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 a Static Single Assignment (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 bitcode 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 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 i32 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 bitcode. The violations pointed out by the verifier pass indicate bugs in transformation passes or input to the parser.

Identifiers

LLVM identifiers come in two basic types: global and local. Global identifiers (functions, global variables) begin with the `'@'` character. Local identifiers (register names, types) begin with the `'%'` character. Additionally, there are three different formats for identifiers, for different purposes:

1. Named values are represented as a string of characters with their 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. Special characters may be escaped using `"\xx"` where `xx` is the ASCII code for the character in hexadecimal. In this way, any character can be used in a name value, even quotes themselves.
2. Unnamed values are represented as an unsigned numeric value with their prefix. For example, `%12`, `@2`, `%44`.
3. Constants, which are described in a section about constants, below.

LLVM requires that values start with a prefix 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', 'bitcast', 'ret', etc...), for primitive type names ('void', 'i32', etc...), and others. These reserved words cannot conflict with variable names, because none of them start with a prefix character ('`%`' or '`@`').

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

The easy way:

```
%result = mul i32 %X, 8
```

After strength reduction:

```
%result = shl i32 %X, i8 3
```

And the hard way:

```
add i32 %X, %X           ; yields {i32}:%0
add i32 %0, %0           ; yields {i32}:%1
%result = add i32 %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 shows 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.

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 i8] c"hello world\0A\00"          ; [13 x i8]*

; External declaration of the puts function
```

```
declare i32 @puts(i8 *)                                         ; i32(i8 *)*

; Definition of main function
define i32 @main() {                                            ; i32()*
        ; Convert [13 x i8]* to i8  *...
        %cast210 = getelementptr [13 x i8]* @.LC0, i64 0, i64 0   ; i8 *

        ; Call puts function to write out the string to stdout...
        call i32 @puts(i8 * %cast210)                           ; i32
        ret i32 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.

Linkage Types

All Global Variables and Functions have one of the following types of linkage:

private:
Global values with private linkage are only directly accessible by objects in the current module. In particular, linking code into a module with an private global value may cause the private to be renamed as necessary to avoid collisions. Because the symbol is private to the module, all references can be updated. This doesn't show up in any symbol table in the object file.

linker_private:
Similar to private, but the symbol is passed through the assembler and removed by the linker after evaluation.

internal:
Similar to private, but the value shows as a local symbol (STB_LOCAL in the case of ELF) in the object file. This corresponds to the notion of the 'static' keyword in C.

available_externally:
Globals with "available_externally" linkage are never emitted into the object file corresponding to the LLVM module. They exist to allow inlining and other optimizations to take place given knowledge of the definition of the global, which is known to be somewhere outside the module. Globals with available_externally linkage are allowed to be discarded at will, and are otherwise the same as linkonce_odr. This linkage type is only allowed on definitions, not declarations.

linkonce:
Globals with "linkonce" linkage are merged with other globals of the same name when linkage occurs. This is typically used to implement inline functions, templates, or other code which must be generated in each translation unit that uses it. Unreferenced linkonce globals are allowed to be discarded.

weak:
"weak" linkage has the same merging semantics as linkonce linkage, except that unreferenced globals with weak linkage may not be discarded. This is used for globals that are declared "weak" in C source code.

common:
"common" linkage is most similar to "weak" linkage, but they are used for tentative definitions in C, such as "int X;" at global scope. Symbols with "common" linkage are merged in the same way as weak symbols, and they may not be deleted if unreferenced. common symbols may not have an explicit section, must have a zero initializer, and may not be marked 'constant'. Functions and aliases may not have common linkage.

appending:
"appending" linkage may only be 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.

extern_weak:

The semantics of this linkage follow the ELF object file model: the symbol is weak until linked, if not linked, the symbol becomes null instead of being an undefined reference.

linkonce_odr:

weak_odr:

Some languages allow differing globals to be merged, such as two functions with different semantics. Other languages, such as `C++`, ensure that only equivalent globals are ever merged (the "one definition rule" - "ODR"). Such languages can use the `linkonce_odr` and `weak_odr` linkage types to indicate that the global will only be merged with equivalent globals. These linkage types are otherwise the same as their non-`odr` versions.

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.

The next two types of linkage are targeted for Microsoft Windows platform only. They are designed to support importing (exporting) symbols from (to) DLLs (Dynamic Link Libraries).

dllimport:

"`dllimport`" linkage causes the compiler to reference a function or variable via a global pointer to a pointer that is set up by the DLL exporting the symbol. On Microsoft Windows targets, the pointer name is formed by combining `__imp_` and the function or variable name.

dllexport:

"`dllexport`" linkage causes the compiler to provide a global pointer to a pointer in a DLL, so that it can be referenced with the `dllimport` attribute. On Microsoft Windows targets, the pointer name is formed by combining `__imp_` and the function or variable name.

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", `dllimport` or `extern_weak`.

Aliases can have only `external`, `internal`, `weak` or `weak_odr` linkages.

Calling Conventions

LLVM functions, calls and invokes can all have an optional calling convention specified for the call. The calling convention of any pair of dynamic caller/callee must match, or the behavior of the program is undefined. The following calling conventions are supported by LLVM, and more may be added in the future:

"`ccc`" - The C calling convention:

This calling convention (the default if no other calling convention is specified) matches the target C calling conventions. This calling convention supports varargs function calls and tolerates some mismatch in the declared prototype and implemented declaration of the function (as does normal C).

"`fastcc`" - The fast calling convention:

This calling convention attempts to make calls as fast as possible (e.g. by passing things in registers). This calling convention allows the target to use whatever tricks it wants to produce fast code for the target, without having to conform to an externally specified ABI (Application Binary Interface). Implementations of this convention should allow arbitrary tail call optimization to be supported. This calling convention does not support varargs and requires the prototype of all callees to exactly match the prototype of the function definition.

"`coldcc`" - The cold calling convention:

This calling convention attempts to make code in the caller as efficient as possible under the assumption that the call is not commonly executed. As such, these calls often preserve all registers so that the call does not break any live ranges in the caller side. This calling convention does not support varargs and requires the prototype of all callees to exactly match the prototype of the function definition.

"`cc <n>`" - Numbered convention:

Any calling convention may be specified by number, allowing target-specific calling conventions to be

used. Target specific calling conventions start at 64.

More calling conventions can be added/defined on an as-needed basis, to support Pascal conventions or any other well-known target-independent convention.

Visibility Styles

All Global Variables and Functions have one of the following visibility styles:

"`default`" - Default style:
On targets that use the ELF object file format, default visibility means that the declaration is visible to other modules and, in shared libraries, means that the declared entity may be overridden. On Darwin, default visibility means that the declaration is visible to other modules. Default visibility corresponds to "external linkage" in the language.

"`hidden`" - Hidden style:
Two declarations of an object with hidden visibility refer to the same object if they are in the same shared object. Usually, hidden visibility indicates that the symbol will not be placed into the dynamic symbol table, so no other module (executable or shared library) can reference it directly.

"`protected`" - Protected style:
On ELF, protected visibility indicates that the symbol will be placed in the dynamic symbol table, but that references within the defining module will bind to the local symbol. That is, the symbol cannot be overridden by another module.

Named Types

LLVM IR allows you to specify name aliases for certain types. This can make it easier to read the IR and make the IR more condensed (particularly when recursive types are involved). An example of a name specification is:

```
%mytype = type { %mytype*, i32 }
```

You may give a name to any type except "void". Type name aliases may be used anywhere a type is expected with the syntax "%mytype".

Note that type names are aliases for the structural type that they indicate, and that you can therefore specify multiple names for the same type. This often leads to confusing behavior when dumping out a .ll file. Since LLVM IR uses structural typing, the name is not part of the type. When printing out LLVM IR, the printer will pick *one name* to render all types of a particular shape. This means that if you have code where two different source types end up having the same LLVM type, that the dumper will sometimes print the "wrong" or unexpected type. This is an important design point and isn't going to change.

Global Variables

Global variables define regions of memory allocated at compilation time instead of run-time. Global variables may optionally be initialized, may have an explicit section to be placed in, and may have an optional explicit alignment specified. A variable may be defined as "thread_local", which means that it will not be shared by threads (each thread will have a separated copy of the variable). A variable may be defined as a global "constant," which indicates that the contents of the variable will **never** be modified (enabling better optimization, allowing the global data to be placed in the read-only section of an executable, etc). Note that variables that need runtime initialization cannot be marked "constant" as there is a store to the variable.

LLVM explicitly allows *declarations* of global variables to be marked constant, even if the final definition of the global is not. This capability can be used to enable slightly better optimization of the program, but requires the language definition to guarantee that optimizations based on the 'constantness' are valid for the translation units that do not include the definition.

As SSA values, global variables define pointer values that are in scope (i.e. they dominate) 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.

A global variable may be declared to reside in a target-specific numbered address space. For targets that support them, address spaces may affect how optimizations are performed and/or what target instructions are used to access the variable. The default address space is zero. The address space qualifier must precede any other attributes.

LLVM allows an explicit section to be specified for globals. If the target supports it, it will emit globals to the section specified.

An explicit alignment may be specified for a global. If not present, or if the alignment is set to zero, the alignment of the global is set by the target to whatever it feels convenient. If an explicit alignment is specified, the global is forced to have at least that much alignment. All alignments must be a power of 2.

For example, the following defines a global in a numbered address space with an initializer, section, and alignment:

```
@G = addrspace(5) constant float 1.0, section "foo", align 4
```

Functions

LLVM function definitions consist of the "`define`" keyord, an optional linkage type, an optional visibility style, an optional calling convention, a return type, an optional parameter attribute for the return type, a function name, a (possibly empty) argument list (each with optional parameter attributes), optional function attributes, an optional section, an optional alignment, an optional garbage collector name, an opening curly brace, a list of basic blocks, and a closing curly brace.

LLVM function declarations consist of the "`declare`" keyword, an optional linkage type, an optional visibility style, an optional calling convention, a return type, an optional parameter attribute for the return type, a function name, a possibly empty list of arguments, an optional alignment, and an optional garbage collector name.

A function definition contains a list of basic blocks, forming the CFG (Control Flow Graph) 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 a function 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.

LLVM allows an explicit section to be specified for functions. If the target supports it, it will emit functions to the section specified.

An explicit alignment may be specified for a function. If not present, or if the alignment is set to zero, the alignment of the function is set by the target to whatever it feels convenient. If an explicit alignment is specified, the function is forced to have at least that much alignment. All alignments must be a power of 2.

Syntax:

```
define [linkage] [visibility]
       [cconv] [ret attrs]
       <ResultType> @<FunctionName> ([argument list])
       [fn Attrs] [section "name"] [align N]
       [gc] { ... }
```

Aliases

Aliases act as "second name" for the aliasee value (which can be either function, global variable, another alias or bitcast of global value). Aliases may have an optional linkage type, and an optional visibility style.

Syntax:

```
@<Name> = alias [Linkage] [Visibility] <AliaseeTy> @<Aliasee>
```

Parameter Attributes

The return type and each parameter of a function type may have a set of *parameter attributes* associated with them. Parameter attributes are used to communicate additional information about the result or parameters of a function. Parameter attributes are considered to be part of the function, not of the function type, so functions with different parameter attributes can have the same function type.

Parameter attributes are simple keywords that follow the type specified. If multiple parameter attributes are needed, they are space separated. For example:

```
declare i32 @printf(i8* noalias nocapture, ...)
declare i32 @atoi(i8 zeroext)
declare signext i8 @returns_signed_char()
```

Note that any attributes for the function result (`nounwind`, `readonly`) come immediately after the argument list.

Currently, only the following parameter attributes are defined:

`zeroext`
: This indicates to the code generator that the parameter or return value should be zero-extended to a 32-bit value by the caller (for a parameter) or the callee (for a return value).

`signext`
: This indicates to the code generator that the parameter or return value should be sign-extended to a 32-bit value by the caller (for a parameter) or the callee (for a return value).

`inreg`
: This indicates that this parameter or return value should be treated in a special target-dependent fashion during while emitting code for a function call or return (usually, by putting it in a register as opposed to memory, though some targets use it to distinguish between two different kinds of registers). Use of this attribute is target-specific.

`byval`
: This indicates that the pointer parameter should really be passed by value to the function. The attribute implies that a hidden copy of the pointee is made between the caller and the callee, so the callee is unable to modify the value in the callee. This attribute is only valid on LLVM pointer arguments. It is generally used to pass structs and arrays by value, but is also valid on pointers to scalars. The copy is considered to belong to the caller not the callee (for example, `readonly` functions should not write to `byval` parameters). This is not a valid attribute for return values. The byval attribute also supports specifying an alignment with the align attribute. This has a target-specific effect on the code generator that usually indicates a desired alignment for the synthesized stack slot.

`sret`
: This indicates that the pointer parameter specifies the address of a structure that is the return value of the function in the source program. This pointer must be guaranteed by the caller to be valid: loads and stores to the structure may be assumed by the callee to not to trap. This may only be applied to the first parameter. This is not a valid attribute for return values.

`noalias`
: This indicates that the pointer does not alias any global or any other parameter. The caller is responsible for ensuring that this is the case. On a function return value, `noalias` additionally indicates that the pointer does not alias any other pointers visible to the caller. For further details, please see the discussion of the NoAlias response in alias analysis.

`nocapture`
: This indicates that the callee does not make any copies of the pointer that outlive the callee itself. This is not a valid attribute for return values.

`nest`
: This indicates that the pointer parameter can be excised using the trampoline intrinsics. This is not a valid attribute for return values.

Garbage Collector Names

Each function may specify a garbage collector name, which is simply a string:

```
define void @f() gc "name" { ...
```

The compiler declares the supported values of *name*. Specifying a collector which will cause the compiler to alter its output in order to support the named garbage collection algorithm.

Function Attributes

Function attributes are set to communicate additional information about a function. Function attributes are considered to be part of the function, not of the function type, so functions with different parameter attributes can have the same function type.

Function attributes are simple keywords that follow the type specified. If multiple attributes are needed, they are space separated. For example:

```
define void @f() noinline { ... }
define void @f() alwaysinline { ... }
define void @f() alwaysinline optsize { ... }
define void @f() optsize
```

`alwaysinline`
: This attribute indicates that the inliner should attempt to inline this function into callers whenever possible, ignoring any active inlining size threshold for this caller.

`noinline`
: This attribute indicates that the inliner should never inline this function in any situation. This attribute may not be used together with the `alwaysinline` attribute.

`optsize`
: This attribute suggests that optimization passes and code generator passes make choices that keep the code size of this function low, and otherwise do optimizations specifically to reduce code size.

`noreturn`
: This function attribute indicates that the function never returns normally. This produces undefined behavior at runtime if the function ever does dynamically return.

`nounwind`
: This function attribute indicates that the function never returns with an unwind or exceptional control flow. If the function does unwind, its runtime behavior is undefined.

`readnone`
: This attribute indicates that the function computes its result (or decides to unwind an exception) based strictly on its arguments, without dereferencing any pointer arguments or otherwise accessing any mutable state (e.g. memory, control registers, etc) visible to caller functions. It does not write through any pointer arguments (including `byval` arguments) and never changes any state visible to callers. This means that it cannot unwind exceptions by calling the `C++` exception throwing methods, but could use the `unwind` instruction.

`readonly`
: This attribute indicates that the function does not write through any pointer arguments (including `byval` arguments) or otherwise modify any state (e.g. memory, control registers, etc) visible to caller functions. It may dereference pointer arguments and read state that may be set in the caller. A readonly function always returns the same value (or unwinds an exception identically) when called with the same set of arguments and global state. It cannot unwind an exception by calling the `C++`

exception throwing methods, but may use the `unwind` instruction.

`ssp`
: This attribute indicates that the function should emit a stack smashing protector. It is in the form of a "canary"—a random value placed on the stack before the local variables that's checked upon return from the function to see if it has been overwritten. A heuristic is used to determine if a function needs stack protectors or not.

  If a function that has an `ssp` attribute is inlined into a function that doesn't have an `ssp` attribute, then the resulting function will have an `ssp` attribute.

`sspreq`
: This attribute indicates that the function should *always* emit a stack smashing protector. This overrides the `ssp` function attribute.

  If a function that has an `sspreq` attribute is inlined into a function that doesn't have an `sspreq` attribute or which has an `ssp` attribute, then the resulting function will have an `sspreq` attribute.

`noredzone`
: This attribute indicates that the code generator should not use a red zone, even if the target-specific ABI normally permits it.

`noimplicitfloat`
: This attributes disables implicit floating point instructions.

`naked`
: This attribute disables prologue / epilogue emission for the function. This can have very system-specific consequences.

Module-Level Inline Assembly

Modules may contain "module-level inline asm" blocks, which corresponds to the GCC "file scope inline asm" blocks. These blocks are internally concatenated by LLVM and treated as a single unit, but may be separated in the `.ll` file if desired. The syntax is very simple:

```
module asm "inline asm code goes here"
module asm "more can go here"
```

The strings can contain any character by escaping non-printable characters. The escape sequence used is simply "\xx" where "xx" is the two digit hex code for the number.

The inline asm code is simply printed to the machine code .s file when assembly code is generated.

Data Layout

A module may specify a target specific data layout string that specifies how data is to be laid out in memory. The syntax for the data layout is simply:

