# LLVM Language Reference Manual

**Written by Chris Lattner and Vikram Adve**


# Abstract


This document is a reference manual for the LLVM assembly language. LLVM is an SSA based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.


# Introduction

The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bytecode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations. The three different forms of LLVM are all equivalent. This document describes the human readable representation and notation.

The LLVM representation aims to be a light-weight and low-level while being expressive, typed, and extensible at the same time. It aims to be a "universal IR" of sorts, by being at a low enough level that high-level ideas may be cleanly mapped to it (similar to how microprocessors are "universal IR's", allowing many source languages to be mapped to them). By providing type information, LLVM can be used as the target of optimizations: for example, through pointer analysis, it can be proven that a C automatic variable is never accessed outside of the current function... allowing it to be promoted to a simple SSA value instead of a memory location.

---

**Well Formedness**

It is important to note that this document describes 'well formed' LLVM assembly language. There is a difference between what the parser accepts and what is considered 'well formed'. For example, the following instruction is syntactically okay, but not well formed:

```
%x = add int 1, %x
```

...because the definition of `%x` does not dominate all of its uses. The LLVM infrastructure provides a verification pass that may be used to verify that an LLVM module is well formed. This pass is automatically run by the parser after parsing input assembly, and by the optimizer before it outputs bytecode. The violations pointed out by the verifier pass indicate bugs in transformation passes or input to the parser.

# Identifiers

LLVM uses three different forms of identifiers, for different purposes:

1. Numeric constants are represented as you would expect: 12, -3 123.421, etc. Floating point constants have an optional hexidecimal notation.
2. Named values are represented as a string of characters with a '%' prefix. For example, %foo, %DivisionByZero, %a.really.long.identifier. The actual regular expression used is '`%[a-zA-Z$._][a-zA-Z$._0-9]*`'. Identifiers which require other characters in their names can be surrounded with quotes. In this way, anything except a `"` character can be used in a name.
3. Unnamed values are represented as an unsigned numeric value with a '%' prefix. For example, %12, %2, %44.

LLVM requires the values start with a '%' sign for two reasons: Compilers don't need to worry about name clashes with reserved words, and the set of reserved words may be expanded in the future without penalty. Additionally, unnamed identifiers allow a compiler to quickly come up with a temporary variable without having to avoid symbol table conflicts.

Reserved words in LLVM are very similar to reserved words in other languages. There are keywords for different opcodes ('`add`', '`cast`', '`ret`', etc...), for primitive type names ('`void`', '`uint`', etc...), and others. These reserved words cannot conflict with variable names, because none of them start with a '%' character.

Here is an example of LLVM code to multiply the integer variable '`%X`' by 8:

The easy way:

```
%result = mul uint %X, 8
```

After strength reduction:

```
%result = shl uint %X, ubyte 3
```

And the hard way:

```
add uint %X, %X           ; yields {uint}:%0
add uint %0, %0           ; yields {uint}:%1
%result = add uint %1, %1
```

This last way of multiplying `%X` by 8 illustrates several important lexical features of LLVM:

1. Comments are delimited with a '`;`' and go until the end of line.
2. Unnamed temporaries are created when the result of a computation is not assigned to a named value.
3. Unnamed temporaries are numbered sequentially

...and it also show a convention that we follow in this document. When demonstrating instructions, we will follow an instruction with a comment that defines the type and name of value produced. Comments are shown in italic text.

The one non-intuitive notation for constants is the optional hexidecimal form of floating point constants. For example, the form '`double 0x432ff973cafa8000`' is equivalent to (but harder to read than) '`double 4.5e+15`' which is also supported by the parser. The only time hexadecimal floating point constants are useful (and the only time that they are generated by the disassembler) is when an FP constant has to be emitted that is not representable as a decimal floating point number exactly. For example, NaN's, infinities, and other special cases are represented in their IEEE hexadecimal format so that assembly and disassembly do not cause any bits to change in the constants.

## Type System

The LLVM type system is one of the most important features of the intermediate representation. Being typed enables a number of optimizations to be performed on the IR directly, without having to do extra

analyses on the side before the transformation. A strong type system makes it easier to read the generated code and enables novel analyses and transformations that are not feasible to perform on normal three address code representations.

### Primitive Types

The primitive types are the fundemental building blocks of the LLVM system. The current set of primitive types are as follows:

| | |
|---|---|
| `void` | No value |
| `ubyte` | Unsigned 8 bit value |
| `ushort` | Unsigned 16 bit value |
| `uint` | Unsigned 32 bit value |
| `ulong` | Unsigned 64 bit value |
| `float` | 32 bit floating point value |
| `label` | Branch destination |

| | |
|---|---|
| `bool` | True or False value |
| `sbyte` | Signed 8 bit value |
| `short` | Signed 16 bit value |
| `int` | Signed 32 bit value |
| `long` | Signed 64 bit value |
| `double` | 64 bit floating point value |

---

## Type Classifications

These different primitive types fall into a few useful classifications:

| | |
|---|---|
| signed | `sbyte, short, int, long, float, double` |
| unsigned | `ubyte, ushort, uint, ulong` |
| integer | `ubyte, sbyte, ushort, short, uint, int, ulong, long` |
| integral | `bool, ubyte, sbyte, ushort, short, uint, int, ulong, long` |
| floating point | `float, double` |
| first class | `bool, ubyte, sbyte, ushort, short,`<br>`uint, int, ulong, long, float, double, pointer` |

### Derived Types

The real power in LLVM comes from the derived types in the system. This is what allows a programmer to represent arrays, functions, pointers, and other useful types. Note that these derived types may be recursive: For example, it is possible to have a two dimensional array.

