LLVM Language Reference Manual

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 the section 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, 3
```

And the hard way:

```
%0 = add i32 %X, %X           ; yields {i32}:%0
%1 = 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 (using a per-function incrementing counter, starting with 0). Note that basic blocks are included in this numbering. For example, if the entry basic block is not given a label name, then it will get number 0.

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.

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.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

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

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!1 = metadata !{i32 42}
!foo = !{!1, null}
```

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

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 evaluated by the linker. Unlike normal strong symbols, they are removed by the linker from the final linked image (executable or dynamic library).

`linker_private_weak`

Similar to "`linker_private`", but the symbol is weak. Note that `linker_private_weak` symbols are subject to coalescing by the linker. The symbols are removed by the linker from the final linked image (executable or dynamic library).

`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 can be used to implement some forms of inline functions, templates, or other code which must be generated in each translation unit that uses it, but where the body may be overridden with a more definitive definition later. Unreferenced `linkonce` globals are allowed to be discarded. Note that `linkonce` linkage does not actually allow the optimizer to inline the body of this function into callers because it doesn't know if this definition of the function is the definitive definition within the program or whether it will be overridden by a stronger definition. To enable inlining and other optimizations, use "`linkonce_odr`" linkage.

`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.

`external`
: 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 `external`, `dllimport` or `extern_weak`.

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). Tail calls can only be optimized when this, the GHC or the HiPE convention is used. 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 10`" - GHC convention
: This calling convention has been implemented specifically for use by the Glasgow Haskell

Compiler (GHC). It passes everything in registers, going to extremes to achieve this by disabling callee save registers. This calling convention should not be used lightly but only for specific situations such as an alternative to the *register pinning* performance technique often used when implementing functional programming languages. At the moment only X86 supports this convention and it has the following limitations:

- On *X86-32* only supports up to 4 bit type parameters. No floating point types are supported.
- On *X86-64* only supports up to 10 bit type parameters and 6 floating point parameters.

This calling convention supports tail call optimization but requires both the caller and callee are using it.

"`cc 11`" - The HiPE calling convention

This calling convention has been implemented specifically for use by the High-Performance Erlang (HiPE) compiler, *the* native code compiler of the Ericsson's Open Source Erlang/OTP system. It uses more registers for argument passing than the ordinary C calling convention and defines no callee-saved registers. The calling convention properly supports tail call optimization but requires that both the caller and the callee use it. It uses a *register pinning* mechanism, similar to GHC's convention, for keeping frequently accessed runtime components pinned to specific hardware registers. At the moment only X86 supports this convention (both 32 and 64 bit).

"`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 definitions must be initialized, may have an explicit section to be placed in, and may have an optional explicit alignment specified.

Global variables in other translation units can also be declared, in which case they don't have an initializer.

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). Not all targets support thread-local variables. Optionally, a TLS model may be specified:

`localdynamic`
: For variables that are only used within the current shared library.

`initialexec`
: For variables in modules that will not be loaded dynamically.

`localexec`
: For variables defined in the executable and only used within it.

The models correspond to the ELF TLS models; see ELF Handling For Thread-Local Storage for more information on under which circumstances the different models may be used. The target may choose a different TLS model if the specified model is not supported, or if a better choice of model can be made.

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.

Global variables can be marked with `unnamed_addr` which indicates that the address is not significant, only the content. Constants marked like this can be merged with other constants if they have the same

initializer. Note that a constant with significant address *can* be merged with a `unnamed_addr` constant, the result being a constant whose address is significant.

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.

By default, global initializers are optimized by assuming that global variables defined within the module are not modified from their initial values before the start of the global initializer. This is true even for variables potentially accessible from outside the module, including those with external linkage or appearing in `@llvm.used`. This assumption may be suppressed by marking the variable with `externally_initialized`.

An explicit alignment may be specified for a global, which must be a power of 2. 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 exactly that alignment. Targets and optimizers are not allowed to over-align the global if the global has an assigned section. In this case, the extra alignment could be observable: for example, code could assume that the globals are densely packed in their section and try to iterate over them as an array, alignment padding would break this iteration.

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
```

The following example just declares a global variable

```
@G = external global i32
```

The following example defines a thread-local global with the `initialexec` TLS model:

```
@G = thread_local(initialexec) global i32 0, align 4
```

Functions

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

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). If an explicit label is not provided, a block is assigned an implicit numbered label,

using the next value from the same counter as used for unnamed temporaries (*see above*). For example, if a function entry block does not have an explicit label, it will be assigned label "%0", then the first unnamed temporary in that block will be "%1", etc.

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.

If the `unnamed_addr` attribute is given, the address is know to not be significant and two identical functions can be merged.

Syntax:

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

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>
```

The linkage must be one of `private`, `linker_private`, `linker_private_weak`, `internal`, `linkonce`, `weak`, `linkonce_odr`, `weak_odr`, `external`. Note that some system linkers might not correctly handle dropping a weak symbol that is aliased by a non weak alias.

Named Metadata

Named metadata is a collection of metadata. *Metadata nodes* (but not metadata strings) are the only valid operands for a named metadata.

Syntax:

```
; Some unnamed metadata nodes, which are referenced by the named metadata.
!0 = metadata !{metadata !"zero"}
!1 = metadata !{metadata !"one"}
!2 = metadata !{metadata !"two"}
; A named metadata.
!name = !{!0, !1, !2}
```

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 the extent required by the target's ABI (which is usually 32-bits, but is 8-bits for a i1 on x86-64) 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 the extent required by the target's ABI (which is usually 32-bits) 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 caller. 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. It indicates the alignment of the stack slot to form and the known alignment of the pointer specified to the call site. If the alignment is not specified, then the code generator makes a target-specific assumption.

`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 not to trap and to be properly aligned. This may only be applied to the first parameter. This is not a valid attribute for return values.

`noalias`
: This indicates that pointer values *based* on the argument or return value do not alias pointer values which are not *based* on it, ignoring certain "irrelevant" dependencies. For a call to the

parent function, dependencies between memory references from before or after the call and from those during the call are "irrelevant" to the `noalias` keyword for the arguments and return value used in that call. The caller shares the responsibility with the callee for ensuring that these requirements are met. For further details, please see the discussion of the NoAlias response in alias analysis.

Note that this definition of `noalias` is intentionally similar to the definition of `restrict` in C99 for function arguments, though it is slightly weaker.

For function return values, C99's `restrict` is not meaningful, while LLVM's `noalias` is.

`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 and can only be applied to one parameter.

`returned`

This indicates that the function always returns the argument as its return value. This is an optimization hint to the code generator when generating the caller, allowing tail call optimization and omission of register saves and restores in some cases; it is not checked or enforced when generating the callee. The parameter and the function return type must be valid operands for the *bitcast instruction*. This is not a valid attribute for return values and can only be applied to one parameter.

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.

Prefix Data

Prefix data is data associated with a function which the code generator will emit immediately before the function body. The purpose of this feature is to allow frontends to associate language-specific runtime metadata with specific functions and make it available through the function pointer while still allowing the function pointer to be called. To access the data for a given function, a program may bitcast the function pointer to a pointer to the constant's type. This implies that the IR symbol points to the start of the prefix data.

To maintain the semantics of ordinary function calls, the prefix data must have a particular format. Specifically, it must begin with a sequence of bytes which decode to a sequence of machine instructions, valid for the module's target, which transfer control to the point immediately succeeding the prefix data, without performing any other visible action. This allows the inliner and other passes to reason about the semantics of the function definition without needing to reason about the prefix data. Obviously this makes the format of the prefix data highly target dependent.

Prefix data is laid out as if it were an initializer for a global variable of the prefix data's type. No padding is automatically placed between the prefix data and the function body. If padding is required, it must be part of the prefix data.

A trivial example of valid prefix data for the x86 architecture is `i8 144`, which encodes the `nop` instruction:

```
define void @f() prefix i8 144 { ... }
```

Generally prefix data can be formed by encoding a relative branch instruction which skips the metadata, as in this example of valid prefix data for the x86_64 architecture, where the first two bytes encode `jmp .+10`:

```
%0 = type <{ i8, i8, i8* }>

define void @f() prefix %0 <{ i8 235, i8 8, i8* @md}> { ... }
```

A function may have prefix data but no body. This has similar semantics to the `available_externally` linkage in that the data may be used by the optimizers but will not be emitted in the object file.

Attribute Groups

Attribute groups are groups of attributes that are referenced by objects within the IR. They are important for keeping `.ll` files readable, because a lot of functions will use the same set of attributes. In the degenerative case of a `.ll` file that corresponds to a single `.c` file, the single attribute group will capture the important command line flags used to build that file.

An attribute group is a module-level object. To use an attribute group, an object references the attribute group's ID (e.g. `#37`). An object may refer to more than one attribute group. In that situation, the attributes from the different groups are merged.

Here is an example of attribute groups for a function that should always be inlined, has a stack alignment of 4, and which shouldn't use SSE instructions:

```
; Target-independent attributes:
attributes #0 = { alwaysinline alignstack=4 }

; Target-dependent attributes:
attributes #1 = { "no-sse" }

; Function @f has attributes: alwaysinline, alignstack=4, and "no-sse".
define void @f() #0 #1 { ... }
```

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 function 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 { ... }
```

`alignstack(<n>)`

> This attribute indicates that, when emitting the prologue and epilogue, the backend should forcibly align the stack pointer. Specify the desired alignment, which must be a power of two, in parentheses.

`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.

`builtin`

This indicates that the callee function at a call site should be recognized as a built-in function, even though the function's declaration uses the `nobuiltin` attribute. This is only valid at call sites for direct calls to functions which are declared with the `nobuiltin` attribute.

`cold`

This attribute indicates that this function is rarely called. When computing edge weights, basic blocks post-dominated by a cold function call are also considered to be cold; and, thus, given low weight.

`inlinehint`

This attribute indicates that the source code contained a hint that inlining this function is desirable (such as the "inline" keyword in C/C++). It is just a hint; it imposes no requirements on the inliner.

`minsize`

This attribute suggests that optimization passes and code generator passes make choices that keep the code size of this function as small as possible and perform optimizations that may sacrifice runtime performance in order to minimize the size of the generated code.

`naked`

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

`nobuiltin`

This indicates that the callee function at a call site is not recognized as a built-in function. LLVM will retain the original call and not replace it with equivalent code based on the semantics of the built-in function, unless the call site uses the `builtin` attribute. This is valid at call sites and on function declarations and definitions.

`noduplicate`

This attribute indicates that calls to the function cannot be duplicated. A call to a `noduplicate` function may be moved within its parent function, but may not be duplicated within its parent function.

A function containing a `noduplicate` call may still be an inlining candidate, provided that the call is not duplicated by inlining. That implies that the function has internal linkage and only has one call site, so the original call is dead after inlining.

`noimplicitfloat`

This attributes disables implicit floating point instructions.

`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.

`nonlazybind`

This attribute suppresses lazy symbol binding for the function. This may make calls to the function faster, at the cost of extra program startup time if the function is not called during program startup.

`noredzone`

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

`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.

`optnone`

This function attribute indicates that the function is not optimized by any optimization or code generator passes with the exception of interprocedural optimization passes. This attribute cannot be used together with the `alwaysinline` attribute; this attribute is also incompatible with the `minsize` attribute and the `optsize` attribute.

This attribute requires the `noinline` attribute to be specified on the function as well, so the function is never inlined into any caller. Only functions with the `alwaysinline` attribute are valid candidates for inlining into the body of this function.

`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 as long as they do not significantly impact runtime performance.

`readnone`

On a function, 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.

On an argument, this attribute indicates that the function does not dereference that pointer argument, even though it may read or write the memory that the pointer points to if accessed through other pointers.

`readonly`

On a function, 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.

On an argument, this attribute indicates that the function does not write through this pointer argument, even though it may write to the memory that the pointer points to.

`returns_twice`

This attribute indicates that this function can return twice. The C `setjmp` is an example of such a function. The compiler disables some optimizations (like tail calls) in the caller of these functions.

`sanitize_address`

This attribute indicates that AddressSanitizer checks (dynamic address safety analysis) are enabled for this function.

`sanitize_memory`

This attribute indicates that MemorySanitizer checks (dynamic detection of accesses to uninitialized memory) are enabled for this function.

`sanitize_thread`

This attribute indicates that ThreadSanitizer checks (dynamic thread safety analysis) are enabled for this function.

`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. The heuristic used will enable protectors for functions with:

- Character arrays larger than `ssp-buffer-size` (default 8).
- Aggregates containing character arrays larger than `ssp-buffer-size`.
- Calls to alloca() with variable sizes or constant sizes greater than `ssp-buffer-size`.

