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

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

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

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

#include "RuntimeMathPch.h"

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 AsmJsMath

{

public:

template<typename T> static T Add( T aLeft, T aRight );

template<typename T> static T Sub( T aLeft, T aRight );

template<typename T> static T Mul( T aLeft, T aRight );

template<typename T> static T Div( T aLeft, T aRight );

template<typename T> static T Rem( T aLeft, T aRight );

template<typename T> static T Min( T aLeft, T aRight );

template<typename T> static T Max( T aLeft, T aRight );

static int And( int aLeft, int aRight );

static int Or( int aLeft, int aRight );

static int Xor( int aLeft, int aRight );

static int Shl( int aLeft, int aRight );

static int Shr( int aLeft, int aRight );

static int ShrU( int aLeft, int aRight );

template<typename T> static T Neg( T aLeft);

static int Not( int aLeft);

static int LogNot( int aLeft);

static int ToBool( int aLeft );

static int Clz32( int value);

template<typename T> static int CmpLt( T aLeft, T aRight );

template<typename T> static int CmpLe( T aLeft, T aRight );

template<typename T> static int CmpGt( T aLeft, T aRight );

template<typename T> static int CmpGe( T aLeft, T aRight );

template<typename T> static int CmpEq( T aLeft, T aRight );

template<typename T> static int CmpNe( T aLeft, T aRight );

};

}

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

// 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<typename T>

\_\_inline T AsmJsMath::Min(T aLeft, T aRight)

{

return aLeft < aRight ? aLeft : aRight;

}

template<>

\_\_inline double AsmJsMath::Min<double>(double aLeft, double aRight)

{

if (NumberUtilities::IsNan(aLeft) || NumberUtilities::IsNan(aRight))

{

return NumberConstants::NaN;

}

return aLeft < aRight ? aLeft : aRight;

}

template<typename T>

\_\_inline T AsmJsMath::Max(T aLeft, T aRight)

{

return aLeft > aRight ? aLeft : aRight;

}

template<>

\_\_inline double AsmJsMath::Max<double>(double aLeft, double aRight)

{

if (NumberUtilities::IsNan(aLeft) || NumberUtilities::IsNan(aRight))

{

return NumberConstants::NaN;

}

return aLeft > aRight ? aLeft : aRight;

}

template<typename T>

\_\_inline T AsmJsMath::Add( T aLeft, T aRight )

{

return aLeft + aRight;

}

template<typename T>

\_\_inline T AsmJsMath::Div( T aLeft, T aRight )

{

return aRight == 0 ? 0 : ( aLeft == (1<<31) && aRight == -1) ? aLeft : aLeft / aRight;

}

template<>

\_\_inline double AsmJsMath::Div<double>( double aLeft, double aRight )

{

return aLeft / aRight;

}

template<typename T>

\_\_inline T AsmJsMath::Mul( T aLeft, T aRight )

{

return aLeft \* aRight;

}

template<>

\_\_inline int AsmJsMath::Mul( int aLeft, int aRight )

{

return (int)((int64)aLeft \* (int64)aRight);

}

template<typename T>

\_\_inline T AsmJsMath::Sub( T aLeft, T aRight )

{

return aLeft - aRight;

}

template<typename T> \_\_inline int AsmJsMath::CmpLt( T aLeft, T aRight ){return (int)(aLeft < aRight);}

template<typename T> \_\_inline int AsmJsMath::CmpLe( T aLeft, T aRight ){return (int)(aLeft <= aRight);}

template<typename T> \_\_inline int AsmJsMath::CmpGt( T aLeft, T aRight ){return (int)(aLeft > aRight);}

template<typename T> \_\_inline int AsmJsMath::CmpGe( T aLeft, T aRight ){return (int)(aLeft >= aRight);}

template<typename T> \_\_inline int AsmJsMath::CmpEq( T aLeft, T aRight ){return (int)(aLeft == aRight);}

template<typename T> \_\_inline int AsmJsMath::CmpNe( T aLeft, T aRight ){return (int)(aLeft != aRight);}

template<typename T>

\_\_inline T AsmJsMath::Rem( T aLeft, T aRight )

{

return (aRight == 0) ? 0 : aLeft % aRight;

}

template<>

\_\_inline int AsmJsMath::Rem<int>( int aLeft, int aRight )

{

return ((aRight == 0) || (aLeft == (1<<31) && aRight == -1)) ? 0 : aLeft % aRight;

}

template<>

\_\_inline double AsmJsMath::Rem<double>( double aLeft, double aRight )

{

return NumberUtilities::Modulus( aLeft, aRight );

}

\_\_inline int AsmJsMath::And( int aLeft, int aRight )

{

return aLeft & aRight;

}

\_\_inline int AsmJsMath::Or( int aLeft, int aRight )

{

return aLeft | aRight;

}

\_\_inline int AsmJsMath::Xor( int aLeft, int aRight )

{

return aLeft ^ aRight;

}

\_\_inline int AsmJsMath::Shl( int aLeft, int aRight )

{

return aLeft << aRight;

}

\_\_inline int AsmJsMath::Shr( int aLeft, int aRight )

{

return aLeft >> aRight;

}

\_\_inline int AsmJsMath::ShrU( int aLeft, int aRight )

{

return (unsigned int)aLeft >> (unsigned int)aRight;

}

template<typename T>

\_\_inline T AsmJsMath::Neg( T aLeft )

{

return -aLeft;

}

\_\_inline int AsmJsMath::Not( int aLeft )

{

return ~aLeft;

}

\_\_inline int AsmJsMath::LogNot( int aLeft )

{

return !aLeft;

}

\_\_inline int AsmJsMath::ToBool( int aLeft )

{

return !!aLeft;

}

inline int AsmJsMath::Clz32( int value)

{

DWORD index;

if (\_BitScanReverse(&index, value))

{

return 31 - index;

}

return 32;

}

}

<?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">

<ProjectGuid>{ABC904AD-9415-46F8-AA23-E33193F81F7C}</ProjectGuid>

<RootNamespace>JS</RootNamespace>

<Keyword>Win32Proj</Keyword>

<TargetName>Chakra.Runtime.Math</TargetName>

</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" />

<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>RuntimeMathPch.h</PrecompiledHeaderFile>

</ClCompile>

</ItemDefinitionGroup>

<ItemDefinitionGroup Condition="'$(Platform)'=='Win32'">

<ClCompile>

<EnableEnhancedInstructionSet>StreamingSIMDExtensions2</EnableEnhancedInstructionSet>

</ClCompile>

</ItemDefinitionGroup>

<ItemGroup>

<ClCompile Include="$(MSBuildThisFileDirectory)AsmJsMath.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)JavascriptSSE2MathOperators.cpp">

<ExcludedFromBuild Condition="'$(Platform)'!='Win32'">true</ExcludedFromBuild>

</ClCompile>

<ClCompile Include="$(MSBuildThisFileDirectory)RuntimeMathPch.cpp">

<PrecompiledHeader>Create</PrecompiledHeader>

</ClCompile>

</ItemGroup>

<ItemGroup>

<None Include="JavascriptMath.cpp" />

<ClInclude Include="AsmJsMath.h" />

<ClInclude Include="CrtSSE2Math.h" />

<ClInclude Include="RuntimeMathPch.h" />

<ClInclude Include="JavascriptMath.h" />

<ClInclude Include="JavascriptSSE2MathOperators.h" />

</ItemGroup>

<ItemGroup>

<None Include="AsmJsMath.inl" />

<None Include="javascriptmath.inl" />

<None Include="JavascriptSSE2MathOperators.inl" />

</ItemGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.targets" Condition="exists('$(BuildConfigPropsPath)Chakra.Build.targets')"/>

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />

</Project>

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

// 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 \_M\_IX86

// This CRT routines skip some special condition checks for FPU state

// These routines are not expected to be called outside of the CRT, so using these should

// be re-evaluated when upgrading toolsets.