---

## Array Type

#### Overview:

The array type is a very simple derived type that arranges elements sequentially in memory. The array type requires a size (number of elements) and an underlying data type.

#### Syntax:

```
[<# elements> x <elementtype>]
```

The number of elements is a constant integer value, elementtype may be any type with a size.

**Examples:**

`[40 x int ]`: Array of 40 integer values.
`[41 x int ]`: Array of 41 integer values.
`[40 x uint]`: Array of 40 unsigned integer values.

Here are some examples of multidimensional arrays:

`[3 x [4 x int]]` : 3x4 array integer values.
`[12 x [10 x float]]` : 12x10 array of single precision floating point values.
`[2 x [3 x [4 x uint]]]`: 2x3x4 array of unsigned integer values.

---

## Function Type

**Overview:**

The function type can be thought of as a function signature. It consists of a return type and a list of formal parameter types. Function types are usually used when to build virtual function tables (which are structures of pointers to functions), for indirect function calls, and when defining a function.

**Syntax:**

```
<returntype> (<parameter list>)
```

Where '`<parameter list>`' is a comma-separated list of type specifiers. Optionally, the parameter list may include a type `...`, which indicates that the function takes a variable number of arguments. Variable argument functions can access their arguments with the variable argument handling intrinsic functions.

**Examples:**

| | |
|---|---|
| `int (int)` | : function taking an `int`, returning an `int` |
| `float (int, int *) *` | : Pointer to a function that takes an `int` and a pointer to `int`, returning `float`. |
| `int (sbyte *, ...)` | : A vararg function that takes at least one pointer to `sbyte` (signed char in C), which returns an integer. This is the signature for `printf` in LLVM. |

---

## Structure Type

**Overview:**

The structure type is used to represent a collection of data members together in memory. The packing of the field types is defined to match the ABI of the underlying processor. The elements of a structure may be any type that has a size.

Structures are accessed using '`load` and '`store`' by getting a pointer to a field with the '`getelementptr`' instruction.

**Syntax:**

```
{ <type list> }
```

**Examples:**

| | |
|---|---|
| `{ int, int, int }` | : a triple of three `int` values |
| `{ float, int (int) * }` | : A pair, where the first element is a `float` and the second element is a pointer to a function that takes an `int`, returning an `int`. |

---

### Pointer Type

**Overview:**

As in many languages, the pointer type represents a pointer or reference to another object, which must live in memory.

**Syntax:**

```
<type> *
```

**Examples:**

`[4x int]*` : pointer to array of four `int` values
`int (int *) *`: A pointer to a function that takes an `int`, returning an `int`.

# High Level Structure

## Module Structure

LLVM programs are composed of "Module"s, each of which is a translation unit of the input programs. Each module consists of functions, global variables, and symbol table entries. Modules may be combined together with the LLVM linker, which merges function (and global variable) definitions, resolves forward declarations, and merges symbol table entries. Here is an example of the "hello world" module:

```
; Declare the string constant as a global constant...
%.LC0 = internal constant [13 x sbyte] c"hello world\0A\00"          ; [13 x sbyte]*

; External declaration of the puts function
declare int %puts(sbyte*)                                           ; int(sbyte*)*

; Definition of main function
int %main() {                                                       ; int()*
        ; Convert [13x sbyte]* to sbyte *...
        %cast210 = getelementptr [13 x sbyte]* %.LC0, long 0, long 0 ; sbyte*

        ; Call puts function to write out the string to stdout...
        call int %puts(sbyte* %cast210)                             ; int
        ret int 0
}
```

This example is made up of a global variable named "`.LC0`", an external declaration of the "`puts`" function, and a function definition for "`main`".

In general, a module is made up of a list of global values, where both functions and global variables are global values. Global values are represented by a pointer to a memory location (in this case, a pointer to an array of char, and a pointer to a function), and have one of the following linkage types:

**`internal`**
: Global values with internal linkage are only directly accessible by objects in the current module. In particular, linking code into a module with an internal global value may cause the internal to be renamed as necessary to avoid collisions. Because the symbol is internal to the module, all references can be updated. This corresponds to the notion of the '`static`' keyword in C, or the idea of "anonymous namespaces" in C++.

**`linkonce`**:
: "`linkonce`" linkage is similar to `internal` linkage, with the twist that linking together two modules defining the same `linkonce` globals will cause one of the globals to be discarded. This is typically used to implement inline functions. Unreferenced `linkonce` globals are allowed to be discarded.

**`weak`**:
: "`weak`" linkage is exactly the same as `linkonce` linkage, except that unreferenced `weak` globals may not be discarded. This is used to implement constructs in C such as "`int X;`" at global scope.

**`appending`**:
: "`appending`" linkage may only applied to global variables of pointer to array type. When two global variables with appending linkage are linked together, the two global arrays are appended together. This is the LLVM, typesafe, equivalent of having the system linker append together "sections" with identical names when .o files are linked.

**`externally visible`**:
: If none of the above identifiers are used, the global is externally visible, meaning that it participates in linkage and can be used to resolve external symbol references.