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` or `sspstrong` attribute, then the resulting function will have an `sspreq` attribute.

`sspstrong`

This attribute indicates that the function should emit a stack smashing protector. This attribute causes a strong heuristic to be used when determining if a function needs stack protectors. The strong heuristic will enable protectors for functions with:

- Arrays of any size and type
- Aggregates containing an array of any size and type.
- Calls to alloca().
- Local variables that have had their address taken.

This overrides the `ssp` function attribute.

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

`uwtable`

This attribute indicates that the ABI being targeted requires that an unwind table entry be produce for this function even if we can show that no exceptions passes by it. This is normally the case for the ELF x86-64 abi, but it can be disabled for some compilation units.

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.

`S<size>`
: Specifies the natural alignment of the stack in bits. Alignment promotion of stack variables is limited to the natural stack alignment to avoid dynamic stack realignment. The stack alignment must be a multiple of 8-bits. If omitted, the natural stack alignment defaults to "unspecified", which does not prevent any alignment promotions.

`p[n]:<size>:<abi>:<pref>`
: This specifies the *size* of a pointer and its `<abi>` and `<pref>`erred alignments for address space `n`. All sizes are in bits. Specifying the `<pref>` alignment is optional. If omitted, the preceding `:` should be omitted too. The address space, `n` is optional, and if not specified, denotes the default address space 0. The value of `n` must be in the range [1,2^23).

`i<size>:<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).

`v<size>:<abi>:<pref>`
: This specifies the alignment for a vector type of a given bit `<size>`.

`f<size>:<abi>:<pref>`
: This specifies the alignment for a floating point type of a given bit `<size>`. Only values of `<size>` that are supported by the target will work. 32 (float) and 64 (double) are supported on all targets; 80 or 128 (different flavors of long double) are also supported on some targets.

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

`s<size>:<abi>:<pref>`
: This specifies the alignment for a stack object of a given bit `<size>`.

`n<size1>:<size2>:<size3>...`

This specifies a set of native integer widths for the target CPU in bits. For example, it might contain `n32` for 32-bit PowerPC, `n32:64` for PowerPC 64, or `n8:16:32:64` for X86-64. Elements of this set are considered to support most general arithmetic operations efficiently.

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

- `E` - big endian
- `p:64:64:64` - 64-bit pointers with 64-bit alignment.
- `p[n]:64:64:64` - Other address spaces are assumed to be the same as the default address space.
- `S0` - natural stack alignment is unspecified
- `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
- `f16:16:16` - half is 16-bit aligned
- `f32:32:32` - float is 32-bit aligned
- `f64:64:64` - double is 64-bit aligned
- `f128:128:128` - quad is 128-bit aligned
- `v64:64:64` - 64-bit vector is 64-bit aligned
- `v128:128:128` - 128-bit vector is 128-bit aligned
- `a0:0:64` - aggregates 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 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.

The function of the data layout string may not be what you expect. Notably, this is not a specification from the frontend of what alignment the code generator should use.

Instead, if specified, the target data layout is required to match what the ultimate *code generator* expects. This string is used by the mid-level optimizers to improve code, and this only works if it matches what the ultimate code generator uses. If you would like to generate IR that does not embed this target-specific detail into the IR, then you don't have to specify the string. This will disable some optimizations that require precise layout information, but this also prevents those optimizations from introducing target specificity into the IR.

Target Triple

A module may specify a target triple string that describes the target host. The syntax for the target triple is simply:

```
target triple = "x86_64-apple-macosx10.7.0"
```

The *target triple* string consists of a series of identifiers delimited by the minus sign character ('-'). The canonical forms are:

```
ARCHITECTURE-VENDOR-OPERATING_SYSTEM
ARCHITECTURE-VENDOR-OPERATING_SYSTEM-ENVIRONMENT
```

This information is passed along to the backend so that it generates code for the proper architecture. It's possible to override this on the command line with the `-mtriple` command line option.

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 is associated with the addresses associated with any value it is *based* on.
- 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.
- 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.

A pointer value is *based* on another pointer value according to the following rules:

- A pointer value formed from a `getelementptr` operation is *based* on the first operand of the `getelementptr`.
- The result value of a `bitcast` is *based* on the operand of the `bitcast`.
- A pointer value formed by an `inttoptr` is *based* on all pointer values that contribute (directly or indirectly) to the computation of the pointer's value.
- The "*based* on" relationship is transitive.

Note that this definition of *"based"* is intentionally similar to the definition of *"based"* in C99, though it is slightly weaker.

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 type 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.

Volatile Memory Accesses

Certain memory accesses, such as *load*'s, *store*'s, and *llvm.memcpy*'s may be marked `volatile`. The optimizers must not change the number of volatile operations or change their order of execution relative to other volatile operations. The optimizers *may* change the order of volatile operations relative to non-volatile operations. This is not Java's "volatile" and has no cross-thread synchronization behavior.

IR-level volatile loads and stores cannot safely be optimized into llvm.memcpy or llvm.memmove intrinsics even when those intrinsics are flagged volatile. Likewise, the backend should never split or

merge target-legal volatile load/store instructions.

Rationale

Platforms may rely on volatile loads and stores of natively supported data width to be executed as single instruction. For example, in C this holds for an l-value of volatile primitive type with native hardware support, but not necessarily for aggregate types. The frontend upholds these expectations, which are intentionally unspecified in the IR. The rules above ensure that IR transformation do not violate the frontend's contract with the language.

Memory Model for Concurrent Operations

The LLVM IR does not define any way to start parallel threads of execution or to register signal handlers. Nonetheless, there are platform-specific ways to create them, and we define LLVM IR's behavior in their presence. This model is inspired by the C++0x memory model.

For a more informal introduction to this model, see the *LLVM Atomic Instructions and Concurrency Guide*.

We define a *happens-before* partial order as the least partial order that

- Is a superset of single-thread program order, and
- When a *synchronizes-with* `b`, includes an edge from `a` to `b`. *Synchronizes-with* pairs are introduced by platform-specific techniques, like pthread locks, thread creation, thread joining, etc., and by atomic instructions. (See also *Atomic Memory Ordering Constraints*).

Note that program order does not introduce *happens-before* edges between a thread and signals executing inside that thread.

Every (defined) read operation (load instructions, memcpy, atomic loads/read-modify-writes, etc.) R reads a series of bytes written by (defined) write operations (store instructions, atomic stores/read-modify-writes, memcpy, etc.). For the purposes of this section, initialized globals are considered to have a write of the initializer which is atomic and happens before any other read or write of the memory in question. For each byte of a read R, R_{byte} may see any write to the same byte, except:

- If $write_1$ happens before $write_2$, and $write_2$ happens before R_{byte}, then R_{byte} does not see $write_1$.
- If R_{byte} happens before $write_3$, then R_{byte} does not see $write_3$.

Given that definition, R_{byte} is defined as follows:

- If R is volatile, the result is target-dependent. (Volatile is supposed to give guarantees which can support `sig_atomic_t` in C/C++, and may be used for accesses to addresses which do not behave like normal memory. It does not generally provide cross-thread synchronization.)
- Otherwise, if there is no write to the same byte that happens before R_{byte}, R_{byte} returns `undef` for that byte.
- Otherwise, if R_{byte} may see exactly one write, R_{byte} returns the value written by that write.
- Otherwise, if R is atomic, and all the writes R_{byte} may see are atomic, it chooses one of the values written. See the *Atomic Memory Ordering Constraints* section for additional constraints on how the choice is made.
- Otherwise R_{byte} returns `undef`.

R returns the value composed of the series of bytes it read. This implies that some bytes within the value may be `undef` **without** the entire value being `undef`. Note that this only defines the semantics of the operation; it doesn't mean that targets will emit more than one instruction to read the series of bytes.

Note that in cases where none of the atomic intrinsics are used, this model places only one restriction on IR transformations on top of what is required for single-threaded execution: introducing a store to a byte which might not otherwise be stored is not allowed in general. (Specifically, in the case where another thread might write to and read from an address, introducing a store can change a load that may see exactly one write into a load that may see multiple writes.)

Atomic Memory Ordering Constraints

Atomic instructions (*cmpxchg*, *atomicrmw*, *fence*, *atomic load*, and *atomic store*) take an ordering parameter that determines which other atomic instructions on the same address they *synchronize with*. These semantics are borrowed from Java and C++0x, but are somewhat more colloquial. If these descriptions aren't precise enough, check those specs (see spec references in the *atomics guide*). *fence* instructions treat these orderings somewhat differently since they don't take an address. See that instruction's documentation for details.

For a simpler introduction to the ordering constraints, see the *LLVM Atomic Instructions and Concurrency Guide*.

`unordered`

The set of values that can be read is governed by the happens-before partial order. A value cannot be read unless some operation wrote it. This is intended to provide a guarantee strong enough to model Java's non-volatile shared variables. This ordering cannot be specified for read-modify-write operations; it is not strong enough to make them atomic in any interesting way.

`monotonic`

In addition to the guarantees of `unordered`, there is a single total order for modifications by `monotonic` operations on each address. All modification orders must be compatible with the happens-before order. There is no guarantee that the modification orders can be combined to a global total order for the whole program (and this often will not be possible). The read in an atomic read-modify-write operation (*cmpxchg* and *atomicrmw*) reads the value in the modification order immediately before the value it writes. If one atomic read happens before another atomic read of the same address, the later read must see the same value or a later value in the address's modification order. This disallows reordering of `monotonic` (or stronger) operations on the same address. If an address is written `monotonic`-ally by one thread, and other threads `monotonic`-ally read that address repeatedly, the other threads must eventually see the write. This corresponds to the C++0x/C1x `memory_order_relaxed`.

`acquire`

In addition to the guarantees of `monotonic`, a *synchronizes-with* edge may be formed with a `release` operation. This is intended to model C++'s `memory_order_acquire`.

`release`

In addition to the guarantees of `monotonic`, if this operation writes a value which is subsequently read by an `acquire` operation, it *synchronizes-with* that operation. (This isn't a complete description; see the C++0x definition of a release sequence.) This corresponds to the C++0x/C1x `memory_order_release`.

`acq_rel` (acquire+release)

Acts as both an `acquire` and `release` operation on its address. This corresponds to the C++0x/C1x `memory_order_acq_rel`.

`seq_cst` (sequentially consistent)

In addition to the guarantees of `acq_rel` (`acquire` for an operation which only reads, `release` for an

operation which only writes), there is a global total order on all sequentially-consistent operations on all addresses, which is consistent with the *happens-before* partial order and with the modification orders of all the affected addresses. Each sequentially-consistent read sees the last preceding write to the same address in this global order. This corresponds to the C++0x/C1x `memory_order_seq_cst` and Java volatile.

If an atomic operation is marked `singlethread`, it only *synchronizes with* or participates in modification and seq_cst total orderings with other operations running in the same thread (for example, in signal handlers).

Fast-Math Flags

LLVM IR floating-point binary ops (*fadd*, *fsub*, *fmul*, *fdiv*, *frem*) have the following flags that can set to enable otherwise unsafe floating point operations

`nnan`
No NaNs - Allow optimizations to assume the arguments and result are not NaN. Such optimizations are required to retain defined behavior over NaNs, but the value of the result is undefined.

`ninf`
No Infs - Allow optimizations to assume the arguments and result are not +/-Inf. Such optimizations are required to retain defined behavior over +/-Inf, but the value of the result is undefined.

`nsz`
No Signed Zeros - Allow optimizations to treat the sign of a zero argument or result as insignificant.

`arcp`
Allow Reciprocal - Allow optimizations to use the reciprocal of an argument rather than perform division.

`fast`
Fast - Allow algebraically equivalent transformations that may dramatically change results in floating point (e.g. reassociate). This flag implies all the others.

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	`half`, `float`, `double`, `x86_fp80`, `fp128`, `ppc_fp128`
first class	*integer*, *floating point*, *pointer*, *vector*, *structure*, *array*, *label*, *metadata*.

Classification	Types
primitive	*label*, *void*, *integer*, *floating point*, *x86mmx*, *metadata*.
derived	*array*, *function*, *pointer*, *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.

Primitive Types

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

Integer Type

Overview:

The integer type is a very simple 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.

Floating Point Types

Type	Description
`half`	16-bit floating point value
`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)

X86mmx Type

Overview:

The x86mmx type represents a value held in an MMX register on an x86 machine. The operations allowed on it are quite limited: parameters and return values, load and store, and bitcast. User-specified MMX instructions are represented as intrinsic or asm calls with arguments and/or results of this type. There are no arrays, vectors or constants of this type.

Syntax:

```
x86mmx
```

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. No derived types may be created from metadata except for *function* arguments.