```
target datalayout = "layout specification"
```

The *layout specification* consists of a list of specifications separated by the minus sign character ('-'). Each specification starts with a letter and may include other information after the letter to define some aspect of the data layout. The specifications accepted are as follows:

`E`
: Specifies that the target lays out data in big-endian form. That is, the bits with the most significance have the lowest address location.

`e`
: Specifies that the target lays out data in little-endian form. That is, the bits with the least significance have the lowest address location.

`p:`*`size`*`:`*`abi`*`:`*`pref`*
: This specifies the *size* of a pointer and its *abi* and *preferred* alignments. All sizes are in bits. Specifying the *pref* alignment is optional. If omitted, the preceding `:` should be omitted too.

`isize:abi:pref`
: This specifies the alignment for an integer type of a given bit *size*. The value of *size* must be in the range [1,2^23).

`vsize:abi:pref`
: This specifies the alignment for a vector type of a given bit *size*.

`fsize:abi:pref`
: This specifies the alignment for a floating point type of a given bit *size*. The value of *size* must be either 32 (float) or 64 (double).

`asize:abi:pref`
: This specifies the alignment for an aggregate type of a given bit *size*.

`ssize:abi:pref`
: This specifies the alignment for a stack object of a given bit *size*.

When constructing the data layout for a given target, LLVM starts with a default set of specifications which are then (possibly) overriden by the specifications in the `datalayout` keyword. The default specifications are given in this list:

- `E` - big endian
- `p:32:64:64` - 32-bit pointers with 64-bit alignment
- `i1:8:8` - i1 is 8-bit (byte) aligned
- `i8:8:8` - i8 is 8-bit (byte) aligned
- `i16:16:16` - i16 is 16-bit aligned
- `i32:32:32` - i32 is 32-bit aligned
- `i64:32:64` - i64 has ABI alignment of 32-bits but preferred alignment of 64-bits
- `f32:32:32` - float is 32-bit aligned
- `f64:64:64` - double is 64-bit aligned
- `v64:64:64` - 64-bit vector is 64-bit aligned
- `v128:128:128` - 128-bit vector is 128-bit aligned
- `a0:0:1` - aggregates are 8-bit aligned
- `s0:64:64` - stack objects are 64-bit aligned

When LLVM is determining the alignment for a given type, it uses the following rules:

1. If the type sought is an exact match for one of the specifications, that specification is used.
2. If no match is found, and the type sought is an integer type, then the smallest integer type that is larger than the bitwidth of the sought type is used. If none of the specifications are larger than the bitwidth then the the largest integer type is used. For example, given the default specifications above, the i7 type will use the alignment of i8 (next largest) while both i65 and i256 will use the alignment of i64 (largest specified).
3. If no match is found, and the type sought is a vector type, then the largest vector type that is smaller than the sought vector type will be used as a fall back. This happens because <128 x double> can be implemented in terms of 64 <2 x double>, for example.

Pointer Aliasing Rules

Any memory access must be done through a pointer value associated with an address range of the memory access, otherwise the behavior is undefined. Pointer values are associated with address ranges according to the following rules:

- A pointer value formed from a `getelementptr` instruction is associated with the addresses associated with the first operand of the `getelementptr`.
- An address of a global variable is associated with the address range of the variable's storage.
- The result value of an allocation instruction is associated with the address range of the allocated storage.
- A null pointer in the default address-space is associated with no address.
- A pointer value formed by an `inttoptr` is associated with all address ranges of all pointer values that contribute (directly or indirectly) to the computation of the pointer's value.
- The result value of a `bitcast` is associated with all addresses associated with the operand of the `bitcast`.
- An integer constant other than zero or a pointer value returned from a function not defined within

LLVM may be associated with address ranges allocated through mechanisms other than those provided by LLVM. Such ranges shall not overlap with any ranges of addresses allocated by mechanisms provided by LLVM.

LLVM IR does not associate types with memory. The result type of a `load` merely indicates the size and alignment of the memory from which to load, as well as the interpretation of the value. The first operand of a `store` similarly only indicates the size and alignment of the store.

Consequently, type-based alias analysis, aka TBAA, aka `-fstrict-aliasing`, is not applicable to general unadorned LLVM IR. Metadata may be used to encode additional information which specialized optimization passes may use to implement type-based alias analysis.

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 intermediate representation 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.

Type Classifications

The types fall into a few useful classifications:

Classification	Types
integer	`i1, i2, i3, ... i8, ... i16, ... i32, ... i64, ...`
floating point	`float, double, x86_fp80, fp128, ppc_fp128`
first class	integer, floating point, pointer, vector, structure, array, label, metadata.
primitive	label, void, floating point, metadata.
derived	integer, array, function, pointer, structure, packed structure, vector, opaque.

The first class types are perhaps the most important. Values of these types are the only ones which can be produced by instructions, passed as arguments, or used as operands to instructions.

Primitive Types

The primitive types are the fundamental building blocks of the LLVM system.

Floating Point Types

Type	Description
`float`	32-bit floating point value
`double`	64-bit floating point value
`fp128`	128-bit floating point value (112-bit mantissa)
`x86_fp80`	80-bit floating point value (X87)
`ppc_fp128`	128-bit floating point value (two 64-bits)

Void Type

Overview:

The void type does not represent any value and has no size.

Syntax:

```
void
```

Label Type

Overview:

The label type represents code labels.

Syntax:

```
label
```

Metadata Type

Overview:

The metadata type represents embedded metadata. The only derived type that may contain metadata is `metadata*` or a function type that returns or takes metadata typed parameters, but not pointer to metadata types.

Syntax:

```
metadata
```

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.

Integer Type

Overview:

The integer type is a very simple derived type that simply specifies an arbitrary bit width for the integer type desired. Any bit width from 1 bit to 2^23-1 (about 8 million) can be specified.

Syntax:

```
iN
```

The number of bits the integer will occupy is specified by the `N` value.

Examples:

`i1`	a single-bit integer.
`i32`	a 32-bit integer.
`i1942652`	a really big integer of over 1 million bits.

Note that the code generator does not yet support large integer types to be used as function return types. The specific limit on how large a return type the code generator can currently handle is target-dependent; currently it's often 64 bits for 32-bit targets and 128 bits for 64-bit targets.

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 i32]`	Array of 40 32-bit integer values.
`[41 x i32]`	Array of 41 32-bit integer values.
`[4 x i8]`	Array of 4 8-bit integer values.

Here are some examples of multidimensional arrays:

`[3 x [4 x i32]]`	3x4 array of 32-bit integer values.
`[12 x [10 x float]]`	12x10 array of single precision floating point values.
`[2 x [3 x [4 x i16]]]`	2x3x4 array of 16-bit integer values.

Note that 'variable sized arrays' can be implemented in LLVM with a zero length array. Normally, accesses past the end of an array are undefined in LLVM (e.g. it is illegal to access the 5th element of a 3 element array). As a special case, however, zero length arrays are recognized to be variable length. This allows implementation of 'pascal style arrays' with the LLVM type "`{ i32, [0 x float]}`", for example.

Note that the code generator does not yet support large aggregate types to be used as function return types. The specific limit on how large an aggregate return type the code generator can currently handle is target-dependent, and also dependent on the aggregate element types.

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. The return type of a function type is a scalar type, a void type, or a struct type. If the return type is a struct type then all struct elements must be of first class types, and the struct must have at least one element.

Syntax:

```
<returntype list> (<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. '`<returntype list>`' is a comma-separated list of first class type specifiers.

Examples:

`i32 (i32)`	function taking an `i32`, returning an `i32`
`float (i16 signext, i32 *) *`	Pointer to a function that takes an `i16` that should be sign extended and a pointer to `i32`, returning `float`.
`i32 (i8*, ...)`	A vararg function that takes at least one pointer to `i8` (char in C), which returns an integer. This is the signature for `printf` in LLVM.

`{i32, i32} (i32)`	A function taking an `i32`, returning two `i32` values as an aggregate of type `{ i32, i32 }`

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:

`{ i32, i32, i32 }`	A triple of three `i32` values
`{ float, i32 (i32) * }`	A pair, where the first element is a `float` and the second element is a pointer to a function that takes an `i32`, returning an `i32`.

Note that the code generator does not yet support large aggregate types to be used as function return types. The specific limit on how large an aggregate return type the code generator can currently handle is target-dependent, and also dependent on the aggregate element types.

Packed Structure Type

Overview:

The packed structure type is used to represent a collection of data members together in memory. There is no padding between fields. Further, the alignment of a packed structure is 1 byte. The elements of a packed 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:

`< { i32, i32, i32 } >`	A triple of three `i32` values
`< { float, i32 (i32)* } >`	A pair, where the first element is a `float` and the second element is a pointer to a function that takes an `i32`, returning an `i32`.

Pointer Type

Overview:

As in many languages, the pointer type represents a pointer or reference to another object, which must live in memory. Pointer types may have an optional address space attribute defining the target-specific numbered address space where the pointed-to object resides. The default address space is zero.

Note that LLVM does not permit pointers to void (`void*`) nor does it permit pointers to labels (`label*`). Use `i8*` instead.

Syntax:

```
<type> *
```

Examples:

`[4 x i32]*`	A pointer to array of four `i32` values.
`i32 (i32 *) *`	A pointer to a function that takes an `i32*`, returning an `i32`.
`i32 addrspace(5)*`	A pointer to an `i32` value that resides in address space #5.

Vector Type

Overview:

A vector type is a simple derived type that represents a vector of elements. Vector types are used when multiple primitive data are operated in parallel using a single instruction (SIMD). A vector type requires a size (number of elements) and an underlying primitive data type. Vectors must have a power of two length (1, 2, 4, 8, 16 ...). Vector types are considered first class.

Syntax:

```
< <# elements> x <elementtype> >
```

The number of elements is a constant integer value; elementtype may be any integer or floating point type.

Examples:

`<4 x i32>`	Vector of 4 32-bit integer values.
`<8 x float>`	Vector of 8 32-bit floating-point values.
`<2 x i64>`	Vector of 2 64-bit integer values.

Note that the code generator does not yet support large vector types to be used as function return types. The specific limit on how large a vector return type codegen can currently handle is target-dependent; currently it's often a few times longer than a hardware vector register.

Opaque Type

Overview:

Opaque types are used to represent unknown types in the system. This corresponds (for example) to the C notion of a forward declared structure type. In LLVM, opaque types can eventually be resolved to any type (not just a structure type).

Syntax:

```
opaque
```

Examples:

`opaque`	An opaque type.

Type Up-references

Overview:

An "up reference" allows you to refer to a lexically enclosing type without requiring it to have a name. For instance, a structure declaration may contain a pointer to any of the types it is lexically a member of. Example of up references (with their equivalent as named type declarations) include:

```
{ \2 * }                %x = type { %x* }
{ \2 }*                 %y = type { %y }*
\1*                     %z = type %z*
```

An up reference is needed by the asmprinter for printing out cyclic types when there is no declared name for a type in the cycle. Because the asmprinter does not want to print out an infinite type string, it needs a syntax to handle recursive types that have no names (all names are optional in llvm IR).

Syntax:

```
\<level>
```

The level is the count of the lexical type that is being referred to.

Examples:

`\1*`	Self-referential pointer.
`{ { \3*, i8 }, i32 }`	Recursive structure where the upref refers to the out-most structure.

Constants

LLVM has several different basic types of constants. This section describes them all and their syntax.

Simple Constants

Boolean constants
The two strings '`true`' and '`false`' are both valid constants of the `i1` type.

Integer constants
Standard integers (such as '4') are constants of the integer type. Negative numbers may be used with integer types.

Floating point constants
Floating point constants use standard decimal notation (e.g. 123.421), exponential notation (e.g. 1.23421e+2), or a more precise hexadecimal notation (see below). The assembler requires the exact decimal value of a floating-point constant. For example, the assembler accepts 1.25 but rejects 1.3 because 1.3 is a repeating decimal in binary. Floating point constants must have a floating point type.

Null pointer constants
The identifier '`null`' is recognized as a null pointer constant and must be of pointer type.

The one non-intuitive notation for constants is the hexadecimal form of floating point constants. For example, the form '`double 0x432ff973cafa8000`' is equivalent to (but harder to read than) '`double 4.5e+15`'. The only time hexadecimal floating point constants are required (and the only time that they are generated by the disassembler) is when a floating point constant must be emitted but it cannot be represented as a decimal floating point number in a reasonable number of digits. For example, NaN's, infinities, and other special values are represented in their IEEE hexadecimal format so that assembly and disassembly do not cause any bits to change in the constants.

When using the hexadecimal form, constants of types float and double are represented using the 16-digit form shown above (which matches the IEEE754 representation for double); float values must, however, be exactly representable as IEE754 single precision. Hexadecimal format is always used for long double,

and there are three forms of long double. The 80-bit format used by x86 is represented as `0xK` followed by 20 hexadecimal digits. The 128-bit format used by PowerPC (two adjacent doubles) is represented by `0xM` followed by 32 hexadecimal digits. The IEEE 128-bit format is represented by `0xL` followed by 32 hexadecimal digits; no currently supported target uses this format. Long doubles will only work if they match the long double format on your target. All hexadecimal formats are big-endian (sign bit at the left).

Complex Constants

Complex constants are a (potentially recursive) combination of simple constants and smaller complex constants.

Structure constants
Structure constants are represented with notation similar to structure type definitions (a comma separated list of elements, surrounded by braces (`{}`)). For example: "`{ i32 4, float 17.0, i32* @G }`", where "`@G`" is declared as "`@G = external global i32`". Structure constants must have structure type, and the number and types of elements must match those specified by the type.

Array constants
Array constants are represented with notation similar to array type definitions (a comma separated list of elements, surrounded by square brackets (`[]`)). For example: "`[ i32 42, i32 11, i32 74 ]`". Array constants must have array type, and the number and types of elements must match those specified by the type.

Vector constants
Vector constants are represented with notation similar to vector type definitions (a comma separated list of elements, surrounded by less-than/greater-than's (`<>`)). For example: "`< i32 42, i32 11, i32 74, i32 100 >`". Vector constants must have vector type, and the number and types of elements must match those specified by the type.

Zero initialization
The string '`zeroinitializer`' can be used to zero initialize a value to zero of *any* type, including scalar and aggregate types. This is often used to avoid having to print large zero initializers (e.g. for large arrays) and is always exactly equivalent to using explicit zero initializers.

Metadata node
A metadata node is a structure-like constant with metadata type. For example: "`metadata !{ i32 0, metadata !"test" }`". Unlike other constants that are meant to be interpreted as part of the instruction stream, metadata is a place to attach additional information such as debug info.

Global Variable and Function Addresses

The addresses of global variables and functions are always implicitly valid (link-time) constants. These constants are explicitly referenced when the identifier for the global is used and always have pointer type. For example, the following is a legal LLVM file:

```
@X = global i32 17
@Y = global i32 42
@Z = global [2 x i32*] [ i32* @X, i32* @Y ]
```

Undefined Values

The string '`undef`' is recognized as a type-less constant that has no specific value. Undefined values may be of any type and be used anywhere a constant is permitted.

Undefined values indicate to the compiler that the program is well defined no matter what value is used, giving the compiler more freedom to optimize.

Constant Expressions

Constant expressions are used to allow expressions involving other constants to be used as constants. Constant expressions may be of any first class type and may involve any LLVM operation that does not

have side effects (e.g. load and call are not supported). The following is the syntax for constant expressions:

trunc (CST to TYPE)
Truncate a constant to another type. The bit size of CST must be larger than the bit size of TYPE. Both types must be integers.

zext (CST to TYPE)
Zero extend a constant to another type. The bit size of CST must be smaller or equal to the bit size of TYPE. Both types must be integers.

sext (CST to TYPE)
Sign extend a constant to another type. The bit size of CST must be smaller or equal to the bit size of TYPE. Both types must be integers.

fptrunc (CST to TYPE)
Truncate a floating point constant to another floating point type. The size of CST must be larger than the size of TYPE. Both types must be floating point.

fpext (CST to TYPE)
Floating point extend a constant to another type. The size of CST must be smaller or equal to the size of TYPE. Both types must be floating point.

fptoui (CST to TYPE)
Convert a floating point constant to the corresponding unsigned integer constant. TYPE must be a scalar or vector integer type. CST must be of scalar or vector floating point type. Both CST and TYPE must be scalars, or vectors of the same number of elements. If the value won't fit in the integer type, the results are undefined.

fptosi (CST to TYPE)
Convert a floating point constant to the corresponding signed integer constant. TYPE must be a scalar or vector integer type. CST must be of scalar or vector floating point type. Both CST and TYPE must be scalars, or vectors of the same number of elements. If the value won't fit in the integer type, the results are undefined.

uitofp (CST to TYPE)
Convert an unsigned integer constant to the corresponding floating point constant. TYPE must be a scalar or vector floating point type. CST must be of scalar or vector integer type. Both CST and TYPE must be scalars, or vectors of the same number of elements. If the value won't fit in the floating point type, the results are undefined.

sitofp (CST to TYPE)
Convert a signed integer constant to the corresponding floating point constant. TYPE must be a scalar or vector floating point type. CST must be of scalar or vector integer type. Both CST and TYPE must be scalars, or vectors of the same number of elements. If the value won't fit in the floating point type, the results are undefined.

ptrtoint (CST to TYPE)
Convert a pointer typed constant to the corresponding integer constant `TYPE` must be an integer type. `CST` must be of pointer type. The `CST` value is zero extended, truncated, or unchanged to make it fit in `TYPE`.

inttoptr (CST to TYPE)
Convert a integer constant to a pointer constant. TYPE must be a pointer type. CST must be of integer type. The CST value is zero extended, truncated, or unchanged to make it fit in a pointer size. This one is *really* dangerous!

bitcast (CST to TYPE)
Convert a constant, CST, to another TYPE. The constraints of the operands are the same as those for the bitcast instruction.

getelementptr (CSTPTR, IDX0, IDX1, ...)
getelementptr inbounds (CSTPTR, IDX0, IDX1, ...)
Perform the getelementptr operation on constants. As with the getelementptr instruction, the index list may have zero or more indexes, which are required to make sense for the type of "CSTPTR".

select (COND, VAL1, VAL2)
Perform the select operation on constants.

icmp COND (VAL1, VAL2)
Performs the icmp operation on constants.

fcmp COND (VAL1, VAL2)
Performs the fcmp operation on constants.

extractelement (VAL, IDX)

Perform the extractelement operation on constants.