For example, since the "`.LC0`" variable is defined to be internal, if another module defined a "`.LC0`" variable and was linked with this one, one of the two would be renamed, preventing a collision. Since "`main`" and "`puts`" are external (i.e., lacking any linkage declarations), they are accessible outside of the current module. It is illegal for a function *declaration* to have any linkage type other than "externally visible".

## Global Variables

Global variables define regions of memory allocated at compilation time instead of run-time. Global variables may optionally be initialized. A variable may be defined as a global "constant", which indicates that the contents of the variable will never be modified (opening options for optimization). Constants must always have an initial value.

As SSA values, global variables define pointer values that are in scope (i.e. they dominate) for all basic blocks in the program. Global variables always define a pointer to their "content" type because they describe a region of memory, and all memory objects in LLVM are accessed through pointers.

## Functions

LLVM functions definitions are composed of a (possibly empty) argument list, an opening curly brace, a list of basic blocks, and a closing curly brace. LLVM function declarations are defined with the "`declare`" keyword, a function name and a function signature.

A function definition contains a list of basic blocks, forming the CFG for the function. Each basic block may optionally start with a label (giving the basic block a symbol table entry), contains a list of

instructions, and ends with a terminator instruction (such as a branch or function return).

The first basic block in program is special in two ways: it is immediately executed on entrance to the function, and it is not allowed to have predecessor basic blocks (i.e. there can not be any branches to the entry block of a function). Because the block can have no predecessors, it also cannot have any PHI nodes.

# Instruction Reference

The LLVM instruction set consists of several different classifications of instructions: terminator instructions, binary instructions, memory instructions, and other instructions.

## Terminator Instructions

As mentioned previously, every basic block in a program ends with a "Terminator" instruction, which indicates which block should be executed after the current block is finished. These terminator instructions typically yield a '`void`' value: they produce control flow, not values (the one exception being the '`invoke`' instruction).

There are five different terminator instructions: the '`ret`' instruction, the '`br`' instruction, the '`switch`' instruction, the '`invoke`' instruction, and the '`unwind`' instruction.

---

### '`ret`' Instruction

**Syntax:**

```
ret <type> <value>       ; Return a value from a non-void function
ret void                 ; Return from void function
```

**Overview:**

The '`ret`' instruction is used to return control flow (and a value) from a function, back to the caller.

There are two forms of the '`ret`' instructruction: one that returns a value and then causes control flow, and one that just causes control flow to occur.

**Arguments:**

The '`ret`' instruction may return any 'first class' type. Notice that a function is not well formed if there exists a '`ret`' instruction inside of the function that returns a value that does not match the return type of the function.

**Semantics:**

When the '`ret`' instruction is executed, control flow returns back to the calling function's context. If the caller is a "`call` instruction, execution continues at the instruction after the call. If the caller was an "`invoke`" instruction, execution continues at the beginning "normal" of the destination block. If the instruction returns a value, that value shall set the call or invoke instruction's return value.

**Example:**

```
ret int 5                       ; Return an integer value of 5
ret void                        ; Return from a void function
```

## 'br' Instruction

**Syntax:**

```
br bool <cond>, label <iftrue>, label <iffalse>
br label <dest>          ; Unconditional branch
```

**Overview:**

The 'br' instruction is used to cause control flow to transfer to a different basic block in the current function. There are two forms of this instruction, corresponding to a conditional branch and an unconditional branch.

**Arguments:**

The conditional branch form of the 'br' instruction takes a single 'bool' value and two 'label' values. The unconditional form of the 'br' instruction takes a single 'label' value as a target.

**Semantics:**

Upon execution of a conditional 'br' instruction, the 'bool' argument is evaluated. If the value is true, control flows to the 'iftrue' label argument. If "cond" is false, control flows to the 'iffalse' label argument.

**Example:**

```
Test:
  %cond = seteq int %a, %b
  br bool %cond, label %IfEqual, label %IfUnequal
IfEqual:
  ret int 1
IfUnequal:
  ret int 0
```

---

## 'switch' Instruction

**Syntax:**

```
switch uint <value>, label <defaultdest> [ int <val>, label &dest>, ... ]
```

**Overview:**

The 'switch' instruction is used to transfer control flow to one of several different places. It is a generalization of the 'br' instruction, allowing a branch to occur to one of many possible destinations.

**Arguments:**

The 'switch' instruction uses three parameters: a 'uint' comparison value 'value', a default 'label' destination, and an array of pairs of comparison value constants and 'label's.

**Semantics:**

The `switch` instruction specifies a table of values and destinations. When the '`switch`' instruction is executed, this table is searched for the given value. If the value is found, the corresponding destination is branched to, otherwise the default value it transfered to.

**Implementation:**

Depending on properties of the target machine and the particular `switch` instruction, this instruction may be code generated as a series of chained conditional branches, or with a lookup table.

**Example:**

```
; Emulate a conditional br instruction
%Val = cast bool %value to uint
switch uint %Val, label %truedest [int 0, label %falsedest ]

; Emulate an unconditional br instruction
switch uint 0, label %dest [ ]

; Implement a jump table:
switch uint %val, label %otherwise [ int 0, label %onzero,
                                     int 1, label %onone,
                                     int 2, label %ontwo ]
```

---

## '`invoke`' Instruction

**Syntax:**

```
<result> = invoke <ptr to function ty> %<function ptr val>(<function args>)
               to label <normal label> except label <exception label>
```

**Overview:**

The '`invoke`' instruction causes control to transfer to a specified function, with the possibility of control flow transfer to either the '`normal`' `label` label or the '`exception`' `label`. If the callee function returns with the "`ret`" instruction, control flow will return to the "normal" label. If the callee (or any indirect callees) returns with the "`unwind`" instruction, control is interrupted, and continued at the dynamically nearest "except" label.