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. Each of these types contain one or more element types which may be a primitive type, or another derived type. For example, it is possible to have a two dimensional array, using an array as the element type of another array.

Aggregate Types

Aggregate Types are a subset of derived types that can contain multiple member types. *Arrays* and *structs* are aggregate types. *Vectors* are not considered to be aggregate types.

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.

There is no restriction on indexing beyond the end of the array implied by a static type (though there are restrictions on indexing beyond the bounds of an allocated object in some cases). This means that single-dimension 'variable sized array' addressing can be implemented in LLVM with a zero length array type. An implementation of 'pascal style arrays' in LLVM could use the type "`{ i32, [0 x float]}`", for example.

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 void type or first class type — except for *label* and *metadata* types.

Syntax:

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

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

Examples:

`i32 (i32)`	function taking an `i32`, returning an `i32`
`float (i16, i32 *) *`	*Pointer* to a function that takes an `i16` 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 a *structure* containing two `i32` values

Structure Type

Overview:

The structure type is used to represent a collection of data members together in memory. The elements of a structure may be any type that has a size.

Structures in memory are accessed using '`load`' and '`store`' by getting a pointer to a field with the '`getelementptr`' instruction. Structures in registers are accessed using the '`extractvalue`' and '`insertvalue`' instructions.

Structures may optionally be "packed" structures, which indicate that the alignment of the struct is one byte, and that there is no padding between the elements. In non-packed structs, padding between field types is inserted as defined by the DataLayout string in the module, which is required to match what the underlying code generator expects.

Structures can either be "literal" or "identified". A literal structure is defined inline with other types (e.g. `{i32, i32}*`) whereas identified types are always defined at the top level with a name. Literal types are uniqued by their contents and can never be recursive or opaque since there is no way to write one. Identified types can be recursive, can be opaqued, and are never uniqued.

Syntax:

```
%T1 = type { <type list> }     ; Identified normal struct type
%T2 = type <{ <type list> }>   ; Identified packed struct type
```

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`.
`<{ i8, i32 }>`	A packed struct known to be 5 bytes in size.

Opaque Structure Types

Overview:

Opaque structure types are used to represent named structure types that do not have a body specified. This corresponds (for example) to the C notion of a forward declared structure.

Syntax:

```
%X = type opaque
%52 = type opaque
```

Examples:

`opaque`	An opaque type.

Pointer Type

Overview:

The pointer type is used to specify memory locations. Pointers are commonly used to reference objects in memory.

Pointer types may have an optional address space attribute defining the numbered address space where the pointed-to object resides. The default address space is number zero. The semantics of non-zero address spaces are target-specific.

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. Vector types are considered *first class*.

Syntax:

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

The number of elements is a constant integer value larger than 0; elementtype may be any integer or floating point type, or a pointer to these types. Vectors of size zero are not allowed.

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.
`<4 x i64*>`	Vector of 4 pointers to 64-bit integer values.

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 half, float, and double are represented using the 16-digit form shown above (which matches the IEEE754 representation for double); half and float values must, however, be exactly representable as IEEE 754 half and single precision, respectively. 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. Long doubles will only work if they match the long double format on your target. The IEEE 16-bit format (half precision) is represented by `0xH` followed by 4 hexadecimal digits. All hexadecimal formats are big-endian (sign bit at the left).

There are no constants of type x86mmx.

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`' can be used anywhere a constant is expected, and indicates that the user of the value may receive an unspecified bit-pattern. Undefined values may be of any type (other than '`label`' or '`void`') and be used anywhere a constant is permitted.

Undefined values are useful because they indicate to the compiler that the program is well defined no matter what value is used. This gives the compiler more freedom to optimize. Here are some examples of (potentially surprising) transformations that are valid (in pseudo IR):

```
  %A = add %X, undef
  %B = sub %X, undef
  %C = xor %X, undef
Safe:
  %A = undef
  %B = undef
  %C = undef
```

This is safe because all of the output bits are affected by the undef bits. Any output bit can have a zero or one depending on the input bits.

```
  %A = or %X, undef
  %B = and %X, undef
Safe:
  %A = -1
  %B = 0
Unsafe:
  %A = undef
  %B = undef
```

These logical operations have bits that are not always affected by the input. For example, if `%X` has a zero bit, then the output of the '`and`' operation will always be a zero for that bit, no matter what the corresponding bit from the '`undef`' is. As such, it is unsafe to optimize or assume that the result of the '`and`' is '`undef`'. However, it is safe to assume that all bits of the '`undef`' could be 0, and optimize the

'`and`' to 0. Likewise, it is safe to assume that all the bits of the '`undef`' operand to the '`or`' could be set, allowing the '`or`' to be folded to -1.

```
  %A = select undef, %X, %Y
  %B = select undef, 42, %Y
  %C = select %X, %Y, undef
Safe:
  %A = %X     (or %Y)
  %B = 42     (or %Y)
  %C = %Y
Unsafe:
  %A = undef
  %B = undef
  %C = undef
```

This set of examples shows that undefined '`select`' (and conditional branch) conditions can go *either way*, but they have to come from one of the two operands. In the `%A` example, if `%X` and `%Y` were both known to have a clear low bit, then `%A` would have to have a cleared low bit. However, in the `%C` example, the optimizer is allowed to assume that the '`undef`' operand could be the same as `%Y`, allowing the whole '`select`' to be eliminated.

```
  %A = xor undef, undef

  %B = undef
  %C = xor %B, %B

  %D = undef
  %E = icmp lt %D, 4
  %F = icmp gte %D, 4

Safe:
  %A = undef
  %B = undef
  %C = undef
  %D = undef
  %E = undef
  %F = undef
```

This example points out that two '`undef`' operands are not necessarily the same. This can be surprising to people (and also matches C semantics) where they assume that "`X^X`" is always zero, even if `X` is undefined. This isn't true for a number of reasons, but the short answer is that an '`undef`' "variable" can arbitrarily change its value over its "live range". This is true because the variable doesn't actually *have a live range*. Instead, the value is logically read from arbitrary registers that happen to be around when needed, so the value is not necessarily consistent over time. In fact, `%A` and `%C` need to have the same semantics or the core LLVM "replace all uses with" concept would not hold.

```
  %A = fdiv undef, %X
  %B = fdiv %X, undef
Safe:
  %A = undef
b: unreachable
```

These examples show the crucial difference between an *undefined value* and *undefined behavior*. An undefined value (like '`undef`') is allowed to have an arbitrary bit-pattern. This means that the `%A` operation can be constant folded to '`undef`', because the '`undef`' could be an SNaN, and `fdiv` is not (currently) defined on SNaN's. However, in the second example, we can make a more aggressive assumption: because the `undef` is allowed to be an arbitrary value, we are allowed to assume that it could be zero. Since a divide by zero has *undefined behavior*, we are allowed to assume that the operation does not execute at all. This allows us to delete the divide and all code after it. Because the undefined operation "can't happen", the optimizer can assume that it occurs in dead code.

```
a:  store undef -> %X
b:  store %X -> undef
Safe:
a: <deleted>
b: unreachable
```

These examples reiterate the `fdiv` example: a store *of* an undefined value can be assumed to not have any effect; we can assume that the value is overwritten with bits that happen to match what was already there. However, a store *to* an undefined location could clobber arbitrary memory, therefore, it has undefined behavior.

Poison Values

Poison values are similar to *undef values*, however they also represent the fact that an instruction or constant expression which cannot evoke side effects has nevertheless detected a condition which results in undefined behavior.

There is currently no way of representing a poison value in the IR; they only exist when produced by operations such as *add* with the `nsw` flag.

Poison value behavior is defined in terms of value *dependence*:

- Values other than *phi* nodes depend on their operands.
- *Phi* nodes depend on the operand corresponding to their dynamic predecessor basic block.
- Function arguments depend on the corresponding actual argument values in the dynamic callers of their functions.
- *Call* instructions depend on the *ret* instructions that dynamically transfer control back to them.
- *Invoke* instructions depend on the *ret*, *resume*, or exception-throwing call instructions that dynamically transfer control back to them.
- Non-volatile loads and stores depend on the most recent stores to all of the referenced memory addresses, following the order in the IR (including loads and stores implied by intrinsics such as *@llvm.memcpy*.)
- An instruction with externally visible side effects depends on the most recent preceding instruction with externally visible side effects, following the order in the IR. (This includes *volatile operations*.)
- An instruction *control-depends* on a *terminator instruction* if the terminator instruction has multiple successors and the instruction is always executed when control transfers to one of the successors, and may not be executed when control is transferred to another.
- Additionally, an instruction also *control-depends* on a terminator instruction if the set of instructions it otherwise depends on would be different if the terminator had transferred control to a different successor.
- Dependence is transitive.

Poison Values have the same behavior as *undef values*, with the additional affect that any instruction which has a *dependence* on a poison value has undefined behavior.

Here are some examples:

```
entry:
  %poison = sub nuw i32 0, 1           ; Results in a poison value.
  %still_poison = and i32 %poison, 0   ; 0, but also poison.
  %poison_yet_again = getelementptr i32* @h, i32 %still_poison
  store i32 0, i32* %poison_yet_again  ; memory at @h[0] is poisoned

  store i32 %poison, i32* @g           ; Poison value stored to memory.
  %poison2 = load i32* @g              ; Poison value loaded back from memory.
```

```
  store volatile i32 %poison, i32* @g  ; External observation; undefined behavior.

  %narrowaddr = bitcast i32* @g to i16*
  %wideaddr = bitcast i32* @g to i64*
  %poison3 = load i16* %narrowaddr     ; Returns a poison value.
  %poison4 = load i64* %wideaddr       ; Returns a poison value.

  %cmp = icmp slt i32 %poison, 0       ; Returns a poison value.
  br i1 %cmp, label %true, label %end  ; Branch to either destination.

true:
  store volatile i32 0, i32* @g        ; This is control-dependent on %cmp, so
                                       ; it has undefined behavior.
  br label %end

end:
  %p = phi i32 [ 0, %entry ], [ 1, %true ]
                                       ; Both edges into this PHI are
                                       ; control-dependent on %cmp, so this
                                       ; always results in a poison value.

  store volatile i32 0, i32* @g        ; This would depend on the store in %true
                                       ; if %cmp is true, or the store in %entry
                                       ; otherwise, so this is undefined behavior.

  br i1 %cmp, label %second_true, label %second_end
                                       ; The same branch again, but this time the
                                       ; true block doesn't have side effects.

second_true:
  ; No side effects!
  ret void

second_end:
  store volatile i32 0, i32* @g        ; This time, the instruction always depends
                                       ; on the store in %end. Also, it is
                                       ; control-equivalent to %end, so this is
                                       ; well-defined (ignoring earlier undefined
                                       ; behavior in this example).
```

Addresses of Basic Blocks

`blockaddress(@function, %block)`

The '`blockaddress`' constant computes the address of the specified basic block in the specified function, and always has an `i8*` type. Taking the address of the entry block is illegal.

This value only has defined behavior when used as an operand to the '*indirectbr*' instruction, or for comparisons against null. Pointer equality tests between labels addresses results in undefined behavior — though, again, comparison against null is ok, and no label is equal to the null pointer. This may be passed around as an opaque pointer sized value as long as the bits are not inspected. This allows `ptrtoint` and arithmetic to be performed on these values so long as the original value is reconstituted before the `indirectbr` instruction.

Finally, some targets may provide defined semantics when using the value as the operand to an inline assembly, but that is target specific.

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 than 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 than 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 an 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*.

`addrspacecast (CST to TYPE)`

Convert a constant pointer or constant vector of pointer, CST, to another TYPE in a different address space. The constraints of the operands are the same as those for the *addrspacecast 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.

`extractvalue (VAL, IDX0, IDX1, ...)`

Perform the *extractvalue operation* on constants. The index list is interpreted in a similar manner as indices in a '*getelementptr*' operation. At least one index value must be specified.

`insertvalue (VAL, ELT, IDX0, IDX1, ...)`

Perform the *insertvalue operation* on constants. The index list is interpreted in a similar manner as indices in a '*getelementptr*' operation. At least one index value must be specified.

`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).

