# 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 that 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. The `"\01"` prefix can be used on global variables to suppress mangling.
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 and unnamed function parameters are included in this numbering. For example, if the entry basic block is not given a label name and all function parameters are named, 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], [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
!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

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.

`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. From the linker's perspective, an `available_externally` global is equivalent to an external declaration. 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 allow inlining and other optimizations. 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.

Unfortunately this doesn't correspond to any feature in .o files, so it can only be used for variables like `llvm.global_ctors` which llvm interprets specially.

`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.

It is illegal for a function *declaration* to have any linkage type other than `external` 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. Furthermore the inliner doesn't consider such function calls for inlining.

"`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).

"`webkit_jscc`" - WebKit's JavaScript calling convention

This calling convention has been implemented for WebKit FTL JIT. It passes arguments on the stack right to left (as cdecl does), and returns a value in the platform's customary return register.

"`anyregcc`" - Dynamic calling convention for code patching

This is a special convention that supports patching an arbitrary code sequence in place of a call site. This convention forces the call arguments into registers but allows them to be dynamically allocated. This can currently only be used with calls to llvm.experimental.patchpoint because only this intrinsic records the location of its arguments in a side table. See *Stack maps and patch points in LLVM*.

"`preserve_mostcc`" - The *`PreserveMost`* calling convention

This calling convention attempts to make the code in the caller as unintrusive as possible. This convention behaves identically to the *`C`* calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers. This alleviates the burden of saving and recovering a large register set before and after the call in the caller. If the arguments are passed in callee-saved registers, then they will be preserved by the callee across the call. This doesn't apply for values returned in callee-saved registers.

- On X86-64 the callee preserves all general purpose registers, except for R11. R11 can be used as a scratch register. Floating-point registers (XMMs/YMMs) are not preserved and need to be saved by the caller.

The idea behind this convention is to support calls to runtime functions that have a hot path and a cold path. The hot path is usually a small piece of code that doesn't use many registers. The cold path might need to call out to another function and therefore only needs to preserve the caller-saved registers, which haven't already been saved by the caller. The *`PreserveMost`*

calling convention is very similar to the *cold* calling convention in terms of caller/callee-saved registers, but they are used for different types of function calls. *coldcc* is for function calls that are rarely executed, whereas *preserve_mostcc* function calls are intended to be on the hot path and definitely executed a lot. Furthermore *preserve_mostcc* doesn't prevent the inliner from inlining the function call.

This calling convention will be used by a future version of the ObjectiveC runtime and should therefore still be considered experimental at this time. Although this convention was created to optimize certain runtime calls to the ObjectiveC runtime, it is not limited to this runtime and might be used by other runtimes in the future too. The current implementation only supports X86-64, but the intention is to support more architectures in the future.

"preserve_allcc" - The *PreserveAll* calling convention

This calling convention attempts to make the code in the caller even less intrusive than the *PreserveMost* calling convention. This calling convention also behaves identical to the *C* calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers. This removes the burden of saving and recovering a large register set before and after the call in the caller. If the arguments are passed in callee-saved registers, then they will be preserved by the callee across the call. This doesn't apply for values returned in callee-saved registers.

- On X86-64 the callee preserves all general purpose registers, except for R11. R11 can be used as a scratch register. Furthermore it also preserves all floating-point registers (XMMs/YMMs).

The idea behind this convention is to support calls to runtime functions that don't need to call out to any other functions.

This calling convention, like the *PreserveMost* calling convention, will be used by a future version of the ObjectiveC runtime and should be considered experimental at this time.

"cxx_fast_tlscc" - The *CXX_FAST_TLS* calling convention for access functions

Clang generates an access function to access C++-style TLS. The access function generally has an entry block, an exit block and an initialization block that is run at the first time. The entry and exit blocks can access a few TLS IR variables, each access will be lowered to a platform-specific sequence.

This calling convention aims to minimize overhead in the caller by preserving as many registers as possible (all the registers that are perserved on the fast path, composed of the entry and exit blocks).

This calling convention behaves identical to the *C* calling convention on how arguments and return values are passed, but it uses a different set of caller/callee-saved registers.

Given that each platform has its own lowering sequence, hence its own set of preserved registers, we can't use the existing *PreserveMost*.

- On X86-64 the callee preserves all general purpose registers, except for RDI and RAX.

"swiftcc" - This calling convention is used for Swift language.

- On X86-64 RCX and R8 are available for additional integer returns, and XMM2 and XMM3 are available for additional FP/vector returns.
- On iOS platforms, we use AAPCS-VFP calling convention.

"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.

A symbol with `internal` or `private` linkage must have `default` visibility.

## DLL Storage Classes

All Global Variables, Functions and Aliases can have one of the following DLL storage class:

`dllimport`

"`dllimport`" 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`" 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. Since this storage class exists for defining a dll interface, the compiler, assembler and linker know it is externally referenced and must refrain from deleting the symbol.

## Thread Local Storage Models

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.

If no explicit model is given, the "general dynamic" model is used.

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 model can also be specified in an alias, but then it only governs how the alias is accessed. It will not have any effect in the aliasee.

For platforms without linker support of ELF TLS model, the -femulated-tls flag can be used to generate GCC compatible emulated TLS code.

## Structure Types

LLVM IR allows you to specify both "identified" and "literal" *structure types*. Literal types are uniqued structurally, but identified types are never uniqued. An *opaque structural type* can also be used to forward declare a type that is not yet available.