`insertelement ( VAL, ELT, IDX )`

Perform the insertelement operation on constants.

`shufflevector ( VEC1, VEC2, IDXMASK )`

Perform the shufflevector operation on constants.

`OPCODE ( LHS, RHS )`

Perform the specified operation of the LHS and RHS constants. OPCODE may be any of the binary or bitwise binary operations. The constraints on operands are the same as those for the corresponding instruction (e.g. no bitwise operations on floating point values are allowed).

Embedded Metadata

Embedded metadata provides a way to attach arbitrary data to the instruction stream without affecting the behaviour of the program. There are two metadata primitives, strings and nodes. All metadata has the `metadata` type and is identified in syntax by a preceding exclamation point ('`!`').

A metadata string is a string surrounded by double quotes. It can contain any character by escaping non-printable characters with "\xx" where "xx" is the two digit hex code. For example: "`!"test\00"`".

Metadata nodes are represented with notation similar to structure constants (a comma separated list of elements, surrounded by braces and preceeded by an exclamation point). For example: "`!{ metadata !"test\00", i32 10}`".

A metadata node will attempt to track changes to the values it holds. In the event that a value is deleted, it will be replaced with a typeless "`null`", such as "`metadata !{null, i32 10}`".

Optimizations may rely on metadata to provide additional information about the program that isn't available in the instructions, or that isn't easily computable. Similarly, the code generator may expect a certain metadata format to be used to express debugging information.

Other Values

Inline Assembler Expressions

LLVM supports inline assembler expressions (as opposed to Module-Level Inline Assembly) through the use of a special value. This value represents the inline assembler as a string (containing the instructions to emit), a list of operand constraints (stored as a string), and a flag that indicates whether or not the inline asm expression has side effects. An example inline assembler expression is:

```
i32 (i32) asm "bswap $0", "=r,r"
```

Inline assembler expressions may **only** be used as the callee operand of a `call` instruction. Thus, typically we have:

```
%X = call i32 asm "bswap $0", "=r,r"(i32 %Y)
```

Inline asms with side effects not visible in the constraint list must be marked as having side effects. This is done through the use of the '`sideeffect`' keyword, like so:

```
call void asm sideeffect "eieio", ""()
```

TODO: The format of the asm and constraints string still need to be documented here. Constraints on what can be done (e.g. duplication, moving, etc need to be documented). This is probably best done by reference to another document that covers inline asm from a holistic perspective.

Intrinsic Global Variables

LLVM has a number of "magic" global variables that contain data that affect code generation or other IR semantics. These are documented here. All globals of this sort should have a section specified as "`llvm.metadata`". This section and all globals that start with "`llvm.`" are reserved for use by LLVM.

The '`llvm.used`' Global Variable

The `@llvm.used` global is an array with i8* element type which has appending linkage. This array contains a list of pointers to global variables and functions which may optionally have a pointer cast formed of bitcast or getelementptr. For example, a legal use of it is:

```
@X = global i8 4
@Y = global i32 123

@llvm.used = appending global [2 x i8*] [
   i8* @X,
   i8* bitcast (i32* @Y to i8*)
], section "llvm.metadata"
```

If a global variable appears in the `@llvm.used` list, then the compiler, assembler, and linker are required to treat the symbol as if there is a reference to the global that it cannot see. For example, if a variable has internal linkage and no references other than that from the `@llvm.used` list, it cannot be deleted. This is commonly used to represent references from inline asms and other things the compiler cannot "see", and corresponds to "attribute((used))" in GNU C.

On some targets, the code generator must emit a directive to the assembler or object file to prevent the assembler and linker from molesting the symbol.

The '`llvm.compiler.used`' Global Variable

The `@llvm.compiler.used` directive is the same as the `@llvm.used` directive, except that it only prevents the compiler from touching the symbol. On targets that support it, this allows an intelligent linker to optimize references to the symbol without being impeded as it would be by `@llvm.used`.

This is a rare construct that should only be used in rare circumstances, and should not be exposed to source languages.

The '`llvm.global_ctors`' Global Variable

TODO: Describe this.

The '`llvm.global_dtors`' Global Variable

TODO: Describe this.

Instruction Reference

The LLVM instruction set consists of several different classifications of instructions: terminator instructions, binary instructions, bitwise 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 six different terminator instructions: the '`ret`' instruction, the '`br`' instruction, the '`switch`' instruction, the '`invoke`' instruction, the '`unwind`' instruction, and the '`unreachable`' 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 optionally a value) from a function back to the caller.

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

Arguments:

The '`ret`' instruction optionally accepts a single argument, the return value. The type of the return value must be a '`first class`' type.

A function is not `well formed` if it it has a non-void return type and contains a '`ret`' instruction with no return value or a return value with a type that does not match its type, or if it has a void return type and contains a '`ret`' instruction with a return value.

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 of the "normal" destination block. If the instruction returns a value, that value shall set the call or invoke instruction's return value.

Example:

```
ret i32 5                       ; Return an integer value of 5
ret void                        ; Return from a void function
ret { i32, i8 } { i32 4, i8 2 } ; Return a struct of values 4 and 2
```

Note that the code generator does not yet fully support large return values. The specific sizes that are currently supported are dependent on the target. For integers, on 32-bit targets the limit is often 64 bits, and on 64-bit targets the limit is often 128 bits. For aggregate types, the current limits are dependent on the element types; for example targets are often limited to 2 total integer elements and 2 total floating-point elements.

'`br`' Instruction

Syntax:

```
br i1 <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 '`i1`' 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 '`i1`' 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 = icmp eq i32 %a, %b
  br i1 %cond, label %IfEqual, label %IfUnequal
IfEqual:
  ret i32 1
IfUnequal:
  ret i32 0
```

'*switch*' Instruction

Syntax:

```
  switch <intty> <value>, label <defaultdest> [ <intty> <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: an integer comparison value '`value`', a default '`label`' destination, and an array of pairs of comparison value constants and '`label`'s. The table is not allowed to contain duplicate constant entries.

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, control flow is transfered to the corresponding destination; otherwise, control flow is transfered to the default destination.

Implementation:

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

Example:

```
 ; Emulate a conditional br instruction
 %Val = zext i1 %value to i32
 switch i32 %Val, label %truedest [ i32 0, label %falsedest ]

 ; Emulate an unconditional br instruction
 switch i32 0, label %dest [ ]

 ; Implement a jump table:
 switch i32 %val, label %otherwise [ i32 0, label %onzero
                                     i32 1, label %onone
                                     i32 2, label %ontwo ]
```

'*invoke*' Instruction

Syntax:

```
<result> = invoke [cconv] [ret attrs] <ptr to function ty> <function ptr val>(<function args>) [fn attrs]
             to label <normal label> unwind 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 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 "exception" label.

Arguments:

This instruction requires several arguments:

1. The optional "cconv" marker indicates which calling convention the call should use. If none is specified, the call defaults to using C calling conventions.
2. The optional Parameter Attributes list for return values. Only '`zeroext`', '`signext`', and '`inreg`' attributes are valid here.
3. '`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.
4. '`function ptr val`': An LLVM value containing a pointer to a function to be invoked.
5. '`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.
6. '`normal label`': the label reached when the called function executes a '`ret`' instruction.
7. '`exception label`': the label reached when a callee returns with the `unwind` instruction.
8. The optional function attributes list. Only '`noreturn`', '`nounwind`', '`readonly`' and '`readnone`' attributes are valid here.

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.

For the purposes of the SSA form, the definition of the value returned by the '`invoke`' instruction is deemed to occur on the edge from the current block to the "normal" label. If the callee unwinds then no return value is available.

Example:

```
%retval = invoke i32 @Test(i32 15) to label %Continue
            unwind label %TestCleanup              ; {i32}:retval set
%retval = invoke coldcc i32 %Testfnptr(i32 15) to label %Continue
            unwind label %TestCleanup              ; {i32}: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`' instruction 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.

'*`unreachable`*' *Instruction*

Syntax:

```
unreachable
```

Overview:

The '`unreachable`' instruction has no defined semantics. This instruction is used to inform the optimizer that a particular portion of the code is not reachable. This can be used to indicate that the code after a no-return function cannot be reached, and other facts.

Semantics:

The '`unreachable`' instruction has no defined semantics.

Binary Operations

Binary operators are used to do most of the computation in a program. They require two operands of the same type, execute an operation on them, and produce a single value. The operands might represent multiple data, as is the case with the vector data type. The result value has the same type as its operands.

There are several different binary operators:

'*`add`*' *Instruction*

Syntax:

```
<result> = add <ty> <op1>, <op2>          ; yields {ty}:result
<result> = add nuw <ty> <op1>, <op2>      ; yields {ty}:result
<result> = add nsw <ty> <op1>, <op2>      ; yields {ty}:result
<result> = add nuw nsw <ty> <op1>, <op2>  ; yields {ty}:result
```

Overview:

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

Arguments:

The two arguments to the '`add`' instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

The value produced is the integer sum of the two operands.

If the sum has unsigned overflow, the result returned is the mathematical result modulo 2^n, where n is the bit width of the result.

Because LLVM integers use a two's complement representation, this instruction is appropriate for both signed and unsigned integers.

`nuw` and `nsw` stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the `nuw` and/or `nsw` keywords are present, the result value of the `add` is undefined if unsigned and/or signed overflow, respectively, occurs.

Example:

```
<result> = add i32 4, %var          ; yields {i32}:result = 4 + %var
```

'*fadd* Instruction

Syntax:

```
<result> = fadd <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

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

Arguments:

The two arguments to the '`fadd`' instruction must be floating point or vector of floating point values. Both arguments must have identical types.

Semantics:

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

Example:

```
<result> = fadd float 4.0, %var          ; yields {float}:result = 4.0 + %var
```

'*sub* Instruction

Syntax:

```
<result> = sub <ty> <op1>, <op2>          ; yields {ty}:result
<result> = sub nuw <ty> <op1>, <op2>      ; yields {ty}:result
<result> = sub nsw <ty> <op1>, <op2>      ; yields {ty}:result
<result> = sub nuw nsw <ty> <op1>, <op2>  ; 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 integer or vector of integer values. Both arguments must have identical types.

Semantics:

The value produced is the integer difference of the two operands.

If the difference has unsigned overflow, the result returned is the mathematical result modulo 2^n, where n

is the bit width of the result.

Because LLVM integers use a two's complement representation, this instruction is appropriate for both signed and unsigned integers.

`nuw` and `nsw` stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the `nuw` and/or `nsw` keywords are present, the result value of the `sub` is undefined if unsigned and/or signed overflow, respectively, occurs.

Example:

```
<result> = sub i32 4, %var          ; yields {i32}:result = 4 - %var
<result> = sub i32 0, %val          ; yields {i32}:result = -%var
```

'`fsub`' Instruction

Syntax:

```
<result> = fsub <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

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

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

Arguments:

The two arguments to the '`fsub`' instruction must be floating point or vector of floating point values. Both arguments must have identical types.

Semantics:

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

Example:

```
<result> = fsub float 4.0, %var           ; yields {float}:result = 4.0 - %var
<result> = fsub float -0.0, %val          ; yields {float}:result = -%var
```

'`mul`' Instruction

Syntax:

```
<result> = mul <ty> <op1>, <op2>          ; yields {ty}:result
<result> = mul nuw <ty> <op1>, <op2>      ; yields {ty}:result
<result> = mul nsw <ty> <op1>, <op2>      ; yields {ty}:result
<result> = mul nuw nsw <ty> <op1>, <op2>  ; yields {ty}:result
```

Overview:

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

Arguments:

The two arguments to the '`mul`' instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

The value produced is the integer product of the two operands.

If the result of the multiplication has unsigned overflow, the result returned is the mathematical result modulo 2^n, where n is the bit width of the result.

Because LLVM integers use a two's complement representation, and the result is the same width as the operands, this instruction returns the correct result for both signed and unsigned integers. If a full product (e.g. `i32`x`i32`->`i64`) is needed, the operands should be sign-extended or zero-extended as appropriate to the width of the full product.

`nuw` and `nsw` stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the `nuw` and/or `nsw` keywords are present, the result value of the `mul` is undefined if unsigned and/or signed overflow, respectively, occurs.

Example:

```
<result> = mul i32 4, %var          ; yields {i32}:result = 4 * %var
```

'`fmul`' Instruction

Syntax:

```
<result> = fmul <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

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

Arguments:

The two arguments to the '`fmul`' instruction must be floating point or vector of floating point values. Both arguments must have identical types.

Semantics:

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

Example:

```
<result> = fmul float 4.0, %var          ; yields {float}:result = 4.0 * %var
```

'`udiv`' Instruction

Syntax:

```
<result> = udiv <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

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

Arguments:

The two arguments to the '`udiv`' instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

The value produced is the unsigned integer quotient of the two operands.

Note that unsigned integer division and signed integer division are distinct operations; for signed integer division, use '`sdiv`'.

Division by zero leads to undefined behavior.

Example:

```
<result> = udiv i32 4, %var          ; yields {i32}:result = 4 / %var
```

'*sdiv*' Instruction

Syntax:

```
<result> = sdiv <ty> <op1>, <op2>         ; yields {ty}:result
<result> = sdiv exact <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

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

Arguments:

The two arguments to the '`sdiv`' instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

The value produced is the signed integer quotient of the two operands rounded towards zero.

Note that signed integer division and unsigned integer division are distinct operations; for unsigned integer division, use '`udiv`'.

Division by zero leads to undefined behavior. Overflow also leads to undefined behavior; this is a rare case, but can occur, for example, by doing a 32-bit division of -2147483648 by -1.

If the `exact` keyword is present, the result value of the `sdiv` is undefined if the result would be rounded or if overflow would occur.

Example:

```
<result> = sdiv i32 4, %var          ; yields {i32}:result = 4 / %var
```

'*fdiv*' Instruction

Syntax:

```
<result> = fdiv <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

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

Arguments:

The two arguments to the '`fdiv`' instruction must be floating point or vector of floating point values. Both arguments must have identical types.

Semantics:

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

Example:

```
<result> = fdiv float 4.0, %var          ; yields {float}:result = 4.0 / %var
```

'*urem*' Instruction

Syntax:

```
<result> = urem <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

The '`urem`' instruction returns the remainder from the unsigned division of its two arguments.

Arguments:

The two arguments to the '`urem`' instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

This instruction returns the unsigned integer *remainder* of a division. This instruction always performs an unsigned division to get the remainder.

Note that unsigned integer remainder and signed integer remainder are distinct operations; for signed integer remainder, use '`srem`'.

Taking the remainder of a division by zero leads to undefined behavior.

Example:

```
<result> = urem i32 4, %var          ; yields {i32}:result = 4 % %var
```

'*srem*' Instruction

Syntax:

```
<result> = srem <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

The '`srem`' instruction returns the remainder from the signed division of its two operands. This instruction can also take vector versions of the values in which case the elements must be integers.

Arguments:

The two arguments to the '`srem`' instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

This instruction returns the *remainder* of a division (where the result has the same sign as the dividend, `op1`), not the *modulo* operator (where the result has the same sign as the divisor, `op2`) of a value. For more information about the difference, see The Math Forum. For a table of how this is implemented in various languages, please see Wikipedia: modulo operation.

Note that signed integer remainder and unsigned integer remainder are distinct operations; for unsigned integer remainder, use '`urem`'.

Taking the remainder of a division by zero leads to undefined behavior. Overflow also leads to undefined

behavior; this is a rare case, but can occur, for example, by taking the remainder of a 32-bit division of -2147483648 by -1. (The remainder doesn't actually overflow, but this rule lets srem be implemented using instructions that return both the result of the division and the remainder.)

Example:

```
<result> = srem i32 4, %var          ; yields {i32}:result = 4 % %var
```

'*frem* Instruction

Syntax:

```
<result> = frem <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

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

Arguments:

The two arguments to the '`frem`' instruction must be floating point or vector of floating point values. Both arguments must have identical types.

Semantics:

This instruction returns the *remainder* of a division. The remainder has the same sign as the dividend.

Example:

```
<result> = frem float 4.0, %var          ; yields {float}:result = 4.0 % %var
```

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 of the same type, execute an operation on them, and produce a single value. The resulting value is the same type as its operands.

'*shl* Instruction

Syntax:

```
<result> = shl <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

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

Arguments:

Both arguments to the '`shl`' instruction must be the same integer or vector of integer type. '`op2`' is treated as an unsigned value.

Semantics:

The value produced is `op1` * 2^{op2} mod 2^n, where `n` is the width of the result. If `op2` is (statically or dynamically) negative or equal to or larger than the number of bits in `op1`, the result is undefined. If the arguments are vectors, each vector element of `op1` is shifted by the corresponding shift amount in `op2`.