**Arguments:**

This instruction requires several arguments:

1. '`ptr to function ty`': shall be the signature of the pointer to function value being invoked. In most cases, this is a direct function invocation, but indirect `invoke`s are just as possible, branching off an arbitrary pointer to function value.
2. '`function ptr val`': An LLVM value containing a pointer to a function to be invoked.
3. '`function args`': argument list whose types match the function signature argument types. If the function signature indicates the function accepts a variable number of arguments, the extra arguments can be specified.
4. '`normal label`': the label reached when the called function executes a '`ret`' instruction.
5. '`exception label`': the label reached when a callee returns with the `unwind` instruction.

**Semantics:**

This instruction is designed to operate as a standard '`call`' instruction in most regards. The primary difference is that it establishes an association with a label, which is used by the runtime library to unwind the stack.

This instruction is used in languages with destructors to ensure that proper cleanup is performed in the case of either a `longjmp` or a thrown exception. Additionally, this is important for implementation of '`catch`' clauses in high-level languages that support them.

**Example:**

```
%retval = invoke int %Test(int 15)
            to label %Continue
            except label %TestCleanup     ; {int}:retval set
```

---

### '`unwind`' Instruction

**Syntax:**

```
unwind
```

**Overview:**

The '`unwind`' instruction unwinds the stack, continuing control flow at the first callee in the dynamic call stack which used an `invoke` instruction to perform the call. This is primarily used to implement exception handling.

**Semantics:**

The '`unwind`' intrinsic causes execution of the current function to immediately halt. The dynamic call stack is then searched for the first `invoke` instruction on the call stack. Once found, execution continues at the "exceptional" destination block specified by the `invoke` instruction. If there is no `invoke` instruction in the dynamic call chain, undefined behavior results.

## Binary Operations

Binary operators are used to do most of the computation in a program. They require two operands, execute an operation on them, and produce a single value. The result value of a binary operator is not necessarily the same type as its operands.

There are several different binary operators:

---

### '`add`' Instruction

**Syntax:**

```
<result> = add <ty> <var1>, <var2>   ; yields {ty}:result
```

**Overview:**

The '`add`' instruction returns the sum of its two operands.

**Arguments:**
The two arguments to the '`add`' instruction must be either integer or floating point values. Both arguments must have identical types.

**Semantics:**

The value produced is the integer or floating point sum of the two operands.

**Example:**

```
<result> = add int 4, %var          ; yields {int}:result = 4 + %var
```

---

## '`sub`' Instruction

**Syntax:**

```
<result> = sub <ty> <var1>, <var2>   ; yields {ty}:result
```

**Overview:**

The '`sub`' instruction returns the difference of its two operands.

Note that the '`sub`' instruction is used to represent the '`neg`' instruction present in most other intermediate representations.

**Arguments:**

The two arguments to the '`sub`' instruction must be either integer or floating point values. Both arguments must have identical types.

**Semantics:**

The value produced is the integer or floating point difference of the two operands.

**Example:**

```
<result> = sub int 4, %var          ; yields {int}:result = 4 - %var
<result> = sub int 0, %val          ; yields {int}:result = -%var
```

---

## '`mul`' Instruction

**Syntax:**

```
<result> = mul <ty> <var1>, <var2>   ; yields {ty}:result
```

**Overview:**

The '`mul`' instruction returns the product of its two operands.

**Arguments:**

The two arguments to the '`mul`' instruction must be either integer or floating point values. Both arguments must have identical types.

**Semantics:**

The value produced is the integer or floating point product of the two operands.

There is no signed vs unsigned multiplication. The appropriate action is taken based on the type of the operand.

**Example:**

```
<result> = mul int 4, %var          ; yields {int}:result = 4 * %var
```

---

## '`div`' Instruction

**Syntax:**

```
<result> = div <ty> <var1>, <var2>   ; yields {ty}:result
```

**Overview:**

The '`div`' instruction returns the quotient of its two operands.

**Arguments:**

The two arguments to the '`div`' instruction must be either integer or floating point values. Both arguments must have identical types.

**Semantics:**

The value produced is the integer or floating point quotient of the two operands.

**Example:**

```
<result> = div int 4, %var          ; yields {int}:result = 4 / %var
```

---

## '`rem`' Instruction

**Syntax:**

```
<result> = rem <ty> <var1>, <var2>   ; yields {ty}:result
```

**Overview:**

The '`rem`' instruction returns the remainder from the division of its two operands.

**Arguments:**

The two arguments to the '`rem`' instruction must be either integer or floating point values. Both arguments must have identical types.

**Semantics:**

This returns the *remainder* of a division (where the result has the same sign as the divisor), not the *modulus* (where the result has the same sign as the dividend) of a value. For more information about the difference, see: The Math Forum.