An example of an identified structure specification is:

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

Prior to the LLVM 3.0 release, identified types were structurally uniqued. Only literal types are uniqued in recent versions of LLVM.

## Global Variables

Global variables define regions of memory allocated at compilation time instead of run-time.

Global variable definitions must be initialized.

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

Either global variable definitions or declarations may have an explicit section to be placed in and may have an optional explicit alignment specified.

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.

If the `local_unnamed_addr` attribute is given, the address is known to not be significant within the module.

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. Additionally, the global can placed in a comdat if the target has the necessary support.

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` or dllexported variables. 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. The maximum alignment is `1 << 29`.

Globals can also have a *DLL storage class* and an optional list of attached *metadata*,

Variables and aliases can have a *Thread Local Storage Model*.

Syntax:

```
@<GlobalVarName> = [Linkage] [Visibility] [DLLStorageClass] [ThreadLocal]
                   [(unnamed_addr|local_unnamed_addr)] [AddrSpace]
                   [ExternallyInitialized]
                   <global | constant> <Type> [<InitializerConstant>]
                   [, section "name"] [, comdat [($name)]]
                   [, align <Alignment>] (, !name !N)*
```

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 *DLL storage class*, 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 *comdat*, an optional *garbage collector name*, an optional *prefix*, an optional *prologue*, an optional *personality*, an optional list of attached *metadata*, 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 *DLL storage class*, an optional *calling convention*, an optional `unnamed_addr` or `local_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*, an optional *prefix*, and an optional *prologue*.

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. Additionally, the function can be placed in a COMDAT.

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 known to not be significant and two identical functions can be merged.

If the `local_unnamed_addr` attribute is given, the address is known to not be significant within the module.

Syntax:

```
define [linkage] [visibility] [DLLStorageClass]
       [cconv] [ret attrs]
       <ResultType> @<FunctionName> ([argument list])
       [(unnamed_addr|local_unnamed_addr)] [fn Attrs] [section "name"]
       [comdat [($name)]] [align N] [gc] [prefix Constant]
       [prologue Constant] [personality Constant] (!name !N)* { ... }
```

The argument list is a comma separated sequence of arguments where each argument is of the following form:

Syntax:

```
<type> [parameter Attrs] [name]
```

## Aliases

Aliases, unlike function or variables, don't create any new data. They are just a new symbol and metadata for an existing position.

Aliases have a name and an aliasee that is either a global value or a constant expression.

Aliases may have an optional *linkage type*, an optional *visibility style*, an optional *DLL storage class* and an optional *tls model*.

Syntax:

```
@<Name> = [Linkage] [Visibility] [DLLStorageClass] [ThreadLocal] [(unnamed_addr|local_u
```

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

Aliases that are not `unnamed_addr` are guaranteed to have the same address as the aliasee expression. `unnamed_addr` ones are only guaranteed to point to the same content.

If the `local_unnamed_addr` attribute is given, the address is known to not be significant within the module.

Since aliases are only a second name, some restrictions apply, of which some can only be checked when producing an object file:

- The expression defining the aliasee must be computable at assembly time. Since it is just a name, no relocations can be used.
- No alias in the expression can be weak as the possibility of the intermediate alias being overridden cannot be represented in an object file.
- No global value in the expression can be a declaration, since that would require a relocation, which is not possible.

## IFuncs

IFuncs, like as aliases, don't create any new data or func. They are just a new symbol that dynamic linker resolves at runtime by calling a resolver function.

IFuncs have a name and a resolver that is a function called by dynamic linker that returns address of another function associated with the name.

IFunc may have an optional *linkage type* and an optional *visibility style*.

Syntax:

```
@<Name> = [Linkage] [Visibility] ifunc <IFuncTy>, <ResolverTy>* @<Resolver>
```

## Comdats

Comdat IR provides access to COFF and ELF object file COMDAT functionality.

Comdats have a name which represents the COMDAT key. All global objects that specify this key will only end up in the final object file if the linker chooses that key over some other key. Aliases are placed in the same COMDAT that their aliasee computes to, if any.

Comdats have a selection kind to provide input on how the linker should choose between keys in two different object files.

Syntax:

```
$<Name> = comdat SelectionKind
```

The selection kind must be one of the following:

`any`
: The linker may choose any COMDAT key, the choice is arbitrary.

`exactmatch`
: The linker may choose any COMDAT key but the sections must contain the same data.

`largest`
: The linker will choose the section containing the largest COMDAT key.

`noduplicates`
: The linker requires that only section with this COMDAT key exist.

`samesize`
: The linker may choose any COMDAT key but the sections must contain the same amount of data.

Note that the Mach-O platform doesn't support COMDATs and ELF only supports `any` as a selection kind.

Here is an example of a COMDAT group where a function will only be selected if the COMDAT key's section is the largest:

```
$foo = comdat largest
@foo = global i32 2, comdat($foo)

define void @bar() comdat($foo) {
  ret void
}
```

As a syntactic sugar the `$name` can be omitted if the name is the same as the global name:

```
$foo = comdat any
@foo = global i32 2, comdat
```

In a COFF object file, this will create a COMDAT section with selection kind `IMAGE_COMDAT_SELECT_LARGEST` containing the contents of the `@foo` symbol and another COMDAT section with selection kind `IMAGE_COMDAT_SELECT_ASSOCIATIVE` which is associated with the first COMDAT section and contains the contents of the `@bar` symbol.

