# Intermediate code

## Registers

Registers can only hold values of the types specified in **BaseType**. Single static assignment (SSA) form is always used for registers; no register may be assigned to twice.

## Stack frames

To store larger values or values which must be in memory, stack frame storage can be used. To avoid specifying exact sizes, storage is allocated in slots. The LdF operation can be used to retrieve the address of a slot in the stack frame. Similarly the LdFA operation can be used to retrieve the address of the arguments to the current function.

## Blocks

Blocks are sequences of operations. Each block except for the last block in a function ends with a jump or branch operation. There always exists a “current block” into which operations are written while converting expressions to intermediate code. This block can change depending on sequence points and control flow statements.

Since SSA form is used for registers, each block contains a sequence of ɸ (phi) operations, stored separately from normal operations.

## Intermediate code format

|  |  |
| --- | --- |
| Operation | Meaning |
| Mov r1 -> r2 | r1 -> r2 |
| CvtC r1 -> r2 | convert r1 as char -> r2 |
| CvtUC r1 -> r2 | convert r1 as unsigned char -> r2 |
| CvtS r1 -> r2 | convert r1 as short -> r2 |
| CvtUS r1 -> r2 | convert r1 as unsigned short -> r2 |
| CvtI r1 -> r2 | convert r1 as int -> r2 |
| CvtUI r1 -> r2 | convert r1 as unsigned int -> r2 |
| CvtL r1 -> r2 | convert r1 as long -> r2 |
| CvtUL r1 -> r2 | convert r1 as unsigned long -> r2 |
| CvtLL r1 -> r2 | convert r1 as long long -> r2 |
| CvtULL r1 -> r2 | convert r1 as unsigned long long -> r2 |
| CvtF r1 -> r2 | convert r1 as float -> r2 |
| CvtD r1 -> r2 | convert r1 as double -> r2 |
| CvtP r1 -> r2 | convert r1 as void \* -> r2 |
| Add r1, r2 -> r3 | r1 + r2 -> r3 |
| AddI r1, c2 -> r3 | r1 + c2 -> r3 |
| Sub r1, r2 -> r3 | r1 - r2 -> r3 |
| SubI r1, c2 -> r3 | r1 - c2 -> r3 |
| SubIR r1, c2 -> r3 | c2 – r1 -> r3 |
| Mul r1, r2 -> r3 | r1 \* r2 -> r3 |
| MulI r1, c2 -> r3 | r1 \* c2 -> r3 |
| MulU r1, r2 -> r3 | r1 \* r2 -> r3 (unsigned multiply) |
| MulUI r1, c2 -> r3 | r1 \* c2 -> r3 (unsigned multiply) |
| Div r1, r2 -> r3 | r1 / r2 -> r3 |
| DivI r1, c2 -> r3 | r1 / c2 -> r3 |
| DivIR r1, c2 -> r3 | c2 / r1 -> r3 |
| DivU r1, r2 -> r3 | r1 / r2 -> r3 (unsigned divide) |
| DivUI r1, c2 -> r3 | r1 / c2 -> r3 (unsigned divide) |
| DivUIR r1, c2 -> r3 | c2 / r1 -> r3 (unsigned divide) |
| Mod r1, r2 -> r3 | r1 % r2 -> r3 |
| ModI r1, c2 -> r3 | r1 % c2 -> r3 |
| ModU r1, r2 -> r3 | r1 % r2 -> r3 (unsigned modulo) |
| ModUI r1, c2 -> r3 | r1 % c2 -> r3 (unsigned modulo) |
| Not r1 -> r2 | ~r1 -> r2 |
| And r1, r2 -> r3 | r1 & r2 -> r3 |
| AndI r1, c2 -> r3 | r1 & c2 -> r3 |
| Or r1, r2 -> r3 | r1 | r2 -> r3 |
| OrI r1, c2 -> r3 | r1 | c2 -> r3 |
| Xor r1, r2 -> r3 | r1 ^ r2 -> r3 |
| XorI r1, c2 -> r3 | r1 ^ c2 -> r3 |
| Shl r1, r2 -> r3 | r1 << r2 -> r3 |
| ShlU r1, r2 -> r3 | r1 << r2 -> r3 |
| Shr r1, r2 -> r3 | r1 >> r2 -> r3 |
| ShrU r1, r2 -> r3 | r1 >> r2 -> r3 |
| Ld r1 -> r2 | Deref(r1) -> r2 |
| LdI c1 -> r2 | c1 -> r2 |
| LdF c1 -> r2 | address of variable slot c1 in stack frame -> r2 |
| LdFA -> r1 | address of arguments in stack frame -> r1 |
| LdN -> r1 | load uninitialized value into r1 |
| St r1, r2 | r1 -> Deref(r2) |
| StI c1, r2 | c1 -> Deref(r2) |
| Srt r1 | r1 -> return value of function |
| CpyI r1, r2, c3 | copy c3 bytes from address r1 to address r2 |
| Jmp b1 | b1 -> next block (placeholder if b1 is NULL) |
| BrE/BrL/BrLE/BrB/BrBE r1, r2, b3, b4 | if relation between r1 and r2 is true then  b3 -> next block  else  b4 -> next block |
| BrEI/BrLI/BrLEI/BrBI/BrBEI r1, c2, b3, b4 | if relation between r1 and c2 is true then  b3 -> next block  else  b4 -> next block |
| ChE/ChL/ChLE/ChB/ChBE r1, r2, c3, c4 -> r5 | if relation between r1 and r2 is true then  c3 -> r5  else  c4 -> r5 |
| ChEI/ChLI/ChLEI/ChBI/ChBEI r1, c2, c3, c4 -> r5 | if relation between r1 and c2 is true then  c3 -> r5  else  c4 -> r5 |
| Call r1, r2, r3, ... | call function r1 with arguments r2, r3, ... |
| CallV r1, r2, r3, ... -> rx | call function r1 with arguments r2, r3, ... and store return value in rx |
| CallI c1, r2, r3, ... | call function c1 with arguments r2, r3, ... |
| CallVI c1, r2, r3, ... -> rx | call function c1 with arguments r2, r3, ... and store return value in rx |
| Phi r1, r2, r3, ... -> rx | choose between r1, r2, r3, ... and store in rx (stored SEPARATELY from normal operations) |
| End | last instruction of every function |