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), a flag that indicates whether or not the inline asm expression has side effects, and a flag indicating whether the function containing the asm needs to align its stack conservatively. 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* or an *invoke* 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", ""()
```

In some cases inline asms will contain code that will not work unless the stack is aligned in some way, such as calls or SSE instructions on x86, yet will not contain code that does that alignment within the asm. The compiler should make conservative assumptions about what the asm might contain and should generate its usual stack alignment code in the prologue if the '`alignstack`' keyword is present:

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

Inline asms also support using non-standard assembly dialects. The assumed dialect is ATT. When the '`inteldialect`' keyword is present, the inline asm is using the Intel dialect. Currently, ATT and Intel are the only supported dialects. An example is:

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

If multiple keywords appear the '`sideeffect`' keyword must come first, the '`alignstack`' keyword second and the '`inteldialect`' keyword last.

Inline Asm Metadata

The call instructions that wrap inline asm nodes may have a "`!srcloc`" MDNode attached to it that contains a list of constant integers. If present, the code generator will use the integer as the location cookie value when report errors through the `LLVMContext` error reporting mechanisms. This allows a front-end to correlate backend errors that occur with inline asm back to the source code that produced it. For example:

```
call void asm sideeffect "something bad", ""(), !srcloc !42
...
!42 = !{ i32 1234567 }
```

It is up to the front-end to make sense of the magic numbers it places in the IR. If the MDNode contains multiple constants, the code generator will use the one that corresponds to the line of the asm that the error occurs on.

Metadata Nodes and Metadata Strings

LLVM IR allows metadata to be attached to instructions in the program that can convey extra information about the code to the optimizers and code generator. One example application of metadata is source-level debug information. 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 preceded by an exclamation point). Metadata nodes can have any values as their operand. For example:

```
!{ metadata !"test\00", i32 10}
```

A *named metadata* is a collection of metadata nodes, which can be looked up in the module symbol table. For example:

```
!foo =  metadata !{!4, !3}
```

Metadata can be used as function arguments. Here `llvm.dbg.value` function is using two metadata arguments:

```
call void @llvm.dbg.value(metadata !24, i64 0, metadata !25)
```

Metadata can be attached with an instruction. Here metadata `!21` is attached to the `add` instruction using the `!dbg` identifier:

```
%indvar.next = add i64 %indvar, 1, !dbg !21
```

More information about specific metadata nodes recognized by the optimizers and code generator is found below.

‘`tbaa`’ Metadata

In LLVM IR, memory does not have types, so LLVM’s own type system is not suitable for doing TBAA. Instead, metadata is added to the IR to describe a type system of a higher level language. This can be used to implement typical C/C++ TBAA, but it can also be used to implement custom alias analysis behavior for other languages.

The current metadata format is very simple. TBAA metadata nodes have up to three fields, e.g.:

```
!0 = metadata !{ metadata !"an example type tree" }
!1 = metadata !{ metadata !"int", metadata !0 }
!2 = metadata !{ metadata !"float", metadata !0 }
!3 = metadata !{ metadata !"const float", metadata !2, i64 1 }
```

The first field is an identity field. It can be any value, usually a metadata string, which uniquely identifies the type. The most important name in the tree is the name of the root node. Two trees with different root node names are entirely disjoint, even if they have leaves with common names.

The second field identifies the type’s parent node in the tree, or is null or omitted for a root node. A type is considered to alias all of its descendants and all of its ancestors in the tree. Also, a type is considered to alias all types in other trees, so that bitcode produced from multiple front-ends is handled conservatively.

If the third field is present, it’s an integer which if equal to 1 indicates that the type is “constant” (meaning `pointsToConstantMemory` should return true; see other useful AliasAnalysis methods).

‘`tbaa.struct`’ Metadata

The *llvm.memcpy* is often used to implement aggregate assignment operations in C and similar languages, however it is defined to copy a contiguous region of memory, which is more than strictly necessary for aggregate types which contain holes due to padding. Also, it doesn’t contain any TBAA information about the fields of the aggregate.

`!tbaa.struct` metadata can describe which memory subregions in a memcpy are padding and what the TBAA tags of the struct are.

The current metadata format is very simple. `!tbaa.struct` metadata nodes are a list of operands which are in conceptual groups of three. For each group of three, the first operand gives the byte offset of a field in bytes, the second gives its size in bytes, and the third gives its tbaa tag. e.g.:

```
!4 = metadata !{ i64 0, i64 4, metadata !1, i64 8, i64 4, metadata !2 }
```

This describes a struct with two fields. The first is at offset 0 bytes with size 4 bytes, and has tbaa tag !1. The second is at offset 8 bytes and has size 4 bytes and has tbaa tag !2.

Note that the fields need not be contiguous. In this example, there is a 4 byte gap between the two fields. This gap represents padding which does not carry useful data and need not be preserved.

'`fpmath`' Metadata

`fpmath` metadata may be attached to any instruction of floating point type. It can be used to express the maximum acceptable error in the result of that instruction, in ULPs, thus potentially allowing the compiler to use a more efficient but less accurate method of computing it. ULP is defined as follows:

> If `x` is a real number that lies between two finite consecutive floating-point numbers `a` and `b`, without being equal to one of them, then `ulp(x) = |b - a|`, otherwise `ulp(x)` is the distance between the two non-equal finite floating-point numbers nearest `x`. Moreover, `ulp(NaN)` is `NaN`.

The metadata node shall consist of a single positive floating point number representing the maximum relative error, for example:

```
!0 = metadata !{ float 2.5 } ; maximum acceptable inaccuracy is 2.5 ULPs
```

'`range`' Metadata

`range` metadata may be attached only to loads of integer types. It expresses the possible ranges the loaded value is in. The ranges are represented with a flattened list of integers. The loaded value is known to be in the union of the ranges defined by each consecutive pair. Each pair has the following properties:

- The type must match the type loaded by the instruction.
- The pair `a,b` represents the range `[a,b)`.
- Both `a` and `b` are constants.
- The range is allowed to wrap.
- The range should not represent the full or empty set. That is, `a!=b`.

In addition, the pairs must be in signed order of the lower bound and they must be non-contiguous.

Examples:

```
  %a = load i8* %x, align 1, !range !0 ; Can only be 0 or 1
  %b = load i8* %y, align 1, !range !1 ; Can only be 255 (-1), 0 or 1
  %c = load i8* %z, align 1, !range !2 ; Can only be 0, 1, 3, 4 or 5
  %d = load i8* %z, align 1, !range !3 ; Can only be -2, -1, 3, 4 or 5
...
!0 = metadata !{ i8 0, i8 2 }
!1 = metadata !{ i8 255, i8 2 }
!2 = metadata !{ i8 0, i8 2, i8 3, i8 6 }
!3 = metadata !{ i8 -2, i8 0, i8 3, i8 6 }
```

'`llvm.loop`'

It is sometimes useful to attach information to loop constructs. Currently, loop metadata is implemented as metadata attached to the branch instruction in the loop latch block. This type of metadata refer to a metadata node that is guaranteed to be separate for each loop. The loop identifier metadata is specified with the name `llvm.loop`.

The loop identifier metadata is implemented using a metadata that refers to itself to avoid merging it with any other identifier metadata, e.g., during module linkage or function inlining. That is, each loop should refer to their own identification metadata even if they reside in separate functions. The following example contains loop identifier metadata for two separate loop constructs:

```
!0 = metadata !{ metadata !0 }
!1 = metadata !{ metadata !1 }
```

The loop identifier metadata can be used to specify additional per-loop metadata. Any operands after the first operand can be treated as user-defined metadata. For example the `llvm.vectorizer.unroll` metadata is understood by the loop vectorizer to indicate how many times to unroll the loop:

```
  br i1 %exitcond, label %._crit_edge, label %.lr.ph, !llvm.loop !0
...
!0 = metadata !{ metadata !0, metadata !1 }
!1 = metadata !{ metadata !"llvm.vectorizer.unroll", i32 2 }
```

‘`llvm.mem`‘

Metadata types used to annotate memory accesses with information helpful for optimizations are prefixed with `llvm.mem`.

‘`llvm.mem.parallel_loop_access`‘ Metadata

For a loop to be parallel, in addition to using the `llvm.loop` metadata to mark the loop latch branch instruction, also all of the memory accessing instructions in the loop body need to be marked with the `llvm.mem.parallel_loop_access` metadata. If there is at least one memory accessing instruction not marked with the metadata, the loop must be considered a sequential loop. This causes parallel loops to be converted to sequential loops due to optimization passes that are unaware of the parallel semantics and that insert new memory instructions to the loop body.

Example of a loop that is considered parallel due to its correct use of both `llvm.loop` and `llvm.mem.parallel_loop_access` metadata types that refer to the same loop identifier metadata.

```
for.body:
  ...
  %0 = load i32* %arrayidx, align 4, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %0, i32* %arrayidx4, align 4, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %for.end, label %for.body, !llvm.loop !0

for.end:
...
!0 = metadata !{ metadata !0 }
```

It is also possible to have nested parallel loops. In that case the memory accesses refer to a list of loop identifier metadata nodes instead of the loop identifier metadata node directly:

```
outer.for.body:
...

inner.for.body:
```

```
  ...
  %0 = load i32* %arrayidx, align 4, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %0, i32* %arrayidx4, align 4, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %inner.for.end, label %inner.for.body, !llvm.loop !1

inner.for.end:
  ...
  %0 = load i32* %arrayidx, align 4, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %0, i32* %arrayidx4, align 4, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %outer.for.end, label %outer.for.body, !llvm.loop !2

outer.for.end:                                    ; preds = %for.body
...
!0 = metadata !{ metadata !1, metadata !2 } ; a list of loop identifiers
!1 = metadata !{ metadata !1 } ; an identifier for the inner loop
!2 = metadata !{ metadata !2 } ; an identifier for the outer loop
```

‘`llvm.vectorizer`‘

Metadata prefixed with `llvm.vectorizer` is used to control per-loop vectorization parameters such as vectorization factor and unroll factor.

`llvm.vectorizer` metadata should be used in conjunction with `llvm.loop` loop identification metadata.

‘`llvm.vectorizer.unroll`‘ Metadata

This metadata instructs the loop vectorizer to unroll the specified loop exactly `N` times.

The first operand is the string `llvm.vectorizer.unroll` and the second operand is an integer specifying the unroll factor. For example:

```
!0 = metadata !{ metadata !"llvm.vectorizer.unroll", i32 4 }
```

Note that setting `llvm.vectorizer.unroll` to 1 disables unrolling of the loop.

If `llvm.vectorizer.unroll` is set to 0 then the amount of unrolling will be determined automatically.

‘`llvm.vectorizer.width`‘ Metadata

This metadata sets the target width of the vectorizer to `N`. Without this metadata, the vectorizer will choose a width automatically. Regardless of this metadata, the vectorizer will only vectorize loops if it believes it is valid to do so.

The first operand is the string `llvm.vectorizer.width` and the second operand is an integer specifying the width. For example:

```
!0 = metadata !{ metadata !"llvm.vectorizer.width", i32 4 }
```

Note that setting `llvm.vectorizer.width` to 1 disables vectorization of the loop.

If `llvm.vectorizer.width` is set to 0 then the width will be determined automatically.

Module Flags Metadata

Information about the module as a whole is difficult to convey to LLVM’s subsystems. The LLVM IR isn’t sufficient to transmit this information. The `llvm.module.flags` named metadata exists in order to

facilitate this. These flags are in the form of key / value pairs — much like a dictionary — making it easy for any subsystem who cares about a flag to look it up.

The `llvm.module.flags` metadata contains a list of metadata triplets. Each triplet has the following form:

- The first element is a *behavior* flag, which specifies the behavior when two (or more) modules are merged together, and it encounters two (or more) metadata with the same ID. The supported behaviors are described below.
- The second element is a metadata string that is a unique ID for the metadata. Each module may only have one flag entry for each unique ID (not including entries with the **Require** behavior).
- The third element is the value of the flag.

When two (or more) modules are merged together, the resulting `llvm.module.flags` metadata is the union of the modules' flags. That is, for each unique metadata ID string, there will be exactly one entry in the merged modules `llvm.module.flags` metadata table, and the value for that entry will be determined by the merge behavior flag, as described below. The only exception is that entries with the *Require* behavior are always preserved.

The following behaviors are supported:

Value	Behavior
1	**Error** Emits an error if two values disagree, otherwise the resulting value is that of the operands.
2	**Warning** Emits a warning if two values disagree. The result value will be the operand for the flag from the first module being linked.
3	**Require** Adds a requirement that another module flag be present and have a specified value after linking is performed. The value must be a metadata pair, where the first element of the pair is the ID of the module flag to be restricted, and the second element of the pair is the value the module flag should be restricted to. This behavior can be used to restrict the allowable results (via triggering of an error) of linking IDs with the **Override** behavior.
4	**Override** Uses the specified value, regardless of the behavior or value of the other module. If both modules specify **Override**, but the values differ, an error will be emitted.
5	**Append** Appends the two values, which are required to be metadata nodes.
6	**AppendUnique** Appends the two values, which are required to be metadata nodes. However, duplicate entries in the second list are dropped during the append operation.

It is an error for a particular unique flag ID to have multiple behaviors, except in the case of **Require** (which adds restrictions on another metadata value) or **Override**.

An example of module flags:

```
!0 = metadata !{ i32 1, metadata !"foo", i32 1 }
!1 = metadata !{ i32 4, metadata !"bar", i32 37 }
!2 = metadata !{ i32 2, metadata !"qux", i32 42 }
!3 = metadata !{ i32 3, metadata !"qux",
  metadata !{
    metadata !"foo", i32 1
  }
}
!llvm.module.flags = !{ !0, !1, !2, !3 }
```

- Metadata `!0` has the ID `!"foo"` and the value '1'. The behavior if two or more `!"foo"` flags are seen is to emit an error if their values are not equal.
- Metadata `!1` has the ID `!"bar"` and the value '37'. The behavior if two or more `!"bar"` flags are seen is to use the value '37'.
- Metadata `!2` has the ID `!"qux"` and the value '42'. The behavior if two or more `!"qux"` flags are seen is to emit a warning if their values are not equal.
- Metadata `!3` has the ID `!"qux"` and the value:

  ```
  metadata !{ metadata !"foo", i32 1 }
  ```

  The behavior is to emit an error if the `llvm.module.flags` does not contain a flag with the ID `!"foo"` that has the value '1' after linking is performed.

Objective-C Garbage Collection Module Flags Metadata

On the Mach-O platform, Objective-C stores metadata about garbage collection in a special section called "image info". The metadata consists of a version number and a bitmask specifying what types of garbage collection are supported (if any) by the file. If two or more modules are linked together their garbage collection metadata needs to be merged rather than appended together.

The Objective-C garbage collection module flags metadata consists of the following key-value pairs:

Key	Value
`Objective-C Version`	**[Required]** — The Objective-C ABI version. Valid values are 1 and 2.
`Objective-C Image Info Version`	**[Required]** — The version of the image info section. Currently always 0.
`Objective-C Image Info Section`	**[Required]** — The section to place the metadata. Valid values are `"__OBJC, __image_info, regular"` for Objective-C ABI version 1, and `"__DATA,__objc_imageinfo, regular, no_dead_strip"` for Objective-C ABI version 2.
`Objective-C Garbage Collection`	**[Required]** — Specifies whether garbage collection is supported or not. Valid values are 0, for no garbage collection, and 2, for garbage collection supported.
`Objective-C GC Only`	**[Optional]** — Specifies that only garbage collection is supported. If present, its value must be 6. This flag requires that the `Objective-C Garbage Collection` flag have the value 2.

Some important flag interactions:

- If a module with `Objective-C Garbage Collection` set to 0 is merged with a module with `Objective-C Garbage Collection` set to 2, then the resulting module has the `Objective-C Garbage Collection` flag set to 0.
- A module with `Objective-C Garbage Collection` set to 0 cannot be merged with a module with `Objective-C GC Only` set to 6.

Automatic Linker Flags Module Flags Metadata

Some targets support embedding flags to the linker inside individual object files. Typically this is used in conjunction with language extensions which allow source files to explicitly declare the libraries they depend on, and have these automatically be transmitted to the linker via object files.

These flags are encoded in the IR using metadata in the module flags section, using the `Linker Options` key. The merge behavior for this flag is required to be `AppendUnique`, and the value for the key is expected to be a metadata node which should be a list of other metadata nodes, each of which should be a list of metadata strings defining linker options.

For example, the following metadata section specifies two separate sets of linker options, presumably to link against `libz` and the `Cocoa` framework:

```
!0 = metadata !{ i32 6, metadata !"Linker Options",
   metadata !{
      metadata !{ metadata !"-lz" },
      metadata !{ metadata !"-framework", metadata !"Cocoa" } } }
!llvm.module.flags = !{ !0 }
```

The metadata encoding as lists of lists of options, as opposed to a collapsed list of options, is chosen so that the IR encoding can use multiple option strings to specify e.g., a single library, while still having that specifier be preserved as an atomic element that can be recognized by a target specific assembly writer or object file emitter.

Each individual option is required to be either a valid option for the target's linker, or an option that is reserved by the target specific assembly writer or object file emitter. No other aspect of these options is defined by the IR.

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 which has *appending linkage*. This array contains a list of pointers to named global variables, functions and aliases 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 symbol 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 symbol that it cannot see (which is why they have to be named). 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

```
%0 = type { i32, void ()* }
@llvm.global_ctors = appending global [1 x %0] [%0 { i32 65535, void ()* @ctor }]
```

The `@llvm.global_ctors` array contains a list of constructor functions and associated priorities. The functions referenced by this array will be called in ascending order of priority (i.e. lowest first) when the module is loaded. The order of functions with the same priority is not defined.

The '`llvm.global_dtors`' Global Variable

```
%0 = type { i32, void ()* }
@llvm.global_dtors = appending global [1 x %0] [%0 { i32 65535, void ()* @dtor }]
```

The `@llvm.global_dtors` array contains a list of destructor functions and associated priorities. The functions referenced by this array will be called in descending order of priority (i.e. highest first) when the module is loaded. The order of functions with the same priority is not defined.

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).

The terminator instructions are: '*ret*', '*br*', '*switch*', '*indirectbr*', '*invoke*', '*resume*', and '*unreachable*'.

'`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
```

'`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 transferred to the corresponding destination; otherwise, control flow is transferred 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 ]
```

‘`indirectbr`‘ Instruction

Syntax:

```
indirectbr <somety>* <address>, [ label <dest1>, label <dest2>, ... ]
```

Overview:

The '`indirectbr`' instruction implements an indirect branch to a label within the current function, whose address is specified by "`address`". Address must be derived from a *blockaddress* constant.

Arguments:

The '`address`' argument is the address of the label to jump to. The rest of the arguments indicate the full set of possible destinations that the address may point to. Blocks are allowed to occur multiple times in the destination list, though this isn't particularly useful.

This destination list is required so that dataflow analysis has an accurate understanding of the CFG.

Semantics:

Control transfers to the block specified in the address argument. All possible destination blocks must be listed in the label list, otherwise this instruction has undefined behavior. This implies that jumps to labels defined in other functions have undefined behavior as well.

Implementation:

This is typically implemented with a jump through a register.

Example:

```
indirectbr i8* %Addr, [ label %bb1, label %bb2, label %bb3 ]
```

'`invoke`' Instruction

Syntax:

```
<result> = invoke [cconv] [ret attrs] <ptr to function ty> <function ptr val>(<function args>)
             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 via the "*resume*" instruction or other exception handling mechanism, control is interrupted and continued at the dynamically nearest "exception" label.

The '`exception`' label is a landing pad for the exception. As such, '`exception`' label is required to have the "*landingpad*" instruction, which contains the information about the behavior of the program after unwinding happens, as its first non-PHI instruction. The restrictions on the "`landingpad`" instruction's tightly couples it to the "`invoke`" instruction, so that the important information contained within the "`landingpad`" instruction can't be lost through normal code motion.

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 and parameter attributes. 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.
6. '`normal label`': the label reached when the called function executes a '`ret`' instruction.
7. '`exception label`': the label reached when a callee returns via the *resume* instruction or other exception handling mechanism.
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
```

'`resume`' Instruction

Syntax:

```
resume <type> <value>
```

Overview:

The '`resume`' instruction is a terminator instruction that has no successors.

Arguments:

The '`resume`' instruction requires one argument, which must have the same type as the result of any '`landingpad`' instruction in the same function.

Semantics:

The '`resume`' instruction resumes propagation of an existing (in-flight) exception whose unwinding was interrupted with a *landingpad* instruction.

Example:

```
resume { i8*, i32 } %exn
```

'`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 a *poison value* if unsigned and/or signed overflow, respectively, occurs.

Example:

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

'`fadd`' Instruction

Syntax:

```
<result> = fadd [fast-math flags]* <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. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

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 a *poison value* 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 [fast-math flags]* <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. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

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` * `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 a *poison value* if unsigned and/or signed overflow, respectively, occurs.

Example:

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

'`fmul`' Instruction

Syntax:

```
<result> = fmul [fast-math flags]* <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. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

Example:

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

'`udiv`' Instruction

Syntax:

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

If the `exact` keyword is present, the result value of the `udiv` is a *poison value* if %op1 is not a multiple of %op2 (as such, "((a udiv exact b) mul b) == a").

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 a *poison value* if the result would be rounded.

Example:

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

'`fdiv`' Instruction

Syntax:

```
<result> = fdiv [fast-math flags]* <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. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

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 is either zero or has the same sign as the dividend, `op1`), not the *modulo* operator (where the result is either zero or 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 [fast-math flags]* <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. This instruction can also take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations:

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

If the `nuw` keyword is present, then the shift produces a *poison value* if it shifts out any non-zero bits. If the `nsw` keyword is present, then the shift produces a *poison value* if it shifts out any bits that disagree with the resultant sign bit. As such, NUW/NSW have the same semantics as they would if the shift were expressed as a mul instruction with the same nsw/nuw bits in (mul %op1, (shl 1, %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,
```

'`lshr`' Instruction

Syntax:

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

If the `exact` keyword is present, the result value of the `lshr` is a *poison value* if any of the bits shifted out are non-zero.

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 = 0x7F
<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
```

'`ashr`' Instruction

Syntax:

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

If the `exact` keyword is present, the result value of the `ashr` is a *poison value* if any of the bits shifted out are non-zero.

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
```

'`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 <t
```

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 the same type. 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 s
<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 >  ;
```

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 member field 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.

The major differences to `getelementptr` indexing are:

- Since the value being indexed is not a pointer, the first index is omitted and assumed to be zero.
- At least one index must be specified.
- Not only struct indices but also array indices must be in bounds.

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> <elt>, <idx>{, <idx>}*    ; yields <aggrega
```

Overview:

The '`insertvalue`' instruction inserts a value into a member field in an *aggregate* value.

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 '`extractvalue`' 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:

```
%agg1 = insertvalue {i32, float} undef, i32 1, 0              ; yields {i32 1, float undef}
%agg2 = insertvalue {i32, float} %agg1, float %val, 1         ; yields {i32 1, float %val}
%agg3 = insertvalue {i32, {float}} %agg1, float %val, 1, 0    ; yields {i32 1, float %val}
```

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, and allocate memory in LLVM.

'`alloca`' Instruction

Syntax:

```
<result> = alloca <type>[, <ty> <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 `resume` instructions), the memory is reclaimed. Allocating zero bytes is legal, but the result is undefined. The order in which memory is allocated (ie., which way the stack grows) is not specified.

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 [volatile] <ty>* <pointer>[, align <alignment>][, !nontemporal !<index>][, !inv
<result> = load atomic [volatile] <ty>* <pointer> [singlethread] <ordering>, align <alignment>
!<index> = !{ i32 1 }
```

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 operations*.

If the `load` is marked as `atomic`, it takes an extra *ordering* and optional `singlethread` argument. The `release` and `acq_rel` orderings are not valid on `load` instructions. Atomic loads produce *defined* results when they may see multiple atomic stores. The type of the pointee must be an integer type whose bit width is a power of two greater than or equal to eight and less than or equal to a target-specific size limit. `align` must be explicitly specified on atomic loads, and the load has undefined behavior if the alignment is not set to a value which is at least the size in bytes of the pointee. `!nontemporal` does not have any defined semantics for atomic loads.

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 ABI 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 undefined behavior. Underestimating the alignment may produce less efficient code. An alignment of 1 is always safe.

The optional `!nontemporal` metadata must reference a single metadata name `<index>` corresponding to a metadata node with one `i32` entry of value 1. The existence of the `!nontemporal` metadata on the instruction tells the optimizer and code generator that this load is not expected to be reused in the cache. The code generator may select special instructions to save cache bandwidth, such as the `MOVNT` instruction on x86.

The optional `!invariant.load` metadata must reference a single metadata name `<index>` corresponding to a metadata node with no entries. The existence of the `!invariant.load` metadata on the instruction tells the optimizer and code generator that this load address points to memory which does not change value during program execution. The optimizer may then move this load around, for example, by hoisting it out of loops using loop invariant code motion.