There are some restrictions on the properties of the global object. It, or an alias to it, must have the same name as the COMDAT group when targeting COFF. The contents and size of this object may be used during link-time to determine which COMDAT groups get selected depending on the selection kind. Because the name of the object must match the name of the COMDAT group, the linkage of the global object must not be local; local symbols can get renamed if a collision occurs in the symbol table.

The combined use of COMDATS and section attributes may yield surprising results. For example:

```
$foo = comdat any
$bar = comdat any
@g1 = global i32 42, section "sec", comdat($foo)
@g2 = global i32 42, section "sec", comdat($bar)
```

From the object file perspective, this requires the creation of two sections with the same name. This is necessary because both globals belong to different COMDAT groups and COMDATs, at the object file level, are represented by sections.

Note that certain IR constructs like global variables and functions may create COMDATs in the object file in addition to any which are specified using COMDAT IR. This arises when the code generator is configured to emit globals in individual sections (e.g. when *`-data-sections`* or *`-function-sections`* is supplied to *`llc`*).

## Named Metadata

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

1. Named metadata are represented as a string of characters with the metadata prefix. The rules for metadata names are the same as for identifiers, but quoted names are not allowed. `"\xx"` type escapes are still valid, which allows any character to be part of a name.

Syntax:

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

`inalloca`

The `inalloca` argument attribute allows the caller to take the address of outgoing stack arguments. An `inalloca` argument must be a pointer to stack memory produced by an `alloca` instruction. The alloca, or argument allocation, must also be tagged with the inalloca keyword. Only the last argument may have the `inalloca` attribute, and that argument is guaranteed to be passed in memory.

An argument allocation may be used by a call at most once because the call may deallocate it. The `inalloca` attribute cannot be used in conjunction with other attributes that affect argument storage, like `inreg`, `nest`, `sret`, or `byval`. The `inalloca` attribute also disables LLVM's implicit lowering of large aggregate return values, which means that frontend authors must lower them with `sret` pointers.

When the call site is reached, the argument allocation must have been the most recent stack allocation that is still live, or the results are undefined. It is possible to allocate

additional stack space after an argument allocation and before its call site, but it must be cleared off with *llvm.stackrestore*.

See *Design and Usage of the InAlloca Attribute* for more information on how to use this attribute.

`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.

`align <n>`

This indicates that the pointer value may be assumed by the optimizer to have the specified alignment.

Note that this attribute has additional semantics when combined with the `byval` attribute.

`noalias`

This indicates that objects accessed via pointer values *based* on the argument or return value are not also accessed, during the execution of the function, via pointer values not *based* on the argument or return value. The attribute on a return value also has additional semantics described below. 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.

For function return values, C99's `restrict` is not meaningful, while LLVM's `noalias` is. Furthermore, the semantics of the `noalias` attribute on return values are stronger than the semantics of the attribute when used on function arguments. On function return values, the `noalias` attribute indicates that the function acts like a system memory allocation function, returning a pointer to allocated storage disjoint from the storage for any other object accessible to the caller.

`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. Addresses used in volatile operations are considered to be captured.

`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 a hint to the optimizer and code generator used when generating the caller, allowing value propagation, 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.

`nonnull`

This indicates that the parameter or return pointer is not null. This attribute may only be applied to pointer typed parameters. This is not checked or enforced by LLVM, the caller must ensure that the pointer passed in is non-null, or the callee must ensure that the returned pointer is non-null.

`dereferenceable(<n>)`

This indicates that the parameter or return pointer is dereferenceable. This attribute may only be applied to pointer typed parameters. A pointer that is dereferenceable can be loaded from speculatively without a risk of trapping. The number of bytes known to be dereferenceable must be provided in parentheses. It is legal for the number of bytes to be less than the size of the pointee type. The `nonnull` attribute does not imply dereferenceability (consider a pointer to one element past the end of an array), however `dereferenceable(<n>)` does imply `nonnull` in `addrspace(0)` (which is the default address space).

`dereferenceable_or_null(<n>)`

This indicates that the parameter or return value isn't both non-null and non-dereferenceable (up to <n> bytes) at the same time. All non-null pointers tagged with `dereferenceable_or_null(<n>)` are `dereferenceable(<n>)`. For address space 0 `dereferenceable_or_null(<n>)` implies that a pointer is exactly one of `dereferenceable(<n>)` or `null`, and in other address spaces `dereferenceable_or_null(<n>)` implies that a pointer is at least one of `dereferenceable(<n>)` or `null` (i.e. it may be both `null` and `dereferenceable(<n>)`). This attribute may only be applied to pointer typed parameters.

`swiftself`

This indicates that the parameter is the self/context parameter. This is not a valid attribute for return values and can only be applied to one parameter.

`swifterror`

This attribute is motivated to model and optimize Swift error handling. It can be applied to a parameter with pointer to pointer type or a pointer-sized alloca. At the call site, the actual argument that corresponds to a `swifterror` parameter has to come from a `swifterror` alloca. A `swifterror` value (either the parameter or the alloca) can only be loaded and stored from, or used as a `swifterror` argument. This is not a valid attribute for return values and can only be applied to one parameter.

These constraints allow the calling convention to optimize access to `swifterror` variables by associating them with a specific register at call boundaries rather than placing them in memory. Since this does change the calling convention, a function which uses the `swifterror` attribute on a parameter is not ABI-compatible with one which does not.

These constraints also allow LLVM to assume that a `swifterror` argument does not alias any other memory visible within a function and that a `swifterror` alloca passed as an argument does not escape.