### Operation modes

Each operation may have a number of possible modes. These modes correspond to the types in **BaseType**. For example, Add<Char> r1, r2 -> r3 assumes that r1, r2 and r3 are Char registers. Ld<Float> r1 -> r2 assumes that r1 is an IntPtr register and r2 is a Float register. Some operations, such as Jmp, BrX, Call and CallI do not have modes. ChXXX is an exception – the output register is always Int, and the mode specifies the type of the inputs.

### Deref(v1) -> r2

The output r2 is set to the contents at IntPtr v1. The operation mode determines the size of the contents at v1.

# Conversion from expressions to intermediate code

In the text below:

* rx, ry, rz mean newly allocated registers.
* “read the value of *x*” or “read from *x*” means to use a register to access *x*, or emit an Ld instruction if x is in indirect storage. *x* may also need to be converted to the type of any operations in which it is used. If *x* is a bit range, emit appropriate shift and mask instructions as well.

## Casts for integer, floating-point or pointer types

From type:

* Char: CvtC
* Unsigned Char: CvtUC
* Short: CvtS
* Unsigned Short: CvtUS
* Int: CvtI
* Unsigned Int: CvtUI
* Long: CvtL
* Unsigned Long: CvtUL
* LongLong: CvtLL
* Unsigned LongLong: CvtULL
* Float: CvtF
* Double: CvtD
* IntPtr: CvtP

To type:

* (Unsigned) Char: CvtXXX<Char>
* (Unsigned) Short: CvtXXX<Short>
* (Unsigned) Int: CvtXXX<Int>
* (Unsigned) Long: CvtXXX<Long>
* (Unsigned) LongLong: CvtXXX<LongLong>
* Float: CvtXXX<Float>
* Double: CvtXXX<Double>
* IntPtr: CvtXXX<IntPtr>

## Implicit casts

### Usual arithmetic conversions

The usual arithmetic conversions attempt to reach a common type for two given expressions.

* If either type is Double, both expressions are converted to Double.
* Otherwise, if either type is Float, both expressions are converted to Float.
* Otherwise, if both expressions have the same type, no conversion is needed.
* Otherwise, if both expressions have signed integer types or both expressions have unsigned integer types, the expression with less precision is converted to the type with greater precision.
* Otherwise, if the unsigned integer type has greater precision than the signed integer type, the signed expression is converted to the unsigned type.
* Otherwise, if the signed integer type can represent all values of the unsigned integer type, the unsigned expression is converted to the signed type.
* Otherwise, both expressions are converted to the unsigned version of the signed type.

### Arrays

Except when it is the operand of the sizeof operator or the address-of (&) operator, an expression that has type “array of *x*” must be converted to a pointer to *x*.

## Sequence points

All pending side effects must be executed at sequence points. These include:

* Function calls, after all arguments have been evaluated.
* The end of the first expression in **AndAndExpression**, **OrOrExpression**, **ConditionalExpression** and **CommaExpression**.
* The end of an initializer.
* The end of a statement.

## PrimaryExpression

### ID {Id}

Find the ordinary symbol ID in the current **Scope**:

* If it is **TypedefSymbolKind**, raise an error.
* If it is **VariableSymbolKind**, emit LdF slot -> rx, where slot is the slot number of the variable (see VariableSlots in **FunctionScope**).
* If it is **ObjectSymbolKind**, emit LdI<IntPtr> value -> rx, where value is an immediate value referencing the linker object for the function or variable. If the object is a function, the resulting type is the function type with one level of indirection (one pointer) and **FunctionDesignator** in RegisterStorageKind is set to True.
* If it is **ConstantSymbolKind**, emit LdI<Int> value -> rx, where value is the value of the enumeration constant.

Return the new register.

### LITERAL\_NUMBER {Number}

Refer to the tables below.

|  |  |  |
| --- | --- | --- |
| Suffix | Decimal constant | Octal or hexadecimal constant |
| none | int  long int  long long int | int  unsigned int  long int  unsigned long int  long long int  unsigned long long int |
| u or U | unsigned int  unsigned long int  unsigned long long int | unsigned int  unsigned long int  unsigned long long int |
| l or L | long int  long long int | long int  unsigned long int  long long int  unsigned long long int |
| Both u or U and l or L | unsigned long int  unsigned long long int | unsigned long int  unsigned long long int |
| ll or LL | long long int | long long int  unsigned long long int |
| Both u or U and ll or LL | unsigned long long int | unsigned long long int |

If a floating-point literal is present, emit LdI<Float> value -> rx if the suffix is f or F. Emit LdI<Double> value -> rx if the suffix is l or L, or if there is no suffix. Return the new register.