// Mark them explicitly as dllimport to workaround VC bug (Dev11:909888)

#ifdef USE\_STATIC\_RUNTIMELIB

#define \_DLLIMPORT

#else

#define \_DLLIMPORT \_\_declspec(dllimport)

#endif

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_acos(double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_asin(double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_atan(double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_atan2(double,double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_cos(double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_exp(double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_pow(double,double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_log(double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_log10(double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_sin(double);

extern "C" double \_DLLIMPORT \_\_cdecl \_\_libm\_sse2\_tan(double);

#undef \_DLLIMPORT

#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

{

#ifdef SSE2MATH

namespace SSE2

{

#endif

Var JavascriptMath::Negate\_Full(Var aRight, ScriptContext\* scriptContext)

{

// Special case for zero. Must return -0

if( aRight == TaggedInt::ToVarUnchecked(0) )

{

return scriptContext->GetLibrary()->GetNegativeZero();

}

double value = Negate\_Helper(aRight, scriptContext);

return JavascriptNumber::ToVarNoCheck(value, scriptContext);

}

Var JavascriptMath::Negate\_InPlace(Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

// Special case for zero. Must return -0

if( aRight == TaggedInt::ToVarUnchecked(0) )

{

return scriptContext->GetLibrary()->GetNegativeZero();

}

double value = Negate\_Helper(aRight, scriptContext);

return JavascriptNumber::InPlaceNew(value, scriptContext, result);

}

Var JavascriptMath::Not\_Full(Var aRight, ScriptContext\* scriptContext)

{

#if \_M\_IX86

AssertMsg(!TaggedInt::Is(aRight), "Should be detected");

#endif

int nValue = JavascriptConversion::ToInt32(aRight, scriptContext);

return JavascriptNumber::ToVar(~nValue, scriptContext);

}

Var JavascriptMath::Not\_InPlace(Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

AssertMsg(!TaggedInt::Is(aRight), "Should be detected");

int nValue = JavascriptConversion::ToInt32(aRight, scriptContext);

return JavascriptNumber::ToVarInPlace(~nValue, scriptContext, result);

}

Var JavascriptMath::Increment\_InPlace(Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if (TaggedInt::Is(aRight))

{

return TaggedInt::Increment(aRight, scriptContext);

}

double inc = Increment\_Helper(aRight, scriptContext);

return JavascriptNumber::InPlaceNew(inc, scriptContext, result);

}

Var JavascriptMath::Increment\_Full(Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aRight))

{

return TaggedInt::Increment(aRight, scriptContext);

}

double inc = Increment\_Helper(aRight, scriptContext);

return JavascriptNumber::ToVarNoCheck(inc, scriptContext);

}

Var JavascriptMath::Decrement\_InPlace(Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if (TaggedInt::Is(aRight))

{

return TaggedInt::Decrement(aRight, scriptContext);

}

double dec = Decrement\_Helper(aRight,scriptContext);

return JavascriptNumber::InPlaceNew(dec, scriptContext, result);

}

Var JavascriptMath::Decrement\_Full(Var aRight, ScriptContext\* scriptContext)

{

if (TaggedInt::Is(aRight))

{

return TaggedInt::Decrement(aRight, scriptContext);

}

double dec = Decrement\_Helper(aRight,scriptContext);

return JavascriptNumber::ToVarNoCheck(dec, scriptContext);

}

Var JavascriptMath::And\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

int32 and = And\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::ToVar(and, scriptContext);

}

Var JavascriptMath::And\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

int32 and = And\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::ToVarInPlace(and, scriptContext, result);

}

Var JavascriptMath::Or\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

int32 or = Or\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::ToVar(or, scriptContext);

}

Var JavascriptMath::Or\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

int32 or = Or\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::ToVarInPlace(or, scriptContext, result);

}

Var JavascriptMath::Xor\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

int32 nLeft = TaggedInt::Is(aLeft) ? TaggedInt::ToInt32(aLeft) : JavascriptConversion::ToInt32(aLeft, scriptContext);

int32 nRight = TaggedInt::Is(aRight) ? TaggedInt::ToInt32(aRight) : JavascriptConversion::ToInt32(aRight, scriptContext);

return JavascriptNumber::ToVar(nLeft ^ nRight,scriptContext);

}

Var JavascriptMath::Xor\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

int32 nLeft = TaggedInt::Is(aLeft) ? TaggedInt::ToInt32(aLeft) : JavascriptConversion::ToInt32(aLeft, scriptContext);

int32 nRight = TaggedInt::Is(aRight) ? TaggedInt::ToInt32(aRight) : JavascriptConversion::ToInt32(aRight, scriptContext);

return JavascriptNumber::ToVarInPlace(nLeft ^ nRight, scriptContext, result);

}

Var JavascriptMath::ShiftLeft\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

int32 nValue = JavascriptConversion::ToInt32(aLeft, scriptContext);

uint32 nShift = JavascriptConversion::ToUInt32(aRight, scriptContext);

int32 nResult = nValue << (nShift & 0x1F);

return JavascriptNumber::ToVar(nResult,scriptContext);

}

Var JavascriptMath::ShiftRight\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

int32 nValue = JavascriptConversion::ToInt32(aLeft, scriptContext);

uint32 nShift = JavascriptConversion::ToUInt32(aRight, scriptContext);

int32 nResult = nValue >> (nShift & 0x1F);

return JavascriptNumber::ToVar(nResult,scriptContext);

}

Var JavascriptMath::ShiftRightU\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

uint32 nValue = JavascriptConversion::ToUInt32(aLeft, scriptContext);

uint32 nShift = JavascriptConversion::ToUInt32(aRight, scriptContext);

uint32 nResult = nValue >> (nShift & 0x1F);

return JavascriptNumber::ToVar(nResult,scriptContext);

}

#if FLOATVAR

Var JavascriptMath::Add\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

// If both sides are numbers, then we can do the addition directly, otherwise

// we need to call the helper.

if(JavascriptNumber::Is(aLeft))

{

if(JavascriptNumber::Is(aRight))

{

double sum = JavascriptNumber::GetValue(aLeft) + JavascriptNumber::GetValue(aRight);

return JavascriptNumber::ToVarNoCheck(sum, scriptContext);

}

else if(TaggedInt::Is(aRight))

{

double sum = TaggedInt::ToDouble(aRight) + JavascriptNumber::GetValue(aLeft);

return JavascriptNumber::ToVarNoCheck(sum, scriptContext);

}

}

else if(JavascriptNumber::Is(aRight))

{

if(TaggedInt::Is(aLeft))

{

double sum = TaggedInt::ToDouble(aLeft) + JavascriptNumber::GetValue(aRight);

return JavascriptNumber::ToVarNoCheck(sum, scriptContext);

}

}

else if(TaggedInt::Is(aLeft))

{

if(TaggedInt::Is(aRight))

{

\_\_int64 sum = TaggedInt::ToInt64(aLeft) + TaggedInt::ToInt64(aRight);

return JavascriptNumber::ToVar(sum, scriptContext);

}

}

else if (TaggedInt::Is(aRight))

{

return Add\_FullHelper\_Wrapper(aLeft, aRight, scriptContext, nullptr, false);

}

else if (RecyclableObject::FromVar(aLeft)->GetTypeId() == TypeIds\_String && RecyclableObject::FromVar(aRight)->GetTypeId() == TypeIds\_String)

{

return JavascriptString::Concat(JavascriptString::FromVar(aLeft), JavascriptString::FromVar(aRight));

}

return Add\_FullHelper\_Wrapper(aLeft, aRight, scriptContext, nullptr, false);

}

#else

Var JavascriptMath::Add\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

Js::TypeId typeLeft = JavascriptOperators::GetTypeId(aLeft);

Js::TypeId typeRight = JavascriptOperators::GetTypeId(aRight);

// Handle combinations of TaggedInt and Number or String pairs directly,

// otherwise call the helper.

switch( typeLeft )

{

case TypeIds\_Integer:

{

switch( typeRight )

{

case TypeIds\_Integer:

{

// Compute the sum using integer addition, then convert to double.