**Example:**

```
<result> = rem int 4, %var          ; yields {int}:result = 4 % %var
```

---

## '`setcc`' Instructions

**Syntax:**

```
<result> = seteq <ty> <var1>, <var2>   ; yields {bool}:result
<result> = setne <ty> <var1>, <var2>   ; yields {bool}:result
<result> = setlt <ty> <var1>, <var2>   ; yields {bool}:result
<result> = setgt <ty> <var1>, <var2>   ; yields {bool}:result
<result> = setle <ty> <var1>, <var2>   ; yields {bool}:result
<result> = setge <ty> <var1>, <var2>   ; yields {bool}:result
```

**Overview:**

The '`setcc`' family of instructions returns a boolean value based on a comparison of their two operands.

**Arguments:**

The two arguments to the '`setcc`' instructions must be of first class or pointer type (it is not possible to compare '`label`'s, '`array`'s, '`structure`' or '`void`' values, etc...). Both arguments must have identical types.

**Semantics:**

The '`seteq`' instruction yields a `true` '`bool`' value if both operands are equal.
The '`setne`' instruction yields a `true` '`bool`' value if both operands are unequal.
The '`setlt`' instruction yields a `true` '`bool`' value if the first operand is less than the second operand.
The '`setgt`' instruction yields a `true` '`bool`' value if the first operand is greater than the second operand.
The '`setle`' instruction yields a `true` '`bool`' value if the first operand is less than or equal to the second operand.
The '`setge`' instruction yields a `true` '`bool`' value if the first operand is greater than or equal to the second operand.

**Example:**

```
<result> = seteq int   4, 5        ; yields {bool}:result = false
<result> = setne float 4, 5        ; yields {bool}:result = true
<result> = setlt uint  4, 5        ; yields {bool}:result = true
<result> = setgt sbyte 4, 5        ; yields {bool}:result = false
<result> = setle sbyte 4, 5        ; yields {bool}:result = true
<result> = setge sbyte 4, 5        ; yields {bool}:result = false
```

## Bitwise Binary Operations

Bitwise binary operators are used to do various forms of bit-twiddling in a program. They are generally very efficient instructions, and can commonly be strength reduced from other instructions. They require two operands, execute an operation on them, and produce a single value. The resulting value of the bitwise binary operators is always the same type as its first operand.

---

### 'and' Instruction

**Syntax:**

```
<result> = and <ty> <var1>, <var2>   ; yields {ty}:result
```

**Overview:**

The 'and' instruction returns the bitwise logical and of its two operands.

**Arguments:**

The two arguments to the 'and' instruction must be integral values. Both arguments must have identical types.

**Semantics:**

The truth table used for the 'and' instruction is:

| In0 | In1 | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Example:**

```
<result> = and int 4, %var         ; yields {int}:result = 4 & %var
<result> = and int 15, 40          ; yields {int}:result = 8
<result> = and int 4, 8            ; yields {int}:result = 0
```

---

### 'or' Instruction

**Syntax:**

```
<result> = or <ty> <var1>, <var2>   ; yields {ty}:result
```

**Overview:**

The 'or' instruction returns the bitwise logical inclusive or of its two operands.

**Arguments:**
The two arguments to the 'or' instruction must be integral values. Both arguments must have identical types.

**Semantics:**

The truth table used for the 'or' instruction is:

| In0 | In1 | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Example:**

```
<result> = or int 4, %var         ; yields {int}:result = 4 | %var
<result> = or int 15, 40          ; yields {int}:result = 47
<result> = or int 4, 8            ; yields {int}:result = 12
```

---

## 'xor' Instruction

**Syntax:**

```
<result> = xor <ty> <var1>, <var2>   ; yields {ty}:result
```

**Overview:**

The 'xor' instruction returns the bitwise logical exclusive or of its two operands. The xor is used to implement the "one's complement" operation, which is the "~" operator in C.

**Arguments:**

The two arguments to the 'xor' instruction must be integral values. Both arguments must have identical types.

**Semantics:**

The truth table used for the 'xor' instruction is:

| In0 | In1 | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Example:**

```
<result> = xor int 4, %var          ; yields {int}:result = 4 ^ %var
<result> = xor int 15, 40           ; yields {int}:result = 39
<result> = xor int 4, 8             ; yields {int}:result = 12
<result> = xor int %V, -1           ; yields {int}:result = ~%V
```

---

## 'shl' Instruction

**Syntax:**

```
<result> = shl <ty> <var1>, ubyte <var2>   ; yields {ty}:result
```

**Overview:**

The 'shl' instruction returns the first operand shifted to the left a specified number of bits.

**Arguments:**

The first argument to the 'shl' instruction must be an integer type. The second argument must be an 'ubyte' type.

**Semantics:**

The value produced is var1 * $2^{var2}$.

**Example:**

```
<result> = shl int 4, ubyte %var   ; yields {int}:result = 4 << %var
<result> = shl int 4, ubyte 2      ; yields {int}:result = 16
<result> = shl int 1, ubyte 10     ; yields {int}:result = 1024
```

---

## 'shr' Instruction

**Syntax:**

```
<result> = shr <ty> <var1>, ubyte <var2>   ; yields {ty}:result
```

**Overview:**

The 'shr' instruction returns the first operand shifted to the right a specified number of bits.

**Arguments:**

The first argument to the 'shr' instruction must be an integer type. The second argument must be an 'ubyte' type.

**Semantics:**

If the first argument is a signed type, the most significant bit is duplicated in the newly free'd bit positions. If the first argument is unsigned, zero bits shall fill the empty positions.