### LITERAL\_CHAR {Char}

If the prefix is L, emit LdI<Short> value -> rx. Otherwise, emit LdI<Char> value -> rx. Return the new register.

### LITERAL\_STRING {String}

Create a linker object containing the string and emit LdI<IntPtr> value -> rx where value is a reference to the linker object. Return the new register, with type short \* if the L prefix is present, or with type char \* if no prefix is present.

### “(“ Expression “)” {Group}

Return the value of the expression.

### PrimaryExpression “++” {Increment}

If the type of the expression is not integer or floating-point, raise an error. Otherwise, add an increment operation for the next sequence point.

### PrimaryExpression “--” {Decrement}

As above, but schedule a decrement operation for the next sequence point.

### PrimaryExpression “(“ “)” {Call}

If the type of the expression is not a pointer to a function, raise an error. Otherwise:

* If the function has a return value, emit CallV r1 -> rx where r1 is read from the expression. Return the new register.
* If the function has no return value, emit Call r1. Return **NoStorageKind**.

### PrimaryExpression “(“ ParameterList “)” {Call2}

As above, but with parameters. For non-register arguments:

* **IndirectStorageKind** arguments must be copied from memory into a sequence of registers. Each register should be an IntPtr. If the value’s size is not a multiple of the size of an IntPtr, the last register must be zero-extended to an IntPtr.

### PrimaryExpression “[“ Expression “]” {Array}

Emit code equivalent to \*(expression1 + expression2), but only if one of the expressions has a pointer type and the other has an integer type.

### PrimaryExpression “.” ID {Member}

If the type of the expression is not a struct type, raise an error. Otherwise:

* If the first expression is **IndirectStorageKind**, emit AddI base, offset -> rx where base is the address of the structure and offset is the byte offset of the field in the structure and return an **IndirectStorageKind** with rx as the Address. If the field spans a number of bits, set BitCount and BitOffset in **IndirectStorageKind**.

### PrimaryExpression “->” ID {IndirectMember}

If the type of the expression is not a pointer to a struct type, raise an error. Otherwise, emit AddI base, offset -> rx where base is read from the expression and offset is the byte offset of the field in the structure. Return an **IndirectStorageKind** with rx as the Address. If the field spans a number of bits, set BitCount and BitOffset in **IndirectStorageKind**.

## UnaryExpression

### “++” CastExpression {Increment}

Emit code equivalent to expression += 1.

### “--“ CastExpression {Decrement}

Emit code equivalent to expression -= 1.

### “+” CastExpression {Positive}

If the type of the expression is not integer or floating-point, raise an error. Otherwise, return the value of the expression.

### “-“ CastExpression {Negative}

If the type of the expression is not integer or floating-point, raise an error. Otherwise, emit SubIR 0, value -> rx where value is read from the expression. Return the new register.

### “!” CastExpression {LogicalNot}

If the type of the expression is not integer, floating-point, or a pointer type, raise an error. Otherwise, emit ChEI value, 0, 1, 0 -> rx where value is read from the expression. Return the new register.

### “~” CastExpression {Not}

If the type of the expression is not integer, raise an error. Otherwise, emit Not value -> rx where value is read from the expression. Return the new register.

### “\*” CastExpression {Dereference}

If the type of the expression is not a pointer type, raise an error. If the type is a pointer to a function, return the value of the expression with FunctionDesignator set to True. Otherwise, return an **IndirectStorageKind** with base as the Address, where base is read from the expression.

### “&” CastExpression {AddressOf}

If the expression is not an l-value and does not have FunctionDesignator set to True, raise an error. If the expression has FunctionDesignator set to True, return the value of the expression with FunctionDesignator set to False. Otherwise, return a **RegisterStorageKind** with the Address in **IndirectStorageKind** as the Register.

### SIZEOF UnaryExpression {SizeOfValue}

If the expression has FunctionDesignator set to True, raise an error. Emit LdI size -> rx, where size is the size of the type of the expression. Return the new register.

### SIZEOF “(“ DeclarationSpecifierList “)” {SizeOfType}

Emit LdI size -> rx, where size is the size of the type. Return the new register.

### SIZEOF “(“ DeclarationSpecifierList AbstractDeclarator “)” {SizeOfType2}

As above.

## CastExpression

### “(“ DeclarationSpecifierList “)” UnaryExpression {Cast}

See **Casts**.

### “(“ DeclarationSpecifierList AbstractDeclarator “)” UnaryExpression {Cast2}

As above.

## MultiplyExpression

### MultiplyExpression “\*” CastExpression {Multiply}

If the type of both expressions is not integer or floating-point, raise an error. Emit Mul value1, value2 -> rx where value1 is read from the first expression and value2 is read from the second expression. Return the new register.

### MultiplyExpression “/” CastExpression {Divide}

As above, but emit Div value1, value2 -> rx instead.

### MultiplyExpression “%” CastExpression {Modulo}

As above, but emit Mod value1, value2 -> rx instead.

## AddExpression

### AddExpression “+” MultiplyExpression {Add}

* If both expressions have integer or floating-point types, emit Add value1, value2 -> rx where value1 is read from the first expression and value2 is read from the second expression. Return the new register.
* If one expression has a pointer type and the other has an integer type, emit Mul offset, size -> rx followed by Add base, rx -> ry where offset is read from the integer expression, size is the size of the type pointed to, and base is read from the pointer expression. Return ry.
* Otherwise, raise an error.