// That way there's only one int->float conversion.

#if INT32VAR

int64 sum = TaggedInt::ToInt64(aLeft) + TaggedInt::ToInt64(aRight);

#else

int32 sum = TaggedInt::ToInt32(aLeft) + TaggedInt::ToInt32(aRight);

#endif

return JavascriptNumber::ToVar(sum, scriptContext );

}

case TypeIds\_Number:

{

double sum = TaggedInt::ToDouble(aLeft) + JavascriptNumber::GetValue(aRight);

return JavascriptNumber::NewInlined( sum, scriptContext );

}

}

break;

}

case TypeIds\_Number:

{

switch( typeRight )

{

case TypeIds\_Integer:

{

double sum = JavascriptNumber::GetValue(aLeft) + TaggedInt::ToDouble(aRight);

return JavascriptNumber::NewInlined( sum, scriptContext );

}

case TypeIds\_Number:

{

double sum = JavascriptNumber::GetValue(aLeft) + JavascriptNumber::GetValue(aRight);

return JavascriptNumber::NewInlined( sum, scriptContext );

}

}

break;

}

case TypeIds\_String:

{

if( typeRight == TypeIds\_String )

{

JavascriptString\* leftString = JavascriptString::FromVar(aLeft);

JavascriptString\* rightString = JavascriptString::FromVar(aRight);

return JavascriptString::Concat(leftString, rightString);

}

break;

}

}

return Add\_FullHelper\_Wrapper(aLeft, aRight, scriptContext, nullptr, false);

}

#endif

Var JavascriptMath::Add\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

Assert(result != nullptr);

// If both sides are numbers, then we can do the addition directly, otherwise

// we need to call the helper.

if( TaggedInt::Is(aLeft) )

{

if( TaggedInt::Is(aRight) )

{

// Compute the sum using integer addition, then convert to double.

// That way there's only one int->float conversion.

#if INT32VAR

int64 sum = TaggedInt::ToInt64(aLeft) + TaggedInt::ToInt64(aRight);

#else

int32 sum = TaggedInt::ToInt32(aLeft) + TaggedInt::ToInt32(aRight);

#endif

return JavascriptNumber::ToVarInPlace(sum, scriptContext, result);

}

else if( JavascriptNumber::Is\_NoTaggedIntCheck(aRight) )

{

double sum = TaggedInt::ToDouble(aLeft) + JavascriptNumber::GetValue(aRight);

return JavascriptNumber::InPlaceNew( sum, scriptContext, result );

}

}

else if( TaggedInt::Is(aRight) )

{

if( JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) )

{

double sum = JavascriptNumber::GetValue(aLeft) + TaggedInt::ToDouble(aRight);

return JavascriptNumber::InPlaceNew( sum, scriptContext, result );

}

}

else if( JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight) )

{

double sum = JavascriptNumber::GetValue(aLeft) + JavascriptNumber::GetValue(aRight);

return JavascriptNumber::InPlaceNew( sum, scriptContext, result );

}

return Add\_FullHelper\_Wrapper(aLeft, aRight, scriptContext, result, false);

}

Var JavascriptMath::AddLeftDead(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber \*result)

{

if (JavascriptOperators::GetTypeId(aLeft) == TypeIds\_String)

{

JavascriptString\* leftString = JavascriptString::FromVar(aLeft);

JavascriptString\* rightString;

TypeId rightType = JavascriptOperators::GetTypeId(aRight);

switch(rightType)

{

case TypeIds\_String:

rightString = JavascriptString::FromVar(aRight);

StringCommon:

return leftString->ConcatDestructive(rightString);

case TypeIds\_Integer:

rightString = scriptContext->GetIntegerString(aRight);

goto StringCommon;

case TypeIds\_Number:

rightString = JavascriptNumber::ToStringRadix10(JavascriptNumber::GetValue(aRight), scriptContext);

goto StringCommon;

}

}

if (TaggedInt::Is(aLeft))

{

if (TaggedInt::Is(aRight))

{

return TaggedInt::Add(aLeft, aRight, scriptContext);

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::ToVarMaybeInPlace(TaggedInt::ToDouble(aLeft) + JavascriptNumber::GetValue(aRight), scriptContext, result);

}

}

else if (TaggedInt::Is(aRight))

{

if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft))

{

return JavascriptNumber::ToVarMaybeInPlace(JavascriptNumber::GetValue(aLeft) + TaggedInt::ToDouble(aRight), scriptContext, result);

}

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(aLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(aRight))

{

return JavascriptNumber::ToVarMaybeInPlace(JavascriptNumber::GetValue(aLeft) + JavascriptNumber::GetValue(aRight), scriptContext, result);

}

return Add\_FullHelper\_Wrapper(aLeft, aRight, scriptContext, result, true);

}

Var JavascriptMath::Add\_FullHelper\_Wrapper(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result, bool leftIsDead)

{

Var aLeftToPrim = JavascriptConversion::ToPrimitive(aLeft, JavascriptHint::None, scriptContext);

Var aRightToPrim = JavascriptConversion::ToPrimitive(aRight, JavascriptHint::None, scriptContext);

return Add\_FullHelper(aLeftToPrim, aRightToPrim, scriptContext, result, leftIsDead);

}

Var JavascriptMath::Add\_FullHelper(Var primLeft, Var primRight, ScriptContext\* scriptContext, JavascriptNumber \*result, bool leftIsDead)

{

// If either side is a string, then the result is also a string

if (JavascriptOperators::GetTypeId(primLeft) == TypeIds\_String)

{

JavascriptString\* stringLeft = JavascriptString::FromVar(primLeft);

JavascriptString\* stringRight = nullptr;

if (JavascriptOperators::GetTypeId(primRight) == TypeIds\_String)

{

stringRight = JavascriptString::FromVar(primRight);

}

else

{

stringRight = JavascriptConversion::ToString(primRight, scriptContext);

}

if(leftIsDead)

{

return stringLeft->ConcatDestructive(stringRight);

}

return JavascriptString::Concat(stringLeft, stringRight);

}

if (JavascriptOperators::GetTypeId(primRight) == TypeIds\_String)

{

JavascriptString\* stringLeft = JavascriptConversion::ToString(primLeft, scriptContext);

JavascriptString\* stringRight = JavascriptString::FromVar(primRight);

if(leftIsDead)

{

return stringLeft->ConcatDestructive(stringRight);

}

return JavascriptString::Concat(stringLeft, stringRight);

}

double sum = Add\_Helper(primLeft, primRight, scriptContext);

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

Var JavascriptMath::MulAddLeft(Var mulLeft, Var mulRight, Var addLeft, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if(TaggedInt::Is(mulLeft))

{

if(TaggedInt::Is(mulRight))

{

// Compute the sum using integer addition, then convert to double.

// That way there's only one int->float conversion.