## Garbage Collector Strategy Names

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

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

The supported values of *name* includes those *built in to LLVM* and any provided by loaded plugins. Specifying a GC strategy will cause the compiler to alter its output in order to support the named garbage collection algorithm. Note that LLVM itself does not contain a garbage collector, this functionality is restricted to generating machine code which can interoperate with a collector provided externally.

## Prefix Data

Prefix data is data associated with a function which the code generator will emit immediately before the function's entrypoint. 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 and dereference index -1. This implies that the IR symbol points just past the

end of the prefix data. For instance, take the example of a function annotated with a single `i32`,

```
define void @f() prefix i32 123 { ... }
```

The prefix data can be referenced as,

```
%0 = bitcast void* () @f to i32*
%a = getelementptr inbounds i32, i32* %0, i32 -1
%b = load i32, i32* %a
```

Prefix data is laid out as if it were an initializer for a global variable of the prefix data's type. The function will be placed such that the beginning of the prefix data is aligned. This means that if the size of the prefix data is not a multiple of the alignment size, the function's entrypoint will not be aligned. If alignment of the function's entrypoint is desired, padding must be added to the prefix data.

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.

## Prologue Data

The `prologue` attribute allows arbitrary code (encoded as bytes) to be inserted prior to the function body. This can be used for enabling function hot-patching and instrumentation.

To maintain the semantics of ordinary function calls, the prologue 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 prologue 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 prologue data. Obviously this makes the format of the prologue data highly target dependent.

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

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

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

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

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

A function may have prologue 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.

## Personality Function

The `personality` attribute permits functions to specify what function to use for exception handling.

## 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.

`allocsize(<EltSizeParam>[, <NumEltsParam>])`

This attribute indicates that the annotated function will always return at least a given number of bytes (or null). Its arguments are zero-indexed parameter numbers; if one argument is provided, then it's assumed that at least `CallSite.Args[EltSizeParam]` bytes will be available at the returned pointer. If two are provided, then it's assumed that `CallSite.Args[EltSizeParam] * CallSite.Args[NumEltsParam]` bytes are available. The referenced parameters must be integer types. No assumptions are made about the contents of the returned block of memory.

`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 that 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.

`convergent`

In some parallel execution models, there exist operations that cannot be made control-dependent on any additional values. We call such operations `convergent`, and mark them with this attribute.

The `convergent` attribute may appear on functions or call/invoke instructions. When it appears on a function, it indicates that calls to this function should not be made control-dependent on additional values. For example, the intrinsic `llvm.nvvm.barrier0` is `convergent`, so calls to this intrinsic cannot be made control-dependent on additional values.

When it appears on a call/invoke, the `convergent` attribute indicates that we should treat the call as though we're calling a convergent function. This is particularly useful on indirect calls; without this we may treat such calls as though the target is non-convergent.

The optimizer may remove the `convergent` attribute on functions when it can prove that the function does not execute any convergent operations. Similarly, the optimizer may remove `convergent` on calls/invokes when it can prove that the call/invoke cannot call a convergent function.

`inaccessiblememonly`

This attribute indicates that the function may only access memory that is not accessible by the module being compiled. This is a weaker form of `readnone`.

`inaccessiblemem_or_argmemonly`

This attribute indicates that the function may only access memory that is either not accessible by the module being compiled, or is pointed to by its pointer arguments. This is a weaker form of `argmemonly`

`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.

`jumptable`

This attribute indicates that the function should be added to a jump-instruction table at code-generation time, and that all address-taken references to this function should be replaced with a reference to the appropriate jump-instruction-table function pointer. Note that this creates a new pointer for the original function, which means that code that depends on function-pointer identity can break. So, any function annotated with `jumptable` must also be `unnamed_addr`.

`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.

`norecurse`

This function attribute indicates that the function does not call itself either directly or indirectly down any possible call path. This produces undefined behavior at runtime if the function ever does recurse.

`nounwind`

This function attribute indicates that the function never raises an exception. If the function does raise an exception, its runtime behavior is undefined. However, functions marked nounwind may still trap or generate asynchronous exceptions. Exception handling schemes that are recognized by LLVM to handle asynchronous exceptions, such as SEH, will still provide their implementation defined semantics.

`optnone`

This function attribute indicates that most optimization passes will skip this function, with the exception of interprocedural optimization passes. Code generation defaults to the “fast” instruction selector. 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.

`"patchable-function"`

This attribute tells the code generator that the code generated for this function needs to follow certain conventions that make it possible for a runtime function to patch over it later. The exact effect of this attribute depends on its string value, for which there currently is one legal possibility:

- `"prologue-short-redirect"` - This style of patchable function is intended to support patching a function prologue to redirect control away from the function in a thread safe manner. It guarantees that the first instruction of the function will be large enough to accommodate a short jump instruction, and will be sufficiently aligned to allow being fully changed via an atomic compare-and-swap instruction. While the first requirement can be satisfied by inserting large enough NOP, LLVM can and will try to re-purpose an

existing instruction (i.e. one that would have to be emitted anyway) as the patchable instruction larger than a short jump.

`"prologue-short-redirect"` is currently only supported on x86-64.

This attribute by itself does not imply restrictions on inter-procedural optimizations. All of the semantic effects the patching may have to be separately conveyed via the linkage type.

`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.

`writeonly`

On a function, this attribute indicates that the function may write to but does not read from memory.