### AddExpression “-“ MultiplyExpression {Subtract}

* If both expressions have integer or floating-point types, emit Sub value1, value2 -> rx where value1 is read from the first expression and value2 is read from the second expression. Return the new register.
* If the first expression has a pointer type and the second expression has an integer type, emit Mul offset, size -> rx followed by Sub base, rx -> ry where offset is read from the integer expression, size is the size of the type pointed to, and base is read from the pointer expression. Return ry.
* If the both expressions have the same type and the type is a pointer type, emit Sub value1, value2 -> rx followed by Div rx, size -> ry where value1 is read from the first expression, value2 is read from the second expression and size is the size of the type pointed to. Return ry.
* Otherwise, raise an error.

## ShiftExpression

### ShiftExpression “<<” AddExpression {LeftShift}

If the type of both expressions is not integer, raise an error. Emit Shl value1, value2 -> rx where value1 is read from the first expression and value2 is read from the second expression. Return the new register.

### ShiftExpression “>>” AddExpression {RightShift}

As above, but emit Shr value1, value2 -> rx instead.

## RelationalExpression

### RelationalExpression “<” ShiftExpression {Less}

If both expressions do not have integer, floating-point or compatible pointer types, raise an error. If the common type of the two expressions is:

* An unsigned integer or pointer type, emit ChB value1, value2, 1, 0 -> rx where value1 is read from the first expression and value2 is read from the second expression, and return rx.
* A signed integer or floating-point type, emit ChL value1, value2, 1, 0 -> rx and return rx.

The type of the returned value must be Int.

Note: Here the definition of “compatible pointer type” does *not* include a pointer to void being compatible with any other kind of pointer.

### RelationalExpression “<=” ShiftExpression {LessEqual}

As above, but if the common type of the two expressions is:

* An unsigned integer or pointer type, emit ChBE value1, value2, 1, 0 -> rx and return rx.
* A signed integer or floating-point type, emit ChLE value1, value2, 1, 0 -> rx and return rx.

### RelationalExpression “>” ShiftExpression {Greater}

As above, but if the common type of the two expressions is:

* An unsigned integer or pointer type, emit ChBE value1, value2, 0, 1 -> rx and return rx.
* A signed integer or floating-point type, emit ChLE value1, value2, 0, 1 -> rx and return rx.

### RelationalExpression “>=” ShiftExpression {GreaterEqual}

As above, but if the common type of the two expressions is:

* An unsigned integer or pointer type, emit ChB value1, value2, 0, 1 -> rx and return rx.
* A signed integer or floating-point type, emit ChL value1, value2, 0, 1 -> rx and return rx.

## EqualityExpression

### EqualityExpression “==” RelationalExpression {Equal}

* If both expressions have integer, floating-point, or compatible pointer types, emit ChE value1, value2, 1, 0 -> rx where value1 is read from the first expression and value2 is read from the second expression. Return rx.
* If one expression has a pointer type and the other is known to be a null pointer constant, emit ChEI pointer, 0, 1, 0 -> rx where pointer is read from the pointer expression.

The type of the returned value must be Int.

Note: Unlike with **RelationalExpression**, here the definition of “compatible pointer type” *does* include a pointer to void being compatible with any other kind of pointer.

### EqualityExpression “!=” RelationalExpression {NotEqual}

As above, but emit ChE value1, value2, 0, 1 -> rx or ChEI pointer, 0, 0, 1 -> rx instead.

## AndExpression

### AndExpression “&” EqualityExpression {And}

If the type of both expressions is not integer, raise an error. Emit And value1, value2 -> rx where value1 is read from the first expression and value2 is read from the second expression. Return the new register.

## XorExpression

### XorExpression “^” AndExpression {Xor}

As with **AndExpression**, but emit Xor value1, value2 -> rx instead.

## OrExpression

### OrExpression “|” XorExpression {Or}

As with **AndExpression**, but emit Or value1, value2 -> rx instead.

## AndAndExpression

### AndAndExpression “&&” OrExpression {AndAnd}

If the type of both expressions is not integer, floating-point, or a pointer type, raise an error. Otherwise, emit code equivalent to

int temp = 0;

if (value1 != 0)

temp = value2 != 0;

where value1 is read from the first expression and value2 is read from the second expression. Return temp. See **IfStatement**.

## OrOrExpression

### OrOrExpression “||” AndAndExpression {OrOr}

If the type of both expressions is not integer, floating-point, or a pointer type, raise an error. Otherwise, emit code equivalent to

int temp = 1;

if (value1 == 0)

temp = value2 != 0;

where value1 is read from the first expression and value2 is read from the second expression. Return temp. See **IfStatement**.

## ConditionalExpression

### OrOrExpression “?” Expression “:” ConditionalExpression {Conditional}

If the type of the first expression is not integer, floating-point, or a pointer type, raise an error. For the other two expressions, if none the following hold, raise an error:

* Both expressions have the same integer, floating-point, or structure type.
* Both expressions are **NoStorageKind** (i.e. both expressions have void type).
* Both expressions have compatible pointer types.
* One expression has a pointer type and the other type is a pointer to void.
* One expression has a pointer type and the other expression is a null pointer constant.

Emit code equivalent to

TYPE temp;

if (value1 != 0)

temp = value2;

else

temp = value3;

where TYPE is the common type derived from the first and second expressions, value1 is read from the first expression, value2 is read from the second expression, and value3 is read from the third expression. Return temp. See **IfStatement**.