JavascriptNumber mulTemp(0, scriptContext->GetLibrary()->GetNumberTypeStatic());

Var mulResult = TaggedInt::MultiplyInPlace(mulLeft, mulRight, scriptContext, &mulTemp);

if (result)

{

return JavascriptMath::Add\_InPlace(addLeft, mulResult, scriptContext, result);

}

else

{

return JavascriptMath::Add\_Full(addLeft, mulResult, scriptContext);

}

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(mulRight))

{

double mulResult = TaggedInt::ToDouble(mulLeft) \* JavascriptNumber::GetValue(mulRight);

return JavascriptMath::Add\_DoubleHelper(addLeft, mulResult, scriptContext, result);

}

}

else if(TaggedInt::Is(mulRight))

{

if(JavascriptNumber::Is\_NoTaggedIntCheck(mulLeft))

{

double mulResult = JavascriptNumber::GetValue(mulLeft) \* TaggedInt::ToDouble(mulRight);

return JavascriptMath::Add\_DoubleHelper(addLeft, mulResult, scriptContext, result);

}

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(mulLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(mulRight))

{

double mulResult = JavascriptNumber::GetValue(mulLeft) \* JavascriptNumber::GetValue(mulRight);

return JavascriptMath::Add\_DoubleHelper(addLeft, mulResult, scriptContext, result);

}

Var aMul;

JavascriptNumber mulTemp(0, scriptContext->GetLibrary()->GetNumberTypeStatic());

aMul = JavascriptMath::Multiply\_InPlace(mulLeft, mulRight, scriptContext, &mulTemp);

if (result)

{

return JavascriptMath::Add\_InPlace(addLeft, aMul, scriptContext, result);

}

else

{

return JavascriptMath::Add\_Full(addLeft, aMul, scriptContext);

}

}

Var JavascriptMath::MulAddRight(Var mulLeft, Var mulRight, Var addRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if(TaggedInt::Is(mulLeft))

{

if(TaggedInt::Is(mulRight))

{

// Compute the sum using integer addition, then convert to double.

// That way there's only one int->float conversion.

JavascriptNumber mulTemp(0, scriptContext->GetLibrary()->GetNumberTypeStatic());

Var mulResult = TaggedInt::MultiplyInPlace(mulLeft, mulRight, scriptContext, &mulTemp);

if (result)

{

return JavascriptMath::Add\_InPlace(mulResult, addRight, scriptContext, result);

}

else

{

return JavascriptMath::Add\_Full(mulResult, addRight, scriptContext);

}

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(mulRight))

{

double mulResult = TaggedInt::ToDouble(mulLeft) \* JavascriptNumber::GetValue(mulRight);

return JavascriptMath::Add\_DoubleHelper(mulResult, addRight, scriptContext, result);

}

}

else if(TaggedInt::Is(mulRight))

{

if(JavascriptNumber::Is\_NoTaggedIntCheck(mulLeft))

{

double mulResult = JavascriptNumber::GetValue(mulLeft) \* TaggedInt::ToDouble(mulRight);

return JavascriptMath::Add\_DoubleHelper(mulResult, addRight, scriptContext, result);

}

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(mulLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(mulRight))

{

double mulResult = JavascriptNumber::GetValue(mulLeft) \* JavascriptNumber::GetValue(mulRight);

return JavascriptMath::Add\_DoubleHelper(mulResult, addRight, scriptContext, result);

}

Var aMul;

JavascriptNumber mulTemp(0, scriptContext->GetLibrary()->GetNumberTypeStatic());

aMul = JavascriptMath::Multiply\_InPlace(mulLeft, mulRight, scriptContext, &mulTemp);

if (result)

{

return JavascriptMath::Add\_InPlace(aMul, addRight, scriptContext, result);

}

else

{

return JavascriptMath::Add\_Full(aMul, addRight, scriptContext);

}

}

Var JavascriptMath::MulSubLeft(Var mulLeft, Var mulRight, Var subLeft, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if(TaggedInt::Is(mulLeft))

{

if(TaggedInt::Is(mulRight))

{

// Compute the sum using integer addition, then convert to double.

// That way there's only one int->float conversion.

JavascriptNumber mulTemp(0, scriptContext->GetLibrary()->GetNumberTypeStatic());

Var mulResult = TaggedInt::MultiplyInPlace(mulLeft, mulRight, scriptContext, &mulTemp);

if (result)

{

return JavascriptMath::Subtract\_InPlace(subLeft, mulResult, scriptContext, result);

}

else

{

return JavascriptMath::Subtract\_Full(subLeft, mulResult, scriptContext);

}

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(mulRight))

{

double mulResult = TaggedInt::ToDouble(mulLeft) \* JavascriptNumber::GetValue(mulRight);

return JavascriptMath::Subtract\_DoubleHelper(subLeft, mulResult, scriptContext, result);

}

}

else if(TaggedInt::Is(mulRight))

{

if(JavascriptNumber::Is\_NoTaggedIntCheck(mulLeft))

{

double mulResult = JavascriptNumber::GetValue(mulLeft) \* TaggedInt::ToDouble(mulRight);

return JavascriptMath::Subtract\_DoubleHelper(subLeft, mulResult, scriptContext, result);

}

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(mulLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(mulRight))

{

double mulResult = JavascriptNumber::GetValue(mulLeft) \* JavascriptNumber::GetValue(mulRight);

return JavascriptMath::Subtract\_DoubleHelper(subLeft, mulResult, scriptContext, result);

}

Var aMul;

JavascriptNumber mulTemp(0, scriptContext->GetLibrary()->GetNumberTypeStatic());

aMul = JavascriptMath::Multiply\_InPlace(mulLeft, mulRight, scriptContext, &mulTemp);

if (result)

{

return JavascriptMath::Subtract\_InPlace(subLeft, aMul, scriptContext, result);

}

else

{

return JavascriptMath::Subtract\_Full(subLeft, aMul, scriptContext);

}

}

Var JavascriptMath::MulSubRight(Var mulLeft, Var mulRight, Var subRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if(TaggedInt::Is(mulLeft))

{

if(TaggedInt::Is(mulRight))

{

// Compute the sum using integer addition, then convert to double.

// That way there's only one int->float conversion.

JavascriptNumber mulTemp(0, scriptContext->GetLibrary()->GetNumberTypeStatic());

Var mulResult = TaggedInt::MultiplyInPlace(mulLeft, mulRight, scriptContext, &mulTemp);

if (result)

{

return JavascriptMath::Subtract\_InPlace(mulResult, subRight, scriptContext, result);

}

else

{

return JavascriptMath::Subtract\_Full(mulResult, subRight, scriptContext);

}

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(mulRight))

{

double mulResult = TaggedInt::ToDouble(mulLeft) \* JavascriptNumber::GetValue(mulRight);

return JavascriptMath::Subtract\_DoubleHelper(mulResult, subRight, scriptContext, result);

}

}

else if(TaggedInt::Is(mulRight))

{

if(JavascriptNumber::Is\_NoTaggedIntCheck(mulLeft))

{

double mulResult = JavascriptNumber::GetValue(mulLeft) \* TaggedInt::ToDouble(mulRight);

return JavascriptMath::Subtract\_DoubleHelper(mulResult, subRight, scriptContext, result);

}

}

else if(JavascriptNumber::Is\_NoTaggedIntCheck(mulLeft) && JavascriptNumber::Is\_NoTaggedIntCheck(mulRight))

{

double mulResult = JavascriptNumber::GetValue(mulLeft) \* JavascriptNumber::GetValue(mulRight);

return JavascriptMath::Subtract\_DoubleHelper(mulResult, subRight, scriptContext, result);

}

Var aMul;

JavascriptNumber mulTemp(0, scriptContext->GetLibrary()->GetNumberTypeStatic());

aMul = JavascriptMath::Multiply\_InPlace(mulLeft, mulRight, scriptContext, &mulTemp);

if (result)

{

return JavascriptMath::Subtract\_InPlace(aMul, subRight, scriptContext, result);

}

else

{

return JavascriptMath::Subtract\_Full(aMul, subRight, scriptContext);

}

}

Var \_\_inline JavascriptMath::Add\_DoubleHelper(double dblLeft, Var addRight, ScriptContext\* scriptContext, JavascriptNumber\*result)

{

if (TaggedInt::Is(addRight))

{

double sum = dblLeft + TaggedInt::ToDouble(addRight);

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(addRight))

{

double sum = dblLeft + JavascriptNumber::GetValue(addRight);

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

else

{

Var aLeft = JavascriptNumber::ToVarMaybeInPlace(dblLeft, scriptContext, result);

return Add\_Full(aLeft, addRight, scriptContext);