Example:

```
<result> = shl i32 4, %var   ; yields {i32}: 4 << %var
<result> = shl i32 4, 2      ; yields {i32}: 16
<result> = shl i32 1, 10     ; yields {i32}: 1024
<result> = shl i32 1, 32     ; undefined
<result> = shl <2 x i32> < i32 1, i32 1>, < i32 1, i32 2>   ; yields: result=<2 x i32> < i32 2, i32 4>
```

'*lshr*' *Instruction*

Syntax:

```
<result> = lshr <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

The '`lshr`' instruction (logical shift right) returns the first operand shifted to the right a specified number of bits with zero fill.

Arguments:

Both arguments to the '`lshr`' instruction must be the same integer or vector of integer type. '`op2`' is treated as an unsigned value.

Semantics:

This instruction always performs a logical shift right operation. The most significant bits of the result will be filled with zero bits after the shift. If `op2` is (statically or dynamically) equal to or larger than the number of bits in `op1`, the result is undefined. If the arguments are vectors, each vector element of `op1` is shifted by the corresponding shift amount in `op2`.

Example:

```
<result> = lshr i32 4, 1   ; yields {i32}:result = 2
<result> = lshr i32 4, 2   ; yields {i32}:result = 1
<result> = lshr i8  4, 3   ; yields {i8}:result = 0
<result> = lshr i8 -2, 1   ; yields {i8}:result = 0x7FFFFFFF
<result> = lshr i32 1, 32  ; undefined
<result> = lshr <2 x i32> < i32 -2, i32 4>, < i32 1, i32 2>   ; yields: result=<2 x i32> < i32 0x7FFFFFFF, i32 1>
```

'*ashr*' *Instruction*

Syntax:

```
<result> = ashr <ty> <op1>, <op2>   ; yields {ty}:result
```

Overview:

The '`ashr`' instruction (arithmetic shift right) returns the first operand shifted to the right a specified number of bits with sign extension.

Arguments:

Both arguments to the '`ashr`' instruction must be the same integer or vector of integer type. '`op2`' is treated as an unsigned value.

Semantics:

This instruction always performs an arithmetic shift right operation, The most significant bits of the result will be filled with the sign bit of `op1`. If `op2` is (statically or dynamically) equal to or larger than the number of bits in `op1`, the result is undefined. If the arguments are vectors, each vector element of `op1` is shifted by

the corresponding shift amount in `op2`.

Example:

```
<result> = ashr i32 4, 1   ; yields {i32}:result = 2
<result> = ashr i32 4, 2   ; yields {i32}:result = 1
<result> = ashr i8  4, 3   ; yields {i8}:result = 0
<result> = ashr i8 -2, 1   ; yields {i8}:result = -1
<result> = ashr i32 1, 32  ; undefined
<result> = ashr <2 x i32> < i32 -2, i32 4>, < i32 1, i32 3>   ; yields: result=<2 x i32> < i32 -1, i32 0>
```

'*and*' Instruction

Syntax:

```
<result> = and <ty> <op1>, <op2>   ; 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 integer or vector of integer 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 i32 4, %var         ; yields {i32}:result = 4 & %var
<result> = and i32 15, 40          ; yields {i32}:result = 8
<result> = and i32 4, 8            ; yields {i32}:result = 0
```

'*or*' Instruction

Syntax:

```
<result> = or <ty> <op1>, <op2>   ; 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 integer or vector of integer 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 i32 4, %var         ; yields {i32}:result = 4 | %var
<result> = or i32 15, 40          ; yields {i32}:result = 47
<result> = or i32 4, 8            ; yields {i32}:result = 12
```

'*xor*' *Instruction*

Syntax:

```
<result> = xor <ty> <op1>, <op2>   ; 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 integer or vector of integer 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 i32 4, %var         ; yields {i32}:result = 4 ^ %var
<result> = xor i32 15, 40          ; yields {i32}:result = 39
<result> = xor i32 4, 8            ; yields {i32}:result = 12
<result> = xor i32 %V, -1          ; yields {i32}:result = ~%V
```

Vector Operations

LLVM supports several instructions to represent vector operations in a target-independent manner. These instructions cover the element-access and vector-specific operations needed to process vectors effectively. While LLVM does directly support these vector operations, many sophisticated algorithms will want to use target-specific intrinsics to take full advantage of a specific target.

'*extractelement*' *Instruction*

Syntax:

```
<result> = extractelement <n x <ty>> <val>, i32 <idx>    ; yields <ty>
```

Overview:

The '`extractelement`' instruction extracts a single scalar element from a vector at a specified index.

Arguments:

The first operand of an '`extractelement`' instruction is a value of vector type. The second operand is an index indicating the position from which to extract the element. The index may be a variable.

Semantics:

The result is a scalar of the same type as the element type of `val`. Its value is the value at position `idx` of `val`. If `idx` exceeds the length of `val`, the results are undefined.

Example:

```
%result = extractelement <4 x i32> %vec, i32 0    ; yields i32
```

'*insertelement*' Instruction

Syntax:

```
<result> = insertelement <n x <ty>> <val>, <ty> <elt>, i32 <idx>    ; yields <n x <ty>>
```

Overview:

The '`insertelement`' instruction inserts a scalar element into a vector at a specified index.

Arguments:

The first operand of an '`insertelement`' instruction is a value of vector type. The second operand is a scalar value whose type must equal the element type of the first operand. The third operand is an index indicating the position at which to insert the value. The index may be a variable.

Semantics:

The result is a vector of the same type as `val`. Its element values are those of `val` except at position `idx`, where it gets the value `elt`. If `idx` exceeds the length of `val`, the results are undefined.

Example:

```
%result = insertelement <4 x i32> %vec, i32 1, i32 0    ; yields <4 x i32>
```

'*shufflevector*' Instruction

Syntax:

```
<result> = shufflevector <n x <ty>> <v1>, <n x <ty>> <v2>, <m x i32> <mask>    ; yields <m x <ty>>
```

Overview:

The '`shufflevector`' instruction constructs a permutation of elements from two input vectors, returning a vector with the same element type as the input and length that is the same as the shuffle mask.

Arguments:

The first two operands of a '`shufflevector`' instruction are vectors with types that match each other. The third argument is a shuffle mask whose element type is always 'i32'. The result of the instruction is a vector whose length is the same as the shuffle mask and whose element type is the same as the element type of the first two operands.

The shuffle mask operand is required to be a constant vector with either constant integer or undef values.

Semantics:

The elements of the two input vectors are numbered from left to right across both of the vectors. The shuffle mask operand specifies, for each element of the result vector, which element of the two input vectors the result element gets. The element selector may be undef (meaning "don't care") and the second operand may be undef if performing a shuffle from only one vector.

Example:

```
%result = shufflevector <4 x i32> %v1, <4 x i32> %v2,
                        <4 x i32> <i32 0, i32 4, i32 1, i32 5>  ; yields <4 x i32>
%result = shufflevector <4 x i32> %v1, <4 x i32> undef,
                        <4 x i32> <i32 0, i32 1, i32 2, i32 3>  ; yields <4 x i32> - Identity shuffle.
%result = shufflevector <8 x i32> %v1, <8 x i32> undef,
                        <4 x i32> <i32 0, i32 1, i32 2, i32 3>  ; yields <4 x i32>
%result = shufflevector <4 x i32> %v1, <4 x i32> %v2,
                        <8 x i32> <i32 0, i32 1, i32 2, i32 3, i32 4, i32 5, i32 6, i32 7 >  ; yields <8 x i32>
```

Aggregate Operations

LLVM supports several instructions for working with aggregate values.

'*extractvalue*' *Instruction*

Syntax:

```
<result> = extractvalue <aggregate type> <val>, <idx>{, <idx>}*
```

Overview:

The '`extractvalue`' instruction extracts the value of a struct field or array element from an aggregate value.

Arguments:

The first operand of an '`extractvalue`' instruction is a value of struct or array type. The operands are constant indices to specify which value to extract in a similar manner as indices in a '`getelementptr`' instruction.

Semantics:

The result is the value at the position in the aggregate specified by the index operands.

Example:

```
%result = extractvalue {i32, float} %agg, 0    ; yields i32
```

'*insertvalue*' *Instruction*

Syntax:

```
<result> = insertvalue <aggregate type> <val>, <ty> <val>, <idx>    ; yields <n x <ty>>
```

Overview:

The '`insertvalue`' instruction inserts a value into a struct field or array element in an aggregate.

Arguments:

The first operand of an '`insertvalue`' instruction is a value of struct or array type. The second operand is a first-class value to insert. The following operands are constant indices indicating the position at which to insert the value in a similar manner as indices in a '`getelementptr`' instruction. The value to insert must have the same type as the value identified by the indices.

Semantics:

The result is an aggregate of the same type as `val`. Its value is that of `val` except that the value at the position specified by the indices is that of `elt`.

Example:

```
%result = insertvalue {i32, float} %agg, i32 1, 0    ; yields {i32, float}
```

Memory Access and Addressing 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>[, i32 <NumElements>][, align <alignment>]     ; yields {type*}:result
```

Overview:

The '`malloc`' instruction allocates memory from the system heap and returns a pointer to it. The object is always allocated in the generic address space (address space zero).

Arguments:

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. If "NumElements" is specified, it is the number of elements allocated, otherwise "NumElements" is defaulted to be one. If a constant alignment is specified, the value result of the allocation is guaranteed to be aligned to at least that boundary. If not specified, or if zero, the target can choose to align the allocation on any convenient boundary compatible with the type.

'`type`' must be a sized type.

Semantics:

Memory is allocated using the system "`malloc`" function, and a pointer is returned. The result of a zero byte allocation is undefined. The result is null if there is insufficient memory available.

Example:

```
%array  = malloc [4 x i8]                    ; yields {[%4 x i8]*}:array

%size   = add i32 2, 2                       ; yields {i32}:size = i32 4
%array1 = malloc i8, i32 4                   ; yields {i8*}:array1
%array2 = malloc [12 x i8], i32 %size        ; yields {[12 x i8]*}:array2
%array3 = malloc i32, i32 4, align 1024      ; yields {i32*}:array3
%array4 = malloc i32, align 1024             ; yields {i32*}:array4
```

Note that the code generator does not yet respect the alignment value.

'*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 no longer defined after this instruction executes. If the pointer is null, the operation is a noop.

Example:

```
%array  = malloc [4 x i8]                     ; yields {[4 x i8]*}:array
          free   [4 x i8]* %array
```

'*alloca*' *Instruction*

Syntax:

```
<result> = alloca <type>[, i32 <NumElements>][, align <alignment>]     ; yields {type*}:result
```

Overview:

The '`alloca`' instruction allocates memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller. The object is always allocated in the generic address space (address space zero).

Arguments:

The '`alloca`' instruction allocates `sizeof(<type>)*NumElements` bytes of memory on the runtime stack, returning a pointer of the appropriate type to the program. If "NumElements" is specified, it is the number of elements allocated, otherwise "NumElements" is defaulted to be one. If a constant alignment is specified, the value result of the allocation is guaranteed to be aligned to at least that boundary. If not specified, or if zero, the target can choose to align the allocation on any convenient boundary compatible with the type.

'`type`' may be any sized type.

Semantics:

Memory is allocated; a pointer is returned. The operation is undefined if there is insufficient stack space for the allocation. '`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 `unwind` instructions), the memory is reclaimed. Allocating zero bytes is legal, but the result is undefined.

Example:

```
%ptr = alloca i32                               ; yields {i32*}:ptr
%ptr = alloca i32, i32 4                        ; yields {i32*}:ptr
%ptr = alloca i32, i32 4, align 1024            ; yields {i32*}:ptr
%ptr = alloca i32, align 1024                   ; yields {i32*}:ptr
```

'`load`' Instruction

Syntax:

```
<result> = load <ty>* <pointer>[, align <alignment>]
<result> = volatile load <ty>* <pointer>[, align <alignment>]
```

Overview:

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

Arguments:

The argument to the '`load`' instruction specifies the memory address from which to load. 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.

The optional constant "align" argument specifies the alignment of the operation (that is, the alignment of the memory address). A value of 0 or an omitted "align" argument means that the operation has the preferential alignment for the target. It is the responsibility of the code emitter to ensure that the alignment information is correct. Overestimating the alignment results in an undefined behavior. Underestimating the alignment may produce less efficient code. An alignment of 1 is always safe.

Semantics:

The location of memory pointed to is loaded. If the value being loaded is of scalar type then the number of bytes read does not exceed the minimum number of bytes needed to hold all bits of the type. For example, loading an `i24` reads at most three bytes. When loading a value of a type like `i20` with a size that is not an integral number of bytes, the result is undefined if the value was not originally written using a store of the same type.

Examples:

```
%ptr = alloca i32                               ; yields {i32*}:ptr
store i32 3, i32* %ptr                          ; yields {void}
%val = load i32* %ptr                           ; yields {i32}:val = i32 3
```

'`store`' Instruction

Syntax:

```
store <ty> <value>, <ty>* <pointer>[, align <alignment>]                    ; yields {void}
volatile store <ty> <value>, <ty>* <pointer>[, align <alignment>]           ; 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 at which to store it. The type of the '`<pointer>`' operand must be a pointer to the first class 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.

The optional constant "align" argument specifies the alignment of the operation (that is, the alignment of

the memory address). A value of 0 or an omitted "align" argument means that the operation has the preferential alignment for the target. It is the responsibility of the code emitter to ensure that the alignment information is correct. Overestimating the alignment results in an undefined behavior. Underestimating the alignment may produce less efficient code. An alignment of 1 is always safe.

Semantics:

The contents of memory are updated to contain '`<value>`' at the location specified by the '`<pointer>`' operand. If '`<value>`' is of scalar type then the number of bytes written does not exceed the minimum number of bytes needed to hold all bits of the type. For example, storing an `i24` writes at most three bytes. When writing a value of a type like `i20` with a size that is not an integral number of bytes, it is unspecified what happens to the extra bits that do not belong to the type, but they will typically be overwritten.

Example:

```
%ptr = alloca i32                               ; yields {i32*}:ptr
store i32 3, i32* %ptr                          ; yields {void}
%val = load i32* %ptr                           ; yields {i32}:val = i32 3
```

'*getelementptr* Instruction'

Syntax:

```
<result> = getelementptr <pty>* <ptrval>{, <ty> <idx>}*
<result> = getelementptr inbounds <pty>* <ptrval>{, <ty> <idx>}*
```

Overview:

The '`getelementptr`' instruction is used to get the address of a subelement of an aggregate data structure. It performs address calculation only and does not access memory.

Arguments:

The first argument is always a pointer, and forms the basis of the calculation. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. The interpretation of each index is dependent on the type being indexed into. The first index always indexes the pointer value given as the first argument, the second index indexes a value of the type pointed to (not necessarily the value directly pointed to, since the first index can be non-zero), etc. The first type indexed into must be a pointer value, subsequent types can be arrays, vectors and structs. Note that subsequent types being indexed into can never be pointers, since that would require loading the pointer before continuing calculation.

The type of each index argument depends on the type it is indexing into. When indexing into a (optionally packed) structure, only `i32` integer **constants** are allowed. When indexing into an array, pointer or vector, integers of any width are allowed, and they are not required to be constant.

For example, let's 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 { i8 , [10 x [20 x i32]], i8  }
%ST = type { i32, double, %RT }

define i32* @foo(%ST* %s) {
entry:
  %reg = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
  ret i32* %reg
}
```

Semantics:

In the example above, the first index is indexing into the '`%ST*`' type, which is a pointer, yielding a '`%ST` = '`{ i32, double, %RT }`' type, a structure. The second index indexes into the third element of the structure, yielding a '`%RT`' = '`{ i8 , [10 x [20 x i32]], i8 }`' type, another structure. The third index indexes into the second element of the structure, yielding a '`[10 x [20 x i32]]`' type, an array. The two dimensions of the array are subscripted into, yielding an '`i32`' type. The '`getelementptr`' instruction returns a pointer to this element, thus computing a value of '`i32*`' 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:

```
define i32* @foo(%ST* %s) {
  %t1 = getelementptr %ST* %s, i32 1                        ; yields %ST*:%t1
  %t2 = getelementptr %ST* %t1, i32 0, i32 2                ; yields %RT*:%t2
  %t3 = getelementptr %RT* %t2, i32 0, i32 1                ; yields [10 x [20 x i32]]*:%t3
  %t4 = getelementptr [10 x [20 x i32]]* %t3, i32 0, i32 5  ; yields [20 x i32]*:%t4
  %t5 = getelementptr [20 x i32]* %t4, i32 0, i32 13        ; yields i32*:%t5
  ret i32* %t5
}
```

If the `inbounds` keyword is present, the result value of the `getelementptr` is undefined if the base pointer is not an *in bounds* address of an allocated object, or if any of the addresses that would be formed by successive addition of the offsets implied by the indices to the base address with infinitely precise arithmetic are not an *in bounds* address of that allocated object. The *in bounds* addresses for an allocated object are all the addresses that point into the object, plus the address one byte past the end.

If the `inbounds` keyword is not present, the offsets are added to the base address with silently-wrapping two's complement arithmetic, and the result value of the `getelementptr` may be outside the object pointed to by the base pointer. The result value may not necessarily be used to access memory though, even if it happens to point into allocated storage. See the Pointer Aliasing Rules section for more information.