## AssignExpression

### ConditionalExpression “=” AssignExpression {Assign}

If none of the following hold, raise an error:

* Both expressions have integer, floating-point or structure types.
* Both expressions have compatible pointer types.
* One expression has a pointer type and the other type is a pointer to void.
* One expression has a pointer type and the other expression is a null pointer constant.

If the storage kind of the first expression is:

* **IndirectStorageKind** and the type of the second expression is **RegisterStorageKind**, emit St right, left where right is read from the second expression and left is the Address of the first expression. Return the value read from the second expression.
* **IndirectStorageKind** and the type of the second expression is **IndirectStorageKind**, emit CpyI right, left, size where right is the Address of the second expression, left is the Address of the first expression, and size is the size of the first expression’s type. Return the value of the second expression.

If the first expression refers to a bit range, emit appropriate bitwise instructions to calculate the resulting value.

### ConditionalExpression “+=” AssignExpression {Add}

Emit code equivalent to

left1 = left2 + right

where left1 is the l-value of the first expression, left2 is read from the first expression and right is read from the second expression. See **AddExpression** for type restrictions.

### ConditionalExpression "-=" AssignExpression {Subtract}

As above, but emit code equivalent to

left1 = left2 - right

### ConditionalExpression "\*=" AssignExpression {Multiply}

As with “Add”, but emit code equivalent to

left1 = left2 \* right

### ConditionalExpression "/=" AssignExpression {Divide}

As with “Add”, but emit code equivalent to

left1 = left2 / right

### ConditionalExpression "%=" AssignExpression {Modulo}

As with “Add”, but emit code equivalent to

left1 = left2 % right

### ConditionalExpression "&=" AssignExpression {And}

As with “Add”, but emit code equivalent to

left1 = left2 & right

### ConditionalExpression "|=" AssignExpression {Or}

As with “Add”, but emit code equivalent to

left1 = left2 | right

### ConditionalExpression "^=" AssignExpression {Xor}

As with “Add”, but emit code equivalent to

left1 = left2 ^ right

### ConditionalExpression "<<=" AssignExpression {LeftShift}

As with “Add”, but emit code equivalent to

left1 = left2 << right

### ConditionalExpression ">>=" AssignExpression {RightShift}

As with “Add”, but emit code equivalent to

left1 = left2 >> right

## CommaExpression

### CommaExpression "," AssignExpression {Comma}

Emit code to evaluate the first expression, followed by code to evaluate the second expression. Return the value of the second expression.

# Conversion from statements to intermediate code

## LabeledStatement

### ID “:” Statement {Id}

If the label is already defined, raise an error. Emit Jmp newBlk where newBlk becomes the new current block. Add the block to Labels in the current **FunctionScope**. If there are PendingGotos for this label, for each pending block adjust the Jmp operation in the block.

### CASE ConditionalExpression “:” Statement {Case}

Find the nearest **ControlScope** with the Switch keyword. If there is none, or the case is already defined, raise an error. Emit Jmp newBlk where newBlk becomes the new current block. Add the block to Cases in the **ControlScope**.

### DEFAULT “:” Statement {Default}

As above, but instead of adding the block to Cases, set DefaultCase in the **ControlScope**.

## IfStatement

### IF “(“ Expression “)” Statement {If}

If the type of the expression is not integer, floating-point or a pointer type, raise an error. Emit BrEI value, 0, nextBlk, nonZeroBlk where:

B0

* nextBlk is a new block that becomes the new current block after the **IfStatement**.
* value is read from the expression.
* nonZeroBlk is a new block in which the statement is executed. Emit Jmp nextBlk as the last instruction in the block.

### IF “(“ Expression “)” Statement ELSE Statement {IfElse}

If the type of the expression is not integer, floating-point or a pointer type, raise an error. Emit BrEI value, 0, zeroBlk, nonZeroBlk where:

B1

B0

* nextBlk is a new block that becomes the new current block after the **IfStatement**.
* value is read from the expression.
* zeroBlk is a new block in which the second statement is executed. Emit Jmp nextBlk as the last instruction in the block.
* nonZeroBlk is a new block in which the first statement is executed. Emit Jmp nextBlk as the last instruction in the block.

## WhileStatement: WHILE “(“ Expression “)” Statement

B1

B0

If the type of the expression is not integer, floating-point or a pointer type, raise an error. Emit Jmp testBlk and create a new **ControlScope** where:

* The **ControlScope** has BreakTarget = nextBlk and ContinueTarget = testBlk.
* nextBlk is a new block that becomes the new current block after the **WhileStatement**.
* testBlk is a new block created by emitting BrEI value, 0, nextBlk, bodyBlk.
* value is read from the expression.
* bodyBlk is a new block in which the statement is executed. Emit Jmp testBlk as the last instruction in the block.

## DoWhileStatement: DO Statement WHILE “(“ Expression “)” “;”

If the type of the expression is not integer, floating-point or a pointer type, raise an error. Emit Jmp bodyBlk and create a new **ControlScope** where:

B0

* The **ControlScope** has BreakTarget = nextBlk and ContinueTarget = bodyBlk.
* nextBlk is a new block that becomes the new current block after the **DoWhileStatement**.
* bodyBlk is a new block in which the statement is executed. Emit BrEI value, 0, nextBlk, bodyBlk as the last instruction in the block.
* value is read from the expression.

## ForStatement

### FOR “(“ ExpressionStatement ExpressionStatement “)” Statement {NoIterate}

As below, except with no third expression to evaluate.