}

}

Var \_\_inline JavascriptMath::Add\_DoubleHelper(Var addLeft, double dblRight, ScriptContext\* scriptContext, JavascriptNumber\*result)

{

if (TaggedInt::Is(addLeft))

{

double sum = TaggedInt::ToDouble(addLeft) + dblRight;

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(addLeft))

{

double sum = JavascriptNumber::GetValue(addLeft) + dblRight;

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

else

{

Var aRight = JavascriptNumber::ToVarMaybeInPlace(dblRight, scriptContext, result);

return Add\_Full(addLeft, aRight, scriptContext);

}

}

Var \_\_inline JavascriptMath::Subtract\_DoubleHelper(double dblLeft, Var subRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if (TaggedInt::Is(subRight))

{

double sum = dblLeft - TaggedInt::ToDouble(subRight);

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(subRight))

{

double sum = dblLeft - JavascriptNumber::GetValue(subRight);

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

else

{

Var aLeft = JavascriptNumber::ToVarMaybeInPlace(dblLeft, scriptContext, result);

return Subtract\_Full(aLeft, subRight, scriptContext);

}

}

Var \_\_inline JavascriptMath::Subtract\_DoubleHelper(Var subLeft, double dblRight, ScriptContext\* scriptContext, JavascriptNumber\*result)

{

if (TaggedInt::Is(subLeft))

{

double sum = TaggedInt::ToDouble(subLeft) - dblRight;

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

else if (JavascriptNumber::Is\_NoTaggedIntCheck(subLeft))

{

double sum = JavascriptNumber::GetValue(subLeft) - dblRight;

return JavascriptNumber::ToVarMaybeInPlace(sum, scriptContext, result);

}

else

{

Var aRight = JavascriptNumber::ToVarMaybeInPlace(dblRight, scriptContext, result);

return Subtract\_Full(subLeft, aRight, scriptContext);

}

}

Var JavascriptMath::Subtract\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

double difference = Subtract\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::ToVarNoCheck(difference, scriptContext);

}

Var JavascriptMath::Subtract\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

double difference = Subtract\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::InPlaceNew(difference, scriptContext, result);

}

Var JavascriptMath::Divide\_Full(Var aLeft,Var aRight, ScriptContext\* scriptContext)

{

// If both arguments are TaggedInt, then try to do integer division

// This case is not handled by the lowerer.

if (TaggedInt::IsPair(aLeft, aRight))

{

return TaggedInt::Divide(aLeft, aRight, scriptContext);

}

return JavascriptNumber::NewInlined( Divide\_Helper(aLeft, aRight, scriptContext), scriptContext );

}

Var JavascriptMath::Exponentiation\_Full(Var aLeft, Var aRight, ScriptContext \*scriptContext)

{

double x = JavascriptConversion::ToNumber(aLeft, scriptContext);

double y = JavascriptConversion::ToNumber(aRight, scriptContext);

return JavascriptNumber::ToVarNoCheck(Math::Pow(x, y), scriptContext);

}

Var JavascriptMath::Exponentiation\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

// The IEEE 754 floating point spec ensures that NaNs are preserved in all operations

double dblLeft = JavascriptConversion::ToNumber(aLeft, scriptContext);

double dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

return JavascriptNumber::InPlaceNew(Math::Pow(dblLeft, dblRight), scriptContext, result);

}

Var JavascriptMath::Multiply\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

if(JavascriptNumber::Is(aLeft))

{

if(JavascriptNumber::Is(aRight))

{

double product = JavascriptNumber::GetValue(aLeft) \* JavascriptNumber::GetValue(aRight);

return JavascriptNumber::ToVarNoCheck(product, scriptContext);

}

else if(TaggedInt::Is(aRight))

{

double product = TaggedInt::ToDouble(aRight) \* JavascriptNumber::GetValue(aLeft);

return JavascriptNumber::ToVarNoCheck(product, scriptContext);

}

}

else if(JavascriptNumber::Is(aRight))

{

if(TaggedInt::Is(aLeft))

{

double product = TaggedInt::ToDouble(aLeft) \* JavascriptNumber::GetValue(aRight);

return JavascriptNumber::ToVarNoCheck(product, scriptContext);

}

}

else if(TaggedInt::IsPair(aLeft, aRight))

{

return TaggedInt::Multiply(aLeft, aRight, scriptContext);

}

double product = Multiply\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::ToVarNoCheck(product, scriptContext);

}

Var JavascriptMath::Multiply\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

if(JavascriptNumber::Is(aLeft))

{

if(JavascriptNumber::Is(aRight))

{

return JavascriptNumber::ToVarInPlace(

JavascriptNumber::GetValue(aLeft) \* JavascriptNumber::GetValue(aRight), scriptContext, result);

}

else if (TaggedInt::Is(aRight))

{

return JavascriptNumber::ToVarInPlace(

JavascriptNumber::GetValue(aLeft) \* TaggedInt::ToDouble(aRight), scriptContext, result);

}

}

else if(JavascriptNumber::Is(aRight))

{

if(TaggedInt::Is(aLeft))

{

return JavascriptNumber::ToVarInPlace(

TaggedInt::ToDouble(aLeft) \* JavascriptNumber::GetValue(aRight), scriptContext, result);

}

}

else if(TaggedInt::IsPair(aLeft, aRight))

{

return TaggedInt::MultiplyInPlace(aLeft, aRight, scriptContext, result);

}

double product = Multiply\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::InPlaceNew(product, scriptContext, result);

}

Var JavascriptMath::Divide\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

// If both arguments are TaggedInt, then try to do integer division

// This case is not handled by the lowerer.

if (TaggedInt::IsPair(aLeft, aRight))

{

return TaggedInt::DivideInPlace(aLeft, aRight, scriptContext, result);

}

double quotient = Divide\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::InPlaceNew(quotient, scriptContext, result);

}

Var JavascriptMath::Modulus\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

// If both arguments are TaggedInt, then try to do integer modulus.

// This case is not handled by the lowerer.

if (TaggedInt::IsPair(aLeft, aRight))

{

return TaggedInt::Modulus(aLeft, aRight, scriptContext);

}

double remainder = Modulus\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::ToVarNoCheck(remainder, scriptContext);

}

Var JavascriptMath::Modulus\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

// If both arguments are TaggedInt, then try to do integer division

// This case is not handled by the lowerer.

if (TaggedInt::IsPair(aLeft, aRight))

{

return TaggedInt::Modulus(aLeft, aRight, scriptContext);

}

double remainder = Modulus\_Helper(aLeft, aRight, scriptContext);

return JavascriptNumber::InPlaceNew(remainder, scriptContext, result);

}

Var JavascriptMath::FinishOddDivByPow2(int32 value, ScriptContext \*scriptContext)

{

return JavascriptNumber::New((double)(value + 0.5), scriptContext);

}

Var JavascriptMath::FinishOddDivByPow2\_InPlace(int32 value, ScriptContext \*scriptContext, JavascriptNumber\* result)

{

return JavascriptNumber::InPlaceNew((double)(value + 0.5), scriptContext, result);

}

Var JavascriptMath::MaxInAnArray(RecyclableObject \* function, CallInfo callInfo, ...)