**Example:**

```
<result> = shr int 4, ubyte %var   ; yields {int}:result = 4 >> %var
<result> = shr uint 4, ubyte 1     ; yields {uint}:result = 2
<result> = shr int 4, ubyte 2      ; yields {int}:result = 1
<result> = shr sbyte 4, ubyte 3    ; yields {sbyte}:result = 0
<result> = shr sbyte -2, ubyte 1   ; yields {sbyte}:result = -1
```

## Memory Access Operations

A key design point of an SSA-based representation is how it represents memory. In LLVM, no memory locations are in SSA form, which makes things very simple. This section describes how to read, write, allocate and free memory in LLVM.

---

### 'malloc' Instruction

**Syntax:**

```
<result> = malloc <type>, uint <NumElements>     ; yields {type*}:result
<result> = malloc <type>                         ; yields {type*}:result
```

**Overview:**

The 'malloc' instruction allocates memory from the system heap and returns a pointer to it.

**Arguments:**

The the 'malloc' instruction allocates sizeof(<type>)*NumElements bytes of memory from the operating system, and returns a pointer of the appropriate type to the program. The second form of the instruction is a shorter version of the first instruction that defaults to allocating one element.

'type' must be a sized type.

**Semantics:**

Memory is allocated using the system "malloc" function, and a pointer is returned.

**Example:**

```
%array  = malloc [4 x ubyte ]                    ; yields {[%4 x ubyte]*}:array

%size   = add uint 2, 2                          ; yields {uint}:size = uint 4
%array1 = malloc ubyte, uint 4                   ; yields {ubyte*}:array1
%array2 = malloc [12 x ubyte], uint %size        ; yields {[12 x ubyte]*}:array2
```

---

### 'free' Instruction

**Syntax:**

```
free <type> <value>                              ; yields {void}
```

**Overview:**

The '`free`' instruction returns memory back to the unused memory heap, to be reallocated in the future.

**Arguments:**

'`value`' shall be a pointer value that points to a value that was allocated with the '`malloc`' instruction.

**Semantics:**

Access to the memory pointed to by the pointer is not longer defined after this instruction executes.

**Example:**

```
%array  = malloc [4 x ubyte]                    ; yields {[4 x ubyte]*}:array
          free   [4 x ubyte]* %array
```

---

## '`alloca`' Instruction

**Syntax:**

```
<result> = alloca <type>, uint <NumElements>  ; yields {type*}:result
<result> = alloca <type>                      ; yields {type*}:result
```

**Overview:**

The '`alloca`' instruction allocates memory on the current stack frame of the procedure that is live until the current function returns to its caller.

**Arguments:**

The the '`alloca`' instruction allocates `sizeof(<type>)*NumElements` bytes of memory on the runtime stack, returning a pointer of the appropriate type to the program. The second form of the instruction is a shorter version of the first that defaults to allocating one element.

'`type`' may be any sized type.

**Semantics:**

Memory is allocated, a pointer is returned. '`alloca`'d memory is automatically released when the function returns. The '`alloca`' instruction is commonly used to represent automatic variables that must have an address available. When the function returns (either with the `ret` or `invoke` instructions), the memory is reclaimed.

**Example:**

```
%ptr = alloca int                             ; yields {int*}:ptr
%ptr = alloca int, uint 4                     ; yields {int*}:ptr
```

---

## '`load`' Instruction

**Syntax:**

```
<result> = load <ty>* <pointer>
<result> = volatile load <ty>* <pointer>
```

**Overview:**
The '`load`' instruction is used to read from memory.

**Arguments:**

The argument to the '`load`' instruction specifies the memory address to load from. The pointer must point to a first class type. If the `load` is marked as `volatile` then the optimizer is not allowed to modify the number or order of execution of this `load` with other volatile `load` and `store` instructions.

**Semantics:**

The location of memory pointed to is loaded.

**Examples:**

```
%ptr = alloca int                               ; yields {int*}:ptr
store int 3, int* %ptr                          ; yields {void}
%val = load int* %ptr                           ; yields {int}:val = int 3
```

---

## '`store`' Instruction

**Syntax:**

```
store <ty> <value>, <ty>* <pointer>                 ; yields {void}
volatile store <ty> <value>, <ty>* <pointer>                ; yields {void}
```

**Overview:**

The '`store`' instruction is used to write to memory.

**Arguments:**

There are two arguments to the '`store`' instruction: a value to store and an address to store it into. The type of the '`<pointer>`' operand must be a pointer to the type of the '`<value>`' operand. If the `store` is marked as `volatile` then the optimizer is not allowed to modify the number or order of execution of this `store` with other volatile `load` and `store` instructions.

**Semantics:**

The contents of memory are updated to contain '`<value>`' at the location specified by the '`<pointer>`' operand.

**Example:**

```
%ptr = alloca int                               ; yields {int*}:ptr
store int 3, int* %ptr                          ; yields {void}
%val = load int* %ptr                           ; yields {int}:val = int 3
```

---

### 'getelementptr' Instruction

**Syntax:**

```
  <result> = getelementptr <ty>* <ptrval>{, long <aidx>|, ubyte <sidx>}*
```

**Overview:**

The 'getelementptr' instruction is used to get the address of a subelement of an aggregate data structure.

**Arguments:**

This instruction takes a list of long values and ubyte constants that indicate what form of addressing to perform. The actual types of the arguments provided depend on the type of the first pointer argument. The 'getelementptr' instruction is used to index down through the type levels of a structure.