### FOR “(“ ExpressionStatement ExpressionStatement Expression “)” Statement {Iterate}

If the type of the second expression is not integer, floating-point or a pointer type, raise an error. Emit code to evaluate the first expression, emit Jmp testBlk, and create a new **ControlScope** where:

B1

B0

* The **ControlScope** has BreakTarget = nextBlk and ContinueTarget = testBlk.
* nextBlk is a new block that becomes the new current block after the **ForStatement**. testBlk is a new block created by emitting BrEI value, 0, nextBlk, bodyBlk.
* value is read from the second expression.
* bodyBlk is a new block in which the statement is executed. Then emit code to evaluate the third expression (if present), and emit Jmp testBlk as the last instruction in the block.

In the diagram, B0 is testBlk and B1 is bodyBlk. The **ForStatement** is very similar to the WhileStatement, the only differences being the addition of the first expression and the third expression.

## SwitchStatement: SWITCH “(“ Expression “)” Statement

If the type of the expression is not integer, raise an error. Emit code to evaluate the expression, create a new **ControlScope**, execute the statement in an isolated block stmtBlk, and then for each case in the **ControlScope**, emit BrEI value, case, caseBlk, testBlk where:

B3

B1

B0

B2

B4

* The **ControlScope** has BreakTarget = nextBlk.
* stmtBlk is a new block in which the statement is executed. Emit Jmp nextBlk as the last instruction in the block.
* nextBlk is a new block that becomes the new current block after the **SwitchStatement**.
* value is read from the expression (this should be done only once, in the block for the first case).
* case is the test value of the case.
* caseBlk is the block associated with the case (inside stmtBlk).
* testBlk is a new block for the test code for the next case (BrEI ...). If there are no more cases and there is a default case, the default case must be used in BrEI as the branch target instead of testBlk. If there is no default case, nextBlk must be used as the branch target.

If there are no cases defined and there is a default case, emit Jmp dfltBlk where dfltBlk is the default case block. If there is no default case, emit Jmp nextBlk.

In the diagram, B0, B2, B4 are within stmtBlk. B1 and B3 are the case test blocks. Note that B­0 does not necessarily have a predecessor, as in the following example:

int a, b;

...

switch (...)

{

b = a + 2; // B­0

case 1: // B2

...

case 2: // B4

...

}

## ContinueStatement: CONTINUE “;”

Find the nearest **ControlScope** with a defined ContinueTarget. If there is none, raise an error. Otherwise, emit Jmp ContinueTarget. Create a new block and set it as the current block.

Note the importance of creating a new block after the Jmp operation. Consider the following code.

do

{

...

if (...)

continue;

...

} while (...);

The **IfStatement** is converted by creating a new block for the sub-statement, executing that sub-statement, and finally emitting a Jmp to the following block. This works even though the sub-statement logically ends with a jump:

bodyBlk:

...

BrEI value, 0, nextBlk, nonZeroBlk

nonZeroBlk: // sub-statement

Jmp bodyBlk // “continue;”

newBlk: // created by **ContinueStatement**

Jmp nextBlk // created by **IfStatement**

nextBlk:

...

newBlk is created by the **ContinueStatement** even though it is never used. This scheme enables greater consistency and does not lead to bigger code because dead code elimination will delete newBlk since it is unused.

## BreakStatement: BREAK “;”

Find the nearest **ControlScope** with a defined BreakTarget. If there is none, raise an error. Otherwise, emit Jmp BreakTarget. Create a new block and set it as the current block.

## ReturnStatement

### RETURN “;” {NoValue}

Emit Jmp end where end is the EndBlock in the current **FunctionScope**.

### RETURN Expression “;” {Value}

If the function does not have a return value, raise an error. Otherwise, update the Value of the variable with ID ReturnVariableId (from the current **FunctionScope**) to point to the value of the expression. Emit Jmp end where end is the EndBlock in the current **FunctionScope**.

## GotoStatement: GOTO ID “;”

If the label is defined in the current **FunctionScope**, Emit Jmp target where target is the label target. Otherwise, emit Jmp NULL and add the current block to the PendingGotos in the current **FunctionScope**. Create a new block and set it as the current block.

## Declaration

For each variable, add it to the current **Scope**, allocate a stack frame slot and emit LdF slot -> rx.

Initializers are then processed in order.

## ExpressionStatement

### “;” {Empty}

No action is performed.

### Expression “;” {NonEmpty}

Emit code to evaluate the expression.

## Block

### “{“ “}” {Empty}

No action is performed.

### “{“ StatementList “}” {NonEmpty}

Emit code to execute each statement in the statement list.

# Functions

## Arguments

Arguments are passed to functions in the Call/CallV/CallI/CallVI operations. These arguments are then logically stored in memory and accessible to the callee through the LdFA operation. When the name of an argument is first used, the address of the argument is loaded and an **IndirectStorageKind** variable is added to ParameterMap in the current **FunctionScope**.

## Structure return values

When a function returns a structure that is larger than 8 bytes, an extra parameter is inserted at the beginning of the parameter list. This parameter is a pointer to the return value, allocated by all callers before calling the function.

