetScriptContext())); \

break; \

}

#define PROCESS\_A2toA1MemNonVar(name, func) PROCESS\_A2toA1MemNonVar\_COMMON(name, func,)

#define PROCESS\_CMMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2), GetScriptContext()) ? JavascriptBoolean::OP\_LdTrue(GetScriptContext()) : \

JavascriptBoolean::OP\_LdFalse(GetScriptContext())); \

break; \

}

#define PROCESS\_CMMem(name, func) PROCESS\_CMMem\_COMMON(name, func,)

#define PROCESS\_ELEM\_RtU\_to\_XX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementRootU, suffix); \

func(playout->PropertyIdIndex); \

break; \

}

#define PROCESS\_ELEM\_RtU\_to\_XX(name, func) PROCESS\_ELEM\_RtU\_to\_XX\_COMMON(name, func,)

#define PROCESS\_ELEM\_C2\_to\_XX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementScopedC, suffix); \

func(GetEnvForEvalCode(), playout->PropertyIdIndex, GetReg(playout->Value)); \

break; \

}

#define PROCESS\_ELEM\_C2\_to\_XX(name, func) PROCESS\_ELEM\_C2\_to\_XX\_COMMON(name, func,)

#define PROCESS\_GET\_ELEM\_SLOT\_FB\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlot, suffix); \

SetReg(playout->Value, \

func((FrameDisplay\*)GetNonVarReg(playout->Instance), reinterpret\_cast<Js::FunctionProxy\*\*>(this->m\_functionBody->GetNestedFuncReference(playout->SlotIndex)))); \

break; \

}

#define PROCESS\_GET\_ELEM\_SLOT\_FB(name, func) PROCESS\_GET\_ELEM\_SLOT\_FB\_COMMON(name, func,)

#define PROCESS\_GET\_SLOT\_FB\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlotI1, suffix); \

SetReg(playout->Value, \

func(this->GetFrameDisplayForNestedFunc(), reinterpret\_cast<Js::FunctionProxy\*\*>(this->m\_functionBody->GetNestedFuncReference(playout->SlotIndex)))); \

break; \

}

#define PROCESS\_GET\_SLOT\_FB(name, func) PROCESS\_GET\_SLOT\_FB\_COMMON(name, func,)

#define PROCESS\_GET\_ELEM\_IMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementI, suffix); \

SetReg(playout->Value, \

func(GetReg(playout->Instance), GetReg(playout->Element), GetScriptContext())); \

break; \

}

#define PROCESS\_GET\_ELEM\_IMem(name, func) PROCESS\_GET\_ELEM\_IMem\_COMMON(name, func,)

#define PROCESS\_GET\_ELEM\_IMem\_Strict\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementI, suffix); \

SetReg(playout->Value, \

func(GetReg(playout->Instance), GetReg(playout->Element), GetScriptContext(), PropertyOperation\_StrictMode)); \

break; \

}

#define PROCESS\_GET\_ELEM\_IMem\_Strict(name, func) PROCESS\_GET\_ELEM\_IMem\_Strict\_COMMON(name, func,)

#define PROCESS\_BR(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Br,); \

ip = func(playout); \

break; \

}

#ifdef BYTECODE\_BRANCH\_ISLAND

#define PROCESS\_BRLONG(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrLong,); \

ip = func(playout); \

break; \

}

#endif

#define PROCESS\_BRS(name,func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrS,); \

if (func(playout->val,GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRB\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg1, suffix); \

if (func(GetReg(playout->R1))) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRB(name, func) PROCESS\_BRB\_COMMON(name, func,)

#define PROCESS\_BRB\_ALLOW\_STACK\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg1, suffix); \

if (func(GetRegAllowStackVar(playout->R1))) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRB\_ALLOW\_STACK(name, func) PROCESS\_BRB\_ALLOW\_STACK\_COMMON(name, func,)

#define PROCESS\_BRBS\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg1, suffix); \

if (func(GetReg(playout->R1), GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRBS(name, func) PROCESS\_BRBS\_COMMON(name, func,)

#define PROCESS\_BRBReturnP1toA1\_COMMON(name, func, type, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg2, suffix); \

SetReg(playout->R1, func((type)GetNonVarReg(playout->R2))); \

if (!GetReg(playout->R1)) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRBReturnP1toA1(name, func, type) PROCESS\_BRBReturnP1toA1\_COMMON(name, func, type,)

#define PROCESS\_BRBMem\_ALLOW\_STACK\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg1, suffix); \

if (func(GetRegAllowStackVar(playout->R1),GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRBMem\_ALLOW\_STACK(name, func) PROCESS\_BRBMem\_ALLOW\_STACK\_COMMON(name, func,)

#define PROCESS\_BRCMem\_COMMON(name, func,suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrReg2, suffix); \

if (func(GetReg(playout->R1), GetReg(playout->R2),GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRCMem(name, func) PROCESS\_BRCMem\_COMMON(name, func,)

#define PROCESS\_BRPROP(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrProperty,); \

if (func(GetReg(playout->Instance), playout->PropertyIdIndex, GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRLOCALPROP(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrLocalProperty,); \

if (func(this->localClosure, playout->PropertyIdIndex, GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_BRENVPROP(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, BrEnvProperty,); \

if (func(LdEnv(), playout->SlotIndex, playout->PropertyIdIndex, GetScriptContext())) \

{ \

ip = m\_reader.SetCurrentRelativeOffset(ip, playout->RelativeJumpOffset); \

} \

break; \

}

#define PROCESS\_W1(name, func) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, W1,); \

func(playout->C1, GetScriptContext()); \

break; \

}

#define PROCESS\_U1toA1\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetReg(playout->R0, \

func(playout->C1,GetScriptContext())); \

break; \

}

#define PROCESS\_U1toA1(name, func) PROCESS\_U1toA1\_COMMON(name, func,)

#define PROCESS\_U1toA1NonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetNonVarReg(playout->R0, \

func(playout->C1)); \

break; \

}

#define PROCESS\_U1toA1NonVar(name, func) PROCESS\_U1toA1NonVar\_COMMON(name, func,)

#define PROCESS\_U1toA1NonVar\_FuncBody\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

SetNonVarReg(playout->R0, \

func(playout->C1,GetScriptContext(), this->m\_functionBody)); \

break; \

}

#define PROCESS\_U1toA1NonVar\_FuncBody(name, func) PROCESS\_U1toA1NonVar\_FuncBody\_COMMON(name, func,)

#define PROCESS\_A1I2toXXNonVar\_FuncBody(name, func) PROCESS\_A1I2toXXNonVar\_FuncBody\_COMMON(name, func,)

#define PROCESS\_A1I2toXXNonVar\_FuncBody\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3, suffix); \

func(playout->R0, playout->R1, playout->R2, GetScriptContext(), this->m\_functionBody); \

break; \

}

#define PROCESS\_A1U1toXX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg1Unsigned1, suffix); \

func(GetNonVarReg(playout->R0), playout->C1); \

break; \

}

#define PROCESS\_A1U1toXX(name, func) PROCESS\_A1U1toXX\_COMMON(name, func,)

#define PROCESS\_EnvU1toXX\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Unsigned1, suffix); \

func(LdEnv(), playout->C1); \

break; \

}

#define PROCESS\_EnvU1toXX(name, func) PROCESS\_EnvU1toXX\_COMMON(name, func,)

#define PROCESS\_GET\_ELEM\_SLOTNonVar\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetNonVarReg(playout->Value, func(GetNonVarReg(playout->Instance), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_SLOTNonVar(name, func, layout) PROCESS\_GET\_ELEM\_SLOTNonVar\_COMMON(name, func, layout,)

#define PROCESS\_GET\_ELEM\_LOCALSLOTNonVar\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetNonVarReg(playout->Value, func((Var\*)GetLocalClosure(), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_LOCALSLOTNonVar(name, func, layout) PROCESS\_GET\_ELEM\_LOCALSLOTNonVar\_COMMON(name, func, layout,)

#define PROCESS\_GET\_ELEM\_INNERSLOTNonVar\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetNonVarReg(playout->Value, func(InnerScopeFromIndex(playout->SlotIndex1), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_INNERSLOTNonVar(name, func, layout) PROCESS\_GET\_ELEM\_INNERSLOTNonVar\_COMMON(name, func, layout,)

#define PROCESS\_GET\_ELEM\_ENVSLOTNonVar\_COMMON(name, func, layout, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, layout, suffix); \

SetNonVarReg(playout->Value, func(LdEnv(), playout)); \

break; \

}

#define PROCESS\_GET\_ELEM\_ENVSLOTNonVar(name, func, layout) PROCESS\_GET\_ELEM\_ENVSLOTNonVar\_COMMON(name, func, layout,)

#define PROCESS\_SET\_ELEM\_SLOTNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlot, suffix); \

func(GetNonVarReg(playout->Instance), playout->SlotIndex, GetRegAllowStackVarEnableOnly(playout->Value)); \

break; \

}

#define PROCESS\_SET\_ELEM\_SLOTNonVar(name, func) PROCESS\_SET\_ELEM\_SLOTNonVar\_COMMON(name, func,)

#define PROCESS\_SET\_ELEM\_LOCALSLOTNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlotI1, suffix); \

func((Var\*)GetLocalClosure(), playout->SlotIndex, GetRegAllowStackVarEnableOnly(playout->Value)); \

break; \

}

#define PROCESS\_SET\_ELEM\_LOCALSLOTNonVar(name, func) PROCESS\_SET\_ELEM\_LOCALSLOTNonVar\_COMMON(name, func,)

#define PROCESS\_SET\_ELEM\_INNERSLOTNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlotI2, suffix); \

func(InnerScopeFromIndex(playout->SlotIndex1), playout->SlotIndex2, GetRegAllowStackVarEnableOnly(playout->Value)); \

break; \

}

#define PROCESS\_SET\_ELEM\_INNERSLOTNonVar(name, func) PROCESS\_SET\_ELEM\_INNERSLOTNonVar\_COMMON(name, func,)

#define PROCESS\_SET\_ELEM\_ENVSLOTNonVar\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, ElementSlotI2, suffix); \

func(LdEnv(), playout->SlotIndex1, playout->SlotIndex2, GetRegAllowStackVarEnableOnly(playout->Value)); \

break; \

}

#define PROCESS\_SET\_ELEM\_ENVSLOTNonVar(name, func) PROCESS\_SET\_ELEM\_ENVSLOTNonVar\_COMMON(name, func,)

/\*---------------------------------------------------------------------------------------------- \*/

#define PROCESS\_A3toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg4, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2), GetReg(playout->R3), GetScriptContext())); \

break; \

}

#define PROCESS\_A3toA1Mem(name, func) PROCESS\_A3toA1Mem\_COMMON(name, func,)

/\*---------------------------------------------------------------------------------------------- \*/

#define PROCESS\_A2I1toA1Mem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3B1, suffix); \

SetReg(playout->R0, \

func(GetReg(playout->R1), GetReg(playout->R2), playout->B3, GetScriptContext())); \

break; \

}

#define PROCESS\_A2I1toA1Mem(name, func) PROCESS\_A2I1toA1Mem\_COMMON(name, func,)

/\*---------------------------------------------------------------------------------------------- \*/

#define PROCESS\_A2I1toXXMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg2B1, suffix); \

func(GetReg(playout->R0), GetReg(playout->R1), playout->B2, scriptContext); \

break; \

}

#define PROCESS\_A2I1toXXMem(name, func) PROCESS\_A2I1toXXMem\_COMMON(name, func,)

/\*---------------------------------------------------------------------------------------------- \*/

#define PROCESS\_A3I1toXXMem\_COMMON(name, func, suffix) \

case OpCode::name: \

{ \

PROCESS\_READ\_LAYOUT(name, Reg3B1, suffix); \

func(GetReg(playout->R0), GetReg(playout->R1), GetReg(playout->R2), playout->B3, scriptContext); \

break; \

}

#define PROCESS\_A3I1toXXMem(name, func) PROCESS\_A3I1toXXMem\_COMMON(name, func,)

#if ENABLE\_PROFILE\_INFO

#define PROCESS\_IP\_TARG\_IMPL(name, func, layoutSize) \

case OpCode::name: \

{ \

Assert(!switchProfileMode); \

ip = func<layoutSize, INTERPRETERPROFILE>(ip); \

if(switchProfileMode) \

{ \

m\_reader.SetIP(ip); \

return nullptr; \

} \

break; \

}

#else

#define PROCESS\_IP\_TARG\_IMPL(name, func, layoutSize) \

case OpCode::name: \

{ \

ip = func<layoutSize, INTERPRETERPROFILE>(ip); \

break; \

}

#endif

#define PROCESS\_IP\_TARG\_COMMON(name, func, suffix) PROCESS\_IP\_TARG##suffix(name, func)

#define PROCESS\_IP\_TARG\_Large(name, func) PROCESS\_IP\_TARG\_IMPL(name, func, Js::LargeLayout)

#define PROCESS\_IP\_TARG\_Medium(name, func) PROCESS\_IP\_TARG\_IMPL(name, func, Js::MediumLayout)

#define PROCESS\_IP\_TARG\_Small(name, func) PROCESS\_IP\_TARG\_IMPL(name, func, Js::SmallLayout)

namespace Js

{

extern const \_\_declspec(selectany) uint32 TypedArrayViewMask[] =

{

(uint32)~0 //TYPE\_INT8

, (uint32)~0 //TYPE\_UINT8

, (uint32)~1 //TYPE\_INT16

, (uint32)~1 //TYPE\_UINT16

, (uint32)~3 //TYPE\_INT32

, (uint32)~3 //TYPE\_UINT32

, (uint32)~3 //TYPE\_FLOAT32

, (uint32)~7 //TYPE\_FLOAT64

};

#ifndef TEMP\_DISABLE\_ASMJS

typedef void(InterpreterStackFrame::\*ArrFunc)(uint32, RegSlot);

const ArrFunc InterpreterStackFrame::StArrFunc[8] =

{

&InterpreterStackFrame::OP\_StArr<int8>,

&InterpreterStackFrame::OP\_StArr<uint8>,

&InterpreterStackFrame::OP\_StArr<int16>,

&InterpreterStackFrame::OP\_StArr<uint16>,

&InterpreterStackFrame::OP\_StArr<int32>,

&InterpreterStackFrame::OP\_StArr<uint32>,

&InterpreterStackFrame::OP\_StArr<float>,

&InterpreterStackFrame::OP\_StArr<double>,

};

const ArrFunc InterpreterStackFrame::LdArrFunc[8] =

{

&InterpreterStackFrame::OP\_LdArr<int8>,

&InterpreterStackFrame::OP\_LdArr<uint8>,

&InterpreterStackFrame::OP\_LdArr<int16>,

&InterpreterStackFrame::OP\_LdArr<uint16>,

&InterpreterStackFrame::OP\_LdArr<int32>,

&InterpreterStackFrame::OP\_LdArr<uint32>,

&InterpreterStackFrame::OP\_LdArr<float>,

&InterpreterStackFrame::OP\_LdArr<double>,

};

#endif

Var InterpreterStackFrame::InnerScopeFromRegSlot(RegSlot reg) const

{

return InnerScopeFromIndex(reg - m\_functionBody->FirstInnerScopeReg());

}

Var InterpreterStackFrame::InnerScopeFromIndex(uint32 index) const

{

if (index >= m\_functionBody->GetInnerScopeCount())

{

AssertMsg(false, "Illegal byte code: bad inner scope index");

Js::Throw::FatalInternalError();

}

Assert(this->innerScopeArray != nullptr);

return this->innerScopeArray[index];

}

void InterpreterStackFrame::SetInnerScopeFromIndex(uint32 index, Var scope)

{

if (index >= m\_functionBody->GetInnerScopeCount())

{

AssertMsg(false, "Illegal byte code: bad inner scope index");

Js::Throw::FatalInternalError();

}

Assert(this->innerScopeArray != nullptr);

this->innerScopeArray[index] = scope;

}

const int k\_stackFrameVarCount = (sizeof(InterpreterStackFrame) + sizeof(Var) - 1) / sizeof(Var);

InterpreterStackFrame::Setup::Setup(Js::ScriptFunction \* function, Js::Arguments& args, bool inlinee)

: function(function), inParams(args.Values), inSlotsCount(args.Info.Count), executeFunction(function->GetFunctionBody()), callFlags(args.Info.Flags), bailedOutOfInlinee(inlinee)

{

SetupInternal();

}

InterpreterStackFrame::Setup::Setup(Js::ScriptFunction \* function, Var \* inParams, int inSlotsCount, bool inlinee)

: function(function), inParams(inParams), inSlotsCount(inSlotsCount), executeFunction(function->GetFunctionBody()), callFlags(CallFlags\_None), bailedOutOfInlinee(inlinee)

{

SetupInternal();

}

void InterpreterStackFrame::Setup::SetupInternal()

{

if (this->function->GetHasInlineCaches() && Js::ScriptFunctionWithInlineCache::Is(this->function))

{

this->inlineCaches = Js::ScriptFunctionWithInlineCache::FromVar(this->function)->GetInlineCaches();

}

else

{

this->inlineCaches = this->executeFunction->GetInlineCaches();

}

this->inlineCacheCount = this->executeFunction->GetInlineCacheCount();

//

// Compute the amount of memory needed on the stack:

// - We compute this in 'Atoms' instead of 'bytes' to keep everything natural word aligned.

//

this->localCount = this->executeFunction->GetLocalsCount();

uint extraVarCount = 0;

#if ENABLE\_PROFILE\_INFO

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(this->executeFunction))

{

extraVarCount += (sizeof(ImplicitCallFlags) \* this->executeFunction->GetLoopCount() + sizeof(Var) - 1) / sizeof(Var);

}

#endif

this->varAllocCount = k\_stackFrameVarCount + localCount + this->executeFunction->GetOutParamsDepth() + extraVarCount + this->executeFunction->GetInnerScopeCount();

if (this->executeFunction->DoStackNestedFunc() && this->executeFunction->GetNestedCount() != 0)

{

// Track stack funcs...

this->varAllocCount += (sizeof(StackScriptFunction) \* this->executeFunction->GetNestedCount()) / sizeof(Var);

if (!this->bailedOutOfInlinee)

{

// Frame display (if environment depth is statically known)...

if (this->executeFunction->DoStackFrameDisplay())

{

uint16 envDepth = this->executeFunction->GetEnvDepth();

Assert(envDepth != (uint16)-1);

this->varAllocCount += sizeof(FrameDisplay) / sizeof(Var) + (envDepth + 1);

}

// ...and scope slots (if any)

if (this->executeFunction->DoStackScopeSlots())

{

uint32 scopeSlots = this->executeFunction->scopeSlotArraySize;

Assert(scopeSlots != 0);

this->varAllocCount += scopeSlots + Js::ScopeSlots::FirstSlotIndex;

}

}

}

}

InterpreterStackFrame \*

InterpreterStackFrame::Setup::InitializeAllocation(\_\_in\_ecount(varAllocCount) Var \* allocation, bool initParams, bool profileParams, Var loopHeaderArray, DWORD\_PTR stackAddr

#if DBG

, Var invalidStackVar

#endif

)

{

//

// Initialize the new InterpreterStackFrame instance on the program stack.

//

//This will fail if InterpreterStackFrame ever gets a non-empty ctor (you'll need to use

//placement\_new(allocation, InterpreterStackFrame) instead, though that will cause problems

//if the placement\_new is surrounded by a try/finally since this would mix C++/SEH exception

//handling.

\_\_analysis\_assume(varAllocCount >= k\_stackFrameVarCount + localCount);

InterpreterStackFrame\* newInstance = (InterpreterStackFrame\*)allocation;

newInstance->scriptContext = this->executeFunction->GetScriptContext();

newInstance->m\_inSlotsCount = this->inSlotsCount;

newInstance->m\_inParams = this->inParams;

newInstance->m\_callFlags = this->callFlags;

newInstance->m\_outParams = newInstance->m\_localSlots + localCount;

newInstance->m\_outSp = newInstance->m\_outParams;

newInstance->m\_arguments = NULL;

newInstance->function = this->function;

newInstance->m\_functionBody = this->executeFunction;

newInstance->inlineCaches = this->inlineCaches;

newInstance->inlineCacheCount = this->inlineCacheCount;

newInstance->currentLoopNum = LoopHeader::NoLoop;

newInstance->currentLoopCounter = 0;

newInstance->m\_flags = InterpreterStackFrameFlags\_None;

newInstance->closureInitDone = false;

#if ENABLE\_PROFILE\_INFO

newInstance->switchProfileMode = false;

newInstance->isAutoProfiling = false;

newInstance->switchProfileModeOnLoopEndNumber = 0u - 1;

#endif

newInstance->ehBailoutData = nullptr;

newInstance->nestedTryDepth = -1;

newInstance->nestedCatchDepth = -1;

newInstance->nestedFinallyDepth = -1;

newInstance->retOffset = 0;

newInstance->localFrameDisplay = nullptr;

newInstance->localClosure = nullptr;

newInstance->innerScopeArray = nullptr;

bool doInterruptProbe = newInstance->scriptContext->GetThreadContext()->DoInterruptProbe(this->executeFunction);

#if ENABLE\_NATIVE\_CODEGEN

bool doJITLoopBody =

!this->executeFunction->GetScriptContext()->GetConfig()->IsNoNative() &&

!(this->executeFunction->GetHasTry() && (PHASE\_OFF((Js::JITLoopBodyInTryCatchPhase), this->executeFunction) || this->executeFunction->GetHasFinally())) &&

(this->executeFunction->ForceJITLoopBody() || this->executeFunction->IsJitLoopBodyPhaseEnabled()) &&

!this->executeFunction->GetScriptContext()->IsInDebugMode();

#else

const bool doJITLoopBody = false;

#endif

// Pick a version of the LoopBodyStart OpCode handlers that is hardcoded to do loop body JIT and

// interrupt probes as needed.

if (doInterruptProbe)

{

#if ENABLE\_NATIVE\_CODEGEN

if (doJITLoopBody)

{

newInstance->opProfiledLoopBodyStart = &InterpreterStackFrame::ProfiledLoopBodyStart<true, true>;

newInstance->opLoopBodyStart = &InterpreterStackFrame::LoopBodyStart<true, true>;

}

else

#endif

{

#if ENABLE\_PROFILE\_INFO

newInstance->opProfiledLoopBodyStart = &InterpreterStackFrame::ProfiledLoopBodyStart<true, false>;

#endif

newInstance->opLoopBodyStart = &InterpreterStackFrame::LoopBodyStart<true, false>;

}

}

else

{

#if ENABLE\_NATIVE\_CODEGEN

if (doJITLoopBody)

{

newInstance->opProfiledLoopBodyStart = &InterpreterStackFrame::ProfiledLoopBodyStart<false, true>;

newInstance->opLoopBodyStart = &InterpreterStackFrame::LoopBodyStart<false, true>;

}

else

#endif

{

#if ENABLE\_PROFILE\_INFO

newInstance->opProfiledLoopBodyStart = &InterpreterStackFrame::ProfiledLoopBodyStart<false, false>;

#endif

newInstance->opLoopBodyStart = &InterpreterStackFrame::LoopBodyStart<false, false>;

}

}

newInstance->loopHeaderArray = loopHeaderArray;

newInstance->m\_stackAddress = stackAddr;

#if ENABLE\_PROFILE\_INFO

// the savedLoopImplicitCallFlags is allocated at the end of the out param array

newInstance->savedLoopImplicitCallFlags = nullptr;

#endif

char \* nextAllocBytes = (char \*)(newInstance->m\_outParams + this->executeFunction->GetOutParamsDepth());

if (this->executeFunction->GetInnerScopeCount())

{

newInstance->innerScopeArray = (Var\*)nextAllocBytes;

nextAllocBytes += this->executeFunction->GetInnerScopeCount() \* sizeof(Var);

}

if (this->executeFunction->DoStackNestedFunc() && this->executeFunction->GetNestedCount() != 0)

{

newInstance->InitializeStackFunctions((StackScriptFunction \*)nextAllocBytes);

nextAllocBytes = nextAllocBytes + sizeof(StackScriptFunction) \* this->executeFunction->GetNestedCount();

if (!this->bailedOutOfInlinee)

{

if (this->executeFunction->DoStackFrameDisplay())

{

uint16 envDepth = this->executeFunction->GetEnvDepth();

Assert(envDepth != (uint16)-1);

newInstance->localFrameDisplay = (FrameDisplay\*)nextAllocBytes;

nextAllocBytes += sizeof(FrameDisplay) + (envDepth + 1) \* sizeof(Var);

}

if (this->executeFunction->DoStackScopeSlots())

{

uint32 scopeSlots = this->executeFunction->scopeSlotArraySize;

Assert(scopeSlots != 0);

newInstance->localClosure = nextAllocBytes;

nextAllocBytes += (scopeSlots + ScopeSlots::FirstSlotIndex) \* sizeof(Var);

}

}

}

#if ENABLE\_PROFILE\_INFO

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(this->executeFunction))

{

/\*

\_\_analysis\_assume(varAllocCount == (k\_stackFrameVarCount + localCount + executeFunction->GetOutParamsDepth()

+ ((sizeof(ImplicitCallFlags) \* executeFunction->GetLoopCount() + sizeof(Var) - 1) / sizeof(Var))));

\*/

newInstance->savedLoopImplicitCallFlags = (ImplicitCallFlags \*)nextAllocBytes;

for (uint i = 0; i < this->executeFunction->GetLoopCount(); i++)

{

#pragma prefast(suppress:26015, "Above analysis assume doesn't work")

newInstance->savedLoopImplicitCallFlags[i] = ImplicitCall\_None;

}

}

#endif

#if DBG

if (CONFIG\_ISENABLED(InitializeInterpreterSlotsWithInvalidStackVarFlag))

{

// Fill the local slots with the invalid stack var so that we will crash deterministically if something goes wrong

for (uint i = 0; i < localCount; ++i)

{

newInstance->m\_localSlots[i] = invalidStackVar;

}

}

else

{

memset(newInstance->m\_localSlots, 0, sizeof(Js::Var) \* localCount);

}

#else

if (newInstance->scriptContext->IsInDebugMode())

{

// In the debug mode zero out the local slot, so this could prevent locals being uninitialized in the case of setNextStatement.

memset(newInstance->m\_localSlots, 0, sizeof(Js::Var) \* localCount);

}

// Zero out only the return slot. This is not a user local, so the byte code will not initialize

// it to "undefined". And it's not an expression temp, so, for instance, a jitted loop body may expect

// it to be valid on entry to the loop, where "valid" means either a var or null.

newInstance->SetNonVarReg(0, NULL);

#endif

// Initialize the low end of the local slots from the constant table.

// Skip the slot for the return value register.

this->executeFunction->InitConstantSlots(&newInstance->m\_localSlots[FunctionBody::FirstRegSlot]);

// Set local FD/SS pointers to null until after we've successfully probed the stack in the process loop.

// That way we avoid trying to box these structures before they've been initialized in the byte code.

if (this->executeFunction->DoStackFrameDisplay())

{

newInstance->SetNonVarReg(executeFunction->GetLocalFrameDisplayReg(), nullptr);

}

if (this->executeFunction->DoStackScopeSlots())

{

Assert(!executeFunction->HasScopeObject());

newInstance->SetNonVarReg(executeFunction->GetLocalClosureReg(), nullptr);

}

Var \*prestDest = &newInstance->m\_localSlots[this->executeFunction->GetConstantCount()];

if (initParams)

{

#if ENABLE\_PROFILE\_INFO

Assert(!this->executeFunction->NeedEnsureDynamicProfileInfo());

#endif

if (profileParams)

{

#if ENABLE\_PROFILE\_INFO

Assert(this->executeFunction->HasExecutionDynamicProfileInfo());

#endif

FunctionBody\* functionBody = this->executeFunction;

InitializeParams(newInstance, [functionBody](Var param, ArgSlot index)

{

#if ENABLE\_PROFILE\_INFO

functionBody->GetDynamicProfileInfo()->RecordParameterInfo(functionBody, index - 1, param);

#endif

}, &prestDest);

}

else

{

InitializeParams(newInstance, [](Var param, ArgSlot index) {}, &prestDest);

}

}

if (this->executeFunction->GetHasRestParameter())

{

InitializeRestParam(newInstance, prestDest);

}

Js::RegSlot envReg = executeFunction->GetEnvReg();

if (envReg != Js::Constants::NoRegister && envReg < executeFunction->GetConstantCount())

{

Assert(this->executeFunction->GetThisRegForEventHandler() == Constants::NoRegister);

// The correct FD (possibly distinct from the one on the function) is passed in the constant table.

this->function->SetEnvironment((Js::FrameDisplay\*)newInstance->GetNonVarReg(envReg));

}

return newInstance;

}

template <class Fn>

void InterpreterStackFrame::Setup::InitializeParams(InterpreterStackFrame \* newInstance, Fn callback, Var \*\*pprestDest)

{

ArgSlot requiredInParamCount = executeFunction->GetInParamsCount();

Assert(requiredInParamCount > 1);

if (this->inSlotsCount >= requiredInParamCount)

{

Var \* pArg = &newInstance->m\_localSlots[executeFunction->GetConstantCount()];

Var \* paGivenSrc = this->inParams + 1;

ArgSlot paramIndex = 1;

do

{

Var src = \*paGivenSrc++;

callback(src, paramIndex);

\*pArg++ = src;

paramIndex++;

}

while (paramIndex < requiredInParamCount);

\*pprestDest = pArg;

}

else

{

InitializeParamsAndUndef(newInstance, callback, pprestDest);

}

}

template <class Fn>

void InterpreterStackFrame::Setup::InitializeParamsAndUndef(InterpreterStackFrame \* newInstance, Fn callback, Var \*\*pprestDest)

{

Var \* pArg = &newInstance->m\_localSlots[executeFunction->GetConstantCount()];

Var \* paGivenSrc = this->inParams + 1;

ArgSlot requiredInParamCount = executeFunction->GetInParamsCount();

ArgSlot paramIndex = 1;

while (paramIndex < this->inSlotsCount)

{

Var src = \*paGivenSrc++;

callback(src, paramIndex);

\*pArg++ = src;

paramIndex++;

}

Var varUndef = executeFunction->GetScriptContext()->GetLibrary()->GetUndefined();

do

{

callback(varUndef, paramIndex);

\*pArg++ = varUndef;

paramIndex++;

}

while (paramIndex < requiredInParamCount);

\*pprestDest = pArg;

}

void InterpreterStackFrame::Setup::InitializeRestParam(InterpreterStackFrame \* newInstance, Var \*dest)

{

Var \*src = this->inParams + executeFunction->GetInParamsCount();

if (this->inSlotsCount > executeFunction->GetInParamsCount())

{

// Create the rest array and copy the args directly into the contiguous head segment.

int excess = this->inSlotsCount - executeFunction->GetInParamsCount();

\*dest = JavascriptArray::OP\_NewScArray(excess, executeFunction->GetScriptContext());

JavascriptArray \*array = static\_cast<JavascriptArray \*>(\*dest);

Var \*elements = ((SparseArraySegment<Var>\*)array->GetHead())->elements;

js\_memcpy\_s(elements, excess \* sizeof(Var), src, excess \* sizeof(Var));

}

else

{

// Rest is an empty array when there are no excess parameters.

\*dest = JavascriptArray::OP\_NewScArray(0, executeFunction->GetScriptContext());

}

}

FrameDisplay \* InterpreterStackFrame::GetEnvForEvalCode()

{

FrameDisplay \*pScope;

if (m\_functionBody->GetIsStrictMode() && m\_functionBody->GetIsGlobalFunc())

{

pScope = this->GetLocalFrameDisplay();

}

else

{

pScope = (FrameDisplay\*)this->LdEnv();

}

return pScope;

}

void InterpreterStackFrame::InitializeClosures()

{

FunctionBody \*executeFunction = this->function->GetFunctionBody();

Var environment;

RegSlot thisRegForEventHandler = executeFunction->GetThisRegForEventHandler();

if (thisRegForEventHandler != Constants::NoRegister)

{

Var varThis = OP\_ArgIn0();

SetReg(thisRegForEventHandler, varThis);

environment = JavascriptOperators::OP\_LdHandlerScope(varThis, GetScriptContext());

this->SetEnv((FrameDisplay\*)environment);

}

else

{

environment = this->LdEnv();

}

RegSlot closureReg = executeFunction->GetLocalClosureReg();

if (closureReg != Js::Constants::NoRegister)

{

Assert(closureReg >= executeFunction->GetConstantCount());

if (executeFunction->HasScopeObject())

{

Js::RegSlot funcExprScopeReg = executeFunction->GetFuncExprScopeReg();

if (funcExprScopeReg != Constants::NoRegister)

{

// t0 = NewPseudoScope

// t1 = LdFrameDisplay t0 env

Var funcExprScope = JavascriptOperators::OP\_NewPseudoScope(GetScriptContext());

SetReg(funcExprScopeReg, funcExprScope);

environment = OP\_LdFrameDisplay(funcExprScope, environment, GetScriptContext());

}

this->NewScopeObject();

}

else

{

this->NewScopeSlots();

}

this->SetNonVarReg(closureReg, nullptr);

}

Js::RegSlot frameDisplayReg = executeFunction->GetLocalFrameDisplayReg();

if (frameDisplayReg != Js::Constants::NoRegister && closureReg != Js::Constants::NoRegister)

{

Assert(frameDisplayReg >= executeFunction->GetConstantCount());

void \*argHead = this->GetLocalClosure();

this->SetLocalFrameDisplay(this->NewFrameDisplay(argHead, environment));

this->SetNonVarReg(frameDisplayReg, nullptr);

}

this->closureInitDone = true;

}

#ifdef \_M\_IX86

#ifndef TEMP\_DISABLE\_ASMJS

int InterpreterStackFrame::GetAsmJsArgSize(AsmJsCallStackLayout\* stack)

{

JavascriptFunction \* func = stack->functionObject;

AsmJsFunctionInfo\* asmInfo = func->GetFunctionBody()->GetAsmJsFunctionInfo();

uint argSize = (uint)(asmInfo->GetArgByteSize());

argSize = ::Math::Align<int32>(argSize, 8);

// 2 \* sizeof(Var) is for functionObject, and another push that DynamicInterpreterThunk does

return argSize + 2 \* sizeof(Var);

}

int InterpreterStackFrame::GetDynamicRetType(AsmJsCallStackLayout\* stack)

{

return GetRetType(stack->functionObject);

}

int InterpreterStackFrame::GetRetType(JavascriptFunction\* func)

{

AsmJsFunctionInfo\* asmInfo = func->GetFunctionBody()->GetAsmJsFunctionInfo();

return asmInfo->GetReturnType().which();

}

DWORD InterpreterStackFrame::GetAsmIntDbValOffSet(AsmJsCallStackLayout\* stack)

{

JavascriptFunction \* func = stack->functionObject;

ScriptContext\* scriptContext = func->GetScriptContext();

return (DWORD)scriptContext + ScriptContext::GetAsmIntDbValOffset();

}

DWORD InterpreterStackFrame::GetAsmSimdValOffSet(AsmJsCallStackLayout\* stack)

{

JavascriptFunction \* func = stack->functionObject;

ScriptContext\* scriptContext = func->GetScriptContext();

return (DWORD)scriptContext + ScriptContext::GetAsmSimdValOffset();

}

#ifdef ASMJS\_PLAT

/\*

AsmInterpreterThunk

-------------------

This is the entrypoint for all Asm Interpreter calls (external and internal)

TODO - Make this a dynamic Interpreter thunk to support ETW

Functionality:

1) Prolog

2) call AsmInterpreter passing the function object

3) Get The return type

4) Check for Double or Float return type

5) If true then retrieve the value stored at a constant offset from the ScriptContext

6) Get Argument Size for callee cleanup

7) EpiLog

a) Retrieve the frame pointer

b) Store the return address in register (edx)

c) Clean the arguments based on the arguments size

d) push the return address back into the stack

\*/

\_\_declspec(naked)

void InterpreterStackFrame::InterpreterAsmThunk(AsmJsCallStackLayout\* layout)

{

enum {

Void = AsmJsRetType::Void,

Signed = AsmJsRetType::Signed,

Float = AsmJsRetType::Float,

Double = AsmJsRetType::Double,

Int32x4 = AsmJsRetType::Int32x4,

Float32x4 = AsmJsRetType::Float32x4,

Float64x2 = AsmJsRetType::Float64x2

};

//Prolog

\_\_asm

{

//Prologue

push ebp;

mov ebp, esp;

push layout; // push stack layout

call InterpreterStackFrame::AsmJsInterpreter;

push eax; // push the return value into the stack

push layout;

call InterpreterStackFrame::GetDynamicRetType;

cmp eax, Void;

je end;

cmp eax, Signed;

je end;

cmp eax, Float;

jne skipFloat;

// float

push layout;

call InterpreterStackFrame::GetAsmIntDbValOffSet;

cvtsd2ss xmm0, [eax];

jmp end;

skipFloat:

cmp eax, Double;

jne skipDouble;

// double

push layout;

call InterpreterStackFrame::GetAsmIntDbValOffSet;

movsd xmm0, [eax];

jmp end;

skipDouble:

// simd value

push layout;

call InterpreterStackFrame::GetAsmSimdValOffSet;

movups xmm0, [eax];

end:

push layout;

call InterpreterStackFrame::GetAsmJsArgSize;

mov ecx, eax;

pop eax; // pop the return value from AsmJsInterpreter to eax

// Epilog, callee cleanup

mov esp, ebp;

pop ebp;

// we need to move stack around in order to do callee cleanup

// unfortunately, we don't really have enough registers to do this cleanly

//

// we are rearranging the stack from this:

// 0x14 caller push scriptArg1

// 0x10 caller push functionObject

// 0x0C DynamicInterpreterThunk return address

// 0x08 DynamicInterpreterThunk push ebp

// 0x04 DynamicInterpreterThunk push functionObject

// 0x00 InterpreterAsmThunk return address <- stack pointer

// to this:

// 0x14 DynamicInterpreterThunk return address

// 0x10 DynamicInterpreterThunk push ebp

// 0x0C InterpreterAsmThunk return address <- stack pointer

push eax; // save eax

mov eax, esp;

add eax, ecx;

add eax, 0xC; // eax will be our stack destination. we need to move backwards because memory might overlap

mov edx, [esp+0x10];

mov [eax], edx; // move the dynamic interpreter thunk return location

sub eax, 0x4;

mov edx, [esp+0xC];

mov [eax], edx; // move the dynamic interpreter thunk "push ebp" location

// skip "push functionObject"

sub eax, 0x4;

mov edx, [esp+0x4];

mov [eax], edx; // move the return location

pop eax;

add esp, ecx; // cleanup arguments

ret;

}

}

#endif

#endif

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

#ifdef \_M\_IX86

\_\_declspec(naked)

Var InterpreterStackFrame::DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

{

\_\_asm

{

push ebp

mov ebp, esp

push [esp+8] // push function object

call InterpreterStackFrame::EnsureDynamicInterpreterThunk;

#ifdef \_CONTROL\_FLOW\_GUARD

// verify that the call target is valid

push eax

mov ecx, eax

call[\_\_guard\_check\_icall\_fptr]

pop eax

#endif

pop ebp

jmp eax

}

}

#endif

#endif

#if ENABLE\_PROFILE\_INFO

JavascriptMethod InterpreterStackFrame::EnsureDynamicInterpreterThunk(Js::ScriptFunction \* function)

{

#if DYNAMIC\_INTERPRETER\_THUNK

Assert(function);

Js::FunctionBody \*functionBody = function->GetFunctionBody();

JavascriptMethod entrypoint = functionBody->EnsureDynamicInterpreterThunk(function->GetFunctionEntryPointInfo());

Assert(!IsDelayDynamicInterpreterThunk(functionBody->GetDirectEntryPoint(function->GetEntryPointInfo())));

if (function->GetEntryPoint() == InterpreterStackFrame::DelayDynamicInterpreterThunk)

{

// If we are not profiling, or the function object is not cross site, this is the direct entry point.

// Change the entry point on the object

Assert(functionBody->GetDirectEntryPoint(function->GetEntryPointInfo()) == entrypoint);

function->ChangeEntryPoint(function->GetEntryPointInfo(), entrypoint);

}

// Return the original entry point to be called

return entrypoint;

#else

return function->GetEntryPoint();

#endif

}

#endif

bool InterpreterStackFrame::IsDelayDynamicInterpreterThunk(void \* entryPoint)

{

return

#if DYNAMIC\_INTERPRETER\_THUNK

#if \_M\_X64

entryPoint == InterpreterStackFrame::AsmJsDelayDynamicInterpreterThunk ||

#endif

entryPoint == InterpreterStackFrame::DelayDynamicInterpreterThunk;

#else

false;

#endif

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

\_\_declspec(thread) int InterpreterThunkStackCountTracker::s\_count = 0;

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

Var InterpreterStackFrame::InterpreterThunk(JavascriptCallStackLayout\* layout)

{

Js::ScriptFunction \* function = Js::ScriptFunction::FromVar(layout->functionObject);

Js::ArgumentReader args(&layout->callInfo, layout->args);

void\* localReturnAddress = \_ReturnAddress();

void\* localAddressOfReturnAddress = \_AddressOfReturnAddress();

return InterpreterHelper(function, args, localReturnAddress, localAddressOfReturnAddress);

}

#else

#pragma optimize("", off)

Var InterpreterStackFrame::InterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

{

ARGUMENTS(args, callInfo);

void\* localReturnAddress = \_ReturnAddress();

void\* localAddressOfReturnAddress = \_AddressOfReturnAddress();

Assert(ScriptFunction::Is(function));

return InterpreterHelper(ScriptFunction::FromVar(function), args, localReturnAddress, localAddressOfReturnAddress);

}

#pragma optimize("", on)

#endif

Var InterpreterStackFrame::InterpreterHelper(ScriptFunction\* function, ArgumentReader args, void\* returnAddress, void\* addressOfReturnAddress, const bool isAsmJs)

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// Support for simulating partially initialized interpreter stack frame.

InterpreterThunkStackCountTracker tracker;

if (CONFIG\_ISENABLED(InjectPartiallyInitializedInterpreterFrameErrorFlag) &&

CONFIG\_FLAG(InjectPartiallyInitializedInterpreterFrameError) == InterpreterThunkStackCountTracker::GetCount())

{

switch (CONFIG\_FLAG(InjectPartiallyInitializedInterpreterFrameErrorType))

{

case 0:

DebugBreak();

break;

case 1:

Js::JavascriptError::MapAndThrowError(function->GetScriptContext(), VBSERR\_InternalError);

break;

default:

DebugBreak();

}

}

#endif

ScriptContext\* functionScriptContext = function->GetScriptContext();

ThreadContext \* threadContext = functionScriptContext->GetThreadContext();

Assert(!threadContext->IsDisableImplicitException());

functionScriptContext->VerifyAlive(!function->IsExternal());

Assert(threadContext->IsScriptActive());

Assert(threadContext->IsInScript());

FunctionBody\* executeFunction = JavascriptFunction::FromVar(function)->GetFunctionBody();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (!isAsmJs && executeFunction->IsByteCodeDebugMode() != functionScriptContext->IsInDebugMode()) // debug mode mismatch

{

if (executeFunction->GetUtf8SourceInfo()->GetIsLibraryCode())

{

Assert(!executeFunction->IsByteCodeDebugMode()); // Library script byteCode is never in debug mode

}

else

{

Throw::FatalInternalError();

}

}

#endif

if (executeFunction->interpretedCount == 0)

{

executeFunction->TraceInterpreterExecutionMode();

}

class AutoRestore

{

private:

ThreadContext \*const threadContext;

const uint8 savedLoopDepth;

public:

AutoRestore(ThreadContext \*const threadContext, FunctionBody \*const executeFunction)

: threadContext(threadContext),

savedLoopDepth(threadContext->LoopDepth())

{

if (savedLoopDepth != 0 && !executeFunction->GetIsAsmJsFunction())

{

executeFunction->SetWasCalledFromLoop();

}

}

~AutoRestore()

{

threadContext->SetLoopDepth(savedLoopDepth);

}

} autoRestore(threadContext, executeFunction);

#if ENABLE\_PROFILE\_INFO

DynamicProfileInfo \* dynamicProfileInfo = nullptr;

const bool doProfile = executeFunction->GetInterpreterExecutionMode(false) == ExecutionMode::ProfilingInterpreter ||

functionScriptContext->IsInDebugMode() && DynamicProfileInfo::IsEnabled(executeFunction);

if (doProfile)

{

#if !DYNAMIC\_INTERPRETER\_THUNK

executeFunction->EnsureDynamicProfileInfo();

#endif

dynamicProfileInfo = executeFunction->GetDynamicProfileInfo();

threadContext->ClearImplicitCallFlags();

}

#else

const bool doProfile = false;

#endif

executeFunction->interpretedCount++;

#ifdef BGJIT\_STATS

functionScriptContext->interpretedCount++;

functionScriptContext->maxFuncInterpret = max(functionScriptContext->maxFuncInterpret, executeFunction->interpretedCount);

#endif

AssertMsg(!executeFunction->IsDeferredParseFunction(),

"Non-intrinsic functions must provide byte-code to execute");

bool fReleaseAlloc = false;

InterpreterStackFrame\* newInstance = nullptr;

Var\* allocation = nullptr;

if (!isAsmJs && executeFunction->IsGenerator())

{

// If the FunctionBody is a generator then this call is being made by one of the three

// generator resuming methods: next(), throw(), or return(). They all pass the generator

// object as the first of two arguments. The real user arguments are obtained from the

// generator object. The second argument is the ResumeYieldData which is only needed

// when resuming a generator and so it only used here if a frame already exists on the

// generator object.

AssertMsg(args.Info.Count == 2, "Generator ScriptFunctions should only be invoked by generator APIs with the pair of arguments they pass in -- the generator object and a ResumeYieldData pointer");

JavascriptGenerator\* generator = JavascriptGenerator::FromVar(args[0]);

newInstance = generator->GetFrame();

if (newInstance != nullptr)

{

ResumeYieldData\* resumeYieldData = static\_cast<ResumeYieldData\*>(args[1]);

newInstance->SetNonVarReg(executeFunction->GetYieldRegister(), resumeYieldData);

// The debugger relies on comparing stack addresses of frames to decide when a step\_out is complete so

// give the InterpreterStackFrame a legit enough stack address to make this comparison work.

newInstance->m\_stackAddress = reinterpret\_cast<DWORD\_PTR>(&generator);

}

else

{

//

// Allocate a new InterpreterStackFrame instance on the recycler heap.

// It will live with the JavascriptGenerator object.

//

Arguments generatorArgs = generator->GetArguments();

InterpreterStackFrame::Setup setup(function, generatorArgs);

size\_t varAllocCount = setup.GetAllocationVarCount();

size\_t varSizeInBytes = varAllocCount \* sizeof(Var);

DWORD\_PTR stackAddr = reinterpret\_cast<DWORD\_PTR>(&generator); // as mentioned above, use any stack address from this frame to ensure correct debugging functionality

Var loopHeaderArray = executeFunction->GetHasAllocatedLoopHeaders() ? executeFunction->GetLoopHeaderArrayPtr() : nullptr;

allocation = RecyclerNewPlus(functionScriptContext->GetRecycler(), varSizeInBytes, Var);

AnalysisAssert(allocation);

#if DBG

// Allocate invalidVar on GC instead of stack since this InterpreterStackFrame will out live the current real frame

Js::RecyclableObject\* invalidVar = (Js::RecyclableObject\*)RecyclerNewPlusLeaf(functionScriptContext->GetRecycler(), sizeof(Js::RecyclableObject), Var);

AnalysisAssert(invalidVar);

memset(invalidVar, 0xFE, sizeof(Js::RecyclableObject));

newInstance = setup.InitializeAllocation(allocation, executeFunction->GetHasImplicitArgIns(), doProfile, loopHeaderArray, stackAddr, invalidVar);

#else

newInstance = setup.InitializeAllocation(allocation, executeFunction->GetHasImplicitArgIns(), doProfile, loopHeaderArray, stackAddr);

#endif

newInstance->m\_reader.Create(executeFunction);

generator->SetFrame(newInstance);

}

}

else

{

InterpreterStackFrame::Setup setup(function, args);

size\_t varAllocCount = setup.GetAllocationVarCount();

size\_t varSizeInBytes = varAllocCount \* sizeof(Var);

//

// Allocate a new InterpreterStackFrame instance on the interpreter's virtual stack.

//

DWORD\_PTR stackAddr;

// If the locals area exceeds a certain limit, allocate it from a private arena rather than

// this frame. The current limit is based on an old assert on the number of locals we would allow here.

if (varAllocCount > InterpreterStackFrame::LocalsThreshold)

{

ArenaAllocator \*tmpAlloc = nullptr;

fReleaseAlloc = functionScriptContext->EnsureInterpreterArena(&tmpAlloc);

allocation = (Var\*)tmpAlloc->Alloc(varSizeInBytes);

stackAddr = reinterpret\_cast<DWORD\_PTR>(&allocation); // use a stack address so the debugger stepping logic works (step-out, for example, compares stack depths to determine when to complete the step)

}

else

{

PROBE\_STACK\_PARTIAL\_INITIALIZED\_INTERPRETER\_FRAME(functionScriptContext, Js::Constants::MinStackInterpreter + varSizeInBytes);

allocation = (Var\*)\_alloca(varSizeInBytes);

stackAddr = reinterpret\_cast<DWORD\_PTR>(allocation);

}

/\*

\* If the function has any loop headers, we allocate an array for the loop headers wrappers, and

\* reference the wrappers in the array. We then push the pointer to the array onto the stack itself.

\* We do this so that while the function is being interpreted, we don't want the jitted loop

\* bodies to be collected, even if the loop body isn't being executed. The loop body will

\* get collected when the function has been JITted, and when the function exits the interpreter.

\* The array contains nulls if the loop body isn't jitted (or hasn't been jitted yet) but

\* it's cheaper to just copy them all into the recycler array rather than just the ones that

\* have been jitted.

\*/

Var loopHeaderArray = nullptr;

if (executeFunction->GetHasAllocatedLoopHeaders())

{

// Loop header array is recycler allocated, so we push it on the stack

// When we scan the stack, we'll recognize it as a recycler allocated

// object, and mark it's contents and keep the individual loop header

// wrappers alive

loopHeaderArray = executeFunction->GetLoopHeaderArrayPtr();

}

#if DBG

Js::RecyclableObject \* invalidStackVar = (Js::RecyclableObject\*)\_alloca(sizeof(Js::RecyclableObject));

memset(invalidStackVar, 0xFE, sizeof(Js::RecyclableObject));

newInstance = setup.InitializeAllocation(allocation, executeFunction->GetHasImplicitArgIns() && !isAsmJs, doProfile, loopHeaderArray, stackAddr, invalidStackVar);

#else

newInstance = setup.InitializeAllocation(allocation, executeFunction->GetHasImplicitArgIns() && !isAsmJs, doProfile, loopHeaderArray, stackAddr);

#endif

newInstance->m\_reader.Create(executeFunction);

}

//

// Execute the function's byte-code, returning the return-value:

// - Mark that the function is current executing and may not be modified.

//

executeFunction->BeginExecution();

Var aReturn = nullptr;

{

if (!isAsmJs && functionScriptContext->IsInDebugMode())

{

#if DYNAMIC\_INTERPRETER\_THUNK

PushPopFrameHelper pushPopFrameHelper(newInstance, returnAddress, addressOfReturnAddress);

aReturn = newInstance->DebugProcess();

#else

aReturn = newInstance->DebugProcessThunk(\_ReturnAddress(), \_AddressOfReturnAddress());

#endif

}

else

{

#if DYNAMIC\_INTERPRETER\_THUNK

PushPopFrameHelper pushPopFrameHelper(newInstance, returnAddress, addressOfReturnAddress);

aReturn = newInstance->Process();

#else

aReturn = newInstance->ProcessThunk(\_ReturnAddress(), \_AddressOfReturnAddress());

#endif

}

}

executeFunction->EndExecution();

if (fReleaseAlloc)

{

functionScriptContext->ReleaseInterpreterArena();

}

#if ENABLE\_PROFILE\_INFO

if (doProfile)

{

dynamicProfileInfo->RecordImplicitCallFlags(threadContext->GetImplicitCallFlags());

}

#endif

if (isAsmJs)

{

return newInstance;

}

return aReturn;

}

#ifndef TEMP\_DISABLE\_ASMJS

#if \_M\_IX86

int InterpreterStackFrame::AsmJsInterpreter(AsmJsCallStackLayout\* stack)

{

ScriptFunction \* function = (ScriptFunction\*)stack->functionObject;

Var\* paramsAddr = stack->args;

int flags = CallFlags\_Value;

ArgSlot nbArgs = UInt16Math::Add(function->GetFunctionBody()->GetAsmJsFunctionInfo()->GetArgCount(), 1);

CallInfo callInfo((CallFlags)flags, nbArgs);

ArgumentReader args(&callInfo, paramsAddr);

void\* returnAddress = \_ReturnAddress();

void\* addressOfReturnAddress = \_AddressOfReturnAddress();

#if ENABLE\_PROFILE\_INFO

function->GetFunctionBody()->EnsureDynamicProfileInfo();

#endif

InterpreterStackFrame\* newInstance = (InterpreterStackFrame\*)InterpreterHelper(function, args, returnAddress, addressOfReturnAddress, true);

//Handle return value

AsmJsRetType::Which retType = (AsmJsRetType::Which) GetRetType(function);

int retVal = 0;

switch (retType)

{

case AsmJsRetType::Int32x4:

case AsmJsRetType::Float32x4:

case AsmJsRetType::Float64x2:

if (function->GetScriptContext()->GetConfig()->IsSimdjsEnabled())

{

function->GetScriptContext()->retAsmSimdVal = newInstance->m\_localSimdSlots[0];

break;

}

Assert(UNREACHED);

// double return

case AsmJsRetType::Double:

function->GetScriptContext()->retAsmIntDbVal = newInstance->m\_localDoubleSlots[0];

break;

// float return

case AsmJsRetType::Float:

function->GetScriptContext()->retAsmIntDbVal = (double)newInstance->m\_localFloatSlots[0];

break;

// signed or void return

case AsmJsRetType::Signed:

case AsmJsRetType::Void:

retVal = newInstance->m\_localIntSlots[0];

break;

default:

Assume(false);

}

return retVal;

}

#elif \_M\_X64

typedef double(\*AsmJsInterpreterDoubleEP)(AsmJsCallStackLayout\*, void \*);

typedef float(\*AsmJsInterpreterFloatEP)(AsmJsCallStackLayout\*, void \*);

typedef int(\*AsmJsInterpreterIntEP)(AsmJsCallStackLayout\*, void \*);

void \* InterpreterStackFrame::GetAsmJsInterpreterEntryPoint(AsmJsCallStackLayout\* stack)

{

JavascriptFunction \* function = stack->functionObject;

void \* entryPoint = nullptr;

switch (function->GetFunctionBody()->GetAsmJsFunctionInfo()->GetReturnType().which())

{

case Js::AsmJsRetType::Double:

{

entryPoint = (AsmJsInterpreterDoubleEP)Js::InterpreterStackFrame::AsmJsInterpreter < double > ;

break;

}

case Js::AsmJsRetType::Float:

{

entryPoint = (AsmJsInterpreterFloatEP)Js::InterpreterStackFrame::AsmJsInterpreter < float > ;

break;

}

case Js::AsmJsRetType::Signed:

case Js::AsmJsRetType::Void:

{

entryPoint = (AsmJsInterpreterIntEP)Js::InterpreterStackFrame::AsmJsInterpreter < int > ;

break;

}

case Js::AsmJsRetType::Int32x4:

case Js::AsmJsRetType::Float32x4:

case Js::AsmJsRetType::Float64x2:

{

entryPoint = Js::InterpreterStackFrame::AsmJsInterpreterSimdJs;

break;

}

default:

Assume(UNREACHED);

}

return entryPoint;

}

template<>

int InterpreterStackFrame::GetAsmJsRetVal<int>(InterpreterStackFrame\* instance)

{

return instance->m\_localIntSlots[0];

}

template<>

double InterpreterStackFrame::GetAsmJsRetVal<double>(InterpreterStackFrame\* instance)

{

return instance->m\_localDoubleSlots[0];

}

template<>

float InterpreterStackFrame::GetAsmJsRetVal<float>(InterpreterStackFrame\* instance)

{

return instance->m\_localFloatSlots[0];

}

template<>

X86SIMDValue InterpreterStackFrame::GetAsmJsRetVal<X86SIMDValue>(InterpreterStackFrame\* instance)

{

return X86SIMDValue::ToX86SIMDValue(instance->m\_localSimdSlots[0]);

}

template<typename T>

T InterpreterStackFrame::AsmJsInterpreter(AsmJsCallStackLayout\* layout)

{

Js::ScriptFunction \* function = Js::ScriptFunction::FromVar(layout->functionObject);

int flags = CallFlags\_Value;

ArgSlot nbArgs = UInt16Math::Add(function->GetFunctionBody()->GetAsmJsFunctionInfo()->GetArgCount(), 1);

CallInfo callInfo((CallFlags)flags, nbArgs);

ArgumentReader args(&callInfo, (Var\*)layout->args);

void\* returnAddress = \_ReturnAddress();

void\* addressOfReturnAddress = \_AddressOfReturnAddress();

function->GetFunctionBody()->EnsureDynamicProfileInfo();

InterpreterStackFrame\* newInstance = (InterpreterStackFrame\*)InterpreterHelper(function, args, returnAddress, addressOfReturnAddress, true);

return GetAsmJsRetVal<T>(newInstance);

}

\_\_m128 InterpreterStackFrame::AsmJsInterpreterSimdJs(AsmJsCallStackLayout\* layout)

{

return AsmJsInterpreter<X86SIMDValue>(layout).m128\_value;

}

#endif

#endif

///----------------------------------------------------------------------------

///

/// InterpreterStackFrame::SetOut()

///

/// SetOut() change the Var value stored in the specified "out parameter"

/// register.

///

///----------------------------------------------------------------------------

inline void InterpreterStackFrame::SetOut(ArgSlot outRegisterID, Var aValue)

{

//

// The "out" parameter slots are located at the end of the local register range, counting

// forwards. This results in the "in" parameter slots being disjoint from the rest of the

// InterpreterStackFrame.

// ..., InterpreterStackFrame A, Locals A[], ..., Out A:0, Out A:1, Out A:2, ...

// | In B:0, In B:1, ..., InterpreterStackFrame B, Locals B[], ...

// (current 'this') |

// (new 'this' after call)

//

Assert(m\_outParams + outRegisterID < m\_outSp);

m\_outParams[outRegisterID] = aValue;

}

inline void InterpreterStackFrame::SetOut(ArgSlot\_OneByte outRegisterID, Var aValue)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

m\_outParams[outRegisterID] = aValue;

}

inline void InterpreterStackFrame::OP\_SetOutAsmDb( RegSlot outRegisterID, double val )

{

Assert( m\_outParams + outRegisterID < m\_outSp );

m\_outParams[outRegisterID] = JavascriptNumber::New( val, scriptContext );

}

inline void InterpreterStackFrame::OP\_SetOutAsmInt( RegSlot outRegisterID, int val )

{

Assert( m\_outParams + outRegisterID < m\_outSp );

m\_outParams[outRegisterID] = JavascriptNumber::ToVar( val, scriptContext );

}

inline void InterpreterStackFrame::OP\_I\_SetOutAsmFlt(RegSlot outRegisterID, float val)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

\*(float\*)(&(m\_outParams[outRegisterID])) = val;

}

inline void InterpreterStackFrame::OP\_I\_SetOutAsmInt(RegSlot outRegisterID, int val)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

\*(int\*)(&(m\_outParams[outRegisterID])) = val;

}

inline void InterpreterStackFrame::OP\_I\_SetOutAsmDb(RegSlot outRegisterID, double val)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

\*(double\*)(&(m\_outParams[outRegisterID])) = val;

}

inline void InterpreterStackFrame::OP\_I\_SetOutAsmSimd(RegSlot outRegisterID, AsmJsSIMDValue val)

{

Assert(m\_outParams + outRegisterID < m\_outSp);

\*(AsmJsSIMDValue\*)(&(m\_outParams[outRegisterID])) = val;

}

inline void InterpreterStackFrame::PushOut(Var aValue)

{

\*m\_outSp++ = aValue;

}

inline void InterpreterStackFrame::PopOut(ArgSlot argCount)

{

m\_outSp -= (argCount+1);

m\_outParams = (Var\*)\*m\_outSp;

AssertMsg(m\_localSlots + this->m\_functionBody->GetLocalsCount() <= m\_outSp &&

m\_outSp < (m\_localSlots + this->m\_functionBody->GetLocalsCount() + this->m\_functionBody->GetOutParamsDepth()),

"out args Stack pointer not in range after Pop");

}

void InterpreterStackFrame::ResetOut()

{

//

// Reset the m\_outParams and m\_outSp

//

m\_outParams = m\_localSlots + this->m\_functionBody->GetLocalsCount();

m\_outSp = m\_outParams;

}

\_\_declspec(noinline)

Var InterpreterStackFrame::DebugProcessThunk(void\* returnAddress, void\* addressOfReturnAddress)

{

PushPopFrameHelper pushPopFrameHelper(this, returnAddress, addressOfReturnAddress);

return this->DebugProcess();

}

//

// Under debug mode allow the exception to be swallowed and execution to continue

// if the debugger has specified that behavior.

//

Var InterpreterStackFrame::DebugProcess()

{

Assert(this->returnAddress != nullptr);

while (true)

{

JavascriptExceptionObject \*exception = nullptr;

try

{

return this->ProcessWithDebugging();

}

catch (JavascriptExceptionObject \*exception\_)

{

Assert(exception\_);

exception = exception\_;

}

if (exception)

{

bool skipException = false;

if (exception != scriptContext->GetThreadContext()->GetPendingSOErrorObject()

&& exception != scriptContext->GetThreadContext()->GetPendingOOMErrorObject())

{

skipException = exception->IsDebuggerSkip();

}

if (skipException)

{

// If we are going to swallow the exception then advance to the beginning of the next user statement

if (exception->IsIgnoreAdvanceToNextStatement()

|| this->scriptContext->GetDebugContext()->GetProbeContainer()->AdvanceToNextUserStatement(this->m\_functionBody, &this->m\_reader))

{

// We must fix up the return value to at least be undefined:

this->SetReg((RegSlot)0,this->scriptContext->GetLibrary()->GetUndefined());

// If we recover from the exception, there may be a chance the out pointers in the InterpreterStackframe are not in a proper state.

// Reset them to correct the stack.

ResetOut();

// If we can successfully advance then continuing processing

continue;

}

}

exception = exception->CloneIfStaticExceptionObject(scriptContext);

throw exception;

}

}

}

template<>

OpCode InterpreterStackFrame::ReadByteOp<OpCode>(const byte \*& ip

#if DBG\_DUMP

, bool isExtended /\*= false\*/

#endif

)

{

#if DBG || DBG\_DUMP

//

// For debugging byte-code, store the current offset before the instruction is read:

// - We convert this to "void \*" to encourage the debugger to always display in hex,

// which matches the displayed offsets used by ByteCodeDumper.

//

this->DEBUG\_currentByteOffset = (void \*) m\_reader.GetCurrentOffset();

#endif

OpCode op = ByteCodeReader::ReadByteOp(ip);

#if DBG\_DUMP

this->scriptContext->byteCodeHistogram[(int)op]++;

if (PHASE\_TRACE(Js::InterpreterPhase, this->m\_functionBody))

{

Output::Print(L"%d.%d:Executing %s at offset 0x%X\n", this->m\_functionBody->GetSourceContextId(), this->m\_functionBody->GetLocalFunctionId(), Js::OpCodeUtil::GetOpCodeName((Js::OpCode)(op+((int)isExtended<<8))), DEBUG\_currentByteOffset);

}

#endif

return op;

}

#ifndef TEMP\_DISABLE\_ASMJS

template<>

OpCodeAsmJs InterpreterStackFrame::ReadByteOp<OpCodeAsmJs>(const byte \*& ip

#if DBG\_DUMP

, bool isExtended /\*= false\*/

#endif

)

{

#if DBG || DBG\_DUMP

//

// For debugging byte-code, store the current offset before the instruction is read:

// - We convert this to "void \*" to encourage the debugger to always display in hex,

// which matches the displayed offsets used by ByteCodeDumper.

//

this->DEBUG\_currentByteOffset = (void \*) m\_reader.GetCurrentOffset();

#endif

OpCodeAsmJs op = (OpCodeAsmJs)ByteCodeReader::ReadByteOp(ip);

#if DBG\_DUMP

if (PHASE\_TRACE(Js::AsmjsInterpreterPhase, this->m\_functionBody))

{

Output::Print(L"%d.%d:Executing %s at offset 0x%X\n", this->m\_functionBody->GetSourceContextId(), this->m\_functionBody->GetLocalFunctionId(), Js::OpCodeUtilAsmJs::GetOpCodeName((Js::OpCodeAsmJs)(op+((int)isExtended<<8))), DEBUG\_currentByteOffset);

}

#endif

return op;

}

#endif

\_\_declspec(noinline)

Var InterpreterStackFrame::ProcessThunk(void\* address, void\* addressOfReturnAddress)

{

PushPopFrameHelper pushPopFrameHelper(this, address, addressOfReturnAddress);

return this->Process();

}

template<> uint32 InterpreterStackFrame::LogSizeOf<uint8>(){return 0;}

template<> uint32 InterpreterStackFrame::LogSizeOf<int8>(){return 0;}

template<> uint32 InterpreterStackFrame::LogSizeOf<uint16>(){return 1;}

template<> uint32 InterpreterStackFrame::LogSizeOf<int16>(){return 1;}

template<> uint32 InterpreterStackFrame::LogSizeOf<uint32>(){return 2;}

template<> uint32 InterpreterStackFrame::LogSizeOf<int32>(){return 2;}

template<> uint32 InterpreterStackFrame::LogSizeOf<float>(){return 2;}

template<> uint32 InterpreterStackFrame::LogSizeOf<double>(){return 3;}

Var InterpreterStackFrame::ProcessAsmJsModule()

{

#ifdef ASMJS\_PLAT

Js::FunctionBody\* asmJsModuleFunctionBody = GetFunctionBody();

AsmJsModuleInfo\* info = asmJsModuleFunctionBody->GetAsmJsModuleInfo();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Configuration::Global.flags.ForceAsmJsLinkFail)

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Forcing link failure");

return this->ProcessLinkFailedAsmJsModule();

}

#endif

if( m\_inSlotsCount != info->GetArgInCount() + 1 )

{

// Error reparse without asm.js

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Invalid module argument count");

return this->ProcessLinkFailedAsmJsModule();

}

const AsmJsModuleMemory& moduleMemory = info->GetModuleMemory();

Var\* moduleMemoryPtr = RecyclerNewArray( scriptContext->GetRecycler(), Var, moduleMemory.mMemorySize );

Var\* arrayBufferPtr = moduleMemoryPtr + moduleMemory.mArrayBufferOffset;

Assert(moduleMemory.mArrayBufferOffset == AsmJsModuleMemory::MemoryTableBeginOffset);

Var\* stdLibPtr = moduleMemoryPtr + moduleMemory.mStdLibOffset;

int\* localIntSlots = (int\*)(moduleMemoryPtr + moduleMemory.mIntOffset);

float\* localFloatSlots = (float\*)(moduleMemoryPtr + moduleMemory.mFloatOffset);

double\* localDoubleSlots = (double\*)(moduleMemoryPtr + moduleMemory.mDoubleOffset);

Var\* localFunctionImports = moduleMemoryPtr + moduleMemory.mFFIOffset ;

Var\* localModuleFunctions = moduleMemoryPtr + moduleMemory.mFuncOffset ;

Var\*\* localFunctionTables = (Var\*\*)(moduleMemoryPtr + moduleMemory.mFuncPtrOffset) ;

AsmJsSIMDValue\* localSimdSlots = nullptr;

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

localSimdSlots = ((AsmJsSIMDValue\*)moduleMemoryPtr) + moduleMemory.mSimdOffset; // simdOffset is in SIMDValues

}

#if 0

// Align SIMD regs to 128 bits.

// We only have space to align if there are any SIMD variables. Otherwise, leave unaligned.

if (info->GetSimdRegCount())

{

AssertMsg((moduleMemory.mMemorySize / SIMD\_SLOTS\_SPACE) - moduleMemory.mSimdOffset >= 1, "Not enough space in module memory to align SIMD vars");

localSimdSlots = (AsmJsSIMDValue\*)::Math::Align<int>((int)localSimdSlots, sizeof(AsmJsSIMDValue));

}

#endif

ThreadContext\* threadContext = this->scriptContext->GetThreadContext();

\*stdLibPtr = (m\_inSlotsCount > 1) ? m\_inParams[1] : nullptr;

Var foreign = (m\_inSlotsCount > 2) ? m\_inParams[2] : nullptr;

\*arrayBufferPtr = (m\_inSlotsCount > 3) ? m\_inParams[3] : nullptr;

//cache the current state of the disable implicit call flag

DisableImplicitFlags prevDisableImplicitFlags = threadContext->GetDisableImplicitFlags();

ImplicitCallFlags saveImplicitcallFlags = threadContext->GetImplicitCallFlags();

// Disable implicit calls to check if any of the VarImport or Function Import leads to implicit calls

threadContext->DisableImplicitCall();

threadContext->SetImplicitCallFlags(ImplicitCallFlags::ImplicitCall\_None);

bool checkParamResult = ASMLink::CheckParams(this->scriptContext, info, \*stdLibPtr, foreign, \*arrayBufferPtr);

if (!checkParamResult)

{

// don't need to print, because checkParams will do it for us

goto linkFailure;

}

else if(this->CheckAndResetImplicitCall(prevDisableImplicitFlags, saveImplicitcallFlags))

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Params have side effects");

return this->ProcessLinkFailedAsmJsModule();

}

// Initialize Variables

for (int i = 0; i < info->GetVarCount(); i++)

{

const auto& var = info->GetVar( i );

const AsmJsVarType type(var.type);

if(type.isInt() )

{

localIntSlots[var.location] = var.initialiser.intInit;

}

else if (type.isFloat())

{

localFloatSlots[var.location] = var.initialiser.floatInit;

}

else if (type.isDouble())

{

localDoubleSlots[var.location] = var.initialiser.doubleInit;

}

else if (scriptContext->GetConfig()->IsSimdjsEnabled() && type.isSIMD())

{

// e.g. var g = f4(0.0, 0.0, 0.0, 0.0);

localSimdSlots[var.location] = var.initialiser.simdInit;

}

else {

Assert(UNREACHED);

}

}

// Load constant variables

for( int i = 0; i < info->GetVarImportCount(); i++ )

{

const auto& import = info->GetVarImport( i );

const AsmJsVarType type(import.type);

// this might throw, but it would anyway in non-asm.js

Var value = JavascriptOperators::OP\_GetProperty( foreign, import.field, scriptContext );

// check if there is implicit call and if there is implicit call then clear the disableimplicitcall flag

if (this->CheckAndResetImplicitCall(prevDisableImplicitFlags, saveImplicitcallFlags))

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Accessing var import %s has side effects", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

return this->ProcessLinkFailedAsmJsModule();

}

if (CONFIG\_FLAG(AsmJsEdge))

{

// emscripten had a bug which caused this check to fail in some circumstances, so this check fails for some demos

if (!TaggedNumber::Is(value) && (!RecyclableObject::Is(value) || DynamicType::Is(RecyclableObject::FromVar(value)->GetTypeId())))

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Var import %s must be primitive", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

goto linkFailure;

}

}

if(type.isInt() )

{

int val = JavascriptMath::ToInt32( value, scriptContext );

localIntSlots[import.location] = val;

}

else if (type.isFloat())

{

float val = (float)JavascriptConversion::ToNumber(value, scriptContext);

localFloatSlots[import.location] = val;

}

else if (type.isDouble())

{

double val = JavascriptConversion::ToNumber( value, scriptContext );

localDoubleSlots[import.location] = val;

}

else if (scriptContext->GetConfig()->IsSimdjsEnabled() && type.isSIMD())

{

// e.g. var g = f4(imports.v);

bool valid = false;

AsmJsSIMDValue val;

val.Zero();

switch (type.which())

{

case AsmJsVarType::Int32x4:

valid = JavascriptSIMDInt32x4::Is(value);

val = ((JavascriptSIMDInt32x4\*)value)->GetValue();

break;

case AsmJsVarType::Float32x4:

valid = JavascriptSIMDFloat32x4::Is(value);

val = ((JavascriptSIMDFloat32x4\*)value)->GetValue();

break;

case AsmJsVarType::Float64x2:

valid = JavascriptSIMDFloat64x2::Is(value);

val = ((JavascriptSIMDFloat64x2\*)value)->GetValue();

break;

default:

Assert(UNREACHED);

};

if (!valid)

{

// link failure of SIMD values imports.

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Foreign var import %s is not SIMD type", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

goto linkFailure;

}

localSimdSlots[import.location] = val;

}

// check for implicit call after converting to number

if (this->CheckAndResetImplicitCall(prevDisableImplicitFlags, saveImplicitcallFlags))

{

// Runtime error

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Accessing var import %s has side effects", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

return this->ProcessLinkFailedAsmJsModule();

}

}

// Load external functions

for( int i = 0; i < info->GetFunctionImportCount(); i++ )

{

const auto& import = info->GetFunctionImport( i );

// this might throw, but it would anyway in non-asm.js

Var importFunc = JavascriptOperators::OP\_GetProperty( foreign, import.field, scriptContext );

// check if there is implicit call and if there is implicit call then clear the disableimplicitcall flag

if (this->CheckAndResetImplicitCall(prevDisableImplicitFlags, saveImplicitcallFlags))

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Accessing foreign function import %s has side effects", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

return this->ProcessLinkFailedAsmJsModule();

}

if( !JavascriptFunction::Is( importFunc ) )

{

AsmJSCompiler::OutputError(this->scriptContext, L"Asm.js Runtime Error : Foreign function import %s is not a function", this->scriptContext->GetPropertyName(import.field)->GetBuffer());

goto linkFailure;

}

localFunctionImports[import.location] = importFunc;

}

if (\*arrayBufferPtr)

{

(\*(ArrayBuffer\*\*)arrayBufferPtr)->SetIsAsmJsBuffer();

}

threadContext->SetDisableImplicitFlags(prevDisableImplicitFlags);

threadContext->SetImplicitCallFlags(saveImplicitcallFlags);

FrameDisplay\* pDisplay = RecyclerNewPlus(scriptContext->GetRecycler(), sizeof(void\*), FrameDisplay, 1);

pDisplay->SetItem( 0, moduleMemoryPtr );

for (int i = 0; i < info->GetFunctionCount(); i++)

{

const auto& modFunc = info->GetFunction(i);

// TODO: add more runtime checks here

auto proxy = m\_functionBody->GetNestedFuncReference(i);

AsmJsScriptFunction\* scriptFuncObj = (AsmJsScriptFunction\*)ScriptFunction::OP\_NewScFunc(pDisplay, (FunctionProxy\*\*)proxy);

localModuleFunctions[modFunc.location] = scriptFuncObj;

if (i == 0 && info->GetUsesChangeHeap())

{

scriptFuncObj->GetDynamicType()->SetEntryPoint(AsmJsChangeHeapBuffer);

}

else

{

if (scriptFuncObj->GetDynamicType()->GetEntryPoint() == DefaultDeferredDeserializeThunk)

{

JavascriptFunction::DeferredDeserialize(scriptFuncObj);

}

scriptFuncObj->GetDynamicType()->SetEntryPoint(AsmJsExternalEntryPoint);

scriptFuncObj->GetFunctionBody()->GetAsmJsFunctionInfo()->SetModuleFunctionBody(asmJsModuleFunctionBody);

}

scriptFuncObj->SetModuleMemory(moduleMemoryPtr);

if (!info->IsRuntimeProcessed())

{

// don't reset entrypoint upon relinking

FunctionEntryPointInfo\* entypointInfo = (FunctionEntryPointInfo\*)scriptFuncObj->GetEntryPointInfo();

entypointInfo->SetIsAsmJSFunction(true);

entypointInfo->SetModuleAddress((uintptr\_t)moduleMemoryPtr);

#if DYNAMIC\_INTERPRETER\_THUNK

if (!PHASE\_ON1(AsmJsJITTemplatePhase))

{

entypointInfo->address = AsmJsDefaultEntryThunk;

}

#endif

}

}

// Initialize function table arrays

for( int i = 0; i < info->GetFunctionTableCount(); i++ )

{

const auto& modFuncTable = info->GetFunctionTable( i );

Var\* funcTableArray = RecyclerNewArray( scriptContext->GetRecycler(), Var, modFuncTable.size );

for (uint j = 0; j < modFuncTable.size ; j++)

{

// get the module function index

const RegSlot index = modFuncTable.moduleFunctionIndex[j];

// assign the module function pointer to the array

Var functionPtr = localModuleFunctions[index];

funcTableArray[j] = functionPtr;

}

localFunctionTables[i] = funcTableArray;

}

// Do MTJRC/MAIC:0 check

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if ((PHASE\_ON1(Js::AsmJsJITTemplatePhase) && CONFIG\_FLAG(MaxTemplatizedJitRunCount) == 0) || (!PHASE\_ON1(Js::AsmJsJITTemplatePhase) && CONFIG\_FLAG(MaxAsmJsInterpreterRunCount) == 0))

{

if (PHASE\_TRACE1(AsmjsEntryPointInfoPhase))

{

Output::Print(L"%s Scheduling For Full JIT at callcount:%d\n", asmJsModuleFunctionBody->GetDisplayName(), 0);

Output::Flush();

}

for (int i = 0; i < info->GetFunctionCount(); i++)

{

ScriptFunction\* functionObj = (ScriptFunction\*)localModuleFunctions[i];

AnalysisAssert(functionObj != nullptr);

// don't want to generate code for APIs like changeHeap

if (functionObj->GetEntryPoint() == Js::AsmJsExternalEntryPoint)

{

GenerateFunction(asmJsModuleFunctionBody->GetScriptContext()->GetNativeCodeGenerator(), functionObj->GetFunctionBody(), functionObj);

}

}

}

#endif

info->SetIsRuntimeProcessed(true);

// create export object

if( info->GetExportsCount() )

{

Var newObj = JavascriptOperators::NewScObjectLiteral( GetScriptContext(), info->GetExportsIdArray(),

this->GetFunctionBody()->GetObjectLiteralTypeRef( 0 ) );

for( int i = 0; i < info->GetExportsCount(); i++ )

{

auto ex = info->GetExport( i );

Var func = localModuleFunctions[\*ex.location];

JavascriptOperators::OP\_InitProperty( newObj, \*ex.id, func );

}

SetReg( (RegSlot) 0, newObj );

return newObj;

}

// export only 1 function

Var exportFunc = localModuleFunctions[info->GetExportFunctionIndex()];

SetReg((RegSlot)0, exportFunc);

return exportFunc;

linkFailure:

threadContext->SetDisableImplicitFlags(prevDisableImplicitFlags);

threadContext->SetImplicitCallFlags(saveImplicitcallFlags);

return this->ProcessLinkFailedAsmJsModule();

}

Var InterpreterStackFrame::ProcessLinkFailedAsmJsModule()

{

AsmJSCompiler::OutputError(this->scriptContext, L"asm.js linking failed.");

Js::FunctionBody\* asmJsModuleFunctionBody = GetFunctionBody();

AsmJsModuleInfo\* info = asmJsModuleFunctionBody->GetAsmJsModuleInfo();

// do not support relinking with failed relink

if (info->IsRuntimeProcessed())

{

Js::Throw::OutOfMemory();

}

ScriptFunction \* funcObj = GetJavascriptFunction();

ScriptFunction::ReparseAsmJsModule(&funcObj);

const bool doProfile =

funcObj->GetFunctionBody()->GetInterpreterExecutionMode(false) == ExecutionMode::ProfilingInterpreter ||

GetScriptContext()->IsInDebugMode() && DynamicProfileInfo::IsEnabled(funcObj->GetFunctionBody());

DynamicProfileInfo \* dynamicProfileInfo = nullptr;

if (doProfile)

{

dynamicProfileInfo = funcObj->GetFunctionBody()->GetDynamicProfileInfo();

funcObj->GetScriptContext()->GetThreadContext()->ClearImplicitCallFlags();

}

// after reparsing, we want to also use a new interpreter stack frame, as it will have different characteristics than the asm.js version

InterpreterStackFrame::Setup setup(funcObj, m\_inParams, m\_inSlotsCount);

size\_t varAllocCount = setup.GetAllocationVarCount();

size\_t varSizeInBytes = varAllocCount \* sizeof(Var);

Var\* allocation = nullptr;

DWORD\_PTR stackAddr;

bool fReleaseAlloc = false;

if (varAllocCount > InterpreterStackFrame::LocalsThreshold)

{

ArenaAllocator \*tmpAlloc = nullptr;

fReleaseAlloc = GetScriptContext()->EnsureInterpreterArena(&tmpAlloc);

allocation = (Var\*)tmpAlloc->Alloc(varSizeInBytes);

// use a stack address so the debugger stepping logic works (step-out, for example, compares stack depths to determine when to complete the step)

// debugger stepping does not matter here, but it's worth being consistent with normal stack frame

stackAddr = reinterpret\_cast<DWORD\_PTR>(&allocation);

}

else

{

PROBE\_STACK\_PARTIAL\_INITIALIZED\_INTERPRETER\_FRAME(GetScriptContext(), Js::Constants::MinStackInterpreter + varSizeInBytes);

allocation = (Var\*)\_alloca(varSizeInBytes);

stackAddr = reinterpret\_cast<DWORD\_PTR>(allocation);

}

#if DBG

Js::RecyclableObject \* invalidStackVar = (Js::RecyclableObject\*)\_alloca(sizeof(Js::RecyclableObject));

memset(invalidStackVar, 0xFE, sizeof(Js::RecyclableObject));

InterpreterStackFrame \* newInstance = newInstance = setup.InitializeAllocation(allocation, funcObj->GetFunctionBody()->GetHasImplicitArgIns(), doProfile, nullptr, stackAddr, invalidStackVar);

#else

InterpreterStackFrame \* newInstance = newInstance = setup.InitializeAllocation(allocation, funcObj->GetFunctionBody()->GetHasImplicitArgIns(), doProfile, nullptr, stackAddr);

#endif

newInstance->m\_reader.Create(funcObj->GetFunctionBody());

// now that we have set up the new frame, let's interpret it!

funcObj->GetFunctionBody()->BeginExecution();

PushPopFrameHelper(newInstance, \_ReturnAddress(), \_AddressOfReturnAddress());

Var retVal = newInstance->ProcessUnprofiled();

if (doProfile)

{

dynamicProfileInfo->RecordImplicitCallFlags(GetScriptContext()->GetThreadContext()->GetImplicitCallFlags());

}

if (fReleaseAlloc)

{

GetScriptContext()->ReleaseInterpreterArena();

}

return retVal;

#else

Assert(UNREACHED);

return nullptr;

#endif

}

#if DBG\_DUMP

int AsmJsCallDepth = 0;

#endif

void InterpreterStackFrame::PrintStack(const int\* const intSrc, const float\* const fltSrc, const double\* const dblSrc, int intConstCount, int floatConstCount, int doubleConstCount, const wchar\_t\* state)

{

Output::Print(L"\n");

Output::Print(L"Interpreter Constant Stack Data(%s)\n", state);

Output::Print(L"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

Output::Print(L"Int Data\n");

Output::Print(L"--------\n");

for (int count = 0; count < intConstCount; count++)

{

Output::Print(L"Index:%d Value:%d \n", count, intSrc[count]);

}

Output::Print(L"\n");

Output::Print(L"Float Data\n");

Output::Print(L"----------\n");

for (int count = 0; count < floatConstCount; count++)

{

Output::Print(L"Index:%d Value:%f \n", count, fltSrc[count]);

}

Output::Print(L"\n");

Output::Print(L"Double Data\n");

Output::Print(L"-----------\n");

for (int count = 0; count < doubleConstCount; count++)

{

Output::Print(L"Index:%d Value:%g \n", count, dblSrc[count]);

}

Output::Print(L"\n");

}

#ifndef TEMP\_DISABLE\_ASMJS

// Function memory allocation should be done the same way as

// T AsmJsCommunEntryPoint(Js::ScriptFunction\* func, ...) (AsmJSJitTemplate.cpp)

// update any changes there

/\*

This function does the following fixup

Stack Before Stack After

============== ================

| VarConstants | | VarConstants |

|--------------| |-----------------

| IntConstants | | IntConstants |

|--------------| | ------------ |

| FloatConst | | Int Vars+Tmps |

|--------------| |----------------|

| DoubleConst | | FloatConst |

|--------------| | ---------- |

| Var&Temps | | Flt Vars+tmps |

|==============| |----------------|

| DoubleConst |

| ----------- |

| Dbl Vars+Tmps |

================

intSrc,FltSrc&DblSrc are pointers to the stack before the change

m\_localIntSlots,m\_localFloatSlots,m\_localDoubleSlots are pointers to the stack after the change

\*/

void InterpreterStackFrame::AlignMemoryForAsmJs()

{

FunctionBody \*const functionBody = GetFunctionBody();

ScriptFunction\* func = GetJavascriptFunction();

//schedule for codegen here only if TJ is collected

if (!functionBody->GetIsAsmJsFullJitScheduled() && !PHASE\_OFF(BackEndPhase, functionBody)

&& !PHASE\_OFF(FullJitPhase, functionBody) && !this->scriptContext->GetConfig()->IsNoNative())

{

int callCount = ++((FunctionEntryPointInfo\*)func->GetEntryPointInfo())->callsCount;

bool doSchedule = false;

const int minAsmJsInterpretRunCount = (int)CONFIG\_FLAG(MinAsmJsInterpreterRunCount);

if (callCount >= minAsmJsInterpretRunCount)

{

doSchedule = true;

}

if (doSchedule && !functionBody->GetIsAsmJsFullJitScheduled())

{

#if ENABLE\_NATIVE\_CODEGEN

if (PHASE\_TRACE1(AsmjsEntryPointInfoPhase))

{

Output::Print(L"Scheduling For Full JIT from Interpreter at callcount:%d\n", callCount);

}

GenerateFunction(functionBody->GetScriptContext()->GetNativeCodeGenerator(), functionBody, func);

#endif

functionBody->SetIsAsmJsFullJitScheduled(true);

}

}

AsmJsFunctionInfo\* info = functionBody->GetAsmJsFunctionInfo();

const int intConstCount = info->GetIntConstCount();

const int doubleConstCount = info->GetDoubleConstCount();

const int floatConstCount = info->GetFloatConstCount();

const int simdConstCount = info->GetSimdConstCount();

// Offset of doubles from (double\*)m\_localSlot

const int intOffset = info->GetIntByteOffset() / sizeof(int);

const int doubleOffset = info->GetDoubleByteOffset() / sizeof(double);

const int floatOffset = info->GetFloatByteOffset() / sizeof(float);

const int simdByteOffset = info->GetSimdByteOffset();// in bytes;

int\* intSrc = (int\*)(m\_localSlots + AsmJsFunctionMemory::RequiredVarConstants);

// Where all int value starts

m\_localIntSlots = ((int\*)m\_localSlots) + intOffset;

// where int arguments starts

// int\* intArgDst = m\_localIntSlots + intConstCount;

// Where float constants currently are

float\* floatSrc = (float\*)(intSrc + intConstCount);

// where all float value starts with the new layout

m\_localFloatSlots = ((float\*)m\_localSlots) + floatOffset;

// Where double arguments starts

// float\* floatArgDst = m\_localFloatSlots + floatConstCount;

// Where double constants currently are

double\* doubleSrc = (double\*)(floatSrc + floatConstCount);

// where all double value starts

m\_localDoubleSlots = ((double\*)m\_localSlots) + doubleOffset;

// Where double arguments starts

// double\* doubleArgDst = m\_localDoubleSlots + doubleConstCount;

AsmJsSIMDValue\* simdSrc = nullptr;

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

simdSrc = (AsmJsSIMDValue\*)(doubleSrc + doubleConstCount);

m\_localSimdSlots = (AsmJsSIMDValue\*)((char\*)m\_localSlots + simdByteOffset);

}

// Load module environment

FrameDisplay\* frame = this->function->GetEnvironment();

m\_localSlots[AsmJsFunctionMemory::ModuleEnvRegister] = frame->GetItem(0);

m\_localSlots[AsmJsFunctionMemory::ArrayBufferRegister] = (Var\*)frame->GetItem(0) + AsmJsModuleMemory::MemoryTableBeginOffset;

m\_localSlots[AsmJsFunctionMemory::ArraySizeRegister] = 0; // do not cache ArraySize in the interpreter

m\_localSlots[AsmJsFunctionMemory::ScriptContextBufferRegister] = functionBody->GetScriptContext();

if (PHASE\_TRACE1(AsmjsInterpreterStackPhase))

{

PrintStack(intSrc, floatSrc, doubleSrc, intConstCount, floatConstCount, doubleConstCount, L"Before Shuffling");

}

// Copying has to happen in that order in order not to overwrite constants

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

memcpy\_s(m\_localSimdSlots, simdConstCount\*sizeof(AsmJsSIMDValue), simdSrc, simdConstCount\*sizeof(AsmJsSIMDValue));

}

// Moving the double and floats to their slot position. We must move the doubles first so that we do not overwrite the doubles stack with floats

memcpy\_s(m\_localDoubleSlots, doubleConstCount\*sizeof(double), doubleSrc, doubleConstCount\*sizeof(double));

memcpy\_s(m\_localFloatSlots, floatConstCount\*sizeof(float), floatSrc, floatConstCount\*sizeof(float));

if (PHASE\_TRACE1(AsmjsInterpreterStackPhase))

{

PrintStack(m\_localIntSlots, m\_localFloatSlots, m\_localDoubleSlots, intConstCount, floatConstCount, doubleConstCount, L"After Shuffling");

}

int\* intArg;

double\* doubleArg;

float\* floatArg;

intArg = m\_localIntSlots + intConstCount;

doubleArg = m\_localDoubleSlots + doubleConstCount;

floatArg = m\_localFloatSlots + floatConstCount;

AsmJsSIMDValue\* simdArg = m\_localSimdSlots + simdConstCount;

// Move the arguments to the right location

ArgSlot argCount = info->GetArgCount();

#if \_M\_X64

uint homingAreaSize = 0;

#endif

uintptr argAddress = (uintptr)m\_inParams;

for (ArgSlot i = 0; i < argCount; i++)

{

#if \_M\_X64

// 3rd Argument should be at the end of the homing area.

Assert(i != 3 || argAddress == (uintptr)m\_inParams + homingAreaSize);

if (i < 3)

{

// for x64 we spill the first 3 floating point args below the rest of the arguments on the stack

// m\_inParams will be from DynamicInterpreterThunk's frame. Floats are in InterpreterAsmThunk's frame. Stack will be set up like so:

// DIT arg2 <- first scriptArg, m\_inParams points here

// DIT arg1

// padding

// IAT r9 home

// IAT r8 home

// IAT rdx home

// IAT rcx home

// IAT return address

// IAT push rbp

// IAT padding

// IAT xmm3 spill

// IAT xmm2 spill

// IAT xmm1 spill <- floatSpillAddress for arg1

// floats are spilled as xmmwords

uintptr floatSpillAddress = (uintptr)m\_inParams - MachPtr \* (15 - 2\*i);

if (info->GetArgType(i).isInt())

{

\*intArg = \*(int\*)argAddress;

++intArg;

homingAreaSize += MachPtr;

}

else if (info->GetArgType(i).isFloat())

{

\*floatArg = \*(float\*)floatSpillAddress;

++floatArg;

homingAreaSize += MachPtr;

}

else if (info->GetArgType(i).isDouble())

{

\*doubleArg = \*(double\*)floatSpillAddress;

++doubleArg;

homingAreaSize += MachPtr;

}

else

{

Assert(info->GetArgType(i).isSIMD());

\*simdArg = \*(AsmJsSIMDValue\*)floatSpillAddress;

++simdArg;

homingAreaSize += sizeof(AsmJsSIMDValue);

}

if (scriptContext->GetConfig()->IsSimdjsEnabled() && i == 2) // last argument ?

{

// If we have simd arguments, the homing area in m\_inParams can be larger than 3 64-bit slots. This is because SIMD values are unboxed there too.

// After unboxing, the homing area is overwritten by rdx, r8 and r9, and we read/skip 64-bit slots from the homing area (argAddress += MachPtr).

// After the last argument of the 3 is read, we need to advance argAddress to skip over the possible extra space and to the start of the rest of the arguments.

argAddress = (uintptr)m\_inParams + homingAreaSize;

}

else

{

argAddress += MachPtr;

}

}

else

#endif

if (info->GetArgType(i).isInt())

{

\*intArg = \*(int\*)argAddress;

++intArg;

argAddress += MachPtr;

}

else if (info->GetArgType(i).isFloat())

{

\*floatArg = \*(float\*)argAddress;

++floatArg;

argAddress += MachPtr;

}

else if (info->GetArgType(i).isDouble())

{

Assert(info->GetArgType(i).isDouble());

\*doubleArg = \*(double\*)argAddress;

++doubleArg;

argAddress += sizeof(double);

}

else if (scriptContext->GetConfig()->IsSimdjsEnabled() && info->GetArgType(i).isSIMD())

{

\*simdArg = \*(AsmJsSIMDValue\*)argAddress;

++simdArg;

argAddress += sizeof(AsmJsSIMDValue);

}

else

{

AssertMsg(UNREACHED, "Invalid function arg type.");

}

}

#if DBG\_DUMP

const bool tracingFunc = PHASE\_TRACE( AsmjsFunctionEntryPhase, functionBody );

if( tracingFunc )

{

if( AsmJsCallDepth )

{

Output::Print( L"%\*c", AsmJsCallDepth,' ');

}

Output::Print( L"Executing function %s", functionBody->GetDisplayName());

++AsmJsCallDepth;

}

#endif

#if DBG\_DUMP

if (tracingFunc)

{

Output::Print(L"){\n");

}

#endif

if( info->GetReturnType() == AsmJsRetType::Void )

{

m\_localSlots[0] = JavascriptOperators::OP\_LdUndef( scriptContext );

}

}

#endif

///----------------------------------------------------------------------------

///

/// InterpreterStackFrame::Process

///

/// Process() processes a single loop of execution for the current

/// JavascriptFunction being executed:

/// - Individual instructions are dispatched to specific handlers for different

/// OpCodes.

///

///----------------------------------------------------------------------------

#if ENABLE\_PROFILE\_INFO

#define INTERPRETERLOOPNAME ProcessProfiled

#define PROVIDE\_INTERPRETERPROFILE

#include "Interpreterloop.inl"

#undef PROVIDE\_INTERPRETERPROFILE

#undef INTERPRETERLOOPNAME

#endif

#define INTERPRETERLOOPNAME ProcessUnprofiled

#include "Interpreterloop.inl"

#undef INTERPRETERLOOPNAME

#ifndef TEMP\_DISABLE\_ASMJS

#define INTERPRETERLOOPNAME ProcessAsmJs

#define INTERPRETER\_ASMJS

#include "InterpreterProcessOpCodeAsmJs.h"

#include "Interpreterloop.inl"

#undef INTERPRETER\_ASMJS

#undef INTERPRETERLOOPNAME

#endif

// For now, always collect profile data when debugging,

// otherwise the backend will be confused if there's no profile data.

#define INTERPRETERLOOPNAME ProcessWithDebugging

#define PROVIDE\_DEBUGGING

#if ENABLE\_PROFILE\_INFO

#define PROVIDE\_INTERPRETERPROFILE

#endif

#include "Interpreterloop.inl"

#if ENABLE\_PROFILE\_INFO

#undef PROVIDE\_INTERPRETERPROFILE

#endif

#undef PROVIDE\_DEBUGGING

#undef INTERPRETERLOOPNAME

Var InterpreterStackFrame::Process()

{

#if ENABLE\_PROFILE\_INFO

class AutoRestore

{

private:

InterpreterStackFrame \*const interpreterStackFrame;

const uint32 savedSwitchProfileModeOnLoopEndNumber;

const bool savedIsAutoProfiling;

const bool savedSwitchProfileMode;

public:

AutoRestore(InterpreterStackFrame \*const interpreterStackFrame)

: interpreterStackFrame(interpreterStackFrame),

savedIsAutoProfiling(interpreterStackFrame->isAutoProfiling),

savedSwitchProfileMode(interpreterStackFrame->switchProfileMode),

savedSwitchProfileModeOnLoopEndNumber(interpreterStackFrame->switchProfileModeOnLoopEndNumber)

{

}

~AutoRestore()

{

interpreterStackFrame->isAutoProfiling = savedIsAutoProfiling;

interpreterStackFrame->switchProfileMode = savedSwitchProfileMode;

interpreterStackFrame->switchProfileModeOnLoopEndNumber = savedSwitchProfileModeOnLoopEndNumber;

}

} autoRestore(this);

#endif

if ((m\_flags & Js::InterpreterStackFrameFlags\_FromBailOut) && !(m\_flags & InterpreterStackFrameFlags\_ProcessingBailOutFromEHCode))

{

if (this->ehBailoutData)

{

m\_flags |= Js::InterpreterStackFrameFlags\_ProcessingBailOutFromEHCode;

EHBailoutData \* topLevelEHBailoutData = this->ehBailoutData;

while (topLevelEHBailoutData->parent->nestingDepth != -1)

{

topLevelEHBailoutData->parent->child = topLevelEHBailoutData;

topLevelEHBailoutData = topLevelEHBailoutData->parent;

}

ProcessTryCatchBailout(topLevelEHBailoutData, this->ehBailoutData->nestingDepth);

m\_flags &= ~Js::InterpreterStackFrameFlags\_ProcessingBailOutFromEHCode;

this->ehBailoutData = nullptr;

}

}

#ifndef TEMP\_DISABLE\_ASMJS

FunctionBody \*const functionBody = GetFunctionBody();

if( functionBody->GetIsAsmjsMode() )

{

AsmJsFunctionInfo\* asmInfo = functionBody->GetAsmJsFunctionInfo();

if (asmInfo)

{

AlignMemoryForAsmJs();

Var returnVar = ProcessAsmJs();

#if DBG\_DUMP

if( PHASE\_TRACE( AsmjsFunctionEntryPhase, functionBody ) )

{

--AsmJsCallDepth;

if( AsmJsCallDepth )

{

Output::Print( L"%\*c}", AsmJsCallDepth, ' ' );

}

else

{

Output::Print( L"}" );

}

switch( asmInfo->GetReturnType().which() )

{

case AsmJsRetType::Void:

break;

case AsmJsRetType::Signed:

Output::Print( L" = %d", JavascriptMath::ToInt32( returnVar, scriptContext ) );

break;

case AsmJsRetType::Float:

case AsmJsRetType::Double:

Output::Print( L" = %.4f", JavascriptConversion::ToNumber( returnVar, scriptContext ) );

break;

default:

break;

}

Output::Print( L";\n" );

}

#endif

return returnVar;

}

else

{

Assert(functionBody->GetAsmJsModuleInfo());

return ProcessAsmJsModule();

}

}

#endif

#if ENABLE\_PROFILE\_INFO

switchProfileMode = false;

switchProfileModeOnLoopEndNumber = 0u - 1;

#endif

#if ENABLE\_PROFILE\_INFO

const ExecutionMode interpreterExecutionMode =

functionBody->GetInterpreterExecutionMode(!!(GetFlags() & InterpreterStackFrameFlags\_FromBailOut));

if(interpreterExecutionMode == ExecutionMode::ProfilingInterpreter)

{

isAutoProfiling = false;

return ProcessProfiled();

}

Assert(

interpreterExecutionMode == ExecutionMode::Interpreter ||

interpreterExecutionMode == ExecutionMode::AutoProfilingInterpreter);

isAutoProfiling = interpreterExecutionMode == ExecutionMode::AutoProfilingInterpreter;

Var result;

while(true)

{

Assert(!switchProfileMode);

result = ProcessUnprofiled();

Assert(!(switchProfileMode && result));

if(switchProfileMode)

{

switchProfileMode = false;

}

else

{

break;

}

Assert(isAutoProfiling);

#if DBG\_DUMP

if(PHASE\_TRACE(InterpreterAutoProfilePhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"InterpreterAutoProfile - Func %s - Started profiling\n", functionBody->GetDebugNumberSet(debugStringBuffer));

Output::Flush();

}

#endif

Assert(!switchProfileMode);

result = ProcessProfiled();

Assert(!(switchProfileMode && result));

if(switchProfileMode)

{

switchProfileMode = false;

}

else

{

break;

}

Assert(isAutoProfiling);

#if DBG\_DUMP

if(PHASE\_TRACE(InterpreterAutoProfilePhase, functionBody))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"InterpreterAutoProfile - Func %s - Stopped profiling\n", functionBody->GetDebugNumberSet(debugStringBuffer));

Output::Flush();

}

#endif

}

return result;

#else

return ProcessUnprofiled();

#endif

}

template <class T>

void InterpreterStackFrame::OP\_GetMethodProperty(unaligned T \*playout)

{

Var varInstance = GetReg(playout->Instance);

OP\_GetMethodProperty(varInstance, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetLocalMethodProperty(unaligned T \*playout)

{

OP\_GetMethodProperty(this->localClosure, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetMethodProperty(Var varInstance, unaligned T \*playout)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(varInstance);

#endif

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

RecyclableObject\* obj = NULL;

if (RecyclableObject::Is(varInstance))

{

obj = RecyclableObject::FromVar(varInstance);

if ((propertyId == PropertyIds::apply || propertyId == PropertyIds::call) && ScriptFunction::Is(obj))

{

// If the property being loaded is "apply"/"call", make an optimistic assumption that apply/call is not overridden and

// undefer the function right here if it was defer parsed before. This is required so that the load of "apply"/"call"

// happens from the same "type". Otherwise, we will have a polymorphic cache for load of "apply"/"call".

ScriptFunction \*fn = ScriptFunction::FromVar(obj);

if(fn->GetType()->GetEntryPoint() == JavascriptFunction::DeferredParsingThunk)

{

JavascriptFunction::DeferredParse(&fn);

}

}

}

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var aValue;

if (obj &&

CacheOperators::TryGetProperty<true, true, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &aValue, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, aValue);

return;

}

OP\_GetMethodProperty\_NoFastPath(varInstance, playout);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetMethodProperty\_NoFastPath(Var instance, unaligned T \*playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = JavascriptOperators::PatchGetMethod<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

instance,

propertyId

);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(instance, propertyId, value, true);

}

#endif

SetReg(playout->Value, value);

}

template <class T>

void InterpreterStackFrame::OP\_GetRootMethodProperty(unaligned T \*playout)

{

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

Js::Var instance = this->GetRootObject();

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

DynamicObject \*obj = DynamicObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var aValue;

if (CacheOperators::TryGetProperty<true, true, false, false, false, false, true, false, false>(

obj, true, obj, propertyId, &aValue, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, aValue);

return;

}

OP\_GetRootMethodProperty\_NoFastPath(playout);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetRootMethodProperty\_NoFastPath(unaligned T \*playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var rootInstance = this->GetRootObject();

Var value = JavascriptOperators::PatchGetRootMethod<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

DynamicObject::FromVar(rootInstance),

propertyId

);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(rootInstance, propertyId, value, true);

}

#endif

SetReg(playout->Value, value);

}

template <class T>

void InterpreterStackFrame::OP\_GetMethodPropertyScoped(unaligned T \*playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

Var varInstance = GetReg(playout->Instance);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

RecyclableObject\* obj = NULL;

if (RecyclableObject::Is(varInstance))

{

obj = RecyclableObject::FromVar(varInstance);

}

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var aValue;

if (obj &&

CacheOperators::TryGetProperty<true, true, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &aValue, GetScriptContext(), nullptr, &info))

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->Value, aValue);

return;

}

OP\_GetMethodPropertyScoped\_NoFastPath(playout);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetMethodPropertyScoped\_NoFastPath(unaligned T \*playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Js::Var instance = GetReg(playout->Instance);

Js::Var value = JavascriptOperators::PatchScopedGetMethod<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

instance,

propertyId

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(instance, propertyId, value, true);

}

#endif

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetMethodProperty(unaligned T \*playout)

{

ProfiledGetProperty<T, false, true, false>(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetLocalMethodProperty(unaligned T \*playout)

{

ProfiledGetProperty<T, false, true, false>(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetRootMethodProperty(unaligned T \*playout)

{

ProfiledGetProperty<T, true, true, false>(playout, GetRootObject());

}

RecyclableObject \*

InterpreterStackFrame::OP\_CallGetFunc(Var target)

{

return JavascriptOperators::GetCallableObjectOrThrow(target, GetScriptContext());

}

void InterpreterStackFrame::OP\_AsmStartCall( const unaligned OpLayoutStartCall \* playout )

{

OP\_StartCall( playout->ArgCount/sizeof(Var) );

m\_outParams[0] = scriptContext->GetLibrary()->GetUndefined();

}

void InterpreterStackFrame::OP\_StartCall(const unaligned OpLayoutStartCall \* playout)

{

OP\_StartCall(playout->ArgCount);

}

void InterpreterStackFrame::OP\_StartCall(uint outParamCount)

{

// Save the outParams for the current callsite on the outparam stack

PushOut(m\_outParams);

// Update outParams for the indicated callsite

m\_outParams = m\_outSp;

m\_outSp += outParamCount;

AssertMsg(m\_localSlots + this->m\_functionBody->GetLocalsCount() < m\_outSp &&

m\_outSp <= (m\_localSlots + this->m\_functionBody->GetLocalsCount() + this->m\_functionBody->GetOutParamsDepth()),

"out args Stack pointer not in range after Push");

}

#ifndef TEMP\_DISABLE\_ASMJS

#if \_M\_X64

void InterpreterStackFrame::OP\_CallAsmInternal(RecyclableObject \* function)

{

AsmJsFunctionInfo\* asmInfo = ((ScriptFunction\*)function)->GetFunctionBody()->GetAsmJsFunctionInfo();

uint argsSize = asmInfo->GetArgByteSize();

ScriptFunction\* scriptFunc = (ScriptFunction\*)function;

ScriptContext \* scriptContext = function->GetScriptContext();

PROBE\_STACK\_CALL(scriptContext, function, argsSize);

Js::FunctionEntryPointInfo\* entrypointInfo = (Js::FunctionEntryPointInfo\*)scriptFunc->GetEntryPointInfo();

switch (asmInfo->GetReturnType().which())

{

case AsmJsRetType::Void:

case AsmJsRetType::Signed:

m\_localIntSlots[0] = JavascriptFunction::CallAsmJsFunction<int>(function, entrypointInfo->address, asmInfo->GetArgCount(), m\_outParams);

break;

case AsmJsRetType::Double:

m\_localDoubleSlots[0] = JavascriptFunction::CallAsmJsFunction<double>(function, entrypointInfo->address, asmInfo->GetArgCount(), m\_outParams);

break;

case AsmJsRetType::Float:

m\_localFloatSlots[0] = JavascriptFunction::CallAsmJsFunction<float>(function, entrypointInfo->address, asmInfo->GetArgCount(), m\_outParams);

break;

case AsmJsRetType::Float32x4:

case AsmJsRetType::Int32x4:

case AsmJsRetType::Float64x2:

X86SIMDValue simdVal;

simdVal.m128\_value = JavascriptFunction::CallAsmJsFunction<\_\_m128>(function, entrypointInfo->address, asmInfo->GetArgCount(), m\_outParams);

m\_localSimdSlots[0] = X86SIMDValue::ToSIMDValue(simdVal);

break;

}

Assert((uint)((ArgSlot)asmInfo->GetArgCount() + 1) == (uint)(asmInfo->GetArgCount() + 1));

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

PopOut((ArgSlot)(asmInfo->GetArgByteSize() / sizeof(Var)) + 1);

}

else

{

PopOut((ArgSlot)asmInfo->GetArgCount() + 1);

}

Assert(function);

}

#elif \_M\_IX86

void InterpreterStackFrame::OP\_CallAsmInternal(RecyclableObject \* function)

{

enum {

Void = AsmJsRetType::Void,

Signed = AsmJsRetType::Signed,

Float = AsmJsRetType::Float,

Double = AsmJsRetType::Double,

Int32x4 = AsmJsRetType::Int32x4,

Float32x4 = AsmJsRetType::Float32x4,

Float64x2 = AsmJsRetType::Float64x2

};

AsmJsFunctionInfo\* asmInfo = ((ScriptFunction\*)function)->GetFunctionBody()->GetAsmJsFunctionInfo();

Assert((uint)((ArgSlot)asmInfo->GetArgCount() + 1) == (uint)(asmInfo->GetArgCount() + 1));

uint argsSize = asmInfo->GetArgByteSize();

uint alignedSize = ::Math::Align<int32>(argsSize, 8);

ScriptFunction\* scriptFunc = (ScriptFunction\*)function;

ScriptContext \* scriptContext = function->GetScriptContext();

PROBE\_STACK\_CALL(scriptContext, function, alignedSize);

Js::FunctionEntryPointInfo\* entrypointInfo = (Js::FunctionEntryPointInfo\*)scriptFunc->GetEntryPointInfo();

int retIntVal = NULL;

float retFloatVal = NULL;

double retDoubleVal = NULL;

AsmJsSIMDValue retSimdVal;

retSimdVal.Zero();

AsmJsRetType::Which retType = (AsmJsRetType::Which) GetRetType(scriptFunc);

void \*data = nullptr;

JavascriptMethod entryPoint = (JavascriptMethod)entrypointInfo->address;

void \*savedEsp = nullptr;

\_\_asm

{

// Save ESP

mov savedEsp, esp;

mov eax, alignedSize;

// Make sure we don't go beyond guard page

cmp eax, 0x1000;

jge alloca\_probe;

sub esp, eax;

jmp dbl\_align;

alloca\_probe :

// Use alloca to allocate more then a page size

// Alloca assumes eax, contains size, and adjust ESP while

// probing each page.

call \_alloca\_probe\_16;

dbl\_align :

and esp,-8

mov data, esp;

}

{

Var\* outParam = m\_outParams + 1;

void\* dest = (void\*)data;

memmove(dest, outParam, argsSize);

}

// call variable argument function provided in entryPoint

\_\_asm

{

#ifdef \_CONTROL\_FLOW\_GUARD

// verify that the call target is valid

mov ecx, entryPoint

call[\_\_guard\_check\_icall\_fptr]

; no need to restore ecx('call entryPoint' is a \_\_cdecl call)

#endif

push function;

call entryPoint;

mov ebx, retType;

cmp ebx, Void;

je VoidLabel;

cmp ebx, Signed;

je SignedLabel;

cmp ebx, Float;

je FloatLabel;

cmp ebx, Double;

je DoubleLabel;

// simd

movups retSimdVal, xmm0;

jmp end

VoidLabel:

SignedLabel:

mov retIntVal, eax;

jmp end;

FloatLabel:

movss retFloatVal, xmm0;

jmp end;

DoubleLabel:

movsd retDoubleVal, xmm0;

end:

// Restore ESP

mov esp, savedEsp;

}

switch (retType)

{

case AsmJsRetType::Int32x4:

case AsmJsRetType::Float32x4:

case AsmJsRetType::Float64x2:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

m\_localSimdSlots[0] = retSimdVal;

break;

}

Assert(UNREACHED);

case AsmJsRetType::Double:

m\_localDoubleSlots[0] = retDoubleVal;

break;

case AsmJsRetType::Float:

m\_localFloatSlots[0] = retFloatVal;

break;

case AsmJsRetType::Signed:

case AsmJsRetType::Void:

m\_localIntSlots[0] = retIntVal;

break;

default:

Assume(false);

}

PopOut((uint)((ArgSlot)argsSize/sizeof(Var)) + 1);

Assert(function);

}

#else

void InterpreterStackFrame::OP\_CallAsmInternal(RecyclableObject \* function)

{

\_\_debugbreak();

}

#endif

#endif

template <class T>

void InterpreterStackFrame::OP\_AsmCall(const unaligned T\* playout)

{

OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), CallFlags\_None);

AsmJsModuleInfo::EnsureHeapAttached(this->function);

}

template <class T>

void InterpreterStackFrame::OP\_CallCommon(const unaligned T \* playout, RecyclableObject \* function, unsigned flags, const Js::AuxArray<uint32> \*spreadIndices)

{

// Always save and restore implicit call flags when calling out

// REVIEW: Can we avoid it if we don't collect dynamic profile info?

ThreadContext \* threadContext = scriptContext->GetThreadContext();

Js::ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

#if DBG

if (scriptContext->IsInDebugMode())

{

JavascriptFunction::CheckValidDebugThunk(scriptContext, function);

}

#endif

if (playout->Return == Js::Constants::NoRegister)

{

flags |= CallFlags\_NotUsed;

Arguments args(CallInfo((CallFlags)flags, playout->ArgCount), m\_outParams);

AssertMsg(args.Info.Flags == flags, "Flags don't fit into the CallInfo field?");

if (spreadIndices != nullptr)

{

JavascriptFunction::CallSpreadFunction(function, function->GetEntryPoint(), args, spreadIndices);

}

else

{

JavascriptFunction::CallFunction<true>(function, function->GetEntryPoint(), args);

}

}

else

{

flags |= CallFlags\_Value;

Arguments args(CallInfo((CallFlags)flags, playout->ArgCount), m\_outParams);

AssertMsg(args.Info.Flags == flags, "Flags don't fit into the CallInfo field?");

if (spreadIndices != nullptr)

{

SetReg((RegSlot)playout->Return, JavascriptFunction::CallSpreadFunction(function, function->GetEntryPoint(), args, spreadIndices));

}

else

{

SetReg((RegSlot)playout->Return, JavascriptFunction::CallFunction<true>(function, function->GetEntryPoint(), args));

}

}

threadContext->SetImplicitCallFlags(savedImplicitCallFlags);

PopOut(playout->ArgCount);

}

template <class T>

void InterpreterStackFrame::OP\_CallCommonI(const unaligned T \* playout, RecyclableObject \* function, unsigned flags)

{

OP\_CallCommon(playout, function, flags); // CallCommon doesn't do anything with Member

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::OP\_ProfileCallCommon(const unaligned T \* playout, RecyclableObject \* function, unsigned flags, ProfileId profileId, InlineCacheIndex inlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices)

{

FunctionBody\* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

FunctionInfo\* functionInfo = function->GetTypeId() == TypeIds\_Function?

JavascriptFunction::FromVar(function)->GetFunctionInfo() : nullptr;

dynamicProfileInfo->RecordCallSiteInfo(functionBody, profileId, functionInfo, functionInfo ? static\_cast<JavascriptFunction\*>(function) : nullptr, playout->ArgCount, false, inlineCacheIndex);

OP\_CallCommon<T>(playout, function, flags, spreadIndices);

if (playout->Return != Js::Constants::NoRegister)

{

dynamicProfileInfo->RecordReturnTypeOnCallSiteInfo(functionBody, profileId, GetReg((RegSlot)playout->Return));

}

}

template <class T>

void InterpreterStackFrame::OP\_ProfileReturnTypeCallCommon(const unaligned T \* playout, RecyclableObject \* function, unsigned flags, ProfileId profileId, const Js::AuxArray<uint32> \*spreadIndices)

{

OP\_CallCommon<T>(playout, function, flags, spreadIndices);

FunctionBody\* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

if (playout->Return != Js::Constants::NoRegister)

{

dynamicProfileInfo->RecordReturnType(functionBody, profileId, GetReg((RegSlot)playout->Return));

}

}

#endif

template <class T>

void InterpreterStackFrame::OP\_CallPutCommon(const unaligned T \*playout, RecyclableObject \* function)

{

Arguments args(CallInfo(CallFlags\_None, playout->ArgCount), m\_outParams);

SetReg((RegSlot)playout->Return, function->InvokePut(args));

PopOut(playout->ArgCount);

}

template <class T>

void InterpreterStackFrame::OP\_CallPutCommonI(const unaligned T \*playout, RecyclableObject \* function)

{

OP\_CallPutCommon(playout, function);

}

template <class T>

void InterpreterStackFrame::OP\_GetRootProperty(unaligned T\* playout)

{

// Same fast path as in the backend.

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

Js::Var instance = this->GetRootObject();

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

DynamicObject \* obj = DynamicObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var value;

if(CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

obj, true, obj, propertyId, &value, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, value);

return;

}

OP\_GetRootProperty\_NoFastPath(playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetRootPropertyForTypeOf(unaligned T\* playout)

{

Var rootInstance = GetRootObject();

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = JavascriptOperators::PatchGetRootValueForTypeOf<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

DynamicObject::FromVar(rootInstance),

propertyId

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(rootInstance, propertyId, value, /\*successful:\*/true);

}

#endif

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetRootProperty\_NoFastPath(unaligned T\* playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var rootInstance = this->GetRootObject();

Var value = JavascriptOperators::PatchGetRootValue<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

DynamicObject::FromVar(rootInstance),

propertyId

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(rootInstance, propertyId, value, /\*successful:\*/true);

}

#endif

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::UpdateFldInfoFlagsForGetSetInlineCandidate(unaligned T\* playout, FldInfoFlags& fldInfoFlags, CacheType cacheType,

DynamicProfileInfo \* dynamicProfileInfo, uint inlineCacheIndex, RecyclableObject \* obj)

{

RecyclableObject \*callee = nullptr;

//TODO: Setter case once we stop sharing inline caches for these callsites.

if ((cacheType & (CacheType\_Getter | CacheType\_Setter)) && GetInlineCache(inlineCacheIndex)->GetGetterSetter(obj->GetType(), &callee))

{

const auto functionBody = this->m\_functionBody;

bool canInline = dynamicProfileInfo->RecordLdFldCallSiteInfo(functionBody, callee, false /\*callApplyTarget\*/);

if (canInline)

{

//updates this fldInfoFlags passed by reference.

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_InlineCandidate);

}

}

}

template <class T>

void InterpreterStackFrame::UpdateFldInfoFlagsForCallApplyInlineCandidate(unaligned T\* playout, FldInfoFlags& fldInfoFlags, CacheType cacheType,

DynamicProfileInfo \* dynamicProfileInfo, uint inlineCacheIndex, RecyclableObject \* obj)

{

RecyclableObject \*callee = nullptr;

if (!(fldInfoFlags & FldInfo\_Polymorphic) && GetInlineCache(inlineCacheIndex)->GetCallApplyTarget(obj, &callee))

{

const auto functionBody = this->m\_functionBody;

bool canInline = dynamicProfileInfo->RecordLdFldCallSiteInfo(functionBody, callee, true /\*callApplyTarget\*/);

if (canInline)

{

//updates this fldInfoFlags passed by reference.

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_InlineCandidate);

}

}

}

template <class T, bool Root, bool Method, bool CallApplyTarget>

void InterpreterStackFrame::ProfiledGetProperty(unaligned T\* playout, const Var instance)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = ProfilingHelpers::ProfiledLdFld<Root, Method, CallApplyTarget>(

instance,

propertyId,

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetFunctionBody(),

instance);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, /\*successful:\*/true);

}

#endif

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetRootProperty(unaligned T\* playout)

{

ProfiledGetProperty<T, true, false, false>(playout, GetRootObject());

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetRootPropertyForTypeOf(unaligned T\* playout)

{

Var rootInstance = GetRootObject();

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = ProfilingHelpers::ProfiledLdFldForTypeOf<true, false, false>(

rootInstance,

propertyId,

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetFunctionBody());

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(rootInstance, propertyId, value, /\*successful:\*/true);

}

#endif

}

#endif

template <class T>

void InterpreterStackFrame::OP\_GetPropertyForTypeOf(unaligned T\* playout)

{

Var instance = GetReg(playout->Instance);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = JavascriptOperators::PatchGetValueForTypeOf<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

instance,

propertyId

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetRootValue throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, /\*successful:\*/true);

}

#endif

}

template <class T>

void InterpreterStackFrame::OP\_GetProperty(unaligned T\* playout)

{

// Same fast path as in the backend.

Var instance = GetReg(playout->Instance);

OP\_GetProperty(instance, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetLocalProperty(unaligned T\* playout)

{

// Same fast path as in the backend.

Var instance = this->localClosure;

OP\_GetProperty(instance, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetProperty(Var instance, unaligned T\* playout)

{

InlineCache \*inlineCache = GetInlineCache(playout->inlineCacheIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (RecyclableObject::Is(instance))

{

RecyclableObject\* obj = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var value;

if (CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &value, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, value);

return;

}

}

OP\_GetProperty\_NoFastPath(instance, playout);

}

template <class T>

void InterpreterStackFrame::OP\_GetSuperProperty(unaligned T\* playout)

{

// Same fast path as in the backend.

Var instance = GetReg(playout->Instance);

Var thisInstance = GetReg(playout->Value2);

InlineCache \*inlineCache = GetInlineCache(playout->PropertyIdIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->PropertyIdIndex);

if (RecyclableObject::Is(instance) && RecyclableObject::Is(thisInstance))

{

RecyclableObject\* superObj = RecyclableObject::FromVar(instance);

RecyclableObject\* thisObj = RecyclableObject::FromVar(thisInstance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->PropertyIdIndex, true);

Var value;

if (CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

thisObj, false, superObj, propertyId, &value, GetScriptContext(), nullptr, &info))

{

SetReg(playout->Value, value);

return;

}

}

SetReg(

playout->Value,

JavascriptOperators::PatchGetValueWithThisPtr<false>(

GetFunctionBody(),

GetInlineCache(playout->PropertyIdIndex),

playout->PropertyIdIndex,

GetReg(playout->Instance),

GetPropertyIdFromCacheId(playout->PropertyIdIndex),

GetReg(playout->Value2)));

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetProperty\_NoFastPath(Var instance, unaligned T\* playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = JavascriptOperators::PatchGetValue<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

instance,

propertyId

);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, true);

}

#endif

SetReg(playout->Value, value);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetProperty(unaligned T\* playout)

{

ProfiledGetProperty<T, false, false, false>(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetLocalProperty(unaligned T\* playout)

{

ProfiledGetProperty<T, false, false, false>(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetSuperProperty(unaligned T\* playout)

{

SetReg(

playout->Value,

ProfilingHelpers::ProfiledLdFld<false, false, false>(

GetReg(playout->Instance),

GetPropertyIdFromCacheId(playout->PropertyIdIndex),

GetInlineCache(playout->PropertyIdIndex),

playout->PropertyIdIndex,

GetFunctionBody(),

GetReg(playout->Value2)));

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetPropertyForTypeOf(unaligned T\* playout)

{

Var instance = GetReg(playout->Instance);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

Var value = ProfilingHelpers::ProfiledLdFldForTypeOf<false, false, false>(

instance,

propertyId,

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetFunctionBody()

);

SetReg(playout->Value, value);

#ifdef TELEMETRY\_INTERPRETER

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, true);

}

#endif

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetPropertyCallApplyTarget(unaligned T\* playout)

{

ProfiledGetProperty<T, false, false, true>(playout, GetReg(playout->Instance));

}

#endif

template <typename T>

void InterpreterStackFrame::OP\_GetPropertyScoped(const unaligned OpLayoutT\_ElementP<T>\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Get the property, using a scope stack rather than an individual instance.

// Use the cache

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

FrameDisplay \*pScope = this->GetEnvForEvalCode();

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

ScriptContext\* scriptContext = GetScriptContext();

int length = pScope->GetLength();

if ( 1 == length )

{

DynamicObject \*obj = (DynamicObject\*)pScope->GetItem(0);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var value;

if (CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &value, scriptContext, nullptr, &info))

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->Value, value);

return;

}

}

OP\_GetPropertyScoped\_NoFastPath(playout);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

template <typename T>

void InterpreterStackFrame::OP\_GetPropertyForTypeOfScoped(const unaligned OpLayoutT\_ElementP<T>\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Get the property, using a scope stack rather than an individual instance.

// Use the cache

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

FrameDisplay \*pScope = this->GetEnvForEvalCode();

InlineCache \*inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

ScriptContext\* scriptContext = GetScriptContext();

int length = pScope->GetLength();

if (1 == length)

{

DynamicObject \*obj = (DynamicObject\*)pScope->GetItem(0);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

Var value;

if (CacheOperators::TryGetProperty<true, false, false, false, false, false, true, false, false>(

obj, false, obj, propertyId, &value, scriptContext, nullptr, &info))

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->Value, value);

return;

}

}

SetReg(

playout->Value,

JavascriptOperators::PatchGetPropertyForTypeOfScoped<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetEnvForEvalCode(),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetReg(Js::FunctionBody::RootObjectRegSlot)));

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

template <typename T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_GetPropertyScoped\_NoFastPath(const unaligned OpLayoutT\_ElementP<T>\* playout)

{

// Implicit root object as default instance

Var defaultInstance = GetReg(Js::FunctionBody::RootObjectRegSlot);

// PatchGetPropertyScoped doesn't update type and slotIndex if the scope is not an array of length 1.

SetReg(

playout->Value,

JavascriptOperators::PatchGetPropertyScoped<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetEnvForEvalCode(),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

defaultInstance));

}

template <class T>

void InterpreterStackFrame::OP\_SetPropertyScoped(unaligned T\* playout, PropertyOperationFlags flags)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Set the property, using a scope stack rather than an individual instance.

// Use the cache

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

FrameDisplay \*pScope = this->GetEnvForEvalCode();

InlineCache \*inlineCache = GetInlineCache(playout->inlineCacheIndex);

ScriptContext\* scriptContext = GetScriptContext();

Var value = GetReg(playout->Value);

DynamicObject \*obj;

int length = pScope->GetLength();

if ( 1 == length )

{

obj = (DynamicObject\*)pScope->GetItem(0);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

if (CacheOperators::TrySetProperty<true, false, false, false, false, true, false, false>(

obj, false, propertyId, value, scriptContext, flags, nullptr, &info))

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return;

}

}

OP\_SetPropertyScoped\_NoFastPath(playout, flags);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::OP\_SetPropertyScoped\_NoFastPath(unaligned T\* playout, PropertyOperationFlags flags)

{

// Implicit root object as default instance

Var defaultInstance = GetReg(Js::FunctionBody::RootObjectRegSlot);

// PatchSetPropertyScoped doesn't update type and slotIndex if the scope is not an array of length 1.

JavascriptOperators::PatchSetPropertyScoped<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetEnvForEvalCode(),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetReg(playout->Value),

defaultInstance,

flags);

}

template <class T>

void InterpreterStackFrame::OP\_SetPropertyScopedStrict(unaligned T\* playout)

{

OP\_SetPropertyScoped(playout, PropertyOperation\_StrictMode);

}

template <class T>

void InterpreterStackFrame::OP\_ConsoleSetPropertyScoped(unaligned T\* playout)

{

OP\_SetPropertyScoped(playout, PropertyOperation\_AllowUndeclInConsoleScope);

}

template <class T>

\_\_inline bool InterpreterStackFrame::TrySetPropertyLocalFastPath(unaligned T\* playout, PropertyId pid, Var instance, InlineCache\*& inlineCache, PropertyOperationFlags flags)

{

Assert(!TaggedNumber::Is(instance));

RecyclableObject\* obj = RecyclableObject::FromVar(instance);

inlineCache = this->GetInlineCache(playout->inlineCacheIndex);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, GetFunctionBody(), inlineCache, playout->inlineCacheIndex, true);

return

CacheOperators::TrySetProperty<true, false, false, false, false, true, false, false>(

obj,

!!(flags & PropertyOperation\_Root),

pid,

GetReg(playout->Value),

GetScriptContext(),

flags,

nullptr,

&info);

}

template <class T>

\_\_inline void InterpreterStackFrame::DoSetProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

// Same fast path as in the backend.

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

InlineCache \*inlineCache;

if (!TaggedNumber::Is(instance)

&& TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache, flags))

{

if(GetJavascriptFunction()->GetConstructorCache()->NeedsUpdateAfterCtor())

{

// This function has only 'this' statements and is being used as a constructor. When the constructor exits, the

// function object's constructor cache will be updated with the type produced by the constructor. From that

// point on, when the same function object is used as a constructor, the a new object with the final type will

// be created. Whatever is stored in the inline cache currently will cause cache misses after the constructor

// cache update. So, just clear it now so that the caches won't be flagged as polymorphic.

inlineCache->Clear();

}

return;

}

DoSetProperty\_NoFastPath(playout, instance, flags);

}

template <class T>

\_\_inline void InterpreterStackFrame::DoSetSuperProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

DoSetSuperProperty\_NoFastPath(playout, instance, flags);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::DoSetProperty\_NoFastPath(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

InlineCache \*const inlineCache = GetInlineCache(playout->inlineCacheIndex);

const auto PatchPutRootValue = &JavascriptOperators::PatchPutRootValueNoLocalFastPath<false, InlineCache>;

const auto PatchPutValue = &JavascriptOperators::PatchPutValueNoLocalFastPath<false, InlineCache>;

const auto PatchPut = flags & PropertyOperation\_Root ? PatchPutRootValue : PatchPutValue;

PatchPut(

GetFunctionBody(),

inlineCache,

playout->inlineCacheIndex,

instance,

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetReg(playout->Value),

flags);

if(!TaggedNumber::Is(instance) && GetJavascriptFunction()->GetConstructorCache()->NeedsUpdateAfterCtor())

{

// This function has only 'this' statements and is being used as a constructor. When the constructor exits, the

// function object's constructor cache will be updated with the type produced by the constructor. From that

// point on, when the same function object is used as a constructor, the a new object with the final type will

// be created. Whatever is stored in the inline cache currently will cause cache misses after the constructor

// cache update. So, just clear it now so that the caches won't be flagged as polymorphic.

inlineCache->Clear();

}

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::DoSetSuperProperty\_NoFastPath(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

InlineCache \*const inlineCache = GetInlineCache(playout->PropertyIdIndex);

JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<false, InlineCache>(

GetFunctionBody(),

inlineCache,

playout->PropertyIdIndex,

instance,

GetPropertyIdFromCacheId(playout->PropertyIdIndex),

GetReg(playout->Value),

GetReg(playout->Value2),

flags);

if (!TaggedNumber::Is(instance) && GetJavascriptFunction()->GetConstructorCache()->NeedsUpdateAfterCtor())

{

// This function has only 'this' statements and is being used as a constructor. When the constructor exits, the

// function object's constructor cache will be updated with the type produced by the constructor. From that

// point on, when the same function object is used as a constructor, the a new object with the final type will

// be created. Whatever is stored in the inline cache currently will cause cache misses after the constructor

// cache update. So, just clear it now so that the caches won't be flagged as polymorphic.

inlineCache->Clear();

}

}

#if ENABLE\_PROFILE\_INFO

template <class T, bool Root>

void InterpreterStackFrame::ProfiledSetProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags)

{

Assert(!Root || flags & PropertyOperation\_Root);

ProfilingHelpers::ProfiledStFld<Root>(

instance,

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetReg(playout->Value),

flags,

GetJavascriptFunction(),

instance);

}

template <class T, bool Root>

void InterpreterStackFrame::ProfiledSetSuperProperty(unaligned T\* playout, Var instance, Var thisInstance, PropertyOperationFlags flags)

{

Assert(!Root || flags & PropertyOperation\_Root);

ProfilingHelpers::ProfiledStFld<Root>(

instance,

GetPropertyIdFromCacheId(playout->PropertyIdIndex),

GetInlineCache(playout->PropertyIdIndex),

playout->PropertyIdIndex,

GetReg(playout->Value),

flags,

GetJavascriptFunction(),

thisInstance);

}

#endif

template <class T>

void InterpreterStackFrame::OP\_SetProperty(unaligned T\* playout)

{

DoSetProperty(playout, GetReg(playout->Instance), PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_SetLocalProperty(unaligned T\* playout)

{

DoSetProperty(playout, this->localClosure, PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_SetSuperProperty(unaligned T\* playout)

{

DoSetSuperProperty(playout, GetReg(playout->Instance), PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetProperty(unaligned T\* playout)

{

ProfiledSetProperty<T, false>(playout, GetReg(playout->Instance), PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetLocalProperty(unaligned T\* playout)

{

ProfiledSetProperty<T, false>(playout, this->localClosure, PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetSuperProperty(unaligned T\* playout)

{

ProfiledSetSuperProperty<T, false>(playout, GetReg(playout->Instance), GetReg(playout->Value2), PropertyOperation\_None);

}

template <class T>

void InterpreterStackFrame::OP\_SetRootProperty(unaligned T\* playout)

{

DoSetProperty(playout, this->GetRootObject(), PropertyOperation\_Root);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetRootProperty(unaligned T\* playout)

{

ProfiledSetProperty<T, true>(playout, this->GetRootObject(), PropertyOperation\_Root);

}

template <class T>

void InterpreterStackFrame::OP\_SetPropertyStrict(unaligned T\* playout)

{

DoSetProperty(playout, GetReg(playout->Instance), PropertyOperation\_StrictMode);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetPropertyStrict(unaligned T\* playout)

{

ProfiledSetProperty<T, false>(playout, GetReg(playout->Instance), PropertyOperation\_StrictMode);

}

template <class T>

void InterpreterStackFrame::OP\_SetRootPropertyStrict(unaligned T\* playout)

{

DoSetProperty(playout, this->GetRootObject(), PropertyOperation\_StrictModeRoot);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledSetRootPropertyStrict(unaligned T\* playout)

{

ProfiledSetProperty<T, true>(playout, this->GetRootObject(), PropertyOperation\_StrictModeRoot);

}

#if ENABLE\_PROFILE\_INFO

template <bool doProfile>

Var InterpreterStackFrame::ProfiledDivide(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId)

{

Var result = JavascriptMath::Divide(aLeft, aRight,scriptContext);

if (doProfile)

{

Js::FunctionBody\* body = this->m\_functionBody;

body->GetDynamicProfileInfo()->RecordDivideResultType(body, profileId, result);

}

return result;

}

template <bool doProfile>

Var InterpreterStackFrame::ProfileModulus(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId)

{

// If both arguments are TaggedInt, then try to do integer division

// This case is not handled by the lowerer.

if (doProfile)

{

Js::FunctionBody\* body = this->function->GetFunctionBody();

if(TaggedInt::IsPair(aLeft, aRight))

{

int nLeft = TaggedInt::ToInt32(aLeft);

int nRight = TaggedInt::ToInt32(aRight);

// nLeft is positive and nRight is +2^i

// Fast path for Power of 2 divisor

if (nLeft > 0 && ::Math::IsPow2(nRight))

{

body->GetDynamicProfileInfo()->RecordModulusOpType(body, profileId, /\*isModByPowerOf2\*/ true);

return TaggedInt::ToVarUnchecked(nLeft & (nRight - 1));

}

}

body->GetDynamicProfileInfo()->RecordModulusOpType(body, profileId, /\*isModByPowerOf2\*/ false);

}

return JavascriptMath::Modulus(aLeft, aRight,scriptContext);

}

template <bool doProfile>

Var InterpreterStackFrame::ProfiledSwitch(Var exp, ProfileId profileId)

{

if (doProfile)

{

Js::FunctionBody\* body = this->m\_functionBody;

body->GetDynamicProfileInfo()->RecordSwitchType(body, profileId, exp);

}

return exp;

}

#else

template <bool doProfile>

Var InterpreterStackFrame::ProfiledDivide(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId)

{

Assert(!doProfile);

return JavascriptMath::Divide(aLeft, aRight, scriptContext);

}

template <bool doProfile>

Var InterpreterStackFrame::ProfileModulus(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId)

{

Assert(!doProfile);

return JavascriptMath::Modulus(aLeft, aRight, scriptContext);

}

template <bool doProfile>

Var InterpreterStackFrame::ProfiledSwitch(Var exp, ProfileId profileId)

{

Assert(!doProfile);

return exp;

}

#endif

template <class T>

void InterpreterStackFrame::DoInitProperty(unaligned T\* playout, Var instance)

{

// Same fast path as in the backend.

InlineCache \*inlineCache = nullptr;

Assert(!TaggedNumber::Is(instance));

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache))

{

return;

}

DoInitProperty\_NoFastPath(playout, instance);

}

template <class T>

\_\_declspec(noinline) void InterpreterStackFrame::DoInitProperty\_NoFastPath(unaligned T\* playout, Var instance)

{

JavascriptOperators::PatchInitValue<false>(

GetFunctionBody(),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

RecyclableObject::FromVar(instance),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMember(const unaligned T \* playout)

{

uint inlineCacheIndex = playout->inlineCacheIndex;

InlineCache \* inlineCache = this->GetInlineCache(inlineCacheIndex);

Var instance = GetReg(playout->Instance);

PropertyOperationFlags flags = PropertyOperation\_None;

Assert(!TaggedNumber::Is(instance));

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (!TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache, flags))

{

JavascriptOperators::OP\_InitClassMember(instance, propertyId, GetReg(playout->Value));

}

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberGet(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberGet(

GetReg(playout->Instance),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberSet(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberSet(

GetReg(playout->Instance),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberSetComputedName(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberSetComputedName(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext());

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberGetComputedName(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberGetComputedName(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext());

}

template <class T>

void InterpreterStackFrame::OP\_InitClassMemberComputedName(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitClassMemberComputedName(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext());

}

template <class T>

void InterpreterStackFrame::DoInitLetFld(const unaligned T \* playout, Var instance, PropertyOperationFlags flags)

{

uint inlineCacheIndex = playout->inlineCacheIndex;

InlineCache \* inlineCache = this->GetInlineCache(inlineCacheIndex);

Assert(!TaggedNumber::Is(instance));

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (!TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache, flags))

{

JavascriptOperators::OP\_InitLetProperty(instance, propertyId, GetReg(playout->Value));

}

}

template <class T>

void InterpreterStackFrame::DoInitConstFld(const unaligned T \* playout, Var instance, PropertyOperationFlags flags)

{

uint inlineCacheIndex = playout->inlineCacheIndex;

InlineCache \* inlineCache = this->GetInlineCache(inlineCacheIndex);

Assert(!TaggedNumber::Is(instance));

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

if (!TrySetPropertyLocalFastPath(playout, propertyId, instance, inlineCache, flags))

{

JavascriptOperators::OP\_InitConstProperty(instance, propertyId, GetReg(playout->Value));

}

}

template <class T>

void InterpreterStackFrame::OP\_InitProperty(unaligned T\* playout)

{

DoInitProperty(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_InitLocalProperty(unaligned T\* playout)

{

DoInitProperty(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_InitInnerFld(const unaligned T\* playout)

{

DoInitProperty(playout, InnerScopeFromIndex(playout->scopeIndex));

}

template <class T>

void InterpreterStackFrame::OP\_InitLetFld(const unaligned T \* playout)

{

DoInitLetFld(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_InitInnerLetFld(const unaligned T \* playout)

{

DoInitLetFld(playout, InnerScopeFromIndex(playout->scopeIndex));

}

template <class T>

void InterpreterStackFrame::OP\_InitLocalLetFld(const unaligned T \* playout)

{

DoInitLetFld(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_InitConstFld(const unaligned T \* playout)

{

DoInitConstFld(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_InitRootProperty(unaligned T\* playout)

{

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

DoInitProperty(playout, this->GetRootObject());

}

template <class T>

void InterpreterStackFrame::OP\_InitRootLetFld(const unaligned T \* playout)

{

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

DoInitLetFld(playout, this->GetRootObject(), PropertyOperation\_Root);

}

template <class T>

void InterpreterStackFrame::OP\_InitRootConstFld(const unaligned T \* playout)

{

Assert(playout->inlineCacheIndex >= this->m\_functionBody->GetRootObjectLoadInlineCacheStart());

DoInitConstFld(playout, this->GetRootObject(), PropertyOperation\_Root);

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclLetProperty(unaligned T\* playout)

{

Var instance = InnerScopeFromIndex(playout->scopeIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

JavascriptOperators::OP\_InitLetProperty(instance, propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclLocalLetProperty(unaligned T\* playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

JavascriptOperators::OP\_InitLetProperty(this->localClosure, propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

void InterpreterStackFrame::OP\_InitUndeclRootLetProperty(uint propertyIdIndex)

{

Var instance = this->GetRootObject();

PropertyId propertyId = this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex);

JavascriptOperators::OP\_InitUndeclRootLetProperty(instance, propertyId);

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclConstProperty(unaligned T\* playout)

{

Var instance = InnerScopeFromIndex(playout->scopeIndex);

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

JavascriptOperators::OP\_InitConstProperty(instance, propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclLocalConstProperty(unaligned T\* playout)

{

PropertyId propertyId = GetPropertyIdFromCacheId(playout->inlineCacheIndex);

JavascriptOperators::OP\_InitConstProperty(this->localClosure, propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

void InterpreterStackFrame::OP\_InitUndeclRootConstProperty(uint propertyIdIndex)

{

Var instance = this->GetRootObject();

PropertyId propertyId = this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex);

JavascriptOperators::OP\_InitUndeclRootConstProperty(instance, propertyId);

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclConsoleLetProperty(unaligned T\* playout)

{

FrameDisplay\* pScope = (FrameDisplay\*)this->LdEnv();

AssertMsg(ConsoleScopeActivationObject::Is((DynamicObject\*)pScope->GetItem(pScope->GetLength() - 1)), "How come we got this opcode without ConsoleScopeActivationObject?");

PropertyId propertyId = m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex);

JavascriptOperators::OP\_InitLetProperty(pScope->GetItem(0), propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

template <class T>

void InterpreterStackFrame::OP\_InitUndeclConsoleConstProperty(unaligned T\* playout)

{

FrameDisplay\* pScope = (FrameDisplay\*)this->LdEnv();

AssertMsg(ConsoleScopeActivationObject::Is((DynamicObject\*)pScope->GetItem(pScope->GetLength() - 1)), "How come we got this opcode without ConsoleScopeActivationObject?");

PropertyId propertyId = m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex);

JavascriptOperators::OP\_InitConstProperty(pScope->GetItem(0), propertyId, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::ProfiledInitProperty(unaligned T\* playout, Var instance)

{

ProfilingHelpers::ProfiledInitFld(

RecyclableObject::FromVar(instance),

GetPropertyIdFromCacheId(playout->inlineCacheIndex),

GetInlineCache(playout->inlineCacheIndex),

playout->inlineCacheIndex,

GetReg(playout->Value),

GetFunctionBody());

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledInitProperty(unaligned T\* playout)

{

ProfiledInitProperty(playout, GetReg(playout->Instance));

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledInitLocalProperty(unaligned T\* playout)

{

ProfiledInitProperty(playout, this->localClosure);

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledInitRootProperty(unaligned T\* playout)

{

ProfiledInitProperty(playout, this->GetRootObject());

}

template <class T>

void InterpreterStackFrame::OP\_ProfiledGetElementI(const unaligned OpLayoutDynamicProfile<T>\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

SetReg(

playout->Value,

ProfilingHelpers::ProfiledLdElem(

GetReg(playout->Instance),

GetReg(playout->Element),

m\_functionBody,

playout->profileId));

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#endif

template <typename T>

void InterpreterStackFrame::OP\_GetElementI(const unaligned T\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Same fast path as in the backend.

Var instance = GetReg(playout->Instance);

// Only enable fast path if the javascript array is not cross site

Var element;

#if ENABLE\_PROFILE\_INFO

if (!TaggedNumber::Is(instance) && VirtualTableInfo<JavascriptArray>::HasVirtualTable(instance))

{

element =

ProfilingHelpers::ProfiledLdElem\_FastPath(

JavascriptArray::FromVar(instance),

GetReg(playout->Element),

GetScriptContext());

}

else

#endif

{

element = JavascriptOperators::OP\_GetElementI(instance, GetReg(playout->Element), GetScriptContext());

}

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->Value, element);

}

template <typename T>

void InterpreterStackFrame::OP\_SetElementI(const unaligned T\* playout, PropertyOperationFlags flags)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

// Same fast path as in the backend.

Var instance = GetReg(playout->Instance);

const Var varIndex = GetReg(playout->Element);

const Var value = GetReg(playout->Value);

#if ENABLE\_PROFILE\_INFO

// Only enable fast path if the javascript array is not cross site

if (!TaggedNumber::Is(instance) &&

VirtualTableInfo<JavascriptArray>::HasVirtualTable(instance) &&

!JavascriptOperators::SetElementMayHaveImplicitCalls(GetScriptContext()))

{

ProfilingHelpers::ProfiledStElem\_FastPath(

JavascriptArray::FromVar(instance),

varIndex,

value,

GetScriptContext(),

flags);

}

else

#endif

{

JavascriptOperators::OP\_SetElementI(instance, varIndex, value, GetScriptContext(), flags);

}

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#if ENABLE\_PROFILE\_INFO

template <typename T>

void InterpreterStackFrame::OP\_ProfiledSetElementI(

const unaligned OpLayoutDynamicProfile<T>\* playout,

PropertyOperationFlags flags)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

ProfilingHelpers::ProfiledStElem(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody,

playout->profileId,

flags);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#endif

template <typename T>

void InterpreterStackFrame::OP\_SetElementIStrict(const unaligned T\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

OP\_SetElementI(playout, PropertyOperation\_StrictMode);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#if ENABLE\_PROFILE\_INFO

template <typename T>

void InterpreterStackFrame::OP\_ProfiledSetElementIStrict(const unaligned OpLayoutDynamicProfile<T>\* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

OP\_ProfiledSetElementI(playout, PropertyOperation\_StrictMode);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

}

#endif

template <class T>

void InterpreterStackFrame::OP\_LdArrayHeadSegment(const unaligned T\* playout)

{

JavascriptArray\* array = JavascriptArray::FromAnyArray(GetReg(playout->R1));

// The array is create by the built-in on the same script context

Assert(array->GetScriptContext() == GetScriptContext());

SetNonVarReg(playout->R0, array->GetHead());

}

template <class T>

void InterpreterStackFrame::OP\_SetArraySegmentItem\_CI4(const unaligned T\* playout)

{

SparseArraySegment<Var> \* segment = (SparseArraySegment<Var> \*)GetNonVarReg(playout->Instance);

uint32 index = playout->Element;

Var value = GetReg(playout->Value);

Assert(segment->left == 0);

Assert(index < segment->length);

segment->elements[index] = value;

}

template <class T>

void InterpreterStackFrame::OP\_NewScArray(const unaligned T \* playout)

{

JavascriptArray \*arr;

arr = scriptContext->GetLibrary()->CreateArrayLiteral(playout->C1);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#if ENABLE\_PROFILE\_INFO

template <bool Profiled, class T>

void InterpreterStackFrame::ProfiledNewScArray(const unaligned OpLayoutDynamicProfile<T> \* playout)

{

if(!Profiled && !isAutoProfiling)

{

OP\_NewScArray(playout);

return;

}

SetReg(

playout->R0,

ProfilingHelpers::ProfiledNewScArray(

playout->C1,

m\_functionBody,

playout->profileId));

}

#else

template <bool Profiled, class T>

void InterpreterStackFrame::ProfiledNewScArray(const unaligned OpLayoutDynamicProfile<T> \* playout)

{

Assert(!Profiled);

OP\_NewScArray(playout);

}

#endif

void InterpreterStackFrame::OP\_NewScIntArray(const unaligned OpLayoutAuxiliary \* playout)

{

#if ENABLE\_PROFILE\_INFO

if(isAutoProfiling)

{

OP\_ProfiledNewScIntArray(static\_cast<const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \*>(playout));

return;

}

#endif

const Js::AuxArray<int32> \*ints = Js::ByteCodeReader::ReadAuxArray<int32>(playout->Offset, this->GetFunctionBody());

JavascriptNativeIntArray \*arr = scriptContext->GetLibrary()->CreateNativeIntArrayLiteral(ints->count);

SparseArraySegment<int32> \* segment = (SparseArraySegment<int32>\*)arr->GetHead();

JavascriptOperators::AddIntsToArraySegment(segment, ints);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#if ENABLE\_PROFILE\_INFO

void InterpreterStackFrame::OP\_ProfiledNewScIntArray(const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \* playout)

{

const Js::AuxArray<int32> \*ints = Js::ByteCodeReader::ReadAuxArray<int32>(playout->Offset, this->GetFunctionBody());

Js::ProfileId profileId = playout->profileId;

FunctionBody \*functionBody = this->m\_functionBody;

ArrayCallSiteInfo \*arrayInfo = functionBody->GetDynamicProfileInfo()->GetArrayCallSiteInfo(functionBody, profileId);

Assert(arrayInfo);

JavascriptArray \*arr;

JavascriptLibrary \*lib = scriptContext->GetLibrary();

if (arrayInfo && arrayInfo->IsNativeIntArray())

{

#if ENABLE\_COPYONACCESS\_ARRAY

if (JavascriptLibrary::IsCopyOnAccessArrayCallSite(lib, arrayInfo, ints->count))

{

Assert(lib->cacheForCopyOnAccessArraySegments);

arr = scriptContext->GetLibrary()->CreateCopyOnAccessNativeIntArrayLiteral(arrayInfo, functionBody, ints);

}

else

#endif

{

arr = scriptContext->GetLibrary()->CreateNativeIntArrayLiteral(ints->count);

SparseArraySegment<int32> \*segment = (SparseArraySegment<int32>\*)arr->GetHead();

JavascriptOperators::AddIntsToArraySegment(segment, ints);

}

JavascriptNativeIntArray \*intArray = reinterpret\_cast<JavascriptNativeIntArray\*>(arr);

Recycler \*recycler = scriptContext->GetRecycler();

intArray->SetArrayCallSite(profileId, recycler->CreateWeakReferenceHandle(functionBody));

}

else if (arrayInfo && arrayInfo->IsNativeFloatArray())

{

arr = scriptContext->GetLibrary()->CreateNativeFloatArrayLiteral(ints->count);

SparseArraySegment<double> \* segment = (SparseArraySegment<double>\*)arr->GetHead();

for (uint i = 0; i < ints->count; i++)

{

segment->elements[i] = (double)ints->elements[i];

}

JavascriptNativeFloatArray \*floatArray = reinterpret\_cast<JavascriptNativeFloatArray\*>(arr);

Recycler \*recycler = scriptContext->GetRecycler();

floatArray->SetArrayCallSite(profileId, recycler->CreateWeakReferenceHandle(functionBody));

}

else

{

arr = scriptContext->GetLibrary()->CreateArrayLiteral(ints->count);

SparseArraySegment<Var> \* segment = (SparseArraySegment<Var>\*)arr->GetHead();

for (uint i = 0; i < ints->count; i++)

{

segment->elements[i] = JavascriptNumber::ToVar(ints->elements[i], scriptContext);

}

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#endif

void InterpreterStackFrame::OP\_NewScFltArray(const unaligned OpLayoutAuxiliary \* playout )

{

#if ENABLE\_PROFILE\_INFO

if(isAutoProfiling)

{

OP\_ProfiledNewScFltArray(static\_cast<const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \*>(playout));

return;

}

#endif

const Js::AuxArray<double> \*doubles = Js::ByteCodeReader::ReadAuxArray<double>(playout->Offset, this->GetFunctionBody());

JavascriptNativeFloatArray \*arr = scriptContext->GetLibrary()->CreateNativeFloatArrayLiteral(doubles->count);

SparseArraySegment<double> \* segment = (SparseArraySegment<double>\*)arr->GetHead();

JavascriptOperators::AddFloatsToArraySegment(segment, doubles);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#if ENABLE\_PROFILE\_INFO

void InterpreterStackFrame::OP\_ProfiledNewScFltArray(const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \* playout)

{

const Js::AuxArray<double> \*doubles = Js::ByteCodeReader::ReadAuxArray<double>(playout->Offset, this->GetFunctionBody());

Js::ProfileId profileId = playout->profileId;

FunctionBody \*functionBody = this->m\_functionBody;

ArrayCallSiteInfo \*arrayInfo = functionBody->GetDynamicProfileInfo()->GetArrayCallSiteInfo(functionBody, profileId);

Assert(arrayInfo);

JavascriptArray \*arr;

if (arrayInfo && arrayInfo->IsNativeFloatArray())

{

arrayInfo->SetIsNotNativeIntArray();

arr = scriptContext->GetLibrary()->CreateNativeFloatArrayLiteral(doubles->count);

SparseArraySegment<double> \* segment = (SparseArraySegment<double>\*)arr->GetHead();

JavascriptOperators::AddFloatsToArraySegment(segment, doubles);

JavascriptNativeFloatArray \*floatArray = reinterpret\_cast<JavascriptNativeFloatArray\*>(arr);

Recycler \*recycler = scriptContext->GetRecycler();

floatArray->SetArrayCallSite(profileId, recycler->CreateWeakReferenceHandle(functionBody));

}

else

{

arr = scriptContext->GetLibrary()->CreateArrayLiteral(doubles->count);

SparseArraySegment<Var> \* segment = (SparseArraySegment<Var>\*)arr->GetHead();

for (uint i = 0; i < doubles->count; i++)

{

segment->elements[i] = JavascriptNumber::ToVarNoCheck(doubles->elements[i], scriptContext);

}

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

arr->CheckForceES5Array();

#endif

SetReg(playout->R0, arr);

}

#endif

void InterpreterStackFrame::OP\_SetArraySegmentVars(const unaligned OpLayoutAuxiliary \* playout)

{

const Js::VarArray \*vars = Js::ByteCodeReader::ReadAuxArray<Var>(playout->Offset, this->GetFunctionBody());

SparseArraySegment<Var> \* segment =(SparseArraySegment<Var> \*)GetNonVarReg(playout->R0);

JavascriptOperators::AddVarsToArraySegment(segment, vars);

}

template <class T>

void InterpreterStackFrame::OP\_SetArrayItemC\_CI4(const unaligned T\* playout)

{

JavascriptArray\* array = JavascriptArray::FromAnyArray(GetReg(playout->Instance));

uint32 index = playout->Element;

Var value = GetReg(playout->Value);

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(value);

#endif

// The array is create by the built-in on the same script context

Assert(array->GetScriptContext() == GetScriptContext());

TypeId typeId = array->GetTypeId();

if (typeId == TypeIds\_NativeIntArray)

{

JavascriptArray::OP\_SetNativeIntElementC(reinterpret\_cast<JavascriptNativeIntArray\*>(array), index, value, array->GetScriptContext());

}

else if (typeId == TypeIds\_NativeFloatArray)

{

JavascriptArray::OP\_SetNativeFloatElementC(reinterpret\_cast<JavascriptNativeFloatArray\*>(array), index, value, array->GetScriptContext());

}

else

{

array->SetArrayLiteralItem(index, value);

}

}

template <class T>

void InterpreterStackFrame::OP\_SetArrayItemI\_CI4(const unaligned T\* playout)

{

// Note that this code assumes that we only get here when we see an array literal,

// so we know that the instance is truly an array, and the index is a uint32.

// If/when we use this for cases like "a[0] = x", we'll at least have to check

// whether "a" is really an array.

JavascriptArray\* array = JavascriptArray::FromAnyArray(GetReg(playout->Instance));

// The array is create by the built-in on the same script context

Assert(array->GetScriptContext() == GetScriptContext());

uint32 index = playout->Element;

Var value = GetReg(playout->Value);

Assert(VirtualTableInfo<JavascriptArray>::HasVirtualTable(array));

SparseArraySegment<Var>\* lastUsedSeg = (SparseArraySegment<Var>\*)array->GetLastUsedSegment();

if (index < lastUsedSeg->left)

{

goto helper;

}

uint32 index2 = index - lastUsedSeg->left;

if (index2 < lastUsedSeg->size)

{

// Successful fastpath

array->DirectSetItemInLastUsedSegmentAt(index2, value);

return;

}

helper:

ScriptContext\* scriptContext = array->GetScriptContext();

JavascriptOperators::SetItem(array, array, index, value, scriptContext);

}

#if ENABLE\_PROFILE\_INFO

Var InterpreterStackFrame::OP\_ProfiledLdThis(Var thisVar, int moduleID, ScriptContext \*scriptContext)

{

FunctionBody \* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

if (JavascriptOperators::IsThisSelf(typeId))

{

Assert(typeId != TypeIds\_GlobalObject || ((Js::GlobalObject\*)thisVar)->ToThis() == thisVar);

Assert(typeId != TypeIds\_ModuleRoot || JavascriptOperators::GetThisFromModuleRoot(thisVar) == thisVar);

// Record the fact that we saw a trivial LdThis.

dynamicProfileInfo->RecordThisInfo(thisVar, ThisType\_Simple);

return thisVar;

}

thisVar = JavascriptOperators::OP\_GetThis(thisVar, moduleID, scriptContext);

// Record the fact that we saw a LdThis that had to map its source to something else, or at least

// forced us to call a helper, e.g., a FastDOM object with an unrecognized type ID.

dynamicProfileInfo->RecordThisInfo(thisVar, ThisType\_Mapped);

return thisVar;

}

Var InterpreterStackFrame::OP\_ProfiledStrictLdThis(Var thisVar, ScriptContext\* scriptContext)

{

FunctionBody \* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

if (typeId == TypeIds\_ActivationObject)

{

thisVar = scriptContext->GetLibrary()->GetUndefined();

dynamicProfileInfo->RecordThisInfo(thisVar, ThisType\_Mapped);

return thisVar;

}

dynamicProfileInfo->RecordThisInfo(thisVar, ThisType\_Simple);

return thisVar;

}

#endif

void InterpreterStackFrame::OP\_InitCachedFuncs(const unaligned OpLayoutAuxNoReg \* playout)

{

const FuncInfoArray \*info = Js::ByteCodeReader::ReadAuxArray<FuncInfoEntry>(playout->Offset, this->GetFunctionBody());

JavascriptOperators::OP\_InitCachedFuncs(this->localClosure, GetLocalFrameDisplay(), info, GetScriptContext());

}

Var InterpreterStackFrame::OP\_GetCachedFunc(Var instance, int32 index)

{

ActivationObjectEx \*obj = (ActivationObjectEx\*)ActivationObjectEx::FromVar(instance);

FuncCacheEntry \*entry = obj->GetFuncCacheEntry((uint)index);

return entry->func;

}

void InterpreterStackFrame::OP\_CommitScope(const unaligned OpLayoutAuxNoReg \* playout)

{

const Js::PropertyIdArray \*propIds = Js::ByteCodeReader::ReadPropertyIdArray(playout->Offset, this->GetFunctionBody());

this->OP\_CommitScopeHelper(playout, propIds);

}

void InterpreterStackFrame::OP\_CommitScopeHelper(const unaligned OpLayoutAuxNoReg \*playout, const PropertyIdArray \*propIds)

{

ActivationObjectEx \*obj = (ActivationObjectEx\*)ActivationObjectEx::FromVar(/\*GetReg(playout->R0)\*/this->localClosure);

ScriptFunction \*func = obj->GetParentFunc();

Assert(obj->GetParentFunc() == func);

if (func->GetCachedScope() == obj)

{

PropertyId firstVarSlot = ActivationObjectEx::GetFirstVarSlot(propIds);

Var undef = scriptContext->GetLibrary()->GetUndefined();

for (uint i = firstVarSlot; i < propIds->count; i++)

{

obj->SetSlot(SetSlotArguments(propIds->elements[i], i, undef));

}

obj->SetCommit(true);

}

}

Var InterpreterStackFrame::OP\_NewScObjectSimple()

{

Var object = scriptContext->GetLibrary()->CreateObject(true);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(object));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

object = JavascriptProxy::AutoProxyWrapper(object);

}

#endif

return object;

}

void InterpreterStackFrame::OP\_NewScObjectLiteral(const unaligned OpLayoutAuxiliary \* playout )

{

const Js::PropertyIdArray \*propIds = Js::ByteCodeReader::ReadPropertyIdArray(playout->Offset, this->GetFunctionBody());

Var newObj = JavascriptOperators::NewScObjectLiteral(GetScriptContext(), propIds,

this->GetFunctionBody()->GetObjectLiteralTypeRef(playout->C1));

SetReg(playout->R0, newObj);

}

void InterpreterStackFrame::OP\_NewScObjectLiteral\_LS(const unaligned OpLayoutAuxiliary \* playout, RegSlot& target)

{

const Js::PropertyIdArray \*propIds = Js::ByteCodeReader::ReadPropertyIdArray(playout->Offset, this->GetFunctionBody());

target = playout->R0;

Var newObj = JavascriptOperators::NewScObjectLiteral(GetScriptContext(), propIds,

this->GetFunctionBody()->GetObjectLiteralTypeRef(playout->C1));

SetReg(playout->R0, newObj);

target = Js::Constants::NoRegister;

}

void InterpreterStackFrame::OP\_LdPropIds(const unaligned OpLayoutAuxiliary \* playout)

{

const Js::PropertyIdArray \*propIds = Js::ByteCodeReader::ReadPropertyIdArray(playout->Offset, this->GetFunctionBody());

SetNonVarReg(playout->R0, (Var)propIds);

}

bool InterpreterStackFrame::IsCurrentLoopNativeAddr(void \* codeAddr) const

{

if (this->GetCurrentLoopNum() == LoopHeader::NoLoop)

{

return false;

}

// TODO: Do more verification?

return true;

}

#if ENABLE\_PROFILE\_INFO

void InterpreterStackFrame::OP\_RecordImplicitCall(uint loopNumber)

{

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()));

Assert(loopNumber < this->m\_functionBody->GetLoopCount());

FunctionBody\* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

ThreadContext \* threadContext = scriptContext->GetThreadContext();

dynamicProfileInfo->RecordLoopImplicitCallFlags(functionBody, loopNumber, threadContext->GetImplicitCallFlags());

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopStart(const byte \* ip)

{

const uint32 C1 = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

if(!profiled && !isAutoProfiling)

{

return ip;

}

ThreadContext \*const threadContext = GetScriptContext()->GetThreadContext();

threadContext->IncrementLoopDepth();

// Save the implicit call flags. The interpreter may switch to profiling mode during LoopBodyStart, so always do this.

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()));

this->savedLoopImplicitCallFlags[C1] = threadContext->GetImplicitCallFlags();

threadContext->SetImplicitCallFlags(ImplicitCall\_None);

this->currentLoopCounter = 0;

if(!profiled)

{

return ip;

}

LayoutSize localLayoutSize;

OpCode peekOp = m\_reader.PeekOp(ip, localLayoutSize);

Assert(peekOp != OpCode::LoopBodyStart);

if (peekOp == OpCode::ProfiledLoopBodyStart)

{

Assert(localLayoutSize == layoutSize);

ip += Js::OpCodeUtil::EncodedSize(peekOp, layoutSize);

// We are doing JIT loop body. Process the first ProfiledLoopBodyStart to avoid recording

// the implicit call before the first iteration

uint32 C2 = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

Assert(C1 == C2);

(this->\*opProfiledLoopBodyStart)(C1, layoutSize, true /\* isFirstIteration \*/);

return m\_reader.GetIP();

}

return ip;

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopEnd(const byte \* ip)

{

uint32 loopNumber = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

if(!profiled && !isAutoProfiling)

{

return ip;

}

this->CheckIfLoopIsHot(this->currentLoopCounter);

Js::FunctionBody \*fn = this->function->GetFunctionBody();

if (fn->HasDynamicProfileInfo())

{

fn->GetAnyDynamicProfileInfo()->SetLoopInterpreted(loopNumber);

// If the counter is 0, there is a high chance that some config disabled tracking that information. (ie: -off:jitloopbody)

// Assume it is valid for memop in this case.

if (this->currentLoopCounter >= (uint)CONFIG\_FLAG(MinMemOpCount) ||

(this->currentLoopCounter == 0 && !this->m\_functionBody->DoJITLoopBody())

)

{

// This flag becomes relevant only if the loop has been interpreted

fn->GetAnyDynamicProfileInfo()->SetMemOpMinReached(loopNumber);

}

}

this->currentLoopCounter = 0;

if (profiled)

{

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()));

OP\_RecordImplicitCall(loopNumber);

if(switchProfileModeOnLoopEndNumber == loopNumber)

{

// Stop profiling since the jitted loop body would be exiting the loop

Assert(!switchProfileMode);

switchProfileMode = true;

switchProfileModeOnLoopEndNumber = 0u - 1;

}

}

// Restore the implicit call flags state and add with flags in the loop as well

ThreadContext \*const threadContext = GetScriptContext()->GetThreadContext();

threadContext->AddImplicitCallFlags(this->savedLoopImplicitCallFlags[loopNumber]);

threadContext->DecrementLoopDepth();

return ip;

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopBodyStart(const byte \* ip)

{

uint32 C1 = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

if(profiled || isAutoProfiling)

{

this->currentLoopCounter++;

}

if (profiled)

{

OP\_RecordImplicitCall(C1);

}

(this->\*(profiled ? opProfiledLoopBodyStart : opLoopBodyStart))(C1, layoutSize, false /\* isFirstIteration \*/);

return m\_reader.GetIP();

}

template<bool InterruptProbe, bool JITLoopBody>

void InterpreterStackFrame::ProfiledLoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration)

{

Assert(Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()));

if (InterruptProbe)

{

this->DoInterruptProbe();

}

if (!JITLoopBody || this->IsInCatchOrFinallyBlock())

{

// For functions having try-catch-finally, jit loop bodies for loops that are contained only in a try block,

// not even indirect containment in a Catch or Finally.

return;

}

LoopHeader const \* loopHeader = DoLoopBodyStart(loopNumber, layoutSize, false, isFirstIteration);

Assert(loopHeader == nullptr || this->m\_functionBody->GetLoopNumber(loopHeader) == loopNumber);

if (loopHeader != nullptr)

{

// We executed jitted loop body, no implicit call information available for this loop

uint currentOffset = m\_reader.GetCurrentOffset();

if (!loopHeader->Contains(currentOffset) || (m\_reader.PeekOp() == OpCode::ProfiledLoopEnd))

{

// Restore the outer loop's implicit call flags

scriptContext->GetThreadContext()->SetImplicitCallFlags(this->savedLoopImplicitCallFlags[loopNumber]);

}

else

{

// We bailout from the loop, just continue collect implicit call flags for this loop

}

}

}

#else

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopStart(const byte \* ip)

{

Assert(!profiled);

return ip;

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopEnd(const byte \* ip)

{

Assert(!profiled);

return ip;

}

template <LayoutSize layoutSize, bool profiled>

const byte \* InterpreterStackFrame::OP\_ProfiledLoopBodyStart(const byte \* ip)

{

uint32 C1 = m\_reader.GetLayout<OpLayoutT\_Unsigned1<LayoutSizePolicy<layoutSize>>>(ip)->C1;

Assert(!profiled);

(this->\*opLoopBodyStart)(C1, layoutSize, false /\* isFirstIteration \*/);

return m\_reader.GetIP();

}

#endif

template<bool InterruptProbe, bool JITLoopBody>

void InterpreterStackFrame::LoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration)

{

if (InterruptProbe)

{

this->DoInterruptProbe();

}

if (!JITLoopBody || this->IsInCatchOrFinallyBlock())

{

// For functions having try-catch-finally, jit loop bodies for loops that are contained only in a try block,

// not even indirect containment in a Catch or Finally.

return;

}

DoLoopBodyStart(loopNumber, layoutSize, true, isFirstIteration);

}

LoopHeader const \* InterpreterStackFrame::DoLoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, const bool doProfileLoopCheck, const bool isFirstIteration)

{

#if ENABLE\_PROFILE\_INFO

class AutoRestoreLoopNumbers

{

private:

InterpreterStackFrame \*const interpreterStackFrame;

uint32 loopNumber;

bool doProfileLoopCheck;

public:

AutoRestoreLoopNumbers(InterpreterStackFrame \*const interpreterStackFrame, uint32 loopNumber, bool doProfileLoopCheck)

: interpreterStackFrame(interpreterStackFrame), loopNumber(loopNumber), doProfileLoopCheck(doProfileLoopCheck)

{

Assert(interpreterStackFrame->currentLoopNum == LoopHeader::NoLoop);

interpreterStackFrame->currentLoopNum = loopNumber;

interpreterStackFrame->m\_functionBody->SetRecentlyBailedOutOfJittedLoopBody(false);

}

~AutoRestoreLoopNumbers()

{

interpreterStackFrame->currentLoopNum = LoopHeader::NoLoop;

interpreterStackFrame->currentLoopCounter = 0;

Js::FunctionBody\* fn = interpreterStackFrame->m\_functionBody;

if (fn->RecentlyBailedOutOfJittedLoopBody())

{

if (doProfileLoopCheck && interpreterStackFrame->isAutoProfiling)

{

// Start profiling the loop after a bailout. Some bailouts require subsequent profile data collection such

// that the rejitted loop body would not bail out again for the same reason.

Assert(!interpreterStackFrame->switchProfileMode);

interpreterStackFrame->switchProfileMode = true;

Assert(interpreterStackFrame->switchProfileModeOnLoopEndNumber == 0u - 1);

interpreterStackFrame->switchProfileModeOnLoopEndNumber = loopNumber;

}

}

else

{

if (interpreterStackFrame->switchProfileModeOnLoopEndNumber == loopNumber)

{

// Stop profiling since the jitted loop body would be exiting the loop

Assert(!interpreterStackFrame->switchProfileMode);

interpreterStackFrame->switchProfileMode = true;

interpreterStackFrame->switchProfileModeOnLoopEndNumber = 0u - 1;

}

interpreterStackFrame->scriptContext->GetThreadContext()->DecrementLoopDepth();

}

}

};

#endif

Js::FunctionBody\* fn = this->m\_functionBody;

Assert(loopNumber < fn->GetLoopCount());

Assert(!this->IsInCatchOrFinallyBlock());

Js::LoopHeader \*loopHeader = fn->GetLoopHeader(loopNumber);

loopHeader->isInTry = (this->m\_flags & Js::InterpreterStackFrameFlags\_WithinTryBlock);

Js::LoopEntryPointInfo \* entryPointInfo = loopHeader->GetCurrentEntryPointInfo();

if (fn->ForceJITLoopBody() && loopHeader->interpretCount == 0 &&

(entryPointInfo != NULL && entryPointInfo->IsNotScheduled()))

{

#if ENABLE\_PROFILE\_INFO

if (Js::DynamicProfileInfo::EnableImplicitCallFlags(GetFunctionBody()))

{

scriptContext->GetThreadContext()->AddImplicitCallFlags(this->savedLoopImplicitCallFlags[loopNumber]);

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

GenerateLoopBody(scriptContext->GetNativeCodeGenerator(), fn, loopHeader, entryPointInfo, fn->GetLocalsCount(), this->m\_localSlots);

#endif

}

#if ENABLE\_NATIVE\_CODEGEN

// If we have JITted the loop, call the JITted code

if (entryPointInfo != NULL && entryPointInfo->IsCodeGenDone())

{

#if DBG\_DUMP

if (PHASE\_TRACE1(Js::JITLoopBodyPhase) && CONFIG\_FLAG(Verbose))

{

fn->DumpFunctionId(true);

Output::Print(L": %-20s LoopBody Execute Loop: %2d\n", fn->GetDisplayName(), loopNumber);

Output::Flush();

}

loopHeader->nativeCount++;

#endif

#ifdef BGJIT\_STATS

entryPointInfo->MarkAsUsed();

#endif

entryPointInfo->EnsureIsReadyToCall();

RegSlot envReg = this->m\_functionBody->GetEnvReg();

if (envReg != Constants::NoRegister)

{

this->SetNonVarReg(envReg, this->LdEnv());

}

RegSlot localClosureReg = this->m\_functionBody->GetLocalClosureReg();

RegSlot localFrameDisplayReg = this->m\_functionBody->GetLocalFrameDisplayReg();

if (entryPointInfo->HasJittedStackClosure())

{

// The jitted code is expecting the closure registers to point to known stack locations where

// the closures can be found and possibly boxed.

// In a jitted loop body, those locations are the local closure fields on the interpreter instance.

if (localClosureReg != Constants::NoRegister)

{

this->SetNonVarReg(localClosureReg, &this->localClosure);

}

if (localFrameDisplayReg != Constants::NoRegister)

{

this->SetNonVarReg(localFrameDisplayReg, &this->localFrameDisplay);

}

}

else

{

// In non-stack-closure jitted code, the closure registers are expected to hold the addresses

// of the actual structures.

if (localClosureReg != Constants::NoRegister)

{

this->SetNonVarReg(localClosureReg, this->localClosure);

}

if (localFrameDisplayReg != Constants::NoRegister)

{

this->SetNonVarReg(localFrameDisplayReg, this->localFrameDisplay);

}

}

uint32 innerScopeCount = this->m\_functionBody->GetInnerScopeCount();

for (uint32 i = 0; i < innerScopeCount; i++)

{

// As with the function-level scope, transfer the inner scopes from the interpreter's side storage

// to their dedicated register slots.

SetNonVarReg(this->m\_functionBody->FirstInnerScopeReg() + i, InnerScopeFromIndex(i));

}

uint newOffset = 0;

if (fn->GetIsAsmJsFunction())

{

AutoRestoreLoopNumbers autoRestore(this, loopNumber, doProfileLoopCheck);

newOffset = this->CallAsmJsLoopBody((JavascriptMethod)entryPointInfo->address);

}

else

{

AutoRestoreLoopNumbers autoRestore(this, loopNumber, doProfileLoopCheck);

newOffset = this->CallLoopBody((JavascriptMethod)entryPointInfo->address);

}

if (envReg != Constants::NoRegister)

{

SetNonVarReg(envReg, nullptr);

}

if (localClosureReg != Constants::NoRegister)

{

SetNonVarReg(localClosureReg, nullptr);

}

if (localFrameDisplayReg != Constants::NoRegister)

{

SetNonVarReg(localFrameDisplayReg, nullptr);

}

for (uint32 i = 0; i < innerScopeCount; i++)

{

// Get the (possibly updated) scopes from their registers and put them back in side storage.

// (Getting the updated values may not be necessary, actually, but it can't hurt.)

// Then null out the registers.

RegSlot reg = this->m\_functionBody->FirstInnerScopeReg() + i;

SetInnerScopeFromIndex(i, GetNonVarReg(reg));

SetNonVarReg(reg, nullptr);

}

Assert(Js::OpCodeUtil::GetOpCodeLayout(OpCode::ProfiledLoopBodyStart) == Js::OpLayoutType::Unsigned1);

Assert(Js::OpCodeUtil::GetOpCodeLayout(OpCode::LoopBodyStart) == Js::OpLayoutType::Unsigned1);

Assert(Js::OpCodeUtil::EncodedSize(Js::OpCode::LoopBodyStart, layoutSize) == Js::OpCodeUtil::EncodedSize(Js::OpCode::ProfiledLoopBodyStart, layoutSize));

uint byteCodeSize = Js::OpCodeUtil::EncodedSize(Js::OpCode::LoopBodyStart, layoutSize);

if (layoutSize == SmallLayout)

{

byteCodeSize += sizeof(OpLayoutUnsigned1\_Small);

}

else if (layoutSize == MediumLayout)

{

byteCodeSize += sizeof(OpLayoutUnsigned1\_Medium);

}

else

{

byteCodeSize += sizeof(OpLayoutUnsigned1\_Large);

}

if (newOffset == loopHeader->startOffset || newOffset == m\_reader.GetCurrentOffset() - byteCodeSize)

{

// If we bail out back the start of the loop, or start of this LoopBodyStart just skip and interpret the loop

// instead of trying to start the loop body again

// Increment the interpret count of the loop

loopHeader->interpretCount++;

}

else

{

// we do not support this in asmjs, need to add support in IrBuilderAsmjs if we need this support for asmjs

if (!entryPointInfo->GetIsAsmJSFunction())

{

this->CheckIfLoopIsHot(loopHeader->profiledLoopCounter);

}

m\_reader.SetCurrentOffset(newOffset);

}

return loopHeader;

}

#endif

// Increment the interpret count of the loop

loopHeader->interpretCount += !isFirstIteration;

const uint loopInterpretCount = GetFunctionBody()->GetLoopInterpretCount(loopHeader);

if (loopHeader->interpretCount > loopInterpretCount)

{

if (this->scriptContext->GetConfig()->IsNoNative())

{

return nullptr;

}

if (!fn->DoJITLoopBody())

{

return nullptr;

}

#if ENABLE\_NATIVE\_CODEGEN

// If the job is not scheduled then we need to schedule it now.

// It is possible a job was scheduled earlier and we find ourselves looking at the same entry point

// again. For example, if the function with the loop was JITed and bailed out then as we finish

// the call in the interpreter we might encounter a loop for which we had scheduled a JIT job before

// the function was initially scheduled. In such cases, that old JIT job will complete. If it completes

// successfully then we can go ahead and use it. If it fails then it will eventually revert to the

// NotScheduled state. Since transitions from NotScheduled can only occur on the main thread,

// by checking the state we are safe from racing with the JIT thread when looking at the other fields

// of the entry point.

if (entryPointInfo != NULL && entryPointInfo->IsNotScheduled())

{

GenerateLoopBody(scriptContext->GetNativeCodeGenerator(), fn, loopHeader, entryPointInfo, fn->GetLocalsCount(), this->m\_localSlots);

}

#endif

}

#if ENABLE\_PROFILE\_INFO

else if(

doProfileLoopCheck &&

isAutoProfiling &&

loopHeader->interpretCount > fn->GetLoopProfileThreshold(loopInterpretCount))

{

// Start profiling the loop so that the jitted loop body will have some profile data to use

Assert(!switchProfileMode);

switchProfileMode = true;

Assert(switchProfileModeOnLoopEndNumber == 0u - 1);

switchProfileModeOnLoopEndNumber = loopNumber;

}

#endif

return nullptr;

}

void

InterpreterStackFrame::CheckIfLoopIsHot(uint profiledLoopCounter)

{

Js::FunctionBody \*fn = this->function->GetFunctionBody();

if (!fn->GetHasHotLoop() && profiledLoopCounter > (uint)CONFIG\_FLAG(JitLoopBodyHotLoopThreshold))

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if(PHASE\_TRACE(Js::JITLoopBodyPhase, fn))

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(

L"Speculate Jit set for this function with loopbody: function: %s (%s)\n",

fn->GetDisplayName(),

fn->GetDebugNumberSet(debugStringBuffer));

Output::Flush();

}

#endif

fn->SetHasHotLoop();

}

}

bool InterpreterStackFrame::CheckAndResetImplicitCall(DisableImplicitFlags prevDisableImplicitFlags, ImplicitCallFlags savedImplicitCallFlags)

{

ImplicitCallFlags curImplicitCallFlags = this->scriptContext->GetThreadContext()->GetImplicitCallFlags();

if (curImplicitCallFlags > ImplicitCall\_None)

{

//error implicit bit is set , reparse without asmjs

this->scriptContext->GetThreadContext()->SetDisableImplicitFlags(prevDisableImplicitFlags);

this->scriptContext->GetThreadContext()->SetImplicitCallFlags(savedImplicitCallFlags);

return true;

}

return false;

}

uint

InterpreterStackFrame::CallLoopBody(JavascriptMethod address)

{

#ifdef \_M\_IX86

void \*savedEsp = NULL;

\_\_asm

{

// Save ESP

mov savedEsp, esp

// 8-byte align frame to improve floating point perf of our JIT'd code.

and esp, -8

// Add an extra 4-bytes to the stack since we'll be pushing 3 arguments

push eax

}

#endif

#if defined(\_M\_ARM32\_OR\_ARM64)

// For ARM we need to make sure that pipeline is synchronized with memory/cache for newly jitted code.

// Note: this does not seem to affect perf, but if it was, we could add a boolean isCalled to EntryPointInfo

// and do ISB only for 1st time this entry point is called (potential working set regression though).

\_InstructionSynchronizationBarrier();

#endif

uint newOffset = ::Math::PointerCastToIntegral<uint>(address(function, CallInfo(CallFlags\_InternalFrame, 1), this));

#ifdef \_M\_IX86

\_asm

{

// Restore ESP

mov esp, savedEsp

}

#endif

return newOffset;

}

uint

InterpreterStackFrame::CallAsmJsLoopBody(JavascriptMethod address)

{

#ifdef \_M\_IX86

void \*savedEsp = NULL;

\_\_asm

{

// Save ESP

mov savedEsp, esp

// Add an extra 4-bytes to the stack since we'll be pushing 3 arguments

push eax

}

#endif

#if defined(\_M\_ARM32\_OR\_ARM64)

// For ARM we need to make sure that pipeline is synchronized with memory/cache for newly jitted code.

// Note: this does not seem to affect perf, but if it was, we could add a boolean isCalled to EntryPointInfo

// and do ISB only for 1st time this entry point is called (potential working set regression though).

\_InstructionSynchronizationBarrier();

#endif

uint newOffset = ::Math::PointerCastToIntegral<uint>(address(function, CallInfo(CallFlags\_InternalFrame, 1), this));

#ifdef \_M\_IX86

\_asm

{

// Restore ESP

mov esp, savedEsp

}

#endif

return newOffset;

}

template <class T>

void InterpreterStackFrame::OP\_NewScObjectNoCtorFull(const unaligned T\* playout)

{

Var function = GetReg(playout->R1);

Var newObj = JavascriptOperators::NewScObjectNoCtorFull(function, GetScriptContext());

SetReg(playout->R0, newObj);

}

///----------------------------------------------------------------------------

///

/// InterpreterStackFrame::OP\_NewScObject

///

/// OP\_NewScObject() allocates a new DynamicObject and initializes it with an

/// optional "constructor" function.

///

/// NOTE: The return register must be carefully chosen to ensure proper

/// behavior:

/// 1. OpCode::NewInstance should never specify "R0" as the register to

/// store the new instance, because it will get whacked from the

/// "constructor" function's return value:

///

/// var a1 = Date(); <-- a1 = string returned from Date() function

/// var a2 = new Date(); <-- a2 = instance return from NewInstance.

/// Date()'s return value is thrown away.

///

/// 2. If an exception is thrown during construction, the destination

/// variable / field should \_\_not\_\_ be modified. Therefore, the destination

/// register should always be a temporary and never a valid local variable.

/// After successfully returning from the constructor function, the new

/// instance is valid and may be stored in its final destination variable /

/// field.

///

/// OPCODE NewObject:

/// T1 = new DynamicObject(Function.Prototype)

/// OutArg[0] = T1

/// Call(Function, ArgCount)

/// Local[Return] = T1

///

/// - R0: Destination "local" register

/// - R1: Optional constructor JavascriptFunction instance or 'null'

///

///----------------------------------------------------------------------------

template <class T, bool Profiled, bool ICIndex>

void InterpreterStackFrame::OP\_NewScObject\_Impl(const unaligned T\* playout, InlineCacheIndex inlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices)

{

if (ICIndex)

{

Assert(inlineCacheIndex != Js::Constants::NoInlineCacheIndex);

}

Var newVarInstance =

#if ENABLE\_PROFILE\_INFO

Profiled ?

ProfiledNewScObject\_Helper(

GetReg(playout->Function),

playout->ArgCount,

static\_cast<const unaligned OpLayoutDynamicProfile<T> \*>(playout)->profileId,

inlineCacheIndex,

spreadIndices) :

#endif

NewScObject\_Helper(GetReg(playout->Function), playout->ArgCount, spreadIndices);

SetReg((RegSlot)playout->Return, newVarInstance);

}

template <class T, bool Profiled>

void InterpreterStackFrame::OP\_NewScObjArray\_Impl(const unaligned T\* playout, const Js::AuxArray<uint32> \*spreadIndices)

{

// Always profile this operation when auto-profiling so that array type changes are tracked

#if ENABLE\_PROFILE\_INFO

if (!Profiled && !isAutoProfiling)

#else

Assert(!Profiled);

#endif

{

OP\_NewScObject\_Impl<T, Profiled, false>(playout, Js::Constants::NoInlineCacheIndex, spreadIndices);

return;

}

#if ENABLE\_PROFILE\_INFO

Arguments args(CallInfo(CallFlags\_New, playout->ArgCount), m\_outParams);

uint32 spreadSize = 0;

if (spreadIndices != nullptr)

{

spreadSize = JavascriptFunction::GetSpreadSize(args, spreadIndices, scriptContext);

// Allocate room on the stack for the spread args.