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

`argmemonly`

This attribute indicates that the only memory accesses inside function are loads and stores from objects pointed to by its pointer-typed arguments, with arbitrary offsets. Or in other words, all memory operations in the function can refer to memory only using pointers based on its function arguments. Note that `argmemonly` can be used together with `readonly` attribute in order to specify that function reads only from its arguments.

`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.

`safestack`

This attribute indicates that SafeStack protection is enabled for this function.

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

`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`.

Variables that are identified as requiring a protector will be arranged on the stack such that they are adjacent to the stack protector guard.

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.

Variables that are identified as requiring a protector will be arranged on the stack such that they are adjacent to the stack protector guard. The specific layout rules are:

1. Large arrays and structures containing large arrays (`>= ssp-buffer-size`) are closest to the stack protector.
2. Small arrays and structures containing small arrays (`< ssp-buffer-size`) are 2nd closest to the protector.
3. Variables that have had their address taken are 3rd closest to the protector.

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.

Variables that are identified as requiring a protector will be arranged on the stack such that they are adjacent to the stack protector guard. The specific layout rules are:

1. Large arrays and structures containing large arrays (`>= ssp-buffer-size`) are closest to the stack protector.
2. Small arrays and structures containing small arrays (`< ssp-buffer-size`) are 2nd closest to the protector.
3. Variables that have had their address taken are 3rd closest to the protector.

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.

`"thunk"`

This attribute indicates that the function will delegate to some other function with a tail call. The prototype of a thunk should not be used for optimization purposes. The caller is expected to cast the thunk prototype to match the thunk target prototype.

`uwtable`

This attribute indicates that the ABI being targeted requires that an unwind table entry be produced 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.

## Operand Bundles

Note: operand bundles are a work in progress, and they should be considered experimental at this time.

Operand bundles are tagged sets of SSA values that can be associated with certain LLVM instructions (currently only `call` s and `invoke` s). In a way they are like metadata, but dropping them is incorrect and will change program semantics.

Syntax:

```
operand bundle set ::= '[' operand bundle (, operand bundle )* ']'
operand bundle ::= tag '(' [ bundle operand ] (, bundle operand )* ')'
bundle operand ::= SSA value
tag ::= string constant
```

Operand bundles are **not** part of a function's signature, and a given function may be called from multiple places with different kinds of operand bundles. This reflects the fact that the operand bundles are conceptually a part of the `call` (or `invoke`), not the callee being dispatched to.

Operand bundles are a generic mechanism intended to support runtime-introspection-like functionality for managed languages. While the exact semantics of an operand bundle depend on the bundle tag, there are certain limitations to how much the presence of an operand bundle can influence the semantics of a program. These restrictions are described as the semantics of an "unknown" operand bundle. As long as the behavior of an operand bundle is describable within these restrictions, LLVM does not need to have special knowledge of the operand bundle to not miscompile programs containing it.

- The bundle operands for an unknown operand bundle escape in unknown ways before control is transferred to the callee or invokee.
- Calls and invokes with operand bundles have unknown read / write effect on the heap on entry and exit (even if the call target is `readnone` or `readonly`), unless they're overridden with callsite specific attributes.
- An operand bundle at a call site cannot change the implementation of the called function. Inter-procedural optimizations work as usual as long as they take into account the first two properties.

More specific types of operand bundles are described below.

### Deoptimization Operand Bundles

Deoptimization operand bundles are characterized by the `"deopt"` operand bundle tag. These operand bundles represent an alternate "safe" continuation for the call site they're attached to, and can be used by a suitable runtime to deoptimize the compiled frame at the specified call site. There can be at most one `"deopt"` operand bundle attached to a call site. Exact details of deoptimization is out of scope for the language reference, but it usually involves rewriting a compiled frame into a set of interpreted frames.

From the compiler's perspective, deoptimization operand bundles make the call sites they're attached to at least `readonly`. They read through all of their pointer typed operands (even if they're not otherwise escaped) and the entire visible heap. Deoptimization operand bundles do not capture their operands except during deoptimization, in which case control will not be returned to the compiled frame.

The inliner knows how to inline through calls that have deoptimization operand bundles. Just like inlining through a normal call site involves composing the normal and exceptional continuations, inlining through a call site with a deoptimization operand bundle needs to appropriately compose the "safe" deoptimization continuation. The inliner does this by prepending the parent's deoptimization continuation to every deoptimization continuation in the inlined body. E.g. inlining `@f` into `@g` in the following example

```
define void @f() {
  call void @x()  ;; no deopt state
  call void @y() [ "deopt"(i32 10) ]
  call void @y() [ "deopt"(i32 10), "unknown"(i8* null) ]
  ret void
}

define void @g() {
  call void @f() [ "deopt"(i32 20) ]
  ret void
}
```

will result in

```
define void @g() {
  call void @x()  ;; still no deopt state
  call void @y() [ "deopt"(i32 20, i32 10) ]
  call void @y() [ "deopt"(i32 20, i32 10), "unknown"(i8* null) ]
  ret void
}
```

It is the frontend's responsibility to structure or encode the deoptimization state in a way that syntactically prepending the caller's deoptimization state to the callee's deoptimization state is semantically equivalent to composing the caller's deoptimization continuation after the callee's deoptimization continuation.

### Funclet Operand Bundles

Funclet operand bundles are characterized by the `"funclet"` operand bundle tag. These operand bundles indicate that a call site is within a particular funclet. There can be at most one `"funclet"` operand bundle attached to a call site and it must have exactly one bundle operand.