{

PROBE\_STACK(function->GetScriptContext(), Js::Constants::MinStackDefault);

ARGUMENTS(args, callInfo);

Assert(args.Info.Count == 2);

Var thisArg = args[0];

Var arrayArg = args[1];

ScriptContext \* scriptContext = function->GetScriptContext();

TypeId typeId = JavascriptOperators::GetTypeId(arrayArg);

if (!JavascriptNativeArray::Is(typeId) && !(TypedArrayBase::Is(typeId) && typeId != TypeIds\_CharArray && typeId != TypeIds\_BoolArray))

{

if (JavascriptArray::IsVarArray(typeId) && JavascriptArray::FromVar(arrayArg)->GetLength() == 0)

{

return scriptContext->GetLibrary()->GetNegativeInfinite();

}

return JavascriptFunction::CalloutHelper<false>(function, thisArg, /\* overridingNewTarget = \*/nullptr, arrayArg, scriptContext);

}

if (JavascriptNativeArray::Is(typeId))

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(arrayArg);

#endif

JavascriptNativeArray \* argsArray = JavascriptNativeArray::FromVar(arrayArg);

uint len = argsArray->GetLength();

if (len == 0)

{

return scriptContext->GetLibrary()->GetNegativeInfinite();

}

if (((Js::SparseArraySegmentBase\*)argsArray->GetHead())->next != nullptr || !argsArray->HasNoMissingValues() ||

((Js::SparseArraySegmentBase\*)argsArray->GetHead())->length != len)

{

return JavascriptFunction::CalloutHelper<false>(function, thisArg, /\* overridingNewTarget = \*/nullptr, arrayArg, scriptContext);

}

return argsArray->FindMinOrMax(scriptContext, true /\*findMax\*/);

}

else

{

TypedArrayBase \* argsArray = TypedArrayBase::FromVar(arrayArg);

uint len = argsArray->GetLength();

if (len == 0)

{

return scriptContext->GetLibrary()->GetNegativeInfinite();

}

Var max = argsArray->FindMinOrMax(scriptContext, typeId, true /\*findMax\*/);

if (max == nullptr)

{

return JavascriptFunction::CalloutHelper<false>(function, thisArg, /\* overridingNewTarget = \*/nullptr, arrayArg, scriptContext);

}

return max;

}

}

Var JavascriptMath::MinInAnArray(RecyclableObject \* function, CallInfo callInfo, ...)

{

PROBE\_STACK(function->GetScriptContext(), Js::Constants::MinStackDefault);

ARGUMENTS(args, callInfo);

Assert(args.Info.Count == 2);

Var thisArg = args[0];

Var arrayArg = args[1];

ScriptContext \* scriptContext = function->GetScriptContext();

TypeId typeId = JavascriptOperators::GetTypeId(arrayArg);

if (!JavascriptNativeArray::Is(typeId) && !(TypedArrayBase::Is(typeId) && typeId != TypeIds\_CharArray && typeId != TypeIds\_BoolArray))

{

if (JavascriptArray::Is(typeId) && JavascriptArray::FromVar(arrayArg)->GetLength() == 0)

{

return scriptContext->GetLibrary()->GetPositiveInfinite();

}

return JavascriptFunction::CalloutHelper<false>(function, thisArg, /\* overridingNewTarget = \*/nullptr, arrayArg, scriptContext);

}

if (JavascriptNativeArray::Is(typeId))

{

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(arrayArg);

#endif

JavascriptNativeArray \* argsArray = JavascriptNativeArray::FromVar(arrayArg);

uint len = argsArray->GetLength();

if (len == 0)

{

return scriptContext->GetLibrary()->GetPositiveInfinite();

}

if (((Js::SparseArraySegmentBase\*)argsArray->GetHead())->next != nullptr || !argsArray->HasNoMissingValues() ||

((Js::SparseArraySegmentBase\*)argsArray->GetHead())->length != len)

{

return JavascriptFunction::CalloutHelper<false>(function, thisArg, /\* overridingNewTarget = \*/nullptr, arrayArg, scriptContext);

}

return argsArray->FindMinOrMax(scriptContext, false /\*findMax\*/);

}

else

{

TypedArrayBase \* argsArray = TypedArrayBase::FromVar(arrayArg);

uint len = argsArray->GetLength();

if (len == 0)

{

return scriptContext->GetLibrary()->GetPositiveInfinite();

}

Var min = argsArray->FindMinOrMax(scriptContext, typeId, false /\*findMax\*/);

if (min == nullptr)

{

return JavascriptFunction::CalloutHelper<false>(function, thisArg, /\* overridingNewTarget = \*/nullptr, arrayArg, scriptContext);

}

return min;

}

}

static const LARGE\_INTEGER multiplier = { 0xDEECE66D, 0x00000005 };

static const double kdbl2to27 = 134217728.0;

double JavascriptMath::Random(ScriptContext \*scriptContext)

{

uint64 seed = scriptContext->GetLibrary()->GetRandSeed();

ulong temp;

if (seed == 0)

{

LARGE\_INTEGER s0;

LARGE\_INTEGER s1;

QueryPerformanceCounter(&s0);

#if DBG\_DUMP

if (Configuration::Global.flags.Trace.IsEnabled(PRNGPhase))

{

Output::Print(L"[PRNG:%x] INIT %I64x\n", scriptContext, s0.QuadPart);

}

#endif

temp = s0.LowPart ^ multiplier.LowPart;

// Put bytes in order 0213.

temp = ((temp & 0xFF000000) >>24) | ((temp & 0x000000FF) <<24) | (temp & 0x00FFFF00);

// Interleave the bits : generator is 3120.

temp = ((temp & 0x0F000F00) >> 4) | ((temp & 0x00F000F0) << 4) | (temp & 0xF00FF00F);

temp = ((temp & 0x30303030) >> 2) | ((temp & 0x0C0C0C0C) << 2) | (temp & 0xC3C3C3C3);

temp = ((temp & 0x44444444) >> 1) | ((temp & 0x22222222) << 1) | (temp & 0x99999999);

s1.HighPart = temp >> 16;

s1.LowPart = (temp << 16) | ((s0.HighPart ^ s0.LowPart)& 0x0000FFFF);

seed = s1.QuadPart;

ThreadContext \*threadContext = scriptContext->GetThreadContext();

threadContext->GetEntropy().AddThreadCycleTime();

threadContext->GetEntropy().AddIoCounters();

seed ^= (threadContext->GetEntropy().GetRand() & 0x0000FFFFFFFFFFFFull);

}

Assert((seed >>32) < 0x00010000 ); // only up to 48 bits should be in the previous value

#if DBG\_DUMP

if (Configuration::Global.flags.Trace.IsEnabled(PRNGPhase))

{

Output::Print(L"[PRNG:%x] SEED %I64x\n", scriptContext, seed);

}

#endif

uint64 sn;

sn = (seed \* multiplier.QuadPart + 11) & 0x0000FFFFFFFFFFFFull; // apply linear recurrence and keep just 48 bits

double res = double((uint)(sn >> 21)); //use for the result the high 27 bits of the 45 bits above

// one more iteration and keep only 48 bits

seed = (sn \* multiplier.QuadPart + 11) & 0x0000FFFFFFFFFFFFull;

// Merge in the high 27 bits and normalize.

res += (double)((uint)(seed >> 21)) / kdbl2to27;

res /= kdbl2to27;

//update the seed

scriptContext->GetLibrary()->SetRandSeed(seed);

#if DBG\_DUMP

if (Configuration::Global.flags.Trace.IsEnabled(PRNGPhase))

{

Output::Print(L"[PRNG:%x] RAND %I64x\n", scriptContext, \*((uint64 \*)&res));

}

#endif

return res;

}

uint32 JavascriptMath::ToUInt32(double T1)

{

// Same as doing ToInt32 and reinterpret the bits as uint32

return (uint32)ToInt32Core(T1);

}

int32 JavascriptMath::ToInt32(double T1)

{

return JavascriptMath::ToInt32Core(T1);

}

int32 JavascriptMath::ToInt32\_Full(Var aValue, ScriptContext\* scriptContext)

{

AssertMsg(!TaggedInt::Is(aValue), "Should be detected");

// 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));

}

return JavascriptConversion::ToInt32\_Full(aValue, scriptContext);

}

#ifdef SSE2MATH

}

#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

{

#ifdef SSE2MATH

namespace SSE2

{

#endif

class JavascriptMath

{

public:

//

// Some frequently-used operations have three flavors optimized for different situations.

//

// 1. Op : Called from the interpreter loop only. Must handle all cases (may call Op\_Full)

// 2. Op\_Full : Called from generated code or from Op (case 1). TaggedInt cases already handled.