The getelementptr instruction is often confusing. For some more insight into how it works, see the getelementptr FAQ.

Example:

```
; yields [12 x i8]*:aptr
%aptr = getelementptr {i32, [12 x i8]}* %saptr, i64 0, i32 1
; yields i8*:vptr
%vptr = getelementptr {i32, <2 x i8>}* %svptr, i64 0, i32 1, i32 1
; yields i8*:eptr
%eptr = getelementptr [12 x i8]* %aptr, i64 0, i32 1
; yields i32*:iptr
%iptr = getelementptr [10 x i32]* @arr, i16 0, i16 0
```

Conversion Operations

The instructions in this category are the conversion instructions (casting) which all take a single operand and a type. They perform various bit conversions on the operand.

'*trunc .. to*' *Instruction*

Syntax:

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

Overview:

The '`trunc`' instruction truncates its operand to the type `ty2`.

Arguments:

The '`trunc`' instruction takes a `value` to trunc, which must be an integer type, and a type that specifies the size and type of the result, which must be an integer type. The bit size of `value` must be larger than the bit size of `ty2`. Equal sized types are not allowed.

Semantics:

The '`trunc`' instruction truncates the high order bits in `value` and converts the remaining bits to `ty2`. Since the source size must be larger than the destination size, `trunc` cannot be a *no-op cast*. It will always truncate bits.

Example:

```
%X = trunc i32 257 to i8              ; yields i8:1
%Y = trunc i32 123 to i1              ; yields i1:true
%Y = trunc i32 122 to i1              ; yields i1:false
```

'`zext .. to`' Instruction

Syntax:

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

Overview:

The '`zext`' instruction zero extends its operand to type `ty2`.

Arguments:

The '`zext`' instruction takes a value to cast, which must be of integer type, and a type to cast it to, which must also be of integer type. The bit size of the `value` must be smaller than the bit size of the destination type, `ty2`.

Semantics:

The `zext` fills the high order bits of the `value` with zero bits until it reaches the size of the destination type, `ty2`.

When zero extending from i1, the result will always be either 0 or 1.

Example:

```
%X = zext i32 257 to i64              ; yields i64:257
%Y = zext i1 true to i32              ; yields i32:1
```

'`sext .. to`' Instruction

Syntax:

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

Overview:

The '`sext`' sign extends `value` to the type `ty2`.

Arguments:

The '`sext`' instruction takes a value to cast, which must be of integer type, and a type to cast it to, which must also be of integer type. The bit size of the `value` must be smaller than the bit size of the destination type, `ty2`.

Semantics:

The '`sext`' instruction performs a sign extension by copying the sign bit (highest order bit) of the `value` until it reaches the bit size of the type `ty2`.

When sign extending from i1, the extension always results in -1 or 0.

Example:

```
%X = sext i8  -1 to i16              ; yields i16   :65535
%Y = sext i1 true to i32             ; yields i32:-1
```

'`fptrunc .. to`' Instruction

Syntax:

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

Overview:

The '`fptrunc`' instruction truncates `value` to type `ty2`.

Arguments:

The '`fptrunc`' instruction takes a floating point value to cast and a floating point type to cast it to. The size of `value` must be larger than the size of `ty2`. This implies that `fptrunc` cannot be used to make a *no-op cast*.

Semantics:

The '`fptrunc`' instruction truncates a `value` from a larger floating point type to a smaller floating point type. If the value cannot fit within the destination type, `ty2`, then the results are undefined.

Example:

```
%X = fptrunc double 123.0 to float         ; yields float:123.0
%Y = fptrunc double 1.0E+300 to float      ; yields undefined
```

'`fpext .. to`' Instruction

Syntax:

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

Overview:

The '`fpext`' extends a floating point `value` to a larger floating point value.

Arguments:

The '`fpext`' instruction takes a floating point `value` to cast, and a floating point type to cast it to. The source type must be smaller than the destination type.

Semantics:

The '`fpext`' instruction extends the `value` from a smaller floating point type to a larger floating point type. The `fpext` cannot be used to make a *no-op cast* because it always changes bits. Use `bitcast` to make a *no-op cast* for a floating point cast.

Example:

```
%X = fpext float 3.1415 to double        ; yields double:3.1415
%Y = fpext float 1.0 to float            ; yields float:1.0 (no-op)
```

'*fptoui .. to*' Instruction

Syntax:

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

Overview:

The '`fptoui`' converts a floating point `value` to its unsigned integer equivalent of type `ty2`.

Arguments:

The '`fptoui`' instruction takes a value to cast, which must be a scalar or vector floating point value, and a type to cast it to `ty2`, which must be an integer type. If `ty` is a vector floating point type, `ty2` must be a vector integer type with the same number of elements as `ty`

Semantics:

The '`fptoui`' instruction converts its floating point operand into the nearest (rounding towards zero) unsigned integer value. If the value cannot fit in `ty2`, the results are undefined.

Example:

```
%X = fptoui double 123.0 to i32      ; yields i32:123
%Y = fptoui float 1.0E+300 to i1     ; yields undefined:1
%X = fptoui float 1.04E+17 to i8     ; yields undefined:1
```

'*fptosi .. to*' Instruction

Syntax:

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

Overview:

The '`fptosi`' instruction converts floating point `value` to type `ty2`.

Arguments:

The '`fptosi`' instruction takes a value to cast, which must be a scalar or vector floating point value, and a type to cast it to `ty2`, which must be an integer type. If `ty` is a vector floating point type, `ty2` must be a vector integer type with the same number of elements as `ty`

Semantics:

The '`fptosi`' instruction converts its floating point operand into the nearest (rounding towards zero) signed integer value. If the value cannot fit in `ty2`, the results are undefined.

Example:

```
%X = fptosi double -123.0 to i32      ; yields i32:-123
%Y = fptosi float 1.0E-247 to i1      ; yields undefined:1
%X = fptosi float 1.04E+17 to i8      ; yields undefined:1
```

'`uitofp .. to`' Instruction

Syntax:

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

Overview:

The '`uitofp`' instruction regards `value` as an unsigned integer and converts that value to the `ty2` type.

Arguments:

The '`uitofp`' instruction takes a value to cast, which must be a scalar or vector integer value, and a type to cast it to `ty2`, which must be an floating point type. If `ty` is a vector integer type, `ty2` must be a vector floating point type with the same number of elements as `ty`

Semantics:

The '`uitofp`' instruction interprets its operand as an unsigned integer quantity and converts it to the corresponding floating point value. If the value cannot fit in the floating point value, the results are undefined.

Example:

```
%X = uitofp i32 257 to float         ; yields float:257.0
%Y = uitofp i8 -1 to double          ; yields double:255.0
```

'`sitofp .. to`' Instruction

Syntax:

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

Overview:

The '`sitofp`' instruction regards `value` as a signed integer and converts that value to the `ty2` type.

Arguments:

The '`sitofp`' instruction takes a value to cast, which must be a scalar or vector integer value, and a type to cast it to `ty2`, which must be an floating point type. If `ty` is a vector integer type, `ty2` must be a vector floating point type with the same number of elements as `ty`

Semantics:

The '`sitofp`' instruction interprets its operand as a signed integer quantity and converts it to the corresponding floating point value. If the value cannot fit in the floating point value, the results are undefined.

Example:

```
%X = sitofp i32 257 to float         ; yields float:257.0
%Y = sitofp i8 -1 to double          ; yields double:-1.0
```

'*ptrtoint .. to*' Instruction

Syntax:

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

Overview:

The '`ptrtoint`' instruction converts the pointer `value` to the integer type `ty2`.

Arguments:

The '`ptrtoint`' instruction takes a `value` to cast, which must be a pointer value, and a type to cast it to `ty2`, which must be an integer type.

Semantics:

The '`ptrtoint`' instruction converts `value` to integer type `ty2` by interpreting the pointer value as an integer and either truncating or zero extending that value to the size of the integer type. If `value` is smaller than `ty2` then a zero extension is done. If `value` is larger than `ty2` then a truncation is done. If they are the same size, then nothing is done (*no-op cast*) other than a type change.

Example:

```
%X = ptrtoint i32* %X to i8           ; yields truncation on 32-bit architecture
%Y = ptrtoint i32* %x to i64          ; yields zero extension on 32-bit architecture
```

'*inttoptr .. to*' Instruction

Syntax:

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

Overview:

The '`inttoptr`' instruction converts an integer `value` to a pointer type, `ty2`.

Arguments:

The '`inttoptr`' instruction takes an integer value to cast, and a type to cast it to, which must be a pointer type.

Semantics:

The '`inttoptr`' instruction converts `value` to type `ty2` by applying either a zero extension or a truncation depending on the size of the integer `value`. If `value` is larger than the size of a pointer then a truncation is done. If `value` is smaller than the size of a pointer then a zero extension is done. If they are the same size, nothing is done (*no-op cast*).

Example:

```
%X = inttoptr i32 255 to i32*          ; yields zero extension on 64-bit architecture
%X = inttoptr i32 255 to i32*          ; yields no-op on 32-bit architecture
%Y = inttoptr i64 0 to i32*            ; yields truncation on 32-bit architecture
```

'*bitcast .. to*' Instruction

Syntax:

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

Overview:

The '`bitcast`' instruction converts `value` to type `ty2` without changing any bits.

Arguments:

The '`bitcast`' instruction takes a value to cast, which must be a non-aggregate first class value, and a type to cast it to, which must also be a non-aggregate first class type. The bit sizes of `value` and the destination type, `ty2`, must be identical. If the source type is a pointer, the destination type must also be a pointer. This instruction supports bitwise conversion of vectors to integers and to vectors of other types (as long as they have the same size).

Semantics:

The '`bitcast`' instruction converts `value` to type `ty2`. It is always a *no-op cast* because no bits change with this conversion. The conversion is done as if the `value` had been stored to memory and read back as type `ty2`. Pointer types may only be converted to other pointer types with this instruction. To convert pointers to other types, use the inttoptr or ptrtoint instructions first.

Example:

```
%X = bitcast i8 255 to i8             ; yields i8 :-1
%Y = bitcast i32* %x to sint*         ; yields sint*:%x
%Z = bitcast <2 x int> %V to i64;     ; yields i64: %V
```

Other Operations

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

'*icmp* Instruction

Syntax:

```
<result> = icmp <cond> <ty> <op1>, <op2>   ; yields {i1} or {<N x i1>}:result
```

Overview:

The '`icmp`' instruction returns a boolean value or a vector of boolean values based on comparison of its two integer, integer vector, or pointer operands.

Arguments:

The '`icmp`' instruction takes three operands. The first operand is the condition code indicating the kind of comparison to perform. It is not a value, just a keyword. The possible condition code are:

1. `eq`: equal
2. `ne`: not equal
3. `ugt`: unsigned greater than
4. `uge`: unsigned greater or equal
5. `ult`: unsigned less than
6. `ule`: unsigned less or equal
7. `sgt`: signed greater than
8. `sge`: signed greater or equal
9. `slt`: signed less than
10. `sle`: signed less or equal

The remaining two arguments must be integer or pointer or integer vector typed. They must also be identical types.

Semantics:

The '`icmp`' compares `op1` and `op2` according to the condition code given as `cond`. The comparison performed always yields either an `i1` or vector of `i1` result, as follows:

1. `eq`: yields `true` if the operands are equal, `false` otherwise. No sign interpretation is necessary or performed.
2. `ne`: yields `true` if the operands are unequal, `false` otherwise. No sign interpretation is necessary or performed.
3. `ugt`: interprets the operands as unsigned values and yields `true` if `op1` is greater than `op2`.
4. `uge`: interprets the operands as unsigned values and yields `true` if `op1` is greater than or equal to `op2`.
5. `ult`: interprets the operands as unsigned values and yields `true` if `op1` is less than `op2`.
6. `ule`: interprets the operands as unsigned values and yields `true` if `op1` is less than or equal to `op2`.
7. `sgt`: interprets the operands as signed values and yields `true` if `op1` is greater than `op2`.
8. `sge`: interprets the operands as signed values and yields `true` if `op1` is greater than or equal to `op2`.
9. `slt`: interprets the operands as signed values and yields `true` if `op1` is less than `op2`.
10. `sle`: interprets the operands as signed values and yields `true` if `op1` is less than or equal to `op2`.

If the operands are pointer typed, the pointer values are compared as if they were integers.

If the operands are integer vectors, then they are compared element by element. The result is an `i1` vector with the same number of elements as the values being compared. Otherwise, the result is an `i1`.

Example:

```
<result> = icmp eq i32 4, 5          ; yields: result=false
<result> = icmp ne float* %X, %X     ; yields: result=false
<result> = icmp ult i16  4, 5        ; yields: result=true
<result> = icmp sgt i16  4, 5        ; yields: result=false
<result> = icmp ule i16 -4, 5        ; yields: result=false
<result> = icmp sge i16  4, 5        ; yields: result=false
```

Note that the code generator does not yet support vector types with the `icmp` instruction.

'*fcmp*' Instruction

Syntax:

```
<result> = fcmp <cond> <ty> <op1>, <op2>     ; yields {i1} or {<N x i1>}:result
```

Overview:

The '`fcmp`' instruction returns a boolean value or vector of boolean values based on comparison of its operands.

If the operands are floating point scalars, then the result type is a boolean (`i1`).

If the operands are floating point vectors, then the result type is a vector of boolean with the same number of elements as the operands being compared.

Arguments:

The '`fcmp`' instruction takes three operands. The first operand is the condition code indicating the kind of comparison to perform. It is not a value, just a keyword. The possible condition code are:

1. `false`: no comparison, always returns false
2. `oeq`: ordered and equal
3. `ogt`: ordered and greater than
4. `oge`: ordered and greater than or equal

5. `olt`: ordered and less than
6. `ole`: ordered and less than or equal
7. `one`: ordered and not equal
8. `ord`: ordered (no nans)
9. `ueq`: unordered or equal
10. `ugt`: unordered or greater than
11. `uge`: unordered or greater than or equal
12. `ult`: unordered or less than
13. `ule`: unordered or less than or equal
14. `une`: unordered or not equal
15. `uno`: unordered (either nans)
16. `true`: no comparison, always returns true

Ordered means that neither operand is a QNAN while *unordered* means that either operand may be a QNAN.

Each of `val1` and `val2` arguments must be either a floating point type or a vector of floating point type. They must have identical types.

Semantics:

The '`fcmp`' instruction compares `op1` and `op2` according to the condition code given as `cond`. If the operands are vectors, then the vectors are compared element by element. Each comparison performed always yields an i1 result, as follows:

1. `false`: always yields `false`, regardless of operands.
2. `oeq`: yields `true` if both operands are not a QNAN and `op1` is equal to `op2`.
3. `ogt`: yields `true` if both operands are not a QNAN and `op1` is greather than `op2`.
4. `oge`: yields `true` if both operands are not a QNAN and `op1` is greater than or equal to `op2`.
5. `olt`: yields `true` if both operands are not a QNAN and `op1` is less than `op2`.
6. `ole`: yields `true` if both operands are not a QNAN and `op1` is less than or equal to `op2`.
7. `one`: yields `true` if both operands are not a QNAN and `op1` is not equal to `op2`.
8. `ord`: yields `true` if both operands are not a QNAN.
9. `ueq`: yields `true` if either operand is a QNAN or `op1` is equal to `op2`.
10. `ugt`: yields `true` if either operand is a QNAN or `op1` is greater than `op2`.
11. `uge`: yields `true` if either operand is a QNAN or `op1` is greater than or equal to `op2`.
12. `ult`: yields `true` if either operand is a QNAN or `op1` is less than `op2`.
13. `ule`: yields `true` if either operand is a QNAN or `op1` is less than or equal to `op2`.
14. `une`: yields `true` if either operand is a QNAN or `op1` is not equal to `op2`.
15. `uno`: yields `true` if either operand is a QNAN.
16. `true`: always yields `true`, regardless of operands.

Example:

```
<result> = fcmp oeq float 4.0, 5.0    ; yields: result=false
<result> = fcmp one float 4.0, 5.0    ; yields: result=true
<result> = fcmp olt float 4.0, 5.0    ; yields: result=true
<result> = fcmp ueq double 1.0, 2.0   ; yields: result=false
```

Note that the code generator does not yet support vector types with the `fcmp` instruction.

'*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 is 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. Only values of first class type may be used as the value arguments to the PHI node. Only labels may be used as the label arguments.

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.

For the purposes of the SSA form, the use of each incoming value is deemed to occur on the edge from the corresponding predecessor block to the current block (but after any definition of an '`invoke`' instruction's return value on the same edge).

Semantics:

At runtime, the '`phi`' instruction logically takes on the value specified by the pair corresponding to the predecessor basic block that executed just prior to the current block.