Arguments outArgs(CallInfo(CallFlags\_New, 0), nullptr);

outArgs.Info.Count = spreadSize;

const unsigned STACK\_ARGS\_ALLOCA\_THRESHOLD = 8; // Number of stack args we allow before using \_alloca

Var stackArgs[STACK\_ARGS\_ALLOCA\_THRESHOLD];

size\_t outArgsSize = 0;

if (outArgs.Info.Count > STACK\_ARGS\_ALLOCA\_THRESHOLD)

{

PROBE\_STACK(scriptContext, outArgs.Info.Count \* sizeof(Var) + Js::Constants::MinStackDefault); // args + function call

outArgsSize = outArgs.Info.Count \* sizeof(Var);

outArgs.Values = (Var\*)\_alloca(outArgsSize);

}

else

{

outArgs.Values = stackArgs;

outArgsSize = STACK\_ARGS\_ALLOCA\_THRESHOLD \* sizeof(Var);

ZeroMemory(outArgs.Values, outArgsSize); // We may not use all of the elements

}

JavascriptFunction::SpreadArgs(args, outArgs, spreadIndices, scriptContext);

SetReg(

(RegSlot)playout->Return,

ProfilingHelpers::ProfiledNewScObjArray(

GetReg(playout->Function),

outArgs,

function,

static\_cast<const unaligned OpLayoutDynamicProfile2<T> \*>(playout)->profileId,

static\_cast<const unaligned OpLayoutDynamicProfile2<T> \*>(playout)->profileId2));

}

else

{

SetReg(

(RegSlot)playout->Return,

ProfilingHelpers::ProfiledNewScObjArray(

GetReg(playout->Function),

args,

function,

static\_cast<const unaligned OpLayoutDynamicProfile2<T> \*>(playout)->profileId,

static\_cast<const unaligned OpLayoutDynamicProfile2<T> \*>(playout)->profileId2));

}

PopOut(playout->ArgCount);

#endif

}

void InterpreterStackFrame::OP\_NewScObject\_A\_Impl(const unaligned OpLayoutAuxiliary \* playout, RegSlot \*target)

{

const Js::VarArrayVarCount \* vars = Js::ByteCodeReader::ReadVarArrayVarCount(playout->Offset, this->GetFunctionBody());

int count = Js::TaggedInt::ToInt32(vars->count);

// Push the parameters to stack

for (int i=0;i<count; i++)

{

SetOut( (ArgSlot)(i+1), vars->elements[i]);

}

Var newVarInstance = NewScObject\_Helper(GetReg((RegSlot)playout->C1), (ArgSlot)count+1);

SetReg((RegSlot)playout->R0, newVarInstance);

}

Var InterpreterStackFrame::NewScObject\_Helper(Var target, ArgSlot ArgCount, const Js::AuxArray<uint32> \*spreadIndices)

{

Arguments args(CallInfo(CallFlags\_New, ArgCount), m\_outParams);

Var newVarInstance = JavascriptOperators::NewScObject(target, args, GetScriptContext(), spreadIndices);

PopOut(ArgCount);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newVarInstance));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newVarInstance = JavascriptProxy::AutoProxyWrapper(newVarInstance);

// this might come from a different scriptcontext.

newVarInstance = CrossSite::MarshalVar(GetScriptContext(), newVarInstance);

}

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

{

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().NewScriptObject( target, args, newVarInstance );

}

#endif

return newVarInstance;

}

#if ENABLE\_PROFILE\_INFO

Var InterpreterStackFrame::ProfiledNewScObject\_Helper(Var target, ArgSlot ArgCount, ProfileId profileId, InlineCacheIndex inlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices)

{

Arguments args(CallInfo(CallFlags\_New, ArgCount), m\_outParams);

Var newVarInstance =

ProfilingHelpers::ProfiledNewScObject(

target,

args,

GetFunctionBody(),

profileId,

inlineCacheIndex,

spreadIndices);

PopOut(ArgCount);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newVarInstance));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newVarInstance = JavascriptProxy::AutoProxyWrapper(newVarInstance);

// this might come from a different scriptcontext.

newVarInstance = CrossSite::MarshalVar(GetScriptContext(), newVarInstance);

}

#endif

#ifdef TELEMETRY\_PROFILED

{

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().NewScriptObject( target, args, newVarInstance );

}

#endif

return newVarInstance;

}

#endif

template <typename T>

void InterpreterStackFrame::OP\_LdElementUndefined(const unaligned OpLayoutT\_ElementU<T>\* playout)

{

if (this->m\_functionBody->IsEval())

{

JavascriptOperators::OP\_LoadUndefinedToElementDynamic(GetReg(playout->Instance),

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

}

else

{

JavascriptOperators::OP\_LoadUndefinedToElement(GetReg(playout->Instance),

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex));

}

}

template <typename T>

void InterpreterStackFrame::OP\_LdLocalElementUndefined(const unaligned OpLayoutT\_ElementRootU<T>\* playout)

{

if (this->m\_functionBody->IsEval())

{

JavascriptOperators::OP\_LoadUndefinedToElementDynamic(this->localClosure,

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

}

else

{

JavascriptOperators::OP\_LoadUndefinedToElement(this->localClosure,

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex));

}

}

template <typename T>

void InterpreterStackFrame::OP\_LdElementUndefinedScoped(const unaligned OpLayoutT\_ElementScopedU<T>\* playout)

{

// Implicit root object as default instance

JavascriptOperators::OP\_LoadUndefinedToElementScoped(GetEnvForEvalCode(),

this->m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetReg(Js::FunctionBody::RootObjectRegSlot), GetScriptContext());

}

void InterpreterStackFrame::OP\_ChkUndecl(Var aValue)

{

if (this->scriptContext->IsUndeclBlockVar(aValue))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

}

void InterpreterStackFrame::OP\_ChkNewCallFlag()

{

if (!(this->m\_callFlags & CallFlags\_New))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ClassConstructorCannotBeCalledWithoutNew);

}

}

void InterpreterStackFrame::OP\_EnsureNoRootProperty(uint propertyIdIndex)

{

Var instance = this->GetRootObject();

JavascriptOperators::OP\_EnsureNoRootProperty(instance, this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex));

}

void InterpreterStackFrame::OP\_EnsureNoRootRedeclProperty(uint propertyIdIndex)

{

Var instance = this->GetRootObject();

JavascriptOperators::OP\_EnsureNoRootRedeclProperty(instance, this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex));

}

void InterpreterStackFrame::OP\_ScopedEnsureNoRedeclProperty(Var aValue, uint propertyIdIndex, Var aValue2)

{

Js::PropertyId propertyId = this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex);

JavascriptOperators::OP\_ScopedEnsureNoRedeclProperty((FrameDisplay\*)aValue, propertyId, aValue2);

}

Var InterpreterStackFrame::OP\_InitUndecl()

{

return this->scriptContext->GetLibrary()->GetUndeclBlockVar();

}

void InterpreterStackFrame::OP\_InitUndeclSlot(Var aValue, int32 slot)

{

this->OP\_StSlot(aValue, slot, this->scriptContext->GetLibrary()->GetUndeclBlockVar());

}

void InterpreterStackFrame::OP\_TryCatch(const unaligned OpLayoutBr\* playout)

{

Js::JavascriptExceptionObject\* exception = NULL;

try

{

this->nestedTryDepth++;

// mark the stackFrame as 'in try block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinTryBlock;

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

if (scriptContext->IsInDebugMode())

{

this->ProcessWithDebugging();

this->TrySetRetOffset();

}

else

{

this->Process();

this->TrySetRetOffset();

}

}

catch (Js::JavascriptExceptionObject \* exceptionObject)

{

// We are using C++ exception handling which does not unwind the stack in the catch block.

// For stack overflow and OOM exceptions, we cannot run user code here because the stack is not unwind.

exception = exceptionObject;

}

if (--this->nestedTryDepth == -1)

{

// unmark the stackFrame as 'in try block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinTryBlock;

}

// Now that the stack is unwound, let's run the catch block.

if (exception)

{

if (exception->IsGeneratorReturnException())

{

// Generator return scenario, so no need to go into the catch block and we must rethrow to propagate the exception to down level

throw exception;

}

exception = exception->CloneIfStaticExceptionObject(scriptContext);

// We've got a JS exception. Grab the exception object and assign it to the

// catch object's location, then call the handler (i.e., we consume the Catch op here).

Var catchObject = exception->GetThrownObject(scriptContext);

m\_reader.SetCurrentRelativeOffset((const byte \*)(playout + 1), playout->RelativeJumpOffset);

LayoutSize layoutSize;

OpCode catchOp = m\_reader.ReadOp(layoutSize);

#ifdef BYTECODE\_BRANCH\_ISLAND

if (catchOp == Js::OpCode::BrLong)

{

Assert(layoutSize == SmallLayout);

auto playoutBrLong = m\_reader.BrLong();

m\_reader.SetCurrentRelativeOffset((const byte \*)(playoutBrLong + 1), playoutBrLong->RelativeJumpOffset);

catchOp = m\_reader.ReadOp(layoutSize);

}

#endif

AssertMsg(catchOp == OpCode::Catch, "Catch op not found at catch offset");

RegSlot reg = layoutSize == SmallLayout ? m\_reader.Reg1\_Small()->R0 :

layoutSize == MediumLayout ? m\_reader.Reg1\_Medium()->R0 : m\_reader.Reg1\_Large()->R0;

SetReg(reg, catchObject);

ResetOut();

this->nestedCatchDepth++;

// mark the stackFrame as 'in catch block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinCatchBlock;

this->ProcessCatch();

if (--this->nestedCatchDepth == -1)

{

// unmark the stackFrame as 'in catch block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinCatchBlock;

}

}

}

void InterpreterStackFrame::ProcessCatch()

{

if (this->scriptContext->IsInDebugMode())

{

this->DebugProcess();

}

else

{

this->Process();

}

}

int InterpreterStackFrame::ProcessFinally()

{

this->nestedFinallyDepth++;

// mark the stackFrame as 'in finally block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinFinallyBlock;

int newOffset = 0;

if (scriptContext->IsInDebugMode())

{

newOffset = ::Math::PointerCastToIntegral<int>(this->DebugProcess());

}

else

{

newOffset = ::Math::PointerCastToIntegral<int>(this->Process());

}

if (--this->nestedFinallyDepth == -1)

{

// unmark the stackFrame as 'in finally block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinFinallyBlock;

}

return newOffset;

}

void InterpreterStackFrame::ProcessTryCatchBailout(EHBailoutData \* ehBailoutData, uint32 tryNestingDepth)

{

int catchOffset = ehBailoutData->catchOffset;

Js::JavascriptExceptionObject\* exception = NULL;

if (catchOffset != 0)

{

try

{

this->nestedTryDepth++;

// mark the stackFrame as 'in try block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinTryBlock;

if (tryNestingDepth != 0)

{

this->ProcessTryCatchBailout(ehBailoutData->child, --tryNestingDepth);

}

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

if (scriptContext->IsInDebugMode())

{

this->ProcessWithDebugging();

this->TrySetRetOffset();

}

else

{

this->Process();

this->TrySetRetOffset();

}

}

catch (Js::JavascriptExceptionObject \* exceptionObject)

{

// We are using C++ exception handling which does not unwind the stack in the catch block.

// For stack overflow and OOM exceptions, we cannot run user code here because the stack is not unwind.

exception = exceptionObject;

}

}

else

{

this->nestedCatchDepth++;

// mark the stackFrame as 'in catch block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinCatchBlock;

if (tryNestingDepth != 0)

{

this->ProcessTryCatchBailout(ehBailoutData->child, --tryNestingDepth);

}

this->ProcessCatch();

if (--this->nestedCatchDepth == -1)

{

// unmark the stackFrame as 'in catch block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinCatchBlock;

}

return;

}

if (--this->nestedTryDepth == -1)

{

// unmark the stackFrame as 'in try block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinTryBlock;

}

// Now that the stack is unwound, let's run the catch block.

if (exception)

{

if (exception->IsGeneratorReturnException())

{

// Generator return scenario, so no need to go into the catch block and we must rethrow to propagate the exception to down level

throw exception;

}

exception = exception->CloneIfStaticExceptionObject(scriptContext);

// We've got a JS exception. Grab the exception object and assign it to the

// catch object's location, then call the handler (i.e., we consume the Catch op here).

Var catchObject = exception->GetThrownObject(scriptContext);

m\_reader.SetCurrentOffset(catchOffset);

LayoutSize layoutSize;

OpCode catchOp = m\_reader.ReadOp(layoutSize);

#ifdef BYTECODE\_BRANCH\_ISLAND

if (catchOp == Js::OpCode::BrLong)

{

Assert(layoutSize == SmallLayout);

auto playoutBrLong = m\_reader.BrLong();

m\_reader.SetCurrentRelativeOffset((const byte \*)(playoutBrLong + 1), playoutBrLong->RelativeJumpOffset);

catchOp = m\_reader.ReadOp(layoutSize);

}

#endif

AssertMsg(catchOp == OpCode::Catch, "Catch op not found at catch offset");

RegSlot reg = layoutSize == SmallLayout ? m\_reader.Reg1\_Small()->R0 :

layoutSize == MediumLayout ? m\_reader.Reg1\_Medium()->R0 : m\_reader.Reg1\_Large()->R0;

SetReg(reg, catchObject);

ResetOut();

this->nestedCatchDepth++;

// mark the stackFrame as 'in catch block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinCatchBlock;

this->ProcessCatch();

if (--this->nestedCatchDepth == -1)

{

// unmark the stackFrame as 'in catch block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinCatchBlock;

}

}

return;

}

void InterpreterStackFrame::TrySetRetOffset()

{

Assert(this->m\_flags & Js::InterpreterStackFrameFlags\_WithinTryBlock);

// It may happen that a JITted loop body returned the offset of RET. If the loop body was

// called from a try, the interpreter "Process()" should also just return.

if (this->retOffset != 0)

{

m\_reader.SetCurrentOffset(this->retOffset);

}

}

bool InterpreterStackFrame::IsInCatchOrFinallyBlock()

{

return (this->m\_flags & Js::InterpreterStackFrameFlags\_WithinCatchBlock) ||

(this->m\_flags & Js::InterpreterStackFrameFlags\_WithinFinallyBlock);

}

void InterpreterStackFrame::OP\_ResumeCatch()

{

this->m\_flags |= InterpreterStackFrameFlags\_WithinCatchBlock;

if (scriptContext->IsInDebugMode())

{

this->DebugProcess();

}

else

{

this->Process();

}

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinCatchBlock;

}

/// ---------------------------------------------------------------------------------------------------

/// The behavior we want is the following:

/// - If the control leaves the user's try without throwing, execute the finally and continue

/// after the end of the try.

/// - If the user code throws, catch this exception and then execute this finally while unwinding to

/// the handler (if any).

/// ---------------------------------------------------------------------------------------------------

void InterpreterStackFrame::ProcessTryFinally(const byte\* ip, Js::JumpOffset jumpOffset, Js::RegSlot regException, Js::RegSlot regOffset, bool hasYield)

{

Js::JavascriptExceptionObject\* pExceptionObject = nullptr;

bool skipFinallyBlock = false;

try

{

Js::Var result = nullptr;

this->nestedTryDepth++;

// mark the stackFrame as 'in try block'

this->m\_flags |= InterpreterStackFrameFlags\_WithinTryBlock;

if (scriptContext->IsInDebugMode())

{

result = this->ProcessWithDebugging();

}

else

{

result = this->Process();

}

if (result == nullptr)

{

Assert(hasYield);

skipFinallyBlock = true;

}

}

catch (Js::JavascriptExceptionObject \* e)

{

pExceptionObject = e;

}

if (--this->nestedTryDepth == -1)

{

// unmark the stackFrame as 'in try block'

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinTryBlock;

}

if (skipFinallyBlock)

{

// A leave occurred due to a yield

return;

}

// Save the current IP so execution can continue there if the finally doesn't

// take control of the flow.

int newOffset = 0;

int currOffset = m\_reader.GetCurrentOffset();

if (hasYield)

{

// save the exception if there is one to a register in case we yield during the finally block

// and need to get that exception object back upon resume in OP\_ResumeFinally

SetNonVarReg(regException, pExceptionObject);

SetNonVarReg(regOffset, reinterpret\_cast<Js::Var>(currOffset));

}

if (pExceptionObject && !pExceptionObject->IsGeneratorReturnException())

{

// Clone static exception object early in case finally block overwrites it

pExceptionObject = pExceptionObject->CloneIfStaticExceptionObject(scriptContext);

}

if (pExceptionObject && scriptContext->IsInDebugMode() &&

pExceptionObject != scriptContext->GetThreadContext()->GetPendingSOErrorObject())

{

// Swallowing an exception that has triggered a finally is not implemented

// (This appears to be the same behavior as ie8)

pExceptionObject->SetDebuggerSkip(false);

}

// Call into the finally by setting the IP, consuming the Finally, and letting the interpreter recurse.

m\_reader.SetCurrentRelativeOffset(ip, jumpOffset);

ResetOut();

newOffset = this->ProcessFinally();

bool endOfFinallyBlock = newOffset == 0;

if (endOfFinallyBlock)

{

// Finally completed without taking over the flow. Resume where we left off before calling it.

m\_reader.SetCurrentOffset(currOffset);

}

else

{

// Finally seized the flow with a jump out of its scope. Resume at the jump target and

// force the runtime to return to this frame without executing the catch.

m\_reader.SetCurrentOffset(newOffset);

return;

}

if (pExceptionObject && (endOfFinallyBlock || !pExceptionObject->IsGeneratorReturnException()))

{

throw pExceptionObject;

}

}

void InterpreterStackFrame::OP\_TryFinally(const unaligned OpLayoutBr\* playout)

{

ProcessTryFinally((const byte\*)(playout + 1), playout->RelativeJumpOffset);

}

void InterpreterStackFrame::OP\_TryFinallyWithYield(const byte\* ip, Js::JumpOffset jumpOffset, Js::RegSlot regException, Js::RegSlot regOffset)

{

ProcessTryFinally(ip, jumpOffset, regException, regOffset, true);

}

void InterpreterStackFrame::OP\_ResumeFinally(const byte\* ip, Js::JumpOffset jumpOffset, RegSlot exceptionRegSlot, RegSlot offsetRegSlot)

{

this->m\_flags |= InterpreterStackFrameFlags\_WithinFinallyBlock;

int newOffset = 0;

if (scriptContext->IsInDebugMode())

{

newOffset = ::Math::PointerCastToIntegral<int>(this->DebugProcess());

}

else

{

newOffset = ::Math::PointerCastToIntegral<int>(this->Process());

}

this->m\_flags &= ~InterpreterStackFrameFlags\_WithinFinallyBlock;

bool endOfFinallyBlock = newOffset == 0;

if (endOfFinallyBlock)

{

// Finally completed without taking over the flow. Resume where we left off before calling it.

int currOffset = ::Math::PointerCastToIntegral<int>(GetNonVarReg(offsetRegSlot));

m\_reader.SetCurrentOffset(currOffset);

}

else

{

// Finally seized the flow with a jump out of its scope. Resume at the jump target and

// force the runtime to return to this frame without executing the catch.

m\_reader.SetCurrentOffset(newOffset);

return;

}

Js::JavascriptExceptionObject\* exceptionObj = (Js::JavascriptExceptionObject\*)GetNonVarReg(exceptionRegSlot);

if (exceptionObj && (endOfFinallyBlock || !exceptionObj->IsGeneratorReturnException()))

{

throw exceptionObj;

}

}

template <typename T>

void InterpreterStackFrame::OP\_IsInst(const unaligned T\* playout)

{

Var instance = GetReg(playout->R1);

Var function = GetReg(playout->R2);

IsInstInlineCache \*inlineCache = this->GetIsInstInlineCache(playout->inlineCacheIndex);

ScriptContext\* scriptContext = GetScriptContext();

Var result = JavascriptOperators::OP\_IsInst(instance, function, scriptContext, inlineCache);

#ifdef ENABLE\_BASIC\_TELEMETRY

{

this->scriptContext->GetTelemetry().GetOpcodeTelemetry().IsInstanceOf(instance, function, result);

}

#endif

SetReg(playout->R0, result);

}

template <typename T>

void InterpreterStackFrame::OP\_ApplyArgs(const unaligned OpLayoutT\_Reg5<T> \* playout)

{

// Always save and restore implicit call flags when calling out

// REVIEW: Can we avoid it if we don't collect dynamic profile info?

ThreadContext \* threadContext = scriptContext->GetThreadContext();

Js::ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

// Currently ApplyArgs is equivalent to CallFldVoid (where we don't use the return value)

Var v=GetNonVarReg(playout->R4);

JavascriptOperators::OP\_ApplyArgs(GetReg(playout->R1),GetReg(playout->R2),

(void\*\*)GetNonVarReg(playout->R3),\*((CallInfo\*)&v),GetScriptContext());

threadContext->SetImplicitCallFlags(savedImplicitCallFlags);

}

void InterpreterStackFrame::OP\_SpreadArrayLiteral(const unaligned OpLayoutReg2Aux \* playout)

{

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

Var instance = GetReg(playout->R1);

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

const Js::AuxArray<uint32> \*spreadIndices = m\_reader.ReadAuxArray<uint32>(playout->Offset, this->GetFunctionBody());

ScriptContext\* scriptContext = GetScriptContext();

Var result = JavascriptArray::SpreadArrayArgs(instance, spreadIndices, scriptContext);

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->R0, result);

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext \*scriptContext)

{

JavascriptOperators::CheckInnerFrameDisplayArgument(argHead);

return OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdInnerFrameDisplayNoParent(void \*argHead, ScriptContext \*scriptContext)

{

JavascriptOperators::CheckInnerFrameDisplayArgument(argHead);

return OP\_LdFrameDisplayNoParent<true>(argHead, scriptContext);

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdFrameDisplay(void \*argHead, void \*argEnv, ScriptContext \*scriptContext)

{

FrameDisplay \*frameDisplay;

bool strict = this->m\_functionBody->GetIsStrictMode();

if (strict)

{

frameDisplay = JavascriptOperators::OP\_LdStrictFrameDisplay(argHead, argEnv, scriptContext);

}

else

{

frameDisplay = JavascriptOperators::OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

}

return frameDisplay;

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdFrameDisplaySetLocal(void \*argHead, void \*argEnv, ScriptContext \*scriptContext)

{

FrameDisplay \*frameDisplay = OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

this->SetLocalFrameDisplay(frameDisplay);

return frameDisplay;

}

FrameDisplay \*

InterpreterStackFrame::NewFrameDisplay(void \*argHead, void \*argEnv)

{

FrameDisplay \*frameDisplay;

bool strict = this->m\_functionBody->GetIsStrictMode();

if (!this->m\_functionBody->DoStackFrameDisplay() || !this->GetLocalFrameDisplay())

{

// Null local frame display probably indicates that we bailed out of an inlinee.

// Once we support stack closures in inlined functions, we can just assert that this value

// is never null if we should be allocating on the stack.

return this->OP\_LdFrameDisplaySetLocal(argHead, argEnv, this->GetScriptContext());

}

frameDisplay = this->GetLocalFrameDisplay();

Assert(frameDisplay != nullptr);

frameDisplay->SetTag(true);

frameDisplay->SetStrictMode(strict);

frameDisplay->SetLength(this->m\_functionBody->GetEnvDepth() + 1);

Assert(frameDisplay->GetLength() == ((FrameDisplay\*)argEnv)->GetLength() + 1);

for (uint i = 0; i < ((FrameDisplay\*)argEnv)->GetLength(); i++)

{

frameDisplay->SetItem(i + 1, ((FrameDisplay\*)argEnv)->GetItem(i));

}

frameDisplay->SetItem(0, argHead);

return frameDisplay;

}

template<bool innerFD>

FrameDisplay \*

InterpreterStackFrame::OP\_LdFrameDisplayNoParent(void \*argHead, ScriptContext \*scriptContext)

{

FrameDisplay \*frameDisplay;

bool strict = this->m\_functionBody->GetIsStrictMode();

Var argEnv = nullptr;

if (innerFD && this->m\_functionBody->GetLocalFrameDisplayReg() != Constants::NoRegister)

{

argEnv = this->GetLocalFrameDisplay();

}

if (argEnv == nullptr && this->m\_functionBody->GetEnvReg() != Constants::NoRegister)

{

argEnv = this->LdEnv();

}

if (argEnv == nullptr)

{

if (strict)

{

frameDisplay = JavascriptOperators::OP\_LdStrictFrameDisplayNoParent(argHead, scriptContext);

}

else

{

frameDisplay = JavascriptOperators::OP\_LdFrameDisplayNoParent(argHead, scriptContext);

}

}

else

{

if (strict)

{

frameDisplay = JavascriptOperators::OP\_LdStrictFrameDisplay(argHead, argEnv, scriptContext);

}

else

{

frameDisplay = JavascriptOperators::OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

}

}

return frameDisplay;

}

FrameDisplay \*

InterpreterStackFrame::OP\_LdFuncExprFrameDisplaySetLocal(void \*argHead1, void \*argHead2, ScriptContext \*scriptContext)

{

FrameDisplay \*frameDisplay = OP\_LdFrameDisplayNoParent<false>(argHead2, scriptContext);

frameDisplay = OP\_LdFrameDisplay(argHead1, frameDisplay, scriptContext);

this->SetLocalFrameDisplay(frameDisplay);

return frameDisplay;

}

FrameDisplay\* InterpreterStackFrame::GetLocalFrameDisplay() const

{

return this->localFrameDisplay;

}

void InterpreterStackFrame::SetLocalFrameDisplay(FrameDisplay\* frameDisplay)

{

this->localFrameDisplay = frameDisplay;

}

Var InterpreterStackFrame::GetLocalClosure() const

{

return this->localClosure;

}

void InterpreterStackFrame::SetLocalClosure(Var closure)

{

this->localClosure = closure;

}

void

InterpreterStackFrame::OP\_NewInnerScopeSlots(uint innerScopeIndex, uint count, int scopeIndex, ScriptContext \*scriptContext, FunctionBody \*functionBody)

{

Var \* slotArray;

slotArray =

JavascriptOperators::OP\_NewScopeSlotsWithoutPropIds(count, scopeIndex, scriptContext, functionBody);

this->SetInnerScopeFromIndex(innerScopeIndex, slotArray);

}

template <typename T>

void InterpreterStackFrame::OP\_CloneInnerScopeSlots(const unaligned OpLayoutT\_Unsigned1<T> \*playout)

{

uint innerScopeIndex = playout->C1;

Var \* slotArray;

slotArray = (Var\*)this->InnerScopeFromIndex(innerScopeIndex);

slotArray = JavascriptOperators::OP\_CloneScopeSlots(slotArray, scriptContext);

this->SetInnerScopeFromIndex(innerScopeIndex, slotArray);

}

template <typename T>

void InterpreterStackFrame::OP\_CloneBlockScope(const unaligned OpLayoutT\_Unsigned1<T> \*playout)

{

uint innerScopeIndex = playout->C1;

Var scope = this->InnerScopeFromIndex(innerScopeIndex);

BlockActivationObject\* blockScope = BlockActivationObject::FromVar(scope);

scope = JavascriptOperators::OP\_CloneBlockScope(blockScope, scriptContext);

this->SetInnerScopeFromIndex(innerScopeIndex, scope);

}

Var \*

InterpreterStackFrame::NewScopeSlots(unsigned int size, ScriptContext \*scriptContext, Var scope)

{

Var \* slotArray;

slotArray = JavascriptOperators::OP\_NewScopeSlots(size, scriptContext, scope);

this->SetLocalClosure(slotArray);

return slotArray;

}

Var \*

InterpreterStackFrame::NewScopeSlots()

{

Var \* slotArray;

FunctionBody \* functionBody = this->m\_functionBody;

uint scopeSlotCount = functionBody->scopeSlotArraySize;

Assert(scopeSlotCount != 0);

if (!functionBody->DoStackScopeSlots())

{

return this->NewScopeSlots(

scopeSlotCount + ScopeSlots::FirstSlotIndex, this->GetScriptContext(), (Var)functionBody);

}

slotArray = (Var\*)this->GetLocalClosure();

Assert(slotArray != nullptr);

ScopeSlots scopeSlots(slotArray);

scopeSlots.SetCount(scopeSlotCount);

scopeSlots.SetScopeMetadata((Var)functionBody);

Var undef = functionBody->GetScriptContext()->GetLibrary()->GetUndefined();

for (unsigned int i = 0; i < scopeSlotCount; i++)

{

scopeSlots.Set(i, undef);

}

return slotArray;

}

Var

InterpreterStackFrame::NewScopeObject()

{

Var scopeObject;

if (m\_functionBody->HasCachedScopePropIds())

{

const Js::PropertyIdArray \*propIds =

Js::ByteCodeReader::ReadPropertyIdArray(0, this->GetFunctionBody(), ActivationObjectEx::ExtraSlotCount());

Var funcExpr = this->GetFunctionExpression();

PropertyId objectId = ActivationObjectEx::GetLiteralObjectRef(propIds);

scopeObject = JavascriptOperators::OP\_InitCachedScope(funcExpr, propIds,

this->GetFunctionBody()->GetObjectLiteralTypeRef(objectId),

propIds->hasNonSimpleParams, GetScriptContext());

}

else

{

scopeObject = JavascriptOperators::OP\_NewScopeObject(GetScriptContext());

}

this->SetLocalClosure(scopeObject);

return scopeObject;

}

FrameDisplay \*

InterpreterStackFrame::GetFrameDisplayForNestedFunc() const

{

if (this->localFrameDisplay == nullptr)

{

return (FrameDisplay\*)LdEnv();

}

return this->localFrameDisplay;

}

template <class T>

void InterpreterStackFrame::OP\_NewStackScFunc(const unaligned T \* playout)

{

uint funcIndex = playout->SlotIndex;

FrameDisplay \*frameDisplay = this->GetFrameDisplayForNestedFunc();

SetRegAllowStackVarEnableOnly(playout->Value,

StackScriptFunction::OP\_NewStackScFunc(frameDisplay,

reinterpret\_cast<Js::FunctionProxy\*\*>(this->m\_functionBody->GetNestedFuncReference(funcIndex)),

this->GetStackNestedFunction(funcIndex)));

}

template <class T>

void InterpreterStackFrame::OP\_NewInnerStackScFunc(const unaligned T \* playout)

{

uint funcIndex = playout->SlotIndex;

FrameDisplay \*frameDisplay = (FrameDisplay\*)GetNonVarReg(playout->Instance);

SetRegAllowStackVarEnableOnly(playout->Value,

StackScriptFunction::OP\_NewStackScFunc(frameDisplay,

reinterpret\_cast<Js::FunctionProxy\*\*>(this->m\_functionBody->GetNestedFuncReference(funcIndex)),

this->GetStackNestedFunction(funcIndex)));

}

template <class T>

void InterpreterStackFrame::OP\_DeleteFld(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteProperty(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

SetReg(playout->Value, result);

}

template <class T>

void InterpreterStackFrame::OP\_DeleteLocalFld(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteProperty(this->localClosure, m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

SetReg(playout->Instance, result);

}

template <class T>

void InterpreterStackFrame::OP\_DeleteRootFld(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteRootProperty(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext());

SetReg(playout->Value, result);

}

template <class T>

void InterpreterStackFrame::OP\_DeleteFldStrict(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteProperty(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext(), PropertyOperation\_StrictMode);

SetReg(playout->Value, result);

}

template <class T>

void InterpreterStackFrame::OP\_DeleteRootFldStrict(const unaligned T \* playout)

{

Var result = JavascriptOperators::OP\_DeleteRootProperty(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetScriptContext(), PropertyOperation\_StrictMode);

SetReg(playout->Value, result);

}

template <typename T>

void InterpreterStackFrame::OP\_ScopedDeleteFld(const unaligned OpLayoutT\_ElementScopedC<T> \* playout)

{

// Implicit root object as default instance

Var result = JavascriptOperators::OP\_DeletePropertyScoped(GetEnvForEvalCode(),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(Js::FunctionBody::RootObjectRegSlot), GetScriptContext());

SetReg(playout->Value, result);

}

template <typename T>

void InterpreterStackFrame::OP\_ScopedDeleteFldStrict(const unaligned OpLayoutT\_ElementScopedC<T> \* playout)

{

// Implicit root object as default instance

Var result = JavascriptOperators::OP\_DeletePropertyScoped(GetEnvForEvalCode(),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(Js::FunctionBody::RootObjectRegSlot), GetScriptContext(), PropertyOperation\_StrictMode);

SetReg(playout->Value, result);

}

template <class T>

void InterpreterStackFrame::OP\_ScopedLdInst(const unaligned T \* playout)

{

Var thisVar;

Var rootObject = GetFunctionBody()->GetRootObject();

Var result = JavascriptOperators::OP\_GetInstanceScoped(GetEnvForEvalCode(),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), rootObject, &thisVar, GetScriptContext());

SetReg(playout->Value, result);

SetReg(playout->Value2, thisVar);

}

template <typename T>

void InterpreterStackFrame::OP\_ScopedInitFunc(const unaligned OpLayoutT\_ElementScopedC<T> \* playout)

{

JavascriptOperators::OP\_InitFuncScoped(GetEnvForEvalCode(),

m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex),

GetReg(playout->Value), GetReg(Js::FunctionBody::RootObjectRegSlot), GetScriptContext());

}

template <class T>

void InterpreterStackFrame::OP\_ClearAttributes(const unaligned T \* playout)

{

JavascriptOperators::OP\_ClearAttributes(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex));

}

template <class T>

void InterpreterStackFrame::OP\_InitGetFld(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitGetter(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitSetFld(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitSetter(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetReg(playout->Value));

}

template <class T>

void InterpreterStackFrame::OP\_InitSetElemI(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitElemSetter(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext()

);

}

template <class T>

void InterpreterStackFrame::OP\_InitGetElemI(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitElemGetter(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext()

);

}

template <class T>

void InterpreterStackFrame::OP\_InitComputedProperty(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitComputedProperty(

GetReg(playout->Instance),

GetReg(playout->Element),

GetReg(playout->Value),

m\_functionBody->GetScriptContext()

);

}

template <class T>

void InterpreterStackFrame::OP\_InitProto(const unaligned T \* playout)

{

JavascriptOperators::OP\_InitProto(GetReg(playout->Instance), m\_functionBody->GetReferencedPropertyId(playout->PropertyIdIndex), GetReg(playout->Value));

}

void InterpreterStackFrame::DoInterruptProbe()

{

PROBE\_STACK(scriptContext, 0);

}

void InterpreterStackFrame::InitializeStackFunctions(StackScriptFunction \* scriptFunctions)

{

this->stackNestedFunctions = scriptFunctions;

FunctionBody \* functionBody = this->m\_functionBody;

uint nestedCount = functionBody->GetNestedCount();

for (uint i = 0; i < nestedCount; i++)

{

StackScriptFunction \* stackScriptFunction = scriptFunctions + i;

FunctionProxy\* nestedProxy = functionBody->GetNestedFunc(i);

ScriptFunctionType\* type = nestedProxy->EnsureDeferredPrototypeType();

new (stackScriptFunction)StackScriptFunction(nestedProxy, type);

}

}

StackScriptFunction \* InterpreterStackFrame::GetStackNestedFunction(uint index)

{

Assert(index < this->m\_functionBody->GetNestedCount());

// Re-check if we have disable stack nested function

if (this->m\_functionBody->DoStackNestedFunc())

{

return this->stackNestedFunctions + index;

}

return nullptr;

}

void InterpreterStackFrame::SetExecutingStackFunction(ScriptFunction \* scriptFunction)

{

Assert(ThreadContext::IsOnStack(this->function));

Assert(this->m\_functionBody == scriptFunction->GetFunctionBody());

this->function = scriptFunction;

}

DWORD\_PTR InterpreterStackFrame::GetStackAddress() const

{

return m\_stackAddress;

}

void\* InterpreterStackFrame::GetAddressOfReturnAddress() const

{

return this->addressOfReturnAddress;

}

template <class T>

const byte \* InterpreterStackFrame::OP\_Br(const unaligned T \* playout)

{

return m\_reader.SetCurrentRelativeOffset((const byte \*)(playout + 1), playout->RelativeJumpOffset);

}

template <class T>

void InterpreterStackFrame::OP\_InitClass(const unaligned OpLayoutT\_Class<T> \* playout)

{

JavascriptOperators::OP\_InitClass(GetReg(playout->Constructor), playout->Extends != Js::Constants::NoRegister ? GetReg(playout->Extends) : NULL, GetScriptContext());

}

template <class T>

void InterpreterStackFrame::OP\_EmitTmpRegCount(const unaligned OpLayoutT\_Unsigned1<T> \* playout)

{

this->scriptContext->GetDebugContext()->GetProbeContainer()->SetCurrentTmpRegCount(playout->C1);

}

Var InterpreterStackFrame::OP\_LdSuper(ScriptContext \* scriptContext)

{

return JavascriptOperators::OP\_LdSuper(function, scriptContext);

}

Var InterpreterStackFrame::OP\_LdSuperCtor(ScriptContext \* scriptContext)

{

return JavascriptOperators::OP\_LdSuperCtor(function, scriptContext);

}

Var InterpreterStackFrame::OP\_ScopedLdSuper(ScriptContext \* scriptContext)

{

return JavascriptOperators::OP\_ScopedLdSuper(function, scriptContext);

}

Var InterpreterStackFrame::OP\_ScopedLdSuperCtor(ScriptContext \* scriptContext)

{

return JavascriptOperators::OP\_ScopedLdSuperCtor(function, scriptContext);

}

void InterpreterStackFrame::ValidateRegValue(Var value, bool allowStackVar, bool allowStackVarOnDisabledStackNestedFunc) const

{

#if DBG

if (value != nullptr && !TaggedNumber::Is(value))

{

if (!allowStackVar || !this->m\_functionBody->DoStackNestedFunc())

{

Assert(!ThreadContext::IsOnStack(value)

|| (allowStackVar && allowStackVarOnDisabledStackNestedFunc && StackScriptFunction::IsBoxed(value)));

}

Assert(!CrossSite::NeedMarshalVar(value, GetScriptContext()));

}

#endif

}

template <typename RegSlotType>

Var InterpreterStackFrame::GetReg(RegSlotType localRegisterID) const

{

Var value = m\_localSlots[localRegisterID];

ValidateRegValue(value);

return value;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetReg(RegSlotType localRegisterID, Var value)

{

Assert(localRegisterID == 0 || localRegisterID >= m\_functionBody->GetConstantCount());

ValidateRegValue(value);

m\_localSlots[localRegisterID] = value;

}

template <typename T>

T InterpreterStackFrame::GetRegRaw(RegSlot localRegisterID) const

{

return (T)m\_localIntSlots[localRegisterID];

}

// specialized version for doubles

template <>

double InterpreterStackFrame::GetRegRaw(RegSlot localRegisterID) const

{

return (double)m\_localDoubleSlots[localRegisterID];

}

template <>

float InterpreterStackFrame::GetRegRaw(RegSlot localRegisterID) const

{

return (float)m\_localFloatSlots[localRegisterID];

}

template <typename T>

void InterpreterStackFrame::SetRegRaw(RegSlot localRegisterID, T bValue)

{

m\_localIntSlots[localRegisterID] = (int)bValue;

}

template <>

void InterpreterStackFrame::SetRegRaw(RegSlot localRegisterID, float bValue)

{

m\_localFloatSlots[localRegisterID] = (float)bValue;

}

template <>

void InterpreterStackFrame::SetRegRaw(RegSlot localRegisterID, double bValue)

{

m\_localDoubleSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

int InterpreterStackFrame::GetRegRawInt(RegSlotType localRegisterID) const

{

return m\_localIntSlots[localRegisterID];

}

template <typename RegSlotType>

double InterpreterStackFrame::GetRegRawDouble(RegSlotType localRegisterID) const

{

return m\_localDoubleSlots[localRegisterID];

}

template <typename RegSlotType>

float InterpreterStackFrame::GetRegRawFloat(RegSlotType localRegisterID) const

{

return m\_localFloatSlots[localRegisterID];

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegRawInt(RegSlotType localRegisterID, int bValue)

{

m\_localIntSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegRawDouble(RegSlotType localRegisterID, double bValue)

{

m\_localDoubleSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegRawFloat(RegSlotType localRegisterID, float bValue)

{

m\_localFloatSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

Var InterpreterStackFrame::GetRegAllowStackVar(RegSlotType localRegisterID) const

{

Var value = m\_localSlots[localRegisterID];

ValidateRegValue(value, true);

return value;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegAllowStackVar(RegSlotType localRegisterID, Var value)

{

Assert(localRegisterID == 0 || localRegisterID >= m\_functionBody->GetConstantCount());

ValidateRegValue(value, true);

m\_localSlots[localRegisterID] = value;

}

template <typename RegSlotType>

Var InterpreterStackFrame::GetRegAllowStackVarEnableOnly(RegSlotType localRegisterID) const

{

Var value = m\_localSlots[localRegisterID];

ValidateRegValue(value, true, false);

return value;

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegAllowStackVarEnableOnly(RegSlotType localRegisterID, Var value)

{

Assert(localRegisterID == 0 || localRegisterID >= m\_functionBody->GetConstantCount());

ValidateRegValue(value, true, false);

m\_localSlots[localRegisterID] = value;

}

template <>

AsmJsSIMDValue InterpreterStackFrame::GetRegRaw(RegSlot localRegisterID) const

{

return (AsmJsSIMDValue)m\_localSimdSlots[localRegisterID];

}

template<>

void InterpreterStackFrame::SetRegRaw(RegSlot localRegisterID, AsmJsSIMDValue bValue)

{

m\_localSimdSlots[localRegisterID] = bValue;

}

template <typename RegSlotType>

AsmJsSIMDValue InterpreterStackFrame::GetRegRawSimd(RegSlotType localRegisterID) const

{

return m\_localSimdSlots[localRegisterID];

}

template <typename RegSlotType>

void InterpreterStackFrame::SetRegRawSimd(RegSlotType localRegisterID, AsmJsSIMDValue bValue)

{

m\_localSimdSlots[localRegisterID] = bValue;

}

template <class T>

void InterpreterStackFrame::OP\_SimdLdArrGeneric(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex) & TypedArrayViewMask[playout->ViewType];

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

uint8 dataWidth = playout->DataWidth;

RegSlot dstReg = playout->Value;

if (index < 0 || index + dataWidth > arr->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

AsmJsSIMDValue \*data = (AsmJsSIMDValue\*)(buffer + index);

AsmJsSIMDValue value;

value = SIMDLdData(data, dataWidth);

SetRegRawSimd(dstReg, value);

}

template <class T>

void InterpreterStackFrame::OP\_SimdLdArrConstIndex(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = playout->SlotIndex;

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

uint8 dataWidth = playout->DataWidth;

RegSlot dstReg = playout->Value;

if (index < 0 || index + dataWidth > arr->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

AsmJsSIMDValue \*data = (AsmJsSIMDValue\*)(buffer + index);

AsmJsSIMDValue value;

value = SIMDLdData(data, dataWidth);

SetRegRawSimd(dstReg, value);

}

template <class T>

void InterpreterStackFrame::OP\_SimdStArrGeneric(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex) & TypedArrayViewMask[playout->ViewType];

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

uint8 dataWidth = playout->DataWidth;

RegSlot srcReg = playout->Value;

if (index < 0 || index + dataWidth > arr->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

AsmJsSIMDValue \*data = (AsmJsSIMDValue\*)(buffer + index);

AsmJsSIMDValue value = GetRegRawSimd(srcReg);

SIMDStData(data, value, dataWidth);

}

template <class T>

void InterpreterStackFrame::OP\_SimdStArrConstIndex(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = playout->SlotIndex;

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

uint8 dataWidth = playout->DataWidth;

RegSlot srcReg = playout->Value;

if (index < 0 || index + dataWidth > arr->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

AsmJsSIMDValue \*data = (AsmJsSIMDValue\*)(buffer + index);

AsmJsSIMDValue value = GetRegRawSimd(srcReg);

SIMDStData(data, value, dataWidth);

}

Var InterpreterStackFrame::GetNonVarReg(RegSlot localRegisterID) const

{

return m\_localSlots[localRegisterID];

}

void InterpreterStackFrame::SetNonVarReg(RegSlot localRegisterID, Var aValue)

{

m\_localSlots[localRegisterID] = aValue;

}

Var InterpreterStackFrame::GetRootObject() const

{

Var rootObject = GetReg(Js::FunctionBody::RootObjectRegSlot);

Assert(rootObject == this->GetFunctionBody()->LoadRootObject());

return rootObject;

}

Var InterpreterStackFrame::OP\_ArgIn0()

{

return m\_inParams[0];

}

#if ENABLE\_PROFILE\_INFO

template <class T>

void InterpreterStackFrame::OP\_ProfiledArgOut\_A(const unaligned T \* playout)

{

FunctionBody\* functionBody = this->m\_functionBody;

DynamicProfileInfo \* dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

Assert(playout->Reg > FunctionBody::FirstRegSlot && playout->Reg < functionBody->GetConstantCount());

Var value = GetReg(playout->Reg);

if (value != nullptr && TaggedInt::Is(value))

{

dynamicProfileInfo->RecordConstParameterAtCallSite(playout->profileId, playout->Arg);

}

SetOut(playout->Arg, GetReg(playout->Reg));

}

#endif

template <class T>

void InterpreterStackFrame::OP\_ArgOut\_A(const unaligned T \* playout)

{

SetOut(playout->Arg, GetReg(playout->Reg));

}

#if DBG

template <class T>

void InterpreterStackFrame::OP\_ArgOut\_ANonVar(const unaligned T \* playout)

{

SetOut(playout->Arg, GetNonVarReg(playout->Reg));

}

#endif

template <class T>

void InterpreterStackFrame::OP\_ArgOut\_Env(const unaligned T \* playout)

{

Var argEnv;

if (this->m\_functionBody->GetLocalFrameDisplayReg() != Constants::NoRegister)

{

argEnv = this->GetLocalFrameDisplay();

}

else

{

argEnv = this->LdEnv();

}

SetOut(playout->Arg, argEnv);

}

BOOL InterpreterStackFrame::OP\_BrFalse\_A(Var aValue, ScriptContext\* scriptContext)

{

return !JavascriptConversion::ToBoolean(aValue, scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrTrue\_A(Var aValue, ScriptContext\* scriptContext)

{

return JavascriptConversion::ToBoolean(aValue, scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrNotNull\_A(Var aValue)

{

return aValue != NULL;

}

BOOL InterpreterStackFrame::OP\_BrUndecl\_A(Var aValue)

{

return this->scriptContext->GetLibrary()->IsUndeclBlockVar(aValue);

}

BOOL InterpreterStackFrame::OP\_BrNotUndecl\_A(Var aValue)

{

return !this->scriptContext->GetLibrary()->IsUndeclBlockVar(aValue);

}

BOOL InterpreterStackFrame::OP\_BrOnHasProperty(Var argInstance, uint propertyIdIndex, ScriptContext\* scriptContext)

{

return JavascriptOperators::OP\_HasProperty(argInstance,

this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex), scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrOnNoProperty(Var argInstance, uint propertyIdIndex, ScriptContext\* scriptContext)

{

return !JavascriptOperators::OP\_HasProperty(argInstance,

this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex), scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrOnNoEnvProperty(Var envInstance, int32 slotIndex, uint propertyIdIndex, ScriptContext\* scriptContext)

{

Var instance = OP\_LdFrameDisplaySlot(envInstance, slotIndex);

return !JavascriptOperators::OP\_HasProperty(instance,

this->m\_functionBody->GetReferencedPropertyId(propertyIdIndex), scriptContext);

}

BOOL InterpreterStackFrame::OP\_BrOnClassConstructor(Var aValue)

{

return JavascriptOperators::IsClassConstructor(aValue);

}

template<class T>

void InterpreterStackFrame::OP\_LdLen(const unaligned T \* const playout)

{

Assert(playout);

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

const auto instance = GetReg(playout->R1);

Var length = JavascriptOperators::OP\_GetLength(instance, GetScriptContext());

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

SetReg(playout->R0, length);

}

#if ENABLE\_PROFILE\_INFO

template<class T>

void InterpreterStackFrame::OP\_ProfiledLdLen(const unaligned OpLayoutDynamicProfile<T> \*const playout)

{

Assert(playout);

const auto functionBody = m\_functionBody;

const auto profileData = functionBody->GetDynamicProfileInfo();

const auto instance = GetReg(playout->R1);

LdElemInfo ldElemInfo;

ldElemInfo.arrayType = ValueType::Uninitialized.Merge(instance);

ThreadContext\* threadContext = this->GetScriptContext()->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

Var length = JavascriptOperators::OP\_GetLength(instance, GetScriptContext());

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

ldElemInfo.elemType = ldElemInfo.elemType.Merge(length);

profileData->RecordElementLoad(functionBody, playout->profileId, ldElemInfo);

SetReg(playout->R0, length);

}

#endif

Var InterpreterStackFrame::GetFunctionExpression()

{

// Make sure we get the boxed function object if is there, (or the function itself)

return StackScriptFunction::GetCurrentFunctionObject(this->function->GetRealFunctionObject());

}

template <class T>

void InterpreterStackFrame::OP\_LdFunctionExpression(const unaligned T \* playout)

{

SetRegAllowStackVar(playout->R0, this->GetFunctionExpression());

}

template <class T>

void InterpreterStackFrame::OP\_StFunctionExpression(const unaligned T \* playout)

{

OP\_StFunctionExpression(GetReg(playout->Instance), GetReg(playout->Value), playout->PropertyIdIndex);

}

template <class T>

void InterpreterStackFrame::OP\_StLocalFunctionExpression(const unaligned T \* playout)

{

OP\_StFunctionExpression(this->localClosure, GetReg(playout->Instance), playout->PropertyIdIndex);

}

void InterpreterStackFrame::OP\_StFunctionExpression(Var instance, Var value, PropertyIdIndexType index)

{

JavascriptOperators::OP\_StFunctionExpression(instance,

this->m\_functionBody->GetReferencedPropertyId(index), value);

}

template <class T>

void InterpreterStackFrame::OP\_LdNewTarget(const unaligned T\* playout)

{

if (this->m\_callFlags & CallFlags\_NewTarget)

{

SetRegAllowStackVar(playout->R0, (Js::RecyclableObject\*)this->m\_inParams[this->m\_inSlotsCount]);

}

else if (this->m\_callFlags & CallFlags\_New)

{

SetRegAllowStackVar(playout->R0, this->GetFunctionExpression());

}

else

{

SetReg(playout->R0, this->GetScriptContext()->GetLibrary()->GetUndefined());

}

}

Var InterpreterStackFrame::OP\_Ld\_A(Var aValue)

{

return aValue;

}

Var InterpreterStackFrame::LdEnv() const

{

return this->function->GetEnvironment();

}

void InterpreterStackFrame::SetEnv(FrameDisplay \*frameDisplay)

{

this->function->SetEnvironment(frameDisplay);

}

Var InterpreterStackFrame::OP\_LdLocalObj()

{

if (!VirtualTableInfo<ActivationObject>::HasVirtualTable(this->localClosure) &&

!VirtualTableInfo<ActivationObjectEx>::HasVirtualTable(this->localClosure))

{

Js::Throw::FatalInternalError();

}

return this->localClosure;

}

#ifndef TEMP\_DISABLE\_ASMJS

template <typename T2>

void InterpreterStackFrame::OP\_StArr(uint32 index, RegSlot value)

{

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

if (index < (arr->GetByteLength()))

{

BYTE\* buffer = arr->GetBuffer();

\*(T2\*)(buffer + index) = (T2)GetRegRaw<T2>(value);

}

}

#endif

template<> inline double InterpreterStackFrame::GetArrayViewOverflowVal()

{

return \*(double\*)&NumberConstants::k\_Nan;

}

template<> inline float InterpreterStackFrame::GetArrayViewOverflowVal()

{

return (float)\*(double\*)&NumberConstants::k\_Nan;

}

template<typename T> T InterpreterStackFrame::GetArrayViewOverflowVal()

{

return 0;

}

template <class T>

void InterpreterStackFrame::OP\_LdArrFunc(const unaligned T\* playout)

{

Var\* arr = (Var\*)GetNonVarReg(playout->Instance);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex);

m\_localSlots[playout->Value] = arr[index];

}

#ifndef TEMP\_DISABLE\_ASMJS

template <typename T2>

void InterpreterStackFrame::OP\_LdArr(uint32 index, RegSlot value)

{

JavascriptArrayBuffer\* arr = \*(JavascriptArrayBuffer\*\*)GetNonVarReg(AsmJsFunctionMemory::ArrayBufferRegister);

BYTE\* buffer = arr->GetBuffer();

T2 val = index < (arr->GetByteLength()) ? \*(T2\*)(buffer + index) : GetArrayViewOverflowVal<T2>();

SetRegRaw<T2>(value, val);

}

#endif

template <class T, typename T2>

void InterpreterStackFrame::OP\_StSlotPrimitive(const unaligned T\* playout)

{

T2\* buffer = (T2\*)GetNonVarReg(playout->Instance);

buffer[playout->SlotIndex] = GetRegRaw<T2>(playout->Value);

}

template <class T>

Var InterpreterStackFrame::OP\_LdAsmJsSlot(Var instance, const unaligned T\* playout)

{

return ((Var\*)instance)[playout->SlotIndex];

}

template <class T, typename T2>

void InterpreterStackFrame::OP\_LdSlotPrimitive(const unaligned T\* playout)

{

T2\* buffer = (T2\*)GetNonVarReg(playout->Instance);

SetRegRaw<T2>(playout->Value, buffer[playout->SlotIndex]);

}

template <class T>

void InterpreterStackFrame::OP\_LdArrGeneric(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex) & TypedArrayViewMask[playout->ViewType];

(this->\*LdArrFunc[playout->ViewType])(index, playout->Value);

}

template <class T>

void InterpreterStackFrame::OP\_LdArrConstIndex(const unaligned T\* playout)

{

const uint32 index = playout->SlotIndex;

Assert(playout->ViewType < 8);

(this->\*LdArrFunc[playout->ViewType])(index, playout->Value);

}

template <class T>

void InterpreterStackFrame::OP\_StArrGeneric(const unaligned T\* playout)

{

Assert(playout->ViewType < 8);

const uint32 index = (uint32)GetRegRawInt(playout->SlotIndex) & TypedArrayViewMask[playout->ViewType];

(this->\*StArrFunc[playout->ViewType])(index, playout->Value);

}

template <class T>

void InterpreterStackFrame::OP\_StArrConstIndex(const unaligned T\* playout)

{

const uint32 index = playout->SlotIndex;

Assert(playout->ViewType < 8);

(this->\*StArrFunc[playout->ViewType])(index, playout->Value);

}

Var InterpreterStackFrame::OP\_LdSlot(Var instance, int32 slotIndex)

{

if (!PHASE\_OFF(ClosureRangeCheckPhase, this->m\_functionBody))

{

if ((uintptr\_t)((Var\*)instance)[ScopeSlots::EncodedSlotCountSlotIndex] <= (uintptr\_t)(slotIndex - ScopeSlots::FirstSlotIndex))

{

Js::Throw::FatalInternalError();

}

}

return ((Var\*)(instance))[slotIndex];

}

template <class T>

Var InterpreterStackFrame::OP\_LdSlot(Var instance, const unaligned T\* playout)

{

return OP\_LdSlot(instance, playout->SlotIndex);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdSlot(Var instance, const unaligned T\* playout)

{

Var value = OP\_LdSlot(instance, playout->SlotIndex);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

template <class T>

Var InterpreterStackFrame::OP\_LdInnerSlot(Var slotArray, const unaligned T\* playout)

{

return OP\_LdSlot(slotArray, playout->SlotIndex2);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdInnerSlot(Var slotArray, const unaligned T\* playout)

{

Var value = OP\_LdInnerSlot(slotArray, playout);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

template <class T>

Var InterpreterStackFrame::OP\_LdInnerObjSlot(Var slotArray, const unaligned T\* playout)

{

return OP\_LdObjSlot(slotArray, playout->SlotIndex2);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdInnerObjSlot(Var slotArray, const unaligned T\* playout)

{

Var value = OP\_LdInnerObjSlot(slotArray, playout);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

Var InterpreterStackFrame::OP\_LdFrameDisplaySlot(Var instance, int32 slotIndex)

{

if (!PHASE\_OFF(ClosureRangeCheckPhase, this->m\_functionBody))

{

if (((FrameDisplay\*)instance)->GetLength() < slotIndex - Js::FrameDisplay::GetOffsetOfScopes()/sizeof(Var))

{

Js::Throw::FatalInternalError();

}

}

return ((Var\*)instance)[slotIndex];

}

template <class T>

Var InterpreterStackFrame::OP\_LdEnvObj(Var instance, const unaligned T\* playout)

{

return OP\_LdFrameDisplaySlot(instance, playout->SlotIndex);

}

template <class T>

Var InterpreterStackFrame::OP\_LdEnvSlot(Var instance, const unaligned T\* playout)

{

Var slotArray = OP\_LdFrameDisplaySlot(instance, playout->SlotIndex1);

return OP\_LdSlot(slotArray, playout->SlotIndex2);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdEnvSlot(Var instance, const unaligned T\* playout)

{

Var value = OP\_LdEnvSlot(instance, playout);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

Var InterpreterStackFrame::OP\_LdObjSlot(Var instance, int32 slotIndex)

{

Var \*slotArray = \*(Var\*\*)((char\*)instance + DynamicObject::GetOffsetOfAuxSlots());

return slotArray[slotIndex];

}

template <class T>

Var InterpreterStackFrame::OP\_LdObjSlot(Var instance, const unaligned T\* playout)

{

return OP\_LdObjSlot(instance, playout->SlotIndex);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdObjSlot(Var instance, const unaligned T\* playout)

{

Var value = OP\_LdObjSlot(instance, playout->SlotIndex);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

template <class T>

Var InterpreterStackFrame::OP\_LdEnvObjSlot(Var instance, const unaligned T\* playout)

{

Var slotArray = OP\_LdFrameDisplaySlot(instance, playout->SlotIndex1);

return OP\_LdObjSlot(slotArray, playout->SlotIndex2);

}

#if ENABLE\_PROFILE\_INFO

template <class T>

Var InterpreterStackFrame::OP\_ProfiledLdEnvObjSlot(Var instance, const unaligned T\* playout)

{

Var value = OP\_LdEnvObjSlot(instance, playout);

ProfilingHelpers::ProfileLdSlot(value, GetFunctionBody(), playout->profileId);

return value;

}

#endif

void InterpreterStackFrame::OP\_StSlot(Var instance, int32 slotIndex, Var value)

{

// We emit OpCode::StSlot in the bytecode only for scope slot arrays, which are not recyclable objects.

if (!PHASE\_OFF(ClosureRangeCheckPhase, this->m\_functionBody))

{

if ((uintptr\_t)((Var\*)instance)[ScopeSlots::EncodedSlotCountSlotIndex] <= (uintptr\_t)(slotIndex - ScopeSlots::FirstSlotIndex))

{

Js::Throw::FatalInternalError();

}

}

((Var\*)(instance))[slotIndex] = value;

}

void InterpreterStackFrame::OP\_StEnvSlot(Var instance, int32 slotIndex1, int32 slotIndex2, Var value)

{

Var slotArray = (Var\*)OP\_LdFrameDisplaySlot(instance, slotIndex1);

OP\_StSlot(slotArray, slotIndex2, value);

}

void InterpreterStackFrame::OP\_StSlotChkUndecl(Var instance, int32 slotIndex, Var value)

{

// We emit OpCode::StSlot in the bytecode only for scope slot arrays, which are not recyclable objects.

if (!PHASE\_OFF(ClosureRangeCheckPhase, this->m\_functionBody))

{

if ((uintptr\_t)((Var\*)instance)[ScopeSlots::EncodedSlotCountSlotIndex] <= (uintptr\_t)(slotIndex - ScopeSlots::FirstSlotIndex))

{

Js::Throw::FatalInternalError();

}

}

OP\_ChkUndecl(((Var\*)instance)[slotIndex]);

((Var\*)(instance))[slotIndex] = value;

}

void InterpreterStackFrame::OP\_StEnvSlotChkUndecl(Var instance, int32 slotIndex1, int32 slotIndex2, Var value)

{

Var slotArray = (Var\*)OP\_LdFrameDisplaySlot(instance, slotIndex1);

OP\_StSlotChkUndecl(slotArray, slotIndex2, value);

}

void InterpreterStackFrame::OP\_StObjSlot(Var instance, int32 slotIndex, Var value)

{

// It would be nice to assert that it's ok to store directly to slot, but we don't have the propertyId.

Var \*slotArray = \*(Var\*\*)((char\*)instance + DynamicObject::GetOffsetOfAuxSlots());

slotArray[slotIndex] = value;

}

void InterpreterStackFrame::OP\_StObjSlotChkUndecl(Var instance, int32 slotIndex, Var value)

{

// It would be nice to assert that it's ok to store directly to slot, but we don't have the propertyId.

Var \*slotArray = \*(Var\*\*)((char\*)instance + DynamicObject::GetOffsetOfAuxSlots());

OP\_ChkUndecl(slotArray[slotIndex]);

slotArray[slotIndex] = value;

}

void InterpreterStackFrame::OP\_StEnvObjSlot(Var instance, int32 slotIndex1, int32 slotIndex2, Var value)

{

// It would be nice to assert that it's ok to store directly to slot, but we don't have the propertyId.

Var envInstance = (Var\*)OP\_LdFrameDisplaySlot(instance, slotIndex1);

OP\_StObjSlot(envInstance, slotIndex2, value);

}

void InterpreterStackFrame::OP\_StEnvObjSlotChkUndecl(Var instance, int32 slotIndex1, int32 slotIndex2, Var value)

{

// It would be nice to assert that it's ok to store directly to slot, but we don't have the propertyId.

Var envInstance = (Var\*)OP\_LdFrameDisplaySlot(instance, slotIndex1);

OP\_StObjSlotChkUndecl(envInstance, slotIndex2, value);

}

Var InterpreterStackFrame::OP\_LdStackArgPtr(void)

{

// Return the address of the first param after "this".

return m\_inParams + 1;

}

// Called for the debug purpose, to create the arguments object explicitly even though script has not declared it.

Var InterpreterStackFrame::CreateHeapArguments(ScriptContext\* scriptContext)

{

return JavascriptOperators::LoadHeapArguments(this->function->GetRealFunctionObject(), this->m\_inSlotsCount - 1, &this->m\_inParams[1], scriptContext->GetLibrary()->GetNull(), (PropertyId\*)scriptContext->GetLibrary()->GetNull(), scriptContext, false);

}

template <bool letArgs>

Var InterpreterStackFrame::LdHeapArgumentsImpl(Var argsArray, ScriptContext\* scriptContext)

{

Var frameObj;

if (m\_functionBody->HasScopeObject() && argsArray != scriptContext->GetLibrary()->GetNull())

{

frameObj = this->localClosure;

Assert(frameObj);

}

else

{

frameObj = scriptContext->GetLibrary()->GetNull();

}

Var args = JavascriptOperators::LoadHeapArguments(this->function->GetRealFunctionObject(), this->m\_inSlotsCount - 1, &this->m\_inParams[1], frameObj, (PropertyId\*)argsArray, scriptContext, letArgs);

this->m\_arguments = args;

return args;

}

Var InterpreterStackFrame::OP\_LdHeapArguments(Var argsArray, ScriptContext\* scriptContext)

{

return LdHeapArgumentsImpl<false>(argsArray, scriptContext);

}

Var InterpreterStackFrame::OP\_LdLetHeapArguments(Var argsArray, ScriptContext\* scriptContext)

{

return LdHeapArgumentsImpl<true>(argsArray, scriptContext);

}

Var InterpreterStackFrame::OP\_LdHeapArgsCached(ScriptContext\* scriptContext)

{

uint32 formalsCount = this->m\_functionBody->GetInParamsCount() - 1;

Var args = JavascriptOperators::LoadHeapArgsCached(this->function->GetRealFunctionObject(), this->m\_inSlotsCount - 1, formalsCount, &this->m\_inParams[1], this->localClosure, scriptContext, false);

this->m\_arguments = args;

return args;

}

Var InterpreterStackFrame::OP\_LdLetHeapArgsCached(ScriptContext\* scriptContext)

{

uint32 formalsCount = this->m\_functionBody->GetInParamsCount() - 1;

Var args = JavascriptOperators::LoadHeapArgsCached(this->function->GetRealFunctionObject(), this->m\_inSlotsCount - 1, formalsCount, &this->m\_inParams[1], this->localClosure, scriptContext, true);

this->m\_arguments = args;

return args;

}

Var InterpreterStackFrame::OP\_LdArgumentsFromFrame()

{

return this->m\_arguments;

}

void\* InterpreterStackFrame::OP\_LdArgCnt()

{

return (void\*)m\_inSlotsCount;

}

Var InterpreterStackFrame::OP\_ResumeYield(Var yieldDataVar, RegSlot yieldStarIterator)

{

ResumeYieldData\* yieldData = static\_cast<ResumeYieldData\*>(yieldDataVar);

RecyclableObject\* iterator = yieldStarIterator != Constants::NoRegister ? RecyclableObject::FromVar(GetNonVarReg(yieldStarIterator)) : nullptr;

return JavascriptOperators::OP\_ResumeYield(yieldData, iterator);

}

void\* InterpreterStackFrame::operator new(size\_t byteSize, void\* previousAllocation)

{

//

// Placement 'new' is used by InterpreterStackFrame to initialize the C++ object on the RcInterpreter's

// program stack:

// - Unlike most other allocations, the previously allocated memory will \_\_not\_\_ be

// zero-initialized, as we do not want the overhead of zero-initializing the frame when

// calling functions.

//

// NOTE: If we wanted to add C# semantics of all locals are automatically zero-initialized,

// need to determine the most efficient mechanism for this.

//

return previousAllocation;

}

void \_\_cdecl InterpreterStackFrame::operator delete(void \* allocationToFree, void \* previousAllocation)

{

AssertMsg(allocationToFree == previousAllocation, "Memory locations should match");

AssertMsg(false, "This function should never actually be called");

}

} // namespace Js

// Make sure the macro and the layout for the op is consistent

#define DEF2(x, op, ...) \

CompileAssert(!Js::OpCodeInfo<Js::OpCode::op>::HasMultiSizeLayout); \

CompileAssert(!Js::OpCodeInfo<Js::OpCode::op>::IsExtendedOpcode);

#define DEF3(x, op, ...) DEF2(x, op)

#define EXDEF2(x, op, ...) \

CompileAssert(!Js::OpCodeInfo<Js::OpCode::op>::HasMultiSizeLayout); \

CompileAssert(Js::OpCodeInfo<Js::OpCode::op>::IsExtendedOpcode);

#define EXDEF3(x, op, ...) EXDEF2(x, op)

#define DEF2\_WMS(x, op, ...) \

CompileAssert(Js::OpCodeInfo<Js::OpCode::op>::HasMultiSizeLayout); \

CompileAssert(!Js::OpCodeInfo<Js::OpCode::op>::IsExtendedOpcode);

#define DEF3\_WMS(x, op, ...) DEF2\_WMS(x, op)

#define EXDEF2\_WMS(x, op, ...) \

CompileAssert(Js::OpCodeInfo<Js::OpCode::op>::HasMultiSizeLayout); \

CompileAssert(Js::OpCodeInfo<Js::OpCode::op>::IsExtendedOpcode);

#define EXDEF3\_WMS(x, op, ...) EXDEF2\_WMS(x, op)

#include "InterpreterHandler.inl"

// Make sure the macro and the layout for the op is consistent

#define DEF2(x, op, ...) \

CompileAssert(!Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::HasMultiSizeLayout); \

CompileAssert(!Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::IsExtendedOpcode);

#define DEF3(x, op, ...) DEF2(x, op)

#define DEF4(x, op, ...) DEF2(x, op)

#define EXDEF2(x, op, ...) \

CompileAssert(!Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::HasMultiSizeLayout); \

CompileAssert(Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::IsExtendedOpcode);

#define EXDEF3(x, op, ...) EXDEF2(x, op)

#define EXDEF4(x, op, ...) EXDEF2(x, op)

#define DEF2\_WMS(x, op, ...) \

CompileAssert(Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::HasMultiSizeLayout); \

CompileAssert(!Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::IsExtendedOpcode);

#define DEF3\_WMS(x, op, ...) DEF2\_WMS(x, op)

#define DEF4\_WMS(x, op, ...) DEF2\_WMS(x, op)

#define EXDEF2\_WMS(x, op, ...) \

CompileAssert(Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::HasMultiSizeLayout); \

CompileAssert(Js::OpCodeInfoAsmJs<Js::OpCodeAsmJs::op>::IsExtendedOpcode);

#define EXDEF3\_WMS(x, op, ...) EXDEF2\_WMS(x, op)

#define EXDEF4\_WMS(x, op, ...) EXDEF2\_WMS(x, op)

#include "InterpreterHandlerAsmJs.inl"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

extern "C" PVOID \_ReturnAddress(VOID);

#pragma intrinsic(\_ReturnAddress)

class BailOutRecord;

extern "C" void \_\_cdecl \_alloca\_probe\_16();

namespace Js

{

class EHBailoutData;

enum InterpreterStackFrameFlags : UINT16

{

InterpreterStackFrameFlags\_None = 0,

InterpreterStackFrameFlags\_WithinTryBlock = 1,

InterpreterStackFrameFlags\_WithinCatchBlock = 2,

InterpreterStackFrameFlags\_WithinFinallyBlock = 4,

InterpreterStackFrameFlags\_FromBailOut = 8,

InterpreterStackFrameFlags\_ProcessingBailOutFromEHCode = 0x10,

InterpreterStackFrameFlags\_All = 0xFFFF,

};

struct InterpreterStackFrame /\* Stack allocated, no virtuals \*/

{

PREVENT\_COPY(InterpreterStackFrame)

friend class BailOutRecord;

friend class JavascriptGeneratorFunction;

friend class JavascriptGenerator;

class Setup

{

public:

Setup(ScriptFunction \* function, Arguments& args, bool inlinee = false);

Setup(ScriptFunction \* function, Var \* inParams, int inSlotsCount, bool inlinee = false);

size\_t GetAllocationVarCount() const { return varAllocCount; }

InterpreterStackFrame \* AllocateAndInitialize(bool doProfile, bool \* releaseAlloc);

#if DBG

InterpreterStackFrame \* InitializeAllocation(\_\_in\_ecount(varAllocCount) Var \* allocation, bool initParams, bool profileParams, Var loopHeaderArray, DWORD\_PTR stackAddr, Var invalidStackVar);

#else

InterpreterStackFrame \* InitializeAllocation(\_\_in\_ecount(varAllocCount) Var \* allocation, bool initParams, bool profileParams, Var loopHeaderArray, DWORD\_PTR stackAddr);

#endif

uint GetLocalCount() const { return localCount; }

private:

template <class Fn>

void InitializeParams(InterpreterStackFrame \* newInstance, Fn callback, Var \*\*pprestDest);

template <class Fn>

void InitializeParamsAndUndef(InterpreterStackFrame \* newInstance, Fn callback, Var \*\*pprestDest);

void InitializeRestParam(InterpreterStackFrame \* newInstance, Var \*dest);

void SetupInternal();

Var \* inParams;

ScriptFunction \* const function;

FunctionBody \* const executeFunction;

void\*\* inlineCaches;

int inSlotsCount;

uint localCount;

uint varAllocCount;

uint inlineCacheCount;

Js::CallFlags callFlags;

bool bailedOutOfInlinee;

};

private:

ByteCodeReader m\_reader; // Reader for current function

int m\_inSlotsCount; // Count of actual incoming parameters to this function

Js::CallFlags m\_callFlags; // CallFlags passed to the current function

Var\* m\_inParams; // Range of 'in' parameters

Var\* m\_outParams; // Range of 'out' parameters (offset in m\_localSlots)

Var\* m\_outSp; // Stack pointer for next outparam

Var m\_arguments; // Dedicated location for this frame's arguments object

StackScriptFunction \* stackNestedFunctions;

FrameDisplay \* localFrameDisplay;

Var localClosure;

Var \*innerScopeArray;

ScriptContext\* scriptContext;

ScriptFunction \* function;

FunctionBody \* m\_functionBody;

void\*\* inlineCaches;

void \* returnAddress;

void \* addressOfReturnAddress; // Tag this frame with stack position, used by (remote) stack walker to test partially initialized interpreter stack frame.

InterpreterStackFrame \*previousInterpreterFrame;

Var loopHeaderArray; // Keeps alive any JITted loop bodies while the function is being interpreted

// 'stack address' of the frame, used for recursion detection during stepping.

// For frames created via interpreter path, we use 'this', for frames created by bailout we use stack addr of actual jitted frame

// the interpreter frame is created for.

DWORD\_PTR m\_stackAddress;

#if ENABLE\_PROFILE\_INFO

ImplicitCallFlags \* savedLoopImplicitCallFlags;

#endif

uint inlineCacheCount;

uint currentLoopNum;

uint currentLoopCounter; // This keeps tracks of how many times the current loop is executed. It's hit only in cases where jitloopbodies are not hit

// such as loops inside try\catch.

UINT16 m\_flags; // based on InterpreterStackFrameFlags

bool closureInitDone : 1;

#if ENABLE\_PROFILE\_INFO

bool switchProfileMode : 1;

bool isAutoProfiling : 1;

uint32 switchProfileModeOnLoopEndNumber;

#endif

int16 nestedTryDepth;

int16 nestedCatchDepth;

uint retOffset;

int16 nestedFinallyDepth;

void (InterpreterStackFrame::\*opLoopBodyStart)(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration);

#if ENABLE\_PROFILE\_INFO

void (InterpreterStackFrame::\*opProfiledLoopBodyStart)(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration);

#endif

#if DBG || DBG\_DUMP

void \* DEBUG\_currentByteOffset;

#endif

// Asm.js stack pointer

int\* m\_localIntSlots;

double\* m\_localDoubleSlots;

float\* m\_localFloatSlots;

\_SIMDValue\* m\_localSimdSlots;

EHBailoutData \* ehBailoutData;

// 16-byte aligned

\_\_declspec(align(16)) Var m\_localSlots[0]; // Range of locals and temporaries

static const int LocalsThreshold = 32 \* 1024; // Number of locals vars we'll allocate on the frame.

// If there are more, we'll use an arena.

typedef void(InterpreterStackFrame::\*ArrFunc)(uint32, RegSlot);

static const ArrFunc StArrFunc[8];

static const ArrFunc LdArrFunc[8];

//This class must have an empty ctor (otherwise it will break the code in InterpreterStackFrame::InterpreterThunk

inline InterpreterStackFrame() { }

void ProcessTryFinally(const byte\* ip, Js::JumpOffset jumpOffset, Js::RegSlot regException = Js::Constants::NoRegister, Js::RegSlot regOffset = Js::Constants::NoRegister, bool hasYield = false);

public:

void OP\_SetOutAsmDb(RegSlot outRegisterID, double val);

void OP\_SetOutAsmInt(RegSlot outRegisterID, int val);

void OP\_I\_SetOutAsmInt(RegSlot outRegisterID, int val);

void OP\_I\_SetOutAsmDb(RegSlot outRegisterID, double val);

void OP\_I\_SetOutAsmFlt(RegSlot outRegisterID, float val);

void OP\_I\_SetOutAsmSimd(RegSlot outRegisterID, AsmJsSIMDValue val);

void SetOut(ArgSlot outRegisterID, Var bValue);

void SetOut(ArgSlot\_OneByte outRegisterID, Var bValue);

void PushOut(Var aValue);

void PopOut(ArgSlot argCount);

FrameDisplay \* GetLocalFrameDisplay() const;

FrameDisplay \* GetFrameDisplayForNestedFunc() const;

Var InnerScopeFromRegSlot(RegSlot reg) const;

void SetClosureInitDone(bool done) { closureInitDone = done; }

void ValidateRegValue(Var value, bool allowStackVar = false, bool allowStackVarOnDisabledStackNestedFunc = true) const;

void ValidateSetRegValue(Var value, bool allowStackVar = false, bool allowStackVarOnDisabledStackNestedFunc = true) const;

template <typename RegSlotType> Var GetReg(RegSlotType localRegisterID) const;

template <typename RegSlotType> void SetReg(RegSlotType localRegisterID, Var bValue);

template <typename RegSlotType> Var GetRegAllowStackVar(RegSlotType localRegisterID) const;

template <typename RegSlotType> void SetRegAllowStackVar(RegSlotType localRegisterID, Var bValue);

template <typename RegSlotType> int GetRegRawInt( RegSlotType localRegisterID ) const;

template <typename RegSlotType> void SetRegRawInt( RegSlotType localRegisterID, int bValue );

template <typename RegSlotType> double GetRegRawDouble(RegSlotType localRegisterID) const;

template <typename RegSlotType> float GetRegRawFloat(RegSlotType localRegisterID) const;

template <typename RegSlotType> void SetRegRawDouble(RegSlotType localRegisterID, double bValue);

template <typename RegSlotType> void SetRegRawFloat(RegSlotType localRegisterID, float bValue);

template <typename T> T GetRegRaw( RegSlot localRegisterID ) const;

template <typename T> void SetRegRaw( RegSlot localRegisterID, T bValue );

template <typename RegSlotType> AsmJsSIMDValue GetRegRawSimd(RegSlotType localRegisterID) const;

template <typename RegSlotType> void SetRegRawSimd(RegSlotType localRegisterID, AsmJsSIMDValue bValue);

static DWORD GetAsmSimdValOffSet(AsmJsCallStackLayout\* stack);

template <class T> void OP\_SimdLdArrGeneric(const unaligned T\* playout);

template <class T> void OP\_SimdLdArrConstIndex(const unaligned T\* playout);

template <class T> void OP\_SimdStArrGeneric(const unaligned T\* playout);

template <class T> void OP\_SimdStArrConstIndex(const unaligned T\* playout);

template <typename RegSlotType>

Var GetRegAllowStackVarEnableOnly(RegSlotType localRegisterID) const;

template <typename RegSlotType>

void SetRegAllowStackVarEnableOnly(RegSlotType localRegisterID, Var bValue);

Var GetNonVarReg(RegSlot localRegisterID) const;

void SetNonVarReg(RegSlot localRegisterID, void \* bValue);

ScriptContext\* GetScriptContext() const { return scriptContext; }

Var GetRootObject() const;

ScriptFunction\* GetJavascriptFunction() const { return function; }

FunctionBody \* GetFunctionBody() const { return m\_functionBody; }

ByteCodeReader\* GetReader() { return &m\_reader;}

uint GetCurrentLoopNum() const { return currentLoopNum; }

InterpreterStackFrame\* GetPreviousFrame() const {return previousInterpreterFrame;}

void SetPreviousFrame(InterpreterStackFrame \*interpreterFrame) {previousInterpreterFrame = interpreterFrame;}

Var GetArgumentsObject() const { return m\_arguments; }

void SetArgumentsObject(Var args) { m\_arguments = args; }

UINT16 GetFlags() const { return m\_flags; }

void OrFlags(UINT16 addTo) { m\_flags |= addTo; }

bool IsInCatchOrFinallyBlock();

static bool IsDelayDynamicInterpreterThunk(void\* entryPoint);

Var LdEnv() const;

void SetEnv(FrameDisplay \*frameDisplay);

Var \* NewScopeSlots(unsigned int size, ScriptContext \*scriptContext, Var scope);

Var \* NewScopeSlots();

Var NewScopeObject();

FrameDisplay \* NewFrameDisplay(void \*argHead, void \*argEnv);

Var CreateHeapArguments(ScriptContext\* scriptContext);

bool IsCurrentLoopNativeAddr(void \* codeAddr) const;

void \* GetReturnAddress() { return returnAddress; }

static uint32 GetOffsetOfLocals() { return offsetof(InterpreterStackFrame, m\_localSlots); }

static uint32 GetOffsetOfArguments() { return offsetof(InterpreterStackFrame, m\_arguments); }

static uint32 GetOffsetOfInParams() { return offsetof(InterpreterStackFrame, m\_inParams); }

static uint32 GetOffsetOfInSlotsCount() { return offsetof(InterpreterStackFrame, m\_inSlotsCount); }

void PrintStack(const int\* const intSrc, const float\* const fltSrc, const double\* const dblSrc, int intConstCount, int floatConstCount, int doubleConstCount, const wchar\_t\* state);

static uint32 GetStartLocationOffset() { return offsetof(InterpreterStackFrame, m\_reader) + ByteCodeReader::GetStartLocationOffset(); }

static uint32 GetCurrentLocationOffset() { return offsetof(InterpreterStackFrame, m\_reader) + ByteCodeReader::GetCurrentLocationOffset(); }

static bool IsBrLong(OpCode op, const byte \* ip)

{

#ifdef BYTECODE\_BRANCH\_ISLAND

return (op == OpCode::ExtendedOpcodePrefix) && ((OpCode)(ByteCodeReader::PeekByteOp(ip) + (OpCode::ExtendedOpcodePrefix << 8)) == OpCode::BrLong);

#else

return false;

#endif

}

DWORD\_PTR GetStackAddress() const;

void\* GetAddressOfReturnAddress() const;

#if \_M\_IX86

static int GetRetType(JavascriptFunction\* func);

static int GetAsmJsArgSize(AsmJsCallStackLayout \* stack);

static int GetDynamicRetType(AsmJsCallStackLayout \* stack);

static DWORD GetAsmIntDbValOffSet(AsmJsCallStackLayout \* stack);

\_\_declspec(noinline) static int AsmJsInterpreter(AsmJsCallStackLayout \* stack);

#elif \_M\_X64

template <typename T>

static T AsmJsInterpreter(AsmJsCallStackLayout\* layout);

static void \* GetAsmJsInterpreterEntryPoint(AsmJsCallStackLayout\* stack);

template <typename T>

static T GetAsmJsRetVal(InterpreterStackFrame\* instance);

static Var AsmJsDelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...);

static \_\_m128 AsmJsInterpreterSimdJs(AsmJsCallStackLayout\* func);

#endif

#ifdef ASMJS\_PLAT

static void InterpreterAsmThunk(AsmJsCallStackLayout\* layout);

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

static Var DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...);

\_\_declspec(noinline) static Var InterpreterThunk(JavascriptCallStackLayout\* layout);

#else

\_\_declspec(noinline) static Var InterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...);

#endif

static Var InterpreterHelper(ScriptFunction\* function, ArgumentReader args, void\* returnAddress, void\* addressOfReturnAddress, const bool isAsmJs = false);

private:

#if DYNAMIC\_INTERPRETER\_THUNK

static JavascriptMethod EnsureDynamicInterpreterThunk(Js::ScriptFunction \* function);

#endif

template<typename T>

T ReadByteOp( const byte \*& ip

#if DBG\_DUMP

, bool isExtended = false

#endif

);

void\* \_\_cdecl operator new(size\_t byteSize, void\* previousAllocation) throw();

void \_\_cdecl operator delete(void\* allocationToFree, void\* previousAllocation) throw();

\_\_declspec(noinline) Var ProcessThunk(void\* returnAddress, void\* addressOfReturnAddress);

\_\_declspec(noinline) Var DebugProcessThunk(void\* returnAddress, void\* addressOfReturnAddress);

void AlignMemoryForAsmJs();

Var Process();

Var ProcessAsmJsModule();

Var ProcessLinkFailedAsmJsModule();

Var ProcessAsmJs();

Var ProcessProfiled();

Var ProcessUnprofiled();

Var ProcessWithDebugging();

Var DebugProcess();

// This will be called for reseting outs when resume from break on error happened

void ResetOut();

Var OP\_ArgIn0();

template <class T> void OP\_ArgOut\_Env(const unaligned T\* playout);

template <class T> void OP\_ArgOut\_A(const unaligned T\* playout);

template <class T> void OP\_ProfiledArgOut\_A(const unaligned T \* playout);

#if DBG

template <class T> void OP\_ArgOut\_ANonVar(const unaligned T\* playout);

#endif

FrameDisplay \* GetEnvForEvalCode();

BOOL OP\_BrFalse\_A(Var aValue, ScriptContext\* scriptContext);

BOOL OP\_BrTrue\_A(Var aValue, ScriptContext\* scriptContext);

BOOL OP\_BrNotNull\_A(Var aValue);

BOOL OP\_BrUndecl\_A(Var aValue);

BOOL OP\_BrNotUndecl\_A(Var aValue);

BOOL OP\_BrOnHasProperty(Var argInstance, uint propertyIdIndex, ScriptContext\* scriptContext);

BOOL OP\_BrOnNoProperty(Var argInstance, uint propertyIdIndex, ScriptContext\* scriptContext);

BOOL OP\_BrOnNoEnvProperty(Var envInstance, int32 slotIndex, uint propertyIdIndex, ScriptContext\* scriptContext);

BOOL OP\_BrOnClassConstructor(Var aValue);

RecyclableObject \* OP\_CallGetFunc(Var target);

template <class T> const byte \* OP\_Br(const unaligned T \* playout);

void OP\_AsmStartCall(const unaligned OpLayoutStartCall \* playout);

void OP\_StartCall( const unaligned OpLayoutStartCall \* playout );

void OP\_StartCall(uint outParamCount);

template <class T> void OP\_CallCommon(const unaligned T \*playout, RecyclableObject \* aFunc, unsigned flags, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

void OP\_CallAsmInternal( RecyclableObject \* function);

template <class T> void OP\_I\_AsmCall(const unaligned T\* playout) { OP\_CallAsmInternal((ScriptFunction\*)OP\_CallGetFunc(GetRegAllowStackVar(playout->Function))); }

template <class T> void OP\_CallCommonI(const unaligned T \*playout, RecyclableObject \* aFunc, unsigned flags);

template <class T> void OP\_ProfileCallCommon(const unaligned T \*playout, RecyclableObject \* aFunc, unsigned flags, ProfileId profileId, InlineCacheIndex inlineCacheIndex = Js::Constants::NoInlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T> void OP\_ProfileReturnTypeCallCommon(const unaligned T \*playout, RecyclableObject \* aFunc, unsigned flags, ProfileId profileId, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T> void OP\_CallPutCommon(const unaligned T \*playout, RecyclableObject \* aFunc);

template <class T> void OP\_CallPutCommonI(const unaligned T \*playout, RecyclableObject \* aFunc);

template <class T> void OP\_AsmCall(const unaligned T\* playout);

template <class T> void OP\_CallI(const unaligned T\* playout, unsigned flags) { OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags); }

template <class T> void OP\_CallIExtended(const unaligned T\* playout, unsigned flags) { OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_CallIExtendedFlags(const unaligned T\* playout, unsigned flags) { OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_CallIFlags(const unaligned T\* playout, unsigned flags) { playout->callFlags == Js::CallFlags::CallFlags\_NewTarget ? OP\_CallPutCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function))) : OP\_CallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags); }

template <class T> void OP\_ProfiledCallI(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId); }

template <class T> void OP\_ProfiledCallIExtended(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId, Js::Constants::NoInlineCacheIndex, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledCallIExtendedFlags(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId, Js::Constants::NoInlineCacheIndex, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledCallIWithICIndex(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId, playout->inlineCacheIndex); }

template <class T> void OP\_ProfiledCallIExtendedWithICIndex(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId, playout->inlineCacheIndex, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledCallIExtendedFlagsWithICIndex(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId, playout->inlineCacheIndex, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledCallIFlags(const unaligned T\* playout, unsigned flags) { playout->callFlags == Js::CallFlags::CallFlags\_NewTarget ? OP\_CallPutCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function))) : OP\_ProfileCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId); }

template <class T> void OP\_ProfiledReturnTypeCallI(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileReturnTypeCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId); }

template <class T> void OP\_ProfiledReturnTypeCallIExtended(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileReturnTypeCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags, playout->profileId, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledReturnTypeCallIExtendedFlags(const unaligned OpLayoutDynamicProfile<T>\* playout, unsigned flags) { OP\_ProfileReturnTypeCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId, (playout->Options & CallIExtended\_SpreadArgs) ? m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody()) : nullptr); }

template <class T> void OP\_ProfiledReturnTypeCallIFlags(const unaligned T\* playout, unsigned flags) { playout->callFlags == Js::CallFlags::CallFlags\_NewTarget ? OP\_CallPutCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function))) : OP\_ProfileReturnTypeCallCommon(playout, OP\_CallGetFunc(GetRegAllowStackVar(playout->Function)), flags | playout->callFlags, playout->profileId); }

// Patching Fastpath Operations

template <class T> void OP\_GetRootProperty(unaligned T\* playout);

template <class T> void OP\_GetRootPropertyForTypeOf(unaligned T\* playout);

template <class T> void OP\_GetRootProperty\_NoFastPath(unaligned T\* playout);

template <class T, bool Root, bool Method, bool CallApplyTarget> void ProfiledGetProperty(unaligned T\* playout, const Var instance);

template <class T> void OP\_ProfiledGetRootProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetRootPropertyForTypeOf(unaligned T\* playout);

template <class T> void OP\_GetProperty(Var instance, unaligned T\* playout);

template <class T> void OP\_GetProperty(unaligned T\* playout);

template <class T> void OP\_GetLocalProperty(unaligned T\* playout);

template <class T> void OP\_GetSuperProperty(unaligned T\* playout);

template <class T> void OP\_GetPropertyForTypeOf(unaligned T\* playout);

template <class T> void OP\_GetProperty\_NoFastPath(Var instance, unaligned T\* playout);

template <class T> void OP\_ProfiledGetProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetLocalProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetSuperProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetPropertyForTypeOf(unaligned T\* playout);

template <class T> void OP\_ProfiledGetPropertyCallApplyTarget(unaligned T\* playout);

template <class T> void OP\_GetRootMethodProperty(unaligned T\* playout);

template <class T> void OP\_GetRootMethodProperty\_NoFastPath(unaligned T\* playout);

template <class T> void OP\_ProfiledGetRootMethodProperty(unaligned T\* playout);

template <class T> void OP\_GetMethodProperty(unaligned T\* playout);

template <class T> void OP\_GetLocalMethodProperty(unaligned T\* playout);

template <class T> void OP\_GetMethodProperty(Var varInstance, unaligned T\* playout);

template <class T> void OP\_GetMethodProperty\_NoFastPath(Var varInstance, unaligned T\* playout);

template <class T> void OP\_ProfiledGetMethodProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledGetLocalMethodProperty(unaligned T\* playout);

template <typename T> void OP\_GetPropertyScoped(const unaligned OpLayoutT\_ElementP<T>\* playout);

template <typename T> void OP\_GetPropertyForTypeOfScoped(const unaligned OpLayoutT\_ElementP<T>\* playout);

template <typename T> void OP\_GetPropertyScoped\_NoFastPath(const unaligned OpLayoutT\_ElementP<T>\* playout);

template <class T> void OP\_GetMethodPropertyScoped(unaligned T\* playout);

template <class T> void OP\_GetMethodPropertyScoped\_NoFastPath(unaligned T\* playout);

#if ENABLE\_PROFILE\_INFO

template <class T> void UpdateFldInfoFlagsForGetSetInlineCandidate(unaligned T\* playout, FldInfoFlags& fldInfoFlags, CacheType cacheType,

DynamicProfileInfo \* dynamicProfileInfo, uint inlineCacheIndex, RecyclableObject \* obj);

template <class T> void UpdateFldInfoFlagsForCallApplyInlineCandidate(unaligned T\* playout, FldInfoFlags& fldInfoFlags, CacheType cacheType,

DynamicProfileInfo \* dynamicProfileInfo, uint inlineCacheIndex, RecyclableObject \* obj);

#endif

template <class T> void OP\_SetProperty(unaligned T\* playout);

template <class T> void OP\_SetLocalProperty(unaligned T\* playout);

template <class T> void OP\_SetSuperProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledSetProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledSetLocalProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledSetSuperProperty(unaligned T\* playout);

template <class T> void OP\_SetRootProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledSetRootProperty(unaligned T\* playout);

template <class T> void OP\_SetPropertyStrict(unaligned T\* playout);

template <class T> void OP\_ProfiledSetPropertyStrict(unaligned T\* playout);

template <class T> void OP\_SetRootPropertyStrict(unaligned T\* playout);

template <class T> void OP\_ProfiledSetRootPropertyStrict(unaligned T\* playout);

template <class T> void OP\_SetPropertyScoped(unaligned T\* playout, PropertyOperationFlags flags = PropertyOperation\_None);

template <class T> void OP\_SetPropertyScoped\_NoFastPath(unaligned T\* playout, PropertyOperationFlags flags);

template <class T> void OP\_SetPropertyScopedStrict(unaligned T\* playout);

template <class T> void OP\_ConsoleSetPropertyScoped(unaligned T\* playout);

template <class T> void DoSetProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T> void DoSetSuperProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T> void DoSetProperty\_NoFastPath(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T> void DoSetSuperProperty\_NoFastPath(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T, bool Root> void ProfiledSetProperty(unaligned T\* playout, Var instance, PropertyOperationFlags flags);

template <class T, bool Root> void ProfiledSetSuperProperty(unaligned T\* playout, Var instance, Var thisInstance, PropertyOperationFlags flags);

template <class T> void OP\_InitProperty(unaligned T\* playout);

template <class T> void OP\_InitLocalProperty(unaligned T\* playout);

template <class T> void OP\_InitRootProperty(unaligned T\* playout);

template <class T> void OP\_InitUndeclLetProperty(unaligned T\* playout);

template <class T> void OP\_InitUndeclLocalLetProperty(unaligned T\* playout);

void OP\_InitUndeclRootLetProperty(uint propertyIdIndex);

template <class T> void OP\_InitUndeclConstProperty(unaligned T\* playout);

template <class T> void OP\_InitUndeclLocalConstProperty(unaligned T\* playout);

void OP\_InitUndeclRootConstProperty(uint propertyIdIndex);

template <class T> void OP\_InitUndeclConsoleLetProperty(unaligned T\* playout);

template <class T> void OP\_InitUndeclConsoleConstProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledInitProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledInitLocalProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledInitRootProperty(unaligned T\* playout);

template <class T> void OP\_ProfiledInitUndeclProperty(unaligned T\* playout);

template <class T> void DoInitProperty(unaligned T\* playout, Var instance);

template <class T> void DoInitProperty\_NoFastPath(unaligned T\* playout, Var instance);

template <class T> void ProfiledInitProperty(unaligned T\* playout, Var instance);

template <class T> bool TrySetPropertyLocalFastPath(unaligned T\* playout, PropertyId pid, Var instance, InlineCache\*& inlineCache, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool doProfile> Var ProfiledDivide(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId);

template <bool doProfile> Var ProfileModulus(Var aLeft, Var aRight, ScriptContext\* scriptContext, ProfileId profileId);

template <bool doProfile> Var ProfiledSwitch(Var exp, ProfileId profileId);

// Non-patching Fastpath operations

template <typename T> void OP\_GetElementI(const unaligned T\* playout);

template <typename T> void OP\_ProfiledGetElementI(const unaligned OpLayoutDynamicProfile<T>\* playout);

template <typename T> void OP\_SetElementI(const unaligned T\* playout, PropertyOperationFlags flags = PropertyOperation\_None);

template <typename T> void OP\_ProfiledSetElementI(const unaligned OpLayoutDynamicProfile<T>\* playout, PropertyOperationFlags flags = PropertyOperation\_None);

template <typename T> void OP\_SetElementIStrict(const unaligned T\* playout);

template <typename T> void OP\_ProfiledSetElementIStrict(const unaligned OpLayoutDynamicProfile<T>\* playout);

template<class T> void OP\_LdLen(const unaligned T \*const playout);

template<class T> void OP\_ProfiledLdLen(const unaligned OpLayoutDynamicProfile<T> \*const playout);

Var OP\_ProfiledLdThis(Var thisVar, int moduleID, ScriptContext\* scriptContext);

Var OP\_ProfiledStrictLdThis(Var thisVar, ScriptContext\* scriptContext);

template <class T> void OP\_SetArrayItemI\_CI4(const unaligned T\* playout);

template <class T> void OP\_SetArrayItemC\_CI4(const unaligned T\* playout);

template <class T> void OP\_SetArraySegmentItem\_CI4(const unaligned T\* playout);

template <class T> void SetArrayLiteralItem(JavascriptArray \*arr, uint32 index, T value);

void OP\_SetArraySegmentVars(const unaligned OpLayoutAuxiliary \* playout);

template <class T> void OP\_NewScArray(const unaligned T \* playout);

template <bool Profiled, class T> void ProfiledNewScArray(const unaligned OpLayoutDynamicProfile<T> \* playout);

template <class T> void OP\_ProfiledNewScArray(const unaligned OpLayoutDynamicProfile<T> \* playout) { ProfiledNewScArray<true, T>(playout); }

template <class T> void OP\_ProfiledNewScArray\_NoProfile(const unaligned OpLayoutDynamicProfile<T> \* playout) { ProfiledNewScArray<false, T>(playout); }

void OP\_NewScIntArray(const unaligned OpLayoutAuxiliary \* playout);

void OP\_NewScFltArray(const unaligned OpLayoutAuxiliary \* playout);

void OP\_ProfiledNewScIntArray(const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \* playout);

void OP\_ProfiledNewScFltArray(const unaligned OpLayoutDynamicProfile<OpLayoutAuxiliary> \* playout);

template <class T> void OP\_LdArrayHeadSegment(const unaligned T\* playout);

inline Var GetFunctionExpression();

template <class T> inline void OP\_LdFunctionExpression(const unaligned T\* playout);

template <class T> inline void OP\_StFunctionExpression(const unaligned T\* playout);

template <class T> inline void OP\_StLocalFunctionExpression(const unaligned T\* playout);

void OP\_StFunctionExpression(Var instance, Var value, PropertyIdIndexType index);

template <class T> inline void OP\_LdNewTarget(const unaligned T\* playout);

inline Var OP\_Ld\_A(Var aValue);

inline Var OP\_LdLocalObj();

void OP\_ChkUndecl(Var aValue);

void OP\_ChkNewCallFlag();

void OP\_EnsureNoRootProperty(uint propertyIdIndex);

void OP\_EnsureNoRootRedeclProperty(uint propertyIdIndex);

void OP\_ScopedEnsureNoRedeclProperty(Var aValue, uint propertyIdIndex, Var aValue2);

Var OP\_InitUndecl();

void OP\_InitUndeclSlot(Var aValue, int32 slot);

template <class T> inline void OP\_InitInnerFld(const unaligned T \* playout);

template <class T> inline void OP\_InitLetFld(const unaligned T \* playout);

template <class T> inline void OP\_InitLocalLetFld(const unaligned T\* playout);

template <class T> inline void OP\_InitInnerLetFld(const unaligned T \* playout);

template <class T> inline void OP\_InitRootLetFld(const unaligned T \* playout);

template <class T> inline void OP\_InitConstFld(const unaligned T \* playout);

template <class T> inline void OP\_InitRootConstFld(const unaligned T \* playout);

template <class T> inline void DoInitLetFld(const unaligned T \* playout, Var instance, PropertyOperationFlags flags = PropertyOperation\_None);

template <class T> inline void DoInitConstFld(const unaligned T \* playout, Var instance, PropertyOperationFlags flags = PropertyOperation\_None);

template <class T> inline void OP\_InitClassMember(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberComputedName(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberGet(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberSet(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberGetComputedName(const unaligned T \* playout);

template <class T> inline void OP\_InitClassMemberSetComputedName(const unaligned T \* playout);

template<typename T> uint32 LogSizeOf();

template <typename T2> inline void OP\_LdArr( uint32 index, RegSlot value );

template <class T> inline void OP\_LdArrFunc(const unaligned T\* playout);

template <class T> inline void OP\_ReturnDb(const unaligned T\* playout);

template<typename T> T GetArrayViewOverflowVal();

template <typename T2> inline void OP\_StArr( uint32 index, RegSlot value );

template <class T> inline Var OP\_LdAsmJsSlot(Var instance, const unaligned T\* playout );

template <class T, typename T2> inline void OP\_StSlotPrimitive(const unaligned T\* playout);

template <class T, typename T2> inline void OP\_LdSlotPrimitive( const unaligned T\* playout );

template <class T> inline void OP\_LdArrGeneric ( const unaligned T\* playout );

template <class T> inline void OP\_LdArrConstIndex( const unaligned T\* playout );

template <class T> inline void OP\_StArrGeneric ( const unaligned T\* playout );

template <class T> inline void OP\_StArrConstIndex( const unaligned T\* playout );

inline Var OP\_LdSlot(Var instance, int32 slotIndex);

inline Var OP\_LdObjSlot(Var instance, int32 slotIndex);

inline Var OP\_LdFrameDisplaySlot(Var instance, int32 slotIndex);

template <class T> inline Var OP\_LdSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdInnerSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdInnerSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdInnerObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdInnerObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdEnvSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdEnvSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdEnvObj(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_LdEnvObjSlot(Var instance, const unaligned T\* playout);

template <class T> inline Var OP\_ProfiledLdEnvObjSlot(Var instance, const unaligned T\* playout);

inline void OP\_StSlot(Var instance, int32 slotIndex, Var value);

inline void OP\_StSlotChkUndecl(Var instance, int32 slotIndex, Var value);

inline void OP\_StEnvSlot(Var instance, int32 slotIndex1, int32 slotIndex2, Var value);

inline void OP\_StEnvSlotChkUndecl(Var instance, int32 slotIndex1, int32 slotIndex2, Var value);

inline void OP\_StObjSlot(Var instance, int32 slotIndex, Var value);

inline void OP\_StObjSlotChkUndecl(Var instance, int32 slotIndex, Var value);

inline void OP\_StEnvObjSlot(Var instance, int32 slotIndex1, int32 slotIndex2, Var value);

inline void OP\_StEnvObjSlotChkUndecl(Var instance, int32 slotIndex1, int32 slotIndex2, Var value);

inline void\* OP\_LdArgCnt();

template <bool letArgs> Var LdHeapArgumentsImpl(Var argsArray, ScriptContext\* scriptContext);

inline Var OP\_LdHeapArguments(Var argsArray, ScriptContext\* scriptContext);

inline Var OP\_LdLetHeapArguments(Var argsArray, ScriptContext\* scriptContext);

inline Var OP\_LdHeapArgsCached(ScriptContext\* scriptContext);

inline Var OP\_LdLetHeapArgsCached(ScriptContext\* scriptContext);

inline Var OP\_LdStackArgPtr();

inline Var OP\_LdArgumentsFromFrame();

Var OP\_NewScObjectSimple();

void OP\_NewScObjectLiteral(const unaligned OpLayoutAuxiliary \* playout);

void OP\_NewScObjectLiteral\_LS(const unaligned OpLayoutAuxiliary \* playout, RegSlot& target);

void OP\_LdPropIds(const unaligned OpLayoutAuxiliary \* playout);

template <bool Profile, bool JITLoopBody> void LoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration);

LoopHeader const \* DoLoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, const bool doProfileLoopCheck, bool isFirstIteration);

template <bool Profile, bool JITLoopBody> void ProfiledLoopBodyStart(uint32 loopNumber, LayoutSize layoutSize, bool isFirstIteration);

void OP\_RecordImplicitCall(uint loopNumber);

template <class T, bool Profiled, bool ICIndex> void OP\_NewScObject\_Impl(const unaligned T\* playout, InlineCacheIndex inlineCacheIndex = Js::Constants::NoInlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T, bool Profiled> void OP\_NewScObjArray\_Impl(const unaligned T\* playout, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T> void OP\_NewScObject(const unaligned T\* playout) { OP\_NewScObject\_Impl<T, false, false>(playout); }

template <class T> void OP\_NewScObjectNoCtorFull(const unaligned T\* playout);

template <class T> void OP\_NewScObjectSpread(const unaligned T\* playout) { OP\_NewScObject\_Impl<T, false, false>(playout, Js::Constants::NoInlineCacheIndex, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

template <class T> void OP\_NewScObjArray(const unaligned T\* playout) { OP\_NewScObjArray\_Impl<T, false>(playout); }

template <class T> void OP\_NewScObjArraySpread(const unaligned T\* playout) { OP\_NewScObjArray\_Impl<T, false>(playout, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

template <class T> void OP\_ProfiledNewScObject(const unaligned OpLayoutDynamicProfile<T>\* playout) { OP\_NewScObject\_Impl<T, true, false>(playout); }

template <class T> void OP\_ProfiledNewScObjectSpread(const unaligned OpLayoutDynamicProfile<T>\* playout) { OP\_NewScObject\_Impl<T, true, false>(playout, Js::Constants::NoInlineCacheIndex, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

template <class T> void OP\_ProfiledNewScObjectWithICIndex(const unaligned OpLayoutDynamicProfile<T>\* playout) { OP\_NewScObject\_Impl<T, true, true>(playout, playout->inlineCacheIndex); }

template <class T> void OP\_ProfiledNewScObjArray(const unaligned OpLayoutDynamicProfile2<T>\* playout) { OP\_NewScObjArray\_Impl<T, true>(playout); }

template <class T> void OP\_ProfiledNewScObjArray\_NoProfile(const unaligned OpLayoutDynamicProfile2<T>\* playout) { OP\_NewScObjArray\_Impl<T, false>(playout); }

template <class T> void OP\_ProfiledNewScObjArraySpread(const unaligned OpLayoutDynamicProfile2<T>\* playout) { OP\_NewScObjArray\_Impl<T, true>(playout, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

template <class T> void OP\_ProfiledNewScObjArraySpread\_NoProfile(const unaligned OpLayoutDynamicProfile2<T>\* playout) { OP\_NewScObjArray\_Impl<T, true>(playout, m\_reader.ReadAuxArray<uint32>(playout->SpreadAuxOffset, this->GetFunctionBody())); }

Var NewScObject\_Helper(Var target, ArgSlot ArgCount, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

Var ProfiledNewScObject\_Helper(Var target, ArgSlot ArgCount, ProfileId profileId, InlineCacheIndex inlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

template <class T, bool Profiled, bool ICIndex> Var OP\_NewScObjectNoArg\_Impl(const unaligned T \*playout, InlineCacheIndex inlineCacheIndex = Js::Constants::NoInlineCacheIndex);

void OP\_NewScObject\_A\_Impl(const unaligned OpLayoutAuxiliary \* playout, RegSlot \*target = nullptr);

void OP\_NewScObject\_A(const unaligned OpLayoutAuxiliary \* playout) { return OP\_NewScObject\_A\_Impl(playout); }

void OP\_InitCachedFuncs(const unaligned OpLayoutAuxNoReg \* playout);

Var OP\_GetCachedFunc(Var instance, int32 index);

void OP\_CommitScope(const unaligned OpLayoutAuxNoReg \* playout);

void OP\_CommitScopeHelper(const unaligned OpLayoutAuxNoReg \*playout, const PropertyIdArray \*propIds);

void OP\_TryCatch(const unaligned OpLayoutBr\* playout);

void ProcessCatch();

int ProcessFinally();

void ProcessTryCatchBailout(EHBailoutData \* innermostEHBailoutData, uint32 tryNestingDepth);

void OP\_TryFinally(const unaligned OpLayoutBr\* playout);

void OP\_TryFinallyWithYield(const byte\* ip, Js::JumpOffset jumpOffset, Js::RegSlot regException, Js::RegSlot regOffset);

void OP\_ResumeCatch();

void OP\_ResumeFinally(const byte\* ip, Js::JumpOffset jumpOffset, RegSlot exceptionRegSlot, RegSlot offsetRegSlot);

inline Var OP\_ResumeYield(Var yieldDataVar, RegSlot yieldStarIterator = Js::Constants::NoRegister);

template <typename T> void OP\_IsInst(const unaligned T \* playout);

template <class T> void OP\_InitClass(const unaligned OpLayoutT\_Class<T> \* playout);

inline Var OP\_LdSuper(ScriptContext \* scriptContext);

inline Var OP\_LdSuperCtor(ScriptContext \* scriptContext);

inline Var OP\_ScopedLdSuper(ScriptContext \* scriptContext);

inline Var OP\_ScopedLdSuperCtor(ScriptContext \* scriptContext);

template <typename T> void OP\_LdElementUndefined(const unaligned OpLayoutT\_ElementU<T>\* playout);

template <typename T> void OP\_LdLocalElementUndefined(const unaligned OpLayoutT\_ElementRootU<T>\* playout);

template <typename T> void OP\_LdElementUndefinedScoped(const unaligned OpLayoutT\_ElementScopedU<T>\* playout);

void OP\_SpreadArrayLiteral(const unaligned OpLayoutReg2Aux \* playout);

template <LayoutSize layoutSize,bool profiled> const byte \* OP\_ProfiledLoopStart(const byte \*ip);

template <LayoutSize layoutSize,bool profiled> const byte \* OP\_ProfiledLoopEnd(const byte \*ip);

template <LayoutSize layoutSize,bool profiled> const byte \* OP\_ProfiledLoopBodyStart(const byte \*ip);

template <typename T> void OP\_ApplyArgs(const unaligned OpLayoutT\_Reg5<T> \* playout);

template <class T> void OP\_EmitTmpRegCount(const unaligned OpLayoutT\_Unsigned1<T> \* ip);

Var InnerScopeFromIndex(uint32 index) const;

void SetInnerScopeFromIndex(uint32 index, Var scope);

void OP\_NewInnerScopeSlots(uint index, uint count, int scopeIndex, ScriptContext \*scriptContext, FunctionBody \*functionBody);

template <typename T> void OP\_CloneInnerScopeSlots(const unaligned OpLayoutT\_Unsigned1<T> \*playout);

template <typename T> void OP\_CloneBlockScope(const unaligned OpLayoutT\_Unsigned1<T> \*playout);

FrameDisplay \* OP\_LdFrameDisplay(void \*argHead, void \*argEnv, ScriptContext \*scriptContext);

FrameDisplay \* OP\_LdFrameDisplaySetLocal(void \*argHead, void \*argEnv, ScriptContext \*scriptContext);

template <bool innerFD> FrameDisplay \* OP\_LdFrameDisplayNoParent(void \*argHead, ScriptContext \*scriptContext);

FrameDisplay \* OP\_LdFuncExprFrameDisplaySetLocal(void \*argHead1, void \*argHead2, ScriptContext \*scriptContext);

FrameDisplay \* OP\_LdInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext \*scriptContext);

FrameDisplay \* OP\_LdInnerFrameDisplayNoParent(void \*argHead, ScriptContext \*scriptContext);

template <class T> void OP\_NewStackScFunc(const unaligned T \* playout);

template <class T> void OP\_NewInnerStackScFunc(const unaligned T \* playout);

template <class T> void OP\_DeleteFld(const unaligned T \* playout);

template <class T> void OP\_DeleteLocalFld(const unaligned T \* playout);

template <class T> void OP\_DeleteRootFld(const unaligned T \* playout);

template <class T> void OP\_DeleteFldStrict(const unaligned T \* playout);

template <class T> void OP\_DeleteRootFldStrict(const unaligned T \* playout);

template <typename T> void OP\_ScopedDeleteFld(const unaligned OpLayoutT\_ElementScopedC<T> \* playout);

template <typename T> void OP\_ScopedDeleteFldStrict(const unaligned OpLayoutT\_ElementScopedC<T> \* playout);

template <class T> void OP\_ScopedLdInst(const unaligned T \* playout);

template <typename T> void OP\_ScopedInitFunc(const unaligned OpLayoutT\_ElementScopedC<T> \* playout);

template <class T> void OP\_ClearAttributes(const unaligned T \* playout);

template <class T> void OP\_InitGetFld(const unaligned T \* playout);

template <class T> void OP\_InitSetFld(const unaligned T \* playout);

template <class T> void OP\_InitSetElemI(const unaligned T \* playout);

template <class T> void OP\_InitGetElemI(const unaligned T \* playout);

template <class T> void OP\_InitComputedProperty(const unaligned T \* playout);

template <class T> void OP\_InitProto(const unaligned T \* playout);

uint CallLoopBody(JavascriptMethod address);

uint CallAsmJsLoopBody(JavascriptMethod address);

void DoInterruptProbe();

void CheckIfLoopIsHot(uint profiledLoopCounter);

bool CheckAndResetImplicitCall(DisableImplicitFlags prevDisableImplicitFlags,ImplicitCallFlags savedImplicitCallFlags);

class PushPopFrameHelper

{

public:

PushPopFrameHelper(InterpreterStackFrame \*interpreterFrame, void \*returnAddress, void \*addressOfReturnAddress)

: m\_threadContext(interpreterFrame->GetScriptContext()->GetThreadContext()), m\_interpreterFrame(interpreterFrame), m\_isHiddenFrame(false)

{

interpreterFrame->returnAddress = returnAddress; // Ensure these are set before pushing to interpreter frame list

interpreterFrame->addressOfReturnAddress = addressOfReturnAddress;

if (interpreterFrame->GetFunctionBody()->GetIsAsmJsFunction())

{

m\_isHiddenFrame = true;

}

else

{

m\_threadContext->PushInterpreterFrame(interpreterFrame);

}

}

~ PushPopFrameHelper()

{

if (!m\_isHiddenFrame)

{

Js::InterpreterStackFrame \*interpreterFrame = m\_threadContext->PopInterpreterFrame();

AssertMsg(interpreterFrame == m\_interpreterFrame,

"Interpreter frame chain corrupted?");

}

}

private:

ThreadContext \*m\_threadContext;

InterpreterStackFrame \*m\_interpreterFrame;

bool m\_isHiddenFrame;

};

inline InlineCache\* GetInlineCache(uint cacheIndex)

{

Assert(this->inlineCaches != nullptr);

Assert(cacheIndex < this->inlineCacheCount);

return reinterpret\_cast<InlineCache \*>(this->inlineCaches[cacheIndex]);

}

inline IsInstInlineCache\* GetIsInstInlineCache(uint cacheIndex)

{

return m\_functionBody->GetIsInstInlineCache(cacheIndex);

}

inline PropertyId GetPropertyIdFromCacheId(uint cacheIndex)

{

return m\_functionBody->GetPropertyIdFromCacheId(cacheIndex);

}

void InitializeStackFunctions(StackScriptFunction \* scriptFunctions);

StackScriptFunction \* GetStackNestedFunction(uint index);

void SetExecutingStackFunction(ScriptFunction \* scriptFunction);

friend class StackScriptFunction;

void InitializeClosures();

void SetLocalFrameDisplay(FrameDisplay \*frameDisplay);

Var GetLocalClosure() const;

void SetLocalClosure(Var closure);

void TrySetRetOffset();

};

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// Used to track how many interpreter stack frames we have on stack.

class InterpreterThunkStackCountTracker

{

public:

InterpreterThunkStackCountTracker() { ++s\_count; }

~InterpreterThunkStackCountTracker() { --s\_count; }

static int GetCount() { return s\_count; }

private:

\_\_declspec(thread) static int s\_count;

};

#endif

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Library\JavascriptNumberObject.h"

#include "Library\JavascriptStringObject.h"

#include "Library\DateImplementation.h"

#include "Library\JavascriptDate.h"

extern "C" PVOID \_ReturnAddress(VOID);

#pragma intrinsic(\_ReturnAddress)

namespace Js

{

static const double k\_2to16 = 65536.0;

static const double k\_2to31 = 2147483648.0;

static const double k\_2to32 = 4294967296.0;

// ES5 9.10 indicates that this method should throw a TypeError if the supplied value is Undefined or Null.

// Our implementation returns FALSE in this scenario, expecting the caller to throw the TypeError.

// This allows the caller to provide more context in the error message without having to unnecessarily

// construct the message string before knowing whether or not the object is coercible.

BOOL JavascriptConversion::CheckObjectCoercible(Var aValue, ScriptContext\* scriptContext)

{

TypeId typeId = JavascriptOperators::GetTypeId(aValue);

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

return FALSE;

}

return TRUE;

}

//ES5 9.11 Undefined, Null, Boolean, Number, String - return false

//If Object has an [[Call]] internal method, then return true, otherwise return false

bool JavascriptConversion::IsCallable(Var aValue)

{

if (!RecyclableObject::Is(aValue))

{

return false;

}

JavascriptMethod entryPoint = RecyclableObject::FromVar(aValue)->GetEntryPoint();

return RecyclableObject::DefaultEntryPoint != entryPoint;

}

//----------------------------------------------------------------------------

// ES5 9.12 SameValue algorithm implementation.

// 1. If Type(x) is different from Type(y), return false.

// 2. If Type(x) is Undefined, return true.

// 3. If Type(x) is Null, return true.

// 4. If Type(x) is Number, then.

// a. If x is NaN and y is NaN, return true.

// b. If x is +0 and y is -0, return false.

// c. If x is -0 and y is +0, return false.

// d. If x is the same number value as y, return true.

// e. Return false.

// 5. If Type(x) is String, then return true if x and y are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return false.

// 6. If Type(x) is Boolean, return true if x and y are both true or both false; otherwise, return false.

// 7. Return true if x and y refer to the same object. Otherwise, return false.

//----------------------------------------------------------------------------

template<bool zero>

bool JavascriptConversion::SameValueCommon(Var aLeft, Var aRight)

{

TypeId leftType = JavascriptOperators::GetTypeId(aLeft);

TypeId rightType = JavascriptOperators::GetTypeId(aRight);

//Check for undefined and null type;

if (leftType == TypeIds\_Undefined )

{

return rightType == TypeIds\_Undefined;

}

if (leftType == TypeIds\_Null)

{

return rightType == TypeIds\_Null;

}

double dblLeft, dblRight;

switch (leftType)

{

case TypeIds\_Integer:

switch (rightType)

{

case TypeIds\_Integer:

return aLeft == aRight;

case TypeIds\_Number:

dblLeft = TaggedInt::ToDouble(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

{

int leftValue = TaggedInt::ToInt32(aLeft);

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

case TypeIds\_UInt64Number:

{

int leftValue = TaggedInt::ToInt32(aLeft);

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

}

break;

case TypeIds\_Int64Number:

switch (rightType)

{

case TypeIds\_Integer:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

int rightValue = TaggedInt::ToInt32(aRight);

return leftValue == rightValue;

}

case TypeIds\_Number:

dblLeft = (double)JavascriptInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

case TypeIds\_UInt64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return ((unsigned \_\_int64)leftValue == rightValue);

}

}

break;

case TypeIds\_UInt64Number:

switch (rightType)

{

case TypeIds\_Integer:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = TaggedInt::ToInt32(aRight);

return (leftValue == (unsigned \_\_int64)rightValue);

}

case TypeIds\_Number:

dblLeft = (double)JavascriptUInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return (leftValue == (unsigned \_\_int64)rightValue);

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

}

break;

case TypeIds\_Number:

switch (rightType)

{

case TypeIds\_Integer:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = TaggedInt::ToDouble(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = (double)JavascriptInt64Number::FromVar(aRight)->GetValue();

goto CommonNumber;

case TypeIds\_UInt64Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = (double)JavascriptUInt64Number::FromVar(aRight)->GetValue();

goto CommonNumber;

case TypeIds\_Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

CommonNumber:

if (JavascriptNumber::IsNan(dblLeft) && JavascriptNumber::IsNan(dblRight))

{

return true;

}

if (zero)

{

// SameValueZero(+0,-0) returns true;

return dblLeft == dblRight;

}

else

{

// SameValue(+0,-0) returns false;

return (NumberUtilities::LuLoDbl(dblLeft) == NumberUtilities::LuLoDbl(dblRight) &&

NumberUtilities::LuHiDbl(dblLeft) == NumberUtilities::LuHiDbl(dblRight));

}

}

break;

case TypeIds\_Boolean:

switch (rightType)

{

case TypeIds\_Boolean:

return aLeft == aRight;

}

break;

case TypeIds\_String:

switch (rightType)

{

case TypeIds\_String:

return JavascriptString::Equals(aLeft, aRight);

}

break;

case TypeIds\_Symbol:

switch (rightType)

{

case TypeIds\_Symbol:

{

JavascriptSymbol\* leftSymbol = JavascriptSymbol::FromVar(aLeft);

JavascriptSymbol\* rightSymbol = JavascriptSymbol::FromVar(aRight);

return leftSymbol->GetValue() == rightSymbol->GetValue();

}

}

return false;

case TypeIds\_Function:

switch (rightType)

{

case TypeIds\_Function:

if (JavascriptFunction::FromVar(aLeft)->IsThrowTypeErrorFunction() &&

JavascriptFunction::FromVar(aRight)->IsThrowTypeErrorFunction())

{

return true;

}

}

break;

}

return aLeft == aRight;

}

template bool JavascriptConversion::SameValueCommon<false>(Var aLeft, Var aRight);

template bool JavascriptConversion::SameValueCommon<true>(Var aLeft, Var aRight);

//----------------------------------------------------------------------------

// ToObject() takes a value and converts it to a Object type

// Implementation of ES5 9.9

// The spec indicates that this method should throw a TypeError if the supplied value is Undefined or Null.

// Our implementation returns FALSE in this scenario, expecting the caller to throw the TypeError.

// This allows the caller to provide more context in the error message without having to unnecessarily

// construct the message string before knowing whether or not the value can be converted to an object.

//

// Undefined Return FALSE.

// Null Return FALSE.

// Boolean Create a new Boolean object whose [[PrimitiveValue]]

// internal property is set to the value of the boolean.

// See 15.6 for a description of Boolean objects.

// Return TRUE.

// Number Create a new Number object whose [[PrimitiveValue]]

// internal property is set to the value of the number.

// See 15.7 for a description of Number objects.

// Return TRUE.

// String Create a new String object whose [[PrimitiveValue]]

// internal property is set to the value of the string.

// See 15.5 for a description of String objects.

// Return TRUE.

// Object The result is the input argument (no conversion).

// Return TRUE.

//----------------------------------------------------------------------------

BOOL JavascriptConversion::ToObject(Var aValue, ScriptContext\* scriptContext, RecyclableObject\*\* object)

{

Assert(object);

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Undefined:

case TypeIds\_Null:

return FALSE;

case TypeIds\_Number:

case TypeIds\_Integer:

\*object = scriptContext->GetLibrary()->CreateNumberObject(aValue);

return TRUE;

default:

{

\*object = RecyclableObject::FromVar(aValue)->ToObject(scriptContext);

return TRUE;

}

}

}

//----------------------------------------------------------------------------

// ToPropertyKey() takes a value and converts it to a property key

// Implementation of ES6 7.1.14

//----------------------------------------------------------------------------

BOOL JavascriptConversion::ToPropertyKey(Var argument, ScriptContext\* scriptContext, const PropertyRecord\*\* propertyRecord)

{

Var key = JavascriptConversion::ToPrimitive(argument, JavascriptHint::HintString, scriptContext);

if (JavascriptSymbol::Is(key))

{

// If we are looking up a property keyed by a symbol, we already have the PropertyId in the symbol

\*propertyRecord = JavascriptSymbol::FromVar(key)->GetValue();

}

else

{

// For all other types, convert the key into a string and use that as the property name

JavascriptString \* propName = JavascriptConversion::ToString(key, scriptContext);

if (VirtualTableInfo<Js::PropertyString>::HasVirtualTable(propName))

{

PropertyString \* propertyString = (PropertyString \*)propName;

\*propertyRecord = propertyString->GetPropertyRecord();

}

else

{

scriptContext->GetOrAddPropertyRecord(propName->GetString(), propName->GetLength(), propertyRecord);

}

}

return TRUE;

}

//----------------------------------------------------------------------------

// ToPrimitive() takes a value and an optional argument and converts it to a non Object type

// Implementation of ES5 9.1

//

// Undefined:The result equals the input argument (no conversion).

// Null: The result equals the input argument (no conversion).

// Boolean: The result equals the input argument (no conversion).

// Number: The result equals the input argument (no conversion).

// String: The result equals the input argument (no conversion).

// Object: Return a default value for the Object.

// The default value of an object is retrieved by calling the [[DefaultValue]]

// internal method of the object, passing the optional hint PreferredType.

// The behavior of the [[DefaultValue]] internal method is defined by this specification

// for all native ECMAScript objects (8.12.9).

//----------------------------------------------------------------------------

Var JavascriptConversion::ToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* requestContext)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_Integer:

case TypeIds\_Boolean:

case TypeIds\_Number:

case TypeIds\_String:

case TypeIds\_Symbol:

return aValue;

case TypeIds\_StringObject:

{

JavascriptStringObject \* stringObject = JavascriptStringObject::FromVar(aValue);

if (stringObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & (hint == JavascriptHint::HintString ? SideEffects\_ToString : SideEffects\_ValueOf))

{

return MethodCallToPrimitive(aValue, hint, requestContext);

}

return CrossSite::MarshalVar(requestContext, stringObject->Unwrap());

}

case TypeIds\_NumberObject:

{

JavascriptNumberObject \* numberObject = JavascriptNumberObject::FromVar(aValue);

if (hint == JavascriptHint::HintString)

{

if (numberObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & SideEffects\_ToString)

{

return MethodCallToPrimitive(aValue, hint, requestContext);

}

return JavascriptNumber::ToStringRadix10(numberObject->GetValue(), requestContext);

}

else

{

if (numberObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & SideEffects\_ValueOf)

{

return MethodCallToPrimitive(aValue, hint, requestContext);

}

return CrossSite::MarshalVar(requestContext, numberObject->Unwrap());

}

}

case TypeIds\_SymbolObject:

{

JavascriptSymbolObject\* symbolObject = JavascriptSymbolObject::FromVar(aValue);

return requestContext->GetLibrary()->CreateSymbol(symbolObject->GetValue());

}

case TypeIds\_Date:

case TypeIds\_WinRTDate:

{

JavascriptDate\* dateObject = JavascriptDate::FromVar(aValue);

if(hint == JavascriptHint::HintNumber)

{

if (dateObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & SideEffects\_ValueOf)

{

// if no Method exists this function falls back to OrdinaryToPrimitive

// if IsES6ToPrimitiveEnabled flag is off we also fall back to OrdinaryToPrimitive

return MethodCallToPrimitive(aValue, hint, requestContext);

}

return JavascriptNumber::ToVarNoCheck(dateObject->GetTime(), requestContext);

}

else

{

if (dateObject->GetScriptContext()->optimizationOverrides.GetSideEffects() & SideEffects\_ToString)

{

// if no Method exists this function falls back to OrdinaryToPrimitive

// if IsES6ToPrimitiveEnabled flag is off we also fall back to OrdinaryToPrimitive

return MethodCallToPrimitive(aValue, hint, requestContext);

}

//NOTE: Consider passing requestContext to JavascriptDate::ToString

return CrossSite::MarshalVar(requestContext, JavascriptDate::ToString(dateObject));

}

}

// convert to JavascriptNumber

case TypeIds\_Int64Number:

return JavascriptInt64Number::FromVar(aValue)->ToJavascriptNumber();

case TypeIds\_UInt64Number:

return JavascriptUInt64Number::FromVar(aValue)->ToJavascriptNumber();

default:

// if no Method exists this function falls back to OrdinaryToPrimitive

// if IsES6ToPrimitiveEnabled flag is off we also fall back to OrdinaryToPrimitive

return MethodCallToPrimitive(aValue, hint, requestContext);

}

}

//----------------------------------------------------------------------------

//7.1.16 CanonicalNumericIndexString(argument)

//1. Assert : Type(argument) is String.

//2. If argument is "-0", then return -0.

//3. Let n be ToNumber(argument).

//4. If SameValue(ToString(n), argument) is false, then return undefined.

//5. Return n.

//----------------------------------------------------------------------------

BOOL JavascriptConversion::CanonicalNumericIndexString(Var aValue, double \*indexValue, ScriptContext \* scriptContext)

{

AssertMsg(JavascriptString::Is(aValue), "CanonicalNumericIndexString expects only string");

if (JavascriptString::IsNegZero(JavascriptString::FromVar(aValue)))

{

\*indexValue = -0;

return TRUE;

}

Var indexNumberValue = JavascriptOperators::ToNumber(aValue, scriptContext);

if (JavascriptString::Equals(JavascriptConversion::ToString(indexNumberValue, scriptContext), aValue))

{

\*indexValue = JavascriptNumber::GetValue(indexNumberValue);

return TRUE;

}

return FALSE;

}

Var JavascriptConversion::MethodCallToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* requestContext)

{

Var result = nullptr;

RecyclableObject \*const recyclableObject = RecyclableObject::FromVar(aValue);

ScriptContext \*const scriptContext = recyclableObject->GetScriptContext();

/\*7.3.7 GetMethod (O, P)

The abstract operation GetMethod is used to get the value of a specific property of an object when the value of the property is expected to be a function.

The operation is called with arguments O and P where O is the object, P is the property key. This abstract operation performs the following steps:

Assert: Type(O) is Object.

Assert: IsPropertyKey(P) is true.

Let func be the result of calling the [[Get]] internal method of O passing P and O as the arguments.

ReturnIfAbrupt(func).

If func is undefined, then return undefined.

If IsCallable(func) is false, then throw a TypeError exception.

Return func.\*/

Var varMethod;

if (!(requestContext->GetConfig()->IsES6ToPrimitiveEnabled()

&& JavascriptOperators::GetPropertyReference(recyclableObject, PropertyIds::\_symbolToPrimitive, &varMethod, requestContext)

&& !JavascriptOperators::IsUndefinedObject(varMethod)))

{

return OrdinaryToPrimitive(aValue, hint, requestContext);

}

if (!JavascriptFunction::Is(varMethod))

{

// Don't error if we disabled implicit calls

JavascriptError::TryThrowTypeError(scriptContext, requestContext, JSERR\_NeedFunction, requestContext->GetPropertyName(PropertyIds::\_symbolToPrimitive)->GetBuffer());

return requestContext->GetLibrary()->GetNull();

}

// Let exoticToPrim be GetMethod(input, @@toPrimitive).

JavascriptFunction\* exoticToPrim = JavascriptFunction::FromVar(varMethod);

JavascriptString\* hintString = nullptr;

if (hint == JavascriptHint::HintString)

{

hintString = requestContext->GetLibrary()->CreateStringFromCppLiteral(L"string");

}

else if (hint == JavascriptHint::HintNumber)

{

hintString = requestContext->GetLibrary()->CreateStringFromCppLiteral(L"number");

}

else

{

hintString = requestContext->GetLibrary()->CreateStringFromCppLiteral(L"default");

}

// If exoticToPrim is not undefined, then

if (nullptr != exoticToPrim)

{

ThreadContext \* threadContext = requestContext->GetThreadContext();

result = threadContext->ExecuteImplicitCall(exoticToPrim, ImplicitCall\_ToPrimitive, [=]()->Js::Var

{

// Stack object should have a pre-op bail on implicit call. We shouldn't see them here.

Assert(!ThreadContext::IsOnStack(recyclableObject));

// Let result be the result of calling the[[Call]] internal method of exoticToPrim, with input as thisArgument and(hint) as argumentsList.

return exoticToPrim->GetEntryPoint()(exoticToPrim, CallInfo(CallFlags\_Value, 2), recyclableObject, hintString);

});

Assert(!CrossSite::NeedMarshalVar(result, requestContext));

if (!result)

{

// There was an implicit call and implicit calls are disabled. This would typically cause a bailout.

Assert(threadContext->IsDisableImplicitCall());

return requestContext->GetLibrary()->GetNull();

}

}

// If result is an ECMAScript language value and Type(result) is not Object, then return result.

if (TaggedInt::Is(result) || JavascriptOperators::IsExposedType(JavascriptOperators::GetTypeId(result)))

{

return result;

}

// Else, throw a TypeError exception.

else

{

// Don't error if we disabled implicit calls

JavascriptError::TryThrowTypeError(scriptContext, requestContext, JSERR\_FunctionArgument\_Invalid, L"[Symbol.toPrimitive]");

return requestContext->GetLibrary()->GetNull();

}

}

Var JavascriptConversion::OrdinaryToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* requestContext)

{

Var result;

RecyclableObject \*const recyclableObject = RecyclableObject::FromVar(aValue);

if (!recyclableObject->ToPrimitive(hint, &result, requestContext))

{

ScriptContext \*const scriptContext = recyclableObject->GetScriptContext();

long hCode;

switch (hint)

{

case JavascriptHint::HintNumber:

hCode = JSERR\_NeedNumber;

break;

case JavascriptHint::HintString:

hCode = JSERR\_NeedString;

break;

default:

hCode = VBSERR\_OLENoPropOrMethod;

break;

}

JavascriptError::TryThrowTypeError(scriptContext, scriptContext, hCode);

return requestContext->GetLibrary()->GetNull();

}

return result;

}

JavascriptString \*JavascriptConversion::CoerseString(Var aValue, ScriptContext\* scriptContext, const wchar\_t\* apiNameForErrorMsg)

{

if (!JavascriptConversion::CheckObjectCoercible(aValue, scriptContext))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_This\_NullOrUndefined, apiNameForErrorMsg);

}

return ToString(aValue, scriptContext);

}

//----------------------------------------------------------------------------

// ToString - abstract operation

// ES5 9.8

//Input Type Result

// Undefined

// "undefined"

// Null

// "null"

// Boolean

// If the argument is true, then the result is "true". If the argument is false, then the result is "false".

// Number

// See 9.8.1 below.

// String

// Return the input argument (no conversion)

// Object

// Apply the following steps:

// 1. Let primValue be ToPrimitive(input argument, hint String).

// 2. Return ToString(primValue).

//----------------------------------------------------------------------------

JavascriptString \*JavascriptConversion::ToString(Var aValue, ScriptContext\* scriptContext)

{

Assert(scriptContext->GetThreadContext()->IsScriptActive());

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Undefined:

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

case TypeIds\_Null:

return scriptContext->GetLibrary()->GetNullDisplayString();

case TypeIds\_Integer:

return scriptContext->GetIntegerString(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? scriptContext->GetLibrary()->GetTrueDisplayString() : scriptContext->GetLibrary()->GetFalseDisplayString();

case TypeIds\_Number:

return JavascriptNumber::ToStringRadix10(JavascriptNumber::GetValue(aValue), scriptContext);

case TypeIds\_Int64Number:

{

\_\_int64 value = JavascriptInt64Number::FromVar(aValue)->GetValue();

if (!TaggedInt::IsOverflow(value))

{

return scriptContext->GetIntegerString((int)value);

}

else

{

return JavascriptInt64Number::ToString(aValue, scriptContext);

}

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 value = JavascriptUInt64Number::FromVar(aValue)->GetValue();

if (!TaggedInt::IsOverflow(value))

{

return scriptContext->GetIntegerString((uint)value);

}

else

{

return JavascriptUInt64Number::ToString(aValue, scriptContext);

}

}

case TypeIds\_String:

return JavascriptString::FromVar(CrossSite::MarshalVar(scriptContext, aValue));

case TypeIds\_VariantDate:

return JavascriptVariantDate::FromVar(aValue)->GetValueString(scriptContext);

case TypeIds\_Symbol:

return JavascriptSymbol::FromVar(aValue)->ToString(scriptContext);

case TypeIds\_GlobalObject:

aValue = static\_cast<Js::GlobalObject\*>(aValue)->ToThis();

// fall through

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToString");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToString, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_InternalError);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintString, scriptContext);

}

}

}

}

JavascriptString \*JavascriptConversion::ToLocaleString(Var aValue, ScriptContext\* scriptContext)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Undefined:

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

case TypeIds\_Null:

return scriptContext->GetLibrary()->GetNullDisplayString();

case TypeIds\_Integer:

return JavascriptNumber::ToLocaleString(TaggedInt::ToInt32(aValue), scriptContext);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? scriptContext->GetLibrary()->GetTrueDisplayString() : scriptContext->GetLibrary()->GetFalseDisplayString();

case TypeIds\_Int64Number:

return JavascriptNumber::ToLocaleString((double)JavascriptInt64Number::FromVar(aValue)->GetValue(), scriptContext);

case TypeIds\_UInt64Number:

return JavascriptNumber::ToLocaleString((double)JavascriptUInt64Number::FromVar(aValue)->GetValue(), scriptContext);

case TypeIds\_Number:

return JavascriptNumber::ToLocaleString(JavascriptNumber::GetValue(aValue), scriptContext);

case TypeIds\_String:

return JavascriptString::FromVar(aValue);

case TypeIds\_VariantDate:

// Legacy behavior was to create an empty object and call toLocaleString on it, which would result in this value

return scriptContext->GetLibrary()->GetObjectDisplayString();

case TypeIds\_Symbol:

return JavascriptSymbol::FromVar(aValue)->ToString(scriptContext);

default:

{

RecyclableObject\* object = RecyclableObject::FromVar(aValue);

Var value = JavascriptOperators::GetProperty(object, PropertyIds::toLocaleString, scriptContext, NULL);

if (JavascriptConversion::IsCallable(value))

{

RecyclableObject\* toLocaleStringFunction = RecyclableObject::FromVar(value);

Var aResult = toLocaleStringFunction->GetEntryPoint()(toLocaleStringFunction, 1, aValue);

if (JavascriptString::Is(aResult))

{

return JavascriptString::FromVar(aResult);

}

else

{

return JavascriptConversion::ToString(aResult, scriptContext);

}

}

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::toLocaleString)->GetBuffer());

}

}

}

//----------------------------------------------------------------------------

// ToBoolean\_Full:

// (ES3.0: S9.2):

//

// Input Output

// ----- ------

// 'undefined' 'false'

// 'null' 'false'

// Boolean Value

// Number 'false' if +0, -0, or Nan

// 'true' otherwise

// String 'false' if argument is ""

// 'true' otherwise

// Object 'true'

// Falsy Object 'false'

//----------------------------------------------------------------------------

BOOL JavascriptConversion::ToBoolean\_Full(Var aValue, ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

AssertMsg(RecyclableObject::Is(aValue), "Should be handled already");

auto type = RecyclableObject::FromVar(aValue)->GetType();

switch (type->GetTypeId())

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_VariantDate:

return false;

case TypeIds\_Symbol:

return true;

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue();

#if !FLOATVAR

case TypeIds\_Number:

{

double value = JavascriptNumber::GetValue(aValue);

return (!JavascriptNumber::IsNan(value)) && (!JavascriptNumber::IsZero(value));

}

#endif

case TypeIds\_Int64Number:

{

\_\_int64 value = JavascriptInt64Number::FromVar(aValue)->GetValue();

return value != 0;

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 value = JavascriptUInt64Number::FromVar(aValue)->GetValue();

return value != 0;

}

case TypeIds\_String:

{

JavascriptString \* pstValue = JavascriptString::FromVar(aValue);

return pstValue->GetLength() > 0;

}

case TypeIds\_SIMDFloat32x4:

case TypeIds\_SIMDFloat64x2:

case TypeIds\_SIMDInt32x4:

{ // SIMD review: may need update once spec is finalized

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return true;

}

}

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToBoolean");

// Falsy objects evaluate to false when converted to Boolean.

return !type->IsFalsy();

}

}

}

void JavascriptConversion::ToFloat\_Helper(Var aValue, float \*pResult, ScriptContext\* scriptContext)

{

\*pResult = (float)ToNumber\_Full(aValue, scriptContext);

}

void JavascriptConversion::ToNumber\_Helper(Var aValue, double \*pResult, ScriptContext\* scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

\*pResult = ToNumber\_Full(aValue, scriptContext);

}

// Used for the JIT's float type specialization

// Convert aValue to double, but only allow primitives. Return false otherwise.

BOOL JavascriptConversion::ToNumber\_FromPrimitive(Var aValue, double \*pResult, BOOL allowUndefined, ScriptContext\* scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

Assert(!TaggedNumber::Is(aValue));

RecyclableObject \*obj = RecyclableObject::FromVar(aValue);

// NOTE: Don't allow strings, otherwise JIT's float type specialization has to worry about concats

if (obj->GetTypeId() >= TypeIds\_String)

{

return false;

}

if (!allowUndefined && obj->GetTypeId() == TypeIds\_Undefined)

{

return false;

}

\*pResult = ToNumber\_Full(aValue, scriptContext);

return true;

}

//----------------------------------------------------------------------------

// ToNumber\_Full:

// Implements ES6 Draft Rev 26 July 18, 2014

//

// Undefined: NaN

// Null: 0

// boolean: v==true ? 1 : 0 ;

// number: v (original number)

// String: conversion by spec algorithm

// object: ToNumber(PrimitiveValue(v, hint\_number))

// Symbol: TypeError

//----------------------------------------------------------------------------

double JavascriptConversion::ToNumber\_Full(Var aValue,ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return NaN if exceptions are disabled

case TypeIds\_Undefined:

return JavascriptNumber::GetValue(scriptContext->GetLibrary()->GetNaN());

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToDouble(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return JavascriptNumber::GetValue(aValue);

case TypeIds\_Int64Number:

return (double)JavascriptInt64Number::FromVar(aValue)->GetValue();

case TypeIds\_UInt64Number:

return (double)JavascriptUInt64Number::FromVar(aValue)->GetValue();

case TypeIds\_String:

return JavascriptString::FromVar(aValue)->ToDouble();

case TypeIds\_VariantDate:

return Js::DateImplementation::GetTvUtc(Js::DateImplementation::JsLocalTimeFromVarDate(JavascriptVariantDate::FromVar(aValue)->GetValue()), scriptContext);

case TypeIds\_SIMDFloat32x4:

case TypeIds\_SIMDInt32x4:

case TypeIds\_SIMDFloat64x2:

JavascriptError::ThrowError(scriptContext, JSERR\_NeedNumber);

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToInteger");

if(fPrimitiveOnly)

{

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

//----------------------------------------------------------------------------

// second part of the ToInteger() implementation.(ES5.0: S9.4).

//----------------------------------------------------------------------------

double JavascriptConversion::ToInteger\_Full(Var aValue,ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToInt32(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return ToInteger(JavascriptNumber::GetValue(aValue));

case TypeIds\_Int64Number:

return ToInteger((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

return ToInteger((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

return ToInteger(JavascriptString::FromVar(aValue)->ToDouble());

case TypeIds\_VariantDate:

return ToInteger(ToNumber\_Full(aValue, scriptContext));

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToInteger");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToInteger\_Full, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

double JavascriptConversion::ToInteger(double val)

{

if(JavascriptNumber::IsNan(val))

return 0;

if(JavascriptNumber::IsPosInf(val) || JavascriptNumber::IsNegInf(val) ||

JavascriptNumber::IsZero(val))

{

return val;

}

return ( ((val < 0) ? -1 : 1 ) \* floor(fabs(val)));

}

//----------------------------------------------------------------------------

// ToInt32() converts the given Var to an Int32 value, as described in

// (ES3.0: S9.5).

//----------------------------------------------------------------------------

int32 JavascriptConversion::ToInt32\_Full(Var aValue, ScriptContext\* scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

// This is used when TaggedInt's overflow but remain under int32

// so Number is our most critical case:

TypeId typeId = JavascriptOperators::GetTypeId(aValue);

if (typeId == TypeIds\_Number)

{

return JavascriptMath::ToInt32Core(JavascriptNumber::GetValue(aValue));

}

switch (typeId)

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToInt32(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToInt32Core((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToInt32Core((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

{

double result;

if (JavascriptString::FromVar(aValue)->ToDouble(&result))

{

return JavascriptMath::ToInt32Core(result);

}

// If the string isn't a valid number, ToDouble returns NaN, and ToInt32 of that is 0

return 0;

}

case TypeIds\_VariantDate:

return ToInt32(ToNumber\_Full(aValue, scriptContext));

default:

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToInteger32");

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToInt32(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return ToInt32(JavascriptNumber::GetValue(aValue));

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToInt32Core((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToInt32Core((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

{

double result;

if (JavascriptString::FromVar(aValue)->ToDouble(&result))

{

return ToInt32(result);

}

// If the string isn't a valid number, ToDouble returns NaN, and ToInt32 of that is 0

return 0;

}

case TypeIds\_VariantDate:

return ToInt32(ToNumber\_Full(aValue, scriptContext));

default:

AssertMsg(FALSE, "wrong call in ToInteger32\_Full, no dynamic objects should get here.");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

}

// a strict version of ToInt32 conversion that returns false for non int32 values like, inf, NaN, undef

BOOL JavascriptConversion::ToInt32Finite(Var aValue, ScriptContext\* scriptContext, int32\* result)

{

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return false and set result to 0 if exceptions are disabled

case TypeIds\_Undefined:

\*result = 0;

return false;

case TypeIds\_Null:

\*result = 0;

return true;

case TypeIds\_Integer:

\*result = TaggedInt::ToInt32(aValue);

return true;

case TypeIds\_Boolean:

\*result = JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

return true;

case TypeIds\_Number:

return ToInt32Finite(JavascriptNumber::GetValue(aValue), result);

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return ToInt32Finite((double)JavascriptInt64Number::FromVar(aValue)->GetValue(), result);

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return ToInt32Finite((double)JavascriptUInt64Number::FromVar(aValue)->GetValue(), result);

case TypeIds\_String:

return ToInt32Finite(JavascriptString::FromVar(aValue)->ToDouble(), result);

case TypeIds\_VariantDate:

return ToInt32Finite(ToNumber\_Full(aValue, scriptContext), result);

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToInteger32");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToInteger32\_Full, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

int32 JavascriptConversion::ToInt32(double T1)

{

return JavascriptMath::ToInt32Core(T1);

}

\_\_int64 JavascriptConversion::ToInt64(Var aValue, ScriptContext\* scriptContext)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Integer:

{

return TaggedInt::ToInt32(aValue);

}

case TypeIds\_Int64Number:

{

JavascriptInt64Number\* int64Number = JavascriptInt64Number::FromVar(aValue);

return int64Number->GetValue();

}

case TypeIds\_UInt64Number:

{

JavascriptUInt64Number\* uint64Number = JavascriptUInt64Number::FromVar(aValue);

return (\_\_int64)uint64Number->GetValue();

}

case TypeIds\_Number:

return JavascriptMath::TryToInt64(JavascriptNumber::GetValue(aValue));

default:

return (unsigned \_\_int64)JavascriptConversion::ToInt32\_Full(aValue, scriptContext);

}

}

unsigned \_\_int64 JavascriptConversion::ToUInt64(Var aValue, ScriptContext\* scriptContext)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Integer:

{

return (unsigned \_\_int64)TaggedInt::ToInt32(aValue);

}

case TypeIds\_Int64Number:

{

JavascriptInt64Number\* int64Number = JavascriptInt64Number::FromVar(aValue);

return (unsigned \_\_int64)int64Number->GetValue();

}

case TypeIds\_UInt64Number:

{

JavascriptUInt64Number\* uint64Number = JavascriptUInt64Number::FromVar(aValue);

return uint64Number->GetValue();

}

case TypeIds\_Number:

return static\_cast<unsigned \_\_int64>(JavascriptMath::TryToInt64(JavascriptNumber::GetValue(aValue)));

default:

return (unsigned \_\_int64)JavascriptConversion::ToInt32\_Full(aValue, scriptContext);

}

}

BOOL JavascriptConversion::ToInt32Finite(double value, int32\* result)

{

if((!NumberUtilities::IsFinite(value)) || JavascriptNumber::IsNan(value))

{

\*result = 0;

return false;

}

else

{

\*result = JavascriptMath::ToInt32Core(value);

return true;

}

}

//----------------------------------------------------------------------------

// (ES3.0: S9.6).

//----------------------------------------------------------------------------

uint32 JavascriptConversion::ToUInt32\_Full(Var aValue, ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToUInt32(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return JavascriptMath::ToUInt32(JavascriptNumber::GetValue(aValue));

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToUInt32((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 32bit boundary; otherwise we need to

// treat it as double anyhow.

return JavascriptMath::ToUInt32((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

{

double result;

if (JavascriptString::FromVar(aValue)->ToDouble(&result))

{

return JavascriptMath::ToUInt32(result);

}

// If the string isn't a valid number, ToDouble returns NaN, and ToUInt32 of that is 0

return 0;

}

case TypeIds\_VariantDate:

return JavascriptMath::ToUInt32(ToNumber\_Full(aValue, scriptContext));

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToUInt32");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToUInt32\_Full, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

uint32 JavascriptConversion::ToUInt32(double T1)

{

// Same as doing ToInt32 and reinterpret the bits as uint32

return (uint32)JavascriptMath::ToInt32Core(T1);

}

//----------------------------------------------------------------------------

// ToUInt16() converts the given Var to a UInt16 value, as described in

// (ES3.0: S9.6).

//----------------------------------------------------------------------------

uint16 JavascriptConversion::ToUInt16\_Full(IN Var aValue, ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

ScriptContext \* objectScriptContext = RecyclableObject::Is(aValue) ? RecyclableObject::FromVar(aValue)->GetScriptContext() : nullptr;

BOOL fPrimitiveOnly = false;

while(true)

{

switch (JavascriptOperators::GetTypeId(aValue))

{

case TypeIds\_Symbol:

JavascriptError::TryThrowTypeError(objectScriptContext, scriptContext, JSERR\_NeedNumber);

// Fallthrough to return 0 if exceptions are disabled

case TypeIds\_Undefined:

case TypeIds\_Null:

return 0;

case TypeIds\_Integer:

return TaggedInt::ToUInt16(aValue);

case TypeIds\_Boolean:

return JavascriptBoolean::FromVar(aValue)->GetValue() ? 1 : +0;

case TypeIds\_Number:

return ToUInt16(JavascriptNumber::GetValue(aValue));

case TypeIds\_Int64Number:

// we won't lose precision if the int64 is within 16bit boundary; otherwise we need to

// treat it as double anyhow.

return ToUInt16((double)JavascriptInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_UInt64Number:

// we won't lose precision if the int64 is within 16bit boundary; otherwise we need to

// treat it as double anyhow.

return ToUInt16((double)JavascriptUInt64Number::FromVar(aValue)->GetValue());

case TypeIds\_String:

{

double result;

if (JavascriptString::FromVar(aValue)->ToDouble(&result))

{

return ToUInt16(result);

}

// If the string isn't a valid number, ToDouble is NaN, and ToUInt16 of that is 0

return 0;

}

case TypeIds\_VariantDate:

return ToUInt16(ToNumber\_Full(aValue, scriptContext));

default:

{

AssertMsg(JavascriptOperators::IsObject(aValue), "bad type object in conversion ToUIn16");

if(fPrimitiveOnly)

{

AssertMsg(FALSE, "wrong call in ToUInt16, no dynamic objects should get here");

JavascriptError::ThrowError(scriptContext, VBSERR\_OLENoPropOrMethod);

}

fPrimitiveOnly = true;

aValue = ToPrimitive(aValue, JavascriptHint::HintNumber, scriptContext);

}

}

}

}

\_\_inline uint16 JavascriptConversion::ToUInt16(double T1)

{

//

// VC does the right thing here, if we first convert to uint32 and then to uint16

// Spec says mod should be done.

//

uint32 result = JavascriptConversion::ToUInt32(T1);

#if defined(\_M\_IX86) && \_MSC\_FULL\_VER < 160030329

// Well VC doesn't actually do the right thing... It takes (uint16)(uint32)double and removes the

// middle uint32 cast to (uint16)double, which isn't the same thing. Somehow, it only seems to be a

// problem for x86. Forcing a store to uint32 prevents the incorrect optimization.

//

// A bug has been filled in the Dev11 database: TF bug id #901495

// Fixed in compiler 16.00.30329.00

volatile uint32 volResult = result;

#endif

return (uint16) result;

}

JavascriptString \* JavascriptConversion::ToPrimitiveString(Var aValue, ScriptContext \* scriptContext)

{

return ToString(ToPrimitive(aValue, JavascriptHint::None, scriptContext), scriptContext);

}

int64 JavascriptConversion::ToLength(Var aValue, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aValue))

{

int64 length = TaggedInt::ToInt64(aValue);

return (length < 0) ? 0 : length;

}

double length = JavascriptConversion::ToInteger(aValue, scriptContext);

if (length < 0.0 || JavascriptNumber::IsNegZero(length))

{

length = 0.0;

}

else if (length > Math::MAX\_SAFE\_INTEGER)

{

length = Math::MAX\_SAFE\_INTEGER;

}

return NumberUtilities::TryToInt64(length);

}

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

typedef int BOOL;

namespace Js {

class JavascriptConversion /\* All static \*/

{

public:

static Var OrdinaryToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* scriptContext);

static Var MethodCallToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* scriptContext);

static Var ToPrimitive(Var aValue, JavascriptHint hint, ScriptContext \* scriptContext);

static BOOL CanonicalNumericIndexString(Var aValue, double \*indexValue, ScriptContext \* scriptContext);

static BOOL ToPropertyKey(Var argument, ScriptContext\* scriptContext, const PropertyRecord\*\* propertyRecord);

static JavascriptString\* ToString(Var aValue, ScriptContext\* scriptContext);

static JavascriptString\* ToLocaleString(Var aValue, ScriptContext\* scriptContext);

static BOOL ToObject(Var aValue, ScriptContext\* scriptContext, RecyclableObject\*\* object);

static BOOL ToBoolean(Var aValue, ScriptContext\* scriptContext);

static BOOL ToBoolean\_Full(Var aValue, ScriptContext\* scriptContext);

static bool ToBool(Var aValue, ScriptContext\* scriptContext);

static double ToNumber(Var aValue, ScriptContext\* scriptContext);

static void ToFloat\_Helper(Var aValue, float \*pResult, ScriptContext\* scriptContext);

static void ToNumber\_Helper(Var aValue, double \*pResult, ScriptContext\* scriptContext);

static BOOL ToNumber\_FromPrimitive(Var aValue, double \*pResult, BOOL allowUndefined, ScriptContext\* scriptContext);

static double ToNumber\_Full(Var aValue, ScriptContext\* scriptContext);

static double ToInteger(Var aValue, ScriptContext\* scriptContext);

static double ToInteger(double value);

static double ToInteger\_Full(Var aValue, ScriptContext\* scriptContext);

static int32 ToInt32(Var aValue, ScriptContext\* scriptContext);

static \_\_int64 ToInt64(Var aValue, ScriptContext\* scriptContext);

static int32 ToInt32(double value);

static int32 ToInt32\_Full(Var aValue, ScriptContext\* scriptContext);

static int8 ToInt8(Var aValue, ScriptContext\* scriptContext);

static uint8 ToUInt8(Var aValue, ScriptContext\* scriptContext);

static uint8 ToUInt8Clamped(Var aValue, ScriptContext\* scriptContext);

static int16 ToInt16(Var aValue, ScriptContext\* scriptContext);

static float ToFloat(Var aValue, ScriptContext\* scriptContext);

static uint32 ToUInt32(Var aValue, ScriptContext\* scriptContext);

static unsigned \_\_int64 ToUInt64(Var aValue, ScriptContext\* scriptContext);

static uint32 ToUInt32(double value);

static uint32 ToUInt32\_Full(Var aValue, ScriptContext\* scriptContext);

static uint16 ToUInt16(Var aValue, ScriptContext\* scriptContext);

static uint16 ToUInt16(double value);

static uint16 ToUInt16\_Full(Var aValue, ScriptContext\* scriptContext);

static JavascriptString \*JavascriptConversion::CoerseString(Var aValue, ScriptContext\* scriptContext, const wchar\_t\* apiNameForErrorMsg);

static BOOL CheckObjectCoercible(Var aValue, ScriptContext\* scriptContext);

static bool SameValue(Var aValue, Var bValue);

static bool SameValueZero(Var aValue, Var bValue);

static bool IsCallable(Var aValue);

static BOOL ToInt32Finite(Var aValue, ScriptContext\* scriptContext, int32\* result);

// ToString(ToPrimitive(aValue), for convert to string on concat

static JavascriptString \* ToPrimitiveString(Var aValue, ScriptContext \* scriptContext);

static int64 ToLength(Var aValue, ScriptContext\* scriptContext);

private:

static BOOL ToInt32Finite(double value, int32\* result);

template<bool zero>

static bool SameValueCommon(Var aValue, Var bValue);

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#if defined(\_M\_IX86) || defined(\_M\_X64)

#include <emmintrin.h>

#endif

namespace Js {

\_\_inline BOOL JavascriptConversion::ToBoolean(Var aValue,ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aValue))

{

return aValue != reinterpret\_cast<Var>(AtomTag\_IntPtr);

}

#if FLOATVAR

else if (JavascriptNumber::Is\_NoTaggedIntCheck(aValue))

{

double value = JavascriptNumber::GetValue(aValue);

return (!JavascriptNumber::IsNan(value)) && (!JavascriptNumber::IsZero(value));

}

#endif

else

{

return ToBoolean\_Full(aValue, scriptContext);

}

}

\_\_inline bool JavascriptConversion::ToBool(Var aValue,ScriptContext\* scriptContext)

{

return !!JavascriptConversion::ToBoolean(aValue, scriptContext);

}

/// ToNumber() returns an integer value, as described in (ES3.0: S9.3).

\_\_inline double JavascriptConversion::ToNumber(Var aValue, ScriptContext\* scriptContext)

{

// Optimize for TaggedInt and double before falling back to ToNumber\_Full

if( TaggedInt::Is(aValue) )

{

return TaggedInt::ToDouble(aValue);

}

if( JavascriptNumber::Is\_NoTaggedIntCheck(aValue) )

{

return JavascriptNumber::GetValue(aValue);

}

return ToNumber\_Full(aValue, scriptContext);

}

\_\_inline double JavascriptConversion::ToInteger(Var aValue, ScriptContext\* scriptContext)

{

return

TaggedInt::Is(aValue) ?

TaggedInt::ToDouble(aValue) :

ToInteger\_Full(aValue, scriptContext);

}

\_\_inline int32 JavascriptConversion::ToInt32(Var aValue, ScriptContext\* scriptContext)

{

return

TaggedInt::Is(aValue) ?

TaggedInt::ToInt32(aValue) :

ToInt32\_Full(aValue, scriptContext);

}

\_\_inline uint32 JavascriptConversion::ToUInt32(Var aValue, ScriptContext\* scriptContext)

{

return

TaggedInt::Is(aValue) ?

TaggedInt::ToUInt32(aValue) :

ToUInt32\_Full(aValue, scriptContext);

}

\_\_inline uint16 JavascriptConversion::ToUInt16(Var aValue, ScriptContext\* scriptContext)

{

return

TaggedInt::Is(aValue) ?

(uint16) TaggedInt::ToUInt32(aValue) :

ToUInt16\_Full(aValue, scriptContext);

}

\_\_inline int8 JavascriptConversion::ToInt8(Var aValue, ScriptContext\* scriptContext)

{

return TaggedInt::Is(aValue) ?

(int8) TaggedInt::ToInt32(aValue) :

(int8) ToInt32(aValue, scriptContext);

}

\_\_inline uint8 JavascriptConversion::ToUInt8(Var aValue, ScriptContext\* scriptContext)

{

return TaggedInt::Is(aValue) ?

(uint8) TaggedInt::ToInt32(aValue) :

(uint8) ToUInt32(aValue, scriptContext);

}

\_\_inline uint8 JavascriptConversion::ToUInt8Clamped(Var aValue, ScriptContext\* scriptContext)

{

double dval;

if (TaggedInt::Is(aValue))

{

int32 val = Js::TaggedInt::ToInt32(aValue);

// Values larger than 0xff should be clamped to 0xff

if (val > UINT8\_MAX)

{

return UINT8\_MAX;

}

// Negative values should be clamped to 0

if (val < 0)

{

return 0;

}

return (uint8) val;

}

else if (JavascriptOperators::GetTypeId(aValue) == TypeIds\_Number)

{

dval = JavascriptNumber::GetValue(aValue);

}

else

{

dval = JavascriptConversion::ToNumber\_Full(aValue, scriptContext);

}

// This will also cover positive infinity

// Note: This is strictly greater-than check because 254.5 rounds to 254

if (dval > 254.5)

{

return UINT8\_MAX;

}

// This will also cover negative infinity, and anything less than INT\_MIN

if (dval < 0)

{

return 0;

}

// We now have a double value which is between 0 and 255 and just need to convert it

// to an integer following IEEE 754 rounding rules which round ties to the nearest

// even integer.

#if defined(\_M\_IX86) || defined(\_M\_X64)

if (AutoSystemInfo::Data.SSE2Available())

{

// On x86 we have a convenient CVTSD2SI intrinsic function to handle this.

\_\_m128d t = \_mm\_load\_sd(&dval);

return (uint8)\_mm\_cvtsd\_si32(t);

}

else

#endif

{

// On ARM, there is not a convenient intrinsic (for VCVTRS32F64).

// Once DevDiv TFS item 656383 is complete, we should replace the below with the intrinsic.

// 1. Calculate the fractional part of the double value

// 2. Round up or down as usual if the fractional part is <> 0.5

// 3. If the fractional part == 0.5, round to nearest even integer:

// Divide by 2, add 0.5, cast to integer, multiply by 2 again.

uint8 u8 = (uint8)dval;

double frac = dval - u8;

if (frac > 0.5)

{

return (uint8)(dval + 0.5);

}

else if (frac < 0.5)

{

return u8;

}

else

{

return ((uint8)(dval / 2.0 + 0.5)) \* 2;

}

}

}

\_\_inline int16 JavascriptConversion::ToInt16(Var aValue, ScriptContext\* scriptContext)

{

return TaggedInt::Is(aValue) ?

(int16) TaggedInt::ToInt32(aValue) :

(int16) ToUInt32(aValue, scriptContext);

}

\_\_inline float JavascriptConversion::ToFloat(Var aValue, ScriptContext\* scriptContext)

{

return (float)ToNumber(aValue, scriptContext);

}

\_\_inline bool JavascriptConversion::SameValue(Var aValue, Var bValue)

{

return SameValueCommon<false>(aValue, bValue);

}

\_\_inline bool JavascriptConversion::SameValueZero(Var aValue, Var bValue)

{

return SameValueCommon<true>(aValue, bValue);

}

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

class JavascriptExceptionContext

{

public:

struct StackFrame

{

private:

// Real script frames: functionBody, byteCodeOffset

// Native library builtin (or potentially virtual) frames: name

FunctionBody\* functionBody;

union

{

uint32 byteCodeOffset; // used for script functions (functionBody != nullptr)

PCWSTR name; // used for native/virtual frames (functionBody == nullptr)

};

StackTraceArguments argumentTypes;

public:

StackFrame() {}

StackFrame(JavascriptFunction\* func, const JavascriptStackWalker& walker, bool initArgumentTypes);

bool IsScriptFunction() const;

FunctionBody\* GetFunctionBody() const;

uint32 GetByteCodeOffset() const { return byteCodeOffset; }

LPCWSTR GetFunctionName() const;

HRESULT GetFunctionNameWithArguments(\_In\_ LPCWSTR \*outResult) const;

};

typedef JsUtil::List<StackFrame> StackTrace;

public:

JavascriptExceptionContext() :

m\_throwingFunction(nullptr),

m\_throwingFunctionByteCodeOffset(0),

m\_stackTrace(nullptr),

m\_originalStackTrace(nullptr)

{

}

JavascriptFunction\* ThrowingFunction() const { return m\_throwingFunction; }

uint32 ThrowingFunctionByteCodeOffset() const { return m\_throwingFunctionByteCodeOffset; }

void SetThrowingFunction(JavascriptFunction\* function, uint32 byteCodeOffset, void \* returnAddress);

bool HasStackTrace() const { return m\_stackTrace && m\_stackTrace->Count() > 0; }

StackTrace\* GetStackTrace() const { return m\_stackTrace; }

void SetStackTrace(StackTrace \*stackTrace) { m\_stackTrace = stackTrace; }

void SetOriginalStackTrace(StackTrace \*stackTrace) { Assert(m\_originalStackTrace == nullptr); m\_originalStackTrace = stackTrace; }

StackTrace\* GetOriginalStackTrace() const { return m\_originalStackTrace; }

private:

JavascriptFunction\* m\_throwingFunction;

uint32 m\_throwingFunctionByteCodeOffset;

StackTrace \*m\_stackTrace;

StackTrace \*m\_originalStackTrace;

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Library\StackScriptFunction.h"

namespace Js

{

void JavascriptExceptionObject::FillError(Js::JavascriptExceptionContext& exceptionContext, ScriptContext \*scriptContext, HostWrapperCreateFuncType hostWrapperCreateFunc)

{

this->scriptContext = scriptContext;

this->exceptionContext = exceptionContext;

this->SetHostWrapperCreateFunc(hostWrapperCreateFunc);

}

void JavascriptExceptionObject::ClearError()

{

Assert(this->isPendingExceptionObject);

memset(this, 0, sizeof(JavascriptExceptionObject));

this->isPendingExceptionObject = true;

}

JavascriptExceptionObject\* JavascriptExceptionObject::CloneIfStaticExceptionObject(ScriptContext\* scriptContext)

{

Assert(scriptContext);

ThreadContext \*threadContext = scriptContext->GetThreadContext();

JavascriptExceptionObject\* exceptionObject = this;

if (this == threadContext->GetPendingOOMErrorObject())

{

AssertMsg(this->thrownObject == NULL, "ThrownObject should be NULL since at time of OOM we will not be able to allocate the thrownObject");

// Let's hope that unwinding has released enough pointers that the

// recycler will find some memory to allocate the real OutOfMemory object.

// If not, it will rethrow outOfMemory

Var thrownObject = scriptContext->GetLibrary()->CreateOutOfMemoryError();

exceptionObject = RecyclerNew(scriptContext->GetRecycler(),

JavascriptExceptionObject,

thrownObject,

scriptContext,

&this->exceptionContext);

threadContext->ClearPendingOOMError();

}

if (this == threadContext->GetPendingSOErrorObject())

{

Var thrownObject = NULL;

if (this->thrownObject == NULL)

{

AssertMsg(!threadContext->GetIsThreadBound(), "ThrownObject could be NULL for Jsrt scenarios because it is cleared in ~EnterScriptEnd. For non-jsrt cases, we should always have an allocated thrown object.");

thrownObject = scriptContext->GetLibrary()->CreateStackOverflowError();

}

else

{

thrownObject = this->GetThrownObject(scriptContext);

}

exceptionObject = RecyclerNew(scriptContext->GetRecycler(),

JavascriptExceptionObject,

thrownObject,

scriptContext,

&this->exceptionContext);

threadContext->ClearPendingSOError();

}

return exceptionObject;

}

// Returns NULL if the exception object is the static out of memory object.

Var JavascriptExceptionObject::GetThrownObject(ScriptContext \* requestingScriptContext)

{

// requestingScriptContext == this->scriptContext when we have A->(cross site thunk)B->(IDispatch)A using and nested A window return

// exception backup. we can go back down to normal code path below.

if (requestingScriptContext != nullptr && hostWrapperCreateFunc != nullptr && (requestingScriptContext != this->scriptContext))

{

return hostWrapperCreateFunc(thrownObject, scriptContext, requestingScriptContext);

}

// We can have cross script context throw in both fastDOM and IE8 mode now.

if (requestingScriptContext && (thrownObject != nullptr))

{

Var rethrownObject = CrossSite::MarshalVar(requestingScriptContext, thrownObject);

// For now, there is no known host for which we need to support cross-domain

// scenario for JSRT. So skip the cross domain check for now.

if (!(scriptContext->GetThreadContext()->GetIsThreadBound()))

{

return rethrownObject;

}

if (rethrownObject)

{

if (JavascriptError::Is(rethrownObject))

{

JavascriptError\* jsErrorObject = JavascriptError::FromVar(rethrownObject);

if (jsErrorObject->GetScriptContext() != requestingScriptContext )

{

Assert(requestingScriptContext->GetHostScriptContext());

HRESULT hr = requestingScriptContext->GetHostScriptContext()->CheckCrossDomainScriptContext(jsErrorObject->GetScriptContext());

if ( S\_OK != hr )

{

JavascriptError\* jsNewErrorObject = requestingScriptContext->GetLibrary()->CreateTypeError();

JavascriptError::SetErrorMessage(jsNewErrorObject, VBSERR\_PermissionDenied, nullptr, requestingScriptContext);

return jsNewErrorObject;

}

}

}

else

{

if (RecyclableObject::Is(rethrownObject))

{

if (((RecyclableObject\*)rethrownObject)->GetScriptContext() != requestingScriptContext)

{

Assert(requestingScriptContext->GetHostScriptContext());

HRESULT hrSecurityCheck = requestingScriptContext->GetHostScriptContext()->CheckCrossDomainScriptContext(((RecyclableObject\*)rethrownObject)->GetScriptContext());

if (hrSecurityCheck != S\_OK)

{

AssertMsg(hrSecurityCheck != E\_ACCESSDENIED, "Invalid cross domain throw. HRESULT must either be S\_OK or !E\_ACCESSDENIED.");

// DOM should not throw cross domain object at all. This is defend in depth that we'll return something in requestScriptContext if they do throw

// something bad.

return requestingScriptContext->GetLibrary()->GetUndefined();

}

}

}

}

}

return rethrownObject;

}

return thrownObject;

}

FunctionBody\* JavascriptExceptionObject::GetFunctionBody() const

{

// If it is a throwing function; it must be deserialized

return exceptionContext.ThrowingFunction() ? exceptionContext.ThrowingFunction()->GetFunctionBody() : NULL;

}

JavascriptExceptionContext::StackFrame::StackFrame(JavascriptFunction\* func, const JavascriptStackWalker& walker, bool initArgumentTypes)

{

this->functionBody = func->GetFunctionBody();

if (this->functionBody)

{

this->byteCodeOffset = walker.GetByteCodeOffset();

}

else

{

this->name = walker.GetCurrentNativeLibraryEntryName();

}

if (this->functionBody && initArgumentTypes)

{

this->argumentTypes.Init(walker);

}

}

bool JavascriptExceptionContext::StackFrame::IsScriptFunction() const

{

return functionBody != nullptr;

}

// Get function body -- available for script functions, null for native library builtin functions.

FunctionBody\* JavascriptExceptionContext::StackFrame::GetFunctionBody() const

{

return functionBody;

}

LPCWSTR JavascriptExceptionContext::StackFrame::GetFunctionName() const

{

return IsScriptFunction() ?

GetFunctionBody()->GetExternalDisplayName() : this->name;

}

// Get function name with arguments info. Used by script WER.

HRESULT JavascriptExceptionContext::StackFrame::GetFunctionNameWithArguments(\_In\_ LPCWSTR \*outResult) const

{

PCWSTR name = GetFunctionName();

HRESULT hr = S\_OK;

if (IsScriptFunction())

{

hr = argumentTypes.ToString(name, functionBody->GetScriptContext(), outResult);

}

else

{

\*outResult = name;

}

return hr;

}

void JavascriptExceptionContext::SetThrowingFunction(JavascriptFunction \* function, uint32 byteCodeOffset, void \* returnAddress)

{

// Unfortunately, window.onerror can ask for argument.callee.caller

// and we will return the thrown function, but the stack already unwound.

// We will need to just box the function

m\_throwingFunction = StackScriptFunction::EnsureBoxed(BOX\_PARAM(function, returnAddress, L"throw"));

m\_throwingFunctionByteCodeOffset = byteCodeOffset;

}

#if DBG

void JavascriptExceptionObject::FillStackBackTrace()

{

// Note: this->scriptContext can be NULL when we throw Out Of Memory exception.

if (this->stackBackTrace == NULL && this->scriptContext != NULL)

{

Recycler\* recycler = scriptContext->GetThreadContext()->GetRecycler();

HRESULT hr = NOERROR;

BEGIN\_TRANSLATE\_OOM\_TO\_HRESULT\_NESTED

{

this->stackBackTrace = StackBackTrace::Capture(recycler, JavascriptExceptionObject::StackToSkip, JavascriptExceptionObject::StackTraceDepth);

}

END\_TRANSLATE\_OOM\_TO\_HRESULT(hr)

}

}

#endif

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

const DWORD ExceptionCode = ('jsc' | 0xE0000000);

// As magic numbers increase, we have to keep track of the versions that we are

// backwards compatible with.

// Old CRTs also recognize unknown magic numbers with a >= test. Therefore, we just increment the

// the magic number by one every time we need another.

//

const DWORD ExceptionParameters = 1;

const int ExceptionObjectIndex = 0;

class JavascriptExceptionContext;

class JavascriptExceptionObject

{

public:

typedef Var (\_\_stdcall \*HostWrapperCreateFuncType)(Var var, ScriptContext \* sourceScriptContext, ScriptContext \* destScriptContext);

JavascriptExceptionObject(Var object, ScriptContext \* scriptContext, JavascriptExceptionContext\* exceptionContextIn, bool isPendingExceptionObject = false) :

thrownObject(object), isPendingExceptionObject(isPendingExceptionObject),

scriptContext(scriptContext), tag(true), isDebuggerSkip(false), byteCodeOffsetAfterDebuggerSkip(Constants::InvalidByteCodeOffset), hasDebuggerLogged(false),

isFirstChance(false), isExceptionCaughtInNonUserCode(false), ignoreAdvanceToNextStatement(false), hostWrapperCreateFunc(nullptr), isGeneratorReturnException(false)

{

if (exceptionContextIn)

{

exceptionContext = \*exceptionContextIn;

}

else

{

memset(&exceptionContext, 0, sizeof(exceptionContext));

}

#if DBG

this->stackBackTrace = nullptr;

#endif

}

Var GetThrownObject(ScriptContext \* requestingScriptContext);

// ScriptContext can be NULL in case of OOM exception.

ScriptContext \* JavascriptExceptionObject::GetScriptContext() const

{

return scriptContext;

}

FunctionBody \* GetFunctionBody() const;

JavascriptFunction\* GetFunction() const

{

return exceptionContext.ThrowingFunction();

}

const JavascriptExceptionContext\* GetExceptionContext() const

{

return &exceptionContext;

}

#if DBG

void FillStackBackTrace();

#endif

void FillError(JavascriptExceptionContext& exceptionContext, ScriptContext \*scriptContext, HostWrapperCreateFuncType hostWrapperCreateFunc = NULL);

void ClearError();

void SetDebuggerSkip(bool skip)

{

isDebuggerSkip = skip;

}

bool IsDebuggerSkip()

{

return isDebuggerSkip;

}

int GetByteCodeOffsetAfterDebuggerSkip()

{

return byteCodeOffsetAfterDebuggerSkip;

}

void SetByteCodeOffsetAfterDebuggerSkip(int offset)

{

byteCodeOffsetAfterDebuggerSkip = offset;

}

void SetDebuggerHasLogged(bool has)

{

hasDebuggerLogged = has;

}

bool HasDebuggerLogged()

{

return hasDebuggerLogged;

}

void SetIsFirstChance(bool is)

{

isFirstChance = is;

}

bool IsFirstChanceException()

{

return isFirstChance;

}

void SetIsExceptionCaughtInNonUserCode(bool is)

{

isExceptionCaughtInNonUserCode = is;

}

bool IsExceptionCaughtInNonUserCode()

{

return isExceptionCaughtInNonUserCode;

}

void SetHostWrapperCreateFunc(HostWrapperCreateFuncType hostWrapperCreateFunc)

{

this->hostWrapperCreateFunc = hostWrapperCreateFunc;

}

uint32 GetByteCodeOffset()

{

return exceptionContext.ThrowingFunctionByteCodeOffset();

}

void ReplaceThrownObject(Var object)

{

AssertMsg(RecyclableObject::Is(object), "Why are we replacing a non recyclable thrown object?");

AssertMsg(this->GetScriptContext() != RecyclableObject::FromVar(object)->GetScriptContext(), "If replaced thrownObject is from same context what's the need to replace?");

this->thrownObject = object;

}

void SetThrownObject(Var object)

{

// Only pending exception object and generator return exception use this API.

Assert(this->isPendingExceptionObject || this->isGeneratorReturnException);

this->thrownObject = object;

}

JavascriptExceptionObject\* JavascriptExceptionObject::CloneIfStaticExceptionObject(ScriptContext\* scriptContext);

void ClearStackTrace()

{

exceptionContext.SetStackTrace(NULL);

}

bool IsPendingExceptionObject() const { return isPendingExceptionObject; }

void SetIgnoreAdvanceToNextStatement(bool is)

{

ignoreAdvanceToNextStatement = is;

}

bool IsIgnoreAdvanceToNextStatement()

{

return ignoreAdvanceToNextStatement;

}

void SetGeneratorReturnException(bool is)

{

isGeneratorReturnException = is;

}

bool IsGeneratorReturnException()

{

// Used by the generators to throw an exception to indicate the return from generator function

return isGeneratorReturnException;

}

private:

Var thrownObject;

ScriptContext \* scriptContext;

int byteCodeOffsetAfterDebuggerSkip;

const bool tag : 1; // Tag the low bit to prevent possible GC false references

bool isPendingExceptionObject : 1;

bool isGeneratorReturnException : 1;

bool isDebuggerSkip : 1;

bool hasDebuggerLogged : 1;

bool isFirstChance : 1; // Mentions whether the current exception is a handled exception or not

bool isExceptionCaughtInNonUserCode : 1; // Mentions if in the caller chain the exception will be handled by the non-user code.

bool ignoreAdvanceToNextStatement : 1; // This will be set when user had setnext while sitting on the exception

// So the exception eating logic shouldn't try and advance to next statement again.

HostWrapperCreateFuncType hostWrapperCreateFunc;

JavascriptExceptionContext exceptionContext;

#if DBG

StackBackTrace \* stackBackTrace;

static const int StackToSkip = 2;

static const int StackTraceDepth = 30;

#endif

};

class GeneratorReturnExceptionObject : public JavascriptExceptionObject

{

public:

GeneratorReturnExceptionObject(Var object, ScriptContext \* scriptContext)

: JavascriptExceptionObject(object, scriptContext, nullptr)

{

this->SetDebuggerSkip(true);

this->SetIgnoreAdvanceToNextStatement(true);

this->SetGeneratorReturnException(true);

}

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "shlwapi.h"

#include "Language\InterpreterStackFrame.h"

#ifdef \_M\_IX86

#ifdef \_CONTROL\_FLOW\_GUARD

extern "C" PVOID \_\_guard\_check\_icall\_fptr;

#endif

#endif

namespace Js

{

void JavascriptExceptionOperators::AutoCatchHandlerExists::FetchNonUserCodeStatus(ScriptContext \* scriptContext)

{

Assert(scriptContext);

bool fFound = false;

// If the outer try catch was already in the user code, no need to go any further.

if (!m\_previousCatchHandlerToUserCodeStatus)

{

Js::JavascriptFunction\* caller;

if (JavascriptStackWalker::GetCaller(&caller, scriptContext))

{

Js::FunctionBody \*funcBody = NULL;

if (caller != NULL && (funcBody = caller->GetFunctionBody()) != NULL)

{

m\_threadContext->SetIsUserCode(funcBody->IsNonUserCode() == false);

fFound = true;

}

}

}

if (!fFound)

{

// If not successfully able to find the caller, set this catch handler belongs to the user code.

m\_threadContext->SetIsUserCode(true);

}

}

JavascriptExceptionOperators::AutoCatchHandlerExists::AutoCatchHandlerExists(ScriptContext\* scriptContext)

{

Assert(scriptContext);

m\_threadContext = scriptContext->GetThreadContext();

Assert(m\_threadContext);

m\_previousCatchHandlerExists = m\_threadContext->HasCatchHandler();

m\_threadContext->SetHasCatchHandler(TRUE);

m\_previousCatchHandlerToUserCodeStatus = m\_threadContext->IsUserCode();

if (scriptContext->IsInDebugMode())

{

FetchNonUserCodeStatus(scriptContext);

}

}

JavascriptExceptionOperators::AutoCatchHandlerExists::~AutoCatchHandlerExists()

{

m\_threadContext->SetHasCatchHandler(m\_previousCatchHandlerExists);

m\_threadContext->SetIsUserCode(m\_previousCatchHandlerToUserCodeStatus);

}

bool JavascriptExceptionOperators::CrawlStackForWER(Js::ScriptContext& scriptContext)

{

return Js::Configuration::Global.flags.WERExceptionSupport && !scriptContext.GetThreadContext()->HasCatchHandler();

}

uint64 JavascriptExceptionOperators::StackCrawlLimitOnThrow(Var thrownObject, ScriptContext& scriptContext)

{

return CrawlStackForWER(scriptContext) ? MaxStackTraceLimit : GetStackTraceLimit(thrownObject, &scriptContext);

}

#ifdef \_M\_X64

void \*JavascriptExceptionOperators::OP\_TryCatch(void \*tryAddr,

void \*catchAddr,

void \*frame,

size\_t spillSize,

size\_t argsSize,

int hasBailedOutOffset,

ScriptContext \*scriptContext)

{

void \*continuation = nullptr;

JavascriptExceptionObject \*exception = nullptr;

PROBE\_STACK(scriptContext, Constants::MinStackDefault + spillSize + argsSize);

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

continuation = amd64\_CallWithFakeFrame(tryAddr, frame, spillSize, argsSize);

}

catch (JavascriptExceptionObject \*caughtException)

{

exception = caughtException;

}

if (exception)

{

exception = exception->CloneIfStaticExceptionObject(scriptContext);

bool hasBailedOut = \*(bool\*)((char\*)frame + hasBailedOutOffset); // stack offsets are negative

if (hasBailedOut)

{

// If we have bailed out, this exception is coming from the interpreter. It should not have been caught;

// it so happens that this catch was on the stack and caught the exception.

// Re-throw!

throw exception;

}

Var exceptionObject = exception->GetThrownObject(scriptContext);

AssertMsg(exceptionObject, "Caught object is null.");

continuation = amd64\_CallWithFakeFrame(catchAddr, frame, spillSize, argsSize, exceptionObject);

}

return continuation;

}

void \*JavascriptExceptionOperators::OP\_TryFinally(void \*tryAddr,

void \*finallyAddr,

void \*frame,

size\_t spillSize,

size\_t argsSize,

ScriptContext \*scriptContext)

{

void \*tryContinuation = nullptr;

void \*finallyContinuation = nullptr;

JavascriptExceptionObject \*exception = nullptr;

PROBE\_STACK(scriptContext, Constants::MinStackDefault + spillSize + argsSize);

try

{

tryContinuation = amd64\_CallWithFakeFrame(tryAddr, frame, spillSize, argsSize);

}

catch (JavascriptExceptionObject \*caughtException)

{

exception = caughtException;

}

if (exception)

{

// Clone static exception object early in case finally block overwrites it

exception = exception->CloneIfStaticExceptionObject(scriptContext);

}

finallyContinuation = amd64\_CallWithFakeFrame(finallyAddr, frame, spillSize, argsSize);

if (finallyContinuation)

{

return finallyContinuation;

}

if (exception)

{

throw exception;

}

return tryContinuation;

}

#elif defined(\_M\_ARM32\_OR\_ARM64)

void \*JavascriptExceptionOperators::OP\_TryCatch(

void \*tryAddr,

void \*catchAddr,

void \*framePtr,

void \*localsPtr,

size\_t argsSize,

int hasBailedOutOffset,

ScriptContext \*scriptContext)

{

void \*continuation = nullptr;

JavascriptExceptionObject \*exception = nullptr;

PROBE\_STACK(scriptContext, Constants::MinStackDefault + argsSize);

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

#if defined(\_M\_ARM)

continuation = arm\_CallEhFrame(tryAddr, framePtr, localsPtr, argsSize);

#elif defined(\_M\_ARM64)

continuation = arm64\_CallEhFrame(tryAddr, framePtr, localsPtr, argsSize);

#endif

}

catch (JavascriptExceptionObject \*caughtException)

{

exception = caughtException;

}

if (exception)

{

exception = exception->CloneIfStaticExceptionObject(scriptContext);

bool hasBailedOut = \*(bool\*)((char\*)localsPtr + hasBailedOutOffset); // stack offsets are sp relative

if (hasBailedOut)

{

// If we have bailed out, this exception is coming from the interpreter. It should not have been caught;

// it so happens that this catch was on the stack and caught the exception.