If any funclet EH pads have been "entered" but not "exited" (per the description in the EH doc), it is undefined behavior to execute a `call` or `invoke` which:

- does not have a `"funclet"` bundle and is not a `call` to a nounwind intrinsic, or
- has a `"funclet"` bundle whose operand is not the most-recently-entered not-yet-exited funclet EH pad.

Similarly, if no funclet EH pads have been entered-but-not-yet-exited, executing a `call` or `invoke` with a `"funclet"` bundle is undefined behavior.

### GC Transition Operand Bundles

GC transition operand bundles are characterized by the `"gc-transition"` operand bundle tag. These operand bundles mark a call as a transition between a function with one GC strategy to a function with a different GC strategy. If coordinating the transition between GC strategies requires additional code generation at the call site, these bundles may contain any values that are needed by the generated code. For more details, see *GC Transitions*.

## 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.

Note that the assembly string *must* be parseable by LLVM's integrated assembler (unless it is disabled), even when emitting a `.s` file.

## 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. 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:<abi>:<pref>`

This specifies the alignment for an object of aggregate type.

`m:<mangling>`

If present, specifies that llvm names are mangled in the output. The options are

- `e`: ELF mangling: Private symbols get a `.L` prefix.
- `m`: Mips mangling: Private symbols get a `$` prefix.
- `o`: Mach-O mangling: Private symbols get `L` prefix. Other symbols get a _ prefix.
- `w`: Windows COFF prefix: Similar to Mach-O, but stdcall and fastcall functions also get a suffix based on the frame size.
- `x`: Windows x86 COFF prefix: Similar to Windows COFF, but use a _ prefix for `__cdecl` functions.

`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.

On every specification that takes a `<abi>:<pref>`, specifying the `<pref>` alignment is optional. If omitted, the preceding `:` should be omitted too and `<pref>` will be equal to `<abi>`.

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
- `a: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. There is no way to generate IR that does not embed this target-specific detail into the IR. If you don't specify the string, the default

specifications will be used to generate a Data Layout and the optimization phases will operate accordingly and introduce target specificity into the IR with respect to these default specifications.

## 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 value 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 transformations 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 that 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 ordering parameters that determine 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 that only reads, `release` for an operation that 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*, *fcmp*) have the following flags that can be 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.

## Use-list Order Directives

Use-list directives encode the in-memory order of each use-list, allowing the order to be recreated. `<order-indexes>` is a comma-separated list of indexes that are assigned to the referenced value's uses. The referenced value's use-list is immediately sorted by these indexes.

Use-list directives may appear at function scope or global scope. They are not instructions, and have no effect on the semantics of the IR. When they're at function scope, they must appear after the terminator of the final basic block.

If basic blocks have their address taken via `blockaddress()` expressions, `uselistorder_bb` can be used to reorder their use-lists from outside their function's scope.

**Syntax:**

```
uselistorder <ty> <value>, { <order-indexes> }
uselistorder_bb @function, %block { <order-indexes> }
```

**Examples:**

```
define void @foo(i32 %arg1, i32 %arg2) {
entry:
  ; ... instructions ...
bb:
```

```
  ; ... instructions ...

  ; At function scope.
  uselistorder i32 %arg1, { 1, 0, 2 }
  uselistorder label %bb, { 1, 0 }
}

; At global scope.
uselistorder i32* @global, { 1, 2, 0 }
uselistorder i32 7, { 1, 0 }
uselistorder i32 (i32) @bar, { 1, 0 }
uselistorder_bb @foo, %bb, { 5, 1, 3, 2, 0, 4 }
```

### Source Filename

The *source filename* string is set to the original module identifier, which will be the name of the compiled source file when compiling from source through the clang front end, for example. It is then preserved through the IR and bitcode.

This is currently necessary to generate a consistent unique global identifier for local functions used in profile data, which prepends the source file name to the local function name.

The syntax for the source file name is simply:

```
source_filename = "/path/to/source.c"
```

## 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.

### Void Type

**Overview:**

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

**Syntax:**

```
void
```

### 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,` | *Pointer* to a function that takes an `i16` and a *pointer* to `i32`, returning `float`. |

| | |
|---|---|
| `i32 *) *` | |
| `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 |

## First Class Types

The *first class* types are perhaps the most important. Values of these types are the only ones which can be produced by instructions.

### Single Value Types

These are the types that are valid in registers from CodeGen's perspective.

#### 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) |

#### X86_mmx Type

**Overview:**

The x86_mmx 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:**

```
x86_mmx
```

### 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, floating point or pointer type. 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. |

### Label Type

**Overview:**

The label type represents code labels.

**Syntax:**

```
label
```

### Token Type

**Overview:**

The token type is used when a value is associated with an instruction but all uses of the value must not attempt to introspect or obscure it. As such, it is not appropriate to have a *phi* or *select* of type token.

**Syntax:**

```
token
```

### Metadata Type

**Overview:**

The metadata type represents embedded metadata. No derived types may be created from metadata except for *function* arguments.

**Syntax:**

```
metadata
```

## 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.

### 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. |

## 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*.

**Token constants**

The identifier '`none`' is recognized as an empty token constant and must be of *token 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 x86_mmx.

## 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. As a special case, character array constants may also be represented as a double-quoted string using the `c` prefix. For example: "`c"Hello World\0A\00"`".