For example, lets consider a C code fragment and how it gets compiled to LLVM:

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

The LLVM code generated by the GCC frontend is:

```
%RT = type { sbyte, [10 x [20 x int]], sbyte }
%ST = type { int, double, %RT }

int* "foo"(%ST* %s) {
  %reg = getelementptr %ST* %s, long 1, ubyte 2, ubyte 1, long 5, long 13
  ret int* %reg
}
```

**Semantics:**

The index types specified for the 'getelementptr' instruction depend on the pointer type that is being index into. Pointer and array types require 'long' values, and structure types require 'ubyte' **constants**.

In the example above, the first index is indexing into the '%ST*' type, which is a pointer, yielding a '%ST' = '{ int, double, %RT }' type, a structure. The second index indexes into the third element of the structure, yielding a '%RT' = '{ sbyte, [10 x [20 x int]], sbyte }' type, another structure. The third index indexes into the second element of the structure, yielding a '[10 x [20 x int]]' type, an array. The two dimensions of the array are subscripted into, yielding an 'int' type. The 'getelementptr' instruction return a pointer to this element, thus yielding a 'int*' type.

Note that it is perfectly legal to index partially through a structure, returning a pointer to an inner element. Because of this, the LLVM code for the given testcase is equivalent to:

```
int* "foo"(%ST* %s) {
  %t1 = getelementptr %ST* %s , long 1                        ; yields %ST*:%t1
  %t2 = getelementptr %ST* %t1, long 0, ubyte 2               ; yields %RT*:%t2
  %t3 = getelementptr %RT* %t2, long 0, ubyte 1               ; yields [10 x [20 x int]]*:%t3
  %t4 = getelementptr [10 x [20 x int]]* %t3, long 0, long 5  ; yields [20 x int]*:%t4
  %t5 = getelementptr [20 x int]* %t4, long 0, long 13        ; yields int*:%t5
  ret int* %t5
}
```

**Example:**

```
  ; yields [12 x ubyte]*:aptr
  %aptr = getelementptr {int, [12 x ubyte]}* %sptr, long 0, ubyte 1
```

## Other Operations

The instructions in this catagory are the "miscellaneous" instructions, which defy better classification.

---

### 'phi' Instruction

**Syntax:**

```
  <result> = phi <ty> [ <val0>, <label0>], ...
```

**Overview:**

The 'phi' instruction is used to implement the φ node in the SSA graph representing the function.

**Arguments:**

The type of the incoming values are specified with the first type field. After this, the 'phi' instruction takes a list of pairs as arguments, with one pair for each predecessor basic block of the current block.

There must be no non-phi instructions between the start of a basic block and the PHI instructions: i.e. PHI instructions must be first in a basic block.

**Semantics:**

At runtime, the 'phi' instruction logically takes on the value specified by the parameter, depending on which basic block we came from in the last terminator instruction.

**Example:**

```
Loop:       ; Infinite loop that counts from 0 on up...
  %indvar = phi uint [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add uint %indvar, 1
  br label %Loop
```

---

### 'cast .. to' Instruction

**Syntax:**

```
  <result> = cast <ty> <value> to <ty2>             ; yields ty2
```

**Overview:**

The '`cast`' instruction is used as the primitive means to convert integers to floating point, change data type sizes, and break type safety (by casting pointers).

**Arguments:**

The '`cast`' instruction takes a value to cast, which must be a first class value, and a type to cast it to, which must also be a first class type.

**Semantics:**

This instruction follows the C rules for explicit casts when determining how the data being cast must change to fit in its new container.

When casting to bool, any value that would be considered true in the context of a C '`if`' condition is converted to the boolean '`true`' values, all else are '`false`'.

When extending an integral value from a type of one signness to another (for example '`sbyte`' to '`ulong`'), the value is sign-extended if the **source** value is signed, and zero-extended if the source value is unsigned. `bool` values are always zero extended into either zero or one.

**Example:**

```
%X = cast int 257 to ubyte              ; yields ubyte:1
%Y = cast int 123 to bool               ; yields bool:true
```

---

## '`call`' Instruction

**Syntax:**

```
<result> = call <ty>* <fnptrval>(<param list>)
```

**Overview:**

The '`call`' instruction represents a simple function call.

**Arguments:**

This instruction requires several arguments:

1. '`ty`': shall be the signature of the pointer to function value being invoked. The argument types must match the types implied by this signature.

2. '`fnptrval`': An LLVM value containing a pointer to a function to be invoked. In most cases, this is a direct function invocation, but indirect `call`s are just as possible, calling an arbitrary pointer to function values.

3. '`function args`': argument list whose types match the function signature argument types. If the function signature indicates the function accepts a variable number of arguments, the extra arguments can be specified.

**Semantics:**

The '`call`' instruction is used to cause control flow to transfer to a specified function, with its incoming arguments bound to the specified values. Upon a '`ret`' instruction in the called function, control flow continues with the instruction after the function call, and the return value of the function is bound to the result argument. This is a simpler case of the invoke instruction.

**Example:**

```
%retval = call int %test(int %argc)
call int(sbyte*, ...) *%printf(sbyte* %msg, int 12, sbyte 42);
```

---

## '`vanext`' Instruction

**Syntax:**

```
<resultarglist> = vanext <va_list> <arglist>, <argty>
```

**Overview:**

The '`vanext`' instruction is used to access arguments passed through the "variable argument" area of a function call. It is used to implement the `va_arg` macro in C.

**Arguments:**

This instruction takes a `valist` value and the type of the argument. It returns another `valist`.