// Re-throw!

throw exception;

}

Var exceptionObject = exception->GetThrownObject(scriptContext);

AssertMsg(exceptionObject, "Caught object is null.");

#if defined(\_M\_ARM)

continuation = arm\_CallCatch(catchAddr, framePtr, localsPtr, argsSize, exceptionObject);

#elif defined(\_M\_ARM64)

continuation = arm64\_CallCatch(catchAddr, framePtr, localsPtr, argsSize, exceptionObject);

#endif

}

return continuation;

}

void \*JavascriptExceptionOperators::OP\_TryFinally(

void \*tryAddr,

void \*finallyAddr,

void \*framePtr,

void \*localsPtr,

size\_t argsSize,

ScriptContext \*scriptContext)

{

void \*tryContinuation = nullptr;

void \*finallyContinuation = nullptr;

JavascriptExceptionObject \*exception = nullptr;

PROBE\_STACK(scriptContext, Constants::MinStackDefault + argsSize);

try

{

#if defined(\_M\_ARM)

tryContinuation = arm\_CallEhFrame(tryAddr, framePtr, localsPtr, argsSize);

#elif defined(\_M\_ARM64)

tryContinuation = arm64\_CallEhFrame(tryAddr, framePtr, localsPtr, argsSize);

#endif

}

catch (JavascriptExceptionObject \*caughtException)

{

exception = caughtException;

}

if (exception)

{

// Clone static exception object early in case finally block overwrites it

exception = exception->CloneIfStaticExceptionObject(scriptContext);

}

#if defined(\_M\_ARM)

finallyContinuation = arm\_CallEhFrame(finallyAddr, framePtr, localsPtr, argsSize);

#elif defined(\_M\_ARM64)

finallyContinuation = arm64\_CallEhFrame(finallyAddr, framePtr, localsPtr, argsSize);

#endif

if (finallyContinuation)

{

return finallyContinuation;

}

if (exception)

{

throw exception;

}

return tryContinuation;

}

#else

#pragma warning(push)

#pragma warning(disable:4731) // frame pointer register 'ebp' modified by inline assembly code

void\* JavascriptExceptionOperators::OP\_TryCatch(void\* tryAddr, void\* handlerAddr, void\* framePtr, int hasBailedOutOffset, ScriptContext \*scriptContext)

{

void\* continuationAddr = NULL;

Js::JavascriptExceptionObject\* pExceptionObject = NULL;

PROBE\_STACK(scriptContext, Constants::MinStackDefault);

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

// Adjust the frame pointer and call into the try.

// If the try completes without raising an exception, it will pass back the continuation address.

// Bug in compiler optimizer: try-catch can be optimized away if the try block contains \_\_asm calls into function

// that may throw. The current workaround is to add the following dummy throw to prevent this optimization.

if (!tryAddr)

{

Js::Throw::InternalError();

}

#ifdef \_M\_IX86

void \*savedEsp;

\_\_asm

{

// Save and restore the callee-saved registers around the call.

// TODO: track register kills by region and generate per-region prologs and epilogs

push esi

push edi

push ebx

// 8-byte align frame to improve floating point perf of our JIT'd code.

// Save ESP

mov ecx, esp

mov savedEsp, ecx

and esp, -8

// Set up the call target, save the current frame ptr, and adjust the frame to access

// locals in native code.

mov eax, tryAddr

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

// verify that the call target is valid

mov ebx, eax ; save call target

mov ecx, eax

call [\_\_guard\_check\_icall\_fptr]

mov eax, ebx ; restore call target

#endif

push ebp

mov ebp, framePtr

call eax

pop ebp

// The native code gives us the address where execution should continue on exit

// from the region.

mov continuationAddr, eax

// Restore ESP

mov ecx, savedEsp

mov esp, ecx

pop ebx

pop edi

pop esi

}

#else

AssertMsg(FALSE, "Unsupported native try-catch handler");

#endif

}

catch(Js::JavascriptExceptionObject \* exceptionObject)

{

pExceptionObject = exceptionObject;

}

// Let's run user catch handler code only after the stack has been unwound.

if(pExceptionObject)

{

pExceptionObject = pExceptionObject->CloneIfStaticExceptionObject(scriptContext);

bool hasBailedOut = \*(bool\*)((char\*)framePtr + hasBailedOutOffset); // stack offsets are negative

if (hasBailedOut)

{

// If we have bailed out, this exception is coming from the interpreter. It should not have been caught;

// it so happens that this catch was on the stack and caught the exception.

// Re-throw!

throw pExceptionObject;

}

Var catchObject = pExceptionObject->GetThrownObject(scriptContext);

AssertMsg(catchObject, "Caught object is NULL");

#ifdef \_M\_IX86

void \*savedEsp;

\_\_asm

{

// Save and restore the callee-saved registers around the call.

// TODO: track register kills by region and generate per-region prologs and epilogs

push esi

push edi

push ebx

// 8-byte align frame to improve floating point perf of our JIT'd code.

// Save ESP

mov ecx, esp

mov savedEsp, ecx

and esp, -8

// Set up the call target

mov ecx, handlerAddr

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

// verify that the call target is valid

mov ebx, ecx ; save call target

call [\_\_guard\_check\_icall\_fptr]

mov ecx, ebx ; restore call target

#endif

// Set up catch object, save the current frame ptr, and adjust the frame to access

// locals in native code.

mov eax, catchObject

push ebp

mov ebp, framePtr

call ecx

pop ebp

// The native code gives us the address where execution should continue on exit

// from the region.

mov continuationAddr, eax

// Restore ESP

mov ecx, savedEsp

mov esp, ecx

pop ebx

pop edi

pop esi

}

#else

AssertMsg(FALSE, "Unsupported native try-catch handler");

#endif

}

return continuationAddr;

}

void\* JavascriptExceptionOperators::OP\_TryFinally(void\* tryAddr, void\* handlerAddr, void\* framePtr, ScriptContext \*scriptContext)

{

Js::JavascriptExceptionObject\* pExceptionObject = NULL;

void\* continuationAddr = NULL;

PROBE\_STACK(scriptContext, Constants::MinStackDefault);

try

{

// Bug in compiler optimizer: try-catch can be optimized away if the try block contains \_\_asm calls into function

// that may throw. The current workaround is to add the following dummy throw to prevent this optimization.

// It seems like compiler got smart and still optimizes if the exception is not JavascriptExceptionObject (see catch handler below).

// In order to circumvent that we are throwing OutOfMemory.

if (!tryAddr)

{

Assert(false);

ThrowOutOfMemory(scriptContext);

}

#ifdef \_M\_IX86

void \*savedEsp;

\_\_asm

{

// Save and restore the callee-saved registers around the call.

// TODO: track register kills by region and generate per-region prologs and epilogs

push esi

push edi

push ebx

// 8-byte align frame to improve floating point perf of our JIT'd code.

// Save ESP

mov ecx, esp

mov savedEsp, ecx

and esp, -8

// Set up the call target, save the current frame ptr, and adjust the frame to access

// locals in native code.

mov eax, tryAddr

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

// verify that the call target is valid

mov ebx, eax ; save call target

mov ecx, eax

call [\_\_guard\_check\_icall\_fptr]

mov eax, ebx ; restore call target

#endif

push ebp

mov ebp, framePtr

call eax

pop ebp

// The native code gives us the address where execution should continue on exit

// from the region.

mov continuationAddr, eax

// Restore ESP

mov ecx, savedEsp

mov esp, ecx

pop ebx

pop edi

pop esi

}

#else

AssertMsg(FALSE, "Unsupported native try-finally handler");

#endif

}

catch(Js::JavascriptExceptionObject\* e)

{

pExceptionObject = e;

}

if (pExceptionObject)

{

// Clone static exception object early in case finally block overwrites it

pExceptionObject = pExceptionObject->CloneIfStaticExceptionObject(scriptContext);

}

void\* newContinuationAddr = NULL;

#ifdef \_M\_IX86

void \*savedEsp;

\_\_asm

{

// Save and restore the callee-saved registers around the call.

// TODO: track register kills by region and generate per-region prologs and epilogs

push esi

push edi

push ebx

// 8-byte align frame to improve floating point perf of our JIT'd code.

// Save ESP

mov ecx, esp

mov savedEsp, ecx

and esp, -8

// Set up the call target

mov eax, handlerAddr

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

// verify that the call target is valid

mov ebx, eax ; save call target

mov ecx, eax

call [\_\_guard\_check\_icall\_fptr]

mov eax, ebx ; restore call target

#endif

// save the current frame ptr, and adjust the frame to access

// locals in native code.

push ebp

mov ebp, framePtr

call eax

pop ebp

// The native code gives us the address where execution should continue on exit

// from the finally, but only if flow leaves the finally before it completes.

mov newContinuationAddr, eax

// Restore ESP

mov ecx, savedEsp

mov esp, ecx

pop ebx

pop edi

pop esi

}

#else

AssertMsg(FALSE, "Unsupported native try-finally handler");

#endif

if (newContinuationAddr != NULL)

{

// Non-null return value from the finally indicates that the finally seized the flow

// with a jump/return out of the region. Continue at that address instead of handling

// the exception.

return newContinuationAddr;

}

if (pExceptionObject)

{

throw pExceptionObject;

}

return continuationAddr;

}

#endif

void \_\_declspec(noreturn) JavascriptExceptionOperators::OP\_Throw(Var object, ScriptContext\* scriptContext)

{

Throw(object, scriptContext);

}

#if defined(DBG) && defined(\_M\_IX86)

extern "C" void \* \_except\_handler4;

void JavascriptExceptionOperators::DbgCheckEHChain()

{

#if 0

// This debug check is disabled until we figure out how to trace an fs:0 chain if we throw from inside

// a finally.

void \*currentFS0;

ThreadContext \* threadContext = ThreadContext::GetContextForCurrentThread();

if (!threadContext->IsScriptActive())

{

return;

}

// Walk the FS:0 chain of exception handlers, until the FS:0 handler in CallRootFunction.

// We should only see SEH frames on the way.

// We do allow C++ EH frames as long as there is no active objects (state = -1).

// That's because we may see frames that have calls to new(). This introduces an EH frame

// to call delete if the constructor throws. Our constructors shouldn't throw, so we should be fine.

currentFS0 = (void\*)\_\_readfsdword(0);

while (currentFS0 != threadContext->callRootFS0)

{

// EH struct:

// void \* next;

// void \* handler;

// int state;

AssertMsg(\*((void\*\*)currentFS0 + 1) == &\_except\_handler4

|| \*((int\*)currentFS0 + 2) == -1, "Found a non SEH exception frame on stack");

currentFS0 = \*(void\*\*)currentFS0;

}

#endif

}

#endif

void JavascriptExceptionOperators::Throw(Var object, ScriptContext \* scriptContext)

{

#if defined(DBG) && defined(\_M\_IX86)

DbgCheckEHChain();

#endif

Assert(scriptContext != nullptr);

// TODO: FastDOM Trampolines will throw JS Exceptions but are not isScriptActive

//AssertMsg(scriptContext->GetThreadContext()->IsScriptActive() ||

// (JavascriptError::Is(object) && (JavascriptError::FromVar(object))->IsExternalError()),

// "Javascript exception raised when script is not active");

AssertMsg(scriptContext->GetThreadContext()->IsInScript() ||

(JavascriptError::Is(object) && (JavascriptError::FromVar(object))->IsExternalError()),

"Javascript exception raised without being in CallRootFunction");

JavascriptError \*javascriptError = nullptr;

if (JavascriptError::Is(object))

{

// We keep track of the JavascriptExceptionObject that was created when this error

// was first thrown so that we can always get the correct metadata.

javascriptError = JavascriptError::FromVar(object);

JavascriptExceptionObject \*exceptionObject = javascriptError->GetJavascriptExceptionObject();

if (exceptionObject)

{

JavascriptExceptionOperators::ThrowExceptionObject(exceptionObject, scriptContext, true);

}

}

JavascriptExceptionObject \* exceptionObject =

RecyclerNew(scriptContext->GetRecycler(), JavascriptExceptionObject, object, scriptContext, NULL);

bool resetStack = false;

if (javascriptError)

{

if (!javascriptError->IsStackPropertyRedefined())

{

/\*

Throwing an error object. Original stack property will be pointing to the stack created at time of Error constructor.

Reset the stack property to match IE11 behavior

\*/

resetStack = true;

}

javascriptError->SetJavascriptExceptionObject(exceptionObject);

}

JavascriptExceptionOperators::ThrowExceptionObject(exceptionObject, scriptContext, /\*considerPassingToDebugger=\*/ true, /\*returnAddress=\*/ nullptr, resetStack);

}

void

JavascriptExceptionOperators::WalkStackForExceptionContext(ScriptContext& scriptContext, JavascriptExceptionContext& exceptionContext, Var thrownObject, uint64 stackCrawlLimit, PVOID returnAddress, bool isThrownException, bool resetSatck)

{

uint32 callerBytecodeOffset;

JavascriptFunction \* jsFunc = WalkStackForExceptionContextInternal(scriptContext, exceptionContext, thrownObject, callerBytecodeOffset, stackCrawlLimit, returnAddress, isThrownException, resetSatck);

if (jsFunc)

{

// If found, the caller is a function, and we can retrieve the debugger info from there

// otherwise it's probably just accessing property. While it is still possible to throw

// from that context, we just won't be able to get the line number etc., which make sense.

exceptionContext.SetThrowingFunction(jsFunc, callerBytecodeOffset, returnAddress);

}

}

JavascriptFunction \*

JavascriptExceptionOperators::WalkStackForExceptionContextInternal(ScriptContext& scriptContext, JavascriptExceptionContext& exceptionContext, Var thrownObject,

uint32& callerByteCodeOffset, uint64 stackCrawlLimit, PVOID returnAddress, bool isThrownException, bool resetStack)

{

JavascriptStackWalker walker(&scriptContext, true, returnAddress);

JavascriptFunction\* jsFunc = nullptr;

if (!GetCaller(walker, jsFunc))

{

return nullptr;

}

// Skip to first non-Library code

// Similar behavior to GetCaller returning false

if(jsFunc->IsLibraryCode() && !walker.GetNonLibraryCodeCaller(&jsFunc))

{

return nullptr;

}

JavascriptFunction \* caller = jsFunc;

callerByteCodeOffset = walker.GetByteCodeOffset();

Assert(!caller->IsLibraryCode());

// NOTE Don't set the throwing exception here, because we might need to box it and will cause a nested stack walker

// instead, return it to be set in WalkStackForExceptionContext

if (stackCrawlLimit == 0)

{

return caller;

}

const bool crawlStackForWER = CrawlStackForWER(scriptContext);

// If we take an OOM (JavascriptException for OOM if script is active), just bail early and return what we've got

HRESULT hr;

JavascriptExceptionContext::StackTrace \*stackTrace = NULL;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

// In WER scenario, we should combine the original stack with latest throw stack as the final throw might be coming form

// a different stack.

uint64 i = 1;

if (crawlStackForWER && thrownObject && Js::JavascriptError::Is(thrownObject))

{

Js::JavascriptError\* errorObject = Js::JavascriptError::FromVar(thrownObject);

Js::JavascriptExceptionContext::StackTrace \*originalStackTrace = NULL;

const Js::JavascriptExceptionObject\* originalExceptionObject = errorObject->GetJavascriptExceptionObject();

if (!resetStack && errorObject->GetInternalProperty(errorObject, InternalPropertyIds::StackTrace, (Js::Var\*) &originalStackTrace, NULL, &scriptContext) &&

(originalStackTrace != nullptr))

{

exceptionContext.SetOriginalStackTrace(originalStackTrace);

}

else

{

if (originalExceptionObject != nullptr)

{

exceptionContext.SetOriginalStackTrace(originalExceptionObject->GetExceptionContext()->GetStackTrace());

}

}

}

stackTrace = RecyclerNew(scriptContext.GetRecycler(), JavascriptExceptionContext::StackTrace, scriptContext.GetRecycler());

do

{

JavascriptExceptionContext::StackFrame stackFrame(jsFunc, walker, crawlStackForWER);

stackTrace->Add(stackFrame);

} while (walker.GetDisplayCaller(&jsFunc) && i++ < stackCrawlLimit);

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_INSCRIPT(hr);

exceptionContext.SetStackTrace(stackTrace);

DumpStackTrace(exceptionContext, isThrownException);

return caller;

}

// We might be trying to raise a stack overflow exception from the interpreter before

// we've executed code in the current script stack frame. In that case the current byte

// code offset is 0. In such cases walk to the caller's caller.

BOOL JavascriptExceptionOperators::GetCaller(JavascriptStackWalker& walker, JavascriptFunction\*& jsFunc)

{

if (! walker.GetCaller(&jsFunc))

{

return FALSE;

}

if (! walker.GetCurrentInterpreterFrame() ||

walker.GetCurrentInterpreterFrame()->GetReader()->GetCurrentOffset() > 0)

{

return TRUE;

}

return walker.GetCaller(&jsFunc);

}

void JavascriptExceptionOperators::DumpStackTrace(JavascriptExceptionContext& exceptionContext, bool isThrownException)

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (! exceptionContext.GetStackTrace()

|| ! Configuration::Global.flags.Dump.IsEnabled(ExceptionStackTracePhase)

|| ! isThrownException)

{

return;

}

Output::Print(L"\nStack trace for thrown exception\n");

JavascriptExceptionContext::StackTrace \*stackTrace = exceptionContext.GetStackTrace();

for (int i=0; i < stackTrace->Count(); i++)

{

Js::JavascriptExceptionContext::StackFrame currFrame = stackTrace->Item(i);

ULONG lineNumber = 0;

LONG characterPosition = 0;

if (currFrame.IsScriptFunction() && !currFrame.GetFunctionBody()->GetUtf8SourceInfo()->GetIsLibraryCode())

{

currFrame.GetFunctionBody()->GetLineCharOffset(currFrame.GetByteCodeOffset(), &lineNumber, &characterPosition);

}

Output::Print(L" %3d: %s (%d, %d)\n", i, currFrame.GetFunctionName(), lineNumber, characterPosition);

}

Output::Flush();

#endif

}

/// ---------------------------------------------------------------------------------------------------

/// When allocators throw out of memory exception - scriptContext is NULL

/// ---------------------------------------------------------------------------------------------------

JavascriptExceptionObject \* JavascriptExceptionOperators::GetOutOfMemoryExceptionObject(ScriptContext \*scriptContext)

{

ThreadContext \*threadContext = scriptContext ?

scriptContext->GetThreadContext() :

ThreadContext::GetContextForCurrentThread();

JavascriptExceptionObject \*oomExceptionObject = threadContext->GetPendingOOMErrorObject();

Assert(oomExceptionObject);

return oomExceptionObject;

}

void JavascriptExceptionOperators::ThrowOutOfMemory(ScriptContext \*scriptContext)

{

ThreadContext \*threadContext = scriptContext ?

scriptContext->GetThreadContext() :

ThreadContext::GetContextForCurrentThread();

threadContext->ClearDisableImplicitFlags();

JavascriptExceptionObject \*oom = JavascriptExceptionOperators::GetOutOfMemoryExceptionObject(scriptContext);

JavascriptExceptionOperators::ThrowExceptionObject(oom, scriptContext);

}

void JavascriptExceptionOperators::ThrowStackOverflow(ScriptContext \*scriptContext, PVOID returnAddress)

{

Assert(scriptContext);

ThreadContext \*threadContext = scriptContext->GetThreadContext();

JavascriptExceptionObject \*so = threadContext->GetPendingSOErrorObject();

Assert(so);

// Disable implicit call before calling into recycler (to prevent QueryContinue/dispose from leave script and stack overflow again)

threadContext->DisableImplicitCall();

Var thrownObject = scriptContext->GetLibrary()->CreateStackOverflowError();

so->SetThrownObject(thrownObject);

// NOTE: Do not ClearDisableImplicitFlags() here. We still need to allocate StackTrace, etc. Keep implicit call disabled till actual

// throw (ThrowExceptionObjectInternal will ClearDisableImplicitFlags before throw). If anything wrong happens in between which throws

// a new exception, the new throw will ClearDisableImplicitFlags.

JavascriptExceptionOperators::ThrowExceptionObject(so, scriptContext, false, returnAddress);

}

void JavascriptExceptionOperators::ThrowExceptionObjectInternal(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext, bool fillExceptionContext, bool considerPassingToDebugger, PVOID returnAddress, bool resetStack)

{

if (scriptContext)

{

if (fillExceptionContext)

{

Assert(exceptionObject);

JavascriptExceptionContext exceptionContext;

Var thrownObject = exceptionObject->GetThrownObject(nullptr);

WalkStackForExceptionContext(\*scriptContext, exceptionContext, thrownObject, StackCrawlLimitOnThrow(thrownObject, \*scriptContext), returnAddress, /\*isThrownException=\*/ true, resetStack);

exceptionObject->FillError(exceptionContext, scriptContext);

AddStackTraceToObject(thrownObject, exceptionContext.GetStackTrace(), \*scriptContext, /\*isThrownException=\*/ true, resetStack);

if (considerPassingToDebugger)

{

DispatchExceptionToDebugger(exceptionObject, scriptContext);

}

}

Assert(!scriptContext ||

// If we disabled implicit calls and we did record an implicit call, do not throw.

// Check your helper to see if a call recorded an implicit call that might cause an invalid value

!(

scriptContext->GetThreadContext()->IsDisableImplicitCall() &&

scriptContext->GetThreadContext()->GetImplicitCallFlags() & (~ImplicitCall\_None)

) ||

// Make sure we didn't disable exceptions

!scriptContext->GetThreadContext()->IsDisableImplicitException()

);

scriptContext->GetThreadContext()->ClearDisableImplicitFlags();

}

if (exceptionObject->IsPendingExceptionObject())

{

ThreadContext \* threadContext = scriptContext? scriptContext->GetThreadContext() : ThreadContext::GetContextForCurrentThread();

threadContext->SetHasThrownPendingException();

}

throw exceptionObject;

}

void JavascriptExceptionOperators::DispatchExceptionToDebugger(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext)

{

Assert(exceptionObject != NULL);

Assert(scriptContext != NULL);

if (scriptContext->IsInDebugMode()

&& scriptContext->GetDebugContext()->GetProbeContainer()->HasAllowedForException(exceptionObject))

{

InterpreterHaltState haltState(STOP\_EXCEPTIONTHROW, /\*executingFunction\*/nullptr);

haltState.exceptionObject = exceptionObject;

scriptContext->GetDebugContext()->GetProbeContainer()->DispatchExceptionBreakpoint(&haltState);

}

}

void JavascriptExceptionOperators::ThrowExceptionObject(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext, bool considerPassingToDebugger, PVOID returnAddress, bool resetStack)

{

ThrowExceptionObjectInternal(exceptionObject, scriptContext, true, considerPassingToDebugger, returnAddress, resetStack);

}

// The purpose of RethrowExceptionObject is to determine if ThrowExceptionObjectInternal should fill in the exception context.

//

// We pretty much always want to fill in the exception context when we throw an exception. The only case where we don't want to do it

// is if we are rethrowing and have the JavascriptExceptionObject from the previous throw with its exception context intact. If

// RethrowExceptionObject is passed a JavascriptExceptionObject with the function already there, that implies we have existing

// exception context and shouldn't step on it on the throw.

//

// RethrowExceptionObject is called only for cross-host calls. When throwing across host calls, we stash our internal JavascriptExceptionObject

// in the TLS. When we are throwing on the same thread (e.g. a throw from one frame to another), we can retrieve that stashed JavascriptExceptionObject

// from the TLS and rethrow it with its exception context intact, so we don't want to step on it. In other cases, e.g. when we throw across threads,

// we cannot retrieve the internal JavascriptExceptionObject from the TLS and have to create a new one. In this case, we need to fill the exception context.

//

void JavascriptExceptionOperators::RethrowExceptionObject(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext, bool considerPassingToDebugger)

{

ThrowExceptionObjectInternal(exceptionObject, scriptContext, ! exceptionObject->GetFunction(), considerPassingToDebugger, /\*returnAddress=\*/ nullptr, /\*resetStack=\*/ false);

}

// Trim the stack trace down to the amount specified for Error.stackTraceLimit. This happens when we do a full crawl for WER, but we only want to store the specified amount in the error object for consistency.

JavascriptExceptionContext::StackTrace\* JavascriptExceptionOperators::TrimStackTraceForThrownObject(JavascriptExceptionContext::StackTrace\* stackTraceIn, Var thrownObject, ScriptContext& scriptContext)

{

Assert(CrawlStackForWER(scriptContext)); // Don't trim if crawl for Error.stack

Assert(stackTraceIn);

int stackTraceLimit = static\_cast<int>(GetStackTraceLimit(thrownObject, &scriptContext));

Assert(stackTraceLimit == 0 || IsErrorInstance(thrownObject));

if (stackTraceIn->Count() <= stackTraceLimit)

{

return stackTraceIn;

}

JavascriptExceptionContext::StackTrace\* stackTraceTrimmed = NULL;

if (stackTraceLimit > 0)

{

HRESULT hr;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

stackTraceTrimmed = RecyclerNew(scriptContext.GetRecycler(), JavascriptExceptionContext::StackTrace, scriptContext.GetRecycler());

for (int i = 0; i < stackTraceLimit; i++)

{

stackTraceTrimmed->Add(stackTraceIn->Item(i));

}

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_INSCRIPT(hr);

}

// ignore OOM and just return what we can get

return stackTraceTrimmed;

}

//

// Check if thrownObject is instanceof Error (but not an Error prototype).

//

bool JavascriptExceptionOperators::IsErrorInstance(Var thrownObject)

{

if (thrownObject && JavascriptError::Is(thrownObject))

{

return !JavascriptError::FromVar(thrownObject)->IsPrototype();

}

if (thrownObject && RecyclableObject::Is(thrownObject))

{

RecyclableObject\* obj = RecyclableObject::FromVar(thrownObject);

while (true)

{

obj = JavascriptOperators::GetPrototype(obj);

if (JavascriptOperators::GetTypeId(obj) == TypeIds\_Null)

{

break;

}

if (JavascriptError::Is(obj))

{

return true;

}

}

}

return false;

}

void JavascriptExceptionOperators::AddStackTraceToObject(Var targetObject, JavascriptExceptionContext::StackTrace\* stackTrace, ScriptContext& scriptContext, bool isThrownException, bool resetStack)

{

if (!stackTrace || stackTrace->Count() == 0 || !scriptContext.GetConfig()->IsErrorStackTraceEnabled())

{

return;

}

if (isThrownException && CrawlStackForWER(scriptContext)) // Trim stack trace for WER

{

stackTrace = TrimStackTraceForThrownObject(stackTrace, targetObject, scriptContext);

if (!stackTrace)

{

return;

}

}

// If we still have stack trace to store and obj is a thrown exception object, obj must be an Error instance.

Assert(!isThrownException || IsErrorInstance(targetObject));

RecyclableObject\* obj = RecyclableObject::FromVar(targetObject);

if (!resetStack && obj->HasProperty(PropertyIds::stack))

{

return; // we don't want to overwrite an existing "stack" property

}

JavascriptFunction\* accessor = scriptContext.GetLibrary()->GetStackTraceAccessorFunction();

PropertyDescriptor stackPropertyDescriptor;

stackPropertyDescriptor.SetSetter(accessor);

stackPropertyDescriptor.SetGetter(accessor);

stackPropertyDescriptor.SetConfigurable(true);

stackPropertyDescriptor.SetEnumerable(false);

HRESULT hr;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

if (JavascriptOperators::DefineOwnPropertyDescriptor(obj, PropertyIds::stack, stackPropertyDescriptor, false, &scriptContext))

{

obj->SetInternalProperty(InternalPropertyIds::StackTrace, stackTrace, PropertyOperationFlags::PropertyOperation\_None, NULL);

obj->SetInternalProperty(InternalPropertyIds::StackTraceCache, NULL, PropertyOperationFlags::PropertyOperation\_None, NULL);

}

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_INSCRIPT(hr)

}

Var JavascriptExceptionOperators::OP\_RuntimeTypeError(MessageId messageId, ScriptContext \*scriptContext)

{

JavascriptError::ThrowTypeError(scriptContext, MAKE\_HR(messageId));

}

Var JavascriptExceptionOperators::OP\_RuntimeRangeError(MessageId messageId, ScriptContext \*scriptContext)

{

JavascriptError::ThrowRangeError(scriptContext, MAKE\_HR(messageId));

}

Var JavascriptExceptionOperators::OP\_RuntimeReferenceError(MessageId messageId, ScriptContext \*scriptContext)

{

JavascriptError::ThrowReferenceError(scriptContext, MAKE\_HR(messageId));

}

Var JavascriptExceptionOperators::ThrowTypeErrorAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

JavascriptError::ThrowTypeError(function->GetScriptContext(), VBSERR\_ActionNotSupported);

}

// Throw type error on access caller when in a restricted context

Var JavascriptExceptionOperators::ThrowTypeErrorCallerAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

JavascriptError::ThrowTypeError(function->GetScriptContext(), JSERR\_AccessCallerRestricted);

}

// Throw type error on access on callee when strict mode

Var JavascriptExceptionOperators::ThrowTypeErrorCalleeAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

JavascriptError::ThrowTypeError(function->GetScriptContext(), JSERR\_AccessCallee);

}

// Throw type error on access arguments when in a restricted context

Var JavascriptExceptionOperators::ThrowTypeErrorArgumentsAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

JavascriptError::ThrowTypeError(function->GetScriptContext(), JSERR\_AccessArgumentsRestricted);

}

Var JavascriptExceptionOperators::StackTraceAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

ARGUMENTS(args, callInfo);

AssertMsg(args.Info.Count > 0, "Should always have implicit 'this'");

ScriptContext \*scriptContext = function->GetScriptContext();

AnalysisAssert(scriptContext);

// If the first argument to the accessor is not a recyclable object, return undefined

// for compat with other browsers

if (!RecyclableObject::Is(args[0]))

{

return scriptContext->GetLibrary()->GetUndefined();

}

RecyclableObject \*obj = RecyclableObject::FromVar(args[0]);

// If an argument was passed to the accessor, it is being called as a setter.

// Set the internal StackTraceCache property accordingly.

if (args.Info.Count > 1)

{

obj->SetInternalProperty(InternalPropertyIds::StackTraceCache, args[1], PropertyOperationFlags::PropertyOperation\_None, NULL);

if (JavascriptError::Is(obj))

{

((JavascriptError \*)obj)->SetStackPropertyRedefined(true);

}

return scriptContext->GetLibrary()->GetEmptyString();

}

// Otherwise, the accessor is being called as a getter.

// Return existing cached value, or obtain the string representation of the StackTrace to return.

Var cache = NULL;

if (obj->GetInternalProperty(obj,InternalPropertyIds::StackTraceCache, (Var\*)&cache, NULL, scriptContext) && cache)

{

return cache;

}

JavascriptString\* stringMessage = scriptContext->GetLibrary()->GetEmptyString();

HRESULT hr;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

Js::JavascriptExceptionContext::StackTrace \*stackTrace = NULL;

if (!obj->GetInternalProperty(obj,InternalPropertyIds::StackTrace, (Js::Var\*) &stackTrace, NULL, scriptContext) ||

stackTrace == nullptr)

{

obj->SetInternalProperty(InternalPropertyIds::StackTraceCache, stringMessage, PropertyOperationFlags::PropertyOperation\_None, NULL);

return stringMessage;

}

if (IsErrorInstance(obj))

{

stringMessage = JavascriptConversion::ToString(obj, scriptContext);

}

CompoundString \*const stringBuilder = CompoundString::NewWithCharCapacity(40, scriptContext->GetLibrary());

stringBuilder->AppendChars(stringMessage);

for (int i = 0; i < stackTrace->Count(); i++)

{

Js::JavascriptExceptionContext::StackFrame currentFrame = stackTrace->Item(i);

// Defend in depth. Discard cross domain frames if somehow they creped in.

if (currentFrame.IsScriptFunction())

{

ScriptContext\* funcScriptContext = currentFrame.GetFunctionBody()->GetScriptContext();

AnalysisAssert(funcScriptContext);

if (scriptContext != funcScriptContext && FAILED(scriptContext->GetHostScriptContext()->CheckCrossDomainScriptContext(funcScriptContext)))

{

continue; // Ignore this frame

}

}

FunctionBody\* functionBody = currentFrame.GetFunctionBody();

const bool isLibraryCode = !functionBody || functionBody->GetUtf8SourceInfo()->GetIsLibraryCode();

if (isLibraryCode)

{

AppendLibraryFrameToStackTrace(stringBuilder, currentFrame.GetFunctionName());

}

else

{

LPCWSTR pUrl = NULL;

ULONG lineNumber = 0;

LONG characterPosition = 0;

functionBody->GetLineCharOffset(currentFrame.GetByteCodeOffset(), &lineNumber, &characterPosition);

pUrl = functionBody->GetSourceName();

LPCWSTR functionName = nullptr;

if (CONFIG\_FLAG(ExtendedErrorStackForTestHost))

{

BEGIN\_LEAVE\_SCRIPT\_INTERNAL(scriptContext)

{

if (currentFrame.GetFunctionNameWithArguments(&functionName) != S\_OK)

{

functionName = functionBody->GetExternalDisplayName();

}

}

END\_LEAVE\_SCRIPT\_INTERNAL(scriptContext)

}

else

{

functionName = functionBody->GetExternalDisplayName();

}

AppendExternalFrameToStackTrace(stringBuilder, functionName, pUrl ? pUrl : L"", lineNumber + 1, characterPosition + 1);

}

}

// Try to create the string object even if we did OOM, but if can't, just return what we've got. We catch and ignore OOM so it doesn’t propagate up.

// With all the stack trace functionality, we do best effort to produce the stack trace in the case of OOM, but don’t want it to trigger an OOM. Idea is if do take

// an OOM, have some chance of producing a stack trace to see where it happened.

stringMessage = stringBuilder;

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_INSCRIPT(hr);

obj->SetInternalProperty(InternalPropertyIds::StackTraceCache, stringMessage, PropertyOperationFlags::PropertyOperation\_None, NULL);

return stringMessage;

}

uint64 JavascriptExceptionOperators::GetStackTraceLimit(Var thrownObject, ScriptContext\* scriptContext)

{

uint64 limit = 0;

if (scriptContext->GetConfig()->IsErrorStackTraceEnabled()

&& IsErrorInstance(thrownObject))

{

HRESULT hr = JavascriptError::GetRuntimeError(RecyclableObject::FromVar(thrownObject), NULL);

JavascriptFunction\* error = scriptContext->GetLibrary()->GetErrorConstructor();

// If we are throwing StackOverflow and Error.stackTraceLimit is a custom getter, we can't make the getter

// call as we don't have stack space. Just bail out without stack trace in such case. Only proceed to get

// Error.stackTraceLimit property if we are not throwing StackOverflow, or there is no implicitCall (in getter case).

DisableImplicitFlags disableImplicitFlags = scriptContext->GetThreadContext()->GetDisableImplicitFlags();

if (hr == VBSERR\_OutOfStack)

{

scriptContext->GetThreadContext()->SetDisableImplicitFlags(DisableImplicitCallAndExceptionFlag);

}

Var var;

if (JavascriptOperators::GetProperty(error, PropertyIds::stackTraceLimit, &var, scriptContext))

{

// Only accept the value if it is a "Number". Avoid potential valueOf() call.

switch (JavascriptOperators::GetTypeId(var))

{

case TypeIds\_Integer:

case TypeIds\_Number:

case TypeIds\_Int64Number:

case TypeIds\_UInt64Number:

double value = JavascriptConversion::ToNumber(var, scriptContext);

limit = JavascriptNumber::IsNan(value) ? 0 :

(NumberUtilities::IsFinite(value) ? JavascriptConversion::ToUInt32(var, scriptContext) : MaxStackTraceLimit);

break;

}

}

if (hr == VBSERR\_OutOfStack)

{

scriptContext->GetThreadContext()->SetDisableImplicitFlags(disableImplicitFlags);

}

}

return limit;

}

void JavascriptExceptionOperators::AppendExternalFrameToStackTrace(CompoundString\* bs, LPCWSTR functionName, LPCWSTR fileName, ULONG lineNumber, LONG characterPosition)

{

// format is equivalent to printf("\n at %s (%s:%d:%d)", functionName, filename, lineNumber, characterPosition);

const CharCount maxULongStringLength = 10; // excluding null terminator

const auto ConvertULongToString = [](const ULONG value, wchar\_t \*const buffer, const CharCount charCapacity)

{

const errno\_t err = \_ultow\_s(value, buffer, charCapacity, 10);

Assert(err == 0);

};

if (CONFIG\_FLAG(ExtendedErrorStackForTestHost))

{

bs->AppendChars(L"\n\tat ");

}

else

{

bs->AppendChars(L"\n at ");

}

bs->AppendCharsSz(functionName);

bs->AppendChars(L" (");

if (CONFIG\_FLAG(ExtendedErrorStackForTestHost) && \*fileName != L'\0')

{

wchar\_t shortfilename[\_MAX\_FNAME];

wchar\_t ext[\_MAX\_EXT];

errno\_t err = \_wsplitpath\_s(fileName, NULL, 0, NULL, 0, shortfilename, \_MAX\_FNAME, ext, \_MAX\_EXT);

if (err != 0)

{

bs->AppendCharsSz(fileName);

}

else

{

bs->AppendCharsSz(shortfilename);

bs->AppendCharsSz(ext);

}

}

else

{

bs->AppendCharsSz(fileName);

}

bs->AppendChars(L':');

bs->AppendChars(lineNumber, maxULongStringLength, ConvertULongToString);

bs->AppendChars(L':');

bs->AppendChars(characterPosition, maxULongStringLength, ConvertULongToString);

bs->AppendChars(L')');

}

void JavascriptExceptionOperators::AppendLibraryFrameToStackTrace(CompoundString\* bs, LPCWSTR functionName)

{

// format is equivalent to printf("\n at %s (native code)", functionName);

bs->AppendChars(L"\n at ");

bs->AppendCharsSz(functionName);

bs->AppendChars(L" (native code)");

}

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifdef \_M\_AMD64

extern "C" void \*amd64\_CallWithFakeFrame(void \*target,

void \*frame,

size\_t spillSize,

size\_t argsSize,

void \*arg0 = nullptr) throw(...);

#elif defined(\_M\_ARM)

extern "C" void \*arm\_CallEhFrame(void \*target, void \*framePtr, void \*localsPtr, size\_t argsSize) throw(...);

extern "C" void \*arm\_CallCatch(void \*target, void \*framePtr, void \*localsPtr, size\_t argsSize, void \*catchObj) throw(...);

#elif defined(\_M\_ARM64)

extern "C" void \*arm64\_CallEhFrame(void \*target, void \*framePtr, void \*localsPtr, size\_t argsSize) throw(...);

extern "C" void \*arm64\_CallCatch(void \*target, void \*framePtr, void \*localsPtr, size\_t argsSize, void \*catchObj) throw(...);

#endif

namespace Js

{

class JavascriptExceptionContext;

class JavascriptExceptionOperators /\* All static \*/

{

public:

static const uint64 DefaultStackTraceLimit = 10;

static const uint64 MaxStackTraceLimit = \_UI64\_MAX;

// AutoCatchHandlerExists tracks where an exception will be caught and not propagated out.

// It should be included wherever an exception is caught and swallowed.

class AutoCatchHandlerExists

{

private:

bool m\_previousCatchHandlerExists;

bool m\_previousCatchHandlerToUserCodeStatus;

ThreadContext\* m\_threadContext;

void FetchNonUserCodeStatus(ScriptContext \*scriptContext);

public:

AutoCatchHandlerExists(ScriptContext\* scriptContext);

~AutoCatchHandlerExists();

};

static void \_\_declspec(noreturn) OP\_Throw(Var object, ScriptContext\* scriptContext);

static void \_\_declspec(noreturn) Throw(Var object, ScriptContext\* scriptContext);

static void \_\_declspec(noreturn) ThrowExceptionObject(Js::JavascriptExceptionObject\* exceptionObject, ScriptContext\* scriptContext, bool considerPassingToDebugger = false, PVOID returnAddress = NULL, bool resetStack = false);

static void \_\_declspec(noreturn) RethrowExceptionObject(Js::JavascriptExceptionObject\* exceptionObject, ScriptContext\* scriptContext, bool considerPassingToDebugger = false);

#ifdef \_M\_X64

static void \*OP\_TryCatch(void \*try\_, void \*catch\_, void \*frame, size\_t spillSize, size\_t argsSize, int hasBailedOutOffset, ScriptContext \*scriptContext);

static void \*OP\_TryFinally(void \*try\_, void \*finally\_, void \*frame, size\_t spillSize, size\_t argsSize, ScriptContext \*scriptContext);

#elif defined(\_M\_ARM32\_OR\_ARM64)

static void\* OP\_TryCatch(void\* continuationAddr, void\* handlerAddr, void\* framePtr, void \*localsPtr, size\_t argsSize, int hasBailedOutOffset, ScriptContext\* scriptContext);

static void\* OP\_TryFinally(void\* continuationAddr, void\* handlerAddr, void\* framePtr, void \*localsPtr, size\_t argsSize, ScriptContext\* scriptContext);

#else

static void\* OP\_TryCatch(void\* continuationAddr, void\* handlerAddr, void\* framePtr, int hasBailedOutOffset, ScriptContext\* scriptContext);

static void\* OP\_TryFinally(void\* continuationAddr, void\* handlerAddr, void\* framePtr, ScriptContext\* scriptContext);

#endif

#if defined(DBG) && defined(\_M\_IX86)

static void DbgCheckEHChain();

#endif

static JavascriptExceptionObject\* GetOutOfMemoryExceptionObject(ScriptContext\* scriptContext);

static Var OP\_RuntimeTypeError(MessageId messageId, ScriptContext\* scriptContext);

static Var OP\_RuntimeRangeError(MessageId messageId, ScriptContext\* scriptContext);

static Var OP\_RuntimeReferenceError(MessageId messageId, ScriptContext\* scriptContext);

static void \_\_declspec(noreturn) ThrowOutOfMemory(ScriptContext\* scriptContext);

static void \_\_declspec(noreturn) ThrowStackOverflow(ScriptContext\* scriptContext, PVOID returnAddress);

static uint64 GetStackTraceLimit(Var thrownObject, ScriptContext\* scriptContext);

static Var ThrowTypeErrorAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static Var ThrowTypeErrorCallerAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static Var ThrowTypeErrorCalleeAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static Var ThrowTypeErrorArgumentsAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static Var StackTraceAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static void WalkStackForExceptionContext(ScriptContext& scriptContext, JavascriptExceptionContext& exceptionContext, Var thrownObject, uint64 stackCrawlLimit, PVOID returnAddress, bool isThrownException = true, bool resetSatck = false);

static void AddStackTraceToObject(Var obj, JavascriptExceptionContext::StackTrace\* stackTrace, ScriptContext& scriptContext, bool isThrownException = true, bool resetSatck = false);

static uint64 StackCrawlLimitOnThrow(Var thrownObject, ScriptContext& scriptContext);

class EntryInfo

{

public:

static FunctionInfo ThrowTypeErrorAccessor;

static FunctionInfo StackTraceAccessor;

// For strict mode

static FunctionInfo ThrowTypeErrorCallerAccessor;

static FunctionInfo ThrowTypeErrorCalleeAccessor;

static FunctionInfo ThrowTypeErrorArgumentsAccessor;

};

private:

static JavascriptFunction \* WalkStackForExceptionContextInternal(ScriptContext& scriptContext, JavascriptExceptionContext& exceptionContext, Var thrownObject, uint32& callerByteCodeOffset,

uint64 stackCrawlLimit, PVOID returnAddress, bool isThrownException, bool resetStack = false);

static void ThrowExceptionObjectInternal(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext, bool fillExceptionContext, bool considerPassingToDebugger, PVOID returnAddress, bool resetStack);

static BOOL GetCaller(JavascriptStackWalker& walker, JavascriptFunction\*& jsFunc);

static void DumpStackTrace(JavascriptExceptionContext& exceptionContext, bool isThrownException = true);

static JavascriptExceptionContext::StackTrace\* TrimStackTraceForThrownObject(JavascriptExceptionContext::StackTrace\* stackTraceOriginal, Var thrownObject, ScriptContext& scriptContext);

static void AppendExternalFrameToStackTrace(CompoundString\* bs, LPCWSTR functionName, LPCWSTR fileName, ULONG lineNumber, LONG characterPosition);

static void AppendLibraryFrameToStackTrace(CompoundString\* bs, LPCWSTR functionName);

static bool IsErrorInstance(Var thrownObject);

static bool CrawlStackForWER(Js::ScriptContext& scriptContext);

static void DispatchExceptionToDebugger(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

enum JavascriptFunctionArgIndex {

JavascriptFunctionArgIndex\_Frame = -2,

JavascriptFunctionArgIndex\_ArgumentsObject = JavascriptFunctionArgIndex\_Frame - Js::Constants::ArgumentLocationOnFrame,

#if \_M\_IX86 || \_M\_AMD64

JavascriptFunctionArgIndex\_StackNestedFuncListWithNoArg = JavascriptFunctionArgIndex\_Frame - Js::Constants::StackNestedFuncListWithNoArg,

JavascriptFunctionArgIndex\_StackFrameDisplayNoArg = JavascriptFunctionArgIndex\_Frame - Js::Constants::StackFrameDisplayWithNoArg,

JavascriptFunctionArgIndex\_StackScopeSlotsNoArg = JavascriptFunctionArgIndex\_Frame - Js::Constants::StackScopeSlotsWithNoArg,

#endif

JavascriptFunctionArgIndex\_StackNestedFuncList = JavascriptFunctionArgIndex\_Frame - Js::Constants::StackNestedFuncList,

JavascriptFunctionArgIndex\_StackFrameDisplay = JavascriptFunctionArgIndex\_Frame - Js::Constants::StackFrameDisplay,

JavascriptFunctionArgIndex\_StackScopeSlots = JavascriptFunctionArgIndex\_Frame - Js::Constants::StackScopeSlots,

JavascriptFunctionArgIndex\_Function = 0,

JavascriptFunctionArgIndex\_CallInfo = 1,

JavascriptFunctionArgIndex\_This = 2, /\* (hidden) first script arg \*/

JavascriptFunctionArgIndex\_SecondScriptArg = 3

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

//JavascriptMath.cpp is a shared file with JavascriptMathOperators.cpp

#include "..\Math\JavascriptMath.cpp"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "..\Math\JavascriptMath.h"

#include "..\Math\AsmJsMath.h"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "..\Math\JavascriptMath.inl"

#include "..\Math\AsmJsMath.inl"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "JavascriptNativeOperators.h"

namespace Js

{

#if ENABLE\_NATIVE\_CODEGEN

void \* JavascriptNativeOperators::Op\_SwitchStringLookUp(JavascriptString\* str, Js::BranchDictionaryWrapper<JavascriptString\*>\* branchTargets, uintptr funcStart, uintptr funcEnd)

{

void\* defaultTarget = branchTargets->defaultTarget;

Js::BranchDictionaryWrapper<JavascriptString\*>::BranchDictionary& stringDictionary = branchTargets->dictionary;

void\* target = stringDictionary.Lookup(str, defaultTarget);

uintptr utarget = (uintptr)target;

if ((utarget - funcStart) > (funcEnd - funcStart))

{

AssertMsg(false, "Switch string dictionary jump target outside of function");

Throw::FatalInternalError();

}

return target;

}

#endif

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

namespace Js

{

#if ENABLE\_NATIVE\_CODEGEN

template <typename T>

class BranchDictionaryWrapper

{

public:

typedef JsUtil::BaseDictionary<T, void\*, NativeCodeData::Allocator> BranchDictionary;

BranchDictionaryWrapper(NativeCodeData::Allocator \* allocator, uint dictionarySize) :

defaultTarget(nullptr), dictionary(allocator)

{

}

BranchDictionary dictionary;

void\* defaultTarget;

static BranchDictionaryWrapper\* New(NativeCodeData::Allocator \* allocator, uint dictionarySize)

{

return NativeCodeDataNew(allocator, BranchDictionaryWrapper, allocator, dictionarySize);

}

};

class JavascriptNativeOperators

{

public:

static void \* Op\_SwitchStringLookUp(JavascriptString\* str, Js::BranchDictionaryWrapper<Js::JavascriptString\*>\* stringDictionary, uintptr funcStart, uintptr funcEnd);

};

#endif

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Types\PathTypeHandler.h"

#include "Types\PropertyIndexRanges.h"

#include "Types\WithScopeObject.h"

#include "Types\SpreadArgument.h"

#include "Library\JavascriptPromise.h"

#include "Library\JavascriptRegularExpression.h"

#include "Library\ThrowErrorObject.h"

#include "Library\JavascriptGeneratorFunction.h"

#include "Types\DynamicObjectEnumerator.h"

#include "Types\DynamicObjectSnapshotEnumerator.h"

#include "Types\DynamicObjectSnapshotEnumeratorWPCache.h"

#include "Library\ForInObjectEnumerator.h"

#include "Library\ES5Array.h"

#ifndef SCRIPT\_DIRECT\_TYPE

typedef enum JsNativeValueType

{

JsInt8Type,

JsUint8Type,

JsInt16Type,

JsUint16Type,

JsInt32Type,

JsUint32Type,

JsInt64Type,

JsUint64Type,

JsFloatType,

JsDoubleType,

JsNativeStringType

} JsNativeValueType;

typedef struct JsNativeString

{

unsigned int length;

LPCWSTR str;

} JsNativeString;

#endif

namespace Js

{

DEFINE\_RECYCLER\_TRACKER\_ARRAY\_PERF\_COUNTER(Var);

DEFINE\_RECYCLER\_TRACKER\_PERF\_COUNTER(FrameDisplay);

enum IndexType

{

IndexType\_Number,

IndexType\_PropertyId,

IndexType\_JavascriptString

};

IndexType GetIndexTypeFromString(wchar\_t const \* propertyName, charcount\_t propertyLength, ScriptContext\* scriptContext, uint32\* index, PropertyRecord const\*\* propertyRecord, bool createIfNotFound)

{

if (JavascriptOperators::TryConvertToUInt32(propertyName, propertyLength, index) &&

(\*index != JavascriptArray::InvalidIndex))

{

return IndexType\_Number;

}

else

{

if (createIfNotFound)

{

scriptContext->GetOrAddPropertyRecord(propertyName, propertyLength, propertyRecord);

}

else

{

scriptContext->FindPropertyRecord(propertyName, propertyLength, propertyRecord);

}

return IndexType\_PropertyId;

}

}

IndexType GetIndexType(Var indexVar, ScriptContext\* scriptContext, uint32\* index, PropertyRecord const \*\* propertyRecord, JavascriptString \*\* propertyNameString, bool createIfNotFound, bool preferJavascriptStringOverPropertyRecord)

{

indexVar = JavascriptConversion::ToPrimitive(indexVar, JavascriptHint::HintString, scriptContext);

// CONSIDER: Only OP\_SetElementI and OP\_GetElementI use and take advantage of the

// IndexType\_JavascriptString result. Consider modifying other callers of GetIndexType to take

// advantage of non-interned property strings where appropriate.

if (TaggedInt::Is(indexVar))

{

int indexInt = TaggedInt::ToInt32(indexVar);

if (indexInt >= 0)

{

\*index = (uint)indexInt;

return IndexType\_Number;

}

else

{

wchar\_t buffer[20];

::\_itow\_s(indexInt, buffer, sizeof(buffer)/sizeof(wchar\_t), 10);

charcount\_t length = JavascriptString::GetBufferLength(buffer);

if (createIfNotFound || preferJavascriptStringOverPropertyRecord)

{

// When preferring JavascriptString objects, just return a PropertyRecord instead

// of creating temporary JavascriptString objects for every negative integer that

// comes through here.

scriptContext->GetOrAddPropertyRecord(buffer, length, propertyRecord);

}

else

{

scriptContext->FindPropertyRecord(buffer, length, propertyRecord);

}

return IndexType\_PropertyId;

}

}

else if (JavascriptSymbol::Is(indexVar))

{

JavascriptSymbol\* symbol = JavascriptSymbol::FromVar(indexVar);

// JavascriptSymbols cannot add a new PropertyRecord - they correspond to one and only one existing PropertyRecord.

// We already know what the PropertyRecord is since it is stored in the JavascriptSymbol itself so just return it.

\*propertyRecord = symbol->GetValue();

return IndexType\_PropertyId;

}

else

{

JavascriptString\* indexStr = JavascriptConversion::ToString(indexVar, scriptContext);

wchar\_t const \* propertyName = indexStr->GetString();

charcount\_t const propertyLength = indexStr->GetLength();

if (!createIfNotFound && preferJavascriptStringOverPropertyRecord)

{

if (JavascriptOperators::TryConvertToUInt32(propertyName, propertyLength, index) &&

(\*index != JavascriptArray::InvalidIndex))

{

return IndexType\_Number;

}

\*propertyNameString = indexStr;

return IndexType\_JavascriptString;

}

return GetIndexTypeFromString(propertyName, propertyLength, scriptContext, index, propertyRecord, createIfNotFound);

}

}

IndexType GetIndexType(Var indexVar, ScriptContext\* scriptContext, uint32\* index, PropertyRecord const \*\* propertyRecord, bool createIfNotFound)

{

return GetIndexType(indexVar, scriptContext, index, propertyRecord, nullptr, createIfNotFound, false);

}

BOOL FEqualDbl(double dbl1, double dbl2)

{

// If the low ulongs don't match, they can't be equal.

if (Js::NumberUtilities::LuLoDbl(dbl1) != Js::NumberUtilities::LuLoDbl(dbl2))

return FALSE;

// If the high ulongs don't match, they can be equal iff one is -0 and

// the other is +0.

if (Js::NumberUtilities::LuHiDbl(dbl1) != Js::NumberUtilities::LuHiDbl(dbl2))

{

return 0x80000000 == (Js::NumberUtilities::LuHiDbl(dbl1) | Js::NumberUtilities::LuHiDbl(dbl2)) &&

0 == Js::NumberUtilities::LuLoDbl(dbl1);

}

// The bit patterns match. They are equal iff they are not Nan.

return !Js::NumberUtilities::IsNan(dbl1);

}

Var JavascriptOperators::OP\_ApplyArgs(Var func, Var instance, \_\_in\_xcount(8) void\*\* stackPtr, CallInfo callInfo, ScriptContext\* scriptContext)

{

int argCount=callInfo.Count;

///

/// Check func has internal [[Call]] property

/// If not, throw TypeError

///

if (!JavascriptConversion::IsCallable(func)) {

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction);

}

// Fix callInfo: expect result/value, and none of other flags are currently applicable.

// OP\_ApplyArgs expects a result. Neither of {jit, interpreted} mode sends correct callFlags:

// LdArgCnt -- jit sends whatever was passed to current function, interpreter always sends 0.

// See Win8 bug 490489.

callInfo.Flags = CallFlags\_Value;

RecyclableObject \*funcPtr = RecyclableObject::FromVar(func);

PROBE\_STACK(scriptContext, Js::Constants::MinStackDefault+argCount\*4);

JavascriptMethod entryPoint=funcPtr->GetEntryPoint();

Var ret;

switch (argCount) {

case 0:

Assert(false);

ret=entryPoint(funcPtr,callInfo);

break;

case 1:

ret=entryPoint(funcPtr,callInfo,instance);

break;

case 2:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0]);

break;

case 3:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1]);

break;

case 4:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1],stackPtr[2]);

break;

case 5:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1],stackPtr[2],stackPtr[3]);

break;

case 6:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1],stackPtr[2],stackPtr[3],stackPtr[4]);

break;

case 7:

ret=entryPoint(funcPtr,callInfo,instance,stackPtr[0],stackPtr[1],stackPtr[2],stackPtr[3],stackPtr[4],stackPtr[5]);

break;

default: {

// Don't need stack probe here- we just did so above

Arguments args(callInfo,stackPtr-1);

ret=JavascriptFunction::CallFunction<false>(funcPtr,entryPoint,args);

}

break;

}

return ret;

}

#ifdef \_M\_IX86

// Alias for overloaded JavascriptNumber::ToVar so it can be called unambiguously from native code

Var JavascriptOperators::Int32ToVar(int32 value, ScriptContext\* scriptContext)

{

return JavascriptNumber::ToVar(value, scriptContext);

}

// Alias for overloaded JavascriptNumber::ToVar so it can be called unambiguously from native code

Var JavascriptOperators::Int32ToVarInPlace(int32 value, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

return JavascriptNumber::ToVarInPlace(value, scriptContext, result);

}

// Alias for overloaded JavascriptNumber::ToVar so it can be called unambiguously from native code

Var JavascriptOperators::UInt32ToVar(uint32 value, ScriptContext\* scriptContext)

{

return JavascriptNumber::ToVar(value, scriptContext);

}

// Alias for overloaded JavascriptNumber::ToVar so it can be called unambiguously from native code

Var JavascriptOperators::UInt32ToVarInPlace(uint32 value, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

return JavascriptNumber::ToVarInPlace(value, scriptContext, result);

}

#endif

Var JavascriptOperators::OP\_FinishOddDivBy2(uint32 value, ScriptContext \*scriptContext)

{

return JavascriptNumber::New((double)(value + 0.5), scriptContext);

}

Var JavascriptOperators::ToNumberInPlace(Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if (TaggedInt::Is(aRight) || JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return aRight;

}

return JavascriptNumber::ToVarInPlace(JavascriptConversion::ToNumber(aRight, scriptContext), scriptContext, result);

}

Var JavascriptOperators::Typeof(Var var, ScriptContext\* scriptContext)

{

switch (JavascriptOperators::GetTypeId(var))

{

case TypeIds\_Undefined:

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

case TypeIds\_Null:

//null

return scriptContext->GetLibrary()->GetObjectTypeDisplayString();

case TypeIds\_Integer:

case TypeIds\_Number:

case TypeIds\_Int64Number:

case TypeIds\_UInt64Number:

return scriptContext->GetLibrary()->GetNumberTypeDisplayString();

case TypeIds\_SIMDFloat32x4:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return scriptContext->GetLibrary()->GetSIMDFloat32x4DisplayString();

}

case TypeIds\_SIMDFloat64x2:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return scriptContext->GetLibrary()->GetSIMDFloat64x2DisplayString();

}

case TypeIds\_SIMDInt32x4:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return scriptContext->GetLibrary()->GetSIMDInt32x4DisplayString();

}

case TypeIds\_SIMDInt8x16:

if (scriptContext->GetConfig()->IsSimdjsEnabled())

{

return scriptContext->GetLibrary()->GetSIMDInt8x16DisplayString();

}

default:

// Falsy objects are typeof 'undefined'.

if (RecyclableObject::FromVar(var)->GetType()->IsFalsy())

{

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

else

{

return RecyclableObject::FromVar(var)->GetTypeOfString(scriptContext);

}

}

}

Var JavascriptOperators::TypeofFld(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

return TypeofFld\_Internal(instance, false, propertyId, scriptContext);

}

Var JavascriptOperators::TypeofRootFld(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

return TypeofFld\_Internal(instance, true, propertyId, scriptContext);

}

Var JavascriptOperators::TypeofFld\_Internal(Var instance, const bool isRoot, PropertyId propertyId, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined , scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

Var value;

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

// In edge mode, spec compat is more important than backward compat. Use spec/web behavior here

if (isRoot

? !JavascriptOperators::GetRootProperty(instance, propertyId, &value, scriptContext)

: !JavascriptOperators::GetProperty(instance, object, propertyId, &value, scriptContext))

{

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

if (!scriptContext->IsUndeclBlockVar(value))

{

return JavascriptOperators::Typeof(value, scriptContext);

}

}

catch(Js::JavascriptExceptionObject \* )

{

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

Assert(scriptContext->IsUndeclBlockVar(value));

Assert(scriptContext->GetConfig()->IsLetAndConstEnabled());

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

Var JavascriptOperators::TypeofElem\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

if (JavascriptOperators::IsNumberFromNativeArray(instance, index, scriptContext))

return scriptContext->GetLibrary()->GetNumberTypeDisplayString();

#if FLOATVAR

return TypeofElem(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return TypeofElem(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::TypeofElem\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

if (JavascriptOperators::IsNumberFromNativeArray(instance, index, scriptContext))

return scriptContext->GetLibrary()->GetNumberTypeDisplayString();

#if FLOATVAR

return TypeofElem(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return TypeofElem(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Js::JavascriptString\* GetPropertyDisplayNameForError(Var prop, ScriptContext\* scriptContext)

{

JavascriptString\* str;

if (JavascriptSymbol::Is(prop))

{

str = JavascriptSymbol::ToString(JavascriptSymbol::FromVar(prop)->GetValue(), scriptContext);

}

else

{

str = JavascriptConversion::ToString(prop, scriptContext);

}

return str;

}

Var JavascriptOperators::TypeofElem(Var instance, Var index, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

Var member;

uint32 indexVal;

PropertyRecord const \* propertyRecord = nullptr;

ThreadContext\* threadContext = scriptContext->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

// For JS Objects, don't create the propertyId if not already added

bool createIfNotFound = !IsJsNativeObject(object) ||

(DynamicType::Is(object->GetTypeId()) && static\_cast<DynamicObject\*>(object)->GetTypeHandler()->IsStringTypeHandler()) || JavascriptProxy::Is(object);

if (GetIndexType(index, scriptContext, &indexVal, &propertyRecord, createIfNotFound) == IndexType\_Number)

{

// In edge mode, we don't need to worry about the special "unknown" behavior. If the item is not available from Get,

// just return undefined.

if (!JavascriptOperators::GetItem(instance, object, indexVal, &member, scriptContext))

{

// If the instance doesn't have the item, typeof result is "undefined".

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

}

else if (propertyRecord == nullptr)

{

Assert(IsJsNativeObject(object));

#if DBG

JavascriptString\* indexStr = JavascriptConversion::ToString(index, scriptContext);

PropertyRecord const \* debugPropertyRecord;

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &debugPropertyRecord);

AssertMsg(!JavascriptOperators::GetProperty(instance, object, debugPropertyRecord->GetPropertyId(), &member, scriptContext), "how did this property come? See OS Bug 2727708 if you see this come from the web");

#endif

// If the instance doesn't have the property, typeof result is "undefined".

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

else

{

if (!JavascriptOperators::GetProperty(instance, object, propertyRecord->GetPropertyId(), &member, scriptContext))

{

// If the instance doesn't have the property, typeof result is "undefined".

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

}

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return JavascriptOperators::Typeof(member, scriptContext);

}

catch(Js::JavascriptExceptionObject \* )

{

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

}

}

//

// Delete the given Var

//

Var JavascriptOperators::Delete(Var var, ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetTrue();

}

BOOL JavascriptOperators::Equal\_Full(Var aLeft, Var aRight, ScriptContext\* requestContext)

{

//

// Fast-path SmInts and paired Number combinations.

//

if (aLeft == aRight)

{

if (JavascriptNumber::Is(aLeft) && JavascriptNumber::IsNan(JavascriptNumber::GetValue(aLeft)))

{

return false;

}

else if (JavascriptVariantDate::Is(aLeft) == false) // only need to check on aLeft - since they are the same var, aRight would do the same

{

return true;

}

else

{

//In ES5 mode strict equals (===) on same instance of object type VariantDate succeeds.

//Hence equals needs to succeed.

return true;

}

}

BOOL result = false;

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// If aLeft == aRight, we would already have returned true above.

return false;

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) == JavascriptNumber::GetValue(aRight);

}

else

{

BOOL res = RecyclableObject::FromVar(aRight)->Equals(aLeft, &result, requestContext);

AssertMsg(res, "Should have handled this");

return result;

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

if (TaggedInt::Is(aRight))

{

return TaggedInt::ToDouble(aRight) == JavascriptNumber::GetValue(aLeft);

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) == JavascriptNumber::GetValue(aRight);

}

else

{

BOOL res = RecyclableObject::FromVar(aRight)->Equals(aLeft, &result, requestContext);

AssertMsg(res, "Should have handled this");

return result;

}

}

if (RecyclableObject::FromVar(aLeft)->Equals(aRight, &result, requestContext))

{

return result;

}

else

{

return false;

}

}

BOOL JavascriptOperators::Greater\_Full(Var aLeft,Var aRight,ScriptContext\* scriptContext)

{

return RelationalComparsionHelper(aRight, aLeft, scriptContext, false, false);

}

BOOL JavascriptOperators::Less\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return RelationalComparsionHelper(aLeft, aRight, scriptContext, true, false);

}

BOOL JavascriptOperators::RelationalComparsionHelper(Var aLeft, Var aRight, ScriptContext\* scriptContext, bool leftFirst, bool undefinedAs)

{

TypeId typeId = JavascriptOperators::GetTypeId(aLeft);

if (typeId == TypeIds\_Null)

{

aLeft=TaggedInt::ToVarUnchecked(0);

}

else if (typeId == TypeIds\_Undefined)

{

aLeft=scriptContext->GetLibrary()->GetNaN();

}

typeId = JavascriptOperators::GetTypeId(aRight);

if (typeId == TypeIds\_Null)

{

aRight=TaggedInt::ToVarUnchecked(0);

}

else if (typeId == TypeIds\_Undefined)

{

aRight=scriptContext->GetLibrary()->GetNaN();

}

double dblLeft, dblRight;

Redo:

TypeId leftType = JavascriptOperators::GetTypeId(aLeft);

TypeId rightType = JavascriptOperators::GetTypeId(aRight);

switch (leftType)

{

case TypeIds\_Integer:

dblLeft = TaggedInt::ToDouble(aLeft);

switch (rightType)

{

case TypeIds\_Integer:

dblRight = TaggedInt::ToDouble(aRight);

break;

case TypeIds\_Number:

dblRight = JavascriptNumber::GetValue(aRight);

break;

default:

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

}

break;

case TypeIds\_Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

switch (rightType)

{

case TypeIds\_Integer:

dblRight = TaggedInt::ToDouble(aRight);

break;

case TypeIds\_Number:

dblRight = JavascriptNumber::GetValue(aRight);

break;

default:

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

}

break;

case TypeIds\_Int64Number:

{

switch (rightType)

{

case TypeIds\_Int64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue < rightValue;

}

break;

case TypeIds\_UInt64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptUInt64Number::FromVar(aRight)->GetValue();

if (rightValue <= INT\_MAX && leftValue >= 0)

{

return leftValue < (\_\_int64)rightValue;

}

}

break;

}

dblLeft = (double)JavascriptInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

}

break;

// we cannot do double conversion between 2 int64 numbers as we can get wrong result after conversion

// i.e., two different numbers become the same after losing precision. We'll continue dbl comparison

// if either number is not an int64 number.

case TypeIds\_UInt64Number:

{

switch (rightType)

{

case TypeIds\_Int64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

if (leftValue < INT\_MAX && rightValue >= 0)

{

return (\_\_int64)leftValue < rightValue;

}

}

break;

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptUInt64Number::FromVar(aRight)->GetValue();

return leftValue < rightValue;

}

break;

}

dblLeft = (double)JavascriptUInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

}

break;

case TypeIds\_String:

switch (rightType)

{

case TypeIds\_Integer:

case TypeIds\_Number:

case TypeIds\_Boolean:

break;

default:

aRight = JavascriptConversion::ToPrimitive(aRight, JavascriptHint::HintNumber, scriptContext);

rightType = JavascriptOperators::GetTypeId(aRight);

if (rightType != TypeIds\_String)

{

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

}

case TypeIds\_String:

return JavascriptString::LessThan(aLeft, aRight);

}

dblLeft = JavascriptConversion::ToNumber(aLeft, scriptContext);

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

case TypeIds\_Boolean:

dblLeft = JavascriptConversion::ToNumber(aLeft, scriptContext);

dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

break;

default:

if (leftFirst)

{

aLeft = JavascriptConversion::ToPrimitive(aLeft, JavascriptHint::HintNumber, scriptContext);

aRight = JavascriptConversion::ToPrimitive(aRight, JavascriptHint::HintNumber, scriptContext);

}

else

{

aRight = JavascriptConversion::ToPrimitive(aRight, JavascriptHint::HintNumber, scriptContext);

aLeft = JavascriptConversion::ToPrimitive(aLeft, JavascriptHint::HintNumber, scriptContext);

}

goto Redo;

}

//

// And +0,-0 that is not implemented fully

//

if (JavascriptNumber::IsNan(dblLeft) || JavascriptNumber::IsNan(dblRight))

{

return undefinedAs;

}

// this will succeed for -0.0 == 0.0 case as well

if (dblLeft == dblRight)

{

return false;

}

return dblLeft < dblRight;

}

BOOL JavascriptOperators::StrictEqualString(Var aLeft, Var aRight)

{

Assert(JavascriptOperators::GetTypeId(aRight) == TypeIds\_String);

if (JavascriptOperators::GetTypeId(aLeft) != TypeIds\_String)

return false;

return JavascriptString::Equals(aLeft, aRight);

}

BOOL JavascriptOperators::StrictEqualEmptyString(Var aLeft)

{

TypeId leftType = JavascriptOperators::GetTypeId(aLeft);

if (leftType != TypeIds\_String)

return false;

return JavascriptString::FromVar(aLeft)->GetLength() == 0;

}

BOOL JavascriptOperators::StrictEqual(Var aLeft, Var aRight, ScriptContext\* requestContext)

{

double dblLeft, dblRight;

TypeId leftType = JavascriptOperators::GetTypeId(aLeft);

TypeId rightType = JavascriptOperators::GetTypeId(aRight);

switch (leftType)

{

case TypeIds\_String:

switch (rightType)

{

case TypeIds\_String:

return JavascriptString::Equals(aLeft, aRight);

}

return FALSE;

case TypeIds\_Integer:

switch (rightType)

{

case TypeIds\_Integer:

return aLeft == aRight;

// we don't need to worry about int64: it cannot equal as we create

// JavascriptInt64Number only in overflow scenarios.

case TypeIds\_Number:

dblLeft = TaggedInt::ToDouble(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

}

return FALSE;

case TypeIds\_Int64Number:

switch (rightType)

{

case TypeIds\_Int64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

case TypeIds\_UInt64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return ((unsigned \_\_int64)leftValue == rightValue);

}

case TypeIds\_Number:

dblLeft = (double)JavascriptInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

}

return FALSE;

case TypeIds\_UInt64Number:

switch (rightType)

{

case TypeIds\_Int64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return (leftValue == (unsigned \_\_int64)rightValue);

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

case TypeIds\_Number:

dblLeft = (double)JavascriptUInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CommonNumber;

}

return FALSE;

case TypeIds\_Number:

switch (rightType)

{

case TypeIds\_Integer:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = TaggedInt::ToDouble(aRight);

goto CommonNumber;

case TypeIds\_Int64Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = (double)JavascriptInt64Number::FromVar(aRight)->GetValue();

goto CommonNumber;

case TypeIds\_UInt64Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = (double)JavascriptUInt64Number::FromVar(aRight)->GetValue();

goto CommonNumber;

case TypeIds\_Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

CommonNumber:

return FEqualDbl(dblLeft, dblRight);

}

return FALSE;

case TypeIds\_Boolean:

switch (rightType)

{

case TypeIds\_Boolean:

return aLeft == aRight;

}

return FALSE;

case TypeIds\_Undefined:

return rightType == TypeIds\_Undefined;

case TypeIds\_Null:

return rightType == TypeIds\_Null;

case TypeIds\_Array:

return (rightType == TypeIds\_Array && aLeft == aRight);

case TypeIds\_Symbol:

switch (rightType)

{

case TypeIds\_Symbol:

{

const PropertyRecord\* leftValue = JavascriptSymbol::FromVar(aLeft)->GetValue();

const PropertyRecord\* rightValue = JavascriptSymbol::FromVar(aRight)->GetValue();

return leftValue == rightValue;

}

}

return false;

case TypeIds\_GlobalObject:

case TypeIds\_HostDispatch:

switch (rightType)

{

case TypeIds\_HostDispatch:

case TypeIds\_GlobalObject:

{

BOOL result;

if(RecyclableObject::FromVar(aLeft)->StrictEquals(aRight, &result, requestContext))

{

return result;

}

return false;

}

}

break;

case TypeIds\_Function:

if (rightType == TypeIds\_Function)

{

// In ES5 in certain cases (ES5 10.6.14(strict), 13.2.19(strict), 15.3.4.5.20-21) we return a function that throws type error.

// For different scenarios we return different instances of the function, which differ by exception/error message.

// According to ES5, this is the same [[ThrowTypeError]] (thrower) internal function, thus they should be equal.

if (JavascriptFunction::FromVar(aLeft)->IsThrowTypeErrorFunction() &&

JavascriptFunction::FromVar(aRight)->IsThrowTypeErrorFunction())

{

return true;

}

}

break;

}

if (RecyclableObject::FromVar(aLeft)->CanHaveInterceptors())

{

BOOL result;

if (RecyclableObject::FromVar(aLeft)->StrictEquals(aRight, &result, requestContext))

{

if (result)

{

return TRUE;

}

}

}

if (!TaggedNumber::Is(aRight) && RecyclableObject::FromVar(aRight)->CanHaveInterceptors())

{

BOOL result;

if (RecyclableObject::FromVar(aRight)->StrictEquals(aLeft, &result, requestContext))

{

if (result)

{

return TRUE;

}

}

}

return aLeft == aRight;

}

BOOL JavascriptOperators::HasOwnProperty(Var instance, PropertyId propertyId, ScriptContext \*requestContext)

{

BOOL result;

if (TaggedNumber::Is(instance))

{

result = false;

}

else

{

RecyclableObject\* object = RecyclableObject::FromVar(instance);

if (JavascriptProxy::Is(instance))

{

PropertyDescriptor desc;

return GetOwnPropertyDescriptor(object, propertyId, requestContext, &desc);

}

else

{

return object && object->HasOwnProperty(propertyId);

}

}

return result;

}

BOOL JavascriptOperators::GetOwnAccessors(Var instance, PropertyId propertyId, Var\* getter, Var\* setter, ScriptContext \* requestContext)

{

BOOL result;

if (TaggedNumber::Is(instance))

{

result = false;

}

else

{

RecyclableObject\* object = RecyclableObject::FromVar(instance);

result = object && object->GetAccessors(propertyId, getter, setter, requestContext);

}

return result;

}

Var JavascriptOperators::GetOwnPropertyNames(Var instance, ScriptContext \*scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::GetOwnPropertyNamesKind);

}

return JavascriptObject::CreateOwnStringPropertiesHelper(object, scriptContext);

}

Var JavascriptOperators::GetOwnPropertySymbols(Var instance, ScriptContext \*scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

CHAKRATEL\_LANGSTATS\_INC\_BUILTINCOUNT(GetOwnPropertySymbolsCount);

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::GetOwnPropertySymbolKind);

}

return JavascriptObject::CreateOwnSymbolPropertiesHelper(object, scriptContext);

}

Var JavascriptOperators::GetOwnPropertyKeys(Var instance, ScriptContext\* scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::KeysKind);

}

return JavascriptObject::CreateOwnStringSymbolPropertiesHelper(object, scriptContext);

}

Var JavascriptOperators::GetOwnEnumerablePropertyNames(Var instance, ScriptContext\* scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

Var result = proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::GetOwnPropertyNamesKind);

AssertMsg(JavascriptArray::Is(result), "PropertyKeysTrap should return JavascriptArray.");

JavascriptArray\* proxyResult;

JavascriptArray\* proxyResultToReturn = scriptContext->GetLibrary()->CreateArray(0);

if (JavascriptArray::Is(result))

{

proxyResult = JavascriptArray::FromVar(result);

}

else

{

return proxyResultToReturn;

}

// filter enumerable keys

uint32 resultLength = proxyResult->GetLength();

Var element;

const Js::PropertyRecord \*propertyRecord;

uint32 index = 0;

for (uint32 i = 0; i < resultLength; i++)

{

element = proxyResult->DirectGetItem(i);

Assert(!JavascriptSymbol::Is(element));

PropertyDescriptor propertyDescriptor;

JavascriptConversion::ToPropertyKey(element, scriptContext, &propertyRecord);

if (JavascriptOperators::GetOwnPropertyDescriptor(RecyclableObject::FromVar(instance), propertyRecord->GetPropertyId(), scriptContext, &propertyDescriptor))

{

if (propertyDescriptor.IsEnumerable())

{

proxyResultToReturn->DirectSetItemAt(index++, element);

}

}

}

return proxyResultToReturn;

}

return JavascriptObject::CreateOwnEnumerableStringPropertiesHelper(object, scriptContext);

}

Var JavascriptOperators::GetOwnEnumerablePropertyNamesSymbols(Var instance, ScriptContext\* scriptContext)

{

RecyclableObject \*object = RecyclableObject::FromVar(ToObject(instance, scriptContext));

if (JavascriptProxy::Is(instance))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->PropertyKeysTrap(JavascriptProxy::KeysTrapKind::KeysKind);

}

return JavascriptObject::CreateOwnEnumerableStringSymbolPropertiesHelper(object, scriptContext);

}

BOOL JavascriptOperators::GetOwnProperty(Var instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext)

{

BOOL result;

if (TaggedNumber::Is(instance))

{

result = false;

}

else

{

RecyclableObject\* object = RecyclableObject::FromVar(instance);

result = object && object->GetProperty(object, propertyId, value, NULL, requestContext);

}

return result;

}

BOOL JavascriptOperators::GetOwnPropertyDescriptor(RecyclableObject\* obj, JavascriptString\* propertyKey, ScriptContext\* scriptContext, PropertyDescriptor\* propertyDescriptor)

{

return JavascriptOperators::GetOwnPropertyDescriptor(obj, JavascriptOperators::GetPropertyId(propertyKey, scriptContext), scriptContext, propertyDescriptor);

}

// ES5's [[GetOwnProperty]].

// Return value:

// FALSE means "undefined" PD.

// TRUE means success. The propertyDescriptor parameter gets the descriptor.

//

BOOL JavascriptOperators::GetOwnPropertyDescriptor(RecyclableObject\* obj, PropertyId propertyId, ScriptContext\* scriptContext, PropertyDescriptor\* propertyDescriptor)

{

Assert(obj);

Assert(scriptContext);

Assert(propertyDescriptor);

if (JavascriptProxy::Is(obj))

{

return JavascriptProxy::GetOwnPropertyDescriptor(obj, propertyId, scriptContext, propertyDescriptor);

}

Var getter, setter;

if (false == JavascriptOperators::GetOwnAccessors(obj, propertyId, &getter, &setter, scriptContext))

{

Var value;

if (false == JavascriptOperators::GetOwnProperty(obj, propertyId, &value, scriptContext))

{

return FALSE;

}

if (nullptr != value)

{

propertyDescriptor->SetValue(value);

}

//CONSIDER : Its expensive to query for each flag from type system. Combine this with the GetOwnProperty to get all the flags

//at once. This will require a new API from type system and override in all the types which overrides IsEnumerable etc.

//Currently there is no performance tuning for ES5. This should be ok.

propertyDescriptor->SetWritable(FALSE != obj->IsWritable(propertyId));

}

else

{

if (nullptr == getter)

{

getter = scriptContext->GetLibrary()->GetUndefined();

}

propertyDescriptor->SetGetter(getter);

if (nullptr == setter)

{

setter = scriptContext->GetLibrary()->GetUndefined();

}

propertyDescriptor->SetSetter(setter);

}

propertyDescriptor->SetConfigurable(FALSE != obj->IsConfigurable(propertyId));

propertyDescriptor->SetEnumerable(FALSE != obj->IsEnumerable(propertyId));

return TRUE;

}

\_\_inline RecyclableObject\* JavascriptOperators::GetPrototypeNoTrap(RecyclableObject\* instance)

{

Type\* type = instance->GetType();

if (type->HasSpecialPrototype())

{

if (type->GetTypeId() == TypeIds\_Proxy)

{

// get back null

Assert(type->GetPrototype() == instance->GetScriptContext()->GetLibrary()->GetNull());

return type->GetPrototype();

}

else

{

return instance->GetPrototypeSpecial();

}

}

return type->GetPrototype();

}

BOOL JavascriptOperators::IsArray(Var instanceVar)

{

if (!RecyclableObject::Is(instanceVar))

{

return FALSE;

}

RecyclableObject\* instance = RecyclableObject::FromVar(instanceVar);

if (DynamicObject::IsAnyArray(instance))

{

return TRUE;

}

if (JavascriptProxy::Is(instanceVar))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instanceVar);

return IsArray(proxy->GetTarget());

}

TypeId remoteTypeId = TypeIds\_Limit;

if (JavascriptOperators::GetRemoteTypeId(instanceVar, &remoteTypeId) &&

DynamicObject::IsAnyArrayTypeId(remoteTypeId))

{

return TRUE;

}

return FALSE;

}

BOOL JavascriptOperators::IsConstructor(Var instanceVar)

{

if (!RecyclableObject::Is(instanceVar))

{

return FALSE;

}

if (JavascriptProxy::Is(instanceVar))

{

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instanceVar);

return IsConstructor(proxy->GetTarget());

}

if (!JavascriptFunction::Is(instanceVar))

{

return FALSE;

}

return JavascriptFunction::FromVar(instanceVar)->IsConstructor();

}

BOOL JavascriptOperators::IsConcatSpreadable(Var instanceVar)

{

// an object is spreadable under two condition, either it is a JsArray

// or you define an isconcatSpreadable flag on it.

if (!JavascriptOperators::IsObject(instanceVar))

{

return false;

}

RecyclableObject\* instance = RecyclableObject::FromVar(instanceVar);

ScriptContext\* scriptContext = instance->GetScriptContext();

Var spreadable = JavascriptOperators::GetProperty(instance, PropertyIds::\_symbolIsConcatSpreadable, scriptContext);

if (spreadable != scriptContext->GetLibrary()->GetUndefined())

{

return JavascriptConversion::ToBoolean(spreadable, scriptContext);

}

if (JavascriptOperators::IsArray(instance))

{

return true;

}

return false;

}

Var JavascriptOperators::OP\_LdCustomSpreadIteratorList(Var aRight, ScriptContext\* scriptContext)

{

RecyclableObject\* function = GetIteratorFunction(aRight, scriptContext);

JavascriptMethod method = function->GetEntryPoint();

if ((JavascriptArray::Is(aRight) && method == JavascriptArray::EntryInfo::Values.GetOriginalEntryPoint()) ||

(TypedArrayBase::Is(aRight) && method == TypedArrayBase::EntryInfo::Values.GetOriginalEntryPoint()))

{

// TODO: There is a compliance bug here in the case where the user has changed %ArrayIteratorPrototype%.next(); we won't call it.

// Checking if the property has been modified is currently not possible without doing a Get on it which might call user code.

// Fixing this bug will require a way to get the value stored in the property without doing the evaluation semantics of a Get.

return aRight;

}

Var iteratorVar = function->GetEntryPoint()(function, CallInfo(Js::CallFlags\_Value, 1), aRight);

if (!JavascriptOperators::IsObject(iteratorVar))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

RecyclableObject\* iterator = RecyclableObject::FromVar(iteratorVar);

return RecyclerNew(scriptContext->GetRecycler(), SpreadArgument, aRight, iterator, scriptContext->GetLibrary()->GetSpreadArgumentType());

}

BOOL JavascriptOperators::IsPropertyUnscopable(Var instanceVar, JavascriptString \*propertyString)

{

// This never gets called.

Throw::InternalError();

}

BOOL JavascriptOperators::IsPropertyUnscopable(Var instanceVar, PropertyId propertyId)

{

RecyclableObject\* instance = RecyclableObject::FromVar(instanceVar);

ScriptContext \* scriptContext = instance->GetScriptContext();

Var unscopables = JavascriptOperators::GetProperty(instance, PropertyIds::\_symbolUnscopables, scriptContext);

if (JavascriptOperators::IsObject(unscopables))

{

DynamicObject \*blackList = DynamicObject::FromVar(unscopables);

Var value;

//8.1.1.2.1.9.c If blocked is not undefined

if (JavascriptOperators::GetProperty(blackList, propertyId, &value, scriptContext))

{

return JavascriptConversion::ToBoolean(value, scriptContext);

}

}

return false;

}

BOOL JavascriptOperators::HasProperty(RecyclableObject\* instance, PropertyId propertyId)

{

while (JavascriptOperators::GetTypeId(instance) != TypeIds\_Null)

{

if (instance->HasProperty(propertyId))

{

return true;

}

instance = JavascriptOperators::GetPrototypeNoTrap(instance);

}

return false;

}

BOOL JavascriptOperators::HasPropertyUnscopables(RecyclableObject\* instance, PropertyId propertyId)

{

return JavascriptOperators::HasProperty(instance, propertyId)

&& !IsPropertyUnscopable(instance, propertyId);

}

BOOL JavascriptOperators::HasRootProperty(RecyclableObject\* instance, PropertyId propertyId)

{

Assert(RootObjectBase::Is(instance));

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(instance);

if (rootObject->HasRootProperty(propertyId))

{

return true;

}

instance = instance->GetPrototype();

return HasProperty(instance, propertyId);

}

BOOL JavascriptOperators::HasProxyOrPrototypeInlineCacheProperty(RecyclableObject\* instance, PropertyId propertyId)

{

TypeId typeId;

typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == Js::TypeIds\_Proxy)

{

// let's be more aggressive to disable inline prototype cache when proxy is presented in the prototypechain

return true;

}

do

{

instance = instance->GetPrototype();

typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == Js::TypeIds\_Proxy)

{

// let's be more aggressive to disable inline prototype cache when proxy is presented in the prototypechain

return true;

}

if (typeId == TypeIds\_Null)

{

break;

}

/\* We can rule out object with deferred type handler, because they would have expanded if they are in the cache \*/

if (!instance->HasDeferredTypeHandler() && instance->HasProperty(propertyId)) { return true; }

} while (typeId != TypeIds\_Null);

return false;

}

BOOL JavascriptOperators::OP\_HasProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

RecyclableObject\* object = TaggedNumber::Is(instance) ?

scriptContext->GetLibrary()->GetNumberPrototype() :

RecyclableObject::FromVar(instance);

BOOL result = HasProperty(object, propertyId);

return result;

}

BOOL JavascriptOperators::OP\_HasOwnProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

RecyclableObject\* object = TaggedNumber::Is(instance) ?

scriptContext->GetLibrary()->GetNumberPrototype() :

RecyclableObject::FromVar(instance);

BOOL result = HasOwnProperty(object, propertyId, scriptContext);

return result;

}

// CONSIDER: Have logic similar to HasOwnPropertyNoHostObjectForHeapEnum

BOOL JavascriptOperators::HasOwnPropertyNoHostObject(Var instance, PropertyId propertyId)

{

AssertMsg(!TaggedNumber::Is(instance), "HasOwnPropertyNoHostObject int passed");

RecyclableObject\* object = RecyclableObject::FromVar(instance);

return object && object->HasOwnPropertyNoHostObject(propertyId);

}

// CONSIDER: Remove HasOwnPropertyNoHostObjectForHeapEnum and use GetOwnPropertyNoHostObjectForHeapEnum in its place by changing it

// to return BOOL, true or false with whether the property exists or not, and return the value if not getter/setter as an out param.

BOOL JavascriptOperators::HasOwnPropertyNoHostObjectForHeapEnum(Var instance, PropertyId propertyId, ScriptContext\* requestContext, Var& getter, Var& setter)

{

AssertMsg(!TaggedNumber::Is(instance), "HasOwnPropertyNoHostObjectForHeapEnum int passed");

RecyclableObject \* object = RecyclableObject::FromVar(instance);

if (StaticType::Is(object->GetTypeId()))

{

return FALSE;

}

getter = setter = NULL;

DynamicObject\* dynamicObject = DynamicObject::FromVar(instance);

Assert(dynamicObject->GetScriptContext()->IsHeapEnumInProgress());

if (dynamicObject->UseDynamicObjectForNoHostObjectAccess())

{

if (!dynamicObject->DynamicObject::GetAccessors(propertyId, &getter, &setter, requestContext))

{

Var value;

if (!dynamicObject->DynamicObject::GetProperty(instance, propertyId, &value, NULL, requestContext) ||

(requestContext->IsUndeclBlockVar(value) && (ActivationObject::Is(instance) || RootObjectBase::Is(instance))))

{

return FALSE;

}

}

}

else

{

if (!object->GetAccessors(propertyId, &getter, &setter, requestContext))

{

Var value;

if (!object->GetProperty(instance, propertyId, &value, NULL, requestContext) ||

(requestContext->IsUndeclBlockVar(value) && (ActivationObject::Is(instance) || RootObjectBase::Is(instance))))

{

return FALSE;

}

}

}

return TRUE;

}

Var JavascriptOperators::GetOwnPropertyNoHostObjectForHeapEnum(Var instance, PropertyId propertyId, ScriptContext\* requestContext, Var& getter, Var& setter)

{

AssertMsg(!TaggedNumber::Is(instance), "GetDataPropertyNoHostObject int passed");

Assert(HasOwnPropertyNoHostObjectForHeapEnum(instance, propertyId, requestContext, getter, setter) || getter || setter);

DynamicObject\* dynamicObject = DynamicObject::FromVar(instance);

getter = setter = NULL;

if (NULL == dynamicObject)

{

return requestContext->GetLibrary()->GetUndefined();

}

Var returnVar = requestContext->GetLibrary()->GetUndefined();

BOOL result = FALSE;

if (dynamicObject->UseDynamicObjectForNoHostObjectAccess())

{

if (! dynamicObject->DynamicObject::GetAccessors(propertyId, &getter, &setter, requestContext))

{

result = dynamicObject->DynamicObject::GetProperty(instance, propertyId, &returnVar, NULL, requestContext);

}

}

else

{

if (! dynamicObject->GetAccessors(propertyId, &getter, &setter, requestContext))

{

result = dynamicObject->GetProperty(instance, propertyId, &returnVar, NULL, requestContext);

}

}

if (result)

{

return returnVar;

}

return requestContext->GetLibrary()->GetUndefined();

}

BOOL JavascriptOperators::OP\_HasOwnPropScoped(Var scope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext)

{

AssertMsg(scope == scriptContext->GetLibrary()->GetNull() || JavascriptArray::Is(scope),

"Invalid scope chain pointer passed - should be null or an array");

if (JavascriptArray::Is(scope))

{

JavascriptArray\* arrScope = JavascriptArray::FromVar(scope);

Var instance = arrScope->DirectGetItem(0);

return JavascriptOperators::OP\_HasOwnProperty(instance, propertyId, scriptContext);

}

return JavascriptOperators::OP\_HasOwnProperty(defaultInstance, propertyId, scriptContext);

}

BOOL JavascriptOperators::GetPropertyUnscopable(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetProperty\_Internal<true>(instance, propertyObject, false, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::GetProperty(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetProperty\_Internal<false>(instance, propertyObject, false, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::GetRootProperty(Var instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetProperty\_Internal<false>(instance, RecyclableObject::FromVar(instance), true, propertyId, value, requestContext, info);

}

template <bool unscopables>

BOOL JavascriptOperators::GetProperty\_Internal(Var instance, RecyclableObject\* propertyObject, const bool isRoot, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

if (TaggedNumber::Is(instance))

{

PropertyValueInfo::ClearCacheInfo(info);

}

RecyclableObject\* object = propertyObject;

BOOL foundProperty = FALSE;

if (isRoot)

{

Assert(RootObjectBase::Is(object));

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(object);

foundProperty = rootObject->GetRootProperty(instance, propertyId, value, info, requestContext);

}

while (!foundProperty && JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (unscopables && IsPropertyUnscopable(object, propertyId))

{

break;

}

else

{

if (object->GetProperty(instance, propertyId, value, info, requestContext))

{

foundProperty = true;

break;

}

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

if (foundProperty)

{

#if DBG

if (DynamicObject::Is(object))

{

DynamicObject\* dynamicObject = (DynamicObject\*)object;

DynamicTypeHandler\* dynamicTypeHandler = dynamicObject->GetDynamicType()->GetTypeHandler();

Var property;

if (dynamicTypeHandler->CheckFixedProperty(requestContext->GetPropertyName(propertyId), &property, requestContext))

{

Assert(value == nullptr || \*value == property);

}

}

#endif

// Don't cache the information if the value is undecl block var

// REVIEW: We might want to only check this if we need to (For LdRootFld or ScopedLdFld)

// Also we might want to throw here instead of checking it again in the caller

if (value && !requestContext->IsUndeclBlockVar(\*value) && !WithScopeObject::Is(object))

{

CacheOperators::CachePropertyRead(instance, object, isRoot, propertyId, false, info, requestContext);

}

#ifdef TELEMETRY\_JSO

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

requestContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, value, /\*successful: \*/true);

}

#endif

return TRUE;

}

else

{

#ifdef MISSING\_PROPERTY\_STATS

if (PHASE\_STATS1(MissingPropertyCachePhase))

{

requestContext->RecordMissingPropertyMiss();

}

#endif

if (PHASE\_TRACE1(MissingPropertyCachePhase))

{

Output::Print(L"MissingPropertyCaching: Missing property %d on slow path.\n", propertyId);

}

// Only cache missing property lookups for non-root field loads on objects that have PathTypeHandlers, because only these objects guarantee a type change when the property is added,

// which obviates the need to explicitly invalidate missing property inline caches.

if (!PHASE\_OFF1(MissingPropertyCachePhase) && !isRoot && DynamicObject::Is(instance) && ((DynamicObject\*)instance)->GetDynamicType()->GetTypeHandler()->IsPathTypeHandler())

{

#ifdef MISSING\_PROPERTY\_STATS

if (PHASE\_STATS1(MissingPropertyCachePhase))

{

requestContext->RecordMissingPropertyCacheAttempt();

}

#endif

if (PHASE\_TRACE1(MissingPropertyCachePhase))

{

Output::Print(L"MissingPropertyCache: Caching missing property for property %d.\n", propertyId);

}

PropertyValueInfo::Set(info, requestContext->GetLibrary()->GetMissingPropertyHolder(), 0);

CacheOperators::CachePropertyRead(instance, requestContext->GetLibrary()->GetMissingPropertyHolder(), isRoot, propertyId, true, info, requestContext);

}

#if defined(TELEMETRY\_JSO) || defined(TELEMETRY\_AddToCache) // enabled for `TELEMETRY\_AddToCache`, because this is the property-not-found codepath where the normal TELEMETRY\_AddToCache code wouldn't be executed.

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

if (info && info->AllowResizingPolymorphicInlineCache()) // If in interpreted mode, not JIT.

{

requestContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, nullptr, /\*successful: \*/false);

}

}

#endif

return FALSE;

}

}

template<typename PropertyKeyType>

BOOL JavascriptOperators::GetPropertyWPCache(Var instance, RecyclableObject\* propertyObject, PropertyKeyType propertyKey, Var\* value, ScriptContext\* requestContext, PropertyString \* propertyString)

{

if (TaggedNumber::Is(instance))

{

propertyString = NULL;

}

PropertyValueInfo info;

RecyclableObject\* object = propertyObject;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->GetProperty(instance, propertyKey, value, &info, requestContext))

{

if (propertyString != NULL)

{

uint16 slotIndex = info.GetPropertyIndex();

if (slotIndex != Constants::NoSlot &&

info.GetInstance() == object &&

info.IsWritable() && !object->CanHaveInterceptors() &&

requestContext == object->GetScriptContext() &&

((info.GetFlags() & (InlineCacheGetterFlag | InlineCacheSetterFlag)) == 0))

{

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

DynamicObject::FromVar(info.GetInstance())->GetTypeHandler()->PropertyIndexToInlineOrAuxSlotIndex(slotIndex, &inlineOrAuxSlotIndex, &isInlineSlot);

propertyString->UpdateCache(info.GetInstance()->GetType(), inlineOrAuxSlotIndex, isInlineSlot, info.IsStoreFieldCacheEnabled());

}

}

return TRUE;

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return FALSE;

}

BOOL JavascriptOperators::GetPropertyObject(Var instance, ScriptContext \* scriptContext, RecyclableObject\*\* propertyObject)

{

Assert(propertyObject);

if (TaggedNumber::Is(instance))

{

\*propertyObject = scriptContext->GetLibrary()->GetNumberPrototype();

return TRUE;

}

RecyclableObject\* object = RecyclableObject::FromVar(instance);

TypeId typeId = object->GetTypeId();

\*propertyObject = object;

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

return FALSE;

}

return TRUE;

}

#if DBG

BOOL JavascriptOperators::IsPropertyObject(RecyclableObject \* instance)

{

TypeId typeId = JavascriptOperators::GetTypeId(instance);

return (typeId != TypeIds\_Integer && typeId != TypeIds\_Null && typeId != TypeIds\_Undefined);

}

#endif

Var JavascriptOperators::OP\_GetProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined, scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

Var result = JavascriptOperators::GetProperty(instance, object, propertyId, scriptContext);

AssertMsg(result != nullptr, "result null in OP\_GetProperty");

return result;

}

Var JavascriptOperators::OP\_GetRootProperty(Var instance, PropertyId propertyId, PropertyValueInfo \* info, ScriptContext\* scriptContext)

{

AssertMsg(RootObjectBase::Is(instance), "Root must be an object!");

Var value;

if (JavascriptOperators::GetRootProperty(RecyclableObject::FromVar(instance), propertyId, &value, scriptContext, info))

{

if (scriptContext->IsUndeclBlockVar(value))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

return value;

}

const wchar\_t\* propertyName = scriptContext->GetPropertyName(propertyId)->GetBuffer();

JavascriptFunction \* caller = nullptr;

if (JavascriptStackWalker::GetCaller(&caller, scriptContext))

{

FunctionBody \* callerBody = caller->GetFunctionBody();

if (callerBody && callerBody->GetUtf8SourceInfo()->GetIsXDomain())

{

propertyName = nullptr;

}

}

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UndefVariable, propertyName);

}

return scriptContext->GetLibrary()->GetUndefined();

}

Var JavascriptOperators::OP\_GetThisScoped(FrameDisplay \*pScope, Var defaultInstance, ScriptContext\* scriptContext)

{

// NOTE: If changes are made to this logic be sure to update the debuggers as well

int length = pScope->GetLength();

for (int i = 0; i < length; i += 1)

{

Var value;

DynamicObject \*obj = DynamicObject::FromVar(pScope->GetItem(i));

if (JavascriptOperators::GetProperty(obj, Js::PropertyIds::\_lexicalThisSlotSymbol, &value, scriptContext))

{

return value;

}

}

return defaultInstance;

}

Var JavascriptOperators::OP\_UnwrapWithObj(Var aValue)

{

return RecyclableObject::FromVar(aValue)->GetThisObjectOrUnWrap();

}

Var JavascriptOperators::OP\_GetInstanceScoped(FrameDisplay \*pScope, PropertyId propertyId, Var rootObject, Var\* thisVar, ScriptContext\* scriptContext)

{

// Similar to GetPropertyScoped, but instead of returning the property value, we return the instance that

// owns it, or the global object if no instance is found.

int i;

int length = pScope->GetLength();

for (i = 0; i < length; i++)

{

RecyclableObject \*obj = (RecyclableObject\*)pScope->GetItem(i);

if (JavascriptOperators::HasProperty(obj, propertyId))

{

// HasProperty will call WithObjects HasProperty which will do the filtering

// All we have to do here is unwrap the object hence the api call

\*thisVar = obj->GetThisObjectOrUnWrap();

return \*thisVar;

}

}

\*thisVar = scriptContext->GetLibrary()->GetUndefined();

if (rootObject != scriptContext->GetGlobalObject())

{

if (JavascriptOperators::OP\_HasProperty(rootObject, propertyId, scriptContext))

{

return rootObject;

}

}

return scriptContext->GetGlobalObject();

}

Var JavascriptOperators::GetPropertyReference(RecyclableObject \*instance, PropertyId propertyId, ScriptContext\* requestContext)

{

Var value = nullptr;

PropertyValueInfo info;

if (JavascriptOperators::GetPropertyReference(instance, propertyId, &value, requestContext, &info))

{

Assert(value != nullptr);

return value;

}

return requestContext->GetLibrary()->GetUndefined();

}

BOOL JavascriptOperators::GetPropertyReference(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetPropertyReference\_Internal(instance, propertyObject, false, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::GetRootPropertyReference(RecyclableObject\* instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return GetPropertyReference\_Internal(instance, instance, true, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::PropertyReferenceWalkUnscopable(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return PropertyReferenceWalk\_Impl<true>(instance, propertyObject, propertyId, value, info, requestContext);

}

BOOL JavascriptOperators::PropertyReferenceWalk(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return PropertyReferenceWalk\_Impl<false>(instance, propertyObject, propertyId, value, info, requestContext);

}

template <bool unscopables>

BOOL JavascriptOperators::PropertyReferenceWalk\_Impl(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

BOOL foundProperty = false;

RecyclableObject\* object = \*propertyObject;

while (!foundProperty && JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (unscopables && JavascriptOperators::IsPropertyUnscopable(object, propertyId))

{

break;

}

else

{

if (object->GetPropertyReference(instance, propertyId, value, info, requestContext))

{

foundProperty = true;

break;

}

}

if (object->SkipsPrototype())

{

break; // will return false

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

\*propertyObject = object;

return foundProperty;

}

BOOL JavascriptOperators::GetPropertyReference\_Internal(Var instance, RecyclableObject\* propertyObject, const bool isRoot, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

if (TaggedNumber::Is(instance))

{

PropertyValueInfo::ClearCacheInfo(info);

}

BOOL foundProperty = FALSE;

RecyclableObject\* object = propertyObject;

if (isRoot)

{

foundProperty = RootObjectBase::FromVar(object)->GetRootPropertyReference(instance, propertyId, value, info, requestContext);

}

if (!foundProperty)

{

foundProperty = PropertyReferenceWalk(instance, &object, propertyId, value, info, requestContext);

}

if (!foundProperty)

{

#if defined(TELEMETRY\_JSO) || defined(TELEMETRY\_AddToCache) // enabled for `TELEMETRY\_AddToCache`, because this is the property-not-found codepath where the normal TELEMETRY\_AddToCache code wouldn't be executed.

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

if (info && info->AllowResizingPolymorphicInlineCache()) // If in interpreted mode, not JIT.

{

requestContext->GetTelemetry().GetOpcodeTelemetry().GetProperty(instance, propertyId, nullptr, /\*successful: \*/false);

}

}

#endif

return foundProperty;

}

if (requestContext->IsUndeclBlockVar(\*value))

{

JavascriptError::ThrowReferenceError(requestContext, JSERR\_UseBeforeDeclaration);

}

#if DBG

if (DynamicObject::Is(object))

{

DynamicObject\* dynamicObject = (DynamicObject\*)object;

DynamicTypeHandler\* dynamicTypeHandler = dynamicObject->GetDynamicType()->GetTypeHandler();

Var property;

if (dynamicTypeHandler->CheckFixedProperty(requestContext->GetPropertyName(propertyId), &property, requestContext))

{

Assert(value == nullptr || \*value == property);

}

}

#endif

CacheOperators::CachePropertyRead(instance, object, isRoot, propertyId, false, info, requestContext);

return TRUE;

}

template <typename PropertyKeyType, bool unscopable>

DescriptorFlags JavascriptOperators::GetterSetter\_Impl(RecyclableObject\* instance, PropertyKeyType propertyKey, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

DescriptorFlags flags = None;

RecyclableObject\* object = instance;

while (flags == None && JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (unscopable && IsPropertyUnscopable(object, propertyKey))

{

break;

}

else

{

flags = object->GetSetter(propertyKey, setterValue, info, scriptContext);

if (flags != None)

{

break;

}

}

// CONSIDER: we should add SkipsPrototype support. DOM has no ES 5 concepts built in that aren't

// already part of our prototype objects which are chakra objects.

object = object->GetPrototype();

}

return flags;

}

DescriptorFlags JavascriptOperators::GetterSetterUnscopable(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

return GetterSetter\_Impl<PropertyId, true>(instance, propertyId, setterValue, info, scriptContext);

}

DescriptorFlags JavascriptOperators::GetterSetter(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

return GetterSetter\_Impl<PropertyId, false>(instance, propertyId, setterValue, info, scriptContext);

}

DescriptorFlags JavascriptOperators::GetterSetter(RecyclableObject\* instance, JavascriptString \* propertyName, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

return GetterSetter\_Impl<JavascriptString\*, false>(instance, propertyName, setterValue, info, scriptContext);

}

// Checks to see if any object in the prototype chain has a property descriptor for the given property

// that specifies either an accessor or a non-writable attribute.

// If TRUE, check flags for details.

template<typename PropertyKeyType, bool doFastProtoChainCheck, bool isRoot>

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritablePropertyCore(RecyclableObject\* instance,

PropertyKeyType propertyKey, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

Assert(setterValue);

Assert(flags);

// Do a quick check to see if all objects in the prototype chain are known to have only

// writable data properties (i.e. no accessors or non-writable properties).

if (doFastProtoChainCheck && CheckIfObjectAndPrototypeChainHasOnlyWritableDataProperties(instance))

{

return FALSE;

}

if (isRoot)

{

\*flags = JavascriptOperators::GetRootSetter(instance, propertyKey, setterValue, info, scriptContext);

}

if (\*flags == None)

{

\*flags = JavascriptOperators::GetterSetter(instance, propertyKey, setterValue, info, scriptContext);

}

return ((\*flags & Accessor) == Accessor) || ((\*flags & Proxy) == Proxy)|| ((\*flags & Data) == Data && (\*flags & Writable) == None);

}

void JavascriptOperators::OP\_InvalidateProtoCaches(PropertyId propertyId, ScriptContext \*scriptContext)

{

scriptContext->InvalidateProtoCaches(propertyId);

}

// Checks to see if any object in the prototype chain has a property descriptor for the given index

// that specifies either an accessor or a non-writable attribute.

// If TRUE, check flags for details.

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritableItem(RecyclableObject\* instance, uint32 index,

Var\* setterValue, DescriptorFlags \*flags, ScriptContext\* scriptContext, BOOL skipPrototypeCheck /\* = FALSE \*/)

{

Assert(setterValue);

Assert(flags);

// Do a quick walk up the prototype chain to see if any of the prototypes has ever had ANY setter or non-writable property.

if (CheckIfObjectAndPrototypeChainHasOnlyWritableDataProperties(instance))

{

return FALSE;

}

RecyclableObject\* object = instance;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

\*flags = object->GetItemSetter(index, setterValue, scriptContext);

if (\*flags != None || skipPrototypeCheck)

{

break;

}

object = object->GetPrototype();

}

return ((\*flags & Accessor) == Accessor) || ((\*flags & Proxy) == Proxy) || ((\*flags & Data) == Data && (\*flags & Writable) == None);

}

BOOL JavascriptOperators::SetGlobalPropertyNoHost(wchar\_t const \* propertyName, charcount\_t propertyLength, Var value, ScriptContext \* scriptContext)

{

GlobalObject \* globalObject = scriptContext->GetGlobalObject();

uint32 index;

PropertyRecord const \* propertyRecord;

IndexType indexType = GetIndexTypeFromString(propertyName, propertyLength, scriptContext, &index, &propertyRecord, true);

if (indexType == IndexType\_Number)

{

return globalObject->DynamicObject::SetItem(index, value, PropertyOperation\_None);

}

return globalObject->DynamicObject::SetProperty(propertyRecord->GetPropertyId(), value, PropertyOperation\_None, NULL);

}

template<typename PropertyKeyType>

BOOL JavascriptOperators::SetPropertyWPCache(Var receiver, RecyclableObject\* object, PropertyKeyType propertyKey, Var newValue, ScriptContext\* requestContext, PropertyString \* propertyString, PropertyOperationFlags propertyOperationFlags)

{

if (receiver)

{

AnalysisAssert(object);

Assert(!TaggedNumber::Is(receiver));

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(object, propertyKey, &setterValueOrProxy, &flags, NULL, requestContext))

{

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

receiver = (RecyclableObject::FromVar(receiver))->GetThisObjectOrUnWrap();

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

JavascriptOperators::CallSetter(func, receiver, newValue, requestContext);

}

return TRUE;

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

auto fn = [&](RecyclableObject\* target) -> BOOL {

return JavascriptOperators::SetPropertyWPCache(receiver, target, propertyKey, newValue, requestContext, propertyString, propertyOperationFlags);

};

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetPropertyWPCacheKind, propertyKey, newValue, requestContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

requestContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

return FALSE;

}

}

else if (!JavascriptOperators::IsObject(receiver))

{

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

RecyclableObject\* receiverObject = RecyclableObject::FromVar(receiver);

if (receiver != object)

{

// If the receiver object has the property and it is an accessor then return false

PropertyDescriptor existingDesc;

if (JavascriptOperators::GetOwnPropertyDescriptor(receiverObject, propertyKey, requestContext, &existingDesc)

&& existingDesc.IsAccessorDescriptor())

{

return FALSE;

}

}

// in 9.1.9, step 5, we should return false if receiver is not object, and that will happen in default RecyclableObject operation anyhow.

PropertyValueInfo info;

if (receiverObject->SetProperty(propertyKey, newValue, propertyOperationFlags, &info))

{

if (propertyString != NULL)

{

uint16 slotIndex = info.GetPropertyIndex();

if (slotIndex != Constants::NoSlot &&

info.GetInstance() == receiverObject &&

!object->CanHaveInterceptors() &&

requestContext == receiverObject->GetScriptContext() &&

(info.GetFlags() != InlineCacheSetterFlag))

{

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

DynamicObject::FromVar(info.GetInstance())->GetTypeHandler()->PropertyIndexToInlineOrAuxSlotIndex(info.GetPropertyIndex(), &inlineOrAuxSlotIndex, &isInlineSlot);

propertyString->UpdateCache(info.GetInstance()->GetType(), inlineOrAuxSlotIndex, isInlineSlot, info.IsStoreFieldCacheEnabled());

}

}

return TRUE;

}

}

return FALSE;

}

BOOL JavascriptOperators::SetItemOnTaggedNumber(Var receiver, RecyclableObject\* object, uint32 index, Var newValue, ScriptContext\* requestContext,

PropertyOperationFlags propertyOperationFlags)

{

Assert(TaggedNumber::Is(receiver));

if (requestContext->optimizationOverrides.GetSideEffects() & SideEffects\_Accessor)

{

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

if (object == nullptr)

{

GetPropertyObject(receiver, requestContext, &object);

}

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableItem(object, index, &setterValueOrProxy, &flags, requestContext))

{

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

JavascriptOperators::CallSetter(func, receiver, newValue, requestContext);

return TRUE;

}

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

const PropertyRecord\* propertyRecord;

proxy->PropertyIdFromInt(index, &propertyRecord);

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetItemOnTaggedNumberKind, propertyRecord->GetPropertyId(), newValue, requestContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

}

}

}

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

BOOL JavascriptOperators::SetPropertyOnTaggedNumber(Var receiver, RecyclableObject\* object, PropertyId propertyId, Var newValue, ScriptContext\* requestContext,

PropertyOperationFlags propertyOperationFlags)

{

Assert (TaggedNumber::Is(receiver));

if (requestContext->optimizationOverrides.GetSideEffects() & SideEffects\_Accessor)

{

Var setterValueOrProxy = nullptr;

PropertyValueInfo info;

DescriptorFlags flags = None;

if (object == nullptr)

{

GetPropertyObject(receiver, requestContext, &object);

}

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(object, propertyId, &setterValueOrProxy, &flags, &info, requestContext))

{

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

Assert(info.GetFlags() == InlineCacheSetterFlag || info.GetPropertyIndex() == Constants::NoSlot);

JavascriptOperators::CallSetter(func, receiver, newValue, requestContext);

return TRUE;

}

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetPropertyOnTaggedNumberKind, propertyId, newValue, requestContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

}

}

}

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

requestContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

BOOL JavascriptOperators::SetPropertyUnscopable(Var instance, RecyclableObject\* receiver, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

return SetProperty\_Internal<true>(instance, receiver, false, propertyId, newValue, info, requestContext, propertyOperationFlags);

}

BOOL JavascriptOperators::SetProperty(Var receiver, RecyclableObject\* object, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

return SetProperty\_Internal<false>(receiver, object, false, propertyId, newValue, info, requestContext, propertyOperationFlags);

}

BOOL JavascriptOperators::SetRootProperty(RecyclableObject\* instance, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

return SetProperty\_Internal<false>(instance, instance, true, propertyId, newValue, info, requestContext, propertyOperationFlags);

}

template <bool unscopables>

BOOL JavascriptOperators::SetProperty\_Internal(Var receiver, RecyclableObject\* object, const bool isRoot, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

if (receiver)

{

Assert(!TaggedNumber::Is(receiver));

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

if ((isRoot && JavascriptOperators::CheckPrototypesForAccessorOrNonWritableRootProperty(object, propertyId, &setterValueOrProxy, &flags, info, requestContext)) ||

(!isRoot && JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(object, propertyId, &setterValueOrProxy, &flags, info, requestContext)))

{

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext) ||

JavascriptError::ThrowIfNotExtensibleUndefinedSetter(propertyOperationFlags, setterValueOrProxy, requestContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

Assert(!info || info->GetFlags() == InlineCacheSetterFlag || info->GetPropertyIndex() == Constants::NoSlot);

if (WithScopeObject::Is(receiver))

{

receiver = (RecyclableObject::FromVar(receiver))->GetThisObjectOrUnWrap();

}

else

{

CacheOperators::CachePropertyWrite(RecyclableObject::FromVar(receiver), isRoot, object->GetType(), propertyId, info, requestContext);

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

if (MutationBreakpoint::IsFeatureEnabled(requestContext))

{

MutationBreakpoint::HandleSetProperty(requestContext, object, propertyId, newValue);

}

#endif

JavascriptOperators::CallSetter(func, receiver, newValue, requestContext);

}

return TRUE;

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

// We can't cache the property at this time. both target and handler can be changed outside of the proxy, so the inline cache needs to be

// invalidate when target, handler, or handler prototype has changed. We don't have a way to achieve this yet.

PropertyValueInfo::SetNoCache(info, proxy);

PropertyValueInfo::DisablePrototypeCache(info, proxy); // We can't cache prototype property either

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetPropertyKind, propertyId, newValue, requestContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

if (flags & Const)

{

JavascriptError::ThrowReferenceError(requestContext, ERRAssignmentToConst);

}

JavascriptError::ThrowCantAssign(propertyOperationFlags, requestContext, propertyId);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

}

else if (!JavascriptOperators::IsObject(receiver))

{

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, requestContext);

return FALSE;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

// Break on mutation if needed

bool doNotUpdateCacheForMbp = MutationBreakpoint::IsFeatureEnabled(requestContext) ?

MutationBreakpoint::HandleSetProperty(requestContext, object, propertyId, newValue) : false;

#endif

// Get the original type before setting the property

Type \*typeWithoutProperty = object->GetType();

BOOL didSetProperty = false;

if (isRoot)

{

AssertMsg(JavascriptOperators::GetTypeId(receiver) == TypeIds\_GlobalObject

|| JavascriptOperators::GetTypeId(receiver) == TypeIds\_ModuleRoot,

"Root must be a global object!");

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(receiver);

didSetProperty = rootObject->SetRootProperty(propertyId, newValue, propertyOperationFlags, info);

}

else

{

RecyclableObject\* instanceObject = RecyclableObject::FromVar(receiver);

while (JavascriptOperators::GetTypeId(instanceObject) != TypeIds\_Null)

{

if (unscopables && JavascriptOperators::IsPropertyUnscopable(instanceObject, propertyId))

{

break;

}

else

{

didSetProperty = instanceObject->SetProperty(propertyId, newValue, propertyOperationFlags, info);

if (didSetProperty || !unscopables)

{

break;

}

}

instanceObject = JavascriptOperators::GetPrototypeNoTrap(instanceObject);

}

}

if (didSetProperty)

{

bool updateCache = true;

#ifdef ENABLE\_MUTATION\_BREAKPOINT

updateCache = updateCache && !doNotUpdateCacheForMbp;

#endif

if (updateCache)

{

if (!JavascriptProxy::Is(receiver))

{

CacheOperators::CachePropertyWrite(RecyclableObject::FromVar(receiver), isRoot, typeWithoutProperty, propertyId, info, requestContext);

}

}

return TRUE;

}

}

return FALSE;

}

BOOL JavascriptOperators::IsNumberFromNativeArray(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

Js::TypeId instanceType = JavascriptOperators::GetTypeId(instance);

// Fast path for native and typed arrays.

if ( (instanceType == TypeIds\_NativeIntArray || instanceType == TypeIds\_NativeFloatArray) || (instanceType >= TypeIds\_Int8Array && instanceType <= TypeIds\_Uint64Array) )

{

RecyclableObject\* object = RecyclableObject::FromVar(instance);

Var member;

// If the item is found in the array own body, then it is a number

if (JavascriptOperators::GetOwnItem(object, index, &member, scriptContext))

return TRUE;

}

return FALSE;

}

BOOL JavascriptOperators::GetAccessors(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, Var\* getter, Var\* setter)

{

RecyclableObject\* object = instance;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->GetAccessors(propertyId, getter, setter, requestContext))

{

\*getter = JavascriptOperators::CanonicalizeAccessor(\*getter, requestContext);

\*setter = JavascriptOperators::CanonicalizeAccessor(\*setter, requestContext);

return TRUE;

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return FALSE;

}

BOOL JavascriptOperators::SetAccessors(RecyclableObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

BOOL result = instance && instance->SetAccessors(propertyId, getter, setter, flags);

return result;

}

BOOL JavascriptOperators::OP\_SetProperty(Var instance, PropertyId propertyId, Var newValue, ScriptContext\* scriptContext, PropertyValueInfo \* info, PropertyOperationFlags flags, Var thisInstance)

{

// The call into ToObject(dynamicObject) is avoided here by checking for null and undefined and doing nothing when dynamicObject is a primitive value.

if (thisInstance == nullptr)

{

thisInstance = instance;

}

TypeId typeId = JavascriptOperators::GetTypeId(thisInstance);

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotSet\_NullOrUndefined, scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

return TRUE;

}

else if (typeId == TypeIds\_VariantDate)

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_VarDate, scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

return TRUE;

}

if (!TaggedNumber::Is(thisInstance))

{

return JavascriptOperators::SetProperty(RecyclableObject::FromVar(thisInstance), RecyclableObject::FromVar(instance), propertyId, newValue, info, scriptContext, flags);

}

JavascriptError::ThrowCantAssignIfStrictMode(flags, scriptContext);

return false;

}

BOOL JavascriptOperators::OP\_StFunctionExpression(Var obj, PropertyId propertyId, Var newValue)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

instance->SetProperty(propertyId, newValue, PropertyOperation\_None, NULL);

instance->SetWritable(propertyId, FALSE);

instance->SetConfigurable(propertyId, FALSE);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitClassMember(Var obj, PropertyId propertyId, Var newValue)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = PropertyOperation\_None;

PropertyAttributes attributes = PropertyClassMemberDefaults;

instance->SetPropertyWithAttributes(propertyId, newValue, attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitLetProperty(Var obj, PropertyId propertyId, Var newValue)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = instance->GetScriptContext()->IsUndeclBlockVar(newValue) ? PropertyOperation\_SpecialValue : PropertyOperation\_None;

PropertyAttributes attributes = PropertyLetDefaults;

if (RootObjectBase::Is(instance))

{

attributes |= PropertyLetConstGlobal;

}

instance->SetPropertyWithAttributes(propertyId, newValue, attributes, NULL, (PropertyOperationFlags)(flags | PropertyOperation\_AllowUndecl));

return TRUE;

}

BOOL JavascriptOperators::OP\_InitConstProperty(Var obj, PropertyId propertyId, Var newValue)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = instance->GetScriptContext()->IsUndeclBlockVar(newValue) ? PropertyOperation\_SpecialValue : PropertyOperation\_None;

PropertyAttributes attributes = PropertyConstDefaults;

if (RootObjectBase::Is(instance))

{

attributes |= PropertyLetConstGlobal;

}

instance->SetPropertyWithAttributes(propertyId, newValue, attributes, NULL, (PropertyOperationFlags)(flags | PropertyOperation\_AllowUndecl));

return TRUE;

}

BOOL JavascriptOperators::OP\_InitUndeclRootLetProperty(Var obj, PropertyId propertyId)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = PropertyLetDefaults | PropertyLetConstGlobal;

instance->SetPropertyWithAttributes(propertyId, instance->GetLibrary()->GetUndeclBlockVar(), attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitUndeclRootConstProperty(Var obj, PropertyId propertyId)

{

RecyclableObject\* instance = RecyclableObject::FromVar(obj);

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = PropertyConstDefaults | PropertyLetConstGlobal;

instance->SetPropertyWithAttributes(propertyId, instance->GetLibrary()->GetUndeclBlockVar(), attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitUndeclConsoleLetProperty(Var obj, PropertyId propertyId)

{

FrameDisplay \*pScope = (FrameDisplay\*)obj;

AssertMsg(ConsoleScopeActivationObject::Is((DynamicObject\*)pScope->GetItem(pScope->GetLength() - 1)), "How come we got this opcode without ConsoleScopeActivationObject?");

RecyclableObject\* instance = RecyclableObject::FromVar(pScope->GetItem(0));

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = PropertyLetDefaults;

instance->SetPropertyWithAttributes(propertyId, instance->GetLibrary()->GetUndeclBlockVar(), attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::OP\_InitUndeclConsoleConstProperty(Var obj, PropertyId propertyId)

{

FrameDisplay \*pScope = (FrameDisplay\*)obj;

AssertMsg(ConsoleScopeActivationObject::Is((DynamicObject\*)pScope->GetItem(pScope->GetLength() - 1)), "How come we got this opcode without ConsoleScopeActivationObject?");

RecyclableObject\* instance = RecyclableObject::FromVar(pScope->GetItem(0));

PropertyOperationFlags flags = static\_cast<PropertyOperationFlags>(PropertyOperation\_SpecialValue | PropertyOperation\_AllowUndecl);

PropertyAttributes attributes = PropertyConstDefaults;

instance->SetPropertyWithAttributes(propertyId, instance->GetLibrary()->GetUndeclBlockVar(), attributes, NULL, flags);

return TRUE;

}

BOOL JavascriptOperators::InitProperty(RecyclableObject\* instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

return instance && instance->InitProperty(propertyId, newValue, flags);

}

BOOL JavascriptOperators::OP\_InitProperty(Var instance, PropertyId propertyId, Var newValue)

{

if(TaggedNumber::Is(instance)) { return false; }

return JavascriptOperators::InitProperty(RecyclableObject::FromVar(instance), propertyId, newValue);

}

BOOL JavascriptOperators::DeleteProperty(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

return DeleteProperty\_Impl<false>(instance, propertyId, propertyOperationFlags);

}

BOOL JavascriptOperators::DeletePropertyUnscopables(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

return DeleteProperty\_Impl<true>(instance, propertyId, propertyOperationFlags);

}

template<bool unscopables>

BOOL JavascriptOperators::DeleteProperty\_Impl(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

if (unscopables && JavascriptOperators::IsPropertyUnscopable(instance, propertyId))

{

return false;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

ScriptContext \*scriptContext = instance->GetScriptContext();

if (MutationBreakpoint::IsFeatureEnabled(scriptContext)

&& scriptContext->HasMutationBreakpoints())

{

MutationBreakpoint::HandleDeleteProperty(scriptContext, instance, propertyId);

}

#endif

// !unscopables will hit the return statement on the first iteration

return instance->DeleteProperty(propertyId, propertyOperationFlags);

}

Var JavascriptOperators::OP\_DeleteProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

if(TaggedNumber::Is(instance))

{

return scriptContext->GetLibrary()->GetTrue();

}

TypeId typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotDelete\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

RecyclableObject \*recyclableObject = RecyclableObject::FromVar(instance);

return scriptContext->GetLibrary()->CreateBoolean(

JavascriptOperators::DeleteProperty(recyclableObject, propertyId, propertyOperationFlags));

}

Var JavascriptOperators::OP\_DeleteRootProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

AssertMsg(RootObjectBase::Is(instance), "Root must be a global object!");

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(instance);

return scriptContext->GetLibrary()->CreateBoolean(

rootObject->DeleteRootProperty(propertyId, propertyOperationFlags));

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchSetPropertyScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags)

{

// Set the property using a scope stack rather than an individual instance.

// Walk the stack until we find an instance that has the property and store

// the new value there.

//

// To propagate 'this' pointer, walk up the stack and update scopes

// where field '\_lexicalThisSlotSymbol' exists and stop at the

// scope where field '\_lexicalNewTargetSymbol' also exists, which

// indicates class constructor.

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

uint16 length = pDisplay->GetLength();

DynamicObject \*object;

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

bool allowUndecInConsoleScope = (propertyOperationFlags & PropertyOperation\_AllowUndeclInConsoleScope) == PropertyOperation\_AllowUndeclInConsoleScope;

bool isLexicalThisSlotSymbol = (propertyId == PropertyIds::\_lexicalThisSlotSymbol);

for (uint16 i = 0; i < length; i++)

{

object = (DynamicObject\*)pDisplay->GetItem(i);

AssertMsg(!ConsoleScopeActivationObject::Is(object) || (i == length - 1), "Invalid location for ConsoleScopeActivationObject");

Type\* type = object->GetType();

if (CacheOperators::TrySetProperty<true, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, propertyOperationFlags, nullptr, &info))

{

if (isLexicalThisSlotSymbol && !JavascriptOperators::HasProperty(object, PropertyIds::\_lexicalNewTargetSymbol))

{

continue;

}

return;

}

// In scoped set property, we need to set the property when it is available; it could be a setter

// or normal property. we need to check setter first, and if no setter is available, but HasProperty

// is true, this must be a normal property.

// TODO: merge OP\_HasProperty and GetSetter in one pass if there is perf problem. In fastDOM we have quite

// a lot of setters so separating the two might be actually faster.

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(object, propertyId, &setterValueOrProxy, &flags, &info, scriptContext))

{

if ((flags & Accessor) == Accessor)

{

if (setterValueOrProxy)

{

JavascriptFunction\* func = (JavascriptFunction\*)setterValueOrProxy;

Assert(info.GetFlags() == InlineCacheSetterFlag || info.GetPropertyIndex() == Constants::NoSlot);

CacheOperators::CachePropertyWrite(object, false, type, propertyId, &info, scriptContext);

JavascriptOperators::CallSetter(func, object, newValue, scriptContext);

}

Assert(!isLexicalThisSlotSymbol);

return;

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

auto fn = [&](RecyclableObject\* target) -> BOOL {

return JavascriptOperators::SetProperty(object, target, propertyId, newValue, scriptContext, propertyOperationFlags);

};

// We can't cache the property at this time. both target and handler can be changed outside of the proxy, so the inline cache needs to be

// invalidate when target, handler, or handler prototype has changed. We don't have a way to achieve this yet.

PropertyValueInfo::SetNoCache(&info, proxy);

PropertyValueInfo::DisablePrototypeCache(&info, proxy); // We can't cache prototype property either

proxy->SetPropertyTrap(object, JavascriptProxy::SetPropertyTrapKind::SetPropertyKind, propertyId, newValue, scriptContext);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

if (!allowUndecInConsoleScope)

{

if (flags & Const)

{

JavascriptError::ThrowReferenceError(scriptContext, ERRAssignmentToConst);

}

Assert(!isLexicalThisSlotSymbol);

return;

}

}

}

else if (!JavascriptOperators::IsObject(object))

{

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, scriptContext);

}

// Need to do a "get" of the current value (if any) to make sure that we're not writing to

// let/const before declaration, but we need to disable implicit calls around the "get",

// so we need to do a "has" first to make sure the "get" is valid (e.g., "get" on a HostDispatch

// with implicit calls disabled will always "succeed").

if (JavascriptOperators::HasProperty(object, propertyId))

{

if (scriptContext->GetConfig()->IsLetAndConstEnabled())

{

DisableImplicitFlags disableImplicitFlags =

scriptContext->GetThreadContext()->GetDisableImplicitFlags();

scriptContext->GetThreadContext()->SetDisableImplicitFlags(DisableImplicitCallAndExceptionFlag);

Var value;

BOOL result = JavascriptOperators::GetProperty(object, propertyId, &value, scriptContext, nullptr);

scriptContext->GetThreadContext()->SetDisableImplicitFlags(disableImplicitFlags);

if (result && scriptContext->IsUndeclBlockVar(value) && !allowUndecInConsoleScope && !isLexicalThisSlotSymbol)

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

PropertyOperationFlags setPropertyOpFlags = allowUndecInConsoleScope ? PropertyOperation\_AllowUndeclInConsoleScope : PropertyOperation\_None;

object->SetProperty(propertyId, newValue, setPropertyOpFlags, &info);

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchSetPropertyScoped", propertyId, scriptContext, object);

}

#endif

if (!JavascriptProxy::Is(object) && !allowUndecInConsoleScope)

{

CacheOperators::CachePropertyWrite(object, false, type, propertyId, &info, scriptContext);

}

if (isLexicalThisSlotSymbol && !JavascriptOperators::HasProperty(object, PropertyIds::\_lexicalNewTargetSymbol))

{

continue;

}

return;

}

}

Assert(!isLexicalThisSlotSymbol);

// If we have console scope and no one in the scope had the property add it to console scope

if ((length > 0) && ConsoleScopeActivationObject::Is(pDisplay->GetItem(length - 1)))

{

RecyclableObject\* obj = RecyclableObject::FromVar((DynamicObject\*)pDisplay->GetItem(length - 1));

OUTPUT\_TRACE(Js::ConsoleScopePhase, L"Adding property '%s' to console scope object\n", scriptContext->GetPropertyName(propertyId)->GetBuffer());

JavascriptOperators::SetProperty(obj, obj, propertyId, newValue, scriptContext, propertyOperationFlags);

return;

}

// No one in the scope stack has the property, so add it to the default instance provided by the caller.

AssertMsg(!TaggedNumber::Is(defaultInstance), "Root object is an int or tagged float?");

Assert(defaultInstance != nullptr);

RecyclableObject\* obj = RecyclableObject::FromVar(defaultInstance);

{

//SetPropertyScoped does not use inline cache for default instance

PropertyValueInfo info;

JavascriptOperators::SetRootProperty(obj, propertyId, newValue, &info, scriptContext, (PropertyOperationFlags)(propertyOperationFlags | PropertyOperation\_Root));

}

}

template void JavascriptOperators::PatchSetPropertyScoped<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags);

template void JavascriptOperators::PatchSetPropertyScoped<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags);

template void JavascriptOperators::PatchSetPropertyScoped<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags);

template void JavascriptOperators::PatchSetPropertyScoped<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags propertyOperationFlags);

BOOL JavascriptOperators::OP\_InitFuncScoped(FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, ScriptContext\* scriptContext)

{

int i;

int length = pScope->GetLength();

DynamicObject \*obj;

for (i = 0; i < length; i++)

{

obj = (DynamicObject\*)pScope->GetItem(i);

if (obj->InitFuncScoped(propertyId, newValue))

{

return TRUE;

}

}

AssertMsg(!TaggedNumber::Is(defaultInstance), "Root object is an int or tagged float?");

return RecyclableObject::FromVar(defaultInstance)->InitFuncScoped(propertyId, newValue);

}

BOOL JavascriptOperators::OP\_InitPropertyScoped(FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, ScriptContext\* scriptContext)

{

int i;

int length = pScope->GetLength();

DynamicObject \*obj;

for (i = 0; i < length; i++)

{

obj = (DynamicObject\*)pScope->GetItem(i);

if (obj->InitPropertyScoped(propertyId, newValue))

{

return TRUE;

}

}

AssertMsg(!TaggedNumber::Is(defaultInstance), "Root object is an int or tagged float?");

return RecyclableObject::FromVar(defaultInstance)->InitPropertyScoped(propertyId, newValue);

}

Var JavascriptOperators::OP\_DeletePropertyScoped(

FrameDisplay \*pScope,

PropertyId propertyId,

Var defaultInstance,

ScriptContext\* scriptContext,

PropertyOperationFlags propertyOperationFlags)

{

int i;

int length = pScope->GetLength();

for (i = 0; i < length; i++)

{

DynamicObject \*obj = (DynamicObject\*)pScope->GetItem(i);

if (JavascriptOperators::HasProperty(obj, propertyId))

{

return scriptContext->GetLibrary()->CreateBoolean(JavascriptOperators::DeleteProperty(obj, propertyId, propertyOperationFlags));

}

}

return JavascriptOperators::OP\_DeleteRootProperty(RecyclableObject::FromVar(defaultInstance), propertyId, scriptContext, propertyOperationFlags);

}

Var JavascriptOperators::OP\_TypeofPropertyScoped(FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext)

{

int i;

int length = pScope->GetLength();

for (i = 0; i < length; i++)

{

DynamicObject \*obj = (DynamicObject\*)pScope->GetItem(i);

if (JavascriptOperators::HasProperty(obj, propertyId))

{

return JavascriptOperators::TypeofFld(obj, propertyId, scriptContext);

}

}

return JavascriptOperators::TypeofRootFld(RecyclableObject::FromVar(defaultInstance), propertyId, scriptContext);

}

BOOL JavascriptOperators::HasOwnItem(RecyclableObject\* object, uint32 index)

{

return object->HasOwnItem(index);

}

BOOL JavascriptOperators::HasItem(RecyclableObject\* object, uint64 index)

{

PropertyRecord const \* propertyRecord;

ScriptContext\* scriptContext = object->GetScriptContext();

JavascriptOperators::GetPropertyIdForInt(index, scriptContext, &propertyRecord);

return JavascriptOperators::HasProperty(object, propertyRecord->GetPropertyId());

}

BOOL JavascriptOperators::HasItem(RecyclableObject\* object, uint32 index)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(object);

#endif

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->HasItem(index))

{

return true;

}

// CONSIDER: Numeric property values shouldn't be on the prototype for now but if this changes

// we should add SkipsPrototype support here as well

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return false;

}

BOOL JavascriptOperators::GetOwnItem(RecyclableObject\* object, uint32 index, Var\* value, ScriptContext\* requestContext)

{

return object->GetItem(object, index, value, requestContext);

}

BOOL JavascriptOperators::GetItem(Var instance, RecyclableObject\* propertyObject, uint32 index, Var\* value, ScriptContext\* requestContext)

{

RecyclableObject\* object = propertyObject;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->GetItem(instance, index, value, requestContext))

{

return true;

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return false;

}

BOOL JavascriptOperators::GetItemReference(Var instance, RecyclableObject\* propertyObject, uint32 index, Var\* value, ScriptContext\* requestContext)

{

RecyclableObject\* object = propertyObject;

while (JavascriptOperators::GetTypeId(object) != TypeIds\_Null)

{

if (object->GetItemReference(instance, index, value, requestContext))

{

return true;

}

if (object->SkipsPrototype())

{

break;

}

object = JavascriptOperators::GetPrototypeNoTrap(object);

}

return false;

}

BOOL JavascriptOperators::SetItem(Var receiver, RecyclableObject\* object, uint64 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

PropertyRecord const \* propertyRecord;

JavascriptOperators::GetPropertyIdForInt(index, scriptContext, &propertyRecord);

return JavascriptOperators::SetProperty(receiver, object, propertyRecord->GetPropertyId(), value, scriptContext, propertyOperationFlags);

}

BOOL JavascriptOperators::SetItem(Var receiver, RecyclableObject\* object, uint32 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags, BOOL skipPrototypeCheck /\* = FALSE \*/)

{

Var setterValueOrProxy = nullptr;

DescriptorFlags flags = None;

Assert(!TaggedNumber::Is(receiver));

if (JavascriptOperators::CheckPrototypesForAccessorOrNonWritableItem(object, index, &setterValueOrProxy, &flags, scriptContext, skipPrototypeCheck))

{

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

if ((flags & Accessor) == Accessor)

{

if (JavascriptError::ThrowIfStrictModeUndefinedSetter(propertyOperationFlags, setterValueOrProxy, scriptContext) ||

JavascriptError::ThrowIfNotExtensibleUndefinedSetter(propertyOperationFlags, setterValueOrProxy, scriptContext))

{

return TRUE;

}

if (setterValueOrProxy)

{

RecyclableObject\* func = RecyclableObject::FromVar(setterValueOrProxy);

JavascriptOperators::CallSetter(func, receiver, value, scriptContext);

}

return TRUE;

}

else if ((flags & Proxy) == Proxy)

{

Assert(JavascriptProxy::Is(setterValueOrProxy));

JavascriptProxy\* proxy = JavascriptProxy::FromVar(setterValueOrProxy);

const PropertyRecord\* propertyRecord;

proxy->PropertyIdFromInt(index, &propertyRecord);

return proxy->SetPropertyTrap(receiver, JavascriptProxy::SetPropertyTrapKind::SetItemKind, propertyRecord->GetPropertyId(), value, scriptContext, skipPrototypeCheck);

}

else

{

Assert((flags & Data) == Data && (flags & Writable) == None);

if ((propertyOperationFlags & PropertyOperationFlags::PropertyOperation\_ThrowIfNotExtensible) == PropertyOperationFlags::PropertyOperation\_ThrowIfNotExtensible)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NonExtensibleObject);

}

JavascriptError::ThrowCantAssign(propertyOperationFlags, scriptContext, index);

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, scriptContext);

return FALSE;

}

}

else if (!JavascriptOperators::IsObject(receiver))

{

JavascriptError::ThrowCantAssignIfStrictMode(propertyOperationFlags, scriptContext);

return FALSE;

}

return (RecyclableObject::FromVar(receiver))->SetItem(index, value, propertyOperationFlags);

}

BOOL JavascriptOperators::DeleteItem(RecyclableObject\* object, uint32 index, PropertyOperationFlags propertyOperationFlags)

{

return object->DeleteItem(index, propertyOperationFlags);

}

BOOL JavascriptOperators::DeleteItem(RecyclableObject\* object, uint64 index, PropertyOperationFlags propertyOperationFlags)

{

PropertyRecord const \* propertyRecord;

JavascriptOperators::GetPropertyIdForInt(index, object->GetScriptContext(), &propertyRecord);

return JavascriptOperators::DeleteProperty(object, propertyRecord->GetPropertyId(), propertyOperationFlags);

}

BOOL JavascriptOperators::OP\_HasItem(Var instance, Var index, ScriptContext\* scriptContext)

{

RecyclableObject\* object = TaggedNumber::Is(instance) ?

scriptContext->GetLibrary()->GetNumberPrototype() :

RecyclableObject::FromVar(instance);

uint32 indexVal;

PropertyRecord const \* propertyRecord;

bool createIfNotFound = (DynamicType::Is(object->GetTypeId()) &&

static\_cast<DynamicObject\*>(object)->GetTypeHandler()->IsStringTypeHandler()) ||

JavascriptProxy::Is(object);

if (GetIndexType(index, scriptContext, &indexVal, &propertyRecord, createIfNotFound) == IndexType\_Number)

{

return HasItem(object, indexVal);

}

else if (propertyRecord == nullptr)

{

Assert(IsJsNativeObject(object));

#if DBG

JavascriptString\* indexStr = JavascriptConversion::ToString(index, scriptContext);

PropertyRecord const \* debugPropertyRecord;

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &debugPropertyRecord);

AssertMsg(!JavascriptOperators::HasProperty(object, debugPropertyRecord->GetPropertyId()), "how did this property come? See OS Bug 2727708 if you see this come from the web");

#endif

return FALSE;

}

else

{

return HasProperty(object, propertyRecord->GetPropertyId());

}

}

#if ENABLE\_PROFILE\_INFO

void JavascriptOperators::UpdateNativeArrayProfileInfoToCreateVarArray(Var instance, const bool expectingNativeFloatArray, const bool expectingVarArray)

{

Assert(instance);

Assert(expectingNativeFloatArray ^ expectingVarArray);

if (!JavascriptNativeArray::Is(instance))

{

return;

}

ArrayCallSiteInfo \*const arrayCallSiteInfo = JavascriptNativeArray::FromVar(instance)->GetArrayCallSiteInfo();

if (!arrayCallSiteInfo)

{

return;

}

if (expectingNativeFloatArray)

{

// Profile data is expecting a native float array. Ensure that at the array's creation site, that a native int array

// is not created, such that the profiled array type would be correct.

arrayCallSiteInfo->SetIsNotNativeIntArray();

}

else

{

// Profile data is expecting a var array. Ensure that at the array's creation site, that a native array is not

// created, such that the profiled array type would be correct.

Assert(expectingVarArray);

arrayCallSiteInfo->SetIsNotNativeArray();

}

}

bool JavascriptOperators::SetElementMayHaveImplicitCalls(ScriptContext \*const scriptContext)

{

return

scriptContext->optimizationOverrides.GetArraySetElementFastPathVtable() ==

ScriptContextOptimizationOverrideInfo::InvalidVtable;

}

#endif

RecyclableObject \*JavascriptOperators::GetCallableObjectOrThrow(const Var callee, ScriptContext \*const scriptContext)

{

Assert(callee);

Assert(scriptContext);

if (TaggedNumber::Is(callee))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction /\* TODO-ERROR: get arg name - aFunc \*/);

}

return RecyclableObject::FromVar(callee);

}

#if ENABLE\_NATIVE\_CODEGEN

Var JavascriptOperators::OP\_GetElementI\_JIT(Var instance, Var index, ScriptContext \*scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

return OP\_GetElementI(instance, index, scriptContext);

}

#else

Var JavascriptOperators::OP\_GetElementI\_JIT(Var instance, Var index, ScriptContext \*scriptContext)

{

return OP\_GetElementI(instance, index, scriptContext);

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

Var JavascriptOperators::OP\_GetElementI\_JIT\_ExpectingNativeFloatArray(Var instance, Var index, ScriptContext \*scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

UpdateNativeArrayProfileInfoToCreateVarArray(instance, true, false);

return OP\_GetElementI\_JIT(instance, index, scriptContext);

}

Var JavascriptOperators::OP\_GetElementI\_JIT\_ExpectingVarArray(Var instance, Var index, ScriptContext \*scriptContext)

{

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

UpdateNativeArrayProfileInfoToCreateVarArray(instance, false, true);

return OP\_GetElementI\_JIT(instance, index, scriptContext);

}

#endif

Var JavascriptOperators::OP\_GetElementI\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetElementI\_JIT(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetElementI\_JIT(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::OP\_GetElementI\_UInt32\_ExpectingNativeFloatArray(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_PROFILE\_INFO

UpdateNativeArrayProfileInfoToCreateVarArray(instance, true, false);

#endif

return OP\_GetElementI\_UInt32(instance, index, scriptContext);

}

Var JavascriptOperators::OP\_GetElementI\_UInt32\_ExpectingVarArray(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_PROFILE\_INFO

UpdateNativeArrayProfileInfoToCreateVarArray(instance, false, true);

#endif

return OP\_GetElementI\_UInt32(instance, index, scriptContext);

}

Var JavascriptOperators::OP\_GetElementI\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetElementI\_JIT(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetElementI\_JIT(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::OP\_GetElementI\_Int32\_ExpectingNativeFloatArray(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_PROFILE\_INFO

UpdateNativeArrayProfileInfoToCreateVarArray(instance, true, false);

#endif

return OP\_GetElementI\_Int32(instance, index, scriptContext);

}

Var JavascriptOperators::OP\_GetElementI\_Int32\_ExpectingVarArray(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if ENABLE\_PROFILE\_INFO

UpdateNativeArrayProfileInfoToCreateVarArray(instance, false, true);

#endif

return OP\_GetElementI\_Int32(instance, index, scriptContext);

}

BOOL JavascriptOperators::GetItemFromArrayPrototype(JavascriptArray \* arr, int32 indexInt, Var \* result, ScriptContext \* scriptContext)

{

// try get from Array prototype

RecyclableObject\* prototype = arr->GetPrototype();

if (JavascriptOperators::GetTypeId(prototype) != TypeIds\_Array) //This can be TypeIds\_ES5Array (or any other object changed through \_\_proto\_\_).

{

return false;

}

JavascriptArray\* arrayPrototype = JavascriptArray::FromVar(prototype); //Prototype must be Array.prototype (unless changed through \_\_proto\_\_)

AssertMsg(scriptContext->GetConfig()->Is\_\_proto\_\_Enabled()

|| arrayPrototype->GetScriptContext()->GetLibrary()->GetArrayPrototype() == arrayPrototype, "This function is supported only for [[class]] Array");

if (arrayPrototype->GetLength() && arrayPrototype->GetItem(arrayPrototype, (uint32)indexInt, result, scriptContext))

{

return true;

}

prototype = arrayPrototype->GetPrototype(); //Its prototype must be Object.prototype (unless changed through \_\_proto\_\_)

AssertMsg(scriptContext->GetConfig()->Is\_\_proto\_\_Enabled()

|| prototype->GetScriptContext()->GetLibrary()->GetObjectPrototype() == prototype, "This function is supported only for [[class]] Array");

if (prototype->GetScriptContext()->GetLibrary()->GetObjectPrototype() != prototype)

{

return false;

}

if (DynamicObject::FromVar(prototype)->HasNonEmptyObjectArray())

{

if (prototype->GetItem(arr, (uint32)indexInt, result, scriptContext))

{

return true;

}

}

\*result = scriptContext->GetMissingItemResult(arr, indexInt);

return true;

}

template <typename T>

BOOL JavascriptOperators::OP\_GetElementI\_ArrayFastPath(T \* arr, int indexInt, Var \* result, ScriptContext \* scriptContext)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(arr);

#endif

if (indexInt >= 0)

{

if (!CrossSite::IsCrossSiteObjectTyped(arr))

{

if (arr->T::DirectGetVarItemAt((uint32)indexInt, result, scriptContext))

{

return true;

}

}

else

{

if (arr->GetItem(arr, (uint32)indexInt, result, scriptContext))

{

return true;

}

}

return GetItemFromArrayPrototype(arr, indexInt, result, scriptContext);

}

return false;

}

Var JavascriptOperators::OP\_GetElementI(Var instance, Var index, ScriptContext\* scriptContext)

{

JavascriptString \*temp = NULL;

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

if (TaggedInt::Is(index))

{

TaggedIntIndex:

switch (JavascriptOperators::GetTypeId(instance))

{

case TypeIds\_Array: //fast path for array

{

Var result;

if (OP\_GetElementI\_ArrayFastPath(JavascriptArray::FromVar(instance), TaggedInt::ToInt32(index), &result, scriptContext))

{

return result;

}

break;

}

case TypeIds\_NativeIntArray:

{

Var result;

if (OP\_GetElementI\_ArrayFastPath(JavascriptNativeIntArray::FromVar(instance), TaggedInt::ToInt32(index), &result, scriptContext))

{

return result;

}

break;

}

case TypeIds\_NativeFloatArray:

{

Var result;

if (OP\_GetElementI\_ArrayFastPath(JavascriptNativeFloatArray::FromVar(instance), TaggedInt::ToInt32(index), &result, scriptContext))

{

return result;

}

break;

}

case TypeIds\_String: // fast path for string

{

charcount\_t indexInt = TaggedInt::ToUInt32(index);

JavascriptString\* string = JavascriptString::FromVar(instance);

Var result;

if (string->JavascriptString::GetItem(instance, indexInt, &result, scriptContext))

{

return result;

}

break;

}

case TypeIds\_Int8Array:

{

// The typed array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Int8VirtualArray>::HasVirtualTable(instance))

{

Int8VirtualArray\* int8Array = Int8VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int8Array) && indexInt >= 0)

{

return int8Array->DirectGetItem(indexInt);

}

}

else

{

Int8Array\* int8Array = Int8Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int8Array) && indexInt >= 0)

{

return int8Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Uint8Array:

{

// The typed array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Uint8VirtualArray>::HasVirtualTable(instance))

{

Uint8VirtualArray\* uint8Array = Uint8VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8Array) && indexInt >= 0)

{

return uint8Array->DirectGetItem(indexInt);

}

}

else

{

Uint8Array\* uint8Array = Uint8Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8Array) && indexInt >= 0)

{

return uint8Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Uint8ClampedArray:

{

// The typed array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Uint8ClampedVirtualArray>::HasVirtualTable(instance))

{

Uint8ClampedVirtualArray\* uint8ClampedArray = Uint8ClampedVirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8ClampedArray) && indexInt >= 0)

{

return uint8ClampedArray->DirectGetItem(indexInt);

}

}

else

{

Uint8ClampedArray\* uint8ClampedArray = Uint8ClampedArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8ClampedArray) && indexInt >= 0)

{

return uint8ClampedArray->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Int16Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Int16VirtualArray>::HasVirtualTable(instance))

{

Int16VirtualArray\* int16Array = Int16VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int16Array) && indexInt >= 0)

{

return int16Array->DirectGetItem(indexInt);

}

}

else

{

Int16Array\* int16Array = Int16Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int16Array) && indexInt >= 0)

{

return int16Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Uint16Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Uint16VirtualArray>::HasVirtualTable(instance))

{

Uint16VirtualArray\* uint16Array = Uint16VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint16Array) && indexInt >= 0)

{

return uint16Array->DirectGetItem(indexInt);

}

}

else

{

Uint16Array\* uint16Array = Uint16Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint16Array) && indexInt >= 0)

{

return uint16Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Int32Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Int32VirtualArray>::HasVirtualTable(instance))

{

Int32VirtualArray\* int32Array = Int32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int32Array) && indexInt >= 0)

{

return int32Array->DirectGetItem(indexInt);

}

}

else

{

Int32Array\* int32Array = Int32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int32Array) && indexInt >= 0)

{

return int32Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Uint32Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Uint32VirtualArray>::HasVirtualTable(instance))

{

Uint32VirtualArray\* uint32Array = Uint32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint32Array) && indexInt >= 0)

{

return uint32Array->DirectGetItem(indexInt);

}

}

else

{

Uint32Array\* uint32Array = Uint32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint32Array) && indexInt >= 0)

{

return uint32Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Float32Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Float32VirtualArray>::HasVirtualTable(instance))

{

Float32VirtualArray\* float32Array = Float32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float32Array) && indexInt >= 0)

{

return float32Array->DirectGetItem(indexInt);

}

}

else

{

Float32Array\* float32Array = Float32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float32Array) && indexInt >= 0)

{

return float32Array->DirectGetItem(indexInt);

}

}

break;

}

case TypeIds\_Float64Array:

{

// The type array will deal with all possible values for the index

int32 indexInt = TaggedInt::ToInt32(index);

if (VirtualTableInfo<Float64VirtualArray>::HasVirtualTable(instance))

{

Float64VirtualArray\* float64Array = Float64VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float64Array) && indexInt >= 0)

{

return float64Array->DirectGetItem(indexInt);

}

}

else

{

Float64Array\* float64Array = Float64Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float64Array) && indexInt >= 0)

{

return float64Array->DirectGetItem(indexInt);

}

}

break;

}

default:

break;

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(index))

{

uint32 uint32Index = JavascriptConversion::ToUInt32(index, scriptContext);

if ((double)uint32Index == JavascriptNumber::GetValue(index) && !TaggedInt::IsOverflow(uint32Index))

{

index = TaggedInt::ToVarUnchecked(uint32Index);

goto TaggedIntIndex;

}

}

else if (JavascriptString::Is(index)) // fastpath for PropertyStrings

{

temp = JavascriptString::FromVar(index);

Assert(temp->GetScriptContext() == scriptContext);

if (VirtualTableInfo<Js::PropertyString>::HasVirtualTable(temp))

{

PropertyString \* propertyString = (PropertyString\*)temp;

PropertyCache const \*cache = propertyString->GetPropertyCache();

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

JavascriptString::FromVar(index)->GetSz());

}

if (object->GetType() == cache->type)

{

#if DBG\_DUMP

scriptContext->forinCache++;

#endif

Assert(object->GetScriptContext() == scriptContext);

Var value;

if (cache->isInlineSlot)

{

value = DynamicObject::FromVar(object)->GetInlineSlot(cache->dataSlotIndex);

}

else

{

value = DynamicObject::FromVar(object)->GetAuxSlot(cache->dataSlotIndex);

}

Assert(!CrossSite::NeedMarshalVar(value, scriptContext));

Assert(value == JavascriptOperators::GetProperty(object, propertyString->GetPropertyRecord()->GetPropertyId(), scriptContext)

|| value == JavascriptOperators::GetRootProperty(object, propertyString->GetPropertyRecord()->GetPropertyId(), scriptContext));

return value;

}

#if DBG\_DUMP

scriptContext->forinNoCache++;

#endif

PropertyRecord const \* propertyRecord = propertyString->GetPropertyRecord();

Var value;

if (propertyRecord->IsNumeric())

{

if (JavascriptOperators::GetItem(instance, object, propertyRecord->GetNumericValue(), &value, scriptContext))

{

return value;

}

}

else

{

if (JavascriptOperators::GetPropertyWPCache(instance, object, propertyRecord->GetPropertyId(), &value, scriptContext, propertyString))

{

return value;

}

}

return scriptContext->GetLibrary()->GetUndefined();

}

#if DBG\_DUMP

scriptContext->forinNoCache++;

#endif

}

return JavascriptOperators::GetElementIHelper(instance, index, instance, scriptContext);

}

Var JavascriptOperators::GetElementIHelper(Var instance, Var index, Var receiver, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

uint32 indexVal;

PropertyRecord const \* propertyRecord;

JavascriptString \* propertyNameString;

Var value;

bool createIfNotFound = !IsJsNativeObject(object);

IndexType indexType = GetIndexType(index, scriptContext, &indexVal, &propertyRecord, &propertyNameString, createIfNotFound, true);

if (indexType == IndexType\_Number)

{

if (JavascriptOperators::GetItem(receiver, object, indexVal, &value, scriptContext))

{

return value;

}

}

else if (indexType == IndexType\_JavascriptString)

{

if (JavascriptOperators::GetPropertyWPCache(receiver, object, propertyNameString, &value, scriptContext, nullptr))

{

return value;

}

}

else

{

// We called GetIndexType with preferJavascriptString as true, so we mush have a propertyRecord

Assert(indexType == IndexType\_PropertyId);

Assert(propertyRecord);

if (JavascriptOperators::GetPropertyWPCache(receiver, object, propertyRecord->GetPropertyId(), &value, scriptContext, nullptr))

{

return value;

}

}

return scriptContext->GetMissingItemResult(object, indexVal);

}

int32 JavascriptOperators::OP\_GetNativeIntElementI(Var instance, Var index)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

if (TaggedInt::Is(index))

{

int32 indexInt = TaggedInt::ToInt32(index);

if (indexInt < 0)

{

return JavascriptNativeIntArray::MissingItem;

}

JavascriptArray \* arr = JavascriptArray::FromVar(instance);

int32 result;

if (arr->DirectGetItemAt((uint32)indexInt, &result))

{

return result;

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(index))

{

int32 indexInt;

bool isInt32;

double dIndex = JavascriptNumber::GetValue(index);

if (JavascriptNumber::TryGetInt32OrUInt32Value(dIndex, &indexInt, &isInt32))

{

if (isInt32 && indexInt < 0)

{

return JavascriptNativeIntArray::MissingItem;

}

JavascriptArray \* arr = JavascriptArray::FromVar(instance);

int32 result;

if (arr->DirectGetItemAt((uint32)indexInt, &result))

{

return result;

}

}

}

else

{

AssertMsg(false, "Non-numerical index in this helper?");

}

return JavascriptNativeIntArray::MissingItem;

}

int32 JavascriptOperators::OP\_GetNativeIntElementI\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetNativeIntElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext));

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetNativeIntElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer));

#endif

}

int32 JavascriptOperators::OP\_GetNativeIntElementI\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetNativeIntElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext));

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetNativeIntElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer));

#endif

}

double JavascriptOperators::OP\_GetNativeFloatElementI(Var instance, Var index)

{

double result = 0;

if (TaggedInt::Is(index))

{

int32 indexInt = TaggedInt::ToInt32(index);

if (indexInt < 0)

{

result = JavascriptNativeFloatArray::MissingItem;

}

else

{

JavascriptArray \* arr = JavascriptArray::FromVar(instance);

if (!arr->DirectGetItemAt((uint32)indexInt, &result))

{

result = JavascriptNativeFloatArray::MissingItem;

}

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(index))

{

int32 indexInt;

bool isInt32;

double dIndex = JavascriptNumber::GetValue(index);

if (JavascriptNumber::TryGetInt32OrUInt32Value(dIndex, &indexInt, &isInt32))

{

if (isInt32 && indexInt < 0)

{

result = JavascriptNativeFloatArray::MissingItem;

}

else

{

JavascriptArray \* arr = JavascriptArray::FromVar(instance);

if (!arr->DirectGetItemAt((uint32)indexInt, &result))

{

result = JavascriptNativeFloatArray::MissingItem;

}

}

}

}

else

{

AssertMsg(false, "Non-numerical index in this helper?");

}

return result;

}

double JavascriptOperators::OP\_GetNativeFloatElementI\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetNativeFloatElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext));

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetNativeFloatElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer));

#endif

}

double JavascriptOperators::OP\_GetNativeFloatElementI\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetNativeFloatElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext));

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetNativeFloatElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer));

#endif

}

Var JavascriptOperators::OP\_GetMethodElement\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetMethodElement(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetMethodElement(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::OP\_GetMethodElement\_Int32(Var instance, int32 index, ScriptContext\* scriptContext)

{

#if FLOATVAR

return OP\_GetElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_GetMethodElement(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext);

#endif

}

Var JavascriptOperators::OP\_GetMethodElement(Var instance, Var index, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

ThreadContext\* threadContext = scriptContext->GetThreadContext();

ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

threadContext->ClearImplicitCallFlags();

uint32 indexVal;

PropertyRecord const \* propertyRecord;

Var value = NULL;

BOOL hasProperty = FALSE;

bool createIfNotFound = !IsJsNativeObject(object) ||

(DynamicType::Is(object->GetTypeId()) && static\_cast<DynamicObject\*>(object)->GetTypeHandler()->IsStringTypeHandler()) || JavascriptProxy::Is(object);

if (GetIndexType(index, scriptContext, &indexVal, &propertyRecord, createIfNotFound) == IndexType\_Number)

{

hasProperty = JavascriptOperators::GetItemReference(instance, object, indexVal, &value, scriptContext);

}

else

{

if (propertyRecord != nullptr)

{

hasProperty = JavascriptOperators::GetPropertyReference(instance, object, propertyRecord->GetPropertyId(), &value, scriptContext, NULL);

}

#if DBG

else

{

Assert(IsJsNativeObject(object));

JavascriptString\* indexStr = JavascriptConversion::ToString(index, scriptContext);

PropertyRecord const \* debugPropertyRecord;

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &debugPropertyRecord);

AssertMsg(!JavascriptOperators::GetPropertyReference(instance, object, debugPropertyRecord->GetPropertyId(), &value, scriptContext, NULL),

"how did this property come? See OS Bug 2727708 if you see this come from the web");

}

#endif

}

if (!hasProperty)

{

JavascriptString\* varName = JavascriptConversion::ToString(index, scriptContext);

// ES5 11.2.3 #2: We evaluate the call target but don't throw yet if target member is missing. We need to evaluate argList

// first (#3). Postpone throwing error to invoke time.

value = ThrowErrorObject::CreateThrowTypeErrorObject(scriptContext, VBSERR\_OLENoPropOrMethod, varName);

}

else if(!JavascriptConversion::IsCallable(value))

{

// ES5 11.2.3 #2: We evaluate the call target but don't throw yet if target member is missing. We need to evaluate argList

// first (#3). Postpone throwing error to invoke time.

JavascriptString\* varName = JavascriptConversion::ToString(index, scriptContext);

value = ThrowErrorObject::CreateThrowTypeErrorObject(scriptContext, JSERR\_Property\_NeedFunction, varName);

}

threadContext->CheckAndResetImplicitCallAccessorFlag();

threadContext->AddImplicitCallFlags(savedImplicitCallFlags);

return value;

}

BOOL JavascriptOperators::OP\_SetElementI\_UInt32(Var instance, uint32 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

#if FLOATVAR

return OP\_SetElementI\_JIT(instance, Js::JavascriptNumber::ToVar(index, scriptContext), value, scriptContext, flags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetElementI\_JIT(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), value, scriptContext, flags);

#endif

}

BOOL JavascriptOperators::OP\_SetElementI\_Int32(Var instance, int32 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

#if FLOATVAR

return OP\_SetElementI\_JIT(instance, Js::JavascriptNumber::ToVar(index, scriptContext), value, scriptContext, flags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetElementI\_JIT(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), value, scriptContext, flags);

#endif

}

BOOL JavascriptOperators::OP\_SetElementI\_JIT(Var instance, Var index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

if (TaggedNumber::Is(instance))

{

return OP\_SetElementI(instance, index, value, scriptContext, flags);

}

INT\_PTR vt = VirtualTableInfoBase::GetVirtualTable(instance);

OP\_SetElementI(instance, index, value, scriptContext, flags);

return vt != VirtualTableInfoBase::GetVirtualTable(instance);

}

BOOL JavascriptOperators::OP\_SetElementI(Var instance, Var index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

TypeId instanceType = JavascriptOperators::GetTypeId(instance);

bool isTypedArray = (instanceType >= TypeIds\_Int8Array && instanceType <= TypeIds\_Float64Array);

if (isTypedArray)

{

if (TaggedInt::Is(index) || JavascriptNumber::Is\_NoTaggedIntCheck(index) || JavascriptString::Is(index))

{

BOOL returnValue = FALSE;

bool isNumericIndex = false;

switch (instanceType)

{

case TypeIds\_Int8Array:

{

// The typed array will deal with all possible values for the index

if (VirtualTableInfo<Int8VirtualArray>::HasVirtualTable(instance))

{

Int8VirtualArray\* int8Array = Int8VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int8Array))

{

returnValue = int8Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Int8Array\* int8Array = Int8Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int8Array))

{

returnValue = int8Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Uint8Array:

{

// The typed array will deal with all possible values for the index

if (VirtualTableInfo<Uint8VirtualArray>::HasVirtualTable(instance))

{

Uint8VirtualArray\* uint8Array = Uint8VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8Array))

{

returnValue = uint8Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Uint8Array\* uint8Array = Uint8Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8Array))

{

returnValue = uint8Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Uint8ClampedArray:

{

// The typed array will deal with all possible values for the index

if (VirtualTableInfo<Uint8ClampedVirtualArray>::HasVirtualTable(instance))

{

Uint8ClampedVirtualArray\* uint8ClampedArray = Uint8ClampedVirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8ClampedArray))

{

returnValue = uint8ClampedArray->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Uint8ClampedArray\* uint8ClampedArray = Uint8ClampedArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint8ClampedArray))

{

returnValue = uint8ClampedArray->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Int16Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Int16VirtualArray>::HasVirtualTable(instance))

{

Int16VirtualArray\* int16Array = Int16VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int16Array))

{

returnValue = int16Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Int16Array\* int16Array = Int16Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int16Array))

{

returnValue = int16Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Uint16Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Uint16VirtualArray>::HasVirtualTable(instance))

{

Uint16VirtualArray\* uint16Array = Uint16VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint16Array))

{

returnValue = uint16Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Uint16Array\* uint16Array = Uint16Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint16Array))

{

returnValue = uint16Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Int32Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Int32VirtualArray>::HasVirtualTable(instance))

{

Int32VirtualArray\* int32Array = Int32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int32Array))

{

returnValue = int32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Int32Array\* int32Array = Int32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(int32Array))

{

returnValue = int32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Uint32Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Uint32VirtualArray>::HasVirtualTable(instance))

{

Uint32VirtualArray\* uint32Array = Uint32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint32Array))

{

returnValue = uint32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Uint32Array\* uint32Array = Uint32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(uint32Array))

{

returnValue = uint32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Float32Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Float32VirtualArray>::HasVirtualTable(instance))

{

Float32VirtualArray\* float32Array = Float32VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float32Array))

{

returnValue = float32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Float32Array\* float32Array = Float32Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float32Array))

{

returnValue = float32Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

case TypeIds\_Float64Array:

{

// The type array will deal with all possible values for the index

if (VirtualTableInfo<Float64VirtualArray>::HasVirtualTable(instance))

{

Float64VirtualArray\* float64Array = Float64VirtualArray::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float64Array))

{

returnValue = float64Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

else

{

Float64Array\* float64Array = Float64Array::FromVar(instance);

if (!CrossSite::IsCrossSiteObjectTyped(float64Array))

{

returnValue = float64Array->ValidateIndexAndDirectSetItem(index, value, &isNumericIndex);

}

}

break;

}

}

// if this was numeric index, return operation status else

// Return the result of calling the default ordinary object [[Set]] internal method (9.1.8) on O passing P, V, and Receiver as arguments.

if (isNumericIndex)

return returnValue;

}

}

else

{

if (TaggedInt::Is(index))

{

TaggedIntIndex:

switch (instanceType)

{

case TypeIds\_NativeIntArray:

case TypeIds\_NativeFloatArray:

case TypeIds\_Array: // fast path for array

{

int indexInt = TaggedInt::ToInt32(index);

if (indexInt >= 0 && scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

JavascriptArray::FromVar(instance)->SetItem((uint32)indexInt, value, flags);

return true;

}

break;

}

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(index))

{

double dIndexValue = JavascriptNumber::GetValue(index);

uint32 uint32Index = JavascriptConversion::ToUInt32(index, scriptContext);

if ((double)uint32Index == dIndexValue && !TaggedInt::IsOverflow(uint32Index))

{

index = TaggedInt::ToVarUnchecked(uint32Index);

goto TaggedIntIndex;

}

}

}

RecyclableObject\* object;

BOOL isNullOrUndefined = !GetPropertyObject(instance, scriptContext, &object);

Assert(object == instance || TaggedNumber::Is(instance));

if (isNullOrUndefined)

{

if (!scriptContext->GetThreadContext()->RecordImplicitException())

{

return FALSE;

}

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotSet\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

return JavascriptOperators::SetElementIHelper(instance, object, index, value, scriptContext, flags);

}

BOOL JavascriptOperators::SetElementIHelper(Var receiver, RecyclableObject\* object, Var index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

PropertyString \* propertyString = nullptr;

Js::IndexType indexType;

uint32 indexVal = 0;

PropertyRecord const \* propertyRecord = nullptr;

JavascriptString \* propertyNameString = nullptr;

if (TaggedNumber::Is(receiver))

{

indexType = GetIndexType(index, scriptContext, &indexVal, &propertyRecord, true);

if (indexType == IndexType\_Number)

{

return JavascriptOperators::SetItemOnTaggedNumber(receiver, object, indexVal, value, scriptContext, flags);

}

else

{

return JavascriptOperators::SetPropertyOnTaggedNumber(receiver, object, propertyRecord->GetPropertyId(), value, scriptContext, flags);

}

}

// fastpath for PropertyStrings only if receiver == object

if (!TaggedInt::Is(index) && JavascriptString::Is(index) &&

VirtualTableInfo<Js::PropertyString>::HasVirtualTable(JavascriptString::FromVar(index)))

{

propertyString = (PropertyString \*)JavascriptString::FromVar(index);

Assert(propertyString->GetScriptContext() == scriptContext);

PropertyCache const \* cache = propertyString->GetPropertyCache();

if (receiver == object && object->GetType() == cache->type && cache->isStoreFieldEnabled)

{

#if DBG

propertyRecord = propertyString->GetPropertyRecord();

#endif

#if DBG\_DUMP

scriptContext->forinCache++;

#endif

Assert(object->GetScriptContext() == scriptContext);

Assert(!CrossSite::NeedMarshalVar(value, scriptContext));

if (cache->isInlineSlot)

{

DynamicObject::FromVar(object)->SetInlineSlot(SetSlotArguments(propertyRecord->GetPropertyId(), cache->dataSlotIndex, value));

}

else

{

DynamicObject::FromVar(object)->SetAuxSlot(SetSlotArguments(propertyRecord->GetPropertyId(), cache->dataSlotIndex, value));

}

return true;

}

propertyRecord = propertyString->GetPropertyRecord();

if (propertyRecord->IsNumeric())

{

indexType = IndexType\_Number;

indexVal = propertyRecord->GetNumericValue();

}

else

{

indexType = IndexType\_PropertyId;

}

#if DBG\_DUMP

scriptContext->forinNoCache++;

#endif

}

else

{

#if DBG\_DUMP

scriptContext->forinNoCache += (!TaggedInt::Is(index) && JavascriptString::Is(index));

#endif

indexType = GetIndexType(index, scriptContext, &indexVal, &propertyRecord, &propertyNameString, false, true);

if (scriptContext->GetThreadContext()->IsDisableImplicitCall() &&

scriptContext->GetThreadContext()->GetImplicitCallFlags() != ImplicitCall\_None)

{

// We hit an implicit call trying to convert the index, and implicit calls are disabled, so

// quit before we try to store the element.

return FALSE;

}

}

if (indexType == IndexType\_Number)

{

return JavascriptOperators::SetItem(receiver, object, indexVal, value, scriptContext, flags);

}

else if (indexType == IndexType\_JavascriptString)

{

Assert(propertyNameString);

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

if (BuiltInPropertyRecords::NaN.Equals(propertyName))

{

// Follow SetProperty convention for NaN

return JavascriptOperators::SetProperty(receiver, object, PropertyIds::NaN, value, scriptContext, flags);

}

else if (BuiltInPropertyRecords::Infinity.Equals(propertyName))

{

// Follow SetProperty convention for Infinity

return JavascriptOperators::SetProperty(receiver, object, PropertyIds::Infinity, value, scriptContext, flags);

}

return JavascriptOperators::SetPropertyWPCache(receiver, object, propertyNameString, value, scriptContext, nullptr, flags);

}

else if (indexType == IndexType\_PropertyId)

{

Assert(propertyRecord);

PropertyId propId = propertyRecord->GetPropertyId();

if (propId == PropertyIds::NaN || propId == PropertyIds::Infinity)

{

// As we no longer convert o[x] into o.x for NaN and Infinity, we need to follow SetProperty convention for these,

// which would check for read-only properties, strict mode, etc.

// Note that "-Infinity" does not qualify as property name, so we don't have to take care of it.

return JavascriptOperators::SetProperty(receiver, object, propId, value, scriptContext, flags);

}

}

return JavascriptOperators::SetPropertyWPCache(receiver, object, propertyRecord->GetPropertyId(), value, scriptContext, propertyString, flags);

}

BOOL JavascriptOperators::OP\_SetNativeIntElementI(

Var instance,

Var aElementIndex,

int32 iValue,

ScriptContext\* scriptContext,

PropertyOperationFlags flags)

{

if (TaggedInt::Is(aElementIndex))

{

int32 indexInt = TaggedInt::ToInt32(aElementIndex);

if (indexInt >= 0 && scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

JavascriptNativeIntArray \*arr = JavascriptNativeIntArray::FromVar(instance);

if (!(arr->TryGrowHeadSegmentAndSetItem<int32, JavascriptNativeIntArray>((uint32)indexInt, iValue)))

{

arr->SetItem(indexInt, iValue);

}

return TRUE;

}

}

return JavascriptOperators::OP\_SetElementI(instance, aElementIndex, JavascriptNumber::ToVar(iValue, scriptContext), scriptContext, flags);

}

BOOL JavascriptOperators::OP\_SetNativeIntElementI\_UInt32(

Var instance,

uint32 aElementIndex,

int32 iValue,

ScriptContext\* scriptContext,

PropertyOperationFlags flags)

{

#if FLOATVAR

return OP\_SetNativeIntElementI(instance, Js::JavascriptNumber::ToVar(aElementIndex, scriptContext), iValue, scriptContext, flags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetNativeIntElementI(instance, Js::JavascriptNumber::ToVarInPlace(aElementIndex, scriptContext,

(Js::JavascriptNumber \*)buffer), iValue, scriptContext, flags);

#endif

}

BOOL JavascriptOperators::OP\_SetNativeIntElementI\_Int32(

Var instance,

int aElementIndex,

int32 iValue,

ScriptContext\* scriptContext,

PropertyOperationFlags flags)

{

#if FLOATVAR

return OP\_SetNativeIntElementI(instance, Js::JavascriptNumber::ToVar(aElementIndex, scriptContext), iValue, scriptContext, flags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetNativeIntElementI(instance, Js::JavascriptNumber::ToVarInPlace(aElementIndex, scriptContext,

(Js::JavascriptNumber \*)buffer), iValue, scriptContext, flags);

#endif

}

BOOL JavascriptOperators::OP\_SetNativeFloatElementI(

Var instance,

Var aElementIndex,

ScriptContext\* scriptContext,

PropertyOperationFlags flags,

double dValue)

{

if (TaggedInt::Is(aElementIndex))

{

int32 indexInt = TaggedInt::ToInt32(aElementIndex);

if (indexInt >= 0 && scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

JavascriptNativeFloatArray \*arr = JavascriptNativeFloatArray::FromVar(instance);

if (!(arr->TryGrowHeadSegmentAndSetItem<double, JavascriptNativeFloatArray>((uint32)indexInt, dValue)))

{

arr->SetItem(indexInt, dValue);

}

return TRUE;

}

}

return JavascriptOperators::OP\_SetElementI(instance, aElementIndex, JavascriptNumber::ToVarWithCheck(dValue, scriptContext), scriptContext, flags);

}

BOOL JavascriptOperators::OP\_SetNativeFloatElementI\_UInt32(

Var instance, uint32

aElementIndex,

ScriptContext\* scriptContext,

PropertyOperationFlags flags,

double dValue)

{

#if FLOATVAR

return OP\_SetNativeFloatElementI(instance, JavascriptNumber::ToVar(aElementIndex, scriptContext), scriptContext, flags, dValue);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetNativeFloatElementI(instance, JavascriptNumber::ToVarInPlace(aElementIndex, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext, flags, dValue);

#endif

}

BOOL JavascriptOperators::OP\_SetNativeFloatElementI\_Int32(

Var instance,

int aElementIndex,

ScriptContext\* scriptContext,

PropertyOperationFlags flags,

double dValue)

{

#if FLOATVAR

return OP\_SetNativeFloatElementI(instance, JavascriptNumber::ToVar(aElementIndex, scriptContext), scriptContext, flags, dValue);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_SetNativeFloatElementI(instance, JavascriptNumber::ToVarInPlace(aElementIndex, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext, flags, dValue);

#endif

}

BOOL JavascriptOperators::OP\_Memcopy(Var dstInstance, int32 dstStart, Var srcInstance, int32 srcStart, int32 length, ScriptContext\* scriptContext)

{

if (length <= 0)

{

return true;

}

TypeId instanceType = JavascriptOperators::GetTypeId(srcInstance);

if (instanceType != JavascriptOperators::GetTypeId(dstInstance))

{

return false;

}

if (srcStart != dstStart)

{

return false;

}

BOOL returnValue = false;

switch (instanceType)

{

case TypeIds\_Int8Array:

{

// The typed array will deal with all possible values for the index

returnValue = Int8Array::FromVar(dstInstance)->DirectSetItemAtRange(Int8Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToInt8);

break;

}

case TypeIds\_Uint8Array:

{

returnValue = Uint8Array::FromVar(dstInstance)->DirectSetItemAtRange(Uint8Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToUInt8);

break;

}

case TypeIds\_Uint8ClampedArray:

{

returnValue = Uint8ClampedArray::FromVar(dstInstance)->DirectSetItemAtRange(Uint8ClampedArray::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToUInt8Clamped);

break;

}

case TypeIds\_Int16Array:

{

returnValue = Int16Array::FromVar(dstInstance)->DirectSetItemAtRange(Int16Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToInt16);

break;

}

case TypeIds\_Uint16Array:

{

returnValue = Uint16Array::FromVar(dstInstance)->DirectSetItemAtRange(Uint16Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToUInt16);

break;

}

case TypeIds\_Int32Array:

{

returnValue = Int32Array::FromVar(dstInstance)->DirectSetItemAtRange(Int32Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToInt32);

break;

}

case TypeIds\_Uint32Array:

{

returnValue = Uint32Array::FromVar(dstInstance)->DirectSetItemAtRange(Uint32Array::FromVar(srcInstance), srcStart, dstStart, length, JavascriptConversion::ToUInt32);

break;

}

case TypeIds\_Array:

case TypeIds\_NativeIntArray:

{

if (dstStart < 0 || srcStart < 0)

{

// This is not supported, Bailout

break;

}

// Upper bounds check for source array

uint32 end;

if (UInt32Math::Add(srcStart, length, &end) || end > ((ArrayObject\*)srcInstance)->GetLength())

{

return false;

}

if (scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

INT\_PTR vt = VirtualTableInfoBase::GetVirtualTable(dstInstance);

if (instanceType == TypeIds\_Array)

{

returnValue = JavascriptArray::FromVar(dstInstance)->DirectSetItemAtRangeFromArray<Var>(dstStart, length, JavascriptArray::FromVar(srcInstance), srcStart);

}

else

{

returnValue = JavascriptArray::FromVar(dstInstance)->DirectSetItemAtRangeFromArray<int32>(dstStart, length, JavascriptArray::FromVar(srcInstance), srcStart);

}

returnValue &= vt == VirtualTableInfoBase::GetVirtualTable(dstInstance);

}

break;

}

default:

{

AssertMsg(false, "We don't support this type for memcopy yet.");

break;

}

}

return returnValue;

}

BOOL JavascriptOperators::OP\_Memset(Var instance, int32 start, Var value, int32 length, ScriptContext\* scriptContext)

{

if (length <= 0)

{

return true;

}

TypeId instanceType = JavascriptOperators::GetTypeId(instance);

BOOL returnValue = false;

// The typed array will deal with all possible values for the index

#define MEMSET\_TYPED\_ARRAY(type, conversion) type ## ::FromVar(instance)->DirectSetItemAtRange(start, length, value, JavascriptConversion:: ## conversion)

switch (instanceType)

{

case TypeIds\_Int8Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Int8Array, ToInt8);

break;

}

case TypeIds\_Uint8Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Uint8Array, ToUInt8);

break;

}

case TypeIds\_Uint8ClampedArray:

{

returnValue = MEMSET\_TYPED\_ARRAY(Uint8ClampedArray, ToUInt8Clamped);

break;

}

case TypeIds\_Int16Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Int16Array, ToInt16);

break;

}

case TypeIds\_Uint16Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Uint16Array, ToUInt16);

break;

}

case TypeIds\_Int32Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Int32Array, ToInt32);

break;

}

case TypeIds\_Uint32Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Uint32Array, ToUInt32);

break;

}

case TypeIds\_Float32Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Float32Array, ToFloat);

break;

}

case TypeIds\_Float64Array:

{

returnValue = MEMSET\_TYPED\_ARRAY(Float64Array, ToNumber);

break;

}

case TypeIds\_NativeFloatArray:

case TypeIds\_NativeIntArray:

case TypeIds\_Array:

{

if (start < 0)

{

for (start; start < 0 && length > 0; ++start, --length)

{

if (!OP\_SetElementI(instance, JavascriptNumber::ToVar(start, scriptContext), value, scriptContext))

{

return false;

}

}

}

if (scriptContext->optimizationOverrides.IsEnabledArraySetElementFastPath())

{

INT\_PTR vt = VirtualTableInfoBase::GetVirtualTable(instance);

if (instanceType == TypeIds\_Array)

{

returnValue = JavascriptArray::FromVar(instance)->DirectSetItemAtRange<Var>(start, length, value);

}

else if (instanceType == TypeIds\_NativeIntArray)

{

returnValue = JavascriptArray::FromVar(instance)->DirectSetItemAtRange<int32>(start, length, JavascriptConversion::ToInt32(value, scriptContext));

}

else

{

returnValue = JavascriptArray::FromVar(instance)->DirectSetItemAtRange<double>(start, length, JavascriptConversion::ToNumber(value, scriptContext));

}

returnValue &= vt == VirtualTableInfoBase::GetVirtualTable(instance);

}

break;

}

default:

{

AssertMsg(false, "We don't support this type for memset yet.");

break;

}

}

#undef MEMSET\_TYPED\_ARRAY

return returnValue;

}

Var JavascriptOperators::OP\_DeleteElementI\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

#if FLOATVAR

return OP\_DeleteElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext, propertyOperationFlags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_DeleteElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext, propertyOperationFlags);

#endif

}

Var JavascriptOperators::OP\_DeleteElementI\_Int32(Var instance, int32 index, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

#if FLOATVAR

return OP\_DeleteElementI(instance, Js::JavascriptNumber::ToVar(index, scriptContext), scriptContext, propertyOperationFlags);

#else

char buffer[sizeof(Js::JavascriptNumber)];

return OP\_DeleteElementI(instance, Js::JavascriptNumber::ToVarInPlace(index, scriptContext,

(Js::JavascriptNumber \*)buffer), scriptContext, propertyOperationFlags);

#endif

}

Var JavascriptOperators::OP\_DeleteElementI(Var instance, Var index, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags)

{

if(TaggedNumber::Is(instance))

{

return scriptContext->GetLibrary()->GetTrue();

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

TypeId typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == TypeIds\_Null || typeId == TypeIds\_Undefined)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotDelete\_NullOrUndefined, GetPropertyDisplayNameForError(index, scriptContext));

}

RecyclableObject\* object = RecyclableObject::FromVar(instance);

uint32 indexVal;

PropertyRecord const \* propertyRecord;

BOOL result = TRUE;

bool createIfNotFound = !IsJsNativeObject(object) ||

(DynamicType::Is(object->GetTypeId()) && static\_cast<DynamicObject\*>(object)->GetTypeHandler()->IsStringTypeHandler()) || JavascriptProxy::Is(object);

if (GetIndexType(index, scriptContext, &indexVal, &propertyRecord, createIfNotFound) == IndexType\_Number)

{

result = JavascriptOperators::DeleteItem(object, indexVal, propertyOperationFlags);

}

else

{

if (propertyRecord)

{

result = JavascriptOperators::DeleteProperty(object, propertyRecord->GetPropertyId(), propertyOperationFlags);

}

#if DBG

else

{

Assert(IsJsNativeObject(object));

JavascriptString\* indexStr = JavascriptConversion::ToString(index, scriptContext);

PropertyRecord const \* debugPropertyRecord;

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &debugPropertyRecord);

AssertMsg(JavascriptOperators::DeleteProperty(object, debugPropertyRecord->GetPropertyId(), propertyOperationFlags), "delete should have been true. See OS Bug 2727708 if you see this come from the web");

}

#endif

}

return scriptContext->GetLibrary()->CreateBoolean(result);

}

Var JavascriptOperators::OP\_GetLength(Var instance, ScriptContext\* scriptContext)

{

return JavascriptOperators::OP\_GetProperty(instance, PropertyIds::length, scriptContext);

}

\_\_inline Var JavascriptOperators::GetThisFromModuleRoot(Var thisVar)

{

RootObjectBase \* rootObject = static\_cast<RootObjectBase\*>(thisVar);

RecyclableObject\* hostObject = rootObject->GetHostObject();

//

// if the module root has the host object, use that as "this"

//

if (hostObject)

{

thisVar = hostObject->GetHostDispatchVar();

}

return thisVar;

}

\_\_inline void JavascriptOperators::TryLoadRoot(Var& thisVar, TypeId typeId, int moduleID, ScriptContext\* scriptContext)

{

bool loadRoot = false;

if (JavascriptOperators::IsUndefinedOrNullType(typeId) || typeId == TypeIds\_ActivationObject)

{

loadRoot = true;

}

else if (typeId == TypeIds\_HostDispatch)

{

TypeId remoteTypeId;

if (RecyclableObject::FromVar(thisVar)->GetRemoteTypeId(&remoteTypeId))

{

if (remoteTypeId == TypeIds\_Null || remoteTypeId == TypeIds\_Undefined || remoteTypeId == TypeIds\_ActivationObject)

{

loadRoot = true;

}

}

}

if (loadRoot)

{

if (moduleID == 0)

{

thisVar = JavascriptOperators::OP\_LdRoot(scriptContext)->ToThis();

}

else

{

Js::ModuleRoot \* moduleRoot = JavascriptOperators::GetModuleRoot(moduleID, scriptContext);

if (moduleRoot == nullptr)

{

Assert(false);

thisVar = scriptContext->GetLibrary()->GetUndefined();

}

else

{

thisVar = GetThisFromModuleRoot(moduleRoot);

}

}

}

}

Var JavascriptOperators::OP\_GetThis(Var thisVar, int moduleID, ScriptContext\* scriptContext)

{

//

// if "this" is null or undefined

// Pass the global object

// Else

// Pass ToObject(this)

//

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

Assert(!JavascriptOperators::IsThisSelf(typeId));

return JavascriptOperators::GetThisHelper(thisVar, typeId, moduleID, scriptContext);

}

Var JavascriptOperators::OP\_GetThisNoFastPath(Var thisVar, int moduleID, ScriptContext\* scriptContext)

{

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

if (JavascriptOperators::IsThisSelf(typeId))

{

Assert(typeId != TypeIds\_GlobalObject || ((Js::GlobalObject\*)thisVar)->ToThis() == thisVar);

Assert(typeId != TypeIds\_ModuleRoot || JavascriptOperators::GetThisFromModuleRoot(thisVar) == thisVar);

return thisVar;

}

return JavascriptOperators::GetThisHelper(thisVar, typeId, moduleID, scriptContext);

}

bool JavascriptOperators::IsThisSelf(TypeId typeId)

{

return (JavascriptOperators::IsObjectType(typeId) && ! JavascriptOperators::IsSpecialObjectType(typeId));

}

Var JavascriptOperators::GetThisHelper(Var thisVar, TypeId typeId, int moduleID, ScriptContext \*scriptContext)

{

if (! JavascriptOperators::IsObjectType(typeId) && ! JavascriptOperators::IsUndefinedOrNullType(typeId))

{

#if !FLOATVAR

// We allowed stack number to be used as the "this" for getter and setter activation of

// n.x and n[prop], where n is the Javascript Number

return JavascriptOperators::ToObject(

JavascriptNumber::BoxStackNumber(thisVar, scriptContext), scriptContext);

#else

return JavascriptOperators::ToObject(thisVar, scriptContext);

#endif

}

else

{

TryLoadRoot(thisVar, typeId, moduleID, scriptContext);

return thisVar;

}

}

Var JavascriptOperators::OP\_StrictGetThis(Var thisVar, ScriptContext\* scriptContext)

{

TypeId typeId = JavascriptOperators::GetTypeId(thisVar);

if (typeId == TypeIds\_ActivationObject)

{

return scriptContext->GetLibrary()->GetUndefined();

}

return thisVar;

}

BOOL JavascriptOperators::GetRemoteTypeId(Var aValue, TypeId\* typeId)

{

if (GetTypeId(aValue) != TypeIds\_HostDispatch)

{

return FALSE;

}

return RecyclableObject::FromVar(aValue)->GetRemoteTypeId(typeId);

}

BOOL JavascriptOperators::IsJsNativeObject(Var aValue)

{

switch(GetTypeId(aValue))

{

case TypeIds\_Object:

case TypeIds\_Function:

case TypeIds\_Array:

case TypeIds\_NativeIntArray:

#if ENABLE\_COPYONACCESS\_ARRAY

case TypeIds\_CopyOnAccessNativeIntArray:

#endif

case TypeIds\_NativeFloatArray:

case TypeIds\_ES5Array:

case TypeIds\_Date:

case TypeIds\_WinRTDate:

case TypeIds\_RegEx:

case TypeIds\_Error:

case TypeIds\_BooleanObject:

case TypeIds\_NumberObject:

case TypeIds\_StringObject:

case TypeIds\_Symbol:

case TypeIds\_SymbolObject:

//case TypeIds\_GlobalObject:

//case TypeIds\_ModuleRoot:

//case TypeIds\_HostObject:

case TypeIds\_Arguments:

case TypeIds\_ActivationObject:

case TypeIds\_Map:

case TypeIds\_Set:

case TypeIds\_WeakMap:

case TypeIds\_WeakSet:

case TypeIds\_ArrayIterator:

case TypeIds\_MapIterator:

case TypeIds\_SetIterator:

case TypeIds\_StringIterator:

case TypeIds\_Generator:

case TypeIds\_Promise:

case TypeIds\_Proxy:

return true;

default:

return false;

}

}

RecyclableObject\* JavascriptOperators::GetPrototype(RecyclableObject\* instance)

{

if (JavascriptOperators::GetTypeId(instance) == TypeIds\_Null)

{

return instance;

}

return instance->GetPrototype();

}

RecyclableObject\* JavascriptOperators::OP\_GetPrototype(Var instance, ScriptContext\* scriptContext)

{

if (TaggedNumber::Is(instance))

{

return scriptContext->GetLibrary()->GetNumberPrototype();

}

else if (JavascriptOperators::GetTypeId(instance) != TypeIds\_Null)

{

return JavascriptOperators::GetPrototype(RecyclableObject::FromVar(instance));

}

else

{

return scriptContext->GetLibrary()->GetNull();

}

}

BOOL JavascriptOperators::OP\_BrFncEqApply(Var instance, ScriptContext \*scriptContext)

{

// JavascriptFunction && !HostDispatch

if (JavascriptOperators::GetTypeId(instance) == TypeIds\_Function)

{

FunctionProxy \*bod= ((JavascriptFunction\*)instance)->GetFunctionProxy();

if (bod != nullptr)

{

return bod->GetDirectEntryPoint(bod->GetDefaultEntryPointInfo()) == &Js::JavascriptFunction::EntryApply;

}

else

{

FunctionInfo\* info = ((JavascriptFunction \*)instance)->GetFunctionInfo();

if (info != nullptr)

{

return &Js::JavascriptFunction::EntryApply == info->GetOriginalEntryPoint();

}

else

{

return false;

}

}

}

return false;

}

BOOL JavascriptOperators::OP\_BrFncNeqApply(Var instance, ScriptContext \*scriptContext)

{

// JavascriptFunction and !HostDispatch

if (JavascriptOperators::GetTypeId(instance) == TypeIds\_Function)

{

FunctionProxy \*bod = ((JavascriptFunction \*)instance)->GetFunctionProxy();

if (bod != nullptr)

{

return bod->GetDirectEntryPoint(bod->GetDefaultEntryPointInfo()) != &Js::JavascriptFunction::EntryApply;

}

else

{

FunctionInfo\* info = ((JavascriptFunction \*)instance)->GetFunctionInfo();

if (info != nullptr)

{

return &Js::JavascriptFunction::EntryApply != info->GetOriginalEntryPoint();

}

else

{

return true;

}

}

}

return true;

}

BOOL JavascriptOperators::OP\_BrHasSideEffects(int se, ScriptContext\* scriptContext)

{

return (scriptContext->optimizationOverrides.GetSideEffects() & se) != SideEffects\_None;

}

BOOL JavascriptOperators::OP\_BrNotHasSideEffects(int se, ScriptContext\* scriptContext)

{

return (scriptContext->optimizationOverrides.GetSideEffects() & se) == SideEffects\_None;

}

// returns NULL if there is no more elements to enumerate.