**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 constant tuple without types. For example: "`!{!0, !{!2, !0}, !"test"}`". Metadata can reference constant values, for example: "`!{!0, i32 0, i8* @global, i64 (i64)* @function, !"str"}`". Unlike other typed 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 slt %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 that cannot evoke side effects has nevertheless detected a condition that 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 effect that any instruction that 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, 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, 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, i16* %narrowaddr ; Returns a poison value.
  %poison4 = load i64, 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 (TY, CSTPTR, IDX0, IDX1, ...)`, `getelementptr inbounds (TY, 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 "pointer to TY".

`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 template 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.

The template string supports argument substitution of the operands using "`$`" followed by a number, to indicate substitution of the given register/memory location, as specified by the constraint string. "`${NUM:MODIFIER}`" may also be used, where `MODIFIER` is a target-specific annotation for how to print the operand (See *Asm template argument modifiers*).

A literal "`$`" may be included by using "`$$`" in the template. To include other special characters into the output, the usual "`\XX`" escapes may be used, just as in other strings. Note that after template substitution, the resulting assembly string is parsed by LLVM's integrated assembler unless it is disabled – even when emitting a `.s` file – and thus must contain assembly syntax known to LLVM.

LLVM's support for inline asm is modeled closely on the requirements of Clang's GCC-compatible inline-asm support. Thus, the feature-set and the constraint and modifier codes listed here are similar or identical to those in GCC's inline asm support. However, to be clear, the syntax of the template and constraint strings described here is *not* the same as the syntax accepted by GCC and Clang, and, while most constraint letters are passed through as-is by Clang, some get translated to other codes when converting from the C source to the LLVM assembly.

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 Constraint String

The constraint list is a comma-separated string, each element containing one or more constraint codes.

For each element in the constraint list an appropriate register or memory operand will be chosen, and it will be made available to assembly template string expansion as `$0` for the first constraint in the list, `$1` for the second, etc.

There are three different types of constraints, which are distinguished by a prefix symbol in front of the constraint code: Output, Input, and Clobber. The constraints must always be given in that order: outputs first, then inputs, then clobbers. They cannot be intermingled.

There are also three different categories of constraint codes:

- Register constraint. This is either a register class, or a fixed physical register. This kind of constraint will allocate a register, and if necessary, bitcast the argument or result to the appropriate type.
- Memory constraint. This kind of constraint is for use with an instruction taking a memory operand. Different constraints allow for different addressing modes used by the target.
- Immediate value constraint. This kind of constraint is for an integer or other immediate value which can be rendered directly into an instruction. The various target-specific constraints allow the selection of a value in the proper range for the instruction you wish to use it with.

#### Output constraints

Output constraints are specified by an "=" prefix (e.g. "=r"). This indicates that the assembly will write to this operand, and the operand will then be made available as a return value of the `asm` expression. Output constraints do not consume an argument from the call instruction. (Except, see below about indirect outputs).

Normally, it is expected that no output locations are written to by the assembly expression until *all* of the inputs have been read. As such, LLVM may assign the same register to an output and an input. If this is not safe (e.g. if the assembly contains two instructions, where the first writes to one output, and the second reads an input and writes to a second output), then the "&" modifier must be used (e.g. "=&r") to specify that the output is an "early-clobber" output. Marking an output as "early-clobber" ensures that LLVM will not use the same register for any inputs (other than an input tied to this output).

#### Input constraints

Input constraints do not have a prefix – just the constraint codes. Each input constraint will consume one argument from the call instruction. It is not permitted for the asm to write to any input register or memory location (unless that input is tied to an output). Note also that multiple inputs may all be assigned to the same register, if LLVM can determine that they necessarily all contain the same value.

Instead of providing a Constraint Code, input constraints may also "tie" themselves to an output constraint, by providing an integer as the constraint string. Tied inputs still consume an argument from the call instruction, and take up a position in the asm template numbering as is usual – they will simply be constrained to always use the same register as the output they've been tied to. For example, a constraint string of "`=r,0`" says to assign a register for output, and use that register as an input as well (it being the 0'th constraint).

It is permitted to tie an input to an "early-clobber" output. In that case, no *other* input may share the same register as the input tied to the early-clobber (even when the other input has the same value).

You may only tie an input to an output which has a register constraint, not a memory constraint. Only a single input may be tied to an output.

There is also an "interesting" feature which deserves a bit of explanation: if a register class constraint allocates a register which is too small for the value type operand provided as input, the input value will be split into multiple registers, and all of them passed to the inline asm.

However, this feature is often not as useful as you might think.

Firstly, the registers are *not* guaranteed to be consecutive. So, on those architectures that have instructions which operate on multiple consecutive instructions, this is not an appropriate way to support them. (e.g. the 32-bit SparcV8 has a 64-bit load, which instruction takes a single 32-bit register. The hardware then loads into both the named register, and the next register. This feature of inline asm would not be useful to support that.)

A few of the targets provide a template string modifier allowing explicit access to the second register of a two-register operand (e.g. MIPS `L`, `M`, and `D`). On such an architecture, you can actually access the second allocated register (yet, still, not any subsequent ones). But, in that case, you're still probably better off simply splitting the value into two separate operands, for clarity. (e.g. see the description of the `A` constraint on X86, which, despite existing only for use with this feature, is not really a good idea to use)

### Indirect inputs and outputs

Indirect output or input constraints can be specified by the "`*`" modifier (which goes after the "`=`" in case of an output). This indicates that the asm will write to or read from the contents of an *address* provided as an input argument. (Note that in this way, indirect outputs act more like an *input* than an output: just like an input, they consume an argument of the call expression, rather than producing a return value. An indirect output constraint is an "output" only in that the asm is expected to write to the contents of the input memory location, instead of just read from it).