**Semantics:**

The '`vanext`' instruction advances the specified `valist` past an argument of the specified type. In conjunction with the `vaarg` instruction, it is used to implement the `va_arg` macro available in C. For more information, see the variable argument handling Intrinsic Functions.

It is legal for this instruction to be called in a function which does not take a variable number of arguments, for example, the `vfprintf` function.

`vanext` is an LLVM instruction instead of an intrinsic function because it takes an type as an argument.

**Example:**

See the variable argument processing section.

---

## '`vaarg`' Instruction

**Syntax:**

```
<resultval> = vaarg <va_list> <arglist>, <argty>
```

**Overview:**

The '`vaarg`' instruction is used to access arguments passed through the "variable argument" area of a

function call. It is used to implement the `va_arg` macro in C.

**Arguments:**

This instruction takes a `valist` value and the type of the argument. It returns a value of the specified argument type.

**Semantics:**

The '`vaarg`' instruction loads an argument of the specified type from the specified `va_list`. In conjunction with the `vanext` instruction, it is used to implement the `va_arg` macro available in C. For more information, see the variable argument handling Intrinsic Functions.

It is legal for this instruction to be called in a function which does not take a variable number of arguments, for example, the `vfprintf` function.

`vaarg` is an LLVM instruction instead of an intrinsic function because it takes an type as an argument.

**Example:**

See the variable argument processing section.

# Intrinsic Functions

LLVM supports the notion of an "intrinsic function". These functions have well known names and semantics, and are required to follow certain restrictions. Overall, these instructions represent an extension mechanism for the LLVM language that does not require changing all of the transformations in LLVM to add to the language (or the bytecode reader/writer, the parser, etc...).

Intrinsic function names must all start with an "`llvm.`" prefix, this prefix is reserved in LLVM for intrinsic names, thus functions may not be named this. Intrinsic functions must always be external functions: you cannot define the body of intrinsic functions. Intrinsic functions may only be used in call or invoke instructions: it is illegal to take the address of an intrinsic function. Additionally, because intrinsic functions are part of the LLVM language, it is required that they all be documented here if any are added.

Unless an intrinsic function is target-specific, there must be a lowering pass to eliminate the intrinsic or all backends must support the intrinsic function.

## Variable Argument Handling Intrinsics

Variable argument support is defined in LLVM with the `vanext` instruction and these three intrinsic functions. These functions are related to the similarly named macros defined in the `<stdarg.h>` header file.

All of these functions operate on arguments that use a target-specific value type "`va_list`". The LLVM assembly language reference manual does not define what this type is, so all transformations should be prepared to handle intrinsics with any type used.

This example shows how the `vanext` instruction and the variable argument handling intrinsic functions are used.

```
int %test(int %X, ...) {
  ; Initialize variable argument processing
  %ap = call sbyte*()* %llvm.va_start()
```

```
  ; Read a single integer argument
  %tmp = vaarg sbyte* %ap, int

  ; Advance to the next argument
  %ap2 = vanext sbyte* %ap, int

  ; Demonstrate usage of llvm.va_copy and llvm.va_end
  %aq = call sbyte* (sbyte*)* %llvm.va_copy(sbyte* %ap2)
  call void %llvm.va_end(sbyte* %aq)

  ; Stop processing of arguments.
  call void %llvm.va_end(sbyte* %ap2)
  ret int %tmp
}
```

---

### 'llvm.va_start' Intrinsic

**Syntax:**

```
call va_list ()* %llvm.va_start()
```

**Overview:**

The 'llvm.va_start' intrinsic returns a new <arglist> for subsequent use by the variable argument intrinsics.

**Semantics:**

The 'llvm.va_start' intrinsic works just like the va_start macro available in C. In a target-dependent way, it initializes and returns a va_list element, so that the next vaarg will produce the first variable argument passed to the function. Unlike the C va_start macro, this intrinsic does not need to know the last argument of the function, the compiler can figure that out.

Note that this intrinsic function is only legal to be called from within the body of a variable argument function.

---

### 'llvm.va_end' Intrinsic

**Syntax:**

```
call void (va_list)* %llvm.va_end(va_list <arglist>)
```

**Overview:**

The 'llvm.va_end' intrinsic destroys <arglist> which has been initialized previously with llvm.va_start or llvm.va_copy.

**Arguments:**

The argument is a va_list to destroy.

**Semantics:**

The '`llvm.va_end`' intrinsic works just like the `va_end` macro available in C. In a target-dependent way, it destroys the `va_list`. Calls to `llvm.va_start` and `llvm.va_copy` must be matched exactly with calls to `llvm.va_end`.

---

### '`llvm.va_copy`' Intrinsic

**Syntax:**

```
call va_list (va_list)* %llvm.va_copy(va_list <destarglist>)
```

**Overview:**

The '`llvm.va_copy`' intrinsic copies the current argument position from the source argument list to the destination argument list.

**Arguments:**

The argument is the `va_list` to copy.

**Semantics:**

The '`llvm.va_copy`' intrinsic works just like the `va_copy` macro available in C. In a target-dependent way, it copies the source `va_list` element into the returned list. This intrinsic is necessary because the `llvm.va_start` intrinsic may be arbitrarily complex and require memory allocation, for example.

---

*Chris Lattner*
The LLVM Compiler Infrastructure
Last modified: Tue Oct 21 10:43:36 CDT 2003