Var JavascriptOperators::OP\_BrOnEmpty(ForInObjectEnumerator \* aEnumerator)

{

PropertyId id;

return aEnumerator->GetCurrentAndMoveNext(id);

}

ForInObjectEnumerator \* JavascriptOperators::OP\_GetForInEnumerator(Var enumerable, ScriptContext\* scriptContext)

{

RecyclableObject\* enumerableObject;

bool isCrossSite;

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(enumerable);

#endif

if (GetPropertyObject(enumerable, scriptContext, &enumerableObject))

{

isCrossSite = enumerableObject->GetScriptContext() != scriptContext;

}

else

{

enumerableObject = nullptr;

isCrossSite = false;

}

if (!isCrossSite)

{

ForInObjectEnumerator \* enumerator = scriptContext->GetLibrary()->GetAndClearForInEnumeratorCache();

if(enumerator != NULL)

{

enumerator->Initialize(enumerableObject, scriptContext);

return enumerator;

}

}

return RecyclerNew(scriptContext->GetRecycler(), ForInObjectEnumerator, enumerableObject, scriptContext);

}

void JavascriptOperators::OP\_ReleaseForInEnumerator(ForInObjectEnumerator \* enumerator, ScriptContext\* scriptContext)

{

// Debugger SetNextStatement may skip OP\_GetForInEnumerator and result in NULL ForInObjectEnumerator here. See Win8 391556

if (enumerator && enumerator->CanBeReused())

{

enumerator->Clear();

scriptContext->GetLibrary()->SetForInEnumeratorCache(enumerator);

}

}

Js::Var JavascriptOperators::OP\_CmEq\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::Equal(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmNeq\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::NotEqual(a,b,scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmSrEq\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::StrictEqual(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmSrEq\_String(Var a, Var b, ScriptContext \*scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::StrictEqualString(a, b), scriptContext);

}

Var JavascriptOperators::OP\_CmSrEq\_EmptyString(Var a, ScriptContext \*scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::StrictEqualEmptyString(a), scriptContext);

}

Var JavascriptOperators::OP\_CmSrNeq\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::NotStrictEqual(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmLt\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::Less(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmLe\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::LessEqual(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmGt\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::Greater(a, b, scriptContext), scriptContext);

}

Var JavascriptOperators::OP\_CmGe\_A(Var a, Var b, ScriptContext\* scriptContext)

{

return JavascriptBoolean::ToVar(JavascriptOperators::GreaterEqual(a, b, scriptContext), scriptContext);

}

DetachedStateBase\* JavascriptOperators::DetachVarAndGetState(Var var)

{

switch (GetTypeId(var))

{

case TypeIds\_ArrayBuffer:

return Js::ArrayBuffer::FromVar(var)->DetachAndGetState();

default:

if (!Js::RecyclableObject::FromVar(var)->IsExternal())

{

AssertMsg(false, "We should explicitly have a case statement for each non-external object that can be detached.");

}

return nullptr;

}

}

bool JavascriptOperators::IsObjectDetached(Var var)

{

switch (GetTypeId(var))

{

case TypeIds\_ArrayBuffer:

return Js::ArrayBuffer::FromVar(var)->IsDetached();

default:

return false;

}

}

Var JavascriptOperators::NewVarFromDetachedState(DetachedStateBase\* state, JavascriptLibrary \*library)

{

switch (state->GetTypeId())

{

case TypeIds\_ArrayBuffer:

return Js::ArrayBuffer::NewFromDetachedState(state, library);

break;

default:

AssertMsg(false, "We should explicitly have a case statement for each object which has detached state.");

return nullptr;

}

}

DynamicType \*

JavascriptOperators::EnsureObjectLiteralType(ScriptContext\* scriptContext, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType)

{

DynamicType \* newType = \*literalType;

if (newType != nullptr)

{

if (!newType->GetIsShared())

{

newType->ShareType();

}

}

else

{

DynamicType\* objectType =

FunctionBody::DoObjectHeaderInliningForObjectLiteral(propIds, scriptContext)

? scriptContext->GetLibrary()->GetObjectHeaderInlinedLiteralType((uint16)propIds->count)

: scriptContext->GetLibrary()->GetObjectLiteralType(

static\_cast<PropertyIndex>(

min(propIds->count, static\_cast<uint32>(MaxPreInitializedObjectTypeInlineSlotCount))));

newType = PathTypeHandlerBase::CreateTypeForNewScObject(scriptContext, objectType, propIds, false);

\*literalType = newType;

}

Assert(GetLiteralInlineSlotCapacity(propIds, scriptContext) == newType->GetTypeHandler()->GetInlineSlotCapacity());

Assert(newType->GetTypeHandler()->GetSlotCapacity() >= 0);

Assert(GetLiteralSlotCapacity(propIds, scriptContext) == (uint)newType->GetTypeHandler()->GetSlotCapacity());

return newType;

}

Var JavascriptOperators::NewScObjectLiteral(ScriptContext\* scriptContext, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType)

{

Assert(propIds->count != 0);

Assert(!propIds->hadDuplicates); // duplicates are removed by parser

#ifdef PROFILE\_OBJECT\_LITERALS

// Empty objects not counted in the object literal counts

scriptContext->objectLiteralInstanceCount++;

if (propIds->count > scriptContext->objectLiteralMaxLength)

{

scriptContext->objectLiteralMaxLength = propIds->count;

}

#endif

DynamicType\* newType = EnsureObjectLiteralType(scriptContext, propIds, literalType);

DynamicObject\* instance = DynamicObject::New(scriptContext->GetRecycler(), newType);

if (!newType->GetIsShared())

{

newType->GetTypeHandler()->SetSingletonInstanceIfNeeded(instance);

}

#ifdef PROFILE\_OBJECT\_LITERALS

else

{

scriptContext->objectLiteralCacheCount++;

}

#endif

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(instance));

// can't auto-proxy here as object literal is not exactly "new" object and cannot be intercepted as proxy.

return instance;

}

uint JavascriptOperators::GetLiteralSlotCapacity(Js::PropertyIdArray const \* propIds, ScriptContext \*const scriptContext)

{

const uint inlineSlotCapacity = GetLiteralInlineSlotCapacity(propIds, scriptContext);

return DynamicTypeHandler::RoundUpSlotCapacity(propIds->count, static\_cast<PropertyIndex>(inlineSlotCapacity));

}

uint JavascriptOperators::GetLiteralInlineSlotCapacity(

Js::PropertyIdArray const \* propIds,

ScriptContext \*const scriptContext)

{

if (propIds->hadDuplicates)

{

return 0;

}

return

FunctionBody::DoObjectHeaderInliningForObjectLiteral(propIds, scriptContext)

? DynamicTypeHandler::RoundUpObjectHeaderInlinedInlineSlotCapacity(static\_cast<PropertyIndex>(propIds->count))

: DynamicTypeHandler::RoundUpInlineSlotCapacity(

static\_cast<PropertyIndex>(

min(propIds->count, static\_cast<uint32>(MaxPreInitializedObjectTypeInlineSlotCount))));

}

Var JavascriptOperators::OP\_InitCachedScope(Var varFunc, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType, bool formalsAreLetDecls, ScriptContext \*scriptContext)

{

ScriptFunction \*func = JavascriptGeneratorFunction::Is(varFunc) ?

JavascriptGeneratorFunction::FromVar(varFunc)->GetGeneratorVirtualScriptFunction() :

ScriptFunction::FromVar(varFunc);

#ifdef PROFILE\_OBJECT\_LITERALS

// Empty objects not counted in the object literal counts

scriptContext->objectLiteralInstanceCount++;

if (propIds->count > scriptContext->objectLiteralMaxLength)

{

scriptContext->objectLiteralMaxLength = propIds->count;

}

#endif

PropertyId cachedFuncCount = ActivationObjectEx::GetCachedFuncCount(propIds);

PropertyId firstFuncSlot = ActivationObjectEx::GetFirstFuncSlot(propIds);

PropertyId firstVarSlot = ActivationObjectEx::GetFirstVarSlot(propIds);

PropertyId lastFuncSlot = Constants::NoProperty;

if (firstFuncSlot != Constants::NoProperty)

{

if (firstVarSlot == Constants::NoProperty)

{

lastFuncSlot = propIds->count - 1;

}

else

{

lastFuncSlot = firstVarSlot - 1;

}

}

DynamicType \*type = \*literalType;

if (type != nullptr)

{

#ifdef PROFILE\_OBJECT\_LITERALS

scriptContext->objectLiteralCacheCount++;

#endif

}

else

{

type = scriptContext->GetLibrary()->GetActivationObjectType();

if (formalsAreLetDecls)

{

uint formalsSlotLimit = (firstFuncSlot != Constants::NoProperty) ? (uint)firstFuncSlot :

(firstVarSlot != Constants::NoProperty) ? (uint)firstVarSlot :

propIds->count;

type = PathTypeHandlerBase::CreateNewScopeObject(scriptContext, type, propIds, PropertyLet, formalsSlotLimit);

}

else

{

type = PathTypeHandlerBase::CreateNewScopeObject(scriptContext, type, propIds);

}

\*literalType = type;

}

Var undef = scriptContext->GetLibrary()->GetUndefined();

ActivationObjectEx \*scopeObjEx = func->GetCachedScope();

if (scopeObjEx && scopeObjEx->IsCommitted())

{

scopeObjEx->ReplaceType(type);

scopeObjEx->SetCommit(false);

#if DBG

for (uint i = firstVarSlot; i < propIds->count; i++)

{

AssertMsg(scopeObjEx->GetSlot(i) == undef, "Var attached to cached scope");

}

#endif

}

else

{

ActivationObjectEx \*tmp = RecyclerNewPlus(scriptContext->GetRecycler(), (cachedFuncCount == 0 ? 0 : cachedFuncCount - 1) \* sizeof(FuncCacheEntry), ActivationObjectEx, type, func, cachedFuncCount, firstFuncSlot, lastFuncSlot);

if (!scopeObjEx)

{

func->SetCachedScope(tmp);

}

scopeObjEx = tmp;

for (uint i = firstVarSlot; i < propIds->count; i++)

{

scopeObjEx->SetSlot(SetSlotArguments(propIds->elements[i], i, undef));

}

}

return scopeObjEx;

}

void JavascriptOperators::OP\_InvalidateCachedScope(void\* varEnv, int32 envIndex)

{

FrameDisplay \*disp = (FrameDisplay\*)varEnv;

RecyclableObject \*objScope = RecyclableObject::FromVar(disp->GetItem(envIndex));

objScope->InvalidateCachedScope();

}

void JavascriptOperators::OP\_InitCachedFuncs(Var varScope, FrameDisplay \*pDisplay, const FuncInfoArray \*info, ScriptContext \*scriptContext)

{

ActivationObjectEx \*scopeObj = (ActivationObjectEx\*)ActivationObjectEx::FromVar(varScope);

Assert(scopeObj->GetTypeHandler()->GetInlineSlotCapacity() == 0);

ScriptFunction \*func;

FuncCacheEntry \*entry;

FunctionProxy \*proxy;

uint scopeSlot;

uint funcCount = info->count;

if (funcCount == 0)

{

// Degenerate case: no nested funcs at all

return;

}

if (scopeObj->HasCachedFuncs())

{

for (uint i = 0; i < funcCount; i++)

{

entry = scopeObj->GetFuncCacheEntry(i);

func = entry->func;

proxy = func->GetFunctionProxy();

if (proxy != proxy->GetFunctionProxy())

{

// The FunctionProxy has changed since the object was cached, e.g., due to execution

// of a deferred function through a different object.

proxy = proxy->GetFunctionProxy();

func->SetFunctionInfo(proxy);

}

// Reset the function's type to the default type with no properties

// Use the cached type on the function proxy rather than the type in the func cache entry

// CONSIDER: Stop caching the function types in the scope object

func->ReplaceType(proxy->EnsureDeferredPrototypeType());

func->ResetConstructorCacheToDefault();

scopeSlot = info->elements[i].scopeSlot;

if (scopeSlot != Constants::NoProperty)

{

// CONSIDER: Store property IDs in FuncInfoArray in debug builds so we can properly assert in SetAuxSlot

scopeObj->SetAuxSlot(SetSlotArguments(Constants::NoProperty, scopeSlot, entry->func));

}

}

return;

}

// No cached functions, so create them and cache them.

JavascriptFunction \*funcParent = scopeObj->GetParentFunc();

for (uint i = 0; i < funcCount; i++)

{

const FuncInfoEntry \*entry = &info->elements[i];

uint nestedIndex = entry->nestedIndex;

scopeSlot = entry->scopeSlot;

proxy = funcParent->GetFunctionBody()->GetNestedFunc(nestedIndex);

func = scriptContext->GetLibrary()->CreateScriptFunction(proxy);

func->SetEnvironment(pDisplay);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_FUNCTION(func, EtwTrace::GetFunctionId(proxy)));

scopeObj->SetCachedFunc(i, func);

if (scopeSlot != Constants::NoProperty)

{

// CONSIDER: Store property IDs in FuncInfoArray in debug builds so we can properly assert in SetAuxSlot

scopeObj->SetAuxSlot(SetSlotArguments(Constants::NoProperty, scopeSlot, func));

}

}

}

Var JavascriptOperators::AddVarsToArraySegment(SparseArraySegment<Var> \* segment, const Js::VarArray \*vars)

{

uint32 count = vars->count;

Assert(segment->left == 0);

Assert(count <= segment->size);

if(count > segment->length)

{

segment->length = count;

}

js\_memcpy\_s(segment->elements, sizeof(Var) \* segment->length, vars->elements, sizeof(Var) \* count);

return segment;

}

void JavascriptOperators::AddIntsToArraySegment(SparseArraySegment<int32> \* segment, const Js::AuxArray<int32> \*ints)

{

uint32 count = ints->count;

Assert(segment->left == 0);

Assert(count <= segment->size);

if(count > segment->length)

{

segment->length = count;

}

js\_memcpy\_s(segment->elements, sizeof(int32) \* segment->length, ints->elements, sizeof(int32) \* count);

}

void JavascriptOperators::AddFloatsToArraySegment(SparseArraySegment<double> \* segment, const Js::AuxArray<double> \*doubles)

{

uint32 count = doubles->count;

Assert(segment->left == 0);

Assert(count <= segment->size);

if(count > segment->length)

{

segment->length = count;

}

js\_memcpy\_s(segment->elements, sizeof(double) \* segment->length, doubles->elements, sizeof(double) \* count);

}

RecyclableObject \* JavascriptOperators::GetPrototypeObject(RecyclableObject \* constructorFunction, ScriptContext \* scriptContext)

{

Var prototypeProperty = JavascriptOperators::GetProperty(constructorFunction, PropertyIds::prototype, scriptContext);

RecyclableObject\* prototypeObject;

PrototypeObject(prototypeProperty, constructorFunction, scriptContext, &prototypeObject);

return prototypeObject;

}

RecyclableObject \* JavascriptOperators::GetPrototypeObjectForConstructorCache(RecyclableObject \* constructor, ScriptContext\* requestContext, bool& canBeCached)

{

PropertyValueInfo info;

Var prototypeValue;

RecyclableObject\* prototypeObject;

canBeCached = false;

// Do a local property lookup. Since a function's prototype property is a non-configurable data property, we don't need to worry

// about the prototype being an accessor property, whose getter returns different values every time it's called.

if (constructor->GetProperty(constructor, PropertyIds::prototype, &prototypeValue, &info, requestContext))

{

if (!JavascriptOperators::PrototypeObject(prototypeValue, constructor, requestContext, &prototypeObject))

{

// The value returned by the property lookup is not a valid prototype object, default to object prototype.

Assert(prototypeObject == constructor->GetLibrary()->GetObjectPrototype());

}

// For these scenarios, we do not want to populate the cache.

if (constructor->GetScriptContext() != requestContext || info.GetInstance() != constructor)

{

return prototypeObject;

}

}

else

{

// It's ok to cache Object.prototype, because Object.prototype cannot be overwritten.

prototypeObject = constructor->GetLibrary()->GetObjectPrototype();

}

canBeCached = true;

return prototypeObject;

}

bool JavascriptOperators::PrototypeObject(Var prototypeProperty, RecyclableObject \* constructorFunction, ScriptContext \* scriptContext, RecyclableObject\*\* prototypeObject)

{

TypeId prototypeType = JavascriptOperators::GetTypeId(prototypeProperty);

if (JavascriptOperators::IsObjectType(prototypeType))

{

\*prototypeObject = RecyclableObject::FromVar(prototypeProperty);

return true;

}

\*prototypeObject = constructorFunction->GetLibrary()->GetObjectPrototype();

return false;

}

FunctionInfo\* JavascriptOperators::GetConstructorFunctionInfo(Var instance, ScriptContext \* scriptContext)

{

TypeId typeId = JavascriptOperators::GetTypeId(instance);

if (typeId == TypeIds\_Function)

{

JavascriptFunction \* function = JavascriptFunction::FromVar(instance);

return function->GetFunctionInfo();

}

if (typeId != TypeIds\_HostDispatch && typeId != TypeIds\_Proxy)

{

if (typeId == TypeIds\_Null)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

JavascriptError::ThrowTypeError(scriptContext, VBSERR\_ActionNotSupported);

}

return nullptr;

}

Var JavascriptOperators::NewJavascriptObjectNoArg(ScriptContext\* requestContext)

{

DynamicObject \* newObject = requestContext->GetLibrary()->CreateObject(true);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newObject));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newObject = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(newObject));

}

#endif

return newObject;

}

Var JavascriptOperators::NewJavascriptArrayNoArg(ScriptContext\* requestContext)

{

JavascriptArray \* newArray = requestContext->GetLibrary()->CreateArray();

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newArray));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newArray = static\_cast<JavascriptArray\*>(JavascriptProxy::AutoProxyWrapper(newArray));

}

#endif

return newArray;

}

Var JavascriptOperators::NewScObjectNoArgNoCtorFull(Var instance, ScriptContext\* requestContext)

{

return NewScObjectNoArgNoCtorCommon(instance, requestContext, true);

}

Var JavascriptOperators::NewScObjectNoArgNoCtor(Var instance, ScriptContext\* requestContext)

{

return NewScObjectNoArgNoCtorCommon(instance, requestContext, false);

}

Var JavascriptOperators::NewScObjectNoArgNoCtorCommon(Var instance, ScriptContext\* requestContext, bool isBaseClassConstructorNewScObject)

{

RecyclableObject \* object = RecyclableObject::FromVar(instance);

FunctionInfo\* functionInfo = JavascriptOperators::GetConstructorFunctionInfo(instance, requestContext);

Assert(functionInfo != &JavascriptObject::EntryInfo::NewInstance); // built-ins are not inlined

Assert(functionInfo != &JavascriptArray::EntryInfo::NewInstance); // built-ins are not inlined

return functionInfo != nullptr ?

JavascriptOperators::NewScObjectCommon(object, functionInfo, requestContext, isBaseClassConstructorNewScObject) :

JavascriptOperators::NewScObjectHostDispatchOrProxy(object, requestContext);

}

Var JavascriptOperators::NewScObjectNoArg(Var instance, ScriptContext \* requestContext)

{

if (JavascriptProxy::Is(instance))

{

Arguments args(CallInfo(CallFlags\_New, 1), &instance);

JavascriptProxy\* proxy = JavascriptProxy::FromVar(instance);

return proxy->ConstructorTrap(args, requestContext, 0);

}

FunctionInfo\* functionInfo = JavascriptOperators::GetConstructorFunctionInfo(instance, requestContext);

RecyclableObject \* object = RecyclableObject::FromVar(instance);

if (functionInfo == &JavascriptObject::EntryInfo::NewInstance)

{

// Fast path for new Object()

Assert((functionInfo->GetAttributes() & FunctionInfo::ErrorOnNew) == 0);

JavascriptLibrary\* library = object->GetLibrary();

DynamicObject \* newObject = library->CreateObject(true);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newObject));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newObject = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(newObject));

}

#endif

#if DBG

DynamicType\* newObjectType = newObject->GetDynamicType();

Assert(newObjectType->GetIsShared());

JavascriptFunction\* constructor = JavascriptFunction::FromVar(instance);

Assert(!constructor->GetConstructorCache()->NeedsUpdateAfterCtor());

#endif

ScriptContext \* scriptContext = library->GetScriptContext();

if (scriptContext != requestContext)

{

CrossSite::MarshalDynamicObjectAndPrototype(requestContext, newObject);

}

return newObject;

}

else if (functionInfo == &JavascriptArray::EntryInfo::NewInstance)

{

Assert((functionInfo->GetAttributes() & FunctionInfo::ErrorOnNew) == 0);

JavascriptLibrary\* library = object->GetLibrary();

JavascriptArray \* newArray = library->CreateArray();

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newArray));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newArray = static\_cast<JavascriptArray\*>(JavascriptProxy::AutoProxyWrapper(newArray));

}

#endif

#if DBG

DynamicType\* newArrayType = newArray->GetDynamicType();

Assert(newArrayType->GetIsShared());

JavascriptFunction\* constructor = JavascriptFunction::FromVar(instance);

Assert(!constructor->GetConstructorCache()->NeedsUpdateAfterCtor());

#endif

ScriptContext \* scriptContext = library->GetScriptContext();

if (scriptContext != requestContext)

{

CrossSite::MarshalDynamicObjectAndPrototype(requestContext, newArray);

}

return newArray;

}

Var newObject = functionInfo != nullptr ?

JavascriptOperators::NewScObjectCommon(object, functionInfo, requestContext) :

JavascriptOperators::NewScObjectHostDispatchOrProxy(object, requestContext);

Var returnVar = object->GetEntryPoint()(object, CallInfo(CallFlags\_New, 1), newObject);

if (JavascriptOperators::IsObject(returnVar))

{

newObject = returnVar;

}

ConstructorCache \* constructorCache = nullptr;

if (JavascriptFunction::Is(instance))

{

constructorCache = JavascriptFunction::FromVar(instance)->GetConstructorCache();

}

if (constructorCache != nullptr && constructorCache->NeedsUpdateAfterCtor())

{

JavascriptOperators::UpdateNewScObjectCache(object, newObject, requestContext);

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newObject = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(newObject));

// this might come from a different scriptcontext.

newObject = CrossSite::MarshalVar(requestContext, newObject);

}

#endif

return newObject;

}

Var JavascriptOperators::NewScObjectNoCtorFull(Var instance, ScriptContext\* requestContext)

{

return NewScObjectNoCtorCommon(instance, requestContext, true);

}

Var JavascriptOperators::NewScObjectNoCtor(Var instance, ScriptContext \* requestContext)

{

return NewScObjectNoCtorCommon(instance, requestContext, false);

}

Var JavascriptOperators::NewScObjectNoCtorCommon(Var instance, ScriptContext\* requestContext, bool isBaseClassConstructorNewScObject)

{

FunctionInfo\* functionInfo = JavascriptOperators::GetConstructorFunctionInfo(instance, requestContext);

if (functionInfo)

{

return JavascriptOperators::NewScObjectCommon(RecyclableObject::FromVar(instance), functionInfo, requestContext, isBaseClassConstructorNewScObject);

}

else

{

return JavascriptOperators::NewScObjectHostDispatchOrProxy(RecyclableObject::FromVar(instance), requestContext);

}

}

Var JavascriptOperators::NewScObjectHostDispatchOrProxy(RecyclableObject \* function, ScriptContext \* requestContext)

{

ScriptContext\* functionScriptContext = function->GetScriptContext();

if (JavascriptProxy::Is(function))

{

// We can still call into NewScObjectNoCtor variations in JIT code for performance; however for proxy we don't

// really need the new object as the trap will handle the "this" pointer separately. pass back nullptr to ensure

// failure in invalid case.

return nullptr;

}

RecyclableObject \* prototype = JavascriptOperators::GetPrototypeObject(function, functionScriptContext);

prototype = RecyclableObject::FromVar(CrossSite::MarshalVar(requestContext, prototype));

Var object = requestContext->GetLibrary()->CreateObject(prototype);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(object));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

object = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(object));

}

#endif

return object;

}

Var JavascriptOperators::NewScObjectCommon(RecyclableObject \* function, FunctionInfo\* functionInfo, ScriptContext \* requestContext, bool isBaseClassConstructorNewScObject)

{

// CONSIDER: Allow for the cache to be repopulated if the type got collected, and a new one got populated with

// the same number of inlined slots. This requires that the JIT-ed code actually load the type from the cache

// (instead of hard-coding it), but it can (and must) keep the hard-coded number of inline slots.

// CONSIDER: Consider also not pinning the type in the cache. This can be done by using a registration based

// weak reference (we need to control the memory address), which we don't yet have, or by allocating the cache from

// the inline cache arena to allow it to be zeroed, but retain a recycler-allocated portion to hold on to the size of

// inlined slots.

JavascriptFunction\* constructor = JavascriptFunction::FromVar(function);

if (functionInfo->IsClassConstructor() && !isBaseClassConstructorNewScObject)

{

// If we are calling new on a class constructor, the contract is that we pass new.target as the 'this' argument.

// function is the constructor on which we called new - which is new.target.

// If we are trying to construct the object for a base class constructor as part of a super call, we should not

// store new.target in the 'this' argument.

return function;

}

ConstructorCache\* constructorCache = constructor->GetConstructorCache();

AssertMsg(constructorCache->GetScriptContext() == nullptr || constructorCache->GetScriptContext() == constructor->GetScriptContext(),

"Why did we populate a constructor cache with a mismatched script context?");

Assert(constructorCache != nullptr);

DynamicType\* type = constructorCache->GetGuardValueAsType();

if (type != nullptr && constructorCache->GetScriptContext() == requestContext)

{

#if DBG

bool cachedProtoCanBeCached;

Assert(type->GetPrototype() == JavascriptOperators::GetPrototypeObjectForConstructorCache(constructor, requestContext, cachedProtoCanBeCached));

Assert(cachedProtoCanBeCached);

Assert(type->GetIsShared());

#endif

#if DBG\_DUMP

TraceUseConstructorCache(constructorCache, constructor, true);

#endif

Var object = DynamicObject::New(requestContext->GetRecycler(), type);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(object));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

object = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(object));

}

#endif

return object;

}

if (constructorCache->SkipDefaultNewObject())

{

Assert(!constructorCache->NeedsUpdateAfterCtor());

#if DBG\_DUMP

TraceUseConstructorCache(constructorCache, constructor, true);

#endif

if (isBaseClassConstructorNewScObject)

{

return JavascriptOperators::CreateFromConstructor(function, requestContext);

}

return nullptr;

}

#if DBG\_DUMP

TraceUseConstructorCache(constructorCache, constructor, false);

#endif

ScriptContext\* constructorScriptContext = function->GetScriptContext();

Assert(!constructorScriptContext->GetThreadContext()->IsDisableImplicitException());

// we shouldn't try to call the constructor if it's closed already.

constructorScriptContext->VerifyAlive(TRUE, requestContext);

FunctionInfo::Attributes attributes = functionInfo->GetAttributes();

if (attributes & FunctionInfo::ErrorOnNew)

{

JavascriptError::ThrowTypeError(requestContext, JSERR\_ErrorOnNew);

}

// Slow path

FunctionProxy \* ctorProxy = constructor->GetFunctionProxy();

FunctionBody \* functionBody = ctorProxy != nullptr ? ctorProxy->EnsureDeserialized()->Parse() : nullptr;

if (attributes & FunctionInfo::SkipDefaultNewObject)

{

// The constructor doesn't use the default new object.

#pragma prefast(suppress:6236, "DevDiv bug 830883. False positive when PHASE\_OFF is #defined as '(false)'.")

if (!PHASE\_OFF1(ConstructorCachePhase) && (functionBody == nullptr || !PHASE\_OFF(ConstructorCachePhase, functionBody)))

{

constructorCache = constructor->EnsureValidConstructorCache();

constructorCache->PopulateForSkipDefaultNewObject(constructorScriptContext);

#if DBG\_DUMP

if ((functionBody != nullptr && PHASE\_TRACE(Js::ConstructorCachePhase, functionBody)) || (functionBody == nullptr && PHASE\_TRACE1(Js::ConstructorCachePhase)))

{

const wchar\_t\* ctorName = functionBody != nullptr ? functionBody->GetDisplayName() : L"<unknown>";

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: populated cache (0x%p) for ctor %s (%s): ", constructorCache, ctorName,

functionBody ? functionBody->GetDebugNumberSet(debugStringBuffer) : L"(null)");

constructorCache->Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

Assert(!constructorCache->NeedsUpdateAfterCtor());

return nullptr;

}

// CONSIDER: Create some form of PatchGetProtoObjForCtorCache, which actually caches the prototype object in the constructor cache.

// Make sure that it does NOT populate the guard field. On the slow path (the only path for cross-context calls) we can do a faster lookup

// after we fail the guard check. When invalidating the cache for proto change, make sure we zap the prototype field of the cache in

// addition to the guard value.

bool prototypeCanBeCached;

RecyclableObject\* prototype = JavascriptOperators::GetPrototypeObjectForConstructorCache(function, constructorScriptContext, prototypeCanBeCached);

prototype = RecyclableObject::FromVar(CrossSite::MarshalVar(requestContext, prototype));

DynamicObject\* newObject = requestContext->GetLibrary()->CreateObject(prototype, 8);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(newObject));

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::autoProxyFlag))

{

newObject = DynamicObject::FromVar(JavascriptProxy::AutoProxyWrapper(newObject));

}

#endif

Assert(newObject->GetTypeHandler()->GetPropertyCount() == 0);

if (prototypeCanBeCached && functionBody != nullptr && requestContext == constructorScriptContext &&

!Js::JavascriptProxy::Is(newObject) &&

!PHASE\_OFF1(ConstructorCachePhase) && !PHASE\_OFF(ConstructorCachePhase, functionBody))

{

DynamicType\* newObjectType = newObject->GetDynamicType();

// Initial type (without any properties) should always be shared up-front. This allows us to populate the cache right away.

Assert(newObjectType->GetIsShared());

// Populate the cache here and set the updateAfterCtor flag. This way, if the ctor is called recursively the

// recursive calls will hit the cache and use the initial type. On the unwind path, we will update the cache

// after the innermost ctor and clear the flag. After subsequent ctors we won't attempt an update anymore.

// As long as the updateAfterCtor flag is set it is safe to update the cache, because it would not have been

// hard-coded in the JIT-ed code.

constructorCache = constructor->EnsureValidConstructorCache();

constructorCache->Populate(newObjectType, constructorScriptContext, functionBody->GetHasNoExplicitReturnValue(), true);

Assert(constructorCache->IsConsistent());

#if DBG\_DUMP

if ((functionBody != nullptr && PHASE\_TRACE(Js::ConstructorCachePhase, functionBody)) || (functionBody == nullptr && PHASE\_TRACE1(Js::ConstructorCachePhase)))

{

const wchar\_t\* ctorName = functionBody != nullptr ? functionBody->GetDisplayName() : L"<unknown>";

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: populated cache (0x%p) for ctor %s (%s): ", constructorCache, ctorName,

functionBody ? functionBody->GetDebugNumberSet(debugStringBuffer) : L"(null)");

constructorCache->Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

else

{

#if DBG\_DUMP

if ((functionBody != nullptr && PHASE\_TRACE(Js::ConstructorCachePhase, functionBody)) || (functionBody == nullptr && PHASE\_TRACE1(Js::ConstructorCachePhase)))

{

const wchar\_t\* ctorName = functionBody != nullptr ? functionBody->GetDisplayName() : L"<unknown>";

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: did not populate cache (0x%p) for ctor %s (%s), because %s: prototype = 0x%p, functionBody = 0x%p, ctor context = 0x%p, request context = 0x%p",

constructorCache, ctorName, functionBody ? functionBody->GetDebugNumberSet(debugStringBuffer) : L"(null)",

!prototypeCanBeCached ? L"prototype cannot be cached" :

functionBody == nullptr ? L"function has no body" :

requestContext != constructorScriptContext ? L"of cross-context call" : L"constructor cache phase is off",

prototype, functionBody, constructorScriptContext, requestContext);

Output::Print(L"\n");

Output::Flush();

}

#endif

}

return newObject;

}

void JavascriptOperators::UpdateNewScObjectCache(Var function, Var instance, ScriptContext\* requestContext)

{

JavascriptFunction\* constructor = JavascriptFunction::FromVar(function);

if(constructor->GetScriptContext() != requestContext)

{

// The cache is populated only when the constructor function's context is the same as the calling context. However,

// the cached type is not finalized yet and may not be until multiple calls to the constructor have been made (see

// flag ConstructorCallsRequiredToFinalizeCachedType). A subsequent call to the constructor may be made from a

// different context, so ignore those cross-context calls and wait for the constructor to be called from its own

// context again to finalize the cached type.

return;

}

// Review : What happens if the cache got invalidated between NewScObject and here?

// Should we allocate new? Should we mark it as polymorphic?

ConstructorCache\* constructorCache = constructor->GetConstructorCache();

Assert(constructorCache->IsConsistent());

Assert(!ConstructorCache::IsDefault(constructorCache));

AssertMsg(constructorCache->GetScriptContext() == constructor->GetScriptContext(), "Why did we populate a constructor cache with a mismatched script context?");

AssertMsg(constructorCache->IsPopulated(), "Why are we updating a constructor cache that hasn't been populated?");

// The presence of the updateAfterCtor flag guarantees that this cache hasn't been used in JIT-ed fast path. Even, if the

// cache is invalidated, this flag is not changed.

AssertMsg(constructorCache->NeedsUpdateAfterCtor(), "Why are we updating a constructor cache that doesn't need to be updated?");

const bool finalizeCachedType =

constructorCache->CallCount() >= CONFIG\_FLAG(ConstructorCallsRequiredToFinalizeCachedType);

if(!finalizeCachedType)

{

constructorCache->IncCallCount();

}

else

{

constructorCache->ClearUpdateAfterCtor();

}

FunctionBody\* constructorBody = constructor->GetFunctionBody();

AssertMsg(constructorBody != nullptr, "Constructor function doesn't have a function body.");

Assert(RecyclableObject::Is(instance));

// The cache might have been invalidated between NewScObjectCommon and UpdateNewScObjectCache. This could occur, for example, if

// the constructor updates its own prototype property. If that happens we don't want to re-populate it here. A new cache will

// be created when the constructor is called again.

if (constructorCache->IsInvalidated())

{

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because cache is invalidated");

#endif

return;

}

Assert(constructorCache->GetGuardValueAsType() != nullptr);

if (DynamicType::Is(RecyclableObject::FromVar(instance)->GetTypeId()))

{

DynamicObject \*object = DynamicObject::FromVar(instance);

DynamicType\* type = object->GetDynamicType();

DynamicTypeHandler\* typeHandler = type->GetTypeHandler();

if (constructorBody->GetHasOnlyThisStmts())

{

if (typeHandler->IsSharable())

{

#if DBG

bool cachedProtoCanBeCached;

Assert(type->GetPrototype() == JavascriptOperators::GetPrototypeObjectForConstructorCache(constructor, requestContext, cachedProtoCanBeCached));

Assert(cachedProtoCanBeCached);

Assert(type->GetScriptContext() == constructorCache->GetScriptContext());

Assert(type->GetPrototype() == constructorCache->GetType()->GetPrototype());

#endif

typeHandler->SetMayBecomeShared();

// CONSIDER: Remove only this for delayed type sharing.

type->ShareType();

#if ENABLE\_PROFILE\_INFO

DynamicProfileInfo\* profileInfo = constructorBody->HasDynamicProfileInfo() ? constructorBody->GetAnyDynamicProfileInfo() : nullptr;

if ((profileInfo != nullptr && profileInfo->GetImplicitCallFlags() <= ImplicitCall\_None) ||

CheckIfPrototypeChainHasOnlyWritableDataProperties(type->GetPrototype()))

{

Assert(typeHandler->GetPropertyCount() < Js::PropertyIndexRanges<PropertyIndex>::MaxValue);

for (PropertyIndex pi = 0; pi < typeHandler->GetPropertyCount(); pi++)

{

requestContext->RegisterConstructorCache(typeHandler->GetPropertyId(requestContext, pi), constructorCache);

}

Assert(constructorBody->GetUtf8SourceInfo()->GetIsLibraryCode() || !constructor->GetScriptContext()->IsInDebugMode());

if (constructorCache->TryUpdateAfterConstructor(type, constructor->GetScriptContext()))

{

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, true, L"");

#endif

}

else

{

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because number of slots > MaxCachedSlotCount");

#endif

}

}

#if DBG\_DUMP

else

{

if (profileInfo &&

((profileInfo->GetImplicitCallFlags() & ~(Js::ImplicitCall\_External | Js::ImplicitCall\_Accessor)) == 0) &&

profileInfo != nullptr && CheckIfPrototypeChainHasOnlyWritableDataProperties(type->GetPrototype()) &&

Js::Configuration::Global.flags.Trace.IsEnabled(Js::HostOptPhase))

{

const wchar\_t\* ctorName = constructorBody->GetDisplayName();

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: %s cache (0x%p) for ctor %s (#%u) did not update because external call",

constructorCache, constructorBody, ctorName, constructorBody ? constructorBody->GetDebugNumberSet(debugStringBuffer) : L"(null)");

Output::Print(L"\n");

Output::Flush();

}

}

#endif

#endif

}

else

{

// Dynamic type created is not sharable.

// So in future don't try to check for "this assignment optimization".

constructorBody->SetHasOnlyThisStmts(false);

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because final type is not shareable");

#endif

}

}

else

{

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because ctor has not only this statements");

#endif

}

}

else

{

// Even though this constructor apparently returned something other than the default object we created,

// it still makes sense to cache the parameters of the default object, since we must create it every time, anyway.

#if DBG\_DUMP

TraceUpdateConstructorCache(constructorCache, constructorBody, false, L"because ctor return a non-object value");

#endif

return;

}

// Whatever the constructor returned, if we're caching a type we want to be sure we shrink its inline slot capacity.

if (finalizeCachedType && constructorCache->IsEnabled())

{

DynamicType\* cachedType = constructorCache->NeedsTypeUpdate() ? constructorCache->GetPendingType() : constructorCache->GetType();

DynamicTypeHandler\* cachedTypeHandler = cachedType->GetTypeHandler();

// Consider: We could delay inline slot capacity shrinking until the second time this constructor is invoked. In some cases

// this might permit more properties to remain inlined if the objects grow after constructor. This would require flagging

// the cache as special (already possible) and forcing the shrinking during work item creation if we happen to JIT this

// constructor while the cache is in this special state.

if (cachedTypeHandler->GetInlineSlotCapacity())

{

#if DBG\_DUMP

int inlineSlotCapacityBeforeShrink = cachedTypeHandler->GetInlineSlotCapacity();

#endif

// Note that after the cache has been updated and might have been used in the JIT-ed code, it is no longer legal to

// shrink the inline slot capacity of the type. That's because we allocate memory for a fixed number of inlined properties

// and if that number changed on the type, this update wouldn't get reflected in JIT-ed code and we would allocate objects

// of a wrong size. This could conceivably happen if the original object got collected, and with it some of the successor

// types also. If then another constructor has the same prototype and needs to populate its own cache, it would attempt to

// shrink inlined slots again. If all surviving type handlers have smaller inline slot capacity, we would shrink it further.

// To address this problem the type handler has a bit indicating its inline slots have been shrunk already. If that bit is

// set ShrinkSlotAndInlineSlotCapacity does nothing.

cachedTypeHandler->ShrinkSlotAndInlineSlotCapacity();

constructorCache->UpdateInlineSlotCount();

#if DBG\_DUMP

Assert(inlineSlotCapacityBeforeShrink >= cachedTypeHandler->GetInlineSlotCapacity());

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::InlineSlotsPhase))

{

if (inlineSlotCapacityBeforeShrink != cachedTypeHandler->GetInlineSlotCapacity())

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"Inline slot capacity shrunk: Function:%04s Before:%d After:%d\n",

constructorBody->GetDebugNumberSet(debugStringBuffer), inlineSlotCapacityBeforeShrink, cachedTypeHandler->GetInlineSlotCapacity());

}

}

#endif

}

}

}

void JavascriptOperators::TraceUseConstructorCache(const ConstructorCache\* ctorCache, const JavascriptFunction\* ctor, bool isHit)

{

#if DBG\_DUMP

// We are under debug, so we can incur the extra check here.

FunctionProxy\* ctorBody = ctor->GetFunctionProxy();

if (ctorBody != nullptr && !ctorBody->GetScriptContext()->IsClosed())

{

ctorBody = ctorBody->EnsureDeserialized();

}

if ((ctorBody != nullptr && PHASE\_TRACE(Js::ConstructorCachePhase, ctorBody)) || (ctorBody == nullptr && PHASE\_TRACE1(Js::ConstructorCachePhase)))

{

const wchar\_t\* ctorName = ctorBody != nullptr ? ctorBody->GetDisplayName() : L"<unknown>";

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: %s cache (0x%p) for ctor %s (%s): ", isHit ? L"hit" : L"missed", ctorCache, ctorName,

ctorBody ? ctorBody->GetDebugNumberSet(debugStringBuffer) : L"(null)");

ctorCache->Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

void JavascriptOperators::TraceUpdateConstructorCache(const ConstructorCache\* ctorCache, const FunctionBody\* ctorBody, bool updated, const wchar\_t\* reason)

{

#if DBG\_DUMP

if (PHASE\_TRACE(Js::ConstructorCachePhase, ctorBody))

{

const wchar\_t\* ctorName = ctorBody->GetDisplayName();

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"CtorCache: %s cache (0x%p) for ctor %s (%s)%s %s: ",

updated ? L"updated" : L"did not update", ctorBody, ctorName,

ctorBody ? const\_cast<Js::FunctionBody \*>(ctorBody)->GetDebugNumberSet(debugStringBuffer) : L"(null)",

updated ? L"" : L", because" , reason);

ctorCache->Dump();

Output::Print(L"\n");

Output::Flush();

}

#endif

}

Var JavascriptOperators::NewScObject(const Var callee, const Arguments args, ScriptContext \*const scriptContext, const Js::AuxArray<uint32> \*spreadIndices)

{

Assert(callee);

Assert(args.Info.Count != 0);

Assert(scriptContext);

// Always save and restore implicit call flags when calling out

// REVIEW: Can we avoid it if we don't collect dynamic profile info?

ThreadContext \*const threadContext = scriptContext->GetThreadContext();

const ImplicitCallFlags savedImplicitCallFlags = threadContext->GetImplicitCallFlags();

const Var newVarInstance = JavascriptFunction::CallAsConstructor(callee, /\* overridingNewTarget = \*/nullptr, args, scriptContext, spreadIndices);

threadContext->SetImplicitCallFlags(savedImplicitCallFlags);

return newVarInstance;

}

Js::GlobalObject \* JavascriptOperators::OP\_LdRoot(ScriptContext\* scriptContext)

{

return scriptContext->GetGlobalObject();

}

Js::ModuleRoot \* JavascriptOperators::GetModuleRoot(int moduleID, ScriptContext\* scriptContext)

{

Assert(moduleID != kmodGlobal);

JavascriptLibrary\* library = scriptContext->GetLibrary();

HostObjectBase \*hostObject = library->GetGlobalObject()->GetHostObject();

if (hostObject)

{

Js::ModuleRoot \* moduleRoot = hostObject->GetModuleRoot(moduleID);

Assert(!CrossSite::NeedMarshalVar(moduleRoot, scriptContext));

return moduleRoot;

}

HostScriptContext \*hostScriptContext = scriptContext->GetHostScriptContext();

if (hostScriptContext)

{

Js::ModuleRoot \* moduleRoot = hostScriptContext->GetModuleRoot(moduleID);

Assert(!CrossSite::NeedMarshalVar(moduleRoot, scriptContext));

return moduleRoot;

}

Assert(FALSE);

return nullptr;

}

Var JavascriptOperators::OP\_LoadModuleRoot(int moduleID, ScriptContext\* scriptContext)

{

Js::ModuleRoot \* moduleRoot = GetModuleRoot(moduleID, scriptContext);

if (moduleRoot)

{

return moduleRoot;

}

Assert(false);

return scriptContext->GetLibrary()->GetUndefined();

}

Var JavascriptOperators::OP\_LdNull(ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetNull();

}

Var JavascriptOperators::OP\_LdUndef(ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetUndefined();

}

Var JavascriptOperators::OP\_LdNaN(ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetNaN();

}

Var JavascriptOperators::OP\_LdInfinity(ScriptContext\* scriptContext)

{

return scriptContext->GetLibrary()->GetPositiveInfinite();

}

void JavascriptOperators::BuildHandlerScope(Var argThis, RecyclableObject \* hostObject, FrameDisplay \* pDisplay, ScriptContext \* scriptContext)

{

Assert(argThis != nullptr);

pDisplay->SetItem(0, TaggedNumber::Is(argThis) ? scriptContext->GetLibrary()->CreateNumberObject(argThis) : argThis);

uint16 i = 1;

Var aChild = argThis;

uint16 length = pDisplay->GetLength();

// Now add any parent scopes

// We need to support the namespace parent lookup in both fastDOM on and off scenario.

while (aChild != NULL)

{

Var aParent = hostObject->GetNamespaceParent(aChild);

if (aParent == nullptr)

{

break;

}

aParent = CrossSite::MarshalVar(scriptContext, aParent);

if (i == length)

{

length += 8;

FrameDisplay \* tmp = RecyclerNewPlus(scriptContext->GetRecycler(), length \* sizeof(void\*), FrameDisplay, length);

js\_memcpy\_s((char\*)tmp + tmp->GetOffsetOfScopes(), tmp->GetLength() \* sizeof(void \*), (char\*)pDisplay + pDisplay->GetOffsetOfScopes(), pDisplay->GetLength() \* sizeof(void\*));

pDisplay = tmp;

}

pDisplay->SetItem(i, aParent);

aChild = aParent;

i++;

}

Assert(i <= pDisplay->GetLength());

pDisplay->SetLength(i);

}

FrameDisplay \* JavascriptOperators::OP\_LdHandlerScope(Var argThis, ScriptContext\* scriptContext)

{

// The idea here is to build a stack of nested scopes in the form of a JS array.

//

// The scope stack for an event handler looks like this:

//

// implicit "this"

// implicit namespace parent scopes

// Put the implicit "this"

if (argThis != NULL)

{

RecyclableObject\* hostObject = scriptContext->GetGlobalObject()->GetHostObject();

if (hostObject == nullptr)

{

hostObject = scriptContext->GetGlobalObject()->GetDirectHostObject();

}

if (hostObject != nullptr)

{

uint16 length = 7;

FrameDisplay \*pDisplay =

RecyclerNewPlus(scriptContext->GetRecycler(), length \* sizeof(void\*), FrameDisplay, length);

BuildHandlerScope(argThis, hostObject, pDisplay, scriptContext);

return pDisplay;

}

}

return const\_cast<FrameDisplay \*>(&Js::NullFrameDisplay);

}

FrameDisplay\* JavascriptOperators::OP\_LdFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext)

{

// Build a display of nested frame objects.

// argHead is the current scope; argEnv is either the lone trailing scope or an array of scopes

// which we append to the new display.

// Note that there are cases in which a function with no local frame must construct a display to pass

// to the function(s) nested within it. In such a case, argHead will be a null object, and it's not

// strictly necessary to include it. But such cases are rare and not perf critical, so it's not

// worth the extra complexity to notify the nested functions that they can "skip" this slot in the

// frame display when they're loading scopes nested outside it.

FrameDisplay \*pDisplay = nullptr;

FrameDisplay \*envDisplay = (FrameDisplay\*)argEnv;

uint16 length = envDisplay->GetLength() + 1;

pDisplay = RecyclerNewPlus(scriptContext->GetRecycler(), length \* sizeof(void\*), FrameDisplay, length);

for (int j = 0; j < length - 1; j++)

{

pDisplay->SetItem(j + 1, envDisplay->GetItem(j));

}

pDisplay->SetItem(0, argHead);

return pDisplay;

}

FrameDisplay\* JavascriptOperators::OP\_LdFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext)

{

return OP\_LdFrameDisplay(argHead, (void\*)&NullFrameDisplay, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdStrictFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext)

{

FrameDisplay \* pDisplay = OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

pDisplay->SetStrictMode(true);

return pDisplay;

}

FrameDisplay\* JavascriptOperators::OP\_LdStrictFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext)

{

return OP\_LdStrictFrameDisplay(argHead, (void\*)&StrictNullFrameDisplay, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext)

{

CheckInnerFrameDisplayArgument(argHead);

return OP\_LdFrameDisplay(argHead, argEnv, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdInnerFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext)

{

CheckInnerFrameDisplayArgument(argHead);

return OP\_LdFrameDisplayNoParent(argHead, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdStrictInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext)

{

CheckInnerFrameDisplayArgument(argHead);

return OP\_LdStrictFrameDisplay(argHead, argEnv, scriptContext);

}

FrameDisplay\* JavascriptOperators::OP\_LdStrictInnerFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext)

{

CheckInnerFrameDisplayArgument(argHead);

return OP\_LdStrictFrameDisplayNoParent(argHead, scriptContext);

}

void JavascriptOperators::CheckInnerFrameDisplayArgument(void \*argHead)

{

if (ThreadContext::IsOnStack(argHead))

{

AssertMsg(false, "Illegal byte code: stack object as with scope");

Js::Throw::FatalInternalError();

}

if (!RecyclableObject::Is(argHead))

{

AssertMsg(false, "Illegal byte code: non-object as with scope");

Js::Throw::FatalInternalError();

}

}

Js::PropertyId JavascriptOperators::GetPropertyId(Var propertyName, ScriptContext\* scriptContext)

{

PropertyRecord const \* propertyRecord = nullptr;

if (JavascriptSymbol::Is(propertyName))

{

propertyRecord = JavascriptSymbol::FromVar(propertyName)->GetValue();

}

else if (JavascriptSymbolObject::Is(propertyName))

{

propertyRecord = JavascriptSymbolObject::FromVar(propertyName)->GetValue();

}

else

{

JavascriptString \* indexStr = JavascriptConversion::ToString(propertyName, scriptContext);

scriptContext->GetOrAddPropertyRecord(indexStr->GetString(), indexStr->GetLength(), &propertyRecord);

}

return propertyRecord->GetPropertyId();

}

void JavascriptOperators::OP\_InitSetter(Var object, PropertyId propertyId, Var setter)

{

AssertMsg(!TaggedNumber::Is(object), "SetMember on a non-object?");

RecyclableObject::FromVar(object)->SetAccessors(propertyId, nullptr, setter);

}

void JavascriptOperators::OP\_InitClassMemberSet(Var object, PropertyId propertyId, Var setter)

{

JavascriptOperators::OP\_InitSetter(object, propertyId, setter);

RecyclableObject::FromVar(object)->SetAttributes(propertyId, PropertyClassMemberDefaults);

}

Js::PropertyId JavascriptOperators::OP\_InitElemSetter(Var object, Var elementName, Var setter, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

AssertMsg(!TaggedNumber::Is(object), "SetMember on a non-object?");

PropertyId propertyId = JavascriptOperators::GetPropertyId(elementName, scriptContext);

RecyclableObject::FromVar(object)->SetAccessors(propertyId, nullptr, setter);

return propertyId;

}

void JavascriptOperators::OP\_InitClassMemberSetComputedName(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

Js::PropertyId propertyId = JavascriptOperators::OP\_InitElemSetter(object, elementName, value, scriptContext);

RecyclableObject\* instance = RecyclableObject::FromVar(object);

// instance will be a function if it is the class constructor (otherwise it would be an object)

if (JavascriptFunction::Is(instance) && Js::PropertyIds::prototype == propertyId)

{

// It is a TypeError to have a static member with a computed name that evaluates to 'prototype'

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ClassStaticMethodCannotBePrototype);

}

instance->SetAttributes(propertyId, PropertyClassMemberDefaults);

}

BOOL JavascriptOperators::IsClassConstructor(Var instance)

{

return JavascriptFunction::Is(instance) && (JavascriptFunction::FromVar(instance)->GetFunctionInfo()->IsClassConstructor() || !JavascriptFunction::FromVar(instance)->IsScriptFunction());

}

void JavascriptOperators::OP\_InitGetter(Var object, PropertyId propertyId, Var getter)

{

AssertMsg(!TaggedNumber::Is(object), "GetMember on a non-object?");

RecyclableObject::FromVar(object)->SetAccessors(propertyId, getter, nullptr);

}

void JavascriptOperators::OP\_InitClassMemberGet(Var object, PropertyId propertyId, Var getter)

{

JavascriptOperators::OP\_InitGetter(object, propertyId, getter);

RecyclableObject::FromVar(object)->SetAttributes(propertyId, PropertyClassMemberDefaults);

}

Js::PropertyId JavascriptOperators::OP\_InitElemGetter(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

AssertMsg(!TaggedNumber::Is(object), "GetMember on a non-object?");

PropertyId propertyId = JavascriptOperators::GetPropertyId(elementName, scriptContext);

RecyclableObject::FromVar(object)->SetAccessors(propertyId, getter, nullptr);

return propertyId;

}

void JavascriptOperators::OP\_InitClassMemberGetComputedName(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

Js::PropertyId propertyId = JavascriptOperators::OP\_InitElemGetter(object, elementName, value, scriptContext);

RecyclableObject\* instance = RecyclableObject::FromVar(object);

// instance will be a function if it is the class constructor (otherwise it would be an object)

if (JavascriptFunction::Is(instance) && Js::PropertyIds::prototype == propertyId)

{

// It is a TypeError to have a static member with a computed name that evaluates to 'prototype'

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ClassStaticMethodCannotBePrototype);

}

instance->SetAttributes(propertyId, PropertyClassMemberDefaults);

}

void JavascriptOperators::OP\_InitComputedProperty(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

PropertyId propertyId = JavascriptOperators::GetPropertyId(elementName, scriptContext);

RecyclableObject::FromVar(object)->InitProperty(propertyId, value, flags);

}

void JavascriptOperators::OP\_InitClassMemberComputedName(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags)

{

PropertyId propertyId = JavascriptOperators::GetPropertyId(elementName, scriptContext);

RecyclableObject\* instance = RecyclableObject::FromVar(object);

// instance will be a function if it is the class constructor (otherwise it would be an object)

if (JavascriptFunction::Is(instance) && Js::PropertyIds::prototype == propertyId)

{

// It is a TypeError to have a static member with a computed name that evaluates to 'prototype'

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ClassStaticMethodCannotBePrototype);

}

instance->SetPropertyWithAttributes(propertyId, value, PropertyClassMemberDefaults, NULL, flags);

}

//

// Used by object literal {..., \_\_proto\_\_: ..., }.

// When \_\_proto\_\_ is enabled, it is effectively same as StFld. However when \_\_proto\_\_ is disabled, it functions same as InitFld.

//

void JavascriptOperators::OP\_InitProto(Var instance, PropertyId propertyId, Var value)

{

AssertMsg(RecyclableObject::Is(instance), "\_\_proto\_\_ member on a non-object?");

Assert(propertyId == PropertyIds::\_\_proto\_\_);

RecyclableObject\* object = RecyclableObject::FromVar(instance);

ScriptContext\* scriptContext = object->GetScriptContext();

if (scriptContext->GetConfig()->Is\_\_proto\_\_Enabled())

{

// B.3.1 \_\_proto\_\_\_ Property Names in Object Initializers

//6.If propKey is the string value "\_\_proto\_\_" and if isComputedPropertyName(propKey) is false, then

// a.If Type(v) is either Object or Null, then

// i.Return the result of calling the [[SetInheritance]] internal method of object with argument propValue.

// b.Return NormalCompletion(empty).

if (JavascriptOperators::IsObjectOrNull(value))

{

JavascriptObject::ChangePrototype(object, RecyclableObject::FromVar(value), /\*validate\*/false, scriptContext);

}

}

else

{

object->InitProperty(propertyId, value);

}

}

Var JavascriptOperators::ConvertToUnmappedArguments(HeapArgumentsObject \*argumentsObject,

uint32 paramCount,

Var \*paramAddr,

DynamicObject\* frameObject,

Js::PropertyIdArray \*propIds,

uint32 formalsCount,

ScriptContext\* scriptContext)

{

Var \*paramIter = paramAddr;

uint32 i = 0;

for (paramIter = paramAddr + i; i < paramCount; i++, paramIter++)

{

JavascriptOperators::SetItem(argumentsObject, argumentsObject, i, \*paramIter, scriptContext, PropertyOperation\_None, /\* skipPrototypeCheck = \*/ TRUE);

}

argumentsObject = argumentsObject->ConvertToUnmappedArgumentsObject();

// Now as the unmapping is done we need to fill those frame object with Undecl

for (i = 0; i < formalsCount; i++)

{

frameObject->SetSlot(SetSlotArguments(propIds != nullptr ? propIds->elements[i] : Js::Constants::NoProperty, i, scriptContext->GetLibrary()->GetUndeclBlockVar()));

}

return argumentsObject;

}

Var JavascriptOperators::LoadHeapArguments(JavascriptFunction \*funcCallee, uint32 paramCount, Var \*paramAddr, Var frameObj, Var vArray, ScriptContext\* scriptContext, bool nonSimpleParamList)

{

AssertMsg(paramCount != (unsigned int)-1, "Loading the arguments object in the global function?");

// Create and initialize the Arguments object.

uint32 formalsCount = 0;

Js::PropertyIdArray \*propIds = nullptr;

if (vArray != scriptContext->GetLibrary()->GetNull())

{

propIds = (Js::PropertyIdArray \*)vArray;

formalsCount = propIds->count;

}

HeapArgumentsObject \*argsObj = JavascriptOperators::CreateHeapArguments(funcCallee, paramCount, formalsCount, frameObj, scriptContext);

// Transfer formal arguments (that were actually passed) from their ArgIn slots to the local frame object.

uint32 i;

Var \*tmpAddr = paramAddr;

if (propIds != nullptr)

{

ActivationObject\* frameObject = (ActivationObject\*)frameObj;

// No fixed fields for formal parameters of the arguments object. Also, mark all fields as initialized up-front, because

// we will set them directly using SetSlot below, so the type handler will not have a chance to mark them as initialized later.

// CONSIDER : When we delay type sharing until the second instance is created, pass an argument indicating we want the types

// and handlers created here to be marked as shared up-front. This is to ensure we don't get any fixed fields and that the handler

// is ready for storing values directly to slots.

DynamicType\* newType = PathTypeHandlerBase::CreateNewScopeObject(scriptContext, frameObject->GetDynamicType(), propIds, nonSimpleParamList ? PropertyLetDefaults : PropertyNone);

int oldSlotCapacity = frameObject->GetDynamicType()->GetTypeHandler()->GetSlotCapacity();

int newSlotCapacity = newType->GetTypeHandler()->GetSlotCapacity();

\_\_analysis\_assume((uint32)newSlotCapacity >= formalsCount);

frameObject->EnsureSlots(oldSlotCapacity, newSlotCapacity, scriptContext, newType->GetTypeHandler());

frameObject->ReplaceType(newType);

if (nonSimpleParamList)

{

return ConvertToUnmappedArguments(argsObj, paramCount, paramAddr, frameObject, propIds, formalsCount, scriptContext);

}

for (i = 0; i < formalsCount && i < paramCount; i++, tmpAddr++)

{

frameObject->SetSlot(SetSlotArguments(propIds->elements[i], i, \*tmpAddr));

}

if (i < formalsCount)

{

// The formals that weren't passed still need to be put in the frame object so that

// their names will be found. Initialize them to "undefined".

for (; i < formalsCount; i++)

{

frameObject->SetSlot(SetSlotArguments(propIds->elements[i], i, scriptContext->GetLibrary()->GetUndefined()));

}

}

}

// Transfer the unnamed actual arguments, if any, to the Arguments object itself.

for (i = formalsCount, tmpAddr = paramAddr + i; i < paramCount; i++, tmpAddr++)

{

// ES5 10.6.11: use [[DefineOwnProperty]] semantics (instead of [[Put]]):

// do not check whether property is non-writable/etc in the prototype.

// ES3 semantics is same.

JavascriptOperators::SetItem(argsObj, argsObj, i, \*tmpAddr, scriptContext, PropertyOperation\_None, /\* skipPrototypeCheck = \*/ TRUE);

}

if (funcCallee->IsStrictMode())

{

// If the formals are let decls, then we just overwrote the frame object slots with

// Undecl sentinels, and we can use the original arguments that were passed to the HeapArgumentsObject.

return argsObj->ConvertToUnmappedArgumentsObject(!nonSimpleParamList);

}

return argsObj;

}

Var JavascriptOperators::LoadHeapArgsCached(JavascriptFunction \*funcCallee, uint32 actualsCount, uint32 formalsCount, Var \*paramAddr, Var frameObj, ScriptContext\* scriptContext, bool nonSimpleParamList)

{

// Disregard the "this" param.

AssertMsg(actualsCount != (uint32)-1 && formalsCount != (uint32)-1,

"Loading the arguments object in the global function?");

// Create and initialize the Arguments object.

HeapArgumentsObject \*argsObj = JavascriptOperators::CreateHeapArguments(funcCallee, actualsCount, formalsCount, frameObj, scriptContext);

// Transfer formal arguments (that were actually passed) from their ArgIn slots to the local frame object.

uint32 i;

Var \*tmpAddr = paramAddr;

if (formalsCount != 0)

{

DynamicObject\* frameObject = DynamicObject::FromVar(frameObj);

\_\_analysis\_assume((uint32)frameObject->GetDynamicType()->GetTypeHandler()->GetSlotCapacity() >= formalsCount);

if (nonSimpleParamList)

{

return ConvertToUnmappedArguments(argsObj, actualsCount, paramAddr, frameObject, nullptr /\*propIds\*/, formalsCount, scriptContext);

}

for (i = 0; i < formalsCount && i < actualsCount; i++, tmpAddr++)

{

// We don't know the propertyId at this point.

frameObject->SetSlot(SetSlotArguments(Constants::NoProperty, i, \*tmpAddr));

}

if (i < formalsCount)

{

// The formals that weren't passed still need to be put in the frame object so that

// their names will be found. Initialize them to "undefined".

for (; i < formalsCount; i++)

{

// We don't know the propertyId at this point.

frameObject->SetSlot(SetSlotArguments(Constants::NoProperty, i, scriptContext->GetLibrary()->GetUndefined()));

}

}

}

// Transfer the unnamed actual arguments, if any, to the Arguments object itself.

for (i = formalsCount, tmpAddr = paramAddr + i; i < actualsCount; i++, tmpAddr++)

{

// ES5 10.6.11: use [[DefineOwnProperty]] semantics (instead of [[Put]]):

// do not check whether property is non-writable/etc in the prototype.

// ES3 semantics is same.

JavascriptOperators::SetItem(argsObj, argsObj, i, \*tmpAddr, scriptContext, PropertyOperation\_None, /\* skipPrototypeCheck = \*/ TRUE);

}

if (funcCallee->IsStrictMode())

{

// If the formals are let decls, then we just overwrote the frame object slots with

// Undecl sentinels, and we can use the original arguments that were passed to the HeapArgumentsObject.

return argsObj->ConvertToUnmappedArgumentsObject(!nonSimpleParamList);

}

return argsObj;

}

HeapArgumentsObject \*JavascriptOperators::CreateHeapArguments(JavascriptFunction \*funcCallee, uint32 actualsCount, uint32 formalsCount, Var frameObj, ScriptContext\* scriptContext)

{

JavascriptLibrary \*library = scriptContext->GetLibrary();

HeapArgumentsObject \*argsObj = library->CreateHeapArguments(frameObj, formalsCount);

//

// Set the number of arguments of Arguments Object

//

argsObj->SetNumberOfArguments(actualsCount);

JavascriptOperators::SetProperty(argsObj, argsObj, PropertyIds::length, JavascriptNumber::ToVar(actualsCount, scriptContext), scriptContext);

if (scriptContext->GetConfig()->IsES6IteratorsEnabled())

{

JavascriptOperators::SetProperty(argsObj, argsObj, PropertyIds::\_symbolIterator, library->GetArrayPrototypeValuesFunction(), scriptContext);

}

if (funcCallee->IsStrictMode())

{

PropertyDescriptor propertyDescriptorCaller;

JavascriptFunction\* callerAccessor = library->GetThrowTypeErrorCallerAccessorFunction();

propertyDescriptorCaller.SetGetter(callerAccessor);

propertyDescriptorCaller.SetSetter(callerAccessor);

propertyDescriptorCaller.SetEnumerable(false);

propertyDescriptorCaller.SetConfigurable(false);

argsObj->SetAccessors(PropertyIds::caller, callerAccessor, callerAccessor, PropertyOperation\_NonFixedValue);

JavascriptOperators::SetAttributes(argsObj, PropertyIds::caller, propertyDescriptorCaller, false);

PropertyDescriptor propertyDescriptorCallee;

JavascriptFunction\* calleeAccessor = library->GetThrowTypeErrorCalleeAccessorFunction();

propertyDescriptorCallee.SetGetter(calleeAccessor);

propertyDescriptorCallee.SetSetter(calleeAccessor);

propertyDescriptorCallee.SetEnumerable(false);

propertyDescriptorCallee.SetConfigurable(false);

argsObj->SetAccessors(PropertyIds::callee, calleeAccessor, calleeAccessor, PropertyOperation\_NonFixedValue);

JavascriptOperators::SetAttributes(argsObj, PropertyIds::callee, propertyDescriptorCallee, false);

}

else

{

JavascriptOperators::SetProperty(argsObj, argsObj, PropertyIds::callee,

StackScriptFunction::EnsureBoxed(BOX\_PARAM(funcCallee, nullptr, L"callee")), scriptContext);

}

return argsObj;

}

Var JavascriptOperators::OP\_NewScopeObject(ScriptContext\*scriptContext)

{

return scriptContext->GetLibrary()->CreateActivationObject();

}

Var\* JavascriptOperators::OP\_NewScopeSlots(unsigned int size, ScriptContext \*scriptContext, Var scope)

{

Assert(size > ScopeSlots::FirstSlotIndex); // Should never see empty slot array

Var\* slotArray = RecyclerNewArray(scriptContext->GetRecycler(), Var, size); // last initialized slot contains reference to array of propertyIds, correspondent to objects in previous slots

uint count = size - ScopeSlots::FirstSlotIndex;

ScopeSlots slots(slotArray);

slots.SetCount(count);

slots.SetScopeMetadata(scope);

Var undef = scriptContext->GetLibrary()->GetUndefined();

for (unsigned int i = 0; i < count; i++)

{

slots.Set(i, undef);

}

return slotArray;

}

Var\* JavascriptOperators::OP\_NewScopeSlotsWithoutPropIds(unsigned int count, int scopeIndex, ScriptContext \*scriptContext, FunctionBody \*functionBody)

{

DebuggerScope\* scope = Constants::FunctionBodyUnavailable;

if (scopeIndex != DebuggerScope::InvalidScopeIndex)

{

AssertMsg(functionBody->GetScopeObjectChain(), "A scope chain should always be created when there are new scope slots for blocks.");

scope = functionBody->GetScopeObjectChain()->pScopeChain->Item(scopeIndex);

}

return OP\_NewScopeSlots(count, scriptContext, scope);

}

Var\* JavascriptOperators::OP\_CloneScopeSlots(Var \*slotArray, ScriptContext \*scriptContext)

{

ScopeSlots slots(slotArray);

uint size = ScopeSlots::FirstSlotIndex + slots.GetCount();

Var\* slotArrayClone = RecyclerNewArray(scriptContext->GetRecycler(), Var, size);

memcpy\_s(slotArrayClone, sizeof(Var) \* size, slotArray, sizeof(Var) \* size);

return slotArrayClone;

}

Var JavascriptOperators::OP\_NewPseudoScope(ScriptContext \*scriptContext)

{

return scriptContext->GetLibrary()->CreatePseudoActivationObject();

}

Var JavascriptOperators::OP\_NewBlockScope(ScriptContext \*scriptContext)

{

return scriptContext->GetLibrary()->CreateBlockActivationObject();

}

Var JavascriptOperators::OP\_CloneBlockScope(BlockActivationObject \*blockScope, ScriptContext \*scriptContext)

{

return blockScope->Clone(scriptContext);

}

Var JavascriptOperators::OP\_IsInst(Var instance, Var aClass, ScriptContext\* scriptContext, IsInstInlineCache\* inlineCache)

{

if (!RecyclableObject::Is(aClass))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Operand\_Invalid\_NeedFunction, L"instanceof");

}

RecyclableObject\* constructor = RecyclableObject::FromVar(aClass);

if (scriptContext->GetConfig()->IsES6HasInstanceEnabled())

{

Var instOfHandler = JavascriptOperators::GetProperty(constructor, PropertyIds::\_symbolHasInstance, scriptContext);

if (JavascriptOperators::IsUndefinedObject(instOfHandler))

{

return JavascriptBoolean::ToVar(constructor->HasInstance(instance, scriptContext, inlineCache), scriptContext);

}

else

{

if (!JavascriptConversion::IsCallable(instOfHandler))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, L"Symbol[Symbol.hasInstance]");

}

RecyclableObject \*instFunc = RecyclableObject::FromVar(instOfHandler);

Js::Var values[2];

Js::CallInfo info(Js::CallFlags\_Value, 2);

Js::Arguments args(info, values);

values[0] = constructor;

values[1] = instance;

Var result = JavascriptFunction::CallFunction<true>(instFunc, instFunc->GetEntryPoint(), args);

return JavascriptBoolean::ToVar(JavascriptConversion::ToBoolean(result, scriptContext) ? TRUE : FALSE, scriptContext);

}

}

else

{

return JavascriptBoolean::ToVar(constructor->HasInstance(instance, scriptContext, inlineCache), scriptContext);

}

}

void JavascriptOperators::OP\_InitClass(Var constructor, Var extends, ScriptContext \* scriptContext)

{

if (JavascriptOperators::GetTypeId(constructor) != Js::TypeId::TypeIds\_Function)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Operand\_Invalid\_NeedFunction, L"class");

}

RecyclableObject \* ctor = RecyclableObject::FromVar(constructor);

// This is a circular reference to the constructor, it associate the constructor with the class and also allows us to check if a

// function is a constructor by comparing the homeObj to the this pointer. see ScriptFunction::IsClassConstructor() for implementation

JavascriptOperators::OP\_SetHomeObj(constructor, constructor);

if (extends)

{

switch (JavascriptOperators::GetTypeId(extends))

{

case Js::TypeId::TypeIds\_Null:

{

Var ctorProto = JavascriptOperators::GetProperty(constructor, ctor, Js::PropertyIds::prototype, scriptContext);

RecyclableObject \* ctorProtoObj = RecyclableObject::FromVar(ctorProto);

ctorProtoObj->SetPrototype(RecyclableObject::FromVar(extends));

ctorProtoObj->EnsureProperty(Js::PropertyIds::constructor);

ctorProtoObj->SetEnumerable(Js::PropertyIds::constructor, FALSE);

break;

}

default:

{

if (!RecyclableObject::Is(extends))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_InvalidPrototype, L"extends");

}

RecyclableObject \* extendsObj = RecyclableObject::FromVar(extends);

if (!JavascriptOperators::IsConstructor(extendsObj))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_ErrorOnNew);

}

if (!extendsObj->HasProperty(Js::PropertyIds::prototype))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_InvalidPrototype);

}

Var extendsProto = JavascriptOperators::GetProperty(extends, extendsObj, Js::PropertyIds::prototype, scriptContext);

uint extendsProtoTypeId = JavascriptOperators::GetTypeId(extendsProto);

if (extendsProtoTypeId <= Js::TypeId::TypeIds\_LastJavascriptPrimitiveType && extendsProtoTypeId != Js::TypeId::TypeIds\_Null)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_InvalidPrototype);

}

Var ctorProto = JavascriptOperators::GetProperty(constructor, ctor, Js::PropertyIds::prototype, scriptContext);

RecyclableObject \* ctorProtoObj = RecyclableObject::FromVar(ctorProto);

ctorProtoObj->SetPrototype(RecyclableObject::FromVar(extendsProto));

ctorProtoObj->EnsureProperty(Js::PropertyIds::constructor);

ctorProtoObj->SetEnumerable(Js::PropertyIds::constructor, FALSE);

Var protoCtor = JavascriptOperators::GetProperty(ctorProto, ctorProtoObj, Js::PropertyIds::constructor, scriptContext);

RecyclableObject \* protoCtorObj = RecyclableObject::FromVar(protoCtor);

protoCtorObj->SetPrototype(extendsObj);

break;

}

}

}

}

void JavascriptOperators::OP\_LoadUndefinedToElement(Var instance, PropertyId propertyId)

{

AssertMsg(!TaggedNumber::Is(instance), "Invalid scope/root object");

JavascriptOperators::EnsureProperty(instance, propertyId);

}

void JavascriptOperators::OP\_LoadUndefinedToElementScoped(FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext)

{

int i;

int length = pScope->GetLength();

Var argInstance;

for (i = 0; i < length; i++)

{

argInstance = pScope->GetItem(i);

if (JavascriptOperators::EnsureProperty(argInstance, propertyId))

{

return;

}

}

if (!JavascriptOperators::HasOwnPropertyNoHostObject(defaultInstance, propertyId))

{

// CONSIDER : Consider adding pre-initialization support to activation objects.

JavascriptOperators::OP\_InitPropertyScoped(pScope, propertyId, scriptContext->GetLibrary()->GetUndefined(), defaultInstance, scriptContext);

}

}

void JavascriptOperators::OP\_LoadUndefinedToElementDynamic(Var instance, PropertyId propertyId, ScriptContext \*scriptContext)

{

if (!JavascriptOperators::HasOwnPropertyNoHostObject(instance, propertyId))

{

RecyclableObject::FromVar(instance)->InitPropertyScoped(propertyId, scriptContext->GetLibrary()->GetUndefined());

}

}

BOOL JavascriptOperators::EnsureProperty(Var instance, PropertyId propertyId)

{

RecyclableObject \*obj = RecyclableObject::FromVar(instance);

return (obj && obj->EnsureProperty(propertyId));

}

void JavascriptOperators::OP\_EnsureNoRootProperty(Var instance, PropertyId propertyId)

{

Assert(RootObjectBase::Is(instance));

RootObjectBase \*obj = RootObjectBase::FromVar(instance);

obj->EnsureNoProperty(propertyId);

}

void JavascriptOperators::OP\_EnsureNoRootRedeclProperty(Var instance, PropertyId propertyId)

{

Assert(RootObjectBase::Is(instance));

RecyclableObject \*obj = RecyclableObject::FromVar(instance);

obj->EnsureNoRedeclProperty(propertyId);

}

void JavascriptOperators::OP\_ScopedEnsureNoRedeclProperty(FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance)

{

int i;

int length = pDisplay->GetLength();

RecyclableObject \*object;

for (i = 0; i < length; i++)

{

object = RecyclableObject::FromVar(pDisplay->GetItem(i));

if (object->EnsureNoRedeclProperty(propertyId))

{

return;

}

}

object = RecyclableObject::FromVar(defaultInstance);

object->EnsureNoRedeclProperty(propertyId);

}

Var JavascriptOperators::IsIn(Var argProperty, Var instance, ScriptContext\* scriptContext)

{

// Note that the fact that we haven't seen a given name before doesn't mean that the instance doesn't

if (!IsObject(instance))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Operand\_Invalid\_NeedObject, L"in");

}

PropertyRecord const \* propertyRecord;

uint32 index;

IndexType indexType = GetIndexType(argProperty, scriptContext, &index, &propertyRecord, true);

RecyclableObject\* object = RecyclableObject::FromVar(instance);

BOOL result;

if( indexType == Js::IndexType\_Number )

{

result = JavascriptOperators::HasItem( object, index );

}

else

{

PropertyId propertyId = propertyRecord->GetPropertyId();

result = JavascriptOperators::HasProperty( object, propertyId );

#ifdef TELEMETRY\_JSO

{

Assert(indexType != Js::IndexType\_JavascriptString);

if( indexType == Js::IndexType\_PropertyId )

{

scriptContext->GetTelemetry().GetOpcodeTelemetry().IsIn( instance, propertyId, result != 0 );

}

}

#endif

}

return JavascriptBoolean::ToVar(result, scriptContext);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

return PatchGetValueWithThisPtr<IsFromFullJit, TInlineCache>(functionBody, inlineCache, inlineCacheIndex, instance, propertyId, instance);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_forceinline Var JavascriptOperators::PatchGetValueWithThisPtr(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

thisInstance, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetValue", propertyId, scriptContext, object);

}

#endif

return JavascriptOperators::GetProperty(thisInstance, object, propertyId, scriptContext, &info);

}

template Var JavascriptOperators::PatchGetValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValueWithThisPtr<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template Var JavascriptOperators::PatchGetValueWithThisPtr<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template Var JavascriptOperators::PatchGetValueWithThisPtr<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template Var JavascriptOperators::PatchGetValueWithThisPtr<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template <bool IsFromFullJit, class TInlineCache>

Var JavascriptOperators::PatchGetValueForTypeOf(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

Assert(Js::JavascriptStackWalker::ValidateTopJitFrame(scriptContext));

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

instance, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetValueForTypeOf", propertyId, scriptContext, object);

}

#endif

Var prop = nullptr;

BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext);

prop = JavascriptOperators::GetProperty(instance, object, propertyId, scriptContext, &info);

END\_TYPEOF\_ERROR\_HANDLER(scriptContext, prop);

return prop;

}

template Var JavascriptOperators::PatchGetValueForTypeOf<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValueForTypeOf<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValueForTypeOf<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetValueForTypeOf<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

Var JavascriptOperators::PatchGetValueUsingSpecifiedInlineCache(InlineCache \* inlineCache, Var instance, RecyclableObject \* object, PropertyId propertyId, ScriptContext\* scriptContext)

{

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, inlineCache);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, true, false, true, !InlineCache::IsPolymorphic, InlineCache::IsPolymorphic, false>(

instance, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetValue", propertyId, scriptContext, object);

}

#endif

return JavascriptOperators::GetProperty(instance, object, propertyId, scriptContext, &info);

}

Var JavascriptOperators::PatchGetValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

return PatchGetValueWithThisPtrNoFastPath(functionBody, inlineCache, inlineCacheIndex, instance, propertyId, instance);

}

Var JavascriptOperators::PatchGetValueWithThisPtrNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

return JavascriptOperators::GetProperty(thisInstance, object, propertyId, scriptContext, &info);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetRootValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId)

{

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetRootValue", propertyId, scriptContext, object);

}

#endif

return JavascriptOperators::OP\_GetRootProperty(object, propertyId, &info, scriptContext);

}

template Var JavascriptOperators::PatchGetRootValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache>

Var JavascriptOperators::PatchGetRootValueForTypeOf(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId)

{

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value = nullptr;

if (CacheOperators::TryGetProperty<true, true, true, false, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetRootValueForTypeOf", propertyId, scriptContext, object);

}

#endif

value = nullptr;

BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext);

if (JavascriptOperators::GetRootProperty(RecyclableObject::FromVar(object), propertyId, &value, scriptContext, &info))

{

if (scriptContext->IsUndeclBlockVar(value))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

return value;

}

END\_TYPEOF\_ERROR\_HANDLER(scriptContext, value);

value = scriptContext->GetLibrary()->GetUndefined();

return value;

}

template Var JavascriptOperators::PatchGetRootValueForTypeOf<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValueForTypeOf<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValueForTypeOf<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootValueForTypeOf<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject \* object, PropertyId propertyId);

Var JavascriptOperators::PatchGetRootValueNoFastPath\_Var(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

return

PatchGetRootValueNoFastPath(

functionBody,

inlineCache,

inlineCacheIndex,

DynamicObject::FromVar(instance),

propertyId);

}

Var JavascriptOperators::PatchGetRootValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId)

{

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

return JavascriptOperators::OP\_GetRootProperty(object, propertyId, &info, scriptContext);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetPropertyScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance)

{

// Get the property, using a scope stack rather than an individual instance.

// Walk the stack until we find an instance that has the property.

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

uint16 length = pDisplay->GetLength();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

for (uint16 i = 0; i < length; i++)

{

DynamicObject\* object = (DynamicObject\*)pDisplay->GetItem(i);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetPropertyScoped", propertyId, scriptContext, object);

}

#endif

if (JavascriptOperators::GetProperty(object, propertyId, &value, scriptContext, &info))

{

if (scriptContext->IsUndeclBlockVar(value) && propertyId != PropertyIds::\_lexicalThisSlotSymbol)

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

return value;

}

}

// No one in the scope stack has the property, so get it from the default instance provided by the caller.

Var value = JavascriptOperators::PatchGetRootValue<IsFromFullJit>(functionBody, inlineCache, inlineCacheIndex, DynamicObject::FromVar(defaultInstance), propertyId);

if (scriptContext->IsUndeclBlockVar(value))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

return value;

}

template Var JavascriptOperators::PatchGetPropertyScoped<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyScoped<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyScoped<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyScoped<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template <bool IsFromFullJit, class TInlineCache>

Var JavascriptOperators::PatchGetPropertyForTypeOfScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance)

{

Var value = nullptr;

ScriptContext \*scriptContext = functionBody->GetScriptContext();

BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext);

value = JavascriptOperators::PatchGetPropertyScoped<IsFromFullJit, TInlineCache>(functionBody, inlineCache, inlineCacheIndex, pDisplay, propertyId, defaultInstance);

END\_TYPEOF\_ERROR\_HANDLER(scriptContext, value)

return value;

}

template Var JavascriptOperators::PatchGetPropertyForTypeOfScoped<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyForTypeOfScoped<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyForTypeOfScoped<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template Var JavascriptOperators::PatchGetPropertyForTypeOfScoped<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pDisplay, PropertyId propertyId, Var defaultInstance);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

Assert(inlineCache != nullptr);

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = nullptr;

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

#ifdef TELEMETRY\_JSO

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(object, propertyId, value, /\*successful:\*/false);

}

#endif

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

instance, false, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetMethod", propertyId, scriptContext, object);

}

#endif

value = Js::JavascriptOperators::PatchGetMethodFromObject(instance, object, propertyId, &info, scriptContext, false);

#ifdef TELEMETRY\_JSO

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(object, propertyId, value, /\*successful:\*/true);

}

#endif

return value;

}

template Var JavascriptOperators::PatchGetMethod<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetMethod<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetMethod<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchGetMethod<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchGetRootMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId)

{

Assert(inlineCache != nullptr);

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetRootMethod", propertyId, scriptContext, object);

}

#endif

value = Js::JavascriptOperators::PatchGetMethodFromObject(object, object, propertyId, &info, scriptContext, true);

#ifdef TELEMETRY\_JSO

if (TELEMETRY\_PROPERTY\_OPCODE\_FILTER(propertyId))

{

// `successful` will be true as PatchGetMethod throws an exception if not found.

scriptContext->GetTelemetry().GetOpcodeTelemetry().GetMethodProperty(object, propertyId, value, /\*successful:\*/ true);

}

#endif

return value;

}

template Var JavascriptOperators::PatchGetRootMethod<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootMethod<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootMethod<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template Var JavascriptOperators::PatchGetRootMethod<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline Var JavascriptOperators::PatchScopedGetMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

Assert(inlineCache != nullptr);

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

const bool isRoot = RootObjectBase::Is(object);

Var value;

if (CacheOperators::TryGetProperty<true, true, true, false, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

instance, isRoot, object, propertyId, &value, scriptContext, nullptr, &info))

{

return value;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchGetMethod", propertyId, scriptContext, object);

}

#endif

return Js::JavascriptOperators::PatchGetMethodFromObject(instance, object, propertyId, &info, scriptContext, isRoot);

}

template Var JavascriptOperators::PatchScopedGetMethod<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchScopedGetMethod<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchScopedGetMethod<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template Var JavascriptOperators::PatchScopedGetMethod<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

Var JavascriptOperators::PatchGetMethodNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptOperators::GetPropertyObject(instance, scriptContext, &object))

{

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotGet\_NullOrUndefined,

scriptContext->GetPropertyName(propertyId)->GetBuffer());

}

else

{

return scriptContext->GetLibrary()->GetUndefined();

}

}

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

return Js::JavascriptOperators::PatchGetMethodFromObject(instance, object, propertyId, &info, scriptContext, false);

}

Var JavascriptOperators::PatchGetRootMethodNoFastPath\_Var(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId)

{

return

PatchGetRootMethodNoFastPath(

functionBody,

inlineCache,

inlineCacheIndex,

DynamicObject::FromVar(instance),

propertyId);

}

Var JavascriptOperators::PatchGetRootMethodNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId)

{

AssertMsg(RootObjectBase::Is(object), "Root must be a global object!");

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

return Js::JavascriptOperators::PatchGetMethodFromObject(object, object, propertyId, &info, functionBody->GetScriptContext(), true);

}

Var JavascriptOperators::PatchGetMethodFromObject(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, PropertyValueInfo \* info, ScriptContext\* scriptContext, bool isRootLd)

{

Assert(IsPropertyObject(propertyObject));

Var value = nullptr;

BOOL foundValue = FALSE;

if (isRootLd)

{

RootObjectBase\* rootObject = RootObjectBase::FromVar(instance);

foundValue = JavascriptOperators::GetRootPropertyReference(rootObject, propertyId, &value, scriptContext, info);

}

else

{

foundValue = JavascriptOperators::GetPropertyReference(instance, propertyObject, propertyId, &value, scriptContext, info);

}

if (!foundValue)

{

// Don't error if we disabled implicit calls

if (scriptContext->GetThreadContext()->RecordImplicitException())

{

const wchar\_t\* propertyName = scriptContext->GetPropertyName(propertyId)->GetBuffer();

value = scriptContext->GetLibrary()->GetUndefined();

JavascriptFunction \* caller = NULL;

if (JavascriptStackWalker::GetCaller(&caller, scriptContext))

{

FunctionBody \* callerBody = caller->GetFunctionBody();

if (callerBody && callerBody->GetUtf8SourceInfo()->GetIsXDomain())

{

propertyName = NULL;

}

}

// Prior to version 12 we had mistakenly immediately thrown an error for property reference method calls

// (i.e. <expr>.foo() form) when the target object is the global object. The spec says that a GetValue

// on a reference should throw if the reference is unresolved, of which a property reference can never be,

// however it can be unresolved in the case of an identifier expression, e.g. foo() with no qualification.

// Such a case would come down to the global object if foo was undefined, hence the check for root object,

// except that it should have been a check for isRootLd to be correct.

//

// // (at global scope)

// foo(x());

//

// should throw an error before evaluating x() if foo is not defined, but

//

// // (at global scope)

// this.foo(x());

//

// should evaluate x() before throwing an error if foo is not a property on the global object.

// Maintain old behavior prior to version 12.

bool isPropertyReference = !isRootLd;

if (!isPropertyReference)

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UndefVariable, propertyName);

}

else

{

// ES5 11.2.3 #2: We evaluate the call target but don't throw yet if target member is missing. We need to evaluate argList

// first (#3). Postpone throwing error to invoke time.

value = ThrowErrorObject::CreateThrowTypeErrorObject(scriptContext, VBSERR\_OLENoPropOrMethod, propertyName);

}

}

}

return value;

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

return PatchPutValueWithThisPtr<IsFromFullJit, TInlineCache>(functionBody, inlineCache, inlineCacheIndex, instance, propertyId, newValue, instance, flags);

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutValueWithThisPtr(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

if (TaggedNumber::Is(instance))

{

JavascriptOperators::SetPropertyOnTaggedNumber(instance, nullptr, propertyId, newValue, scriptContext, flags);

return;

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

RecyclableObject\* object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<true, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutValue", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::OP\_SetProperty(object, propertyId, newValue, scriptContext, &info, flags, thisInstance))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutRootValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<true, true, true, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutRootValue", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::SetRootProperty(object, propertyId, newValue, &info, scriptContext, flags))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutRootValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutValueNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

if (TaggedNumber::Is(instance))

{

JavascriptOperators::SetPropertyOnTaggedNumber(instance,

nullptr,

propertyId,

newValue,

scriptContext,

flags);

return;

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

RecyclableObject \*object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<!TInlineCache::IsPolymorphic, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutValueNoLocalFastPath", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::OP\_SetProperty(instance, propertyId, newValue, scriptContext, &info, flags))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutValueNoLocalFastPath<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueNoLocalFastPath<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueNoLocalFastPath<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueNoLocalFastPath<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

if (TaggedNumber::Is(instance))

{

JavascriptOperators::SetPropertyOnTaggedNumber(instance,

nullptr,

propertyId,

newValue,

scriptContext,

flags);

return;

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

RecyclableObject \*object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<!TInlineCache::IsPolymorphic, true, true, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutValueNoLocalFastPath", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::OP\_SetProperty(instance, propertyId, newValue, scriptContext, &info, flags, thisInstance))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutValueWithThisPtrNoLocalFastPath<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags);

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchPutRootValueNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject \*object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<!TInlineCache::IsPolymorphic, true, true, true, false, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, true, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchPutRootValueNoLocalFastPath", propertyId, scriptContext, object);

}

#endif

ImplicitCallFlags prevImplicitCallFlags = ImplicitCall\_None;

ImplicitCallFlags currImplicitCallFlags = ImplicitCall\_None;

bool hasThisOnlyStatements = functionBody->GetHasOnlyThisStmts();

if (hasThisOnlyStatements)

{

prevImplicitCallFlags = CacheAndClearImplicitBit(scriptContext);

}

if (!JavascriptOperators::SetRootProperty(object, propertyId, newValue, &info, scriptContext, flags))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

if (hasThisOnlyStatements)

{

currImplicitCallFlags = CheckAndUpdateFunctionBodyWithImplicitFlag(functionBody);

RestoreImplicitFlag(scriptContext, prevImplicitCallFlags, currImplicitCallFlags);

}

}

template void JavascriptOperators::PatchPutRootValueNoLocalFastPath<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValueNoLocalFastPath<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValueNoLocalFastPath<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

template void JavascriptOperators::PatchPutRootValueNoLocalFastPath<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags);

void JavascriptOperators::PatchPutValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

PatchPutValueWithThisPtrNoFastPath(functionBody, inlineCache, inlineCacheIndex, instance, propertyId, newValue, instance, flags);

}

void JavascriptOperators::PatchPutValueWithThisPtrNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

if (TaggedNumber::Is(instance))

{

JavascriptOperators::SetPropertyOnTaggedNumber(instance, nullptr, propertyId, newValue, scriptContext, flags);

return;

}

RecyclableObject\* object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

if (!JavascriptOperators::OP\_SetProperty(object, propertyId, newValue, scriptContext, &info, flags, thisInstance))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

}

void JavascriptOperators::PatchPutRootValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

RecyclableObject\* object = RecyclableObject::FromVar(instance);

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

if (!JavascriptOperators::SetRootProperty(object, propertyId, newValue, &info, scriptContext, flags))

{

// Add implicit call flags, to bail out if field copy prop may propagate the wrong value.

scriptContext->GetThreadContext()->AddImplicitCallFlags(ImplicitCall\_NoOpSet);

}

}

template <bool IsFromFullJit, class TInlineCache>

\_\_inline void JavascriptOperators::PatchInitValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue)

{

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

const PropertyOperationFlags flags = newValue == NULL ? PropertyOperation\_SpecialValue : PropertyOperation\_None;

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, !IsFromFullJit);

if (CacheOperators::TrySetProperty<true, true, false, true, true, !TInlineCache::IsPolymorphic, TInlineCache::IsPolymorphic, false>(

object, false, propertyId, newValue, scriptContext, flags, nullptr, &info))

{

return;

}

#if DBG\_DUMP

if (PHASE\_VERBOSE\_TRACE1(Js::InlineCachePhase))

{

CacheOperators::TraceCache(inlineCache, L"PatchInitValue", propertyId, scriptContext, object);

}

#endif

Type \*typeWithoutProperty = object->GetType();

// Ideally the lowerer would emit a call to the right flavor of PatchInitValue, so that we can ensure that we only

// ever initialize to NULL in the right cases. But the backend uses the StFld opcode for initialization, and it

// would be cumbersome to thread the different helper calls all the way down

if (object->InitProperty(propertyId, newValue, flags, &info))

{

CacheOperators::CachePropertyWrite(object, false, typeWithoutProperty, propertyId, &info, scriptContext);

}

}

template void JavascriptOperators::PatchInitValue<false, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

template void JavascriptOperators::PatchInitValue<true, InlineCache>(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

template void JavascriptOperators::PatchInitValue<false, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

template void JavascriptOperators::PatchInitValue<true, PolymorphicInlineCache>(FunctionBody \*const functionBody, PolymorphicInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

void JavascriptOperators::PatchInitValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue)

{

PropertyValueInfo info;

PropertyValueInfo::SetCacheInfo(&info, functionBody, inlineCache, inlineCacheIndex, true);

Type \*typeWithoutProperty = object->GetType();

if (object->InitProperty(propertyId, newValue, PropertyOperation\_None, &info))

{

CacheOperators::CachePropertyWrite(object, false, typeWithoutProperty, propertyId, &info, functionBody->GetScriptContext());

}

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void JavascriptOperators::TracePropertyEquivalenceCheck(const JitEquivalentTypeGuard\* guard, const Type\* type, const Type\* refType, bool isEquivalent, uint failedPropertyIndex)

{

if (PHASE\_TRACE1(Js::EquivObjTypeSpecPhase))

{

uint propertyCount = guard->GetCache()->record.propertyCount;

Output::Print(L"EquivObjTypeSpec: checking %u properties on operation %u, (type = 0x%p, ref type = 0x%p):\n",

propertyCount, guard->GetObjTypeSpecFldId(), type, refType);

const Js::TypeEquivalenceRecord& record = guard->GetCache()->record;

ScriptContext\* scriptContext = type->GetScriptContext();

if (isEquivalent)

{

if (Js::Configuration::Global.flags.Verbose)

{

Output::Print(L" <start>, ");

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &record.properties[pi];

const PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(refInfo->propertyId);

Output::Print(L"%s(#%d)@%ua%dw%d, ", propertyRecord->GetBuffer(), propertyRecord->GetPropertyId(), refInfo->slotIndex, refInfo->isAuxSlot, refInfo->mustBeWritable);

}

Output::Print(L"<end>\n");

}

}

else

{

const EquivalentPropertyEntry\* refInfo = &record.properties[failedPropertyIndex];

Js::PropertyEquivalenceInfo info(Constants::NoSlot, false, false);

const PropertyRecord\* propertyRecord = scriptContext->GetPropertyName(refInfo->propertyId);

if (DynamicType::Is(type->GetTypeId()))

{

Js::DynamicTypeHandler\* typeHandler = (static\_cast<const DynamicType\*>(type))->GetTypeHandler();

typeHandler->GetPropertyEquivalenceInfo(propertyRecord, info);

}

Output::Print(L"EquivObjTypeSpec: check failed for %s (#%d) on operation %u:\n",

propertyRecord->GetBuffer(), propertyRecord->GetPropertyId(), guard->GetObjTypeSpecFldId());

Output::Print(L" type = 0x%p, ref type = 0x%p, slot = 0x%u (%d), ref slot = 0x%u (%d), is writable = %d, required writable = %d\n",

type, refType, info.slotIndex, refInfo->slotIndex, info.isAuxSlot, refInfo->isAuxSlot, info.isWritable, refInfo->mustBeWritable);

}

Output::Flush();

}

}

#endif

bool JavascriptOperators::IsStaticTypeObjTypeSpecEquivalent(const TypeEquivalenceRecord& equivalenceRecord, uint& failedIndex)

{

uint propertyCount = equivalenceRecord.propertyCount;

Js::EquivalentPropertyEntry\* properties = equivalenceRecord.properties;

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &properties[pi];

if (!IsStaticTypeObjTypeSpecEquivalent(refInfo))

{

failedIndex = pi;

return false;

}

}

return true;

}

bool JavascriptOperators::IsStaticTypeObjTypeSpecEquivalent(const EquivalentPropertyEntry \*entry)

{

// Objects of static types have no local properties, but they may load fields from their prototypes.

return entry->slotIndex == Constants::NoSlot && !entry->mustBeWritable;

}

bool JavascriptOperators::CheckIfTypeIsEquivalent(Type\* type, JitEquivalentTypeGuard\* guard)

{

if (guard->GetValue() == 0)

{

return false;

}

if (guard->GetType()->GetScriptContext() != type->GetScriptContext())

{

// Can't cache cross-context objects

return false;

}

// CONSIDER : Add stats on how often the cache hits, and simply force bailout if

// the efficacy is too low.

EquivalentTypeCache\* cache = guard->GetCache();

// CONSIDER : Consider emitting o.type == equivTypes[hash(o.type)] in machine code before calling

// this helper, particularly if we want to handle polymorphism with frequently changing types.

Assert(EQUIVALENT\_TYPE\_CACHE\_SIZE == 8);

Type\*\* equivTypes = cache->types;

if (type == equivTypes[0] || type == equivTypes[1] || type == equivTypes[2] || type == equivTypes[3] ||

type == equivTypes[4] || type == equivTypes[5] || type == equivTypes[6] || type == equivTypes[7])

{

guard->SetType(type);

return true;

}

// If we didn't find the type in the cache, let's check if it's equivalent the slow way, by comparing

// each of its relevant property slots to its equivalent in one of the cached types.

// We are making a few assumption that simplify the process:

// 1. If two types have the same prototype, any properties loaded from a prototype must come from the same slot.

// If any of the prototypes in the chain was altered such that this is no longer true, the corresponding

// property guard would have been invalidated and we would bail out at the guard check (either on this

// type check or downstream, but before the property load is attempted).

// 2. For polymorphic field loads fixed fields are only supported on prototypes. Hence, if two types have the

// same prototype, any of the equivalent fixed properties will match. If any has been overwritten, the

// corresponding guard would have been invalidated and we would bail out (as above).

Type\* refType = equivTypes[0];

if (refType == nullptr)

{

return false;

}

if (cache->IsLoadedFromProto() && type->GetPrototype() != refType->GetPrototype())

{

if (PHASE\_TRACE1(Js::EquivObjTypeSpecPhase))

{

Output::Print(L"EquivObjTypeSpec: failed check on operation %u (type = 0x%x, ref type = 0x%x, proto = 0x%x, ref proto = 0x%x) \n",

guard->GetObjTypeSpecFldId(), type, refType, type->GetPrototype(), refType->GetPrototype());

Output::Flush();

}

return false;

}

if (type->GetTypeId() != refType->GetTypeId())

{

if (PHASE\_TRACE1(Js::EquivObjTypeSpecPhase))

{

Output::Print(L"EquivObjTypeSpec: failed check on operation %u (type = 0x%x, ref type = 0x%x, proto = 0x%x, ref proto = 0x%x) \n",

guard->GetObjTypeSpecFldId(), type, refType, type->GetPrototype(), refType->GetPrototype());

Output::Flush();

}

return false;

}

// Review : This is quite slow. We could make it somewhat faster, by keeping slot indexes instead

// of property IDs, but that would mean we would need to look up property IDs from slot indexes when installing

// property guards, or maintain a whole separate list of equivalent slot indexes.

Assert(cache->record.propertyCount > 0);

// CONSIDER (EquivObjTypeSpec): Impose a limit on the number of properties guarded by an equivalent type check.

// The trick is where in the glob opt to make the cut off. Perhaps in the forward pass we could track the number of

// field operations protected by a type check (keep a counter on the type's value info), and if that counter exceeds

// some threshold, simply stop optimizing any further instructions.

bool isEquivalent;

uint failedPropertyIndex;

if (DynamicType::Is(type->GetTypeId()))

{

Js::DynamicTypeHandler\* typeHandler = (static\_cast<DynamicType\*>(type))->GetTypeHandler();

isEquivalent = typeHandler->IsObjTypeSpecEquivalent(type, cache->record, failedPropertyIndex);

}

else

{

Assert(StaticType::Is(type->GetTypeId()));

isEquivalent = IsStaticTypeObjTypeSpecEquivalent(cache->record, failedPropertyIndex);

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

TracePropertyEquivalenceCheck(guard, type, refType, isEquivalent, failedPropertyIndex);

#endif

if (!isEquivalent)

{

return false;

}

// CONSIDER (EquivObjTypeSpec): Invent some form of least recently used eviction scheme.

uintptr\_t index = (reinterpret\_cast<uintptr\_t>(type) >> 4) & (EQUIVALENT\_TYPE\_CACHE\_SIZE - 1);

if (cache->nextEvictionVictim == EQUIVALENT\_TYPE\_CACHE\_SIZE)

{

\_\_analysis\_assume(index < EQUIVALENT\_TYPE\_CACHE\_SIZE);

if (equivTypes[index] != nullptr)

{

uintptr\_t initialIndex = index;

index = (initialIndex + 1) & (EQUIVALENT\_TYPE\_CACHE\_SIZE - 1);

for (; index != initialIndex; index = (index + 1) & (EQUIVALENT\_TYPE\_CACHE\_SIZE - 1))

{

if (equivTypes[index] == nullptr) break;

}

}

\_\_analysis\_assume(index < EQUIVALENT\_TYPE\_CACHE\_SIZE);

if (equivTypes[index] != nullptr)

{

cache->nextEvictionVictim = 0;

}

}

else

{

Assert(cache->nextEvictionVictim < EQUIVALENT\_TYPE\_CACHE\_SIZE);

\_\_analysis\_assume(cache->nextEvictionVictim < EQUIVALENT\_TYPE\_CACHE\_SIZE);

equivTypes[cache->nextEvictionVictim] = equivTypes[index];

cache->nextEvictionVictim = (cache->nextEvictionVictim + 1) & (EQUIVALENT\_TYPE\_CACHE\_SIZE - 1);

}

Assert(index < EQUIVALENT\_TYPE\_CACHE\_SIZE);

\_\_analysis\_assume(index < EQUIVALENT\_TYPE\_CACHE\_SIZE);

equivTypes[index] = type;

if (cache->HasFixedValue())

{

// Fixed field checks allow us to assume a specific type ID, but the assumption is only

// valid if we lock the type. Otherwise, the type ID may change out from under us without

// evolving the type.

if (DynamicType::Is(type->GetTypeId()))

{

DynamicType \*dynamicType = static\_cast<DynamicType\*>(type);

if (!dynamicType->GetIsLocked())

{

dynamicType->LockType();

}

}

}

guard->SetType(type);

return true;

}

void JavascriptOperators::GetPropertyIdForInt(uint64 value, ScriptContext\* scriptContext, PropertyRecord const \*\* propertyRecord)

{

wchar\_t buffer[20];

::\_ui64tow\_s(value, buffer, sizeof(buffer)/sizeof(wchar\_t), 10);

scriptContext->GetOrAddPropertyRecord(buffer, JavascriptString::GetBufferLength(buffer), propertyRecord);

}

void JavascriptOperators::GetPropertyIdForInt(uint32 value, ScriptContext\* scriptContext, PropertyRecord const \*\* propertyRecord)

{

GetPropertyIdForInt(static\_cast<uint64>(value), scriptContext, propertyRecord);

}

Var JavascriptOperators::FromPropertyDescriptor(PropertyDescriptor descriptor, ScriptContext\* scriptContext)

{

DynamicObject\* object = scriptContext->GetLibrary()->CreateObject();

// ES5 Section 8.10.4 specifies the order for adding these properties.

if (descriptor.IsDataDescriptor())

{

if (descriptor.ValueSpecified())

{

JavascriptOperators::InitProperty(object, PropertyIds::value, descriptor.GetValue());

}

JavascriptOperators::InitProperty(object, PropertyIds::writable, JavascriptBoolean::ToVar(descriptor.IsWritable(),scriptContext));

}

else if (descriptor.IsAccessorDescriptor())

{

JavascriptOperators::InitProperty(object, PropertyIds::get, JavascriptOperators::CanonicalizeAccessor(descriptor.GetGetter(), scriptContext));

JavascriptOperators::InitProperty(object, PropertyIds::set, JavascriptOperators::CanonicalizeAccessor(descriptor.GetSetter(), scriptContext));

}

if (descriptor.EnumerableSpecified())

{

JavascriptOperators::InitProperty(object, PropertyIds::enumerable, JavascriptBoolean::ToVar(descriptor.IsEnumerable(), scriptContext));

}

if (descriptor.ConfigurableSpecified())

{

JavascriptOperators::InitProperty(object, PropertyIds::configurable, JavascriptBoolean::ToVar(descriptor.IsConfigurable(), scriptContext));

}

return object;

}

// ES5 8.12.9 [[DefineOwnProperty]].

// Return value:

// - TRUE = success.

// - FALSE (can throw depending on throwOnError parameter) = unsuccessful.

BOOL JavascriptOperators::DefineOwnPropertyDescriptor(RecyclableObject\* obj, PropertyId propId, const PropertyDescriptor& descriptor, bool throwOnError, ScriptContext\* scriptContext)

{

Assert(obj);

Assert(scriptContext);

if (JavascriptProxy::Is(obj))

{

return JavascriptProxy::DefineOwnPropertyDescriptor(obj, propId, descriptor, throwOnError, scriptContext);

}

PropertyDescriptor currentDescriptor;

BOOL isCurrentDescriptorDefined = JavascriptOperators::GetOwnPropertyDescriptor(obj, propId, scriptContext, &currentDescriptor);

bool isExtensible = !!obj->IsExtensible();

return ValidateAndApplyPropertyDescriptor<true>(obj, propId, descriptor, isCurrentDescriptorDefined ? &currentDescriptor : nullptr, isExtensible, throwOnError, scriptContext);

}

BOOL JavascriptOperators::IsCompatiblePropertyDescriptor(const PropertyDescriptor& descriptor, PropertyDescriptor\* currentDescriptor, bool isExtensible, bool throwOnError, ScriptContext\* scriptContext)

{

return ValidateAndApplyPropertyDescriptor<false>(nullptr, Constants::NoProperty, descriptor, currentDescriptor, isExtensible, throwOnError, scriptContext);

}

template<bool needToSetProperty>

BOOL JavascriptOperators::ValidateAndApplyPropertyDescriptor(RecyclableObject\* obj, PropertyId propId, const PropertyDescriptor& descriptor,

PropertyDescriptor\* currentDescriptor, bool isExtensible, bool throwOnError, ScriptContext\* scriptContext)

{

Var defaultDataValue = scriptContext->GetLibrary()->GetUndefined();

Var defaultAccessorValue = scriptContext->GetLibrary()->GetDefaultAccessorFunction();

if (currentDescriptor == nullptr)

{

if (!isExtensible) // ES5 8.12.9.3.

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotExtensible, propId);

}

else // ES5 8.12.9.4.

{

if (needToSetProperty)

{

if (descriptor.IsGenericDescriptor() || descriptor.IsDataDescriptor())

{

// ES5 8.12.9.4a: Create an own data property named P of object O whose [[Value]], [[Writable]],

// [[Enumerable]] and [[Configurable]] attribute values are described by Desc.

// If the value of an attribute field of Desc is absent, the attribute of the newly created property

// is set to its default value.

PropertyDescriptor filledDescriptor = FillMissingPropertyDescriptorFields<false>(descriptor, scriptContext);

BOOL tempResult = obj->SetPropertyWithAttributes(propId, filledDescriptor.GetValue(), filledDescriptor.GetAttributes(), nullptr);

Assert(tempResult || obj->IsExternal());

}

else

{

// ES5 8.12.9.4b: Create an own accessor property named P of object O whose [[Get]], [[Set]], [[Enumerable]]

// and [[Configurable]] attribute values are described by Desc. If the value of an attribute field of Desc is absent,

// the attribute of the newly created property is set to its default value.

Assert(descriptor.IsAccessorDescriptor());

PropertyDescriptor filledDescriptor = FillMissingPropertyDescriptorFields<true>(descriptor, scriptContext);

BOOL isSetAccessorsSuccess = obj->SetAccessors(propId, filledDescriptor.GetGetter(), filledDescriptor.GetSetter());

// It is valid for some objects to not-support getters and setters, specifically, for projection of an ABI method

// (CustomExternalObject => MapWithStringKey) which SetAccessors returns VBSErr\_ActionNotSupported.

// But for non-external objects SetAccessors should succeed.

Assert(isSetAccessorsSuccess || obj->CanHaveInterceptors());

// If SetAccessors failed, the property wasn't created, so no need to change the attributes.

if (isSetAccessorsSuccess)

{

JavascriptOperators::SetAttributes(obj, propId, filledDescriptor, true); // use 'force' as default attributes in type system are different from ES5.

}

}

}

return TRUE;

}

}

// ES5 8.12.9.5: Return true, if every field in Desc is absent.

if (!descriptor.ConfigurableSpecified() && !descriptor.EnumerableSpecified() && !descriptor.WritableSpecified() &&

!descriptor.ValueSpecified() && !descriptor.GetterSpecified() && !descriptor.SetterSpecified())

{

return TRUE;

}

// ES5 8.12.9.6: Return true, if every field in Desc also occurs in current and the value of every field in Desc is the same value

// as the corresponding field in current when compared using the SameValue algorithm (9.12).

PropertyDescriptor filledDescriptor = descriptor.IsAccessorDescriptor() ? FillMissingPropertyDescriptorFields<true>(descriptor, scriptContext)

: FillMissingPropertyDescriptorFields<false>(descriptor, scriptContext);

if (JavascriptOperators::AreSamePropertyDescriptors(&filledDescriptor, currentDescriptor, scriptContext))

{

return TRUE;

}

if (!currentDescriptor->IsConfigurable()) // ES5 8.12.9.7.

{

if (descriptor.ConfigurableSpecified() && descriptor.IsConfigurable())

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotConfigurable, propId);

}

if (descriptor.EnumerableSpecified() && descriptor.IsEnumerable() != currentDescriptor->IsEnumerable())

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotConfigurable, propId);

}

}

// Whether to merge attributes from tempDescriptor into descriptor to keep original values

// of some attributes from the object/use tempDescriptor for SetAttributes, or just use descriptor.

// This is optimization to avoid 2 calls to SetAttributes.

bool mergeDescriptors = false;

// Whether to call SetAttributes with 'force' flag which forces setting all attributes

// rather than only specified or which have true values.

// This is to make sure that the object has correct attributes, as default values in the object are not for ES5.

bool forceSetAttributes = false;

PropertyDescriptor tempDescriptor;

// ES5 8.12.9.8: If IsGenericDescriptor(Desc) is true, then no further validation is required.

if (!descriptor.IsGenericDescriptor())

{

if (currentDescriptor->IsDataDescriptor() != descriptor.IsDataDescriptor())

{

// ES5 8.12.9.9: Else, if IsDataDescriptor(current) and IsDataDescriptor(Desc) have different results...

if (!currentDescriptor->IsConfigurable())

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotConfigurable, propId);

}

if (needToSetProperty)

{

if (currentDescriptor->IsDataDescriptor())

{

// ES5 8.12.9.9.b: Convert the property named P of object O from a data property to an accessor property.

// Preserve the existing values of the converted property's [[Configurable]] and [[Enumerable]] attributes

// and set the rest of the property's attributes to their default values.

PropertyAttributes preserveFromObject = currentDescriptor->GetAttributes() & (PropertyConfigurable | PropertyEnumerable);

BOOL isSetAccessorsSuccess = obj->SetAccessors(propId, defaultAccessorValue, defaultAccessorValue);

// It is valid for some objects to not-support getters and setters, specifically, for projection of an ABI method

// (CustomExternalObject => MapWithStringKey) which SetAccessors returns VBSErr\_ActionNotSupported.

// But for non-external objects SetAccessors should succeed.

Assert(isSetAccessorsSuccess || obj->CanHaveInterceptors());

if (isSetAccessorsSuccess)

{

tempDescriptor.SetAttributes(preserveFromObject, PropertyConfigurable | PropertyEnumerable);

forceSetAttributes = true; // use SetAttrbiutes with 'force' as default attributes in type system are different from ES5.

mergeDescriptors = true;

}

}

else

{

// ES5 8.12.9.9.c: Convert the property named P of object O from an accessor property to a data property.

// Preserve the existing values of the converted property's [[Configurable]] and [[Enumerable]] attributes

// and set the rest of the property's attributes to their default values.

// Note: avoid using SetProperty/SetPropertyWithAttributes here because they has undesired side-effects:

// it calls previous setter and in some cases of attribute values throws.

// To walk around, call DeleteProperty and then AddProperty.

PropertyAttributes preserveFromObject = currentDescriptor->GetAttributes() & (PropertyConfigurable | PropertyEnumerable);

tempDescriptor.SetAttributes(preserveFromObject, PropertyConfigurable | PropertyEnumerable);

tempDescriptor.MergeFrom(descriptor); // Update only fields specified in 'descriptor'.

Var descriptorValue = descriptor.ValueSpecified() ? descriptor.GetValue() : defaultDataValue;

// Note: HostDispath'es implementation of DeleteProperty currently throws E\_NOTIMPL.

obj->DeleteProperty(propId, PropertyOperation\_None);

BOOL tempResult = obj->SetPropertyWithAttributes(propId, descriptorValue, tempDescriptor.GetAttributes(), NULL, PropertyOperation\_Force);

Assert(tempResult);

// At this time we already set value and attributes to desired values,

// thus we can skip step ES5 8.12.9.12 and simply return true.

return TRUE;

}

}

}

else if (currentDescriptor->IsDataDescriptor() && descriptor.IsDataDescriptor())

{

// ES5 8.12.9.10: Else, if IsDataDescriptor(current) and IsDataDescriptor(Desc) are both true...

if (!currentDescriptor->IsConfigurable())

{

if (!currentDescriptor->IsWritable())

{

if ((descriptor.WritableSpecified() && descriptor.IsWritable()) || // ES5 8.12.9.10.a.i

(descriptor.ValueSpecified() &&

!JavascriptConversion::SameValue(descriptor.GetValue(), currentDescriptor->GetValue()))) // ES5 8.12.9.10.a.ii

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotWritable, propId);

}

}

}

// ES5 8.12.9.10.b: else, the [[Configurable]] field of current is true, so any change is acceptable.

}

else

{

// ES5 8.12.9.11: Else, IsAccessorDescriptor(current) and IsAccessorDescriptor(Desc) are both true, so...

Assert(currentDescriptor->IsAccessorDescriptor() && descriptor.IsAccessorDescriptor());

if (!currentDescriptor->IsConfigurable())

{

if ((descriptor.SetterSpecified() &&

!JavascriptConversion::SameValue(

JavascriptOperators::CanonicalizeAccessor(descriptor.GetSetter(), scriptContext),

JavascriptOperators::CanonicalizeAccessor(currentDescriptor->GetSetter(), scriptContext))) ||

(descriptor.GetterSpecified() &&

!JavascriptConversion::SameValue(

JavascriptOperators::CanonicalizeAccessor(descriptor.GetGetter(), scriptContext),

JavascriptOperators::CanonicalizeAccessor(currentDescriptor->GetGetter(), scriptContext))))

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotConfigurable, propId);

}

}

}

// This part is only for non-generic descriptors:

// ES5 8.12.9.12: For each attribute field of Desc that is present,

// set the correspondingly named attribute of the property named P of object O to the value of the field.

if (descriptor.IsDataDescriptor())

{

if (descriptor.ValueSpecified() && needToSetProperty)

{

// Set just the value by passing the current attributes of the property.

// If the property's attributes are also changing (perhaps becoming non-writable),

// this will be taken care of in the call to JavascriptOperators::SetAttributes below.

// Built-in Function.prototype properties 'length', 'arguments', and 'caller' are special cases.

BOOL tempResult = obj->SetPropertyWithAttributes(propId, descriptor.GetValue(), currentDescriptor->GetAttributes(), nullptr);

AssertMsg(tempResult || JavascriptFunction::IsBuiltinProperty(obj, propId), "If you hit this assert, most likely there is something wrong with the object/type.");

}

}

else if (descriptor.IsAccessorDescriptor() && needToSetProperty)

{

Assert(descriptor.GetterSpecified() || descriptor.SetterSpecified());

Var oldGetter = defaultAccessorValue, oldSetter = defaultAccessorValue;

if (!descriptor.GetterSpecified() || !descriptor.SetterSpecified())

{

// Unless both getter and setter are specified, make sure we don't overwrite old accessor.

obj->GetAccessors(propId, &oldGetter, &oldSetter, scriptContext);

}

Var getter = descriptor.GetterSpecified() ? descriptor.GetGetter() : oldGetter;

Var setter = descriptor.SetterSpecified() ? descriptor.GetSetter() : oldSetter;

obj->SetAccessors(propId, getter, setter);

}

} // if (!descriptor.IsGenericDescriptor())

// Continue for all descriptors including generic:

// ES5 8.12.9.12: For each attribute field of Desc that is present,

// set the correspondingly named attribute of the property named P of object O to the value of the field.

if (needToSetProperty)

{

if (mergeDescriptors)

{

tempDescriptor.MergeFrom(descriptor);

JavascriptOperators::SetAttributes(obj, propId, tempDescriptor, forceSetAttributes);

}

else

{

JavascriptOperators::SetAttributes(obj, propId, descriptor, forceSetAttributes);

}

}

return TRUE;

}

template <bool isAccessor>

PropertyDescriptor JavascriptOperators::FillMissingPropertyDescriptorFields(PropertyDescriptor descriptor, ScriptContext\* scriptContext)

{

PropertyDescriptor newDescriptor;

const PropertyDescriptor\* defaultDescriptor = scriptContext->GetLibrary()->GetDefaultPropertyDescriptor();

if (isAccessor)

{

newDescriptor.SetGetter(descriptor.GetterSpecified() ? descriptor.GetGetter() : defaultDescriptor->GetGetter());

newDescriptor.SetSetter(descriptor.SetterSpecified() ? descriptor.GetSetter() : defaultDescriptor->GetSetter());

}

else

{

newDescriptor.SetValue(descriptor.ValueSpecified() ? descriptor.GetValue() : defaultDescriptor->GetValue());

newDescriptor.SetWritable(descriptor.WritableSpecified() ? descriptor.IsWritable() : defaultDescriptor->IsWritable());

}

newDescriptor.SetConfigurable(descriptor.ConfigurableSpecified() ? descriptor.IsConfigurable() : defaultDescriptor->IsConfigurable());

newDescriptor.SetEnumerable(descriptor.EnumerableSpecified() ? descriptor.IsEnumerable() : defaultDescriptor->IsEnumerable());

return newDescriptor;

}

// ES5: 15.4.5.1

BOOL JavascriptOperators::DefineOwnPropertyForArray(JavascriptArray\* arr, PropertyId propId, const PropertyDescriptor& descriptor, bool throwOnError, ScriptContext\* scriptContext)

{

if (propId == PropertyIds::length)

{

if (!descriptor.ValueSpecified())

{

return DefineOwnPropertyDescriptor(arr, PropertyIds::length, descriptor, throwOnError, scriptContext);

}

PropertyDescriptor newLenDesc = descriptor;

uint32 newLen = ES5Array::ToLengthValue(descriptor.GetValue(), scriptContext);

newLenDesc.SetValue(JavascriptNumber::ToVar(newLen, scriptContext));

uint32 oldLen = arr->GetLength();

if (newLen >= oldLen)

{

return DefineOwnPropertyDescriptor(arr, PropertyIds::length, newLenDesc, throwOnError, scriptContext);

}

BOOL oldLenWritable = arr->IsWritable(PropertyIds::length);

if (!oldLenWritable)

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_NotWritable, propId);

}

bool newWritable = (!newLenDesc.WritableSpecified() || newLenDesc.IsWritable());

if (!newWritable)

{

// Need to defer setting writable to false in case any elements cannot be deleted

newLenDesc.SetWritable(true);

}

BOOL succeeded = DefineOwnPropertyDescriptor(arr, PropertyIds::length, newLenDesc, throwOnError, scriptContext);

//

// Our SetProperty(length) is also responsible to trim elements. When succeeded is

//

// false:

// \* length attributes rejected

// \* elements not touched

// true:

// \* length attributes are set successfully

// \* elements trimming may be either completed or incompleted, length value is correct

//

// \* Strict mode TODO: Currently SetProperty(length) does not throw. If that throws, we need

// to update here to set correct newWritable even on exception.

//

if (!succeeded)

{

return false;

}

if (!newWritable) // Now set requested newWritable.

{

PropertyDescriptor newWritableDesc;

newWritableDesc.SetWritable(false);

DefineOwnPropertyDescriptor(arr, PropertyIds::length, newWritableDesc, false, scriptContext);

}

if (arr->GetLength() > newLen) // Delete incompleted

{

// Since SetProperty(length) not throwing, we'll reject here

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_Default, propId);

}

return true;

}

uint32 index;

if (scriptContext->IsNumericPropertyId(propId, &index))

{

if (index >= arr->GetLength() && !arr->IsWritable(PropertyIds::length))

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_LengthNotWritable, propId);

}

BOOL succeeded = DefineOwnPropertyDescriptor(arr, propId, descriptor, false, scriptContext);

if (!succeeded)

{

return Reject(throwOnError, scriptContext, JSERR\_DefineProperty\_Default, propId);

}

// Out SetItem takes care of growing "length". we are done.

return true;

}

return DefineOwnPropertyDescriptor(arr, propId, descriptor, throwOnError, scriptContext);

}

BOOL JavascriptOperators::SetPropertyDescriptor(RecyclableObject\* object, PropertyId propId, PropertyDescriptor descriptor)

{

if (descriptor.ValueSpecified())

{

ScriptContext\* requestContext = object->GetScriptContext(); // Real requestContext?

JavascriptOperators::SetProperty(object, object, propId, descriptor.GetValue(), requestContext);

}

else if (descriptor.GetterSpecified() || descriptor.SetterSpecified())

{

JavascriptOperators::SetAccessors(object, propId, descriptor.GetGetter(), descriptor.GetSetter());

}

if (descriptor.EnumerableSpecified())

{

object->SetEnumerable(propId, descriptor.IsEnumerable());

}

if (descriptor.ConfigurableSpecified())

{

object->SetConfigurable(propId, descriptor.IsConfigurable());

}

if (descriptor.WritableSpecified())

{

object->SetWritable(propId, descriptor.IsWritable());

}

return true;

}

BOOL JavascriptOperators::ToPropertyDescriptorForProxyObjects(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext)

{

if (!JavascriptOperators::IsObject(propertySpec))

{

return FALSE;

}

Var value;

RecyclableObject\* propertySpecObj = RecyclableObject::FromVar(propertySpec);

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::enumerable) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::enumerable, &value, scriptContext))

{

descriptor->SetEnumerable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'enumerable'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::configurable) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::configurable, &value, scriptContext))

{

descriptor->SetConfigurable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'configurable'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::value) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::value, &value, scriptContext))

{

descriptor->SetValue(value);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'value'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::writable) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::writable, &value, scriptContext))

{

descriptor->SetWritable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'writable'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::get) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::get, &value, scriptContext))

{

if (JavascriptOperators::GetTypeId(value) != TypeIds\_Undefined && (false == JavascriptConversion::IsCallable(value)))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::get)->GetBuffer());

}

descriptor->SetGetter(value);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'get'.");

}

}

if (JavascriptOperators::HasProperty(propertySpecObj, PropertyIds::set) == TRUE)

{

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::set, &value, scriptContext))

{

if (JavascriptOperators::GetTypeId(value) != TypeIds\_Undefined && (false == JavascriptConversion::IsCallable(value)))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::set)->GetBuffer());

}

descriptor->SetSetter(value);

}

else

{

AssertMsg(FALSE, "Proxy : HasProperty and GetProperty's result don't match for 'set'.");

}

}

return TRUE;

}

BOOL JavascriptOperators::ToPropertyDescriptorForGenericObjects(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext)

{

if (!JavascriptOperators::IsObject(propertySpec))

{

return FALSE;

}

Var value;

RecyclableObject\* propertySpecObj = RecyclableObject::FromVar(propertySpec);

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::enumerable, &value, scriptContext))

{

descriptor->SetEnumerable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::configurable, &value, scriptContext))

{

descriptor->SetConfigurable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::value, &value, scriptContext))

{

descriptor->SetValue(value);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::writable, &value, scriptContext))

{

descriptor->SetWritable(JavascriptConversion::ToBoolean(value, scriptContext) ? true : false);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::get, &value, scriptContext))

{

if (JavascriptOperators::GetTypeId(value) != TypeIds\_Undefined && (false == JavascriptConversion::IsCallable(value)))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::get)->GetBuffer());

}

descriptor->SetGetter(value);

}

if (JavascriptOperators::GetProperty(propertySpecObj, PropertyIds::set, &value, scriptContext))

{

if (JavascriptOperators::GetTypeId(value) != TypeIds\_Undefined && (false == JavascriptConversion::IsCallable(value)))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_NeedFunction, scriptContext->GetPropertyName(PropertyIds::set)->GetBuffer());

}

descriptor->SetSetter(value);

}

return TRUE;

}

BOOL JavascriptOperators::ToPropertyDescriptor(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext)

{

if (JavascriptProxy::Is(propertySpec))

{

if (ToPropertyDescriptorForProxyObjects(propertySpec, descriptor, scriptContext) == FALSE)

{

return FALSE;

}

}

else

{

if (ToPropertyDescriptorForGenericObjects(propertySpec, descriptor, scriptContext) == FALSE)

{

return FALSE;

}

}

if (descriptor->GetterSpecified() || descriptor->SetterSpecified())

{

if (descriptor->ValueSpecified())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Property\_CannotHaveAccessorsAndValue);

}

if (descriptor->WritableSpecified())

{

long hCode = descriptor->IsWritable() ? JSERR\_InvalidAttributeTrue : JSERR\_InvalidAttributeFalse;

JavascriptError::ThrowTypeError(scriptContext, hCode, L"writable");

}

}

descriptor->SetOriginal(propertySpec);

return TRUE;

}

void JavascriptOperators::CompletePropertyDescriptor(PropertyDescriptor\* resultDescriptor, PropertyDescriptor\* likeDescriptor, ScriptContext\* requestContext)

{

const PropertyDescriptor\* likePropertyDescriptor = likeDescriptor;

// 1. Assert: LikeDesc is either a Property Descriptor or undefined.

// 2. ReturnIfAbrupt(Desc).

// 3. Assert : Desc is a Property Descriptor

// 4. If LikeDesc is undefined, then set LikeDesc to Record{ [[Value]]: undefined, [[Writable]] : false, [[Get]] : undefined, [[Set]] : undefined, [[Enumerable]] : false, [[Configurable]] : false }.

if (likePropertyDescriptor == nullptr)

{

likePropertyDescriptor = requestContext->GetLibrary()->GetDefaultPropertyDescriptor();

}

// 5. If either IsGenericDescriptor(Desc) or IsDataDescriptor(Desc) is true, then

if (resultDescriptor->IsDataDescriptor() || resultDescriptor->IsGenericDescriptor())

{

// a.If Desc does not have a[[Value]] field, then set Desc.[[Value]] to LikeDesc.[[Value]].

// b.If Desc does not have a[[Writable]] field, then set Desc.[[Writable]] to LikeDesc.[[Writable]].

if (!resultDescriptor->ValueSpecified())

{

resultDescriptor->SetValue(likePropertyDescriptor->GetValue());

}

if (!resultDescriptor->WritableSpecified())

{

resultDescriptor->SetWritable(likePropertyDescriptor->IsWritable());

}

}

else

{

// 6. Else,

// a.If Desc does not have a[[Get]] field, then set Desc.[[Get]] to LikeDesc.[[Get]].

// b.If Desc does not have a[[Set]] field, then set Desc.[[Set]] to LikeDesc.[[Set]].

if (!resultDescriptor->GetterSpecified())

{

resultDescriptor->SetGetter(likePropertyDescriptor->GetGetter());

}

if (!resultDescriptor->SetterSpecified())

{

resultDescriptor->SetSetter(likePropertyDescriptor->GetSetter());

}

}

// 7. If Desc does not have an[[Enumerable]] field, then set Desc.[[Enumerable]] to LikeDesc.[[Enumerable]].

// 8. If Desc does not have a[[Configurable]] field, then set Desc.[[Configurable]] to LikeDesc.[[Configurable]].

// 9. Return Desc.

if (!resultDescriptor->EnumerableSpecified())

{

resultDescriptor->SetEnumerable(likePropertyDescriptor->IsEnumerable());

}

if (!resultDescriptor->ConfigurableSpecified())

{

resultDescriptor->SetConfigurable(likePropertyDescriptor->IsConfigurable());

}

}

Var JavascriptOperators::OP\_InvokePut(Js::ScriptContext \*scriptContext, Var instance, CallInfo callInfo, ...)

{

// Handle a store to a call result: x(y) = z.

// This is not strictly permitted in JScript, but some scripts expect to be able to use

// the syntax to set properties of ActiveX objects.

// We handle this by deferring to a virtual method of type. This incurs an extra level of

// indirection but seems preferable to adding the "put" method as a member of every type

// and using the normal JScript calling mechanism.

RUNTIME\_ARGUMENTS(args, callInfo);

AssertMsg(args.Info.Count > 0, "Missing this argument in InvokePut");

if (TaggedNumber::Is(instance))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction /\* TODO-ERROR: get arg name - aFunc \*/);

}

RecyclableObject\* function = RecyclableObject::FromVar(instance);

return function->InvokePut(args);

}

// Conformance to: ES5 8.6.1.

// Set attributes on the object as provided by property descriptor.

// If force parameter is true, we force SetAttributes call even if none of the attributes are defined by the descriptor.

// NOTE: does not set [[Get]], [Set]], [[Value]]

void JavascriptOperators::SetAttributes(RecyclableObject\* object, PropertyId propId, const PropertyDescriptor& descriptor, bool force)

{

Assert(object);

BOOL isWritable = FALSE;

if (descriptor.IsDataDescriptor())

{

isWritable = descriptor.WritableSpecified() ? descriptor.IsWritable() : FALSE;

}

else if (descriptor.IsAccessorDescriptor())

{

// The reason is that JavascriptOperators::OP\_SetProperty checks for RecyclableObject::FromVar(instance)->IsWritableOrAccessor(propertyId),

// which should in fact check for 'is writable or accessor' but since there is no GetAttributes, we can't do that efficiently.

isWritable = TRUE;

}

// CONSIDER: call object->SetAttributes which is much more efficient as that's 1 call instead of 3.

// Can't do that now as object->SetAttributes doesn't provide a way which attributes to modify and which not.

if (force || descriptor.ConfigurableSpecified())

{

object->SetConfigurable(propId, descriptor.ConfigurableSpecified() ? descriptor.IsConfigurable() : FALSE);

}

if (force || descriptor.EnumerableSpecified())

{

object->SetEnumerable(propId, descriptor.EnumerableSpecified() ? descriptor.IsEnumerable() : FALSE);

}

if (force || descriptor.WritableSpecified() || isWritable)

{

object->SetWritable(propId, isWritable);

}

}

void JavascriptOperators::OP\_ClearAttributes(Var instance, PropertyId propertyId)

{

Assert(instance);

if (RecyclableObject::Is(instance))

{

RecyclableObject\* obj = RecyclableObject::FromVar(instance);

obj->SetAttributes(propertyId, PropertyNone);

}

}

void JavascriptOperators::OP\_Freeze(Var instance)

{

Assert(instance);

if (RecyclableObject::Is(instance))

{

RecyclableObject\* obj = RecyclableObject::FromVar(instance);

obj->Freeze();

}

}

BOOL JavascriptOperators::Reject(bool throwOnError, ScriptContext\* scriptContext, long errorCode, PropertyId propertyId)

{

Assert(scriptContext);

if (throwOnError)

{

JavascriptError::ThrowTypeError(scriptContext, errorCode, scriptContext->GetThreadContext()->GetPropertyName(propertyId)->GetBuffer());

}

return FALSE;

}

bool JavascriptOperators::AreSamePropertyDescriptors(const PropertyDescriptor\* x, const PropertyDescriptor\* y, ScriptContext\* scriptContext)

{

Assert(scriptContext);

if (x->ConfigurableSpecified() != y->ConfigurableSpecified() || x->IsConfigurable() != y->IsConfigurable() ||

x->EnumerableSpecified() != y->EnumerableSpecified() || x->IsEnumerable() != y->IsEnumerable())

{

return false;

}

if (x->IsDataDescriptor())

{

if (!y->IsDataDescriptor() || x->WritableSpecified() != y->WritableSpecified() || x->IsWritable() != y->IsWritable())

{

return false;

}

if (x->ValueSpecified())

{

if (!y->ValueSpecified() || !JavascriptConversion::SameValue(x->GetValue(), y->GetValue()))

{

return false;

}

}

}

else if (x->IsAccessorDescriptor())

{

if (!y->IsAccessorDescriptor())

{

return false;

}

if (x->GetterSpecified())

{

if (!y->GetterSpecified() || !JavascriptConversion::SameValue(

JavascriptOperators::CanonicalizeAccessor(x->GetGetter(), scriptContext),

JavascriptOperators::CanonicalizeAccessor(y->GetGetter(), scriptContext)))

{

return false;

}

}

if (x->SetterSpecified())

{

if (!y->SetterSpecified() || !JavascriptConversion::SameValue(

JavascriptOperators::CanonicalizeAccessor(x->GetSetter(), scriptContext),

JavascriptOperators::CanonicalizeAccessor(y->GetSetter(), scriptContext)))

{

return false;

}

}

}

return true;

}

// Check if an accessor is undefined (null or defaultAccessor)

bool JavascriptOperators::IsUndefinedAccessor(Var accessor, ScriptContext\* scriptContext)

{

return nullptr == accessor || scriptContext->GetLibrary()->GetDefaultAccessorFunction() == accessor;

}

// Converts default accessor to undefined.

// Can be used when comparing accessors.

Var JavascriptOperators::CanonicalizeAccessor(Var accessor, ScriptContext\* scriptContext)

{

Assert(scriptContext);

if (IsUndefinedAccessor(accessor, scriptContext))

{

return scriptContext->GetLibrary()->GetUndefined();

}

return accessor;

}

Var JavascriptOperators::DefaultAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

return function->GetLibrary()->GetUndefined();

}

void FrameDisplay::SetItem(uint index, void\* item)

{

AssertMsg(index < this->length, "Invalid frame display access");

scopes[index] = item;

}

void \*FrameDisplay::GetItem(uint index)

{

AssertMsg(index < this->length, "Invalid frame display access");

return scopes[index];

}

// Grab the "this" pointer, mapping a root object to its associated host object.

Var JavascriptOperators::RootToThisObject(const Var object, ScriptContext\* scriptContext)

{

Js::Var thisVar = object;

TypeId typeId = Js::JavascriptOperators::GetTypeId(thisVar);

switch (typeId)

{

case Js::TypeIds\_GlobalObject:

return ((Js::GlobalObject\*)thisVar)->ToThis();

case Js::TypeIds\_ModuleRoot:

return Js::JavascriptOperators::GetThisFromModuleRoot(thisVar);

default:

if (typeId == scriptContext->GetDirectHostTypeId())

{

return ((RecyclableObject\*)thisVar)->GetLibrary()->GetGlobalObject()->ToThis();

}

}

return thisVar;

}

Var JavascriptOperators::CallGetter(RecyclableObject \* const function, Var const object, ScriptContext \* requestContext)

{

ScriptContext \* scriptContext = function->GetScriptContext();

ThreadContext \* threadContext = scriptContext->GetThreadContext();

return threadContext->ExecuteImplicitCall(function, ImplicitCall\_Accessor, [=]() -> Js::Var

{

// Stack object should have a pre-op bail on implicit call. We shouldn't see them here.

// Stack numbers are ok, as we will call ToObject to wrap it in a number object anyway

// See JavascriptOperators::GetThisHelper

Assert(JavascriptOperators::GetTypeId(object) == TypeIds\_Integer ||

JavascriptOperators::GetTypeId(object) == TypeIds\_Number || !ThreadContext::IsOnStack(object));

// Verify that the scriptcontext is alive before firing getter/setter

if (!scriptContext->VerifyAlive(!function->IsExternal(), requestContext))

{

return nullptr;

}

CallFlags flags = CallFlags\_Value;

Var thisVar = RootToThisObject(object, scriptContext);

RecyclableObject\* marshalledFunction = RecyclableObject::FromVar(CrossSite::MarshalVar(requestContext, function));

Var result = marshalledFunction->GetEntryPoint()(function, CallInfo(flags, 1), thisVar);

result = CrossSite::MarshalVar(requestContext, result);

return result;

});

}

void JavascriptOperators::CallSetter(RecyclableObject \* const function, Var const object, Var const value, ScriptContext \* requestContext)

{

ScriptContext \* scriptContext = function->GetScriptContext();

ThreadContext \* threadContext = scriptContext->GetThreadContext();

threadContext->ExecuteImplicitCall(function, ImplicitCall\_Accessor, [=]() -> Js::Var

{

// Stack object should have a pre-op bail on implicit call. We shouldn't see them here.

// Stack numbers are ok, as we will call ToObject to wrap it in a number object anyway

// See JavascriptOperators::GetThisHelper

Assert(JavascriptOperators::GetTypeId(object) == TypeIds\_Integer ||

JavascriptOperators::GetTypeId(object) == TypeIds\_Number || !ThreadContext::IsOnStack(object));

// Verify that the scriptcontext is alive before firing getter/setter

if (!scriptContext->VerifyAlive(!function->IsExternal(), requestContext))

{

return nullptr;

}

CallFlags flags = CallFlags\_Value;

Var putValue = value;

// CONSIDER: Have requestContext everywhere, even in the setProperty related codepath.

if (requestContext)

{

putValue = CrossSite::MarshalVar(requestContext, value);

}

Var thisVar = RootToThisObject(object, scriptContext);

RecyclableObject\* marshalledFunction = function;

if (requestContext)

{

marshalledFunction = RecyclableObject::FromVar(CrossSite::MarshalVar(requestContext, function));

}

Var result = marshalledFunction->GetEntryPoint()(function, CallInfo(flags, 2), thisVar, putValue);

Assert(result);

return nullptr;

});

}

void \* JavascriptOperators::AllocMemForVarArray(size\_t size, Recycler\* recycler)

{

TRACK\_ALLOC\_INFO(recycler, Js::Var, Recycler, 0, (size\_t)(size / sizeof(Js::Var)));

return recycler->AllocZero(size);

}

void \* JavascriptOperators::AllocUninitializedNumber(Js::RecyclerJavascriptNumberAllocator \* allocator)

{

TRACK\_ALLOC\_INFO(allocator->GetRecycler(), Js::JavascriptNumber, Recycler, 0, (size\_t)-1);

return allocator->Alloc(sizeof(Js::JavascriptNumber));

}

void JavascriptOperators::ScriptAbort()

{

throw ScriptAbortException();

}

void PolymorphicInlineCache::Finalize(bool isShutdown)

{

if (size == 0)

{

// Already finalized

Assert(!inlineCaches && !prev && !next);

return;

}

Assert(inlineCaches && size > 0);

// If we're not shutting down (as in closing the script context), we need to remove our inline caches from

// thread context's invalidation lists, and release memory back to the arena. During script context shutdown,

// we leave everything in place, because the inline cache arena will stay alive until script context is destroyed

// (as in destructor has been called) and thus the invalidation lists are safe to keep references to caches from this

// script context. We will, however, zero all inline caches so that we don't have to process them on subsequent

// collections, which may still happen from other script contexts.

if (isShutdown)

{

memset(inlineCaches, 0, size \* sizeof(InlineCache));

}

else

{

for (int i = 0; i < size; i++)

{

inlineCaches[i].RemoveFromInvalidationList();

}

AllocatorDeleteArray(InlineCacheAllocator, functionBody->GetScriptContext()->GetInlineCacheAllocator(), size, inlineCaches);

#ifdef POLY\_INLINE\_CACHE\_SIZE\_STATS

functionBody->GetScriptContext()->GetInlineCacheAllocator()->LogPolyCacheFree(size \* sizeof(InlineCache));

#endif

}

// Remove this PolymorphicInlineCache from the list

if (this == functionBody->GetPolymorphicInlineCachesHead())

{

Assert(!prev);

if (next)

{

Assert(next->prev == this);

next->prev = nullptr;

}

functionBody->SetPolymorphicInlineCachesHead(next);

}

else

{

if (prev)

{

Assert(prev->next == this);

prev->next = next;

}

if (next)

{

Assert(next->prev == this);

next->prev = prev;

}

}

prev = next = nullptr;

inlineCaches = nullptr;

size = 0;

}

JavascriptString \* JavascriptOperators::Concat3(Var aLeft, Var aCenter, Var aRight, ScriptContext \* scriptContext)

{

// Make sure we do the conversion in order from left to right

JavascriptString \* strLeft = JavascriptConversion::ToPrimitiveString(aLeft, scriptContext);

JavascriptString \* strCenter = JavascriptConversion::ToPrimitiveString(aCenter, scriptContext);

JavascriptString \* strRight = JavascriptConversion::ToPrimitiveString(aRight, scriptContext);

return JavascriptString::Concat3(strLeft, strCenter, strRight);

}

JavascriptString \*

JavascriptOperators::NewConcatStrMulti(Var a1, Var a2, uint count, ScriptContext \* scriptContext)

{

// Make sure we do the conversion in order

JavascriptString \* str1 = JavascriptConversion::ToPrimitiveString(a1, scriptContext);

JavascriptString \* str2 = JavascriptConversion::ToPrimitiveString(a2, scriptContext);

return ConcatStringMulti::New(count, str1, str2, scriptContext);

}

void

JavascriptOperators::SetConcatStrMultiItem(Var concatStr, Var str, uint index, ScriptContext \* scriptContext)

{

ConcatStringMulti::FromVar(concatStr)->SetItem(index,

JavascriptConversion::ToPrimitiveString(str, scriptContext));

}

void

JavascriptOperators::SetConcatStrMultiItem2(Var concatStr, Var str1, Var str2, uint index, ScriptContext \* scriptContext)

{

ConcatStringMulti \* cs = ConcatStringMulti::FromVar(concatStr);

cs->SetItem(index, JavascriptConversion::ToPrimitiveString(str1, scriptContext));

cs->SetItem(index + 1, JavascriptConversion::ToPrimitiveString(str2, scriptContext));

}

void JavascriptOperators::OP\_SetComputedNameVar(Var method, Var computedNameVar)

{

ScriptFunctionBase \*scriptFunction = ScriptFunctionBase::FromVar(method);

scriptFunction->SetComputedNameVar(computedNameVar);

}

void JavascriptOperators::OP\_SetHomeObj(Var method, Var homeObj)

{

ScriptFunctionBase \*scriptFunction = ScriptFunctionBase::FromVar(method);

scriptFunction->SetHomeObj(homeObj);

}

Var JavascriptOperators::OP\_LdSuper(Var scriptFunction, ScriptContext \* scriptContext)

{

// Ensure this is not a stack ScriptFunction

if (!ScriptFunction::Is(scriptFunction) || ThreadContext::IsOnStack(scriptFunction))

{

return scriptContext->GetLibrary()->GetUndefined();

}

ScriptFunction \*instance = ScriptFunction::FromVar(scriptFunction);

// We keep a reference to the current class rather than its super prototype

// since the prototype could change.

Var homeObj = instance->GetHomeObj();

if (homeObj == nullptr || !RecyclableObject::Is(homeObj))

{

return scriptContext->GetLibrary()->GetUndefined();

}

RecyclableObject \*thisObjPrototype = RecyclableObject::FromVar(homeObj);

Assert(thisObjPrototype != nullptr);

RecyclableObject \*superBase = thisObjPrototype->GetPrototype();

if (superBase == nullptr || !RecyclableObject::Is(superBase))

{

return scriptContext->GetLibrary()->GetUndefined();

}

return superBase;

}

Var JavascriptOperators::OP\_LdSuperCtor(Var scriptFunction, ScriptContext \* scriptContext)

{

// use self as value of [[FunctionObject]] - this is true only for constructors

Assert(RecyclableObject::Is(scriptFunction));

Assert(JavascriptOperators::IsClassConstructor(scriptFunction)); // non-constructors cannot have direct super

RecyclableObject \*superCtor = RecyclableObject::FromVar(scriptFunction)->GetPrototype();

if (superCtor == nullptr || !IsConstructor(superCtor))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NotAConstructor, L"super");

}

return superCtor;

}

Var JavascriptOperators::ScopedLdSuperHelper(Var scriptFunction, Js::PropertyId propertyId, ScriptContext \* scriptContext)

{

ScriptFunction \*instance = ScriptFunction::FromVar(scriptFunction);

Var superRef = nullptr;

FrameDisplay \*frameDisplay = instance->GetEnvironment();

if (frameDisplay->GetLength() == 0)

{

// Globally scoped evals are a syntax error

JavascriptError::ThrowSyntaxError(scriptContext, ERRSuperInGlobalEval, L"super");

}

// Iterate over the scopes in the FrameDisplay, looking for the super property.

for (unsigned i = 0; i < frameDisplay->GetLength(); ++i)

{

void \*currScope = frameDisplay->GetItem(i);

if (RecyclableObject::Is(currScope))

{

if (BlockActivationObject::Is(currScope))

{

// We won't find super in a block scope.

continue;

}

RecyclableObject \*recyclableObject = RecyclableObject::FromVar(currScope);

if (GetProperty(recyclableObject, propertyId, &superRef, scriptContext))

{

return superRef;

}

if (HasProperty(recyclableObject, Js::PropertyIds::\_lexicalThisSlotSymbol))

{

// If we reach 'this' and haven't found the super reference, we don't need to look any further.

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_BadSuperReference, L"super");

}

}

}

if (superRef == nullptr)

{

// We didn't find a super reference. Emit a reference error.

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_BadSuperReference, L"super");

}

return superRef;

}

Var JavascriptOperators::OP\_ScopedLdSuper(Var scriptFunction, ScriptContext \* scriptContext)

{

return JavascriptOperators::ScopedLdSuperHelper(scriptFunction, Js::PropertyIds::\_superReferenceSymbol, scriptContext);

}

Var JavascriptOperators::OP\_ScopedLdSuperCtor(Var scriptFunction, ScriptContext \* scriptContext)

{

return JavascriptOperators::ScopedLdSuperHelper(scriptFunction, Js::PropertyIds::\_superCtorReferenceSymbol, scriptContext);

}

Var JavascriptOperators::OP\_ResumeYield(ResumeYieldData\* yieldData, RecyclableObject\* iterator)

{

// CONSIDER: Fast path this early out return path in JITed code before helper call to avoid the helper call overhead in the common case e.g. next() calls.

if (yieldData->exceptionObj == nullptr)

{

return yieldData->data;

}

ScriptContext\* scriptContext = yieldData->exceptionObj->GetScriptContext();

bool isReturn = yieldData->exceptionObj->IsGeneratorReturnException();

if (iterator != nullptr)

{

PropertyId propertyId = isReturn ? PropertyIds::return\_ : PropertyIds::throw\_;

Var prop = nullptr;

Var args[] = { iterator, yieldData->data };

CallInfo callInfo(CallFlags\_Value, \_countof(args));

if (JavascriptOperators::GetProperty(iterator, iterator, propertyId, &prop, iterator->GetScriptContext())

&& prop != iterator->GetLibrary()->GetUndefined())

{

RecyclableObject\* method = RecyclableObject::FromVar(prop);

Var result = JavascriptFunction::CallFunction<true>(method, method->GetEntryPoint(), Arguments(callInfo, args));

if (isReturn)

{

if (!JavascriptOperators::IsObject(result))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

Var value = JavascriptOperators::GetProperty(RecyclableObject::FromVar(result), PropertyIds::value, scriptContext);

// CONSIDER: Using an exception to carry the return value and force finally code to execute is a bit of a janky

// solution since we have to override the value here in the case of yield\* expressions. It works but is there

// a more elegant way?

//

// Instead what if ResumeYield was a "set Dst then optionally branch" opcode, that could also throw? Then we could

// avoid using a special exception entirely with byte code something like this:

//

// ;; Ry is the yieldData

//

// ResumeYield Rx Ry $returnPathLabel

// ... code like normal

// $returnPathLabel:

// Ld\_A R0 Rx

// Br $exitFinallyAndReturn

//

// This would probably give better performance for the common case of calling next() on generators since we wouldn't

// have to wrap the call to the generator code in a try catch.

yieldData->exceptionObj->SetThrownObject(value);

}

}

else if (!isReturn)

{

// Throw is called on yield\* but the iterator does not have a throw method. This is a protocol violation.

// So we have to call IteratorClose().

if (JavascriptOperators::GetProperty(iterator, iterator, PropertyIds::return\_, &prop, iterator->GetScriptContext())

&& prop != iterator->GetLibrary()->GetUndefined())

{

// As per the spec we ignore the inner result after checking whether it is a valid object

RecyclableObject\* method = RecyclableObject::FromVar(prop);

Var result = JavascriptFunction::CallFunction<true>(method, method->GetEntryPoint(), Arguments(callInfo, args));

if (!JavascriptOperators::IsObject(result))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

}

}

}

if (!isReturn)

{

// Use ThrowExceptionObject() to get debugger support for breaking on throw

JavascriptExceptionOperators::ThrowExceptionObject(yieldData->exceptionObj, scriptContext, true);

}

// Do not use ThrowExceptionObject for return() API exceptions since these exceptions are not real exceptions

throw yieldData->exceptionObj;

}

Var JavascriptOperators::OP\_AsyncSpawn(Var aGenerator, Var aThis, ScriptContext\* scriptContext)

{

JavascriptLibrary\* library = scriptContext->GetLibrary();

JavascriptExceptionObject\* e = nullptr;

JavascriptPromiseResolveOrRejectFunction\* resolve;

JavascriptPromiseResolveOrRejectFunction\* reject;

JavascriptPromiseAsyncSpawnExecutorFunction\* executor = library->CreatePromiseAsyncSpawnExecutorFunction(JavascriptPromise::EntryJavascriptPromiseAsyncSpawnExecutorFunction, (JavascriptGenerator\*)aGenerator, aThis);

JavascriptPromise\* promise = library->CreatePromise();

JavascriptPromise::InitializePromise(promise, &resolve, &reject, scriptContext);

try

{

executor->GetEntryPoint()(executor, CallInfo(CallFlags\_Value, 3), library->GetUndefined(), resolve, reject);

}

catch (JavascriptExceptionObject\* ex)

{

e = ex;

}

if (e != nullptr)

{

reject->GetEntryPoint()(reject, CallInfo(CallFlags\_Value, 2), library->GetUndefined(), e->GetThrownObject(scriptContext));

}

return promise;

}

Js::Var

JavascriptOperators::BoxStackInstance(Js::Var instance, ScriptContext \* scriptContext, bool allowStackFunction)

{

if (!ThreadContext::IsOnStack(instance) || (allowStackFunction && !TaggedNumber::Is(instance) && (\*(int\*)instance & 1)))

{

return instance;

}

TypeId typeId = JavascriptOperators::GetTypeId(instance);

switch (typeId)

{

case Js::TypeIds\_Number:

#if !FLOATVAR

return JavascriptNumber::BoxStackInstance(instance, scriptContext);

#endif

// fall-through

case Js::TypeIds\_Integer:

return instance;

case Js::TypeIds\_RegEx:

return JavascriptRegExp::BoxStackInstance(JavascriptRegExp::FromVar(instance));

case Js::TypeIds\_Object:

return DynamicObject::BoxStackInstance(DynamicObject::FromVar(instance));

case Js::TypeIds\_Array:

return JavascriptArray::BoxStackInstance(JavascriptArray::FromVar(instance));

case Js::TypeIds\_NativeIntArray:

return JavascriptNativeIntArray::BoxStackInstance(JavascriptNativeIntArray::FromVar(instance));

case Js::TypeIds\_NativeFloatArray:

return JavascriptNativeFloatArray::BoxStackInstance(JavascriptNativeFloatArray::FromVar(instance));

case Js::TypeIds\_Function:

Assert(allowStackFunction);

// Stack functions are deal with not mar mark them, but by nested function escape analysis

// in the front end. No need to box here.

return instance;

#if ENABLE\_COPYONACCESS\_ARRAY

case Js::TypeIds\_CopyOnAccessNativeIntArray:

Assert(false);

// fall-through

#endif

default:

Assert(false);

return instance;

};

}

ImplicitCallFlags

JavascriptOperators::CacheAndClearImplicitBit(ScriptContext\* scriptContext)

{

ImplicitCallFlags prevImplicitCallFlags = scriptContext->GetThreadContext()->GetImplicitCallFlags();

scriptContext->GetThreadContext()->ClearImplicitCallFlags();

return prevImplicitCallFlags;

}

ImplicitCallFlags

JavascriptOperators::CheckAndUpdateFunctionBodyWithImplicitFlag(FunctionBody\* functionBody)

{

ScriptContext\* scriptContext = functionBody->GetScriptContext();

ImplicitCallFlags currImplicitCallFlags = scriptContext->GetThreadContext()->GetImplicitCallFlags();

if ((currImplicitCallFlags > ImplicitCall\_None))

{

functionBody->SetHasOnlyThisStmts(false);

}

return currImplicitCallFlags;

}

void

JavascriptOperators::RestoreImplicitFlag(ScriptContext\* scriptContext, ImplicitCallFlags prevImplicitCallFlags, ImplicitCallFlags currImplicitCallFlags)

{

scriptContext->GetThreadContext()->SetImplicitCallFlags((ImplicitCallFlags)(prevImplicitCallFlags | currImplicitCallFlags));

}

FunctionProxy\*

JavascriptOperators::GetDeferredDeserializedFunctionProxy(JavascriptFunction\* func)

{

FunctionProxy\* proxy = func->GetFunctionProxy();

if (proxy->GetFunctionProxy() != proxy)

{

proxy = proxy->GetFunctionProxy();

}

return proxy;

}

template <>

Js::Var JavascriptOperators::GetElementAtIndex(Js::JavascriptArray\* arrayObject, UINT index, Js::ScriptContext\* scriptContext)

{

Js::Var result;

if (Js::JavascriptOperators::OP\_GetElementI\_ArrayFastPath(arrayObject, index, &result, scriptContext))

{

return result;

}

return scriptContext->GetMissingItemResult(arrayObject, index);

}

template<>

Js::Var JavascriptOperators::GetElementAtIndex(Js::JavascriptNativeIntArray\* arrayObject, UINT index, Js::ScriptContext\* scriptContext)

{

Js::Var result;

if (Js::JavascriptOperators::OP\_GetElementI\_ArrayFastPath(arrayObject, index, &result, scriptContext))

{

return result;

}

return scriptContext->GetMissingItemResult(arrayObject, index);

}

template<>

Js::Var JavascriptOperators::GetElementAtIndex(Js::JavascriptNativeFloatArray\* arrayObject, UINT index, Js::ScriptContext\* scriptContext)

{

Js::Var result;

if (Js::JavascriptOperators::OP\_GetElementI\_ArrayFastPath(arrayObject, index, &result, scriptContext))

{

return result;

}

return scriptContext->GetMissingItemResult(arrayObject, index);

}

template<>

Js::Var JavascriptOperators::GetElementAtIndex(Js::Var\* arrayObject, UINT index, Js::ScriptContext\* scriptContext)

{

return Js::JavascriptOperators::OP\_GetElementI\_Int32(\*arrayObject, index, scriptContext);

}

template<typename T>

void JavascriptOperators::ObjectToNativeArray(T\* arrayObject,

JsNativeValueType valueType,

\_\_in UINT length,

\_\_in UINT elementSize,

\_\_out\_bcount(length\*elementSize) byte\* buffer,

Js::ScriptContext\* scriptContext)

{

Var element;

uint64 allocSize = length \* elementSize;

// TODO:further fast path the call for things like IntArray convert to int, floatarray convert to float etc.

// such that we don't need boxing.

switch (valueType)

{

case JsInt8Type:

AnalysisAssert(elementSize == sizeof(int8));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(int8) <= allocSize);

((int8\*)buffer)[i] = Js::JavascriptConversion::ToInt8(element, scriptContext);

}

break;

case JsUint8Type:

AnalysisAssert(elementSize == sizeof(uint8));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(uint8) <= allocSize);

((uint8\*)buffer)[i] = Js::JavascriptConversion::ToUInt8(element, scriptContext);

}

break;

case JsInt16Type:

AnalysisAssert(elementSize == sizeof(int16));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(int16) <= allocSize);

((int16\*)buffer)[i] = Js::JavascriptConversion::ToInt16(element, scriptContext);

}

break;

case JsUint16Type:

AnalysisAssert(elementSize == sizeof(uint16));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(uint16) <= allocSize);

((uint16\*)buffer)[i] = Js::JavascriptConversion::ToUInt16(element, scriptContext);

}

break;

case JsInt32Type:

AnalysisAssert(elementSize == sizeof(int32));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(int32) <= allocSize);

((int32\*)buffer)[i] = Js::JavascriptConversion::ToInt32(element, scriptContext);

}

break;

case JsUint32Type:

AnalysisAssert(elementSize == sizeof(uint32));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(uint32) <= allocSize);

((uint32\*)buffer)[i] = Js::JavascriptConversion::ToUInt32(element, scriptContext);

}

break;

case JsInt64Type:

AnalysisAssert(elementSize == sizeof(int64));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(int64) <= allocSize);

((int64\*)buffer)[i] = Js::JavascriptConversion::ToInt64(element, scriptContext);

}

break;

case JsUint64Type:

AnalysisAssert(elementSize == sizeof(uint64));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(uint64) <= allocSize);

((uint64\*)buffer)[i] = Js::JavascriptConversion::ToUInt64(element, scriptContext);

}

break;

case JsFloatType:

AnalysisAssert(elementSize == sizeof(float));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(float) <= allocSize);

((float\*)buffer)[i] = Js::JavascriptConversion::ToFloat(element, scriptContext);

}

break;

case JsDoubleType:

AnalysisAssert(elementSize == sizeof(double));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(double) <= allocSize);

((double\*)buffer)[i] = Js::JavascriptConversion::ToNumber(element, scriptContext);

}

break;

case JsNativeStringType:

AnalysisAssert(elementSize == sizeof(JsNativeString));

for (UINT i = 0; i < length; i++)

{

element = GetElementAtIndex(arrayObject, i, scriptContext);

AnalysisAssert((i + 1) \* sizeof(JsNativeString) <= allocSize);

Js::JavascriptString\* string = Js::JavascriptConversion::ToString(element, scriptContext);

(((JsNativeString\*)buffer)[i]).str = string->GetSz();

(((JsNativeString\*)buffer)[i]).length = string->GetLength();

}

break;

default:

Assert(FALSE);

}

}

void JavascriptOperators::VarToNativeArray(Var arrayObject,

JsNativeValueType valueType,

\_\_in UINT length,

\_\_in UINT elementSize,

\_\_out\_bcount(length\*elementSize) byte\* buffer,

Js::ScriptContext\* scriptContext)

{

Js::DynamicObject\* dynamicObject = DynamicObject::FromVar(arrayObject);

if (dynamicObject->IsCrossSiteObject() || Js::TaggedInt::IsOverflow(length))

{

Js::JavascriptOperators::ObjectToNativeArray(&arrayObject, valueType, length, elementSize, buffer, scriptContext);

}

else

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(arrayObject);

#endif

switch (Js::JavascriptOperators::GetTypeId(arrayObject))

{

case TypeIds\_Array:

Js::JavascriptOperators::ObjectToNativeArray(Js::JavascriptArray::FromVar(arrayObject), valueType, length, elementSize, buffer, scriptContext);

break;

case TypeIds\_NativeFloatArray:

Js::JavascriptOperators::ObjectToNativeArray(Js::JavascriptNativeFloatArray::FromVar(arrayObject), valueType, length, elementSize, buffer, scriptContext);

break;

case TypeIds\_NativeIntArray:

Js::JavascriptOperators::ObjectToNativeArray(Js::JavascriptNativeIntArray::FromVar(arrayObject), valueType, length, elementSize, buffer, scriptContext);

break;

// We can have more specialized template if needed.

default:

Js::JavascriptOperators::ObjectToNativeArray(&arrayObject, valueType, length, elementSize, buffer, scriptContext);

}

}

}

// SpeciesConstructor abstract operation as described in ES6.0 Section 7.3.20

Var JavascriptOperators::SpeciesConstructor(RecyclableObject\* object, Var defaultConstructor, ScriptContext\* scriptContext)

{

//1.Assert: Type(O) is Object.

Assert(JavascriptOperators::IsObject(object));

//2.Let C be Get(O, "constructor").

//3.ReturnIfAbrupt(C).

Var constructor = JavascriptOperators::GetProperty(object, PropertyIds::constructor, scriptContext);

if (scriptContext->GetConfig()->IsES6SpeciesEnabled())

{

//4.If C is undefined, return defaultConstructor.

if (JavascriptOperators::IsUndefinedObject(constructor))

{

return defaultConstructor;

}

//5.If Type(C) is not Object, throw a TypeError exception.

if (!JavascriptOperators::IsObject(constructor))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject, L"[constructor]");

}

//6.Let S be Get(C, @@species).

//7.ReturnIfAbrupt(S).

Var species = nullptr;

if (!JavascriptOperators::GetProperty(RecyclableObject::FromVar(constructor), PropertyIds::\_symbolSpecies, &species, scriptContext)

|| JavascriptOperators::IsUndefinedOrNullType(JavascriptOperators::GetTypeId(species)))

{

//8.If S is either undefined or null, return defaultConstructor.

return defaultConstructor;

}

constructor = species;

}

//9.If IsConstructor(S) is true, return S.

if (JavascriptOperators::IsConstructor(constructor))

{

return constructor;

}

//10.Throw a TypeError exception.

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NotAConstructor, L"constructor[Symbol.species]");

}

BOOL JavascriptOperators::GreaterEqual(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// Works whether it is TaggedInt31 or TaggedInt32

return ::Math::PointerCastToIntegralTruncate<int>(aLeft) >= ::Math::PointerCastToIntegralTruncate<int>(aRight);

}

if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) >= JavascriptNumber::GetValue(aRight);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::GetValue(aLeft) >= TaggedInt::ToDouble(aRight);

}

}

else

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) >= JavascriptNumber::GetValue(aRight);

}

}

return !RelationalComparsionHelper(aLeft, aRight, scriptContext, true, true);

}

BOOL JavascriptOperators::LessEqual(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// Works whether it is TaggedInt31 or TaggedInt32

return ::Math::PointerCastToIntegralTruncate<int>(aLeft) <= ::Math::PointerCastToIntegralTruncate<int>(aRight);

}

if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) <= JavascriptNumber::GetValue(aRight);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::GetValue(aLeft) <= TaggedInt::ToDouble(aRight);

}

}

else

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) <= JavascriptNumber::GetValue(aRight);

}

}

return !RelationalComparsionHelper(aRight, aLeft, scriptContext, false, true);

}

BOOL JavascriptOperators::NotEqual(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

//

// TODO: Change to use Abstract Equality Comparison Algorithm (ES3.0: S11.9.3):

// - Evaluate left, then right, operands to preserve correct evaluation order.

// - Call algorithm, potentially reversing arguments.

//

return !Equal(aLeft, aRight, scriptContext);

}

// NotStrictEqual() returns whether the two vars have strict equality, as

// described in (ES3.0: S11.9.5, S11.9.6).

BOOL JavascriptOperators::NotStrictEqual(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return !StrictEqual(aLeft, aRight, scriptContext);

}

bool JavascriptOperators::CheckIfObjectAndPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* object)

{

Assert(object);

if (object->GetType()->HasSpecialPrototype())

{

TypeId typeId = object->GetTypeId();

if (typeId == TypeIds\_Null)

{

return true;

}

if (typeId == TypeIds\_Proxy)

{

return false;

}

}

if (!object->HasOnlyWritableDataProperties())

{

return false;

}

return CheckIfPrototypeChainHasOnlyWritableDataProperties(object->GetPrototype());

}

bool JavascriptOperators::CheckIfPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* prototype)

{

Assert(prototype);

if (prototype->GetType()->AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties())

{

Assert(DoCheckIfPrototypeChainHasOnlyWritableDataProperties(prototype));

return true;

}

return DoCheckIfPrototypeChainHasOnlyWritableDataProperties(prototype);

}

// Does a quick check to see if the specified object (which should be a prototype object) and all objects in its prototype

// chain have only writable data properties (i.e. no accessors or non-writable properties).

bool JavascriptOperators::DoCheckIfPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* prototype)

{

Assert(prototype);

Type \*const originalType = prototype->GetType();

ScriptContext \*const scriptContext = prototype->GetScriptContext();

bool onlyOneScriptContext = true;

TypeId typeId;

for (; (typeId = prototype->GetTypeId()) != TypeIds\_Null; prototype = prototype->GetPrototype())

{

if (typeId == TypeIds\_Proxy)

{

return false;

}

if (!prototype->HasOnlyWritableDataProperties())

{

return false;

}

if (prototype->GetScriptContext() != scriptContext)

{

onlyOneScriptContext = false;

}

}

if (onlyOneScriptContext)

{

// See JavascriptLibrary::typesEnsuredToHaveOnlyWritableDataPropertiesInItAndPrototypeChain for a description of

// this cache. Technically, we could register all prototypes in the chain but this is good enough for now.

originalType->SetAreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties(true);

}

return true;

}

BOOL JavascriptOperators::Equal(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (aLeft == aRight)

{

if (TaggedInt::Is(aLeft) || JavascriptObject::Is(aLeft))

{

return true;

}

else

{

return Equal\_Full(aLeft, aRight, scriptContext);

}

}

if (JavascriptString::Is(aLeft) && JavascriptString::Is(aRight))

{

JavascriptString\* left = (JavascriptString\*)aLeft;

JavascriptString\* right = (JavascriptString\*)aRight;

if (left->GetLength() == right->GetLength())

{

if (left->UnsafeGetBuffer() != NULL && right->UnsafeGetBuffer() != NULL)

{

if (left->GetLength() == 1)

{

return left->UnsafeGetBuffer()[0] == right->UnsafeGetBuffer()[0];

}

return memcmp(left->UnsafeGetBuffer(), right->UnsafeGetBuffer(), left->GetLength() \* sizeof(left->UnsafeGetBuffer()[0])) == 0;

}

// fall through to Equal\_Full

}

else

{

return false;

}

}

return Equal\_Full(aLeft, aRight, scriptContext);

}

BOOL JavascriptOperators::Greater(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// Works whether it is TaggedInt31 or TaggedInt32

return ::Math::PointerCastToIntegralTruncate<int>(aLeft) > ::Math::PointerCastToIntegralTruncate<int>(aRight);

}

if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) > JavascriptNumber::GetValue(aRight);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::GetValue(aLeft) > TaggedInt::ToDouble(aRight);

}

}

else

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) > JavascriptNumber::GetValue(aRight);

}

}

return Greater\_Full(aLeft, aRight, scriptContext);

}

BOOL JavascriptOperators::Less(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

// Works whether it is TaggedInt31 or TaggedInt32

return ::Math::PointerCastToIntegralTruncate<int>(aLeft) < ::Math::PointerCastToIntegralTruncate<int>(aRight);

}

if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return TaggedInt::ToDouble(aLeft) < JavascriptNumber::GetValue(aRight);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::GetValue(aLeft) < TaggedInt::ToDouble(aRight);

}

}

else

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::GetValue(aLeft) < JavascriptNumber::GetValue(aRight);

}

}

return Less\_Full(aLeft, aRight, scriptContext);

}

Var JavascriptOperators::ToObject(Var aRight, ScriptContext\* scriptContext)

{

RecyclableObject\* object = nullptr;

if (FALSE == JavascriptConversion::ToObject(aRight, scriptContext, &object))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject /\* TODO-ERROR: get arg name - aValue \*/);

}

return object;

}

Var JavascriptOperators::ToWithObject(Var aRight, ScriptContext\* scriptContext)

{

RecyclableObject\* object = RecyclableObject::FromVar(aRight);

WithScopeObject\* withWrapper = RecyclerNew(scriptContext->GetRecycler(), WithScopeObject, object, scriptContext->GetLibrary()->GetWithType());

return withWrapper;

}

Var JavascriptOperators::ToNumber(Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aRight) || (JavascriptNumber::Is\_NoTaggedIntCheck(aRight)))

{

return aRight;

}

return JavascriptNumber::ToVarNoCheck(JavascriptConversion::ToNumber\_Full(aRight, scriptContext), scriptContext);

}

BOOL JavascriptOperators::IsObject(Var aValue)

{

return GetTypeId(aValue) > TypeIds\_LastJavascriptPrimitiveType;

}

BOOL JavascriptOperators::IsObjectType(TypeId typeId)

{

return typeId > TypeIds\_LastJavascriptPrimitiveType;

}

BOOL JavascriptOperators::IsExposedType(TypeId typeId)

{

return typeId <= TypeIds\_LastTrueJavascriptObjectType && typeId != TypeIds\_HostDispatch;

}

BOOL JavascriptOperators::IsObjectOrNull(Var instance)

{

TypeId typeId = GetTypeId(instance);

return IsObjectType(typeId) || typeId == TypeIds\_Null;

}

BOOL JavascriptOperators::IsUndefinedOrNullType(TypeId typeId)

{

return typeId <= TypeIds\_UndefinedOrNull;

}

BOOL JavascriptOperators::IsSpecialObjectType(TypeId typeId)

{

return typeId > TypeIds\_LastTrueJavascriptObjectType;

}

BOOL JavascriptOperators::IsUndefinedObject(Var instance)

{

return JavascriptOperators::GetTypeId(instance) == TypeIds\_Undefined;

}

BOOL JavascriptOperators::IsUndefinedObject(Var instance, RecyclableObject \*libraryUndefined)

{

Assert(JavascriptOperators::IsUndefinedObject(libraryUndefined));

return instance == libraryUndefined;

}

BOOL JavascriptOperators::IsUndefinedObject(Var instance, ScriptContext \*scriptContext)

{

return JavascriptOperators::IsUndefinedObject(instance, scriptContext->GetLibrary()->GetUndefined());

}

BOOL JavascriptOperators::IsUndefinedObject(Var instance, JavascriptLibrary\* library)

{

return JavascriptOperators::IsUndefinedObject(instance, library->GetUndefined());

}

BOOL JavascriptOperators::IsAnyNumberValue(Var instance)

{

TypeId typeId = GetTypeId(instance);

return TypeIds\_FirstNumberType <= typeId && typeId <= TypeIds\_LastNumberType;

}

BOOL JavascriptOperators::IsIterable(RecyclableObject\* instance, ScriptContext\* scriptContext)

{

if (JavascriptProxy::Is(instance))

{

Var func = JavascriptOperators::GetProperty(instance, PropertyIds::\_symbolIterator, scriptContext);

if (JavascriptOperators::IsUndefinedObject(func))

{

return FALSE;

}

else

{

return TRUE;

}

}

else

{

return JavascriptOperators::HasProperty(instance, PropertyIds::\_symbolIterator);

}

}

// GetIterator as described in ES6.0 (draft 22) Section 7.4.1

RecyclableObject\* JavascriptOperators::GetIterator(Var iterable, ScriptContext\* scriptContext)

{

RecyclableObject\* iterableObj = RecyclableObject::FromVar(JavascriptOperators::ToObject(iterable, scriptContext));

return JavascriptOperators::GetIterator(iterableObj, scriptContext);

}

RecyclableObject\* JavascriptOperators::GetIteratorFunction(Var iterable, ScriptContext\* scriptContext)

{

RecyclableObject\* iterableObj = RecyclableObject::FromVar(JavascriptOperators::ToObject(iterable, scriptContext));

return JavascriptOperators::GetIteratorFunction(iterableObj, scriptContext);

}

RecyclableObject\* JavascriptOperators::GetIteratorFunction(RecyclableObject\* instance, ScriptContext \* scriptContext)

{

Var func = JavascriptOperators::GetProperty(instance, PropertyIds::\_symbolIterator, scriptContext);

if (!JavascriptConversion::IsCallable(func))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction);

}

RecyclableObject\* function = RecyclableObject::FromVar(func);

return function;

}

RecyclableObject\* JavascriptOperators::GetIterator(RecyclableObject\* instance, ScriptContext \* scriptContext)

{

RecyclableObject\* function = GetIteratorFunction(instance, scriptContext);

Var iterator = function->GetEntryPoint()(function, CallInfo(Js::CallFlags\_Value, 1), instance);

if (!JavascriptOperators::IsObject(iterator))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

return RecyclableObject::FromVar(iterator);

}

// IteratorNext as described in ES6.0 (draft 22) Section 7.4.2

RecyclableObject\* JavascriptOperators::IteratorNext(RecyclableObject\* iterator, ScriptContext\* scriptContext, Var value)

{

Var func = JavascriptOperators::GetProperty(iterator, PropertyIds::next, scriptContext);

if (!JavascriptConversion::IsCallable(func))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedFunction);

}

RecyclableObject\* callable = RecyclableObject::FromVar(func);

Js::Var args[] = { iterator, value };

Js::CallInfo callInfo(Js::CallFlags\_Value, \_countof(args) + (value == nullptr ? -1 : 0));

Var result = JavascriptFunction::CallFunction<true>(callable, callable->GetEntryPoint(), Js::Arguments(callInfo, args));

if (!JavascriptOperators::IsObject(result))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedObject);

}

return RecyclableObject::FromVar(result);

}

// IteratorComplete as described in ES6.0 (draft 22) Section 7.4.3

bool JavascriptOperators::IteratorComplete(RecyclableObject\* iterResult, ScriptContext\* scriptContext)

{

Var done = JavascriptOperators::GetProperty(iterResult, Js::PropertyIds::done, scriptContext);

return JavascriptConversion::ToBool(done, scriptContext);

}

// IteratorValue as described in ES6.0 (draft 22) Section 7.4.4

Var JavascriptOperators::IteratorValue(RecyclableObject\* iterResult, ScriptContext\* scriptContext)

{

return JavascriptOperators::GetProperty(iterResult, Js::PropertyIds::value, scriptContext);

}

// IteratorStep as described in ES6.0 (draft 22) Section 7.4.5

bool JavascriptOperators::IteratorStep(RecyclableObject\* iterator, ScriptContext\* scriptContext, RecyclableObject\*\* result)

{

Assert(result);

\*result = JavascriptOperators::IteratorNext(iterator, scriptContext);

return !JavascriptOperators::IteratorComplete(\*result, scriptContext);

}

bool JavascriptOperators::IteratorStepAndValue(RecyclableObject\* iterator, ScriptContext\* scriptContext, Var\* resultValue)

{

RecyclableObject\* result = JavascriptOperators::IteratorNext(iterator, scriptContext);

if (!JavascriptOperators::IteratorComplete(result, scriptContext))

{

\*resultValue = JavascriptOperators::IteratorValue(result, scriptContext);

return true;

}

return false;

}

RecyclableObject\* JavascriptOperators::CreateFromConstructor(RecyclableObject\* constructor, ScriptContext\* scriptContext)

{

// Create a regular object and set the internal proto from the constructor

return JavascriptOperators::OrdinaryCreateFromConstructor(constructor, scriptContext->GetLibrary()->CreateObject(), nullptr, scriptContext);

}

RecyclableObject\* JavascriptOperators::OrdinaryCreateFromConstructor(RecyclableObject\* constructor, RecyclableObject\* obj, DynamicObject\* intrinsicProto, ScriptContext\* scriptContext)

{

// There isn't a good way for us to add internal properties to objects in Chakra.

// Thus, caller should take care to create obj with the correct internal properties.

Var proto = JavascriptOperators::GetProperty(constructor, Js::PropertyIds::prototype, scriptContext);

// If constructor.prototype is an object, we should use that as the [[Prototype]] for our obj.

// Else, we set the [[Prototype]] internal slot of obj to %intrinsicProto% - which should be the default.

if (JavascriptOperators::IsObjectType(JavascriptOperators::GetTypeId(proto)) &&

DynamicObject::FromVar(proto) != intrinsicProto)

{

JavascriptObject::ChangePrototype(obj, RecyclableObject::FromVar(proto), /\*validate\*/true, scriptContext);

}

return obj;

}

Var JavascriptOperators::GetProperty(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return JavascriptOperators::GetProperty(instance, instance, propertyId, requestContext, info);

}

BOOL JavascriptOperators::GetProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return JavascriptOperators::GetProperty(instance, instance, propertyId, value, requestContext, info);

}

Var JavascriptOperators::GetProperty(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

Var value;

if (JavascriptOperators::GetProperty(instance, propertyObject, propertyId, &value, requestContext, info))

{

return value;

}

return requestContext->GetMissingPropertyResult(propertyObject, propertyId);

}

Var JavascriptOperators::GetRootProperty(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

Var value;

if (JavascriptOperators::GetRootProperty(instance, propertyId, &value, requestContext, info))

{

return value;

}

return requestContext->GetMissingPropertyResult(instance, propertyId);

}

BOOL JavascriptOperators::GetPropertyReference(RecyclableObject \*instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info)

{

return JavascriptOperators::GetPropertyReference(instance, instance, propertyId, value, requestContext, info);

}

BOOL JavascriptOperators::GetItem(RecyclableObject\* instance, uint64 index, Var\* value, ScriptContext\* requestContext)

{

PropertyRecord const \* propertyRecord;

JavascriptOperators::GetPropertyIdForInt(index, requestContext, &propertyRecord);

return JavascriptOperators::GetProperty(instance, propertyRecord->GetPropertyId(), value, requestContext);

}

BOOL JavascriptOperators::GetItem(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext)

{

return JavascriptOperators::GetItem(instance, instance, index, value, requestContext);

}

BOOL JavascriptOperators::GetItemReference(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext)

{

return GetItemReference(instance, instance, index, value, requestContext);

}

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

if (propertyId == Js::PropertyIds::\_\_proto\_\_)

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyId, false, false>(instance, propertyId, setterValue, flags, info, scriptContext);

}

else

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyId, true, false>(instance, propertyId, setterValue, flags, info, scriptContext);

}

}

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritableRootProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

if (propertyId == Js::PropertyIds::\_\_proto\_\_)

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyId, false, true>(instance, propertyId, setterValue, flags, info, scriptContext);

}

else

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyId, true, true>(instance, propertyId, setterValue, flags, info, scriptContext);

}

}

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritableProperty(RecyclableObject\* instance, JavascriptString\* propertyNameString, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext)

{

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

if (Js::BuiltInPropertyRecords::\_\_proto\_\_.Equals(propertyName))

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<JavascriptString\*, false, false>(instance, propertyNameString, setterValue, flags, info, scriptContext);

}

else

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<JavascriptString\*, true, false>(instance, propertyNameString, setterValue, flags, info, scriptContext);

}

}

template<typename PropertyKeyType>

BOOL JavascriptOperators::CheckPrototypesForAccessorOrNonWritablePropertySlow(RecyclableObject\* instance, PropertyKeyType propertyKey, Var\* setterValue, DescriptorFlags\* flags, bool isRoot, ScriptContext\* scriptContext)

{

// This is used in debug verification, do not doFastProtoChainCheck to avoid side effect (doFastProtoChainCheck may update HasWritableDataOnly flags).

if (isRoot)

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyKeyType, /\*doFastProtoChainCheck\*/false, true>(instance, propertyKey, setterValue, flags, nullptr, scriptContext);

}

else

{

return CheckPrototypesForAccessorOrNonWritablePropertyCore<PropertyKeyType, /\*doFastProtoChainCheck\*/false, false>(instance, propertyKey, setterValue, flags, nullptr, scriptContext);

}

}

BOOL JavascriptOperators::SetProperty(Var instance, RecyclableObject\* object, PropertyId propertyId, Var newValue, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags)

{

PropertyValueInfo info;

return JavascriptOperators::SetProperty(instance, object, propertyId, newValue, &info, requestContext, propertyOperationFlags);

}

BOOL JavascriptOperators::TryConvertToUInt32(const wchar\_t\* str, int length, uint32\* intVal)

{

return NumberUtilities::TryConvertToUInt32(str, length, intVal);

}

template <typename TPropertyKey>

DescriptorFlags JavascriptOperators::GetRootSetter(RecyclableObject\* instance, TPropertyKey propertyKey, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

// This is provided only so that CheckPrototypesForAccessorOrNonWritablePropertyCore will compile.