# Optimizations

## Constant propagation

Constant propagation is performed using a variant of “conditional constant propagation”. In this algorithm, every register is described using a predicate of one of the following forms:

If necessary, the algorithm can easily be expanded to allow arbitrary predicate forms. The standard laws of Boolean algebra apply, including:

Every block contains a lookup table mapping each register to a predicate. There is also a work queue containing the blocks that need to be processed by the algorithm. The algorithm begins with the start block in the work queue. For example:

Start:

LdI 1 -> r1

BrLI r1, 2, Block1, Block2

Block1:

LdI 2 -> r2

Jmp Common

Block2:

LdI 3 -> r3

Jmp Common

Common:

ɸ(r2,r3) -> r4

The algorithm begins by associating every register with in the Start block:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table for Start | r1 | r2 | r3 | r4 |
|  |  |  |  |

By the end of the first pass:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table for Start | r1 | r2 | r3 | r4 |
|  |  |  |  |

At the BrLI operation, the algorithm computes for the positive branch and records this in the table for Block1 (along with the other inherited values) while adding Block1 to the work queue. It also computes for the negative branch, but does not add Block2 to the work queue; execution cannot reach Block2 since the truth value computed is . By the end of the second pass:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table for Block1 | r1 | r2 | r3 | r4 |
|  |  |  |  |

Due to the Jmp operation, Common has been added to the work queue. In the third pass, the algorithm computes . By the end of the third pass:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table for Common | r1 | r2 | r3 | r4 |
|  |  |  |  |

For this example, the algorithm terminates here.

### Algorithm outline

The algorithm must also account for cycles in the control flow graph (CFG); these occur naturally when there are loops. An outline of the algorithm is presented below.

for every block B:

B.LastIteration starts as 0

B.Table starts as {}

B.PredTable starts as {}

B.VisitCount starts as 0

WorkQueue = { B0 } // start block

Iteration = 1

while WorkQueue ≠ {}:

remove block B from WorkQueue

Process(B)

rewrite all operations with constant values

algorithm terminates

Process(B):

Restarted = False

if B.LastIteration = Iteration:

Iteration = Iteration + 1 // cycle detected; restart in new iteration

Restarted = True

WorkQueue = {}

OldTable = B.Table

for each (R, V) in B.PredTable:

if R B.Table:

B.Table[R] =

B.PredTable = {}

ChangeList = {}

for each operation P in B that defines a register (including ɸ operations):

assume P defines register R

V = compute new predicate for P taking into consideration B.VisitCount

B.Table[R] = V

if (R OldTable and V ≠ ) or (R OldTable and V ≠ OldTable[R]):

add R to ChangeList

if (ChangeList ≠ {}) or Restarted:

examine the Jmp/Br that ends B to determine reachable successors

for each reachable successor S of B:

if S.LastIteration = Iteration:

for each R in ChangeList:

S.Table[R] =

else:

S.Table = B.Table // inherit value table

if B ends with a Br operation with a computable predicate:

// “computable predicate” means the second operand is

// effectively constant

assume R is the first operand in the Br operation

assume V is the effective predicate for the Br operation

if S is the negative branch:

V =

if R S.PredTable:

S.PredTable[R] =

else:

S.PredTable[R] = V

add S to WorkQueue

B.LastIteration = Iteration

B.VisitCount = B.VisitCount + 1

The algorithm could be improved by storing the register-to-predicate table in a global variable instead of a variable local to each block.

### Interpretation of top and bottom (tautology and contradiction)

At the end of the algorithm, every register should be associated with a predicate. If a register’s predicate is , the algorithm has essentially “given up” on tracking the register’s possible values (see below) – the register’s value is completely unknown. If a register’s predicate is , the register has no value, either because its definition depends on uninitialized values or because its definition occurs in an unreachable block.

### Computing new predicates for operations

The constant propagation algorithm effectively “executes” the intermediate code while determining possible values for each register. Loops in the code create cycles in the CFG, and these are handled in the algorithm by discarding results from previous “iterations”. Without a pre-determined number of iterations to run through, the algorithm is not guaranteed to terminate. Therefore, the VisitCount variable holds an integer indicating the number of times a block has been processed. If the count exceeds a set threshold, the predicate computation functions below must begin to return , eventually causing the algorithm to halt. Any constant propagation algorithm that did not require such a halting mechanism would effectively solve the halting problem.

### Examples of predicate computation functions

* and same with all other operators
* and

### Example

Consider the following C code fragment:

int a = 0;

int i = 0;

for (i = 0; i < 10; i++)

{

a++;

}

i += 10;

This may translate to:

Start:

LdI 0 -> r1

LdI 0 -> r2

Jmp Test

Test:

ɸ(r1,r3) -> r4

ɸ(r2,r5) -> r6

BrLI r6, 10, Body, Next

Body:

AddI r4, 1 -> r3

AddI r6, 1 -> r5

Jmp Test

Next:

AddI r6, 10 -> r7

The constant propagation algorithm would run as follows, assuming that the threshold for VisitCount is 1:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Register | r1 | r2 | r3 | r4 | r5 | r6 | r7 |
|  |  |  |  |  |  |  |  |
| Start |  |  |  |  |  |  |  |
| Test |  |  |  |  |  |  |  |
| Body |  |  |  |  |  |  |  |
| Test |  |  |  |  |  |  |  |
| Body |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Test |  |  |  |  |  |  |  |
| Body |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Test |  |  |  |  |  |  |  |
| Body |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Next |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

## Store forwarding

# Definitions

## BaseType

These are the basic types:

* Char (integer type)
* Short (integer type)
* Int (integer type)
* Long (integer type)
* LongLong (integer type)
* Float (floating-point type)
* Double (floating-point type)
* IntPtr (pointer type) – only for intermediate code; must not be used as an actual type (e.g. in **Type**)
* Void (void type) – only for the type system; must not be used for intermediate code

## Block

* PhiOperations : List of ɸ operations
* Operations : List of intermediate code operations

When a block is created, it must be added to “Blocks” in the current **FunctionScope**.