Example:

```
Loop:       ; Infinite loop that counts from 0 on up...
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

'*select*' *Instruction*

Syntax:

```
  <result> = select selty <cond>, <ty> <val1>, <ty> <val2>             ; yields ty

  selty is either i1 or {<N x i1>}
```

Overview:

The '`select`' instruction is used to choose one value based on a condition, without branching.

Arguments:

The '`select`' instruction requires an 'i1' value or a vector of 'i1' values indicating the condition, and two values of the same first class type. If the val1/val2 are vectors and the condition is a scalar, then entire vectors are selected, not individual elements.

Semantics:

If the condition is an i1 and it evaluates to 1, the instruction returns the first value argument; otherwise, it returns the second value argument.

If the condition is a vector of i1, then the value arguments must be vectors of the same size, and the selection is done element by element.

Example:

```
  %X = select i1 true, i8 17, i8 42          ; yields i8:17
```

Note that the code generator does not yet support conditions with vector type.

'*call*' *Instruction*

Syntax:

```
<result> = [tail] call [cconv] [ret attrs] <ty> [<fnty>*] <fnptrval>(<function args>) [fn attrs]
```

Overview:

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

Arguments:

This instruction requires several arguments:

1. The optional "tail" marker indicates whether the callee function accesses any allocas or varargs in the caller. If the "tail" marker is present, the function call is eligible for tail call optimization. Note that calls may be marked "tail" even if they do not occur before a `ret` instruction.
2. The optional "cconv" marker indicates which calling convention the call should use. If none is specified, the call defaults to using C calling conventions.
3. The optional Parameter Attributes list for return values. Only '`zeroext`', '`signext`', and '`inreg`' attributes are valid here.
4. '`ty`': the type of the call instruction itself which is also the type of the return value. Functions that return no value are marked `void`.
5. '`fnty`': shall be the signature of the pointer to function value being invoked. The argument types must match the types implied by this signature. This type can be omitted if the function is not varargs and if the function type does not return a pointer to a function.
6. '`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 value.
7. '`function args`': argument list whose types match the function signature argument types. All arguments must be of first class type. If the function signature indicates the function accepts a variable number of arguments, the extra arguments can be specified.
8. The optional function attributes list. Only '`noreturn`', '`nounwind`', '`readonly`' and '`readnone`' attributes are valid here.

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.

Example:

```
%retval = call i32 @test(i32 %argc)
call i32 (i8 *, ...)* @printf(i8 * %msg, i32 12, i8 42)      ; yields i32
%X = tail call i32 @foo()                                    ; yields i32
%Y = tail call fastcc i32 @foo()  ; yields i32
call void %foo(i8 97 signext)

%struct.A = type { i32, i8 }
%r = call %struct.A @foo()                        ; yields { 32, i8 }
%gr = extractvalue %struct.A %r, 0                ; yields i32
%gr1 = extractvalue %struct.A %r, 1               ; yields i8
%Z = call void @foo() noreturn                    ; indicates that %foo never returns normally
%ZZ = call zeroext i32 @bar()                     ; Return value is %zero extended
```

'*va_arg*' Instruction

Syntax:

```
<resultval> = va_arg <va_list*> <arglist>, <argty>
```

Overview:

The '`va_arg`' 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 `va_list*` value and the type of the argument. It returns a value of the specified argument type and increments the `va_list` to point to the next argument. The actual type of `va_list` is target specific.

Semantics:

The '`va_arg`' instruction loads an argument of the specified type from the specified `va_list` and causes the `va_list` to point to the next argument. 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.

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

Example:

See the variable argument processing section.

Note that the code generator does not yet fully support va_arg on many targets. Also, it does not currently support va_arg with aggregate types on any target.

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 intrinsics represent an extension mechanism for the LLVM language that does not require changing all of the transformations in LLVM when adding to the language (or the bitcode 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, function names may not begin with this prefix. 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 if any are added that they be documented here.

Some intrinsic functions can be overloaded, i.e., the intrinsic represents a family of functions that perform the same operation but on different data types. Because LLVM can represent over 8 million different integer types, overloading is used commonly to allow an intrinsic function to operate on any integer type. One or more of the argument types or the result type can be overloaded to accept any integer type. Argument types may also be defined as exactly matching a previous argument's type or the result type. This allows an intrinsic function which accepts multiple arguments, but needs all of them to be of the same type, to only be overloaded with respect to a single argument or the result.

Overloaded intrinsics will have the names of its overloaded argument types encoded into its function name, each preceded by a period. Only those types which are overloaded result in a name suffix. Arguments whose type is matched against another type do not. For example, the `llvm.ctpop` function can take an integer of any width and returns an integer of exactly the same integer width. This leads to a family of functions such as `i8 @llvm.ctpop.i8(i8 %val)` and `i29 @llvm.ctpop.i29(i29 %val)`. Only one type, the return type, is overloaded, and only one type suffix is required. Because the argument's type is matched against the return type, it does not require its own name suffix.

To learn how to add an intrinsic function, please see the Extending LLVM Guide.

Variable Argument Handling Intrinsics

Variable argument support is defined in LLVM with the va_arg 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 these functions regardless of the type used.

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

```
define i32 @test(i32 %X, ...) {
  ; Initialize variable argument processing
  %ap = alloca i8*
  %ap2 = bitcast i8** %ap to i8*
  call void @llvm.va_start(i8* %ap2)

  ; Read a single integer argument
  %tmp = va_arg i8** %ap, i32

  ; Demonstrate usage of llvm.va_copy and llvm.va_end
  %aq = alloca i8*
  %aq2 = bitcast i8** %aq to i8*
  call void @llvm.va_copy(i8* %aq2, i8* %ap2)
  call void @llvm.va_end(i8* %aq2)

  ; Stop processing of arguments.
  call void @llvm.va_end(i8* %ap2)
  ret i32 %tmp
}

declare void @llvm.va_start(i8*)
declare void @llvm.va_copy(i8*, i8*)
declare void @llvm.va_end(i8*)
```

'*llvm.va_start*' Intrinsic

Syntax:

```
declare void %llvm.va_start(i8* <arglist>)
```

Overview:

The 'llvm.va_start' intrinsic initializes *<arglist> for subsequent use by va_arg.

Arguments:

The argument is a pointer to a va_list element to initialize.

Semantics:

The 'llvm.va_start' intrinsic works just like the va_start macro available in C. In a target-dependent way, it initializes the va_list element to which the argument points, so that the next call to va_arg 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 as the compiler can figure that out.

'*llvm.va_end*' Intrinsic

Syntax:

```
declare void @llvm.va_end(i8* <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 pointer to 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` element to which the argument points. Calls to `llvm.va_start` and `llvm.va_copy` must be matched exactly with calls to `llvm.va_end`.

'*llvm.va_copy*' Intrinsic

Syntax:

```
declare void @llvm.va_copy(i8* <destarglist>, i8* <srcarglist>)
```

Overview:

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

Arguments:

The first argument is a pointer to a `va_list` element to initialize. The second argument is a pointer to a `va_list` element to copy from.

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 destination `va_list` element. This intrinsic is necessary because the `llvm.va_start` intrinsic may be arbitrarily complex and require, for example, memory allocation.

Accurate Garbage Collection Intrinsics

LLVM support for Accurate Garbage Collection (GC) requires the implementation and generation of these intrinsics. These intrinsics allow identification of GC roots on the stack, as well as garbage collector implementations that require read and write barriers. Front-ends for type-safe garbage collected languages should generate these intrinsics to make use of the LLVM garbage collectors. For more details, see Accurate Garbage Collection with LLVM.

The garbage collection intrinsics only operate on objects in the generic address space (address space zero).

'*llvm.gcroot*' Intrinsic

Syntax:

```
declare void @llvm.gcroot(i8** %ptrloc, i8* %metadata)
```

Overview:

The '`llvm.gcroot`' intrinsic declares the existence of a GC root to the code generator, and allows some metadata to be associated with it.

Arguments:

The first argument specifies the address of a stack object that contains the root pointer. The second pointer (which must be either a constant or a global value address) contains the meta-data to be associated with the root.

Semantics:

At runtime, a call to this intrinsic stores a null pointer into the "ptrloc" location. At compile-time, the code generator generates information to allow the runtime to find the pointer at GC safe points. The '`llvm.gcroot`' intrinsic may only be used in a function which specifies a GC algorithm.

'*llvm.gcread*' *Intrinsic*

Syntax:

```
declare i8* @llvm.gcread(i8* %ObjPtr, i8** %Ptr)
```

Overview:

The '`llvm.gcread`' intrinsic identifies reads of references from heap locations, allowing garbage collector implementations that require read barriers.

Arguments:

The second argument is the address to read from, which should be an address allocated from the garbage collector. The first object is a pointer to the start of the referenced object, if needed by the language runtime (otherwise null).

Semantics:

The '`llvm.gcread`' intrinsic has the same semantics as a load instruction, but may be replaced with substantially more complex code by the garbage collector runtime, as needed. The '`llvm.gcread`' intrinsic may only be used in a function which specifies a GC algorithm.

'*llvm.gcwrite*' *Intrinsic*

Syntax:

```
declare void @llvm.gcwrite(i8* %P1, i8* %Obj, i8** %P2)
```

Overview:

The '`llvm.gcwrite`' intrinsic identifies writes of references to heap locations, allowing garbage collector implementations that require write barriers (such as generational or reference counting collectors).

Arguments:

The first argument is the reference to store, the second is the start of the object to store it to, and the third is the address of the field of Obj to store to. If the runtime does not require a pointer to the object, Obj may be null.

Semantics:

The '`llvm.gcwrite`' intrinsic has the same semantics as a store instruction, but may be replaced with substantially more complex code by the garbage collector runtime, as needed. The '`llvm.gcwrite`' intrinsic may only be used in a function which specifies a GC algorithm.

Code Generator Intrinsics

These intrinsics are provided by LLVM to expose special features that may only be implemented with code generator support.

'`llvm.returnaddress`' Intrinsic

Syntax:

```
declare i8  *@llvm.returnaddress(i32 <level>)
```

Overview:

The '`llvm.returnaddress`' intrinsic attempts to compute a target-specific value indicating the return address of the current function or one of its callers.

Arguments:

The argument to this intrinsic indicates which function to return the address for. Zero indicates the calling function, one indicates its caller, etc. The argument is **required** to be a constant integer value.

Semantics:

The '`llvm.returnaddress`' intrinsic either returns a pointer indicating the return address of the specified call frame, or zero if it cannot be identified. The value returned by this intrinsic is likely to be incorrect or 0 for arguments other than zero, so it should only be used for debugging purposes.

Note that calling this intrinsic does not prevent function inlining or other aggressive transformations, so the value returned may not be that of the obvious source-language caller.

'`llvm.frameaddress`' Intrinsic

Syntax:

```
declare i8 *@llvm.frameaddress(i32 <level>)
```

Overview:

The '`llvm.frameaddress`' intrinsic attempts to return the target-specific frame pointer value for the specified stack frame.

Arguments:

The argument to this intrinsic indicates which function to return the frame pointer for. Zero indicates the calling function, one indicates its caller, etc. The argument is **required** to be a constant integer value.

Semantics:

The '`llvm.frameaddress`' intrinsic either returns a pointer indicating the frame address of the specified call frame, or zero if it cannot be identified. The value returned by this intrinsic is likely to be incorrect or 0 for arguments other than zero, so it should only be used for debugging purposes.

Note that calling this intrinsic does not prevent function inlining or other aggressive transformations, so the value returned may not be that of the obvious source-language caller.

'`llvm.stacksave`' Intrinsic

Syntax:

```
declare i8 *@llvm.stacksave()
```

Overview:

The '`llvm.stacksave`' intrinsic is used to remember the current state of the function stack, for use with `llvm.stackrestore`. This is useful for implementing language features like scoped automatic variable sized arrays in C99.

Semantics:

This intrinsic returns a opaque pointer value that can be passed to `llvm.stackrestore`. When an `llvm.stackrestore` intrinsic is executed with a value saved from `llvm.stacksave`, it effectively restores the state of the stack to the state it was in when the `llvm.stacksave` intrinsic executed. In practice, this pops any alloca blocks from the stack that were allocated after the `llvm.stacksave` was executed.

'*llvm.stackrestore*' Intrinsic

Syntax:

```
declare void @llvm.stackrestore(i8 * %ptr)
```

Overview:

The '`llvm.stackrestore`' intrinsic is used to restore the state of the function stack to the state it was in when the corresponding `llvm.stacksave` intrinsic executed. This is useful for implementing language features like scoped automatic variable sized arrays in C99.

Semantics:

See the description for `llvm.stacksave`.

'*llvm.prefetch*' Intrinsic

Syntax:

```
declare void @llvm.prefetch(i8* <address>, i32 <rw>, i32 <locality>)
```

Overview:

The '`llvm.prefetch`' intrinsic is a hint to the code generator to insert a prefetch instruction if supported; otherwise, it is a noop. Prefetches have no effect on the behavior of the program but can change its performance characteristics.

Arguments:

`address` is the address to be prefetched, `rw` is the specifier determining if the fetch should be for a read (0) or write (1), and `locality` is a temporal locality specifier ranging from (0) - no locality, to (3) - extremely local keep in cache. The `rw` and `locality` arguments must be constant integers.

Semantics:

This intrinsic does not modify the behavior of the program. In particular, prefetches cannot trap and do not produce a value. On targets that support this intrinsic, the prefetch can provide hints to the processor cache for better performance.

'*llvm.pcmarker*' Intrinsic

Syntax:

```
declare void @llvm.pcmarker(i32 <id>)
```

Overview:

The '`llvm.pcmarker`' intrinsic is a method to export a Program Counter (PC) in a region of code to simulators and other tools. The method is target specific, but it is expected that the marker will use exported symbols to transmit the PC of the marker. The marker makes no guarantees that it will remain with any specific instruction after optimizations. It is possible that the presence of a marker will inhibit optimizations. The intended use is to be inserted after optimizations to allow correlations of simulation runs.

Arguments:

`id` is a numerical id identifying the marker.

Semantics:

This intrinsic does not modify the behavior of the program. Backends that do not support this intrinisic may ignore it.

'`llvm.readcyclecounter`' Intrinsic

Syntax:

```
declare i64 @llvm.readcyclecounter( )
```

Overview:

The '`llvm.readcyclecounter`' intrinsic provides access to the cycle counter register (or similar low latency, high accuracy clocks) on those targets that support it. On X86, it should map to RDTSC. On Alpha, it should map to RPCC. As the backing counters overflow quickly (on the order of 9 seconds on alpha), this should only be used for small timings.

Semantics:

When directly supported, reading the cycle counter should not modify any memory. Implementations are allowed to either return a application specific value or a system wide value. On backends without support, this is lowered to a constant 0.

Standard C Library Intrinsics

LLVM provides intrinsics for a few important standard C library functions. These intrinsics allow source-language front-ends to pass information about the alignment of the pointer arguments to the code generator, providing opportunity for more efficient code generation.

'`llvm.memcpy`' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.memcpy` on any integer bit width. Not all targets support all bit widths however.

```
declare void @llvm.memcpy.i8(i8 * <dest>, i8 * <src>,
                             i8 <len>, i32 <align>)
declare void @llvm.memcpy.i16(i8 * <dest>, i8 * <src>,
                              i16 <len>, i32 <align>)
declare void @llvm.memcpy.i32(i8 * <dest>, i8 * <src>,
                              i32 <len>, i32 <align>)
declare void @llvm.memcpy.i64(i8 * <dest>, i8 * <src>,
                              i64 <len>, i32 <align>)
```

Overview:

The '`llvm.memcpy.*`' intrinsics copy a block of memory from the source location to the destination location.

Note that, unlike the standard libc function, the `llvm.memcpy.*` intrinsics do not return a value, and takes an extra alignment argument.

Arguments:

The first argument is a pointer to the destination, the second is a pointer to the source. The third argument is an integer argument specifying the number of bytes to copy, and the fourth argument is the alignment of the source and destination locations.

If the call to this intrinisic has an alignment value that is not 0 or 1, then the caller guarantees that both the source and destination pointers are aligned to that boundary.

Semantics:

The '`llvm.memcpy.*`' intrinsics copy a block of memory from the source location to the destination location, which are not allowed to overlap. It copies "len" bytes of memory over. If the argument is known to be aligned to some boundary, this can be specified as the fourth argument, otherwise it should be set to 0 or 1.

'*llvm.memmove* Intrinsic

Syntax:

This is an overloaded intrinsic. You can use llvm.memmove on any integer bit width. Not all targets support all bit widths however.

```
declare void @llvm.memmove.i8(i8 * <dest>, i8 * <src>,
                              i8 <len>, i32 <align>)
declare void @llvm.memmove.i16(i8 * <dest>, i8 * <src>,
                               i16 <len>, i32 <align>)
declare void @llvm.memmove.i32(i8 * <dest>, i8 * <src>,
                               i32 <len>, i32 <align>)
declare void @llvm.memmove.i64(i8 * <dest>, i8 * <src>,
                               i64 <len>, i32 <align>)
```

Overview:

The '`llvm.memmove.*`' intrinsics move a block of memory from the source location to the destination location. It is similar to the '`llvm.memcpy`' intrinsic but allows the two memory locations to overlap.

Note that, unlike the standard libc function, the `llvm.memmove.*` intrinsics do not return a value, and takes an extra alignment argument.

Arguments:

The first argument is a pointer to the destination, the second is a pointer to the source. The third argument is an integer argument specifying the number of bytes to copy, and the fourth argument is the alignment of the source and destination locations.

If the call to this intrinisic has an alignment value that is not 0 or 1, then the caller guarantees that the source and destination pointers are aligned to that boundary.

Semantics:

The '`llvm.memmove.*`' intrinsics copy a block of memory from the source location to the destination location, which may overlap. It copies "len" bytes of memory over. If the argument is known to be aligned to some boundary, this can be specified as the fourth argument, otherwise it should be set to 0 or 1.