// It will never be called.

Throw::FatalInternalError();

}

template <>

inline DescriptorFlags JavascriptOperators::GetRootSetter(RecyclableObject\* instance, PropertyId propertyId, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(JavascriptOperators::GetTypeId(instance) == TypeIds\_GlobalObject

|| JavascriptOperators::GetTypeId(instance) == TypeIds\_ModuleRoot,

"Root must be a global object!");

RootObjectBase\* rootObject = static\_cast<RootObjectBase\*>(instance);

return rootObject->GetRootSetter(propertyId, setterValue, info, requestContext);

}

// Helper to fetch @@species from a constructor object

Var JavascriptOperators::GetSpecies(RecyclableObject\* constructor, ScriptContext\* scriptContext)

{

if (scriptContext->GetConfig()->IsES6SpeciesEnabled())

{

Var species = nullptr;

// Let S be Get(C, @@species)

if (JavascriptOperators::GetProperty(constructor, PropertyIds::\_symbolSpecies, &species, scriptContext)

&& !JavascriptOperators::IsUndefinedOrNullType(JavascriptOperators::GetTypeId(species)))

{

// If S is neither undefined nor null, let C be S

return species;

}

}

return constructor;

}

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace IR

{

class LabelInstr;

}

enum JsNativeValueType;

namespace Js

{

struct ResumeYieldData;

#define DeclareExceptionPointer(ep) \

EXCEPTION\_RECORD ep##er; \

CONTEXT ep##c; \

EXCEPTION\_POINTERS ep = {&ep##er, &ep##c};

#define TYPEOF\_ERROR\_HANDLER\_CATCH(scriptContext, var) \

} \

catch (Js::JavascriptExceptionObject \*exceptionObject) \

{ \

Js::Var errorObject = exceptionObject->GetThrownObject(nullptr); \

if (errorObject != nullptr && Js::JavascriptError::Is(errorObject)) \

{ \

HRESULT hr = Js::JavascriptError::GetRuntimeError(Js::RecyclableObject::FromVar(errorObject), nullptr); \

if (JavascriptError::GetErrorNumberFromResourceID(JSERR\_Property\_CannotGet\_NullOrUndefined) == (long)hr \

|| JavascriptError::GetErrorNumberFromResourceID(JSERR\_UseBeforeDeclaration) == (long)hr) \

{ \

if (scriptContext->IsInDebugMode()) \

{ \

JavascriptExceptionOperators::ThrowExceptionObject(exceptionObject, scriptContext, true); \

} \

else \

{ \

throw exceptionObject; \

} \

} \

} \

var = scriptContext->GetLibrary()->GetUndefined();

#define TYPEOF\_ERROR\_HANDLER\_THROW(scriptContext, var) \

} \

if (scriptContext->IsUndeclBlockVar(var)) \

{ \

Assert(scriptContext->GetConfig()->IsLetAndConstEnabled()); \

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration); \

}

#define BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext) \

try { \

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext); \

class AutoCleanup \

{ \

private: \

ScriptContext \*const scriptContext; \

public: \

AutoCleanup(ScriptContext \*const scriptContext) : scriptContext(scriptContext) \

{ \

if (scriptContext->IsInDebugMode()) \

{ \

scriptContext->GetDebugContext()->GetProbeContainer()->SetThrowIsInternal(true); \

} \

} \

~AutoCleanup() \

{ \

if (scriptContext->IsInDebugMode()) \

{ \

scriptContext->GetDebugContext()->GetProbeContainer()->SetThrowIsInternal(false); \

} \

} \

} autoCleanup(scriptContext);

#define END\_TYPEOF\_ERROR\_HANDLER(scriptContext, var) \

TYPEOF\_ERROR\_HANDLER\_CATCH(scriptContext, var) \

TYPEOF\_ERROR\_HANDLER\_THROW(scriptContext, var)

#define BEGIN\_PROFILED\_TYPEOF\_ERROR\_HANDLER(scriptContext) \

BEGIN\_TYPEOF\_ERROR\_HANDLER(scriptContext)

#define END\_PROFILED\_TYPEOF\_ERROR\_HANDLER(scriptContext, var, functionBody, inlineCacheIndex) \

TYPEOF\_ERROR\_HANDLER\_CATCH(scriptContext, var) \

functionBody->GetDynamicProfileInfo()->RecordFieldAccess(functionBody, inlineCacheIndex, var, FldInfo\_NoInfo); \

TYPEOF\_ERROR\_HANDLER\_THROW(scriptContext, var)

class JavascriptOperators /\* All static \*/

{

// Methods

public:

static void FreeTemp(Var aValue);

static BOOL IsArray(Var instanceVar);

static BOOL IsConstructor(Var instanceVar);

static BOOL IsConcatSpreadable(Var instanceVar);

static Var ToObject(Var aRight,ScriptContext\* scriptContext);

static Var ToWithObject(Var aRight, ScriptContext\* scriptContext);

static Var OP\_LdCustomSpreadIteratorList(Var aRight, ScriptContext\* scriptContext);

static Var ToNumber(Var aRight,ScriptContext\* scriptContext);

static Var ToNumberInPlace(Var aRight,ScriptContext\* scriptContext, JavascriptNumber\* result);

#ifdef \_M\_IX86

static Var Int32ToVar(int32 value, ScriptContext\* scriptContext);

static Var Int32ToVarInPlace(int32 value, ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var UInt32ToVar(uint32 value, ScriptContext\* scriptContext);

static Var UInt32ToVarInPlace(uint32 value, ScriptContext\* scriptContext, JavascriptNumber \*result);

#endif

static Var OP\_FinishOddDivBy2(uint32 value, ScriptContext \*scriptContext);

static Var OP\_ApplyArgs(Var func,Var instance,\_\_in\_xcount(8)void\*\* stackPtr,CallInfo callInfo,ScriptContext\* scriptContext);

static Var Typeof(Var var, ScriptContext\* scriptContext);

static Var TypeofFld(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static Var TypeofRootFld(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static Var TypeofElem(Var instance, Var index, ScriptContext\* scriptContext);

static Var TypeofElem\_UInt32(Var instance, uint32 index, ScriptContext\* scriptContext);

static Var TypeofElem\_Int32(Var instance, int32 index, ScriptContext\* scriptContext);

static Var Delete(Var var, ScriptContext\* scriptContext);

static JavascriptString \* Concat3(Var aLeft, Var aCenter, Var aRight, ScriptContext \* scriptContext);

static JavascriptString \* NewConcatStrMulti(Var a1, Var a2, uint count, ScriptContext \* scriptContext);

static void SetConcatStrMultiItem(Var concatStr, Var str, uint index, ScriptContext \* scriptContext);

static void SetConcatStrMultiItem2(Var concatStr, Var str1, Var str2, uint index, ScriptContext \* scriptContext);

static BOOL Equal(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Equal\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Greater(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Greater\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL GreaterEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL GreaterEqual\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Less(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL Less\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL LessEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL LessEqual\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL NotEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL NotEqual\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL StrictEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL StrictEqualString(Var aLeft, Var aRight);

static BOOL StrictEqualEmptyString(Var aLeft);

static BOOL NotStrictEqual(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL HasOwnProperty(Var instance, PropertyId propertyId, ScriptContext \* requestContext);

static BOOL GetOwnProperty(Var instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext);

static BOOL GetOwnAccessors(Var instance, PropertyId propertyId, Var\* getter, Var\* setter, ScriptContext \* requestContext);

static BOOL EnsureProperty(Var instance, PropertyId propertyId);

static void OP\_EnsureNoRootProperty(Var instance, PropertyId propertyId);

static void OP\_EnsureNoRootRedeclProperty(Var instance, PropertyId propertyId);

static void OP\_ScopedEnsureNoRedeclProperty(FrameDisplay \*pDisplay, PropertyId propertyId, Var instanceDefault);

static Var GetOwnPropertyNames(Var instance, ScriptContext \*scriptContext);

static Var GetOwnPropertySymbols(Var instance, ScriptContext \*scriptContext);

static Var GetOwnPropertyKeys(Var instance, ScriptContext \*scriptContext);

static Var GetOwnEnumerablePropertyNames(Var instance, ScriptContext \*scriptContext);

static Var GetOwnEnumerablePropertyNamesSymbols(Var instance, ScriptContext \*scriptContext);

static BOOL GetOwnPropertyDescriptor(RecyclableObject\* obj, PropertyId propertyId, ScriptContext\* scriptContext, PropertyDescriptor\* propertyDescriptor);

static BOOL GetOwnPropertyDescriptor(RecyclableObject\* obj, JavascriptString\* propertyKey, ScriptContext\* scriptContext, PropertyDescriptor\* propertyDescriptor);

static BOOL IsPropertyUnscopable (Var instanceVar, PropertyId propertyId);

static BOOL IsPropertyUnscopable (Var instanceVar, JavascriptString \*propertyString);

template<bool unscopables>

static BOOL HasProperty\_Impl(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasPropertyUnscopables(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasProperty(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasRootProperty(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasProxyOrPrototypeInlineCacheProperty(RecyclableObject\* instance, PropertyId propertyId);

static BOOL HasProxyInPrototypeChain(RecyclableObject\* instance);

template<typename PropertyKeyType>

static BOOL GetPropertyWPCache(Var instance, RecyclableObject\* propertyObject, PropertyKeyType propertyKey, Var\* value, ScriptContext\* requestContext, PropertyString \* propertyString);

static BOOL GetPropertyUnscopable(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info=NULL);

static Var GetProperty(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static Var GetProperty(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetProperty(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetPropertyObject(Var instance, ScriptContext \* scriptContext, RecyclableObject\*\* propertyObject);

static BOOL GetRootProperty(Var instance, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static Var GetRootProperty(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static Var GetPropertyReference(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext);

static BOOL GetPropertyReference(RecyclableObject\* instance, PropertyId propertyId, Var\* value,ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetPropertyReference(Var instance, RecyclableObject\* propertyObject, PropertyId propertyId, Var\* value,ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

static BOOL GetRootPropertyReference(RecyclableObject\* instance, PropertyId propertyId, Var\* value,ScriptContext\* requestContext, PropertyValueInfo\* info = NULL);

template<typename PropertyKeyType>

static BOOL SetPropertyWPCache(Var instance, RecyclableObject\* object, PropertyKeyType propertyKey, Var newValue, ScriptContext\* requestContext, PropertyString \* propertyString, PropertyOperationFlags flags);

static BOOL SetPropertyUnscopable(Var instance, RecyclableObject\* receiver, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetProperty(Var instance, RecyclableObject\* object, PropertyId propertyId, Var newValue, ScriptContext\* requestContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetProperty(Var instance, RecyclableObject\* receiver, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetRootProperty(RecyclableObject\* instance, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL GetAccessors(RecyclableObject\* instance, PropertyId propertyId, ScriptContext\* requestContext, Var\* getter, Var\* setter);

static BOOL SetAccessors(RecyclableObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL InitProperty(RecyclableObject\* instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL DeleteProperty(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static BOOL DeletePropertyUnscopables(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

template<bool unscopables>

static BOOL DeleteProperty\_Impl(RecyclableObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static TypeId GetTypeId(Var instance);

static BOOL IsObject(Var instance);

static BOOL IsExposedType(TypeId typeId);

static BOOL IsObjectType(TypeId typeId);

static BOOL IsObjectOrNull(Var instance);

static BOOL IsUndefinedOrNullType(TypeId);

static BOOL IsSpecialObjectType(TypeId typeId);

static BOOL IsJsNativeObject(Var instance);

static BOOL IsUndefinedObject(Var instance);

static BOOL IsUndefinedObject(Var instance, ScriptContext \*scriptContext);

static BOOL IsUndefinedObject(Var instance, RecyclableObject \*libraryUndefined);

static BOOL IsUndefinedObject(Var isntance, JavascriptLibrary\* library);

static BOOL IsAnyNumberValue(Var instance);

static BOOL IsIterable(RecyclableObject\* instance, ScriptContext\* scriptContext);

static BOOL IsClassConstructor(Var instance);

static BOOL HasOwnItem(RecyclableObject\* instance, uint32 index);

static BOOL HasItem(RecyclableObject\* instance, uint32 index);

static BOOL HasItem(RecyclableObject\* instance, uint64 index);

static BOOL GetOwnItem(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItem(RecyclableObject\* instance, uint64 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItem(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItem(Var instance, RecyclableObject\* propertyObject, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItemReference(RecyclableObject\* instance, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL GetItemReference(Var instance, RecyclableObject\* propertyObject, uint32 index, Var\* value, ScriptContext\* requestContext);

static BOOL SetItem(Var instance, RecyclableObject\* object, uint64 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetItem(Var instance, RecyclableObject\* object, uint32 index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None, BOOL skipPrototypeCheck = FALSE);

static BOOL DeleteItem(RecyclableObject\* instance, uint32 index, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static BOOL DeleteItem(RecyclableObject\* instance, uint64 index, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var Construct(RecyclableObject\* constructor, const Arguments args, ScriptContext\* scriptContext);

static RecyclableObject\* CreateFromConstructor(RecyclableObject\* constructor, ScriptContext\* scriptContext);

static RecyclableObject\* OrdinaryCreateFromConstructor(RecyclableObject\* constructor, RecyclableObject\* obj, DynamicObject\* intrinsicProto, ScriptContext\* scriptContext);

template<typename PropertyKeyType>

static BOOL CheckPrototypesForAccessorOrNonWritablePropertySlow(RecyclableObject\* instance, PropertyKeyType propertyKey, Var\* setterValueOrProxy, DescriptorFlags\* flags, bool isRoot, ScriptContext\* scriptContext);

static BOOL CheckPrototypesForAccessorOrNonWritableProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValueOrProxy, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static BOOL CheckPrototypesForAccessorOrNonWritableProperty(RecyclableObject\* instance, JavascriptString\* propertyNameString, Var\* setterValueOrProxy, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static BOOL CheckPrototypesForAccessorOrNonWritableRootProperty(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValueOrProxy, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static BOOL CheckPrototypesForAccessorOrNonWritableItem(RecyclableObject\* instance, uint32 index, Var\* setterValueOrProxy, DescriptorFlags\* flags, ScriptContext\* scriptContext, BOOL skipPrototypeCheck = FALSE);

template <typename PropertyKeyType, bool unscopable>

static DescriptorFlags GetterSetter\_Impl(RecyclableObject\* instance, PropertyKeyType propertyKey, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static DescriptorFlags GetterSetterUnscopable(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static DescriptorFlags GetterSetter(RecyclableObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static DescriptorFlags GetterSetter(RecyclableObject\* instance, JavascriptString \* propertyName, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static void OP\_InvalidateProtoCaches(PropertyId propertyId, ScriptContext \*scriptContext);

static BOOL SetGlobalPropertyNoHost(wchar\_t const \* propertyName, charcount\_t propertyLength, Var value, ScriptContext \* scriptContext);

static RecyclableObject\* GetPrototype(RecyclableObject\* instance);

static RecyclableObject\* OP\_GetPrototype(Var instance, ScriptContext\* scriptContext);

static BOOL OP\_HasProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static BOOL OP\_HasOwnProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static BOOL HasOwnPropertyNoHostObject(Var instance, PropertyId propertyId);

static BOOL HasOwnPropertyNoHostObjectForHeapEnum(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, Var& getter, Var& setter);

static Var GetOwnPropertyNoHostObjectForHeapEnum(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, Var& getter, Var &setter);

static BOOL OP\_HasOwnPropScoped(Var instance, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext);

static Var OP\_GetProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static Var OP\_GetRootProperty(Var instance, PropertyId propertyId, PropertyValueInfo \* info, ScriptContext\* scriptContext);

static BOOL OP\_SetProperty(Var instance, PropertyId propertyId, Var newValue, ScriptContext\* scriptContext, PropertyValueInfo \* info = nullptr, PropertyOperationFlags flags = PropertyOperation\_None, Var thisInstance = nullptr);

static BOOL SetPropertyOnTaggedNumber(Var instance, RecyclableObject\* object, PropertyId propertyId, Var newValue, ScriptContext\* requestContext, PropertyOperationFlags flags);

static BOOL SetItemOnTaggedNumber(Var instance, RecyclableObject\* object, uint32 index, Var newValue, ScriptContext\* requestContext, PropertyOperationFlags propertyOperationFlags);

static BOOL OP\_StFunctionExpression(Var instance, PropertyId propertyId, Var newValue);

static BOOL OP\_InitProperty(Var instance, PropertyId propertyId, Var newValue);

static Var OP\_DeleteProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var OP\_DeleteRootProperty(Var instance, PropertyId propertyId, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static BOOL OP\_InitLetProperty(Var instance, PropertyId propertyId, Var newValue);

static BOOL OP\_InitConstProperty(Var instance, PropertyId propertyId, Var newValue);

static BOOL OP\_InitUndeclRootLetProperty(Var instance, PropertyId propertyId);

static BOOL OP\_InitUndeclRootConstProperty(Var instance, PropertyId propertyId);

static BOOL OP\_InitUndeclConsoleLetProperty(Var instance, PropertyId propertyId);

static BOOL OP\_InitUndeclConsoleConstProperty(Var instance, PropertyId propertyId);

static BOOL OP\_InitClassMember(Var instance, PropertyId propertyId, Var newValue);

static void OP\_InitClassMemberComputedName(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitClassMemberGet(Var object, PropertyId propertyId, Var getter);

static void OP\_InitClassMemberGetComputedName(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitClassMemberSet(Var object, PropertyId propertyId, Var setter);

static void OP\_InitClassMemberSetComputedName(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static Js::PropertyId GetPropertyId(Var propertyName, ScriptContext\* scriptContext);

static BOOL OP\_HasItem(Var instance, Var aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI(Var instance, Var aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_JIT(Var instance, Var index, ScriptContext \*scriptContext);

#if ENABLE\_NATIVE\_CODEGEN

static Var OP\_GetElementI\_JIT\_ExpectingNativeFloatArray(Var instance, Var index, ScriptContext \*scriptContext);

static Var OP\_GetElementI\_JIT\_ExpectingVarArray(Var instance, Var index, ScriptContext \*scriptContext);

#endif

static Var OP\_GetElementI\_UInt32(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_UInt32\_ExpectingNativeFloatArray(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_UInt32\_ExpectingVarArray(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_Int32(Var instance, int32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_Int32\_ExpectingNativeFloatArray(Var instance, int32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetElementI\_Int32\_ExpectingVarArray(Var instance, int32 aElementIndex, ScriptContext\* scriptContext);

static Var GetElementIHelper(Var instance, Var index, Var receiver, ScriptContext\* scriptContext);

static int32 OP\_GetNativeIntElementI(Var instance, Var index);

static int32 OP\_GetNativeIntElementI\_Int32(Var instance, int32 index, ScriptContext \*scriptContext);

static int32 OP\_GetNativeIntElementI\_UInt32(Var instance, uint32 index, ScriptContext \*scriptContext);

static double OP\_GetNativeFloatElementI(Var instance, Var index);

static double OP\_GetNativeFloatElementI\_Int32(Var instance, int32 index, ScriptContext \*scriptContext);

static double OP\_GetNativeFloatElementI\_UInt32(Var instance, uint32 index, ScriptContext \*scriptContext);

static Var OP\_GetMethodElement(Var instance, Var aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetMethodElement\_UInt32(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext);

static Var OP\_GetMethodElement\_Int32(Var instance, int32 aElementIndex, ScriptContext\* scriptContext);

static BOOL OP\_SetElementI(Var instance, Var aElementIndex, Var aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetElementI\_JIT(Var instance, Var aElementIndex, Var aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetElementI\_UInt32(Var instance, uint32 aElementIndex, Var aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetElementI\_Int32(Var instance, int aElementIndex, Var aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL SetElementIHelper(Var receiver, RecyclableObject\* object, Var index, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags);

static BOOL OP\_SetNativeIntElementI(Var instance, Var aElementIndex, int32 aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetNativeIntElementI\_UInt32(Var instance, uint32 aElementIndex, int32 aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetNativeIntElementI\_Int32(Var instance, int aElementIndex, int32 aValue, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static BOOL OP\_SetNativeFloatElementI(Var instance, Var aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags flags, double value);

static BOOL OP\_SetNativeFloatElementI\_UInt32(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags flags, double value);

static BOOL OP\_SetNativeFloatElementI\_Int32(Var instance, int aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags flags, double value);

static Var OP\_DeleteElementI(Var instance, Var aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var OP\_DeleteElementI\_UInt32(Var instance, uint32 aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var OP\_DeleteElementI\_Int32(Var instance, int aElementIndex, ScriptContext\* scriptContext, PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static BOOL OP\_Memset(Var instance, int32 start, Var value, int32 length, ScriptContext\* scriptContext);

static BOOL OP\_Memcopy(Var dstInstance, int32 dstStart, Var srcInstance, int32 srcStart, int32 length, ScriptContext\* scriptContext);

static Var OP\_GetLength(Var instance, ScriptContext\* scriptContext);

static Var OP\_GetThis(Var thisVar, int moduleID, ScriptContext\* scriptContext);

static Var OP\_GetThisNoFastPath(Var thisVar, int moduleID, ScriptContext\* scriptContext);

static Var OP\_StrictGetThis(Var thisVar, ScriptContext\* scriptContext);

static bool IsThisSelf(TypeId typeId);

static Var GetThisHelper(Var thisVar, TypeId typeId, int moduleID, ScriptContext \*scriptContext);

static Var GetThisFromModuleRoot(Var thisVar);

static Var OP\_GetThisScoped(FrameDisplay \*pScope, Var defaultInstance, ScriptContext\* scriptContext);

static Var OP\_UnwrapWithObj(Var aValue);

static Var OP\_GetInstanceScoped(FrameDisplay \*pScope, PropertyId propertyId, Var rootObject, Var\* result2, ScriptContext\* scriptContext);

static BOOL OP\_InitPropertyScoped(FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, ScriptContext\* scriptContext);

static BOOL OP\_InitFuncScoped(FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, ScriptContext\* scriptContext);

static Var OP\_DeletePropertyScoped(

FrameDisplay \*pScope,

PropertyId propertyId,

Var defaultInstance,

ScriptContext\* scriptContext,

PropertyOperationFlags propertyOperationFlags = PropertyOperation\_None);

static Var OP\_TypeofPropertyScoped(FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext);

static void OP\_InitGetter(Var object, PropertyId propertyId, Var getter);

static Js::PropertyId OP\_InitElemGetter(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitSetter(Var object, PropertyId propertyId, Var setter);

static Js::PropertyId OP\_InitElemSetter(Var object, Var elementName, Var getter, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitComputedProperty(Var object, Var elementName, Var value, ScriptContext\* scriptContext, PropertyOperationFlags flags = PropertyOperation\_None);

static void OP\_InitProto(Var object, PropertyId propertyId, Var value);

static ForInObjectEnumerator \* OP\_GetForInEnumerator(Var enumerable, ScriptContext\* scriptContext);

static void OP\_ReleaseForInEnumerator(ForInObjectEnumerator \* enumerator, ScriptContext\* scriptContext);

static Var OP\_BrOnEmpty(ForInObjectEnumerator \* enumerator);

static BOOL OP\_BrHasSideEffects(int se,ScriptContext\* scriptContext);

static BOOL OP\_BrNotHasSideEffects(int se,ScriptContext\* scriptContext);

static BOOL OP\_BrFncEqApply(Var instance,ScriptContext\* scriptContext);

static BOOL OP\_BrFncNeqApply(Var instance,ScriptContext\* scriptContext);

static Var OP\_CmEq\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmNeq\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmSrEq\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmSrEq\_String(Var a, Var b, ScriptContext \*scriptContext);

static Var OP\_CmSrEq\_EmptyString(Var a, ScriptContext \*scriptContext);

static Var OP\_CmSrNeq\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmLt\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmLe\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmGt\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static Var OP\_CmGe\_A(Js::Var a,Js::Var b,ScriptContext\* scriptContext);

static FunctionInfo \* JavascriptOperators::GetConstructorFunctionInfo(Var instance, ScriptContext \* scriptContext);

// Detach the type array buffer, if possible, and returns the state of the object which can be used to initialize another object

static DetachedStateBase\* DetachVarAndGetState(Var var);

static bool IsObjectDetached(Var var);

// This will return a new object from the state returned by the above operation

static Var NewVarFromDetachedState(DetachedStateBase\* state, JavascriptLibrary \*library);

static Var NewScObjectLiteral(ScriptContext\* scriptContext, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType);

static DynamicType \* EnsureObjectLiteralType(ScriptContext\* scriptContext, const Js::PropertyIdArray \*propIds, DynamicType \*\* literalType);

static uint GetLiteralSlotCapacity(Js::PropertyIdArray const \* propIds, ScriptContext \*const scriptContext);

static uint GetLiteralInlineSlotCapacity(Js::PropertyIdArray const \* propIds, ScriptContext \*const scriptContext);

static Var NewJavascriptObjectNoArg(ScriptContext\* requestContext);

static Var NewJavascriptArrayNoArg(ScriptContext\* requestContext);

static Var NewScObjectNoCtorCommon(Var instance, ScriptContext\* requestContext, bool isBaseClassConstructorNewScObject = false);

static Var NewScObjectNoCtor(Var instance, ScriptContext\* requestContext);

static Var NewScObjectNoCtorFull(Var instance, ScriptContext\* requestContext);

static Var NewScObjectNoArgNoCtorCommon(Var instance, ScriptContext\* requestContext, bool isBaseClassConstructorNewScObject = false);

static Var NewScObjectNoArgNoCtor(Var instance, ScriptContext\* requestContext);

static Var NewScObjectNoArgNoCtorFull(Var instance, ScriptContext\* requestContext);

static Var NewScObjectNoArg(Var instance, ScriptContext\* requestContext);

static Var NewScObject(const Var callee, const Arguments args, ScriptContext \*const scriptContext, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

static Var AddVarsToArraySegment(SparseArraySegment<Var> \* segment, const Js::VarArray \*vars);

static void AddIntsToArraySegment(SparseArraySegment<int32> \* segment, const Js::AuxArray<int32> \*ints);

static void AddFloatsToArraySegment(SparseArraySegment<double> \* segment, const Js::AuxArray<double> \*doubles);

static void UpdateNewScObjectCache(Var function, Var instance, ScriptContext\* requestContext);

static RecyclableObject\* GetIteratorFunction(Var iterable, ScriptContext\* scriptContext);

static RecyclableObject\* GetIteratorFunction(RecyclableObject\* instance, ScriptContext \* scriptContext);

static RecyclableObject\* GetIterator(Var instance, ScriptContext\* scriptContext);

static RecyclableObject\* GetIterator(RecyclableObject\* instance, ScriptContext\* scriptContext);

static RecyclableObject\* IteratorNext(RecyclableObject\* iterator, ScriptContext\* scriptContext, Var value = nullptr);

static bool IteratorComplete(RecyclableObject\* iterResult, ScriptContext\* scriptContext);

static Var IteratorValue(RecyclableObject\* iterResult, ScriptContext\* scriptContext);

static bool IteratorStep(RecyclableObject\* iterator, ScriptContext\* scriptContext, RecyclableObject\*\* result);

static bool IteratorStepAndValue(RecyclableObject\* iterator, ScriptContext\* scriptContext, Var\* resultValue);

static void TraceUseConstructorCache(const ConstructorCache\* ctorCache, const JavascriptFunction\* ctor, bool isHit);

static void TraceUpdateConstructorCache(const ConstructorCache\* ctorCache, const FunctionBody\* ctorBody, bool updated, const wchar\_t\* reason);

static Var ConvertToUnmappedArguments(HeapArgumentsObject \*argumentsObject, uint32 paramCount, Var \*paramAddr, DynamicObject\* frameObject, Js::PropertyIdArray \*propIds, uint32 formalsCount, ScriptContext\* scriptContext);

static Js::GlobalObject \* OP\_LdRoot(ScriptContext\* scriptContext);

static Js::ModuleRoot \* GetModuleRoot(int moduleID, ScriptContext\* scriptContext);

static Js::Var OP\_LoadModuleRoot(int moduleID, ScriptContext\* scriptContext);

static Var OP\_LdNull(ScriptContext\* scriptContext);

static Var OP\_LdUndef(ScriptContext\* scriptContext);

static Var OP\_LdNaN(ScriptContext\* scriptContext);

static Var OP\_LdInfinity(ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdHandlerScope(Var argThis, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdStrictFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdStrictFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdInnerFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdStrictInnerFrameDisplay(void \*argHead, void \*argEnv, ScriptContext\* scriptContext);

static FrameDisplay\* OP\_LdStrictInnerFrameDisplayNoParent(void \*argHead, ScriptContext\* scriptContext);

static void CheckInnerFrameDisplayArgument(void \*argHead);

static Var LoadHeapArguments(JavascriptFunction \*funcCallee, unsigned int count, Var \*pParams, Var frameObj, Var vArray, ScriptContext\* scriptContext, bool nonSimpleParamList);

static Var LoadHeapArgsCached(JavascriptFunction \*funcCallee, uint32 actualsCount, uint32 formalsCount, Var \*pParams, Var frameObj, ScriptContext\* scriptContext, bool nonSimpleParamList);

static HeapArgumentsObject \*CreateHeapArguments(JavascriptFunction \*funcCallee, uint32 actualsCount, uint32 formalsCount, Var frameObj, ScriptContext\* scriptContext);

static Var OP\_InitCachedScope(Var varFunc, const PropertyIdArray \*propIds, DynamicType \*\* literalType, bool formalsAreLetDecls, ScriptContext \*scriptContext);

static void OP\_InvalidateCachedScope(Var varEnv, int32 envIndex);

static void OP\_InitCachedFuncs(Var varScope, FrameDisplay \*pDisplay, const FuncInfoArray \*info, ScriptContext \*scriptContext);

static Var OP\_NewScopeObject(ScriptContext\*scriptContext);

static Var\* OP\_NewScopeSlots(unsigned int count, ScriptContext \*scriptContext, Var scope);

static Var\* OP\_NewScopeSlotsWithoutPropIds(unsigned int count, int index, ScriptContext \*scriptContext, FunctionBody \*functionBody);

static Var\* OP\_CloneScopeSlots(Var \*scopeSlots, ScriptContext \*scriptContext);

static Var OP\_NewPseudoScope(ScriptContext \*scriptContext);

static Var OP\_NewBlockScope(ScriptContext \*scriptContext);

static Var OP\_CloneBlockScope(BlockActivationObject \*blockScope, ScriptContext \*scriptContext);

static void OP\_InitClass(Var constructor, Var extends, ScriptContext \* scriptContext);

static void OP\_LoadUndefinedToElement(Var instance, PropertyId propertyId);

static void OP\_LoadUndefinedToElementDynamic(Var instance, PropertyId propertyId, ScriptContext\* scriptContext);

static void OP\_LoadUndefinedToElementScoped(FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance, ScriptContext\* scriptContext);

static Var OP\_IsInst(Var instance, Var aClass, ScriptContext\* scriptContext, IsInstInlineCache \*inlineCache);

static Var IsIn(Var argProperty, Var instance, ScriptContext\* scriptContext);

static BOOL GetRemoteTypeId(Var instance, TypeId\* typeId);

static FunctionProxy\* GetDeferredDeserializedFunctionProxy(JavascriptFunction\* func);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetValueWithThisPtr(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetValueForTypeOf(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetValueUsingSpecifiedInlineCache(InlineCache \* inlineCache, Var instance, RecyclableObject \* object, PropertyId propertyId, ScriptContext\* scriptContext);

static Var PatchGetValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetValueWithThisPtrNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var thisInstance);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetRootValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetRootValueForTypeOf(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

static Var PatchGetRootValueNoFastPath\_Var(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetRootValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetPropertyScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance);

template <bool IsFromFullJit, class TInlineCache> static void PatchSetPropertyScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pScope, PropertyId propertyId, Var newValue, Var defaultInstance, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetPropertyForTypeOfScoped(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FrameDisplay \*pScope, PropertyId propertyId, Var defaultInstance);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutValueWithThisPtr(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutRootValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutValueNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutValueWithThisPtrNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchPutRootValueNoLocalFastPath(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

static void PatchPutValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

static void PatchPutValueWithThisPtrNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, Var thisInstance, PropertyOperationFlags flags = PropertyOperation\_None);

static void PatchPutRootValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var obj, PropertyId propertyId, Var newValue, PropertyOperationFlags flags = PropertyOperation\_None);

template <bool IsFromFullJit, class TInlineCache> static void PatchInitValue(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

static void PatchInitValueNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, RecyclableObject\* object, PropertyId propertyId, Var newValue);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchGetRootMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

template <bool IsFromFullJit, class TInlineCache> static Var PatchScopedGetMethod(FunctionBody \*const functionBody, TInlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetMethodNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetRootMethodNoFastPath\_Var(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, Var instance, PropertyId propertyId);

static Var PatchGetRootMethodNoFastPath(FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, DynamicObject\* object, PropertyId propertyId);

static Var PatchGetMethodFromObject(Var instance, RecyclableObject \* propertyObject, PropertyId propertyId, PropertyValueInfo \* info, ScriptContext \* scriptContext, bool isRootLd);

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

static void TracePropertyEquivalenceCheck(const JitEquivalentTypeGuard\* guard, const Type\* type, const Type\* refType, bool isEquivalent, uint failedPropertyIndex);

#endif

static bool IsStaticTypeObjTypeSpecEquivalent(const TypeEquivalenceRecord& equivalenceRecord, uint& failedIndex);

static bool IsStaticTypeObjTypeSpecEquivalent(const EquivalentPropertyEntry \*entry);

static bool CheckIfTypeIsEquivalent(Type\* type, JitEquivalentTypeGuard\* guard);

static void GetPropertyIdForInt(uint64 value, ScriptContext\* scriptContext, PropertyRecord const \*\* propertyRecord);

static void GetPropertyIdForInt(uint32 value, ScriptContext\* scriptContext, PropertyRecord const \*\* propertyRecord);

static BOOL TryConvertToUInt32(const wchar\_t\* str, int length, uint32\* value);

static BOOL ToPropertyDescriptor(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext);

static Var FromPropertyDescriptor(PropertyDescriptor descriptor, ScriptContext\* scriptContext);

static void CompletePropertyDescriptor(PropertyDescriptor\* resultDescriptor, PropertyDescriptor\* likePropertyDescriptor, ScriptContext\* requestContext);

static BOOL SetPropertyDescriptor(RecyclableObject\* object, PropertyId propId, PropertyDescriptor descriptor);

static BOOL DefineOwnPropertyDescriptor(RecyclableObject\* object, PropertyId propId, const PropertyDescriptor& descriptor, bool throwOnError, ScriptContext\* scriptContext);

static BOOL DefineOwnPropertyForArray(JavascriptArray\* arr, PropertyId propId, const PropertyDescriptor& descriptor, bool throwOnError, ScriptContext\* scriptContext);

static BOOL IsCompatiblePropertyDescriptor(const PropertyDescriptor& descriptor, PropertyDescriptor\* currentDescriptor, bool isExtensible, bool throwOnError, ScriptContext\* scriptContext);

template <bool needToSetProperty>

static BOOL ValidateAndApplyPropertyDescriptor(RecyclableObject\* obj, PropertyId propId, const PropertyDescriptor& descriptor,

PropertyDescriptor\* currentPropertyDescriptor, bool isExtensible, bool throwOnError, ScriptContext\* scriptContext);

template <bool isAccessor>

static PropertyDescriptor FillMissingPropertyDescriptorFields(PropertyDescriptor descriptor, ScriptContext\* scriptContext);

static Var OP\_InvokePut(Js::ScriptContext \*scriptContext, Var function, CallInfo callInfo, ...);

static Var DefaultAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static bool IsUndefinedAccessor(Var accessor, ScriptContext\* scriptContext);

static void SetAttributes(RecyclableObject\* object, PropertyId propId, const PropertyDescriptor& descriptor, bool force);

static void OP\_ClearAttributes(Var instance, PropertyId propertyId);

static void OP\_Freeze(Var instance);

static Var RootToThisObject(const Var object, ScriptContext \* const scriptContext);

static Var CallGetter(RecyclableObject \* const function, Var const object, ScriptContext \* const scriptContext);

static void CallSetter(RecyclableObject \* const function, Var const object, Var const value, ScriptContext \* const scriptContext);

static bool CheckIfObjectAndPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* object);

static bool CheckIfPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* prototype);

static bool DoCheckIfPrototypeChainHasOnlyWritableDataProperties(RecyclableObject\* prototype);

static void OP\_SetComputedNameVar(Var method, Var computedNameVar);

static void OP\_SetHomeObj(Var method, Var homeObj);

static Var OP\_LdSuper(Var scriptFunction, ScriptContext \* scriptContext);

static Var OP\_LdSuperCtor(Var scriptFunction, ScriptContext \* scriptContext);

static Var OP\_ScopedLdSuper(Var scriptFunction, ScriptContext \* scriptContext);

static Var OP\_ScopedLdSuperCtor(Var scriptFunction, ScriptContext \* scriptContext);

static Var ScopedLdSuperHelper(Var scriptFunction, Js::PropertyId propertyId, ScriptContext \* scriptContext);

static Var OP\_ResumeYield(ResumeYieldData\* yieldData, RecyclableObject\* iterator);

static Var OP\_AsyncSpawn(Js::Var aGenerator, Js::Var aThis, ScriptContext\* scriptContext);

template <typename T>

static void \* JitRecyclerAlloc(size\_t size, Recycler\* recycler)

{

TRACK\_ALLOC\_INFO(recycler, T, Recycler, size - sizeof(T), (size\_t)-1);

return recycler->AllocZero(size);

}

static void \* AllocMemForVarArray(size\_t size, Recycler\* recycler);

static void \* AllocUninitializedNumber(RecyclerJavascriptNumberAllocator \* allocator);

static void ScriptAbort();

class EntryInfo

{

public:

static FunctionInfo DefaultAccessor;

};

template <BOOL stopAtProxy, class Func>

static void MapObjectAndPrototypes(RecyclableObject\* object, Func func);

template <BOOL stopAtProxy, class Func>

static bool MapObjectAndPrototypesUntil(RecyclableObject\* object, Func func);

#if ENABLE\_PROFILE\_INFO

static void UpdateNativeArrayProfileInfoToCreateVarArray(Var instance, const bool expectingNativeFloatArray, const bool expectingVarArray);

static bool SetElementMayHaveImplicitCalls(ScriptContext \*const scriptContext);

#endif

static RecyclableObject \*GetCallableObjectOrThrow(const Var callee, ScriptContext \*const scriptContext);

static Js::Var BoxStackInstance(Js::Var value, ScriptContext \* scriptContext, bool allowStackFunction = false);

static BOOL PropertyReferenceWalkUnscopable(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

static BOOL PropertyReferenceWalk(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

static void VarToNativeArray(Var arrayObject,

JsNativeValueType valueType,

\_\_in UINT length,

\_\_in UINT elementSize,

\_\_out\_bcount(length\*elementSize) byte\* contentBuffer,

Js::ScriptContext\* scriptContext);

static Var SpeciesConstructor(RecyclableObject\* object, Var defaultConstructor, ScriptContext\* scriptContext);

static Var GetSpecies(RecyclableObject\* constructor, ScriptContext\* scriptContext);

private:

static BOOL RelationalComparsionHelper(Var aLeft, Var aRight, ScriptContext\* scriptContext, bool leftFirst, bool undefinedAs);

template <typename ArrayType>

static void ObjectToNativeArray(ArrayType\* arrayObject,

JsNativeValueType valueType,

\_\_in UINT length,

\_\_in UINT elementSize,

\_\_out\_bcount(length\*elementSize) byte\* contentBuffer,

Js::ScriptContext\* scriptContext);

template <typename ArrayType>

static Js::Var GetElementAtIndex(ArrayType\* arrayObject, UINT index, Js::ScriptContext\* scriptContext);

#if DBG

static BOOL IsPropertyObject(RecyclableObject \* instance);

#endif

template<typename PropertyKeyType, bool doFastProtoChainCheck, bool isRoot>

static BOOL CheckPrototypesForAccessorOrNonWritablePropertyCore(RecyclableObject\* instance,

PropertyKeyType propertyKey, Var\* setterValue, DescriptorFlags\* flags, PropertyValueInfo\* info, ScriptContext\* scriptContext);

static RecyclableObject \* GetPrototypeObject(RecyclableObject \* constructorFunction, ScriptContext \* scriptContext);

static RecyclableObject \* GetPrototypeObjectForConstructorCache(RecyclableObject \* constructor, ScriptContext \* scriptContext, bool& canBeCached);

static bool PrototypeObject(Var prototypeProperty, RecyclableObject \* constructorFunction, ScriptContext \* scriptContext, RecyclableObject\*\* prototypeObject);

static Var NewScObjectHostDispatchOrProxy(RecyclableObject \* function, ScriptContext \* requestContext);

static Var NewScObjectCommon(RecyclableObject \* functionObject, FunctionInfo \* functionInfo, ScriptContext \* scriptContext, bool isBaseClassConstructorNewScObject = false);

static BOOL Reject(bool throwOnError, ScriptContext\* scriptContext, long errorCode, PropertyId propertyId);

static bool AreSamePropertyDescriptors(const PropertyDescriptor\* x, const PropertyDescriptor\* y, ScriptContext\* scriptContext);

static Var CanonicalizeAccessor(Var accessor, ScriptContext\* scriptContext);

static void BuildHandlerScope(Var argThis, RecyclableObject \* hostObject, FrameDisplay \* pScopes, ScriptContext \* scriptContext);

static void TryLoadRoot(Var& thisVar, TypeId typeId, int moduleID, ScriptContext\* scriptContext);

template <bool unscopables>

static BOOL GetProperty\_Internal(Var instance, RecyclableObject\* propertyObject, const bool isRoot, PropertyId propertyId, Var\* value, ScriptContext\* requestContext, PropertyValueInfo\* info);

static RecyclableObject\* GetPrototypeNoTrap(RecyclableObject\* instance);

static BOOL GetPropertyReference\_Internal(Var instance, RecyclableObject\* propertyObject, const bool isRoot, PropertyId propertyId, Var\* value,ScriptContext\* requestContext, PropertyValueInfo\* info);

template <bool unscopables>

static BOOL PropertyReferenceWalk\_Impl(Var instance, RecyclableObject\*\* propertyObject, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

static Var TypeofFld\_Internal(Var instance, const bool isRoot, PropertyId propertyId, ScriptContext\* scriptContext);

template <bool unscopables>

static BOOL SetProperty\_Internal(Var instance, RecyclableObject\* object, const bool isRoot, PropertyId propertyId, Var newValue, PropertyValueInfo \* info, ScriptContext\* requestContext, PropertyOperationFlags flags);

template <typename TPropertyKey>

static DescriptorFlags GetRootSetter(RecyclableObject\* instance, TPropertyKey propertyKey, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext);

static BOOL IsNumberFromNativeArray(Var instance, uint32 index, ScriptContext\* scriptContext);

static BOOL GetItemFromArrayPrototype(JavascriptArray \* arr, int32 indexInt, Var \* result, ScriptContext \* scriptContext);

template <typename T>

static BOOL OP\_GetElementI\_ArrayFastPath(T \* arr, int indexInt, Var \* result, ScriptContext \* scriptContext);

static ImplicitCallFlags CacheAndClearImplicitBit(ScriptContext\* scriptContext);

static ImplicitCallFlags CheckAndUpdateFunctionBodyWithImplicitFlag(FunctionBody\* functionBody);

static void RestoreImplicitFlag(ScriptContext\* scriptContext, ImplicitCallFlags prevImplicitCallFlags, ImplicitCallFlags currImplicitCallFlags);

static BOOL ToPropertyDescriptorForProxyObjects(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext);

static BOOL ToPropertyDescriptorForGenericObjects(Var propertySpec, PropertyDescriptor\* descriptor, ScriptContext\* scriptContext);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

\_\_forceinline TypeId JavascriptOperators::GetTypeId(const Var aValue)

{

AssertMsg(aValue != nullptr, "GetTypeId aValue is null");

if (TaggedInt::Is(aValue))

{

return TypeIds\_Integer;

}

#if FLOATVAR

else if (JavascriptNumber::Is\_NoTaggedIntCheck(aValue))

{

return TypeIds\_Number;

}

#endif

else

{

auto typeId = RecyclableObject::FromVar(aValue)->GetTypeId();

#if DBG

auto isExternal = RecyclableObject::FromVar(aValue)->CanHaveInterceptors();

AssertMsg(typeId < TypeIds\_Limit || isExternal, "GetTypeId aValue has invalid TypeId");

#endif

return typeId;

}

}

template <BOOL stopAtProxy, class Func>

void JavascriptOperators::MapObjectAndPrototypes(RecyclableObject\* object, Func func)

{

MapObjectAndPrototypesUntil<stopAtProxy>(object, [=](RecyclableObject\* obj)

{

func(obj);

return false; // this will map whole prototype chain

});

}

template <BOOL stopAtProxy, class Func>

bool JavascriptOperators::MapObjectAndPrototypesUntil(RecyclableObject\* object, Func func)

{

TypeId typeId = JavascriptOperators::GetTypeId(object);

while (typeId != TypeIds\_Null && (!stopAtProxy || typeId != TypeIds\_Proxy))

{

if (func(object))

{

return true;

}

object = object->GetPrototype();

typeId = JavascriptOperators::GetTypeId(object);

}

return false;

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Language\JavascriptFunctionArgIndex.h"

#include "Language\InterpreterStackFrame.h"

#define FAligned(VALUE, TYPE) ((((LONG\_PTR)VALUE) & (sizeof(TYPE)-1)) == 0)

#define AlignIt(VALUE, TYPE) (~(~((LONG\_PTR)(VALUE) + (sizeof(TYPE)-1)) | (sizeof(TYPE)-1)))

namespace Js

{

Js::ArgumentsObject \* JavascriptCallStackLayout::GetArgumentsObject() const

{

return (Js::ArgumentsObject \*)((void \*\*)this)[JavascriptFunctionArgIndex\_ArgumentsObject];

}

Js::Var\* JavascriptCallStackLayout::GetArgumentsObjectLocation() const

{

return (Js::Var \*)&((void \*\*)this)[JavascriptFunctionArgIndex\_ArgumentsObject];

}

void JavascriptCallStackLayout::SetArgumentsObject(Js::ArgumentsObject \* obj)

{

((void \*\*)this)[JavascriptFunctionArgIndex\_ArgumentsObject] = obj;

}

Js::Var JavascriptCallStackLayout::GetOffset(int offset) const

{

Js::Var \*varPtr = (Js::Var \*)(((char \*)this) + offset);

Assert(FAligned(varPtr, Js::Var));

return \*varPtr;

}

double JavascriptCallStackLayout::GetDoubleAtOffset(int offset) const

{

double \*dblPtr = (double \*)(((char \*)this) + offset);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::CheckAlignmentFlag))

{

Assert(FAligned(dblPtr, double));

}

#endif

return \*dblPtr;

}

int32 JavascriptCallStackLayout::GetInt32AtOffset(int offset) const

{

int32 \*intPtr = (int32 \*)(((char \*)this) + offset);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.IsEnabled(Js::CheckAlignmentFlag))

{

Assert(FAligned(intPtr, int32));

}

#endif

return \*intPtr;

}

SIMDValue JavascriptCallStackLayout::GetSimdValueAtOffset(int offset) const

{

return \*((SIMDValue \*)(((char \*)this) + offset));

}

char \* JavascriptCallStackLayout::GetValueChangeOffset(int offset) const

{

Js::Var \*varPtr = (Js::Var \*)(((char \*)this) + offset);

Assert(FAligned(varPtr, Js::Var));

return (char \*)varPtr;

}

JavascriptCallStackLayout \*JavascriptCallStackLayout::FromFramePointer(void \*const framePointer)

{

return

reinterpret\_cast<JavascriptCallStackLayout \*>(

static\_cast<void \*\*>(framePointer) + (JavascriptFunctionArgIndex\_Function - JavascriptFunctionArgIndex\_Frame));

}

void\* const JavascriptCallStackLayout::ToFramePointer(JavascriptCallStackLayout\* callstackLayout)

{

return

reinterpret\_cast<void \* const>(

reinterpret\_cast<void \*\*>(callstackLayout) - (JavascriptFunctionArgIndex\_Function - JavascriptFunctionArgIndex\_Frame));

}

Js::Var\* JavascriptCallStackLayout::GetArgv() const

{

return const\_cast<Js::Var\*>(&this->args[0]);

}

ScriptContext\* JavascriptStackWalker::GetCurrentScriptContext() const

{

return this->GetCurrentInterpreterFrame() ? this->GetCurrentInterpreterFrame()->GetScriptContext() : this->scriptContext;

}

Var JavascriptStackWalker::GetCurrentArgumentsObject() const

{

#if ENABLE\_PROFILE\_INFO

if (interpreterFrame)

#else

Assert(interpreterFrame);

#endif

{

return interpreterFrame->GetArgumentsObject();

}

#if ENABLE\_NATIVE\_CODEGEN

else

{

if (inlinedFramesBeingWalked)

{

return inlinedFrameWalker.GetArgumentsObject();

}

else

{

return this->GetCurrentNativeArgumentsObject();

}

}

#endif

}

void JavascriptStackWalker::SetCurrentArgumentsObject(Var args)

{

#if ENABLE\_NATIVE\_CODEGEN

if (interpreterFrame)

#else

Assert(interpreterFrame);

#endif

{

interpreterFrame->SetArgumentsObject(args);

}

#if ENABLE\_NATIVE\_CODEGEN

else

{

if (inlinedFramesBeingWalked)

{

inlinedFrameWalker.SetArgumentsObject(args);

}

else

{

this->SetCurrentNativeArgumentsObject(args);

}

}

#endif

}

Var JavascriptStackWalker::GetPermanentArguments() const

{

Assert(IsJavascriptFrame());

AssertMsg(this->GetCurrentFunction()->IsScriptFunction(), "GetPermanentArguments should not be called for non-script function as there is no slot allocated for it.");

const uint32 paramCount = GetCallInfo()->Count;

if (paramCount == 0)

{

// glob function doesn't allocate ArgumentsObject slot on stack

return nullptr;

}

// Get the heap-allocated args for this frame.

Var args = this->GetCurrentArgumentsObject();

if (args && ArgumentsObject::Is(args))

{

args = ((ArgumentsObject\*)args)->GetHeapArguments();

}

return args;

}

void JavascriptStackWalker::SetPermanentArguments(Var heapArgs)

{

AssertMsg(this->GetCurrentFunction()->IsScriptFunction(), "SetPermanentArguments should not be called for non-script function as there is no slot allocated for it.");

// Set the heap-allocated args on this frame.

Var args = this->GetCurrentArgumentsObject();

if (args && ArgumentsObject::Is(args))

{

((ArgumentsObject\*)args)->SetHeapArguments((HeapArgumentsObject\*)heapArgs);

}

else

{

this->SetCurrentArgumentsObject(heapArgs);

}

}

BOOL JavascriptStackWalker::WalkToArgumentsFrame(ArgumentsObject \*args)

{

// Move the walker up the stack until we find the given arguments object on the frame.

while (this->Walk(/\*includeInlineFrame\*/ true))

{

if (this->IsJavascriptFrame())

{

Var currArgs = this->GetCurrentArgumentsObject();

if (currArgs == args)

{

return TRUE;

}

}

}

return FALSE;

}

bool JavascriptStackWalker::GetThis(Var\* pVarThis, int moduleId) const

{

#if ENABLE\_NATIVE\_CODEGEN

if (inlinedFramesBeingWalked)

{

if (inlinedFrameWalker.GetArgc() == 0)

{

\*pVarThis = JavascriptOperators::OP\_GetThis(this->scriptContext->GetLibrary()->GetUndefined(), moduleId, scriptContext);

return false;

}

\*pVarThis = inlinedFrameWalker.GetThisObject();

Assert(\*pVarThis);

return true;

}

else

#endif

{

CallInfo const \*callInfo = this->GetCallInfo();

if (callInfo->Count == 0)

{

\*pVarThis = JavascriptOperators::OP\_GetThis(scriptContext->GetLibrary()->GetUndefined(), moduleId, scriptContext);

return false;

}

\*pVarThis = this->GetThisFromFrame();

return (\*pVarThis) != nullptr;

}

}

BOOL IsEval(const CallInfo\* callInfo)

{

return (callInfo->Flags & CallFlags\_Eval) != 0;

}

BOOL JavascriptStackWalker::IsCallerGlobalFunction() const

{

CallInfo const\* callInfo = this->GetCallInfo();

JavascriptFunction\* function = this->GetCurrentFunction();

if (IsLibraryStackFrameEnabled(this->scriptContext) && !function->IsScriptFunction())

{

return false; // native library code can't be global function

}

FunctionInfo\* funcInfo = function->GetFunctionInfo();

if (funcInfo->HasParseableInfo())

{

return funcInfo->GetParseableFunctionInfo()->GetIsGlobalFunc() || IsEval(callInfo);

}

else

{

AssertMsg(FALSE, "Here we should only have script functions which were already parsed/deserialized.");

return callInfo->Count == 0 || IsEval(callInfo);

}

}

BOOL JavascriptStackWalker::IsEvalCaller() const

{

CallInfo const\* callInfo = this->GetCallInfo();

return (callInfo->Flags & CallFlags\_Eval) != 0;

}

Var JavascriptStackWalker::GetCurrentNativeArgumentsObject() const

{

Assert(this->IsJavascriptFrame() && this->interpreterFrame == nullptr);

return this->GetCurrentArgv()[JavascriptFunctionArgIndex\_ArgumentsObject];

}

void JavascriptStackWalker::SetCurrentNativeArgumentsObject(Var args)

{

Assert(this->IsJavascriptFrame() && this->interpreterFrame == nullptr);

this->GetCurrentArgv()[JavascriptFunctionArgIndex\_ArgumentsObject] = args;

}

Js::Var \* JavascriptStackWalker::GetJavascriptArgs() const

{

Assert(this->IsJavascriptFrame());

#if ENABLE\_NATIVE\_CODEGEN

if (inlinedFramesBeingWalked)

{

return inlinedFrameWalker.GetArgv(/\* includeThis = \*/ false);

}

else

#endif

if (this->GetCurrentFunction()->GetFunctionInfo()->IsGenerator())

{

JavascriptGenerator\* gen = JavascriptGenerator::FromVar(this->GetCurrentArgv()[JavascriptFunctionArgIndex\_This]);

return gen->GetArguments().Values;

}

else

{

return &this->GetCurrentArgv()[JavascriptFunctionArgIndex\_SecondScriptArg];

}

}

uint32 JavascriptStackWalker::GetByteCodeOffset() const

{

if (this->IsJavascriptFrame())

{

if (this->interpreterFrame && this->lastInternalFrameInfo.codeAddress == nullptr)

{

uint32 offset = this->interpreterFrame->GetReader()->GetCurrentOffset();

if (offset == 0)

{

// This will be the case when we are broken on the debugger on very first statement (due to async break).

// Or the interpreter loop can throw OOS on entrance before executing bytecode.

return offset;

}

else

{

// Note : For many cases, we move the m\_currentLocation of ByteCodeReader already to next available opcode.

// This could create problem in binding the exception to proper line offset.

// Reducing by 1 will make sure the current offset falls under, current executing opcode.

return offset - 1;

}

}

#if ENABLE\_NATIVE\_CODEGEN

DWORD\_PTR pCodeAddr;

uint loopNum = LoopHeader::NoLoop;

if (this->lastInternalFrameInfo.codeAddress != nullptr)

{

if (this->lastInternalFrameInfo.loopBodyFrameType == InternalFrameType\_LoopBody)

{

AnalysisAssert(this->interpreterFrame);

loopNum = this->interpreterFrame->GetCurrentLoopNum();

Assert(loopNum != LoopHeader::NoLoop);

}

pCodeAddr = (DWORD\_PTR)this->lastInternalFrameInfo.codeAddress;

}

else

{

if (this->IsCurrentPhysicalFrameForLoopBody())

{

// Internal frame but codeAddress on lastInternalFrameInfo not set. We must be in an inlined frame in the loop body.

Assert(this->tempInterpreterFrame);

loopNum = this->tempInterpreterFrame->GetCurrentLoopNum();

Assert(loopNum != LoopHeader::NoLoop);

}

pCodeAddr = (DWORD\_PTR)this->GetCurrentCodeAddr();

}

// If the current instruction's return address is the beginning of the next statement then we will show error for the next line, which would be completely wrong.

// The quick fix would be to look the address which is at least lesser than current return address.

// Assert to verify at what places this can happen.

Assert(pCodeAddr);

if (pCodeAddr)

{

#if defined(\_M\_ARM32\_OR\_ARM64)

// Note that DWORD\_PTR is not actually a pointer type (!) but is simple unsigned long/\_\_int64 (see BaseTsd.h).

// Thus, decrement would be by 1 byte and not 4 bytes as in pointer arithmetic. That's exactly what we need.

// For ARM the 'return address' is always odd and is 'next instr addr' + 1 byte, so to get to the BLX instr, we need to subtract 2 bytes from it.

AssertMsg(pCodeAddr % 2 == 1, "Got even number for pCodeAddr! It's expected to be return address, which should be odd.");

pCodeAddr--;

#endif

pCodeAddr--;

}

JavascriptFunction \*function = nullptr;

FunctionBody \*inlinee = nullptr;

StatementData data;

// For inlined frames, translation from native offset -> source code happens in two steps.

// The native offset is first translated into a statement index using the physical frame's

// source context info. This statement index is then looked up in the \*inlinee\*'s source

// context info to get the bytecode offset.

//

// For all inlined frames contained within a physical frame we have only one offset == (IP - entry).

// Since we can't use that to get the other inlined callers' IPs, we save the IP of all inlined

// callers in its "callinfo" (See InlineeCallInfo). The top most inlined frame uses the IP

// of the physical frame. All other inlined frames use the preceding inlined frame's offset.

//

function = this->GetCurrentFunctionFromPhysicalFrame();

inlinee = inlinedFramesBeingWalked ? inlinedFrameWalker.GetFunctionObject()->GetFunctionBody() : nullptr;

InlinedFrameWalker tmpFrameWalker;

if (inlinedFramesBeingWalked)

{

// Inlined frames are being walked right now. The top most frame is where the IP is.

if (!inlinedFrameWalker.IsTopMostFrame())

{

if (function->GetFunctionBody()->GetMatchingStatementMapFromNativeOffset(pCodeAddr,

inlinedFrameWalker.GetCurrentInlineeOffset(),

data,

loopNum,

inlinee))

{

return data.bytecodeBegin;

}

}

}

else if (ScriptFunction::Is(function) &&

InlinedFrameWalker::FromPhysicalFrame(tmpFrameWalker, currentFrame, ScriptFunction::FromVar(function), previousInterpreterFrameIsFromBailout, loopNum, this))

{

// Inlined frames are not being walked right now. However, if there

// are inlined frames on the stack the InlineeCallInfo of the first inlined frame

// has the native offset of the current physical frame.

Assert(!inlinee);

uint32 inlineeOffset = tmpFrameWalker.GetBottomMostInlineeOffset();

tmpFrameWalker.Close();

if (this->GetCurrentFunctionFromPhysicalFrame()->GetFunctionBody()->GetMatchingStatementMapFromNativeOffset(pCodeAddr,

inlineeOffset,

data,

loopNum,

inlinee))

{

return data.bytecodeBegin;

}

}

if (function->GetFunctionBody() && function->GetFunctionBody()->GetMatchingStatementMapFromNativeAddress(pCodeAddr, data, loopNum, inlinee))

{

return data.bytecodeBegin;

}

#endif

}

return 0;

}

bool JavascriptStackWalker::GetSourcePosition(const WCHAR\*\* sourceFileName, ULONG\* line, LONG\* column)

{

uint byteCodeoffset = this->GetByteCodeOffset();

if(byteCodeoffset)

{

Js::FunctionBody\* functionBody = this->GetCurrentFunction()->GetFunctionBody();

if (functionBody->GetLineCharOffset(byteCodeoffset, line, column))

{

if(functionBody->GetUtf8SourceInfo()->IsDynamic())

{

\*sourceFileName = L"Dynamic Code";

}

else

{

\*sourceFileName = functionBody->GetUtf8SourceInfo()->GetSrcInfo()->sourceContextInfo->url;

}

return true;

}

}

return false;

}

Js::JavascriptFunction \* JavascriptStackWalker::UpdateFrame(bool includeInlineFrames)

{

this->isJavascriptFrame = this->CheckJavascriptFrame(includeInlineFrames);

if (this->IsJavascriptFrame())

{

// In case we have a cross site thunk, update the script context

Js::JavascriptFunction \*function = this->GetCurrentFunction();

// We might've bailed out of an inlinee, so check if there were any inlinees.

if (this->interpreterFrame && (this->interpreterFrame->GetFlags() & InterpreterStackFrameFlags\_FromBailOut))

{

previousInterpreterFrameIsFromBailout = true;

#if ENABLE\_NATIVE\_CODEGEN

bool isCurrentPhysicalFrameForLoopBody = this->IsCurrentPhysicalFrameForLoopBody();

Assert(!inlinedFramesBeingWalked);

if (includeInlineFrames)

{

int loopNum = -1;

if (isCurrentPhysicalFrameForLoopBody)

{

loopNum = this->tempInterpreterFrame->GetCurrentLoopNum();

}

bool inlinedFramesOnStack = InlinedFrameWalker::FromPhysicalFrame(inlinedFrameWalker, currentFrame,

ScriptFunction::FromVar(function), true /\*fromBailout\*/, loopNum, this);

if (inlinedFramesOnStack)

{

inlinedFramesBeingWalked = inlinedFrameWalker.Next(inlinedFrameCallInfo);

Assert(inlinedFramesBeingWalked);

Assert(StackScriptFunction::GetCurrentFunctionObject(this->interpreterFrame->GetJavascriptFunction()) == inlinedFrameWalker.GetFunctionObject());

// We're now back in the state where currentFrame == physical frame of the inliner, but

// since interpreterFrame != null, we'll pick values from the interpreterFrame (the bailout

// frame of the inliner). Set a flag to tell the stack walker that it needs to start from the

// inlinee frames on the stack when Walk() is called.

}

else

{

Assert(!isCurrentPhysicalFrameForLoopBody);

}

}

else if (isCurrentPhysicalFrameForLoopBody)

{

// Getting here is only possible when the current interpreterFrame is for a function which

// encountered a bailout after getting inlined in a jitted loop body. If we are not including

// inlined frames in the stack walk, we need to set the codeAddress on lastInternalFrameInfo,

// which would have otherwise been set upon closing the inlinedFrameWalker, now.

// Note that we already have an assert in CheckJavascriptFrame to ensure this.

SetCachedInternalFrameInfo(InternalFrameType\_LoopBody, InternalFrameType\_LoopBody);

}

#else

// How did we bail out when JIT was disabled?

Assert(false);

#endif

}

else

{

Assert(this->interpreterFrame == nullptr || StackScriptFunction::GetCurrentFunctionObject(this->interpreterFrame->GetJavascriptFunction()) == function);

if (this->interpreterFrame)

{

previousInterpreterFrameIsFromBailout = false;

}

}

this->scriptContext = function->GetScriptContext();

return function;

}

return nullptr;

}

#if \_M\_X64

extern "C" void \*amd64\_ReturnFromCallWithFakeFrame(void);

#endif

// Note: noinline is to make sure that when we unwind to the unwindToAddress, there is at least one frame to unwind.

\_\_declspec(noinline)

JavascriptStackWalker::JavascriptStackWalker(ScriptContext \* scriptContext, bool useEERContext, PVOID returnAddress, bool \_forceFullWalk /\*=false\*/) :

inlinedFrameCallInfo(CallFlags\_None, 0), shouldDetectPartiallyInitializedInterpreterFrame(true), forceFullWalk(\_forceFullWalk),

previousInterpreterFrameIsFromBailout(false), ehFramesBeingWalkedFromBailout(false)

{

if (scriptContext == NULL)

{

Throw::InternalError();

}

this->scriptContext = scriptContext;

// Pull the current script state from the thread context.

ThreadContext \* threadContext = scriptContext->GetThreadContext();

this->entryExitRecord = threadContext->GetScriptEntryExit();

this->nativeLibraryEntry = threadContext->PeekNativeLibraryEntry();

this->prevNativeLibraryEntry = nullptr;

this->interpreterFrame = NULL;

this->isJavascriptFrame = false;

this->isNativeLibraryFrame = false;

if (entryExitRecord->frameIdOfScriptExitFunction != NULL)

{

// We're currently outside the script, so grab the frame from which we left.

this->scriptContext = entryExitRecord->scriptContext;

this->isInitialFrame = this->currentFrame.InitializeByFrameId(entryExitRecord->frameIdOfScriptExitFunction, this->scriptContext);

}

else

{

// Just start with the caller

this->isInitialFrame = this->currentFrame.InitializeByReturnAddress(\_ReturnAddress(), this->scriptContext);

}

if (useEERContext)

{

this->tempInterpreterFrame = this->scriptContext->GetThreadContext()->GetLeafInterpreterFrame();

}

else

{

// We need to generate stack for the passed script context, so use the leaf interpreter frame for passed script context

this->tempInterpreterFrame = scriptContext->GetThreadContext()->GetLeafInterpreterFrame();

}

inlinedFramesBeingWalked = false;

}

BOOL JavascriptStackWalker::Walk(bool includeInlineFrames)

{

// Walk one frame up the call stack.

this->interpreterFrame = NULL;

#if ENABLE\_NATIVE\_CODEGEN

if (inlinedFramesBeingWalked)

{

Assert(includeInlineFrames);

if (this->lastInternalFrameInfo.frameConsumed)

{

ClearCachedInternalFrameInfo();

}

inlinedFramesBeingWalked = inlinedFrameWalker.Next(inlinedFrameCallInfo);

if (!inlinedFramesBeingWalked)

{

inlinedFrameWalker.Close();

if ((this->IsCurrentPhysicalFrameForLoopBody()))

{

// Done walking inlined frames in a loop body, cache the native code address now

// in order to skip the loop body frame.

this->SetCachedInternalFrameInfo(InternalFrameType\_LoopBody, InternalFrameType\_LoopBody);

isJavascriptFrame = false;

}

}

return true;

}

#endif

if (this->isInitialFrame)

{

this->isInitialFrame = false; // Only walk initial frame once

}

else if (!this->currentFrame.Next())

{

this->isJavascriptFrame = false;

return false;

}

// If we're at the entry from a host frame, hop to the frame from which we left the script.

if (this->currentFrame.GetInstructionPointer() == this->entryExitRecord->returnAddrOfScriptEntryFunction)

{

BOOL hasCaller = this->entryExitRecord->hasCaller || this->forceFullWalk;

#ifdef CHECK\_STACKWALK\_EXCEPTION

BOOL ignoreStackWalkException = this->entryExitRecord->ignoreStackWalkException;

#endif

this->entryExitRecord = this->entryExitRecord->next;

if (this->entryExitRecord == NULL)

{

this->isJavascriptFrame = false;

return false;

}

if (!hasCaller && !this->scriptContext->IsDiagnosticsScriptContext())

{

#ifdef CHECK\_STACKWALK\_EXCEPTION

if (!ignoreStackWalkException)

{

AssertMsg(false, "walk pass no caller frame");

}

#endif

this->isJavascriptFrame = false;

return false;

}

this->scriptContext = this->entryExitRecord->scriptContext;

this->currentFrame.SkipToFrame(this->entryExitRecord->frameIdOfScriptExitFunction);

}

this->UpdateFrame(includeInlineFrames);

return true;

}

BOOL JavascriptStackWalker::GetCallerWithoutInlinedFrames(JavascriptFunction \*\* ppFunc)

{

return GetCaller(ppFunc, /\*includeInlineFrames\*/ false);

}

BOOL JavascriptStackWalker::GetCaller(JavascriptFunction \*\* ppFunc, bool includeInlineFrames)

{

while (this->Walk(includeInlineFrames))

{

if (this->IsJavascriptFrame())

{

Assert(entryExitRecord != NULL);

\*ppFunc = this->GetCurrentFunction();

AssertMsg(!this->shouldDetectPartiallyInitializedInterpreterFrame, "must have skipped first frame if needed");

return true;

}

}

\*ppFunc = (JavascriptFunction\*)this->scriptContext->GetLibrary()->GetNull();

return false;

}

BOOL JavascriptStackWalker::GetNonLibraryCodeCaller(JavascriptFunction \*\* ppFunc)

{

while (this->GetCaller(ppFunc))

{

if (!(\*ppFunc)->IsLibraryCode())

{

return true;

}

}

return false;

}

/\*static\*/

bool JavascriptStackWalker::IsLibraryStackFrameEnabled(Js::ScriptContext \* scriptContext)

{

Assert(scriptContext != nullptr);

return CONFIG\_FLAG(LibraryStackFrame);

}

// Check if a function is a display caller: user code, or native library / boundary script library code

bool JavascriptStackWalker::IsDisplayCaller(JavascriptFunction\* func)

{

FunctionBody\* body = func->GetFunctionBody();

if (IsLibraryStackFrameEnabled(func->GetScriptContext()))

{

return !func->IsScriptFunction() || !body->GetUtf8SourceInfo()->GetIsLibraryCode() || body->IsPublicLibraryCode();

}

else

{

return !body->GetUtf8SourceInfo()->GetIsLibraryCode();

}

}

bool JavascriptStackWalker::GetDisplayCaller(JavascriptFunction \*\* ppFunc)

{

while (this->GetCaller(ppFunc))

{

if (IsDisplayCaller(\*ppFunc))

{

return true;

}

}

return false;

}

PCWSTR JavascriptStackWalker::GetCurrentNativeLibraryEntryName() const

{

Assert(IsLibraryStackFrameEnabled(this->scriptContext)

&& this->prevNativeLibraryEntry

&& this->prevNativeLibraryEntry->next == this->nativeLibraryEntry);

return this->prevNativeLibraryEntry->name;

}

// WalkToTarget skips internal frames

BOOL JavascriptStackWalker::WalkToTarget(JavascriptFunction \* funcTarget)

{

// Walk up the call stack until we find the frame that belongs to the given function.

while (this->Walk(/\*includeInlineFrames\*/ true))

{

if (this->IsJavascriptFrame() && this->GetCurrentFunction() == funcTarget)

{

// Skip internal names

Assert( !(this->GetCallInfo()->Flags & CallFlags\_InternalFrame) );

return true;

}

}

return false;

}

void \*\* JavascriptStackWalker::GetCurrentArgv() const

{

Assert(this->IsJavascriptFrame());

Assert(this->interpreterFrame != nullptr ||

(this->prevNativeLibraryEntry && this->currentFrame.GetAddressOfReturnAddress() == this->prevNativeLibraryEntry->addr) ||

JavascriptFunction::IsNativeAddress(this->scriptContext, (void\*)this->currentFrame.GetInstructionPointer()));

bool isNativeAddr = (this->interpreterFrame == nullptr) &&

(!this->prevNativeLibraryEntry || (this->currentFrame.GetAddressOfReturnAddress() != this->prevNativeLibraryEntry->addr));

void \*\* argv = currentFrame.GetArgv(isNativeAddr, false /\*shouldCheckForNativeAddr\*/);

Assert(argv);

return argv;

}

bool JavascriptStackWalker::CheckJavascriptFrame(bool includeInlineFrames)

{

if (this->lastInternalFrameInfo.frameConsumed)

{

ClearCachedInternalFrameInfo();

}

this->isNativeLibraryFrame = false; // Clear previous result

void \* codeAddr = this->currentFrame.GetInstructionPointer();

if (this->tempInterpreterFrame && codeAddr == this->tempInterpreterFrame->GetReturnAddress())

{

// We need to skip over the first interpreter frame on the stack if it is the partially initialized frame

// otherwise it is a real frame and we should continue.

// For fully initialized frames (PushPopHelper was called) the thunk stack addr is equal or below addressOfReturnAddress

// as the latter one is obtained in InterpreterStackFrame::InterpreterThunk called by the thunk.

bool isPartiallyInitializedFrame = this->shouldDetectPartiallyInitializedInterpreterFrame &&

this->currentFrame.GetAddressOfReturnAddress(false /\*isCurrentContextNative\*/, false /\*shouldCheckForNativeAddr\*/) < this->tempInterpreterFrame->GetAddressOfReturnAddress();

this->shouldDetectPartiallyInitializedInterpreterFrame = false;

if (isPartiallyInitializedFrame)

{

return false; // Skip it.

}

void \*\* argv = this->currentFrame.GetArgv(false /\*isCurrentContextNative\*/, false /\*shouldCheckForNativeAddr\*/);

if (argv == nullptr)

{

// NOTE: When we switch to walking the stack ourselves and skip non engine frames, this should never happen.

return false;

}

#if defined(\_M\_AMD64)

if (argv[JavascriptFunctionArgIndex\_Function] == amd64\_ReturnFromCallWithFakeFrame)

{

this->ehFramesBeingWalkedFromBailout = true;

return false;

}

#endif

this->interpreterFrame = this->tempInterpreterFrame;

this->tempInterpreterFrame = this->interpreterFrame->GetPreviousFrame();

#if DBG && ENABLE\_NATIVE\_CODEGEN

if (((CallInfo const \*)&argv[JavascriptFunctionArgIndex\_CallInfo])->Flags & CallFlags\_InternalFrame)

{

// The return address of the interpreterFrame is the same as the entryPoint for a jitted loop body.

// This can only ever happen when we have bailed out from a function inlined in the loop body. We

// wouldn't have created a new interpreterFrame if the bailout were from the loop body itself.

Assert((this->interpreterFrame->GetFlags() & Js::InterpreterStackFrameFlags\_FromBailOut) != 0);

InlinedFrameWalker tmpFrameWalker;

Assert(InlinedFrameWalker::FromPhysicalFrame(tmpFrameWalker, currentFrame, Js::ScriptFunction::FromVar(argv[JavascriptFunctionArgIndex\_Function]),

true /\*fromBailout\*/, this->tempInterpreterFrame->GetCurrentLoopNum(), this, true /\*noAlloc\*/));

tmpFrameWalker.Close();

}

#endif

if (!this->interpreterFrame->IsCurrentLoopNativeAddr(this->lastInternalFrameInfo.codeAddress))

{

ClearCachedInternalFrameInfo();

}

else

{

Assert(this->lastInternalFrameInfo.codeAddress);

this->lastInternalFrameInfo.frameConsumed = true;

}

return true;

}

if (IsLibraryStackFrameEnabled(this->scriptContext) && this->nativeLibraryEntry)

{

void\* addressOfReturnAddress = this->currentFrame.GetAddressOfReturnAddress();

AssertMsg(addressOfReturnAddress <= this->nativeLibraryEntry->addr, "Missed matching native library entry?");

if (addressOfReturnAddress == this->nativeLibraryEntry->addr)

{

this->isNativeLibraryFrame = true;

this->shouldDetectPartiallyInitializedInterpreterFrame = false;

this->prevNativeLibraryEntry = this->nativeLibraryEntry; // Saves match in prevNativeLibraryEntry

this->nativeLibraryEntry = this->nativeLibraryEntry->next;

return true;

}

}

#if ENABLE\_NATIVE\_CODEGEN

BOOL isNativeAddr = JavascriptFunction::IsNativeAddress(this->scriptContext, codeAddr);

if (isNativeAddr)

{

this->shouldDetectPartiallyInitializedInterpreterFrame = false;

void \*\* argv = this->currentFrame.GetArgv(true /\*isCurrentContextNative\*/, false /\*shouldCheckForNativeAddr\*/);

if (argv == nullptr)

{

// NOTE: When we switch to walking the stack ourselves and skip non engine frames, this should never happen.

return false;

}

#if defined(\_M\_AMD64)

if (argv[JavascriptFunctionArgIndex\_Function] == amd64\_ReturnFromCallWithFakeFrame)

{

// There could be nested internal frames in the case of try...catch..finally

// let's not set the last internal frame address if it has already been set.

if(!this->lastInternalFrameInfo.codeAddress && !this->ehFramesBeingWalkedFromBailout)

{

SetCachedInternalFrameInfo(InternalFrameType\_EhFrame, InternalFrameType\_None);

}

return false;

}

#endif

ScriptFunction\* funcObj = Js::ScriptFunction::FromVar(argv[JavascriptFunctionArgIndex\_Function]);

if (funcObj->GetFunctionBody()->GetIsAsmjsMode())

{

return false;

}

// Note: this check has to happen after asm.js check, because call info is not valid for asm.js

if (((CallInfo const \*)&argv[JavascriptFunctionArgIndex\_CallInfo])->Flags & CallFlags\_InternalFrame)

{

if (includeInlineFrames &&

InlinedFrameWalker::FromPhysicalFrame(inlinedFrameWalker, currentFrame, Js::ScriptFunction::FromVar(argv[JavascriptFunctionArgIndex\_Function]),

false /\*fromBailout\*/, this->tempInterpreterFrame->GetCurrentLoopNum(), this))

{

// Found inlined frames in a jitted loop body. We dont want to skip the inlined frames; walk all of them before setting codeAddress on lastInternalFrameInfo.

inlinedFramesBeingWalked = inlinedFrameWalker.Next(inlinedFrameCallInfo);

Assert(inlinedFramesBeingWalked);

return true;

}

SetCachedInternalFrameInfo(InternalFrameType\_LoopBody, InternalFrameType\_LoopBody);

return false;

}

if (this->lastInternalFrameInfo.codeAddress)

{

this->lastInternalFrameInfo.frameConsumed = true;

}

if (includeInlineFrames &&

InlinedFrameWalker::FromPhysicalFrame(inlinedFrameWalker, currentFrame, Js::ScriptFunction::FromVar(argv[JavascriptFunctionArgIndex\_Function])))

{

inlinedFramesBeingWalked = inlinedFrameWalker.Next(inlinedFrameCallInfo);

Assert(inlinedFramesBeingWalked);

}

if (this->ehFramesBeingWalkedFromBailout)

{

AnalysisAssert(this->tempInterpreterFrame != nullptr);

this->interpreterFrame = this->tempInterpreterFrame;

this->tempInterpreterFrame = this->tempInterpreterFrame->GetPreviousFrame();

if (!this->interpreterFrame->IsCurrentLoopNativeAddr(this->lastInternalFrameInfo.codeAddress))

{

ClearCachedInternalFrameInfo();

}

else

{

Assert(this->lastInternalFrameInfo.codeAddress);

this->lastInternalFrameInfo.frameConsumed = true;

}

this->ehFramesBeingWalkedFromBailout = false;

}

return true;

}

#endif

return false;

}

void \* JavascriptStackWalker::GetCurrentCodeAddr() const

{

return this->currentFrame.GetInstructionPointer();

}

JavascriptFunction \* JavascriptStackWalker::GetCurrentFunction(bool includeInlinedFrames /\* = true \*/) const

{

Assert(this->IsJavascriptFrame());

#if ENABLE\_NATIVE\_CODEGEN

if (includeInlinedFrames && inlinedFramesBeingWalked)

{

return inlinedFrameWalker.GetFunctionObject();

}

else

#endif

if (this->isNativeLibraryFrame)

{

// Return saved function. Do not read from stack as compiler may stackpack/optimize args.

return JavascriptFunction::FromVar(this->prevNativeLibraryEntry->function);

}

else

{

return StackScriptFunction::GetCurrentFunctionObject((JavascriptFunction \*)this->GetCurrentArgv()[JavascriptFunctionArgIndex\_Function]);

}

}

void JavascriptStackWalker::SetCurrentFunction(JavascriptFunction \* function)

{

Assert(this->IsJavascriptFrame());

#if ENABLE\_NATIVE\_CODEGEN

if (inlinedFramesBeingWalked)

{

inlinedFrameWalker.SetFunctionObject(function);

}

else

#endif

{

this->GetCurrentArgv()[JavascriptFunctionArgIndex\_Function] = function;

}

}

JavascriptFunction \*JavascriptStackWalker::GetCurrentFunctionFromPhysicalFrame() const

{

return GetCurrentFunction(false);

}

CallInfo const \* JavascriptStackWalker::GetCallInfo(bool includeInlinedFrames /\* = true \*/) const

{

Assert(this->IsJavascriptFrame());

if (includeInlinedFrames && inlinedFramesBeingWalked)

{

// Since we don't support inlining constructors yet, its questionable if we should handle the

// hidden frame display here?

return (CallInfo const \*)&inlinedFrameCallInfo;

}

else if (this->GetCurrentFunction()->GetFunctionInfo()->IsGenerator())

{

JavascriptGenerator\* gen = JavascriptGenerator::FromVar(this->GetCurrentArgv()[JavascriptFunctionArgIndex\_This]);

return &gen->GetArguments().Info;

}

else if (this->isNativeLibraryFrame)

{

// Return saved callInfo. Do not read from stack as compiler may stackpack/optimize args.

return &this->prevNativeLibraryEntry->callInfo;

}

else

{

return (CallInfo const \*)&this->GetCurrentArgv()[JavascriptFunctionArgIndex\_CallInfo];

}

}

CallInfo const \*JavascriptStackWalker::GetCallInfoFromPhysicalFrame() const

{

return GetCallInfo(false);

}

Var JavascriptStackWalker::GetThisFromFrame() const

{

Assert(!inlinedFramesBeingWalked);

Assert(this->IsJavascriptFrame());

if (this->GetCurrentFunction()->GetFunctionInfo()->IsGenerator())

{

JavascriptGenerator\* gen = JavascriptGenerator::FromVar(this->GetCurrentArgv()[JavascriptFunctionArgIndex\_This]);

return gen->GetArguments()[0];

}

return this->GetCurrentArgv()[JavascriptFunctionArgIndex\_This];

}

void JavascriptStackWalker::ClearCachedInternalFrameInfo()

{

this->lastInternalFrameInfo.Clear();

}

void JavascriptStackWalker::SetCachedInternalFrameInfo(InternalFrameType frameType, InternalFrameType loopBodyFrameType)

{

if (!this->lastInternalFrameInfo.codeAddress)

{

this->lastInternalFrameInfo.Set(this->GetCurrentCodeAddr(), this->currentFrame.GetFrame(), this->currentFrame.GetStackCheckCodeHeight(), frameType, loopBodyFrameType);

}

this->lastInternalFrameInfo.loopBodyFrameType = loopBodyFrameType;

}

bool JavascriptStackWalker::IsCurrentPhysicalFrameForLoopBody() const

{

return !!(this->GetCallInfoFromPhysicalFrame()->Flags & CallFlags\_InternalFrame);

}

BOOL JavascriptStackWalker::GetCaller(JavascriptFunction\*\* ppFunc, ScriptContext\* scriptContext)

{

JavascriptStackWalker walker(scriptContext);

return walker.GetCaller(ppFunc);

}

BOOL JavascriptStackWalker::GetCaller(JavascriptFunction\*\* ppFunc, uint32\* byteCodeOffset, ScriptContext\* scriptContext)

{

JavascriptStackWalker walker(scriptContext);

if (walker.GetCaller(ppFunc))

{

\*byteCodeOffset = walker.GetByteCodeOffset();

return TRUE;

}

return FALSE;

}

bool JavascriptStackWalker::GetThis(Var\* pThis, int moduleId, ScriptContext\* scriptContext)

{

JavascriptStackWalker walker(scriptContext);

JavascriptFunction\* caller;

return walker.GetCaller(&caller) && walker.GetThis(pThis, moduleId);

}

bool JavascriptStackWalker::GetThis(Var\* pThis, int moduleId, JavascriptFunction\* func, ScriptContext\* scriptContext)

{

JavascriptStackWalker walker(scriptContext);

JavascriptFunction\* caller;

while (walker.GetCaller(&caller))

{

if (caller == func)

{

walker.GetThis(pThis, moduleId);

return true;

}

}

return false;

}

// Try to see whether there is a top-most javascript frame, and if there is return true if it's native.

// Returns true if top most frame is javascript frame, in this case the isNative parameter receives true

// when top-most frame is native, false otherwise.

// Returns false if top most frame is not a JavaScript frame.

/\* static \*/

bool JavascriptStackWalker::TryIsTopJavaScriptFrameNative(ScriptContext\* scriptContext, bool\* isNative, bool ignoreLibraryCode /\* = false \*/)

{

Assert(scriptContext);

Assert(isNative);

Js::JavascriptFunction\* caller;

Js::JavascriptStackWalker walker(scriptContext);

BOOL isSuccess;

if (ignoreLibraryCode)

{

isSuccess = walker.GetNonLibraryCodeCaller(&caller);

}

else

{

isSuccess = walker.GetCaller(&caller);

}

if (isSuccess)

{

\*isNative = (walker.GetCurrentInterpreterFrame() == NULL);

return true;

}

return false;

}

#if ENABLE\_NATIVE\_CODEGEN

bool InlinedFrameWalker::FromPhysicalFrame(InlinedFrameWalker& self, StackFrame& physicalFrame, Js::ScriptFunction \*parent, bool fromBailout, int loopNum, const JavascriptStackWalker \* const stackWalker, bool noAlloc)

{

bool inlinedFramesFound = false;

FunctionBody\* parentFunctionBody = parent->GetFunctionBody();

EntryPointInfo \*entryPointInfo;

if (loopNum != -1)

{

Assert(stackWalker);

}

void \*nativeCodeAddress = (loopNum == -1 || !stackWalker->GetCachedInternalFrameInfo().codeAddress) ? physicalFrame.GetInstructionPointer() : stackWalker->GetCachedInternalFrameInfo().codeAddress;

void \*framePointer = (loopNum == -1 || !stackWalker->GetCachedInternalFrameInfo().codeAddress) ? physicalFrame.GetFrame() : stackWalker->GetCachedInternalFrameInfo().framePointer;

if (loopNum != -1)

{

entryPointInfo = (Js::EntryPointInfo\*)parentFunctionBody->GetLoopEntryPointInfoFromNativeAddress((DWORD\_PTR)nativeCodeAddress, loopNum);

}

else

{

entryPointInfo = (Js::EntryPointInfo\*)parentFunctionBody->GetEntryPointFromNativeAddress((DWORD\_PTR)physicalFrame.GetInstructionPointer());

}

AssertMsg(entryPointInfo != nullptr, "Inlined frame should resolve to the right parent address");

if (entryPointInfo->HasInlinees())

{

void \*entry = reinterpret\_cast<void\*>(entryPointInfo->GetNativeAddress());

InlinedFrameWalker::InlinedFrame \*outerMostFrame = InlinedFrame::FromPhysicalFrame(physicalFrame, stackWalker, entry, entryPointInfo);

if (!outerMostFrame)

{

return inlinedFramesFound;

}

if (!fromBailout)

{

InlineeFrameRecord\* record = entryPointInfo->FindInlineeFrame((void\*)nativeCodeAddress);

if (record)

{

record->RestoreFrames(parent->GetFunctionBody(), outerMostFrame, JavascriptCallStackLayout::FromFramePointer(framePointer));

}

}

if (outerMostFrame->callInfo.Count)

{

inlinedFramesFound = true;

if (noAlloc)

{

return inlinedFramesFound;

}

int32 frameCount = 0;

InlinedFrameWalker::InlinedFrame \*frameIterator = outerMostFrame;

while (frameIterator->callInfo.Count)

{

frameCount++;

frameIterator = frameIterator->Next();

}

InlinedFrameWalker::InlinedFrame \*\*frames = HeapNewArray(InlinedFrameWalker::InlinedFrame\*, frameCount);

frameIterator = outerMostFrame;

for (int index = frameCount - 1; index >= 0; index--)

{

Assert(frameIterator);

frames[index] = frameIterator;

frameIterator = frameIterator->Next();

}

self.Initialize(frameCount, frames, parent);

}

}

return inlinedFramesFound;

}

void InlinedFrameWalker::Close()

{

parentFunction = nullptr;

HeapDeleteArray(frameCount, frames);

frames = nullptr;

currentIndex = -1;

frameCount = 0;

}

bool InlinedFrameWalker::Next(CallInfo& callInfo)

{

MoveNext();

InlinedFrameWalker::InlinedFrame \*const currentFrame = GetCurrentFrame();

if (currentFrame)

{

callInfo.Flags = CallFlags\_None;

callInfo.Count = (currentFrame->callInfo.Count & 0xFFFF);

}

return currentFrame != nullptr;

}

size\_t InlinedFrameWalker::GetArgc() const

{

InlinedFrameWalker::InlinedFrame \*const currentFrame = GetCurrentFrame();

Assert(currentFrame);

return currentFrame->callInfo.Count;

}

Js::Var \*InlinedFrameWalker::GetArgv(bool includeThis /\* = true \*/) const

{

InlinedFrameWalker::InlinedFrame \*const currentFrame = GetCurrentFrame();

Assert(currentFrame);

uint firstArg = includeThis ? InlinedFrameArgIndex\_This : InlinedFrameArgIndex\_SecondScriptArg;

Js::Var \*args = &currentFrame->argv[firstArg];

this->FinalizeStackValues(args, this->GetArgc() - firstArg);

return args;

}

void InlinedFrameWalker::FinalizeStackValues(\_\_in\_ecount(argCount) Js::Var args[], size\_t argCount) const

{

ScriptContext \*scriptContext = this->GetFunctionObject()->GetScriptContext();

for (size\_t i = 0; i < argCount; i++)

{

args[i] = Js::JavascriptOperators::BoxStackInstance(args[i], scriptContext);

}

}

Js::JavascriptFunction \*InlinedFrameWalker::GetFunctionObject() const

{

InlinedFrameWalker::InlinedFrame \*const currentFrame = GetCurrentFrame();

Assert(currentFrame);

return StackScriptFunction::GetCurrentFunctionObject(currentFrame->function);

}

void InlinedFrameWalker::SetFunctionObject(Js::JavascriptFunction \* function)

{

InlinedFrameWalker::InlinedFrame \*const currentFrame = GetCurrentFrame();

Assert(currentFrame);

currentFrame->function = function;

}

Js::Var InlinedFrameWalker::GetArgumentsObject() const

{

InlinedFrameWalker::InlinedFrame \*const currentFrame = GetCurrentFrame();

Assert(currentFrame);

return currentFrame->arguments;

}

void InlinedFrameWalker::SetArgumentsObject(Js::Var arguments)

{

InlinedFrameWalker::InlinedFrame \*currentFrame = (InlinedFrameWalker::InlinedFrame \*)GetCurrentFrame();

Assert(currentFrame);

currentFrame->arguments = arguments;

}

Js::Var InlinedFrameWalker::GetThisObject() const

{

InlinedFrameWalker::InlinedFrame \*const currentFrame = GetCurrentFrame();

Assert(currentFrame);

return currentFrame->argv[InlinedFrameArgIndex\_This];

}

bool InlinedFrameWalker::IsCallerPhysicalFrame() const

{

return currentIndex == (frameCount - 1);

}

bool InlinedFrameWalker::IsTopMostFrame() const

{

return currentIndex == 0;

}

uint32 InlinedFrameWalker::GetCurrentInlineeOffset() const

{

Assert(!IsTopMostFrame());

Assert(currentIndex);

return GetFrameAtIndex(currentIndex - 1)->callInfo.InlineeStartOffset;

}

uint32 InlinedFrameWalker::GetBottomMostInlineeOffset() const

{

Assert(frameCount);

return GetFrameAtIndex(frameCount - 1)->callInfo.InlineeStartOffset;

}

Js::JavascriptFunction \*InlinedFrameWalker::GetBottomMostFunctionObject() const

{

Assert(frameCount);

return GetFrameAtIndex(frameCount - 1)->function;

}

InlinedFrameWalker::InlinedFrame \*const InlinedFrameWalker::GetCurrentFrame() const

{

return GetFrameAtIndex(currentIndex);

}

InlinedFrameWalker::InlinedFrame \*const InlinedFrameWalker::GetFrameAtIndex(signed index) const

{

Assert(frames);

Assert(frameCount);

InlinedFrameWalker::InlinedFrame \*frame = nullptr;

if (index < frameCount)

{

frame = frames[index];

}

return frame;

}

void InlinedFrameWalker::MoveNext()

{

currentIndex++;

}

void InlinedFrameWalker::Initialize(int32 frameCount, \_\_in\_ecount(frameCount) InlinedFrame \*\*frames, Js::ScriptFunction \*parent)

{

Assert(!parentFunction);

Assert(!this->frames);

Assert(!this->frameCount);

Assert(currentIndex == -1);

this->parentFunction = parent;

this->frames = frames;

this->frameCount = frameCount;

this->currentIndex = -1;

}

InlinedFrameWalker::InlinedFrame\* InlinedFrameWalker::InlinedFrame::FromPhysicalFrame(StackFrame& currentFrame, const JavascriptStackWalker \* const stackWalker, void \*entry, EntryPointInfo\* entryPointInfo)

{

// If the current javascript frame is a native frame, get the inlined frame from it, otherwise

// it may be possible that current frame is the interpreter frame for a jitted loop body

// If the loop body had some inlinees in it, retrieve the inlined frame using the cached info,

// viz. instruction pointer, frame pointer, and stackCheckCodeHeight, about the loop body frame.

struct InlinedFrame \*inlinedFrame = nullptr;

void \*codeAddr, \*framePointer;

size\_t stackCheckCodeHeight;

if (entryPointInfo->IsLoopBody() && stackWalker && stackWalker->GetCachedInternalFrameInfo().codeAddress)

{

codeAddr = stackWalker->GetCachedInternalFrameInfo().codeAddress;

framePointer = stackWalker->GetCachedInternalFrameInfo().framePointer;

stackCheckCodeHeight = stackWalker->GetCachedInternalFrameInfo().stackCheckCodeHeight;

}

else

{

codeAddr = currentFrame.GetInstructionPointer();

framePointer = currentFrame.GetFrame();

stackCheckCodeHeight = currentFrame.GetStackCheckCodeHeight();

}

if (!StackFrame::IsInStackCheckCode(entry, codeAddr, stackCheckCodeHeight))

{

inlinedFrame = (struct InlinedFrame \*)(((uint8 \*)framePointer) - entryPointInfo->frameHeight);

}

return inlinedFrame;

}

void InternalFrameInfo::Set(void \*codeAddress, void \*framePointer, size\_t stackCheckCodeHeight, InternalFrameType frameType, InternalFrameType loopBodyFrameType)

{

// We skip a jitted loop body's native frame when walking the stack and refer to the loop body's interpreter frame to get the function.

// However, if the loop body has inlinees, to retrieve inlinee frames we need to cache some info about the loop body's native frame.

this->codeAddress = codeAddress;

this->framePointer = framePointer;

this->stackCheckCodeHeight = stackCheckCodeHeight;

this->frameType = frameType;

this->loopBodyFrameType = loopBodyFrameType;

this->frameConsumed = false;

}

void InternalFrameInfo::Clear()

{

this->codeAddress = nullptr;

this->framePointer = nullptr;

this->stackCheckCodeHeight = (uint)-1;

this->frameType = InternalFrameType\_None;

this->loopBodyFrameType = InternalFrameType\_None;

this->frameConsumed = false;

}

#endif

#if DBG

// Force a stack walk which till we find an interpreter frame

// This will ensure inlined frames are decoded.

bool JavascriptStackWalker::ValidateTopJitFrame(Js::ScriptContext\* scriptContext)

{

if (!Configuration::Global.flags.ValidateInlineStack)

{

return true;

}

Js::JavascriptStackWalker walker(scriptContext);

Js::JavascriptFunction\* function;

while (walker.GetCaller(&function))

{

Assert(function);

if (walker.GetCurrentInterpreterFrame() != nullptr)

{

break;

}

}

// If no asserts have fired yet - we should have succeeded.

return true;

}

#endif

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#if defined(\_M\_IX86)

#include "Language\i386\stackframe.h"

typedef Js::X86StackFrame StackFrame;

#elif defined(\_M\_X64)

#include "Language\amd64\stackframe.h"

#include "Language\amd64\stackframe.inl"

typedef Js::Amd64StackFrame StackFrame;

#elif defined(\_M\_ARM)

#include "Language\arm\stackframe.h"

typedef Js::ArmStackFrame StackFrame;

#elif defined(\_M\_ARM64)

#include "Language\arm64\stackframe.h"

typedef Js::Arm64StackFrame StackFrame;

#else

#error JavascriptStackWalker is not supported on this architecture.

#endif

namespace Js

{

struct ScriptEntryExitRecord;

enum InternalFrameType {

InternalFrameType\_None,

InternalFrameType\_EhFrame,

InternalFrameType\_LoopBody

};

struct AsmJsCallStackLayout

{

Js::JavascriptFunction \* functionObject;

Js::Var args[0];

};

class JavascriptCallStackLayout

{

public:

Js::JavascriptFunction \* functionObject;

Js::CallInfo callInfo;

Js::Var args[0];

Js::ArgumentsObject \* GetArgumentsObject() const;

Js::Var \* GetArgumentsObjectLocation() const;

void SetArgumentsObject(Js::ArgumentsObject\* obj);

Js::Var\* GetArgv() const;

Js::Var GetOffset(int offset) const;

double GetDoubleAtOffset(int offset) const;

int32 GetInt32AtOffset(int offset) const;

SIMDValue GetSimdValueAtOffset(int offset) const;

char \* GetValueChangeOffset(int offset) const;

static JavascriptCallStackLayout \*FromFramePointer(void \*const framePointer);

static void\* const ToFramePointer(JavascriptCallStackLayout\* callstackLayout);

private:

JavascriptCallStackLayout() : callInfo(0) {};

};

#if ENABLE\_NATIVE\_CODEGEN

/\*

\* The InlinedFrameStackWalker knows how to walk an inlinee's virtual frames inside a

\* physical frame. If the stack walker is in the inlineeFramesBeingWalked mode it

\* defers pretty much all functionality to its helpers. The virtual stack frames themselves

\* are laid out in the reverse order on the stack. So we do one pass to find out the count of

\* frames and another to construct an array of pointers to frames in the correct order

\* (top most first like a real stack). Each frame begins with a count. Since frames are laid

\* out continuously in memory, this is all the stack walker needs to find the next frame.

\* We don't maintain explicit prev, next pointers. We also clear the count field of the frame

\* next to the top most frame to indicate that the top most frame is, well, the top most frame.

\* Whenever an inlinee's code ends, the count field in its frame gets set to 0 indicating this

\* frame isn't valid anymore. This keeps the fast case fast and offloads the heavy lifting

\* to the stack walker.

\*/

class InlinedFrameWalker

{

public:

InlinedFrameWalker()

: parentFunction(nullptr)

, frames(nullptr)

, currentIndex(-1)

, frameCount(0)

{

}

~InlinedFrameWalker()

{

Assert(!parentFunction);

Assert(!this->frames);

Assert(!frameCount);

Assert(currentIndex == -1);

}

static bool FromPhysicalFrame(InlinedFrameWalker& self, StackFrame& physicalFrame, Js::ScriptFunction \*parent, bool fromBailout = false, int loopNum = -1, const JavascriptStackWalker \* const walker = nullptr, bool noAlloc = false);

void Close();

bool Next(CallInfo& callInfo);

size\_t GetArgc() const;

Js::Var \*GetArgv(bool includeThis = true) const;

Js::JavascriptFunction \*GetFunctionObject() const;

void SetFunctionObject(Js::JavascriptFunction \* function);

Js::Var GetArgumentsObject() const;

void SetArgumentsObject(Js::Var arguments);

Js::Var GetThisObject() const;

bool IsCallerPhysicalFrame() const;

bool IsTopMostFrame() const;

int32 GetFrameIndex() const { Assert(currentIndex != -1); return currentIndex; }

uint32 GetCurrentInlineeOffset() const;

uint32 GetBottomMostInlineeOffset() const;

Js::JavascriptFunction \*GetBottomMostFunctionObject() const;

void FinalizeStackValues(\_\_in\_ecount(argCount) Js::Var args[], size\_t argCount) const;

private:

enum {

InlinedFrameArgIndex\_This = 0,

InlinedFrameArgIndex\_SecondScriptArg = 1

};

struct InlinedFrame : public InlinedFrameLayout

{

Js::Var argv[0]; // It's defined here as in C++ can't have 0-size array in the base class.

struct InlinedFrame \*Next()

{

InlinedFrameLayout \*next = \_\_super::Next();

return (InlinedFrame\*)next;

}

static InlinedFrame \*FromPhysicalFrame(StackFrame& currentFrame, const JavascriptStackWalker \* const stackWalker, void \*entry, EntryPointInfo\* entryPointInfo);

};

void Initialize(int32 frameCount, \_\_in\_ecount(frameCount) InlinedFrame \*\*frames, Js::ScriptFunction \*parent);

void MoveNext();

InlinedFrame \*const GetCurrentFrame() const;

InlinedFrame \*const GetFrameAtIndex(signed index) const;

Js::ScriptFunction \*parentFunction;

InlinedFrame \*\*frames;

int32 currentIndex;

int32 frameCount;

};

class InternalFrameInfo

{

public:

void \*codeAddress;

void \*framePointer;

size\_t stackCheckCodeHeight;

InternalFrameType frameType;

InternalFrameType loopBodyFrameType;

bool frameConsumed;

InternalFrameInfo() :

codeAddress(nullptr),

framePointer(nullptr),

stackCheckCodeHeight((uint)-1),

frameType(InternalFrameType\_None),

loopBodyFrameType(InternalFrameType\_None),

frameConsumed(false)

{

}

void Clear();

void Set(void \*codeAddress, void \*framePointer, size\_t stackCheckCodeHeight, InternalFrameType frameType, InternalFrameType loopBodyFrameType);

};

#endif

class JavascriptStackWalker

{

friend Js::ScriptContext;

public:

JavascriptStackWalker(ScriptContext \* scriptContext, bool useEERContext = TRUE /\* use leafinterpreterframe of entryexit record \*/, PVOID returnAddress = NULL, bool \_forceFullWalk = false);

#if ENABLE\_NATIVE\_CODEGEN

~JavascriptStackWalker() { inlinedFrameWalker.Close(); }

#endif

BOOL Walk(bool includeInlineFrames = true);

BOOL GetCaller(JavascriptFunction \*\* ppFunc, bool includeInlineFrames = true);

BOOL GetCallerWithoutInlinedFrames(JavascriptFunction \*\* ppFunc);

BOOL GetNonLibraryCodeCaller(JavascriptFunction \*\* ppFunc);

BOOL WalkToTarget(JavascriptFunction \* funcTarget);

BOOL WalkToArgumentsFrame(ArgumentsObject \*argsObj);

uint32 GetByteCodeOffset() const;

BOOL IsCallerGlobalFunction() const;

BOOL IsEvalCaller() const;

bool IsJavascriptFrame() const { return inlinedFramesBeingWalked || isJavascriptFrame; }

bool IsInlineFrame() const { return inlinedFramesBeingWalked; }

bool IsBailedOutFromInlinee() const

{

Assert(this->IsJavascriptFrame() && this->interpreterFrame);

return IsInlineFrame();

}

bool IsBailedOutFromFunction() const

{

Assert(this->IsJavascriptFrame() && this->interpreterFrame);

return !!JavascriptFunction::IsNativeAddress(this->scriptContext, this->currentFrame.GetInstructionPointer());

}

Var GetPermanentArguments() const;

void SetPermanentArguments(Var args);

void \*GetCurrentCodeAddr() const;

JavascriptFunction \*GetCurrentFunction(bool includeInlinedFrames = true) const;

void SetCurrentFunction(JavascriptFunction \* function);

CallInfo const \*GetCallInfo(bool includeInlinedFrames = true) const;

CallInfo const \*GetCallInfoFromPhysicalFrame() const;

bool GetThis(Var \*pThis, int moduleId) const;

Js::Var \* GetJavascriptArgs() const;

void \*\*GetCurrentArgv() const;

ScriptContext\* GetCurrentScriptContext() const;

InterpreterStackFrame\* GetCurrentInterpreterFrame() const

{

Assert(this->IsJavascriptFrame());

return interpreterFrame;

}

bool GetSourcePosition(const WCHAR\*\* sourceFileName, ULONG\* line, LONG\* column);

static bool TryIsTopJavaScriptFrameNative(ScriptContext\* scriptContext, bool\* istopFrameNative, bool ignoreLibraryCode = false);

void ClearCachedInternalFrameInfo();

void SetCachedInternalFrameInfo(InternalFrameType frameType, InternalFrameType loopBodyFrameType);

InternalFrameInfo GetCachedInternalFrameInfo() const { return this->lastInternalFrameInfo; }

bool IsCurrentPhysicalFrameForLoopBody() const;

// noinline, we want to use own stack frame.

static \_\_declspec(noinline) BOOL GetCaller(JavascriptFunction\*\* ppFunc, ScriptContext\* scriptContext);

static \_\_declspec(noinline) BOOL GetCaller(JavascriptFunction\*\* ppFunc, uint32\* byteCodeOffset, ScriptContext\* scriptContext);

static \_\_declspec(noinline) bool GetThis(Var\* pThis, int moduleId, ScriptContext\* scriptContext);

static \_\_declspec(noinline) bool GetThis(Var\* pThis, int moduleId, JavascriptFunction\* func, ScriptContext\* scriptContext);

static bool IsDisplayCaller(JavascriptFunction\* func);

bool GetDisplayCaller(JavascriptFunction \*\* ppFunc);

PCWSTR GetCurrentNativeLibraryEntryName() const;

static bool IsLibraryStackFrameEnabled(Js::ScriptContext \* scriptContext);

// Walk frames (until walkFrame returns true)

template <class WalkFrame>

ushort WalkUntil(ushort stackTraceLimit, WalkFrame walkFrame, bool onlyOnDebugMode = false, bool filterDiagnosticsOM = false)

{

ushort frameIndex = 0;

JavascriptFunction\* jsFunction;

BOOL foundCaller = GetNonLibraryCodeCaller(&jsFunction);

while (foundCaller)

{

if (IsDisplayCaller(jsFunction))

{

bool needToPass = (!onlyOnDebugMode || jsFunction->GetScriptContext()->IsInDebugMode())

&& (!filterDiagnosticsOM || !jsFunction->GetScriptContext()->IsDiagnosticsScriptContext());

if (needToPass)

{

if (walkFrame(jsFunction, frameIndex))

{

break;

}

frameIndex++;

}

}

foundCaller = frameIndex < stackTraceLimit && GetCaller(&jsFunction);

}

return frameIndex;

}

template <class WalkFrame>

ushort WalkUntil(WalkFrame walkFrame, bool onlyOnDebugMode = false, bool filterDiagnosticsOM = false)

{

return WalkUntil(USHORT\_MAX, walkFrame, onlyOnDebugMode, filterDiagnosticsOM);

}

BYTE\*\* GetCurrentAddresOfReturnAddress() const

{

return (BYTE\*\*)this->currentFrame.GetAddressOfReturnAddress();

}

BYTE\*\* GetCurrentAddressOfInstructionPointer() const

{

return (BYTE\*\*)this->currentFrame.GetAddressOfInstructionPointer();

}

void\* GetInstructionPointer() const

{

return this->currentFrame.GetInstructionPointer();

}

bool GetCurrentFrameFromBailout() const

{

return previousInterpreterFrameIsFromBailout;

}

#if DBG

static bool ValidateTopJitFrame(Js::ScriptContext\* scriptContext);

#endif

private:

ScriptContext \*scriptContext;

ScriptEntryExitRecord \*entryExitRecord;

const NativeLibraryEntryRecord::Entry \*nativeLibraryEntry;

const NativeLibraryEntryRecord::Entry \*prevNativeLibraryEntry; // Saves previous nativeLibraryEntry when it moves to next

InterpreterStackFrame \*interpreterFrame;

InterpreterStackFrame \*tempInterpreterFrame;

#if ENABLE\_NATIVE\_CODEGEN

Js::InlinedFrameWalker inlinedFrameWalker;

#endif

CallInfo inlinedFrameCallInfo;

bool inlinedFramesBeingWalked : 1;

bool isJavascriptFrame : 1;

bool isNativeLibraryFrame : 1;

bool isInitialFrame : 1; // If we need to walk the initial frame

bool shouldDetectPartiallyInitializedInterpreterFrame : 1;

bool previousInterpreterFrameIsFromBailout : 1;

bool ehFramesBeingWalkedFromBailout : 1;

bool forceFullWalk; // ignoring hasCaller

Var GetThisFromFrame() const;

Var GetCurrentArgumentsObject() const;

void SetCurrentArgumentsObject(Var args);

Var GetCurrentNativeArgumentsObject() const;

void SetCurrentNativeArgumentsObject(Var args);

InternalFrameInfo lastInternalFrameInfo;

mutable StackFrame currentFrame;

Js::JavascriptFunction \* UpdateFrame(bool includeInlineFrames);

bool CheckJavascriptFrame(bool includeInlineFrames);

JavascriptFunction \*JavascriptStackWalker::GetCurrentFunctionFromPhysicalFrame() const;

};

class AutoPushReturnAddressForStackWalker

{

private:

ScriptContext \*m\_scriptContext;

public:

AutoPushReturnAddressForStackWalker(ScriptContext \*scriptContext, void\* returnAddress) : m\_scriptContext(scriptContext)

{

scriptContext->SetFirstInterpreterFrameReturnAddress(returnAddress);

}

~AutoPushReturnAddressForStackWalker()

{

m\_scriptContext->SetFirstInterpreterFrameReturnAddress(NULL);

}

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

namespace Js

{

#if ENABLE\_PROFILE\_INFO

Var ProfilingHelpers::ProfiledLdElem(

const Var base,

const Var varIndex,

FunctionBody \*const functionBody,

const ProfileId profileId)

{

Assert(base);

Assert(varIndex);

Assert(functionBody);

Assert(profileId != Constants::NoProfileId);

LdElemInfo ldElemInfo;

// Only enable fast path if the javascript array is not cross site

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(base);

#endif

const bool isJsArray = !TaggedNumber::Is(base) && VirtualTableInfo<JavascriptArray>::HasVirtualTable(base);

const bool fastPath = isJsArray;

if(fastPath)

{

JavascriptArray \*const array = JavascriptArray::FromVar(base);

ldElemInfo.arrayType = ValueType::FromArray(ObjectType::Array, array, TypeIds\_Array).ToLikely();

const Var element = ProfiledLdElem\_FastPath(array, varIndex, functionBody->GetScriptContext(), &ldElemInfo);

ldElemInfo.elemType = ldElemInfo.elemType.Merge(element);

functionBody->GetDynamicProfileInfo()->RecordElementLoad(functionBody, profileId, ldElemInfo);

return element;

}

Assert(!isJsArray);

bool isObjectWithArray;

TypeId arrayTypeId;

JavascriptArray \*const array =

JavascriptArray::GetArrayForArrayOrObjectWithArray(base, &isObjectWithArray, &arrayTypeId);

do // while(false)

{

// The fast path is only for JavascriptArray and doesn't cover native arrays, objects with internal arrays, or typed

// arrays, but we still need to profile the array

uint32 headSegmentLength;

if(array)

{

ldElemInfo.arrayType =

(

isObjectWithArray

? ValueType::FromObjectArray(array)

: ValueType::FromArray(ObjectType::Array, array, arrayTypeId)

).ToLikely();

SparseArraySegmentBase \*const head = array->GetHead();

Assert(head->left == 0);

headSegmentLength = head->length;

}

else if(TypedArrayBase::TryGetLengthForOptimizedTypedArray(base, &headSegmentLength, &arrayTypeId))

{

bool isVirtual = (VirtualTableInfoBase::GetVirtualTable(base) == ValueType::GetVirtualTypedArrayVtable(arrayTypeId));

ldElemInfo.arrayType = ValueType::FromTypeId(arrayTypeId, isVirtual).ToLikely();

}

else

{

break;

}

if(!TaggedInt::Is(varIndex))

{

ldElemInfo.neededHelperCall = true;

break;

}

const int32 index = TaggedInt::ToInt32(varIndex);

const uint32 offset = index;

if(index < 0 || offset >= headSegmentLength || array && array->IsMissingHeadSegmentItem(offset))

{

ldElemInfo.neededHelperCall = true;

break;

}

} while(false);

const Var element = JavascriptOperators::OP\_GetElementI(base, varIndex, functionBody->GetScriptContext());

const ValueType arrayType(ldElemInfo.GetArrayType());

if(!arrayType.IsUninitialized())

{

if(arrayType.IsLikelyObject() && arrayType.GetObjectType() == ObjectType::Array && !arrayType.HasIntElements())

{

JavascriptOperators::UpdateNativeArrayProfileInfoToCreateVarArray(

array,

arrayType.HasFloatElements(),

arrayType.HasVarElements());

}

ldElemInfo.elemType = ValueType::Uninitialized.Merge(element);

functionBody->GetDynamicProfileInfo()->RecordElementLoad(functionBody, profileId, ldElemInfo);

return element;

}

functionBody->GetDynamicProfileInfo()->RecordElementLoadAsProfiled(functionBody, profileId);

return element;

}

Var ProfilingHelpers::ProfiledLdElem\_FastPath(

JavascriptArray \*const array,

const Var varIndex,

ScriptContext \*const scriptContext,

LdElemInfo \*const ldElemInfo)

{

Assert(array);

Assert(varIndex);

Assert(scriptContext);

do // while(false)

{

Assert(!array->IsCrossSiteObject());

if (!TaggedInt::Is(varIndex))

{

break;

}

int32 index = TaggedInt::ToInt32(varIndex);

if (index < 0)

{

break;

}

if(ldElemInfo)

{

SparseArraySegment<Var> \*const head = static\_cast<SparseArraySegment<Var> \*>(array->GetHead());

Assert(head->left == 0);

const uint32 offset = index;

if(offset < head->length)

{

const Var element = head->elements[offset];

if(!SparseArraySegment<Var>::IsMissingItem(&element))

{

// Successful fastpath

return element;

}

}

ldElemInfo->neededHelperCall = true;

}

SparseArraySegment<Var> \*seg = (SparseArraySegment<Var>\*)array->GetLastUsedSegment();

if ((uint32) index < seg->left)

{

break;

}

uint32 index2 = index - seg->left;

if (index2 < seg->length)

{

Var elem = seg->elements[index2];

if (elem != SparseArraySegment<Var>::GetMissingItem())

{

// Successful fastpath

return elem;

}

}

} while(false);

if(ldElemInfo)

{

ldElemInfo->neededHelperCall = true;

}

return JavascriptOperators::OP\_GetElementI(array, varIndex, scriptContext);

}

void ProfilingHelpers::ProfiledStElem\_DefaultFlags(

const Var base,

const Var varIndex,

const Var value,

FunctionBody \*const functionBody,

const ProfileId profileId)

{

ProfiledStElem(base, varIndex, value, functionBody, profileId, PropertyOperation\_None);

}

void ProfilingHelpers::ProfiledStElem(

const Var base,

const Var varIndex,

const Var value,

FunctionBody \*const functionBody,

const ProfileId profileId,

const PropertyOperationFlags flags)

{

Assert(base);

Assert(varIndex);

Assert(value);

Assert(functionBody);

Assert(profileId != Constants::NoProfileId);

StElemInfo stElemInfo;

// Only enable fast path if the javascript array is not cross site

const bool isJsArray = !TaggedNumber::Is(base) && VirtualTableInfo<JavascriptArray>::HasVirtualTable(base);

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

const bool fastPath = isJsArray && !JavascriptOperators::SetElementMayHaveImplicitCalls(scriptContext);

if(fastPath)

{

JavascriptArray \*const array = JavascriptArray::FromVar(base);

stElemInfo.arrayType = ValueType::FromArray(ObjectType::Array, array, TypeIds\_Array).ToLikely();

stElemInfo.createdMissingValue = array->HasNoMissingValues();

ProfiledStElem\_FastPath(array, varIndex, value, scriptContext, flags, &stElemInfo);

stElemInfo.createdMissingValue &= !array->HasNoMissingValues();

functionBody->GetDynamicProfileInfo()->RecordElementStore(functionBody, profileId, stElemInfo);

return;

}

JavascriptArray \*array;

bool isObjectWithArray;

TypeId arrayTypeId;

if(isJsArray)

{

array = JavascriptArray::FromVar(base);

isObjectWithArray = false;

arrayTypeId = TypeIds\_Array;

}

else

{

array = JavascriptArray::GetArrayForArrayOrObjectWithArray(base, &isObjectWithArray, &arrayTypeId);

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(base);

#endif

do // while(false)

{

// The fast path is only for JavascriptArray and doesn't cover native arrays, objects with internal arrays, or typed

// arrays, but we still need to profile the array

uint32 length;

uint32 headSegmentLength;

if(array)

{

stElemInfo.arrayType =

(

isObjectWithArray

? ValueType::FromObjectArray(array)

: ValueType::FromArray(ObjectType::Array, array, arrayTypeId)

).ToLikely();

stElemInfo.createdMissingValue = array->HasNoMissingValues();

length = array->GetLength();

SparseArraySegmentBase \*const head = array->GetHead();

Assert(head->left == 0);

headSegmentLength = head->length;

}

else if(TypedArrayBase::TryGetLengthForOptimizedTypedArray(base, &headSegmentLength, &arrayTypeId))

{

length = headSegmentLength;

bool isVirtual = (VirtualTableInfoBase::GetVirtualTable(base) == ValueType::GetVirtualTypedArrayVtable(arrayTypeId));

stElemInfo.arrayType = ValueType::FromTypeId(arrayTypeId, isVirtual).ToLikely();

}

else

{

break;

}

if(!TaggedInt::Is(varIndex))

{

stElemInfo.neededHelperCall = true;

break;

}

const int32 index = TaggedInt::ToInt32(varIndex);

if(index < 0)

{

stElemInfo.neededHelperCall = true;

break;

}

const uint32 offset = index;

if(offset >= headSegmentLength)

{

stElemInfo.storedOutsideHeadSegmentBounds = true;

if(!isObjectWithArray && offset >= length)

{

stElemInfo.storedOutsideArrayBounds = true;

}

break;

}

if(array && array->IsMissingHeadSegmentItem(offset))

{

stElemInfo.filledMissingValue = true;

}

} while(false);

JavascriptOperators::OP\_SetElementI(base, varIndex, value, scriptContext, flags);

if(!stElemInfo.GetArrayType().IsUninitialized())

{

if(array)

{

stElemInfo.createdMissingValue &= !array->HasNoMissingValues();

}

functionBody->GetDynamicProfileInfo()->RecordElementStore(functionBody, profileId, stElemInfo);

return;

}

functionBody->GetDynamicProfileInfo()->RecordElementStoreAsProfiled(functionBody, profileId);

}

void ProfilingHelpers::ProfiledStElem\_FastPath(

JavascriptArray \*const array,

const Var varIndex,

const Var value,

ScriptContext \*const scriptContext,

const PropertyOperationFlags flags,

StElemInfo \*const stElemInfo)

{

Assert(array);

Assert(varIndex);

Assert(value);

Assert(scriptContext);

Assert(!JavascriptOperators::SetElementMayHaveImplicitCalls(scriptContext));

do // while(false)

{

if (!TaggedInt::Is(varIndex))

{

break;

}

int32 index = TaggedInt::ToInt32(varIndex);

if (index < 0)

{

break;

}

if(stElemInfo)

{

SparseArraySegmentBase \*const head = array->GetHead();

Assert(head->left == 0);

const uint32 offset = index;

if(offset >= head->length)

{

stElemInfo->storedOutsideHeadSegmentBounds = true;

if(offset >= array->GetLength())

{

stElemInfo->storedOutsideArrayBounds = true;

}

}

if(offset < head->size)

{

array->DirectProfiledSetItemInHeadSegmentAt(offset, value, stElemInfo);

return;

}

}

SparseArraySegment<Var>\* lastUsedSeg = (SparseArraySegment<Var>\*)array->GetLastUsedSegment();

if (lastUsedSeg == NULL ||

(uint32) index < lastUsedSeg->left)

{

break;

}

uint32 index2 = index - lastUsedSeg->left;

if (index2 < lastUsedSeg->size)

{

// Successful fastpath

array->DirectSetItemInLastUsedSegmentAt(index2, value);

return;

}

} while(false);

if(stElemInfo)

{

stElemInfo->neededHelperCall = true;

}

JavascriptOperators::OP\_SetElementI(array, varIndex, value, scriptContext, flags);

}

JavascriptArray \*ProfilingHelpers::ProfiledNewScArray(

const uint length,

FunctionBody \*const functionBody,

const ProfileId profileId)

{

Assert(functionBody);

Assert(profileId != Constants::NoProfileId);

// Not creating native array here if the function is unoptimized, because it turns out to be tricky to

// get the initialization right if GlobOpt doesn't give us bailout. It's possible, but we should see

// a use case before spending time on it.

ArrayCallSiteInfo \*const arrayInfo =

functionBody->GetDynamicProfileInfo()->GetArrayCallSiteInfo(functionBody, profileId);

Assert(arrayInfo);

if (length > SparseArraySegmentBase::INLINE\_CHUNK\_SIZE || (functionBody->GetHasTry() && PHASE\_OFF((Js::OptimizeTryCatchPhase), functionBody)))

{

arrayInfo->SetIsNotNativeArray();

}

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

JavascriptArray \*array;

if (arrayInfo->IsNativeIntArray())

{

JavascriptNativeIntArray \*const intArray = scriptContext->GetLibrary()->CreateNativeIntArrayLiteral(length);

Recycler \*recycler = scriptContext->GetRecycler();

intArray->SetArrayCallSite(profileId, recycler->CreateWeakReferenceHandle(functionBody));

array = intArray;

}

else if (arrayInfo->IsNativeFloatArray())

{

JavascriptNativeFloatArray \*const floatArray = scriptContext->GetLibrary()->CreateNativeFloatArrayLiteral(length);

Recycler \*recycler = scriptContext->GetRecycler();

floatArray->SetArrayCallSite(profileId, recycler->CreateWeakReferenceHandle(functionBody));

array = floatArray;

}

else

{

array = scriptContext->GetLibrary()->CreateArrayLiteral(length);

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

array->CheckForceES5Array();

#endif

return array;

}

Var ProfilingHelpers::ProfiledNewScObjArray\_Jit(

const Var callee,

void \*const framePointer,

const ProfileId profileId,

const ProfileId arrayProfileId,

CallInfo callInfo,

...)

{

ARGUMENTS(args, callInfo);

return

ProfiledNewScObjArray(

callee,

args,

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject),

profileId,

arrayProfileId);

}

Var ProfilingHelpers::ProfiledNewScObjArraySpread\_Jit(

const Js::AuxArray<uint32> \*spreadIndices,

const Var callee,

void \*const framePointer,

const ProfileId profileId,

const ProfileId arrayProfileId,

CallInfo callInfo,

...)

{

ARGUMENTS(args, callInfo);

Js::ScriptFunction \*function = ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

ScriptContext\* scriptContext = function->GetScriptContext();

// GetSpreadSize ensures that spreadSize < 2^24

uint32 spreadSize = 0;

if (spreadIndices != nullptr)

{

Arguments outArgs(CallInfo(args.Info.Flags, 0), nullptr);

spreadSize = JavascriptFunction::GetSpreadSize(args, spreadIndices, scriptContext);

Assert(spreadSize == (((1 << 24) - 1) & spreadSize));

// Allocate room on the stack for the spread args.

outArgs.Info.Count = spreadSize;

const unsigned STACK\_ARGS\_ALLOCA\_THRESHOLD = 8; // Number of stack args we allow before using \_alloca

Var stackArgs[STACK\_ARGS\_ALLOCA\_THRESHOLD];

size\_t outArgsSize = 0;

if (outArgs.Info.Count > STACK\_ARGS\_ALLOCA\_THRESHOLD)

{

PROBE\_STACK(scriptContext, outArgs.Info.Count \* sizeof(Var) + Js::Constants::MinStackDefault); // args + function call

outArgsSize = outArgs.Info.Count \* sizeof(Var);

outArgs.Values = (Var\*)\_alloca(outArgsSize);

}

else

{

outArgs.Values = stackArgs;

outArgsSize = STACK\_ARGS\_ALLOCA\_THRESHOLD \* sizeof(Var);

ZeroMemory(outArgs.Values, outArgsSize); // We may not use all of the elements

}

JavascriptFunction::SpreadArgs(args, outArgs, spreadIndices, scriptContext);

return

ProfiledNewScObjArray(

callee,

outArgs,

function,

profileId,

arrayProfileId);

}

else

{

return

ProfiledNewScObjArray(

callee,

args,

function,

profileId,

arrayProfileId);

}

}

Var ProfilingHelpers::ProfiledNewScObjArray(

const Var callee,

const Arguments args,

ScriptFunction \*const caller,

const ProfileId profileId,

const ProfileId arrayProfileId)

{

Assert(callee);

Assert(args.Info.Count != 0);

Assert(caller);

Assert(profileId != Constants::NoProfileId);

Assert(arrayProfileId != Constants::NoProfileId);

FunctionBody \*const callerFunctionBody = caller->GetFunctionBody();

DynamicProfileInfo \*const profileInfo = callerFunctionBody->GetDynamicProfileInfo();

ArrayCallSiteInfo \*const arrayInfo = profileInfo->GetArrayCallSiteInfo(callerFunctionBody, arrayProfileId);

Assert(arrayInfo);

ScriptContext \*const scriptContext = callerFunctionBody->GetScriptContext();

FunctionInfo \*const calleeFunctionInfo = JavascriptOperators::GetConstructorFunctionInfo(callee, scriptContext);

if (calleeFunctionInfo != &JavascriptArray::EntryInfo::NewInstance)

{

// It may be worth checking the object that we actually got back from the ctor, but

// we should at least not keep bailing out at this call site.

arrayInfo->SetIsNotNativeArray();

return ProfiledNewScObject(callee, args, callerFunctionBody, profileId);

}

profileInfo->RecordCallSiteInfo(

callerFunctionBody,

profileId,

calleeFunctionInfo,

caller,

args.Info.Count,

true);

args.Values[0] = nullptr;

Var array;

if (arrayInfo->IsNativeIntArray())

{

array = JavascriptNativeIntArray::NewInstance(RecyclableObject::FromVar(callee), args);

if (VirtualTableInfo<JavascriptNativeIntArray>::HasVirtualTable(array))

{

JavascriptNativeIntArray \*const intArray = static\_cast<JavascriptNativeIntArray \*>(array);

intArray->SetArrayCallSite(arrayProfileId, scriptContext->GetRecycler()->CreateWeakReferenceHandle(callerFunctionBody));

}

else

{

arrayInfo->SetIsNotNativeIntArray();

if (VirtualTableInfo<JavascriptNativeFloatArray>::HasVirtualTable(array))

{

JavascriptNativeFloatArray \*const floatArray = static\_cast<JavascriptNativeFloatArray \*>(array);

floatArray->SetArrayCallSite(arrayProfileId, scriptContext->GetRecycler()->CreateWeakReferenceHandle(callerFunctionBody));

}

else

{

arrayInfo->SetIsNotNativeArray();

}

}

}

else if (arrayInfo->IsNativeFloatArray())

{

array = JavascriptNativeFloatArray::NewInstance(RecyclableObject::FromVar(callee), args);

if (VirtualTableInfo<JavascriptNativeFloatArray>::HasVirtualTable(array))

{

JavascriptNativeFloatArray \*const floatArray = static\_cast<JavascriptNativeFloatArray \*>(array);

floatArray->SetArrayCallSite(arrayProfileId, scriptContext->GetRecycler()->CreateWeakReferenceHandle(callerFunctionBody));

}

else

{

arrayInfo->SetIsNotNativeArray();

}

}

else

{

array = JavascriptArray::NewInstance(RecyclableObject::FromVar(callee), args);

}

return CrossSite::MarshalVar(scriptContext, array);

}

Var ProfilingHelpers::ProfiledNewScObject(

const Var callee,

const Arguments args,

FunctionBody \*const callerFunctionBody,

const ProfileId profileId,

const InlineCacheIndex inlineCacheIndex,

const Js::AuxArray<uint32> \*spreadIndices)

{

Assert(callee);

Assert(args.Info.Count != 0);

Assert(callerFunctionBody);

Assert(profileId != Constants::NoProfileId);

ScriptContext \*const scriptContext = callerFunctionBody->GetScriptContext();

if(!TaggedNumber::Is(callee))

{

const auto calleeObject = JavascriptOperators::GetCallableObjectOrThrow(callee, scriptContext);

const auto calleeFunctionInfo =

calleeObject->GetTypeId() == TypeIds\_Function

? JavascriptFunction::FromVar(calleeObject)->GetFunctionInfo()

: nullptr;

callerFunctionBody->GetDynamicProfileInfo()->RecordCallSiteInfo(

callerFunctionBody,

profileId,

calleeFunctionInfo,

calleeFunctionInfo ? static\_cast<JavascriptFunction \*>(calleeObject) : nullptr,

args.Info.Count,

true,

inlineCacheIndex);

}

return JavascriptOperators::NewScObject(callee, args, scriptContext, spreadIndices);

}

void ProfilingHelpers::ProfileLdSlot(const Var value, FunctionBody \*const functionBody, const ProfileId profileId)

{

Assert(value);

Assert(functionBody);

Assert(profileId != Constants::NoProfileId);

functionBody->GetDynamicProfileInfo()->RecordSlotLoad(functionBody, profileId, value);

}

Var ProfilingHelpers::ProfiledLdFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

return

ProfiledLdFld<false, false, false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

scriptFunction->GetFunctionBody(),

instance);

}

Var ProfilingHelpers::ProfiledLdSuperFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

void \*const framePointer,

const Var thisInstance)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

return

ProfiledLdFld<false, false, false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

scriptFunction->GetFunctionBody(),

thisInstance);

}

Var ProfilingHelpers::ProfiledLdFldForTypeOf\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

return ProfiledLdFldForTypeOf<false, false, false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

scriptFunction->GetFunctionBody());

}

Var ProfilingHelpers::ProfiledLdFld\_CallApplyTarget\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

return

ProfiledLdFld<false, false, true>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

scriptFunction->GetFunctionBody(),

instance);

}

Var ProfilingHelpers::ProfiledLdMethodFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

return

ProfiledLdFld<false, true, false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

scriptFunction->GetFunctionBody(),

instance);

}

Var ProfilingHelpers::ProfiledLdRootFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

return

ProfiledLdFld<true, false, false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

scriptFunction->GetFunctionBody(),

instance);

}

Var ProfilingHelpers::ProfiledLdRootFldForTypeOf\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

return ProfiledLdFldForTypeOf<true, false, false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

scriptFunction->GetFunctionBody());

}

Var ProfilingHelpers::ProfiledLdRootMethodFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

return

ProfiledLdFld<true, true, false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

scriptFunction->GetFunctionBody(),

instance);

}

template<bool Root, bool Method, bool CallApplyTarget>

Var ProfilingHelpers::ProfiledLdFld(

const Var instance,

const PropertyId propertyId,

InlineCache \*const inlineCache,

const InlineCacheIndex inlineCacheIndex,

FunctionBody \*const functionBody,

const Var thisInstance)

{

Assert(instance);

Assert(thisInstance);

Assert(propertyId != Constants::NoProperty);

Assert(inlineCache);

Assert(functionBody);

Assert(inlineCacheIndex < functionBody->GetInlineCacheCount());

Assert(!Root || inlineCacheIndex >= functionBody->GetRootObjectLoadInlineCacheStart());

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

DynamicProfileInfo \*const dynamicProfileInfo = functionBody->GetDynamicProfileInfo();

Var value;

FldInfoFlags fldInfoFlags = FldInfo\_NoInfo;

if (Root || (RecyclableObject::Is(instance) && RecyclableObject::Is(thisInstance)))

{

RecyclableObject \*const object = RecyclableObject::FromVar(instance);

RecyclableObject \*const thisObject = RecyclableObject::FromVar(thisInstance);

if (!Root && Method && (propertyId == PropertyIds::apply || propertyId == PropertyIds::call) && ScriptFunction::Is(object))

{

// If the property being loaded is "apply"/"call", make an optimistic assumption that apply/call is not overridden and

// undefer the function right here if it was defer parsed before. This is required so that the load of "apply"/"call"

// happens from the same "type". Otherwise, we will have a polymorphic cache for load of "apply"/"call".

ScriptFunction \*fn = ScriptFunction::FromVar(object);

if (fn->GetType()->GetEntryPoint() == JavascriptFunction::DeferredParsingThunk)

{

JavascriptFunction::DeferredParse(&fn);

}

}

PropertyCacheOperationInfo operationInfo;

PropertyValueInfo propertyValueInfo;

PropertyValueInfo::SetCacheInfo(&propertyValueInfo, functionBody, inlineCache, inlineCacheIndex, true);

if (!CacheOperators::TryGetProperty<true, true, true, !Root && !Method, true, !Root, true, false, true>(

thisObject,

Root,

object,

propertyId,

&value,

scriptContext,

&operationInfo,

&propertyValueInfo))

{

const auto PatchGetValue = &JavascriptOperators::PatchGetValueWithThisPtrNoFastPath;

const auto PatchGetRootValue = &JavascriptOperators::PatchGetRootValueNoFastPath\_Var;

const auto PatchGetMethod = &JavascriptOperators::PatchGetMethodNoFastPath;

const auto PatchGetRootMethod = &JavascriptOperators::PatchGetRootMethodNoFastPath\_Var;

const auto PatchGet =

Root

? Method ? PatchGetRootMethod : PatchGetRootValue

: PatchGetMethod ;

value = (!Root && !Method) ? PatchGetValue(functionBody, inlineCache, inlineCacheIndex, object, propertyId, thisObject) :

PatchGet(functionBody, inlineCache, inlineCacheIndex, object, propertyId);

CacheOperators::PretendTryGetProperty<true, false>(object->GetType(), &operationInfo, &propertyValueInfo);

}

else if (!Root && !Method)

{

// Inline cache hit. oldflags must match the new ones. If not there is mark it as polymorphic as there is likely

// a bailout to interpreter and change in the inline cache type.

const FldInfoFlags oldflags = dynamicProfileInfo->GetFldInfo(functionBody, inlineCacheIndex)->flags;

if ((oldflags != FldInfo\_NoInfo) &&

!(oldflags & DynamicProfileInfo::FldInfoFlagsFromCacheType(operationInfo.cacheType)))

{

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_Polymorphic);

}

}

if (!Root && operationInfo.isPolymorphic)

{

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_Polymorphic);

}

fldInfoFlags =

DynamicProfileInfo::MergeFldInfoFlags(

fldInfoFlags,

DynamicProfileInfo::FldInfoFlagsFromCacheType(operationInfo.cacheType));

fldInfoFlags =

DynamicProfileInfo::MergeFldInfoFlags(

fldInfoFlags,

DynamicProfileInfo::FldInfoFlagsFromSlotType(operationInfo.slotType));

if (!Method)

{

UpdateFldInfoFlagsForGetSetInlineCandidate(

object,

fldInfoFlags,

operationInfo.cacheType,

inlineCache,

functionBody);

if (!Root && CallApplyTarget)

{

UpdateFldInfoFlagsForCallApplyInlineCandidate(

object,

fldInfoFlags,

operationInfo.cacheType,

inlineCache,

functionBody);

}

}

}

else

{

Assert(!Root);

const auto PatchGetValue = &JavascriptOperators::PatchGetValue<false, InlineCache>;

const auto PatchGetMethod = &JavascriptOperators::PatchGetMethod<false, InlineCache>;

const auto PatchGet = Method ? PatchGetMethod : PatchGetValue;

value = PatchGet(functionBody, inlineCache, inlineCacheIndex, instance, propertyId);

}

dynamicProfileInfo->RecordFieldAccess(functionBody, inlineCacheIndex, value, fldInfoFlags);

return value;

}

template<bool Root, bool Method, bool CallApplyTarget>

Var ProfilingHelpers::ProfiledLdFldForTypeOf(

const Var instance,

const PropertyId propertyId,

InlineCache \*const inlineCache,

const InlineCacheIndex inlineCacheIndex,

FunctionBody \*const functionBody)

{

Var val = nullptr;

ScriptContext \*scriptContext = functionBody->GetScriptContext();

BEGIN\_PROFILED\_TYPEOF\_ERROR\_HANDLER(scriptContext);

val = ProfiledLdFld<Root, Method, CallApplyTarget>(

instance,

propertyId,

inlineCache,

inlineCacheIndex,

functionBody,

instance);

END\_PROFILED\_TYPEOF\_ERROR\_HANDLER(scriptContext, val, functionBody, inlineCacheIndex);

return val;

}

void ProfilingHelpers::ProfiledStFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

const Var value,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

ProfiledStFld<false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

value,

PropertyOperation\_None,

scriptFunction,

instance);

}

void ProfilingHelpers::ProfiledStSuperFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

const Var value,

void \*const framePointer,

const Var thisInstance)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

ProfiledStFld<false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

value,

PropertyOperation\_None,

scriptFunction,

thisInstance);

}

void ProfilingHelpers::ProfiledStFld\_Strict\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

const Var value,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

ProfiledStFld<false>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

value,

PropertyOperation\_StrictMode,

scriptFunction,

instance);

}

void ProfilingHelpers::ProfiledStRootFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

const Var value,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

ProfiledStFld<true>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

value,

PropertyOperation\_Root,

scriptFunction,

instance);

}

void ProfilingHelpers::ProfiledStRootFld\_Strict\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

const Var value,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

ProfiledStFld<true>(

instance,

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

value,

PropertyOperation\_StrictModeRoot,

scriptFunction,

instance);

}

template<bool Root>

void ProfilingHelpers::ProfiledStFld(

const Var instance,

const PropertyId propertyId,

InlineCache \*const inlineCache,

const InlineCacheIndex inlineCacheIndex,

const Var value,

const PropertyOperationFlags flags,

ScriptFunction \*const scriptFunction,

const Var thisInstance)

{

Assert(instance);

Assert(thisInstance);

Assert(propertyId != Constants::NoProperty);

Assert(inlineCache);

Assert(scriptFunction);

FunctionBody \*const functionBody = scriptFunction->GetFunctionBody();

Assert(inlineCacheIndex < functionBody->GetInlineCacheCount());

Assert(value);

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

FldInfoFlags fldInfoFlags = FldInfo\_NoInfo;

if(Root || (RecyclableObject::Is(instance) && RecyclableObject::Is(thisInstance)))

{

RecyclableObject \*const object = RecyclableObject::FromVar(instance);

RecyclableObject \*const thisObject = RecyclableObject::FromVar(thisInstance);

PropertyCacheOperationInfo operationInfo;

PropertyValueInfo propertyValueInfo;

PropertyValueInfo::SetCacheInfo(&propertyValueInfo, functionBody, inlineCache, inlineCacheIndex, true);

if(!CacheOperators::TrySetProperty<true, true, true, true, !Root, true, false, true>(

thisObject,

Root,

propertyId,

value,

scriptContext,

flags,

&operationInfo,

&propertyValueInfo))

{

Type \*const oldType = object->GetType();

if (Root)

{

JavascriptOperators::PatchPutRootValueNoFastPath(functionBody, inlineCache, inlineCacheIndex, object, propertyId, value, flags);

}

else

{

JavascriptOperators::PatchPutValueWithThisPtrNoFastPath(functionBody, inlineCache, inlineCacheIndex, object, propertyId, value, thisObject, flags);

}

CacheOperators::PretendTrySetProperty<true, false>(

object->GetType(),

oldType,

&operationInfo,

&propertyValueInfo);

}

// Only make the field polymorphic if we are not using the root object inline cache

if(operationInfo.isPolymorphic && inlineCacheIndex < functionBody->GetRootObjectStoreInlineCacheStart())

{

// should not be a load inline cache

Assert(inlineCacheIndex < functionBody->GetRootObjectLoadInlineCacheStart());

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_Polymorphic);

}

fldInfoFlags =

DynamicProfileInfo::MergeFldInfoFlags(

fldInfoFlags,

DynamicProfileInfo::FldInfoFlagsFromCacheType(operationInfo.cacheType));

fldInfoFlags =

DynamicProfileInfo::MergeFldInfoFlags(

fldInfoFlags,

DynamicProfileInfo::FldInfoFlagsFromSlotType(operationInfo.slotType));

UpdateFldInfoFlagsForGetSetInlineCandidate(

object,

fldInfoFlags,

operationInfo.cacheType,

inlineCache,

functionBody);

if(scriptFunction->GetConstructorCache()->NeedsUpdateAfterCtor())

{

// This function has only 'this' statements and is being used as a constructor. When the constructor exits, the

// function object's constructor cache will be updated with the type produced by the constructor. From that

// point on, when the same function object is used as a constructor, the a new object with the final type will

// be created. Whatever is stored in the inline cache currently will cause cache misses after the constructor

// cache update. So, just clear it now so that the caches won't be flagged as polymorphic.

inlineCache->Clear();

}

}

else

{

JavascriptOperators::PatchPutValueNoLocalFastPath<false>(

functionBody,

inlineCache,

inlineCacheIndex,

instance,

propertyId,

value,

flags);

}

functionBody->GetDynamicProfileInfo()->RecordFieldAccess(functionBody, inlineCacheIndex, nullptr, fldInfoFlags);

}

void ProfilingHelpers::ProfiledInitFld\_Jit(

const Var instance,

const PropertyId propertyId,

const InlineCacheIndex inlineCacheIndex,

const Var value,

void \*const framePointer)

{

ScriptFunction \*const scriptFunction =

ScriptFunction::FromVar(JavascriptCallStackLayout::FromFramePointer(framePointer)->functionObject);

ProfiledInitFld(

RecyclableObject::FromVar(instance),

propertyId,

GetInlineCache(scriptFunction, inlineCacheIndex),

inlineCacheIndex,

value,

scriptFunction->GetFunctionBody());

}

void ProfilingHelpers::ProfiledInitFld(

RecyclableObject \*const object,

const PropertyId propertyId,

InlineCache \*const inlineCache,

const InlineCacheIndex inlineCacheIndex,

const Var value,

FunctionBody \*const functionBody)

{

Assert(object);

Assert(propertyId != Constants::NoProperty);

Assert(inlineCache);

Assert(functionBody);

Assert(inlineCacheIndex < functionBody->GetInlineCacheCount());

Assert(value);

ScriptContext \*const scriptContext = functionBody->GetScriptContext();

FldInfoFlags fldInfoFlags = FldInfo\_NoInfo;

PropertyCacheOperationInfo operationInfo;

PropertyValueInfo propertyValueInfo;

PropertyValueInfo::SetCacheInfo(&propertyValueInfo, functionBody, inlineCache, inlineCacheIndex, true);

if(!CacheOperators::TrySetProperty<true, true, true, true, true, true, false, true>(

object,

false,

propertyId,

value,

scriptContext,

PropertyOperation\_None,

&operationInfo,

&propertyValueInfo))

{

Type \*const oldType = object->GetType();

JavascriptOperators::PatchInitValueNoFastPath(

functionBody,

inlineCache,

inlineCacheIndex,

object,

propertyId,

value);

CacheOperators::PretendTrySetProperty<true, false>(object->GetType(), oldType, &operationInfo, &propertyValueInfo);

}

// Only make the field polymorphic if the we are not using the root object inline cache

if(operationInfo.isPolymorphic && inlineCacheIndex < functionBody->GetRootObjectStoreInlineCacheStart())

{

// should not be a load inline cache

Assert(inlineCacheIndex < functionBody->GetRootObjectLoadInlineCacheStart());

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_Polymorphic);

}

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, DynamicProfileInfo::FldInfoFlagsFromCacheType(operationInfo.cacheType));

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, DynamicProfileInfo::FldInfoFlagsFromSlotType(operationInfo.slotType));

functionBody->GetDynamicProfileInfo()->RecordFieldAccess(functionBody, inlineCacheIndex, nullptr, fldInfoFlags);

}

void ProfilingHelpers::UpdateFldInfoFlagsForGetSetInlineCandidate(

RecyclableObject \*const object,

FldInfoFlags &fldInfoFlags,

const CacheType cacheType,

InlineCache \*const inlineCache,

FunctionBody \*const functionBody)

{

RecyclableObject \*callee = nullptr;

if((cacheType & (CacheType\_Getter | CacheType\_Setter)) &&

inlineCache->GetGetterSetter(object->GetType(), &callee))

{

const bool canInline = functionBody->GetDynamicProfileInfo()->RecordLdFldCallSiteInfo(functionBody, callee, false /\*callApplyTarget\*/);

if(canInline)

{

//updates this fldInfoFlags passed by reference.

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_InlineCandidate);

}

}

}

void ProfilingHelpers::UpdateFldInfoFlagsForCallApplyInlineCandidate(

RecyclableObject \*const object,

FldInfoFlags &fldInfoFlags,

const CacheType cacheType,

InlineCache \*const inlineCache,

FunctionBody \*const functionBody)

{

RecyclableObject \*callee = nullptr;

if(!(fldInfoFlags & FldInfo\_Polymorphic) && inlineCache->GetCallApplyTarget(object, &callee))

{

const bool canInline = functionBody->GetDynamicProfileInfo()->RecordLdFldCallSiteInfo(functionBody, callee, true /\*callApplyTarget\*/);

if(canInline)

{

//updates the fldInfoFlags passed by reference.

fldInfoFlags = DynamicProfileInfo::MergeFldInfoFlags(fldInfoFlags, FldInfo\_InlineCandidate);

}

}

}

InlineCache \*ProfilingHelpers::GetInlineCache(ScriptFunction \*const scriptFunction, const InlineCacheIndex inlineCacheIndex)

{

Assert(scriptFunction);

Assert(inlineCacheIndex < scriptFunction->GetFunctionBody()->GetInlineCacheCount());

return

scriptFunction->GetHasInlineCaches()

? ScriptFunctionWithInlineCache::FromVar(scriptFunction)->GetInlineCache(inlineCacheIndex)

: scriptFunction->GetFunctionBody()->GetInlineCache(inlineCacheIndex);

}

#endif

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

#if ENABLE\_PROFILE\_INFO

class ProfilingHelpers

{

public:

static Var ProfiledLdElem(const Var base, const Var varIndex, FunctionBody \*const functionBody, const ProfileId profileId);

static Var ProfiledLdElem\_FastPath(JavascriptArray \*const array, const Var varIndex, ScriptContext \*const scriptContext, LdElemInfo \*const ldElemInfo = nullptr);

public:

static void ProfiledStElem\_DefaultFlags(const Var base, const Var varIndex, const Var value, FunctionBody \*const functionBody, const ProfileId profileId);

static void ProfiledStElem(const Var base, const Var varIndex, const Var value, FunctionBody \*const functionBody, const ProfileId profileId, const PropertyOperationFlags flags);

static void ProfiledStElem\_FastPath(JavascriptArray \*const array, const Var varIndex, const Var value, ScriptContext \*const scriptContext, const PropertyOperationFlags flags, StElemInfo \*const stElemInfo = nullptr);

public:

static JavascriptArray \*ProfiledNewScArray(const uint length, FunctionBody \*const functionBody, const ProfileId profileId);

public:

static Var ProfiledNewScObjArray\_Jit(const Var callee, void \*const framePointer, const ProfileId profileId, const ProfileId arrayProfileId, CallInfo callInfo, ...);

static Var ProfiledNewScObjArraySpread\_Jit(const Js::AuxArray<uint32> \*spreadIndices, const Var callee, void \*const framePointer, const ProfileId profileId, const ProfileId arrayProfileId, CallInfo callInfo, ...);

static Var ProfiledNewScObjArray(const Var callee, const Arguments args, ScriptFunction \*const caller, const ProfileId profileId, const ProfileId arrayProfileId);

public:

static Var ProfiledNewScObject(const Var callee, const Arguments args, FunctionBody \*const callerFunctionBody, const ProfileId profileId, const InlineCacheIndex inlineCacheIndex = Constants::NoInlineCacheIndex, const Js::AuxArray<uint32> \*spreadIndices = nullptr);

public:

static void ProfileLdSlot(const Var value, FunctionBody \*const functionBody, const ProfileId profileId);

public:

static Var ProfiledLdFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, void \*const framePointer);

static Var ProfiledLdSuperFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, void \*const framePointer, const Var thisInstance);

static Var ProfiledLdFldForTypeOf\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, void \*const framePointer);

static Var ProfiledLdRootFldForTypeOf\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, void \*const framePointer);

static Var ProfiledLdFld\_CallApplyTarget\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, void \*const framePointer);

static Var ProfiledLdMethodFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, void \*const framePointer);

static Var ProfiledLdRootFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, void \*const framePointer);

static Var ProfiledLdRootMethodFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, void \*const framePointer);

template<bool Root, bool Method, bool CallApplyTarget> static Var ProfiledLdFld(const Var instance, const PropertyId propertyId, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FunctionBody \*const functionbody, const Var thisInstance);

template<bool Root, bool Method, bool CallApplyTarget> static Var ProfiledLdFldForTypeOf(const Var instance, const PropertyId propertyId, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, FunctionBody \*const functionbody);

public:

static void ProfiledStFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, const Var value, void \*const framePointer);

static void ProfiledStSuperFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, const Var value, void \*const framePointer, const Var thisInstance);

static void ProfiledStFld\_Strict\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, const Var value, void \*const framePointer);

static void ProfiledStRootFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, const Var value, void \*const framePointer);

static void ProfiledStRootFld\_Strict\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, const Var value, void \*const framePointer);

template<bool Root> static void ProfiledStFld(const Var instance, const PropertyId propertyId, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, const Var value, const PropertyOperationFlags flags, ScriptFunction \*const scriptFunction, const Var thisInstance);

public:

static void ProfiledInitFld\_Jit(const Var instance, const PropertyId propertyId, const InlineCacheIndex inlineCacheIndex, const Var value, void \*const framePointer);

static void ProfiledInitFld(RecyclableObject \*const object, const PropertyId propertyId, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, const Var value, FunctionBody \*const functionBody);

private:

static void UpdateFldInfoFlagsForGetSetInlineCandidate(RecyclableObject \*const object, FldInfoFlags &fldInfoFlags, const CacheType cacheType, InlineCache \*const inlineCache, FunctionBody \*const functionBody);

static void UpdateFldInfoFlagsForCallApplyInlineCandidate(RecyclableObject \*const object, FldInfoFlags &fldInfoFlags, const CacheType cacheType, InlineCache \*const inlineCache, FunctionBody \*const functionBody);

static InlineCache \*GetInlineCache(ScriptFunction \*const scriptFunction, const InlineCacheIndex inlineCacheIndex);

};

#endif

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "Language\ReadOnlyDynamicProfileInfo.h"

#if ENABLE\_PROFILE\_INFO

namespace Js

{

const LdElemInfo \*ReadOnlyDynamicProfileInfo::GetLdElemInfo(FunctionBody \*functionBody, ProfileId ldElemId)

{

Assert(functionBody);

Assert(ldElemId < functionBody->GetProfiledLdElemCount());

// This data is accessed multiple times. Since the original profile data may be changing on the foreground thread,

// the first time it's accessed it will be copied from the original profile data (if we're jitting in the

// background).

if(!ldElemInfo)

{

if(backgroundAllocator)

{

// Jitting in the background

LdElemInfo \*const info = AnewArray(backgroundAllocator, LdElemInfo, functionBody->GetProfiledLdElemCount());

js\_memcpy\_s(

info,

functionBody->GetProfiledLdElemCount() \* sizeof(info[0]),

profileInfo->GetLdElemInfo(),

functionBody->GetProfiledLdElemCount() \* sizeof(info[0]));

ldElemInfo = info;

}

else

{

// Jitting in the foreground

ldElemInfo = profileInfo->GetLdElemInfo();

}

}

return &ldElemInfo[ldElemId];

}

const StElemInfo \*ReadOnlyDynamicProfileInfo::GetStElemInfo(FunctionBody \*functionBody, ProfileId stElemId)

{

Assert(functionBody);

Assert(stElemId < functionBody->GetProfiledStElemCount());

// This data is accessed multiple times. Since the original profile data may be changing on the foreground thread,

// the first time it's accessed it will be copied from the original profile data (if we're jitting in the

// background).

if(!stElemInfo)

{

if(backgroundAllocator)

{

// Jitting in the background

StElemInfo \*const info = AnewArray(backgroundAllocator, StElemInfo, functionBody->GetProfiledStElemCount());

js\_memcpy\_s(

info,

functionBody->GetProfiledStElemCount() \* sizeof(info[0]),

profileInfo->GetStElemInfo(),

functionBody->GetProfiledStElemCount() \* sizeof(info[0]));

stElemInfo = info;

}

else

{

// Jitting in the foreground

stElemInfo = profileInfo->GetStElemInfo();

}

}

return &stElemInfo[stElemId];

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#if ENABLE\_PROFILE\_INFO

namespace Js

{

// Provides a thread-safe, read-only view of a profile. This is the view used

// by codegen. We copy settings that are no thread-safe and we pass-through

// the rest.

// Note that this class' ctor is called on the codegen thread, so if the

// copying itself (as opposed to accesses) ever becomes thread-unsafe then a locking

// scheme would need to be added.

class ReadOnlyDynamicProfileInfo

{

public:

ReadOnlyDynamicProfileInfo(const DynamicProfileInfo \* profileInfo, ArenaAllocator \*const backgroundAllocator) :

profileInfo(profileInfo),

backgroundAllocator(backgroundAllocator),

isAggressiveIntTypeSpecDisabled(false),

isAggressiveIntTypeSpecDisabled\_jitLoopBody(false),

isAggressiveMulIntTypeSpecDisabled(false),

isAggressiveMulIntTypeSpecDisabled\_jitLoopBody(false),

isDivIntTypeSpecDisabled(false),

isDivIntTypeSpecDisabled\_jitLoopBody(false),

isLossyIntTypeSpecDisabled(false),

isTrackCompoundedIntOverflowDisabled(false),

isFloatTypeSpecDisabled(false),

isArrayCheckHoistDisabled(false),

isArrayCheckHoistDisabled\_jitLoopBody(false),

isArrayMissingValueCheckHoistDisabled(false),

isArrayMissingValueCheckHoistDisabled\_jitLoopBody(false),

isJsArraySegmentHoistDisabled(false),

isJsArraySegmentHoistDisabled\_jitLoopBody(false),

isArrayLengthHoistDisabled(false),

isArrayLengthHoistDisabled\_jitLoopBody(false),

isTypedArrayTypeSpecDisabled(false),

isTypedArrayTypeSpecDisabled\_jitLoopBody(false),

isLdLenIntSpecDisabled(false),

isBoundCheckHoistDisabled(false),

isBoundCheckHoistDisabled\_jitLoopBody(false),

isLoopCountBasedBoundCheckHoistDisabled(false),

isLoopCountBasedBoundCheckHoistDisabled\_jitLoopBody(false),

isFloorInliningDisabled(false),

isNoProfileBailoutsDisabled(false),

isSwitchOptDisabled(false),

isEquivalentObjTypeSpecDisabled(false),

isObjTypeSpecDisabled\_jitLoopBody(false),

ldElemInfo(nullptr),

stElemInfo(nullptr)

{

if (profileInfo == nullptr)

{

return;

}

this->isAggressiveIntTypeSpecDisabled = profileInfo->IsAggressiveIntTypeSpecDisabled(false);

this->isAggressiveIntTypeSpecDisabled\_jitLoopBody = profileInfo->IsAggressiveIntTypeSpecDisabled(true);

this->isAggressiveMulIntTypeSpecDisabled = profileInfo->IsAggressiveMulIntTypeSpecDisabled(false);

this->isAggressiveMulIntTypeSpecDisabled\_jitLoopBody = profileInfo->IsAggressiveMulIntTypeSpecDisabled(true);

this->isDivIntTypeSpecDisabled = profileInfo->IsDivIntTypeSpecDisabled(false);

this->isDivIntTypeSpecDisabled\_jitLoopBody = profileInfo->IsDivIntTypeSpecDisabled(true);

this->isLossyIntTypeSpecDisabled = profileInfo->IsLossyIntTypeSpecDisabled();

this->isTrackCompoundedIntOverflowDisabled = profileInfo->IsTrackCompoundedIntOverflowDisabled();

this->isFloatTypeSpecDisabled = profileInfo->IsFloatTypeSpecDisabled();

this->isArrayCheckHoistDisabled = profileInfo->IsArrayCheckHoistDisabled(false);

this->isArrayCheckHoistDisabled\_jitLoopBody = profileInfo->IsArrayCheckHoistDisabled(true);

this->isArrayMissingValueCheckHoistDisabled = profileInfo->IsArrayMissingValueCheckHoistDisabled(false);

this->isArrayMissingValueCheckHoistDisabled\_jitLoopBody = profileInfo->IsArrayMissingValueCheckHoistDisabled(true);

this->isJsArraySegmentHoistDisabled = profileInfo->IsJsArraySegmentHoistDisabled(false);

this->isJsArraySegmentHoistDisabled\_jitLoopBody = profileInfo->IsJsArraySegmentHoistDisabled(true);

this->isArrayLengthHoistDisabled = profileInfo->IsArrayLengthHoistDisabled(false);

this->isArrayLengthHoistDisabled\_jitLoopBody = profileInfo->IsArrayLengthHoistDisabled(true);

this->isTypedArrayTypeSpecDisabled = profileInfo->IsTypedArrayTypeSpecDisabled(false);

this->isTypedArrayTypeSpecDisabled\_jitLoopBody = profileInfo->IsTypedArrayTypeSpecDisabled(true);

this->isLdLenIntSpecDisabled = profileInfo->IsLdLenIntSpecDisabled();

this->isBoundCheckHoistDisabled = profileInfo->IsBoundCheckHoistDisabled(false);

this->isBoundCheckHoistDisabled\_jitLoopBody = profileInfo->IsBoundCheckHoistDisabled(true);

this->isLoopCountBasedBoundCheckHoistDisabled = profileInfo->IsLoopCountBasedBoundCheckHoistDisabled(false);

this->isLoopCountBasedBoundCheckHoistDisabled\_jitLoopBody = profileInfo->IsLoopCountBasedBoundCheckHoistDisabled(true);

this->isFloorInliningDisabled = profileInfo->IsFloorInliningDisabled();

this->isNoProfileBailoutsDisabled = profileInfo->IsNoProfileBailoutsDisabled();

this->isSwitchOptDisabled = profileInfo->IsSwitchOptDisabled();

this->isEquivalentObjTypeSpecDisabled = profileInfo->IsEquivalentObjTypeSpecDisabled();

this->isObjTypeSpecDisabled\_jitLoopBody = profileInfo->IsObjTypeSpecDisabledInJitLoopBody();

}

void OnBackgroundAllocatorReset()

{

// The background allocator was reset, so need to clear any references to data cloned using that allocator

Assert(backgroundAllocator);

ldElemInfo = nullptr;

stElemInfo = nullptr;

}

bool HasProfileInfo() const

{

return this->profileInfo != NULL;

}

const LdElemInfo \*GetLdElemInfo(FunctionBody \*functionBody, ProfileId ldElemId);

const StElemInfo \*GetStElemInfo(FunctionBody \*functionBody, ProfileId stElemId);

ArrayCallSiteInfo \*GetArrayCallSiteInfo(FunctionBody \*functionBody, ProfileId index)

{

return this->profileInfo->GetArrayCallSiteInfo(functionBody, index);

}

FldInfo \* GetFldInfo(FunctionBody\* functionBody, uint fieldAccessId) const

{

return this->profileInfo->GetFldInfo(functionBody, fieldAccessId);

}

ThisInfo GetThisInfo() const

{

return this->profileInfo->GetThisInfo();

}

ValueType GetReturnType(FunctionBody\* functionBody, Js::OpCode opcode, ProfileId callSiteId) const

{

return this->profileInfo->GetReturnType(functionBody, opcode, callSiteId);

}

ValueType GetDivProfileInfo(FunctionBody\* functionBody, ProfileId divideId) const

{

return this->profileInfo->GetDivideResultType(functionBody, divideId);

}

bool IsModulusOpByPowerOf2(FunctionBody\* functionBody, ProfileId moduleId) const

{

return this->profileInfo->IsModulusOpByPowerOf2(functionBody, moduleId);

}

ValueType GetSwitchProfileInfo(FunctionBody\* functionBody, ProfileId switchId) const

{

return this->profileInfo->GetSwitchType(functionBody, switchId);

}

ValueType GetParameterInfo(FunctionBody\* functionBody, ArgSlot index) const

{

return this->profileInfo->GetParameterInfo(functionBody, index);

}

ImplicitCallFlags GetLoopImplicitCallFlags(FunctionBody\* functionBody, uint loopNum) const

{

return this->profileInfo->GetLoopImplicitCallFlags(functionBody, loopNum);

}

ImplicitCallFlags GetImplicitCallFlags() const

{

return this->profileInfo->GetImplicitCallFlags();

}

LoopFlags GetLoopFlags(uint loopNum) const

{

return this->profileInfo->GetLoopFlags(loopNum);

}

bool IsAggressiveIntTypeSpecDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isAggressiveIntTypeSpecDisabled\_jitLoopBody

: this->isAggressiveIntTypeSpecDisabled;

}

void DisableAggressiveIntTypeSpec(const bool isJitLoopBody)

{

this->isAggressiveIntTypeSpecDisabled\_jitLoopBody = true;

if(!isJitLoopBody)

{

this->isAggressiveIntTypeSpecDisabled = true;

}

}

bool IsSwitchOptDisabled() const

{

return this->isSwitchOptDisabled;

}

void DisableSwitchOpt()

{

this->isSwitchOptDisabled = true;

}

bool IsEquivalentObjTypeSpecDisabled() const

{

return this->isEquivalentObjTypeSpecDisabled;

}

void DisableEquivalentObjTypeSpec()

{

this->isEquivalentObjTypeSpecDisabled = true;

}

bool IsObjTypeSpecDisabledInJitLoopBody() const

{

return this->isObjTypeSpecDisabled\_jitLoopBody;

}

void DisableObjTypeSpecInJitLoopBody()

{

this->isObjTypeSpecDisabled\_jitLoopBody = true;

}

bool IsAggressiveMulIntTypeSpecDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isAggressiveMulIntTypeSpecDisabled\_jitLoopBody

: this->isAggressiveMulIntTypeSpecDisabled;

}

void DisableAggressiveMulIntTypeSpec(const bool isJitLoopBody)

{

this->isAggressiveMulIntTypeSpecDisabled\_jitLoopBody = true;

if(!isJitLoopBody)

{

this->isAggressiveMulIntTypeSpecDisabled = true;

}

}

bool IsDivIntTypeSpecDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isDivIntTypeSpecDisabled\_jitLoopBody

: this->isDivIntTypeSpecDisabled;

}

void DisableDivIntTypeSpec(const bool isJitLoopBody)

{

this->isDivIntTypeSpecDisabled\_jitLoopBody = true;

if (!isJitLoopBody)

{

this->isDivIntTypeSpecDisabled = true;

}

}

bool IsLossyIntTypeSpecDisabled() const

{

return this->isLossyIntTypeSpecDisabled;

}

bool IsMemOpDisabled() const

{

return this->profileInfo->IsMemOpDisabled();

}

bool IsTrackCompoundedIntOverflowDisabled() const

{

return this->isTrackCompoundedIntOverflowDisabled;

}

void DisableTrackCompoundedIntOverflow()

{

this->isTrackCompoundedIntOverflowDisabled = true;

}

bool IsFloatTypeSpecDisabled() const

{

return this->isFloatTypeSpecDisabled;

}

bool IsCheckThisDisabled() const

{

return this->profileInfo->IsCheckThisDisabled();

}

bool IsArrayCheckHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isArrayCheckHoistDisabled\_jitLoopBody

: this->isArrayCheckHoistDisabled;

}

bool IsArrayMissingValueCheckHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isArrayMissingValueCheckHoistDisabled\_jitLoopBody

: this->isArrayMissingValueCheckHoistDisabled;

}

bool IsJsArraySegmentHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isJsArraySegmentHoistDisabled\_jitLoopBody

: this->isJsArraySegmentHoistDisabled;

}

bool IsArrayLengthHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isArrayLengthHoistDisabled\_jitLoopBody

: this->isArrayLengthHoistDisabled;

}

bool IsTypedArrayTypeSpecDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isTypedArrayTypeSpecDisabled\_jitLoopBody

: this->isTypedArrayTypeSpecDisabled;

}

bool IsLdLenIntSpecDisabled() const

{

return this->isLdLenIntSpecDisabled;

}

bool IsBoundCheckHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isBoundCheckHoistDisabled\_jitLoopBody

: this->isBoundCheckHoistDisabled;

}

bool IsLoopCountBasedBoundCheckHoistDisabled(const bool isJitLoopBody) const

{

return

isJitLoopBody

? this->isLoopCountBasedBoundCheckHoistDisabled\_jitLoopBody

: this->isLoopCountBasedBoundCheckHoistDisabled;

}

bool IsFloorInliningDisabled() const

{

return this->isFloorInliningDisabled;

}

bool IsNoProfileBailoutsDisabled() const

{

return this->isNoProfileBailoutsDisabled;

}

private:

const DynamicProfileInfo \* profileInfo;

ArenaAllocator \*const backgroundAllocator; // null if the work item is being jitted in the foreground

// These settings need to be copied because they are used at multiple points in the globopt,

// and the readings need to be consistent.

bool isAggressiveIntTypeSpecDisabled : 1;

bool isAggressiveIntTypeSpecDisabled\_jitLoopBody : 1;

bool isAggressiveMulIntTypeSpecDisabled : 1;

bool isAggressiveMulIntTypeSpecDisabled\_jitLoopBody : 1;

bool isDivIntTypeSpecDisabled : 1;

bool isDivIntTypeSpecDisabled\_jitLoopBody : 1;

bool isLossyIntTypeSpecDisabled : 1;

bool isTrackCompoundedIntOverflowDisabled : 1;

bool isFloatTypeSpecDisabled : 1;

bool isArrayCheckHoistDisabled : 1;

bool isArrayCheckHoistDisabled\_jitLoopBody : 1;

bool isArrayMissingValueCheckHoistDisabled : 1;

bool isArrayMissingValueCheckHoistDisabled\_jitLoopBody : 1;

bool isJsArraySegmentHoistDisabled : 1;

bool isJsArraySegmentHoistDisabled\_jitLoopBody : 1;

bool isArrayLengthHoistDisabled : 1;

bool isArrayLengthHoistDisabled\_jitLoopBody : 1;

bool isTypedArrayTypeSpecDisabled : 1;

bool isTypedArrayTypeSpecDisabled\_jitLoopBody : 1;

bool isLdLenIntSpecDisabled : 1;

bool isBoundCheckHoistDisabled : 1;

bool isBoundCheckHoistDisabled\_jitLoopBody : 1;

bool isLoopCountBasedBoundCheckHoistDisabled : 1;

bool isLoopCountBasedBoundCheckHoistDisabled\_jitLoopBody : 1;

bool isFloorInliningDisabled : 1;

bool isNoProfileBailoutsDisabled : 1;

bool isSwitchOptDisabled : 1;

bool isEquivalentObjTypeSpecDisabled : 1;

bool isObjTypeSpecDisabled\_jitLoopBody : 1;

const LdElemInfo \*ldElemInfo;

const StElemInfo \*stElemInfo;

// Other settings are safe to be accessed concurrently. If that changes then they need

// to be copied.

};

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "Parser.h"

#include "Runtime.h"

#include "Language\AsmJsTypes.h"

#include "Language\AsmJsUtils.h"

#include "Language\AsmJsLink.h"

#include "Language\AsmJsModule.h"

#include "Language\AsmJs.h"

#ifdef ASMJS\_PLAT

#include "Language\AsmJSJitTemplate.h"

#include "Language\AsmJSEncoder.h"

#include "Language\AsmJSCodeGenerator.h"

#endif

#include "Language\FunctionCodeGenJitTimeData.h"

#include "Language\ProfilingHelpers.h"

#include "Language\CacheOperators.h"

#include "Language\JavascriptMathOperators.h"

#include "Language\JavascriptStackWalker.h"

#ifdef DYNAMIC\_PROFILE\_STORAGE

#include "Language\DynamicProfileStorage.h"

#endif

#include "Language\SourceDynamicProfileManager.h"

#include "Base\EtwTrace.h"

#include "Library\ArgumentsObject.h"

#include "Types\TypePropertyCache.h"

#include "Library\JavascriptVariantDate.h"

#include "Library\JavascriptProxy.h"

#include "Library\JavascriptSymbol.h"

#include "Library\JavascriptSymbolObject.h"

#include "Library\JavascriptGenerator.h"

#include "Library\StackScriptFunction.h"

#include "Library\HostObjectBase.h"

#ifdef ENABLE\_MUTATION\_BREAKPOINT

// REVIEW: ChakraCore Dependency

#include "activdbg\_private.h"

#include "Debug\MutationBreakpoint.h"

#endif

// SIMD\_JS

// SIMD types

#include "Library\JavascriptSIMDFloat32x4.h"

#include "Library\JavascriptSIMDFloat64x2.h"

#include "Library\JavascriptSIMDInt32x4.h"

#include "Library\JavascriptSIMDInt8x16.h"

// SIMD operations

#include "Language\SIMDFloat32x4Operation.h"

#include "Language\SIMDFloat64x2Operation.h"

#include "Language\SIMDInt32x4Operation.h"

#include "Language\SIMDInt8x16Operation.h"

#include "Language\SIMDUtils.h"

// SIMD libs

#include "Library\SIMDFloat32x4Lib.h"

#include "Library\SIMDFloat64x2Lib.h"

#include "Library\SIMDInt32x4Lib.h"

#include "Library\SIMDInt8x16Lib.h"

#include "Debug\DebuggingFlags.h"

#include "Debug\DiagProbe.h"

#include "Debug\DebugManager.h"

#include "Debug\ProbeContainer.h"

#include "Debug\DebugContext.h"

#ifdef ENABLE\_BASIC\_TELEMETRY

#include "ScriptContextTelemetry.h"

#endif

// .inl files

#include "Language\CacheOperators.inl"

#include "Language\JavascriptMathOperators.inl"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if defined(\_M\_ARM32\_OR\_ARM64)

namespace Js

{

SIMDValue SIMDFloat32x4Operation::OpFloat32x4(float x, float y, float z, float w)

{

SIMDValue result;

result.f32[SIMD\_X] = x;

result.f32[SIMD\_Y] = y;

result.f32[SIMD\_Z] = z;

result.f32[SIMD\_W] = w;

return result;

}

SIMDValue SIMDFloat32x4Operation::OpFloat32x4(const SIMDValue& v)

{

// overload function with input paramter as SIMDValue for completeness

SIMDValue result;

result = v;

return result;

}

SIMDValue SIMDFloat32x4Operation::OpZero()

{

SIMDValue result;

result.f32[SIMD\_X] = result.f32[SIMD\_Y] = result.f32[SIMD\_Z] = result.f32[SIMD\_W] = 0;

return result;

}

SIMDValue SIMDFloat32x4Operation::OpSplat(float x)

{

SIMDValue result;

result.f32[SIMD\_X] = result.f32[SIMD\_Y] = result.f32[SIMD\_Z] = result.f32[SIMD\_W] = x;

return result;

}

SIMDValue SIMDFloat32x4Operation::OpSplat(const SIMDValue& v)

{

SIMDValue result;

result.f32[SIMD\_X] = result.f32[SIMD\_Y] = result.f32[SIMD\_Z] = result.f32[SIMD\_W] = v.f32[SIMD\_X];

return result;

}

// Conversions

SIMDValue SIMDFloat32x4Operation::OpFromFloat64x2(const SIMDValue& v)

{

SIMDValue result;

result.f32[SIMD\_X] = (float)(v.f64[SIMD\_X]);

result.f32[SIMD\_Y] = (float)(v.f64[SIMD\_Y]);

result.f32[SIMD\_Z] = result.f32[SIMD\_W] = 0;

return result;

}

SIMDValue SIMDFloat32x4Operation::OpFromFloat64x2Bits(const SIMDValue& v)

{

SIMDValue result;

result.f64[SIMD\_X] = v.f64[SIMD\_X];

result.f64[SIMD\_Y] = v.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpFromInt32x4(const SIMDValue& v)

{

SIMDValue result;

result.f32[SIMD\_X] = (float)(v.i32[SIMD\_X]);

result.f32[SIMD\_Y] = (float)(v.i32[SIMD\_Y]);

result.f32[SIMD\_Z] = (float)(v.i32[SIMD\_Z]);

result.f32[SIMD\_W] = (float)(v.i32[SIMD\_W]);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpFromInt32x4Bits(const SIMDValue& v)

{

return OpFromFloat64x2Bits(v);

}

// Unary Ops

SIMDValue SIMDFloat32x4Operation::OpAbs(const SIMDValue& value)

{

SIMDValue result;

result.f32[SIMD\_X] = (value.f32[SIMD\_X] < 0) ? -1 \* value.f32[SIMD\_X] : value.f32[SIMD\_X];

result.f32[SIMD\_Y] = (value.f32[SIMD\_Y] < 0) ? -1 \* value.f32[SIMD\_Y] : value.f32[SIMD\_Y];

result.f32[SIMD\_Z] = (value.f32[SIMD\_Z] < 0) ? -1 \* value.f32[SIMD\_Z] : value.f32[SIMD\_Z];

result.f32[SIMD\_W] = (value.f32[SIMD\_W] < 0) ? -1 \* value.f32[SIMD\_W] : value.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpNeg(const SIMDValue& value)

{

SIMDValue result;

result.f32[SIMD\_X] = -1 \* value.f32[SIMD\_X];

result.f32[SIMD\_Y] = -1 \* value.f32[SIMD\_Y];

result.f32[SIMD\_Z] = -1 \* value.f32[SIMD\_Z];

result.f32[SIMD\_W] = -1 \* value.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpNot(const SIMDValue& value)

{

SIMDValue result;

result = SIMDInt32x4Operation::OpNot(value);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpReciprocal(const SIMDValue& value)

{

SIMDValue result;

result.f32[SIMD\_X] = (float)(1.0 / (value.f32[SIMD\_X]));

result.f32[SIMD\_Y] = (float)(1.0 / (value.f32[SIMD\_Y]));

result.f32[SIMD\_Z] = (float)(1.0 / (value.f32[SIMD\_Z]));

result.f32[SIMD\_W] = (float)(1.0 / (value.f32[SIMD\_W]));

return result;

}

SIMDValue SIMDFloat32x4Operation::OpReciprocalSqrt(const SIMDValue& value)

{

SIMDValue result;

result.f32[SIMD\_X] = (float)sqrt(1.0 / (value.f32[SIMD\_X]));

result.f32[SIMD\_Y] = (float)sqrt(1.0 / (value.f32[SIMD\_Y]));

result.f32[SIMD\_Z] = (float)sqrt(1.0 / (value.f32[SIMD\_Z]));

result.f32[SIMD\_W] = (float)sqrt(1.0 / (value.f32[SIMD\_W]));

return result;

}

SIMDValue SIMDFloat32x4Operation::OpSqrt(const SIMDValue& value)

{

SIMDValue result;

result.f32[SIMD\_X] = sqrtf(value.f32[SIMD\_X]);

result.f32[SIMD\_Y] = sqrtf(value.f32[SIMD\_Y]);

result.f32[SIMD\_Z] = sqrtf(value.f32[SIMD\_Z]);

result.f32[SIMD\_W] = sqrtf(value.f32[SIMD\_W]);

return result;

}

// Binary Ops

SIMDValue SIMDFloat32x4Operation::OpAdd(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f32[SIMD\_X] = aValue.f32[SIMD\_X] + bValue.f32[SIMD\_X];

result.f32[SIMD\_Y] = aValue.f32[SIMD\_Y] + bValue.f32[SIMD\_Y];

result.f32[SIMD\_Z] = aValue.f32[SIMD\_Z] + bValue.f32[SIMD\_Z];

result.f32[SIMD\_W] = aValue.f32[SIMD\_W] + bValue.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpSub(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f32[SIMD\_X] = aValue.f32[SIMD\_X] - bValue.f32[SIMD\_X];

result.f32[SIMD\_Y] = aValue.f32[SIMD\_Y] - bValue.f32[SIMD\_Y];

result.f32[SIMD\_Z] = aValue.f32[SIMD\_Z] - bValue.f32[SIMD\_Z];

result.f32[SIMD\_W] = aValue.f32[SIMD\_W] - bValue.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpMul(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f32[SIMD\_X] = aValue.f32[SIMD\_X] \* bValue.f32[SIMD\_X];

result.f32[SIMD\_Y] = aValue.f32[SIMD\_Y] \* bValue.f32[SIMD\_Y];

result.f32[SIMD\_Z] = aValue.f32[SIMD\_Z] \* bValue.f32[SIMD\_Z];

result.f32[SIMD\_W] = aValue.f32[SIMD\_W] \* bValue.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpDiv(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f32[SIMD\_X] = aValue.f32[SIMD\_X] / bValue.f32[SIMD\_X];

result.f32[SIMD\_Y] = aValue.f32[SIMD\_Y] / bValue.f32[SIMD\_Y];

result.f32[SIMD\_Z] = aValue.f32[SIMD\_Z] / bValue.f32[SIMD\_Z];

result.f32[SIMD\_W] = aValue.f32[SIMD\_W] / bValue.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpAnd(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result = SIMDInt32x4Operation::OpAnd(aValue, bValue);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpOr(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result = SIMDInt32x4Operation::OpOr(aValue, bValue);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpXor(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result = SIMDInt32x4Operation::OpXor(aValue, bValue);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpMin(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f32[SIMD\_X] = (aValue.f32[SIMD\_X] < bValue.f32[SIMD\_X]) ? aValue.f32[SIMD\_X] : bValue.f32[SIMD\_X];

result.f32[SIMD\_Y] = (aValue.f32[SIMD\_Y] < bValue.f32[SIMD\_Y]) ? aValue.f32[SIMD\_Y] : bValue.f32[SIMD\_Y];

result.f32[SIMD\_Z] = (aValue.f32[SIMD\_Z] < bValue.f32[SIMD\_Z]) ? aValue.f32[SIMD\_Z] : bValue.f32[SIMD\_Z];

result.f32[SIMD\_W] = (aValue.f32[SIMD\_W] < bValue.f32[SIMD\_W]) ? aValue.f32[SIMD\_W] : bValue.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpMax(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f32[SIMD\_X] = (aValue.f32[SIMD\_X] > bValue.f32[SIMD\_X]) ? aValue.f32[SIMD\_X] : bValue.f32[SIMD\_X];

result.f32[SIMD\_Y] = (aValue.f32[SIMD\_Y] > bValue.f32[SIMD\_Y]) ? aValue.f32[SIMD\_Y] : bValue.f32[SIMD\_Y];

result.f32[SIMD\_Z] = (aValue.f32[SIMD\_Z] > bValue.f32[SIMD\_Z]) ? aValue.f32[SIMD\_Z] : bValue.f32[SIMD\_Z];

result.f32[SIMD\_W] = (aValue.f32[SIMD\_W] > bValue.f32[SIMD\_W]) ? aValue.f32[SIMD\_W] : bValue.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpScale(const SIMDValue& Value, float scaleValue)

{

SIMDValue result;

result.f32[SIMD\_X] = Value.f32[SIMD\_X] \* scaleValue;

result.f32[SIMD\_Y] = Value.f32[SIMD\_Y] \* scaleValue;

result.f32[SIMD\_Z] = Value.f32[SIMD\_Z] \* scaleValue;

result.f32[SIMD\_W] = Value.f32[SIMD\_W] \* scaleValue;

return result;

}

SIMDValue SIMDFloat32x4Operation::OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f32[SIMD\_X] < bValue.f32[SIMD\_X];

int y = aValue.f32[SIMD\_Y] < bValue.f32[SIMD\_Y];

int z = aValue.f32[SIMD\_Z] < bValue.f32[SIMD\_Z];

int w = aValue.f32[SIMD\_W] < bValue.f32[SIMD\_W];

result = SIMDInt32x4Operation::OpBool(x, y, z, w);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpLessThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f32[SIMD\_X] <= bValue.f32[SIMD\_X];

int y = aValue.f32[SIMD\_Y] <= bValue.f32[SIMD\_Y];

int z = aValue.f32[SIMD\_Z] <= bValue.f32[SIMD\_Z];

int w = aValue.f32[SIMD\_W] <= bValue.f32[SIMD\_W];

result = SIMDInt32x4Operation::OpBool(x, y, z, w);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f32[SIMD\_X] == bValue.f32[SIMD\_X];

int y = aValue.f32[SIMD\_Y] == bValue.f32[SIMD\_Y];

int z = aValue.f32[SIMD\_Z] == bValue.f32[SIMD\_Z];

int w = aValue.f32[SIMD\_W] == bValue.f32[SIMD\_W];

result = SIMDInt32x4Operation::OpBool(x, y, z, w);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpNotEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f32[SIMD\_X] != bValue.f32[SIMD\_X];

int y = aValue.f32[SIMD\_Y] != bValue.f32[SIMD\_Y];

int z = aValue.f32[SIMD\_Z] != bValue.f32[SIMD\_Z];

int w = aValue.f32[SIMD\_W] != bValue.f32[SIMD\_W];

result = SIMDInt32x4Operation::OpBool(x, y, z, w);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f32[SIMD\_X] > bValue.f32[SIMD\_X];

int y = aValue.f32[SIMD\_Y] > bValue.f32[SIMD\_Y];

int z = aValue.f32[SIMD\_Z] > bValue.f32[SIMD\_Z];

int w = aValue.f32[SIMD\_W] > bValue.f32[SIMD\_W];

result = SIMDInt32x4Operation::OpBool(x, y, z, w);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpGreaterThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f32[SIMD\_X] >= bValue.f32[SIMD\_X];

int y = aValue.f32[SIMD\_Y] >= bValue.f32[SIMD\_Y];

int z = aValue.f32[SIMD\_Z] >= bValue.f32[SIMD\_Z];

int w = aValue.f32[SIMD\_W] >= bValue.f32[SIMD\_W];

result = SIMDInt32x4Operation::OpBool(x, y, z, w);

return result;

}

SIMDValue SIMDFloat32x4Operation::OpClamp(const SIMDValue& value, const SIMDValue& lower, const SIMDValue& upper)

{

SIMDValue result;

// lower clamp

result.f32[SIMD\_X] = value.f32[SIMD\_X] < lower.f32[SIMD\_X] ? lower.f32[SIMD\_X] : value.f32[SIMD\_X];

result.f32[SIMD\_Y] = value.f32[SIMD\_Y] < lower.f32[SIMD\_Y] ? lower.f32[SIMD\_Y] : value.f32[SIMD\_Y];

result.f32[SIMD\_Z] = value.f32[SIMD\_Z] < lower.f32[SIMD\_Z] ? lower.f32[SIMD\_Z] : value.f32[SIMD\_Z];

result.f32[SIMD\_W] = value.f32[SIMD\_W] < lower.f32[SIMD\_W] ? lower.f32[SIMD\_W] : value.f32[SIMD\_W];

// upper clamp

result.f32[SIMD\_X] = result.f32[SIMD\_X] > upper.f32[SIMD\_X] ? upper.f32[SIMD\_X] : result.f32[SIMD\_X];

result.f32[SIMD\_Y] = result.f32[SIMD\_Y] > upper.f32[SIMD\_Y] ? upper.f32[SIMD\_Y] : result.f32[SIMD\_Y];

result.f32[SIMD\_Z] = result.f32[SIMD\_Z] > upper.f32[SIMD\_Z] ? upper.f32[SIMD\_Z] : result.f32[SIMD\_Z];

result.f32[SIMD\_W] = result.f32[SIMD\_W] > upper.f32[SIMD\_W] ? upper.f32[SIMD\_W] : result.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV)

{

SIMDValue result;

SIMDValue trueResult = SIMDInt32x4Operation::OpAnd(mV, tV);

SIMDValue notValue = SIMDInt32x4Operation::OpNot(mV);

SIMDValue falseResult = SIMDInt32x4Operation::OpAnd(notValue, fV);

result = SIMDInt32x4Operation::OpOr(trueResult, falseResult);

return result;

}

// Get SignMask

int SIMDFloat32x4Operation::OpGetSignMask(const SIMDValue& v)

{

int result;

int mx = (v.f32[SIMD\_X] < 0.0 || 1 / v.f32[SIMD\_X] == JavascriptNumber::NEGATIVE\_INFINITY) ? 1 : 0;

int my = (v.f32[SIMD\_Y] < 0.0 || 1 / v.f32[SIMD\_Y] == JavascriptNumber::NEGATIVE\_INFINITY) ? 1 : 0;

int mz = (v.f32[SIMD\_Z] < 0.0 || 1 / v.f32[SIMD\_Z] == JavascriptNumber::NEGATIVE\_INFINITY) ? 1 : 0;

int mw = (v.f32[SIMD\_W] < 0.0 || 1 / v.f32[SIMD\_W] == JavascriptNumber::NEGATIVE\_INFINITY) ? 1 : 0;

result = mx | my << 1 | mz << 2 | mw << 3;

return result;

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

struct SIMDFloat32x4Operation

{

// following are operation wrappers of SIMD.Float32x4 general implementation

static SIMDValue OpFloat32x4(float x, float y, float z, float w);

static SIMDValue OpFloat32x4(const SIMDValue& v);

static SIMDValue OpZero();

static SIMDValue OpSplat(float x);

static SIMDValue OpSplat(const SIMDValue& v);

// conversion

static SIMDValue OpFromFloat64x2(const SIMDValue& value);

static SIMDValue OpFromFloat64x2Bits(const SIMDValue& value);

static SIMDValue OpFromInt32x4(const SIMDValue& value);

static SIMDValue OpFromInt32x4Bits(const SIMDValue& value);

// Unary Ops

static SIMDValue OpAbs(const SIMDValue& v);

static SIMDValue OpNeg(const SIMDValue& v);

static SIMDValue OpNot(const SIMDValue& v);

static SIMDValue OpReciprocal(const SIMDValue& v);

static SIMDValue OpReciprocalSqrt(const SIMDValue& v);

static SIMDValue OpSqrt(const SIMDValue& v);

// Binary Ops

static SIMDValue OpAdd(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpSub(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMul(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpDiv(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpAnd(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpOr (const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpXor(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMin(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMax(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpScale(const SIMDValue& Value, float scaleValue);

static SIMDValue OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpLessThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpNotEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpGreaterThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpClamp(const SIMDValue& value, const SIMDValue& upper, const SIMDValue& lower);

static SIMDValue OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV);

// Get SignMask

static int OpGetSignMask(const SIMDValue& mV);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if \_M\_IX86 || \_M\_AMD64

namespace Js

{

SIMDValue SIMDFloat32x4Operation::OpFloat32x4(float x, float y, float z, float w)

{

X86SIMDValue x86Result;

// Sets the 4 single-precision, floating-point values, note order starts with W below

x86Result.m128\_value = \_mm\_set\_ps(w, z, y, x);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpFloat32x4(const SIMDValue& v)

{

X86SIMDValue x86Result;

// Sets the 4 single-precision, floating-point values, note in revised order: W, Z, Y, X

x86Result.m128\_value = \_mm\_set\_ps(v.f32[SIMD\_W], v.f32[SIMD\_Z], v.f32[SIMD\_Y], v.f32[SIMD\_X]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpZero()

{

X86SIMDValue x86Result;

// Sets the 128-bit value to zero

x86Result.m128\_value = \_mm\_setzero\_ps();

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpSplat(float x)

{

X86SIMDValue x86Result;

// Sets the four single-precision, floating-point values to x

x86Result.m128\_value = \_mm\_set1\_ps(x);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpSplat(const SIMDValue& v)

{

X86SIMDValue x86Result;

// Sets the four single-precision, floating-point values to v.f32[SIMD\_X]

x86Result.m128\_value = \_mm\_set1\_ps(v.f32[SIMD\_X]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

// Conversions

SIMDValue SIMDFloat32x4Operation::OpFromFloat64x2(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Converts the two double-precision, floating-point values of v.m128d\_value

// to single-precision, floating-point values.

x86Result.m128\_value = \_mm\_cvtpd\_ps(v.m128d\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpFromFloat64x2Bits(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

\_mm\_store\_ps(x86Result.simdValue.f32, v.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpFromInt32x4(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Converts the 4 signed 32-bit integer values of v.m128i\_value

// to single-precision, floating-point values.

x86Result.m128\_value = \_mm\_cvtepi32\_ps(v.m128i\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpFromInt32x4Bits(const SIMDValue& value)

{

return OpFromFloat64x2Bits(value);

}

// Unary Ops

SIMDValue SIMDFloat32x4Operation::OpAbs(const SIMDValue& value)

{

X86SIMDValue x86Result, SIGNMASK;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

SIGNMASK.m128\_value = \_mm\_castsi128\_ps(\_mm\_set1\_epi32(0x7FFFFFFF));

// bitwise AND of the 4 single - precision, floats of SIGNMASK and v

x86Result.m128\_value = \_mm\_and\_ps(SIGNMASK.m128\_value, v.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpNeg(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue SIGNMASK;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

SIGNMASK.m128\_value = \_mm\_castsi128\_ps(\_mm\_set1\_epi32(0x80000000));

// bitwise EXOR (exclusive-or) of the 4 single-precision, floats of value and signmask

x86Result.m128\_value = \_mm\_xor\_ps(v.m128\_value, SIGNMASK.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpNot(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue negativeOnes = { { -1, -1, -1, -1 } };

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// bitwise EXOR (exclusive-or) of the 4 single-precision, floats of value and -1

x86Result.m128\_value = \_mm\_xor\_ps(v.m128\_value, negativeOnes.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpReciprocal(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue floatOnes;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// RCPPS is not precise. Using DIVPS

floatOnes.m128\_value = \_mm\_set\_ps(1.0, 1.0, 1.0, 1.0);

// Divides the four single-precision, floating-point values of 1.0 and value

x86Result.m128\_value = \_mm\_div\_ps(floatOnes.m128\_value, v.m128\_value); // result = 1.0/value

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpReciprocalSqrt(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue floatOnes, temp;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

floatOnes.m128\_value = \_mm\_set\_ps(1.0, 1.0, 1.0, 1.0);

temp.m128\_value = \_mm\_div\_ps(floatOnes.m128\_value, v.m128\_value); // temp = 1.0/value

x86Result.m128\_value = \_mm\_sqrt\_ps(temp.m128\_value); // result = sqrt(1.0/value)

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpSqrt(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

x86Result.m128\_value = \_mm\_sqrt\_ps(v.m128\_value); // result = sqrt(value)

return X86SIMDValue::ToSIMDValue(x86Result);

}

// Binary Ops

SIMDValue SIMDFloat32x4Operation::OpAdd(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_add\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a + b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpSub(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_sub\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a - b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpMul(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_mul\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a \* b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpDiv(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_div\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a / b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpAnd(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_and\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a & b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpOr(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_or\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a | b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpXor(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_xor\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a ^ b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpMin(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

// choose the smaller value of the two parameters a and b

x86Result.m128\_value = \_mm\_min\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpMax(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

// choose the larger value of the two parameters a and b

x86Result.m128\_value = \_mm\_max\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpScale(const SIMDValue& Value, float scaleValue)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(Value);

X86SIMDValue scaleVector;

scaleVector.m128\_value = \_mm\_set1\_ps(scaleValue);

x86Result.m128\_value = \_mm\_mul\_ps(v.m128\_value, scaleVector.m128\_value); // v \* scale

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_cmplt\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a < b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpLessThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_cmple\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a <= b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_cmpeq\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a == b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpNotEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_cmpneq\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a != b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_cmpgt\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a > b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpGreaterThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128\_value = \_mm\_cmpge\_ps(tmpaValue.m128\_value, tmpbValue.m128\_value); // a >= b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat32x4Operation::OpClamp(const SIMDValue& value, const SIMDValue& lower, const SIMDValue& upper)

{ // SIMD review: do we have intrinsic for the implementation?