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 [volatile] <ty> <value>, <ty>* <pointer>[, align <alignment>][, !nontemporal !<index>]
store atomic [volatile] <ty> <value>, <ty>* <pointer> [singlethread] <ordering>, align <alignme
```

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 operations*.

If the `store` is marked as `atomic`, it takes an extra *ordering* and optional `singlethread` argument. The `acquire` and `acq_rel` orderings aren't valid on `store` instructions. Atomic loads produce *defined* results when they may see multiple atomic stores. The type of the pointee must be an integer type whose bit width is a power of two greater than or equal to eight and less than or equal to a target-specific size

limit. `align` must be explicitly specified on atomic stores, and the store has undefined behavior if the alignment is not set to a value which is at least the size in bytes of the pointee. `!nontemporal` does not have any defined semantics for atomic stores.

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 ABI 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 undefined behavior. Underestimating the alignment may produce less efficient code. An alignment of 1 is always safe.

The optional `!nontemporal` metadata must reference a single metadata name `<index>` corresponding to a metadata node with one `i32` entry of value 1. The existence of the `!nontemporal` metadata on the instruction tells the optimizer and code generator that this load is not expected to be reused in the cache. The code generator may select special instructions to save cache bandwidth, such as the MOVNT instruction on x86.

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
```

‘fence‘ Instruction

Syntax:

```
fence [singlethread] <ordering>                   ; yields {void}
```

Overview:

The ‘`fence`‘ instruction is used to introduce happens-before edges between operations.

Arguments:

‘`fence`‘ instructions take an *ordering* argument which defines what *synchronizes-with* edges they add. They can only be given `acquire`, `release`, `acq_rel`, and `seq_cst` orderings.

Semantics:

A fence A which has (at least) `release` ordering semantics *synchronizes with* a fence B with (at least) `acquire` ordering semantics if and only if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M (either directly or through some side effect of a sequence headed by X), Y is sequenced before B, and Y observes M. This provides a *happens-before* dependency between A and B. Rather than an explicit `fence`, one (but not

both) of the atomic operations X or Y might provide a `release` or `acquire` (resp.) ordering constraint and still *synchronize-with* the explicit `fence` and establish the *happens-before* edge.

A `fence` which has `seq_cst` ordering, in addition to having both `acquire` and `release` semantics specified above, participates in the global program order of other `seq_cst` operations and/or fences.

The optional "*singlethread*" argument specifies that the fence only synchronizes with other fences in the same thread. (This is useful for interacting with signal handlers.)

Example:

```
fence acquire                          ; yields {void}
fence singlethread seq_cst             ; yields {void}
```

'`cmpxchg`' Instruction

Syntax:

```
cmpxchg [volatile] <ty>* <pointer>, <ty> <cmp>, <ty> <new> [singlethread] <ordering>  ; yields
```

Overview:

The '`cmpxchg`' instruction is used to atomically modify memory. It 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:

There are three arguments to the '`cmpxchg`' instruction: an address to operate on, a value to compare to the value currently be at that address, and a new value to place at that address if the compared values are equal. The type of '<cmp>' must be an integer type whose bit width is a power of two greater than or equal to eight and less than or equal to a target-specific size limit. '<cmp>' and '<new>' must have the same type, and the type of '<pointer>' must be a pointer to that type. If the `cmpxchg` is marked as `volatile`, then the optimizer is not allowed to modify the number or order of execution of this `cmpxchg` with other *volatile operations*.

The *ordering* argument specifies how this `cmpxchg` synchronizes with other atomic operations.

The optional "`singlethread`" argument declares that the `cmpxchg` is only atomic with respect to code (usually signal handlers) running in the same thread as the `cmpxchg`. Otherwise the cmpxchg is atomic with respect to all other code in the system.

The pointer passed into cmpxchg must have alignment greater than or equal to the size in memory of the operand.

Semantics:

The contents of memory at the location specified by the '`<pointer>`' operand is read and compared to '`<cmp>`'; if the read value is the equal, '`<new>`' is written. The original value at the location is returned.

A successful `cmpxchg` is a read-modify-write instruction for the purpose of identifying release sequences. A failed `cmpxchg` is equivalent to an atomic load with an ordering parameter determined by dropping any `release` part of the `cmpxchg`'s ordering.

Example:

```
entry:
  %orig = atomic load i32* %ptr unordered                     ; yields {i32}
  br label %loop

loop:
  %cmp = phi i32 [ %orig, %entry ], [%old, %loop]
  %squared = mul i32 %cmp, %cmp
  %old = cmpxchg i32* %ptr, i32 %cmp, i32 %squared          ; yields {i32}
  %success = icmp eq i32 %cmp, %old
  br i1 %success, label %done, label %loop

done:
  ...
```

'`atomicrmw`' Instruction

Syntax:

```
atomicrmw [volatile] <operation> <ty>* <pointer>, <ty> <value> [singlethread] <ordering>
```

Overview:

The '`atomicrmw`' instruction is used to atomically modify memory.

Arguments:

There are three arguments to the '`atomicrmw`' instruction: an operation to apply, an address whose value to modify, an argument to the operation. The operation must be one of the following keywords:

- xchg
- add
- sub
- and
- nand
- or
- xor
- max
- min
- umax
- umin

The type of '<value>' must be an integer type whose bit width is a power of two greater than or equal to eight and less than or equal to a target-specific size limit. The type of the '`<pointer>`' operand must be a pointer to that type. If the `atomicrmw` is marked as `volatile`, then the optimizer is not allowed to modify the number or order of execution of this `atomicrmw` with other *volatile operations*.

Semantics:

The contents of memory at the location specified by the '`<pointer>`' operand are atomically read, modified, and written back. The original value at the location is returned. The modification is specified by the operation argument:

- xchg: `*ptr = val`
- add: `*ptr = *ptr + val`
- sub: `*ptr = *ptr - val`

- and: `*ptr = *ptr & val`
- nand: `*ptr = ~(*ptr & val)`
- or: `*ptr = *ptr | val`
- xor: `*ptr = *ptr ^ val`
- max: `*ptr = *ptr > val ? *ptr : val` (using a signed comparison)
- min: `*ptr = *ptr < val ? *ptr : val` (using a signed comparison)
- umax: `*ptr = *ptr > val ? *ptr : val` (using an unsigned comparison)
- umin: `*ptr = *ptr < val ? *ptr : val` (using an unsigned comparison)

Example:

```
%old = atomicrmw add i32* %ptr, i32 1 acquire                    ; yields {i32}
```

‘`getelementptr`‘ Instruction

Syntax:

```
<result> = getelementptr <pty>* <ptrval>{, <ty> <idx>}*
<result> = getelementptr inbounds <pty>* <ptrval>{, <ty> <idx>}*
<result> = getelementptr <ptr vector> ptrval, <vector index type> 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 or a vector of pointers, 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 using a vector of indices they must all be the **same** `i32` integer constant). When indexing into an array, pointer or vector, integers of any width are allowed, and they are not required to be constant. These integers are treated as signed values where relevant.

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 Clang is:

```
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define i32* @foo(%struct.ST* %s) nounwind uwtable readnone optsize ssp {
entry:
  %arrayidx = getelementptr inbounds %struct.ST* %s, i64 1, i32 2, i32 1, i64 5, i64 13
  ret i32* %arrayidx
}
```

Semantics:

In the example above, the first index is indexing into the '`%struct.ST*`' type, which is a pointer, yielding a '`%struct.ST`' = '`{ i32, double, %struct.RT }`' type, a structure. The second index indexes into the third element of the structure, yielding a '`%struct.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(%struct.ST* %s) {
  %t1 = getelementptr %struct.ST* %s, i32 1                        ; yields %struct.ST*:%t1
  %t2 = getelementptr %struct.ST* %t1, i32 0, i32 2                ; yields %struct.RT*:%t2
  %t3 = getelementptr %struct.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 a *poison value* 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 signed 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. In cases where the base is a vector of pointers the `inbounds` keyword applies to each of the computations element-wise.

If the `inbounds` keyword is not present, the offsets are added to the base address with silently-wrapping two's complement arithmetic. If the offsets have a different width from the pointer, they are sign-extended or truncated to the width of the pointer. 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
```

In cases where the pointer argument is a vector of pointers, each index must be a vector with the same number of elements. For example:

```
%A = getelementptr <4 x i8*> %ptrs, <4 x i64> %offsets,
```

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, and a type to trunc it to. Both types must be of *integer* types, or vectors of the same number of integers. The bit size of the `value` must be larger than the bit size of the destination type, `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
%Z = trunc i32 122 to i1                         ; yields i1:false
%W = trunc <2 x i16> <i16 8, i16 7> to <2 x i8> ; yields <i8 8, i8 7>
```

'`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, and a type to cast it to. Both types must be of *integer* types, or vectors of the same number of integers. 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
%Z = zext <2 x i16> <i16 8, i16 7> to <2 x i32> ; yields <i32 8, i32 7>
```

'`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, and a type to cast it to. Both types must be of *integer* types, or vectors of the same number of integers. 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
%Z = sext <2 x i16> <i16 8, i16 7> to <2 x i32> ; yields <i32 8, i32 7>
```

'`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 3.125 to double         ; yields double:3.125000e+00
%Y = fpext double %X to fp128            ; yields fp128:0xL00000000000000004000900000000000
```

'`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
%Z = 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
%Z = 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 or a vector of pointers `value` to the integer (or vector of integers) type `ty2`.

Arguments:

The ‘`ptrtoint`‘ instruction takes a `value` to cast, which must be a a value of type *pointer* or a vector of pointers, and a type to cast it to `ty2`, which must be an *integer* or a vector of integers 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* %P to i8                         ; yields truncation on 32-bit architecture
%Y = ptrtoint i32* %P to i64                        ; yields zero extension on 32-bit architect
%Z = ptrtoint <4 x i32*> %P to <4 x i64>; yields vector zero extension for a vector of addresse
```

‘`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
%Y = inttoptr i32 255 to i32*          ; yields no-op on 32-bit architecture
%Z = inttoptr i64 0 to i32*            ; yields truncation on 32-bit architecture
%Z = inttoptr <4 x i32> %G to <4 x i8*>; yields truncation of vector G to four pointers
```

'`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 of the same size. 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 (or vector of pointers) types may only be converted to other pointer (or vector of pointers) types with the same address space through 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
%Z = bitcast <2 x i32*> %V to <2 x i64*> ; yields <2 x i64*>
```

'`addrspacecast .. to`' Instruction

Syntax:

```
<result> = addrspacecast <pty> <ptrval> to <pty2>       ; yields pty2
```

Overview:

The '`addrspacecast`' instruction converts `ptrval` from `pty` in address space `n` to type `pty2` in address space `m`.

Arguments:

The '`addrspacecast`' instruction takes a pointer or vector of pointer value to cast and a pointer type to cast it to, which must have a different address space.

Semantics:

The '`addrspacecast`' instruction converts the pointer value `ptrval` to type `pty2`. It can be a *no-op cast* or a complex value modification, depending on the target and the address space pair. Pointer conversions within the same address space must be performed with the `bitcast` instruction. Note that if the address space conversion is legal then both result and operand refer to the same memory location.

Example:

```
%X = addrspacecast i32* %x to i32 addrspace(1)*    ; yields i32 addrspace(1)*:%x
%Y = addrspacecast i32 addrspace(1)* %y to i64 addrspace(2)*    ; yields i64 addrspace(2)*:%y
%Z = addrspacecast <4 x i32*> %z to <4 x float addrspace(3)*>   ; yields <4 x float addrspace(3
```

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, pointer, or pointer vector 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 greater 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
```

'`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 that the callee function does not access any allocas or varargs in the caller. Note that calls may be marked "tail" even if they do not occur before a *ret* instruction. If the "tail" marker is present, the function call is eligible for tail call optimization, but might not in fact be optimized into a jump. The code generator may optimize calls marked "tail" with either 1) automatic sibling call optimization when the caller and callee have matching signatures, or 2) forced tail call optimization when the following extra requirements are met:
   - Caller and callee both have the calling convention `fastcc`.
   - The call is in tail position (ret immediately follows call and ret uses value of call or is void).
   - Option `-tailcallopt` is enabled, or `llvm::GuaranteedTailCallOpt` is `true`.
   - Platform specific constraints are met.
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. The calling convention of the call must match the calling convention of the target function, or else the behavior is undefined.
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 and parameter attributes. 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
```

llvm treats calls to some functions with names and arguments that match the standard C99 library as being the C99 library functions, and may perform optimizations or generate code for them under that assumption. This is something we'd like to change in the future to provide better support for freestanding environments and non-C-based languages.

'`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.

'`landingpad`' Instruction

Syntax:

```
<resultval> = landingpad <resultty> personality <type> <pers_fn> <clause>+
<resultval> = landingpad <resultty> personality <type> <pers_fn> cleanup <clause>*