'*llvm.memset.**' Intrinsics

Syntax:

This is an overloaded intrinsic. You can use llvm.memset on any integer bit width. Not all targets support all bit widths however.

```
declare void @llvm.memset.i8(i8 * <dest>, i8 <val>,
                             i8 <len>, i32 <align>)
declare void @llvm.memset.i16(i8 * <dest>, i8 <val>,
                              i16 <len>, i32 <align>)
declare void @llvm.memset.i32(i8 * <dest>, i8 <val>,
                              i32 <len>, i32 <align>)
declare void @llvm.memset.i64(i8 * <dest>, i8 <val>,
                              i64 <len>, i32 <align>)
```

Overview:

The '`llvm.memset.*`' intrinsics fill a block of memory with a particular byte value.

Note that, unlike the standard libc function, the `llvm.memset` intrinsic does not return a value, and takes an extra alignment argument.

Arguments:

The first argument is a pointer to the destination to fill, the second is the byte value to fill it with, the third argument is an integer argument specifying the number of bytes to fill, and the fourth argument is the known alignment of destination location.

If the call to this intrinisic has an alignment value that is not 0 or 1, then the caller guarantees that the destination pointer is aligned to that boundary.

Semantics:

The '`llvm.memset.*`' intrinsics fill "len" bytes of memory starting at the destination location. If the argument is known to be aligned to some boundary, this can be specified as the fourth argument, otherwise it should be set to 0 or 1.

'*llvm.sqrt.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.sqrt` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.sqrt.f32(float %Val)
declare double    @llvm.sqrt.f64(double %Val)
declare x86_fp80  @llvm.sqrt.f80(x86_fp80 %Val)
declare fp128     @llvm.sqrt.f128(fp128 %Val)
declare ppc_fp128 @llvm.sqrt.ppcf128(ppc_fp128 %Val)
```

Overview:

The '`llvm.sqrt`' intrinsics return the sqrt of the specified operand, returning the same value as the libm '`sqrt`' functions would. Unlike `sqrt` in libm, however, `llvm.sqrt` has undefined behavior for negative numbers other than -0.0 (which allows for better optimization, because there is no need to worry about errno being set). `llvm.sqrt(-0.0)` is defined to return -0.0 like IEEE sqrt.

Arguments:

The argument and return value are floating point numbers of the same type.

Semantics:

This function returns the sqrt of the specified operand if it is a nonnegative floating point number.

'*llvm.powi.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.powi` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.powi.f32(float  %Val, i32 %power)
declare double    @llvm.powi.f64(double %Val, i32 %power)
declare x86_fp80  @llvm.powi.f80(x86_fp80  %Val, i32 %power)
declare fp128     @llvm.powi.f128(fp128 %Val, i32 %power)
declare ppc_fp128 @llvm.powi.ppcf128(ppc_fp128  %Val, i32 %power)
```

Overview:

The '`llvm.powi.*`' intrinsics return the first operand raised to the specified (positive or negative) power. The order of evaluation of multiplications is not defined. When a vector of floating point type is used, the second argument remains a scalar integer value.

Arguments:

The second argument is an integer power, and the first is a value to raise to that power.

Semantics:

This function returns the first value raised to the second power with an unspecified sequence of rounding operations.

'*llvm.sin.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.sin` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.sin.f32(float  %Val)
declare double    @llvm.sin.f64(double %Val)
declare x86_fp80  @llvm.sin.f80(x86_fp80  %Val)
declare fp128     @llvm.sin.f128(fp128 %Val)
declare ppc_fp128 @llvm.sin.ppcf128(ppc_fp128  %Val)
```

Overview:

The '`llvm.sin.*`' intrinsics return the sine of the operand.

Arguments:

The argument and return value are floating point numbers of the same type.

Semantics:

This function returns the sine of the specified operand, returning the same values as the libm `sin` functions would, and handles error conditions in the same way.

'*llvm.cos.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.cos` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.cos.f32(float  %Val)
declare double    @llvm.cos.f64(double %Val)
declare x86_fp80  @llvm.cos.f80(x86_fp80  %Val)
declare fp128     @llvm.cos.f128(fp128 %Val)
declare ppc_fp128 @llvm.cos.ppcf128(ppc_fp128  %Val)
```

Overview:

The '`llvm.cos.*`' intrinsics return the cosine of the operand.

Arguments:

The argument and return value are floating point numbers of the same type.

Semantics:

This function returns the cosine of the specified operand, returning the same values as the libm `cos` functions would, and handles error conditions in the same way.

'*llvm.pow.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.pow` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.pow.f32(float  %Val, float %Power)
declare double    @llvm.pow.f64(double %Val, double %Power)
declare x86_fp80  @llvm.pow.f80(x86_fp80  %Val, x86_fp80 %Power)
declare fp128     @llvm.pow.f128(fp128 %Val, fp128 %Power)
declare ppc_fp128 @llvm.pow.ppcf128(ppc_fp128  %Val, ppc_fp128 Power)
```

Overview:

The '`llvm.pow.*`' intrinsics return the first operand raised to the specified (positive or negative) power.

Arguments:

The second argument is a floating point power, and the first is a value to raise to that power.

Semantics:

This function returns the first value raised to the second power, returning the same values as the libm `pow` functions would, and handles error conditions in the same way.

Bit Manipulation Intrinsics

LLVM provides intrinsics for a few important bit manipulation operations. These allow efficient code generation for some algorithms.

'*llvm.bswap.**' Intrinsics

Syntax:

This is an overloaded intrinsic function. You can use bswap on any integer type that is an even number of bytes (i.e. BitWidth % 16 == 0).

```
declare i16 @llvm.bswap.i16(i16 <id>)
declare i32 @llvm.bswap.i32(i32 <id>)
declare i64 @llvm.bswap.i64(i64 <id>)
```

Overview:

The '`llvm.bswap`' family of intrinsics is used to byte swap integer values with an even number of bytes (positive multiple of 16 bits). These are useful for performing operations on data that is not in the target's native byte order.

Semantics:

The `llvm.bswap.i16` intrinsic returns an i16 value that has the high and low byte of the input i16 swapped. Similarly, the `llvm.bswap.i32` intrinsic returns an i32 value that has the four bytes of the input i32 swapped, so that if the input bytes are numbered 0, 1, 2, 3 then the returned i32 will have its bytes in 3, 2, 1, 0 order. The `llvm.bswap.i48`, `llvm.bswap.i64` and other intrinsics extend this concept to additional even-byte lengths (6 bytes, 8 bytes and more, respectively).

'*llvm.ctpop.**' *Intrinsic*

Syntax:

This is an overloaded intrinsic. You can use llvm.ctpop on any integer bit width. Not all targets support all bit widths however.

```
declare i8 @llvm.ctpop.i8(i8  <src>)
declare i16 @llvm.ctpop.i16(i16 <src>)
declare i32 @llvm.ctpop.i32(i32 <src>)
declare i64 @llvm.ctpop.i64(i64 <src>)
declare i256 @llvm.ctpop.i256(i256 <src>)
```

Overview:

The '`llvm.ctpop`' family of intrinsics counts the number of bits set in a value.

Arguments:

The only argument is the value to be counted. The argument may be of any integer type. The return type must match the argument type.

Semantics:

The '`llvm.ctpop`' intrinsic counts the 1's in a variable.

'*llvm.ctlz.**' *Intrinsic*

Syntax:

This is an overloaded intrinsic. You can use `llvm.ctlz` on any integer bit width. Not all targets support all bit widths however.

```
declare i8 @llvm.ctlz.i8 (i8  <src>)
declare i16 @llvm.ctlz.i16(i16 <src>)
declare i32 @llvm.ctlz.i32(i32 <src>)
declare i64 @llvm.ctlz.i64(i64 <src>)
declare i256 @llvm.ctlz.i256(i256 <src>)
```

Overview:

The '`llvm.ctlz`' family of intrinsic functions counts the number of leading zeros in a variable.

Arguments:

The only argument is the value to be counted. The argument may be of any integer type. The return type must match the argument type.

Semantics:

The '`llvm.ctlz`' intrinsic counts the leading (most significant) zeros in a variable. If the src == 0 then the result is the size in bits of the type of src. For example, `llvm.ctlz(i32 2) = 30`.

'*llvm.cttz.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.cttz` on any integer bit width. Not all targets support all bit widths however.

```
declare i8 @llvm.cttz.i8 (i8  <src>)
declare i16 @llvm.cttz.i16(i16 <src>)
declare i32 @llvm.cttz.i32(i32 <src>)
declare i64 @llvm.cttz.i64(i64 <src>)
declare i256 @llvm.cttz.i256(i256 <src>)
```

Overview:

The '`llvm.cttz`' family of intrinsic functions counts the number of trailing zeros.

Arguments:

The only argument is the value to be counted. The argument may be of any integer type. The return type must match the argument type.

Semantics:

The '`llvm.cttz`' intrinsic counts the trailing (least significant) zeros in a variable. If the src == 0 then the result is the size in bits of the type of src. For example, `llvm.cttz(2) = 1`.

Arithmetic with Overflow Intrinsics

LLVM provides intrinsics for some arithmetic with overflow operations.

'*llvm.sadd.with.overflow.**' Intrinsics

Syntax:

This is an overloaded intrinsic. You can use `llvm.sadd.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.sadd.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.sadd.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.sadd.with.overflow.i64(i64 %a, i64 %b)
```

Overview:

The '`llvm.sadd.with.overflow`' family of intrinsic functions perform a signed addition of the two arguments, and indicate whether an overflow occurred during the signed summation.

Arguments:

The arguments (%a and %b) and the first element of the result structure may be of integer types of any

bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo signed addition.

Semantics:

The '`llvm.sadd.with.overflow`' family of intrinsic functions perform a signed addition of the two variables. They return a structure — the first element of which is the signed summation, and the second element of which is a bit specifying if the signed summation resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.sadd.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

'*llvm.uadd.with.overflow.**' Intrinsics

Syntax:

This is an overloaded intrinsic. You can use `llvm.uadd.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.uadd.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.uadd.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.uadd.with.overflow.i64(i64 %a, i64 %b)
```

Overview:

The '`llvm.uadd.with.overflow`' family of intrinsic functions perform an unsigned addition of the two arguments, and indicate whether a carry occurred during the unsigned summation.

Arguments:

The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo unsigned addition.

Semantics:

The '`llvm.uadd.with.overflow`' family of intrinsic functions perform an unsigned addition of the two arguments. They return a structure — the first element of which is the sum, and the second element of which is a bit specifying if the unsigned summation resulted in a carry.

Examples:

```
%res = call {i32, i1} @llvm.uadd.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %carry, label %normal
```

'*llvm.ssub.with.overflow.**' Intrinsics

Syntax:

This is an overloaded intrinsic. You can use `llvm.ssub.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.ssub.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.ssub.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.ssub.with.overflow.i64(i64 %a, i64 %b)
```

Overview:

The '`llvm.ssub.with.overflow`' family of intrinsic functions perform a signed subtraction of the two arguments, and indicate whether an overflow occurred during the signed subtraction.

Arguments:

The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo signed subtraction.

Semantics:

The '`llvm.ssub.with.overflow`' family of intrinsic functions perform a signed subtraction of the two arguments. They return a structure — the first element of which is the subtraction, and the second element of which is a bit specifying if the signed subtraction resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.ssub.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

'*llvm.usub.with.overflow.**' Intrinsics

Syntax:

This is an overloaded intrinsic. You can use `llvm.usub.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.usub.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.usub.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.usub.with.overflow.i64(i64 %a, i64 %b)
```

Overview:

The '`llvm.usub.with.overflow`' family of intrinsic functions perform an unsigned subtraction of the two arguments, and indicate whether an overflow occurred during the unsigned subtraction.

Arguments:

The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo unsigned subtraction.

Semantics:

The '`llvm.usub.with.overflow`' family of intrinsic functions perform an unsigned subtraction of the two arguments. They return a structure — the first element of which is the subtraction, and the second element of which is a bit specifying if the unsigned subtraction resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.usub.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

'*llvm.smul.with.overflow.**' Intrinsics

Syntax:

This is an overloaded intrinsic. You can use `llvm.smul.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.smul.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.smul.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.smul.with.overflow.i64(i64 %a, i64 %b)
```

Overview:

The '`llvm.smul.with.overflow`' family of intrinsic functions perform a signed multiplication of the two arguments, and indicate whether an overflow occurred during the signed multiplication.

Arguments:

The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo signed multiplication.

Semantics:

The '`llvm.smul.with.overflow`' family of intrinsic functions perform a signed multiplication of the two arguments. They return a structure — the first element of which is the multiplication, and the second element of which is a bit specifying if the signed multiplication resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.smul.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
br i1 %obit, label %overflow, label %normal
```

'*llvm.umul.with.overflow.**' *Intrinsics*

Syntax:

This is an overloaded intrinsic. You can use `llvm.umul.with.overflow` on any integer bit width.

```
declare {i16, i1} @llvm.umul.with.overflow.i16(i16 %a, i16 %b)
declare {i32, i1} @llvm.umul.with.overflow.i32(i32 %a, i32 %b)
declare {i64, i1} @llvm.umul.with.overflow.i64(i64 %a, i64 %b)
```

Overview:

The '`llvm.umul.with.overflow`' family of intrinsic functions perform a unsigned multiplication of the two arguments, and indicate whether an overflow occurred during the unsigned multiplication.

Arguments:

The arguments (%a and %b) and the first element of the result structure may be of integer types of any bit width, but they must have the same bit width. The second element of the result structure must be of type `i1`. `%a` and `%b` are the two values that will undergo unsigned multiplication.

Semantics:

The '`llvm.umul.with.overflow`' family of intrinsic functions perform an unsigned multiplication of the two arguments. They return a structure — the first element of which is the multiplication, and the second element of which is a bit specifying if the unsigned multiplication resulted in an overflow.

Examples:

```
%res = call {i32, i1} @llvm.umul.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%obit = extractvalue {i32, i1} %res, 1
```

```
br i1 %obit, label %overflow, label %normal
```

Debugger Intrinsics

The LLVM debugger intrinsics (which all start with `llvm.dbg.` prefix), are described in the LLVM Source Level Debugging document.

Exception Handling Intrinsics

The LLVM exception handling intrinsics (which all start with `llvm.eh.` prefix), are described in the LLVM Exception Handling document.

Trampoline Intrinsic

This intrinsic makes it possible to excise one parameter, marked with the `nest` attribute, from a function. The result is a callable function pointer lacking the nest parameter - the caller does not need to provide a value for it. Instead, the value to use is stored in advance in a "trampoline", a block of memory usually allocated on the stack, which also contains code to splice the nest value into the argument list. This is used to implement the GCC nested function address extension.

For example, if the function is `i32 f(i8* nest %c, i32 %x, i32 %y)` then the resulting function pointer has signature `i32 (i32, i32)*`. It can be created as follows:

```
%tramp = alloca [10 x i8], align 4 ; size and alignment only correct for X86
%tramp1 = getelementptr [10 x i8]* %tramp, i32 0, i32 0
%p = call i8* @llvm.init.trampoline( i8* %tramp1, i8* bitcast (i32 (i8* nest , i32, i32)* @f to i8*), i8* %nval )
%fp = bitcast i8* %p to i32 (i32, i32)*
```

The call `%val = call i32 %fp( i32 %x, i32 %y )` is then equivalent to `%val = call i32 %f( i8* %nval, i32 %x, i32 %y )`.

'*llvm.init.trampoline*' *Intrinsic*

Syntax:

```
declare i8* @llvm.init.trampoline(i8* <tramp>, i8* <func>, i8* <nval>)
```

Overview:

This fills the memory pointed to by `tramp` with code and returns a function pointer suitable for executing it.

Arguments:

The `llvm.init.trampoline` intrinsic takes three arguments, all pointers. The `tramp` argument must point to a sufficiently large and sufficiently aligned block of memory; this memory is written to by the intrinsic. Note that the size and the alignment are target-specific - LLVM currently provides no portable way of determining them, so a front-end that generates this intrinsic needs to have some target-specific knowledge. The `func` argument must hold a function bitcast to an `i8*`.

Semantics:

The block of memory pointed to by `tramp` is filled with target dependent code, turning it into a function. A pointer to this function is returned, but needs to be bitcast to an appropriate function pointer type before being called. The new function's signature is the same as that of `func` with any arguments marked with the `nest` attribute removed. At most one such `nest` argument is allowed, and it must be of pointer type. Calling the new function is equivalent to calling `func` with the same argument list, but with `nval` used for the missing `nest` argument. If, after calling `llvm.init.trampoline`, the memory pointed to by `tramp` is modified, then the effect of any later call to the returned function pointer is undefined.

Atomic Operations and Synchronization Intrinsics

These intrinsic functions expand the "universal IR" of LLVM to represent hardware constructs for atomic operations and memory synchronization. This provides an interface to the hardware, not an interface to the programmer. It is aimed at a low enough level to allow any programming models or APIs (Application Programming Interfaces) which need atomic behaviors to map cleanly onto it. It is also modeled primarily on hardware behavior. Just as hardware provides a "universal IR" for source languages, it also provides a starting point for developing a "universal" atomic operation and synchronization IR.

These do *not* form an API such as high-level threading libraries, software transaction memory systems, atomic primitives, and intrinsic functions as found in BSD, GNU libc, atomic_ops, APR, and other system and application libraries. The hardware interface provided by LLVM should allow a clean implementation of all of these APIs and parallel programming models. No one model or paradigm should be selected above others unless the hardware itself ubiquitously does so.

'`llvm.memory.barrier`' Intrinsic

Syntax:

```
declare void @llvm.memory.barrier( i1 <ll>, i1 <ls>, i1 <sl>, i1 <ss>, i1 <device> )
```

Overview:

The `llvm.memory.barrier` intrinsic guarantees ordering between specific pairs of memory access types.

Arguments:

The `llvm.memory.barrier` intrinsic requires five boolean arguments. The first four arguments enables a specific barrier as listed below. The fith argument specifies that the barrier applies to io or device or uncached memory.

- `ll`: load-load barrier
- `ls`: load-store barrier
- `sl`: store-load barrier
- `ss`: store-store barrier
- `device`: barrier applies to device and uncached memory also.

Semantics:

This intrinsic causes the system to enforce some ordering constraints upon the loads and stores of the program. This barrier does not indicate *when* any events will occur, it only enforces an *order* in which they occur. For any of the specified pairs of load and store operations (f.ex. load-load, or store-load), all of the first operations preceding the barrier will complete before any of the second operations succeeding the barrier begin. Specifically the semantics for each pairing is as follows:

- `ll`: All loads before the barrier must complete before any load after the barrier begins.
- `ls`: All loads before the barrier must complete before any store after the barrier begins.
- `ss`: All stores before the barrier must complete before any store after the barrier begins.
- `sl`: All stores before the barrier must complete before any load after the barrier begins.

These semantics are applied with a logical "and" behavior when more than one is enabled in a single memory barrier intrinsic.

Backends may implement stronger barriers than those requested when they do not support as fine grained a barrier as requested. Some architectures do not need all types of barriers and on such architectures, these become noops.

Example:

```
%ptr      = malloc i32
```

```
            store i32 4, %ptr

%result1  = load i32* %ptr      ; yields {i32}:result1 = 4
            call void @llvm.memory.barrier( i1 false, i1 true, i1 false, i1 false )
                                ; guarantee the above finishes
            store i32 8, %ptr   ; before this begins
```

'*llvm.atomic.cmp.swap.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.atomic.cmp.swap` on any integer bit width and for different address spaces. Not all targets support all bit widths however.