SIMDValue result;

// lower clamp

result.f32[SIMD\_X] = value.f32[SIMD\_X] < lower.f32[SIMD\_X] ? lower.f32[SIMD\_X] : value.f32[SIMD\_X];

result.f32[SIMD\_Y] = value.f32[SIMD\_Y] < lower.f32[SIMD\_Y] ? lower.f32[SIMD\_Y] : value.f32[SIMD\_Y];

result.f32[SIMD\_Z] = value.f32[SIMD\_Z] < lower.f32[SIMD\_Z] ? lower.f32[SIMD\_Z] : value.f32[SIMD\_Z];

result.f32[SIMD\_W] = value.f32[SIMD\_W] < lower.f32[SIMD\_W] ? lower.f32[SIMD\_W] : value.f32[SIMD\_W];

// upper clamp

result.f32[SIMD\_X] = result.f32[SIMD\_X] > upper.f32[SIMD\_X] ? upper.f32[SIMD\_X] : result.f32[SIMD\_X];

result.f32[SIMD\_Y] = result.f32[SIMD\_Y] > upper.f32[SIMD\_Y] ? upper.f32[SIMD\_Y] : result.f32[SIMD\_Y];

result.f32[SIMD\_Z] = result.f32[SIMD\_Z] > upper.f32[SIMD\_Z] ? upper.f32[SIMD\_Z] : result.f32[SIMD\_Z];

result.f32[SIMD\_W] = result.f32[SIMD\_W] > upper.f32[SIMD\_W] ? upper.f32[SIMD\_W] : result.f32[SIMD\_W];

return result;

}

SIMDValue SIMDFloat32x4Operation::OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV)

{

X86SIMDValue x86Result;

X86SIMDValue maskValue = X86SIMDValue::ToX86SIMDValue(mV);

X86SIMDValue trueValue = X86SIMDValue::ToX86SIMDValue(tV);

X86SIMDValue falseValue = X86SIMDValue::ToX86SIMDValue(fV);

X86SIMDValue tempTrue, tempFalse;

tempTrue.m128\_value = \_mm\_and\_ps(maskValue.m128\_value, trueValue.m128\_value); // mask & True

tempFalse.m128\_value = \_mm\_andnot\_ps(maskValue.m128\_value, falseValue.m128\_value); // !mask & False

x86Result.m128\_value = \_mm\_or\_ps(tempTrue.m128\_value, tempFalse.m128\_value); // tempTrue | tempFalse

return X86SIMDValue::ToSIMDValue(x86Result);

}

// Get SignMask

int SIMDFloat32x4Operation::OpGetSignMask(const SIMDValue& value)

{

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Creates a 4-bit mask from the most significant bits of

// the 4 single-precision, floating-point values

return \_mm\_movemask\_ps(v.m128\_value);

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if defined(\_M\_ARM32\_OR\_ARM64)

namespace Js

{

SIMDValue SIMDFloat64x2Operation::OpFloat64x2(double x, double y)

{

SIMDValue result;

result.f64[SIMD\_X] = x;

result.f64[SIMD\_Y] = y;

return result;

}

SIMDValue SIMDFloat64x2Operation::OpFloat64x2(const SIMDValue& v)

{// overload function with input parameter as SIMDValue for completeness

SIMDValue result;

result = v;

return result;

}

SIMDValue SIMDFloat64x2Operation::OpZero()

{

SIMDValue result;

result.f64[SIMD\_X] = result.f64[SIMD\_Y] = 0;

return result;

}

SIMDValue SIMDFloat64x2Operation::OpSplat(double x)

{

SIMDValue result;

result.f64[SIMD\_X] = result.f64[SIMD\_Y] = x;

return result;

}

SIMDValue SIMDFloat64x2Operation::OpSplat(const SIMDValue& v)

{

SIMDValue result;

result.f64[SIMD\_X] = result.f64[SIMD\_Y] = v.f64[SIMD\_X];

return result;

}

// Conversions

SIMDValue SIMDFloat64x2Operation::OpFromFloat32x4(const SIMDValue& v)

{

SIMDValue result;

result.f64[SIMD\_X] = (double)(v.f32[SIMD\_X]);

result.f64[SIMD\_Y] = (double)(v.f32[SIMD\_Y]);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpFromInt32x4(const SIMDValue& v)

{

SIMDValue result;

result.f64[SIMD\_X] = (double)(v.i32[SIMD\_X]);

result.f64[SIMD\_Y] = (double)(v.i32[SIMD\_Y]);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpFromFloat32x4Bits(const SIMDValue& v)

{

SIMDValue result;

result.f64[SIMD\_X] = v.f64[SIMD\_X];

result.f64[SIMD\_Y] = v.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpFromInt32x4Bits(const SIMDValue& v)

{

return OpFromFloat32x4Bits(v);

}

// Unary Ops

SIMDValue SIMDFloat64x2Operation::OpAbs(const SIMDValue& value)

{

SIMDValue result;

result.f64[SIMD\_X] = (value.f64[SIMD\_X] < 0) ? -1 \* value.f64[SIMD\_X] : value.f64[SIMD\_X];

result.f64[SIMD\_Y] = (value.f64[SIMD\_Y] < 0) ? -1 \* value.f64[SIMD\_Y] : value.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpNeg(const SIMDValue& value)

{

SIMDValue result;

result.f64[SIMD\_X] = -1 \* value.f64[SIMD\_X];

result.f64[SIMD\_Y] = -1 \* value.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpNot(const SIMDValue& value)

{

SIMDValue result;

result = SIMDInt32x4Operation::OpNot(value);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpReciprocal(const SIMDValue& value)

{

SIMDValue result;

result.f64[SIMD\_X] = 1.0/(value.f64[SIMD\_X]);

result.f64[SIMD\_Y] = 1.0/(value.f64[SIMD\_Y]);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpReciprocalSqrt(const SIMDValue& value)

{

SIMDValue result;

result.f64[SIMD\_X] = sqrt(1.0 / (value.f64[SIMD\_X]));

result.f64[SIMD\_Y] = sqrt(1.0 / (value.f64[SIMD\_Y]));

return result;

}

SIMDValue SIMDFloat64x2Operation::OpSqrt(const SIMDValue& value)

{

SIMDValue result;

result.f64[SIMD\_X] = sqrt(value.f64[SIMD\_X]);

result.f64[SIMD\_Y] = sqrt(value.f64[SIMD\_Y]);

return result;

}

// Binary Ops

SIMDValue SIMDFloat64x2Operation::OpAdd(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f64[SIMD\_X] = aValue.f64[SIMD\_X] + bValue.f64[SIMD\_X];

result.f64[SIMD\_Y] = aValue.f64[SIMD\_Y] + bValue.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpSub(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f64[SIMD\_X] = aValue.f64[SIMD\_X] - bValue.f64[SIMD\_X];

result.f64[SIMD\_Y] = aValue.f64[SIMD\_Y] - bValue.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpMul(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f64[SIMD\_X] = aValue.f64[SIMD\_X] \* bValue.f64[SIMD\_X];

result.f64[SIMD\_Y] = aValue.f64[SIMD\_Y] \* bValue.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpDiv(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f64[SIMD\_X] = aValue.f64[SIMD\_X] / bValue.f64[SIMD\_X];

result.f64[SIMD\_Y] = aValue.f64[SIMD\_Y] / bValue.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpAnd(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result = SIMDInt32x4Operation::OpAnd(aValue, bValue);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpOr(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result = SIMDInt32x4Operation::OpOr(aValue, bValue);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpXor(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result = SIMDInt32x4Operation::OpXor(aValue, bValue);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpMin(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f64[SIMD\_X] = (aValue.f64[SIMD\_X] < bValue.f64[SIMD\_X]) ? aValue.f64[SIMD\_X] : bValue.f64[SIMD\_X];

result.f64[SIMD\_Y] = (aValue.f64[SIMD\_Y] < bValue.f64[SIMD\_Y]) ? aValue.f64[SIMD\_Y] : bValue.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpMax(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.f64[SIMD\_X] = (aValue.f64[SIMD\_X] > bValue.f64[SIMD\_X]) ? aValue.f64[SIMD\_X] : bValue.f64[SIMD\_X];

result.f64[SIMD\_Y] = (aValue.f64[SIMD\_Y] > bValue.f64[SIMD\_Y]) ? aValue.f64[SIMD\_Y] : bValue.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpScale(const SIMDValue& Value, double scaleValue)

{

SIMDValue result;

result.f64[SIMD\_X] = Value.f64[SIMD\_X] \* scaleValue;

result.f64[SIMD\_Y] = Value.f64[SIMD\_Y] \* scaleValue;

return result;

}

SIMDValue SIMDFloat64x2Operation::OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f64[SIMD\_X] < bValue.f64[SIMD\_X];

int y = aValue.f64[SIMD\_Y] < bValue.f64[SIMD\_Y];

result = SIMDInt32x4Operation::OpBool(x, x, y, y);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpLessThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f64[SIMD\_X] <= bValue.f64[SIMD\_X];

int y = aValue.f64[SIMD\_Y] <= bValue.f64[SIMD\_Y];

result = SIMDInt32x4Operation::OpBool(x, x, y, y);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f64[SIMD\_X] == bValue.f64[SIMD\_X];

int y = aValue.f64[SIMD\_Y] == bValue.f64[SIMD\_Y];

result = SIMDInt32x4Operation::OpBool(x, x, y, y);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpNotEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f64[SIMD\_X] != bValue.f64[SIMD\_X];

int y = aValue.f64[SIMD\_Y] != bValue.f64[SIMD\_Y];

result = SIMDInt32x4Operation::OpBool(x, x, y, y);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f64[SIMD\_X] > bValue.f64[SIMD\_X];

int y = aValue.f64[SIMD\_Y] > bValue.f64[SIMD\_Y];

result = SIMDInt32x4Operation::OpBool(x, x, y, y);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpGreaterThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

int x = aValue.f64[SIMD\_X] >= bValue.f64[SIMD\_X];

int y = aValue.f64[SIMD\_Y] >= bValue.f64[SIMD\_Y];

result = SIMDInt32x4Operation::OpBool(x, x, y, y);

return result;

}

SIMDValue SIMDFloat64x2Operation::OpClamp(const SIMDValue& value, const SIMDValue& lower, const SIMDValue& upper)

{

SIMDValue result;

// lower clamp

result.f64[SIMD\_X] = value.f64[SIMD\_X] < lower.f64[SIMD\_X] ? lower.f64[SIMD\_X] : value.f64[SIMD\_X];

result.f64[SIMD\_Y] = value.f64[SIMD\_Y] < lower.f64[SIMD\_Y] ? lower.f64[SIMD\_Y] : value.f64[SIMD\_Y];

// upper clamp

result.f64[SIMD\_X] = result.f64[SIMD\_X] > upper.f64[SIMD\_X] ? upper.f64[SIMD\_X] : result.f64[SIMD\_X];

result.f64[SIMD\_Y] = result.f64[SIMD\_Y] > upper.f64[SIMD\_Y] ? upper.f64[SIMD\_Y] : result.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV)

{

SIMDValue result;

SIMDValue trueResult = SIMDInt32x4Operation::OpAnd(mV, tV);

SIMDValue notValue = SIMDInt32x4Operation::OpNot(mV);

SIMDValue falseResult = SIMDInt32x4Operation::OpAnd(notValue, fV);

result = SIMDInt32x4Operation::OpOr(trueResult, falseResult);

return result;

}

// Get SignMask

int SIMDFloat64x2Operation::OpGetSignMask(const SIMDValue& v)

{

int result;

int mx = (v.f64[SIMD\_X] < 0.0 || 1 / v.f64[SIMD\_X] == JavascriptNumber::NEGATIVE\_INFINITY) ? 1 : 0;

int my = (v.f64[SIMD\_Y] < 0.0 || 1 / v.f64[SIMD\_Y] == JavascriptNumber::NEGATIVE\_INFINITY) ? 1 : 0;

result = mx | my << 1;

return result;

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

struct SIMDFloat64x2Operation

{

// following are operation wrappers for SIMD.Float64x2 general implementation

static SIMDValue OpFloat64x2(double x, double y);

static SIMDValue OpFloat64x2(const SIMDValue& v);

static SIMDValue OpZero();

static SIMDValue OpSplat(double x);

static SIMDValue OpSplat(const SIMDValue& v);

// conversion

static SIMDValue OpFromFloat32x4(const SIMDValue& value);

static SIMDValue OpFromFloat32x4Bits(const SIMDValue& value);

static SIMDValue OpFromInt32x4(const SIMDValue& value);

static SIMDValue OpFromInt32x4Bits(const SIMDValue& value);

// Unary Ops

static SIMDValue OpAbs(const SIMDValue& v);

static SIMDValue OpNeg(const SIMDValue& v);

static SIMDValue OpNot(const SIMDValue& v);

static SIMDValue OpReciprocal(const SIMDValue& v);

static SIMDValue OpReciprocalSqrt(const SIMDValue& v);

static SIMDValue OpSqrt(const SIMDValue& v);

// Binary Ops

static SIMDValue OpAdd(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpSub(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMul(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpDiv(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpAnd(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpOr (const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpXor(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMin(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMax(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpScale(const SIMDValue& Value, double scaleValue);

static SIMDValue OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpLessThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpNotEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpGreaterThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpClamp(const SIMDValue& value, const SIMDValue& upper, const SIMDValue& lower);

static SIMDValue OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV);

// Get SignMask

static int OpGetSignMask(const SIMDValue& mV);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if \_M\_IX86 || \_M\_AMD64

namespace Js

{

SIMDValue SIMDFloat64x2Operation::OpFloat64x2(double x, double y)

{

X86SIMDValue x86Result;

// Sets the lower double-precision, floating-point value to x

// and sets the upper double-precision, floating-point value to y.

x86Result.m128d\_value = \_mm\_set\_pd(y, x);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpFloat64x2(const SIMDValue& v)

{

X86SIMDValue x86Result;

// Sets the lower double-precision, floating-point value to x

// and sets the upper double-precision, floating-point value to y.

x86Result.m128d\_value = \_mm\_set\_pd(v.f64[SIMD\_Y], v.f64[SIMD\_X]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpZero()

{

X86SIMDValue x86Result;

// Sets the 2 double-precision, floating-point values to zero

x86Result.m128d\_value = \_mm\_setzero\_pd();

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpSplat(double x)

{

X86SIMDValue x86Result;

// Sets the 2 double-precision, floating-point values to x

x86Result.m128d\_value = \_mm\_set1\_pd(x);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpSplat(const SIMDValue& v)

{

X86SIMDValue x86Result;

// Sets the 2 double-precision, floating-point values to v.f64[SIMD\_X]

x86Result.m128d\_value = \_mm\_set1\_pd(v.f64[SIMD\_X]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

// Conversions

SIMDValue SIMDFloat64x2Operation::OpFromFloat32x4(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Converts the lower 2 single-precision, floating-point values

// to two double-precision, floating-point values

x86Result.m128d\_value = \_mm\_cvtps\_pd(v.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpFromInt32x4(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Converts the lower 2 signed 32-bit integer values of

// to double-precision, floating-point values

x86Result.m128d\_value = \_mm\_cvtepi32\_pd(v.m128i\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpFromFloat32x4Bits(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

\_mm\_store\_ps(x86Result.simdValue.f32, v.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpFromInt32x4Bits(const SIMDValue& value)

{

return OpFromFloat32x4Bits(value);

}

// Unary Ops

SIMDValue SIMDFloat64x2Operation::OpAbs(const SIMDValue& value)

{

X86SIMDValue x86Result, SIGNMASK;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

SIGNMASK.m128d\_value = \_mm\_castsi128\_pd(\_mm\_set\_epi32(0x7fffffff, 0xffffffff, 0x7fffffff, 0xffffffff));

x86Result.m128d\_value = \_mm\_and\_pd(v.m128d\_value, SIGNMASK.m128d\_value); // v & SIGNMASK

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpNeg(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue SIGNMASK;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

SIGNMASK.m128d\_value = \_mm\_castsi128\_pd(\_mm\_set\_epi32(0x80000000, 0x0, 0x80000000, 0x0));

x86Result.m128d\_value = \_mm\_xor\_pd(v.m128d\_value, SIGNMASK.m128d\_value); // v ^ mask

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpNot(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue negativeOnes = { { -1, -1, -1, -1 } };

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

x86Result.m128d\_value = \_mm\_xor\_pd(v.m128d\_value, negativeOnes.m128d\_value); // v ^ -1

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpReciprocal(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue doubleOnes;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

doubleOnes.m128d\_value = \_mm\_set\_pd(1.0, 1.0);

x86Result.m128d\_value = \_mm\_div\_pd(doubleOnes.m128d\_value, v.m128d\_value); // result = 1.0/value

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpReciprocalSqrt(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue doubleOnes, temp;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

doubleOnes.m128d\_value = \_mm\_set\_pd(1.0, 1.0);

temp.m128d\_value = \_mm\_div\_pd(doubleOnes.m128d\_value, v.m128d\_value); // temp = 1.0/value

x86Result.m128d\_value = \_mm\_sqrt\_pd(temp.m128d\_value); // result = sqrt(1.0/value)

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpSqrt(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

x86Result.m128d\_value = \_mm\_sqrt\_pd(v.m128d\_value); // result = sqrt(value)

return X86SIMDValue::ToSIMDValue(x86Result);

}

// Binary Ops

SIMDValue SIMDFloat64x2Operation::OpAdd(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_add\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a + b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpSub(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_sub\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a - b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpMul(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_mul\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a \* b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpDiv(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_div\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a / b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpAnd(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_and\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a & b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpOr(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_or\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a | b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpXor(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_xor\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a ^ b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpMin(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

// choose the smaller value of the two parameters a and b

x86Result.m128d\_value = \_mm\_min\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpMax(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

// choose the larger value of the two parameters a and b

x86Result.m128d\_value = \_mm\_max\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpScale(const SIMDValue& Value, double scaleValue)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(Value);

X86SIMDValue scaleVector;

scaleVector.m128d\_value = \_mm\_set1\_pd(scaleValue);

x86Result.m128d\_value = \_mm\_mul\_pd(v.m128d\_value, scaleVector.m128d\_value); // v \* scale

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_cmplt\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a < b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpLessThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_cmple\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a <= b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_cmpeq\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a == b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpNotEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_cmpneq\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a != b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_cmpgt\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a > b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpGreaterThanOrEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128d\_value = \_mm\_cmpge\_pd(tmpaValue.m128d\_value, tmpbValue.m128d\_value); // a >= b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDFloat64x2Operation::OpClamp(const SIMDValue& value, const SIMDValue& lower, const SIMDValue& upper)

{ // SIMD review: do we have intrinsic for the implementation?

SIMDValue result;

// lower clamp

result.f64[SIMD\_X] = value.f64[SIMD\_X] < lower.f64[SIMD\_X] ? lower.f64[SIMD\_X] : value.f64[SIMD\_X];

result.f64[SIMD\_Y] = value.f64[SIMD\_Y] < lower.f64[SIMD\_Y] ? lower.f64[SIMD\_Y] : value.f64[SIMD\_Y];

// upper clamp

result.f64[SIMD\_X] = result.f64[SIMD\_X] > upper.f64[SIMD\_X] ? upper.f64[SIMD\_X] : result.f64[SIMD\_X];

result.f64[SIMD\_Y] = result.f64[SIMD\_Y] > upper.f64[SIMD\_Y] ? upper.f64[SIMD\_Y] : result.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDFloat64x2Operation::OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV)

{

X86SIMDValue x86Result;

X86SIMDValue maskValue = X86SIMDValue::ToX86SIMDValue(mV);

X86SIMDValue trueValue = X86SIMDValue::ToX86SIMDValue(tV);

X86SIMDValue falseValue = X86SIMDValue::ToX86SIMDValue(fV);

X86SIMDValue tempTrue, tempFalse;

tempTrue.m128d\_value = \_mm\_and\_pd(maskValue.m128d\_value, trueValue.m128d\_value); // mask & True

tempFalse.m128d\_value = \_mm\_andnot\_pd(maskValue.m128d\_value, falseValue.m128d\_value); // !mask & False

x86Result.m128d\_value = \_mm\_or\_pd(tempTrue.m128d\_value, tempFalse.m128d\_value); // tempTrue | tempFalse

return X86SIMDValue::ToSIMDValue(x86Result);

}

// Get SignMask

int SIMDFloat64x2Operation::OpGetSignMask(const SIMDValue& value)

{

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Creates a two-bit mask from the sign bits of the two double-precision, floating-point

// values of v.m128d\_value

return \_mm\_movemask\_pd(v.m128d\_value);

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if defined(\_M\_ARM32\_OR\_ARM64)

namespace Js

{

SIMDValue SIMDInt32x4Operation::OpInt32x4(int x, int y, int z, int w)

{

SIMDValue result;

result.i32[SIMD\_X] = x;

result.i32[SIMD\_Y] = y;

result.i32[SIMD\_Z] = z;

result.i32[SIMD\_W] = w;

return result;

}

SIMDValue SIMDInt32x4Operation::OpInt32x4(const SIMDValue& v)

{

// overload function with input parameter as SIMDValue for completeness, may not need

SIMDValue result;

result = v;

return result;

}

SIMDValue SIMDInt32x4Operation::OpZero()

{

SIMDValue result;

result.i32[SIMD\_X] = result.i32[SIMD\_Y] = result.i32[SIMD\_Z] = result.i32[SIMD\_W] = 0;

return result;

}

SIMDValue SIMDInt32x4Operation::OpSplat(int x)

{

SIMDValue result;

result.i32[SIMD\_X] = result.i32[SIMD\_Y] = result.i32[SIMD\_Z] = result.i32[SIMD\_W] = x;

return result;

}

SIMDValue SIMDInt32x4Operation::OpSplat(const SIMDValue& v)

{

SIMDValue result;

result.i32[SIMD\_X] = result.i32[SIMD\_Y] = result.i32[SIMD\_Z] = result.i32[SIMD\_W] = v.i32[SIMD\_X];

return result;

}

SIMDValue SIMDInt32x4Operation::OpBool(int x, int y, int z, int w)

{

SIMDValue result;

int nX = x ? -1 : 0x0;

int nY = y ? -1 : 0x0;

int nZ = z ? -1 : 0x0;

int nW = w ? -1 : 0x0;

result.i32[SIMD\_X] = nX;

result.i32[SIMD\_Y] = nY;

result.i32[SIMD\_Z] = nZ;

result.i32[SIMD\_W] = nW;

return result;

}

SIMDValue SIMDInt32x4Operation::OpBool(const SIMDValue& v)

{

SIMDValue result;

// incoming 4 signed integers has to be 0 or -1

Assert(v.i32[SIMD\_X] == 0 || v.i32[SIMD\_X] == -1);

Assert(v.i32[SIMD\_Y] == 0 || v.i32[SIMD\_Y] == -1);

Assert(v.i32[SIMD\_Z] == 0 || v.i32[SIMD\_Z] == -1);

Assert(v.i32[SIMD\_W] == 0 || v.i32[SIMD\_W] == -1);

result = v;

return result;

}

// Conversions

SIMDValue SIMDInt32x4Operation::OpFromBool(const SIMDValue& v)

{

SIMDValue result;

result.i32[SIMD\_X] = (v.i32[SIMD\_X]) ? 0xFFFFFFFF : 0x0;

result.i32[SIMD\_Y] = (v.i32[SIMD\_Y]) ? 0xFFFFFFFF : 0x0;

result.i32[SIMD\_Z] = (v.i32[SIMD\_Z]) ? 0xFFFFFFFF : 0x0;

result.i32[SIMD\_W] = (v.i32[SIMD\_W]) ? 0xFFFFFFFF : 0x0;

return result;

}

SIMDValue SIMDInt32x4Operation::OpFromFloat32x4(const SIMDValue& v)

{

SIMDValue result;

result.i32[SIMD\_X] = (int)(v.f32[SIMD\_X]);

result.i32[SIMD\_Y] = (int)(v.f32[SIMD\_Y]);

result.i32[SIMD\_Z] = (int)(v.f32[SIMD\_Z]);

result.i32[SIMD\_W] = (int)(v.f32[SIMD\_W]);

return result;

}

SIMDValue SIMDInt32x4Operation::OpFromFloat64x2(const SIMDValue& v)

{

SIMDValue result;

result.i32[SIMD\_X] = (int)(v.f64[SIMD\_X]);

result.i32[SIMD\_Y] = (int)(v.f64[SIMD\_Y]);

result.i32[SIMD\_Z] = result.i32[SIMD\_W] = 0;

return result;

}

SIMDValue SIMDInt32x4Operation::OpFromFloat32x4Bits(const SIMDValue& v)

{

SIMDValue result;

result.f64[SIMD\_X] = v.f64[SIMD\_X];

result.f64[SIMD\_Y] = v.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDInt32x4Operation::OpFromFloat64x2Bits(const SIMDValue& v)

{

return OpFromFloat32x4Bits(v);

}

// Unary Ops

SIMDValue SIMDInt32x4Operation::OpAbs(const SIMDValue& value)

{

SIMDValue result;

result.i32[SIMD\_X] = (value.i32[SIMD\_X] < 0) ? -1 \* value.i32[SIMD\_X] : value.i32[SIMD\_X];

result.i32[SIMD\_Y] = (value.i32[SIMD\_Y] < 0) ? -1 \* value.i32[SIMD\_Y] : value.i32[SIMD\_Y];

result.i32[SIMD\_Z] = (value.i32[SIMD\_Z] < 0) ? -1 \* value.i32[SIMD\_Z] : value.i32[SIMD\_Z];

result.i32[SIMD\_W] = (value.i32[SIMD\_W] < 0) ? -1 \* value.i32[SIMD\_W] : value.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpNeg(const SIMDValue& value)

{

SIMDValue result;

result.i32[SIMD\_X] = -1 \* value.i32[SIMD\_X];

result.i32[SIMD\_Y] = -1 \* value.i32[SIMD\_Y];

result.i32[SIMD\_Z] = -1 \* value.i32[SIMD\_Z];

result.i32[SIMD\_W] = -1 \* value.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpNot(const SIMDValue& value)

{

SIMDValue result;

result.i32[SIMD\_X] = ~(value.i32[SIMD\_X]);

result.i32[SIMD\_Y] = ~(value.i32[SIMD\_Y]);

result.i32[SIMD\_Z] = ~(value.i32[SIMD\_Z]);

result.i32[SIMD\_W] = ~(value.i32[SIMD\_W]);

return result;

}

SIMDValue SIMDInt32x4Operation::OpAdd(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = aValue.i32[SIMD\_X] + bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = aValue.i32[SIMD\_Y] + bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = aValue.i32[SIMD\_Z] + bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = aValue.i32[SIMD\_W] + bValue.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpSub(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = aValue.i32[SIMD\_X] - bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = aValue.i32[SIMD\_Y] - bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = aValue.i32[SIMD\_Z] - bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = aValue.i32[SIMD\_W] - bValue.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpMul(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = aValue.i32[SIMD\_X] \* bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = aValue.i32[SIMD\_Y] \* bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = aValue.i32[SIMD\_Z] \* bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = aValue.i32[SIMD\_W] \* bValue.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpAnd(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = aValue.i32[SIMD\_X] & bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = aValue.i32[SIMD\_Y] & bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = aValue.i32[SIMD\_Z] & bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = aValue.i32[SIMD\_W] & bValue.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpOr(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = aValue.i32[SIMD\_X] | bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = aValue.i32[SIMD\_Y] | bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = aValue.i32[SIMD\_Z] | bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = aValue.i32[SIMD\_W] | bValue.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpXor(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = aValue.i32[SIMD\_X] ^ bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = aValue.i32[SIMD\_Y] ^ bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = aValue.i32[SIMD\_Z] ^ bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = aValue.i32[SIMD\_W] ^ bValue.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpMin(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = (aValue.i32[SIMD\_X] < bValue.i32[SIMD\_X]) ? aValue.i32[SIMD\_X] : bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = (aValue.i32[SIMD\_Y] < bValue.i32[SIMD\_Y]) ? aValue.i32[SIMD\_Y] : bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = (aValue.i32[SIMD\_Z] < bValue.i32[SIMD\_Z]) ? aValue.i32[SIMD\_Z] : bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = (aValue.i32[SIMD\_W] < bValue.i32[SIMD\_W]) ? aValue.i32[SIMD\_W] : bValue.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpMax(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = (aValue.i32[SIMD\_X] > bValue.i32[SIMD\_X]) ? aValue.i32[SIMD\_X] : bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = (aValue.i32[SIMD\_Y] > bValue.i32[SIMD\_Y]) ? aValue.i32[SIMD\_Y] : bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = (aValue.i32[SIMD\_Z] > bValue.i32[SIMD\_Z]) ? aValue.i32[SIMD\_Z] : bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = (aValue.i32[SIMD\_W] > bValue.i32[SIMD\_W]) ? aValue.i32[SIMD\_W] : bValue.i32[SIMD\_W];

return result;

}

SIMDValue SIMDInt32x4Operation::OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = (aValue.i32[SIMD\_X] < bValue.i32[SIMD\_X]) ? 0xffffffff : 0x0;

result.i32[SIMD\_Y] = (aValue.i32[SIMD\_Y] < bValue.i32[SIMD\_Y]) ? 0xffffffff : 0x0;

result.i32[SIMD\_Z] = (aValue.i32[SIMD\_Z] < bValue.i32[SIMD\_Z]) ? 0xffffffff : 0x0;

result.i32[SIMD\_W] = (aValue.i32[SIMD\_W] < bValue.i32[SIMD\_W]) ? 0xffffffff : 0x0;

return result;

}

SIMDValue SIMDInt32x4Operation::OpEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = (aValue.i32[SIMD\_X] == bValue.i32[SIMD\_X]) ? 0xffffffff : 0x0;

result.i32[SIMD\_Y] = (aValue.i32[SIMD\_Y] == bValue.i32[SIMD\_Y]) ? 0xffffffff : 0x0;

result.i32[SIMD\_Z] = (aValue.i32[SIMD\_Z] == bValue.i32[SIMD\_Z]) ? 0xffffffff : 0x0;

result.i32[SIMD\_W] = (aValue.i32[SIMD\_W] == bValue.i32[SIMD\_W]) ? 0xffffffff : 0x0;

return result;

}

SIMDValue SIMDInt32x4Operation::OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

result.i32[SIMD\_X] = (aValue.i32[SIMD\_X] > bValue.i32[SIMD\_X]) ? 0xffffffff : 0x0;

result.i32[SIMD\_Y] = (aValue.i32[SIMD\_Y] > bValue.i32[SIMD\_Y]) ? 0xffffffff : 0x0;

result.i32[SIMD\_Z] = (aValue.i32[SIMD\_Z] > bValue.i32[SIMD\_Z]) ? 0xffffffff : 0x0;

result.i32[SIMD\_W] = (aValue.i32[SIMD\_W] > bValue.i32[SIMD\_W]) ? 0xffffffff : 0x0;

return result;

}

SIMDValue SIMDInt32x4Operation::OpShiftLeft(const SIMDValue& value, int count)

{

SIMDValue result;

result.i32[SIMD\_X] = value.i32[SIMD\_X] << count;

result.i32[SIMD\_Y] = value.i32[SIMD\_Y] << count;

result.i32[SIMD\_Z] = value.i32[SIMD\_Z] << count;

result.i32[SIMD\_W] = value.i32[SIMD\_W] << count;

return result;

}

SIMDValue SIMDInt32x4Operation::OpShiftRightLogical(const SIMDValue& value, int count)

{

SIMDValue result;

int nIntMin = INT\_MIN; // INT\_MIN = -2147483648 = 0x80000000

int mask = ~((nIntMin >> count) << 1); // now first count bits are 0

// right shift count bits and shift in with 0

result.i32[SIMD\_X] = (value.i32[SIMD\_X] >> count) & mask;

result.i32[SIMD\_Y] = (value.i32[SIMD\_Y] >> count) & mask;

result.i32[SIMD\_Z] = (value.i32[SIMD\_Z] >> count) & mask;

result.i32[SIMD\_W] = (value.i32[SIMD\_W] >> count) & mask;

return result;

}

SIMDValue SIMDInt32x4Operation::OpShiftRightArithmetic(const SIMDValue& value, int count)

{

SIMDValue result;

result.i32[SIMD\_X] = value.i32[SIMD\_X] >> count;

result.i32[SIMD\_Y] = value.i32[SIMD\_Y] >> count;

result.i32[SIMD\_Z] = value.i32[SIMD\_Z] >> count;

result.i32[SIMD\_W] = value.i32[SIMD\_W] >> count;

return result;

}

SIMDValue SIMDInt32x4Operation::OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV)

{

SIMDValue result;

SIMDValue trueResult = SIMDInt32x4Operation::OpAnd(mV, tV);

SIMDValue notValue = SIMDInt32x4Operation::OpNot(mV);

SIMDValue falseResult = SIMDInt32x4Operation::OpAnd(notValue, fV);

result = SIMDInt32x4Operation::OpOr(trueResult, falseResult);

return result;

}

// Get SignMask

int SIMDInt32x4Operation::OpGetSignMask(const SIMDValue& v)

{

int result;

// shift right 31 bits while shifting in with zero

SIMDValue value = SIMDInt32x4Operation::OpShiftRightLogical(v, 31);

// extract sign bit from each lane

int mx = value.i32[SIMD\_X];

int my = value.i32[SIMD\_Y];

int mz = value.i32[SIMD\_Z];

int mw = value.i32[SIMD\_W];

result = mx | my << 1 | mz << 2 | mw << 3;

return result;

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

struct SIMDInt32x4Operation

{

// following are operation wrappers for SIMDInt32x4 general implementation

// input and output are typically SIMDValue

static SIMDValue OpInt32x4(int x, int y, int z, int w);

static SIMDValue OpInt32x4(const SIMDValue& v);

static SIMDValue OpZero();

static SIMDValue OpSplat(int x);

static SIMDValue OpSplat(const SIMDValue& v);

static SIMDValue OpBool(int x, int y, int z, int w);

static SIMDValue OpBool(const SIMDValue& v);

// conversion

static SIMDValue OpFromBool(const SIMDValue& value);

static SIMDValue OpFromFloat32x4(const SIMDValue& value);

static SIMDValue OpFromFloat64x2(const SIMDValue& value);

static SIMDValue OpFromFloat32x4Bits(const SIMDValue& value);

static SIMDValue OpFromFloat64x2Bits(const SIMDValue& value);

// Unary Ops

static SIMDValue OpAbs(const SIMDValue& v);

static SIMDValue OpNeg(const SIMDValue& v);

static SIMDValue OpNot(const SIMDValue& v);

static SIMDValue OpAdd(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpSub(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMul(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpAnd(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpOr (const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpXor(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMin(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMax(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpShiftLeft(const SIMDValue& value, int count);

static SIMDValue OpShiftRightLogical(const SIMDValue& value, int count);

static SIMDValue OpShiftRightArithmetic(const SIMDValue& value, int count);

static SIMDValue OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV);

// Get SignMask

static int OpGetSignMask(const SIMDValue& mV);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if \_M\_IX86 || \_M\_AMD64

namespace Js

{

// SIMD.Int32x4 operation wrappers that cover intrinsics for x86/x64 system

SIMDValue SIMDInt32x4Operation::OpInt32x4(int x, int y, int z, int w)

{

X86SIMDValue x86Result;

x86Result.m128i\_value = \_mm\_set\_epi32(w, z, y, x);

// Sets the 4 signed 32-bit integer values, note in revised order: starts with W below

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpInt32x4(const SIMDValue& v)

{

X86SIMDValue x86Result;

// Sets the 4 signed 32-bit integer values, note in revised order: starts with W below

x86Result.m128i\_value = \_mm\_set\_epi32(v.i32[SIMD\_W], v.i32[SIMD\_Z], v.i32[SIMD\_Y], v.i32[SIMD\_X]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpZero()

{

X86SIMDValue x86Result;

// Sets the 128-bit value to zero

x86Result.m128i\_value = \_mm\_setzero\_si128();

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpSplat(int x)

{

X86SIMDValue x86Result;

// set 4 signed 32-bit integers values to input value x

x86Result.m128i\_value = \_mm\_set1\_epi32(x);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpSplat(const SIMDValue& v)

{

X86SIMDValue x86Result;

// set 4 signed 32-bit integers values to input value(v.i32[SIMD\_X])

x86Result.m128i\_value = \_mm\_set1\_epi32(v.i32[SIMD\_X]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpBool(int x, int y, int z, int w)

{

X86SIMDValue x86Result;

x86Result.m128i\_value = \_mm\_set\_epi32(w, z, y, x); // Sets the 4 signed 32-bit int value

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpBool(const SIMDValue& v)

{

X86SIMDValue x86Result;

// Sets the 4 signed 32-bit integer value

x86Result.m128i\_value = \_mm\_set\_epi32(v.i32[SIMD\_W], v.i32[SIMD\_Z], v.i32[SIMD\_Y], v.i32[SIMD\_X]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

// Conversions

SIMDValue SIMDInt32x4Operation::OpFromBool(const SIMDValue& value)

{

X86SIMDValue x86Result;

// assuming incoming value is any number, construct a new instance of SIMD.int32x4 with

// 0xFFFFFFFF or 0x0 in each lane

int x = value.i32[SIMD\_X] ? 0xFFFFFFFF : 0x0;

int y = value.i32[SIMD\_Y] ? 0xFFFFFFFF : 0x0;

int z = value.i32[SIMD\_Z] ? 0xFFFFFFFF : 0x0;

int w = value.i32[SIMD\_W] ? 0xFFFFFFFF : 0x0;

// Sets the 4 signed 32-bit integer value, starts with W

x86Result.m128i\_value = \_mm\_set\_epi32(w, z, y, x);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpFromFloat32x4(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Converts the 4 single-precision, floating-point values to signed 32-bit integer values

// using truncate, using truncate one instead of \_mm\_cvtps\_epi32

x86Result.m128i\_value = \_mm\_cvttps\_epi32(v.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpFromFloat64x2(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Converts the 2 double-precision, floating-point values to 32-bit signed integers

// using truncate. using truncate one instead of \_mm\_cvtpd\_epi32

x86Result.m128i\_value = \_mm\_cvttpd\_epi32(v.m128d\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpFromFloat32x4Bits(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

\_mm\_store\_ps(x86Result.simdValue.f32, v.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpFromFloat64x2Bits(const SIMDValue& value)

{

return OpFromFloat32x4Bits(value);

}

// Unary Ops

SIMDValue SIMDInt32x4Operation::OpAbs(const SIMDValue& value)

{

SIMDValue result;

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

if (AutoSystemInfo::Data.SSE3Available())

{

x86Result.m128i\_value = \_mm\_abs\_epi32(v.m128i\_value); // only available after SSE3

result = X86SIMDValue::ToSIMDValue(x86Result);

}

else if (AutoSystemInfo::Data.SSE2Available())

{

X86SIMDValue temp, SIGNMASK;

SIGNMASK.m128i\_value = \_mm\_srai\_epi32(v.m128i\_value, 31); // mask = value >> 31

temp.m128i\_value = \_mm\_xor\_si128(v.m128i\_value, SIGNMASK.m128i\_value); // temp = value ^ mask

x86Result.m128i\_value = \_mm\_sub\_epi32(temp.m128i\_value, SIGNMASK.m128i\_value); // temp - mask

result = X86SIMDValue::ToSIMDValue(x86Result);

}

else

{

result.i32[SIMD\_X] = (value.i32[SIMD\_X] < 0) ? -1 \* value.i32[SIMD\_X] : value.i32[SIMD\_X];

result.i32[SIMD\_Y] = (value.i32[SIMD\_Y] < 0) ? -1 \* value.i32[SIMD\_Y] : value.i32[SIMD\_Y];

result.i32[SIMD\_Z] = (value.i32[SIMD\_Z] < 0) ? -1 \* value.i32[SIMD\_Z] : value.i32[SIMD\_Z];

result.i32[SIMD\_W] = (value.i32[SIMD\_W] < 0) ? -1 \* value.i32[SIMD\_W] : value.i32[SIMD\_W];

}

return result;

}

SIMDValue SIMDInt32x4Operation::OpNeg(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue SIGNMASK, temp;

X86SIMDValue negativeOnes = { { -1, -1, -1, -1 } };

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

temp.m128i\_value = \_mm\_andnot\_si128(v.m128i\_value, negativeOnes.m128i\_value); // (~value) & (negative ones)

SIGNMASK.m128i\_value = \_mm\_set1\_epi32(0x00000001); // set SIGNMASK to 1

x86Result.m128i\_value = \_mm\_add\_epi32(SIGNMASK.m128i\_value, temp.m128i\_value);// add 4 integers respectively

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpNot(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue negativeOnes = { { -1, -1, -1, -1 } };

X86SIMDValue temp = X86SIMDValue::ToX86SIMDValue(value);

x86Result.m128i\_value = \_mm\_andnot\_si128(temp.m128i\_value, negativeOnes.m128i\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpAdd(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_add\_epi32(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a + b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpSub(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_sub\_epi32(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a - b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpMul(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

if (AutoSystemInfo::Data.SSE4\_1Available())

{ // a \* b, only available in SSE4

x86Result.m128i\_value = \_mm\_mullo\_epi32(tmpaValue.m128i\_value, tmpbValue.m128i\_value);

result = X86SIMDValue::ToSIMDValue(x86Result);

}

else if (AutoSystemInfo::Data.SSE2Available())

{

// mul 2,0: r0 = a0\*b0; r1 = a2\*b2

\_\_m128i tmp1 = \_mm\_mul\_epu32(tmpaValue.m128i\_value, tmpbValue.m128i\_value);

// mul 3,1: r0=a1\*b1; r1=a3\*b3

\_\_m128i tmp2 = \_mm\_mul\_epu32(\_mm\_srli\_si128(tmpaValue.m128i\_value, 4), \_mm\_srli\_si128(tmpbValue.m128i\_value, 4));

// shuffle x86Results to [63..0] and pack

x86Result.m128i\_value = \_mm\_unpacklo\_epi32(\_mm\_shuffle\_epi32(tmp1, \_MM\_SHUFFLE(0, 0, 2, 0)), \_mm\_shuffle\_epi32(tmp2, \_MM\_SHUFFLE(0, 0, 2, 0)));

result = X86SIMDValue::ToSIMDValue(x86Result);

}

else

{

result.i32[SIMD\_X] = aValue.i32[SIMD\_X] \* bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = aValue.i32[SIMD\_Y] \* bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = aValue.i32[SIMD\_Z] \* bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = aValue.i32[SIMD\_W] \* bValue.i32[SIMD\_W];

}

return result;

}

SIMDValue SIMDInt32x4Operation::OpAnd(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_and\_si128(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a & b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpOr(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_or\_si128(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a | b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpXor(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_xor\_si128(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a ^ b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpMin(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

if (AutoSystemInfo::Data.SSE4\_1Available())

{ // choose the smaller value of the two parameters, only available after SSE4

x86Result.m128i\_value = \_mm\_min\_epi32(tmpaValue.m128i\_value, tmpbValue.m128i\_value);

result = X86SIMDValue::ToSIMDValue(x86Result);

}

else

{

result.i32[SIMD\_X] = (aValue.i32[SIMD\_X] < bValue.i32[SIMD\_X]) ? aValue.i32[SIMD\_X] : bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = (aValue.i32[SIMD\_Y] < bValue.i32[SIMD\_Y]) ? aValue.i32[SIMD\_Y] : bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = (aValue.i32[SIMD\_Z] < bValue.i32[SIMD\_Z]) ? aValue.i32[SIMD\_Z] : bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = (aValue.i32[SIMD\_W] < bValue.i32[SIMD\_W]) ? aValue.i32[SIMD\_W] : bValue.i32[SIMD\_W];

}

return result;

}

SIMDValue SIMDInt32x4Operation::OpMax(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

if (AutoSystemInfo::Data.SSE4\_1Available())

{ // choose the larger value of the two parameters, only available after SSE4

x86Result.m128i\_value = \_mm\_max\_epi32(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a ^ b

result = X86SIMDValue::ToSIMDValue(x86Result);

}

else

{

result.i32[SIMD\_X] = (aValue.i32[SIMD\_X] > bValue.i32[SIMD\_X]) ? aValue.i32[SIMD\_X] : bValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = (aValue.i32[SIMD\_Y] > bValue.i32[SIMD\_Y]) ? aValue.i32[SIMD\_Y] : bValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = (aValue.i32[SIMD\_Z] > bValue.i32[SIMD\_Z]) ? aValue.i32[SIMD\_Z] : bValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = (aValue.i32[SIMD\_W] > bValue.i32[SIMD\_W]) ? aValue.i32[SIMD\_W] : bValue.i32[SIMD\_W];

}

return result;

}

SIMDValue SIMDInt32x4Operation::OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_cmplt\_epi32(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // compare a < b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_cmpeq\_epi32(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // compare a == b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_cmpgt\_epi32(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // compare a > b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpShiftLeft(const SIMDValue& value, int count)

{

X86SIMDValue x86Result;

X86SIMDValue tmpValue = X86SIMDValue::ToX86SIMDValue(value);

// Shifts the 4 signed 32-bit integers in a left by count bits while shifting in zeros

x86Result.m128i\_value = \_mm\_slli\_epi32(tmpValue.m128i\_value, count);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpShiftRightLogical(const SIMDValue& value, int count)

{

X86SIMDValue x86Result;

X86SIMDValue tmpValue = X86SIMDValue::ToX86SIMDValue(value);

// Shifts the 4 signed 32-bit integers in a right by count bits while shifting in zeros

x86Result.m128i\_value = \_mm\_srli\_epi32(tmpValue.m128i\_value, count);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpShiftRightArithmetic(const SIMDValue& value, int count)

{

X86SIMDValue x86Result;

X86SIMDValue tmpValue = X86SIMDValue::ToX86SIMDValue(value);

// Shifts the 4 signed 32-bit integers right by count bits while shifting in the sign bit

x86Result.m128i\_value = \_mm\_srai\_epi32(tmpValue.m128i\_value, count);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt32x4Operation::OpSelect(const SIMDValue& mV, const SIMDValue& tV, const SIMDValue& fV)

{

X86SIMDValue x86Result;

X86SIMDValue maskValue = X86SIMDValue::ToX86SIMDValue(mV);

X86SIMDValue trueValue = X86SIMDValue::ToX86SIMDValue(tV);

X86SIMDValue falseValue = X86SIMDValue::ToX86SIMDValue(fV);

X86SIMDValue tempTrue, tempFalse;

tempTrue.m128i\_value = \_mm\_and\_si128(maskValue.m128i\_value, trueValue.m128i\_value); // mask & T

tempFalse.m128i\_value = \_mm\_andnot\_si128(maskValue.m128i\_value, falseValue.m128i\_value); //!mask & F

x86Result.m128i\_value = \_mm\_or\_si128(tempTrue.m128i\_value, tempFalse.m128i\_value); // tempT | temp F

return X86SIMDValue::ToSIMDValue(x86Result);

}

// Get SignMask

int SIMDInt32x4Operation::OpGetSignMask(const SIMDValue& value)

{

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

// Creates a 4-bit mask from the most significant bits of

// the 4 single-precision, floating-point values

// SIMD review: no suitable integer intrinsics, the float version seems working fine

return \_mm\_movemask\_ps(v.m128\_value);

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "SIMDInt8x16Operation.h"

#if defined(\_M\_ARM32\_OR\_ARM64)

namespace Js

{

SIMDValue SIMDInt8x16Operation::OpInt8x16(int8 x0, int8 x1, int8 x2, int8 x3

, int8 x4, int8 x5, int8 x6, int8 x7

, int8 x8, int8 x9, int8 x10, int8 x11

, int8 x12, int8 x13, int8 x14, int8 x15)

{

SIMDValue result;

result.i8[0] = x0;

result.i8[1] = x1;

result.i8[2] = x2;

result.i8[3] = x3;

result.i8[4] = x4;

result.i8[5] = x5;

result.i8[6] = x6;

result.i8[7] = x7;

result.i8[8] = x8;

result.i8[9] = x9;

result.i8[10] = x10;

result.i8[11] = x11;

result.i8[12] = x12;

result.i8[13] = x13;

result.i8[14] = x14;

result.i8[15] = x15;

return result;

}

SIMDValue SIMDInt8x16Operation::OpInt8x16(const SIMDValue& v)

{// overload function with input parameter as SIMDValue for completeness, may not need

SIMDValue result;

result = v;

return result;

}

SIMDValue SIMDInt8x16Operation::OpZero()

{

SIMDValue result;

result.i8[0] = result.i8[1] = result.i8[2] = result.i8[3] = result.i8[4] = result.i8[5] = result.i8[6] = result.i8[7] = result.i8[8] = result.i8[9] = result.i8[10] = result.i8[11] = result.i8[12] = result.i8[13] = result.i8[14] = result.i8[15] = 0;

return result;

}

SIMDValue SIMDInt8x16Operation::OpSplat(int8 x)

{

SIMDValue result;

result.i8[0] = result.i8[1] = result.i8[2] = result.i8[3] = result.i8[4] = result.i8[5] = result.i8[6]= result.i8[7] = result.i8[8] = result.i8[9]= result.i8[10] = result.i8[11] = result.i8[12]= result.i8[13] = result.i8[14] = result.i8[15] = x;

return result;

}

SIMDValue SIMDInt8x16Operation::OpSplat(const SIMDValue& v) // not in polyfill or spec

{

SIMDValue result;

result.i8[0] = result.i8[1] = result.i8[2] = result.i8[3] = result.i8[4] = result.i8[5] = result.i8[6]= result.i8[7] = result.i8[8] = result.i8[9]= result.i8[10] = result.i8[11] = result.i8[12]= result.i8[13] = result.i8[14] = result.i8[15] = v.i8[0];

return result;

}

SIMDValue SIMDInt8x16Operation::OpFromFloat32x4Bits(const SIMDValue& v)

{

SIMDValue result;

result.f64[SIMD\_X] = v.f64[SIMD\_X];

result.f64[SIMD\_Y] = v.f64[SIMD\_Y];

return result;

}

SIMDValue SIMDInt8x16Operation::OpFromInt32x4Bits(const SIMDValue& v)

{

return OpFromFloat32x4Bits(v);

}

//// Unary Ops

SIMDValue SIMDInt8x16Operation::OpNeg(const SIMDValue& value)

{

SIMDValue result;

result.i8[0] = -1 \* value.i8[0];

result.i8[1] = -1 \* value.i8[1];

result.i8[2] = -1 \* value.i8[2];

result.i8[3] = -1 \* value.i8[3];

result.i8[4] = -1 \* value.i8[4];

result.i8[5] = -1 \* value.i8[5];

result.i8[6] = -1 \* value.i8[6];

result.i8[7] = -1 \* value.i8[7];

result.i8[8] = -1 \* value.i8[8];

result.i8[9] = -1 \* value.i8[9];

result.i8[10] = -1 \* value.i8[10];

result.i8[11] = -1 \* value.i8[11];

result.i8[12] = -1 \* value.i8[12];

result.i8[13] = -1 \* value.i8[13];

result.i8[14] = -1 \* value.i8[14];

result.i8[15] = -1 \* value.i8[15];

return result;

}

SIMDValue SIMDInt8x16Operation::OpNot(const SIMDValue& value)

{

SIMDValue result;

result.i8[0] = ~(value.i8[0]);

result.i8[1] = ~(value.i8[1]);

result.i8[2] = ~(value.i8[2]);

result.i8[3] = ~(value.i8[3]);

result.i8[4] = ~(value.i8[4]);

result.i8[5] = ~(value.i8[5]);

result.i8[6] = ~(value.i8[6]);

result.i8[7] = ~(value.i8[7]);

result.i8[8] = ~(value.i8[8]);

result.i8[9] = ~(value.i8[9]);

result.i8[10] = ~(value.i8[10]);

result.i8[11] = ~(value.i8[11]);

result.i8[12] = ~(value.i8[12]);

result.i8[13] = ~(value.i8[13]);

result.i8[14] = ~(value.i8[14]);

result.i8[15] = ~(value.i8[15]);

return result;

}

SIMDValue SIMDInt8x16Operation::OpAdd(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = aValue.i8[idx] + bValue.i8[idx];

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpSub(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = aValue.i8[idx] - bValue.i8[idx];

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpMul(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = aValue.i8[idx] \* bValue.i8[idx];

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpAnd(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = aValue.i8[idx] & bValue.i8[idx];

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpOr(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = aValue.i8[idx] | bValue.i8[idx];

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpXor(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = aValue.i8[idx] ^ bValue.i8[idx];

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue) // TODO arun: return bool types

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = (aValue.i8[idx] < bValue.i8[idx]) ? 0xff : 0x0;

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpEqual(const SIMDValue& aValue, const SIMDValue& bValue) // TODO arun: return bool types

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = (aValue.i8[idx] == bValue.i8[idx]) ? 0xff : 0x0;

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue) // TODO arun: return bool types

{

SIMDValue result;

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = (aValue.i8[idx] > bValue.i8[idx]) ? 0xff : 0x0; //Return should be bool vector according to spec

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpShiftLeftByScalar(const SIMDValue& value, int8 count)

{

SIMDValue result;

if (count < 0 || count > 8) // Similar to polyfill, maximum shift will happen if the shift amounts and invalid

{

count = 8;

}

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = value.i8[idx] << count;

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpShiftRightLogicalByScalar(const SIMDValue& value, int8 count)

{

SIMDValue result;

int nIntMin = INT\_MIN; // INT\_MIN = -2147483648 = 0x80000000

int mask = ~((nIntMin >> count) << 1); // now first count bits are 0

// right shift count bits and shift in with 0

for (uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = (value.i8[idx] >> count) & mask;

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpShiftRightArithmeticByScalar(const SIMDValue& value, int8 count)

{

SIMDValue result;

if (count < 0 || count > 8) // Similar to polyfill, maximum shift will happen if the shift amounts and invalid

{

count = 8;

}

for(uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = value.i8[idx] >> count;

}

return result;

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

struct SIMDInt8x16Operation

{

// following are operation wrappers for SIMDInt8x16 general implementation

// input and output are typically SIMDValue

static SIMDValue OpInt8x16(int8 x0, int8 x1, int8 x2, int8 x3, int8 x4, int8 x5, int8 x6, int8 x7, int8 x8, int8 x9, int8 x10, int8 x11, int8 x12, int8 x13, int8 x14, int8 x15);

static SIMDValue OpInt8x16(const SIMDValue& v);

static SIMDValue OpZero();

static SIMDValue OpSplat(int8 x);

static SIMDValue OpSplat(const SIMDValue& v);

//// conversion

static SIMDValue OpFromInt32x4Bits(const SIMDValue& value);

static SIMDValue OpFromFloat32x4Bits(const SIMDValue& value);

//// Unary Ops

static SIMDValue OpNeg(const SIMDValue& v);

static SIMDValue OpNot(const SIMDValue& v);

static SIMDValue OpAdd(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpSub(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpMul(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpAnd(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpOr(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpXor(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpEqual(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue);

static SIMDValue OpShiftLeftByScalar(const SIMDValue& value, int8 count);

static SIMDValue OpShiftRightLogicalByScalar(const SIMDValue& value, int8 count);

static SIMDValue OpShiftRightArithmeticByScalar(const SIMDValue& value, int8 count);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if \_M\_IX86 || \_M\_AMD64

namespace Js

{

// SIMD.Int8x16 operation wrappers that cover intrinsics for x86/x64 system

SIMDValue SIMDInt8x16Operation::OpInt8x16(int8 x0, int8 x1, int8 x2, int8 x3

, int8 x4, int8 x5, int8 x6, int8 x7

, int8 x8, int8 x9, int8 x10, int8 x11

, int8 x12, int8 x13, int8 x14, int8 x15)

{

X86SIMDValue x86Result;

// Sets the 16 signed 8-bit integer values, note in revised order: starts with x15 below

x86Result.m128i\_value = \_mm\_set\_epi8((int8)x15, (int8)x14, (int8)x13, (int8)x12, (int8)x11, (int8)x10, (int8)x9, (int8)x8, (int8)x7, (int8)x6, (int8)x5, (int8)x4, (int8)x3, (int8)x2, (int8)x1, (int8)x0);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpInt8x16(const SIMDValue& v)

{

X86SIMDValue x86Result;

// Sets the 16 signed 8-bit integer values, note in revised order: starts with x15 below

x86Result.m128i\_value = \_mm\_set\_epi8(v.i8[15], v.i8[14], v.i8[13], v.i8[12], v.i8[11]

, v.i8[10], v.i8[9], v.i8[8], v.i8[7], v.i8[6]

, v.i8[5], v.i8[4], v.i8[3], v.i8[2], v.i8[1], v.i8[0]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpZero()

{

X86SIMDValue x86Result;

// Sets the 128-bit value to zero

x86Result.m128i\_value = \_mm\_setzero\_si128();

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpSplat(int8 x)

{

X86SIMDValue x86Result;

// set 16 signed 8-bit integers values to input value x

x86Result.m128i\_value = \_mm\_set1\_epi8(x);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpSplat(const SIMDValue& v) // TODO arun: not in spec

{

X86SIMDValue x86Result;

// set 16 signed 8-bit integers values to input value(v.i8[SIMD\_X])

x86Result.m128i\_value = \_mm\_set1\_epi8(v.i8[0]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpFromFloat32x4Bits(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

\_mm\_store\_ps(x86Result.simdValue.f32, v.m128\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpFromInt32x4Bits(const SIMDValue& value)

{

X86SIMDValue x86Result;

x86Result.m128i\_value = \_mm\_set\_epi8(value.i8[15], value.i8[14], value.i8[13], value.i8[12]

, value.i8[11], value.i8[10], value.i8[9], value.i8[8]

, value.i8[7], value.i8[6], value.i8[5], value.i8[4]

, value.i8[5], value.i8[2], value.i8[1], value.i8[0]);

return X86SIMDValue::ToSIMDValue(x86Result);

}

//// Unary Ops

SIMDValue SIMDInt8x16Operation::OpNeg(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue SIGNMASK, temp;

X86SIMDValue negativeOnes = { { -1, -1, -1, -1 } };

X86SIMDValue v = X86SIMDValue::ToX86SIMDValue(value);

temp.m128i\_value = \_mm\_andnot\_si128(v.m128i\_value, negativeOnes.m128i\_value); // (~value) & (negative ones)

SIGNMASK.m128i\_value = \_mm\_set1\_epi8(0x00000001); // set SIGNMASK to 1

x86Result.m128i\_value = \_mm\_add\_epi8(SIGNMASK.m128i\_value, temp.m128i\_value);// add 16 integers respectively

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpNot(const SIMDValue& value)

{

X86SIMDValue x86Result;

X86SIMDValue negativeOnes = { { -1, -1, -1, -1 } };

X86SIMDValue temp = X86SIMDValue::ToX86SIMDValue(value);

x86Result.m128i\_value = \_mm\_andnot\_si128(temp.m128i\_value, negativeOnes.m128i\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpAdd(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_add\_epi8(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a + b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpSub(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_sub\_epi8(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a - b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpMul(const SIMDValue& aValue, const SIMDValue& bValue)

{

SIMDValue result;

X86SIMDValue x86Result;

X86SIMDValue x86tmp1;

X86SIMDValue x86tmp2;

X86SIMDValue x86tmp3;

const \_x86\_SIMDValue X86\_LOWBYTE\_MASK = { 0x00ff00ff, 0x00ff00ff, 0x00ff00ff, 0x00ff00ff };

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

if (AutoSystemInfo::Data.SSE2Available())

{

// (ah\* 2^8 + al) \* (bh \*2^8 + bl) = (ah\*bh\* 2^8 + al\*bh + ah\* bl) \* 2^8 + al \* bl

x86tmp1.m128i\_value = \_mm\_mullo\_epi16(tmpaValue.m128i\_value, tmpbValue.m128i\_value);

x86tmp2.m128i\_value = \_mm\_and\_si128(x86tmp1.m128i\_value, X86\_LOWBYTE\_MASK.m128i\_value);

tmpaValue.m128i\_value = \_mm\_srli\_epi16(tmpaValue.m128i\_value, 8);

tmpbValue.m128i\_value = \_mm\_srli\_epi16(tmpbValue.m128i\_value, 8);

x86tmp3.m128i\_value = \_mm\_mullo\_epi16(tmpaValue.m128i\_value, tmpbValue.m128i\_value);

x86tmp3.m128i\_value = \_mm\_slli\_epi16(x86tmp3.m128i\_value, 8);

x86Result.m128i\_value = \_mm\_or\_si128(x86tmp2.m128i\_value, x86tmp3.m128i\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

else

{

for (uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = aValue.i8[idx] \* bValue.i8[idx];

}

}

return result;

}

SIMDValue SIMDInt8x16Operation::OpAnd(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_and\_si128(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a & b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpOr(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_or\_si128(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a | b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpXor(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_xor\_si128(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // a ^ b

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpLessThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_cmplt\_epi8(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // compare a < b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpEqual(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_cmpeq\_epi8(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // compare a == b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpGreaterThan(const SIMDValue& aValue, const SIMDValue& bValue)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(aValue);

X86SIMDValue tmpbValue = X86SIMDValue::ToX86SIMDValue(bValue);

x86Result.m128i\_value = \_mm\_cmpgt\_epi8(tmpaValue.m128i\_value, tmpbValue.m128i\_value); // compare a > b?

return X86SIMDValue::ToSIMDValue(x86Result);

}

SIMDValue SIMDInt8x16Operation::OpShiftLeftByScalar(const SIMDValue& value, int8 count)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(value);

X86SIMDValue x86tmp1;

const \_x86\_SIMDValue X86\_LOWBYTE\_MASK = { 0x00ff00ff, 0x00ff00ff, 0x00ff00ff, 0x00ff00ff };

const \_x86\_SIMDValue X86\_HIGHBYTE\_MASK = { 0xff00ff00, 0xff00ff00, 0xff00ff00, 0xff00ff00 };

if (count < 0 || count > 8)

count = 8;

if (AutoSystemInfo::Data.SSE2Available())

{

x86tmp1.m128i\_value = \_mm\_and\_si128(tmpaValue.m128i\_value, X86\_HIGHBYTE\_MASK.m128i\_value);

x86tmp1.m128i\_value = \_mm\_slli\_epi16(x86tmp1.m128i\_value, count);

tmpaValue.m128i\_value = \_mm\_slli\_epi16(tmpaValue.m128i\_value, count);

tmpaValue.m128i\_value = \_mm\_and\_si128(tmpaValue.m128i\_value, X86\_LOWBYTE\_MASK.m128i\_value);

x86Result.m128i\_value = \_mm\_or\_si128(tmpaValue.m128i\_value, x86tmp1.m128i\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

else

{

SIMDValue result;

for (uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = value.i8[idx] << count;

}

return result;

}

}

SIMDValue SIMDInt8x16Operation::OpShiftRightLogicalByScalar(const SIMDValue& value, int8 count)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(value);

X86SIMDValue x86tmp1;

const \_x86\_SIMDValue X86\_LOWBYTE\_MASK = { 0x00ff00ff, 0x00ff00ff, 0x00ff00ff, 0x00ff00ff };

const \_x86\_SIMDValue X86\_HIGHBYTE\_MASK = { 0xff00ff00, 0xff00ff00, 0xff00ff00, 0xff00ff00 };

if (count < 0 || count > 8)

count = 8;

if (AutoSystemInfo::Data.SSE2Available())

{

x86tmp1.m128i\_value = \_mm\_and\_si128(tmpaValue.m128i\_value, X86\_LOWBYTE\_MASK.m128i\_value);

x86tmp1.m128i\_value = \_mm\_srli\_epi16(x86tmp1.m128i\_value, count);

tmpaValue.m128i\_value = \_mm\_srli\_epi16(tmpaValue.m128i\_value, count);

tmpaValue.m128i\_value = \_mm\_and\_si128(tmpaValue.m128i\_value, X86\_HIGHBYTE\_MASK.m128i\_value);

x86Result.m128i\_value = \_mm\_or\_si128(tmpaValue.m128i\_value, x86tmp1.m128i\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

else

{

SIMDValue result;

int nIntMin = INT\_MIN; // INT\_MIN = -2147483648 = 0x80000000

int mask = ~((nIntMin >> count) << 1); // now first count bits are 0

// right shift count bits and shift in with 0

result.i8[7] = (value.i8[7] >> count) & mask;

for (uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = (value.i8[idx] >> count) & mask;

}

return result;

}

}

SIMDValue SIMDInt8x16Operation::OpShiftRightArithmeticByScalar(const SIMDValue& value, int8 count)

{

X86SIMDValue x86Result;

X86SIMDValue tmpaValue = X86SIMDValue::ToX86SIMDValue(value);

X86SIMDValue x86tmp1;

const \_x86\_SIMDValue X86\_LOWBYTE\_MASK = { 0x00ff00ff, 0x00ff00ff, 0x00ff00ff, 0x00ff00ff };

const \_x86\_SIMDValue X86\_HIGHBYTE\_MASK = { 0xff00ff00, 0xff00ff00, 0xff00ff00, 0xff00ff00 };

if (count < 0 || count > 8)

{

count = 8;

}

if (AutoSystemInfo::Data.SSE2Available())

{

x86tmp1.m128i\_value = \_mm\_slli\_epi16(tmpaValue.m128i\_value, 8);

x86tmp1.m128i\_value = \_mm\_srai\_epi16(x86tmp1.m128i\_value, count + 8);

x86tmp1.m128i\_value = \_mm\_and\_si128(x86tmp1.m128i\_value, X86\_LOWBYTE\_MASK.m128i\_value);

tmpaValue.m128i\_value = \_mm\_srai\_epi16(tmpaValue.m128i\_value, count);

tmpaValue.m128i\_value = \_mm\_and\_si128(tmpaValue.m128i\_value, X86\_HIGHBYTE\_MASK.m128i\_value);

x86Result.m128i\_value = \_mm\_or\_si128(tmpaValue.m128i\_value, x86tmp1.m128i\_value);

return X86SIMDValue::ToSIMDValue(x86Result);

}

else

{

SIMDValue result;

for (uint idx = 0; idx < 16; ++idx)

{

result.i8[idx] = value.i8[idx] >> count;

}

return result;

}

}

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

// SIMD operations

#include "Language\SIMDFloat32x4Operation.h"

#include "Language\SIMDFloat64x2Operation.h"

#include "Language\SIMDInt32x4Operation.h"

#include "Language\SIMDInt8x16Operation.h"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

namespace Js

{

int32 SIMDCheckTypedArrayIndex(ScriptContext\* scriptContext, Var index)

{

int32 int32Value;

Assert(index != NULL);

int32Value = SIMDCheckInt32Number(scriptContext, index);

return int32Value;

}

int32 SIMDCheckLaneIndex(ScriptContext\* scriptContext, Var lane, const int32 range)

{

int32 int32Value;

Assert(lane != NULL);

int32Value = SIMDCheckInt32Number(scriptContext, lane);

if (int32Value < 0 || int32Value >= range)

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_SimdLaneRangeError);

}

return int32Value;

}

// Is Number with int32 value.

int32 SIMDCheckInt32Number(ScriptContext\* scriptContext, Var value)

{

int32 int32Value;

if (JavascriptNumber::Is(value))

{

if (!JavascriptNumber::TryGetInt32Value(JavascriptNumber::GetValue(value), &int32Value))

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange);

}

}

else if (TaggedInt::Is(value))

{

int32Value = TaggedInt::ToInt32(value);

}

else

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NeedNumber, L"index");

}

return int32Value;

}

template <int laneCount> SIMDValue SIMD128InnerShuffle(SIMDValue src1, SIMDValue src2, int32 lane0, int32 lane1, int32 lane2, int32 lane3)

{

SIMDValue result;

CompileAssert(laneCount == 4 || laneCount == 2);

if (laneCount == 4)

{

result.i32[SIMD\_X] = lane0 < 4 ? src1.i32[lane0] : src2.i32[lane0 - 4];

result.i32[SIMD\_Y] = lane1 < 4 ? src1.i32[lane1] : src2.i32[lane1 - 4];

result.i32[SIMD\_Z] = lane2 < 4 ? src1.i32[lane2] : src2.i32[lane2 - 4];

result.i32[SIMD\_W] = lane3 < 4 ? src1.i32[lane3] : src2.i32[lane3 - 4];

}

else

{

result.f64[SIMD\_X] = lane0 < 2 ? src1.f64[lane0] : src2.f64[lane0 - 2];

result.f64[SIMD\_Y] = lane1 < 2 ? src1.f64[lane1] : src2.f64[lane1 - 2];

}

return result;

}

template <class SIMDType, int laneCount> Var SIMD128SlowShuffle(Var src1, Var src2, Var lane0, Var lane1, Var lane2, Var lane3, int range, ScriptContext\* scriptContext)

{

SIMDType \*a = SIMDType::FromVar(src1);

SIMDType \*b = SIMDType::FromVar(src2);

Assert(a);

Assert(b);

int32 lane0Value = 0;

int32 lane1Value = 0;

int32 lane2Value = 0;

int32 lane3Value = 0;

SIMDValue src1Value = a->GetValue();

SIMDValue src2Value = b->GetValue();

SIMDValue result;

CompileAssert(laneCount == 4 || laneCount == 2);

if (laneCount == 4)

{

lane0Value = SIMDCheckLaneIndex(scriptContext, lane0, range);

lane1Value = SIMDCheckLaneIndex(scriptContext, lane1, range);

lane2Value = SIMDCheckLaneIndex(scriptContext, lane2, range);

lane3Value = SIMDCheckLaneIndex(scriptContext, lane3, range);

Assert(lane0Value >= 0 && lane0Value < range);

Assert(lane1Value >= 0 && lane1Value < range);

Assert(lane2Value >= 0 && lane2Value < range);

Assert(lane3Value >= 0 && lane3Value < range);

result = SIMD128InnerShuffle(src1Value, src2Value, lane0Value, lane1Value, lane2Value, lane3Value);

}

else

{

lane0Value = SIMDCheckLaneIndex(scriptContext, lane0, range);

lane1Value = SIMDCheckLaneIndex(scriptContext, lane1, range);

Assert(lane0Value >= 0 && lane0Value < range);

Assert(lane1Value >= 0 && lane1Value < range);

result = SIMD128InnerShuffle<2>(src1Value, src2Value, lane0Value, lane1Value, lane2Value, lane3Value);

}

return SIMDType::New(&result, scriptContext);

}

template Var SIMD128SlowShuffle<JavascriptSIMDInt32x4, 4> (Var src1, Var src2, Var lane0, Var lane1, Var lane2, Var lane3, int range, ScriptContext\* scriptContext);

template Var SIMD128SlowShuffle<JavascriptSIMDFloat32x4, 4> (Var src1, Var src2, Var lane0, Var lane1, Var lane2, Var lane3, int range, ScriptContext\* scriptContext);

template Var SIMD128SlowShuffle<JavascriptSIMDFloat64x2, 2> (Var src1, Var src2, Var lane0, Var lane1, Var lane2, Var lane3, int range, ScriptContext\* scriptContext);

//Int8x16 LaneAccess

inline int8 SIMD128InnerExtractLaneI16(const SIMDValue& src1, const int32 lane)

{

return src1.i8[lane];

}

inline SIMDValue SIMD128InnerReplaceLaneI16(const SIMDValue& src1, const int32 lane, const int8 value)

{

SIMDValue result = src1;

result.i8[lane] = value;

return result;

}

static inline int8 SIMD128GetLaneValue(JavascriptSIMDInt8x16 \*jsVal, const int laneValue)

{

Assert(jsVal);

return SIMD128InnerExtractLaneI16(jsVal->GetValue(), laneValue);

}

static inline SIMDValue SIMD128SetLaneValue(JavascriptSIMDInt8x16 \*jsVal, const int laneValue, int8 value)

{

Assert(jsVal);

return SIMD128InnerReplaceLaneI16(jsVal->GetValue(), laneValue, value);

}

//Int32x4 LaneAccess

inline int SIMD128InnerExtractLaneI4(const SIMDValue& src1, const int32 lane)

{

return src1.i32[lane];

}

inline SIMDValue SIMD128InnerReplaceLaneI4(const SIMDValue& src1, const int32 lane, const int value)

{

SIMDValue result = src1;

result.i32[lane] = value;

return result;

}

static inline int SIMD128GetLaneValue(JavascriptSIMDInt32x4 \*jsVal, const int laneValue)

{

Assert(jsVal);

return SIMD128InnerExtractLaneI4(jsVal->GetValue(), laneValue);

}

static inline SIMDValue SIMD128SetLaneValue(JavascriptSIMDInt32x4 \*jsVal, const int laneValue, int value)

{

Assert(jsVal);

return SIMD128InnerReplaceLaneI4(jsVal->GetValue(), laneValue, value);

}

//Float32x4 LaneAccess

inline float SIMD128InnerExtractLaneF4(const SIMDValue& src1, const int32 lane)

{

return src1.f32[lane];

}

inline SIMDValue SIMD128InnerReplaceLaneF4(const SIMDValue& src1, const int32 lane, const float value)

{

SIMDValue result = src1;

result.f32[lane] = value;

return result;

}

static inline float SIMD128GetLaneValue(JavascriptSIMDFloat32x4 \*jsVal, const int laneValue)

{

Assert(jsVal);

return SIMD128InnerExtractLaneF4(jsVal->GetValue(), laneValue);

}

static inline SIMDValue SIMD128SetLaneValue(JavascriptSIMDFloat32x4 \*jsVal, const int laneValue, float value)

{

Assert(jsVal);

return SIMD128InnerReplaceLaneF4(jsVal->GetValue(), laneValue, value);

}

template<class SIMDType, int laneCount, typename T>

inline T SIMD128ExtractLane(const Var src, const Var lane, ScriptContext\* scriptContext)

{

SIMDType \*jsVal = SIMDType::FromVar(src);

Assert(jsVal);

int32 laneValue = SIMDCheckLaneIndex(scriptContext, lane, laneCount);

Assert(laneValue >= 0 && laneValue < laneCount);

return SIMD128GetLaneValue(jsVal, laneValue);

}

template<class SIMDType, int laneCount, typename T>

inline SIMDValue SIMD128ReplaceLane(const Var src, const Var lane, const T value, ScriptContext\* scriptContext)

{

SIMDType \*jsVal = SIMDType::FromVar(src);

Assert(jsVal);

int32 laneValue = SIMDCheckLaneIndex(scriptContext, lane, laneCount);

Assert(laneValue >= 0 && laneValue < laneCount);

return SIMD128SetLaneValue(jsVal, laneValue, value);

}

template int8 SIMD128ExtractLane<JavascriptSIMDInt8x16, 16, int8>(Var src, Var lane, ScriptContext\* scriptContext);

template SIMDValue SIMD128ReplaceLane<JavascriptSIMDInt8x16, 16, int8>(Var src, Var lane, int8 value, ScriptContext\* scriptContext);

template int SIMD128ExtractLane<JavascriptSIMDInt32x4, 4, int>(Var src, Var lane, ScriptContext\* scriptContext);

template SIMDValue SIMD128ReplaceLane<JavascriptSIMDInt32x4, 4, int>(Var src, Var lane, int value, ScriptContext\* scriptContext);

template float SIMD128ExtractLane<JavascriptSIMDFloat32x4, 4, float>(Var src, Var lane, ScriptContext\* scriptContext);

template SIMDValue SIMD128ReplaceLane<JavascriptSIMDFloat32x4, 4, float>(Var src, Var lane, float value, ScriptContext\* scriptContext);

bool SIMDIsSupportedTypedArray(Var value)

{

return JavascriptOperators::GetTypeId(value) >= TypeIds\_Int8Array && JavascriptOperators::GetTypeId(value) <= TypeIds\_Float64Array;

}

/\*

Checks if:

1. Array is supported typed array

2. Lane index is a Number/TaggedInt and int32 value

3. Lane index is within array bounds

\*/

SIMDValue\* SIMDCheckTypedArrayAccess(Var arg1, Var arg2, TypedArrayBase \*\*tarray, int32 \*index, uint32 dataWidth, ScriptContext \*scriptContext)

{

if (!SIMDIsSupportedTypedArray(arg1))

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_SimdInvalidArgType, L"Simd typed array access");

}

\*index = SIMDCheckInt32Number(scriptContext, arg2);

// bound check

\*tarray = TypedArrayBase::FromVar(arg1);

uint32 bpe = (\*tarray)->GetBytesPerElement();

int32 offset = (\*index) \* bpe;

if (offset < 0 || (offset + dataWidth) >(int32)(\*tarray)->GetByteLength())

{

JavascriptError::ThrowRangeError(scriptContext, JSERR\_ArgumentOutOfRange, L"Simd typed array access");

}

return (SIMDValue\*)((\*tarray)->GetByteBuffer() + offset);

}

SIMDValue SIMDLdData(SIMDValue \*data, uint8 dataWidth)

{

SIMDValue result = { 0, 0, 0, 0 };

// bitwise copy. Always use integer fields to avoid wrong copy of NaNs.

switch (dataWidth)

{

case 16:

result.i32[SIMD\_W] = data->i32[SIMD\_W];

// fall through

case 12:

result.i32[SIMD\_Z] = data->i32[SIMD\_Z];

// fall through

case 8:

result.i32[SIMD\_Y] = data->i32[SIMD\_Y];

// fall through

case 4:

result.i32[SIMD\_X] = data->i32[SIMD\_X];

break;

default:

Assert(UNREACHED);

}

return result;

}

void SIMDStData(SIMDValue \*data, SIMDValue simdValue, uint8 dataWidth)

{

// bitwise copy. Always use integer fields to avoid wrong copy of NaNs.

switch (dataWidth)

{

case 16:

data->i32[SIMD\_W] = simdValue.i32[SIMD\_W];

// fall through

case 12:

data->i32[SIMD\_Z] = simdValue.i32[SIMD\_Z];

// fall through

case 8:

data->i32[SIMD\_Y] = simdValue.i32[SIMD\_Y];

// fall through

case 4:

data->i32[SIMD\_X] = simdValue.i32[SIMD\_X];

break;

default:

Assert(UNREACHED);

}

}

template <class SIMDType>

Var SIMD128TypedArrayLoad(Var arg1, Var arg2, uint32 dataWidth, ScriptContext \*scriptContext)

{

Assert(dataWidth >= 4 && dataWidth <= 16);

TypedArrayBase \*tarray = NULL;

int32 index = -1;

SIMDValue\* data = NULL;

data = SIMDCheckTypedArrayAccess(arg1, arg2, &tarray, &index, dataWidth, scriptContext);

Assert(tarray != NULL);

Assert(index >= 0);

Assert(data != NULL);

SIMDValue result = SIMDLdData(data, (uint8)dataWidth);

return SIMDType::New(&result, scriptContext);

}

template Var SIMD128TypedArrayLoad<JavascriptSIMDFloat32x4>(Var arg1, Var arg2, uint32 dataWidth, ScriptContext \*scriptContext);

template Var SIMD128TypedArrayLoad<JavascriptSIMDInt32x4>(Var arg1, Var arg2, uint32 dataWidth, ScriptContext \*scriptContext);

template Var SIMD128TypedArrayLoad<JavascriptSIMDFloat64x2>(Var arg1, Var arg2, uint32 dataWidth, ScriptContext \*scriptContext);

template <class SIMDType>

void SIMD128TypedArrayStore(Var arg1, Var arg2, Var simdVar, uint32 dataWidth, ScriptContext \*scriptContext)

{

Assert(dataWidth >= 4 && dataWidth <= 16);

TypedArrayBase \*tarray = NULL;

int32 index = -1;

SIMDValue\* data = NULL;

data = SIMDCheckTypedArrayAccess(arg1, arg2, &tarray, &index, dataWidth, scriptContext);

Assert(tarray != NULL);

Assert(index >= 0);

Assert(data != NULL);

SIMDValue simdValue = SIMDType::FromVar(simdVar)->GetValue();

SIMDStData(data, simdValue, (uint8)dataWidth);

}

template void SIMD128TypedArrayStore<JavascriptSIMDFloat32x4>(Var arg1, Var arg2, Var simdVar, uint32 dataWidth, ScriptContext \*scriptContext);

template void SIMD128TypedArrayStore<JavascriptSIMDInt32x4>(Var arg1, Var arg2, Var simdVar, uint32 dataWidth, ScriptContext \*scriptContext);

template void SIMD128TypedArrayStore<JavascriptSIMDFloat64x2>(Var arg1, Var arg2, Var simdVar, uint32 dataWidth, ScriptContext \*scriptContext);

#if ENABLE\_NATIVE\_CODEGEN

// Maps Simd opcodes which are non-contigous to a zero-based linear space. Used to index a table using an Simd opcode.

uint32 SimdOpcodeAsIndex(Js::OpCode op)

{

if (op <= Js::OpCode::Simd128\_End)

{

return (uint32)((Js::OpCode)op - Js::OpCode::Simd128\_Start);

}

else

{

Assert(op >= Js::OpCode::Simd128\_Start\_Extend && op <= Js::OpCode::Simd128\_End\_Extend);

return (uint32)((Js::OpCode)op - Js::OpCode::Simd128\_Start\_Extend) + (uint32)(Js::OpCode::Simd128\_End - Js::OpCode::Simd128\_Start) + 1;

}

}

#endif

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#define SIMD128\_TYPE\_SPEC\_FLAG Js::Configuration::Global.flags.Simd128TypeSpec

// The representations below assume little-endian.

#define SIMD\_X 0

#define SIMD\_Y 1

#define SIMD\_Z 2

#define SIMD\_W 3

struct \_SIMDValue

{

union{

int i32[4];

float f32[4];

double f64[2];

int8 i8[16];

};

void SetValue(\_SIMDValue value)

{

i32[SIMD\_X] = value.i32[SIMD\_X];

i32[SIMD\_Y] = value.i32[SIMD\_Y];

i32[SIMD\_Z] = value.i32[SIMD\_Z];

i32[SIMD\_W] = value.i32[SIMD\_W];

}

void Zero()

{

f64[SIMD\_X] = f64[SIMD\_Y] = 0;

}

bool operator==(const \_SIMDValue& r)

{

// don't compare f64/f32 because NaN bit patterns will not be considered equal.

return (this->i32[SIMD\_X] == r.i32[SIMD\_X] &&

this->i32[SIMD\_Y] == r.i32[SIMD\_Y] &&

this->i32[SIMD\_Z] == r.i32[SIMD\_Z] &&

this->i32[SIMD\_W] == r.i32[SIMD\_W]);

}

bool IsZero()

{

return (i32[SIMD\_X] == 0 && i32[SIMD\_Y] == 0 && i32[SIMD\_Z] == 0 && i32[SIMD\_W] == 0);

}

};

typedef \_SIMDValue SIMDValue;

// For dictionary use

template <>

struct DefaultComparer<\_SIMDValue>

{

\_\_forceinline static bool Equals(\_SIMDValue x, \_SIMDValue y)

{

return x == y;

}

\_\_forceinline static hash\_t GetHashCode(\_SIMDValue d)

{

return (hash\_t)(d.i32[SIMD\_X] ^ d.i32[SIMD\_Y] ^ d.i32[SIMD\_Z] ^ d.i32[SIMD\_W]);

}

};

#if \_M\_IX86 || \_M\_AMD64

struct \_x86\_SIMDValue

{

union{

\_SIMDValue simdValue;

\_\_m128 m128\_value;

\_\_m128d m128d\_value;

\_\_m128i m128i\_value;

};

static \_x86\_SIMDValue ToX86SIMDValue(const SIMDValue& val)

{

\_x86\_SIMDValue result;

result.simdValue.i32[SIMD\_X] = val.i32[SIMD\_X];

result.simdValue.i32[SIMD\_Y] = val.i32[SIMD\_Y];

result.simdValue.i32[SIMD\_Z] = val.i32[SIMD\_Z];

result.simdValue.i32[SIMD\_W] = val.i32[SIMD\_W];

return result;

}

static SIMDValue ToSIMDValue(const \_x86\_SIMDValue& val)

{

SIMDValue result;

result.i32[SIMD\_X] = val.simdValue.i32[SIMD\_X];

result.i32[SIMD\_Y] = val.simdValue.i32[SIMD\_Y];

result.i32[SIMD\_Z] = val.simdValue.i32[SIMD\_Z];

result.i32[SIMD\_W] = val.simdValue.i32[SIMD\_W];

return result;

}

};

// These global values are 16-byte aligned.

const \_x86\_SIMDValue X86\_ABS\_MASK\_F4 = { 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff };

const \_x86\_SIMDValue X86\_ABS\_MASK\_I4 = { 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff };

const \_x86\_SIMDValue X86\_ABS\_MASK\_D2 = { 0xffffffff, 0x7fffffff, 0xffffffff, 0x7fffffff };

const \_x86\_SIMDValue X86\_NEG\_MASK\_F4 = { 0x80000000, 0x80000000, 0x80000000, 0x80000000 };

const \_x86\_SIMDValue X86\_NEG\_MASK\_D2 = { 0x00000000, 0x80000000, 0x00000000, 0x80000000 };

const \_x86\_SIMDValue X86\_ALL\_ONES\_F4 = { 0x3f800000, 0x3f800000, 0x3f800000, 0x3f800000 }; // {1.0, 1.0, 1.0, 1.0}

const \_x86\_SIMDValue X86\_ALL\_ONES\_I4 = { 0x00000001, 0x00000001, 0x00000001, 0x00000001 }; // {1, 1, 1, 1}

const \_x86\_SIMDValue X86\_ALL\_ONES\_D2 = { 0x00000000, 0x3ff00000, 0x00000000, 0x3ff00000 }; // {1.0, 1.0}

const \_x86\_SIMDValue X86\_ALL\_NEG\_ONES = { 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff };

const \_x86\_SIMDValue X86\_ALL\_ZEROS = { 0x00000000, 0x00000000, 0x00000000, 0x00000000 };

const \_x86\_SIMDValue X86\_LANE\_W\_ZEROS = { 0xffffffff, 0xffffffff, 0xffffffff, 0x00000000 };

typedef \_x86\_SIMDValue X86SIMDValue;

CompileAssert(sizeof(X86SIMDValue) == 16);

#endif

typedef SIMDValue AsmJsSIMDValue; // alias for asmjs

CompileAssert(sizeof(SIMDValue) == 16);

namespace Js {

int32 SIMDCheckTypedArrayIndex(ScriptContext\* scriptContext, Var index);

int32 SIMDCheckLaneIndex(ScriptContext\* scriptContext, Var lane, const int32 range = 4);

template <int laneCount = 4>

SIMDValue SIMD128InnerShuffle(SIMDValue src1, SIMDValue src2, int32 lane0, int32 lane1, int32 lane2, int32 lane3);

template <class SIMDType, int laneCount = 4>

Var SIMD128SlowShuffle(Var src1, Var src2, Var lane0, Var lane1, Var lane2, Var lane3, int range, ScriptContext\* scriptContext);

//Lane Access

template<class SIMDType, int laneCount, typename T>

inline T SIMD128ExtractLane(Var src, Var lane, ScriptContext\* scriptContext);

template<class SIMDType, int laneCount, typename T>

inline SIMDValue SIMD128ReplaceLane(Var src, Var lane, T value, ScriptContext\* scriptContext);

//Lane Access

template<class SIMDType, int laneCount, typename T>

inline T SIMD128ExtractLane(Var src, Var lane, ScriptContext\* scriptContext);

template<class SIMDType, int laneCount, typename T>

inline SIMDValue SIMD128ReplaceLane(Var src, Var lane, T value, ScriptContext\* scriptContext);

SIMDValue SIMD128InnerReplaceLaneF4(const SIMDValue& src1, const int32 lane, const float value);

float SIMD128InnerExtractLaneF4(const SIMDValue& src1, const int32 lane);

SIMDValue SIMD128InnerReplaceLaneI4(const SIMDValue& src1, const int32 lane, const int value);

int SIMD128InnerExtractLaneI4(const SIMDValue& src1, const int32 lane);

SIMDValue SIMD128InnerReplaceLaneI16(const SIMDValue& src1, const int32 lane, const int8 value);

int8 SIMD128InnerExtractLaneI16(const SIMDValue& src1, const int32 lane);

int32 SIMDCheckInt32Number(ScriptContext\* scriptContext, Var value);

bool SIMDIsSupportedTypedArray(Var value);

SIMDValue\* SIMDCheckTypedArrayAccess(Var arg1, Var arg2, TypedArrayBase \*\*tarray, int32 \*index, uint8 dataWidth, ScriptContext \*scriptContext);

AsmJsSIMDValue SIMDLdData(AsmJsSIMDValue \*data, uint8 dataWidth);

void SIMDStData(AsmJsSIMDValue \*data, AsmJsSIMDValue simdValue, uint8 dataWidth);

template <class SIMDType>

Var SIMD128TypedArrayLoad(Var arg1, Var arg2, uint32 dataWidth, ScriptContext \*scriptContext);

template <class SIMDType>

void SIMD128TypedArrayStore(Var arg1, Var arg2, Var simdVar, uint32 dataWidth, ScriptContext \*scriptContext);

enum class OpCode : ushort;

uint32 SimdOpcodeAsIndex(Js::OpCode op);

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if ENABLE\_PROFILE\_INFO

#ifdef ENABLE\_WININET\_PROFILE\_DATA\_CACHE

#include "activscp\_private.h"

#endif

namespace Js

{

ExecutionFlags

SourceDynamicProfileManager::IsFunctionExecuted(Js::LocalFunctionId functionId)

{

if (cachedStartupFunctions == nullptr || cachedStartupFunctions->Length() <= functionId)

{

return ExecutionFlags\_HasNoInfo;

}

return (ExecutionFlags)cachedStartupFunctions->Test(functionId);

}

DynamicProfileInfo \*

SourceDynamicProfileManager::GetDynamicProfileInfo(FunctionBody \* functionBody)

{

Js::LocalFunctionId functionId = functionBody->GetLocalFunctionId();

DynamicProfileInfo \* dynamicProfileInfo = nullptr;

if (dynamicProfileInfoMap.Count() > 0 && dynamicProfileInfoMap.TryGetValue(functionId, &dynamicProfileInfo))

{

if (dynamicProfileInfo->MatchFunctionBody(functionBody))

{

return dynamicProfileInfo;

}

#if DBG\_DUMP

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::DynamicProfilePhase))

{

Output::Print(L"TRACE: DynamicProfile: Profile data rejected for function %d in %s\n",

functionId, functionBody->GetSourceContextInfo()->url);

Output::Flush();

}

#endif

// NOTE: We have profile mismatch, we can invalidate all other profile here.

}

return nullptr;

}

void

SourceDynamicProfileManager::Reset(uint numberOfFunctions)

{

dynamicProfileInfoMap.Clear();

}

void SourceDynamicProfileManager::UpdateDynamicProfileInfo(LocalFunctionId functionId, DynamicProfileInfo \* dynamicProfileInfo)

{

Assert(dynamicProfileInfo != nullptr);

dynamicProfileInfoMap.Item(functionId, dynamicProfileInfo);

}

void SourceDynamicProfileManager::MarkAsExecuted(LocalFunctionId functionId)

{

Assert(startupFunctions != nullptr);

Assert(functionId <= startupFunctions->Length());

startupFunctions->Set(functionId);

}

void SourceDynamicProfileManager::EnsureStartupFunctions(uint numberOfFunctions)

{

Assert(numberOfFunctions != 0);

if(!startupFunctions || numberOfFunctions > startupFunctions->Length())

{

BVFixed\* oldStartupFunctions = this->startupFunctions;

startupFunctions = BVFixed::New(numberOfFunctions, this->GetRecycler());

if(oldStartupFunctions)

{

this->startupFunctions->Copy(oldStartupFunctions);

}

}

}

//

// Enables re-use of profile managers across script contexts - on every re-use the

// previous script contexts list of startup functions are transferred over as input to this new script context.

//

void SourceDynamicProfileManager::Reuse()

{

AssertMsg(profileDataCache == nullptr, "Persisted profiles cannot be re-used");

cachedStartupFunctions = startupFunctions;

}

//

// Loads the profile from the WININET cache

//

bool SourceDynamicProfileManager::LoadFromProfileCache(IActiveScriptDataCache\* profileDataCache, LPCWSTR url)

{

#ifdef ENABLE\_WININET\_PROFILE\_DATA\_CACHE

AssertMsg(CONFIG\_FLAG(WininetProfileCache), "Profile caching should be enabled for us to get here");

Assert(profileDataCache);

AssertMsg(!IsProfileLoadedFromWinInet(), "Duplicate profile cache loading?");

// Keep a copy of this and addref it

profileDataCache->AddRef();

this->profileDataCache = profileDataCache;

IStream\* readStream;

HRESULT hr = profileDataCache->GetReadDataStream(&readStream);

if(SUCCEEDED(hr))

{

Assert(readStream != nullptr);

// stream reader owns the stream and will close it on destruction

SimpleStreamReader streamReader(readStream);

DWORD jscriptMajorVersion;

DWORD jscriptMinorVersion;

if(FAILED(AutoSystemInfo::GetJscriptFileVersion(&jscriptMajorVersion, &jscriptMinorVersion)))

{

return false;

}

DWORD majorVersion;

if(!streamReader.Read(&majorVersion) || majorVersion != jscriptMajorVersion)

{

return false;

}

DWORD minorVersion;

if(!streamReader.Read(&minorVersion) || minorVersion != jscriptMinorVersion)

{

return false;

}

uint numberOfFunctions;

if(!streamReader.Read(&numberOfFunctions) || numberOfFunctions > MAX\_FUNCTION\_COUNT)

{

return false;

}

BVFixed\* functions = BVFixed::New(numberOfFunctions, this->recycler);

if(!streamReader.ReadArray(functions->GetData(), functions->WordCount()))

{

return false;

}

this->cachedStartupFunctions = functions;

OUTPUT\_TRACE(Js::DynamicProfilePhase, L"Profile load succeeded. Function count: %d %s\n", numberOfFunctions, url);

#if DBG\_DUMP

if(PHASE\_TRACE1(Js::DynamicProfilePhase) && Js::Configuration::Global.flags.Verbose)

{

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Profile loaded:\n");

functions->Dump();

}

#endif

return true;

}

else if (hr == HRESULT\_FROM\_WIN32(ERROR\_WRITE\_PROTECT))

{

this->isNonCachableScript = true;

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Profile load failed. Non-cacheable resource. %s\n", url);

}

else

{

OUTPUT\_TRACE(Js::DynamicProfilePhase, L"Profile load failed. No read stream. %s\n", url);

}

#endif

return false;

}

//

// Saves the profile to the WININET cache and returns the bytes written

//

uint SourceDynamicProfileManager::SaveToProfileCacheAndRelease(SourceContextInfo\* info)

{

uint bytesWritten = 0;

#ifdef ENABLE\_WININET\_PROFILE\_DATA\_CACHE

if(profileDataCache)

{

if(ShouldSaveToProfileCache(info))

{

OUTPUT\_TRACE(Js::DynamicProfilePhase, L"Saving profile. Number of functions: %d Url: %s...\n", startupFunctions->Length(), info->url);

bytesWritten = SaveToProfileCache();

if(bytesWritten == 0)

{

OUTPUT\_TRACE(Js::DynamicProfilePhase, L"Profile saving FAILED\n");

}

}

profileDataCache->Release();

profileDataCache = nullptr;

}

#endif

return bytesWritten;

}

//

// Saves the profile to the WININET cache

//

uint SourceDynamicProfileManager::SaveToProfileCache()

{

AssertMsg(CONFIG\_FLAG(WininetProfileCache), "Profile caching should be enabled for us to get here");

Assert(startupFunctions);

uint bytesWritten = 0;

#ifdef ENABLE\_WININET\_PROFILE\_DATA\_CACHE

//TODO: Add some diffing logic to not write unless necessary

IStream\* writeStream;

HRESULT hr = profileDataCache->GetWriteDataStream(&writeStream);

if(FAILED(hr))

{

return 0;

}

Assert(writeStream != nullptr);

// stream writer owns the stream and will close it on destruction

SimpleStreamWriter streamWriter(writeStream);

DWORD jscriptMajorVersion;

DWORD jscriptMinorVersion;

if(FAILED(AutoSystemInfo::GetJscriptFileVersion(&jscriptMajorVersion, &jscriptMinorVersion)))

{

return 0;

}

if(!streamWriter.Write(jscriptMajorVersion))

{

return 0;

}

if(!streamWriter.Write(jscriptMinorVersion))

{

return 0;

}

if(!streamWriter.Write(startupFunctions->Length()))

{

return 0;

}

if(streamWriter.WriteArray(startupFunctions->GetData(), startupFunctions->WordCount()))

{

STATSTG stats;

if(SUCCEEDED(writeStream->Stat(&stats, STATFLAG\_NONAME)))

{

bytesWritten = stats.cbSize.LowPart;

Assert(stats.cbSize.LowPart > 0);

AssertMsg(stats.cbSize.HighPart == 0, "We should not be writing such long data that the high part is non-zero");

}

hr = profileDataCache->SaveWriteDataStream(writeStream);

if(FAILED(hr))

{

return 0;

}

#if DBG\_DUMP

if(PHASE\_TRACE1(Js::DynamicProfilePhase) && Js::Configuration::Global.flags.Verbose)

{

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Saved profile:\n");

startupFunctions->Dump();

}

#endif

}

#endif

return bytesWritten;

}

//

// Do not save the profile:

// - If it is a non-cacheable WININET resource

// - If there are no or small number of functions executed

// - If there is not substantial difference in number of functions executed.

//

bool SourceDynamicProfileManager::ShouldSaveToProfileCache(SourceContextInfo\* info) const

{

if(isNonCachableScript)

{

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Skipping save of profile. Non-cacheable resource. %s\n", info->url);

return false;

}

if(!startupFunctions || startupFunctions->Length() <= DEFAULT\_CONFIG\_MinProfileCacheSize)

{

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Skipping save of profile. Small number of functions. %s\n", info->url);

return false;

}

if(cachedStartupFunctions)

{

AssertMsg(cachedStartupFunctions != startupFunctions, "Ensure they are not shallow copies of each other - Reuse() does this for dynamic sources. We should not be invoked for dynamic sources");

uint numberOfBitsDifferent = cachedStartupFunctions->DiffCount(startupFunctions);

uint saveThreshold = (cachedStartupFunctions->Length() \* DEFAULT\_CONFIG\_ProfileDifferencePercent) / 100;

if(numberOfBitsDifferent <= saveThreshold)

{

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Skipping save of profile. Number of functions different: %d %s\n", numberOfBitsDifferent, info->url);

return false;

}

else

{

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Number of functions different: %d ", numberOfBitsDifferent);

}

}

return true;

}

SourceDynamicProfileManager \*

SourceDynamicProfileManager::LoadFromDynamicProfileStorage(SourceContextInfo\* info, ScriptContext\* scriptContext, IActiveScriptDataCache\* profileDataCache)

{

SourceDynamicProfileManager\* manager = nullptr;

Recycler\* recycler = scriptContext->GetRecycler();

#ifdef DYNAMIC\_PROFILE\_STORAGE

if(DynamicProfileStorage::IsEnabled() && info->url != nullptr)

{

manager = DynamicProfileStorage::Load(info->url, [recycler](char const \* buffer, uint length) -> SourceDynamicProfileManager \*

{

BufferReader reader(buffer, length);

return SourceDynamicProfileManager::Deserialize(&reader, recycler);

});

}

#endif

if(manager == nullptr)

{

manager = RecyclerNew(recycler, SourceDynamicProfileManager, recycler);

}

if(profileDataCache != nullptr)

{

bool profileLoaded = manager->LoadFromProfileCache(profileDataCache, info->url);

if(profileLoaded)

{

JS\_ETW(EventWriteJSCRIPT\_PROFILE\_LOAD(info->dwHostSourceContext, scriptContext));

}

}

return manager;

}

#ifdef DYNAMIC\_PROFILE\_STORAGE

void

SourceDynamicProfileManager::SaveDynamicProfileInfo(LocalFunctionId functionId, DynamicProfileInfo \* dynamicProfileInfo)

{

Assert(dynamicProfileInfo->GetFunctionBody()->HasExecutionDynamicProfileInfo());

dynamicProfileInfoMap.Item(functionId, dynamicProfileInfo);

}

template <typename T>

SourceDynamicProfileManager \*

SourceDynamicProfileManager::Deserialize(T \* reader, Recycler\* recycler)

{

uint functionCount;

if (!reader->Peek(&functionCount))

{

return nullptr;

}

BVFixed \* startupFunctions = BVFixed::New(functionCount, recycler);

if (!reader->ReadArray(((char \*)startupFunctions),

BVFixed::GetAllocSize(functionCount)))

{

return nullptr;

}

uint profileCount;

if (!reader->Read(&profileCount))

{

return nullptr;

}

ThreadContext\* threadContext = ThreadContext::GetContextForCurrentThread();

SourceDynamicProfileManager \* sourceDynamicProfileManager = RecyclerNew(threadContext->GetRecycler(), SourceDynamicProfileManager, recycler);

sourceDynamicProfileManager->cachedStartupFunctions = startupFunctions;

#if DBG\_DUMP

if(Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase))

{

Output::Print(L"Loaded: Startup functions bit vector:");

startupFunctions->Dump();

}

#endif

for (uint i = 0; i < profileCount; i++)

{

Js::LocalFunctionId functionId;

DynamicProfileInfo \* dynamicProfileInfo = DynamicProfileInfo::Deserialize(reader, recycler, &functionId);

if (dynamicProfileInfo == nullptr || functionId >= functionCount)

{

return nullptr;

}

sourceDynamicProfileManager->dynamicProfileInfoMap.Add(functionId, dynamicProfileInfo);

}

return sourceDynamicProfileManager;

}

template <typename T>

bool

SourceDynamicProfileManager::Serialize(T \* writer)

{

// To simulate behavior of in memory profile cache - let's keep functions marked as executed if they were loaded

// to be so from the profile - this helps with ensure inlined functions are marked as executed.

if(!this->startupFunctions)

{

this->startupFunctions = const\_cast<BVFixed\*>(this->cachedStartupFunctions);

}

else if(cachedStartupFunctions && this->cachedStartupFunctions->Length() == this->startupFunctions->Length())

{

this->startupFunctions->Or(cachedStartupFunctions);

}

if(this->startupFunctions)

{

#if DBG\_DUMP

if(Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase))

{

Output::Print(L"Saving: Startup functions bit vector:");

this->startupFunctions->Dump();

}

#endif

size\_t bvSize = BVFixed::GetAllocSize(this->startupFunctions->Length()) ;

if (!writer->WriteArray((char \*)this->startupFunctions, bvSize)

|| !writer->Write(this->dynamicProfileInfoMap.Count()))

{

return false;

}

}

for (int i = 0; i < this->dynamicProfileInfoMap.Count(); i++)

{

DynamicProfileInfo \* dynamicProfileInfo = this->dynamicProfileInfoMap.GetValueAt(i);

if (dynamicProfileInfo == nullptr || !dynamicProfileInfo->HasFunctionBody())

{

continue;

}

if (!dynamicProfileInfo->Serialize(writer))

{

return false;

}

}

return true;

}

void

SourceDynamicProfileManager::SaveToDynamicProfileStorage(wchar\_t const \* url)

{

Assert(DynamicProfileStorage::IsEnabled());

BufferSizeCounter counter;

if (!this->Serialize(&counter))

{

return;

}

if (counter.GetByteCount() > UINT\_MAX)

{

// too big

return;

}

char \* record = DynamicProfileStorage::AllocRecord(static\_cast<DWORD>(counter.GetByteCount()));

#if DBG\_DUMP

if (PHASE\_STATS1(DynamicProfilePhase))

{

Output::Print(L"%-180s : %d bytes\n", url, counter.GetByteCount());

}

#endif

BufferWriter writer(DynamicProfileStorage::GetRecordBuffer(record), counter.GetByteCount());

if (!this->Serialize(&writer))

{

Assert(false);

DynamicProfileStorage::DeleteRecord(record);

}

DynamicProfileStorage::SaveRecord(url, record);

}

#endif

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

class SourceContextInfo;

#if ENABLE\_PROFILE\_INFO

namespace Js

{

enum ExecutionFlags : BYTE

{

ExecutionFlags\_NotExecuted = 0x00,

ExecutionFlags\_Executed = 0x01,

ExecutionFlags\_HasNoInfo = 0x02

};

//

// For every source file, an instance of SourceDynamicProfileManager is used to save/load data.

// It uses the WININET cache to save/load profile data.

// For testing scenarios enabled using DYNAMIC\_PROFILE\_STORAGE macro, this can persist the profile info into a file as well.

class SourceDynamicProfileManager

{

public:

SourceDynamicProfileManager(Recycler\* allocator) : isNonCachableScript(false), cachedStartupFunctions(nullptr), recycler(allocator), dynamicProfileInfoMap(allocator), startupFunctions(nullptr), profileDataCache(nullptr) {}

ExecutionFlags IsFunctionExecuted(Js::LocalFunctionId functionId);

DynamicProfileInfo \* GetDynamicProfileInfo(FunctionBody \* functionBody);

Recycler\* GetRecycler() { return recycler; }

void UpdateDynamicProfileInfo(LocalFunctionId functionId, DynamicProfileInfo \* dynamicProfileInfo);

void MarkAsExecuted(LocalFunctionId functionId);

static SourceDynamicProfileManager \* LoadFromDynamicProfileStorage(SourceContextInfo\* info, ScriptContext\* scriptContext, IActiveScriptDataCache\* profileDataCache);

void EnsureStartupFunctions(uint numberOfFunctions);

void Reuse();

uint SaveToProfileCacheAndRelease(SourceContextInfo\* info);

bool IsProfileLoaded() { return cachedStartupFunctions != nullptr; }

bool IsProfileLoadedFromWinInet() { return profileDataCache != nullptr; }

bool LoadFromProfileCache(IActiveScriptDataCache\* profileDataCache, LPCWSTR url);

IActiveScriptDataCache\* GetProfileCache() { return profileDataCache; }

uint GetStartupFunctionsLength() { return (this->startupFunctions ? this->startupFunctions->Length() : 0); }

private:

friend class DynamicProfileInfo;

Recycler\* recycler;

#ifdef DYNAMIC\_PROFILE\_STORAGE

void SaveDynamicProfileInfo(LocalFunctionId functionId, DynamicProfileInfo \* dynamicProfileInfo);

void SaveToDynamicProfileStorage(wchar\_t const \* url);

template <typename T>

static SourceDynamicProfileManager \* Deserialize(T \* reader, Recycler\* allocator);

template <typename T>

bool Serialize(T \* writer);

#endif

uint SaveToProfileCache();

bool ShouldSaveToProfileCache(SourceContextInfo\* info) const;

void Reset(uint numberOfFunctions);

//------ Private data members -------- /

private:

bool isNonCachableScript; // Indicates if this script can be cached in WININET

IActiveScriptDataCache\* profileDataCache; // WININET based cache to store profile info

BVFixed\* startupFunctions; // Bit vector representing functions that are executed at startup

BVFixed const \* cachedStartupFunctions; // Bit vector representing functions executed at startup that are loaded from a persistent or in-memory cache

// It's not modified but used as an input for deferred parsing/bytecodegen

JsUtil::BaseDictionary<LocalFunctionId, DynamicProfileInfo \*, Recycler, PowerOf2SizePolicy> dynamicProfileInfoMap;

static const uint MAX\_FUNCTION\_COUNT = 10000; // Consider data corrupt if there are more functions than this

//

// Simple read-only wrapper around IStream - templatized and returns boolean result to indicate errors

//

class SimpleStreamReader

{

public:

SimpleStreamReader(IStream\* stream) : stream(stream) {}

~SimpleStreamReader()

{

this->Close();

}

template <typename T>

bool Read(T \* data)

{

ULONG bytesRead;

HRESULT hr = stream->Read(data, sizeof(T), &bytesRead);

// hr is S\_FALSE if bytesRead < sizeOf(T)

return hr == S\_OK;

}

template <typename T>

bool ReadArray(\_Out\_writes\_(len) T \* data, ULONG len)

{

ULONG bytesSize = sizeof(T) \* len;

ULONG bytesRead;

HRESULT hr = stream->Read(data, bytesSize, &bytesRead);

// hr is S\_FALSE if bytesRead < bytesSize

return hr == S\_OK;

}

private:

void Close()

{

Assert(stream);

stream->Release();

stream = NULL;

}

IStream\* stream;

};

//

// Simple write-only wrapper around IStream - templatized and returns boolean result to indicate errors

//

class SimpleStreamWriter

{

public:

SimpleStreamWriter(IStream\* stream) : stream(stream) {}

~SimpleStreamWriter()

{

this->Close();

}

template <typename T>

bool Write(\_In\_ T const& data)

{

ULONG bytesWritten;

HRESULT hr = stream->Write(&data, sizeof(T), &bytesWritten);

// hr is S\_FALSE if bytesRead < sizeOf(T)

return hr == S\_OK;

}

template <typename T>

bool WriteArray(\_In\_reads\_(len) T \* data, \_In\_ ULONG len)

{

ULONG bytesSize = sizeof(T) \* len;

ULONG bytesWritten;

HRESULT hr = stream->Write(data, bytesSize, &bytesWritten);

// hr is S\_FALSE if bytesRead < bytesSize

return hr == S\_OK;

}

private:

void Close()

{

Assert(stream);

stream->Release();

stream = NULL;

}

IStream\* stream;

};

};

};

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "shlwapi.h"

namespace Js {

uint64 StackTraceArguments::ObjectToTypeCode(Js::Var object)

{

switch(JavascriptOperators::GetTypeId(object))

{

case TypeIds\_Null:

return nullValue;

case TypeIds\_Undefined:

return undefinedValue;

case TypeIds\_Boolean:

return booleanValue;

case TypeIds\_String:

return stringValue;

case TypeIds\_Symbol:

return symbolValue;

case TypeIds\_Number:

if (Js::JavascriptNumber::IsNan(JavascriptNumber::GetValue(object)))

{

return nanValue;

}

else

{

return numberValue;

}

case TypeIds\_Integer:

case TypeIds\_Int64Number:

case TypeIds\_UInt64Number:

return numberValue;

}

return objectValue;

}

JavascriptString \*StackTraceArguments::TypeCodeToTypeName(unsigned typeCode, ScriptContext \*scriptContext)

{

switch(typeCode)

{

case nullValue:

return scriptContext->GetLibrary()->GetNullDisplayString();

case undefinedValue:

return scriptContext->GetLibrary()->GetUndefinedDisplayString();

case booleanValue:

return scriptContext->GetLibrary()->GetBooleanTypeDisplayString();

case stringValue:

return scriptContext->GetLibrary()->GetStringTypeDisplayString();

case nanValue:

return scriptContext->GetLibrary()->GetNaNDisplayString();

case numberValue:

return scriptContext->GetLibrary()->GetNumberTypeDisplayString();

case symbolValue:

return scriptContext->GetLibrary()->GetSymbolTypeDisplayString();

case objectValue:

return scriptContext->GetLibrary()->GetObjectTypeDisplayString();

default:

AssertMsg(0, "Unknown type code");

return scriptContext->GetLibrary()->GetEmptyString();

}

}

void StackTraceArguments::Init(const JavascriptStackWalker &walker)

{

types = 0;

if (!walker.IsCallerGlobalFunction())

{

int64 numberOfArguments = walker.GetCallInfo()->Count;

if (numberOfArguments > 0) numberOfArguments --; // Don't consider 'this'

if (walker.GetCallInfo()->Flags & Js::CallFlags\_ExtraArg)

{

Assert(numberOfArguments > 0 );

// skip the last FrameDisplay argument.

numberOfArguments--;

}

for (int64 j = 0; j < numberOfArguments && j < MaxNumberOfDisplayedArgumentsInStack; j ++)

{

types |= ObjectToTypeCode(walker.GetJavascriptArgs()[j]) << 3\*j; // maximal code is 7, so we can use 3 bits to store it

}

if (numberOfArguments > MaxNumberOfDisplayedArgumentsInStack)

{

types |= fTooManyArgs; // two upper bits are flags

}

}

else

{

types |= fCallerIsGlobal; // two upper bits are flags

}

}

HRESULT StackTraceArguments::ToString(LPCWSTR functionName, Js::ScriptContext \*scriptContext, \_In\_ LPCWSTR \*outResult) const

{

HRESULT hr = S\_OK;

uint64 argumentsTypes = types;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

CompoundString \*const stringBuilder = CompoundString::NewWithCharCapacity(40, scriptContext->GetLibrary());

stringBuilder->AppendCharsSz(functionName);

bool calleIsGlobalFunction = (argumentsTypes & fCallerIsGlobal) != 0;

bool toManyArgs = (argumentsTypes & fTooManyArgs) != 0;

argumentsTypes &= ~fCallerIsGlobal; // erase flags to prevent them from being treated as values

argumentsTypes &= ~fTooManyArgs;

if (!calleIsGlobalFunction)

{

stringBuilder->AppendChars(L'(');

}

for (uint64 i = 0; i < MaxNumberOfDisplayedArgumentsInStack && argumentsTypes != 0; i ++)

{

if (i > 0)

{

stringBuilder->AppendChars(L", ");

}

stringBuilder->AppendChars(TypeCodeToTypeName(argumentsTypes & 7, scriptContext)); // we use 3 bits to store one code

argumentsTypes >>= 3;

}

if (toManyArgs)

{

stringBuilder->AppendChars(L", ...");

}

if (!calleIsGlobalFunction)

{

stringBuilder->AppendChars(L')');

}

\*outResult = stringBuilder->GetString();

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT(hr);

return hr;

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

namespace Js {

class StackTraceArguments

{

private:

static const uint64 MaxNumberOfDisplayedArgumentsInStack = 20; // 1 << (3\*MaxNumberOfDisplayedArgumentsInStack + 1) must fit in uint64 (or you have to change it's type)

static const uint64 fCallerIsGlobal = 1ull << (3\*MaxNumberOfDisplayedArgumentsInStack + 1);

static const uint64 fTooManyArgs = 1ull << (3\*MaxNumberOfDisplayedArgumentsInStack);

static uint64 ObjectToTypeCode(Js::Var object);

static JavascriptString \*TypeCodeToTypeName(unsigned typeCode, ScriptContext \*scriptContext);

// We use 3 bits to store the value type. If we add another type, we need to use more bits.

static enum valueTypes

{

nullValue = 0,

undefinedValue = 1,

booleanValue = 2,

stringValue = 3,

nanValue = 4,

numberValue = 5,

symbolValue = 6,

objectValue = 7

};

uint64 types;

public:

HRESULT ToString(LPCWSTR functionName, Js::ScriptContext \*scriptContext, \_In\_ LPCWSTR\* outResult) const;

void Init(const JavascriptStackWalker &walker);

StackTraceArguments() : types(fCallerIsGlobal) {}

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

namespace Js

{

Var TaggedInt::Negate(Var aRight,ScriptContext\* scriptContext)

{

int32 nValue = ToInt32(aRight);

return JavascriptNumber::ToVar(-nValue,scriptContext);

}

Var TaggedInt::Not(Var aRight,ScriptContext\* scriptContext)

{

int32 nValue = ToInt32(aRight);

return JavascriptNumber::ToVar(~nValue,scriptContext);

}

// Explicitly marking noinline and stdcall since this is called from inline asm

\_\_declspec(noinline) Var \_\_stdcall TaggedInt::OverflowHelper(int overflowValue, ScriptContext\* scriptContext)

{

Assert( IsOverflow(overflowValue) );

return JavascriptNumber::NewInlined(static\_cast<double>(overflowValue), scriptContext);

}

// noinline since it's a rare edge case and we don't want to bloat mainline code

\_\_declspec(noinline) Var TaggedInt::DivideByZero(int nLeft, ScriptContext\* scriptContext)

{

if (nLeft == 0)

{

return scriptContext->GetLibrary()->GetNaN();

}

if (nLeft < 0)

{

return scriptContext->GetLibrary()->GetNegativeInfinite();

}

return scriptContext->GetLibrary()->GetPositiveInfinite();

}

Var TaggedInt::Divide(Var aLeft,Var aRight,ScriptContext\* scriptContext)

{

int nLeft = ToInt32(aLeft);

int nRight = ToInt32(aRight);

if (nRight == 0)

{

return DivideByZero(nLeft, scriptContext);

}

//

// If the operands produce an integer, keep the result as an integer to improve performance:

// - This also bypasses conversion to / from doubles, which is expensive and potentially

// lossy.

//

#if INT32VAR

\_\_try

{

#endif

if ((nLeft % nRight) == 0)

{

//

// Check that result is not -0. !(Dividend is 0 and Divisor is negative)

//

if ((nLeft != 0) || (nRight > 0))

{

return JavascriptNumber::ToVar(nLeft/nRight, scriptContext);

}

}

#if INT32VAR

}

// 0x80000000 / -1 will trigger an integer overflow exception

\_\_except(GetExceptionCode() == STATUS\_INTEGER\_OVERFLOW)

{}

#endif

//

// Fallback to creating a floating-point number to preserve the fractional portion.

//

double dblResult = (double) nLeft / (double) nRight;

return JavascriptNumber::ToVarNoCheck(dblResult, scriptContext);

}

Var TaggedInt::Modulus(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

int nLeft = ToInt32(aLeft);

int nRight = ToInt32(aRight);

// nLeft is positive and nRight is +2^i

// Fast path for Power of 2 divisor

if (nLeft > 0 && ::Math::IsPow2(nRight))

{

return ToVarUnchecked(nLeft & (nRight - 1));

}

if (nRight == 0)

{

return scriptContext->GetLibrary()->GetNaN();

}

if (nLeft == 0)

{

return ToVarUnchecked(0);

}

int result;

#if INT32VAR

\_\_try

{

#endif

result = nLeft % nRight;

#if INT32VAR

}

// 0x80000000 / -1 will trigger an integer overflow exception

\_\_except(GetExceptionCode() == STATUS\_INTEGER\_OVERFLOW)

{

int64 left64 = nLeft;

int64 right64 = nRight;

result = (int)(left64 % right64);

}

#endif

if (result != 0)

{

return ToVarUnchecked(result);

}

else

{

//

// return -0 if left is negative

//

if (nLeft >= 0)

{

return ToVarUnchecked(0);

}

else

{

return scriptContext->GetLibrary()->GetNegativeZero();

}

}

}

Var TaggedInt::DivideInPlace(Var aLeft,Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result)

{

int nLeft = ToInt32(aLeft);

int nRight = ToInt32(aRight);

if (nRight == 0)

{

return DivideByZero(nLeft, scriptContext);

}

//

// If the operands produce an integer, keep the result as an integer to improve performance:

// - This also bypasses conversion to / from doubles, which is expensive and potentially

// lossy.

//

if ((nLeft % nRight) == 0)

{

//

// Check that result is not -0. !(Dividend is 0 and Divisor is negative)

//

if ((nLeft != 0) || (nRight > 0))

{

return JavascriptNumber::ToVar(nLeft/nRight, scriptContext);

}

}

//

// Fallback to creating a floating-point number to preserve the fractional portion.

//

double dblResult = (double) nLeft / (double) nRight;

return JavascriptNumber::InPlaceNew(dblResult, scriptContext, result);

}

Var TaggedInt::Multiply(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

//

// Perform the signed integer multiplication.

//

int nLeft = ToInt32(aLeft);

int nRight = ToInt32(aRight);

int nResult;

\_\_int64 int64Result = (\_\_int64)nLeft \* (\_\_int64)nRight;

nResult = (int)int64Result;

if (((int64Result >> 32) == 0 && (nResult > 0 || nResult == 0 && nLeft+nRight >= 0))

|| ((int64Result >> 32) == -1 && nResult < 0))

{

return JavascriptNumber::ToVar(nResult,scriptContext);

}

else if (int64Result == 0)

{

return JavascriptNumber::ToVarNoCheck((double)nLeft \* (double)nRight, scriptContext);

}

else

{

return JavascriptNumber::ToVarNoCheck((double)int64Result, scriptContext);

}

}

Var TaggedInt::MultiplyInPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber \*result)

{

//

// Perform the signed integer multiplication.

//

int nLeft = ToInt32(aLeft);

int nRight = ToInt32(aRight);

int nResult;

\_\_int64 int64Result = (\_\_int64)nLeft \* (\_\_int64)nRight;

nResult = (int)int64Result;

if (((int64Result >> 32) == 0 && nResult > 0)

|| (int64Result >> 32) == -1 && nResult < 0)

{

if (!TaggedInt::IsOverflow(nResult))

{

return TaggedInt::ToVarUnchecked(nResult);

}

else

{

return JavascriptNumber::InPlaceNew((double)nResult, scriptContext, result);

}

}

else if (int64Result == 0)

{

return JavascriptNumber::InPlaceNew((double)nLeft \* (double)nRight, scriptContext, result);

}

else

{

return JavascriptNumber::InPlaceNew((double)int64Result, scriptContext, result);

}

}

Var TaggedInt::Subtract(Var aLeft,Var aRight,ScriptContext\* scriptContext)

#ifdef DBG

{

Var difference = DbgSubtract(aLeft, aRight, scriptContext);

AssertMsg(JavascriptConversion::ToNumber(difference,scriptContext) == ToDouble(aLeft) - ToDouble(aRight), "TaggedInt fast subtraction is broken");

return difference;

}

Var TaggedInt::DbgSubtract(Var aLeft,Var aRight,ScriptContext\* scriptContext)

#endif

{

#if \_M\_IX86

//

// Perform the signed, integer subtraction directly on Atoms without converting to integers:

//

// T = AtomTag\_Int32

// nResult = A1 - A2

// Step 1: (N1 \* S + T) - (N2 \* S + T)

// Step 2: ((N1 - N2) \* S + T) - T

// Step 3: A3 - T

//

// NOTE: As demonstrated above, the FromVar() / ToVar() calls in (T) will cancel out,

// enabling an optimized operation.

//

\_\_asm

{

mov eax, aLeft

sub eax, aRight

jno LblDone // Check for overflow/underflow

// The carry flag indicates whether the sum has

// overflowed (>INT\_MAX) or underflowed (< INT\_MIN)

push scriptContext

cmc // For subtraction, CF=1 indicates an overflow, so reverse the flag

rcr eax, 1 // Convert to int32 and set the sign to the carry bit

push eax

call TaggedInt::OverflowHelper

dec eax // Adjust for the upcoming inc eax

LblDone:

inc eax

// Difference is in eax

}

#elif defined(\_M\_X64) || defined(\_M\_ARM32\_OR\_ARM64)

//

// Perform the signed, integer subtraction directly on Atoms using 64-bit math for overflow

// checking.

//

int64 nResult64 = ToInt64(aLeft) - ToInt64(aRight);

if (IsOverflow(nResult64))

{

//

// Promote result to double.

//

return JavascriptNumber::ToVarNoCheck((double) nResult64, scriptContext);

}

else

{

//

// Return A3

//

int nResult32 = (int) nResult64;

return ToVarUnchecked(nResult32);

}

#else

#error Unsupported processor type

#endif

}

// Without checking, bitwise 'and' the two values together. If the result passes the "Is" test

// then both arguments were valid Int31s and the result is correct.

Var TaggedInt::Speculative\_And(Var aLeft, Var aRight)

{

return (Var) (((size\_t) aLeft) & ((size\_t) aRight));

}

Var TaggedInt::And(Var aLeft, Var aRight)

{

//

// Perform the integer "bitwise and":

//

// A1 = N1' | T

// A2 = N2' | T

// A1 & A2 = N1' & N2' | T & T

// Result = N1' & N2' | T

//

Var aResult = Speculative\_And(aLeft, aRight);

AssertMsg(TaggedInt::Is(aResult), "Ensure result is properly marked");

return aResult;

}

Var TaggedInt::Or(Var aLeft, Var aRight)

{

//

// Perform the integer "bitwise or":

//

// A1 = N1' | T

// A2 = N2' | T

// A1 | A2 = N1' | N2' | T | T

// Result = N1' | N2' | T

//

Var aResult = (Var) (((size\_t) aLeft) | ((size\_t) aRight));

AssertMsg(TaggedInt::Is(aResult), "Ensure result is properly marked");

return aResult;

}

#if INT32VAR

Var TaggedInt::Xor(Var aLeft, Var aRight)

{

int32 nResult = ToInt32(aLeft) ^ ToInt32(aRight);

return TaggedInt::ToVarUnchecked(nResult);

}

#else

Var TaggedInt::Xor(Var aLeft, Var aRight)

{

//

// Perform the integer "bitwise xor":

//

// A1 = N1' | T

// A2 = N2' | T

// A1 ^ A2 = N1' ^ N2' | T ^ T

// Result - T = N1' ^ N2' | 0

//

size\_t nResult = ((size\_t) aLeft) ^ ((size\_t) aRight);

AssertMsg((nResult & AtomTag) == 0, "Ensure tag-bits cancelled out");

return (Var) (nResult | AtomTag\_IntPtr);

}

#endif

Var TaggedInt::ShiftLeft(Var aLeft,Var aRight,ScriptContext\* scriptContext)

{

//

// Shifting an integer left will always remain an integer, but it may overflow the Int31

// range. Therefore, we must call JavascriptNumber::ToVar() to check.

//

int nValue = ToInt32(aLeft);

uint32 nShift = ToUInt32(aRight);

return JavascriptNumber::ToVar(nValue << (nShift & 0x1F),scriptContext);

}

Var TaggedInt::ShiftRight(Var aLeft, Var aRight)

{

//

// If aLeft was a Int31 coming in, then the result must always be a Int31 going out because

// shifting right only makes value smaller. Therefore, we may call ToVarUnchecked()

// directly.

//

int nValue = ToInt32(aLeft);

uint32 nShift = ToUInt32(aRight);

return ToVarUnchecked(nValue >> (nShift & 0x1F));

}

Var TaggedInt::ShiftRightU(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

//

// If aLeft was a Int31 coming in, then the result must always be a Int31 going out because

// shifting right only makes value smaller. Therefore, we may call ToVarUnchecked()

// directly.

//

uint32 uValue = ToUInt32(aLeft);

uint32 nShift = ToUInt32(aRight);

return JavascriptNumber::ToVar(uValue >> (nShift & 0x1F), scriptContext);

}

void TaggedInt::ToBuffer(Var aValue, \_\_out\_ecount\_z(bufSize) wchar\_t \* buffer, uint bufSize)

{

return ToBuffer(ToInt32(aValue), buffer, bufSize);

}

void TaggedInt::ToBuffer(int value, \_\_out\_ecount\_z(bufSize) wchar\_t \* buffer, uint bufSize)

{

Assert(bufSize > 10);

\_itow\_s(value, buffer, bufSize, 10);

}

void TaggedInt::ToBuffer(uint value, \_\_out\_ecount\_z(bufSize) wchar\_t \* buffer, uint bufSize)

{

Assert(bufSize > 10);

\_ultow\_s(value, buffer, bufSize, 10);

}

JavascriptString\* TaggedInt::ToString(Var aValue,ScriptContext\* scriptContext)

{

return ToString(ToInt32(aValue), scriptContext);

}

JavascriptString\* TaggedInt::ToString(int value, ScriptContext\* scriptContext)

{

wchar\_t szBuffer[20];

ToBuffer(value, szBuffer, \_countof(szBuffer));

return JavascriptString::NewCopySz(szBuffer, scriptContext);

}

JavascriptString\* TaggedInt::ToString(uint value, ScriptContext\* scriptContext)

{

wchar\_t szBuffer[20];

ToBuffer(value, szBuffer, \_countof(szBuffer));

return JavascriptString::NewCopySz(szBuffer, scriptContext);

}

Var TaggedInt::NegateUnchecked(Var aValue)

{

AssertMsg( Is(aValue), "Ensure var is actually an 'TaggedInt'");

AssertMsg( aValue != ToVarUnchecked(0), "Do not use NegateUnchecked on zero because NegativeZero is special");

AssertMsg( aValue != ToVarUnchecked(k\_nMinValue), "Do not use NegateUnchecked on min value because it cannot be represented");

#if INT32VAR

int n = ToInt32(aValue);

Var result = ToVarUnchecked( 0 - n );

#else

int n = reinterpret\_cast<int>(aValue);

// Negation can be done by subtracting from "zero". The following method

// is just two operations: "load constant; sub"

// The constant 2 in the following expression

// a) adjusts for the bias of ToVarUnchecked(0) and

// b) ensures the tag bit is set

Var result = reinterpret\_cast<Var>( 2 - n );

#endif

// Check against the long way (shift, negate, shift, or)

AssertMsg( result == ToVarUnchecked( -ToInt32(aValue) ), "Logic error in NegateUnchecked" );

return result;

}

// Explicitly marking noinline and stdcall since this is called from inline asm

\_\_declspec(noinline) Var \_\_stdcall TaggedInt::IncrementOverflowHelper(ScriptContext\* scriptContext)

{

return JavascriptNumber::New( k\_nMaxValue + 1.0, scriptContext );

}

Var TaggedInt::Increment(Var aValue, ScriptContext\* scriptContext)

{

#if \_M\_IX86

\_\_asm

{

mov eax, aValue

add eax, 2

jno LblDone

push scriptContext

call TaggedInt::IncrementOverflowHelper

LblDone:

; result is in eax

}

#else

#if INT32VAR

Var result = aValue;

(\*(int \*)&result)++;

#else

int n = reinterpret\_cast<int>(aValue);

n += 2;

Var result = reinterpret\_cast<Var>(n);

#endif

// Wrap-around

if( result == MinVal() )

{

// Use New instead of ToVar for this constant

return IncrementOverflowHelper(scriptContext);

}

AssertMsg( result == ToVarUnchecked( ToInt32(aValue) + 1 ), "Logic error in Int31::Increment" );

return result;

#endif

}

// Explicitly marking noinline and stdcall since this is called from inline asm

\_\_declspec(noinline) Var \_\_stdcall TaggedInt::DecrementUnderflowHelper(ScriptContext\* scriptContext)

{

return JavascriptNumber::New( k\_nMinValue - 1.0, scriptContext );

}

Var TaggedInt::Decrement(Var aValue, ScriptContext\* scriptContext)

{

#if \_M\_IX86

\_\_asm

{

mov eax, aValue

sub eax, 2

jno LblDone

push scriptContext

call TaggedInt::DecrementUnderflowHelper

LblDone:

; result is in eax

}

#else

#if INT32VAR

Var result = aValue;

(\*(int \*)&result)--;

#else

int n = reinterpret\_cast<int>(aValue);

n -= 2;

Var result = reinterpret\_cast<Var>(n);

#endif

// Wrap-around

if( result == MaxVal() )

{

// Use New instead of ToVar for this constant

return DecrementUnderflowHelper(scriptContext);

}

AssertMsg( result == ToVarUnchecked( ToInt32(aValue) - 1 ), "Logic error in Int31::Decrement" );

return result;

#endif

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

class TaggedInt

{

public:

static Var Increment(Var aRight, ScriptContext\* scriptContext);

static Var Decrement(Var aRight, ScriptContext\* scriptContext);

static Var Negate(Var aRight, ScriptContext\* scriptContext);

static Var NegateUnchecked(Var aRight);

static Var Not(Var aRight, ScriptContext\* scriptContext);

static Var Add(Var aLeft, Var aRight, ScriptContext\* scriptContext);

static Var Divide(Var aLeft, Var aRight, ScriptContext\* scriptContext);

static Var DivideInPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var Modulus(Var aLeft, Var aRight, ScriptContext\* scriptContext);

static Var Multiply(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var MultiplyInPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var Subtract(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var And(Var aLeft, Var aRight);

static Var Or(Var aLeft, Var aRight);

static Var Xor(Var aLeft, Var aRight);

static Var ShiftLeft(Var aLeft, Var aShift, ScriptContext\* scriptContext);

static Var ShiftRight(Var aLeft, Var aShift);

static Var ShiftRightU(Var aLeft, Var aShift, ScriptContext\* scriptContext);

static Var Speculative\_And(Var aLeft, Var aRight);

static bool IsOverflow(int32 nValue);

static bool IsOverflow(uint32 nValue);

static bool IsOverflow(int64 nValue);

static bool IsOverflow(uint64 nValue);

static bool Is(Var aValue);

static bool IsPair(Var aLeft, Var aRight);

static double ToDouble(Var aValue);

static int32 ToInt32(Var aValue);

static uint32 ToUInt32(Var aValue);

static int64 ToInt64(Var aValue);

static uint16 ToUInt16(Var aValue);

static Var ToVarUnchecked(int nValue);

static void TaggedInt::ToBuffer(Var aValue, \_\_out\_ecount\_z(bufSize) wchar\_t \* buffer, uint bufSize);

static void TaggedInt::ToBuffer(int value, \_\_out\_ecount\_z(bufSize) wchar\_t \* buffer, uint bufSize);

static void TaggedInt::ToBuffer(uint value, \_\_out\_ecount\_z(bufSize) wchar\_t \* buffer, uint bufSize);

static JavascriptString\* ToString(Var aValue,ScriptContext\* scriptContext);

static JavascriptString\* ToString(int value,ScriptContext\* scriptContext);

static JavascriptString\* ToString(uint value,ScriptContext\* scriptContext);

static Var MinVal() { return ToVarUnchecked(k\_nMinValue); }

static Var MaxVal() { return ToVarUnchecked(k\_nMaxValue); }

private:

static Var DivideByZero(int nLeft, ScriptContext\* scriptContext);

static Var \_\_stdcall OverflowHelper(int overflowValue, ScriptContext\* scriptContext);

static Var \_\_stdcall IncrementOverflowHelper(ScriptContext\* scriptContext);

static Var \_\_stdcall DecrementUnderflowHelper(ScriptContext\* scriptContext);

#ifdef DBG

static Var DbgAdd(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var DbgSubtract(Var aLeft, Var aRight,ScriptContext\* scriptContext);

#endif

static const int k\_nMinValue = INT\_MIN / AtomTag\_Multiply;

static const int k\_nMaxValue = INT\_MAX / AtomTag\_Multiply;

};

// Helper representing operations and checks on TaggedInteger

// and JavascriptNumber ( aka TaggedFloat) - JavascriptNumber is a Tagged value

// only for 64-bit platforms.

class TaggedNumber

{

public:

static bool Is(Var aValue);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js

{

\_\_inline bool TaggedInt::IsOverflow(int32 nValue)

{

return (nValue < k\_nMinValue) || (nValue > k\_nMaxValue);

}

\_\_inline bool TaggedInt::IsOverflow(uint32 nValue)

{

return nValue > k\_nMaxValue;

}

\_\_inline bool TaggedInt::IsOverflow(int64 nValue)

{

return (nValue < k\_nMinValue) || (nValue > k\_nMaxValue);

}

\_\_inline bool TaggedInt::IsOverflow(uint64 nValue)

{

return nValue > k\_nMaxValue;

}

#if INT32VAR

\_\_inline bool TaggedInt::Is(Var aValue)

{

bool result = (((uintptr) aValue) >> VarTag\_Shift) == AtomTag;

if(result)

{

Assert((uintptr)aValue >> 32 == (AtomTag << 16));

}

return result;

}

\_\_inline bool TaggedInt::IsPair(Var aLeft, Var aRight)

{

uint32 tags = (uint32)((uint64)aLeft >> 32 | (uint64)aRight >> 48);

bool result = (tags == AtomTag\_Pair);

AssertMsg(result == (TaggedInt::Is(aLeft) && TaggedInt::Is(aRight)), "TaggedInt::IsPair() logic is incorrect");

return result;

}

\_\_inline int32 TaggedInt::ToInt32(Var aValue)

{

AssertMsg(Is(aValue), "Ensure var is actually an 'TaggedInt'");

return ::Math::PointerCastToIntegralTruncate<int32>(aValue);

}

\_\_inline uint32 TaggedInt::ToUInt32(Var aValue)

{

AssertMsg(Is(aValue), "Ensure var is actually an 'TaggedInt'");

return ::Math::PointerCastToIntegralTruncate<uint32>(aValue);

}

\_\_inline int64 TaggedInt::ToInt64(Var aValue)

{

AssertMsg(Is(aValue), "Ensure var is actually an 'TaggedInt'");

int64 nValue = (int64)(::Math::PointerCastToIntegralTruncate<int32>(aValue));

AssertMsg(nValue == (int64) ToInt32(aValue),

"Ensure 32-bit and 64-bit operations return same result");

return nValue;

}

\_\_inline uint16 TaggedInt::ToUInt16(Var aValue)

{

AssertMsg(Is(aValue), "Ensure var is actually an 'TaggedInt'");

return ::Math::PointerCastToIntegralTruncate<uint16>(aValue);

}

\_\_inline double TaggedInt::ToDouble(Var aValue)

{

return (double)::Math::PointerCastToIntegralTruncate<int32>(aValue);

}

\_\_inline Var TaggedInt::ToVarUnchecked(int nValue)

{

//

// To convert to an var we first cast to uint32 to lose the signedness and then

// extend it to a 64-bit uintptr before OR'ing the 64-bit atom tag.

//

AssertMsg(!IsOverflow(nValue), "Ensure no information loss from conversion");

return reinterpret\_cast<Var>(((uintptr)(uint32)nValue) | AtomTag\_IntPtr);

}

#else

\_\_inline bool TaggedInt::Is(const Var aValue)

{

return (((uintptr) aValue) & AtomTag) == AtomTag\_IntPtr;

}

\_\_inline bool TaggedInt::IsPair(Var aLeft, Var aRight)

{

//

// Check if both Atoms are tagged as "SmInts":

// - Because we're checking tag bits, we don't need to check against 'null', since it won't

// be tagged as an TaggedInt.

// - This degenerates into bitwise arithmetic that avoids the overhead of branching and

// short-circuit evaluation.

//

return (((uintptr) aLeft) & ((uintptr) aRight) & AtomTag) == AtomTag\_IntPtr;

}

\_\_inline int32 TaggedInt::ToInt32(Var aValue)

{

//

// To convert from an var, must first convert to an 'int32' to properly sign-extend

// negative values. Then, use shift operation to remove the tag bits.

//

AssertMsg(Is(aValue), "Ensure var is actually an 'TaggedInt'");

return ((int) aValue) >> VarTag\_Shift;

}

\_\_inline uint32 TaggedInt::ToUInt32(Var aValue)

{

//

// To convert from an var, must use ToInt32() to properly sign-extend negative values, then

// convert to an (unsigned) uint32.

//

return (uint32) ToInt32(aValue);

}

\_\_inline int64 TaggedInt::ToInt64(Var aValue)

{

//

// To convert from an var, must first convert to an 'int64' to properly sign-extend

// negative values. Then, use shift operation to remove the tag bits.

//

AssertMsg(Is(aValue), "Ensure var is actually an 'TaggedInt'");

int64 nValue = ((int32) aValue) >> VarTag\_Shift;

AssertMsg(nValue == (int64) ToInt32(aValue),

"Ensure 32-bit and 64-bit operations return same result");

return nValue;

}

\_\_inline uint16 TaggedInt::ToUInt16(Var aValue)

{

AssertMsg(Is(aValue), "Ensure var is actually an 'TaggedInt'");

return (uint16)(((int) aValue) >> VarTag\_Shift);

}

\_\_inline double TaggedInt::ToDouble(Var aValue)

{

return (double) ToInt32(aValue);

}

\_\_inline Var TaggedInt::ToVarUnchecked(int nValue)

{

//

// To convert to an var, must first multiply (which will be converted into a shift

// operation) to create space for the tag while properly preserving negative values. Then,

// add the tag.

//

AssertMsg(!IsOverflow(nValue), "Ensure no information loss from conversion");

return reinterpret\_cast<Var>((nValue << VarTag\_Shift) | AtomTag\_IntPtr);

}

#endif

\_\_inline Var TaggedInt::Add(Var aLeft,Var aRight,ScriptContext\* scriptContext)

#ifdef DBG

{

Var sum = DbgAdd(aLeft, aRight, scriptContext);

AssertMsg(JavascriptConversion::ToNumber(sum,scriptContext) == ToDouble(aLeft) + ToDouble(aRight), "TaggedInt fast addition is broken");

return sum;

}

\_\_inline Var TaggedInt::DbgAdd(Var aLeft,Var aRight,ScriptContext\* scriptContext)

#endif

{

#if \_M\_IX86

//

// Perform the signed, integer addition directly on Atoms without converting to integers:

//

// T = AtomTag\_Int32

// nResult = A1 + A2

// Step 1: (N1 \* S + T) + (N2 \* S + T)

// Step 2: ((N1 + N2) \* S + T) + T

// Step 3: A3 + T

//

// NOTE: As demonstrated above, the FromVar() / ToVar() calls in (T) will cancel out,

// enabling an optimized operation.

//

\_\_asm

{

mov eax, aLeft

dec eax // Get rid of one of the tags, since they'll add up

add eax, aRight // Perform the addition

jno LblDone // Check for overflow/underflow

// The carry flag indicates whether the sum has

// overflowed (>INT\_MAX) or underflowed (< INT\_MIN)

push scriptContext

rcr eax, 1 // Convert to int32 and set the sign to the carry bit

push eax

call TaggedInt::OverflowHelper

LblDone:

// Sum is in eax

}

#elif defined(\_M\_X64) || defined(\_M\_ARM32\_OR\_ARM64)

//

// Perform the signed, integer addition directly on Atoms using 64-bit math for overflow

// checking.

//

int64 nResult64 = ToInt64(aLeft) + ToInt64(aRight);

if (IsOverflow(nResult64))

{

//

// Promote result to double.

//

return JavascriptNumber::ToVarNoCheck((double) nResult64, scriptContext);

}

else

{

//

// Return A3

//

int nResult32 = (int) nResult64;

return ToVarUnchecked(nResult32);

}

#else

#error Unsupported processor type

#endif

}

//

// True if the value is a tagged number representation (for x64 - float & integers) - otherwise false.

//

\_\_inline bool TaggedNumber::Is(const Var aValue)

{

bool isTaggedNumber;

#if FLOATVAR

// If we add another tagged representation that is not numerical - this will not work.

isTaggedNumber = !RecyclableObject::Is(aValue);

#else

isTaggedNumber = TaggedInt::Is(aValue);

#endif

return isTaggedNumber;

}

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#define BASE\_VALUE\_TYPE(t, b) const ValueType ValueType::##t(b);

#include "ValueTypes.h"

#undef BASE\_VALUE\_TYPE

const ValueType ValueType::AnyNumber(

Bits::Int | Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged | Bits::Float | Bits::Number);

void ValueType::Initialize()

{

InitializeTypeIdToBitsMap();

#if DBG

RunUnitTests();

#endif

}

\_\_inline ValueType::Bits ValueType::BitPattern(const TSize onCount)

{

CompileAssert(sizeof(TSize) <= sizeof(size\_t));

Assert(onCount && onCount <= sizeof(TSize) \* 8);

#pragma prefast(suppress:6235, "Non-Zero Constant TSize and size\_t in Condition. This is By Design to allow for TSize to be increased in size to uint32 without breaking anything")

return

static\_cast<Bits>(

sizeof(TSize) < sizeof(size\_t) || onCount < sizeof(TSize) \* 8

? (static\_cast<size\_t>(1) << onCount) - 1

: static\_cast<size\_t>(-1));

}

\_\_inline ValueType::Bits ValueType::BitPattern(const TSize onCount, const TSize offCount)

{

Assert(onCount && onCount <= sizeof(TSize) \* 8);

Assert(offCount && offCount <= sizeof(TSize) \* 8);

return BitPattern(onCount + offCount) - BitPattern(offCount);

}

ValueType ValueType::GetTaggedInt()

{

return Verify(Int);

}

ValueType ValueType::GetInt(const bool isLikelyTagged)

{

Bits intBits = Bits::Int | Bits::IntCanBeUntagged | Bits::CanBeTaggedValue;

if(!isLikelyTagged)

intBits |= Bits::IntIsLikelyUntagged;

return Verify(intBits);

}

ValueType ValueType::GetNumberAndLikelyInt(const bool isLikelyTagged)

{

return Verify(GetInt(isLikelyTagged).bits | Bits::Number);

}

\_\_inline ValueType ValueType::GetObject(const ObjectType objectType)

{

ValueType valueType(UninitializedObject);

valueType.SetObjectType(objectType);

if(objectType == ObjectType::Array || objectType == ObjectType::ObjectWithArray)

{

// Default to the most conservative array-specific information. This just a safeguard to guarantee that the returned

// value type has a valid set of array-specific information. Callers should not rely on these defaults, and should

// instead always set each piece of information explicitly.

valueType = valueType.SetHasNoMissingValues(false).SetArrayTypeId(Js::TypeIds\_Array);

}

return Verify(valueType);

}

ValueType ValueType::GetSimd128(const ObjectType objectType)

{

Assert(objectType >= ObjectType::Simd128Float32x4 && objectType <= ObjectType::Simd128Float64x2);

return GetObject(objectType);

}

\_\_inline ValueType ValueType::GetArray(const ObjectType objectType)

{

// Should typically use GetObject instead. This function should only be used for performance, when the array info is

// guaranteed to be updated correctly by the caller.

Assert(objectType == ObjectType::Array || objectType == ObjectType::ObjectWithArray);

ValueType valueType(UninitializedObject);

valueType.SetObjectType(objectType);

return Verify(valueType);

}

ValueType::ValueType() : bits(Uninitialized.bits)

{

CompileAssert(sizeof(ValueType) == sizeof(TSize));

CompileAssert(sizeof(ObjectType) == sizeof(TSize));

}

ValueType::ValueType(const Bits bits) : bits(bits)

{

}

ValueType ValueType::Verify(const Bits bits)

{

return Verify(ValueType(bits));

}

ValueType ValueType::Verify(const ValueType valueType)

{

Assert(valueType.bits);

Assert(!valueType.OneOn(Bits::Object) || valueType.GetObjectType() < ObjectType::Count);

Assert(

valueType.OneOn(Bits::Object) ||

valueType.OneOn(Bits::Int) ||

valueType.AnyOnExcept(Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged));

Assert(

valueType.OneOn(Bits::Object) ||

!valueType.AllEqual(Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged, Bits::IntIsLikelyUntagged));

return valueType;

}

bool ValueType::OneOn(const Bits b) const

{

return AnyOn(b);

}

bool ValueType::AnyOn(const Bits b) const

{

Assert(b);

return !!(bits & b);

}

bool ValueType::AllEqual(const Bits b, const Bits e) const

{

Assert(b);

return (bits & b) == e;

}

bool ValueType::AllOn(const Bits b) const

{

return AllEqual(b, b);

}

bool ValueType::OneOnOneOff(const Bits on, const Bits off) const

{

return AllOnAllOff(on, off);

}

bool ValueType::AllOnAllOff(const Bits on, const Bits off) const

{

return AllEqual(on | off, on);

}

bool ValueType::OneOnOthersOff(const Bits b) const

{

return AllOnOthersOff(b);

}

bool ValueType::OneOnOthersOff(const Bits b, const Bits ignore) const

{

return AllOnOthersOff(b, ignore);

}

bool ValueType::AnyOnOthersOff(const Bits b) const

{

Assert(b);

return !(bits & ~b);

}

bool ValueType::AnyOnOthersOff(const Bits b, const Bits ignore) const

{

Assert(b);

Assert(ignore);

Assert(!(b & ignore)); // not necessary for this function to work correctly, but generally not expected

return AnyOn(b) && AnyOnOthersOff(b | ignore);

}

bool ValueType::AllOnOthersOff(const Bits b) const

{

Assert(b);

return bits == b;

}

bool ValueType::AllOnOthersOff(const Bits b, const Bits ignore) const

{

Assert(b);

Assert(ignore);

Assert(!(b & ignore));

return AllEqual(~ignore, b);

}

bool ValueType::AnyOnExcept(const Bits b) const

{

Assert(!!b && !!~b);

return !AnyOn(b);

}

bool ValueType::IsUninitialized() const

{

return AllOnOthersOff(Bits::Likely, Bits::CanBeTaggedValue);

}

bool ValueType::IsDefinite() const

{

return !OneOn(Bits::Likely);

}

bool ValueType::IsTaggedInt() const

{

return AllOnOthersOff(Bits::Int | Bits::CanBeTaggedValue);

}

bool ValueType::IsIntAndLikelyTagged() const

{

return AllOnOthersOff(Bits::Int | Bits::CanBeTaggedValue, Bits::IntCanBeUntagged);

}

bool ValueType::IsLikelyTaggedInt() const

{

return AllOnOthersOff(Bits::Int | Bits::CanBeTaggedValue, Bits::Likely | Bits::IntCanBeUntagged | Bits::Number);

}

bool ValueType::HasBeenUntaggedInt() const

{

return OneOnOneOff(Bits::IntIsLikelyUntagged, Bits::Object);

}

bool ValueType::IsIntAndLikelyUntagged() const

{

return AllOnOthersOff(Bits::Int | Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged, Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelyUntaggedInt() const

{

return AllOnOthersOff(Bits::Int | Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged,

Bits::Likely | Bits::Number | Bits::CanBeTaggedValue);

}

bool ValueType::IsNotTaggedValue() const

{

return IsNotNumber() || !OneOn(Bits::CanBeTaggedValue);

}

bool ValueType::CanBeTaggedValue() const

{

return !IsNotTaggedValue();

}

ValueType ValueType::SetCanBeTaggedValue(const bool b) const

{

if (b)

{

Assert(!IsNotNumber());

return Verify(bits | Bits::CanBeTaggedValue);

}

return Verify(bits & ~Bits::CanBeTaggedValue);

}

bool ValueType::HasBeenInt() const

{

return OneOnOneOff(Bits::Int, Bits::Object);

}

bool ValueType::IsInt() const

{

return OneOnOthersOff(Bits::Int, Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged | Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelyInt() const

{

return OneOnOthersOff(

Bits::Int,

Bits::Likely | Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged | Bits::Number | Bits::CanBeTaggedValue);

}

bool ValueType::IsNotInt() const

{

return

AnyOnExcept(Bits::Likely | Bits::Object | Bits::Int | Bits::CanBeTaggedValue | Bits::Float | Bits::Number) ||

OneOnOneOff(Bits::Object, Bits::Likely);

}

bool ValueType::IsNotNumber() const

{

// These are the same for now.

return IsNotInt();

}

bool ValueType::HasBeenFloat() const

{

return OneOnOneOff(Bits::Float, Bits::Object);

}

bool ValueType::IsFloat() const

{

// TODO: Require that the int bits are off. We can then use (!IsFloat() && IsNumber()) to determine that a tagged int check

// needs to be done but not a JavascriptNumber/TaggedFloat check.

return

OneOnOthersOff(

Bits::Float,

(

Bits::Int |

Bits::IntCanBeUntagged |

Bits::IntIsLikelyUntagged |

Bits::CanBeTaggedValue |

Bits::Number

));

}

bool ValueType::IsLikelyFloat() const

{

return

OneOnOthersOff(

Bits::Float,

(

Bits::Likely |

Bits::Undefined |

Bits::Int |

Bits::IntCanBeUntagged |

Bits::IntIsLikelyUntagged |

Bits::CanBeTaggedValue |

Bits::Number

));

}

bool ValueType::HasBeenNumber() const

{

return !OneOn(Bits::Object) && AnyOn(Bits::Int | Bits::Float | Bits::Number);

}

bool ValueType::IsNumber() const

{

return AnyOnOthersOff(Bits::Int | Bits::Float | Bits::Number,

Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged | Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelyNumber() const

{

return

AnyOnOthersOff(

Bits::Int | Bits::Float | Bits::Number,

Bits::Likely | Bits::Undefined | Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged | Bits::CanBeTaggedValue);

}

bool ValueType::HasBeenUnknownNumber() const

{

return OneOnOneOff(Bits::Number, Bits::Object);

}

bool ValueType::IsUnknownNumber() const

{

// Equivalent to IsNumber() && !IsLikelyInt() && !IsLikelyFloat()

return OneOnOthersOff(Bits::Number, Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelyUnknownNumber() const

{

// If true, equivalent to IsLikelyNumber() && !IsLikelyInt() && !IsLikelyFloat()

return OneOnOthersOff(Bits::Number, Bits::Likely | Bits::Undefined | Bits::CanBeTaggedValue);

}

bool ValueType::HasBeenUndefined() const

{

return OneOn(Bits::Undefined);

}

bool ValueType::IsUndefined() const

{

return OneOnOthersOff(Bits::Undefined, Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelyUndefined() const

{

return OneOnOthersOff(Bits::Undefined, Bits::Likely | Bits::CanBeTaggedValue);

}

bool ValueType::HasBeenNull() const

{

return OneOn(Bits::Null);

}

bool ValueType::IsNull() const

{

return OneOnOthersOff(Bits::Null, Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelyNull() const

{

return OneOnOthersOff(Bits::Null, Bits::Likely | Bits::CanBeTaggedValue);

}

bool ValueType::HasBeenBoolean() const

{

return OneOnOneOff(Bits::Boolean, Bits::Object);

}

bool ValueType::IsBoolean() const

{

return OneOnOthersOff(Bits::Boolean, Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelyBoolean() const

{

return OneOnOthersOff(Bits::Boolean, Bits::Likely | Bits::CanBeTaggedValue);

}

bool ValueType::HasBeenString() const

{

return OneOnOneOff(Bits::String, Bits::Object);

}

bool ValueType::IsString() const

{

return OneOnOthersOff(Bits::String, Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelyString() const

{

return OneOnOthersOff(Bits::String, Bits::Likely | Bits::CanBeTaggedValue);

}

bool ValueType::IsNotString() const

{

return AnyOnExcept(Bits::Likely | Bits::Object | Bits::String | Bits::CanBeTaggedValue)

|| OneOnOneOff(Bits::Object, Bits::Likely);

}

bool ValueType::HasBeenSymbol() const

{

return OneOnOneOff(Bits::Symbol, Bits::Object);

}

bool ValueType::IsSymbol() const

{

return OneOnOthersOff(Bits::Symbol, Bits::CanBeTaggedValue);

}

bool ValueType::IsLikelySymbol() const

{

return OneOnOthersOff(Bits::Symbol, Bits::Likely | Bits::CanBeTaggedValue);

}

bool ValueType::IsNotSymbol() const

{

return AnyOnExcept(Bits::Likely | Bits::Object | Bits::Symbol | Bits::CanBeTaggedValue)

|| OneOnOneOff(Bits::Object, Bits::Likely);

}

bool ValueType::HasBeenPrimitive() const

{

return

OneOn(Bits::Object)

?

AnyOn(Bits::Undefined | Bits::Null)

|| GetObjectType() >= ObjectType::Simd128Float32x4

:

AnyOn(

Bits::Undefined |

Bits::Null |

Bits::Int |

Bits::Float |

Bits::Number |

Bits::Boolean |

Bits::String |

Bits::Symbol |

Bits::PrimitiveOrObject);

}

bool ValueType::IsPrimitive() const

{

bool result =

AnyOnOthersOff(

Bits::Undefined | Bits::Null | Bits::Int | Bits::Float | Bits::Number | Bits::Boolean | Bits::String | Bits::Symbol,

Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged | Bits::CanBeTaggedValue);

#if ENABLE\_NATIVE\_CODEGEN

result = result || IsSimd128();

#endif

return result;

}

bool ValueType::IsLikelyPrimitive() const

{

bool result =

AnyOnOthersOff(

Bits::Undefined | Bits::Null | Bits::Int | Bits::Float | Bits::Number | Bits::Boolean | Bits::String | Bits::Symbol,

Bits::Likely | Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged | Bits::CanBeTaggedValue);

#if ENABLE\_NATIVE\_CODEGEN

result = result || IsLikelySimd128();

#endif

return result;

}

bool ValueType::HasBeenObject() const

{

return AnyOn(Bits::Object | Bits::PrimitiveOrObject);

}

bool ValueType::IsObject() const

{

return AllOnAllOff(Bits::Object, Bits::Likely | Bits::Undefined | Bits::Null);

}

bool ValueType::IsLikelyObject() const

{

// For syms that are typically used as objects, they often also have Undefined or Null values, and they are used as objects

// only after checking for Undefined or Null. So, for the purpose of determining whether a value type is likely object, the

// Undefined and Null bits are ignored.

return OneOn(Bits::Object);

}

bool ValueType::IsNotObject() const

{

return AnyOnExcept(Bits::Likely | Bits::Object | Bits::PrimitiveOrObject);

}

bool ValueType::CanMergeToObject() const

{

Assert(!IsLikelyObject());

return AnyOnExcept(BitPattern(VALUE\_TYPE\_NONOBJECT\_BIT\_COUNT, VALUE\_TYPE\_COMMON\_BIT\_COUNT));

}

bool ValueType::CanMergeToSpecificObjectType() const

{

return IsLikelyObject() ? GetObjectType() == ObjectType::UninitializedObject : CanMergeToObject();

}

bool ValueType::IsRegExp() const

{

return IsObject() && GetObjectType() == ObjectType::RegExp;

}

bool ValueType::IsLikelyRegExp() const

{

return IsLikelyObject() && GetObjectType() == ObjectType::RegExp;

}

bool ValueType::IsArray() const

{

return IsObject() && GetObjectType() == ObjectType::Array;

}

bool ValueType::IsLikelyArray() const

{

return IsLikelyObject() && GetObjectType() == ObjectType::Array;

}

bool ValueType::IsNotArray() const

{

return IsNotObject() || IsObject() && GetObjectType() > ObjectType::Object && GetObjectType() != ObjectType::Array;

}

bool ValueType::IsArrayOrObjectWithArray() const

{

return IsObject() && (GetObjectType() == ObjectType::ObjectWithArray || GetObjectType() == ObjectType::Array);

}

bool ValueType::IsLikelyArrayOrObjectWithArray() const

{

return IsLikelyObject() && (GetObjectType() == ObjectType::ObjectWithArray || GetObjectType() == ObjectType::Array);

}

bool ValueType::IsNotArrayOrObjectWithArray() const

{

return

IsNotObject() ||

IsObject() && GetObjectType() != ObjectType::ObjectWithArray && GetObjectType() != ObjectType::Array;

}

bool ValueType::IsNativeArray() const

{

return IsArrayOrObjectWithArray() && !HasVarElements();

}

bool ValueType::IsLikelyNativeArray() const

{

return IsLikelyArrayOrObjectWithArray() && !HasVarElements();

}

bool ValueType::IsNotNativeArray() const

{

return

IsNotObject() ||

(

IsObject() &&

(GetObjectType() != ObjectType::ObjectWithArray && GetObjectType() != ObjectType::Array || HasVarElements())

);

}

bool ValueType::IsNativeIntArray() const

{

return IsArrayOrObjectWithArray() && HasIntElements();

}

bool ValueType::IsLikelyNativeIntArray() const

{

return IsLikelyArrayOrObjectWithArray() && HasIntElements();

}

bool ValueType::IsNativeFloatArray() const

{

return IsArrayOrObjectWithArray() && HasFloatElements();

}

bool ValueType::IsLikelyNativeFloatArray() const

{

return IsLikelyArrayOrObjectWithArray() && HasFloatElements();

}

bool ValueType::IsTypedIntArray() const

{

return IsObject() && GetObjectType() >= ObjectType::Int8Array && GetObjectType() <= ObjectType::Uint32Array;

}

bool ValueType::IsLikelyTypedIntArray() const

{

return IsLikelyObject() && GetObjectType() >= ObjectType::Int8Array && GetObjectType() <= ObjectType::Uint32Array;

}

bool ValueType::IsTypedArray() const

{

return IsObject() && GetObjectType() >= ObjectType::Int8Array && GetObjectType() <= ObjectType::CharArray;

}

bool ValueType::IsLikelyTypedArray() const

{

return IsLikelyObject() && GetObjectType() >= ObjectType::Int8Array && GetObjectType() <= ObjectType::CharArray;

}

bool ValueType::IsTypedIntOrFloatArray() const

{

return IsObject() && ((GetObjectType() >= ObjectType::Int8Array && GetObjectType() <= ObjectType::Float64Array));

}

bool ValueType::IsOptimizedTypedArray() const

{

return IsObject() && ((GetObjectType() >= ObjectType::Int8Array && GetObjectType() <= ObjectType::Float64MixedArray));

}

bool ValueType::IsLikelyOptimizedTypedArray() const

{

return IsLikelyObject() && ((GetObjectType() >= ObjectType::Int8Array && GetObjectType() <= ObjectType::Float64MixedArray));

}

bool ValueType::IsLikelyOptimizedVirtualTypedArray() const

{

return IsLikelyObject() && (GetObjectType() >= ObjectType::Int8VirtualArray && GetObjectType() <= ObjectType::Float64VirtualArray);

}

bool ValueType::IsAnyArrayWithNativeFloatValues() const

{

if(!IsObject())

return false;

switch(GetObjectType())

{

case ObjectType::ObjectWithArray:

case ObjectType::Array:

return HasFloatElements();

case ObjectType::Float32Array:

case ObjectType::Float32VirtualArray:

case ObjectType::Float32MixedArray:

case ObjectType::Float64Array:

case ObjectType::Float64VirtualArray:

case ObjectType::Float64MixedArray:

return true;

}

return false;

}

bool ValueType::IsLikelyAnyArrayWithNativeFloatValues() const

{

if(!IsLikelyObject())

return false;

switch(GetObjectType())

{

case ObjectType::ObjectWithArray:

case ObjectType::Array:

return HasFloatElements();

case ObjectType::Float32Array:

case ObjectType::Float32VirtualArray:

case ObjectType::Float32MixedArray:

case ObjectType::Float64Array:

case ObjectType::Float64VirtualArray:

case ObjectType::Float64MixedArray:

return true;

}

return false;

}

bool ValueType::IsAnyArray() const

{

return IsObject() && GetObjectType() >= ObjectType::ObjectWithArray && GetObjectType() <= ObjectType::CharArray;

}

bool ValueType::IsLikelyAnyArray() const

{

return IsLikelyObject() && GetObjectType() >= ObjectType::ObjectWithArray && GetObjectType() <= ObjectType::CharArray;

}

bool ValueType::IsAnyOptimizedArray() const

{

return IsObject() && ((GetObjectType() >= ObjectType::ObjectWithArray && GetObjectType() <= ObjectType::Float64MixedArray));

}

bool ValueType::IsLikelyAnyOptimizedArray() const

{

return IsLikelyObject() && ((GetObjectType() >= ObjectType::ObjectWithArray && GetObjectType() <= ObjectType::Float64MixedArray));

}

bool ValueType::IsLikelyAnyUnOptimizedArray() const

{

return IsLikelyObject() && GetObjectType() >= ObjectType::Int64Array && GetObjectType() <= ObjectType::CharArray;

}

#if ENABLE\_NATIVE\_CODEGEN

// Simd128 values

// Note that SIMD types are primitives

bool ValueType::IsSimd128() const

{

return IsObject() && (GetObjectType() >= ObjectType::Simd128Float32x4 && GetObjectType() <= ObjectType::Simd128Float64x2);

}

bool ValueType::IsSimd128(IRType type) const

{

switch (type)

{

case TySimd128F4:

return IsSimd128Float32x4();

case TySimd128I4:

return IsSimd128Int32x4();

case TySimd128D2:

return IsSimd128Float64x2();

default:

Assert(UNREACHED);

return false;

}

}

bool ValueType::IsSimd128Float32x4() const

{

return IsObject() && GetObjectType() == ObjectType::Simd128Float32x4;

}

bool ValueType::IsSimd128Int32x4() const

{

return IsObject() && GetObjectType() == ObjectType::Simd128Int32x4;

}

bool ValueType::IsSimd128Int8x16() const

{

return IsObject() && GetObjectType() == ObjectType::Simd128Int8x16;

}

bool ValueType::IsSimd128Float64x2() const

{

return IsObject() && GetObjectType() == ObjectType::Simd128Float64x2;

}

bool ValueType::IsLikelySimd128() const

{

return IsLikelyObject() && (GetObjectType() >= ObjectType::Simd128Float32x4 && GetObjectType() <= ObjectType::Simd128Float64x2);

}

bool ValueType::IsLikelySimd128Float32x4() const

{

return IsLikelyObject() && GetObjectType() == ObjectType::Simd128Float32x4;

}

bool ValueType::IsLikelySimd128Int32x4() const

{

return IsLikelyObject() && GetObjectType() == ObjectType::Simd128Int32x4;

}

bool ValueType::IsLikelySimd128Int8x16() const

{

return IsLikelyObject() && GetObjectType() == ObjectType::Simd128Int8x16;

}

bool ValueType::IsLikelySimd128Float64x2() const

{

return IsLikelyObject() && GetObjectType() == ObjectType::Simd128Float64x2;

}

#endif

ObjectType ValueType::GetObjectType() const

{

Assert(OneOn(Bits::Object));

return \_objectType;

}

void ValueType::SetObjectType(const ObjectType objectType)

{

Assert(OneOn(Bits::Object));

Assert(objectType < ObjectType::Count);

\_objectType = objectType;

}

ValueType ValueType::SetIsNotAnyOf(const ValueType other) const

{

Assert(other.IsDefinite());

Assert(!other.HasBeenObject());

return bits & ~other.bits;

}

bool ValueType::HasNoMissingValues() const

{

Assert(IsLikelyArrayOrObjectWithArray());

return OneOn(Bits::NoMissingValues);

}

ValueType ValueType::SetHasNoMissingValues(const bool noMissingValues) const

{

Assert(IsLikelyArrayOrObjectWithArray());

if(noMissingValues)

return Verify(bits | Bits::NoMissingValues);

return Verify(bits & ~Bits::NoMissingValues);

}

bool ValueType::HasNonInts() const

{

Assert(IsLikelyArrayOrObjectWithArray());

return OneOn(Bits::NonInts);

}

bool ValueType::HasNonFloats() const

{

Assert(IsLikelyArrayOrObjectWithArray());

return OneOn(Bits::NonFloats);

}

bool ValueType::HasIntElements() const

{

Assert(IsLikelyArrayOrObjectWithArray());

return !OneOn(Bits::NonInts);

}

bool ValueType::HasFloatElements() const

{

Assert(IsLikelyArrayOrObjectWithArray());

return OneOnOneOff(Bits::NonInts, Bits::NonFloats);

}

bool ValueType::HasVarElements() const

{

Assert(IsLikelyArrayOrObjectWithArray());

return AllOn(Bits::NonInts | Bits::NonFloats);

}

ValueType ValueType::SetArrayTypeId(const Js::TypeId typeId) const

{

using namespace Js;

Assert(IsLikelyArrayOrObjectWithArray());

Assert(JavascriptArray::Is(typeId));

Assert(typeId == TypeIds\_Array || IsLikelyObject() && GetObjectType() == ObjectType::Array); // objects with native arrays are currently not supported

Bits newBits = bits & ~(Bits::NonInts | Bits::NonFloats);

switch(typeId)

{

case TypeIds\_Array:

newBits |= Bits::NonFloats;

// fall through

case TypeIds\_NativeFloatArray:

newBits |= Bits::NonInts;

break;

}

return Verify(newBits);

}

bool ValueType::IsSubsetOf(

const ValueType other,

const bool isAggressiveIntTypeSpecEnabled,

const bool isFloatSpecEnabled,

const bool isArrayMissingValueCheckHoistEnabled,

const bool isNativeArrayEnabled) const

{

if(IsUninitialized())

return other.IsUninitialized();

if(other.IsUninitialized())

return true;

if(IsLikelyNumber() && other.IsLikelyNumber())

{

// Special case for numbers since there are multiple combinations of bits and a bit-subset produces incorrect results in

// some cases

// When type specialization is enabled, a value type that is likely int or float is considered to be more specific than

// a value type that is definitely number but unknown as to whether it was int or float, because the former can

// participate in type specialization while the latter cannot. When type specialization is disabled, the definite value

// type is considered to be a subset of the indefinite value type because neither will participate in type

// specialization.

if(other.IsUnknownNumber() &&

(isAggressiveIntTypeSpecEnabled && IsLikelyInt() || isFloatSpecEnabled && IsLikelyFloat()))

{

return true;

}

// The following number types are listed in order of most specific type to least specific type. Types are considered to

// be subsets of the less specific types below it, with the exception that int types are not considered to be subsets of

// float types.

// TaggedInt

// IntAndLikelyTagged

// Int

// Float

// Number

// This logic doesn't play well with Bits::Undefined, so remove it before proceeding.

ValueType \_this = this->bits & ~Bits::Undefined;

ValueType \_other = other.bits & ~Bits::Undefined;

return

(!\_this.OneOn(Bits::Likely) || \_other.OneOn(Bits::Likely)) &&

(

(

\_this.IsTaggedInt() ||

\_this.IsLikelyTaggedInt() && !\_other.IsTaggedInt() ||

\_this.IsLikelyInt() && !\_other.IsLikelyTaggedInt()

) && !\_other.IsLikelyFloat() ||

\_this.IsLikelyFloat() && !\_other.IsLikelyInt() ||

\_other.IsLikelyUnknownNumber()

);

}

const Bits commonBits = bits & (BitPattern(VALUE\_TYPE\_COMMON\_BIT\_COUNT) - Bits::Object);

if(!!commonBits && !other.AllOn(commonBits))

return false;

if(OneOn(Bits::Object))

{

if(!other.OneOn(Bits::Object))

return other.OneOn(Bits::PrimitiveOrObject) || !other.IsDefinite() && other.CanMergeToObject();

}

else

{

if(!other.OneOn(Bits::Object))

return other.AllOn(bits);

return CanMergeToObject();

}

if(other.GetObjectType() == ObjectType::UninitializedObject && GetObjectType() != ObjectType::UninitializedObject)

return true; // object types other than UninitializedObject are a subset of UninitializedObject regardless of the Likely bit

if(GetObjectType() != other.GetObjectType())

return false;

if(!OneOn(Bits::Likely) && other.OneOn(Bits::Likely) ||

other.GetObjectType() != ObjectType::ObjectWithArray && other.GetObjectType() != ObjectType::Array)

{

return true;

}

// The following javascript array types are listed in the order of most specific type to least specific type. Types are

// considered to be subsets of the less specific types below it.

// Int32 !HasNonInts() (!HasNonFloats() is implied)

// Float64 HasNonInts() && !HasNonFloats()

// Var HasNonFloats() (HasNonInts() is implied)

return

(HasNoMissingValues() || !other.HasNoMissingValues() || !isArrayMissingValueCheckHoistEnabled) &&

(

(!HasNonInts() || other.HasNonInts()) && (!HasNonFloats() || other.HasNonFloats()) ||

!isNativeArrayEnabled

);

}

ValueType ValueType::ToDefinite() const

{

Assert(!IsUninitialized());

return Verify(bits & ~Bits::Likely);

}

ValueType ValueType::ToLikelyUntaggedInt() const

{

Assert(IsLikelyInt());

Assert(!IsInt());

return Verify(bits | (Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged));

}

ValueType ValueType::ToDefiniteNumber\_PreferFloat() const

{

return IsNumber() ? \*this : ToDefiniteAnyFloat();

}

ValueType ValueType::ToDefiniteAnyFloat() const

{

// Not asserting on expected value type because float specialization allows specializing values of arbitrary types, even

// values that are definitely not float

return

Verify(

OneOn(Bits::Object)

? (Bits::Float | Bits::CanBeTaggedValue)

: bits & (Bits::Int | Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged | Bits::CanBeTaggedValue | Bits::Number) | Bits::Float);

}

ValueType ValueType::ToDefiniteNumber() const

{

Assert(IsLikelyNumber());

return IsNumber() ? \*this : ToDefiniteAnyNumber();

}

ValueType ValueType::ToDefiniteAnyNumber() const

{

// Not asserting on expected value type because Conv\_Num allows converting values of arbitrary types to number

if(OneOn(Bits::Object))

return Verify(Bits::Number | Bits::CanBeTaggedValue);

Bits numberBits =

bits &

(

Bits::Int |

Bits::IntCanBeUntagged |

Bits::IntIsLikelyUntagged |

Bits::CanBeTaggedValue |

Bits::Float |

Bits::Number

);

if(!(numberBits & (Bits::Float | Bits::Number)))

numberBits |= Bits::Number | Bits::CanBeTaggedValue;

return Verify(numberBits);

}

ValueType ValueType::ToDefinitePrimitiveSubset() const

{

// This function does not do a safe conversion of an arbitrary type to a definitely-primitive type. It only obtains the

// primitive subset of bits from the type.

// When Undefined (or Null) merge with Object, the resulting value type is still likely Object (IsLikelyObject() returns

// true). ToDefinite() on the merged type would return a type that is definitely Undefined or Object. Usually, that type is

// not interesting and a test for one or more of the primitive types may have been done. ToDefinitePrimitive() removes the

// object-specific bits.

Assert(HasBeenPrimitive());

Assert(HasBeenObject());

// If we have an object format of a type (object bit = 1) that represents a primitive (e.g. SIMD128),

// we want to keep the object bit and type along with other merged primitives (Undefined and/or Null).

if (IsLikelyObject() && IsLikelyPrimitive())

return Verify(bits & (Bits::Undefined | Bits::Null) | ToDefiniteObject().bits);

return

Verify(

bits &

(

OneOn(Bits::Object)

?

BitPattern(VALUE\_TYPE\_COMMON\_BIT\_COUNT) - (Bits::Likely | Bits::Object)

:

BitPattern(VALUE\_TYPE\_COMMON\_BIT\_COUNT + VALUE\_TYPE\_NONOBJECT\_BIT\_COUNT) -

(Bits::Likely | Bits::Object | Bits::PrimitiveOrObject)

));

}

ValueType ValueType::ToDefiniteObject() const

{

// When Undefined (or Null) merge with Object, the resulting value type is still likely Object (IsLikelyObject() returns

// true). ToDefinite() on the merged type would return a type that is definitely Undefined or Object. Usually, that type is

// not interesting and a test for the Object type may have been done to ensure the Object type. ToDefiniteObject() removes

// the Undefined and Null bits as well.

Assert(IsLikelyObject());

return Verify(bits & ~(BitPattern(VALUE\_TYPE\_COMMON\_BIT\_COUNT) - Bits::Object));

}

ValueType ValueType::ToLikely() const

{

return Verify(bits | Bits::Likely | Bits::CanBeTaggedValue);

}

ValueType ValueType::ToArray() const

{

Assert(GetObjectType() == ObjectType::ObjectWithArray);

ValueType valueType(\*this);

valueType.SetObjectType(ObjectType::Array);

return Verify(valueType);

}

ValueType ValueType::ToPrimitiveOrObject() const

{

// When an object type is merged with a non-object type, the PrimitiveOrObject bit is set in the merged type by converting

// the object type to a PrimitiveOrObject type (preserving only the common bits other than Object) and merging it with the

// non-object type. The PrimitiveOrObject type will not have the Object bit set, so that it can still be queried for whether

// it was any of the other types.

Assert(OneOn(Bits::Object));

return Verify(bits & (BitPattern(VALUE\_TYPE\_COMMON\_BIT\_COUNT) - Bits::Object) | Bits::PrimitiveOrObject);

}

\_\_forceinline ValueType ValueType::Merge(const ValueType other) const

{

Verify(\*this);

Verify(other);

if(\*this == other)

return \*this;

const ValueType merged(bits | other.bits);

if(!merged.OneOn(Bits::Object)) // neither has the Object bit set

return Verify(merged);

return MergeWithObject(other);

}

ValueType ValueType::MergeWithObject(const ValueType other) const

{

ValueType merged(bits | other.bits);

Assert(merged.OneOn(Bits::Object));

if(ValueType(bits & other.bits).OneOn(Bits::Object)) // both have the Object bit set

{

if (GetObjectType() == other.GetObjectType())

return Verify(merged);

const ObjectType typedArrayMergedObjectType =

TypedArrayMergeMap[static\_cast<uint16>(GetObjectType())][static\_cast<uint16>(other.GetObjectType())];

if (typedArrayMergedObjectType != ObjectType::UninitializedObject)

{

merged.SetObjectType(typedArrayMergedObjectType);

return Verify(merged);

}

if(GetObjectType() != ObjectType::UninitializedObject && other.GetObjectType() != ObjectType::UninitializedObject)

{

// Any two different specific object types (excludes UninitializedObject and Object, which don't indicate any

// specific type of object) merge to Object since the resulting type is not guaranteed to indicate any specific type

merged.SetObjectType(ObjectType::Object);

return Verify(merged);

}

// Since UninitializedObject is a generic object type, when merged with a specific object type, the resulting object

// type is not guaranteed to be any specific object type. However, since UninitializedObject means that we don't have

// info about the object type, we can assume that the merged type is likely an object of the specific type.