// 3. Op\_InPlace : Called from generated code only where result may be "in-place new'd"

//

static Var Increment(Var aRight,ScriptContext\* scriptContext);

static Var Increment\_Full(Var aRight,ScriptContext\* scriptContext);

static Var Increment\_InPlace(Var aRight,ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var Decrement(Var aRight,ScriptContext\* scriptContext);

static Var Decrement\_Full(Var aRight,ScriptContext\* scriptContext);

static Var Decrement\_InPlace(Var aRight,ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var Negate(Var aRight,ScriptContext\* scriptContext);

static Var Negate\_Full(Var aRight,ScriptContext\* scriptContext);

static Var Negate\_InPlace(Var aRight,ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var Not(Var aRight,ScriptContext\* scriptContext);

static Var Not\_Full(Var aRight,ScriptContext\* scriptContext);

static Var Not\_InPlace(Var aRight,ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var Add(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Add\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Add\_InPlace(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var AddLeftDead(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var MulAddLeft(Var mulLeft, Var mulRight, Var addLeft, ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var MulAddRight(Var mulLeft, Var mulRight, Var addRight, ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var MulSubLeft(Var mulLeft, Var mulRight, Var subLeft, ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var MulSubRight(Var mulLeft, Var mulRight, Var subRight, ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var Subtract(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Subtract\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Subtract\_InPlace(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var Multiply(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Multiply\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Multiply\_InPlace(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var Divide(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Divide\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Divide\_InPlace(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var Exponentiation(Var aLeft, Var aRight, ScriptContext\* scriptContext);

static Var Exponentiation\_Full(Var aLeft, Var aRight, ScriptContext\* scriptContext);

static Var Exponentiation\_InPlace(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var Modulus(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Modulus\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Modulus\_InPlace(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var And(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var And\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var And\_InPlace(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var Or(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Or\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Or\_InPlace(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var Xor(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Xor\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Xor\_InPlace(Var aLeft, Var aRight,ScriptContext\* scriptContext, JavascriptNumber \*result);

static Var ShiftLeft(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var ShiftLeft\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var ShiftRight(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var ShiftRight\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var ShiftRightU(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var ShiftRightU\_Full(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var FinishOddDivByPow2(int32 value, ScriptContext \*scriptContext);

static Var FinishOddDivByPow2\_InPlace(int32 value, ScriptContext \*scriptContext, JavascriptNumber\* result);

static Var MaxInAnArray(RecyclableObject \* function, CallInfo callInfo, ...);

static Var MinInAnArray(RecyclableObject \* function, CallInfo callInfo, ...);

static double Random(ScriptContext \*scriptContext);

static int32 ToInt32Core(double T1);

static uint32 ToUInt32(double value);

static int64 TryToInt64(double T1);

static int32 ToInt32(Var aValue, ScriptContext\* scriptContext);

static int32 ToInt32(double value);

static int32 ToInt32\_Full(Var aValue, ScriptContext\* scriptContext);

private:

static Var Add\_FullHelper(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result, bool leftIsDead);

static Var Add\_FullHelper\_Wrapper(Var aLeft, Var aRight, ScriptContext\* scriptContext, JavascriptNumber\* result, bool leftIsDead);

static double Add\_Helper(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static double Subtract\_Helper(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static double Multiply\_Helper(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static double Divide\_Helper(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static double Modulus\_Helper(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static Var Add\_DoubleHelper(double dblLeft, Var addRight, ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var Add\_DoubleHelper(Var addLeft, double dblRight, ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var Subtract\_DoubleHelper(double dblLeft, Var subRight, ScriptContext\* scriptContext, JavascriptNumber\* result);

static Var Subtract\_DoubleHelper(Var subLeft, double dblRight, ScriptContext\* scriptContext, JavascriptNumber\* result);

static int32 And\_Helper(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static double Decrement\_Helper(Var aLeft, ScriptContext\* scriptContext);

static double Increment\_Helper(Var aLeft, ScriptContext\* scriptContext);

static double Negate\_Helper(Var aRight,ScriptContext\* scriptContext);

static int32 Or\_Helper(Var aLeft, Var aRight,ScriptContext\* scriptContext);

static BOOL IsNanInfZero(double v);

static \_\_int64 ToInt32ES5OverflowHelper(double d);

};

#ifdef SSE2MATH

}

#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

{

#ifdef SSE2MATH

namespace SSE2

{

#endif

\_\_inline Var JavascriptMath::Increment(Var aRight, ScriptContext\* scriptContext)

{

return Increment\_Full(aRight, scriptContext);

}

\_\_inline Var JavascriptMath::Decrement(Var aRight, ScriptContext\* scriptContext)

{

return Decrement\_Full(aRight, scriptContext);

}

\_\_inline Var JavascriptMath::Negate(Var aRight, ScriptContext\* scriptContext)

{

return

(TaggedInt::Is(aRight) && aRight != TaggedInt::ToVarUnchecked(0) && aRight != TaggedInt::MinVal()) ?

TaggedInt::NegateUnchecked(aRight) :

Negate\_Full(aRight,scriptContext);

}

\_\_inline Var JavascriptMath::Not(Var aRight, ScriptContext\* scriptContext)

{

return

TaggedInt::Is(aRight) ?

TaggedInt::Not(aRight,scriptContext) :

Not\_Full(aRight,scriptContext);

}

\_\_inline Var JavascriptMath::Or(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return

TaggedInt::IsPair(aLeft,aRight) ?

TaggedInt::Or(aLeft,aRight) :

Or\_Full(aLeft,aRight,scriptContext);

}

\_\_inline Var JavascriptMath::And(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

#if FLOATVAR

return

TaggedInt::IsPair(aLeft,aRight) ?

TaggedInt::And(aLeft,aRight) :

And\_Full(aLeft,aRight,scriptContext);

#else

Var varSpeculative = TaggedInt::Speculative\_And(aLeft, aRight);

if (TaggedInt::Is(varSpeculative))

{

return varSpeculative;

}

return And\_Full(aLeft, aRight, scriptContext);

#endif

}

\_\_inline Var JavascriptMath::ShiftLeft(Var aLeft,Var aRight,ScriptContext\* scriptContext)

{

return

TaggedInt::IsPair(aLeft, aRight) ?

TaggedInt::ShiftLeft(aLeft, aRight,scriptContext) :

ShiftLeft\_Full(aLeft, aRight,scriptContext);

}

\_\_inline Var JavascriptMath::ShiftRight(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return

TaggedInt::IsPair(aLeft, aRight) ?

TaggedInt::ShiftRight(aLeft, aRight) :

ShiftRight\_Full(aLeft, aRight,scriptContext);

}

\_\_inline Var JavascriptMath::ShiftRightU(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return

TaggedInt::IsPair(aLeft, aRight) ?

TaggedInt::ShiftRightU(aLeft, aRight, scriptContext) :

ShiftRightU\_Full(aLeft, aRight,scriptContext);

}

\_\_inline Var JavascriptMath::Xor(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return

TaggedInt::IsPair(aLeft, aRight) ?