<clause> := catch <type> <value>
<clause> := filter <array constant type> <array constant>
```

Overview:

The '`landingpad`' instruction is used by LLVM's exception handling system to specify that a basic block is a landing pad — one where the exception lands, and corresponds to the code found in the `catch` portion of a `try`/`catch` sequence. It defines values supplied by the personality function (`pers_fn`) upon re-entry to the function. The `resultval` has the type `resultty`.

Arguments:

This instruction takes a `pers_fn` value. This is the personality function associated with the unwinding mechanism. The optional `cleanup` flag indicates that the landing pad block is a cleanup.

A `clause` begins with the clause type — `catch` or `filter` — and contains the global variable representing the "type" that may be caught or filtered respectively. Unlike the `catch` clause, the `filter` clause takes an array constant as its argument. Use "`[0 x i8**] undef`" for a filter which cannot throw. The '`landingpad`' instruction must contain *at least* one `clause` or the `cleanup` flag.

Semantics:

The '`landingpad`' instruction defines the values which are set by the personality function (`pers_fn`) upon re-entry to the function, and therefore the "result type" of the `landingpad` instruction. As with calling conventions, how the personality function results are represented in LLVM IR is target specific.

The clauses are applied in order from top to bottom. If two `landingpad` instructions are merged together through inlining, the clauses from the calling function are appended to the list of clauses. When the call stack is being unwound due to an exception being thrown, the exception is compared against each `clause` in turn. If it doesn't match any of the clauses, and the `cleanup` flag is not set, then

unwinding continues further up the call stack.

The `landingpad` instruction has several restrictions:

- A landing pad block is a basic block which is the unwind destination of an '`invoke`' instruction.
- A landing pad block must have a '`landingpad`' instruction as its first non-PHI instruction.
- There can be only one '`landingpad`' instruction within the landing pad block.
- A basic block that is not a landing pad block may not include a '`landingpad`' instruction.
- All '`landingpad`' instructions in a function must have the same personality function.

Example:

```
;; A landing pad which can catch an integer.
%res = landingpad { i8*, i32 } personality i32 (...)* @__gxx_personality_v0
         catch i8** @_ZTIi
;; A landing pad that is a cleanup.
%res = landingpad { i8*, i32 } personality i32 (...)* @__gxx_personality_v0
         cleanup
;; A landing pad which can catch an integer and can only throw a double.
%res = landingpad { i8*, i32 } personality i32 (...)* @__gxx_personality_v0
         catch i8** @_ZTIi
         filter [1 x i8**] [@_ZTId]
```

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>, i32 <cache type>)
```

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 `cache type` specifies whether the prefetch is performed on the data (1) or instruction (0) cache. The `rw`, `locality` and `cache type` 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 intrinsic 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.

Note that runtime support may be conditional on the privilege-level code is running at and the host platform.

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 and for different address spaces. Not all targets support all bit widths however.

```
declare void @llvm.memcpy.p0i8.p0i8.i32(i8* <dest>, i8* <src>,
                                        i32 <len>, i32 <align>, i1 <isvolatile>)
declare void @llvm.memcpy.p0i8.p0i8.i64(i8* <dest>, i8* <src>,
                                        i64 <len>, i32 <align>, i1 <isvolatile>)
```

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, takes extra alignment/isvolatile arguments and the pointers can be in specified address spaces.

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, the fourth argument is the alignment of the source and destination locations, and the fifth is a boolean indicating a volatile access.

If the call to this intrinsic 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.

If the `isvolatile` parameter is `true`, the `llvm.memcpy` call is a *volatile operation*. The detailed access behavior is not very cleanly specified and it is unwise to depend on it.

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 (both meaning no alignment).

'`llvm.memmove`' Intrinsic

Syntax:

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

```
declare void @llvm.memmove.p0i8.p0i8.i32(i8* <dest>, i8* <src>,
                                         i32 <len>, i32 <align>, i1 <isvolatile>)
declare void @llvm.memmove.p0i8.p0i8.i64(i8* <dest>, i8* <src>,
                                         i64 <len>, i32 <align>, i1 <isvolatile>)
```

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, takes extra alignment/isvolatile arguments and the pointers can be in specified address spaces.

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, the fourth argument is the alignment of the source and destination locations, and the fifth is a boolean indicating a volatile access.

If the call to this intrinsic 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.

If the `isvolatile` parameter is `true`, the `llvm.memmove` call is a *volatile operation*. The detailed access behavior is not very cleanly specified and it is unwise to depend on it.

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 (both meaning no alignment).

'`llvm.memset.*`' Intrinsics

Syntax:

This is an overloaded intrinsic. You can use llvm.memset on any integer bit width and for different address spaces. However, not all targets support all bit widths.

```
declare void @llvm.memset.p0i8.i32(i8* <dest>, i8 <val>,
                                   i32 <len>, i32 <align>, i1 <isvolatile>)
declare void @llvm.memset.p0i8.i64(i8* <dest>, i8 <val>,
                                   i64 <len>, i32 <align>, i1 <isvolatile>)
```

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 extra alignment/volatile arguments. Also, the destination can be in an arbitrary address space.

Arguments:

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

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

If the `isvolatile` parameter is `true`, the `llvm.memset` call is a *volatile operation*. The detailed access behavior is not very cleanly specified and it is unwise to depend on it.

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 (both meaning no alignment).

'`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.

‘`llvm.exp.*`‘ Intrinsic

Syntax:

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

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

Overview:

The ‘`llvm.exp.*`‘ intrinsics perform the exp function.

Arguments:

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

Semantics:

This function returns the same values as the libm `exp` functions would, and handles error conditions in the same way.

‘`llvm.exp2.*`‘ Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.exp2` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.exp2.f32(float  %Val)
declare double    @llvm.exp2.f64(double %Val)
declare x86_fp80  @llvm.exp2.f80(x86_fp80  %Val)
declare fp128     @llvm.exp2.f128(fp128 %Val)
declare ppc_fp128 @llvm.exp2.ppcf128(ppc_fp128  %Val)
```

Overview:

The ‘`llvm.exp2.*`‘ intrinsics perform the exp2 function.

Arguments:

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

Semantics:

This function returns the same values as the libm `exp2` functions would, and handles error conditions

in the same way.

'`llvm.log.*`' Intrinsic

Syntax:

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

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

Overview:

The '`llvm.log.*`' intrinsics perform the log function.

Arguments:

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

Semantics:

This function returns the same values as the libm `log` functions would, and handles error conditions in the same way.

'`llvm.log10.*`' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.log10` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.log10.f32(float  %Val)
declare double    @llvm.log10.f64(double %Val)
declare x86_fp80  @llvm.log10.f80(x86_fp80  %Val)
declare fp128     @llvm.log10.f128(fp128 %Val)
declare ppc_fp128 @llvm.log10.ppcf128(ppc_fp128  %Val)
```

Overview:

The '`llvm.log10.*`' intrinsics perform the log10 function.

Arguments:

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

Semantics:

This function returns the same values as the libm `log10` functions would, and handles error conditions in the same way.

'`llvm.log2.*`' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.log2` on any floating point or vector of floating point type. Not all targets support all types however.

```
declare float     @llvm.log2.f32(float  %Val)
declare double    @llvm.log2.f64(double %Val)
declare x86_fp80  @llvm.log2.f80(x86_fp80  %Val)
declare fp128     @llvm.log2.f128(fp128 %Val)
declare ppc_fp128 @llvm.log2.ppcf128(ppc_fp128  %Val)
```

Overview:

The '`llvm.log2.*`' intrinsics perform the log2 function.

Arguments:

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

Semantics:

This function returns the same values as the libm `log2` functions would, and handles error conditions in the same way.

'`llvm.fma.*`' Intrinsic

Syntax:

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

```
declare float     @llvm.fma.f32(float  %a, float  %b, float  %c)
declare double    @llvm.fma.f64(double %a, double %b, double %c)
declare x86_fp80  @llvm.fma.f80(x86_fp80 %a, x86_fp80 %b, x86_fp80 %c)
declare fp128     @llvm.fma.f128(fp128 %a, fp128 %b, fp128 %c)
declare ppc_fp128 @llvm.fma.ppcf128(ppc_fp128 %a, ppc_fp128 %b, ppc_fp128 %c)
```

Overview:

The '`llvm.fma.*`' intrinsics perform the fused multiply-add operation.

Arguments:

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

Semantics:

This function returns the same values as the libm `fma` functions would.

'`llvm.fabs.*`' Intrinsic

Syntax:

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

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

Overview:

The '`llvm.fabs.*`' intrinsics return the absolute value of the operand.

Arguments:

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

Semantics:

This function returns the same values as the libm `fabs` functions would, and handles error conditions in the same way.

'`llvm.copysign.*`' Intrinsic

Syntax:

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

```
declare float     @llvm.copysign.f32(float  %Mag, float  %Sgn)
declare double    @llvm.copysign.f64(double %Mag, double %Sgn)
declare x86_fp80  @llvm.copysign.f80(x86_fp80  %Mag, x86_fp80  %Sgn)
declare fp128     @llvm.copysign.f128(fp128 %Mag, fp128 %Sgn)
declare ppc_fp128 @llvm.copysign.ppcf128(ppc_fp128  %Mag, ppc_fp128  %Sgn)
```

Overview:

The '`llvm.copysign.*`' intrinsics return a value with the magnitude of the first operand and the sign of the second operand.

Arguments:

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

Semantics:

This function returns the same values as the libm `copysign` functions would, and handles error conditions in the same way.

'`llvm.floor.*`' Intrinsic

Syntax:

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

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

Overview:

The '`llvm.floor.*`' intrinsics return the floor of the operand.

Arguments:

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

Semantics:

This function returns the same values as the libm `floor` functions would, and handles error conditions in the same way.

'`llvm.ceil.*`' Intrinsic

Syntax:

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

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

Overview:

The '`llvm.ceil.*`' intrinsics return the ceiling of the operand.

Arguments:

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

Semantics:

This function returns the same values as the libm `ceil` functions would, and handles error conditions in the same way.

'`llvm.trunc.*`' Intrinsic

Syntax:

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

```
declare float     @llvm.trunc.f32(float  %Val)
declare double    @llvm.trunc.f64(double %Val)
declare x86_fp80  @llvm.trunc.f80(x86_fp80  %Val)
```

```
declare fp128     @llvm.trunc.f128(fp128 %Val)
declare ppc_fp128 @llvm.trunc.ppcf128(ppc_fp128  %Val)
```

Overview:

The '`llvm.trunc.*`' intrinsics returns the operand rounded to the nearest integer not larger in magnitude than the operand.

Arguments:

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

Semantics:

This function returns the same values as the libm `trunc` functions would, and handles error conditions in the same way.

'`llvm.rint.*`' Intrinsic

Syntax:

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

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

Overview:

The '`llvm.rint.*`' intrinsics returns the operand rounded to the nearest integer. It may raise an inexact floating-point exception if the operand isn't an integer.

Arguments:

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

Semantics:

This function returns the same values as the libm `rint` functions would, and handles error conditions in the same way.

'`llvm.nearbyint.*`' Intrinsic

Syntax:

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

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

```
declare ppc_fp128 @llvm.nearbyint.ppcf128(ppc_fp128  %Val)
```

Overview:

The '`llvm.nearbyint.*`' intrinsics returns the operand rounded to the nearest integer.

Arguments:

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

Semantics:

This function returns the same values as the libm `nearbyint` functions would, and handles error conditions in the same way.

'`llvm.round.*`' Intrinsic

Syntax:

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

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

Overview:

The '`llvm.round.*`' intrinsics returns the operand rounded to the nearest integer.

Arguments:

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

Semantics:

This function returns the same values as the libm `round` 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, or on any vector with integer elements. Not all targets support all bit widths or vector types, 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>)
declare <2 x i32> @llvm.ctpop.v2i32(<2 x i32> <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, or a vector with integer elements. The return type must match the argument type.

Semantics:

The '`llvm.ctpop`' intrinsic counts the 1's in a variable, or within each element of a vector.

'`llvm.ctlz.*`' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use `llvm.ctlz` on any integer bit width, or any vector whose elements are integers. Not all targets support all bit widths or vector types, however.

```
declare i8   @llvm.ctlz.i8  (i8   <src>, i1 <is_zero_undef>)
declare i16  @llvm.ctlz.i16 (i16  <src>, i1 <is_zero_undef>)
declare i32  @llvm.ctlz.i32 (i32  <src>, i1 <is_zero_undef>)
declare i64  @llvm.ctlz.i64 (i64  <src>, i1 <is_zero_undef>)
```

```
declare i256 @llvm.ctlz.i256(i256 <src>, i1 <is_zero_undef>)
declase <2 x i32> @llvm.ctlz.v2i32(<2 x i32> <src>, i1 <is_zero_undef>)
```

Overview:

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

Arguments:

The first argument is the value to be counted. This argument may be of any integer type, or a vectory with integer element type. The return type must match the first argument type.

The second argument must be a constant and is a flag to indicate whether the intrinsic should ensure that a zero as the first argument produces a defined result. Historically some architectures did not provide a defined result for zero values as efficiently, and many algorithms are now predicated on avoiding zero-value inputs.