//

// If both were definitely object, we lose the information that the merged type is also definitely object. For now

// though, it's better to have "likely object of a specific type" than "definitely object of an unknown type". It may

// eventually become necessary to distinguish these and have a "definitely object, likely of a specific type".

return

Verify(

GetObjectType() > ObjectType::Object || other.GetObjectType() > ObjectType::Object

? merged.ToLikely()

: merged);

}

if(OneOn(Bits::Object))

{

if(other.CanMergeToObject())

return Verify(merged);

return Verify(ToPrimitiveOrObject().bits | other.bits); // see ToPrimitiveOrObject

}

Assert(other.OneOn(Bits::Object));

if(CanMergeToObject())

return Verify(merged);

return Verify(bits | other.ToPrimitiveOrObject().bits); // see ToPrimitiveOrObject

}

\_\_inline ValueType ValueType::Merge(const Js::Var var) const

{

using namespace Js;

Assert(var);

if(TaggedInt::Is(var))

return Merge(GetTaggedInt());

if(JavascriptNumber::Is\_NoTaggedIntCheck(var))

{

return

Merge(

(IsUninitialized() || IsLikelyInt()) && JavascriptNumber::IsInt32\_NoChecks(var)

? GetInt(false)

: ValueType::Float);

}

return Merge(FromObject(RecyclableObject::FromVar(var)));

}

ValueType::Bits ValueType::TypeIdToBits[Js::TypeIds\_Limit];

ValueType::Bits ValueType::VirtualTypeIdToBits[Js::TypeIds\_Limit];

INT\_PTR ValueType::TypeIdToVtable[Js::TypeIds\_Limit];

ObjectType ValueType::VirtualTypedArrayPair[(uint16)ObjectType::Count];

ObjectType ValueType::MixedTypedArrayPair[(uint16)ObjectType::Count];

ObjectType ValueType::TypedArrayMergeMap[(uint16)ObjectType::Count][(uint16)ObjectType::Count];

ObjectType ValueType::MixedTypedToVirtualTypedArray[(uint16)ObjectType::Count];

void ValueType::InitializeTypeIdToBitsMap()

{

using namespace Js;

// Initialize all static types to Uninitialized first, so that a zero will indicate that it's a dynamic type

for (TypeId typeId = static\_cast<TypeId>(0); typeId <= TypeIds\_LastStaticType; typeId = static\_cast<TypeId>(typeId + 1))

{

TypeIdToBits[typeId] = ValueType::Uninitialized.bits;

VirtualTypeIdToBits[typeId] = ValueType::Uninitialized.bits;

TypeIdToVtable[typeId] = (INT\_PTR)nullptr;

}

for (ObjectType objType = static\_cast<ObjectType>(0); objType <ObjectType::Count; objType = static\_cast<ObjectType>((uint16)(objType) + 1))

{

VirtualTypedArrayPair[(uint16)objType] = ObjectType::UninitializedObject;

MixedTypedArrayPair[(uint16)objType] = ObjectType::UninitializedObject;

MixedTypedToVirtualTypedArray[(uint16)objType] = ObjectType::UninitializedObject;

}

for (ObjectType objType = static\_cast<ObjectType>(0); objType < ObjectType::Count; objType = static\_cast<ObjectType>((uint16)(objType)+1))

{

for (ObjectType objTypeInner = static\_cast<ObjectType>(0); objTypeInner < ObjectType::Count; objTypeInner = static\_cast<ObjectType>((uint16)(objTypeInner)+1))

TypedArrayMergeMap[(uint16)objType][(uint16)objTypeInner] = ObjectType::UninitializedObject;

}

TypeIdToBits[TypeIds\_Undefined ] = ValueType::Undefined.bits;

TypeIdToBits[TypeIds\_Null ] = ValueType::Null.bits;

TypeIdToBits[TypeIds\_Boolean ] = ValueType::Boolean.bits;

TypeIdToBits[TypeIds\_String ] = ValueType::String.bits;

TypeIdToBits[TypeIds\_Symbol ] = ValueType::Symbol.bits;

TypeIdToBits[TypeIds\_RegEx ] = GetObject(ObjectType::RegExp).bits;

TypeIdToBits[TypeIds\_Int8Array ] = GetObject(ObjectType::Int8Array).bits;

TypeIdToBits[TypeIds\_Uint8Array ] = GetObject(ObjectType::Uint8Array).bits;

TypeIdToBits[TypeIds\_Uint8ClampedArray ] = GetObject(ObjectType::Uint8ClampedArray).bits;

TypeIdToBits[TypeIds\_Int16Array ] = GetObject(ObjectType::Int16Array).bits;

TypeIdToBits[TypeIds\_Uint16Array ] = GetObject(ObjectType::Uint16Array).bits;

TypeIdToBits[TypeIds\_Int32Array ] = GetObject(ObjectType::Int32Array).bits;

TypeIdToBits[TypeIds\_Uint32Array ] = GetObject(ObjectType::Uint32Array).bits;

TypeIdToBits[TypeIds\_Float32Array ] = GetObject(ObjectType::Float32Array).bits;

TypeIdToBits[TypeIds\_Float64Array ] = GetObject(ObjectType::Float64Array).bits;

TypeIdToBits[TypeIds\_Int64Array ] = GetObject(ObjectType::Int64Array).bits;

TypeIdToBits[TypeIds\_Uint64Array ] = GetObject(ObjectType::Uint64Array).bits;

TypeIdToBits[TypeIds\_CharArray ] = GetObject(ObjectType::CharArray).bits;

TypeIdToBits[TypeIds\_BoolArray ] = GetObject(ObjectType::BoolArray).bits;

TypeIdToBits[TypeIds\_SIMDFloat32x4 ] = GetObject(ObjectType::Simd128Float32x4).bits;

TypeIdToBits[TypeIds\_SIMDInt32x4 ] = GetObject(ObjectType::Simd128Int32x4).bits;

TypeIdToBits[TypeIds\_SIMDInt8x16 ] = GetObject(ObjectType::Simd128Int8x16).bits;

TypeIdToBits[TypeIds\_SIMDFloat64x2 ] = GetObject(ObjectType::Simd128Float64x2).bits;

VirtualTypeIdToBits[TypeIds\_Int8Array] = GetObject(ObjectType::Int8VirtualArray).bits;

VirtualTypeIdToBits[TypeIds\_Uint8Array] = GetObject(ObjectType::Uint8VirtualArray).bits;

VirtualTypeIdToBits[TypeIds\_Uint8ClampedArray] = GetObject(ObjectType::Uint8ClampedArray).bits;

VirtualTypeIdToBits[TypeIds\_Int16Array] = GetObject(ObjectType::Int16VirtualArray).bits;

VirtualTypeIdToBits[TypeIds\_Uint16Array] = GetObject(ObjectType::Uint16VirtualArray).bits;

VirtualTypeIdToBits[TypeIds\_Int32Array] = GetObject(ObjectType::Int32VirtualArray).bits;

VirtualTypeIdToBits[TypeIds\_Uint32Array] = GetObject(ObjectType::Uint32VirtualArray).bits;

VirtualTypeIdToBits[TypeIds\_Float32Array] = GetObject(ObjectType::Float32VirtualArray).bits;

VirtualTypeIdToBits[TypeIds\_Float64Array] = GetObject(ObjectType::Float64VirtualArray).bits;

TypeIdToVtable[TypeIds\_Int8Array] = VirtualTableInfo<Int8VirtualArray>::Address;

TypeIdToVtable[TypeIds\_Uint8Array] = VirtualTableInfo<Uint8VirtualArray>::Address;

TypeIdToVtable[TypeIds\_Uint8ClampedArray] = VirtualTableInfo<Uint8ClampedVirtualArray>::Address;

TypeIdToVtable[TypeIds\_Int16Array] = VirtualTableInfo<Int16VirtualArray>::Address;

TypeIdToVtable[TypeIds\_Uint16Array] = VirtualTableInfo<Uint16VirtualArray>::Address;

TypeIdToVtable[TypeIds\_Int32Array] = VirtualTableInfo<Int32VirtualArray>::Address;

TypeIdToVtable[TypeIds\_Uint32Array] = VirtualTableInfo<Uint32VirtualArray>::Address;

TypeIdToVtable[TypeIds\_Float32Array] = VirtualTableInfo<Float32VirtualArray>::Address;

TypeIdToVtable[TypeIds\_Float64Array] = VirtualTableInfo<Float64VirtualArray>::Address;

VirtualTypedArrayPair[(int)ObjectType::Int8VirtualArray] = ObjectType::Int8Array;

VirtualTypedArrayPair[(int)ObjectType::Int8Array] = ObjectType::Int8VirtualArray;

VirtualTypedArrayPair[(int)ObjectType::Uint8VirtualArray] = ObjectType::Uint8Array;

VirtualTypedArrayPair[(int)ObjectType::Uint8Array] = ObjectType::Uint8VirtualArray;

VirtualTypedArrayPair[(int)ObjectType::Uint8ClampedVirtualArray] = ObjectType::Uint8ClampedArray;

VirtualTypedArrayPair[(int)ObjectType::Uint8ClampedArray] = ObjectType::Uint8ClampedVirtualArray;

VirtualTypedArrayPair[(int)ObjectType::Int16VirtualArray] = ObjectType::Int16Array;

VirtualTypedArrayPair[(int)ObjectType::Int16Array] = ObjectType::Int16VirtualArray;

VirtualTypedArrayPair[(int)ObjectType::Uint16VirtualArray] = ObjectType::Uint16Array;

VirtualTypedArrayPair[(int)ObjectType::Uint16Array] = ObjectType::Uint16VirtualArray;

VirtualTypedArrayPair[(int)ObjectType::Int32VirtualArray] = ObjectType::Int32Array;

VirtualTypedArrayPair[(int)ObjectType::Int32Array] = ObjectType::Int32VirtualArray;

VirtualTypedArrayPair[(int)ObjectType::Uint32VirtualArray] = ObjectType::Uint32Array;

VirtualTypedArrayPair[(int)ObjectType::Uint32Array] = ObjectType::Uint32VirtualArray;

VirtualTypedArrayPair[(int)ObjectType::Float32VirtualArray] = ObjectType::Float32Array;

VirtualTypedArrayPair[(int)ObjectType::Float32Array] = ObjectType::Float32VirtualArray;

VirtualTypedArrayPair[(int)ObjectType::Float64VirtualArray] = ObjectType::Float64Array;

VirtualTypedArrayPair[(int)ObjectType::Float64Array] = ObjectType::Float64VirtualArray;

MixedTypedArrayPair[(int)ObjectType::Int8VirtualArray] = ObjectType::Int8MixedArray;

MixedTypedArrayPair[(int)ObjectType::Int8Array] = ObjectType::Int8MixedArray;

MixedTypedArrayPair[(int)ObjectType::Uint8VirtualArray] = ObjectType::Uint8MixedArray;

MixedTypedArrayPair[(int)ObjectType::Uint8Array] = ObjectType::Uint8MixedArray;

MixedTypedArrayPair[(int)ObjectType::Uint8ClampedVirtualArray] = ObjectType::Uint8ClampedMixedArray;

MixedTypedArrayPair[(int)ObjectType::Uint8ClampedArray] = ObjectType::Uint8ClampedMixedArray;

MixedTypedArrayPair[(int)ObjectType::Int16VirtualArray] = ObjectType::Int16MixedArray;

MixedTypedArrayPair[(int)ObjectType::Int16Array] = ObjectType::Int16MixedArray;

MixedTypedArrayPair[(int)ObjectType::Uint16VirtualArray] = ObjectType::Uint16MixedArray;

MixedTypedArrayPair[(int)ObjectType::Uint16Array] = ObjectType::Uint16MixedArray;

MixedTypedArrayPair[(int)ObjectType::Int32VirtualArray] = ObjectType::Int32MixedArray;

MixedTypedArrayPair[(int)ObjectType::Int32Array] = ObjectType::Int32MixedArray;

MixedTypedArrayPair[(int)ObjectType::Uint32VirtualArray] = ObjectType::Uint32MixedArray;

MixedTypedArrayPair[(int)ObjectType::Uint32Array] = ObjectType::Uint32MixedArray;

MixedTypedArrayPair[(int)ObjectType::Float32VirtualArray] = ObjectType::Float32MixedArray;

MixedTypedArrayPair[(int)ObjectType::Float32Array] = ObjectType::Float32MixedArray;

MixedTypedArrayPair[(int)ObjectType::Float64VirtualArray] = ObjectType::Float64MixedArray;

MixedTypedArrayPair[(int)ObjectType::Float64Array] = ObjectType::Float64MixedArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Int8MixedArray] = ObjectType::Int8VirtualArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Uint8MixedArray] = ObjectType::Uint8VirtualArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Uint8ClampedMixedArray] = ObjectType::Uint8ClampedVirtualArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Int16MixedArray] = ObjectType::Int16VirtualArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Uint16MixedArray] = ObjectType::Uint16VirtualArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Int32MixedArray] = ObjectType::Int32VirtualArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Uint32MixedArray] = ObjectType::Uint32VirtualArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Float32MixedArray] = ObjectType::Float32VirtualArray;

MixedTypedToVirtualTypedArray[(int)ObjectType::Float64MixedArray] = ObjectType::Float64VirtualArray;

TypedArrayMergeMap[(int)ObjectType::Int8Array][(int)ObjectType::Int8VirtualArray] = ObjectType::Int8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int8VirtualArray][(int)ObjectType::Int8Array] = ObjectType::Int8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int8MixedArray][(int)ObjectType::Int8Array] = ObjectType::Int8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int8MixedArray][(int)ObjectType::Int8VirtualArray] = ObjectType::Int8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int8Array][(int)ObjectType::Int8MixedArray] = ObjectType::Int8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int8VirtualArray][(int)ObjectType::Int8MixedArray] = ObjectType::Int8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8Array][(int)ObjectType::Uint8VirtualArray] = ObjectType::Uint8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8VirtualArray][(int)ObjectType::Uint8Array] = ObjectType::Uint8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8MixedArray][(int)ObjectType::Uint8VirtualArray] = ObjectType::Uint8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8MixedArray][(int)ObjectType::Uint8Array] = ObjectType::Uint8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8Array][(int)ObjectType::Uint8MixedArray] = ObjectType::Uint8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8VirtualArray][(int)ObjectType::Uint8MixedArray] = ObjectType::Uint8MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8ClampedArray][(int)ObjectType::Uint8ClampedVirtualArray] = ObjectType::Uint8ClampedMixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8ClampedVirtualArray][(int)ObjectType::Uint8ClampedArray] = ObjectType::Uint8ClampedMixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8ClampedMixedArray][(int)ObjectType::Uint8ClampedVirtualArray] = ObjectType::Uint8ClampedMixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8ClampedMixedArray][(int)ObjectType::Uint8ClampedArray] = ObjectType::Uint8ClampedMixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8ClampedArray][(int)ObjectType::Uint8ClampedMixedArray] = ObjectType::Uint8ClampedMixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint8ClampedVirtualArray][(int)ObjectType::Uint8ClampedMixedArray] = ObjectType::Uint8ClampedMixedArray;

TypedArrayMergeMap[(int)ObjectType::Int16Array][(int)ObjectType::Int16VirtualArray] = ObjectType::Int16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int16VirtualArray][(int)ObjectType::Int16Array] = ObjectType::Int16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int16MixedArray][(int)ObjectType::Int16VirtualArray] = ObjectType::Int16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int16MixedArray][(int)ObjectType::Int16Array] = ObjectType::Int16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int16Array][(int)ObjectType::Int16MixedArray] = ObjectType::Int16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int16VirtualArray][(int)ObjectType::Int16MixedArray] = ObjectType::Int16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint16Array][(int)ObjectType::Uint16VirtualArray] = ObjectType::Uint16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint16VirtualArray][(int)ObjectType::Uint16Array] = ObjectType::Uint16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint16MixedArray][(int)ObjectType::Uint16VirtualArray] = ObjectType::Uint16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint16MixedArray][(int)ObjectType::Uint16Array] = ObjectType::Uint16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint16Array][(int)ObjectType::Uint16MixedArray] = ObjectType::Uint16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint16VirtualArray][(int)ObjectType::Uint16MixedArray] = ObjectType::Uint16MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int32Array][(int)ObjectType::Int32VirtualArray] = ObjectType::Int32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int32VirtualArray][(int)ObjectType::Int32Array] = ObjectType::Int32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int32MixedArray][(int)ObjectType::Int32VirtualArray] = ObjectType::Int32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int32MixedArray][(int)ObjectType::Int32Array] = ObjectType::Int32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int32Array][(int)ObjectType::Int32MixedArray] = ObjectType::Int32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Int32VirtualArray][(int)ObjectType::Int32MixedArray] = ObjectType::Int32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint32Array][(int)ObjectType::Uint32VirtualArray] = ObjectType::Uint32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint32VirtualArray][(int)ObjectType::Uint32Array] = ObjectType::Uint32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint32MixedArray][(int)ObjectType::Uint32VirtualArray] = ObjectType::Uint32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint32MixedArray][(int)ObjectType::Uint32Array] = ObjectType::Uint32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint32Array][(int)ObjectType::Uint32MixedArray] = ObjectType::Uint32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Uint32VirtualArray][(int)ObjectType::Uint32MixedArray] = ObjectType::Uint32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float32Array][(int)ObjectType::Float32VirtualArray] = ObjectType::Float32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float32VirtualArray][(int)ObjectType::Float32Array] = ObjectType::Float32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float32MixedArray][(int)ObjectType::Float32VirtualArray] = ObjectType::Float32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float32MixedArray][(int)ObjectType::Float32Array] = ObjectType::Float32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float32Array][(int)ObjectType::Float32MixedArray] = ObjectType::Float32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float32VirtualArray][(int)ObjectType::Float32MixedArray] = ObjectType::Float32MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float64Array][(int)ObjectType::Float64VirtualArray] = ObjectType::Float64MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float64VirtualArray][(int)ObjectType::Float64Array] = ObjectType::Float64MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float64MixedArray][(int)ObjectType::Float64VirtualArray] = ObjectType::Float64MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float64MixedArray][(int)ObjectType::Float64Array] = ObjectType::Float64MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float64Array][(int)ObjectType::Float64MixedArray] = ObjectType::Float64MixedArray;

TypedArrayMergeMap[(int)ObjectType::Float64VirtualArray][(int)ObjectType::Float64MixedArray] = ObjectType::Float64MixedArray;

}

INT\_PTR ValueType::GetVirtualTypedArrayVtable(const Js::TypeId typeId)

{

if (typeId < \_countof(TypeIdToVtable))

{

return TypeIdToVtable[typeId];

}

return NULL;

}

ValueType ValueType::FromTypeId(const Js::TypeId typeId, bool useVirtual)

{

if(typeId < \_countof(TypeIdToBits))

{

if (useVirtual)

{

const Bits bits = VirtualTypeIdToBits[typeId];

if (!!bits)

return bits;

}

else

{

const Bits bits = TypeIdToBits[typeId];

if (!!bits)

return bits;

}

}

return Uninitialized;

}

ValueType ValueType::FromObject(Js::RecyclableObject \*const recyclableObject)

{

using namespace Js;

Assert(recyclableObject);

const TypeId typeId = recyclableObject->GetTypeId();

if (typeId < \_countof(TypeIdToBits))

{

const Bits bits = TypeIdToBits[typeId];

if (!!bits)

{

const ValueType valueType = Verify(bits);

if (!valueType.IsLikelyOptimizedTypedArray())

return valueType;

bool isVirtual = (VirtualTableInfoBase::GetVirtualTable(recyclableObject) == ValueType::GetVirtualTypedArrayVtable(typeId));

if (!isVirtual)

return valueType;

return GetObject(VirtualTypedArrayPair[static\_cast<uint16>(valueType.GetObjectType())]);

}

}

Assert(DynamicType::Is(typeId)); // all static type IDs have nonzero values in TypeIdToBits

if(!JavascriptArray::Is(typeId))

{

// TODO: Once the issue with loop bodies and uninitialized interpreter local slots is fixed, use FromVar

DynamicObject \*const object = static\_cast<DynamicObject \*>(recyclableObject);

if(!VirtualTableInfo<DynamicObject>::HasVirtualTable(object) || !object->HasObjectArray())

return GetObject(ObjectType::Object);

return FromObjectWithArray(object);

}

return FromArray(ObjectType::Array, static\_cast<JavascriptArray \*>(recyclableObject), typeId);

}

ValueType ValueType::FromObjectWithArray(Js::DynamicObject \*const object)

{

using namespace Js;

Assert(object);

Assert(VirtualTableInfo<DynamicObject>::HasVirtualTable(object));

Assert(object->GetTypeId() == TypeIds\_Object); // this check should be a superset of the DynamicObject vtable check above

Assert(object->HasObjectArray());

ArrayObject \*const objectArray = object->GetObjectArray();

Assert(objectArray);

if(!VirtualTableInfo<JavascriptArray>::HasVirtualTable(objectArray))

return GetObject(ObjectType::Object);

return FromObjectArray(JavascriptArray::FromVar(objectArray));

}

\_\_inline ValueType ValueType::FromObjectArray(Js::JavascriptArray \*const objectArray)

{

using namespace Js;

Assert(objectArray);

return FromArray(ObjectType::ObjectWithArray, objectArray, TypeIds\_Array); // objects with native arrays are currently not supported

}

\_\_inline ValueType ValueType::FromArray(

const ObjectType objectType,

Js::JavascriptArray \*const array,

const Js::TypeId arrayTypeId)

{

Assert(array);

Assert(array->GetTypeId() == arrayTypeId);

// TODO: Once the issue with loop bodies and uninitialized interpreter local slots is fixed, use FromVar and the checked version of HasNoMissingValues

return

Verify(

GetArray(objectType)

.SetHasNoMissingValues(array->HasNoMissingValues\_Unchecked())

.SetArrayTypeId(arrayTypeId));

}

bool ValueType::operator ==(const ValueType other) const

{

return bits == other.bits;

}

bool ValueType::operator !=(const ValueType other) const

{

return !(\*this == other);

}

uint ValueType::GetHashCode() const

{

return static\_cast<uint>(bits);

}

const char \*const ValueType::BitNames[] =

{

#define VALUE\_TYPE\_BIT(t, b) "" STRINGIZE(t) "",

#include "ValueTypes.h"

#undef VALUE\_TYPE\_BIT

};

const char \*const ObjectTypeNames[] =

{

#define OBJECT\_TYPE(ot) "" STRINGIZE(ot) "",

#include "ValueTypes.h"

#undef OBJECT\_TYPE

};

size\_t ValueType::GetLowestBitIndex(const Bits b)

{

Assert(b);

DWORD i;

::GetFirstBitSet(&i, static\_cast<UnitWord32>(b));

return i;

}

void ValueType::ToVerboseString(char (&str)[VALUE\_TYPE\_MAX\_STRING\_SIZE]) const

{

if(IsUninitialized())

{

strcpy\_s(str, "Uninitialized");

return;

}

Bits b = bits;

if(OneOn(Bits::Object))

{

// Exclude the object type for enumerating bits, and exclude bits specific to a different object type

b = \_objectBits;

if(IsLikelyArrayOrObjectWithArray())

b &= ~(Bits::NonInts | Bits::NonFloats); // these are handled separately for better readability

else

b &= ~BitPattern(VALUE\_TYPE\_ARRAY\_BIT\_COUNT, VALUE\_TYPE\_COMMON\_BIT\_COUNT);

}

else if(!CONFIG\_FLAG(Verbose))

b &= ~(Bits::IntCanBeUntagged | Bits::IntIsLikelyUntagged); // these will be simplified

size\_t length = 0;

bool addUnderscore = false;

size\_t nameIndexOffset = 0;

do

{

const char \*name;

switch(b & -b) // bit to be printed

{

case Bits::Object:

if(IsLikelyNativeArray())

{

Assert(GetObjectType() == ObjectType::Array || GetObjectType() == ObjectType::ObjectWithArray);

Assert(HasIntElements() || HasFloatElements());

name =

GetObjectType() == ObjectType::Array

? HasIntElements() ? "NativeIntArray" : "NativeFloatArray"

: HasIntElements() ? "ObjectWithNativeIntArray" : "ObjectWithNativeFloatArray";

break;

}

name = ObjectTypeNames[static\_cast<TSize>(GetObjectType())]; // print the object type instead

break;

case Bits::Int:

if(!CONFIG\_FLAG(Verbose) && !OneOn(Bits::Object))

{

if(AnyOnExcept(Bits::Likely | Bits::IntCanBeUntagged | Bits::CanBeTaggedValue))

{

name = "TaggedInt";

break;

}

if(OneOn(Bits::IntIsLikelyUntagged))

{

name = "IntAndLikelyUntagged";

break;

}

}

// fall through

default:

size\_t nameIndex = nameIndexOffset + GetLowestBitIndex(b);

Assert(nameIndex < sizeof(BitNames) / sizeof(BitNames[0]));

\_\_analysis\_assume(nameIndex < sizeof(BitNames) / sizeof(BitNames[0])); // function is not used in shipping builds, satisfy oacr

name = BitNames[nameIndex];

break;

}

size\_t nameLength = strlen(name);

if(addUnderscore)

++nameLength;

if(length + nameLength >= sizeof(str) / sizeof(str[0]))

break;

if(addUnderscore)

{

str[length++] = '\_';

--nameLength;

}

else

addUnderscore = !(b & Bits::Likely);

js\_memcpy\_s(&str[length], sizeof(str) / sizeof(str[0]) - 1 - length, name, nameLength);

length += nameLength;

if((b & -b) == BitPattern(1, VALUE\_TYPE\_OBJECT\_BIT\_INDEX)) // if the bit that was just printed is the last common bit

nameIndexOffset += VALUE\_TYPE\_NONOBJECT\_BIT\_COUNT; // skip bit names for bits that only apply when the Object bit is set

b &= b - 1; // unset the least significant set bit

} while(!!b);

Assert(length < sizeof(str) / sizeof(str[0]));

str[length] = '\0';

}

void ValueType::ToString(wchar (&str)[VALUE\_TYPE\_MAX\_STRING\_SIZE]) const

{

char charStr[VALUE\_TYPE\_MAX\_STRING\_SIZE];

ToString(charStr);

for(int i = 0; i < VALUE\_TYPE\_MAX\_STRING\_SIZE; ++i)

{

str[i] = charStr[i];

if(!charStr[i])

break;

}

}

void ValueType::ToString(char (&str)[VALUE\_TYPE\_MAX\_STRING\_SIZE]) const

{

if(IsUninitialized() || CONFIG\_FLAG(Verbose))

{

ToVerboseString(str);

return;

}

bool canBeTaggedValue = CanBeTaggedValue();

const ValueType definiteType = ToDefinite();

ValueType generalizedType;

if(definiteType.IsInt())

generalizedType = definiteType;

else if(definiteType.IsFloat())

generalizedType = Float;

else if(definiteType.IsNumber())

{

generalizedType = Number;

if(definiteType.IsLikelyInt())

generalizedType = generalizedType.Merge(GetInt(definiteType.IsLikelyTaggedInt()));

}

else if(definiteType.IsUndefined())

generalizedType = Undefined;

else if(definiteType.IsNull())

generalizedType = Null;

else if(definiteType.IsBoolean())

generalizedType = Boolean;

else if (definiteType.IsString())

generalizedType = String;

else if (definiteType.IsSymbol())

generalizedType = Symbol;

else if(definiteType.IsPrimitive() && !IsLikelyObject())

{

strcpy\_s(str, IsDefinite() ? "Primitive" : "LikelyPrimitive");

return;

}

else if(definiteType.IsLikelyObject())

{

generalizedType = definiteType.ToDefiniteObject();

if(!definiteType.IsObject())

generalizedType = generalizedType.ToLikely();

}

else

{

strcpy\_s(str, IsDefinite() ? "Mixed" : "LikelyMixed");

return;

}

if(!IsDefinite())

generalizedType = generalizedType.ToLikely();

generalizedType.SetCanBeTaggedValue(canBeTaggedValue).ToVerboseString(str);

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void ValueType::ToStringDebug(\_\_out\_ecount(strSize) char \*const str, const size\_t strSize) const

{

Assert(str);

if(strSize == 0)

return;

char generalizedStr[VALUE\_TYPE\_MAX\_STRING\_SIZE];

ToString(generalizedStr);

char verboseStr[VALUE\_TYPE\_MAX\_STRING\_SIZE];

ToVerboseString(verboseStr);

const size\_t generalizedStrLength = strlen(generalizedStr);

if(strcmp(generalizedStr, verboseStr) == 0)

{

if(generalizedStrLength >= strSize)

{

str[0] = '\0';

return;

}

strcpy\_s(str, strSize, generalizedStr);

return;

}

const size\_t verboseStrLength = strlen(verboseStr);

if(generalizedStrLength + verboseStrLength + 3 >= strSize)

{

str[0] = '\0';

return;

}

sprintf\_s(str, strSize, "%s (%s)", generalizedStr, verboseStr);

}

#endif

bool ValueType::FromString(const wchar \*const str, ValueType \*valueType)

{

Assert(str);

Assert(valueType);

char charStr[VALUE\_TYPE\_MAX\_STRING\_SIZE];

int i = 0;

for(; i < VALUE\_TYPE\_MAX\_STRING\_SIZE - 1 && str[i]; ++i)

{

Assert(static\_cast<wchar>(static\_cast<char>(str[i])) == str[i]);

charStr[i] = static\_cast<char>(str[i]);

}

charStr[i] = '\0';

return FromString(charStr, valueType);

}

bool ValueType::FromString(const char \*const str, ValueType \*valueType)

{

Assert(str);

Assert(valueType);

bool found = false;

MapInitialValueTypesUntil([&](const ValueType initialValueType, const size\_t) -> bool

{

char valueTypeStr[VALUE\_TYPE\_MAX\_STRING\_SIZE];

initialValueType.ToString(valueTypeStr);

if(strcmp(str, valueTypeStr))

return false;

\*valueType = initialValueType;

found = true;

return true;

});

return found;

}

ValueType::TSize ValueType::GetRawData() const

{

return static\_cast<TSize>(bits);

}

ValueType ValueType::FromRawData(const TSize rawData)

{

return Verify(static\_cast<Bits>(rawData));

}

// Virtual and Mixed Typed Array Methods

bool ValueType::IsVirtualTypedArrayPair(const ObjectType other) const

{

return (VirtualTypedArrayPair[(int)GetObjectType()] == other);

}

bool ValueType::IsLikelyMixedTypedArrayType() const

{

return (IsLikelyObject() && GetObjectType() >= ObjectType::Int8MixedArray && GetObjectType() <= ObjectType::Float64MixedArray);

}

bool ValueType::IsMixedTypedArrayPair(const ValueType other) const

{

return ( IsLikelyObject() && other.IsLikelyObject() &&

( (MixedTypedArrayPair[(int)GetObjectType()] == other.GetObjectType()) ||

(MixedTypedArrayPair[(int)other.GetObjectType()] == GetObjectType()) ||

(IsLikelyMixedTypedArrayType() && other.IsLikelyMixedTypedArrayType())

)

);

}

ValueType ValueType::ChangeToMixedTypedArrayType() const

{

ObjectType objType = MixedTypedArrayPair[(int)GetObjectType()];

Assert(objType);

ValueType valueType(bits);

valueType.SetObjectType(objType);

return Verify(valueType);

}

ObjectType ValueType::GetMixedTypedArrayObjectType() const

{

return MixedTypedArrayPair[(int)GetObjectType()];

}

ObjectType ValueType::GetMixedToVirtualTypedArrayObjectType() const

{

return MixedTypedToVirtualTypedArray[(int)GetObjectType()];

}

#if DBG

void ValueType::RunUnitTests()

{

Assert(Uninitialized.bits == (Bits::Likely | Bits::CanBeTaggedValue));

Assert(!ObjectType::UninitializedObject); // this is assumed in Merge

const ValueType TaggedInt(GetTaggedInt());

const ValueType IntAndLikelyTagged(GetInt(true));

const ValueType IntAndLikelyUntagged(GetInt(false));

Assert(TaggedInt.IsTaggedInt());

Assert(TaggedInt.IsIntAndLikelyTagged());

Assert(TaggedInt.IsLikelyTaggedInt());

Assert(!TaggedInt.IsLikelyUntaggedInt());

Assert(TaggedInt.IsInt());

Assert(TaggedInt.IsLikelyInt());

Assert(!TaggedInt.IsLikelyFloat());

Assert(TaggedInt.IsNumber());

Assert(TaggedInt.IsLikelyNumber());

Assert(TaggedInt.IsPrimitive());

Assert(TaggedInt.IsLikelyPrimitive());

Assert(!IntAndLikelyTagged.IsTaggedInt());

Assert(IntAndLikelyTagged.IsIntAndLikelyTagged());

Assert(IntAndLikelyTagged.IsLikelyTaggedInt());

Assert(!IntAndLikelyTagged.IsLikelyUntaggedInt());

Assert(IntAndLikelyTagged.IsInt());

Assert(IntAndLikelyTagged.IsLikelyInt());

Assert(!IntAndLikelyTagged.IsLikelyFloat());

Assert(IntAndLikelyTagged.IsNumber());

Assert(IntAndLikelyTagged.IsLikelyNumber());

Assert(IntAndLikelyTagged.IsPrimitive());

Assert(IntAndLikelyTagged.IsLikelyPrimitive());

Assert(GetNumberAndLikelyInt(true).IsLikelyTaggedInt());

Assert(!GetNumberAndLikelyInt(true).IsLikelyUntaggedInt());

Assert(!GetNumberAndLikelyInt(true).IsInt());

Assert(GetNumberAndLikelyInt(true).IsLikelyInt());

Assert(!GetNumberAndLikelyInt(true).IsLikelyFloat());

Assert(GetNumberAndLikelyInt(true).IsNumber());

Assert(GetNumberAndLikelyInt(true).IsLikelyNumber());

Assert(GetNumberAndLikelyInt(true).IsPrimitive());

Assert(GetNumberAndLikelyInt(true).IsLikelyPrimitive());

Assert(TaggedInt.ToLikely().IsLikelyTaggedInt());

Assert(!TaggedInt.ToLikely().IsLikelyUntaggedInt());

Assert(TaggedInt.ToLikely().IsLikelyInt());

Assert(!TaggedInt.ToLikely().IsLikelyFloat());

Assert(TaggedInt.ToLikely().IsLikelyNumber());

Assert(!TaggedInt.ToLikely().IsPrimitive());

Assert(TaggedInt.ToLikely().IsLikelyPrimitive());

Assert(IntAndLikelyTagged.ToLikely().IsLikelyTaggedInt());

Assert(!IntAndLikelyTagged.ToLikely().IsLikelyUntaggedInt());

Assert(IntAndLikelyTagged.ToLikely().IsLikelyInt());

Assert(!IntAndLikelyTagged.ToLikely().IsLikelyFloat());

Assert(IntAndLikelyTagged.ToLikely().IsLikelyNumber());

Assert(!IntAndLikelyTagged.ToLikely().IsPrimitive());

Assert(IntAndLikelyTagged.ToLikely().IsLikelyPrimitive());

Assert(!IntAndLikelyUntagged.IsLikelyTaggedInt());

Assert(IntAndLikelyUntagged.IsIntAndLikelyUntagged());

Assert(IntAndLikelyUntagged.IsLikelyUntaggedInt());

Assert(IntAndLikelyUntagged.IsInt());

Assert(IntAndLikelyUntagged.IsLikelyInt());

Assert(!IntAndLikelyUntagged.IsLikelyFloat());

Assert(IntAndLikelyUntagged.IsNumber());

Assert(IntAndLikelyUntagged.IsLikelyNumber());

Assert(IntAndLikelyUntagged.IsPrimitive());

Assert(IntAndLikelyUntagged.IsLikelyPrimitive());

Assert(!GetNumberAndLikelyInt(false).IsLikelyTaggedInt());

Assert(!GetNumberAndLikelyInt(false).IsIntAndLikelyUntagged());

Assert(GetNumberAndLikelyInt(false).IsLikelyUntaggedInt());

Assert(!GetNumberAndLikelyInt(false).IsInt());

Assert(GetNumberAndLikelyInt(false).IsLikelyInt());

Assert(!GetNumberAndLikelyInt(false).IsLikelyFloat());

Assert(GetNumberAndLikelyInt(false).IsNumber());

Assert(GetNumberAndLikelyInt(false).IsLikelyNumber());

Assert(GetNumberAndLikelyInt(false).IsPrimitive());

Assert(GetNumberAndLikelyInt(false).IsLikelyPrimitive());

Assert(!IntAndLikelyUntagged.ToLikely().IsLikelyTaggedInt());

Assert(IntAndLikelyUntagged.ToLikely().IsLikelyUntaggedInt());

Assert(IntAndLikelyUntagged.ToLikely().IsLikelyInt());

Assert(!IntAndLikelyUntagged.ToLikely().IsLikelyFloat());

Assert(IntAndLikelyUntagged.ToLikely().IsLikelyNumber());

Assert(!IntAndLikelyUntagged.ToLikely().IsPrimitive());

Assert(IntAndLikelyUntagged.ToLikely().IsLikelyPrimitive());

Assert(!Float.IsLikelyInt());

Assert(Float.IsFloat());

Assert(Float.IsLikelyFloat());

Assert(Float.IsNumber());

Assert(Float.IsLikelyNumber());

Assert(Float.IsPrimitive());

Assert(Float.IsLikelyPrimitive());

Assert(!Float.ToLikely().IsLikelyInt());

Assert(Float.ToLikely().IsLikelyFloat());

Assert(Float.ToLikely().IsLikelyNumber());

Assert(!Float.ToLikely().IsPrimitive());

Assert(!Float.ToLikely().IsPrimitive());

Assert(Float.ToLikely().IsLikelyPrimitive());

Assert(!Number.IsLikelyInt());

Assert(!Number.IsLikelyFloat());

Assert(Number.IsNumber());

Assert(Number.IsUnknownNumber());

Assert(Number.IsLikelyNumber());

Assert(Number.IsPrimitive());

Assert(Number.IsLikelyPrimitive());

Assert(!Number.ToLikely().IsLikelyInt());

Assert(!Number.ToLikely().IsLikelyFloat());

Assert(Number.ToLikely().IsLikelyNumber());

Assert(Number.ToLikely().IsLikelyUnknownNumber());

Assert(!Number.ToLikely().IsPrimitive());

Assert(Number.ToLikely().IsLikelyPrimitive());

Assert(!UninitializedObject.IsLikelyPrimitive());

Assert(UninitializedObject.IsObject());

Assert(UninitializedObject.IsLikelyObject());

Assert(!UninitializedObject.ToLikely().IsLikelyPrimitive());

Assert(!UninitializedObject.ToLikely().IsObject());

Assert(UninitializedObject.ToLikely().IsLikelyObject());

Assert(Undefined.IsNotInt());

Assert(!Undefined.ToLikely().IsNotInt());

Assert(Null.IsNotInt());

Assert(!Null.ToLikely().IsNotInt());

Assert(Boolean.IsNotInt());

Assert(!Boolean.ToLikely().IsNotInt());

Assert(String.IsNotInt());

Assert(!String.ToLikely().IsNotInt());

Assert(UninitializedObject.IsNotInt());

Assert(!UninitializedObject.ToLikely().IsNotInt());

{

const ValueType m(IntAndLikelyUntagged.Merge(Null));

Assert(m.IsPrimitive());

Assert(m.IsLikelyPrimitive());

Assert(IntAndLikelyUntagged.IsSubsetOf(m, true, true, true, true));

Assert(!m.IsSubsetOf(IntAndLikelyUntagged, true, true, true, true));

}

{

const ValueType m(IntAndLikelyUntagged.Merge(UninitializedObject));

Assert(m.HasBeenInt());

Assert(!m.IsLikelyPrimitive());

Assert(m.HasBeenObject());

Assert(!m.IsLikelyObject());

}

{

const ValueType m(Uninitialized.Merge(IntAndLikelyTagged));

Assert(!m.IsPrimitive());

Assert(!m.IsDefinite());

Assert(m.IsLikelyTaggedInt());

}

{

const ValueType m(UninitializedObject.Merge(Null));

Assert(UninitializedObject.IsSubsetOf(m, true, true, true, true));

Assert(!m.IsSubsetOf(UninitializedObject, true, true, true, true));

Assert(Null.IsSubsetOf(m, true, true, true, true));

Assert(!m.IsSubsetOf(Null, true, true, true, true));

Assert(!TaggedInt.IsSubsetOf(m, true, true, true, true));

Assert(!m.IsSubsetOf(TaggedInt, true, true, true, true));

const ValueType po = m.Merge(TaggedInt);

Assert(m.IsSubsetOf(po, true, true, true, true));

Assert(!po.IsSubsetOf(m, true, true, true, true));

}

MapInitialValueTypesUntil([](const ValueType valueType0, const size\_t i) -> bool

{

MapInitialValueTypesUntil([=](const ValueType t1, const size\_t j) -> bool

{

if(j < i)

return false;

const ValueType t0(valueType0);

const ValueType m(t0.Merge(t1));

Assert(m.bits == t1.Merge(t0).bits);

Assert(m.IsUninitialized() == (t0.IsUninitialized() && t1.IsUninitialized()));

const bool isSubsetWithTypeSpecEnabled = t0.IsSubsetOf(t1, true, true, true, true);

if(t0.IsUninitialized())

{

Assert(isSubsetWithTypeSpecEnabled == t1.IsUninitialized());

return false;

}

else if(t1.IsUninitialized())

{

Assert(isSubsetWithTypeSpecEnabled);

return false;

}

Assert(m.IsIntAndLikelyTagged() == (t0.IsIntAndLikelyTagged() && t1.IsIntAndLikelyTagged()));

Assert(

m.IsLikelyTaggedInt() ==

(

t0.IsLikelyNumber() && t1.IsLikelyNumber() && // both are likely number

!t0.IsLikelyFloat() && !t1.IsLikelyFloat() && // neither is likely float

!t0.IsLikelyUntaggedInt() && !t1.IsLikelyUntaggedInt() && // neither is likely untagged int

(t0.IsLikelyTaggedInt() || t1.IsLikelyTaggedInt()) // one is likely tagged int

));

Assert(m.IsInt() == (t0.IsInt() && t1.IsInt()));

Assert(

m.IsLikelyInt() ==

(

t0.IsLikelyNumber() && t1.IsLikelyNumber() && // both are likely number

!t0.IsLikelyFloat() && !t1.IsLikelyFloat() && // neither is likely float

(t0.IsLikelyInt() || t1.IsLikelyInt()) // one is likely int

));

if(!(

t0.IsObject() && t1.IsObject() && // both are objects

(

t0.GetObjectType() == ObjectType::UninitializedObject ||

t1.GetObjectType() == ObjectType::UninitializedObject

) && // one has an uninitialized object type

(t0.GetObjectType() > ObjectType::Object || t1.GetObjectType() > ObjectType::Object) // one has a specific object type

)) // then the resulting object type is not guaranteed

{

Assert(m.IsNotInt() == (t0.IsNotInt() && t1.IsNotInt()));

}

Assert(m.IsFloat() == (t0.IsNumber() && t1.IsNumber() && (t0.IsFloat() || t1.IsFloat())));

Assert(

m.IsLikelyFloat() ==

(

(t0.IsLikelyFloat() || t1.IsLikelyFloat()) && // one is likely float

(t0.IsLikelyUndefined() || t0.IsLikelyNumber()) &&

(t1.IsLikelyUndefined() || t1.IsLikelyNumber()) // both are likely undefined or number

));

Assert(m.IsNumber() == (t0.IsNumber() && t1.IsNumber()));

Assert(

m.IsLikelyNumber() ==

(

(t0.IsLikelyNumber() || t1.IsLikelyNumber()) && // one is likely number

(t0.IsLikelyUndefined() || t0.IsLikelyNumber()) &&

(t1.IsLikelyUndefined() || t1.IsLikelyNumber()) // both are likely undefined or number

));

Assert(m.IsUnknownNumber() == (m.IsNumber() && !m.IsLikelyInt() && !m.IsLikelyFloat()));

Assert(!m.IsLikelyUnknownNumber() || m.IsLikelyNumber() && !m.IsLikelyInt() && !m.IsLikelyFloat());

Assert(m.IsUndefined() == (t0.IsUndefined() && t1.IsUndefined()));

Assert(m.IsLikelyUndefined() == (t0.IsLikelyUndefined() && t1.IsLikelyUndefined()));

Assert(m.IsNull() == (t0.IsNull() && t1.IsNull()));

Assert(m.IsLikelyNull() == (t0.IsLikelyNull() && t1.IsLikelyNull()));

Assert(m.IsBoolean() == (t0.IsBoolean() && t1.IsBoolean()));

Assert(m.IsLikelyBoolean() == (t0.IsLikelyBoolean() && t1.IsLikelyBoolean()));

Assert(m.IsString() == (t0.IsString() && t1.IsString()));

Assert(m.IsLikelyString() == (t0.IsLikelyString() && t1.IsLikelyString()));

if(!(

t0.IsObject() && t1.IsObject() && // both are objects

(

t0.GetObjectType() == ObjectType::UninitializedObject ||

t1.GetObjectType() == ObjectType::UninitializedObject

) && // one has an uninitialized object type

(t0.GetObjectType() > ObjectType::Object || t1.GetObjectType() > ObjectType::Object) // one has a specific object type

)) // then the resulting object type is not guaranteed

{

Assert(m.IsObject() == (t0.IsObject() && t1.IsObject()));

}

Assert(

m.IsLikelyObject() ==

(

(t0.IsLikelyObject() || t1.IsLikelyObject()) && // one is likely object

(t0.IsLikelyUndefined() || t0.IsLikelyNull() || t0.IsLikelyObject()) &&

(t1.IsLikelyUndefined() || t1.IsLikelyNull() || t1.IsLikelyObject()) // both are likely undefined, null, or object

));

if(t1.IsUnknownNumber())

{

Assert(isSubsetWithTypeSpecEnabled == (t0.IsNumber() || t0.IsLikelyInt() || t0.IsLikelyFloat()));

Assert(t0.IsSubsetOf(t1, false, true, true, true) == (t0.IsNumber() || t0.IsLikelyFloat()));

Assert(t0.IsSubsetOf(t1, true, false, true, true) == (t0.IsNumber() || t0.IsLikelyInt()));

}

else if(t0.IsLikelyInt() && t1.IsLikelyInt())

{

Assert(

isSubsetWithTypeSpecEnabled ==

(

(t0.IsDefinite() || !t1.IsDefinite()) &&

(

t0.IsTaggedInt() ||

t0.IsLikelyTaggedInt() && !t1.IsTaggedInt() ||

!t1.IsLikelyTaggedInt()

)

));

}

else if(t0.IsLikelyFloat() && t1.IsLikelyFloat())

{

Assert(isSubsetWithTypeSpecEnabled == (t0.IsDefinite() || !t1.IsDefinite()));

}

else if(t0.IsLikelyNumber() && t1.IsLikelyNumber())

{

Assert(

isSubsetWithTypeSpecEnabled ==

(

(t0.IsDefinite() || !t1.IsDefinite()) &&

(

t0.IsLikelyInt() && !t1.IsLikelyFloat() ||

t0.IsLikelyFloat() && !t1.IsLikelyInt() ||

t1.IsLikelyUnknownNumber()

)

));

}

else if(t0.IsLikelyObject() && (t1.IsLikelyUndefined() || t1.IsLikelyNull()))

{

Assert(isSubsetWithTypeSpecEnabled);

}

else if(t0.IsLikelyObject() && t1.IsLikelyObject())

{

if(t1.GetObjectType() == ObjectType::UninitializedObject &&

t0.GetObjectType() != ObjectType::UninitializedObject)

{

Assert(isSubsetWithTypeSpecEnabled);

}

else if(!t0.IsDefinite() && t1.IsDefinite() || t0.GetObjectType() != t1.GetObjectType())

{

Assert(!isSubsetWithTypeSpecEnabled);

}

else if(

t0.IsDefinite() && !t1.IsDefinite() ||

t0.GetObjectType() != ObjectType::ObjectWithArray && t0.GetObjectType() != ObjectType::Array)

{

Assert(isSubsetWithTypeSpecEnabled);

}

else

{

Assert(

isSubsetWithTypeSpecEnabled ==

(

(t0.HasNoMissingValues() || !t1.HasNoMissingValues()) &&

(

(!t0.HasNonInts() || t1.HasNonInts()) && (!t0.HasNonFloats() || t1.HasNonFloats())

)

));

Assert(

t0.IsSubsetOf(t1, true, true, false, true) ==

(

(!t0.HasNonInts() || t1.HasNonInts()) && (!t0.HasNonFloats() || t1.HasNonFloats())

));

Assert(t0.IsSubsetOf(t1, true, true, false, false));

}

}

else

{

Assert(

isSubsetWithTypeSpecEnabled ==

(

(t0.IsDefinite() || !t1.IsDefinite()) &&

!t0.IsLikelyObject() && !t1.IsLikelyObject() &&

t1.AllOn(t0.bits)

));

}

return false;

});

return false;

});

}

#endif

void ValueType::InstantiateForceInlinedMembers()

{

// Force-inlined functions defined in a translation unit need a reference from an extern non-force-inlined function in the

// same translation unit to force an instantiation of the force-inlined function. Otherwise, if the force-inlined function

// is not referenced in the same translation unit, it will not be generated and the linker is not able to find the

// definition to inline the function in other translation units.

AnalysisAssert(false);

const Js::Var var = nullptr;

ValueType \*const t = nullptr;

t->Merge(\*t);

t->Merge(var);

}

bool ValueTypeComparer::Equals(const ValueType t0, const ValueType t1)

{

return t0 == t1;

}

uint ValueTypeComparer::GetHashCode(const ValueType t)

{

return t.GetHashCode();

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

// Max string size for ToString, in number of characters including the null terminator

#define VALUE\_TYPE\_MAX\_STRING\_SIZE (256)

enum class ObjectType : uint16

{

#define OBJECT\_TYPE(ot) ot,

#include "ValueTypes.h"

#undef OBJECT\_TYPE

};

ENUM\_CLASS\_HELPERS(ObjectType, uint16);

extern const char \*const ObjectTypeNames[];

class ValueType

{

public:

typedef uint16 TSize;

private:

enum class Bits : TSize

{

#define VALUE\_TYPE\_BIT(t, b) t = (b),

#include "ValueTypes.h"

#undef VALUE\_TYPE\_BIT

};

public:

#define BASE\_VALUE\_TYPE(t, b) static const ValueType t;

#include "ValueTypes.h"

#undef BASE\_VALUE\_TYPE

static const ValueType AnyNumber;

public:

static void Initialize();

private:

static Bits BitPattern(const TSize onCount);

static Bits BitPattern(const TSize onCount, const TSize offCount);

public:

static ValueType GetTaggedInt();

static ValueType GetInt(const bool isLikelyTagged);

static ValueType GetNumberAndLikelyInt(const bool isLikelyTagged);

static ValueType GetObject(const ObjectType objectType);

// SIMD\_JS

static ValueType GetSimd128(const ObjectType objectType);

private:

static ValueType GetArray(const ObjectType objectType);

private:

union

{

// Don't use the following directly because they only apply to specific types. They're mostly for debugger-friendliness.

struct

{

TSize : VALUE\_TYPE\_OBJECT\_BIT\_INDEX;

TSize \_objectBit : 1;

};

struct

{

Bits \_nonObjectBits : VALUE\_TYPE\_COMMON\_BIT\_COUNT + VALUE\_TYPE\_NONOBJECT\_BIT\_COUNT;

};

struct

{

Bits \_objectBits : VALUE\_TYPE\_COMMON\_BIT\_COUNT + VALUE\_TYPE\_OBJECT\_BIT\_COUNT;

ObjectType \_objectType : sizeof(TSize) \* 8 - (VALUE\_TYPE\_COMMON\_BIT\_COUNT + VALUE\_TYPE\_OBJECT\_BIT\_COUNT); // use remaining bits

};

Bits bits;

};

public:

ValueType();

private:

ValueType(const Bits bits);

static ValueType Verify(const Bits bits);

static ValueType Verify(const ValueType valueType);

private:

bool OneOn(const Bits b) const;

bool AnyOn(const Bits b) const;

bool AllEqual(const Bits b, const Bits e) const;

bool AllOn(const Bits b) const;

bool OneOnOneOff(const Bits on, const Bits off) const;

bool AllOnAllOff(const Bits on, const Bits off) const;

bool OneOnOthersOff(const Bits b) const;

bool OneOnOthersOff(const Bits b, const Bits ignore) const;

bool AnyOnOthersOff(const Bits b) const;

bool AnyOnOthersOff(const Bits b, const Bits ignore) const;

bool AllOnOthersOff(const Bits b) const;

bool AllOnOthersOff(const Bits b, const Bits ignore) const;

bool AnyOnExcept(const Bits b) const;

public:

bool IsUninitialized() const;

bool IsDefinite() const;

bool IsTaggedInt() const;

bool IsIntAndLikelyTagged() const;

bool IsLikelyTaggedInt() const;

bool HasBeenUntaggedInt() const;

bool IsIntAndLikelyUntagged() const;

bool IsLikelyUntaggedInt() const;

bool IsNotTaggedValue() const;

bool CanBeTaggedValue() const;

ValueType SetCanBeTaggedValue(const bool b) const;

bool HasBeenInt() const;

bool IsInt() const;

bool IsLikelyInt() const;

bool IsNotInt() const;

bool IsNotNumber() const;

bool HasBeenFloat() const;

bool IsFloat() const;

bool IsLikelyFloat() const;

bool HasBeenNumber() const;

bool IsNumber() const;

bool IsLikelyNumber() const;

bool HasBeenUnknownNumber() const;

bool IsUnknownNumber() const;

bool IsLikelyUnknownNumber() const;

bool HasBeenUndefined() const;

bool IsUndefined() const;

bool IsLikelyUndefined() const;

bool HasBeenNull() const;

bool IsNull() const;

bool IsLikelyNull() const;

bool HasBeenBoolean() const;

bool IsBoolean() const;

bool IsLikelyBoolean() const;

bool HasBeenString() const;

bool IsString() const;

bool IsLikelyString() const;

bool IsNotString() const;

bool HasBeenSymbol() const;

bool IsSymbol() const;

bool IsLikelySymbol() const;

bool IsNotSymbol() const;

bool HasBeenPrimitive() const;

bool IsPrimitive() const;

bool IsLikelyPrimitive() const;

#if ENABLE\_NATIVE\_CODEGEN

// SIMD\_JS

bool IsSimd128() const;

bool IsSimd128(IRType type) const;

bool IsSimd128Float32x4() const;

bool IsSimd128Int32x4() const;

bool IsSimd128Int8x16() const;

bool IsSimd128Float64x2() const;

bool IsLikelySimd128() const;

bool IsLikelySimd128Float32x4() const;

bool IsLikelySimd128Int32x4() const;

bool IsLikelySimd128Int8x16() const;

bool IsLikelySimd128Float64x2() const;

#endif

bool HasBeenObject() const;

bool IsObject() const;

bool IsLikelyObject() const;

bool IsNotObject() const;

bool CanMergeToObject() const;

bool CanMergeToSpecificObjectType() const;

bool IsRegExp() const;

bool IsLikelyRegExp() const;

bool IsArray() const;

bool IsLikelyArray() const;

bool IsNotArray() const;

bool IsArrayOrObjectWithArray() const;

bool IsLikelyArrayOrObjectWithArray() const;

bool IsNotArrayOrObjectWithArray() const;

bool IsNativeArray() const;

bool IsLikelyNativeArray() const;

bool IsNotNativeArray() const;

bool IsNativeIntArray() const;

bool IsLikelyNativeIntArray() const;

bool IsNativeFloatArray() const;

bool IsLikelyNativeFloatArray() const;

bool IsTypedArray() const;

bool IsLikelyTypedArray() const;

bool IsTypedIntArray() const;

bool IsLikelyTypedIntArray() const;

bool IsTypedIntOrFloatArray() const;

bool IsOptimizedTypedArray() const;

bool IsLikelyOptimizedTypedArray() const;

bool IsLikelyOptimizedVirtualTypedArray() const;

bool IsAnyArrayWithNativeFloatValues() const;

bool IsLikelyAnyArrayWithNativeFloatValues() const;

bool IsAnyArray() const;

bool IsLikelyAnyArray() const;

bool IsAnyOptimizedArray() const;

bool IsLikelyAnyOptimizedArray() const;

bool IsLikelyAnyUnOptimizedArray() const;

// The following apply to object types only

public:

ObjectType GetObjectType() const;

private:

void SetObjectType(const ObjectType objectType);

public:

ValueType SetIsNotAnyOf(const ValueType other) const;

// The following apply to javascript array types only

public:

bool HasNoMissingValues() const;

ValueType SetHasNoMissingValues(const bool noMissingValues) const;

private:

bool HasNonInts() const;

bool HasNonFloats() const;

public:

bool HasIntElements() const;

bool HasFloatElements() const;

bool HasVarElements() const;

ValueType SetArrayTypeId(const Js::TypeId typeId) const;

public:

bool IsSubsetOf(const ValueType other, const bool isAggressiveIntTypeSpecEnabled, const bool isFloatSpecEnabled, const bool isArrayMissingValueCheckHoistEnabled, const bool isNativeArrayEnabled) const;

public:

ValueType ToDefinite() const;

ValueType ToLikelyUntaggedInt() const;

ValueType ToDefiniteNumber\_PreferFloat() const;

ValueType ToDefiniteAnyFloat() const;

ValueType ToDefiniteNumber() const;

ValueType ToDefiniteAnyNumber() const;

ValueType ToDefinitePrimitiveSubset() const;

ValueType ToDefiniteObject() const;

ValueType ToLikely() const;

ValueType ToArray() const;

private:

ValueType ToPrimitiveOrObject() const;

public:

ValueType Merge(const ValueType other) const;

private:

ValueType MergeWithObject(const ValueType other) const;

public:

ValueType Merge(const Js::Var var) const;

private:

static Bits TypeIdToBits[Js::TypeIds\_Limit];

static Bits VirtualTypeIdToBits[Js::TypeIds\_Limit];

static INT\_PTR TypeIdToVtable[Js::TypeIds\_Limit];

static ObjectType VirtualTypedArrayPair[(size\_t)ObjectType::Count];

static ObjectType MixedTypedArrayPair[(size\_t)ObjectType::Count];

static ObjectType MixedTypedToVirtualTypedArray[(size\_t)ObjectType::Count];

static ObjectType TypedArrayMergeMap[(size\_t)ObjectType::Count][(size\_t)ObjectType::Count];

static void InitializeTypeIdToBitsMap();

public:

static ValueType FromTypeId(const Js::TypeId typeId, bool useVirtual);

static INT\_PTR GetVirtualTypedArrayVtable(const Js::TypeId typeId);

static ValueType FromObject(Js::RecyclableObject \*const recyclableObject);

static ValueType FromObjectWithArray(Js::DynamicObject \*const object);

static ValueType FromObjectArray(Js::JavascriptArray \*const objectArray);

static ValueType FromArray(const ObjectType objectType, Js::JavascriptArray \*const array, const Js::TypeId arrayTypeId);

public:

bool operator ==(const ValueType other) const;

bool operator !=(const ValueType other) const;

uint GetHashCode() const;

public:

template<class F> static void MapInitialDefiniteValueTypesUntil(const F f);

template<class F> static void MapInitialIndefiniteValueTypesUntil(const F f);

template<class F> static void MapInitialValueTypesUntil(const F f);

private:

static const char \*const BitNames[];

static size\_t GetLowestBitIndex(const Bits b);

void ToVerboseString(char (&str)[VALUE\_TYPE\_MAX\_STRING\_SIZE]) const;

public:

void ToString(wchar (&str)[VALUE\_TYPE\_MAX\_STRING\_SIZE]) const;

void ToString(char (&str)[VALUE\_TYPE\_MAX\_STRING\_SIZE]) const;

void ToStringDebug(\_\_out\_ecount(strSize) char \*const str, const size\_t strSize) const;

static bool FromString(const wchar \*const str, ValueType \*valueType);

static bool FromString(const char \*const str, ValueType \*valueType);

public:

TSize GetRawData() const;

static ValueType FromRawData(const TSize rawData);

public:

bool IsVirtualTypedArrayPair(const ObjectType other) const;

bool IsLikelyMixedTypedArrayType() const;

bool IsMixedTypedArrayPair(const ValueType other) const;

ObjectType GetMixedTypedArrayObjectType() const;

ObjectType GetMixedToVirtualTypedArrayObjectType() const;

ValueType ChangeToMixedTypedArrayType() const;

#if DBG

private:

static void RunUnitTests();

#endif

private:

static void InstantiateForceInlinedMembers();

ENUM\_CLASS\_HELPER\_FRIENDS(Bits, TSize);

};

ENUM\_CLASS\_HELPERS(ValueType::Bits, ValueType::TSize);

struct ValueTypeComparer

{

static bool Equals(const ValueType t0, const ValueType t1);

static uint GetHashCode(const ValueType t);

};

template<>

struct DefaultComparer<ValueType> : public ValueTypeComparer

{

};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Template function definitions

template<class F> void ValueType::MapInitialDefiniteValueTypesUntil(const F f)

{

size\_t i = 0;

// Enumerate variations of Int types

if(f(GetTaggedInt(), i++))

return;

if(f(GetInt(true), i++))

return;

if(f(GetInt(false), i++))

return;

// Enumerate base value types except uninitialized, types covered above, and object types

const ValueType BaseValueTypes[] =

{

#define BASE\_VALUE\_TYPE(t, b) t,

#include "ValueTypes.h"

#undef BASE\_VALUE\_TYPE

};

for(size\_t j = 0; j < sizeof(BaseValueTypes) / sizeof(BaseValueTypes[0]); ++j)

{

if(BaseValueTypes[j] == Uninitialized || BaseValueTypes[j] == Int || BaseValueTypes[j] == UninitializedObject)

continue;

if(f(BaseValueTypes[j], i++))

return;

}

// Enumerate object types

for(ObjectType objectType = ObjectType::UninitializedObject; objectType < ObjectType::Count; ++objectType)

{

if(objectType != ObjectType::ObjectWithArray && objectType != ObjectType::Array)

{

if(f(GetObject(objectType), i++))

return;

continue;

}

// Permute all combinations of array information

for(Js::TypeId arrayTypeId = Js::TypeIds\_ArrayFirst;

arrayTypeId <= Js::TypeIds\_ArrayLast;

arrayTypeId = static\_cast<Js::TypeId>(arrayTypeId + 1))

{

if(objectType == ObjectType::ObjectWithArray && arrayTypeId != Js::TypeIds\_Array) // objects with native arrays are currently not supported

continue;

for(TSize noMissingValues = 0; noMissingValues < 2; ++noMissingValues)

{

const ValueType valueType(

ValueType::GetObject(objectType)

.SetHasNoMissingValues(!!noMissingValues)

.SetArrayTypeId(arrayTypeId));

if(f(valueType, i++))

return;

}

}

}

}

template<class F> void ValueType::MapInitialIndefiniteValueTypesUntil(const F f)

{

if(f(Uninitialized, 0))

return;

MapInitialDefiniteValueTypesUntil([&](const ValueType valueType, const size\_t i) -> bool

{

return f(valueType.ToLikely(), i + 1);

});

}

template<class F> void ValueType::MapInitialValueTypesUntil(const F f)

{

if(f(Uninitialized, 0))

return;

MapInitialDefiniteValueTypesUntil([&](const ValueType valueType, const size\_t i) -> bool

{

return f(valueType, i \* 2 + 1) || f(valueType.ToLikely(), i \* 2 + 2);

});

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Value type bits

#ifdef VALUE\_TYPE\_BIT

VALUE\_TYPE\_BIT(Likely, static\_cast<ValueType::TSize>(1 << 0 ))

VALUE\_TYPE\_BIT(Undefined, static\_cast<ValueType::TSize>(1 << 1 ))

VALUE\_TYPE\_BIT(Null, static\_cast<ValueType::TSize>(1 << 2 ))

VALUE\_TYPE\_BIT(CanBeTaggedValue, static\_cast<ValueType::TSize>(1 << 3 ))

VALUE\_TYPE\_BIT(Object, static\_cast<ValueType::TSize>(1 << 4 ))

#if !defined(VALUE\_TYPE\_OBJECT\_BIT\_INDEX)

#define VALUE\_TYPE\_OBJECT\_BIT\_INDEX static\_cast<ValueType::TSize>(4)

#endif

#if !defined(VALUE\_TYPE\_COMMON\_BIT\_COUNT)

#define VALUE\_TYPE\_COMMON\_BIT\_COUNT static\_cast<ValueType::TSize>(5)

#endif

// The following bits only apply when the Object bit is not set

VALUE\_TYPE\_BIT(Int, static\_cast<ValueType::TSize>(1 << 5 ))

VALUE\_TYPE\_BIT(IntCanBeUntagged, static\_cast<ValueType::TSize>(1 << 6 ))

VALUE\_TYPE\_BIT(IntIsLikelyUntagged, static\_cast<ValueType::TSize>(1 << 7 ))

VALUE\_TYPE\_BIT(Float, static\_cast<ValueType::TSize>(1 << 8 ))

VALUE\_TYPE\_BIT(Number, static\_cast<ValueType::TSize>(1 << 9 ))

VALUE\_TYPE\_BIT(Boolean, static\_cast<ValueType::TSize>(1 << 10 ))

VALUE\_TYPE\_BIT(String, static\_cast<ValueType::TSize>(1 << 11 ))

VALUE\_TYPE\_BIT(Symbol, static\_cast<ValueType::TSize>(1 << 12 ))

VALUE\_TYPE\_BIT(PrimitiveOrObject, static\_cast<ValueType::TSize>(1 << 13 ))

#if !defined(VALUE\_TYPE\_NONOBJECT\_BIT\_COUNT)

#define VALUE\_TYPE\_NONOBJECT\_BIT\_COUNT static\_cast<ValueType::TSize>(9)

#endif

// The following bits only apply when the Object bit is set

VALUE\_TYPE\_BIT(NoMissingValues, static\_cast<ValueType::TSize>(1 << 5 )) // array

VALUE\_TYPE\_BIT(NonInts, static\_cast<ValueType::TSize>(1 << 6 )) // array

VALUE\_TYPE\_BIT(NonFloats, static\_cast<ValueType::TSize>(1 << 7 )) // array

#if !defined(VALUE\_TYPE\_ARRAY\_BIT\_COUNT)

#define VALUE\_TYPE\_ARRAY\_BIT\_COUNT static\_cast<ValueType::TSize>(3)

#endif

#if !defined(VALUE\_TYPE\_OBJECT\_BIT\_COUNT)

#define VALUE\_TYPE\_OBJECT\_BIT\_COUNT static\_cast<ValueType::TSize>(3)

#endif

#endif

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Object types

#ifdef OBJECT\_TYPE

OBJECT\_TYPE(UninitializedObject )

OBJECT\_TYPE(Object )

// TODO (jedmiad): Add JsFunction to this list, once we have an optimization that can take advantage of it.

// Other objects

OBJECT\_TYPE(RegExp )

// Optimized arrays

OBJECT\_TYPE(ObjectWithArray )

OBJECT\_TYPE(Array )

// Typed arrays

// Typed arrays that are optimized by the JIT

OBJECT\_TYPE(Int8Array )

OBJECT\_TYPE(Uint8Array )

OBJECT\_TYPE(Uint8ClampedArray )

OBJECT\_TYPE(Int16Array )

OBJECT\_TYPE(Uint16Array )

OBJECT\_TYPE(Int32Array )

OBJECT\_TYPE(Uint32Array )

OBJECT\_TYPE(Float32Array )

OBJECT\_TYPE(Float64Array )

// Virtual Arrays

OBJECT\_TYPE(Int8VirtualArray)

OBJECT\_TYPE(Uint8VirtualArray)

OBJECT\_TYPE(Uint8ClampedVirtualArray)

OBJECT\_TYPE(Int16VirtualArray)

OBJECT\_TYPE(Uint16VirtualArray)

OBJECT\_TYPE(Int32VirtualArray)

OBJECT\_TYPE(Uint32VirtualArray)

OBJECT\_TYPE(Float32VirtualArray)

OBJECT\_TYPE(Float64VirtualArray)

//Mixed Arrays

OBJECT\_TYPE(Int8MixedArray)

OBJECT\_TYPE(Uint8MixedArray)

OBJECT\_TYPE(Uint8ClampedMixedArray)

OBJECT\_TYPE(Int16MixedArray)

OBJECT\_TYPE(Uint16MixedArray)

OBJECT\_TYPE(Int32MixedArray)

OBJECT\_TYPE(Uint32MixedArray)

OBJECT\_TYPE(Float32MixedArray)

OBJECT\_TYPE(Float64MixedArray)

// Typed arrays that are not optimized by the JIT

OBJECT\_TYPE(Int64Array)

OBJECT\_TYPE(Uint64Array)

OBJECT\_TYPE(BoolArray)

OBJECT\_TYPE(CharArray)

// SIMD\_JS

// Only Simd128 sub-types. Currently no need to track top Simd128 type

OBJECT\_TYPE(Simd128Float32x4 )

OBJECT\_TYPE(Simd128Int32x4 )

OBJECT\_TYPE(Simd128Int8x16 )

OBJECT\_TYPE(Simd128Float64x2 )

OBJECT\_TYPE(Count)

#endif

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Base value types

#ifdef BASE\_VALUE\_TYPE

BASE\_VALUE\_TYPE(Uninitialized, Bits::Likely | Bits::CanBeTaggedValue )

BASE\_VALUE\_TYPE(Int, Bits::Int | Bits::CanBeTaggedValue )

BASE\_VALUE\_TYPE(Float, Bits::Float | Bits::CanBeTaggedValue )

BASE\_VALUE\_TYPE(Number, Bits::Number | Bits::CanBeTaggedValue )

BASE\_VALUE\_TYPE(Undefined, Bits::Undefined )

BASE\_VALUE\_TYPE(Null, Bits::Null )

BASE\_VALUE\_TYPE(Boolean, Bits::Boolean )

BASE\_VALUE\_TYPE(String, Bits::String )

BASE\_VALUE\_TYPE(Symbol, Bits::Symbol )

BASE\_VALUE\_TYPE(UninitializedObject, Bits::Object )

BASE\_VALUE\_TYPE(PrimitiveOrObject, Bits::PrimitiveOrObject )

#endif

;-------------------------------------------------------------------------------------------------------

; Copyright (C) Microsoft. All rights reserved.

; Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

;-------------------------------------------------------------------------------------------------------

include ksamd64.inc

\_TEXT SEGMENT

ifdef \_CONTROL\_FLOW\_GUARD

extrn \_\_guard\_check\_icall\_fptr:QWORD

endif

ifdef \_ENABLE\_DYNAMIC\_THUNKS

;;============================================================================================================

;; InterpreterStackFrame::DelayDynamicInterpreterThunk

;;============================================================================================================

;; JavascriptMethod InterpreterStackFrame::EnsureDynamicInterpreterThunk(ScriptFunction \* function)

extrn ?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z : PROC

;; Var InterpreterStackFrame::DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

align 16

?DelayDynamicInterpreterThunk@InterpreterStackFrame@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ PROC FRAME

;; save volatile registers

mov qword ptr [rsp + 8h], rcx

mov qword ptr [rsp + 10h], rdx

mov qword ptr [rsp + 18h], r8

mov qword ptr [rsp + 20h], r9

push rbp

.pushreg rbp

lea rbp, [rsp]

.setframe rbp, 0

.endprolog

sub rsp, 20h ;allocate stack space for the callee params(min 4 slots is mandate)

call ?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z

ifdef \_CONTROL\_FLOW\_GUARD

mov rcx, rax ; \_\_guard\_check\_icall\_fptr requires the call target in rcx.

call [\_\_guard\_check\_icall\_fptr] ; verify that the call target is valid

mov rax, rcx ;restore call target

endif

add rsp, 20h ;de-allocate stack space for the callee params(min 4 slots is mandate)

;;EPILOGUE starts here

lea rsp, [rbp]

pop rbp

;; restore volatile registers

mov rcx, qword ptr [rsp + 8h]

mov rdx, qword ptr [rsp + 10h]

mov r8, qword ptr [rsp + 18h]

mov r9, qword ptr [rsp + 20h]

rex\_jmp\_reg rax

?DelayDynamicInterpreterThunk@InterpreterStackFrame@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ ENDP

;;============================================================================================================

;; InterpreterStackFrame::AsmJsDelayDynamicInterpreterThunk

;;============================================================================================================

;; JavascriptMethod InterpreterStackFrame::EnsureDynamicInterpreterThunk(ScriptFunction \* function)

extrn ?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z : PROC

;; Var InterpreterStackFrame::AsmJsDelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

align 16

?AsmJsDelayDynamicInterpreterThunk@InterpreterStackFrame@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ PROC FRAME

;; save volatile registers

mov qword ptr [rsp + 8h], rcx

mov qword ptr [rsp + 10h], rdx

mov qword ptr [rsp + 18h], r8

mov qword ptr [rsp + 20h], r9

push rbp

.pushreg rbp

lea rbp, [rsp]

.setframe rbp, 0

.endprolog

sub rsp, 60h

; spill potential floating point arguments to stack

movaps xmmword ptr [rsp + 30h], xmm1

movaps xmmword ptr [rsp + 40h], xmm2

movaps xmmword ptr [rsp + 50h], xmm3

ifdef \_CONTROL\_FLOW\_GUARD

call ?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z

mov rcx, rax ; \_\_guard\_check\_icall\_fptr requires the call target in rcx.

call [\_\_guard\_check\_icall\_fptr] ; verify that the call target is valid

mov rax, rcx ;restore call target

else

call ?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z

endif

; restore potential floating point arguments from stack

movaps xmm1, xmmword ptr [rsp + 30h]

movaps xmm2, xmmword ptr [rsp + 40h]

movaps xmm3, xmmword ptr [rsp + 50h]

add rsp, 60h

;;EPILOGUE starts here

lea rsp, [rbp]

pop rbp

;; restore volatile registers

mov rcx, qword ptr [rsp + 8h]

mov rdx, qword ptr [rsp + 10h]

mov r8, qword ptr [rsp + 18h]

mov r9, qword ptr [rsp + 20h]

rex\_jmp\_reg rax

?AsmJsDelayDynamicInterpreterThunk@InterpreterStackFrame@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ ENDP

;;============================================================================================================

;; DynamicProfileInfo::EnsureDynamicProfileInfoThunk

;;============================================================================================================

;; JavascriptMethod DynamicProfileInfo::EnsureDynamicProfileInfo(ScriptFunction \* function)

extrn ?EnsureDynamicProfileInfo@DynamicProfileInfo@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z : PROC

;; Var DynamicProfileInfo::EnsureDynamicProfileInfoThunk(RecyclableObject\* function, CallInfo callInfo, ...)

align 16

?EnsureDynamicProfileInfoThunk@DynamicProfileInfo@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ PROC FRAME

;; save volatile registers

mov qword ptr [rsp + 8h], rcx

mov qword ptr [rsp + 10h], rdx

mov qword ptr [rsp + 18h], r8

mov qword ptr [rsp + 20h], r9

push rbp

.pushreg rbp

lea rbp, [rsp]

.setframe rbp, 0

.endprolog

sub rsp, 20h

call ?EnsureDynamicProfileInfo@DynamicProfileInfo@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z

ifdef \_CONTROL\_FLOW\_GUARD

mov rcx, rax ; \_\_guard\_check\_icall\_fptr requires the call target in rcx.

call [\_\_guard\_check\_icall\_fptr] ; verify that the call target is valid

mov rax, rcx ;restore call target

endif

add rsp, 20h

lea rsp, [rbp]

pop rbp

;; restore volatile registers

mov rcx, qword ptr [rsp + 8h]

mov rdx, qword ptr [rsp + 10h]

mov r8, qword ptr [rsp + 18h]

mov r9, qword ptr [rsp + 20h]

rex\_jmp\_reg rax

?EnsureDynamicProfileInfoThunk@DynamicProfileInfo@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ ENDP

endif ;; \_ENABLE\_DYNAMIC\_THUNKS

;;============================================================================================================

;; ScriptContext::ProfileModeDeferredParsingThunk

;;============================================================================================================

;; Js::JavascriptMethod ScriptContext::ProfileModeDeferredParse(ScriptFunction \*function)

extrn ?ProfileModeDeferredParse@ScriptContext@Js@@SAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAPEAVScriptFunction@2@@Z : PROC

;; Var ScriptContext::ProfileModeDeferredParsingThunk(RecyclableObject\* function, CallInfo callInfo, ...)

align 16

?ProfileModeDeferredParsingThunk@ScriptContext@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ PROC FRAME

;; save volatile registers

mov qword ptr [rsp + 8h], rcx

mov qword ptr [rsp + 10h], rdx

mov qword ptr [rsp + 18h], r8

mov qword ptr [rsp + 20h], r9

push rbp

.pushreg rbp

lea rbp, [rsp]

.setframe rbp, 0

.endprolog

sub rsp, 20h

lea rcx, [rsp + 30h]

call ?ProfileModeDeferredParse@ScriptContext@Js@@SAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAPEAVScriptFunction@2@@Z

ifdef \_CONTROL\_FLOW\_GUARD

mov rcx, rax ; \_\_guard\_check\_icall\_fptr requires the call target in rcx.

call [\_\_guard\_check\_icall\_fptr] ; verify that the call target is valid

mov rax, rcx ;restore call target

endif

add rsp, 20h

lea rsp, [rbp]

pop rbp

;; restore volatile registers

mov rcx, qword ptr [rsp + 8h]

mov rdx, qword ptr [rsp + 10h]

mov r8, qword ptr [rsp + 18h]

mov r9, qword ptr [rsp + 20h]

rex\_jmp\_reg rax

?ProfileModeDeferredParsingThunk@ScriptContext@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ ENDP

;;============================================================================================================

;;============================================================================================================

;; ScriptContext::ProfileModeDeferredDeserializeThunk

;;============================================================================================================

;; Js::JavascriptMethod ScriptContext::ProfileModeDeferredDeserialize(ScriptFunction \*function)

extrn ?ProfileModeDeferredDeserialize@ScriptContext@Js@@SAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z : PROC

;; Var ScriptContext::ProfileModeDeferredDeserializeThunk(RecyclableObject\* function, CallInfo callInfo, ...)

align 16

?ProfileModeDeferredDeserializeThunk@ScriptContext@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ PROC FRAME

;; save volatile registers

mov qword ptr [rsp + 8h], rcx

mov qword ptr [rsp + 10h], rdx

mov qword ptr [rsp + 18h], r8

mov qword ptr [rsp + 20h], r9

push rbp

.pushreg rbp

lea rbp, [rsp]

.setframe rbp, 0

.endprolog

sub rsp, 20h

call ?ProfileModeDeferredDeserialize@ScriptContext@Js@@SAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z

ifdef \_CONTROL\_FLOW\_GUARD

mov rcx, rax ; \_\_guard\_check\_icall\_fptr requires the call target in rcx.

call [\_\_guard\_check\_icall\_fptr] ; verify that the call target is valid

mov rax, rcx ;restore call target

endif

add rsp, 20h

lea rsp, [rbp]

pop rbp

;; restore volatile registers

mov rcx, qword ptr [rsp + 8h]

mov rdx, qword ptr [rsp + 10h]

mov r8, qword ptr [rsp + 18h]

mov r9, qword ptr [rsp + 20h]

rex\_jmp\_reg rax

?ProfileModeDeferredDeserializeThunk@ScriptContext@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ ENDP

ifdef \_ENABLE\_DYNAMIC\_THUNKS

;;============================================================================================================

;; Js::AsmJsInterpreterThunk

;;============================================================================================================

extern ?GetAsmJsInterpreterEntryPoint@InterpreterStackFrame@Js@@SAPEAXPEAUAsmJsCallStackLayout@2@@Z : PROC

; AsmJsInterpreterThunk (AsmJsCallStackLayout \*function, ...)

align 16

?InterpreterAsmThunk@InterpreterStackFrame@Js@@SAXPEAUAsmJsCallStackLayout@2@@Z PROC FRAME

; spill arguments

mov qword ptr [rsp + 8h], rcx

mov qword ptr [rsp + 10h], rdx

mov qword ptr [rsp + 18h], r8

mov qword ptr [rsp + 20h], r9

push rbp

.pushreg rbp

lea rbp, [rsp]

.setframe rbp, 0

.endprolog

sub rsp, 60h

; spill potential floating point arguments to stack

movups xmmword ptr [rsp + 30h], xmm1

movups xmmword ptr [rsp + 40h], xmm2

movups xmmword ptr [rsp + 50h], xmm3

; get correct interpreter entrypoint

call ?GetAsmJsInterpreterEntryPoint@InterpreterStackFrame@Js@@SAPEAXPEAUAsmJsCallStackLayout@2@@Z

ifdef \_CONTROL\_FLOW\_GUARD

mov rcx, rax ; \_\_guard\_check\_icall\_fptr requires the call target in rcx.

call [\_\_guard\_check\_icall\_fptr] ; verify that the call target is valid

mov rax, rcx ;restore call target

endif

mov rcx, qword ptr [rsp + 70h] ; restore rcx

call rax ; call appropriate template

lea rsp, [rbp]

pop rbp

ret

?InterpreterAsmThunk@InterpreterStackFrame@Js@@SAXPEAUAsmJsCallStackLayout@2@@Z ENDP

;;============================================================================================================

;; Js::AsmJsExternalEntryPoint

;;============================================================================================================

extrn ?GetStackSizeForAsmJsUnboxing@Js@@YAHPEAVScriptFunction@1@@Z: PROC

extrn ?UnboxAsmJsArguments@Js@@YAPEAXPEAVScriptFunction@1@PEAPEAXPEADUCallInfo@1@@Z : PROC

; extrn ?BoxAsmJsReturnValue@Js@@YAPEAXPEAVScriptFunction@1@HNM@Z : PROC

extrn ?BoxAsmJsReturnValue@Js@@YAPEAXPEAVScriptFunction@1@HNMT\_\_m128@@@Z : PROC

extrn ?GetArgsSizesArray@Js@@YAPEAIPEAVScriptFunction@1@@Z : PROC

;; int Js::AsmJsExternalEntryPoint(RecyclableObject\* entryObject, CallInfo callInfo, ...);

align 16

?AsmJsExternalEntryPoint@Js@@YAPEAXPEAVRecyclableObject@1@UCallInfo@1@ZZ PROC FRAME

mov qword ptr [rsp + 8h], rcx

mov qword ptr [rsp + 10h], rdx

mov qword ptr [rsp + 18h], r8

mov qword ptr [rsp + 20h], r9

push rbp

.pushreg rbp

lea rbp, [rsp]

.setframe rbp, 0

.endprolog

sub rsp, 40h

mov [rsp + 28h], rsi

mov [rsp + 30h], rdi

mov rsi, rcx ; store entryObject in rsi

mov rdi, rdx ; store callInfo in rdi

; allocate stack space for unboxed values

; int GetStackSizeForAsmJsUnboxing(ScriptFunction\* func)

call ?GetStackSizeForAsmJsUnboxing@Js@@YAHPEAVScriptFunction@1@@Z

mov r9, rdi

mov rdx, rsp ; orig stack pointer is arg for the unboxing helper

mov rdi, rdx ; save orig stack pointer, so that we can add it back later

add rdx, 68h ; account for the changes we have already made to rsp

sub rsp, rax ; allocate additional stack space for args

; UnboxAsmJsArguments(func, origArgsLoc, argDst, callInfo)

mov rcx, rsi

mov r8, rsp

sub rsp, 20h ; so stack space for unboxing function isn't same as where it is unboxing into. allocate args spill space for unboxing function.

; unboxing function also does stack probe

call ?UnboxAsmJsArguments@Js@@YAPEAXPEAVScriptFunction@1@PEAPEAXPEADUCallInfo@1@@Z

; rax = target function address

ifdef \_CONTROL\_FLOW\_GUARD

mov rcx, rax

; it is guaranteed that icall check will preserve rcx

call [\_\_guard\_check\_icall\_fptr]

mov rax, rcx ; restore entry point to rax

endif

add rsp, 20h

; move first 4 arguments into registers.

; don't know types other than arg0 (which is ScriptFunction \*), so put in both xmm and general purpose registers

mov rcx, rsi

; int GetArgsSizesArray(ScriptFunction\* func)

; get args sizes of target asmjs function

; rcx has ScriptFunction\*

push rdi

push rax

push rcx

sub rsp, 20h

call ?GetArgsSizesArray@Js@@YAPEAIPEAVScriptFunction@1@@Z

mov rdi, rax

add rsp, 20h

pop rcx

pop rax

; Move 3 args to regs per convention. rcx already has first arg: ScriptFunction\*

push rsi

; rsi->unboxed args

lea rsi, [rsp + 18h] ; rsp + size of(rdi + rsi + ScriptFunction\*)

; rdi is arg size

cmp dword ptr [rdi], 10h

je SIMDArg2

mov rdx, [rsi]

movq xmm1, qword ptr [rsi]

add rsi, 8h

jmp Arg3

SIMDArg2:

movups xmm1, xmmword ptr[rsi]

add rsi, 10h

Arg3:

cmp dword ptr [rdi + 4h], 10h

je SIMDArg3

mov r8, [rsi]

movq xmm2, qword ptr [rsi]

add rsi, 8h

jmp Arg4

SIMDArg3:

movups xmm2, xmmword ptr[rsi]

add rsi, 10h

Arg4:

cmp dword ptr [rdi + 8h], 10h

je SIMDArg4

mov r9, [rsi]

movq xmm3, qword ptr [rsi]

jmp ArgsDone

SIMDArg4:

movups xmm3, xmmword ptr [rsi]

ArgsDone:

pop rsi

pop rdi

; call entry point

call rax

; Var BoxAsmJsReturnValue(ScriptFunction\* func, int intRetVal, double doubleRetVal, float floatRetVal)

mov rcx, rsi

mov edx, eax

movsd xmm2, xmm0

movss xmm3, xmm0

; store SIMD xmm value and pointer to it as argument to box function

sub rsp, 40h

movups [rsp + 30h], xmm0

lea rsi, [rsp + 30h]

mov qword ptr [rsp + 20h], rsi

call ?BoxAsmJsReturnValue@Js@@YAPEAXPEAVScriptFunction@1@HNMT\_\_m128@@@Z

mov rsp, rdi ; restore stack pointer

Epilogue:

mov rsi, [rsp + 28h]

mov rdi, [rsp + 30h]

lea rsp, [rbp]

pop rbp

ret

?AsmJsExternalEntryPoint@Js@@YAPEAXPEAVRecyclableObject@1@UCallInfo@1@ZZ ENDP

endif ;; \_ENABLE\_DYNAMIC\_THUNKS

;;============================================================================================================

\_TEXT ENDS

end

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#ifndef TEMP\_DISABLE\_ASMJS

namespace Js

{

#if DBG\_DUMP

FunctionBody\* AsmJsJitTemplate::Globals::CurrentEncodingFunction = nullptr;

#endif

void\* AsmJsJitTemplate::InitTemplateData()

{

\_\_debugbreak();

return nullptr;

}

void AsmJsJitTemplate::FreeTemplateData(void\* userData)

{

\_\_debugbreak();

}

}

#endif

;-------------------------------------------------------------------------------------------------------

; Copyright (C) Microsoft. All rights reserved.

; Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

;-------------------------------------------------------------------------------------------------------

include ksamd64.inc

\_TEXT SEGMENT

ifdef \_CONTROL\_FLOW\_GUARD

extrn \_\_guard\_check\_icall\_fptr:QWORD

endif

extrn \_\_chkstk: PROC

;; [rsp + 8h] = arg0.

;; r9 = args size.

;; r8 = spill size.

;; rdx = original frame pointer.

;; rcx = target.

align 16

amd64\_CallWithFakeFrame PROC

;; Call \_\_chkstk to ensure the stack is extended properly. It expects size in rax.

mov rax, r8

add rax, r9

cmp rax, 1000h

jl chkstk\_done

call \_\_chkstk

chkstk\_done:

;; The stack walker uses this marker to skip this frame.

lea rax, amd64\_ReturnFromCallWithFakeFrame

mov [rsp+8h], rax

mov rax, [rsp + 28h]

push rbp

mov rbp, rdx

;; Frame spill size.

sub rsp, r8

;; Save callee-saved xmm registers

movapd xmmword ptr [rsp + 90h], xmm15

movapd xmmword ptr [rsp + 80h], xmm14

movapd xmmword ptr [rsp + 70h], xmm13

movapd xmmword ptr [rsp + 60h], xmm12

movapd xmmword ptr [rsp + 50h], xmm11

movapd xmmword ptr [rsp + 40h], xmm10

movapd xmmword ptr [rsp + 30h], xmm9

movapd xmmword ptr [rsp + 20h], xmm8

movapd xmmword ptr [rsp + 10h], xmm7

movapd xmmword ptr [rsp], xmm6

;; Save all callee saved registers.

push r15

push r14

push r13

push r12

push rdi

push rsi

push rbx

;; Frame args size.

sub rsp, r9

rex\_jmp\_reg rcx

amd64\_CallWithFakeFrame ENDP

;; r9 = args size.

;; r8 = spill size.

align 16

amd64\_ReturnFromCallWithFakeFrame PROC

add rsp, r9

pop rbx

pop rsi

pop rdi

pop r12

pop r13

pop r14

pop r15

;; Restore callee-saved xmm registers

movapd xmm6, xmmword ptr [rsp]

movapd xmm7, xmmword ptr [rsp + 10h]

movapd xmm8, xmmword ptr [rsp + 20h]

movapd xmm9, xmmword ptr [rsp + 30h]

movapd xmm10, xmmword ptr [rsp + 40h]

movapd xmm11, xmmword ptr [rsp + 50h]

movapd xmm12, xmmword ptr [rsp + 60h]

movapd xmm13, xmmword ptr [rsp + 70h]

movapd xmm14, xmmword ptr [rsp + 80h]

movapd xmm15, xmmword ptr [rsp + 90h]

add rsp, r8

pop rbp

;; Return to the real caller.

ret

amd64\_ReturnFromCallWithFakeFrame ENDP

\_TEXT ENDS

end

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if !defined(\_M\_X64)

#error Amd64StackFrame is not supported on this architecture.

#endif

Js::Amd64StackFrame::Amd64StackFrame()

: scriptContext(nullptr),

imageBase(0),

functionEntry(nullptr),

currentContext(nullptr),

hasCallerContext(false),

callerContext(nullptr),

addressOfCodeAddr(nullptr)

{

}

Js::Amd64StackFrame::~Amd64StackFrame()

{

if (currentContext)

{

scriptContext->GetThreadContext()->GetAmd64ContextsManager()->Release(currentContext);

}

}

// InitializeByReturnAddress.

// Parameters:

// unwindToAddress: specifies the address we need to unwind the stack before any walks can be done.

// RtlVirtualUnwind API requires that all unwinds are done within same call stack --

// this means that we can't capture context here, go 2 frames back (this ctor -> JavascriptStackWalker ctor),

// then create new frames (e.g. for Next()) and unwind stack in them.

bool Js::Amd64StackFrame::InitializeByReturnAddress(void \*returnAddress, ScriptContext\* scriptContext)

{

CONTEXT\* pair = scriptContext->GetThreadContext()->GetAmd64ContextsManager()->Allocate();

this->scriptContext = scriptContext;

this->currentContext = pair;

this->callerContext = pair + 1;

this->stackCheckCodeHeight =

scriptContext->GetThreadContext()->DoInterruptProbe() ? stackCheckCodeHeightWithInterruptProbe

: scriptContext->GetThreadContext()->GetIsThreadBound() ? stackCheckCodeHeightThreadBound

: stackCheckCodeHeightNotThreadBound;

// this is the context for the current function

RtlCaptureContext(currentContext);

OnCurrentContextUpdated();

// Unwind stack to the frame where RIP is the returnAddress

bool found = SkipToFrame(returnAddress);

if (!found)

{

AssertMsg(FALSE, "Amd64StackFrame: can't initialize: can't unwind the stack to specified unwindToAddress.");

RtlCaptureContext(currentContext); // Restore trashed context, the best we can do.

}

return found;

}

bool Js::Amd64StackFrame::Next()

{

if (hasCallerContext)

{

\*currentContext = \*callerContext;

OnCurrentContextUpdated();

return true;

}

if (JavascriptFunction::IsNativeAddress(this->scriptContext, (void\*)this->currentContext->Rip))

{

this->addressOfCodeAddr = this->GetAddressOfReturnAddress(true /\*isCurrentContextNative\*/, false /\*shouldCheckForNativeAddr\*/);

if (NextFromNativeAddress(this->currentContext))

{

OnCurrentContextUpdated();

return true;

}

return false;

}

EnsureFunctionEntry();

this->addressOfCodeAddr = this->GetAddressOfReturnAddress();

if (Next(currentContext, imageBase, functionEntry))

{

OnCurrentContextUpdated();

return true;

}

return false;

}

VOID \*Js::Amd64StackFrame::GetInstructionPointer()

{

return (VOID \*)currentContext->Rip;

}

void \*Js::Amd64StackFrame::GetFrame() const

{

return (void \*)currentContext->Rbp;

}

VOID \*\*Js::Amd64StackFrame::GetArgv(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

if (EnsureCallerContext(isCurrentContextNative || (shouldCheckForNativeAddr && JavascriptFunction::IsNativeAddress(this->scriptContext, (void\*)this->currentContext->Rip))))

{

return (VOID \*\*)callerContext->Rsp;

}

return nullptr;

}

VOID \*Js::Amd64StackFrame::GetReturnAddress(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

if (EnsureCallerContext(isCurrentContextNative || (shouldCheckForNativeAddr && JavascriptFunction::IsNativeAddress(this->scriptContext, (void\*)this->currentContext->Rip))))

{

return (VOID \*)callerContext->Rip;

}

return nullptr;

}

void \*Js::Amd64StackFrame::GetAddressOfReturnAddress(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

if (EnsureCallerContext(isCurrentContextNative || (shouldCheckForNativeAddr && JavascriptFunction::IsNativeAddress(this->scriptContext, (void\*)this->currentContext->Rip))))

{

return (void\*)((VOID \*\*)callerContext->Rsp - 1);

}

return nullptr;

}

bool Js::Amd64StackFrame::Next(CONTEXT \*context, ULONG64 imageBase, RUNTIME\_FUNCTION \*functionEntry)

{

Assert(context);

VOID \*handlerData = nullptr;

ULONG64 establisherFrame = 0;

if (!context->Rip)

return false;

if (functionEntry)

{

RtlVirtualUnwind(0 /\* UNW\_FLAG\_NHANDLER \*/,

imageBase,

context->Rip,

functionEntry,

context,

&handlerData,

&establisherFrame,

nullptr);

}

else

{

// Leaf frames are not listed in the PDATA section because they

// don't use the stack.

// Manually crawl to the next frame.

context->Rip = \*((DWORD64 \*)context->Rsp);

context->Rsp += 8;

}

return true;

}

bool

Js::Amd64StackFrame::NextFromNativeAddress(CONTEXT \* context)

{

if (!context->Rip)

{

return false;

}

//Restore Rip, Rsp and Rbp

// Rip - to check if the context is in native address range

// - to check if the current frame is javascript frame

// - to do virtual unwind

// - to pass to RtlLookupFunctionEntry if the next frame is not native

//

// Rsp - To easily get to the arguments passed in

//

// Rbp - to walk to the next frame

context->Rip = \*((DWORD64\*)context->Rbp + 1);

context->Rsp = (DWORD64)((DWORD64\*)context->Rbp + 2);

context->Rbp = \*((DWORD64\*)context->Rbp);

return true;

}

bool

Js::Amd64StackFrame::SkipToFrame(void \* returnAddress)

{

bool found = false;

while (Next())

{

if (((PVOID)currentContext->Rip) == returnAddress)

{

found = true;

break;

}

else if (!ThreadContext::IsOnStack((PVOID)currentContext->Rsp))

{

AssertMsg(FALSE, "Amd64StackFrame: while doing initial unwind SP got out of stack.");

break;

}

}

return found;

}

bool

Js::Amd64StackFrame::IsInStackCheckCode(void \*entry, void \*codeAddr, size\_t stackCheckCodeHeight)

{

return ((size\_t(codeAddr) - size\_t(entry)) <= stackCheckCodeHeight);

}

Js::Amd64ContextsManager::Amd64ContextsManager()

: curIndex(GENERAL\_CONTEXT)

{

}

\_Ret\_writes\_(CONTEXT\_PAIR\_COUNT)

CONTEXT\* Js::Amd64ContextsManager::InternalGet(

\_In\_range\_(GENERAL\_CONTEXT, OOM\_CONTEXT) ContextsIndex index)

{

Assert(index < NUM\_CONTEXTS);

return &contexts[CONTEXT\_PAIR\_COUNT \* index];

}

\_Ret\_writes\_(CONTEXT\_PAIR\_COUNT)

CONTEXT\* Js::Amd64ContextsManager::Allocate()

{

CONTEXT\* pair = NULL;

switch(curIndex)

{

case GENERAL\_CONTEXT: //0

pair = InternalGet(curIndex++);

Assert(curIndex == OOM\_CONTEXT); // Next available is OOM\_CONTEXT

break;

case OOM\_CONTEXT: //1

pair = HeapNewNoThrowArray(CONTEXT, CONTEXT\_PAIR\_COUNT);

if (!pair)

{

pair = InternalGet(curIndex++);

Assert(curIndex == NUM\_CONTEXTS); // Used up all stock contexts

}

break;

default:

AssertMsg(false, "Unexpected usage of JavascriptStackWalker. We run out of CONTEXTs on amd64.");

Amd64StackWalkerOutOfContexts\_fatal\_error((ULONG\_PTR)this);

}

AnalysisAssert(pair);

memset(pair, 0, sizeof(CONTEXT) \* CONTEXT\_PAIR\_COUNT);

return pair;

}

void Js::Amd64ContextsManager::Release(\_In\_ CONTEXT\* contexts)

{

switch(curIndex)

{

case GENERAL\_CONTEXT:

AssertMsg(false, "Unexpected release of CONTEXTs. No contexts allocated.");

break;

case OOM\_CONTEXT:

if (contexts != InternalGet(curIndex - 1))

{

HeapDeleteArray(CONTEXT\_PAIR\_COUNT, contexts);

}

else

{

--curIndex;

Assert(curIndex == GENERAL\_CONTEXT); // GENERAL\_CONTEXT is now available

}

break;

case NUM\_CONTEXTS:

AssertMsg(contexts == InternalGet(curIndex - 1), "Invalid CONTEXT releasing sequence. Expect to release stock contexts for OOM.");

--curIndex;

Assert(curIndex == OOM\_CONTEXT); // OOM\_CONTEXT is now available

break;

default:

Assert(false); // Invalid state

break;

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

/\*

\* Stackwalking on x86-64:

\* ----------------------

\*

\* On x86 the EBP register always points to the current stack frame, which, at

\* offset 0 contains a pointer to its caller's stack frame. Walking the stack

\* is accomplished by walking this linked list. Currently VC++ does not build

\* RBP frames on x86-64. However, the format of a function's stack frame is fairly

\* restricted. This and the fact that each function has metadata makes a stack walk

\* possible.

\*

\* The x86-64 ABI (that VC++ uses) is an x86 "fastcall" like calling convention

\* that uses RCX, RDX, R8 and R9 to pass the first 4 QWORD parameters with stack backing

\* for those registers. The caller is responsible for allocating space for parameters to

\* the callee and \*always\* allocates 4 extra QWORDs that the callee uses to "home" the

\* argument registers (typically). Home addresses are required for the register arguments

\* so a contiguous area is available in case the callee uses a va\_list. However, the callee

\* is not required to save the value in the registers into these slots and generally does

\* not unless the compiler deems it necessary. The caller always cleans the stack.

\*

\* All non-leaf functions (functions that either alloca / call other functions / use exception

\* handling) are annotated with data that describes how non-volatile registers

\* can be restored (say during a stack unwind). This data structure lives in the PDATA section

\* of a PE image. It describes the address limits of the prolog which

\* - saves argument registers in their home addresses

\* - pushes non-volatile registers used by the function on the stack

\* - allocates locals

\* - establishes a frame pointer (usually only if the function uses alloca)

\* The epilog trims the stack by reversing the effects of the prolog. Epilogs follow a

\* strict set of rules so that the operating system can scan a code stream to identify one.

\* This eliminates the need for more metadata. Epilogs are usually of the form

\*

\* add RSP, stack\_frame\_fixed\_size

\* pop Rn

\* ...

\* ret

\*

\* Since there is no linked list on the stack to follow, the x86-64 stack walker

\* (the platform specific portions of which are implemented in Amd64StackFrame and Javascript

\* specific portions in JavascriptStackWalker) uses the above mentioned metadata to

\* "unwind" the stack to get to a caller from its callee. Since we don't really want

\* to unwind the stack, we use the RtlVirtualUnwind API that does the unwind using a copy

\* of portions of the real stack using an instance of CONTEXT as the register file.

\*

\* The Javascript stack walker needs to find out what the return address, argv is for a given

\* stack frame.

\* - return address: to match against one in a script context that serves to identify a

\* stack frame where a stack walk should start (frameIdOfScriptExitFunction)

\* or terminate (returnAddrOfScriptEntryFunction).

\* - argv: used to implement caller args, find the line number (from the function object) when

\* an exception is thrown etc.

\*

\* During stack unwind, the return address for a given stack frame is the address RIP is

\* pointing to when we unwind to its caller. Similarly argv is the address that RSP is pointing

\* to when we unwind to its caller. We make sure that the arguments passed in registers are

\* homed by making use of the /homeparams switch.

\*

\* A "JavascriptFrame" is a stack frame established either by JITted script code or by the

\* interpreter whose arguments are always:

\* JavascriptFunction \* <-- instance of the javascript function object that this frame

\* corresponds to.

\* CallInfo

\* this <-- first script argument (hidden "this" pointer).

\* ... <-- rest of the script arguments.

\*/

namespace Js {

class ScriptContext;

class Amd64StackFrame {

public:

Amd64StackFrame();

~Amd64StackFrame();

bool InitializeByFrameId(void \* returnAddress, ScriptContext\* scriptContext) { return InitializeByReturnAddress(returnAddress, scriptContext); }

bool InitializeByReturnAddress(void \* returnAddress, ScriptContext\* scriptContext);

bool Next();

void \*GetInstructionPointer();

void \*\*GetArgv(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

void \*GetReturnAddress(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

void \*GetAddressOfReturnAddress(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

void \*GetAddressOfInstructionPointer() { return this->addressOfCodeAddr; }

bool SkipToFrame(void \* returnAddress);

void \*GetFrame() const;

size\_t GetStackCheckCodeHeight() { return this->stackCheckCodeHeight; }

static bool IsInStackCheckCode(void \*entry, void \* codeAddr, size\_t stackCheckCodeHeight);

private:

void\* addressOfCodeAddr;

ScriptContext \*scriptContext;

ULONG64 imageBase;

RUNTIME\_FUNCTION \*functionEntry;

CONTEXT \*currentContext;

// We store the context of the caller the first time we retrieve it so that

// consecutive operations that need the caller context don't have to retrieve

// it again. The callerContext is only valid if hasCallerContext is true. When

// currentContext is changed, callerContext must be invalidated by calling

// OnCurrentContextUpdated().

bool hasCallerContext;

CONTEXT \*callerContext;

size\_t stackCheckCodeHeight;

\_\_inline void EnsureFunctionEntry();

\_\_inline bool EnsureCallerContext(bool isCurrentContextNative);

\_\_inline void OnCurrentContextUpdated();

static bool NextFromNativeAddress(CONTEXT \* context);

static bool Next(CONTEXT \*context, ULONG64 imageBase, RUNTIME\_FUNCTION \*runtimeFunction);

static const size\_t stackCheckCodeHeightThreadBound = StackFrameConstants::StackCheckCodeHeightThreadBound;

static const size\_t stackCheckCodeHeightNotThreadBound = StackFrameConstants::StackCheckCodeHeightNotThreadBound;

static const size\_t stackCheckCodeHeightWithInterruptProbe = StackFrameConstants::StackCheckCodeHeightWithInterruptProbe;

};

class Amd64ContextsManager

{

private:

static const int CONTEXT\_PAIR\_COUNT = 2;

enum

{

GENERAL\_CONTEXT = 0,

OOM\_CONTEXT = 1,

NUM\_CONTEXTS = 2

};

typedef int ContextsIndex;

CONTEXT contexts[CONTEXT\_PAIR\_COUNT \* NUM\_CONTEXTS];

\_Field\_range\_(GENERAL\_CONTEXT, NUM\_CONTEXTS)

ContextsIndex curIndex;

\_Ret\_writes\_(CONTEXT\_PAIR\_COUNT) CONTEXT\* InternalGet(

\_In\_range\_(GENERAL\_CONTEXT, OOM\_CONTEXT) ContextsIndex index);

public:

Amd64ContextsManager();

private:

friend class Amd64StackFrame;

\_Ret\_writes\_(CONTEXT\_PAIR\_COUNT) CONTEXT\* Allocate();

void Release(\_In\_ CONTEXT\* contexts);

};

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

\_\_inline void Js::Amd64StackFrame::EnsureFunctionEntry()

{

if (!functionEntry)

{

functionEntry = RtlLookupFunctionEntry(currentContext->Rip, &imageBase, nullptr);

}

}

\_\_inline bool Js::Amd64StackFrame::EnsureCallerContext(bool isCurrentContextNative)

{

if (!hasCallerContext)

{

\*callerContext = \*currentContext;

if (isCurrentContextNative)

{

if (NextFromNativeAddress(callerContext))

{

hasCallerContext = true;

return true;

}

return false;

}

EnsureFunctionEntry();

if (Next(callerContext, imageBase, functionEntry))

{

hasCallerContext = true;

return true;

}

else

{

return false;

}

}

return true;

}

\_\_inline void Js::Amd64StackFrame::OnCurrentContextUpdated()

{

imageBase = 0;

functionEntry = nullptr;

hasCallerContext = false;

}

;-------------------------------------------------------------------------------------------------------

; Copyright (C) Microsoft. All rights reserved.

; Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

;-------------------------------------------------------------------------------------------------------

;

; arm\_CallEhFrame() and arm\_CallCatch() both thunk into jitted code at the

; start of an EH region. The purpose is to restore the frame pointer (r11)

; and locals pointer (r7) to the appropriate values for executing the parent

; function and to create a local frame that can be unwound using the parent

; function's pdata. The parent's frame looks like this:

;

;-------------------

; {r0-r3} -- homed parameters

; lr -- address from which parent was called

; r11 -- saved frame pointer, pointed to by current r11

; arg obj

; {r4-r10} -- non-volatile registers: all of them are saved

; {d8-d15} -- non-volatile double registers: all of them are saved

; locals area -- pointed to by r7

; pointer to non-volatile register area above

; stack args

;-------------------

;

; The reason for the "pointer to non-volatile register area" is to allow the

; unwinder to deallocate the locals area regardless of its size. So this thunk can skip

; the allocation of the locals area altogether, and unwinding still works.

; The unwind pseudo-codes for the above prolog look like:

;

; 1. Deallocate stack args (sp now points to "pointer to non-volatile register area")

; 2. Restore rN (rN now points to first saved register)

; 3. Copy rN to sp (sp now points to first saved register)

; 4. Restore {d8-d15} (non-volatile double registers restored)

; 5. Restore {r4-r10,r12} (non-volatile registers restored, sp points to saved r11)

; 6. Restore r11

; 7. Load lr into pc and deallocate remaining stack.

;

; The prologs for the assembly thunks allocate a frame that can be unwound by executing

; the above steps, although we don't allocate a locals area and don't know the size of the

; stack args. The caller doesn't return to this thunk; it executes its own epilog and

; returns to the caller of the thunk (one of the runtime try helpers).

; Windows version

OPT 2 ; disable listing

#include "ksarm.h"

OPT 1 ; re-enable listing

TTL Lib\Common\arm\arm\_CallEhFrame.asm

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

IMPORT \_\_guard\_check\_icall\_fptr

#endif

IMPORT \_\_chkstk

EXPORT arm\_CallEhFrame

TEXTAREA

NESTED\_ENTRY arm\_CallEhFrame

; Params:

; r0 -- thunk target

; r1 -- frame pointer

; r2 -- locals pointer

; r3 -- size of stack args area

; Home params and save registers

; Push r11 to create the equivalent of the stack nested function list (doesn't matter what is stored there)

; Push r12 to create the equivalent of the arguments object slot (doesn't matter what is stored there)

PROLOG\_PUSH {r0-r3}

PROLOG\_PUSH {r11,lr}

PROLOG\_PUSH {r4-r12}

PROLOG\_VPUSH {d8-d15}

; Save a pointer to the saved registers

PROLOG\_STACK\_SAVE r4

PROLOG\_PUSH r4

; Set up the frame pointer and locals pointer

mov r7,r2

mov r11,r1

; Allocate the arg out area, calling chkstk if necessary

cmp r3,4096

bge chkstk\_call

sub sp,r3

; Verify that the call target is valid

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

; we have used the r1-r3 info to set up the fake frame

; they aren't needed by the jitted code that we're going to thunk to

; so we don't preserve them across the \_\_guard\_check\_icall\_fptr call

mov r5, r0

mov32 r12, \_\_guard\_check\_icall\_fptr

ldr r12, [r12]

blx r12

mov r0, r5

#endif

; Thunk to the jitted code (and don't return)

bx r0

|chkstk\_call|

; Call chkstk, passing a DWORD count in r4

lsr r4,r3,#2

bl |\_\_chkstk|

; r4 is now the byte count to be allocated

sub sp,r4

; Thunk to the jitted code (and don't return)

bx r0

NESTED\_END arm\_CallEhFrame

; arm\_CallCatch() is similar to arm\_CallEhFrame() except that we also pass the catch object to the jitted code

EXPORT arm\_CallCatch

TEXTAREA

NESTED\_ENTRY arm\_CallCatch

; Params:

; r0 -- thunk target

; r1 -- frame pointer

; r2 -- locals pointer

; r3 -- size of stack args area

; [sp] -- exception object

; Home params and save registers

; Push r11 to create the equivalent of the stack nested function list (doesn't matter what is stored there)

; Push r12 to create the equivalent of the arguments object slot (doesn't matter what is stored there)

PROLOG\_PUSH {r0-r3}

PROLOG\_PUSH {r11,lr}

PROLOG\_PUSH {r4-r12}

PROLOG\_VPUSH {d8-d15}

; Save a pointer to the saved registers

PROLOG\_STACK\_SAVE r4

PROLOG\_PUSH r4

; Set up the frame pointer and locals pointer

mov r7,r2

mov r11,r1

; Load the exception object from [sp, 16 (homed params) + 44 (saved registers) + 64 (double registers) + 4 (saved SP) == 128]

ldr r1,[sp,#128]

; Allocate the arg out area, calling chkstk if necessary

cmp r3,4096

bge chkstk\_call\_catch

sub sp,r3

; Verify that the call target is valid

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

; we have used the r1-r3 info to set up the fake frame

; they aren't needed by the jitted code that we're going to thunk to

; so we don't preserve them across the \_\_guard\_check\_icall\_fptr call

mov r5, r0

mov32 r12, \_\_guard\_check\_icall\_fptr

ldr r12, [r12]

blx r12

mov r0, r5

#endif

; Thunk to the jitted code (and don't return)

bx r0

|chkstk\_call\_catch|

; Call chkstk, passing a DWORD count in r4

lsr r4,r3,#2

bl |\_\_chkstk|

; r4 is now the byte count to be allocated

sub sp,r4

; Thunk to the jitted code (and don't return)

bx r0

NESTED\_END arm\_CallCatch

END

;-------------------------------------------------------------------------------------------------------

; Copyright (C) Microsoft. All rights reserved.

; Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

;-------------------------------------------------------------------------------------------------------

OPT 2 ; disable listing

#include "ksarm.h"

OPT 1 ; re-enable listing

TTL Lib\Runtime\Language\arm\arm\_DelayDynamicInterpreterThunk.asm

;Var InterpreterStackFrame::DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

EXPORT |?DelayDynamicInterpreterThunk@InterpreterStackFrame@Js@@SAPAXPAVRecyclableObject@2@UCallInfo@2@ZZ|

;Var DynamicProfileInfo::EnsureDynamicProfileInfoThunk(RecyclableObject\* function, CallInfo callInfo, ...)

EXPORT |?EnsureDynamicProfileInfoThunk@DynamicProfileInfo@Js@@SAPAXPAVRecyclableObject@2@UCallInfo@2@ZZ|

; Var ScriptContext::ProfileModeDeferredParsingThunk(RecyclableObject\* function, CallInfo callInfo, ...)

EXPORT |?ProfileModeDeferredParsingThunk@ScriptContext@Js@@SAPAXPAVRecyclableObject@2@UCallInfo@2@ZZ|

;JavascriptMethod InterpreterStackFrame::EnsureDynamicInterpreterThunk(Js::ScriptFunction \* function)

IMPORT |?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APAXPAVRecyclableObject@2@UCallInfo@2@ZZPAVScriptFunction@2@@Z|

;JavascriptMethod DynamicProfileInfo::EnsureDynamicProfileInfoThunk(Js::ScriptFunction \* function)

IMPORT |?EnsureDynamicProfileInfo@DynamicProfileInfo@Js@@CAP6APAXPAVRecyclableObject@2@UCallInfo@2@ZZPAVScriptFunction@2@@Z|

;JavascriptMethod ScriptContext::ProfileModeDeferredParse(ScriptFunction \*\*function)

IMPORT |?ProfileModeDeferredParse@ScriptContext@Js@@SAP6APAXPAVRecyclableObject@2@UCallInfo@2@ZZPAPAVScriptFunction@2@@Z|

;JavascriptMethod ScriptContext::ProfileModeDeferredDeserialize(ScriptFunction \*function)

IMPORT |?ProfileModeDeferredDeserialize@ScriptContext@Js@@SAP6APAXPAVRecyclableObject@2@UCallInfo@2@ZZPAVScriptFunction@2@@Z|

TEXTAREA

;;============================================================================================================

;; InterpreterStackFrame::DelayDynamicInterpreterThunk

;;============================================================================================================

;Var InterpreterStackFrame::DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

NESTED\_ENTRY ?DelayDynamicInterpreterThunk@InterpreterStackFrame@Js@@SAPAXPAVRecyclableObject@2@UCallInfo@2@ZZ

PROLOG\_PUSH r0-r5,r11,lr ; save volatile registers and non-volatile registers; r5 is pushed for aligned purposes

bl |?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APAXPAVRecyclableObject@2@UCallInfo@2@ZZPAVScriptFunction@2@@Z| ; call InterpreterStackFrame::EnsureDynamicInterpreterThunk

#if defined(\_CONTROL\_FLOW\_GUARD)

mov r4, r0 ; save entryPoint in r4

mov32 r12, \_\_guard\_check\_icall\_fptr

ldr r12, [r12]

blx r12

mov r0, r4 ; restore entryPoint in r0

#endif

mov r12, r0 ; back up entryPoint in R12

EPILOG\_POP r0-r5,r11,lr ; restore arguments and return address

EPILOG\_NOP bx r12 ; jump (tail call) to new entryPoint

NESTED\_END

;;============================================================================================================

;; DynamicProfileInfo::EnsureDynamicProfileInfoThunk

;;============================================================================================================

;Var DynamicProfileInfo::EnsureDynamicProfileInfoThunk(RecyclableObject\* function, CallInfo callInfo, ...)

NESTED\_ENTRY ?EnsureDynamicProfileInfoThunk@DynamicProfileInfo@Js@@SAPAXPAVRecyclableObject@2@UCallInfo@2@ZZ

PROLOG\_PUSH r0-r3,r11,lr ; save volatile registers

bl |?EnsureDynamicProfileInfo@DynamicProfileInfo@Js@@CAP6APAXPAVRecyclableObject@2@UCallInfo@2@ZZPAVScriptFunction@2@@Z| ; call DynamicProfileInfo::EnsureDynamicProfileInfo

mov r12, r0 ; back up entryPoint in R12

EPILOG\_POP r0-r3,r11,lr ; restore arguments and return address

EPILOG\_NOP bx r12 ; jump (tail call) to new entryPoint

NESTED\_END

;;============================================================================================================

;; ScriptContext::ProfileModeDeferredParsingThunk

;;============================================================================================================

;; Var ScriptContext::ProfileModeDeferredParsingThunk(RecyclableObject\* function, CallInfo callInfo, ...)

NESTED\_ENTRY ?ProfileModeDeferredParsingThunk@ScriptContext@Js@@SAPAXPAVRecyclableObject@2@UCallInfo@2@ZZ

PROLOG\_PUSH r0-r3,r11,lr ; save volatile registers

mov r0, sp ; Pass the address of the function at the saved r0 in case it need to be boxed

bl |?ProfileModeDeferredParse@ScriptContext@Js@@SAP6APAXPAVRecyclableObject@2@UCallInfo@2@ZZPAPAVScriptFunction@2@@Z| ; call ScriptContext::ProfileModeDeferredParse

mov r12, r0 ; back up entryPoint in R12

EPILOG\_POP r0-r3,r11,lr ; restore arguments and return address

EPILOG\_NOP bx r12 ; jump (tail call) to new entryPoint

NESTED\_END

;;============================================================================================================

;; ScriptContext::ProfileModeDeferredDeserializeThunk

;;============================================================================================================

;; Var ScriptContext::ProfileModeDeferredDeserializeThunk(RecyclableObject\* function, CallInfo callInfo, ...)

NESTED\_ENTRY ?ProfileModeDeferredDeserializeThunk@ScriptContext@Js@@SAPAXPAVRecyclableObject@2@UCallInfo@2@ZZ

PROLOG\_PUSH r0-r3,r11,lr ; save volatile registers

bl |?ProfileModeDeferredDeserialize@ScriptContext@Js@@SAP6APAXPAVRecyclableObject@2@UCallInfo@2@ZZPAVScriptFunction@2@@Z| ; call ScriptContext::ProfileModeDeferredDeserialize

mov r12, r0 ; back up entryPoint in R12

EPILOG\_POP r0-r3,r11,lr ; restore arguments and return address

EPILOG\_NOP bx r12 ; jump (tail call) to new entryPoint

NESTED\_END

;;============================================================================================================

END

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if !defined(\_M\_ARM)

#error ArmStackFrame is not supported on this architecture.

#endif

// For ARM, we walk the r11 chain (similar to the EBP chain on x86). This allows us to do what the internal

// stack walker needs to do - find and visit non-leaf Javascript frames on the call stack and retrieve return

// addresses and parameter values. Note that we don't need the equivalent of CONTEXT\_UNWOUND\_TO\_CALL here,

// or any PC adjustment to account for return from tail call, because the PC is only used to determine whether

// we're in a Javascript frame or not, not to control unwinding (via pdata). So we only require that Javascript

// functions not end in call instructions.

// We also require that register parameters be homed on entry to a Javascript function, something that jitted

// functions and the ETW interpreter thunk do, and which C++ vararg functions currently do as well. The guidance

// from C++ is that we not rely on this behavior in future. If we have to visit interpreted Javascript frames

// that don't pass through the ETW thunk, we'll have to use some other mechanism to force homing of parameters.

namespace Js

{

bool

ArmStackFrame::InitializeByFrameId(void \* frame, ScriptContext\* scriptContext)

{

return SkipToFrame(frame);

}

// InitializeByReturnAddress.

// Parameters:

// unwindToAddress: specifies the address we need to unwind the stack before any walks can be done.

// This is expected to be return address i.e. address of the instruction right after the blx instruction

// and not the address of blx itself.

bool

ArmStackFrame::InitializeByReturnAddress(void \* returnAddress, ScriptContext\* scriptContext)

{

this->frame = (void\*\*)arm\_GET\_CURRENT\_FRAME();

while (Next())

{

if (this->codeAddr == returnAddress)

{

return true;

}

}

return false;

}

bool

ArmStackFrame::Next()

{

this->addressOfCodeAddr = this->GetAddressOfReturnAddress();

this->codeAddr = this->GetReturnAddress();

this->frame = (void \*\*)this->frame[0];

return frame != nullptr;

}

bool

ArmStackFrame::SkipToFrame(void \* frameAddress)

{

this->frame = (void \*\*)frameAddress;

return Next();

}

void \*

ArmStackFrame::GetInstructionPointer()

{

return codeAddr;

}

void \*\*

ArmStackFrame::GetArgv(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

UNREFERENCED\_PARAMETER(isCurrentContextNative);

UNREFERENCED\_PARAMETER(shouldCheckForNativeAddr);

return this->frame + ArgOffsetFromFramePtr;

}

void \*

ArmStackFrame::GetReturnAddress(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

UNREFERENCED\_PARAMETER(isCurrentContextNative);

UNREFERENCED\_PARAMETER(shouldCheckForNativeAddr);

return this->frame[ReturnAddrOffsetFromFramePtr];

}

void \*

ArmStackFrame::GetAddressOfReturnAddress(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

UNREFERENCED\_PARAMETER(isCurrentContextNative);

UNREFERENCED\_PARAMETER(shouldCheckForNativeAddr);

return &this->frame[ReturnAddrOffsetFromFramePtr];

}

bool

ArmStackFrame::IsInStackCheckCode(void \*entry, void \*codeAddr, size\_t stackCheckCodeHeight)

{

return ((size\_t(codeAddr) - size\_t(entry)) <= stackCheckCodeHeight);

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

/\*

\* Stackwalking on Evanesco:

\* ------------------------

\* NOTE: This structure is currently only used for \_M\_ARM, which walks a chain of frame pointers.

\* This requires frame chaining using r11 (i.e., no FPO) and the homing of register parameters on

\* entry to Javascript functions.

\*

\* The relevant part of the frame looks like this (high addresses at the top, low ones at the bottom):

\*

\* ----------------------

\* r3 <=== Homed input parameters

\* r2 <

\* r1 <

\* r0 <===

\* lr <=== return address

\* r11 <=== current r11 (frame pointer)

\* ...

\*

\*/

#if !defined(\_M\_ARM)

#error This is only for ARM

#endif

namespace Js

{

class ArmStackFrame

{

public:

ArmStackFrame() : frame(nullptr), codeAddr(nullptr), addressOfCodeAddr(nullptr)

{

}

bool InitializeByFrameId(void \* returnAddress, ScriptContext\* scriptContext);

bool InitializeByReturnAddress(void \* returnAddress, ScriptContext\* scriptContext);

bool Next();

void \*GetInstructionPointer();

void \*\*GetArgv(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

void \*GetReturnAddress(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

void \*GetAddressOfInstructionPointer() { Assert(addressOfCodeAddr != nullptr); return addressOfCodeAddr; }

void \*GetAddressOfReturnAddress(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

bool SkipToFrame(void \* returnAddress);

void \*GetFrame() { return (void \*)frame;};

size\_t GetStackCheckCodeHeight() { return this->stackCheckCodeHeight; }

static bool IsInStackCheckCode(void \*entry, void \*codeAddr, size\_t stackCheckCodeHeight);

private:

void \*\* frame; // r11 (frame pointer) - other interesting values are relative to this address

void \* codeAddr; // return address from the current frame

void\* addressOfCodeAddr;

//ProbeStack height is 40 when stack probe is before prolog & 58 when it is after prolog (for small stacks)

//Choosing the higher number is safe as @58 we are still in the prolog and inlinee frame is not setup yet and

//we shouldn't try to query the frame height for the inlined function.

//TODO (abchatra): Get rid of this magic number in future and design a more safe way of handling the stack check code height.

static const size\_t stackCheckCodeHeight = StackFrameConstants::StackCheckCodeHeight;

};

};

;-------------------------------------------------------------------------------------------------------

; Copyright (C) Microsoft. All rights reserved.

; Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

;-------------------------------------------------------------------------------------------------------

;

; arm64\_CallEhFrame() and arm64\_CallCatch() both thunk into jitted code at the

; start of an EH region. The purpose is to restore the frame pointer (fp)

; and locals pointer (x28) to the appropriate values for executing the parent

; function and to create a local frame that can be unwound using the parent

; function's pdata. The parent's frame looks like this:

;

;-------------------

; {x0-x7} -- homed parameters

; lr -- address from which parent was called

; fp -- saved frame pointer, pointed to by current fp

; arg obj

; {x19-x28} -- non-volatile registers: all of them are saved

; {q8-q15} -- non-volatile double registers: all of them are saved

; locals area -- pointed to by x28

; pointer to non-volatile register area above

; stack args

;-------------------

;

; The reason for the "pointer to non-volatile register area" is to allow the

; unwinder to deallocate the locals area regardless of its size. So this thunk can skip

; the allocation of the locals area altogether, and unwinding still works.

; The unwind pseudo-codes for the above prolog look like:

;

; 1. Deallocate stack args (sp now points to "pointer to non-volatile register area")

; 2. Restore rN (rN now points to first saved register)

; 3. Copy rN to sp (sp now points to first saved register)

; 4. Restore {q8-q15} (non-volatile double registers restored)

; 5. Restore {x19-x28} (non-volatile registers restored, sp points to saved r11)

; 6. Restore fp

; 7. Load lr into pc and deallocate remaining stack.

;

; The prologs for the assembly thunks allocate a frame that can be unwound by executing

; the above steps, although we don't allocate a locals area and don't know the size of the

; stack args. The caller doesn't return to this thunk; it executes its own epilog and

; returns to the caller of the thunk (one of the runtime try helpers).

; Windows version

OPT 2 ; disable listing

#include "ksarm64.h"

OPT 1 ; re-enable listing

TTL Lib\Common\arm\arm64\_CallEhFrame.asm

IMPORT \_\_chkstk

EXPORT arm64\_CallEhFrame

TEXTAREA

NESTED\_ENTRY arm64\_CallEhFrame

; Params:

; x0 -- thunk target

; x1 -- frame pointer

; x2 -- locals pointer

; x3 -- size of stack args area

; Home params and save registers

PROLOG\_SAVE\_REG\_PAIR fp, lr, #-80!

PROLOG\_NOP stp x0, x1, [sp, #16]

PROLOG\_NOP stp x2, x3, [sp, #32]

PROLOG\_NOP stp x4, x5, [sp, #48]

PROLOG\_NOP stp x6, x7, [sp, #64]

PROLOG\_STACK\_ALLOC (10\*8 + 8\*16 + 32)

PROLOG\_NOP stp q8, q9, [sp, #(16 + 0\*16)]

PROLOG\_NOP stp q10, q11, [sp, #(16 + 2\*16)]

PROLOG\_NOP stp q12, q13, [sp, #(16 + 4\*16)]

PROLOG\_NOP stp q14, q15, [sp, #(16 + 6\*16)]

PROLOG\_SAVE\_REG\_PAIR x19, x20, #(16 + 8\*16 + 0\*8)

PROLOG\_SAVE\_REG\_PAIR x21, x22, #(16 + 8\*16 + 2\*8)

PROLOG\_SAVE\_REG\_PAIR x23, x24, #(16 + 8\*16 + 4\*8)

PROLOG\_SAVE\_REG\_PAIR x25, x26, #(16 + 8\*16 + 6\*8)

PROLOG\_SAVE\_REG\_PAIR x27, x28, #(16 + 8\*16 + 8\*8)

; Save a pointer to the saved registers

mov x16, sp

str x16, [sp, #0]

; Set up the frame pointer and locals pointer

mov x28, x2

mov fp, x1

; Allocate the arg out area, calling chkstk if necessary

cmp x3,#4095

bgt chkstk\_call

sub sp,sp,x3

; Thunk to the jitted code (and don't return)

br x0

|chkstk\_call|

; Call chkstk, passing a size/16 count in x15

lsr x15,x3,#4

bl |\_\_chkstk|

sub sp,sp,x15,lsl #4

; Thunk to the jitted code (and don't return)

br x0

NESTED\_END arm64\_CallEhFrame

; arm64\_CallCatch() is similar to arm64\_CallEhFrame() except that we also pass the catch object to the jitted code

EXPORT arm64\_CallCatch

TEXTAREA

NESTED\_ENTRY arm64\_CallCatch

; Params:

; x0 -- thunk target

; x1 -- frame pointer

; x2 -- locals pointer

; x3 -- size of stack args area

; x4 -- exception object

; Home params and save registers

PROLOG\_SAVE\_REG\_PAIR fp, lr, #-80!

PROLOG\_NOP stp x0, x1, [sp, #16]

PROLOG\_NOP stp x2, x3, [sp, #32]

PROLOG\_NOP stp x4, x5, [sp, #48]

PROLOG\_NOP stp x6, x7, [sp, #64]

PROLOG\_STACK\_ALLOC (10\*8 + 8\*16 + 32)

PROLOG\_NOP stp q8, q9, [sp, #(16 + 0\*16)]

PROLOG\_NOP stp q10, q11, [sp, #(16 + 2\*16)]

PROLOG\_NOP stp q12, q13, [sp, #(16 + 4\*16)]

PROLOG\_NOP stp q14, q15, [sp, #(16 + 6\*16)]

PROLOG\_SAVE\_REG\_PAIR x19, x20, #(16 + 8\*16 + 0\*8)

PROLOG\_SAVE\_REG\_PAIR x21, x22, #(16 + 8\*16 + 2\*8)

PROLOG\_SAVE\_REG\_PAIR x23, x24, #(16 + 8\*16 + 4\*8)

PROLOG\_SAVE\_REG\_PAIR x25, x26, #(16 + 8\*16 + 6\*8)

PROLOG\_SAVE\_REG\_PAIR x27, x28, #(16 + 8\*16 + 8\*8)

; Save a pointer to the saved registers

mov x16, sp

str x16, [sp, #0]

; Set up the frame pointer and locals pointer

mov x28, x2

mov fp, x1

; Allocate the arg out area, calling chkstk if necessary

cmp x3,#4095

mov x1, x4

bgt chkstk\_call

sub sp,sp,x3

; Thunk to the jitted code (and don't return)

br x0

|chkstk\_call\_catch|

; Call chkstk, passing a size/16 count in x15

lsr x15,x3,#4

bl |\_\_chkstk|

sub sp,sp,x15,lsl #4

; Thunk to the jitted code (and don't return)

br x0

NESTED\_END arm64\_CallCatch

END

;-------------------------------------------------------------------------------------------------------

; Copyright (C) Microsoft. All rights reserved.

; Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

;-------------------------------------------------------------------------------------------------------

OPT 2 ; disable listing

#include "ksarm64.h"

OPT 1 ; re-enable listing

TTL Lib\Runtime\Language\arm64\arm64\_DelayDynamicInterpreterThunk.asm

;Var InterpreterStackFrame::DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

EXPORT |?DelayDynamicInterpreterThunk@InterpreterStackFrame@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ|

;Var DynamicProfileInfo::EnsureDynamicProfileInfoThunk(RecyclableObject\* function, CallInfo callInfo, ...)

EXPORT |?EnsureDynamicProfileInfoThunk@DynamicProfileInfo@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ|

; Var ScriptContext::ProfileModeDeferredParsingThunk(RecyclableObject\* function, CallInfo callInfo, ...)

EXPORT |?ProfileModeDeferredParsingThunk@ScriptContext@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ|

;JavascriptMethod InterpreterStackFrame::EnsureDynamicInterpreterThunk(Js::ScriptFunction \* function)

IMPORT |?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z|

;JavascriptMethod DynamicProfileInfo::EnsureDynamicProfileInfoThunk(Js::ScriptFunction \* function)

IMPORT |?EnsureDynamicProfileInfo@DynamicProfileInfo@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z|

;JavascriptMethod ScriptContext::ProfileModeDeferredParse(ScriptFunction \*\*function)

IMPORT |?ProfileModeDeferredParse@ScriptContext@Js@@SAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAPEAVScriptFunction@2@@Z|

;JavascriptMethod ScriptContext::ProfileModeDeferredDeserialize(ScriptFunction \*function)

IMPORT |?ProfileModeDeferredDeserialize@ScriptContext@Js@@SAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z|

TEXTAREA

;;============================================================================================================

;; InterpreterStackFrame::DelayDynamicInterpreterThunk

;;============================================================================================================

;Var InterpreterStackFrame::DelayDynamicInterpreterThunk(RecyclableObject\* function, CallInfo callInfo, ...)

NESTED\_ENTRY ?DelayDynamicInterpreterThunk@InterpreterStackFrame@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ

PROLOG\_SAVE\_REG\_PAIR fp, lr, #-80! ; save volatile registers

stp x0, x1, [sp, #16]

stp x2, x3, [sp, #32]

stp x4, x5, [sp, #48]

stp x6, x7, [sp, #64]

bl |?EnsureDynamicInterpreterThunk@InterpreterStackFrame@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z| ; call InterpreterStackFrame::EnsureDynamicInterpreterThunk

mov x16, x0 ; back up entryPoint in x16

ldp x6, x7, [sp, #64] ; restore arguments and return address

ldp x4, x5, [sp, #48]

ldp x2, x3, [sp, #32]

ldp x0, x1, [sp, #16]

EPILOG\_RESTORE\_REG\_PAIR fp, lr, #80!

EPILOG\_NOP br x16 ; jump (tail call) to new entryPoint

NESTED\_END

;;============================================================================================================

;; DynamicProfileInfo::EnsureDynamicProfileInfoThunk

;;============================================================================================================

;Var DynamicProfileInfo::EnsureDynamicProfileInfoThunk(RecyclableObject\* function, CallInfo callInfo, ...)

NESTED\_ENTRY ?EnsureDynamicProfileInfoThunk@DynamicProfileInfo@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ

PROLOG\_SAVE\_REG\_PAIR fp, lr, #-80! ; save volatile registers

stp x0, x1, [sp, #16]

stp x2, x3, [sp, #32]

stp x4, x5, [sp, #48]

stp x6, x7, [sp, #64]

bl |?EnsureDynamicProfileInfo@DynamicProfileInfo@Js@@CAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z| ; call DynamicProfileInfo::EnsureDynamicProfileInfo

mov x16, x0 ; back up entryPoint in x16

ldp x6, x7, [sp, #64] ; restore arguments and return address

ldp x4, x5, [sp, #48]

ldp x2, x3, [sp, #32]

ldp x0, x1, [sp, #16]

EPILOG\_RESTORE\_REG\_PAIR fp, lr, #80!

EPILOG\_NOP br x16 ; jump (tail call) to new entryPoint

NESTED\_END

;;============================================================================================================

;; ScriptContext::ProfileModeDeferredParsingThunk

;;============================================================================================================

;; Var ScriptContext::ProfileModeDeferredParsingThunk(RecyclableObject\* function, CallInfo callInfo, ...)

NESTED\_ENTRY ?ProfileModeDeferredParsingThunk@ScriptContext@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ

PROLOG\_SAVE\_REG\_PAIR fp, lr, #-80! ; save volatile registers

stp x0, x1, [sp, #16]

stp x2, x3, [sp, #32]

stp x4, x5, [sp, #48]

stp x6, x7, [sp, #64]

mov x0, sp ; Pass the address of the function at the saved x0 in case it need to be boxed

bl |?ProfileModeDeferredParse@ScriptContext@Js@@SAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAPEAVScriptFunction@2@@Z| ; call ScriptContext::ProfileModeDeferredParse

mov x16, x0 ; back up entryPoint in x16

ldp x6, x7, [sp, #64] ; restore arguments and return address

ldp x4, x5, [sp, #48]

ldp x2, x3, [sp, #32]

ldp x0, x1, [sp, #16]

EPILOG\_RESTORE\_REG\_PAIR fp, lr, #80!

EPILOG\_NOP br x16 ; jump (tail call) to new entryPoint

NESTED\_END

;;============================================================================================================

;; ScriptContext::ProfileModeDeferredDeserializeThunk

;;============================================================================================================

;; Var ScriptContext::ProfileModeDeferredDeserializeThunk(RecyclableObject\* function, CallInfo callInfo, ...)

NESTED\_ENTRY ?ProfileModeDeferredDeserializeThunk@ScriptContext@Js@@SAPEAXPEAVRecyclableObject@2@UCallInfo@2@ZZ

PROLOG\_SAVE\_REG\_PAIR fp, lr, #-80! ; save volatile registers

stp x0, x1, [sp, #16]

stp x2, x3, [sp, #32]

stp x4, x5, [sp, #48]

stp x6, x7, [sp, #64]

bl |?ProfileModeDeferredDeserialize@ScriptContext@Js@@SAP6APEAXPEAVRecyclableObject@2@UCallInfo@2@ZZPEAVScriptFunction@2@@Z| ; call ScriptContext::ProfileModeDeferredDeserialize

mov x16, x0 ; back up entryPoint in x16

ldp x6, x7, [sp, #64] ; restore arguments and return address

ldp x4, x5, [sp, #48]

ldp x2, x3, [sp, #32]

ldp x0, x1, [sp, #16]

EPILOG\_RESTORE\_REG\_PAIR fp, lr, #80!

EPILOG\_NOP br x16 ; jump (tail call) to new entryPoint

NESTED\_END

;;============================================================================================================

END

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if !defined(\_M\_ARM64)

#error Arm64StackFrame is not supported on this architecture.

#endif

// For ARM64, we walk the fp chain (similar to the EBP chain on x86). This allows us to do what the internal

// stack walker needs to do - find and visit non-leaf Javascript frames on the call stack and retrieve return

// addresses and parameter values. Note that we don't need the equivalent of CONTEXT\_UNWOUND\_TO\_CALL here,

// or any PC adjustment to account for return from tail call, because the PC is only used to determine whether

// we're in a Javascript frame or not, not to control unwinding (via pdata). So we only require that Javascript

// functions not end in call instructions.

// We also require that register parameters be homed on entry to a Javascript function, something that jitted

// functions and the ETW interpreter thunk do, and which C++ vararg functions currently do as well. The guidance

// from C++ is that we not rely on this behavior in future. If we have to visit interpreted Javascript frames

// that don't pass through the ETW thunk, we'll have to use some other mechanism to force homing of parameters.

namespace Js

{

bool

Arm64StackFrame::InitializeByFrameId(void \* frame, ScriptContext\* scriptContext)

{

return SkipToFrame(frame);

}

// InitializeByReturnAddress.

// Parameters:

// unwindToAddress: specifies the address we need to unwind the stack before any walks can be done.

// This is expected to be return address i.e. address of the instruction right after the blx instruction

// and not the address of blx itself.

bool

Arm64StackFrame::InitializeByReturnAddress(void \* returnAddress, ScriptContext\* scriptContext)

{

this->frame = (void\*\*)arm64\_GET\_CURRENT\_FRAME();

while (Next())

{

if (this->codeAddr == returnAddress)

{

return true;

}

}

return false;

}

bool

Arm64StackFrame::Next()

{

this->addressOfCodeAddr = this->GetAddressOfReturnAddress();

this->codeAddr = this->GetReturnAddress();

this->frame = (void \*\*)this->frame[0];

return frame != nullptr;

}

bool

Arm64StackFrame::SkipToFrame(void \* frameAddress)

{

this->frame = (void \*\*)frameAddress;

return Next();

}

void \*

Arm64StackFrame::GetInstructionPointer()

{

return codeAddr;

}

void \*\*

Arm64StackFrame::GetArgv(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

UNREFERENCED\_PARAMETER(isCurrentContextNative);

UNREFERENCED\_PARAMETER(shouldCheckForNativeAddr);

return this->frame + ArgOffsetFromFramePtr;

}

void \*

Arm64StackFrame::GetReturnAddress(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

UNREFERENCED\_PARAMETER(isCurrentContextNative);

UNREFERENCED\_PARAMETER(shouldCheckForNativeAddr);

return this->frame[ReturnAddrOffsetFromFramePtr];

}

void \*

Arm64StackFrame::GetAddressOfReturnAddress(bool isCurrentContextNative, bool shouldCheckForNativeAddr)

{

UNREFERENCED\_PARAMETER(isCurrentContextNative);

UNREFERENCED\_PARAMETER(shouldCheckForNativeAddr);

return &this->frame[ReturnAddrOffsetFromFramePtr];

}

bool

Arm64StackFrame::IsInStackCheckCode(void \*entry, void \*codeAddr, size\_t stackCheckCodeHeight)

{

return ((size\_t(codeAddr) - size\_t(entry)) <= stackCheckCodeHeight);

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

/\*

\* Stackwalking on Evanesco:

\* ------------------------

\* NOTE: This structure is currently only used for \_M\_ARM64, which walks a chain of frame pointers.

\* This requires frame chaining using fp (i.e., no FPO) and the homing of register parameters on

\* entry to Javascript functions.

\*

\* The relevant part of the frame looks like this (high addresses at the top, low ones at the bottom):

\*

\* ----------------------

\* x7 <=== Homed input parameters

\* x6 <

\* x5 <

\* x4 <

\* x3 <

\* x2 <

\* x1 <

\* x0 <===

\* lr <=== return address

\* fp <=== current fp (frame pointer)

\* ...

\*

\*/

#if !defined(\_M\_ARM64)

#error This is only for ARM64

#endif

namespace Js

{

class Arm64StackFrame

{

public:

Arm64StackFrame() : frame(nullptr), codeAddr(nullptr), addressOfCodeAddr(nullptr)

{

}

bool InitializeByFrameId(void \* returnAddress, ScriptContext\* scriptContext);

bool InitializeByReturnAddress(void \* returnAddress, ScriptContext\* scriptContext);

bool Next();

void \*GetInstructionPointer();

void \*\*GetArgv(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

void \*GetReturnAddress(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

void \*GetAddressOfInstructionPointer() { Assert(addressOfCodeAddr != nullptr); return addressOfCodeAddr; }

void \*GetAddressOfReturnAddress(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true);

bool SkipToFrame(void \* returnAddress);

void \*GetFrame() { return (void \*)frame;};

size\_t GetStackCheckCodeHeight() { return this->stackCheckCodeHeight; }

static bool IsInStackCheckCode(void \*entry, void \*codeAddr, size\_t stackCheckCodeHeight);

private:

void \*\* frame; // fp (frame pointer) - other interesting values are relative to this address

void \* codeAddr; // return address from the current frame

void\* addressOfCodeAddr;

//ProbeStack height is 40 when stack probe is before prolog & 58 when it is after prolog (for small stacks)

//Choosing the higher number is safe as @58 we are still in the prolog and inlinee frame is not setup yet and

//we shouldn't try to query the frame height for the inlined function.

//TODO (abchatra): Get rid of this magic number in future and design a more safe way of handling the stack check code height.

static const size\_t stackCheckCodeHeight = StackFrameConstants::StackCheckCodeHeight;

};

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

namespace Js

{

namespace AsmJsJitTemplate

{

#define IsPowerOfTwo(N) ((N >= 1) & !(N & (N - 1)))

#define Is64BitsReg(reg) (reg >= FIRST\_FLOAT\_REG)

#define Is8BitsReg(reg) (reg <= RegEBX)

#define Is64BitsOper() (sizeof(OperationSize) == 8)

#define Is128BitsOper() (sizeof(OperationSize) == 16)

#define Is128BitsReg(reg) Is64BitsReg(reg)

const BYTE MOD0 = 0x0;

const BYTE MOD1 = 0x40;

const BYTE MOD2 = 0x80;

const BYTE MOD3 = 0xC0;

enum InstructionFlags

{

NoFlag = 0,

AffectOp1 = 1 << 0,

};

struct AddressDefinition

{

AddressDefinition( RegNum \_regEffAddr, int \_offset ) :

regEffAddr( \_regEffAddr ),

regEffAddr2( RegNOREG ),

multiplier( 1 ),

offset( \_offset )

{

}

AddressDefinition( RegNum \_regEffAddr, RegNum \_regEffAddr2, int \_multplier, int \_offset ) :

regEffAddr( \_regEffAddr ),

regEffAddr2( \_regEffAddr2 ),

multiplier( \_multplier ),

offset( \_offset )

{

}

RegNum regEffAddr;

RegNum regEffAddr2;

int multiplier;

int offset;

#if DBG\_DUMP

void dump() const

{

if( regEffAddr2 == RegNOREG )

{

if (offset < 0)

Output::Print(L"[%s-0x%X]", RegNamesW[regEffAddr], offset \* -1);

else

Output::Print( L"[%s+0x%X]", RegNamesW[regEffAddr], offset );

}

else

{

if (offset < 0)

Output::Print(L"[%s+%s\*%d+0x%X]", RegNamesW[regEffAddr], RegNamesW[regEffAddr2], multiplier, offset);

else

Output::Print( L"[%s+%s\*%d+0x%X]", RegNamesW[regEffAddr], RegNamesW[regEffAddr2], multiplier, offset );

}

}

#endif

};

// X1 : Unary instruction

// X1\_X2 : X1 <--- X2

// X1\_X2\_X3 : X1 <--- X2 op X3

enum FormatType

{

EMPTY,

REG,

ADDR,

PTR,

IMM,

REG\_PTR,

REG\_REG,

REG\_ADDR,

ADDR\_REG,

REG\_IMM,

ADDR\_IMM,

REG\_REG\_IMM,

REG\_ADDR\_IMM,

};

struct InstrParamsReg

{

InstrParamsReg(RegNum \_reg ) :reg(\_reg){}

RegNum reg;

static const FormatType FORMAT\_TYPE = REG;

#if DBG\_DUMP

void dump() const

{

Output::Print( L"%s", RegNamesW[reg] );

}

#endif

};

template<typename T>

struct InstrParamsImm

{

InstrParamsImm(T \_imm ) : imm(\_imm){}

T imm;

static const FormatType FORMAT\_TYPE = IMM;

#if DBG\_DUMP

void dump() const

{

Output::Print( L"0x%X", imm );

}

#endif

};

struct InstrParamsEmpty

{

InstrParamsEmpty() {}

static const FormatType FORMAT\_TYPE = EMPTY;

#if DBG\_DUMP

void dump() const

{

}

#endif

};

struct InstrParamsPtr

{

InstrParamsPtr(const void\* \_addr ) : addr(\_addr){}

const void\* addr;

static const FormatType FORMAT\_TYPE = PTR;

#if DBG\_DUMP

void dump() const

{

Output::Print( L"ptr:0x%X", (int)addr );

}

#endif

};

struct InstrParamsRegPtr

{

InstrParamsRegPtr(RegNum \_reg, const void\* \_addr ) : reg(\_reg),addr(\_addr){}

RegNum reg;

const void\* addr;

static const FormatType FORMAT\_TYPE = REG\_PTR;

#if DBG\_DUMP

void dump() const

{

Output::Print( L"%s, ptr:0x%X", RegNamesW[reg], (int)addr);

}

#endif

};

struct InstrParamsAddr

{

InstrParamsAddr(RegNum \_regEffAddr, int \_offset ) :addr(\_regEffAddr, \_offset){}

AddressDefinition addr;

static const FormatType FORMAT\_TYPE = ADDR;

#if DBG\_DUMP

void dump() const

{

addr.dump();

}

#endif

};

struct InstrParams2Reg

{

InstrParams2Reg(RegNum \_reg, RegNum \_reg2) :reg(\_reg), reg2(\_reg2){}

RegNum reg, reg2;

static const FormatType FORMAT\_TYPE = REG\_REG;

#if DBG\_DUMP

void dump() const

{

Output::Print( L"%s, %s", RegNamesW[reg], RegNamesW[reg2] );

}

#endif

};

// op reg, [regEffAddr1 + regEffAddr2\*multiplier + offset]

struct InstrParamsRegAddr

{

InstrParamsRegAddr( RegNum \_reg, RegNum \_regEffAddr, int \_offset ) :

reg( \_reg ),

addr( \_regEffAddr, \_offset )

{

}

InstrParamsRegAddr( RegNum \_reg, RegNum \_regEffAddr, RegNum \_regEffAddr2, int \_multplier, int \_offset ) :

reg( \_reg ),

addr( \_regEffAddr , \_regEffAddr2 , \_multplier , \_offset )

{

}

RegNum reg;

AddressDefinition addr;

static const FormatType FORMAT\_TYPE = REG\_ADDR;

#if DBG\_DUMP

void dump() const

{

Output::Print( L"%s, ", RegNamesW[reg] );

addr.dump();

}

#endif

};

struct InstrParamsAddrReg

{

InstrParamsAddrReg( RegNum \_regEffAddr, int \_offset, RegNum \_reg ) :

reg( \_reg ),

addr( \_regEffAddr, \_offset )

{

}

InstrParamsAddrReg( RegNum \_regEffAddr, RegNum \_regEffAddr2, int \_multplier, int \_offset, RegNum \_reg ) :

reg( \_reg ),

addr( \_regEffAddr , \_regEffAddr2 , \_multplier , \_offset )

{

}

RegNum reg;

AddressDefinition addr;

static const FormatType FORMAT\_TYPE = ADDR\_REG;

#if DBG\_DUMP

void dump() const

{

addr.dump();

Output::Print( L" , %s", RegNamesW[reg] );

}

#endif

};

template<typename ImmType>

struct InstrParamsRegImm

{

InstrParamsRegImm(RegNum \_reg, ImmType \_imm) :reg(\_reg), imm(\_imm){}

RegNum reg;

ImmType imm;

static const FormatType FORMAT\_TYPE = REG\_IMM;

#if DBG\_DUMP

void dump() const

{

if (imm < 0)

Output::Print(L"%s, -0x%X", RegNamesW[reg], imm \* -1);

else

Output::Print(L"%s, 0x%X", RegNamesW[reg], imm);

}

#endif

};

template<typename ImmType>

struct InstrParamsAddrImm

{

InstrParamsAddrImm(RegNum \_regEffAddr, int \_offset, ImmType \_imm) :

addr(\_regEffAddr, \_offset),

imm(\_imm)

{}

InstrParamsAddrImm(RegNum \_regEffAddr, RegNum \_regEffAddr2, int \_multplier, int \_offset, ImmType \_imm) :

addr( \_regEffAddr , \_regEffAddr2 , \_multplier , \_offset ),

imm(\_imm)

{}

AddressDefinition addr;

ImmType imm;

static const FormatType FORMAT\_TYPE = ADDR\_IMM;

#if DBG\_DUMP

void dump() const

{

addr.dump();

if (imm < 0)

Output::Print(L", -0x%X", imm \* -1);

else

Output::Print( L", 0x%X", imm );

}

#endif

};

// op reg, [regEffAddr1 + regEffAddr2\*multiplier + offset], imm8

template<typename ImmType>

struct InstrParamsRegAddrImm

{

CompileAssert(sizeof(ImmType) == 1);

InstrParamsRegAddrImm(RegNum \_reg, RegNum \_regEffAddr, int \_offset, ImmType imm) :

reg(\_reg),

addr(\_regEffAddr, \_offset),

imm(imm)

{

}

InstrParamsRegAddrImm(RegNum \_reg, RegNum \_regEffAddr, RegNum \_regEffAddr2, int \_multplier, int \_offset, ImmType imm) :

reg(\_reg),

addr(\_regEffAddr, \_regEffAddr2, \_multplier, \_offset),

imm(imm)

{

}

RegNum reg;

AddressDefinition addr;

ImmType imm;

static const FormatType FORMAT\_TYPE = REG\_ADDR\_IMM;

#if DBG\_DUMP

void dump() const

{

Output::Print(L"%s, ", RegNamesW[reg]);

addr.dump();

if (imm < 0)

Output::Print(L", -0x%X", imm \* -1);

else

Output::Print(L", 0x%X", imm);

}

#endif

};

// op reg, reg, imm8

template<typename ImmType>

struct InstrParams2RegImm

{

CompileAssert(sizeof(ImmType) == 1);

InstrParams2RegImm(RegNum \_reg, RegNum \_reg2, ImmType imm) :reg(\_reg), reg2(\_reg2), imm(imm){}

RegNum reg, reg2;

ImmType imm;

static const FormatType FORMAT\_TYPE = REG\_REG\_IMM;

#if DBG\_DUMP

void dump() const

{

Output::Print(L"%s, %s", RegNamesW[reg], RegNamesW[reg2]);

if (imm < 0)

Output::Print(L", -0x%X", imm \* -1);

else

Output::Print(L", 0x%X", imm);

}

#endif

};

bool FitsInByte(size\_t value)

{

return ((size\_t)(signed char)(value & 0xFF) == value);

}

bool FitsInByteUnsigned(size\_t value)

{

return ((size\_t)(value & 0xFF) == value);

}

template <typename FormatType>

int EncodeModRM\_2Reg(BYTE\*& buffer, const FormatType& params)

{

\*buffer++ = MOD3 | ( RegEncode[params.reg] << 3 ) | RegEncode[params.reg2];

return 1;

}

template<BYTE b, typename FormatType>

int EncodeModRM\_ByteReg( BYTE\*& buffer, const FormatType& params )

{

\*buffer++ = MOD3 | ( b << 3 ) | RegEncode[params.reg];

return 1;

}

int EncodeModRM\_Min( BYTE\*& buffer, BYTE regByte, RegNum regEffAddr, int offset )

{

// [offset]

if( regEffAddr == RegNOREG )

{

\*buffer++ = MOD0 | ( regByte << 3 ) | 0x05;

for( int i = 0; i < 4; i++ )

{

\*buffer++ = (BYTE)offset & 0xFF;

offset >>= 8;

}

return 5;

}

// [reg+offset] or [ebp]

if( offset || regEffAddr == RegEBP )

{

// [reg + byte]

if( FitsInByte( offset ) )

{

\*buffer++ = MOD1 | ( regByte << 3 ) | RegEncode[regEffAddr];

// special case for esp

if( regEffAddr == RegESP )

{

\*buffer++ = 0x24; // SIB byte to esp scaled index none

}

\*buffer++ = (BYTE)offset;

return 2;

}

// [reg + int]

\*buffer++ = MOD2 | ( regByte << 3 ) | RegEncode[regEffAddr];

// special case for esp

if( regEffAddr == RegESP )

{

\*buffer++ = 0x24; // SIB byte to esp scaled index none

}

for( int i = 0; i < 4; i++ )

{

\*buffer++ = (BYTE)offset & 0xFF;

offset >>= 8;

}

return 5;

}

// [reg]

Assert( regEffAddr != RegEBP );

if( regEffAddr == RegESP )

{

// special case [esp]

\*buffer++ = MOD0 | ( regByte << 3 ) | 0x04;

\*buffer++ = 0x24;

return 2;

}

\*buffer++ = MOD0 | ( regByte << 3 ) | RegEncode[regEffAddr];

return 1;

}

int EncodeModRM( BYTE\*& buffer, BYTE regByte, RegNum regEffAddr, RegNum regEffAddr2, int multiplier, int offset )

{

Assert( !Is64BitsReg( regEffAddr ) );

Assert( !Is64BitsReg( regEffAddr2 ) );

AssertMsg( regEffAddr2 != RegESP, "Invalid encoding" );

// Cannot have a multiplier with no register for second regAddr

Assert( !(( regEffAddr2 == RegNOREG ) && ( multiplier != 1 )) );

if( regEffAddr2 == RegNOREG )

{

return EncodeModRM\_Min( buffer, regByte, regEffAddr, offset );

}

// encode modr/m byte

const bool offsetFitsInByte = FitsInByte( offset );

BYTE mod = 0;

// 0 = noEncoding, 1 = encode 1 byte, 2 = encode 4 bytes

int offsetEncoding = 0;

if( offset == 0 || regEffAddr == RegNOREG )

{

mod = MOD0;

if( regEffAddr == RegNOREG )

{

// encode 4 bytes even if offset is 0

offsetEncoding = 2;

}

}

else if( offsetFitsInByte )

{

mod = MOD1;

offsetEncoding = 1;

}

else

{

mod = MOD2;

offsetEncoding = 2;

}

\*buffer++ = mod | ( regByte << 3 ) | 0x04;

BYTE ss = 0;

// encode SIB byte

switch( multiplier )

{

case 1:

ss = MOD0;

break;

case 2:

ss = MOD1;

break;

case 4:

ss = MOD2;

break;

case 8:

ss = MOD3;

break;

default:

Assume( false );

}

BYTE sibReg = RegEncode[regEffAddr];

if( regEffAddr == RegNOREG )

{

sibReg = 0x05;

}

\*buffer++ = ss | ( RegEncode[regEffAddr2] << 3 ) | sibReg;

// encode offset

if( offsetEncoding & 1 )

{

\*buffer++ = (BYTE)offset & 0xFF;

return 3;

}

else if( offsetEncoding & 2 )

{

for( int i = 0; i < 4; i++ )

{

\*buffer++ = (BYTE)offset & 0xFF;

offset >>= 8;

}

return 6;

}

return 2;

}

template <typename FormatType>

int EncodeModRM\_RegRM(BYTE\*& buffer, const FormatType& params)

{

Assert( params.reg != RegNOREG );

return EncodeModRM( buffer, RegEncode[params.reg], params.addr.regEffAddr, params.addr.regEffAddr2, params.addr.multiplier, params.addr.offset );

}

int EncodeModRM\_RegPtr( BYTE\*& buffer, const InstrParamsRegPtr& params )

{

\*buffer++ = MOD0 | RegEncode[params.reg] << 3 | 0x05;

int addr = (int)params.addr;

for( int i = 0; i < 4; i++ )

{

\*buffer++ = (BYTE)addr & 0xFF;

addr >>= 8;

}

return 5;

}

template<BYTE b, typename FormatType>

int EncodeModRM\_ByteRM( BYTE\*& buffer, const FormatType& params )

{

return EncodeModRM( buffer, b, params.addr.regEffAddr, params.addr.regEffAddr2, params.addr.multiplier, params.addr.offset );

}

// encodes the opcode + register

template<BYTE op, typename FormatType>

int EncodeOpReg( BYTE\*& buffer, const FormatType& params )

{

\*buffer++ = op | RegEncode[params.reg];

return 1;

}

template<typename ImmType>

int Encode\_Immutable( BYTE\*& buffer, ImmType imm )

{

for( int i = 0; i < sizeof(ImmType); i++ )

{

\*buffer++ = imm & 0xFF;

imm >>= 8;

}

return sizeof(ImmType);

}

int EncodeFarAddress( BYTE\*& buffer, const InstrParamsPtr& params )

{

AssertMsg( false, "Todo:: need more work for encoding far addresses" );

\*(int32\*)buffer = (int32)params.addr;

buffer += 4;

int16 a;

\_\_asm{

mov a,cs

};

\*(int16\*)buffer = a;

buffer += 2;

return 6;

}

template<typename T>

int Encode\_Empty( BYTE\*& buffer, const T& params )

{

return 0;

}

//////////////////////////////////////////////////////////////////////////

/// OpCode encoding functions

#define OpFuncSignature(name) template<int instrSize, typename ImmType> \

int name##\_OpFunc( BYTE\*& buffer, FormatType formatType, void\* params )

template<int instrSize, typename ImmType, int opReg, int opAddr, int opImm>

int GenericBinary\_OpFunc( BYTE\*& buffer, FormatType formatType )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG\_REG:

case Js::AsmJsJitTemplate::REG\_ADDR:

\*buffer++ = opReg | (int)( instrSize != 1 );

break;

case Js::AsmJsJitTemplate::ADDR\_REG:

\*buffer++ = opAddr | (int)( instrSize != 1 );

break;

case Js::AsmJsJitTemplate::REG\_IMM:

case Js::AsmJsJitTemplate::ADDR\_IMM:

if( instrSize == sizeof( ImmType ) )

{

\*buffer++ = opImm | (int)( instrSize != 1 );

}

else if( sizeof( ImmType ) == 1 )

{

\*buffer++ = 0x83;

}

else

{

AssertMsg( false, "Invalid format" );

}

break;

break;

default:

Assume( false );

}

return 1;

}

OpFuncSignature( ADD )

{

return GenericBinary\_OpFunc<instrSize, ImmType, 0x02, 0x00, 0x80>( buffer, formatType );

}

OpFuncSignature( ADDSD )

{

\*buffer++ = 0xF2;

\*buffer++ = 0x0F;

\*buffer++ = 0x58;

return 3;

}

OpFuncSignature(ADDSS)

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x58;

return 3;

}

OpFuncSignature(MULSS)

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x59;

return 3;

}

OpFuncSignature(DIVSS)

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x5E;

return 3;

}

OpFuncSignature( AND )

{

return GenericBinary\_OpFunc<instrSize, ImmType, 0x22, 0x20, 0x80>( buffer, formatType );

}

OpFuncSignature( BSR )

{

\*buffer++ = 0x0F;

\*buffer++ = 0xBD;

return 2;

}

OpFuncSignature( CALL )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG:

case Js::AsmJsJitTemplate::ADDR:

\*buffer++ = 0xFF;

break;

case Js::AsmJsJitTemplate::PTR:

\*buffer++ = 0x9A;

break;

default:

Assume( false );

}

return 1;

}

OpFuncSignature( CDQ )

{

\*buffer++ = 0x99;

return 1;

}

OpFuncSignature( CMP )

{

return GenericBinary\_OpFunc<instrSize, ImmType, 0x3A, 0x38, 0x80>( buffer, formatType );

}

OpFuncSignature( COMISD )

{

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x2F;

return 3;

}

OpFuncSignature(COMISS)

{

\*buffer++ = 0x0F;

\*buffer++ = 0x2F;

return 2;

}

OpFuncSignature( CVTDQ2PD )

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0xE6;

return 3;

}

OpFuncSignature( CVTPS2PD )

{

\*buffer++ = 0x0F;

\*buffer++ = 0x5A;

return 2;

}

OpFuncSignature( CVTSI2SD )

{

\*buffer++ = 0xF2;

\*buffer++ = 0x0F;

\*buffer++ = 0x2A;

return 3;

}

OpFuncSignature( CVTTSD2SI )

{

\*buffer++ = 0xF2;

\*buffer++ = 0x0F;

\*buffer++ = 0x2C;

return 3;

}

OpFuncSignature( CVTSS2SD )

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x5A;

return 3;

}

OpFuncSignature(CVTTSS2SI)

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x2C;

return 3;

}

OpFuncSignature( CVTSD2SS )

{

\*buffer++ = 0xF2;

\*buffer++ = 0x0F;

\*buffer++ = 0x5A;

return 3;

}

OpFuncSignature(CVTSI2SS)

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x2A;

return 3;

}

OpFuncSignature( DIV )

{

\*buffer++ = 0xF6 | (int)( instrSize != 1 );

return 1;

}

OpFuncSignature(IDIV)

{

\*buffer++ = 0xF6 | (int)(instrSize != 1);

return 1;

}

OpFuncSignature( DIVSD )

{

\*buffer++ = 0xF2;

\*buffer++ = 0x0F;

\*buffer++ = 0x5E;

return 3;

}

OpFuncSignature( FLD )

{

// Add 4 if 64 bits

\*buffer++ = 0xD9 | ( ( instrSize == 8 ) << 2 );

return 1;

}

OpFuncSignature( FSTP )

{

// Add 4 if 64 bits

\*buffer++ = 0xD9 | ( ( instrSize == 8 ) << 2 );

return 1;

}

OpFuncSignature( IMUL )

{

\*buffer++ = 0x0F;

\*buffer++ = 0xAF;

return 2;

}

OpFuncSignature( INC )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG:

return 0; // encode nothing for op

case Js::AsmJsJitTemplate::ADDR:

\*buffer++ = 0xFE | (int)( instrSize != 1 );

return 1;

default:

Assume( false );

break;

}

return 0;

}

OpFuncSignature( JMP )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG:

case Js::AsmJsJitTemplate::ADDR:

\*buffer++ = 0xFF;

break;

case Js::AsmJsJitTemplate::IMM:

// add 2 if in 8bits

\*buffer++ = 0xE9 | ( ( sizeof(ImmType) == 1 ) << 1 );

break;

default:

Assume( false );

break;

}

return 1;

}

OpFuncSignature( LAHF )

{

\*buffer++ = 0x9F;

return 1;

}

OpFuncSignature( MOV )

{

int size = 1;

if( instrSize == 2 )

{

\*buffer++ = 0x66;

++size;

}

switch( formatType )

{

case Js::AsmJsJitTemplate::REG\_REG:

case Js::AsmJsJitTemplate::REG\_ADDR:

\*buffer++ = 0x8A | (int)( instrSize != 1 );

break;

case Js::AsmJsJitTemplate::ADDR\_REG:

\*buffer++ = 0x88 | (int)( instrSize != 1 );

break;

case Js::AsmJsJitTemplate::REG\_IMM:

case Js::AsmJsJitTemplate::ADDR\_IMM:

\*buffer++ = 0xC6 | (int)( instrSize != 1 );

break;

default:

Assume( false );

}

return size;

}

OpFuncSignature( MOVD )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG\_REG:{

InstrParams2Reg\* fparams = (InstrParams2Reg\*)params;

if( Is64BitsReg( fparams->reg ) )

{

Assert( !Is64BitsReg( fparams->reg2 ) );

Assert( instrSize == 8 || instrSize == 4); // Remove == 8 ? we are copying double-word.