TaggedInt::Xor(aLeft, aRight) :

Xor\_Full(aLeft, aRight,scriptContext);

}

\_\_inline double JavascriptMath::Decrement\_Helper(Var aRight, ScriptContext\* scriptContext)

{

#if defined(DBG)

if (TaggedInt::Is(aRight))

{

// The only reason to be here is if TaggedInt increment underflowed

AssertMsg(aRight == TaggedInt::MinVal(), "TaggedInt decrement should be handled in generated code.");

}

#endif

double value = JavascriptConversion::ToNumber(aRight, scriptContext);

return --value;

}

\_\_inline double JavascriptMath::Increment\_Helper(Var aRight, ScriptContext\* scriptContext)

{

#if defined(DBG)

if (TaggedInt::Is(aRight))

{

// The only reason to be here is if TaggedInt increment overflowed

AssertMsg(aRight == TaggedInt::MaxVal(), "TaggedInt increment should be handled in generated code.");

}

#endif

double value = JavascriptConversion::ToNumber(aRight, scriptContext);

return ++value;

}

\_\_inline double JavascriptMath::Negate\_Helper(Var aRight,ScriptContext\* scriptContext)

{

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

double value = JavascriptConversion::ToNumber(aRight, scriptContext);

return -value;

}

\_\_inline int32 JavascriptMath::And\_Helper(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

#if \_M\_IX86

AssertMsg(!TaggedInt::IsPair(aLeft, aRight), "TaggedInt bitwise and should have been handled already");

#endif

int32 nLeft = JavascriptConversion::ToInt32(aLeft, scriptContext);

int32 nRight = JavascriptConversion::ToInt32(aRight, scriptContext);

return nLeft & nRight;

}

\_\_inline int32 JavascriptMath::Or\_Helper(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

#if \_M\_IX86

AssertMsg(!TaggedInt::IsPair(aLeft, aRight), "TaggedInt bitwise or should have been handled already");

#endif

int32 nLeft = JavascriptConversion::ToInt32(aLeft, scriptContext);

int32 nRight = JavascriptConversion::ToInt32(aRight, scriptContext);

return nLeft | nRight;

}

\_\_inline double JavascriptMath::Add\_Helper(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

AssertMsg( !JavascriptString::Is(aLeft), "Strings should have been handled already" );

AssertMsg( !JavascriptString::Is(aRight), "Strings should have been handled already" );

double dblLeft = JavascriptConversion::ToNumber(aLeft, scriptContext);

double dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

return dblLeft + dblRight;

}

\_\_inline Var JavascriptMath::Add(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return

TaggedInt::IsPair(aLeft,aRight) ?

TaggedInt::Add(aLeft, aRight, scriptContext) :

Add\_Full(aLeft, aRight, scriptContext);

}

\_\_inline Var JavascriptMath::Subtract(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return

TaggedInt::IsPair(aLeft,aRight) ?

TaggedInt::Subtract(aLeft, aRight, scriptContext) :

Subtract\_Full(aLeft, aRight, scriptContext);

}

\_\_inline double JavascriptMath::Subtract\_Helper(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

// The IEEE 754 floating point spec ensures that NaNs are preserved in all operations

double dblLeft = JavascriptConversion::ToNumber(aLeft, scriptContext);

double dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

return dblLeft - dblRight;

}

\_\_inline Var JavascriptMath::Multiply(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return

TaggedInt::IsPair(aLeft,aRight) ?

TaggedInt::Multiply(aLeft, aRight, scriptContext) :

Multiply\_Full(aLeft, aRight, scriptContext);

}

\_\_inline Var JavascriptMath::Exponentiation(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return Exponentiation\_Full(aLeft, aRight, scriptContext);

}

\_\_inline double JavascriptMath::Multiply\_Helper(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

// The IEEE 754 floating point spec ensures that NaNs are preserved in all operations

return JavascriptConversion::ToNumber(aLeft, scriptContext) \* JavascriptConversion::ToNumber(aRight, scriptContext);

}

\_\_inline Var JavascriptMath::Divide(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

// The TaggedInt,TaggedInt case is handled within Divide\_Full

return Divide\_Full(aLeft, aRight,scriptContext);

}

\_\_inline double JavascriptMath::Divide\_Helper(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

Assert(aLeft != nullptr);

Assert(aRight != nullptr);

Assert(scriptContext != nullptr);

#if !defined(\_M\_ARM32\_OR\_ARM64)

AssertMsg(!TaggedInt::IsPair(aLeft, aRight), "Integer division should have been handled already");

#endif

// The IEEE 754 floating point spec ensures that NaNs are preserved in all operations

return JavascriptConversion::ToNumber(aLeft, scriptContext) / JavascriptConversion::ToNumber(aRight, scriptContext);

}

\_\_inline Var JavascriptMath::Modulus(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

return Modulus\_Full(aLeft, aRight, scriptContext);

}

\_\_inline double JavascriptMath::Modulus\_Helper(Var aLeft, Var aRight, ScriptContext\* scriptContext)

{

double dblLeft = JavascriptConversion::ToNumber(aLeft, scriptContext);

double dblRight = JavascriptConversion::ToNumber(aRight, scriptContext);

return NumberUtilities::Modulus(dblLeft, dblRight);

}

#if defined(\_M\_ARM32\_OR\_ARM64)

\_\_inline int32 JavascriptMath::ToInt32Core(double T1)

{

// Try the int32 conversion first and only do the more expensive (& closer to spec)

// i64 conversion if it fails.

\_\_int32 i32 = (\_\_int32)T1;

if ((i32 != 0x80000000) && (i32 != 0x7fffffff))

return i32; //No overflow so just return i32

int64 T4\_64 = TryToInt64(T1);

if (!NumberUtilities::IsValidTryToInt64(T4\_64)) // overflow

{

T4\_64 = ToInt32ES5OverflowHelper(T1);

}

return static\_cast<int32>(T4\_64);

}

#else

\_\_inline int32 JavascriptMath::ToInt32Core(double T1)

{

// ES5 Spec for ToUInt32

//

// T3 = sign(T1) \* floor(abs(T1))

// T4 = T3 % 2^32

//

// Casting gives equivalent result, except when T1 > INT64\_MAX, or T1 < INT64\_MIN (or NaN Inf Zero),

// in which case we'll use slow path.

// Try casting to int32 first. Results in 0x80000000 if it overflows.

int32 T4\_32 = static\_cast<int32>(T1);

if (T4\_32 != 0x80000000)

{

return T4\_32;

}

int64 T4\_64 = TryToInt64(T1);

if (T4\_64 == 0x8000000000000000) // overflow && ES5

{

T4\_64 = ToInt32ES5OverflowHelper(T1);

}

return static\_cast<int32>(T4\_64);

}

#endif

// Implements platform-agnostic part of handling overflow when converting Number to int32, ES5 version.

\_\_inline \_\_int64 JavascriptMath::ToInt32ES5OverflowHelper(double d)

{

if (IsNanInfZero(d)) // ShortCut NaN Inf Zero

{

return 0;

}

const double k\_2to32 = 4294967296.0;

double floored;

#pragma prefast(suppress:6031, "We don't care about the fraction part")

modf(d, &floored); // take out the floating point part.

double m2to32 = fmod(floored, k\_2to32); // divide modulo 2^32.

\_\_int64 result = TryToInt64(m2to32);

AssertMsg(NumberUtilities::IsValidTryToInt64(result), "No more overflow expected");

return result;

}

\_\_inline BOOL JavascriptMath::IsNanInfZero(double v)

{

return JavascriptNumber::IsNan(v) || JavascriptNumber::IsZero(v) || JavascriptNumber::IsPosInf(v) || JavascriptNumber::IsNegInf(v);

}

\_\_inline int64 JavascriptMath::TryToInt64(double T1)

{

return Js::NumberUtilities::TryToInt64(T1);

}

\_\_inline int32 JavascriptMath::ToInt32(Var aValue, ScriptContext\* scriptContext)

{

return

TaggedInt::Is(aValue) ?

TaggedInt::ToInt32(aValue) :

ToInt32\_Full(aValue, scriptContext);

}

#ifdef SSE2MATH

}

#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 "RuntimeMathPch.h"

#define SSE2MATH

#include "JavascriptMath.cpp"

#undef SSE2MATH

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

// 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 SSE2MATH

#include "JavascriptMath.h"

#undef SSE2MATH

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

// 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 SSE2MATH

#include "JavascriptMath.inl"

#undef SSE2MATH

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

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

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

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

#include "RuntimeMathPch.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 "Runtime.h"

#include "Math\JavascriptSSE2MathOperators.h"

#include "Math\JavascriptSSE2MathOperators.inl"