Semantics:

The '`llvm.ctlz`' intrinsic counts the leading (most significant) zeros in a variable, or within each element of the vector. If `src == 0` then the result is the size in bits of the type of `src` if `is_zero_undef == 0` and `undef` otherwise. 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, or any vector of integer elements. Not all targets support all bit widths or vector types, however.

```
declare i8   @llvm.cttz.i8  (i8   <src>, i1 <is_zero_undef>)
declare i16  @llvm.cttz.i16 (i16  <src>, i1 <is_zero_undef>)
declare i32  @llvm.cttz.i32 (i32  <src>, i1 <is_zero_undef>)
declare i64  @llvm.cttz.i64 (i64  <src>, i1 <is_zero_undef>)
declare i256 @llvm.cttz.i256(i256 <src>, i1 <is_zero_undef>)
declase <2 x i32> @llvm.cttz.v2i32(<2 x i32> <src>, i1 <is_zero_undef>)
```

Overview:

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

Arguments:

The first argument is the value to be counted. This argument may be of any integer type, or a vectory with integer element type. The return type must match the first argument type.

The second argument must be a constant and is a flag to indicate whether the intrinsic should ensure that a zero as the first argument produces a defined result. Historically some architectures did not provide a defined result for zero values as efficiently, and many algorithms are now predicated on avoiding zero-value inputs.

Semantics:

The '`llvm.cttz`' intrinsic counts the trailing (least significant) zeros in a variable, or within each element of a vector. If `src == 0` then the result is the size in bits of the type of `src` if `is_zero_undef ==`

`0` and `undef` otherwise. 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
```

Specialised Arithmetic Intrinsics

'`llvm.fmuladd.*`' Intrinsic

Syntax:

```
declare float @llvm.fmuladd.f32(float %a, float %b, float %c)
declare double @llvm.fmuladd.f64(double %a, double %b, double %c)
```

Overview:

The '`llvm.fmuladd.*`' intrinsic functions represent multiply-add expressions that can be fused if the code generator determines that (a) the target instruction set has support for a fused operation, and (b) that the fused operation is more efficient than the equivalent, separate pair of mul and add instructions.

Arguments:

The '`llvm.fmuladd.*`' intrinsics each take three arguments: two multiplicands, a and b, and an addend c.

Semantics:

The expression:

```
%0 = call float @llvm.fmuladd.f32(%a, %b, %c)
```

is equivalent to the expression a * b + c, except that rounding will not be performed between the multiplication and addition steps if the code generator fuses the operations. Fusion is not guaranteed, even if the target platform supports it. If a fused multiply-add is required the corresponding llvm.fma.* intrinsic function should be used instead.

Examples:

```
%r2 = call float @llvm.fmuladd.f32(float %a, float %b, float %c) ; yields {float}:r2 = (a * b)
```

Half Precision Floating Point Intrinsics

For most target platforms, half precision floating point is a storage-only format. This means that it is a dense encoding (in memory) but does not support computation in the format.

This means that code must first load the half-precision floating point value as an i16, then convert it to float with *llvm.convert.from.fp16*. Computation can then be performed on the float value (including extending to double etc). To store the value back to memory, it is first converted to float if needed, then converted to i16 with *llvm.convert.to.fp16*, then storing as an i16 value.

'`llvm.convert.to.fp16`' Intrinsic

Syntax:

```
declare i16 @llvm.convert.to.fp16(f32 %a)
```

Overview:

The '`llvm.convert.to.fp16`' intrinsic function performs a conversion from single precision floating point format to half precision floating point format.

Arguments:

The intrinsic function contains single argument - the value to be converted.

Semantics:

The '`llvm.convert.to.fp16`' intrinsic function performs a conversion from single precision floating point format to half precision floating point format. The return value is an `i16` which contains the converted number.

Examples:

```
%res = call i16 @llvm.convert.to.fp16(f32 %a)
store i16 %res, i16* @x, align 2
```

'`llvm.convert.from.fp16`' Intrinsic

Syntax:

```
declare f32 @llvm.convert.from.fp16(i16 %a)
```

Overview:

The '`llvm.convert.from.fp16`' intrinsic function performs a conversion from half precision floating point format to single precision floating point format.

Arguments:

The intrinsic function contains single argument - the value to be converted.

Semantics:

The '`llvm.convert.from.fp16`' intrinsic function performs a conversion from half single precision floating point format to single precision floating point format. The input half-float value is represented by an `i16` value.

Examples:

```
%a = load i16* @x, align 2
%res = call f32 @llvm.convert.from.fp16(i16 %a)
```

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 Intrinsics

These intrinsics make 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
call i8* @llvm.init.trampoline(i8* %tramp1, i8* bitcast (i32 (i8*, i32, i32)* @f to i8*), i8* %
%p = call i8* @llvm.adjust.trampoline(i8* %tramp1)
%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 void @llvm.init.trampoline(i8* <tramp>, i8* <func>, i8* <nval>)
```

Overview:

This fills the memory pointed to by `tramp` with executable code, turning it into a trampoline.

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. Then `tramp` needs to be passed to *llvm.adjust.trampoline* to get a pointer which can be *bitcast (to a new function) and 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.

'`llvm.adjust.trampoline`' Intrinsic

Syntax:

```
declare i8* @llvm.adjust.trampoline(i8* <tramp>)
```

Overview:

This performs any required machine-specific adjustment to the address of a trampoline (passed as `tramp`).

Arguments:

`tramp` must point to a block of memory which already has trampoline code filled in by a previous call to *llvm.init.trampoline*.

Semantics:

On some architectures the address of the code to be executed needs to be different to the address where the trampoline is actually stored. This intrinsic returns the executable address corresponding to `tramp` after performing the required machine specific adjustments. The pointer returned can then be *bitcast and executed*.

Memory Use Markers

This class of intrinsics exists to information about the lifetime of memory objects and ranges where variables are immutable.

'`llvm.lifetime.start`' Intrinsic

Syntax:

```
declare void @llvm.lifetime.start(i64 <size>, i8* nocapture <ptr>)
```

Overview:

The '`llvm.lifetime.start`' intrinsic specifies the start of a memory object's lifetime.

Arguments:

The first argument is a constant integer representing the size of the object, or -1 if it is variable sized. The second argument is a pointer to the object.

Semantics:

This intrinsic indicates that before this point in the code, the value of the memory pointed to by `ptr` is dead. This means that it is known to never be used and has an undefined value. A load from the pointer that precedes this intrinsic can be replaced with `'undef'`.

'`llvm.lifetime.end`' Intrinsic

Syntax:

```
declare void @llvm.lifetime.end(i64 <size>, i8* nocapture <ptr>)
```

Overview:

The '`llvm.lifetime.end`' intrinsic specifies the end of a memory object's lifetime.

Arguments:

The first argument is a constant integer representing the size of the object, or -1 if it is variable sized. The second argument is a pointer to the object.

Semantics:

This intrinsic indicates that after this point in the code, the value of the memory pointed to by `ptr` is dead. This means that it is known to never be used and has an undefined value. Any stores into the memory object following this intrinsic may be removed as dead.

'`llvm.invariant.start`' Intrinsic

Syntax:

```
declare {}* @llvm.invariant.start(i64 <size>, i8* nocapture <ptr>)
```

Overview:

The '`llvm.invariant.start`' intrinsic specifies that the contents of a memory object will not change.

Arguments:

The first argument is a constant integer representing the size of the object, or -1 if it is variable sized. The second argument is a pointer to the object.

Semantics:

This intrinsic indicates that until an `llvm.invariant.end` that uses the return value, the referenced memory location is constant and unchanging.

'`llvm.invariant.end`' Intrinsic

Syntax:

```
declare void @llvm.invariant.end({}* <start>, i64 <size>, i8* nocapture <ptr>)
```

Overview:

The '`llvm.invariant.end`' intrinsic specifies that the contents of a memory object are mutable.

Arguments:

The first argument is the matching `llvm.invariant.start` intrinsic. The second argument is a constant integer representing the size of the object, or -1 if it is variable sized and the third argument is a pointer to the object.

Semantics:

This intrinsic indicates that the memory is mutable again.

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.ptr.annotation.*`' Intrinsic

Syntax:

This is an overloaded intrinsic. You can use '`llvm.ptr.annotation`' on a pointer to an integer of any width. *NOTE* you must specify an address space for the pointer. The identifier for the default address space is the integer '`0`'.

```
declare i8*   @llvm.ptr.annotation.p<address space>i8(i8* <val>, i8* <str>, i8* <str>, i32  <ir
declare i16*  @llvm.ptr.annotation.p<address space>i16(i16* <val>, i8* <str>, i8* <str>, i32  <
declare i32*  @llvm.ptr.annotation.p<address space>i32(i32* <val>, i8* <str>, i8* <str>, i32  <
declare i64*  @llvm.ptr.annotation.p<address space>i64(i64* <val>, i8* <str>, i8* <str>, i32  <
declare i256* @llvm.ptr.annotation.p<address space>i256(i256* <val>, i8* <str>, i8* <str>, i32
```

Overview:

The '`llvm.ptr.annotation`' intrinsic.

Arguments:

The first argument is a pointer to an integer value of arbitrary bitwidth (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 annotation of a pointer to an integer 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() noreturn nounwind
```

Overview:

The '`llvm.trap`' intrinsic.

Arguments:

None.

Semantics:

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

'`llvm.debugtrap`' Intrinsic

Syntax:

```
declare void @llvm.debugtrap() nounwind
```

Overview:

The '`llvm.debugtrap`' intrinsic.

Arguments:

None.

Semantics:

This intrinsic is lowered to code which is intended to cause an execution trap with the intention of requesting the attention of a debugger.

‘`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 by `llvm.stackprotectorcheck`. If they are different, then `llvm.stackprotectorcheck` causes the program to abort by calling the `__stack_chk_fail()` function.

‘`llvm.stackprotectorcheck`‘ Intrinsic

Syntax:

```
declare void @llvm.stackprotectorcheck(i8** <guard>)
```

Overview:

The `llvm.stackprotectorcheck` intrinsic compares `guard` against an already created stack protector and if they are not equal calls the `__stack_chk_fail()` function.

Arguments:

The `llvm.stackprotectorcheck` intrinsic requires one pointer argument, the the variable `@__stack_chk_guard`.

Semantics:

This intrinsic is provided to perform the stack protector check by comparing `guard` with the stack slot created by `llvm.stackprotector` and if the values do not match call the `__stack_chk_fail()` function.

The reason to provide this as an IR level intrinsic instead of implementing it via other IR operations is that in order to perform this operation at the IR level without an intrinsic, one would need to create additional basic blocks to handle the success/failure cases. This makes it difficult to stop the stack protector check from disrupting sibling tail calls in Codegen. With this intrinsic, we are able to generate the stack protector basic blocks late in codegen after the tail call decision has occurred.

'`llvm.objectsize`' Intrinsic

Syntax:

```
declare i32 @llvm.objectsize.i32(i8* <object>, i1 <min>)
declare i64 @llvm.objectsize.i64(i8* <object>, i1 <min>)
```

Overview:

The `llvm.objectsize` intrinsic is designed to provide information to the optimizers to determine at compile time whether a) an operation (like memcpy) will overflow a buffer that corresponds to an object, or b) that a runtime check for overflow isn't necessary. An object in this context means an allocation of a specific class, structure, array, or other object.

Arguments:

The `llvm.objectsize` intrinsic takes two arguments. The first argument is a pointer to or into the `object`. The second argument is a boolean and determines whether `llvm.objectsize` returns 0 (if true) or -1 (if false) when the object size is unknown. The second argument only accepts constants.

Semantics:

The `llvm.objectsize` intrinsic is lowered to a constant representing the size of the object concerned. If the size cannot be determined at compile time, `llvm.objectsize` returns `i32/i64 -1 or 0` (depending on the `min` argument).

'`llvm.expect`' Intrinsic

Syntax:

```
declare i32 @llvm.expect.i32(i32 <val>, i32 <expected_val>)
declare i64 @llvm.expect.i64(i64 <val>, i64 <expected_val>)
```

Overview:

The `llvm.expect` intrinsic provides information about expected (the most probable) value of `val`, which can be used by optimizers.

Arguments:

The `llvm.expect` intrinsic takes two arguments. The first argument is a value. The second argument is an expected value, this needs to be a constant value, variables are not allowed.

Semantics:

This intrinsic is lowered to the `val`.

'`llvm.donothing`' Intrinsic

Syntax:

```
declare void @llvm.donothing() nounwind readnone
```

Overview:

The `llvm.donothing` intrinsic doesn't perform any operation. It's the only intrinsic that can be called with an invoke instruction.

Arguments:

None.

Semantics:

This intrinsic does nothing, and it's removed by optimizers and ignored by codegen.