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x6E;

}

else

{

Assert( Is64BitsReg( fparams->reg2 ) );

Assert( instrSize == 4 );

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x7E;

}

return 3;

}

case Js::AsmJsJitTemplate::REG\_ADDR:{

InstrParamsRegAddr\* fparams = (InstrParamsRegAddr\*)params;

Assert( Is64BitsReg( fparams->reg ) );

Assert( instrSize == 8 );

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x6E;

return 3;

}

case Js::AsmJsJitTemplate::ADDR\_REG:{

InstrParamsAddrReg\* fparams = (InstrParamsAddrReg\*)params;

Assert( Is64BitsReg( fparams->reg ) );

Assert( instrSize == 4 );

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x7E;

return 3;

}

default:

Assume( false );

}

return 0;

}

OpFuncSignature( MOVSD )

{

\*buffer++ = 0xF2;

\*buffer++ = 0x0F;

\*buffer++ = 0x10 | (int)( formatType == ADDR\_REG );

return 3;

}

OpFuncSignature( MOVSS )

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x10 | (int)( formatType == ADDR\_REG );

return 3;

}

OpFuncSignature( MOVSX )

{

\*buffer++ = 0x0F;

\*buffer++ = 0xBE | (int)( instrSize != 1 );

return 2;

}

OpFuncSignature( MOVZX )

{

\*buffer++ = 0x0F;

\*buffer++ = 0xB6 | (int)( instrSize != 1 );

return 2;

}

OpFuncSignature( MUL )

{

\*buffer++ = 0xF6 | (int)( instrSize != 1 );

return 1;

}

OpFuncSignature( MULSD )

{

\*buffer++ = 0xF2;

\*buffer++ = 0x0F;

\*buffer++ = 0x59;

return 3;

}

OpFuncSignature( NEG )

{

\*buffer++ = 0xF6 | (int)( instrSize != 1 );

return 1;

}

OpFuncSignature( NOT )

{

\*buffer++ = 0xF6 | (int)( instrSize != 1 );

return 1;

}

OpFuncSignature( OR )

{

return GenericBinary\_OpFunc<instrSize, ImmType, 0x0A, 0x08, 0x80>( buffer, formatType );

}

OpFuncSignature( POP )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG:

{

InstrParamsReg\* p = (InstrParamsReg\*)params;

\*buffer++ = 0x58 | RegEncode[p->reg];

}

break;

case Js::AsmJsJitTemplate::ADDR:

\*buffer++ = 0x8F;

break;

default:

Assume( false );

break;

}

return 1;

}

OpFuncSignature( PUSH )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG:

case Js::AsmJsJitTemplate::ADDR:

\*buffer++ = 0xFF;

break;

case Js::AsmJsJitTemplate::IMM:

// add 2 if in 8bits

\*buffer++ = 0x68 | ( ( sizeof(ImmType) == 1 ) << 1 );

break;

default:

Assume( false );

break;

}

return 1;

}

OpFuncSignature( RET )

{

\*buffer++ = 0xC2;

return 1;

}

#define ShiftInstruction(name)\

OpFuncSignature( name )\

{\

switch( formatType )\

{\

case Js::AsmJsJitTemplate::REG:\

case Js::AsmJsJitTemplate::ADDR:\

\*buffer++ = 0xD0 | (BYTE)(instrSize!= 1);\

break;\

case Js::AsmJsJitTemplate::REG\_REG:\

Assert( ((InstrParams2Reg\*)params)->reg2 == RegECX );\

\*buffer++ = 0xD2 | (BYTE)(instrSize!= 1);\

break;\

case Js::AsmJsJitTemplate::ADDR\_REG:\

Assert( ((InstrParamsAddrReg\*)params)->reg == RegECX );\

\*buffer++ = 0xD2 | (BYTE)(instrSize!= 1);\

break;\

case Js::AsmJsJitTemplate::REG\_IMM:\

case Js::AsmJsJitTemplate::ADDR\_IMM:\

Assert( sizeof( ImmType ) == 1 );\

\*buffer++ = 0xC0 | (BYTE)(instrSize!= 1);\

break;\

default:\

Assume( false );\

break;\

}\

return 1;\

}

ShiftInstruction(SAL)

ShiftInstruction(SAR)

ShiftInstruction(SHL)

ShiftInstruction(SHR)

#undef ShiftInstruction

OpFuncSignature( SUB )

{

return GenericBinary\_OpFunc<instrSize, ImmType, 0x2A, 0x28, 0x80>( buffer, formatType );

}

OpFuncSignature( SUBSD )

{

\*buffer++ = 0xF2;

\*buffer++ = 0x0F;

\*buffer++ = 0x5C;

return 3;

}

OpFuncSignature(SUBSS)

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x5C;

return 3;

}

OpFuncSignature( TEST )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG\_REG:

case Js::AsmJsJitTemplate::ADDR\_REG:

\*buffer++ = 0x84 | (int)( instrSize != 1 );

break;

case Js::AsmJsJitTemplate::REG\_IMM:

case Js::AsmJsJitTemplate::ADDR\_IMM:

\*buffer++ = 0xF6 | (int)( instrSize != 1 );

break;

default:

Assume( false );

}

return 1;

}

OpFuncSignature( UCOMISD )

{

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x2E;

return 3;

}

OpFuncSignature(UCOMISS)

{

\*buffer++ = 0x0F;

\*buffer++ = 0x2E;

return 2;

}

OpFuncSignature( XOR )

{

switch( formatType )

{

case Js::AsmJsJitTemplate::REG\_REG:

case Js::AsmJsJitTemplate::REG\_ADDR:

\*buffer++ = 0x32 | (int)( instrSize != 1 );

break;

case Js::AsmJsJitTemplate::ADDR\_REG:

\*buffer++ = 0x30 | (int)( instrSize != 1 );

break;

case Js::AsmJsJitTemplate::REG\_IMM:

case Js::AsmJsJitTemplate::ADDR\_IMM:

\*buffer++ = 0x80 | (int)( instrSize != 1 );

break;

default:

Assume( false );

}

return 1;

}

OpFuncSignature( XORPS )

{

\*buffer++ = 0x0F;

\*buffer++ = 0x57;

return 2;

}

template<typename ImmType, int op>

int JmpGeneric\_OpFunc( BYTE\*& buffer, FormatType formatType )

{

if( sizeof(ImmType) != 1 )

{

\*buffer++ = 0x0F;

\*buffer++ = op ^ 0xF0;

return 2;

}

\*buffer++ = op;

return 1;

}

#define Jcc(name,op) \

OpFuncSignature(name){return JmpGeneric\_OpFunc<ImmType, op>( buffer, formatType );}

Jcc(JA , 0x77 )

Jcc(JAE , 0x73 )

Jcc(JB , 0x72 )

Jcc(JBE , 0x76 )

Jcc(JC , 0x72 )

Jcc(JE , 0x74 )

Jcc(JG , 0x7F )

Jcc(JGE , 0x7D )

Jcc(JL , 0x7C )

Jcc(JLE , 0x7E )

Jcc(JNA , 0x75 )

Jcc(JNAE, 0x72 )

Jcc(JNB , 0x73 )

Jcc(JNBE, 0x77 )

Jcc(JNC , 0x73 )

Jcc(JNE , 0x75 )

Jcc(JNG , 0x7E )

Jcc(JNGE, 0x7C )

Jcc(JNL , 0x7D )

Jcc(JNLE, 0x7F )

Jcc(JNO , 0x71 )

Jcc(JNP , 0x7B )

Jcc(JNS , 0x79 )

Jcc(JNZ , 0x75 )

Jcc(JO , 0x70 )

Jcc(JP , 0x7A )

Jcc(JPE , 0x7A )

Jcc(JPO , 0x7B )

Jcc(JS , 0x78 )

Jcc(JZ , 0x74 )

#undef Jcc

template<int op>

int SetFlagGeneric\_OpFunc( BYTE\*& buffer, FormatType formatType )

{

\*buffer++ = op;

return 1;

}

#define SETFLAG(name,op) \

OpFuncSignature(name){\

\*buffer++ = 0x0F;\

\*buffer++ = op;\

return 1;\

}

SETFLAG(SETA ,0x97)

SETFLAG(SETAE ,0x93)

SETFLAG(SETB ,0x92)

SETFLAG(SETBE ,0x96)

SETFLAG(SETC ,0x92)

SETFLAG(SETE ,0x94)

SETFLAG(SETG ,0x9F)

SETFLAG(SETGE ,0x9D)

SETFLAG(SETL ,0x9C)

SETFLAG(SETLE ,0x9E)

SETFLAG(SETNA ,0x96)

SETFLAG(SETNAE,0x92)

SETFLAG(SETNB ,0x93)

SETFLAG(SETNBE,0x97)

SETFLAG(SETNC ,0x93)

SETFLAG(SETNE ,0x95)

SETFLAG(SETNG ,0x9E)

SETFLAG(SETNGE,0x9C)

SETFLAG(SETNL ,0x9D)

SETFLAG(SETNLE,0x9F)

SETFLAG(SETNO ,0x91)

SETFLAG(SETNP ,0x9B)

SETFLAG(SETNS ,0x99)

SETFLAG(SETNZ ,0x95)

SETFLAG(SETO ,0x90)

SETFLAG(SETP ,0x9A)

SETFLAG(SETPE ,0x9A)

SETFLAG(SETPO ,0x9B)

SETFLAG(SETS ,0x98)

SETFLAG(SETZ ,0x94)

#undef SETFLAG

#define CMOV(name,op) \

OpFuncSignature(name){\

\*buffer++ = 0x0F;\

\*buffer++ = op;\

return 1;\

}

CMOV(CMOVA , 0x47)

CMOV(CMOVAE , 0x43)

CMOV(CMOVB , 0x42)

CMOV(CMOVBE , 0x46)

CMOV(CMOVC , 0x42)

CMOV(CMOVE , 0x44)

CMOV(CMOVG , 0x4F)

CMOV(CMOVGE , 0x4D)

CMOV(CMOVL , 0x4C)

CMOV(CMOVLE , 0x4E)

CMOV(CMOVNA , 0x46)

CMOV(CMOVNAE, 0x42)

CMOV(CMOVNB , 0x43)

CMOV(CMOVNBE, 0x47)

CMOV(CMOVNC , 0x43)

CMOV(CMOVNE , 0x45)

CMOV(CMOVNG , 0x4E)

CMOV(CMOVNGE, 0x4C)

CMOV(CMOVNL , 0x4D)

CMOV(CMOVNLE, 0x4F)

CMOV(CMOVNO , 0x41)

CMOV(CMOVNP , 0x4B)

CMOV(CMOVNS , 0x49)

CMOV(CMOVNZ , 0x45)

CMOV(CMOVO , 0x40)

CMOV(CMOVP , 0x4A)

CMOV(CMOVPE , 0x4A)

CMOV(CMOVPO , 0x4B)

CMOV(CMOVS , 0x48)

CMOV(CMOVZ , 0x44)

#undef CMOV

// SSE2 instructions

OpFuncSignature( MOVUPS ){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x10 | (int)(formatType == ADDR\_REG);

return 2;

}

OpFuncSignature(MOVAPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x28 | (int)(formatType == ADDR\_REG);

return 2;

}

OpFuncSignature(MOVHPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x16 | (int)(formatType == ADDR\_REG);

return 3;

}

OpFuncSignature(MOVHLPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x12;

return 2;

}

OpFuncSignature(MOVLHPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x16;

return 2;

}

OpFuncSignature( SHUFPS ){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0xC6;

return 2;

}

OpFuncSignature(SHUFPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xC6;

return 3;

}

OpFuncSignature(PSHUFD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x70;

return 3;

}

OpFuncSignature(CVTPD2PS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x5A;

return 3;

}

OpFuncSignature(CVTDQ2PS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x5B;

return 2;

}

OpFuncSignature(CVTTPS2DQ)

{

\*buffer++ = 0xF3;

\*buffer++ = 0x0F;

\*buffer++ = 0x5B;

return 3;

}

OpFuncSignature(CVTTPD2DQ)

{

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xE6;

return 3;

}

OpFuncSignature(ANDPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x54;

return 3;

}

OpFuncSignature(ANDNPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x55;

return 2;

}

OpFuncSignature(ANDNPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x55;

return 3;

}

OpFuncSignature(PXOR){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xEF;

return 3;

}

OpFuncSignature(DIVPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x5E;

return 2;

}

OpFuncSignature(DIVPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x5E;

return 3;

}

OpFuncSignature(SQRTPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x51;

return 2;

}

OpFuncSignature(SQRTPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x51;

return 3;

}

OpFuncSignature(ADDPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x58;

return 2;

}

OpFuncSignature(ADDPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x58;

return 3;

}

OpFuncSignature(PADDD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xFE;

return 3;

}

OpFuncSignature(SUBPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x5C;

return 2;

}

OpFuncSignature(SUBPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x5C;

return 3;

}

OpFuncSignature(PSUBD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xFA;

return 3;

}

OpFuncSignature(MULPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x59;

return 2;

}

OpFuncSignature(MULPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x59;

return 3;

}

OpFuncSignature(PMULUDQ){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xF4;

return 3;

}

OpFuncSignature(PSRLDQ){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x73;

return 3;

}

OpFuncSignature(PUNPCKLDQ){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x62;

return 3;

}

OpFuncSignature(MINPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x5D;

return 3;

}

OpFuncSignature(MAXPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x5F;

return 3;

}

OpFuncSignature(MINPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x5D;

return 3;

}

OpFuncSignature(MAXPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x5F;

return 3;

}

OpFuncSignature(CMPPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0xC2;

return 2;

}

OpFuncSignature(CMPPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xC2;

return 3;

}

OpFuncSignature(PCMPGTD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x66;

return 3;

}

OpFuncSignature(PCMPEQD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x76;

return 3;

}

OpFuncSignature(ANDPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x54;

return 2;

}

OpFuncSignature(PAND){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xDB;

return 2;

}

OpFuncSignature(ORPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x56;

return 2;

}

OpFuncSignature(POR){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0xEB;

return 3;

}

OpFuncSignature(MOVMSKPS){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x0F;

\*buffer++ = 0x50;

return 2;

}

OpFuncSignature(MOVMSKPD){

CompileAssert(instrSize == sizeof(AsmJsSIMDValue));

\*buffer++ = 0x66;

\*buffer++ = 0x0F;

\*buffer++ = 0x50;

return 3;

}

struct EncodingInfo

{

int opSize, operandSize, immSize;

void Fill( int \_opSize, int \_operandSize, int \_immSize )

{

opSize = \_opSize;

operandSize = \_operandSize;

immSize = \_immSize;

}

int GetSizeBeforeImm() const {return opSize + operandSize;}

int GetSizeBeforeOperand()const {return opSize;}

int GetSizeBeforeOpCOde() const {return 0;}

};

// Dump generated bytes

#define DUMP\_ASM\_CODE\_NB\_BYTES 5

#define DUMP\_ASM\_CODE\_PADDING(size) ((DUMP\_ASM\_CODE\_NB\_BYTES-size%DUMP\_ASM\_CODE\_NB\_BYTES)%DUMP\_ASM\_CODE\_NB\_BYTES)\*5+1

template<typename T>

void DumpAsmCode( const BYTE\* buffer, const int size, const wchar\_t\* instructionName, T\* params )

{

#if DBG\_DUMP

if( PHASE\_TRACE( AsmjsEncoderPhase, AsmJsJitTemplate::Globals::CurrentEncodingFunction ) )

{

int j = 0;

for( int i = size; i > 0; --i, ++j )

{

if( j == DUMP\_ASM\_CODE\_NB\_BYTES )

{

Output::Print( L"\n" ); j = 0;

}

Output::Print( L"0x%02X ", buffer[-i] );

}

Output::Print( L"%\*c %s ", DUMP\_ASM\_CODE\_PADDING( size ), ' ', instructionName );

if( params )

{

params->dump();

}

Output::Print( L" (size: %d)\n", size );

}

#endif

}

#if DBG\_DUMP

#define InstructionMembers(name, supInstrSize, flags) \

static const int SupportedInstrSize = supInstrSize;\

static const wchar\_t\* InstructionName;\

static const int Flags = flags;\

static const wchar\_t\* GetInstructionName() { return InstructionName; }

#define InstructionMemberInit(name)\

const wchar\_t\* name::InstructionName = L#name;

#else

#define InstructionMembers(name, supInstrSize, flags) \

static const int SupportedInstrSize = supInstrSize;\

static const int Flags = flags;\

static const wchar\_t\* GetInstructionName() { return L""; }

#define InstructionMemberInit(name)

#endif

#define InstructionStart(name, supInstrSize, maxSize, flags) \

struct name {\

InstructionMembers(name, supInstrSize, flags)\

private:\

template<int instrSize, typename ImmType> \

static int EncodeOpFunc( BYTE\*& buffer, FormatType formatType, void\* params )\

{\

return name##\_OpFunc<instrSize,ImmType>(buffer,formatType,params);\

}\

public:

#define InstructionEnd(name) \

};\

InstructionMemberInit(name);

// Structure for instructions

#define InstructionEmpty() \

template<typename OperationSize> static int EncodeInstruction( BYTE\*& buffer, EncodingInfo\* info = nullptr )\

{\

CompileAssert((sizeof(OperationSize)&(SupportedInstrSize)));\

CompileAssert(IsPowerOfTwo(sizeof(OperationSize)));\

const int size = EncodeOpFunc<sizeof(OperationSize),int>(buffer,EMPTY,nullptr);\

if(info) info->Fill(size,0,0); \

DumpAsmCode<InstrParamsEmpty>(buffer,size,GetInstructionName(),nullptr);\

return size;\

}

#define InstructionFormat(check,Format,encodingfunc) \

template<typename OperationSize> static int EncodeInstruction( BYTE\*& buffer, const Format& params, EncodingInfo\* info = nullptr )\

{\

CompileAssert((sizeof(OperationSize)&(SupportedInstrSize))); \

CompileAssert(IsPowerOfTwo(sizeof(OperationSize))); \

Assert(check);\

const int opsize = EncodeOpFunc<sizeof(OperationSize),int>(buffer,Format::FORMAT\_TYPE,(void\*)&params);\

const int operandSize = encodingfunc(buffer,params);\

const int size = opsize+operandSize;\

if(info) info->Fill(opsize,operandSize,0); \

DumpAsmCode(buffer,size,GetInstructionName(),&params);\

return size;\

}

// Structure for instructions with a constant value

#define InstructionFormat\_Imm(check,Format,encodingfunc) \

template<typename OperationSize, typename ImmType> static int EncodeInstruction( BYTE\*& buffer, const Format<ImmType>& params, EncodingInfo\* info = nullptr )\

{\

CompileAssert((sizeof(OperationSize)&(SupportedInstrSize)));\

CompileAssert(IsPowerOfTwo(sizeof(OperationSize)));\

Assert(check);\

const int opsize = EncodeOpFunc<sizeof(OperationSize),ImmType>(buffer,Format<ImmType>::FORMAT\_TYPE, (void\*)&params) ;\

const int operandSize = encodingfunc(buffer,params);\

const int immSize = Encode\_Immutable<ImmType>(buffer,params.imm);\

const int size = opsize+operandSize+immSize;\

if(info) info->Fill(opsize,operandSize,immSize); \

DumpAsmCode(buffer,size,GetInstructionName(),&params);\

return size;\

}

#define FormatEmpty() \

InstructionEmpty()

#define FormatUnaryPtr(encodingfunc) \

InstructionFormat(\

(true)\

,InstrParamsPtr\

,encodingfunc\

)

#define FormatRegPtr(encodingfunc) \

InstructionFormat(\

(!Is64BitsOper() || Is64BitsReg(params.reg))\

,InstrParamsRegPtr\

,encodingfunc\

)

#define FormatUnaryRegCustomCheck(encodingfunc,check) \

InstructionFormat(\

(check)\

,InstrParamsReg\

,encodingfunc\

)

#define FormatUnaryReg(encodingfunc) \

FormatUnaryRegCustomCheck(\

encodingfunc,\

(!(Is64BitsOper()^Is64BitsReg(params.reg)))\

)

// Support only al,cl,dl,bl

#define FormatUnaryReg8Bits(encodingfunc) \

FormatUnaryRegCustomCheck(\

encodingfunc,\

(!Is64BitsOper() && Is8BitsReg(params.reg))\

)

#define FormatUnaryAddr(encodingfunc) \

InstructionFormat(\

(true)\

,InstrParamsAddr\

,encodingfunc\

)

#define Format2RegCustomCheck(encodingfunc, check) \

InstructionFormat(\

(check)\

,InstrParams2Reg\

,encodingfunc\

)

// Left register must be 64bits and right register must be 32 bits

// op xmm,r32

#define Format2Reg64\_32(encodingfunc) \

Format2RegCustomCheck(\

encodingfunc,\

Is64BitsReg(params.reg) && !Is64BitsReg(params.reg2)\

)

// Left register must be 32 bits and right register must be 64 bits

// op r32,xmm

#define Format2Reg32\_64(encodingfunc) \

Format2RegCustomCheck(\

encodingfunc,\

!Is64BitsReg(params.reg) && Is64BitsReg(params.reg2)\

)

#define Format2Reg(encodingfunc) \

Format2RegCustomCheck(\

encodingfunc,\

((!Is64BitsOper() || Is64BitsReg(params.reg)) && (!Is64BitsOper() || Is64BitsReg(params.reg2)))\

)

#define FormatRegAddrCustomCheck(encodingfunc,check) \

InstructionFormat(\

(check)\

,InstrParamsRegAddr\

,encodingfunc\

)

#define FormatRegAddr(encodingfunc) \

FormatRegAddrCustomCheck(\

encodingfunc,\

(!Is64BitsOper() || Is64BitsReg(params.reg))\

)

#define FormatAddrRegCustomCheck(encodingfunc,check) \

InstructionFormat(\

(check)\

,InstrParamsAddrReg\

,encodingfunc\

)

#define FormatAddrReg(encodingfunc) \

FormatAddrRegCustomCheck(\

encodingfunc,\

(!Is64BitsOper() || Is64BitsReg(params.reg))\

)

#define FormatRegImm(encodingfunc) \

InstructionFormat\_Imm(\

( !Is64BitsOper() && !Is64BitsReg(params.reg) )\

,InstrParamsRegImm\

,encodingfunc\

)

#define FormatAddrImm(encodingfunc) \

InstructionFormat\_Imm(\

( !Is64BitsOper() )\

,InstrParamsAddrImm\

,encodingfunc\

)

#define FormatUnaryImm(encodingfunc) \

InstructionFormat\_Imm(\

(!(Is64BitsOper()))\

,InstrParamsImm\

,encodingfunc\

)

#define Format2RegImm8(encodingfunc) \

InstructionFormat\_Imm(\

(Is128BitsOper() && Is128BitsReg(params.reg) && Is128BitsReg(params.reg2) && FitsInByteUnsigned(params.imm))\

, InstrParams2RegImm\

, encodingfunc\

)

#define FormatRegAddrImm8(encodingfunc) \

InstructionFormat\_Imm(\

(Is128BitsOper() && Is128BitsReg(params.reg) && FitsInByte(params.imm))\

, InstrParamsRegAddrImm\

, encodingfunc\

)

#define FormatRegImm8(encodingfunc) \

InstructionFormat\_Imm(\

(Is128BitsOper() && Is128BitsReg(params.reg) && FitsInByte(params.imm))\

, InstrParamsRegImm\

, encodingfunc\

)

#include "AsmJsInstructionTemplate.inl"

// cleanup macros

#undef InstructionStart

#undef Format2Reg

#undef FormatRegAddr

#undef FormatAddrReg

#undef FormatRegImm

#undef FormatAddrImm

#undef InstructionEnd

#undef InstructionMembers

#undef InstructionMemberInit

#undef InstructionFormat\_Imm

#undef InstructionFormat

int MovHigh8Bits( BYTE\*& buffer, RegNum reg, int8 imm )

{

Assert( reg <= RegEBX );

BYTE\* opDst = buffer;

int size = MOV::EncodeInstruction<int8>( buffer, InstrParamsRegImm<int8>(reg, imm) );

\*opDst |= 4;

return size;

}

enum High8BitsRegType

{

LOW\_HIGH = 0x04,

HIGH\_LOW = 0x20,

HIGH\_HIGH = 0x24,

};

int MovHigh8Bits( BYTE\*& buffer, RegNum reg, RegNum reg2, High8BitsRegType high8BitsRegType )

{

Assert( reg <= RegEBX );

Assert( reg2 <= RegEBX );

BYTE\* opDst = buffer;

int size = MOV::EncodeInstruction<int8>( buffer, InstrParams2Reg(reg, reg2) );

opDst[1] |= high8BitsRegType;

return size;

}

int MovHigh8Bits( BYTE\*& buffer, RegNum reg, RegNum regEffAddr, int offset )

{

Assert( reg <= RegEBX );

BYTE\* opDst = buffer;

int size = MOV::EncodeInstruction<int8>( buffer, InstrParamsRegAddr(reg, regEffAddr, offset) );

opDst[1] |= 0x20;

return size;

}

int MovHigh8Bits( BYTE\*& buffer, RegNum regEffAddr, int offset, RegNum reg )

{

Assert( reg <= RegEBX );

BYTE\* opDst = buffer;

int size = MOV::EncodeInstruction<int8>( buffer, InstrParamsAddrReg(regEffAddr, offset, reg) );

opDst[1] |= 0x20;

return size;

}

int ApplyCustomTemplate( BYTE\*& buffer, const BYTE\* src, const int size )

{

memcpy\_s( buffer, size, src, size );

buffer += size;

DumpAsmCode<InstrParamsEmpty>( buffer, size, L"Custom",nullptr);

return size;

}

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#ifndef InstructionStart

#define InstructionStart(name, SupportedOperationSize, MaxTemplateSize)

#endif

#ifndef InstructionEnd

#define InstructionEnd(name)

#endif

InstructionStart( ADD , 1|4 , 11, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<0>)

FormatAddrImm(EncodeModRM\_ByteRM<0>)

InstructionEnd ( ADD )

InstructionStart( ADDSD , 8 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( ADDSD )

InstructionStart(ADDSS, 4|8, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(ADDSS)

InstructionStart( AND , 1|4 , 11, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<4>)

FormatAddrImm(EncodeModRM\_ByteRM<4>)

InstructionEnd ( AND )

InstructionStart( BSR , 2|4 , 7, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( BSR )

InstructionStart( CALL , 4 , 7 , NoFlag )

FormatUnaryPtr(EncodeFarAddress)

FormatUnaryReg(EncodeModRM\_ByteReg<2>)

FormatUnaryAddr(EncodeModRM\_ByteRM<2>)

InstructionEnd( CALL )

InstructionStart( CDQ , 4 , 1 , NoFlag )

FormatEmpty()

InstructionEnd( CDQ )

InstructionStart( CMP , 1|4 , 11, NoFlag )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<7>)

FormatAddrImm(EncodeModRM\_ByteRM<7>)

InstructionEnd ( CMP )

InstructionStart( COMISD , 8 , 8 , NoFlag )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( COMISD )

InstructionStart(COMISS, 4|8, 8, NoFlag)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(COMISS)

InstructionStart( CVTDQ2PD , 8 | 16 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( CVTDQ2PD )

InstructionStart( CVTPS2PD , 8 |16 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( CVTPS2PD )

InstructionStart( CVTSI2SD , 8 , 8 , AffectOp1 )

Format2Reg64\_32(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( CVTSI2SD )

InstructionStart( CVTTSD2SI , 4 , 8 , AffectOp1 )

Format2Reg32\_64(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( CVTTSD2SI )

InstructionStart( CVTSS2SD , 4|8 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( CVTSS2SD )

InstructionStart(CVTTSS2SI, 4 | 8, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(CVTTSS2SI)

InstructionStart( CVTSD2SS , 8 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( CVTSD2SS )

InstructionStart(CVTSI2SS, 4|8, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(CVTSI2SS)

// Unsigned division

InstructionStart( DIV , 1|4 , 7 , AffectOp1 )

FormatUnaryReg(EncodeModRM\_ByteReg<6>)

FormatUnaryAddr(EncodeModRM\_ByteRM<6>)

InstructionEnd ( DIV )

InstructionStart(IDIV, 1 | 4, 7, AffectOp1)

FormatUnaryReg(EncodeModRM\_ByteReg<7>)

FormatUnaryAddr(EncodeModRM\_ByteRM<7>)

InstructionEnd(IDIV)

InstructionStart( DIVSD , 8 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( DIVSD )

InstructionStart(DIVSS, 4|8, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(DIVSS)

InstructionStart( FLD , 4|8 , 6 , NoFlag )

FormatUnaryAddr(EncodeModRM\_ByteRM<0>)

InstructionEnd ( FLD )

InstructionStart( FSTP , 4|8 , 6 , AffectOp1 )

FormatUnaryAddr(EncodeModRM\_ByteRM<3>)

InstructionEnd ( FSTP )

//format imul reg,[addr+offset],imm possible but not implemented

InstructionStart( IMUL , 4 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd( IMUL )

InstructionStart( INC , 1|4 , 7 , AffectOp1 )

FormatUnaryAddr(EncodeModRM\_ByteRM<0>)

FormatUnaryReg(EncodeOpReg<0x40>)

InstructionEnd ( INC )

InstructionStart( JMP , 1|4 , 6 , NoFlag )

FormatUnaryImm(Encode\_Empty)

FormatUnaryAddr(EncodeModRM\_ByteRM<4>)

FormatUnaryReg(EncodeModRM\_ByteReg<4>)

InstructionEnd ( JMP )

InstructionStart( LAHF , 1|4|8 , 1 , NoFlag )

FormatEmpty()

InstructionEnd ( LAHF )

InstructionStart( MOV , 1|2|4 , 11, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<0>)

FormatAddrImm(EncodeModRM\_ByteRM<0>)

InstructionEnd ( MOV )

InstructionStart( MOVD , 4|8 , 8 , AffectOp1 )

Format2RegCustomCheck(EncodeModRM\_2Reg,true)

FormatRegAddrCustomCheck(EncodeModRM\_RegRM,true)

FormatAddrRegCustomCheck(EncodeModRM\_RegRM,true)

InstructionEnd ( MOVD )

InstructionStart( MOVSD , 8 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd ( MOVSD )

InstructionStart( MOVSS , 4|8 , 8, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd ( MOVSS )

InstructionStart( MOVSX , 1|2 , 11, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( MOVSX )

InstructionStart( MOVZX , 1|2 , 11, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( MOVZX )

// Unsigned multiplication

InstructionStart( MUL , 1|4 , 7 , AffectOp1 )

FormatUnaryReg(EncodeModRM\_ByteReg<4>)

FormatUnaryAddr(EncodeModRM\_ByteRM<4>)

InstructionEnd ( MUL )

InstructionStart( MULSD , 8 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( MULSD )

InstructionStart(MULSS, 4|8, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(MULSS)

InstructionStart( NEG , 1|4 , 6 , AffectOp1 )

FormatUnaryReg(EncodeModRM\_ByteReg<3>)

FormatUnaryAddr(EncodeModRM\_ByteRM<3>)

InstructionEnd ( NEG )

InstructionStart( NOT , 1|4 , 6 , AffectOp1 )

FormatUnaryReg(EncodeModRM\_ByteReg<2>)

FormatUnaryAddr(EncodeModRM\_ByteRM<2>)

InstructionEnd ( NOT )

InstructionStart( OR , 1|4 , 11, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<1>)

FormatAddrImm(EncodeModRM\_ByteRM<1>)

InstructionEnd ( OR )

InstructionStart( POP , 4 , 7 , AffectOp1 )

FormatUnaryReg(Encode\_Empty)

FormatUnaryAddr(EncodeModRM\_ByteRM<0>)

InstructionEnd( POP )

InstructionStart( PUSH , 1|4 , 7 , NoFlag )

FormatUnaryImm(Encode\_Empty)

FormatUnaryReg(EncodeModRM\_ByteReg<6>)

FormatUnaryAddr(EncodeModRM\_ByteRM<6>)

InstructionEnd( PUSH )

InstructionStart( RET , 1|2|4|8 , 1 , NoFlag )

FormatUnaryImm(Encode\_Empty)

InstructionEnd( RET )

#define ShiftInstruction(name, val)\

InstructionStart( name , 1|4 , 7 , AffectOp1 )\

FormatUnaryReg(EncodeModRM\_ByteReg<val>)\

FormatUnaryAddr(EncodeModRM\_ByteRM<val>)\

Format2Reg(EncodeModRM\_ByteReg<val>)\

FormatAddrReg(EncodeModRM\_ByteRM<val>)\

FormatRegImm(EncodeModRM\_ByteReg<val>)\

FormatAddrImm(EncodeModRM\_ByteRM<val>)\

InstructionEnd ( name )

ShiftInstruction(SAL, 4)

ShiftInstruction(SAR, 7)

ShiftInstruction(SHL, 4)

ShiftInstruction(SHR, 5)

#undef ShiftInstruction

InstructionStart( SUB , 1|4 , 11, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<5>)

FormatAddrImm(EncodeModRM\_ByteRM<5>)

InstructionEnd ( SUB )

InstructionStart(SUBSS, 4 | 8, 11, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<5>)

FormatAddrImm(EncodeModRM\_ByteRM<5>)

InstructionEnd(SUBSS)

InstructionStart( SUBSD , 8 , 8 , AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( SUBSD )

InstructionStart( TEST , 1|4 , 7 , NoFlag )

Format2Reg(EncodeModRM\_2Reg)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<0>)

FormatAddrImm(EncodeModRM\_ByteRM<0>)

InstructionEnd ( TEST )

InstructionStart( UCOMISD , 8 , 8 , NoFlag )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd ( UCOMISD )

InstructionStart(UCOMISS, 4|8, 8, NoFlag)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(UCOMISS)

InstructionStart( XOR , 1|4 , 11, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegImm(EncodeModRM\_ByteReg<6>)

FormatAddrImm(EncodeModRM\_ByteRM<6>)

InstructionEnd ( XOR )

InstructionStart( XORPS , 8 | 16 , 8, AffectOp1 )

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd ( XORPS )

#define Jcc(name) \

InstructionStart( name, 1|4 , 6, NoFlag )\

FormatUnaryImm(Encode\_Empty)\

InstructionEnd ( name )

Jcc(JA )

Jcc(JAE )

Jcc(JB )

Jcc(JBE )

Jcc(JC )

Jcc(JE )

Jcc(JG )

Jcc(JGE )

Jcc(JL )

Jcc(JLE )

Jcc(JNA )

Jcc(JNAE)

Jcc(JNB )

Jcc(JNBE)

Jcc(JNC )

Jcc(JNE )

Jcc(JNG )

Jcc(JNGE)

Jcc(JNL )

Jcc(JNLE)

Jcc(JNO )

Jcc(JNP )

Jcc(JNS )

Jcc(JNZ )

Jcc(JO )

Jcc(JP )

Jcc(JPE )

Jcc(JPO )

Jcc(JS )

Jcc(JZ )

#undef Jcc

#define SETFLAG(name) \

InstructionStart( name, 1 , 3, AffectOp1 )\

FormatUnaryReg8Bits(EncodeModRM\_ByteReg<0>)\

InstructionEnd ( name )

SETFLAG(SETA)

SETFLAG(SETAE)

SETFLAG(SETB)

SETFLAG(SETBE)

SETFLAG(SETC)

SETFLAG(SETE)

SETFLAG(SETG)

SETFLAG(SETGE)

SETFLAG(SETL)

SETFLAG(SETLE)

SETFLAG(SETNA)

SETFLAG(SETNAE)

SETFLAG(SETNB)

SETFLAG(SETNBE)

SETFLAG(SETNC)

SETFLAG(SETNE)

SETFLAG(SETNG)

SETFLAG(SETNGE)

SETFLAG(SETNL)

SETFLAG(SETNLE)

SETFLAG(SETNO)

SETFLAG(SETNP)

SETFLAG(SETNS)

SETFLAG(SETNZ)

SETFLAG(SETO)

SETFLAG(SETP)

SETFLAG(SETPE)

SETFLAG(SETPO)

SETFLAG(SETS)

SETFLAG(SETZ)

#undef SETFLAG

#define CMOV(name) \

InstructionStart( name, 4 , 7, AffectOp1 )\

Format2Reg(EncodeModRM\_2Reg)\

FormatRegAddr(EncodeModRM\_RegRM)\

InstructionEnd ( name )

CMOV(CMOVA)

CMOV(CMOVAE)

CMOV(CMOVB)

CMOV(CMOVBE)

CMOV(CMOVC)

CMOV(CMOVE)

CMOV(CMOVG)

CMOV(CMOVGE)

CMOV(CMOVL)

CMOV(CMOVLE)

CMOV(CMOVNA)

CMOV(CMOVNAE)

CMOV(CMOVNB)

CMOV(CMOVNBE)

CMOV(CMOVNC)

CMOV(CMOVNE)

CMOV(CMOVNG)

CMOV(CMOVNGE)

CMOV(CMOVNL)

CMOV(CMOVNLE)

CMOV(CMOVNO)

CMOV(CMOVNP)

CMOV(CMOVNS)

CMOV(CMOVNZ)

CMOV(CMOVO)

CMOV(CMOVP)

CMOV(CMOVPE)

CMOV(CMOVPO)

CMOV(CMOVS)

CMOV(CMOVZ)

#undef CMOV

//SSE2 instructions

InstructionStart(MOVUPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd(MOVUPS)

InstructionStart(MOVAPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd(MOVAPS)

InstructionStart(MOVHPD, 16, 9, AffectOp1)

FormatRegAddr(EncodeModRM\_RegRM)

FormatAddrReg(EncodeModRM\_RegRM)

InstructionEnd(MOVHPD)

InstructionStart(MOVHLPS, 16, 3, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

InstructionEnd(MOVHLPS)

InstructionStart(MOVLHPS, 16, 3, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

InstructionEnd(MOVLHPS)

InstructionStart(SHUFPS, 16, 9, AffectOp1)

Format2RegImm8(EncodeModRM\_2Reg)

FormatRegAddrImm8(EncodeModRM\_RegRM)

InstructionEnd(SHUFPS)

InstructionStart(SHUFPD, 16, 10, AffectOp1)

Format2RegImm8(EncodeModRM\_2Reg)

FormatRegAddrImm8(EncodeModRM\_RegRM)

InstructionEnd(SHUFPD)

InstructionStart(PSHUFD, 16, 10, AffectOp1)

Format2RegImm8(EncodeModRM\_2Reg)

FormatRegAddrImm8(EncodeModRM\_RegRM)

InstructionEnd(PSHUFD)

InstructionStart(CVTPD2PS, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(CVTPD2PS)

InstructionStart(CVTDQ2PS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(CVTDQ2PS)

InstructionStart(CVTTPS2DQ, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(CVTTPS2DQ)

InstructionStart(CVTTPD2DQ, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(CVTTPD2DQ)

InstructionStart(ANDPD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd(ANDPD)

InstructionStart(PXOR, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd(PXOR)

InstructionStart(DIVPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(DIVPS)

InstructionStart(DIVPD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(DIVPD)

InstructionStart(SQRTPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(SQRTPS)

InstructionStart(SQRTPD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(SQRTPD)

InstructionStart(ADDPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(ADDPS)

InstructionStart(ADDPD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(ADDPD)

InstructionStart(PADDD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd(PADDD)

InstructionStart(SUBPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(SUBPS)

InstructionStart(SUBPD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(SUBPD)

InstructionStart(PSUBD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(PSUBD)

InstructionStart(MULPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(MULPS)

InstructionStart(MULPD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(MULPD)

InstructionStart(PMULUDQ, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(PMULUDQ)

InstructionStart(PSRLDQ, 16, 9, AffectOp1)

FormatRegImm8(EncodeModRM\_ByteReg<3>)

InstructionEnd(PSRLDQ)

InstructionStart(PUNPCKLDQ, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(PUNPCKLDQ)

InstructionStart(MINPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(MINPS)

InstructionStart(MAXPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(MAXPS)

InstructionStart(MINPD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(MINPD)

InstructionStart(MAXPD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(MAXPD)

enum CMP\_IMM8

{

EQ,

LT,

LE,

UNORD,

NEQ,

NLT,

NLE,

ORD

};

InstructionStart(CMPPS, 16, 8, AffectOp1)

Format2RegImm8(EncodeModRM\_2Reg)

FormatRegAddrImm8(EncodeModRM\_RegRM)

InstructionEnd(CMPPS)

InstructionStart(CMPPD, 16, 9, AffectOp1)

Format2RegImm8(EncodeModRM\_2Reg)

FormatRegAddrImm8(EncodeModRM\_RegRM)

InstructionEnd(CMPPD)

InstructionStart(PCMPGTD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(PCMPGTD)

InstructionStart(PCMPEQD, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(PCMPEQD)

InstructionStart(ANDPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd(ANDPS)

InstructionStart(ORPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(ORPS)

InstructionStart(PAND, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(PAND)

InstructionStart(ANDNPS, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

FormatRegPtr(EncodeModRM\_RegPtr)

InstructionEnd(ANDNPS)

InstructionStart(ANDNPD, 16, 8, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(ANDNPD)

InstructionStart(POR, 16, 9, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

FormatRegAddr(EncodeModRM\_RegRM)

InstructionEnd(POR)

InstructionStart(MOVMSKPS, 4|16, 3, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

InstructionEnd(MOVMSKPS)

InstructionStart(MOVMSKPD, 4 | 16, 4, AffectOp1)

Format2Reg(EncodeModRM\_2Reg)

InstructionEnd(MOVMSKPD)

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if ENABLE\_NATIVE\_CODEGEN

#include "..\BackEnd\i386\Reg.h"

static const BYTE RegEncode[] =

{

#define REGDAT(Name, Listing, Encoding, ...) Encoding,

#include "..\BackEnd\i386\RegList.h"

#undef REGDAT

};

#if DBG\_DUMP || ENABLE\_DEBUG\_CONFIG\_OPTIONS

extern wchar\_t const \* const RegNamesW[];

#endif

#include "AsmJsInstructionTemplate.h"

namespace Js

{

// Mask of Registers that can be saved through function calls

const uint MaskNonVolatileReg = 1 << RegEBX | 1 << RegEDI | 1 << RegESI;

// Reserved RegEDI for ArrayBuffer length

const RegNum ModuleEnvReg = RegEDI;

const RegNum ArrayBufferReg = RegESI;

// Registers that can't be chosen for general purposes

const uint MaskUnavailableReg = 1 << RegESP | 1 << RegEBP | 1 << ModuleEnvReg | 1 << ArrayBufferReg | 1 << RegNOREG;

// Mask for Register in enum RegNum [EAX,ECX,EDX,EBX,ESI,EDI]

const uint Mask32BitsReg = ( ( 1 << ( FIRST\_FLOAT\_REG ) ) - 1 ) & ~MaskUnavailableReg ;

// Mask for Register in enum RegNum [EAX,ECX,EDX,EBX] aka [al,cl,dl,bl]

const uint Mask8BitsReg = Mask32BitsReg &~(1<<RegEBP|1<<RegESP|1<<RegESI|1<<RegEDI);

// Mask for Register in enum RegNum [XMM0,XMM1,XMM2,XMM3,XMM4,XMM5,XMM6,XMM7]

const uint Mask64BitsReg = ((1 << (FIRST\_FLOAT\_REG+XMM\_REGCOUNT))-1) & ~MaskUnavailableReg & ~Mask32BitsReg;

// Template version to access register mask

template<typename T> uint GetRegMask();

template<> uint GetRegMask<int>() { return Mask32BitsReg; }

template<> uint GetRegMask<double>() { return Mask64BitsReg; }

template<> uint GetRegMask<float>() { return Mask64BitsReg; }

template<> uint GetRegMask<AsmJsSIMDValue>() { return Mask64BitsReg; }

// Template version to access first register available

template<typename T> RegNum GetFirstReg();

template<> RegNum GetFirstReg<int>() { return FIRST\_INT\_REG; }

template<> RegNum GetFirstReg<double>() { return FIRST\_FLOAT\_REG; }

template<> RegNum GetFirstReg<float>() { return FIRST\_FLOAT\_REG; }

template<> RegNum GetFirstReg<AsmJsSIMDValue>() { return FIRST\_FLOAT\_REG; }

// Returns the last register available + 1, forms an upper bound [GetFirstReg, GetLastReg[

template<typename T> RegNum GetLastReg() { return RegNum(GetFirstReg<T>()+8); }

struct InternalCallInfo

{

// size in bytes of arguments

int argByteSize;

int nextArgIndex;

int currentOffset;

InternalCallInfo\* next;

};

struct X86TemplateData

{

private:

InternalCallInfo\* mCallInfoList;

// Bit vector : 1 means a useful information is known for this RegNum.

// Never set an unavailable register flag to 1

int mAnyStackSaved;

// Stack offset saved for registers

int mRegisterStackOffsetSaved[RegNumCount];

// Value range [0,8[ add GetFirstReg() for RegNum

RegNum mNext32BitsReg, mNext64BitsReg;

// Template version to access the Next Register

template<typename T> RegNum GetNextRegister();

template<typename T> void SetNextRegister(RegNum reg);

int mBaseOffset;

int mScriptContextOffSet;

int mModuleSlotOffset;

int mModuleEnvOffset;

int mArrayBufferOffSet;

int mArraySizeOffset;

// Applies the register choosing algorithm and returns it

template<typename T> RegNum GetNextReg(RegNum reg);

template<> RegNum GetNextReg<int>(RegNum reg)

{

return RegNum((reg + 1) % GetLastReg<int>());

}

template<> RegNum GetNextReg<double>(RegNum reg)

{

RegNum nextReg = RegNum((reg + 1) % GetLastReg<double>());

if (nextReg < GetFirstReg<double>())

{

return RegNum(GetFirstReg<double>());

}

return nextReg;

}

template<typename T> RegNum NextReg(const int registerRestriction)

{

RegNum reg = GetNextRegister<T>();

const uint unavailable = registerRestriction | MaskUnavailableReg;

Assert( unavailable != GetRegMask<T>() );

if( (1<<reg) & unavailable )

{

while( (1<<reg) & unavailable )

{

reg = GetNextReg<T>(reg);

}

Assert( !(1 << reg & unavailable) );

return reg; // do not change the next register

}

RegNum next = reg;

do

{

next = GetNextReg<T>(next);

} while( ( 1 << next ) & MaskUnavailableReg );

SetNextRegister<T>( next );

Assert( !(1 << reg & unavailable) );

return reg;

}

public:

X86TemplateData()

{

Assert( !( (1<<GetFirstReg<int>()) & MaskUnavailableReg ) );

Assert(!((1 << GetFirstReg<double>()) & MaskUnavailableReg));

Assert(!((1 << GetFirstReg<float>()) & MaskUnavailableReg));

mNext32BitsReg = GetFirstReg<int>();

mNext64BitsReg = GetFirstReg<double>(); // it is the same for float

mAnyStackSaved = 0;

mCallInfoList = nullptr;

for (int i = 0; i < RegNumCount ; i++)

{

mRegisterStackOffsetSaved[i] = 0;

}

}

~X86TemplateData()

{

Assert( !mCallInfoList );

}

InternalCallInfo\* GetInternalCallInfo() const

{

return mCallInfoList;

}

void StartInternalCall( int argSizeByte )

{

InternalCallInfo\* info = HeapNew( InternalCallInfo );

info->argByteSize = argSizeByte;

info->currentOffset = 0;

info->nextArgIndex = 1;

info->next = mCallInfoList;

mCallInfoList = info;

}

void InternalCallDone()

{

Assert( mCallInfoList );

Assert( mCallInfoList->currentOffset + MachPtr == mCallInfoList->argByteSize );

InternalCallInfo\* next = mCallInfoList->next;

HeapDelete( mCallInfoList );

mCallInfoList = next;

}

// Tells this register is holding the content located at the stackOffset

void SetStackInfo( RegNum reg, int stackOffset )

{

Assert( !( 1 << reg & MaskUnavailableReg ) );

mRegisterStackOffsetSaved[reg] = stackOffset;

mAnyStackSaved |= 1 << reg;

}

// Call when register content is data dependent

void InvalidateReg( RegNum reg )

{

mAnyStackSaved &= ~( 1 << reg );

}

void InvalidateAllVolatileReg()

{

mAnyStackSaved &= MaskNonVolatileReg;

}

void InvalidateAllReg()

{

mAnyStackSaved = 0;

}

// Call when stack value has changed

void OverwriteStack( int stackOffset )

{

if( mAnyStackSaved )

{

// check all register with a stack offset saved

int stackSavedReg = mAnyStackSaved;

int reg = 0;

while( stackSavedReg )

{

// skip reg with no stack info

while( !(stackSavedReg & 1) )

{

stackSavedReg >>= 1;

++reg;

}

// invalidate register with this stack location

if( mRegisterStackOffsetSaved[reg] == stackOffset )

{

InvalidateReg( RegNum( reg ) );

}

// next register

stackSavedReg >>= 1;

++reg;

}

}

}

// Gets a register to use

// registerRestriction : bit vector, 1 means the register cannot be chosen

template<typename T> RegNum GetReg(const int registerRestriction = 0)

{

CompileAssert( sizeof(T) == 4 || sizeof(T) == 8 );

const int mask = GetRegMask<T>() & ~registerRestriction;

int stackSavedReg = mAnyStackSaved & mask;

// No more register available

if( stackSavedReg == mask )

{

RegNum reg = NextReg<T>(registerRestriction);

Assert( !(1 << reg & registerRestriction) );

return reg;

}

// making sure we don't choose the unavailable registers

stackSavedReg |= MaskUnavailableReg|registerRestriction;

int reg = GetFirstReg<T>();

stackSavedReg >>= reg;

// will always find a value under these conditions

while( 1 )

{

// if the register hold no useful info, return it

if( !( stackSavedReg & 1 ) )

{

Assert( !(1 << reg & registerRestriction) );

return RegNum( reg );

}

stackSavedReg >>= 1;

++reg;

}

}

// Gets a register to use

// registerRestriction : bit vector, 1 means the register cannot be chosen

template<> RegNum GetReg<float>(const int registerRestriction)

{

const int mask = GetRegMask<double>() & ~registerRestriction;

int stackSavedReg = mAnyStackSaved & mask;

// No more register available

if (stackSavedReg == mask)

{

RegNum reg = NextReg<double>(registerRestriction);

Assert(!(1 << reg & registerRestriction));

return reg;

}

// making sure we don't choose the unavailable registers

stackSavedReg |= MaskUnavailableReg | registerRestriction;

int reg = GetFirstReg<double>();

stackSavedReg >>= reg;

// will always find a value under these conditions

while (1)

{

// if the register hold no useful info, return it

if (!(stackSavedReg & 1))

{

Assert(!(1 << reg & registerRestriction));

return RegNum(reg);

}

stackSavedReg >>= 1;

++reg;

}

}

template<> RegNum GetReg<AsmJsSIMDValue>(const int registerRestriction)

{

return GetReg<float>(registerRestriction);

}

// Search for a register already holding the value at this location

template<typename T> bool FindRegWithStackOffset( RegNum& outReg, int stackOffset, int registerRestriction = 0 )

{

CompileAssert( sizeof(T) == 4 || sizeof(T) == 8 || sizeof(T) == 16);

int stackSavedReg = mAnyStackSaved & GetRegMask<T>() & ~registerRestriction;

if( stackSavedReg )

{

int reg = GetFirstReg<T>();

stackSavedReg >>= reg;

while( stackSavedReg )

{

// skip reg with no stack info

while( !(stackSavedReg & 1) )

{

stackSavedReg >>= 1;

++reg;

}

// invalidate register with this stack location

if( mRegisterStackOffsetSaved[reg] == stackOffset )

{

outReg = RegNum( reg );

return true;

}

// next register

stackSavedReg >>= 1;

++reg;

}

}

return false;

}

void SetBaseOffset(int baseOffSet)

{

// We subtract with the baseoffset as the layout of the stack has changed from the interpreter

// Assume Stack is growing downwards

// Interpreter - Stack is above EBP and offsets are positive

// TJ - Stack is below EBP and offsets are negative

mBaseOffset = baseOffSet;

mModuleSlotOffset = AsmJsJitTemplate::Globals::ModuleSlotOffset - mBaseOffset;

mModuleEnvOffset = AsmJsJitTemplate::Globals::ModuleEnvOffset - mBaseOffset;

mArrayBufferOffSet = AsmJsJitTemplate::Globals::ArrayBufferOffset - mBaseOffset;

mArraySizeOffset = AsmJsJitTemplate::Globals::ArraySizeOffset - mBaseOffset;

mScriptContextOffSet = AsmJsJitTemplate::Globals::ScriptContextOffset - mBaseOffset;

}

int GetBaseOffSet()

{

return mBaseOffset;

}

int GetModuleSlotOffset()

{

return mModuleSlotOffset;

}

int GetModuleEnvOffset()

{

return mModuleEnvOffset;

}

int GetArrayBufferOffset()

{

return mArrayBufferOffSet;

}

int GetArraySizeOffset()

{

return mArraySizeOffset;

}

int GetScriptContextOffset()

{

return mScriptContextOffSet;

}

const int GetCalleSavedRegSizeInByte()

{

//EBX,ESI,EDI

return 3 \* sizeof(void\*);

}

const int GetEBPOffsetCorrection()

{

//We computed the offset in BCG adjusting for push ebp and ret address

return 2 \* sizeof(void\*);

}

};

template<> RegNum X86TemplateData::GetNextRegister<int>() { return mNext32BitsReg; }

template<> RegNum X86TemplateData::GetNextRegister<double>() { return mNext64BitsReg; }

template<> void X86TemplateData::SetNextRegister<int>(RegNum reg) { mNext32BitsReg = reg; }

template<> void X86TemplateData::SetNextRegister<double>(RegNum reg) { mNext64BitsReg = reg; }

struct ReturnContent

{

union

{

int intVal;

double doubleVal;

};

template<typename T> T GetReturnVal()const;

#if DBG\_DUMP

template<typename T> void Print()const;

#endif

};

template<> int ReturnContent::GetReturnVal<int>()const

{

return intVal;

}

template<> float ReturnContent::GetReturnVal<float>()const

{

return (float)doubleVal;

}

template<> double ReturnContent::GetReturnVal<double>()const

{

return doubleVal;

}

#if DBG\_DUMP

template<> void ReturnContent::Print<int>()const

{

Output::Print( L" = %d", intVal );

}

template<> void ReturnContent::Print<double>()const

{

Output::Print( L" = %.4f", doubleVal );

}

template<> void ReturnContent::Print<float>()const

{

Output::Print( L" = %.4f", doubleVal );

}

int AsmJsCallDepth = 0;

#endif

uint CallLoopBody(JavascriptMethod address, ScriptFunction\* function, Var frameAddress)

{

void \*savedEsp = NULL;

\_\_asm

{

// Save ESP

mov savedEsp, esp

// Add an extra 4-bytes to the stack since we'll be pushing 3 arguments

push eax

}

uint newOffset = (uint)address(function, CallInfo(CallFlags\_InternalFrame, 1), frameAddress);

\_asm

{

// Restore ESP

mov esp, savedEsp

}

return newOffset;

}

uint DoLoopBodyStart(Js::ScriptFunction\* function,Var ebpPtr,uint32 loopNumber)

{

FunctionBody\* fn = function->GetFunctionBody();

Assert(loopNumber < fn->GetLoopCount());

Js::LoopHeader \*loopHeader = fn->GetLoopHeader(loopNumber);

Js::LoopEntryPointInfo \* entryPointInfo = loopHeader->GetCurrentEntryPointInfo();

ScriptContext\* scriptContext = fn->GetScriptContext();

// If we have JITted the loop, call the JITted code

if (entryPointInfo != NULL && entryPointInfo->IsCodeGenDone())

{

#if DBG\_DUMP

if (PHASE\_TRACE1(Js::JITLoopBodyPhase) && CONFIG\_FLAG(Verbose))

{

fn->DumpFunctionId(true);

Output::Print(L": %-20s LoopBody Execute Loop: %2d\n", fn->GetDisplayName(), loopNumber);

Output::Flush();

}

loopHeader->nativeCount++;

#endif

#ifdef BGJIT\_STATS

entryPointInfo->MarkAsUsed();

#endif

Assert(entryPointInfo->address);

uint newOffset = CallLoopBody((JavascriptMethod)entryPointInfo->address, function, ebpPtr);

ptrdiff\_t value = NULL;

fn->GetAsmJsFunctionInfo()->mbyteCodeTJMap->TryGetValue(newOffset, &value);

Assert(value != NULL); // value cannot be null

BYTE\* newAddress = fn->GetAsmJsFunctionInfo()->mTJBeginAddress + value;

Assert(newAddress);

return (uint)newAddress;

}

// interpreCount for loopHeader is incremented before calling DoLoopBody

const uint loopInterpretCount = fn->GetLoopInterpretCount(loopHeader);

if (loopHeader->interpretCount > loopInterpretCount)

{

if (!fn->DoJITLoopBody())

{

return 0;

}

// If the job is not scheduled then we need to schedule it now.

// It is possible a job was scheduled earlier and we find ourselves looking at the same entry point

// again. For example, if the function with the loop was JITed and bailed out then as we finish

// the call in the interpreter we might encounter a loop for which we had scheduled a JIT job before

// the function was initially scheduled. In such cases, that old JIT job will complete. If it completes

// successfully then we can go ahead and use it. If it fails then it will eventually revert to the

// NotScheduled state. Since transitions from NotScheduled can only occur on the main thread,

// by checking the state we are safe from racing with the JIT thread when looking at the other fields

// of the entry point.

if (entryPointInfo != NULL && entryPointInfo->IsNotScheduled())

{

entryPointInfo->SetIsAsmJSFunction(true);

entryPointInfo->SetIsTJMode(true);

GenerateLoopBody(scriptContext->GetNativeCodeGenerator(), fn, loopHeader, entryPointInfo, fn->GetLocalsCount(), &ebpPtr);

//reset InterpretCount

loopHeader->interpretCount = 0;

}

}

return 0;

}

// Function memory allocation should be done the same way as

// void InterpreterStackFrame::AlignMemoryForAsmJs() (InterpreterStackFrame.cpp)

// update any changes there

void AsmJsCommonEntryPoint(Js::ScriptFunction\* func, void\* savedEbpPtr)

{

int savedEbp = (int)savedEbpPtr;

FunctionBody\* body = func->GetFunctionBody();

Js::FunctionEntryPointInfo \* entryPointInfo = body->GetDefaultFunctionEntryPointInfo();

//CodeGenDone status is set by TJ

if ((entryPointInfo->IsNotScheduled() || entryPointInfo->IsCodeGenDone()) && !PHASE\_OFF(BackEndPhase, body) && !PHASE\_OFF(FullJitPhase, body))

{

if (entryPointInfo->callsCount < 255)

{

++entryPointInfo->callsCount;

}

const int minTemplatizedJitRunCount = (int)CONFIG\_FLAG(MinTemplatizedJitRunCount);

if ((entryPointInfo->callsCount >= minTemplatizedJitRunCount || body->IsHotAsmJsLoop()))

{

if (PHASE\_TRACE1(AsmjsEntryPointInfoPhase))

{

Output::Print(L"Scheduling %s For Full JIT at callcount:%d\n", body->GetDisplayName(), entryPointInfo->callsCount);

}

GenerateFunction(body->GetScriptContext()->GetNativeCodeGenerator(), body, func);

}

}

void\* constTable = body->GetConstTable();

constTable = (void\*)(((Var\*)constTable)+AsmJsFunctionMemory::RequiredVarConstants-1);

AsmJsFunctionInfo\* asmInfo = body->GetAsmJsFunctionInfo();

const int intConstCount = asmInfo->GetIntConstCount();

const int doubleConstCount = asmInfo->GetDoubleConstCount();

const int floatConstCount = asmInfo->GetFloatConstCount();

const int simdConstCount = asmInfo->GetSimdConstCount();

// Offset of doubles from (double\*)m\_localSlot

const int intOffsets = asmInfo->GetIntByteOffset() / sizeof(int);

const int doubleOffsets = asmInfo->GetDoubleByteOffset() / sizeof(double);

const int floatOffset = asmInfo->GetFloatByteOffset() / sizeof(float);

const int simdByteOffset = asmInfo->GetSimdByteOffset(); // in bytes

// (2\*sizeof(Var)) -- push ebp and ret address

//sizeof(ScriptFunction\*) -- this is the argument passed to the TJ function

int argoffset = (2\*sizeof(Var)) + sizeof(ScriptFunction\*);

argoffset = argoffset + savedEbp ;

// initialize argument location

int\* intArg;

double\* doubleArg;

float\* floatArg;

AsmJsSIMDValue\* simdArg;

// setup stack memory

FrameDisplay\* frame = func->GetEnvironment();

Var moduleEnv = frame->GetItem(0);

Var\* arrayBufferVar = (Var\*)frame->GetItem(0) + AsmJsModuleMemory::MemoryTableBeginOffset;

int arraySize = 0;

BYTE\* arrayPtr = nullptr;

if (\*arrayBufferVar && JavascriptArrayBuffer::Is(\*arrayBufferVar))

{

JavascriptArrayBuffer\* arrayBuffer = \*(JavascriptArrayBuffer\*\*)arrayBufferVar;

arrayPtr = arrayBuffer->GetBuffer();

arraySize = arrayBuffer->GetByteLength();

}

Var\* m\_localSlots;

int\* m\_localIntSlots;

double\* m\_localDoubleSlots;

float\* m\_localFloatSlots;

AsmJsSIMDValue\* m\_localSimdSlots;

#if DBG\_DUMP

const bool tracingFunc = PHASE\_TRACE( AsmjsFunctionEntryPhase, body );

if( tracingFunc )

{

if( AsmJsCallDepth )

{

Output::Print( L"%\*c", AsmJsCallDepth,' ');

}

Output::Print( L"Executing function %s(", body->GetDisplayName());

++AsmJsCallDepth;

}

#endif

// two args i.e. (ScriptFunction and savedEbp) + 2\* (void\*) i.e.(ebp + return address)

int beginSlotOffset = sizeof(ScriptFunction\*) + sizeof(void\*) + 2 \* sizeof(void\*);

\_\_asm

{

mov eax, ebp

add eax, beginSlotOffset

mov m\_localSlots,eax

};

{

const ArgSlot argCount = asmInfo->GetArgCount();

m\_localSlots[AsmJsFunctionMemory::ModuleEnvRegister] = moduleEnv;

m\_localSlots[AsmJsFunctionMemory::ArrayBufferRegister] = (Var)arrayPtr;

m\_localSlots[AsmJsFunctionMemory::ArraySizeRegister] = (Var)arraySize;

m\_localSlots[AsmJsFunctionMemory::ScriptContextBufferRegister] = body->GetScriptContext();

m\_localIntSlots = ((int\*)m\_localSlots) + intOffsets;

memcpy\_s(m\_localIntSlots, intConstCount\*sizeof(int), constTable, intConstCount\*sizeof(int));

constTable = (void\*)(((int\*)constTable) + intConstCount);

m\_localFloatSlots = ((float\*)m\_localSlots) + floatOffset;

memcpy\_s(m\_localFloatSlots, floatConstCount\*sizeof(float), constTable, floatConstCount\*sizeof(float));

constTable = (void\*)(((float\*)constTable) + floatConstCount);

m\_localDoubleSlots = ((double\*)m\_localSlots) + doubleOffsets;

memcpy\_s(m\_localDoubleSlots, doubleConstCount\*sizeof(double), constTable, doubleConstCount\*sizeof(double));

if (func->GetScriptContext()->GetConfig()->IsSimdjsEnabled())

{

// Copy SIMD constants to TJ stack frame. No data alignment.

constTable = (void\*)(((double\*)constTable) + doubleConstCount);

m\_localSimdSlots = (AsmJsSIMDValue\*)((char\*)m\_localSlots + simdByteOffset);

memcpy\_s(m\_localSimdSlots, simdConstCount\*sizeof(AsmJsSIMDValue), constTable, simdConstCount\*sizeof(AsmJsSIMDValue));

}

intArg = m\_localIntSlots + intConstCount;

doubleArg = m\_localDoubleSlots + doubleConstCount;

floatArg = m\_localFloatSlots + floatConstCount;

simdArg = m\_localSimdSlots + simdConstCount;

for(ArgSlot i = 0; i < argCount; i++ )

{

if(asmInfo->GetArgType(i).isInt())

{

\_\_asm

{

mov eax, argoffset

mov eax, [eax]

mov ecx, intArg

mov [ecx], eax

};

#if DBG\_DUMP

if( tracingFunc )

{

Output::Print( L" %d%c", \*intArg, i+1 < argCount ? ',':' ');

}

#endif

++intArg;

argoffset += sizeof( int );

}

else if (asmInfo->GetArgType(i).isFloat())

{

\_\_asm

{

mov eax, argoffset

movss xmm0, [eax]

mov eax, floatArg

movss[eax], xmm0

};

#if DBG\_DUMP

if (tracingFunc)

{

Output::Print(L" %.4f%c", \*floatArg, i + 1 < argCount ? ',' : ' ');

}

#endif

++floatArg;

argoffset += sizeof(float);

}

else if (asmInfo->GetArgType(i).isDouble())

{

\_\_asm

{

mov eax, argoffset

movsd xmm0, [eax]

mov eax, doubleArg

movsd [eax], xmm0

};

#if DBG\_DUMP

if( tracingFunc )

{

Output::Print( L" %.4f%c", \*doubleArg, i+1 < argCount ? ',':' ');

}

#endif

++doubleArg;

argoffset += sizeof( double );

}

else if (asmInfo->GetArgType(i).isSIMD())

{

\_\_asm

{

mov eax, argoffset

movups xmm0, [eax]

mov eax, simdArg

movups[eax], xmm0

};

#if DBG\_DUMP

if (tracingFunc)

{

switch (asmInfo->GetArgType(i).which())

{

case AsmJsType::Int32x4:

Output::Print(L" I4(%d, %d, %d, %d)", \

simdArg->i32[SIMD\_X], simdArg->i32[SIMD\_Y], simdArg->i32[SIMD\_Z], simdArg->i32[SIMD\_W]);

break;

case AsmJsType::Float32x4:

Output::Print(L" F4(%.4f, %.4f, %.4f, %.4f)", \

simdArg->f32[SIMD\_X], simdArg->f32[SIMD\_Y], simdArg->f32[SIMD\_Z], simdArg->f32[SIMD\_W]);

break;

case AsmJsType::Float64x2:

Output::Print(L" D2(%.4f, %.4f)%c", \

simdArg->f64[SIMD\_X], simdArg->f64[SIMD\_Y]);

break;

}

Output::Print(L"%c", i + 1 < argCount ? ',' : ' ');

}

#endif

++simdArg;

argoffset += sizeof(AsmJsSIMDValue);

}

}

}

#if DBG\_DUMP

if( tracingFunc )

{

Output::Print( L"){\n");

}

#endif

}

#if DBG\_DUMP

void AsmJSCommonCallHelper(Js::ScriptFunction\* func)

{

FunctionBody\* body = func->GetFunctionBody();

AsmJsFunctionInfo\* asmInfo = body->GetAsmJsFunctionInfo();

const bool tracingFunc = PHASE\_TRACE(AsmjsFunctionEntryPhase, body);

if (tracingFunc)

{

--AsmJsCallDepth;

if (AsmJsCallDepth)

{

Output::Print(L"%\*c}", AsmJsCallDepth, ' ');

}

else

{

Output::Print(L"}");

}

if (asmInfo->GetReturnType() != AsmJsRetType::Void)

{

//returnContent.Print<T>();

}

Output::Print(L";\n");

}

}

#endif

Var ExternalCallHelper( JavascriptFunction\* function, int nbArgs, Var\* paramsAddr )

{

int flags = CallFlags\_Value;

Arguments args(CallInfo((CallFlags)flags, (ushort)nbArgs), paramsAddr);

return JavascriptFunction::CallFunction<true>(function, function->GetEntryPoint(), args);

}

namespace AsmJsJitTemplate

{

const int Globals::ModuleSlotOffset = (AsmJsFunctionMemory::ModuleSlotRegister + Globals::StackVarCount)\*sizeof(Var);

const int Globals::ModuleEnvOffset = (AsmJsFunctionMemory::ModuleEnvRegister + Globals::StackVarCount)\*sizeof(Var);

const int Globals::ArrayBufferOffset = (AsmJsFunctionMemory::ArrayBufferRegister + Globals::StackVarCount)\*sizeof(Var);

const int Globals::ArraySizeOffset = (AsmJsFunctionMemory::ArraySizeRegister + Globals::StackVarCount)\*sizeof(Var);

const int Globals::ScriptContextOffset = (AsmJsFunctionMemory::ScriptContextBufferRegister + Globals::StackVarCount)\*sizeof(Var);

#if DBG\_DUMP

FunctionBody\* Globals::CurrentEncodingFunction = nullptr;

#endif

// Jump relocation : fix the jump offset for a later point in the same template

struct JumpRelocation

{

// buffer : where the instruction will be encoded

// size : address of a variable tracking the instructions size encoded after the jump

JumpRelocation( BYTE\* buffer, int\* size )

{

#if DBG

mRelocDone = false;

mEncodingImmSize = -1;

#endif

Init( buffer, size );

}

// Default Constructor, must call Init before using

JumpRelocation()

{

#if DBG

mRelocDone = false;

mEncodingImmSize = -1;

#endif

}

#if DBG

~JumpRelocation()

{

// Make sure the relocation is done when destruction the object

Assert( mRelocDone );

}

#endif

void Init( BYTE\* buffer, int\* size )

{

#if DBG

// this cannot be called twice

Assert( mEncodingImmSize == -1 );

#endif

mBuffer = buffer;

mSize = size;

mInitialSize = \*mSize;

}

// to be called right after encoding a jump

void JumpEncoded( const EncodingInfo& info )

{

#if DBG

// this cannot be called twice

Assert( mEncodingImmSize == -1 );

#endif

const int curSize = \*mSize;

// move the buffer to the point where we need to fix the value

mBuffer += curSize - mInitialSize - info.immSize;

mInitialSize = curSize;

#if DBG

mEncodingImmSize = info.immSize;

#endif

}

// use when only 1 Byte was allocated

template<typename OffsetType>

void ApplyReloc()

{

#if DBG

Assert( mEncodingImmSize == sizeof(OffsetType) );

mRelocDone = true;

#endif

const int relocSize = \*mSize - mInitialSize;

// if we encoded only 1 byte, make sure it fits

Assert( sizeof(OffsetType) != 1 || FitsInByte( relocSize ) );

\*(OffsetType\*)mBuffer = (OffsetType)relocSize;

}

#if DBG

bool mRelocDone;

int mEncodingImmSize;

#endif

BYTE\* mBuffer;

int\* mSize;

int mInitialSize;

};

#define GetTemplateData(context) ((X86TemplateData\*)context->GetTemplateData())

// Initialize template data

void\* InitTemplateData()

{

return HeapNew( X86TemplateData );

}

// Free template data for architecture specific

void FreeTemplateData( void\* userData )

{

HeapDelete( (X86TemplateData\*)userData );

}

// Typedef to map a type to an instruction

template<typename InstructionSize> struct InstructionBySize;

template<> struct InstructionBySize < int > { typedef MOV MoveInstruction; };

template<> struct InstructionBySize < double > { typedef MOVSD MoveInstruction; };

template<> struct InstructionBySize < float > { typedef MOVSS MoveInstruction; };

template<> struct InstructionBySize < AsmJsSIMDValue > { typedef MOVUPS MoveInstruction; };

namespace EncodingHelpers

{

// put the value on the stack into a register

template<typename RegisterSize>

RegNum GetStackReg( BYTE\*& buffer, X86TemplateData\* templateData, int varOffset, int &size, const int registerRestriction = 0 )

{

RegNum reg;

if( !templateData->FindRegWithStackOffset<RegisterSize>( reg, varOffset, registerRestriction ) )

{

reg = templateData->GetReg<RegisterSize>( registerRestriction );

size += InstructionBySize<RegisterSize>::MoveInstruction::EncodeInstruction<RegisterSize>( buffer, InstrParamsRegAddr( reg, RegEBP, varOffset ) );

templateData->SetStackInfo( reg, varOffset );

}

return reg;

}

// put the value of a register on the stack

template<typename RegisterSize>

int SetStackReg( BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, RegNum reg )

{

CompileAssert(sizeof(RegisterSize) == 4 || sizeof(RegisterSize) == 8);

templateData->OverwriteStack( targetOffset );

templateData->SetStackInfo( reg, targetOffset );

return InstructionBySize<RegisterSize>::MoveInstruction::EncodeInstruction<RegisterSize>( buffer, InstrParamsAddrReg( RegEBP, targetOffset, reg ) );

}

template<typename LaneType=int>

int SIMDSetStackReg(BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, RegNum reg)

{

CompileAssert(sizeof(LaneType) == 4 || sizeof(LaneType) == 8);

AssertMsg(((1<<reg) & GetRegMask<AsmJsSIMDValue>()), "Expecting XMM reg.");

// On a stack spill, we need to invalidate any registers holding lane values.

int laneOffset = 0;

while (laneOffset < sizeof(AsmJsSIMDValue))

{

templateData->OverwriteStack(targetOffset + laneOffset);

laneOffset += sizeof(LaneType);

}

templateData->SetStackInfo(reg, targetOffset);

return InstructionBySize<AsmJsSIMDValue>::MoveInstruction::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsAddrReg(RegEBP, targetOffset, reg));

}

/\*

Simply copy data from memory to memory.

TODO: Optimize to initialize in XMM reg and then store to mem.

\*/

template<typename LaneType>

int SIMDInitFromPrimitives(BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, int srcOffset1, int srcOffset2, int srcOffset3 = 0, int srcOffset4 = 0)

{

CompileAssert(sizeof(LaneType) == 4 || sizeof(LaneType) == 8);

int size = 0;

int laneOffset = 0;

RegNum reg;

targetOffset -= templateData->GetBaseOffSet();

srcOffset1 -= templateData->GetBaseOffSet();

srcOffset2 -= templateData->GetBaseOffSet();

srcOffset3 -= templateData->GetBaseOffSet();

srcOffset4 -= templateData->GetBaseOffSet();

// Since we overwrite all lanes, any register holding any lane value is invalidated.

reg = EncodingHelpers::GetStackReg<LaneType>(buffer, templateData, srcOffset1, size);

size += EncodingHelpers::SetStackReg<LaneType>(buffer, templateData, targetOffset + laneOffset, reg);

templateData->InvalidateReg(reg);

laneOffset += sizeof(LaneType);

reg = EncodingHelpers::GetStackReg<LaneType>(buffer, templateData, srcOffset2, size);

size += EncodingHelpers::SetStackReg<LaneType>(buffer, templateData, targetOffset + laneOffset, reg);

templateData->InvalidateReg(reg);

laneOffset += sizeof(LaneType);

if (laneOffset < sizeof(AsmJsSIMDValue))

{

reg = EncodingHelpers::GetStackReg<LaneType>(buffer, templateData, srcOffset3, size);

size += EncodingHelpers::SetStackReg<LaneType>(buffer, templateData, targetOffset + laneOffset, reg);

templateData->InvalidateReg(reg);

laneOffset += sizeof(LaneType);

reg = EncodingHelpers::GetStackReg<LaneType>(buffer, templateData, srcOffset4, size);

size += EncodingHelpers::SetStackReg<LaneType>(buffer, templateData, targetOffset + laneOffset, reg);

templateData->InvalidateReg(reg);

}

return size;

}

// Since SIMD data is unaligned, we cannot support "OP reg, [mem]" operations.

template <typename Operation, typename LaneType=int>

int SIMDUnaryOperation(BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, int srcOffset, int registerRestriction = 0)

{

int size = 0;

RegNum dstReg, srcReg;

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

// MOVUPS

srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset, size);

// Get a new reg for dst, and keep src reg alive

dstReg = templateData->GetReg<AsmJsSIMDValue>(1 << srcReg);

// OP reg1, reg2

size += Operation::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, srcReg));

// MOVUPS

size += EncodingHelpers::SIMDSetStackReg<LaneType>(buffer, templateData, targetOffset, dstReg);

return size;

}

template <typename Operation, typename LaneType = int>

int SIMDBinaryOperation(BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, int srcOffset1, int srcOffset2)

{

int size = 0;

RegNum srcReg1, srcReg2, dstReg;

targetOffset -= templateData->GetBaseOffSet();

srcOffset1 -= templateData->GetBaseOffSet();

srcOffset2 -= templateData->GetBaseOffSet();

// MOVUPS srcReg1, [srcOffset1]

srcReg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset1, size);

// MOVUPS srcReg2, [srcOffset2]

srcReg2 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset2, size);

// keep src regs alive

// MOVAPS dstReg, srcReg1

dstReg = templateData->GetReg<AsmJsSIMDValue>((1 << srcReg1) | (1 << srcReg2));

size += MOVAPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, srcReg1));

// OP dstReg, srcReg2

size += Operation::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, srcReg2));

// MOVUPS

size += EncodingHelpers::SIMDSetStackReg<LaneType>(buffer, templateData, targetOffset, dstReg);

return size;

}

// for CMP and Shuffle operations

template <typename Operation, typename LaneType = int>

int SIMDBinaryOperation(BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, int srcOffset1, int srcOffset2, byte imm8)

{

int size = 0;

RegNum srcReg1, srcReg2, dstReg;

targetOffset -= templateData->GetBaseOffSet();

srcOffset1 -= templateData->GetBaseOffSet();

srcOffset2 -= templateData->GetBaseOffSet();

// MOVUPS srcReg1, [srcOffset1]

srcReg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset1, size);

// MOVUPS srcReg2, [srcOffset2]

srcReg2 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset2, size);

// keep src regs alive

// MOVAPS dstReg, srcReg1

dstReg = templateData->GetReg<AsmJsSIMDValue>((1 << srcReg1) | (1 << srcReg2));

size += MOVAPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, srcReg1));

// OP dstReg, srcReg2, imm8

size += Operation::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2RegImm<byte>(dstReg, srcReg2, imm8));

// MOVUPS

size += EncodingHelpers::SIMDSetStackReg<LaneType>(buffer, templateData, targetOffset, dstReg);

return size;

}

template <typename Operation, typename LaneType>

RegNum SIMDRcpOperation(BYTE\*& buffer, X86TemplateData\* templateData, RegNum srcReg, void \*ones, int &size)

{

RegNum reg;

// MOVAPS reg, [mask]

reg = templateData->GetReg<AsmJsSIMDValue>(1 << srcReg);

size += MOVAPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegPtr(reg, ones));

// OP reg, srcReg

size += Operation::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(reg, srcReg));

return reg;

}

template <typename Operation, typename LaneType>

int SIMDLdLaneOperation(BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, int srcOffset, const int index, const bool reUseResult = true)

{

CompileAssert(sizeof(LaneType) == 4 || sizeof(LaneType) == 8);

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

RegNum srcReg, tmpReg;

int size = 0;

// MOVUPS

srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset, size);

// MOVAPS tmpReg, srcReg

tmpReg = templateData->GetReg<AsmJsSIMDValue>((1 << srcReg));

size += MOVAPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tmpReg, srcReg));

// PSRLDQ tmpREg, (index \* sizeof(lane))

size += PSRLDQ::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegImm<byte>(tmpReg, (byte)(sizeof(LaneType)\* index)));

templateData->OverwriteStack(targetOffset);

if (reUseResult)

{

// can re-use register for floats and doubles only.

templateData->SetStackInfo(tmpReg, targetOffset);

}

size += Operation::EncodeInstruction<LaneType>(buffer, InstrParamsAddrReg(RegEBP, targetOffset, tmpReg));

return size;

}

template <typename LaneType, typename ShufOperation = SHUFPS>

int SIMDSetLaneOperation(BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, int srcOffset, int valOffset, const int laneIndex)

{

CompileAssert(sizeof(LaneType) == 4);

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

valOffset -= templateData->GetBaseOffSet();

int size = 0;

RegNum srcReg, tmpReg, valReg;

// load regs

// MOVUPS srcReg, [src]

srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset, size);

// keep src alive

// MOVAPS tmpReg, srcReg

tmpReg = templateData->GetReg<AsmJsSIMDValue>(1 << srcReg);

size += MOVAPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tmpReg, srcReg));

// MOVSS valReg, [val] ; valReg is XMM

valReg = EncodingHelpers::GetStackReg<float>(buffer, templateData, valOffset, size, (1 << srcReg) | (1 << tmpReg));

if (laneIndex == 0)

{

// MOVSS tmpReg, valReg

// Note: we use MOVSS for both F4 and I4. MOVD sets upper bits to zero, MOVSS leaves them unmodified.

size += MOVSS::EncodeInstruction<LaneType>(buffer, InstrParams2Reg(tmpReg, valReg));

}

else if (laneIndex == 1 || laneIndex == 3)

{

// shuf, mov, shuf

byte shufMask;

shufMask = 0xE4; // 11 10 01 00

shufMask |= laneIndex; // 11 10 01 id

shufMask &= ~(0x03 << (laneIndex << 1)); // set 2 bits corresponding to lane index to 00

// shuf tempReg, tmpReg, shufMask

size += ShufOperation::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2RegImm<byte>(tmpReg, tmpReg, shufMask));

// MOVSS tmpReg, valReg

size += MOVSS::EncodeInstruction<LaneType>(buffer, InstrParams2Reg(tmpReg, valReg));

// shuf tempReg, tmpReg, shufMask

size += ShufOperation::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2RegImm<byte>(tmpReg, tmpReg, shufMask));

}

else

{

Assert(laneIndex == 2);

RegNum tmpReg2 = templateData->GetReg<AsmJsSIMDValue>((1 << srcReg) | (1 << tmpReg));

// MOVHLPS tmpReg2, tmpReg

size += MOVHLPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tmpReg2, tmpReg));

// MOVSS tmpReg2, valReg

size += MOVSS::EncodeInstruction<LaneType>(buffer, InstrParams2Reg(tmpReg2, valReg));

// MOVHLPS tmpReg, tmpReg2

size += MOVLHPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tmpReg, tmpReg2));

}

size += EncodingHelpers::SIMDSetStackReg<LaneType>(buffer, templateData, targetOffset, tmpReg);

return size;

}

template <>

int SIMDSetLaneOperation<double>(BYTE\*& buffer, X86TemplateData\* templateData, int targetOffset, int srcOffset, int valOffset, const int laneIndex)

{

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

valOffset -= templateData->GetBaseOffSet();

int size = 0;

RegNum srcReg, tmpReg, valReg;

// load regs

// MOVUPS srcReg, [src]

srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset, size);

// keep src alive

// MOVAPS tmpReg, srcReg

tmpReg = templateData->GetReg<AsmJsSIMDValue>(1 << srcReg);

size += MOVAPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tmpReg, srcReg));

if (laneIndex == 0)

{

// We have to load val to reg. MOVSD reg, [val] will zero upper bits.

// MOVSD valReg, [val]

valReg = EncodingHelpers::GetStackReg<double>(buffer, templateData, valOffset, size, (1 << srcReg) | (1 << tmpReg));

// MOVSD tmpReg, value

size += MOVSD::EncodeInstruction<double>(buffer, InstrParams2Reg(tmpReg, valReg));

}

else

{

Assert(laneIndex == 1);

// MOVHPD tmpReg, [val]

size += MOVHPD::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegAddr(tmpReg, RegEBP, valOffset));

}

// MOVUPS

size += EncodingHelpers::SIMDSetStackReg<double>(buffer, templateData, targetOffset, tmpReg);

return size;

}

// Retrieve the value of the array buffer and put it in a register to use

RegNum GetArrayBufferRegister( BYTE\*& buffer, TemplateContext context, int &size, const int registerRestriction = 0 )

{

return ArrayBufferReg;

}

// Retrieve the value of the module environment and put it in a register to use

RegNum GetModuleEnvironmentRegister( BYTE\*& buffer, TemplateContext context, int &size, const int registerRestriction = 0 )

{

return ModuleEnvReg;

}

// Retrieve the value of the script context and put it in a register to use

RegNum GetScriptContextRegister( BYTE\*& buffer, TemplateContext context, int &size, const int registerRestriction = 0 )

{

X86TemplateData\* templateData = GetTemplateData(context);

return GetStackReg<int>(buffer, GetTemplateData(context), templateData->GetScriptContextOffset(), size, registerRestriction);

}

// Encode a Compare instruction between a register and the array length : format cmp length, reg

int CompareRegisterToArrayLength( BYTE\*& buffer, TemplateContext context, RegNum reg, const int registerRestriction = 0 )

{

X86TemplateData\* templateData = GetTemplateData(context);

return CMP::EncodeInstruction<int>(buffer, InstrParamsAddrReg(RegEBP, templateData->GetArraySizeOffset(), reg));

}

// Encode a Compare instruction between an immutable value and the array length : format cmp length, imm

template<typename T>

int CompareImmutableToArrayLength( BYTE\*& buffer, TemplateContext context, T imm, const int registerRestriction = 0 )

{

X86TemplateData\* templateData = GetTemplateData(context);

return CMP::EncodeInstruction<int>(buffer, InstrParamsAddrImm<T>(RegEBP, templateData->GetArraySizeOffset(), imm));

}

// Encodes a short(1 Byte offset) jump instruction

template<typename JCC>

void EncodeShortJump( BYTE\*& buffer, JumpRelocation& reloc, int\* size )

{

Assert( size != nullptr );

reloc.Init( buffer, size );

EncodingInfo info;

\*size += JCC::EncodeInstruction<int8>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc.JumpEncoded( info );

}

template<typename Operation, typename OperationSize>

int CommutativeOperation( TemplateContext context, BYTE\*& buffer, int leftOffset, int rightOffset, int\* targetOffset = nullptr, RegNum\* outReg = nullptr, int registerRestriction = 0 )

{

X86TemplateData\* templateData = GetTemplateData( context );

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

\*targetOffset -= templateData->GetBaseOffSet();

RegNum reg1, reg2;

RegNum resultReg = RegNOREG;

const int reg1Found = templateData->FindRegWithStackOffset<OperationSize>( reg1, leftOffset, registerRestriction );

const int reg2Found = templateData->FindRegWithStackOffset<OperationSize>( reg2, rightOffset, registerRestriction );

int size = 0;

switch( reg1Found & ( reg2Found << 1 ) )

{

case 0: // none found

reg1 = templateData->GetReg<OperationSize>( registerRestriction );

size += InstructionBySize<OperationSize>::MoveInstruction::EncodeInstruction<OperationSize>(buffer, InstrParamsRegAddr(reg1, RegEBP, leftOffset));

if( leftOffset == rightOffset )

{

size += Operation::EncodeInstruction<OperationSize>( buffer, InstrParams2Reg( reg1, reg1 ) );

}

else

{

size += Operation::EncodeInstruction<OperationSize>(buffer, InstrParamsRegAddr(reg1, RegEBP, rightOffset));

}

resultReg = reg1;

break;

case 1: // found 1 and not 2

size += Operation::EncodeInstruction<OperationSize>(buffer, InstrParamsRegAddr(reg1, RegEBP, rightOffset));

resultReg = reg1;

break;

case 2: // found 2 and not 1

size += Operation::EncodeInstruction<OperationSize>(buffer, InstrParamsRegAddr(reg2, RegEBP, leftOffset));

resultReg = reg2;

break;

case 3: // found both

size += Operation::EncodeInstruction<OperationSize>( buffer, InstrParams2Reg( reg1, reg2 ) );

resultReg = reg1;

break;

default:

Assume(UNREACHED);

}

if( Operation::Flags & AffectOp1 )

{

templateData->InvalidateReg( resultReg );

}

if( targetOffset )

{

const int offset = \*targetOffset;

size += InstructionBySize<OperationSize>::MoveInstruction::EncodeInstruction<OperationSize>(buffer, InstrParamsAddrReg(RegEBP, offset, resultReg));

templateData->OverwriteStack( offset );

templateData->SetStackInfo( resultReg, offset );

}

if( outReg )

{

\*outReg = resultReg;

}

return size;

}

template<typename Operation, typename OperationSize>

int NonCommutativeOperation( TemplateContext context, BYTE\*& buffer, int leftOffset, int rightOffset, int\* targetOffset = nullptr, RegNum\* outReg = nullptr, int registerRestriction = 0 )

{

X86TemplateData\* templateData = GetTemplateData( context );

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg1, reg2;

const int reg1Found = templateData->FindRegWithStackOffset<OperationSize>( reg1, leftOffset, registerRestriction );

int size = 0;

if( !reg1Found )

{

reg1 = templateData->GetReg<OperationSize>( registerRestriction );

size += InstructionBySize<OperationSize>::MoveInstruction::EncodeInstruction<OperationSize>( buffer, InstrParamsRegAddr( reg1, RegEBP, leftOffset ) );

templateData->SetStackInfo( reg1, leftOffset );

}

const int reg2Found = templateData->FindRegWithStackOffset<OperationSize>( reg2, rightOffset, registerRestriction );

if( reg2Found )

{

size += Operation::EncodeInstruction<OperationSize>( buffer, InstrParams2Reg( reg1, reg2 ) );

}

else

{

size += Operation::EncodeInstruction<OperationSize>( buffer, InstrParamsRegAddr( reg1, RegEBP, rightOffset ) );

}

templateData->InvalidateReg( reg1 );

if( targetOffset )

{

int offset = \*targetOffset;

offset -= templateData->GetBaseOffSet();

size += InstructionBySize<OperationSize>::MoveInstruction::EncodeInstruction<OperationSize>( buffer, InstrParamsAddrReg( RegEBP, offset, reg1 ) );

templateData->OverwriteStack( offset );

templateData->SetStackInfo( reg1, offset );

}

if( outReg )

{

\*outReg = reg1;

}

return size;

}

int ReloadArrayBuffer(TemplateContext context, BYTE\*& buffer)

{

int size = 0;

if (!context->GetFunctionBody()->GetAsmJsFunctionInfo()->IsHeapBufferConst())

{

X86TemplateData\* templateData = GetTemplateData(context);

// mov buffer, [mod+bufferOffset]

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(ArrayBufferReg, ModuleEnvReg, AsmJsModuleMemory::MemoryTableBeginOffset));

// mov tmpReg, [buffer+lenOffset]

RegNum reg = templateData->GetReg<int>(1 << RegEAX);

templateData->InvalidateReg(reg);

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(reg, ArrayBufferReg, ArrayBuffer::GetByteLengthOffset()));

// mov [mod+offset], tmpReg

size += EncodingHelpers::SetStackReg<int>(buffer, templateData, templateData->GetArraySizeOffset(), reg);

// mov buffer, [buffer+buffOffset]

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(ArrayBufferReg, ArrayBufferReg, ArrayBuffer::GetBufferOffset()));

}

return size;

}

int CheckForArrayBufferDetached(TemplateContext context, BYTE\*& buffer)

{

int size = 0;

if (context->GetFunctionBody()->GetAsmJsFunctionInfo()->UsesHeapBuffer())

{

X86TemplateData\* templateData = GetTemplateData(context);

RegNum reg = templateData->GetReg<int>(1 << RegEAX);

templateData->InvalidateReg(reg);

// mov reg, [mod+bufferOffset]

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(reg, ModuleEnvReg, AsmJsModuleMemory::MemoryTableBeginOffset));

// mov reg, [reg+detachedOffset]

size += MOV::EncodeInstruction<int8>(buffer, InstrParamsRegAddr(reg, reg, ArrayBuffer::GetIsDetachedOffset()));

// test reg,reg

size += TEST::EncodeInstruction<int8>(buffer, InstrParams2Reg(reg, reg));

// JE Done

JumpRelocation relocDone;

EncodingHelpers::EncodeShortJump<JE>(buffer, relocDone, &size);

//call Js::Throw::OutOfMemory

int32 throwOOM = (int32)(void(\*)())Throw::OutOfMemory;

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(reg, throwOOM));

size += CALL::EncodeInstruction<int>(buffer, InstrParamsReg(reg));

// Done:

relocDone.ApplyReloc<int8>();

}

return size;

}

}

int Br::ApplyTemplate(TemplateContext context, BYTE\*& buffer, BYTE\*\* relocAddr, bool isBackEdge)

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

if (isBackEdge)

{

RegNum regInc = templateData->GetReg<int>(0);

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(regInc, (int32)context->GetFunctionBody()));

size += INC::EncodeInstruction<int>(buffer, InstrParamsAddr(regInc, context->GetFunctionBody()->GetAsmJsTotalLoopCountOffset()));

templateData->InvalidateReg(regInc);

}

\*relocAddr = buffer;

EncodingInfo info;

size += JMP::EncodeInstruction<int>( buffer, InstrParamsImm<int32>( 0 ), &info );

\*relocAddr += info.opSize;

return size;

}

int BrEq::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int leftOffset, int rightOffset, BYTE\*\* relocAddr, bool isBackEdge)

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

if (isBackEdge)

{

RegNum regInc = templateData->GetReg<int>(0);

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(regInc, (int32)context->GetFunctionBody()));

size += INC::EncodeInstruction<int>(buffer, InstrParamsAddr(regInc, context->GetFunctionBody()->GetAsmJsTotalLoopCountOffset()));

templateData->InvalidateReg(regInc);

}

RegNum reg1, reg2;

const int reg1Found = templateData->FindRegWithStackOffset<int>( reg1, leftOffset );

const int reg2Found = templateData->FindRegWithStackOffset<int>( reg2, rightOffset );

switch( reg1Found & (reg2Found<<1) )

{

case 0:

reg1 = templateData->GetReg<int>();

size += MOV::EncodeInstruction<int32>( buffer, InstrParamsRegAddr( reg1, RegEBP, leftOffset ) );

size += CMP::EncodeInstruction<int32>( buffer, InstrParamsRegAddr( reg1, RegEBP, rightOffset ) );

templateData->SetStackInfo( reg1, leftOffset );

break;

case 1:

size += CMP::EncodeInstruction<int32>( buffer, InstrParamsRegAddr( reg1, RegEBP, rightOffset ) );

break;

case 2:

size += CMP::EncodeInstruction<int32>( buffer, InstrParamsRegAddr( reg2, RegEBP, leftOffset ) );

break;

case 3:

if( reg1 == reg2 )

{

templateData->InvalidateAllReg();

\*relocAddr = buffer;

EncodingInfo info;

size += JMP::EncodeInstruction<int>( buffer, InstrParamsImm<int32>( 0 ), &info );

\*relocAddr += info.opSize;

return size;

}

else

{

size += CMP::EncodeInstruction<int32>( buffer, InstrParams2Reg( reg1, reg2 ) );

}

default:

\_\_assume( false );

}

\*relocAddr = buffer;

EncodingInfo info;

size += JE::EncodeInstruction<int>( buffer, InstrParamsImm<int32>( 0 ), &info );

\*relocAddr += info.opSize;

return size;

}

int BrTrue::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int offset, BYTE\*\* relocAddr, bool isBackEdge)

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

offset -= templateData->GetBaseOffSet();

RegNum reg;

if (isBackEdge)

{

RegNum regInc = templateData->GetReg<int>(0);

//see if we can change this to just Inc

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(regInc, (int32)context->GetFunctionBody()));

size += INC::EncodeInstruction<int>(buffer, InstrParamsAddr(regInc, context->GetFunctionBody()->GetAsmJsTotalLoopCountOffset()));

templateData->InvalidateReg(regInc);

}

if( templateData->FindRegWithStackOffset<int>( reg, offset ) )

{

size += CMP::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( reg, 0 ) );

}

else

{

size += CMP::EncodeInstruction<int>( buffer, InstrParamsAddrImm<int32>( RegEBP, offset, 0 ) );

}

\*relocAddr = buffer;

EncodingInfo info;

size += JNE::EncodeInstruction<int>( buffer, InstrParamsImm<int32>( 0 ), &info );

\*relocAddr += info.opSize;

return size;

}

int Label::ApplyTemplate( TemplateContext context, BYTE\*& buffer )

{

X86TemplateData\* templateData = GetTemplateData( context );

templateData->InvalidateAllReg();

return 0;

}

int FunctionEntry::ApplyTemplate( TemplateContext context, BYTE\*& buffer )

{

int size = 0;

X86TemplateData\* templateData = GetTemplateData(context);

Var CommonEntryPoint = (void(\*)(Js::ScriptFunction\*, void\*))AsmJsCommonEntryPoint;

// Get the stack size

FunctionBody\* funcBody = context->GetFunctionBody();

AsmJsFunctionInfo\* asmInfo = funcBody->GetAsmJsFunctionInfo();

int32 stackSize = asmInfo->GetTotalSizeinBytes();

stackSize = ::Math::Align<int32>(stackSize, 8);

//Prolog , save EBP and callee saved reg

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( RegEBP ) );

size += MOV::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEBP, RegESP ) );

//Start Stack Probe:

// cmp esp, ThreadContext::scriptStackLimit + frameSize

// jg done

// push frameSize

// call ThreadContext::ProbeCurrentStack

int scriptStackLimit = (int)funcBody->GetScriptContext()->GetThreadContext()->GetScriptStackLimit();

// cmp esp, ThreadContext::scriptStackLimit + frameSize

size += CMP::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegESP, scriptStackLimit + stackSize));

// jg Done

JumpRelocation relocDone;

EncodingHelpers::EncodeShortJump<JG>(buffer, relocDone, &size);

// call ThreadContext::ProbeCurrentStack

int probeStack = (int) (void(\*)(size\_t, Js::ScriptContext\*, int, int))ThreadContext::ProbeCurrentStack;

//push args

//push scriptcontext

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsImm<int>((int)funcBody->GetScriptContext()));

// push frameSize

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsImm<int>(stackSize));

//call probestack

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, probeStack));

size += CALL::EncodeInstruction<int>(buffer, InstrParamsReg(RegEAX));

// Done:

relocDone.ApplyReloc<int8>();

//End Stack Probe:

if (stackSize <= PAGESIZE)

{

//Stack Size

size += SUB::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegESP, stackSize));

}

else

{

// call ChkStack

int chkStk = (int)(void(\*)(int))\_chkstk;

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, stackSize));

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegECX, chkStk));

size += CALL::EncodeInstruction<int>(buffer, InstrParamsReg(RegECX));

}

// Move the arg registers here ??? TODO

//push ebx, push edi and push esi + 8 (offsets int bytecode calculated with two args i.e func obj and var args ??)

int baseOffSet = stackSize + templateData->GetEBPOffsetCorrection();

templateData->SetBaseOffset(baseOffSet);

// push EBP and push funcobj

int funcOffSet = 2 \* sizeof(Var);

//push args for CEP

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsReg(RegEBP));

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsAddr(RegEBP, funcOffSet));

// Call CEP

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, (int32)CommonEntryPoint));

size += CALL::EncodeInstruction<int>(buffer, InstrParamsReg(RegEAX));

//Push callee saved registers

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsReg(RegEBX));

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsReg(RegESI));

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsReg(RegEDI));

//SetESI and EDI

//EnvOffset - SlotIndex 1 in the stack

//ArrptrOffset - Slot Index 2 in the stack

// stackSize + templateData->GetCalleSavedRegSizeInByte() - this gives the offset of the beginning of stack from EBP

int envOffset = sizeof(Var) - stackSize;

int arrPtrOffset = 2 \* sizeof(Var) - stackSize;

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(RegEDI, RegEBP, envOffset));

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(RegESI, RegEBP, arrPtrOffset));

size += EncodingHelpers::ReloadArrayBuffer(context, buffer);

templateData->InvalidateAllVolatileReg();

return size;

}

int FunctionExit::ApplyTemplate( TemplateContext context, BYTE\*& buffer )

{

int size = 0;

#if DBG\_DUMP

if (PHASE\_ON1(AsmjsFunctionEntryPhase))

{

Var CommonCallHelper = (void(\*)(Js::ScriptFunction\*))AsmJSCommonCallHelper;

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, (int32)CommonCallHelper));

size += CALL::EncodeInstruction<int>(buffer, InstrParamsReg(RegEAX));

}

#endif

size += POP::EncodeInstruction<int>( buffer, InstrParamsReg( RegEDI ) );

size += POP::EncodeInstruction<int>( buffer, InstrParamsReg( RegESI ) );

size += POP::EncodeInstruction<int>(buffer, InstrParamsReg(RegEBX));

size += MOV::EncodeInstruction<int>(buffer, InstrParams2Reg(RegESP, RegEBP));

size += POP::EncodeInstruction<int>( buffer, InstrParamsReg( RegEBP ) );

//arg size + func

int argSize = context->GetFunctionBody()->GetAsmJsFunctionInfo()->GetArgByteSize();

// to keep 8 byte alignment after the pop EIP in RET, we add MachPtr for the func object after alignment

argSize = ::Math::Align<int32>(argSize, 8) + MachPtr;

EncodingInfo info;

size += RET::EncodeInstruction<int>(buffer, InstrParamsImm<int>(argSize), &info);

return size;

}

int LdSlot\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int slotIndex )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetModuleEnvironmentRegister( buffer, context, size );

RegNum reg2 = templateData->GetReg<int>(1<<reg);

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg2, reg, slotIndex\*sizeof( int ) ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset, reg2 );

return size;

}

int LdSlot\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int slotIndex)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetModuleEnvironmentRegister(buffer, context, size);

RegNum reg2 = templateData->GetReg<float>(1 << reg);

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsRegAddr(reg2, reg, slotIndex\*sizeof(float)));

size += EncodingHelpers::SetStackReg<float>(buffer, templateData, targetOffset, reg2);

return size;

}

int StSlot\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int srcOffset, int slotIndex )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetModuleEnvironmentRegister( buffer, context, size );

RegNum reg2;

if( !templateData->FindRegWithStackOffset<int>( reg2, srcOffset ) )

{

reg2 = templateData->GetReg<int>( 1 << reg );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg2, RegEBP, srcOffset ) );

templateData->SetStackInfo( reg2, srcOffset );

}

size += MOV::EncodeInstruction<int>( buffer, InstrParamsAddrReg( reg, slotIndex\*sizeof( int ) , reg2 ) );

return size;

}

int StSlot\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffset, int slotIndex)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetModuleEnvironmentRegister(buffer, context, size);

RegNum reg2;

if (!templateData->FindRegWithStackOffset<float>(reg2, srcOffset))

{

reg2 = templateData->GetReg<float>(1 << reg);

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsRegAddr(reg2, RegEBP, srcOffset));

templateData->SetStackInfo(reg2, srcOffset);

}

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsAddrReg(reg, slotIndex\*sizeof(float), reg2));

return size;

}

int Ld\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

if( targetOffset == rightOffset )

{

return 0;

}

int size = 0;

RegNum reg = EncodingHelpers::GetStackReg<int>( buffer, templateData, rightOffset, size );

size += MOV::EncodeInstruction<int32>( buffer, InstrParamsAddrReg(RegEBP, targetOffset, reg) );

templateData->OverwriteStack( targetOffset );

templateData->SetStackInfo( reg, targetOffset );

return size;

}

int LdConst\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int offset, int value )

{

X86TemplateData\* templateData = GetTemplateData( context );

offset -= templateData->GetBaseOffSet();

templateData->OverwriteStack( offset );

int size = MOV::EncodeInstruction<int32>( buffer, InstrParamsAddrImm<int32>(RegEBP, offset, value) );

return size;

}

int SetReturn\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int offset )

{

X86TemplateData\* templateData = GetTemplateData( context );

offset -= templateData->GetBaseOffSet();

RegNum reg = RegEAX;

if( !templateData->FindRegWithStackOffset<int>( reg, offset ) )

{

templateData->SetStackInfo( RegEAX, offset );

return MOV::EncodeInstruction<int32>( buffer, InstrParamsRegAddr(RegEAX, RegEBP, offset) );

}

else if( reg != RegEAX )

{

templateData->SetStackInfo( RegEAX, offset );

return MOV::EncodeInstruction<int32>( buffer, InstrParams2Reg(RegEAX, reg) );

}

// value already in eax, do nothing

return 0;

}

int Neg\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

rightOffset -= templateData->GetBaseOffSet();

targetOffset -= templateData->GetBaseOffSet();

int size = 0;

if( targetOffset == rightOffset )

{

size += NEG::EncodeInstruction<int32>( buffer, InstrParamsAddr( RegEBP, targetOffset ) );

templateData->OverwriteStack( targetOffset );

}

else

{

RegNum reg;

if( !templateData->FindRegWithStackOffset<int>( reg, rightOffset ) )

{

reg = templateData->GetReg<int>();

MOV::EncodeInstruction<int32>( buffer, InstrParamsRegAddr( reg, RegEBP, rightOffset ) );

}

size += NEG::EncodeInstruction<int32>( buffer, InstrParamsReg( reg ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg);

}

return size;

}

int Not\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

int size = 0;

if( targetOffset == rightOffset )

{

size += NOT::EncodeInstruction<int32>( buffer, InstrParamsAddr( RegEBP, targetOffset ) );

templateData->OverwriteStack( targetOffset );

}

else

{

RegNum reg;

if( !templateData->FindRegWithStackOffset<int>( reg, rightOffset ) )

{

reg = templateData->GetReg<int>();

MOV::EncodeInstruction<int32>( buffer, InstrParamsRegAddr( reg, RegEBP, rightOffset ) );

}

size += NOT::EncodeInstruction<int32>( buffer, InstrParamsReg( reg ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg);

}

return size;

}

int Int\_To\_Bool::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<int>(~Mask8BitsReg);

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( reg, reg ) );

size += CMP::EncodeInstruction<int>( buffer, InstrParamsAddrImm<int8>( RegEBP, rightOffset, 0 ) );

size += SETNE::EncodeInstruction<int8>( buffer, InstrParamsReg( reg ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg);

return size;

}

int LogNot\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<int>(~Mask8BitsReg);

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( reg, reg ) );

size += CMP::EncodeInstruction<int>( buffer, InstrParamsAddrImm<int8>( RegEBP, rightOffset, 0 ) );

size += SETE::EncodeInstruction<int8>( buffer, InstrParamsReg( reg ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg);

return size;

}

int Or\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<OR,int32>( context, buffer, leftOffset, rightOffset, &targetOffset );

return size;

}

int And\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<AND,int32>( context, buffer, leftOffset, rightOffset, &targetOffset );

return size;

}

int Xor\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<XOR,int32>( context, buffer, leftOffset, rightOffset, &targetOffset );

return size;

}

int Shr\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg1, reg2;

if( leftOffset != rightOffset )

{

if( !templateData->FindRegWithStackOffset<int>( reg1, leftOffset, 1<<RegECX ) )

{

reg1 = templateData->GetReg<int>( 1 << RegECX );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg1, RegEBP, leftOffset ) );

templateData->SetStackInfo( reg1, leftOffset );

}

}

else

{

reg1 = RegECX;

}

if( !templateData->FindRegWithStackOffset<int>( reg2, rightOffset ) )

{

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( RegECX, RegEBP, rightOffset ) );

templateData->SetStackInfo( RegECX, rightOffset );

}

else if( reg2 != RegECX )

{

size += MOV::EncodeInstruction<int>( buffer, InstrParams2Reg( RegECX, reg2) );

templateData->SetStackInfo( RegECX, rightOffset );

}

size += SAR::EncodeInstruction<int>( buffer, InstrParams2Reg( reg1, RegECX ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg1);

return size;

}

int Shl\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg1, reg2;

if( leftOffset != rightOffset )

{

if( !templateData->FindRegWithStackOffset<int>( reg1, leftOffset, 1<<RegECX ) )

{

reg1 = templateData->GetReg<int>( 1 << RegECX );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg1, RegEBP, leftOffset ) );

templateData->SetStackInfo( reg1, leftOffset );

}

}

else

{

reg1 = RegECX;

}

if( !templateData->FindRegWithStackOffset<int>( reg2, rightOffset ) )

{

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( RegECX, RegEBP, rightOffset ) );

templateData->SetStackInfo( RegECX, rightOffset );

}

else if( reg2 != RegECX )

{

size += MOV::EncodeInstruction<int>( buffer, InstrParams2Reg( RegECX, reg2) );

templateData->SetStackInfo( RegECX, rightOffset );

}

// Encode shl reg,cl

size += SHL::EncodeInstruction<int>( buffer, InstrParams2Reg( reg1, RegECX ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg1);

return size;

}

int ShrU\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg1, reg2;

if( leftOffset != rightOffset )

{

if( !templateData->FindRegWithStackOffset<int>( reg1, leftOffset, 1<<RegECX ) )

{

reg1 = templateData->GetReg<int>( 1 << RegECX );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg1, RegEBP, leftOffset ) );

templateData->SetStackInfo( reg1, leftOffset );

}

}

else

{

reg1 = RegECX;

}

if( !templateData->FindRegWithStackOffset<int>( reg2, rightOffset ) )

{

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( RegECX, RegEBP, rightOffset ) );

templateData->SetStackInfo( RegECX, rightOffset );

}

else if( reg2 != RegECX )

{

size += MOV::EncodeInstruction<int>( buffer, InstrParams2Reg( RegECX, reg2) );

templateData->SetStackInfo( RegECX, rightOffset );

}

// Encode shr reg,cl

size += SHR::EncodeInstruction<int>( buffer, InstrParams2Reg( reg1, RegECX ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg1);

return size;

}

int Add\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<ADD,int32>( context, buffer, leftOffset, rightOffset, &targetOffset );

return size;

}

int Sub\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

int size = 0;

size += EncodingHelpers::NonCommutativeOperation<SUB,int32>( context, buffer, leftOffset, rightOffset, &targetOffset );

return size;

}

int Mul\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<IMUL,int32>( context, buffer, leftOffset, rightOffset, &targetOffset );

return size;

}

int Div\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum rhsReg = EncodingHelpers::GetStackReg<int>(buffer,templateData,rightOffset,size,1<<RegEAX|1<<RegEDX);

// test reg,reg

size += TEST::EncodeInstruction<int>(buffer, InstrParams2Reg(rhsReg, rhsReg));

// JNE Label1

JumpRelocation relocLabel1;

EncodingHelpers::EncodeShortJump<JNE>( buffer, relocLabel1, &size );

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEAX, RegEAX ) );

// JMP LabelEnd

JumpRelocation relocLabelEnd;

EncodingHelpers::EncodeShortJump<JMP>( buffer, relocLabelEnd, &size );

// Label1:

relocLabel1.ApplyReloc<int8>();

// MOV eax, [leftOffset]

RegNum lhsReg;

if (!templateData->FindRegWithStackOffset<int>(lhsReg, leftOffset))

{

size += MOV::EncodeInstruction<int32>( buffer, InstrParamsRegAddr(RegEAX, RegEBP, leftOffset) );

}

else if (lhsReg != RegEAX)

{

size += MOV::EncodeInstruction<int32>(buffer, InstrParams2Reg(RegEAX, lhsReg));

}

size += CMP::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegEAX, 0x80000000 ) );

// JNE LabelDoDiv

JumpRelocation relocLabelDoDiv;

EncodingHelpers::EncodeShortJump<JNE>( buffer, relocLabelDoDiv, &size );

// CMP reg,-1

size += CMP::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(rhsReg, -1));

// JNE LabelDoDiv

JumpRelocation relocLabelDoDiv2;

EncodingHelpers::EncodeShortJump<JNE>( buffer, relocLabelDoDiv2, &size );

// JMP LabelEnd

JumpRelocation relocLabelEnd2;

EncodingHelpers::EncodeShortJump<JMP>( buffer, relocLabelEnd2, &size );

// LabelDoDiv:

relocLabelDoDiv.ApplyReloc<int8>();

relocLabelDoDiv2.ApplyReloc<int8>();

// cdq

size += CDQ::EncodeInstruction<int>( buffer );

// idiv reg

size += IDIV::EncodeInstruction<int>(buffer, InstrParamsReg(rhsReg));

// LabelEnd:

relocLabelEnd.ApplyReloc<int8>();

relocLabelEnd2.ApplyReloc<int8>();

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset, RegEAX );

templateData->InvalidateReg( RegEDX );

return size;

}

int Rem\_Int::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

//Xor regedx , regedx

size += XOR::EncodeInstruction<int>(buffer, InstrParams2Reg(RegEDX, RegEDX));

//mov eax , leftoffset

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(RegEAX, RegEBP, leftOffset));

RegNum rhsReg = EncodingHelpers::GetStackReg<int>(buffer, templateData, rightOffset, size, 1 << RegEAX | 1 << RegEDX);

//cmp rightoffset, 0

size += CMP::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(rhsReg, 0));

//je :L4

JumpRelocation reloc(buffer, &size);

EncodingInfo info;

size += JE::EncodeInstruction<int8>(buffer, InstrParamsImm<int8>(0), &info);

reloc.JumpEncoded(info);

//cmp leftoffset -2147483648

size += CMP::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, 0x80000000));

//jne :L3

JumpRelocation reloc2(buffer, &size);

EncodingInfo info2;

size += JNE::EncodeInstruction<int8>(buffer, InstrParamsImm<int8>(0), &info2);

reloc2.JumpEncoded(info2);

//cmp rightoffset -1

size += CMP::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(rhsReg, -1));

//je :L4

JumpRelocation reloc3(buffer, &size);

EncodingInfo info3;

size += JE::EncodeInstruction<int8>(buffer, InstrParamsImm<int8>(0), &info3);

reloc3.JumpEncoded(info3);

//:L3

reloc2.ApplyReloc<int8>();

//cdq

size += CDQ::EncodeInstruction<int>(buffer);

//idiv rightoffset

size += IDIV::EncodeInstruction<int>(buffer, InstrParamsReg(rhsReg));

//mov targetoffset , edx

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset,RegEDX);

//jmp :L5

JumpRelocation reloc4(buffer, &size);

EncodingInfo info4;

size += JMP::EncodeInstruction<int8>(buffer, InstrParamsImm<int8>(0), &info4);

reloc4.JumpEncoded(info4);

//:L4

reloc.ApplyReloc<int8>();

reloc3.ApplyReloc<int8>();

//mov targetoffset , 0

size += MOV::EncodeInstruction<int>(buffer, InstrParamsAddrImm<int32>(RegEBP, targetOffset, 0));

//:L5

reloc4.ApplyReloc<int8>();

//mov eax, targetoffset

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(RegEAX, RegEBP, targetOffset));

templateData->InvalidateReg(RegEAX);

templateData->InvalidateReg(RegEDX);

return size;

}

#define IntCmp(name, jmp) \

int name::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )\

{\

X86TemplateData\* templateData = GetTemplateData( context );\

int size = 0;\

RegNum resultReg = templateData->GetReg<int>();\

size += XOR::EncodeInstruction<int32>( buffer, InstrParams2Reg( resultReg, resultReg ) );\

size += EncodingHelpers::NonCommutativeOperation<CMP,int>( context, buffer, leftOffset, rightOffset, nullptr, nullptr, 1 << resultReg );\

size += jmp::EncodeInstruction<int8>( buffer, InstrParamsImm<int8>(1) );\

size += INC::EncodeInstruction<int32>( buffer, InstrParamsReg( resultReg ) );\

targetOffset -= templateData->GetBaseOffSet();\

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , resultReg);\

return size;\

}

IntCmp(Lt\_Int,JGE)

IntCmp(Le\_Int,JG)

IntCmp(Gt\_Int,JLE)

IntCmp(Ge\_Int,JL)

IntCmp(Eq\_Int,JNE)

IntCmp(Ne\_Int,JE)

IntCmp(Lt\_UInt,JAE)

IntCmp(Le\_UInt,JA)

IntCmp(Gt\_UInt,JBE)

IntCmp(Ge\_UInt,JB)

#undef IntCmp

int Min\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

RegNum reg1;

size += EncodingHelpers::NonCommutativeOperation<CMP, int>( context, buffer, leftOffset, rightOffset, nullptr, &reg1 );

RegNum reg2;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

if( templateData->FindRegWithStackOffset<int>( reg2, rightOffset ) )

{

size += CMOVG::EncodeInstruction<int>( buffer, InstrParams2Reg( reg1, reg2 ) );

}

else

{

size += CMOVG::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg1, RegEBP, rightOffset) );

}

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg1);

return size;

}

int Max\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

RegNum reg1;

size += EncodingHelpers::NonCommutativeOperation<CMP, int>( context, buffer, leftOffset, rightOffset, nullptr, &reg1 );

RegNum reg2;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

if( templateData->FindRegWithStackOffset<int>( reg2, rightOffset ) )

{

size += CMOVL::EncodeInstruction<int>( buffer, InstrParams2Reg( reg1, reg2 ) );

}

else

{

size += CMOVL::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg1, RegEBP, rightOffset) );

}

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , reg1);

return size;

}

int Abs\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg;

if( templateData->FindRegWithStackOffset<int>( reg, rightOffset ) )

{

if( reg != RegEAX )

{

size += MOV::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEAX, reg ) );

}

}

else

{

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( RegEAX, RegEBP, rightOffset ) );

}

size += CDQ::EncodeInstruction<int>( buffer );

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEAX, RegEDX ) );

size += SUB::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEAX, RegEDX ) );

templateData->InvalidateReg( RegEDX );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , RegEAX);

return size;

}

int Clz32\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

// BSR tmp, src

// JE $label32

// MOV dst, 31

// SUB dst, tmp

// JMP $done

// label32:

// MOV dst, 32

// $done

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum tmpReg = templateData->GetReg<int>();

RegNum srcReg;

if (templateData->FindRegWithStackOffset<int>(srcReg, rightOffset))

{

size += BSR::EncodeInstruction<int>(buffer, InstrParams2Reg(tmpReg, srcReg));

}

else

{

size += BSR::EncodeInstruction<int>(buffer, InstrParamsRegAddr(tmpReg, RegEBP, rightOffset));

}

JumpRelocation relocLabel32;

EncodingHelpers::EncodeShortJump<JE>(buffer, relocLabel32, &size);

RegNum dstReg = templateData->GetReg<int>(1 << tmpReg);

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int>(dstReg, 31));

size += SUB::EncodeInstruction<int8>(buffer, InstrParams2Reg(dstReg, tmpReg));

JumpRelocation relocLabelDone;

EncodingHelpers::EncodeShortJump<JMP>(buffer, relocLabelDone, &size);

relocLabel32.ApplyReloc<int8>();

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int>(dstReg, 32));

relocLabelDone.ApplyReloc<int8>();

templateData->InvalidateReg(tmpReg);

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , dstReg);

return size;

}

int Mul\_UInt::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg1, reg2;

const int reg1Found = templateData->FindRegWithStackOffset<int>( reg1, rightOffset, 1<<RegEDX );

const int reg2Found = templateData->FindRegWithStackOffset<int>( reg2, leftOffset, 1<<RegEDX );

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg(RegEDX,RegEDX) );

switch( reg1Found & ( reg2Found << 1 ) )

{

case 0: // none found

reg1 = RegEAX;

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg1, RegEBP, leftOffset ) );

templateData->SetStackInfo( reg1, leftOffset );

size += MUL::EncodeInstruction<int>( buffer, InstrParamsAddr(RegEBP, rightOffset) );

break;

case 1: // found 2

if( reg2 == RegEAX )

{

size += MUL::EncodeInstruction<int>( buffer, InstrParamsAddr(RegEBP, leftOffset) );

}

else

{

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( RegEAX, RegEBP, leftOffset ) );

size += MUL::EncodeInstruction<int>( buffer, InstrParamsReg(reg2) );

}

break;

case 2: // found 1

if( reg1 == RegEAX )

{

size += MUL::EncodeInstruction<int>( buffer, InstrParamsAddr(RegEBP, rightOffset) );

}

else

{

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( RegEAX, RegEBP, rightOffset ) );

size += MUL::EncodeInstruction<int>( buffer, InstrParamsReg(reg1) );

}

break;

case 3: // found both

if( reg1 == RegEAX )

{

size += MUL::EncodeInstruction<int>( buffer, InstrParamsReg(reg2) );

}

else if( reg2 == RegEAX )

{

size += MUL::EncodeInstruction<int>( buffer, InstrParamsReg(reg1) );

}

else

{

size += MOV::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEAX, reg1 ) );

size += MUL::EncodeInstruction<int>( buffer, InstrParamsReg(reg2) );

}

break;

default:

\_\_assume( false );

}

size += TEST::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEDX, RegEDX ) );

JumpRelocation reloc;

EncodingHelpers::EncodeShortJump<JE>( buffer, reloc, &size );

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEAX, RegEAX ) );

reloc.ApplyReloc<int8>();

templateData->InvalidateReg( RegEDX );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , RegEAX);

return size;

}

int Div\_UInt::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetStackReg<int>( buffer, templateData, rightOffset, size, 1 << RegEDX | 1 << RegEAX );

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEAX, RegEAX ) );

size += CMP::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( reg, 0 ) );

JumpRelocation reloc( buffer, &size );

EncodingInfo info1;

// JNE labelEnd

size += JE::EncodeInstruction<int8>( buffer, InstrParamsImm<int8>( 0 ), &info1 );

reloc.JumpEncoded( info1 );

size += MOV::EncodeInstruction<int32>( buffer, InstrParamsRegAddr( RegEAX, RegEBP, leftOffset ) );

size += XOR::EncodeInstruction<int32>( buffer, InstrParams2Reg( RegEDX, RegEDX ) );

size += DIV::EncodeInstruction<int32>( buffer, InstrParamsReg( reg ) );

// labelEnd:

reloc.ApplyReloc<int8>();

templateData->InvalidateReg( RegEDX );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , RegEAX);

return size;

}

int Rem\_UInt::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( RegEDX, RegEDX ) );

size += CMP::EncodeInstruction<int>( buffer, InstrParamsAddrImm<int>( RegEBP, rightOffset, 0 ) );

size += JE::EncodeInstruction<int8>( buffer, InstrParamsImm<int8>( 0 ) );

BYTE\* reloc = &buffer[-1];

int relocSize = 0;

relocSize += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( RegEAX, RegEBP, leftOffset ) );

relocSize += DIV::EncodeInstruction<int>( buffer, InstrParamsAddr( RegEBP, rightOffset ) );

Assert( FitsInByte( relocSize ) );

\*reloc = (BYTE)relocSize;

size += relocSize;

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , RegEDX);

templateData->InvalidateReg(RegEAX);

return size;

}

int SetReturn\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int offset )

{

X86TemplateData\* templateData = GetTemplateData(context);

offset -= templateData->GetBaseOffSet();

RegNum reg = RegXMM0;

if (!templateData->FindRegWithStackOffset<double>(reg, offset))

{

templateData->SetStackInfo(RegXMM0, offset);

return MOVSD::EncodeInstruction<double>(buffer, InstrParamsRegAddr(RegXMM0, RegEBP, offset));

}

if (reg != RegXMM0)

{

return MOVSD::EncodeInstruction<double>(buffer, InstrParams2Reg(RegXMM0, reg));

}

return 0;

}

int SetReturn\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int offset)

{

X86TemplateData\* templateData = GetTemplateData(context);

offset -= templateData->GetBaseOffSet();

RegNum reg = RegXMM0;

if (!templateData->FindRegWithStackOffset<float>(reg, offset))

{

templateData->SetStackInfo(RegXMM0, offset);

return MOVSS::EncodeInstruction<float>(buffer, InstrParamsRegAddr(RegXMM0, RegEBP, offset));

}

if (reg != RegXMM0)

{

return MOVSS::EncodeInstruction<float>(buffer, InstrParams2Reg(RegXMM0, reg));

}

return 0;

}

int SetFround\_Db::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset,int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<float>();

RegNum reg1;

int size = 0;

if (templateData->FindRegWithStackOffset<double>(reg1, rightOffset))

{

size += CVTSD2SS::EncodeInstruction<double>(buffer, InstrParams2Reg(reg, reg1));

}

else

{

size += CVTSD2SS::EncodeInstruction<double>(buffer, InstrParamsRegAddr(reg, RegEBP, rightOffset));

}

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsAddrReg(RegEBP, targetOffset, reg));

templateData->OverwriteStack(targetOffset);

templateData->SetStackInfo(reg,targetOffset);

return size;

}

int SetFround\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<float>();

int size = 0;

if (!templateData->FindRegWithStackOffset<float>(reg, rightOffset))

{

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsRegAddr(reg, RegEBP, rightOffset));

templateData->SetStackInfo(reg, rightOffset);

}

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsAddrReg(RegEBP, targetOffset, reg));

templateData->OverwriteStack(targetOffset);

return size;

}

int SetFround\_Int::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<float>();

RegNum reg1;

int size = 0;

if (templateData->FindRegWithStackOffset<int32>(reg1, rightOffset))

{

size += CVTSI2SS::EncodeInstruction<int32>(buffer, InstrParams2Reg(reg, reg1));

}

else

{

size += CVTSI2SS::EncodeInstruction<int32>(buffer, InstrParamsRegAddr(reg, RegEBP, rightOffset));

}

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsAddrReg(RegEBP, targetOffset, reg));

templateData->OverwriteStack(targetOffset);

templateData->SetStackInfo(reg, targetOffset);

return size;

}

int StSlot\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int srcOffset, int slotIndex )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetModuleEnvironmentRegister( buffer, context, size );

RegNum reg2;

if( !templateData->FindRegWithStackOffset<double>( reg2, srcOffset ) )

{

reg2 = templateData->GetReg<double>();

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( reg2, RegEBP, srcOffset ) );

templateData->SetStackInfo( reg2, srcOffset );

}

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsAddrReg( reg, slotIndex\*sizeof( double ) , reg2 ) );

return size;

}

int LdSlot\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int slotIndex )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetModuleEnvironmentRegister( buffer, context, size );

RegNum reg2 = templateData->GetReg<double>();

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( reg2, reg, slotIndex\*sizeof( double ) ) );

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , reg2);

return size;

}

int LdAddr\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, const double\* dbAddr )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<double>();

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegPtr( reg, (void\*)dbAddr ) );

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , reg);

return size;

}

int Ld\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

if( targetOffset == rightOffset )

{

return 0;

}

RegNum reg = templateData->GetReg<double>();

int size = 0;

if( !templateData->FindRegWithStackOffset<double>( reg, rightOffset ) )

{

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr(reg, RegEBP, rightOffset) );

templateData->SetStackInfo( reg, rightOffset );

}

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsAddrReg(RegEBP, targetOffset, reg) );

templateData->OverwriteStack( targetOffset );

return size;

}

int Ld\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

if (targetOffset == rightOffset)

{

return 0;

}

//get reg can be double registers for float too

RegNum reg = templateData->GetReg<float>();

int size = 0;

if (!templateData->FindRegWithStackOffset<float>(reg, rightOffset))

{

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsRegAddr(reg, RegEBP, rightOffset));

templateData->SetStackInfo(reg, rightOffset);

}

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsAddrReg(RegEBP, targetOffset, reg));

templateData->OverwriteStack(targetOffset);

return size;

}

int Add\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<ADDSS, float>(context, buffer, leftOffset, rightOffset, &targetOffset);

return size;

}

int Add\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<ADDSD, double>( context, buffer, leftOffset, rightOffset, &targetOffset );

return size;

}

int Sub\_Db::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

int size = 0;

size += EncodingHelpers::NonCommutativeOperation<SUBSD, double>(context, buffer, leftOffset, rightOffset, &targetOffset);

return size;

}

int Mul\_Db::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<MULSD, double>(context, buffer, leftOffset, rightOffset, &targetOffset);

return size;

}

int Div\_Db::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<DIVSD, double>(context, buffer, leftOffset, rightOffset, &targetOffset);

return size;

}

int Sub\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

int size = 0;

size += EncodingHelpers::NonCommutativeOperation<SUBSS, float>(context, buffer, leftOffset, rightOffset, &targetOffset);

return size;

}

int Mul\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<MULSS, float>(context, buffer, leftOffset, rightOffset, &targetOffset);

return size;

}

int Div\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

int size = 0;

size += EncodingHelpers::CommutativeOperation<DIVSS, double>(context, buffer, leftOffset, rightOffset, &targetOffset);

return size;

}

int Rem\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

leftOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

//AsmJsMath::Rem < int > ;

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int8>( RegESP, 16 ) );

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( RegXMM0, RegEBP, rightOffset ) );

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsAddrReg( RegESP, 8, RegXMM0 ) );

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( RegXMM0, RegEBP, leftOffset ) );

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsAddrReg( RegESP, 0, RegXMM0 ) );

void\* ptr = (double (\*)(double,double)) AsmJsMath::Rem < double > ;

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>(RegEAX,(int)ptr) );

size += CALL::EncodeInstruction<int>( buffer, InstrParamsReg(RegEAX) );

templateData->InvalidateAllVolatileReg();

size += FSTP::EncodeInstruction<double>( buffer, InstrParamsAddr( RegEBP, targetOffset ) );

templateData->InvalidateReg( RegEAX );

templateData->OverwriteStack( targetOffset );

return size;

}

template<typename JCC, typename OperationSignature, typename Size>

int CompareEq(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

RegNum resultReg = templateData->GetReg<int>(1 << RegEAX);

size += XOR::EncodeInstruction<int32>(buffer, InstrParams2Reg(resultReg, resultReg));

size += EncodingHelpers::NonCommutativeOperation<OperationSignature, Size>(context, buffer, leftOffset, rightOffset);

size += LAHF::EncodeInstruction<int32>(buffer);

size += TEST::EncodeInstruction<int8>(buffer, InstrParamsRegImm<int8>(RegNum(RegEAX), 0x44));

/\*fix for ah\*/buffer[-2] |= 0x04;

size += JCC::EncodeInstruction<int8>(buffer, InstrParamsImm<int8>(1));

size += INC::EncodeInstruction<int32>(buffer, InstrParamsReg(resultReg));

templateData->InvalidateReg(RegEAX);

targetOffset -= templateData->GetBaseOffSet();

size += EncodingHelpers::SetStackReg<int>(buffer, templateData, targetOffset, resultReg);

return size;

}

int CmpEq\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

return CompareEq<JP, UCOMISS, float>(context, buffer, targetOffset, leftOffset, rightOffset);

}

int CmpNe\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

return CompareEq<JNP, UCOMISS, float>(context, buffer, targetOffset, leftOffset, rightOffset);

}

int CmpEq\_Db::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

return CompareEq<JP, UCOMISD, double>(context, buffer, targetOffset, leftOffset, rightOffset);

}

int CmpNe\_Db::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

return CompareEq<JNP, UCOMISD, double>(context, buffer, targetOffset, leftOffset, rightOffset);

}

template<typename JCC, typename OperationSignature, typename Size>

int CompareDbOrFlt( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

// we are modifying the rightoffset and leftoffset in the call to EncodingHelpers::NonCommutativeOperation

targetOffset -= templateData->GetBaseOffSet();

RegNum resultReg = templateData->GetReg<int>();

size += XOR::EncodeInstruction<int32>( buffer, InstrParams2Reg( resultReg, resultReg ) );

size += EncodingHelpers::NonCommutativeOperation<OperationSignature, Size>(context, buffer, leftOffset, rightOffset);

size += JCC::EncodeInstruction<int8>( buffer, InstrParamsImm<int8>(1) );

size += INC::EncodeInstruction<int32>( buffer, InstrParamsReg( resultReg ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , resultReg);

return size;

}

int CmpLt\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

return CompareDbOrFlt<JBE, COMISS, float>(context, buffer, targetOffset, rightOffset, leftOffset);

}

int CmpLe\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

return CompareDbOrFlt<JB, COMISS, float>(context, buffer, targetOffset, rightOffset, leftOffset);

}

int CmpGt\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

return CompareDbOrFlt<JBE, COMISS, float>(context, buffer, targetOffset, leftOffset, rightOffset);

}

int CmpGe\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset)

{

return CompareDbOrFlt<JB, COMISS, float>(context, buffer, targetOffset, leftOffset, rightOffset);

}

int CmpLt\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

return CompareDbOrFlt<JBE, COMISD, double>(context, buffer, targetOffset, rightOffset, leftOffset);

}

int CmpLe\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

return CompareDbOrFlt<JB, COMISD, double>(context, buffer, targetOffset, rightOffset, leftOffset);

}

int CmpGt\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

return CompareDbOrFlt<JBE, COMISD, double>(context, buffer, targetOffset, leftOffset, rightOffset);

}

int CmpGe\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int leftOffset, int rightOffset )

{

return CompareDbOrFlt<JB, COMISD, double>(context, buffer, targetOffset, leftOffset, rightOffset);

}

int UInt\_To\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum regInt;

RegNum regDouble = templateData->GetReg<double>();

if( !templateData->FindRegWithStackOffset<int>( regInt, rightOffset ) )

{

regInt = templateData->GetReg<int>();

size += MOV::EncodeInstruction<int32>( buffer, InstrParamsRegAddr( regInt, RegEBP, rightOffset ) );

}

size += MOVD::EncodeInstruction<double>( buffer, InstrParams2Reg(regDouble,regInt) );

size += CVTDQ2PD::EncodeInstruction<double>( buffer, InstrParams2Reg( regDouble, regDouble ) );

size += SHR::EncodeInstruction<int32>( buffer, InstrParamsRegImm<int8>( regInt, 31 ) );

templateData->InvalidateReg( regInt );

static \_\_declspec(align(8)) const double MaskConvUintDouble[] = { 0.0, 4294967296.0 };

size += ADDSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( regDouble, RegNOREG, regInt, 8, (int)MaskConvUintDouble ) );

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , regDouble);

return size;

}

int Int\_To\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<double>(), regInt;

if( templateData->FindRegWithStackOffset<int>( regInt, rightOffset ) )

{

size += CVTSI2SD::EncodeInstruction<double>( buffer, InstrParams2Reg( reg, regInt ) );

}

else

{

size += CVTSI2SD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( reg, RegEBP, rightOffset ) );

}

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , reg);

return size;

}

int Float\_To\_Db::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<double>();

RegNum reg1;

if (templateData->FindRegWithStackOffset<float>(reg1, rightOffset))

{

size += CVTSS2SD::EncodeInstruction<float>(buffer, InstrParams2Reg(reg, reg1));

}

else

{

size += CVTSS2SD::EncodeInstruction<float>(buffer, InstrParamsRegAddr(reg, RegEBP, rightOffset));

}

size += EncodingHelpers::SetStackReg<double>(buffer, templateData, targetOffset, reg);

return size;

}

int Float\_To\_Int::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg = templateData->GetReg<int>();

RegNum reg1;

if (templateData->FindRegWithStackOffset<float>(reg1, rightOffset))

{

size += CVTTSS2SI::EncodeInstruction<float>(buffer, InstrParams2Reg(reg, reg1));

}

else

{

size += CVTTSS2SI::EncodeInstruction<float>(buffer, InstrParamsRegAddr(reg, RegEBP, rightOffset));

}

size += EncodingHelpers::SetStackReg<int>(buffer, templateData, targetOffset, reg);

return size;

}

int Db\_To\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg;

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int8>( RegESP, 8 ) );

if( !templateData->FindRegWithStackOffset<double>( reg, rightOffset ) )

{

reg = templateData->GetReg<double>();

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( reg, RegEBP, rightOffset ) );

templateData->SetStackInfo( reg, rightOffset );

}

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsAddrReg( RegESP, 0, reg ) );

void\* addr = ((int(\*)(double))JavascriptMath::ToInt32Core);

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegEAX, (int)addr ) );

size += CALL::EncodeInstruction<int>( buffer, InstrParamsReg( RegEAX ) );

templateData->InvalidateAllVolatileReg();

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , RegEAX);

return size;

}

int Neg\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg;

if( !templateData->FindRegWithStackOffset<double>( reg, rightOffset ) )

{

reg = templateData->GetReg<double>();

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( reg, RegEBP, rightOffset ) );

}

static \_\_declspec(align(16)) const double MaskNegDouble[] = { -0.0, -0.0 };

static const BYTE temp[] = {

0x66, 0x0F, 0x57, 0x05,

(BYTE)( ( (int)( MaskNegDouble ) ) & 0xFF ),

(BYTE)( ( ( (int)( MaskNegDouble ) ) >> 8 ) & 0xFF ),

(BYTE)( ( ( (int)( MaskNegDouble ) ) >> 16 ) & 0xFF ),

(BYTE)( ( ( (int)( MaskNegDouble ) ) >> 24 ) & 0xFF ),

};

size += ApplyCustomTemplate( buffer, temp, 8 );

//fix template for register

buffer[-5] |= RegEncode[reg] << 3;

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , reg);

return size;

}

int Neg\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int rightOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

rightOffset -= templateData->GetBaseOffSet();

RegNum reg;

if (!templateData->FindRegWithStackOffset<float>(reg, rightOffset))

{

reg = templateData->GetReg<float>();

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsRegAddr(reg, RegEBP, rightOffset));

}

static const BYTE temp[] = {

0x0F, 0x57, 0x05,

(BYTE)(((int)(JavascriptNumber::MaskNegFloat)) & 0xFF),

(BYTE)((((int)(JavascriptNumber::MaskNegFloat)) >> 8) & 0xFF),

(BYTE)((((int)(JavascriptNumber::MaskNegFloat)) >> 16) & 0xFF),

(BYTE)((((int)(JavascriptNumber::MaskNegFloat)) >> 24) & 0xFF),

};

size += ApplyCustomTemplate(buffer, temp, 7);

//fix template for register

buffer[-5] |= RegEncode[reg] << 3;

size += EncodingHelpers::SetStackReg<float>(buffer, templateData, targetOffset, reg);

return size;

}

int Call\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int nbOffsets, int\* offsets, void\* addr, bool addEsp )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

\*offsets -= templateData->GetBaseOffSet();

Assert( nbOffsets >= 1 );

const int nbArgs = nbOffsets - 1;

const int targetOffset = offsets[0];

int\* args = offsets + 1;

int stackSize = nbArgs << 3;

Assert( stackSize > nbArgs ); // check for overflow

if( nbArgs > 0 )

{

RegNum reg = templateData->GetReg<double>();

if( FitsInByte( stackSize ) )

{

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int8>( RegESP, (int8)( stackSize ) ) );

}

else

{

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegESP, stackSize ) );

}

int espOffset = stackSize - 8;

for( int i = nbArgs - 1; i >= 0; i-- )

{

// TODO: check for reg in template

int argOffset = args[i] - templateData->GetBaseOffSet();

size += MOVSD::EncodeInstruction<double>(buffer, InstrParamsRegAddr(reg, RegEBP, argOffset));

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsAddrReg( RegESP, espOffset, reg ) );

espOffset -= 8;

}

templateData->InvalidateReg( reg );

}

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>(RegEAX,(int)addr) );

size += CALL::EncodeInstruction<int>( buffer, InstrParamsReg(RegEAX) );

templateData->InvalidateAllVolatileReg();

size += FSTP::EncodeInstruction<double>( buffer, InstrParamsAddr( RegEBP, targetOffset ) );

templateData->InvalidateReg( RegEAX );

templateData->OverwriteStack( targetOffset );

if( addEsp )

{

if( FitsInByte( stackSize ) )

{

size += ADD::EncodeInstruction<int>( buffer, InstrParamsRegImm<int8>( RegESP, (int8)stackSize ) );

}

else

{

size += ADD::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegESP, stackSize ) );

}

}

return size;

}

int Call\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int nbOffsets, int\* offsets, void\* addr, bool addEsp)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

\*offsets -= templateData->GetBaseOffSet();

Assert(nbOffsets >= 1);

const int nbArgs = nbOffsets - 1;

const int targetOffset = offsets[0];

int\* args = offsets + 1;

// REVIEW: 4 bytes per arg for floats, do we want to maintain 8 byte stack alignment?

int stackSize = nbArgs << 2;

Assert(stackSize > nbArgs); // check for overflow

if (nbArgs > 0)

{

RegNum reg = templateData->GetReg<float>();

if (FitsInByte(stackSize))

{

size += SUB::EncodeInstruction<int>(buffer, InstrParamsRegImm<int8>(RegESP, (int8)(stackSize)));

}

else

{

size += SUB::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegESP, stackSize));

}

int espOffset = stackSize - 4;

for (int i = nbArgs - 1; i >= 0; i--)

{

// TODO: check for reg in template

int argOffset = args[i] - templateData->GetBaseOffSet();

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsRegAddr(reg, RegEBP, argOffset));

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsAddrReg(RegESP, espOffset, reg));

espOffset -= 4;

}

templateData->InvalidateReg(reg);

}

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, (int)addr));

size += CALL::EncodeInstruction<int>(buffer, InstrParamsReg(RegEAX));

templateData->InvalidateAllVolatileReg();

size += FSTP::EncodeInstruction<float>(buffer, InstrParamsAddr(RegEBP, targetOffset));

templateData->InvalidateReg(RegEAX);

templateData->OverwriteStack(targetOffset);

if (addEsp)

{

if (FitsInByte(stackSize))

{

size += ADD::EncodeInstruction<int>(buffer, InstrParamsRegImm<int8>(RegESP, (int8)stackSize));

}

else

{

size += ADD::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegESP, stackSize));

}

}

return size;

}

int StartCall::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int argBytesSize )

{

int size = 0;

// remove extra var from sub because we are using push to add it

argBytesSize -= sizeof(Var);

if( FitsInByte( argBytesSize ) )

{

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int8>( RegESP, (int8)argBytesSize ) );

}

else

{

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegESP, argBytesSize ) );

}

// pushing undefined as the first var

const int undefinedVar = (int)context->GetFunctionBody()->GetScriptContext()->GetLibrary()->GetUndefined();

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsImm<int32>(undefinedVar) );

return size;

}

int ArgOut\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int argIndex, int offset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

offset -= templateData->GetBaseOffSet();

RegNum regScriptContext, regVariable;

if( !templateData->FindRegWithStackOffset<int>( regScriptContext, templateData->GetScriptContextOffset() ) )

{

regScriptContext = templateData->GetReg<int>(1<<RegEAX);

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(regScriptContext, RegEBP, templateData->GetScriptContextOffset()));

templateData->SetStackInfo(regScriptContext, templateData->GetScriptContextOffset());

}

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( regScriptContext ) );

if( !templateData->FindRegWithStackOffset<int>( regVariable, offset ) )

{

regVariable = templateData->GetReg<int>(1<<RegEAX);

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( regVariable, RegEBP, offset ) );

templateData->SetStackInfo( regVariable, offset );

}

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( regVariable ) );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegEAX, (int32)(Var(\*)(int,ScriptContext\*))JavascriptNumber::ToVar) );

size += CALL::EncodeInstruction<int>( buffer, InstrParamsReg( RegEAX ) );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsAddrReg( RegESP, argIndex << 2, RegEAX ) );

templateData->InvalidateAllVolatileReg();

return size;

}

int ArgOut\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int argIndex, int offset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

offset -= templateData->GetBaseOffSet();

RegNum regScriptContext = EncodingHelpers::GetScriptContextRegister( buffer, context, size, 1 << RegEAX ), regVariable;

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( regScriptContext ) );

if( !templateData->FindRegWithStackOffset<double>( regVariable, offset ) )

{

regVariable = templateData->GetReg<double>();

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegAddr( regVariable, RegEBP, offset ) );

templateData->SetStackInfo( regVariable, offset );

}

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int8>( RegESP, 8 ) );

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsAddrReg( RegESP, 0, regVariable ) );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegEAX, (int32)(Var(\*)(double,ScriptContext\*))JavascriptNumber::New) );

size += CALL::EncodeInstruction<int>( buffer, InstrParamsReg( RegEAX ) );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsAddrReg( RegESP, argIndex << 2, RegEAX ) );

templateData->InvalidateAllVolatileReg();

return size;

}

int Call::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int funcOffset, int nbArgs )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

funcOffset -= templateData->GetBaseOffSet();

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( RegESP ) );

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( (int8)nbArgs ) );

RegNum reg;

if( !templateData->FindRegWithStackOffset<int>( reg, funcOffset ) )

{

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsAddr( RegEBP, funcOffset ) );

}

else

{

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( reg ) );

}

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegEAX, (int32)( Var( \*)( JavascriptFunction\*, int, Var\* ) )ExternalCallHelper ));

size += CALL::EncodeInstruction<int>( buffer, InstrParamsReg( RegEAX ) );

const int stackSize = nbArgs << 2;

Assert( FitsInByte( stackSize ) );

size += ADD::EncodeInstruction<int>( buffer, InstrParamsRegImm<int8>( RegESP, (int8)stackSize ) );

templateData->InvalidateAllVolatileReg();

size += EncodingHelpers::ReloadArrayBuffer(context, buffer);

size += EncodingHelpers::CheckForArrayBufferDetached(context, buffer);

if (targetOffset != templateData->GetModuleSlotOffset())

{

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , RegEAX);

}

templateData->SetStackInfo( RegEAX, targetOffset );

return size;

}

int Conv\_VTI::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

RegNum reg;

if( !templateData->FindRegWithStackOffset<int>( reg, srcOffset ) )

{

reg = templateData->GetReg<int>();

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg, RegEBP, srcOffset ) );

templateData->SetStackInfo( reg, srcOffset );

}

RegNum regScriptContext = EncodingHelpers::GetScriptContextRegister( buffer, context, size, 1 << reg );

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( regScriptContext ) );

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( reg ) );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegEAX, (int32)(int32(\*)(Var,ScriptContext\*))JavascriptMath::ToInt32) );

size += CALL::EncodeInstruction<int>( buffer, InstrParamsReg( RegEAX ) );

templateData->InvalidateAllVolatileReg();

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , RegEAX);

return size;

}

int Conv\_VTD::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

RegNum reg;

if( !templateData->FindRegWithStackOffset<int>( reg, srcOffset ) )

{

reg = templateData->GetReg<int>();

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg, RegEBP, srcOffset ) );

templateData->SetStackInfo( reg, srcOffset );

}

RegNum regScriptContext = EncodingHelpers::GetScriptContextRegister( buffer, context, size, 1 << reg );

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( regScriptContext ) );

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg( reg ) );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegEAX, (int32)(double(\*)(Var,ScriptContext\*))JavascriptConversion::ToNumber) );

size += CALL::EncodeInstruction<int>( buffer, InstrParamsReg( RegEAX ) );

templateData->InvalidateAllVolatileReg();

size += FSTP::EncodeInstruction<double>( buffer, InstrParamsAddr( RegEBP, targetOffset ) );

templateData->OverwriteStack( targetOffset );

return size;

}

//TODO - consider changing this to template (Conv\_vtd and Conv\_vtf)

int Conv\_VTF::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

RegNum reg;

if (!templateData->FindRegWithStackOffset<int>(reg, srcOffset))

{

reg = templateData->GetReg<int>();

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(reg, RegEBP, srcOffset));

templateData->SetStackInfo(reg, srcOffset);

}

RegNum regScriptContext = EncodingHelpers::GetScriptContextRegister(buffer, context, size, 1 << reg);

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsReg(regScriptContext));

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsReg(reg));

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, (int32)(float(\*)(Var, ScriptContext\*))JavascriptConversion::ToNumber));

size += CALL::EncodeInstruction<int>(buffer, InstrParamsReg(RegEAX));

templateData->InvalidateAllVolatileReg();

size += FSTP::EncodeInstruction<float>(buffer, InstrParamsAddr(RegEBP, targetOffset));

templateData->OverwriteStack(targetOffset);

return size;

}

int I\_StartCall::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int argBytesSize )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

templateData->StartInternalCall(argBytesSize);

argBytesSize = ::Math::Align<int32>(argBytesSize - MachPtr, 8);

if( FitsInByte( argBytesSize ) )

{

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int8>( RegESP, (int8)argBytesSize ) );

}

else

{

size += SUB::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( RegESP, argBytesSize ) );

}

return size;

}

int I\_ArgOut\_Int::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int argIndex, int offset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

offset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetStackReg<int>( buffer, templateData, offset, size );

InternalCallInfo\* callInfo = templateData->GetInternalCallInfo();

Assert( callInfo->nextArgIndex == argIndex );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsAddrReg( RegESP, callInfo->currentOffset, reg ) );

callInfo->currentOffset += sizeof( int );

++callInfo->nextArgIndex;

return size;

}

int I\_ArgOut\_Db::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int argIndex, int offset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

offset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetStackReg<double>( buffer, templateData, offset, size );

InternalCallInfo\* callInfo = templateData->GetInternalCallInfo();

Assert( callInfo->nextArgIndex == argIndex );

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsAddrReg( RegESP, callInfo->currentOffset, reg ) );

callInfo->currentOffset += sizeof( double );

callInfo->nextArgIndex += sizeof(double)/sizeof(Var);

return size;

}

int I\_ArgOut\_Flt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int argIndex, int offset)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

offset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetStackReg<float>(buffer, templateData, offset, size);

InternalCallInfo\* callInfo = templateData->GetInternalCallInfo();

Assert(callInfo->nextArgIndex == argIndex);

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsAddrReg(RegESP, callInfo->currentOffset, reg));

callInfo->currentOffset += sizeof(float);

++callInfo->nextArgIndex;

return size;

}

int I\_Call::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int funcOffset, int nbArgs, AsmJsRetType retType)

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

funcOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetStackReg<int>( buffer, templateData, funcOffset, size );

size += PUSH::EncodeInstruction<int>( buffer, InstrParamsReg(reg));

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(RegEAX, reg, RecyclableObject::GetTypeOffset()));

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(RegEAX, RegEAX, ScriptFunctionType::GetEntryPointInfoOffset()));

//GetAddressOffset from entrypointinfo

size += CALL::EncodeInstruction<int>(buffer, InstrParamsAddr(RegEAX, ProxyEntryPointInfo::GetAddressOffset()));

templateData->InvalidateAllVolatileReg();

templateData->InternalCallDone();

size += EncodingHelpers::ReloadArrayBuffer(context, buffer);

return size;

}

int AsmJsLoopBody::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int loopNumber)

{

int size = 0;

X86TemplateData\* templateData = GetTemplateData(context);

AsmJsFunctionInfo\* funcInfo = context->GetFunctionBody()->GetAsmJsFunctionInfo();

LoopHeader\* loopHeader = context->GetFunctionBody()->GetLoopHeader(loopNumber);

Var LoopEntryPoint = (LoopHeader(\*)(Js::FunctionBody\*, Var, uint32))DoLoopBodyStart;

int offsetCorrection = templateData->GetEBPOffsetCorrection() - templateData->GetBaseOffSet(); // no EBP correction is needed here as the offset is not coming from bytecode

int intOffset = funcInfo->GetIntByteOffset() + offsetCorrection;

int floatOffset = funcInfo->GetFloatByteOffset() + offsetCorrection;

int doubleOffset = funcInfo->GetDoubleByteOffset() + offsetCorrection;

// Increment the loop count(TJCount) , reusing interpretcount in the loopHeader

RegNum regInc = templateData->GetReg<int>(0);

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(regInc, (int32)loopHeader));

size += INC::EncodeInstruction<int>(buffer, InstrParamsAddr(regInc, loopHeader->GetOffsetOfInterpretCount()));

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(regInc, regInc,loopHeader->GetOffsetOfInterpretCount()));

// Compare InterpretCount(TJCount) with the threshold set for LIC and if it is less then do schedule for JitLoopBody

size += CMP::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(regInc, context->GetFunctionBody()->GetLoopInterpretCount(loopHeader)));

// Jmp $LabelCount in case count is not equal to the threshold

JumpRelocation relocLabelCount;

EncodingHelpers::EncodeShortJump<JL>(buffer, relocLabelCount, &size);

// If the loop is hot, Push the current EBP and loopNumber on the stack along with the function object

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, loopNumber));

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsReg(RegEAX));

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsReg(RegEBP));

size += PUSH::EncodeInstruction<int>(buffer, InstrParamsAddr(RegEBP, 2 \* sizeof(Var)));

// Call DoLoopBodyStart

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, (int32)LoopEntryPoint));

size += CALL::EncodeInstruction<int>(buffer, InstrParamsReg(RegEAX));

// invalidate all the volatile reg's as it is a return from a function call

templateData->InvalidateAllVolatileReg();

// Check the return value in EAX, this is the bytecode offset, if it is zero then loopBody is not yet jitted and we need to continue with TJ

// Else Jump to the offset location stored in EAX

size += CMP::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(RegEAX, 0));

JumpRelocation relocLabel1;

EncodingHelpers::EncodeShortJump<JE>(buffer, relocLabel1, &size);

// reload the array buffer after JIT loop body

size += EncodingHelpers::ReloadArrayBuffer(context, buffer);

// Before we jump, move the result to EAX in case we return from there

size += MOV::EncodeInstruction<int>(buffer, InstrParams2Reg(RegECX, RegEAX));

templateData->InvalidateReg(RegECX);

//get the output in the right register

Js::AsmJsRetType retType = funcInfo->GetReturnType();

switch (retType.which())

{

case Js::AsmJsRetType::Signed:

case Js::AsmJsRetType::Void:

size += MOV::EncodeInstruction<int>(buffer, InstrParamsRegAddr(RegEAX, RegEBP, intOffset));

templateData->InvalidateReg(RegEAX);

break;

case Js::AsmJsRetType::Float:

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsRegAddr(RegXMM0, RegEBP, floatOffset));

templateData->InvalidateReg(RegXMM0);

break;

case Js::AsmJsRetType::Double:

size += MOVSD::EncodeInstruction<double>(buffer, InstrParamsRegAddr(RegXMM0, RegEBP, doubleOffset));

templateData->InvalidateReg(RegXMM0);

break;

default:

Assume(UNREACHED);

}

// Jump to the offset

size += JMP::EncodeInstruction<int>(buffer, InstrParamsReg(RegECX));

// Label1:

relocLabel1.ApplyReloc<int8>();

//$LabelCount:

relocLabelCount.ApplyReloc<int8>();

return size;

}

int I\_Conv\_VTI::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , RegEAX);

return size;

}

int I\_Conv\_VTD::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

size += MOVSD::EncodeInstruction<double>(buffer, InstrParamsAddrReg(RegEBP, targetOffset,RegXMM0));

templateData->OverwriteStack( targetOffset );

return size;

}

int I\_Conv\_VTF::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

size += MOVSS::EncodeInstruction<float>(buffer, InstrParamsAddrReg(RegEBP, targetOffset, RegXMM0));

templateData->OverwriteStack(targetOffset);

return size;

}

int LdUndef::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

const int undefinedVar = (int)context->GetFunctionBody()->GetScriptContext()->GetLibrary()->GetUndefined();

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>(RegEAX,undefinedVar) );

templateData->InvalidateReg( RegEAX );

return size;

}

int LdArr\_Func::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int arrOffset, int slotVarIndexOffset )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

arrOffset -= templateData->GetBaseOffSet();

slotVarIndexOffset -= templateData->GetBaseOffSet();

RegNum regArr, regIndex;

if( !templateData->FindRegWithStackOffset<int>( regArr, arrOffset ) )

{

regArr = templateData->GetReg<int>();

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( regArr, RegEBP, arrOffset ) );

templateData->SetStackInfo( regArr, arrOffset );

}

if( !templateData->FindRegWithStackOffset<int>(regIndex,slotVarIndexOffset) )

{

regIndex = templateData->GetReg<int>( 1 << regArr );

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( regIndex, RegEBP, slotVarIndexOffset ) );

templateData->SetStackInfo( regIndex, slotVarIndexOffset );

}

// optimization because this value will be read only once right after this bytecode

RegNum targetReg = targetOffset == templateData->GetModuleSlotOffset() ? RegEAX : templateData->GetReg<int>();

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( targetReg, regArr, regIndex, 4, 0 ) );

if (targetOffset == templateData->GetModuleSlotOffset())

{

templateData->OverwriteStack( targetOffset );

templateData->SetStackInfo( RegEAX, targetOffset );

}

else

{

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , targetReg);

}

return size;

}

int LdSlot::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int arrOffset, int slotIndex )

{

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

arrOffset -= templateData->GetBaseOffSet();

RegNum reg;

if( !templateData->FindRegWithStackOffset<int>( reg, arrOffset ) )

{

reg = templateData->GetReg<int>();

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( reg, RegEBP, arrOffset ) );

templateData->SetStackInfo( reg, arrOffset );

}

if (targetOffset == templateData->GetModuleSlotOffset())

{

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( RegEAX, reg, slotIndex\*sizeof(Var) ) );

templateData->OverwriteStack( targetOffset );

templateData->SetStackInfo( RegEAX, targetOffset );

}

else

{

RegNum targetReg = templateData->GetReg<int>(1<<reg);

size += MOV::EncodeInstruction<int>( buffer, InstrParamsRegAddr( targetReg, reg, slotIndex\*sizeof(Var) ) );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , targetReg);

}

return size;

}

typedef int( \*MovEncodingFunc )( BYTE\*&, const InstrParamsRegAddr&, EncodingInfo\* info );

static const MovEncodingFunc ldArrMovEncodingFunc[] = {

MOVSX::EncodeInstruction<int8>//TYPE\_INT8 = 0,

,MOVZX::EncodeInstruction<int8>//TYPE\_UINT8,

,MOVSX::EncodeInstruction<int16>//TYPE\_INT16,

,MOVZX::EncodeInstruction<int16>//TYPE\_UINT16,

,MOV::EncodeInstruction<int>//TYPE\_INT32,

,MOV::EncodeInstruction<int>//TYPE\_UINT32,

,MOVSS::EncodeInstruction<float>//TYPE\_FLOAT32,

,MOVSD::EncodeInstruction<double>//TYPE\_FLOAT64,

};

typedef int( \*StArrMovEncodingFunc )( BYTE\*&, const InstrParamsAddrReg&, EncodingInfo\* info );

static const StArrMovEncodingFunc stArrMovEncodingFunc[] = {

MOV::EncodeInstruction<int8>//TYPE\_INT8 = 0,

,MOV::EncodeInstruction<int8>//TYPE\_UINT8,

,MOV::EncodeInstruction<int16>//TYPE\_INT16,

,MOV::EncodeInstruction<int16>//TYPE\_UINT16,

,MOV::EncodeInstruction<int>//TYPE\_INT32,

,MOV::EncodeInstruction<int>//TYPE\_UINT32,

,MOVSS::EncodeInstruction<float>//TYPE\_FLOAT32,

,MOVSD::EncodeInstruction<double>//TYPE\_FLOAT64,

};

static const uint32 TypedArrayViewMask[] =

{

(uint32)~0 //TYPE\_INT8

,(uint32)~0 //TYPE\_UINT8

,(uint32)~1 //TYPE\_INT16

,(uint32)~1 //TYPE\_UINT16

,(uint32)~3 //TYPE\_INT32

,(uint32)~3 //TYPE\_UINT32

,(uint32)~3 //TYPE\_FLOAT32

,(uint32)~7 //TYPE\_FLOAT64

};

int LdArrDb::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int slotVarIndex, ArrayBufferView::ViewType viewType )

{

AnalysisAssert(viewType == ArrayBufferView::TYPE\_FLOAT64);

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

slotVarIndex -= templateData->GetBaseOffSet();

RegNum regIndex = EncodingHelpers::GetStackReg<int>( buffer, templateData, slotVarIndex, size );

RegNum resultReg = templateData->GetReg<double>();

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister( buffer, context, size, 1 << regIndex );

size += AND::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( regIndex, TypedArrayViewMask[viewType] ) );

templateData->InvalidateReg( regIndex );

size += EncodingHelpers::CompareRegisterToArrayLength( buffer, context, regIndex );

// Jump to load value

JumpRelocation reloc( buffer, &size );

EncodingInfo info;

size += JBE::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc.JumpEncoded( info );

size += ldArrMovEncodingFunc[viewType]( buffer, InstrParamsRegAddr( resultReg, regArrayBuffer, regIndex, 1, 0 ), nullptr );

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , resultReg);

// Jump to load default value

JumpRelocation reloc2( buffer, &size );

size += JMP::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc2.JumpEncoded( info );

reloc.ApplyReloc<int8>();

int\* nanAddr = (int\*)&NumberConstants::k\_Nan;

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegPtr( resultReg, (void\*)nanAddr ) );

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , resultReg);

reloc2.ApplyReloc<int8>();

return size;

}

int LdArrFlt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int slotVarIndex, ArrayBufferView::ViewType viewType)

{

AnalysisAssert(viewType == ArrayBufferView::TYPE\_FLOAT32);

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

slotVarIndex -= templateData->GetBaseOffSet();

RegNum regIndex = EncodingHelpers::GetStackReg<int>(buffer, templateData, slotVarIndex, size);

RegNum resultReg = templateData->GetReg<float>();

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister(buffer, context, size, 1 << regIndex);

size += AND::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(regIndex, TypedArrayViewMask[viewType]));

templateData->InvalidateReg(regIndex);

size += EncodingHelpers::CompareRegisterToArrayLength(buffer, context, regIndex);

// Jump to load value

JumpRelocation reloc(buffer, &size);

EncodingInfo info;

size += JBE::EncodeInstruction<int>(buffer, InstrParamsImm<int8>(0), &info);

reloc.JumpEncoded(info);

size += ldArrMovEncodingFunc[viewType](buffer, InstrParamsRegAddr(resultReg, regArrayBuffer, regIndex, 1, 0), nullptr);

size += EncodingHelpers::SetStackReg<float>(buffer, templateData, targetOffset, resultReg);

// Jump to load default value

JumpRelocation reloc2(buffer, &size);

size += JMP::EncodeInstruction<int>(buffer, InstrParamsImm<int8>(0), &info);

reloc2.JumpEncoded(info);

reloc.ApplyReloc<int8>();

int\* nanAddr = (int\*)&NumberConstants::k\_Nan;

size += MOVSD::EncodeInstruction<double>(buffer, InstrParamsRegPtr(resultReg, (void\*)nanAddr));

size += CVTSD2SS::EncodeInstruction<double>(buffer, InstrParams2Reg(resultReg, resultReg));

size += EncodingHelpers::SetStackReg<float>(buffer, templateData, targetOffset, resultReg);

reloc2.ApplyReloc<int8>();

return size;

}

int LdArr::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int slotVarIndex, ArrayBufferView::ViewType viewType )

{

AnalysisAssert(viewType >= ArrayBufferView::TYPE\_INT8 && viewType < ArrayBufferView::TYPE\_FLOAT32);

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

slotVarIndex -= templateData->GetBaseOffSet();

RegNum regIndex = EncodingHelpers::GetStackReg<int>( buffer, templateData, slotVarIndex, size );

RegNum resultReg = templateData->GetReg<int>( 1 << regIndex );

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister( buffer, context, size, 1 << regIndex | 1 << resultReg );

if( viewType != ArrayBufferView::TYPE\_INT8 && viewType != ArrayBufferView::TYPE\_UINT8 )

{

size += AND::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( regIndex, TypedArrayViewMask[viewType] ) );

templateData->InvalidateReg( regIndex );

}

size += EncodingHelpers::CompareRegisterToArrayLength( buffer, context, regIndex );

// Jump to load value

JumpRelocation reloc( buffer, &size );

EncodingInfo info;

size += JBE::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc.JumpEncoded( info );

size += ldArrMovEncodingFunc[viewType]( buffer, InstrParamsRegAddr( resultReg, regArrayBuffer, regIndex, 1, 0 ), nullptr );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , resultReg);

// Jump to load default value

JumpRelocation reloc2( buffer, &size );

size += JMP::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc2.JumpEncoded( info );

reloc.ApplyReloc<int8>();

size += MOV::EncodeInstruction<int>( buffer, InstrParamsAddrImm<int32>( RegEBP, targetOffset, 0 ) );

// load the value into a register now since it will most likely be used very soon + avoids discrepancies in templateData between the 2 jumps

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( resultReg, resultReg) );

reloc2.ApplyReloc<int8>();

return size;

}

int StArrDb::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int srcOffset, int slotVarIndex, ArrayBufferView::ViewType viewType )

{

AnalysisAssert(viewType == ArrayBufferView::TYPE\_FLOAT64);

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

slotVarIndex -= templateData->GetBaseOffSet();

RegNum regIndex = EncodingHelpers::GetStackReg<int>( buffer, templateData, slotVarIndex, size );

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister( buffer, context, size, 1 << regIndex );

size += AND::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( regIndex, TypedArrayViewMask[viewType] ) );

templateData->InvalidateReg( regIndex );

size += EncodingHelpers::CompareRegisterToArrayLength( buffer, context, regIndex );

// Jump to load value

JumpRelocation reloc( buffer, &size );

EncodingInfo info;

size += JBE::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc.JumpEncoded( info );

RegNum regVal;

regVal = EncodingHelpers::GetStackReg<double>( buffer, templateData, srcOffset, size );

size += stArrMovEncodingFunc[viewType]( buffer, InstrParamsAddrReg( regArrayBuffer, regIndex, 1, 0, regVal ), nullptr );

// do nothing if index is out of range

reloc.ApplyReloc<int8>();

return size;

}

int StArrFlt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffset, int slotVarIndex, ArrayBufferView::ViewType viewType)

{

AnalysisAssert(viewType == ArrayBufferView::TYPE\_FLOAT32);

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

slotVarIndex -= templateData->GetBaseOffSet();

RegNum regIndex = EncodingHelpers::GetStackReg<int>(buffer, templateData, slotVarIndex, size);

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister(buffer, context, size, 1 << regIndex);

size += AND::EncodeInstruction<int>(buffer, InstrParamsRegImm<int32>(regIndex, TypedArrayViewMask[viewType]));

templateData->InvalidateReg(regIndex);

size += EncodingHelpers::CompareRegisterToArrayLength(buffer, context, regIndex);

// Jump to load value

JumpRelocation reloc(buffer, &size);

EncodingInfo info;

size += JBE::EncodeInstruction<int>(buffer, InstrParamsImm<int8>(0), &info);

reloc.JumpEncoded(info);

RegNum regVal;

regVal = EncodingHelpers::GetStackReg<float>(buffer, templateData, srcOffset, size);

size += stArrMovEncodingFunc[viewType](buffer, InstrParamsAddrReg(regArrayBuffer, regIndex, 1, 0, regVal), nullptr);

// do nothing if index is out of range

reloc.ApplyReloc<int8>();

return size;

}

int StArr::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int srcOffset, int slotVarIndex, ArrayBufferView::ViewType viewType )

{

AnalysisAssert(viewType >= ArrayBufferView::TYPE\_INT8 && viewType < ArrayBufferView::TYPE\_FLOAT32);

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

slotVarIndex -= templateData->GetBaseOffSet();

RegNum regIndex = EncodingHelpers::GetStackReg<int>( buffer, templateData, slotVarIndex, size );

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister( buffer, context, size, 1 << regIndex );

if( viewType != ArrayBufferView::TYPE\_INT8 && viewType != ArrayBufferView::TYPE\_UINT8 )

{

size += AND::EncodeInstruction<int>( buffer, InstrParamsRegImm<int32>( regIndex, TypedArrayViewMask[viewType] ) );

templateData->InvalidateReg( regIndex );

}

size += EncodingHelpers::CompareRegisterToArrayLength( buffer, context, regIndex );

// Jump to load value

JumpRelocation reloc( buffer, &size );

EncodingInfo info;

size += JBE::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc.JumpEncoded( info );

int extraRestriction = viewType == ArrayBufferView::TYPE\_INT8 || viewType == ArrayBufferView::TYPE\_UINT8 ? ~Mask8BitsReg : 0;

RegNum regVal = EncodingHelpers::GetStackReg<int>( buffer, templateData, srcOffset, size, 1 << regIndex | extraRestriction | 1 << regArrayBuffer );

size += stArrMovEncodingFunc[viewType]( buffer, InstrParamsAddrReg( regArrayBuffer, regIndex, 1, 0, regVal ), nullptr );

// do nothing if index is out of range

reloc.ApplyReloc<int8>();

return size;

}

// Version with const index

int ConstLdArrDb::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int constIndex, ArrayBufferView::ViewType viewType )

{

AnalysisAssert(viewType == ArrayBufferView::TYPE\_FLOAT64);

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

RegNum resultReg = templateData->GetReg<double>();

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister( buffer, context, size );

size += EncodingHelpers::CompareImmutableToArrayLength<int32>( buffer, context, constIndex );

// Jump to load value

JumpRelocation reloc( buffer, &size );

EncodingInfo info;

size += JBE::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc.JumpEncoded( info );

size += ldArrMovEncodingFunc[viewType]( buffer, InstrParamsRegAddr( resultReg, regArrayBuffer, constIndex ), nullptr );

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , resultReg);

// Jump to load default value

JumpRelocation reloc2( buffer, &size );

size += JMP::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc2.JumpEncoded( info );

reloc.ApplyReloc<int8>();

int\* nanAddr = (int\*)&NumberConstants::k\_Nan;

size += MOVSD::EncodeInstruction<double>( buffer, InstrParamsRegPtr( resultReg, (void\*)nanAddr ) );

size += EncodingHelpers::SetStackReg<double>( buffer, templateData, targetOffset , resultReg);

reloc2.ApplyReloc<int8>();

return size;

}

// Version with const index

int ConstLdArrFlt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int constIndex, ArrayBufferView::ViewType viewType)

{

AnalysisAssert(viewType == ArrayBufferView::TYPE\_FLOAT32);

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

RegNum resultReg = templateData->GetReg<float>();

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister(buffer, context, size);

size += EncodingHelpers::CompareImmutableToArrayLength<int32>(buffer, context, constIndex);

// Jump to load value

JumpRelocation reloc(buffer, &size);

EncodingInfo info;

size += JBE::EncodeInstruction<int>(buffer, InstrParamsImm<int8>(0), &info);

reloc.JumpEncoded(info);

size += ldArrMovEncodingFunc[viewType](buffer, InstrParamsRegAddr(resultReg, regArrayBuffer, constIndex), nullptr);

size += EncodingHelpers::SetStackReg<float>(buffer, templateData, targetOffset, resultReg);

// Jump to load default value

JumpRelocation reloc2(buffer, &size);

size += JMP::EncodeInstruction<int>(buffer, InstrParamsImm<int8>(0), &info);

reloc2.JumpEncoded(info);

reloc.ApplyReloc<int8>();

int\* nanAddr = (int\*)&NumberConstants::k\_Nan;

size += MOVSD::EncodeInstruction<double>(buffer, InstrParamsRegPtr(resultReg, (void\*)nanAddr));

size += CVTSD2SS::EncodeInstruction<double>(buffer, InstrParams2Reg(resultReg, resultReg));

size += EncodingHelpers::SetStackReg<float>(buffer, templateData, targetOffset, resultReg);

reloc2.ApplyReloc<int8>();

return size;

}

int ConstLdArr::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int targetOffset, int constIndex, ArrayBufferView::ViewType viewType )

{

AnalysisAssert(viewType < ArrayBufferView::TYPE\_FLOAT32 && viewType >= ArrayBufferView::TYPE\_INT8);

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

RegNum resultReg = templateData->GetReg<int>( );

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister( buffer, context, size, 1 << resultReg );

size += EncodingHelpers::CompareImmutableToArrayLength<int32>( buffer, context, constIndex );

// Jump to load value

JumpRelocation reloc( buffer, &size );

EncodingInfo info;

size += JBE::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc.JumpEncoded( info );

size += ldArrMovEncodingFunc[viewType]( buffer, InstrParamsRegAddr( resultReg, regArrayBuffer, constIndex ), nullptr );

size += EncodingHelpers::SetStackReg<int>( buffer, templateData, targetOffset , resultReg);

// Jump to load default value

JumpRelocation reloc2( buffer, &size );

size += JMP::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc2.JumpEncoded( info );

reloc.ApplyReloc<int8>();

size += MOV::EncodeInstruction<int>( buffer, InstrParamsAddrImm<int32>( RegEBP, targetOffset, 0 ) );

// load the value into a register now since it will most likely be used very soon + avoids discrepancies in templateData between the 2 jumps

size += XOR::EncodeInstruction<int>( buffer, InstrParams2Reg( resultReg, resultReg) );

reloc2.ApplyReloc<int8>();

return size;

}

template<typename Size>

int ConstStArrDbOrFlt(TemplateContext context, BYTE\*& buffer, int srcOffset, int constIndex, ArrayBufferView::ViewType viewType)

{

AnalysisAssert(viewType == ArrayBufferView::TYPE\_FLOAT32 || viewType == ArrayBufferView::TYPE\_FLOAT64);

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister(buffer, context, size);

size += EncodingHelpers::CompareImmutableToArrayLength<int32>(buffer, context, constIndex);

// Jump to load value

JumpRelocation reloc(buffer, &size);

EncodingInfo info;

size += JBE::EncodeInstruction<int>(buffer, InstrParamsImm<int8>(0), &info);

reloc.JumpEncoded(info);

RegNum regVal;

regVal = EncodingHelpers::GetStackReg<Size>(buffer, templateData, srcOffset, size);

size += stArrMovEncodingFunc[viewType](buffer, InstrParamsAddrReg(regArrayBuffer, constIndex, regVal), nullptr);

// do nothing if index is out of range

reloc.ApplyReloc<int8>();

return size;

}

int ConstStArrDb::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int srcOffset, int constIndex, ArrayBufferView::ViewType viewType )

{

Assert(viewType == ArrayBufferView::TYPE\_FLOAT64);

return ConstStArrDbOrFlt<double>(context, buffer, srcOffset, constIndex, viewType);

}

int ConstStArrFlt::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffset, int constIndex, ArrayBufferView::ViewType viewType)

{

Assert(viewType == ArrayBufferView::TYPE\_FLOAT32);

return ConstStArrDbOrFlt<float>(context, buffer, srcOffset, constIndex, viewType);

}

int ConstStArr::ApplyTemplate( TemplateContext context, BYTE\*& buffer, int srcOffset, int constIndex, ArrayBufferView::ViewType viewType )

{

AnalysisAssert(viewType < ArrayBufferView::TYPE\_FLOAT32 && viewType >= ArrayBufferView::TYPE\_INT8);

X86TemplateData\* templateData = GetTemplateData( context );

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

RegNum regArrayBuffer = EncodingHelpers::GetArrayBufferRegister( buffer, context, size );

size += EncodingHelpers::CompareImmutableToArrayLength<int32>( buffer, context, constIndex );

// Jump to load value

JumpRelocation reloc( buffer, &size );

EncodingInfo info;

size += JBE::EncodeInstruction<int>( buffer, InstrParamsImm<int8>( 0 ), &info );

reloc.JumpEncoded( info );

int extraRestriction = viewType == ArrayBufferView::TYPE\_INT8 || viewType == ArrayBufferView::TYPE\_UINT8 ? ~Mask8BitsReg : 0;

RegNum regVal = EncodingHelpers::GetStackReg<int>( buffer, templateData, srcOffset, size, extraRestriction | 1 << regArrayBuffer );

size += stArrMovEncodingFunc[viewType]( buffer, InstrParamsAddrReg( regArrayBuffer, constIndex, regVal ), nullptr );

// do nothing if index is out of range

reloc.ApplyReloc<int8>();

return size;

}

int Simd128\_Ld\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4, int srcOffsetF4)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4 -= templateData->GetBaseOffSet();

srcOffsetF4 -= templateData->GetBaseOffSet();

if (targetOffsetF4 == srcOffsetF4)

{

return 0;

}

int size = 0;

RegNum reg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4, size);

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4, reg);

return size;

}

int Simd128\_Ld\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4, int srcOffsetI4)

{

return Simd128\_Ld\_F4::ApplyTemplate(context, buffer, targetOffsetI4, srcOffsetI4);

}

int Simd128\_Ld\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2, int srcOffsetD2)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetD2 -= templateData->GetBaseOffSet();

srcOffsetD2 -= templateData->GetBaseOffSet();

if (targetOffsetD2 == srcOffsetD2)

{

return 0;

}

int size = 0;

RegNum reg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetD2, size);

size += EncodingHelpers::SIMDSetStackReg<double>(buffer, templateData, targetOffsetD2, reg);

return size;

}

int Simd128\_LdSlot\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int slotIndex)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

targetOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetModuleEnvironmentRegister(buffer, context, size);

RegNum reg2 = templateData->GetReg<AsmJsSIMDValue>(1 << reg);

size += MOVUPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegAddr(reg2, reg, slotIndex\*sizeof(AsmJsSIMDValue)));

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffset, reg2);

return size;

}

int Simd128\_LdSlot\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int slotIndex)

{

return Simd128\_LdSlot\_F4::ApplyTemplate(context, buffer, targetOffset, slotIndex);

}

int Simd128\_LdSlot\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int slotIndex)

{

return Simd128\_LdSlot\_F4::ApplyTemplate(context, buffer, targetOffset, slotIndex);

}

int Simd128\_StSlot\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffset, int slotIndex)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

srcOffset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetModuleEnvironmentRegister(buffer, context, size);

RegNum reg2 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffset, size);

size += MOVUPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsAddrReg(reg, slotIndex\*sizeof(AsmJsSIMDValue), reg2));

return size;

}

int Simd128\_StSlot\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffset, int slotIndex)

{

return Simd128\_StSlot\_F4::ApplyTemplate(context, buffer, srcOffset, slotIndex);

}

int Simd128\_StSlot\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffset, int slotIndex)

{

return Simd128\_StSlot\_F4::ApplyTemplate(context, buffer, srcOffset, slotIndex);

}

int Simd128\_FloatsToF4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF1, int srcOffsetF2, int srcOffsetF3, int srcOffsetF4)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDInitFromPrimitives<float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF1, srcOffsetF2, srcOffsetF3, srcOffsetF4);

}

int Simd128\_IntsToI4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI1, int srcOffsetI2, int srcOffsetI3, int srcOffsetI4)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDInitFromPrimitives<int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI1, srcOffsetI2, srcOffsetI3, srcOffsetI4);

}

int Simd128\_DoublesToD2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD1, int srcOffsetD2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDInitFromPrimitives<double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD1, srcOffsetD2);

}

int Simd128\_Return\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffsetF4)

{

X86TemplateData\* templateData = GetTemplateData(context);

srcOffsetF4 -= templateData->GetBaseOffSet();

RegNum reg = RegXMM0;

if (!templateData->FindRegWithStackOffset<AsmJsSIMDValue>(reg, srcOffsetF4))

{

templateData->SetStackInfo(RegXMM0, srcOffsetF4);

return MOVUPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegAddr(RegXMM0, RegEBP, srcOffsetF4));

}

if (reg != RegXMM0)

{

return MOVUPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(RegXMM0, reg));

}

return 0;

}

int Simd128\_Return\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffsetI4)

{

return Simd128\_Return\_F4::ApplyTemplate(context, buffer, srcOffsetI4);

}

int Simd128\_Return\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int srcOffsetD2)

{

return Simd128\_Return\_F4::ApplyTemplate(context, buffer, srcOffsetD2);

}

int Simd128\_Splat\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4\_0 -= templateData->GetBaseOffSet();

srcOffsetF1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg = EncodingHelpers::GetStackReg<float>(buffer, templateData, srcOffsetF1, size);

size += SHUFPS::EncodeInstruction<AsmJsSIMDValue, byte>(buffer, InstrParams2RegImm<byte>(reg, reg, 0x00));

// MOVUPS

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4\_0, reg);

return size;

}

int Simd128\_Splat\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetI4\_0 -= templateData->GetBaseOffSet();

srcOffsetI1 -= templateData->GetBaseOffSet();

int size = 0;

// load as float: MOVSS XMM, [intVal]

RegNum reg = EncodingHelpers::GetStackReg<float>(buffer, templateData, srcOffsetI1, size);

size += PSHUFD::EncodeInstruction<AsmJsSIMDValue, byte>(buffer, InstrParams2RegImm<byte>(reg, reg, 0x00));

// MOVUPS

size += EncodingHelpers::SIMDSetStackReg<int>(buffer, templateData, targetOffsetI4\_0, reg);

return size;

}

int Simd128\_Splat\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetD2\_0 -= templateData->GetBaseOffSet();

srcOffsetD1 -= templateData->GetBaseOffSet();

int size = 0;

// MOVSD

RegNum reg = EncodingHelpers::GetStackReg<double>(buffer, templateData, srcOffsetD1, size);

size += SHUFPD::EncodeInstruction<AsmJsSIMDValue, byte>(buffer, InstrParams2RegImm<byte>(reg, reg, 0x00));

size += EncodingHelpers::SIMDSetStackReg<double>(buffer, templateData, targetOffsetD2\_0, reg);

return size;

}

// Type conversions

int Simd128\_FromFloat64x2\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetD2\_1)

{

return EncodingHelpers::SIMDUnaryOperation<CVTPD2PS, float>(buffer, GetTemplateData(context), targetOffsetF4\_0, srcOffsetD2\_1);

}

int Simd128\_FromInt32x4\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetI4\_1)

{

return EncodingHelpers::SIMDUnaryOperation<CVTDQ2PS, float>(buffer, GetTemplateData(context), targetOffsetF4\_0, srcOffsetI4\_1);

}

int Simd128\_FromFloat32x4\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetF4\_1)

{

return EncodingHelpers::SIMDUnaryOperation<CVTTPS2DQ, int>(buffer, GetTemplateData(context), targetOffsetI4\_0, srcOffsetF4\_1);

}

int Simd128\_FromFloat64x2\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetD2\_1)

{

return EncodingHelpers::SIMDUnaryOperation<CVTTPD2DQ, int>(buffer, GetTemplateData(context), targetOffsetI4\_0, srcOffsetD2\_1);

}

int Simd128\_FromFloat32x4\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetF4\_1)

{

return EncodingHelpers::SIMDUnaryOperation<CVTPS2PD, double>(buffer, GetTemplateData(context), targetOffsetD2\_0, srcOffsetF4\_1);

}

int Simd128\_FromInt32x4\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetI4\_1)

{

return EncodingHelpers::SIMDUnaryOperation<CVTDQ2PD, float>(buffer, GetTemplateData(context), targetOffsetD2\_0, srcOffsetI4\_1);

}

// Bits conversions

int Simd128\_FromFloat64x2Bits\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetD2\_1)

{

return Simd128\_Ld\_F4::ApplyTemplate(context, buffer, targetOffsetF4\_0, srcOffsetD2\_1);

}

int Simd128\_FromInt32x4Bits\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetI4\_1)

{

return Simd128\_Ld\_F4::ApplyTemplate(context, buffer, targetOffsetF4\_0, srcOffsetI4\_1);

}

int Simd128\_FromFloat32x4Bits\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetF4\_1)

{

return Simd128\_Ld\_I4::ApplyTemplate(context, buffer, targetOffsetI4\_0, srcOffsetF4\_1);

}

int Simd128\_FromFloat64x2Bits\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetD2\_1)

{

return Simd128\_Ld\_I4::ApplyTemplate(context, buffer, targetOffsetI4\_0, srcOffsetD2\_1);

}

int Simd128\_FromFloat32x4Bits\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetF4\_1)

{

return Simd128\_Ld\_D2::ApplyTemplate(context, buffer, targetOffsetD2\_0, srcOffsetF4\_1);

}

int Simd128\_FromInt32x4Bits\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetI4\_1)

{

return Simd128\_Ld\_D2::ApplyTemplate(context, buffer, targetOffsetD2\_0, srcOffsetI4\_1);

}

// Unary operations

int Simd128\_Abs\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4\_0 -= templateData->GetBaseOffSet();

srcOffsetF4\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg1;

// MOVUPS reg, [src]

reg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_1, size);

// ANDPS reg, [mask]

size += ANDPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegPtr(reg1, &(X86\_ABS\_MASK\_F4)));

// MOVUPS [dst], reg

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4\_0, reg1);

return size;

}

int Simd128\_Abs\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetD2\_0 -= templateData->GetBaseOffSet();

srcOffsetD2\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg1;

// MOVUPS reg, [src]

reg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetD2\_1, size);

// ANDPS reg, [mask]

size += ANDPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegPtr(reg1, &(X86\_ABS\_MASK\_D2)));

// MOVUPS [dst], reg

size += EncodingHelpers::SIMDSetStackReg<double>(buffer, templateData, targetOffsetD2\_0, reg1);

return size;

}

int Simd128\_Neg\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4\_0 -= templateData->GetBaseOffSet();

srcOffsetF4\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg1;

// MOVUPS reg, [src]

reg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_1, size);

// XORPS reg, [mask]

size += XORPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegPtr(reg1, &(X86\_NEG\_MASK\_F4)));

// MOVUPS [dst], reg

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4\_0, reg1);

return size;

}

int Simd128\_Neg\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetI4\_0 -= templateData->GetBaseOffSet();

srcOffsetI4\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg1;

// MOVUPS reg, [src]

reg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetI4\_1, size);

// ANDNPS reg, [mask]

size += ANDNPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegPtr(reg1, &(X86\_ALL\_NEG\_ONES)));

// PADDD reg, [all\_ones]

size += PADDD::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegPtr(reg1, &(X86\_ALL\_ONES\_I4)));

// MOVUPS [dst], reg

size += EncodingHelpers::SIMDSetStackReg<int>(buffer, templateData, targetOffsetI4\_0, reg1);

return size;

}

int Simd128\_Neg\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetD2\_0 -= templateData->GetBaseOffSet();

srcOffsetD2\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg1;

// MOVUPS reg, [src]

reg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetD2\_1, size);

// XORPS reg, [mask]

size += XORPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegPtr(reg1, &(X86\_NEG\_MASK\_D2)));

// MOVUPS [dst], reg

size += EncodingHelpers::SIMDSetStackReg<double>(buffer, templateData, targetOffsetD2\_0, reg1);

return size;

}

int Simd128\_Rcp\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4\_0 -= templateData->GetBaseOffSet();

srcOffsetF4\_1 -= templateData->GetBaseOffSet();

int size = 0;

// MOVUPS srcReg, [src]

RegNum srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_1, size);

RegNum rcpReg = EncodingHelpers::SIMDRcpOperation<DIVPS, float>(buffer, templateData, srcReg, (void\*)(&X86\_ALL\_ONES\_F4), size);

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4\_0, rcpReg);

return size;

}

int Simd128\_Rcp\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetD2\_0 -= templateData->GetBaseOffSet();

srcOffsetD2\_1 -= templateData->GetBaseOffSet();

int size = 0;

// MOVUPS reg1, [src]

RegNum srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetD2\_1, size);

RegNum rcpReg = EncodingHelpers::SIMDRcpOperation<DIVPD, double>(buffer, templateData, srcReg, (void\*)(&X86\_ALL\_ONES\_D2), size);

size += EncodingHelpers::SIMDSetStackReg<double>(buffer, templateData, targetOffsetD2\_0, rcpReg);

return size;

}

int Simd128\_RcpSqrt\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4\_0 -= templateData->GetBaseOffSet();

srcOffsetF4\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_1, size);

RegNum dstReg = EncodingHelpers::SIMDRcpOperation<DIVPS, float>(buffer, templateData, srcReg, (void\*)(&X86\_ALL\_ONES\_F4), size);

size += SQRTPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, dstReg));

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4\_0, dstReg);

return size;

}

int Simd128\_RcpSqrt\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetD2\_0 -= templateData->GetBaseOffSet();

srcOffsetD2\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetD2\_1, size);

RegNum dstReg = EncodingHelpers::SIMDRcpOperation<DIVPD, double>(buffer, templateData, srcReg, (void\*)(&X86\_ALL\_ONES\_D2), size);

size += SQRTPD::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, dstReg));

size += EncodingHelpers::SIMDSetStackReg<double>(buffer, templateData, targetOffsetD2\_0, dstReg);

return size;

}

int Simd128\_Sqrt\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4\_0 -= templateData->GetBaseOffSet();

srcOffsetF4\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_1, size);

size += SQRTPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(reg, reg));

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4\_0, reg);

return size;

}

int Simd128\_Sqrt\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetD2\_0 -= templateData->GetBaseOffSet();

srcOffsetD2\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetD2\_1, size);

size += SQRTPD::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(reg, reg));

size += EncodingHelpers::SIMDSetStackReg<double>(buffer, templateData, targetOffsetD2\_0, reg);

return size;

}

int Simd128\_Not\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4\_0 -= templateData->GetBaseOffSet();

srcOffsetF4\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum reg1;

// MOVUPS reg, [src]

reg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_1, size);

// XORPS reg, [mask]

size += XORPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegPtr(reg1, &(X86\_ALL\_NEG\_ONES)));

// MOVUPS [dst], reg

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4\_0, reg1);

return size;

}

int Simd128\_Not\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1)

{

return Simd128\_Not\_F4::ApplyTemplate(context, buffer, targetOffsetI4\_0, srcOffsetI4\_1);

}

int Simd128\_Add\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<ADDPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2);

}

int Simd128\_Add\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<PADDD, int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_1, srcOffsetI4\_2);

}

int Simd128\_Add\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<ADDPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2);

}

int Simd128\_Sub\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<SUBPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2);

}

int Simd128\_Sub\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<PSUBD, int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_1, srcOffsetI4\_2);

}

int Simd128\_Sub\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<SUBPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2);

}

int Simd128\_Mul\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<MULPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2);

}

int Simd128\_Mul\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

RegNum srcReg1, srcReg2, tmpReg;

int size = 0;

targetOffsetI4\_0 -= templateData->GetBaseOffSet();

srcOffsetI4\_1 -= templateData->GetBaseOffSet();

srcOffsetI4\_2 -= templateData->GetBaseOffSet();

// MOVUPS srcReg1, [src1]

// MOVUPS srcReg1, [src2]

srcReg1 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetI4\_1, size);

srcReg2 = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetI4\_2, size);

tmpReg = templateData->GetReg<AsmJsSIMDValue>((1 << srcReg1) | (1 << srcReg2));

// MOVAPS tmpReg, srcReg1

size += MOVAPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tmpReg, srcReg1));

// PMULUDQ tmpReg, srcReg2

size += PMULUDQ::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tmpReg, srcReg2));

// PSRLDQ srcReg1, 0x04

size += PSRLDQ::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegImm<byte>(srcReg1, 0x04));

templateData->InvalidateReg(srcReg1);

if (srcReg1 != srcReg2)

{

// PSRLDQ srcReg2, 0x04

size += PSRLDQ::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsRegImm<byte>(srcReg2, 0x04));

templateData->InvalidateReg(srcReg2);

}

// PMULUDQ srcReg1, srcReg2

size += PMULUDQ::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(srcReg1, srcReg2));

// PSHUFD tmpReg, tmpReg, b00001000

size += PSHUFD::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2RegImm<byte>(tmpReg, tmpReg, 0x08));

// PSHUFD srcReg1, srcReg1, b00001000

size += PSHUFD::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2RegImm<byte>(srcReg1, srcReg1, 0x08));

// PUNPCKLDQ srcReg1, tmpReg

size += PUNPCKLDQ::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tmpReg, srcReg1));

// MOVUPS [dst] srcReg1

size += EncodingHelpers::SIMDSetStackReg<int>(buffer, templateData, targetOffsetI4\_0, tmpReg);

return size;

}

int Simd128\_Mul\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<MULPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2);

}

int Simd128\_Div\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<DIVPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2);

}

int Simd128\_Div\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<DIVPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2);

}

int Simd128\_Min\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<MINPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2);

}

int Simd128\_Min\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<MINPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2);

}

int Simd128\_Max\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<MAXPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2);

}

int Simd128\_Max\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<MAXPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2);

}

// comparison

int Simd128\_Lt\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<CMPPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2, CMP\_IMM8::LT);

}

int Simd128\_Lt\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<PCMPGTD, int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_2, srcOffsetI4\_1);

}

int Simd128\_Lt\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<CMPPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2, CMP\_IMM8::LT);

}

int Simd128\_Gt\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<CMPPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_2, srcOffsetF4\_1, CMP\_IMM8::LT);

}

int Simd128\_Gt\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<PCMPGTD, int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_1, srcOffsetI4\_2);

}

int Simd128\_Gt\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<CMPPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_2, srcOffsetD2\_1, CMP\_IMM8::LT);

}

int Simd128\_LtEq\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<CMPPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2, CMP\_IMM8::LE);

}

int Simd128\_LtEq\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<CMPPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2, CMP\_IMM8::LE);

}

int Simd128\_GtEq\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<CMPPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_2, srcOffsetF4\_1, CMP\_IMM8::LE);

}

int Simd128\_GtEq\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<CMPPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_2, srcOffsetD2\_1, CMP\_IMM8::LE);

}

int Simd128\_Eq\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<CMPPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2, CMP\_IMM8::EQ);

}

int Simd128\_Eq\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<PCMPEQD, int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_2, srcOffsetI4\_1);

}

int Simd128\_Eq\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<CMPPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2, CMP\_IMM8::EQ);

}

int Simd128\_Neq\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<CMPPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF4\_2, CMP\_IMM8::NEQ);

}

int Simd128\_Neq\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetD2\_1, int srcOffsetD2\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<CMPPD, double>(buffer, templateData, targetOffsetD2\_0, srcOffsetD2\_1, srcOffsetD2\_2, CMP\_IMM8::NEQ);

}

int Simd128\_And\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<ANDPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_2, srcOffsetF4\_1);

}

int Simd128\_And\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<PAND, int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_1, srcOffsetI4\_2);

}

int Simd128\_Or\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<ORPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_2, srcOffsetF4\_1);

}

int Simd128\_Or\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<POR, int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_1, srcOffsetI4\_2);

}

int Simd128\_Xor\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

// reversed operands

return EncodingHelpers::SIMDBinaryOperation<XORPS, float>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_2, srcOffsetF4\_1);

}

int Simd128\_Xor\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2)

{

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDBinaryOperation<PXOR, int>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_1, srcOffsetI4\_2);

}

int Simd128\_Select\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetI4\_1, int srcOffsetF4\_2, int srcOffsetF4\_3)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetF4\_0 -= templateData->GetBaseOffSet();

srcOffsetI4\_1 -= templateData->GetBaseOffSet();

srcOffsetF4\_2 -= templateData->GetBaseOffSet();

srcOffsetF4\_3 -= templateData->GetBaseOffSet();

RegNum maskReg, tReg, fReg, tempReg;

int size = 0;

int restrictions = 0;

maskReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetI4\_1, size);

restrictions |= (1 << maskReg);

tReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_2, size, restrictions);

restrictions |= (1 << tReg);

fReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_3, size, restrictions);

restrictions |= (1 << fReg);

tempReg = templateData->GetReg<AsmJsSIMDValue>(restrictions);

// MOVAPS tempReg, maskReg

size += MOVAPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tempReg, maskReg));

// ANDPS tempReg, tReg

size += ANDPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tempReg, tReg));

// ANDNPS maskReg, fReg

size += ANDNPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(maskReg, fReg));

templateData->InvalidateReg(maskReg);

// ORPS tempReg, maskReg

size += ORPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(tempReg, maskReg));

// MOVUPS [dst], tempReg

size += EncodingHelpers::SIMDSetStackReg<float>(buffer, templateData, targetOffsetF4\_0, tempReg);

return size;

}

int Simd128\_Select\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI4\_2, int srcOffsetI4\_3)

{

// ok to re-use F4, size of I4 lane >= size of F4 lane. Important for correct invalidation of regs upon store to stack.

return Simd128\_Select\_F4::ApplyTemplate(context, buffer, targetOffsetI4\_0, srcOffsetI4\_1, srcOffsetI4\_2, srcOffsetI4\_3);

}

int Simd128\_Select\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetD2\_0, int srcOffsetI4\_1, int srcOffsetD2\_2, int srcOffsetD2\_3)

{

// ok to re-use F4, size of D2 lane >= size of F4 lane. Important for correct invalidation of regs upon store to stack.

return Simd128\_Select\_F4::ApplyTemplate(context, buffer, targetOffsetD2\_0, srcOffsetI4\_1, srcOffsetD2\_2, srcOffsetD2\_3);

}

//Lane Access

int Simd128\_ExtractLane\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI0, int srcOffsetI4\_1, int index)

{

AssertMsg(index >= 0 && index < 4, "Invalid lane index");

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDLdLaneOperation<MOVSS, int>(buffer, templateData, targetOffsetI0, srcOffsetI4\_1, index, false);

}

int Simd128\_ExtractLane\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF0, int srcOffsetF4\_1, int index)

{

AssertMsg(index >= 0 && index < 4, "Invalid lane index");

X86TemplateData\* templateData = GetTemplateData(context);

return EncodingHelpers::SIMDLdLaneOperation<MOVSS, int>(buffer, templateData, targetOffsetF0, srcOffsetF4\_1, index, false);

}

int Simd128\_ReplaceLane\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetF4\_0, int srcOffsetF4\_1, int srcOffsetF2, int laneIndex)

{

X86TemplateData\* templateData = GetTemplateData(context);

AssertMsg(laneIndex >= 0 && laneIndex < 4, "Invalid lane index");

return EncodingHelpers::SIMDSetLaneOperation<float, SHUFPS>(buffer, templateData, targetOffsetF4\_0, srcOffsetF4\_1, srcOffsetF2, laneIndex);

}

int Simd128\_ReplaceLane\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI4\_0, int srcOffsetI4\_1, int srcOffsetI2, int laneIndex)

{

X86TemplateData\* templateData = GetTemplateData(context);

AssertMsg(laneIndex >= 0 && laneIndex < 4, "Invalid lane index");

return EncodingHelpers::SIMDSetLaneOperation<int, PSHUFD>(buffer, templateData, targetOffsetI4\_0, srcOffsetI4\_1, srcOffsetI2, laneIndex);

}

int Simd128\_LdSignMask\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI0, int srcOffsetF4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetI0 -= templateData->GetBaseOffSet();

srcOffsetF4\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum srcReg, dstReg;

srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetF4\_1, size);

dstReg = templateData->GetReg<int>();

size += MOVMSKPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, srcReg));

size += EncodingHelpers::SetStackReg<int>(buffer, templateData, targetOffsetI0, dstReg);

return size;

}

int Simd128\_LdSignMask\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI0, int srcOffsetI4\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetI0 -= templateData->GetBaseOffSet();

srcOffsetI4\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum srcReg, dstReg;

srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetI4\_1, size);

dstReg = templateData->GetReg<int>();

size += MOVMSKPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, srcReg));

size += EncodingHelpers::SetStackReg<int>(buffer, templateData, targetOffsetI0, dstReg);

return size;

}

int Simd128\_LdSignMask\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffsetI0, int srcOffsetD2\_1)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffsetI0 -= templateData->GetBaseOffSet();

srcOffsetD2\_1 -= templateData->GetBaseOffSet();

int size = 0;

RegNum srcReg, dstReg;

srcReg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, srcOffsetD2\_1, size);

dstReg = templateData->GetReg<int>();

size += MOVMSKPD::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParams2Reg(dstReg, srcReg));

size += EncodingHelpers::SetStackReg<int>(buffer, templateData, targetOffsetI0, dstReg);

return size;

}

int Simd128\_I\_ArgOut\_F4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int argIndex, int offset)

{

X86TemplateData\* templateData = GetTemplateData(context);

int size = 0;

offset -= templateData->GetBaseOffSet();

RegNum reg = EncodingHelpers::GetStackReg<AsmJsSIMDValue>(buffer, templateData, offset, size);

InternalCallInfo\* callInfo = templateData->GetInternalCallInfo();

Assert(callInfo->nextArgIndex == argIndex);

size += MOVUPS::EncodeInstruction<AsmJsSIMDValue>(buffer, InstrParamsAddrReg(RegESP, callInfo->currentOffset, reg));

callInfo->currentOffset += sizeof(AsmJsSIMDValue);

callInfo->nextArgIndex += sizeof(AsmJsSIMDValue) / sizeof(Var);

return size;

}

int Simd128\_I\_ArgOut\_I4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int argIndex, int offset)

{

return Simd128\_I\_ArgOut\_F4::ApplyTemplate(context, buffer, argIndex, offset);

}

int Simd128\_I\_ArgOut\_D2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int argIndex, int offset)

{

return Simd128\_I\_ArgOut\_F4::ApplyTemplate(context, buffer, argIndex, offset);

}

int Simd128\_I\_Conv\_VTF4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset)

{

X86TemplateData\* templateData = GetTemplateData(context);

targetOffset -= templateData->GetBaseOffSet();

srcOffset -= templateData->GetBaseOffSet();

return EncodingHelpers::SIMDSetStackReg(buffer, templateData, targetOffset, RegXMM0);

}

int Simd128\_I\_Conv\_VTI4::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset)

{

return Simd128\_I\_Conv\_VTF4::ApplyTemplate(context, buffer, targetOffset, srcOffset);

}

int Simd128\_I\_Conv\_VTD2::ApplyTemplate(TemplateContext context, BYTE\*& buffer, int targetOffset, int srcOffset)

{

return Simd128\_I\_Conv\_VTF4::ApplyTemplate(context, buffer, targetOffset, srcOffset);

}

};

}

#endif

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#if !defined(\_M\_IX86)

#error X86StackFrame is not supported on this architecture.

#endif

namespace Js

{

bool

X86StackFrame::InitializeByFrameId(void \* frame, ScriptContext\* scriptContext)

{

this->frame = (void \*\*)frame;

this->stackCheckCodeHeight =

scriptContext->GetThreadContext()->DoInterruptProbe() ? stackCheckCodeHeightWithInterruptProbe

: scriptContext->GetThreadContext()->GetIsThreadBound() ? stackCheckCodeHeightThreadBound

: stackCheckCodeHeightNotThreadBound;

return Next();

}

bool

X86StackFrame::InitializeByReturnAddress(void \* returnAddress, ScriptContext\* scriptContext)

{

void \*\* framePtr;

\_\_asm

{

mov framePtr, ebp;

}

this->frame = framePtr;

this->stackCheckCodeHeight =

scriptContext->GetThreadContext()->DoInterruptProbe() ? stackCheckCodeHeightWithInterruptProbe

: scriptContext->GetThreadContext()->GetIsThreadBound() ? stackCheckCodeHeightThreadBound

: stackCheckCodeHeightNotThreadBound;

while (Next())

{

if (this->codeAddr == returnAddress)

{

return true;

}

}

return false;

}

bool

X86StackFrame::Next()

{

this->addressOfCodeAddr = this->GetAddressOfReturnAddress();

this->codeAddr = this->GetReturnAddress();

this->frame = (void \*\*)this->frame[0];

return frame != nullptr;

}

bool

X86StackFrame::SkipToFrame(void \* frameAddress)

{

this->frame = (void \*\*)frameAddress;

return Next();

}

bool

X86StackFrame::IsInStackCheckCode(void \*entry, void \*codeAddr, size\_t stackCheckCodeHeight)

{

return ((size\_t(codeAddr) - size\_t(entry)) <= stackCheckCodeHeight);

}

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

#ifndef \_M\_IX86

#error This is only for x86

#endif

class X86StackFrame {

public:

X86StackFrame() : frame(nullptr), codeAddr(nullptr), stackCheckCodeHeight(0), addressOfCodeAddr(nullptr) {};

bool InitializeByFrameId(void \* frameAddress, ScriptContext\* scriptContext);

bool InitializeByReturnAddress(void \* returnAddress, ScriptContext\* scriptContext);

bool Next();

void \* GetInstructionPointer() { return codeAddr; }

void \*\* GetArgv(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true) { return frame + 2; } // parameters unused for x86, arm and arm64

void \* GetReturnAddress(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true) { return frame[1]; } // parameters unused for x86, arm and arm64

void \* GetAddressOfReturnAddress(bool isCurrentContextNative = false, bool shouldCheckForNativeAddr = true) { return &frame[1]; } // parameters unused for x86, arm and arm64

void \* GetAddressOfInstructionPointer() const { return addressOfCodeAddr; }

void \* GetFrame() const { return (void \*)frame; }

void SetReturnAddress(void \* address) { frame[1] = address; }

bool SkipToFrame(void \* frameAddress);

size\_t GetStackCheckCodeHeight() { return this->stackCheckCodeHeight; }

static bool IsInStackCheckCode(void \*entry, void \*codeAddr, size\_t stackCheckCodeHeight);

private:

void \*\* frame; // ebp

void \* codeAddr; // eip

void \* addressOfCodeAddr;

size\_t stackCheckCodeHeight;

static const size\_t stackCheckCodeHeightThreadBound = StackFrameConstants::StackCheckCodeHeightThreadBound;

static const size\_t stackCheckCodeHeightNotThreadBound = StackFrameConstants::StackCheckCodeHeightNotThreadBound;

static const size\_t stackCheckCodeHeightWithInterruptProbe = StackFrameConstants::StackCheckCodeHeightWithInterruptProbe;

};

};