## ControlKeyword

* While
* DoWhile
* For
* Switch

## ControlScope

* Parent : ControlScope
* Keyword : ControlKeyword
* BreakTarget : Block (optional)
* ContinueTarget : Block (optional)
* Cases : Map from LongLong to Block
* DefaultCase : Block (optional)

## EnumField

* Name : String
* Value : Int

## EnumType

* Tag : String
* Fields : List of EnumField

## FunctionScope

* StartBlock : Block
* EndBlock : Block
* Blocks : Set of Block
* BlockMap : Map from Integer to Block
* Slots : Map from Integer to FrameSlot
* Variables : Map from VariableId to FunctionVariable
* SlotCache : Map from Integer to Register
* ReturnVariableId : VariableId
* ParameterMap : Map from Int to VariableId
* Labels : Map from String to Block
* PendingGotos : Multimap from String to Block

If the function has a return value, ReturnVariableId contains the ID of the “return variable”, which stores the return value. This variable must be added to the StartBlock before conversion begins. When conversion of the function’s statements has finished and ɸ operations have been inserted into EndBlock, a Srt operation is emitted as the second last instruction in EndBlock.

## FunctionType

* ReturnType : Type
* Variadic : Bool
* Parameters : List of Type

## FunctionVariable

* Parameter : Bool
* Index : Int
* AddressRegister : Register
* Type : Type

## FrameSlot

* Size : Int
* VariableId : VariableId

## ImmediateValue

* Type : BaseType
* Value : <type specified in BaseType>
* IsReference : Bool
* Reference : ObjectReference
* ReferenceOffset : Int

This structure describes a value used for literal values and immediate operands. If IsReference is True, Reference contains the ID of a linker object and ReferenceOffset contains the number of bytes to add to the address of the linker object. Type must be IntPtr.

## ObjectReference

An integer ID for a linker data object – e.g. global variable or function.

## OrdinarySymbol

### All sub-types

* Name : String

### VariableSymbolKind

* Id : VariableId

Id specifies a unique identifier for a variable. Since names can shadow names from an outer scope, Id distinguishes between two variables with the same name.

### ObjectSymbolKind

* Type : Type
* Reference : ObjectReference

This type is used for functions, global variables and static local variables.

### TypedefSymbolKind

* Target : Type

### ConstantSymbolKind

* Value : Int

## Register

An integer ID for a register.

## Scope

* Parent : Scope
* Symbols : Map from String to OrdinarySymbol
* Tags : Map from String to TagSymbol

This structure represents a lexical scope. It is only used during conversion to intermediate code in order to keep track of visible names. Labels are not included here because they have function-level scope and can only be resolved in a separate pass.

## Storage

### RegisterStorageKind

* Register : Register
* FunctionDesignator : Bool
* BitCount : Int
* BitOffset : Int

Register storage can never act as an l-value.

### IndirectStorageKind

* Address : Register
* LValue : Bool
* BitCount : Int
* BitOffset : Int

Indirect storage sometimes cannot act as an l-value (e.g. a struct returned from a function).

### NoStorageKind

This represents no value, i.e. “void”.

## StructField

* Name : String
* Type : Type
* Bits : Int

## StructType

* Keyword : TypeKeyword
* Tag : String
* Fields : List of StructField

## TagSymbol

* Name : String
* Type : Type

Any usage of a tag must be checked against the keyword with which it was originally defined. An error must be raised if there is a mismatch.

## Type

### All sub-types

* Traits : TypeTraits

### BaseTypeKind

* Base : BaseType
* EnumTag : String (optional)

Enumeration types are treated identically to Int types, but the enumeration tag is stored in EnumTag.

### PointerTypeKind

* Child : Type (optional)

If Child is not specified, the type is a void pointer.

### ArrayTypeKind

* Size : IntPtr (optional)
* Child : Type

If Size is not specified, an error must be raised when the size of the type is being calculated. The type then behaves identically to **PointerTypeKind** with the same Child.

### StructTypeKind

* Struct : StructType

### EnumTypeKind

* Enum : EnumType

This type is not used for definitions. It is only used for **TagSymbol** instances.

### IncompleteTypeKind

* Keyword : TypeKeyword
* Tag : String

Incomplete type specifications must be treated identically to struct type specifications if the struct is fully defined at a later point. For example, in

typedef struct my\_struct other\_type\_name;  
other\_type\_name \*a;  
struct my\_struct { int b; };

the variable “a” must be treated as a pointer to a full instance of my\_struct.

### FunctionTypeKind

* Function : FunctionType

## TypeKeyword

These correspond to the types which have “tags”.

* Struct
* Union
* Enum

## TypeTraits (flags)

* Signed (applies only to basic types Char, Short, Int, Long, LongLong)
* Unsigned (applies only to certain types as with Signed)
* Volatile
* Const

## TypedValue

* Type : Type
* Value : Storage

This structure describes a storage location along with a type. **TypedValues** are the result of expressions, and operators use them as input and for type checking.

## VariableId

An integer ID for a variable, preferably a 64-bit value. This is necessary to distinguish between different instances of variables with the same name (e.g. “i” in an outer scope and “i” in an inner scope).