```
declare i8 @llvm.atomic.cmp.swap.i8.p0i8( i8* <ptr>, i8 <cmp>, i8 <val> )
declare i16 @llvm.atomic.cmp.swap.i16.p0i16( i16* <ptr>, i16 <cmp>, i16 <val> )
declare i32 @llvm.atomic.cmp.swap.i32.p0i32( i32* <ptr>, i32 <cmp>, i32 <val> )
declare i64 @llvm.atomic.cmp.swap.i64.p0i64( i64* <ptr>, i64 <cmp>, i64 <val> )
```

Overview:

This loads a value in memory and compares it to a given value. If they are equal, it stores a new value into the memory.

Arguments:

The `llvm.atomic.cmp.swap` intrinsic takes three arguments. The result as well as both `cmp` and `val` must be integer values with the same bit width. The `ptr` argument must be a pointer to a value of this integer type. While any bit width integer may be used, targets may only lower representations they support in hardware.

Semantics:

This entire intrinsic must be executed atomically. It first loads the value in memory pointed to by `ptr` and compares it with the value `cmp`. If they are equal, `val` is stored into the memory. The loaded value is yielded in all cases. This provides the equivalent of an atomic compare-and-swap operation within the SSA framework.

Examples:

```
%ptr      = malloc i32
            store i32 4, %ptr

%val1     = add i32 4, 4
%result1  = call i32 @llvm.atomic.cmp.swap.i32.p0i32( i32* %ptr, i32 4, %val1 )
                                      ; yields {i32}:result1 = 4
%stored1  = icmp eq i32 %result1, 4       ; yields {i1}:stored1 = true
%memval1  = load i32* %ptr                ; yields {i32}:memval1 = 8

%val2     = add i32 1, 1
%result2  = call i32 @llvm.atomic.cmp.swap.i32.p0i32( i32* %ptr, i32 5, %val2 )
                                      ; yields {i32}:result2 = 8
%stored2  = icmp eq i32 %result2, 5       ; yields {i1}:stored2 = false

%memval2  = load i32* %ptr                ; yields {i32}:memval2 = 8
```

'*llvm.atomic.swap.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.atomic.swap` on any integer bit width. Not all targets support all bit widths however.

```
declare i8 @llvm.atomic.swap.i8.p0i8( i8* <ptr>, i8 <val> )
```

```
declare i16 @llvm.atomic.swap.i16.p0i16( i16* <ptr>, i16 <val> )
declare i32 @llvm.atomic.swap.i32.p0i32( i32* <ptr>, i32 <val> )
declare i64 @llvm.atomic.swap.i64.p0i64( i64* <ptr>, i64 <val> )
```

Overview:

This intrinsic loads the value stored in memory at `ptr` and yields the value from memory. It then stores the value in `val` in the memory at `ptr`.

Arguments:

The `llvm.atomic.swap` intrinsic takes two arguments. Both the `val` argument and the result must be integers of the same bit width. The first argument, `ptr`, must be a pointer to a value of this integer type. The targets may only lower integer representations they support.

Semantics:

This intrinsic loads the value pointed to by `ptr`, yields it, and stores `val` back into `ptr` atomically. This provides the equivalent of an atomic swap operation within the SSA framework.

Examples:

```
%ptr      = malloc i32
            store i32 4, %ptr

%val1     = add i32 4, 4
%result1  = call i32 @llvm.atomic.swap.i32.p0i32( i32* %ptr, i32 %val1 )
                                      ; yields {i32}:result1 = 4
%stored1  = icmp eq i32 %result1, 4     ; yields {i1}:stored1 = true
%memval1  = load i32* %ptr              ; yields {i32}:memval1 = 8

%val2     = add i32 1, 1
%result2  = call i32 @llvm.atomic.swap.i32.p0i32( i32* %ptr, i32 %val2 )
                                      ; yields {i32}:result2 = 8

%stored2  = icmp eq i32 %result2, 8     ; yields {i1}:stored2 = true
%memval2  = load i32* %ptr              ; yields {i32}:memval2 = 2
```

'*llvm.atomic.load.add.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.atomic.load.add` on any integer bit width. Not all targets support all bit widths however.

```
declare i8 @llvm.atomic.load.add.i8..p0i8( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.add.i16..p0i16( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.add.i32..p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.add.i64..p0i64( i64* <ptr>, i64 <delta> )
```

Overview:

This intrinsic adds `delta` to the value stored in memory at `ptr`. It yields the original value at `ptr`.

Arguments:

The intrinsic takes two arguments, the first a pointer to an integer value and the second an integer value. The result is also an integer value. These integer types can have any bit width, but they must all have the same bit width. The targets may only lower integer representations they support.

Semantics:

This intrinsic does a series of operations atomically. It first loads the value stored at `ptr`. It then adds `delta`,

stores the result to `ptr`. It yields the original value stored at `ptr`.

Examples:

```
%ptr      = malloc i32
        store i32 4, %ptr
%result1  = call i32 @llvm.atomic.load.add.i32.p0i32( i32* %ptr, i32 4 )
                                ; yields {i32}:result1 = 4
%result2  = call i32 @llvm.atomic.load.add.i32.p0i32( i32* %ptr, i32 2 )
                                ; yields {i32}:result2 = 8
%result3  = call i32 @llvm.atomic.load.add.i32.p0i32( i32* %ptr, i32 5 )
                                ; yields {i32}:result3 = 10
%memval1  = load i32* %ptr      ; yields {i32}:memval1 = 15
```

'*llvm.atomic.load.sub.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.atomic.load.sub` on any integer bit width and for different address spaces. Not all targets support all bit widths however.

```
declare i8 @llvm.atomic.load.sub.i8.p0i32( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.sub.i16.p0i32( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.sub.i32.p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.sub.i64.p0i32( i64* <ptr>, i64 <delta> )
```

Overview:

This intrinsic subtracts `delta` to the value stored in memory at `ptr`. It yields the original value at `ptr`.

Arguments:

The intrinsic takes two arguments, the first a pointer to an integer value and the second an integer value. The result is also an integer value. These integer types can have any bit width, but they must all have the same bit width. The targets may only lower integer representations they support.

Semantics:

This intrinsic does a series of operations atomically. It first loads the value stored at `ptr`. It then subtracts `delta`, stores the result to `ptr`. It yields the original value stored at `ptr`.

Examples:

```
%ptr      = malloc i32
        store i32 8, %ptr
%result1  = call i32 @llvm.atomic.load.sub.i32.p0i32( i32* %ptr, i32 4 )
                                ; yields {i32}:result1 = 8
%result2  = call i32 @llvm.atomic.load.sub.i32.p0i32( i32* %ptr, i32 2 )
                                ; yields {i32}:result2 = 4
%result3  = call i32 @llvm.atomic.load.sub.i32.p0i32( i32* %ptr, i32 5 )
                                ; yields {i32}:result3 = 2
%memval1  = load i32* %ptr      ; yields {i32}:memval1 = -3
```

'*llvm.atomic.load.and.**' Intrinsic
'*llvm.atomic.load.nand.**' Intrinsic
'*llvm.atomic.load.or.**' Intrinsic
'*llvm.atomic.load.xor.**' Intrinsic

Syntax:

These are overloaded intrinsics. You can use `llvm.atomic.load_and`, `llvm.atomic.load_nand`, `llvm.atomic.load_or`, and `llvm.atomic.load_xor` on any integer bit width and for different address spaces. Not all targets support all bit widths however.

```
declare i8 @llvm.atomic.load.and.i8.p0i8( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.and.i16.p0i16( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.and.i32.p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.and.i64.p0i64( i64* <ptr>, i64 <delta> )

declare i8 @llvm.atomic.load.or.i8.p0i8( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.or.i16.p0i16( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.or.i32.p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.or.i64.p0i64( i64* <ptr>, i64 <delta> )

declare i8 @llvm.atomic.load.nand.i8.p0i32( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.nand.i16.p0i32( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.nand.i32.p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.nand.i64.p0i32( i64* <ptr>, i64 <delta> )

declare i8 @llvm.atomic.load.xor.i8.p0i32( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.xor.i16.p0i32( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.xor.i32.p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.xor.i64.p0i32( i64* <ptr>, i64 <delta> )
```

Overview:

These intrinsics bitwise the operation (and, nand, or, xor) `delta` to the value stored in memory at `ptr`. It yields the original value at `ptr`.

Arguments:

These intrinsics take two arguments, the first a pointer to an integer value and the second an integer value. The result is also an integer value. These integer types can have any bit width, but they must all have the same bit width. The targets may only lower integer representations they support.

Semantics:

These intrinsics does a series of operations atomically. They first load the value stored at `ptr`. They then do the bitwise operation `delta`, store the result to `ptr`. They yield the original value stored at `ptr`.

Examples:

```
%ptr      = malloc i32
        store i32 0x0F0F, %ptr
%result0  = call i32 @llvm.atomic.load.nand.i32.p0i32( i32* %ptr, i32 0xFF )
                                ; yields {i32}:result0 = 0x0F0F
%result1  = call i32 @llvm.atomic.load.and.i32.p0i32( i32* %ptr, i32 0xFF )
                                ; yields {i32}:result1 = 0xFFFFFFF0
%result2  = call i32 @llvm.atomic.load.or.i32.p0i32( i32* %ptr, i32 0F )
                                ; yields {i32}:result2 = 0xF0
%result3  = call i32 @llvm.atomic.load.xor.i32.p0i32( i32* %ptr, i32 0F )
                                ; yields {i32}:result3 = FF
%memval1  = load i32* %ptr      ; yields {i32}:memval1 = F0
```

'*llvm.atomic.load.max.**' Intrinsic
'*llvm.atomic.load.min.**' Intrinsic
'*llvm.atomic.load.umax.**' Intrinsic
'*llvm.atomic.load.umin.**' Intrinsic

Syntax:

These are overloaded intrinsics. You can use `llvm.atomic.load_max`, `llvm.atomic.load_min`, `llvm.atomic.load_umax`, and `llvm.atomic.load_umin` on any integer bit width and for different address spaces. Not all targets support all bit widths however.

```
declare i8 @llvm.atomic.load.max.i8.p0i8( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.max.i16.p0i16( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.max.i32.p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.max.i64.p0i64( i64* <ptr>, i64 <delta> )

declare i8 @llvm.atomic.load.min.i8.p0i8( i8* <ptr>, i8 <delta> )
```

```
declare i16 @llvm.atomic.load.min.i16.p0i16( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.min.i32..p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.min.i64..p0i64( i64* <ptr>, i64 <delta> )

declare i8 @llvm.atomic.load.umax.i8.p0i8( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.umax.i16.p0i16( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.umax.i32.p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.umax.i64.p0i64( i64* <ptr>, i64 <delta> )

declare i8 @llvm.atomic.load.umin.i8..p0i8( i8* <ptr>, i8 <delta> )
declare i16 @llvm.atomic.load.umin.i16.p0i16( i16* <ptr>, i16 <delta> )
declare i32 @llvm.atomic.load.umin.i32..p0i32( i32* <ptr>, i32 <delta> )
declare i64 @llvm.atomic.load.umin.i64..p0i64( i64* <ptr>, i64 <delta> )
```

Overview:

These intrinsics takes the signed or unsigned minimum or maximum of `delta` and the value stored in memory at `ptr`. It yields the original value at `ptr`.

Arguments:

These intrinsics take two arguments, the first a pointer to an integer value and the second an integer value. The result is also an integer value. These integer types can have any bit width, but they must all have the same bit width. The targets may only lower integer representations they support.

Semantics:

These intrinsics does a series of operations atomically. They first load the value stored at `ptr`. They then do the signed or unsigned min or max `delta` and the value, store the result to `ptr`. They yield the original value stored at `ptr`.

Examples:

```
%ptr      = malloc i32
        store i32 7, %ptr
%result0  = call i32 @llvm.atomic.load.min.i32.p0i32( i32* %ptr, i32 -2 )
                                ; yields {i32}:result0 = 7
%result1  = call i32 @llvm.atomic.load.max.i32.p0i32( i32* %ptr, i32 8 )
                                ; yields {i32}:result1 = -2
%result2  = call i32 @llvm.atomic.load.umin.i32.p0i32( i32* %ptr, i32 10 )
                                ; yields {i32}:result2 = 8
%result3  = call i32 @llvm.atomic.load.umax.i32.p0i32( i32* %ptr, i32 30 )
                                ; yields {i32}:result3 = 8
%memval1  = load i32* %ptr      ; yields {i32}:memval1 = 30
```

General Intrinsics

This class of intrinsics is designed to be generic and has no specific purpose.

'*llvm.var.annotation*' Intrinsic

Syntax:

```
declare void @llvm.var.annotation(i8* <val>, i8* <str>, i8* <str>, i32  <int> )
```

Overview:

The '`llvm.var.annotation`' intrinsic.

Arguments:

The first argument is a pointer to a value, the second is a pointer to a global string, the third is a pointer to a global string which is the source file name, and the last argument is the line number.

Semantics:

This intrinsic allows annotation of local variables with arbitrary strings. This can be useful for special purpose optimizations that want to look for these annotations. These have no other defined use, they are ignored by code generation and optimization.

'*llvm.annotation.**' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use '`llvm.annotation`' on any integer bit width.

```
declare i8 @llvm.annotation.i8(i8 <val>, i8* <str>, i8* <str>, i32  <int> )
declare i16 @llvm.annotation.i16(i16 <val>, i8* <str>, i8* <str>, i32  <int> )
declare i32 @llvm.annotation.i32(i32 <val>, i8* <str>, i8* <str>, i32  <int> )
declare i64 @llvm.annotation.i64(i64 <val>, i8* <str>, i8* <str>, i32  <int> )
declare i256 @llvm.annotation.i256(i256 <val>, i8* <str>, i8* <str>, i32  <int> )
```

Overview:

The '`llvm.annotation`' intrinsic.

Arguments:

The first argument is an integer value (result of some expression), the second is a pointer to a global string, the third is a pointer to a global string which is the source file name, and the last argument is the line number. It returns the value of the first argument.

Semantics:

This intrinsic allows annotations to be put on arbitrary expressions with arbitrary strings. This can be useful for special purpose optimizations that want to look for these annotations. These have no other defined use, they are ignored by code generation and optimization.

'*llvm.trap*' Intrinsic

Syntax:

```
declare void @llvm.trap()
```

Overview:

The '`llvm.trap`' intrinsic.

Arguments:

None.

Semantics:

This intrinsics is lowered to the target dependent trap instruction. If the target does not have a trap instruction, this intrinsic will be lowered to the call of the `abort()` function.

'*llvm.stackprotector*' Intrinsic

Syntax:

```
declare void @llvm.stackprotector( i8* <guard>, i8** <slot> )
```

Overview:

The `llvm.stackprotector` intrinsic takes the `guard` and stores it onto the stack at `slot`. The stack slot is adjusted to ensure that it is placed on the stack before local variables.

Arguments:

The `llvm.stackprotector` intrinsic requires two pointer arguments. The first argument is the value loaded from the stack guard `@__stack_chk_guard`. The second variable is an `alloca` that has enough space to hold the value of the guard.

Semantics:

This intrinsic causes the prologue/epilogue inserter to force the position of the `AllocaInst` stack slot to be before local variables on the stack. This is to ensure that if a local variable on the stack is overwritten, it will destroy the value of the guard. When the function exits, the guard on the stack is checked against the original guard. If they're different, then the program aborts by calling the `__stack_chk_fail()` function.

Chris Lattner
The LLVM Compiler Infrastructure
Last modified: $Date: 2009/10/24 04:13:14 $