This is most typically used for memory constraint, e.g. "`=*m`", to pass the address of a variable as a value.

It is also possible to use an indirect *register* constraint, but only on output (e.g. "`=*r`"). This will cause LLVM to allocate a register for an output value normally, and then, separately emit a store to the address provided as input, after the provided inline asm. (It's not clear what value this functionality provides, compared to writing the store explicitly after the asm statement, and it can only produce worse code, since it bypasses many optimization passes. I would recommend not using it.)

### Clobber constraints

A clobber constraint is indicated by a "~" prefix. A clobber does not consume an input operand, nor generate an output. Clobbers cannot use any of the general constraint code letters – they may use only explicit register constraints, e.g. "~{eax}". The one exception is that a clobber string of "~{memory}" indicates that the assembly writes to arbitrary undeclared memory locations – not only the memory pointed to by a declared indirect output.

### Constraint Codes

After a potential prefix comes constraint code, or codes.

A Constraint Code is either a single letter (e.g. "r"), a "^" character followed by two letters (e.g. "^wc"), or "{" register-name "}" (e.g. "{eax}").

The one and two letter constraint codes are typically chosen to be the same as GCC's constraint codes.

A single constraint may include one or more than constraint code in it, leaving it up to LLVM to choose which one to use. This is included mainly for compatibility with the translation of GCC inline asm coming from clang.

There are two ways to specify alternatives, and either or both may be used in an inline asm constraint list:

1. Append the codes to each other, making a constraint code set. E.g. "im" or "{eax}m". This means "choose any of the options in the set". The choice of constraint is made independently for each constraint in the constraint list.
2. Use "|" between constraint code sets, creating alternatives. Every constraint in the constraint list must have the same number of alternative sets. With this syntax, the same alternative in *all* of the items in the constraint list will be chosen together.

Putting those together, you might have a two operand constraint string like "rm|r,ri|rm". This indicates that if operand 0 is r or m, then operand 1 may be one of r or i. If operand 0 is r, then operand 1 may be one of r or m. But, operand 0 and 1 cannot both be of type m.

However, the use of either of the alternatives features is *NOT* recommended, as LLVM is not able to make an intelligent choice about which one to use. (At the point it currently needs to choose, not enough information is available to do so in a smart way.) Thus, it simply tries to make a choice that's most likely to compile, not one that will be optimal performance. (e.g., given "rm", it'll always choose to use memory, not registers). And, if given multiple registers, or multiple register classes, it will simply choose the first one. (In fact, it doesn't currently even ensure explicitly specified physical registers are unique, so specifying multiple physical registers as alternatives, like {r11}{r12},{r11}{r12}, will assign r11 to both operands, not at all what was intended.)

### Supported Constraint Code List

The constraint codes are, in general, expected to behave the same way they do in GCC. LLVM's support is often implemented on an 'as-needed' basis, to support C inline asm code which was supported by GCC. A mismatch in behavior between LLVM and GCC likely indicates a bug in LLVM.

Some constraint codes are typically supported by all targets:

- r: A register in the target's general purpose register class.
- m: A memory address operand. It is target-specific what addressing modes are supported, typical examples are register, or register + register offset, or register + immediate offset (of some target-specific size).
- i: An integer constant (of target-specific width). Allows either a simple immediate, or a relocatable value.
- n: An integer constant – *not* including relocatable values.
- s: An integer constant, but allowing *only* relocatable values.

- `X`: Allows an operand of any kind, no constraint whatsoever. Typically useful to pass a label for an asm branch or call.
- `{register-name}`: Requires exactly the named physical register.

Other constraints are target-specific:

AArch64:

- `z`: An immediate integer 0. Outputs `WZR` or `XZR`, as appropriate.
- `I`: An immediate integer valid for an `ADD` or `SUB` instruction, i.e. 0 to 4095 with optional shift by 12.
- `J`: An immediate integer that, when negated, is valid for an `ADD` or `SUB` instruction, i.e. -1 to -4095 with optional left shift by 12.
- `K`: An immediate integer that is valid for the 'bitmask immediate 32' of a logical instruction like `AND`, `EOR`, or `ORR` with a 32-bit register.
- `L`: An immediate integer that is valid for the 'bitmask immediate 64' of a logical instruction like `AND`, `EOR`, or `ORR` with a 64-bit register.
- `M`: An immediate integer for use with the `MOV` assembly alias on a 32-bit register. This is a superset of `K`: in addition to the bitmask immediate, also allows immediate integers which can be loaded with a single `MOVZ` or `MOVL` instruction.
- `N`: An immediate integer for use with the `MOV` assembly alias on a 64-bit register. This is a superset of `L`.
- `Q`: Memory address operand must be in a single register (no offsets). (However, LLVM currently does this for the `m` constraint as well.)
- `r`: A 32 or 64-bit integer register (W* or X*).
- `w`: A 32, 64, or 128-bit floating-point/SIMD register.
- `x`: A lower 128-bit floating-point/SIMD register (`V0` to `V15`).

AMDGPU:

- `r`: A 32 or 64-bit integer register.
- `[0-9]v`: The 32-bit VGPR register, number 0-9.
- `[0-9]s`: The 32-bit SGPR register, number 0-9.

All ARM modes:

- `Q`, `Um`, `Un`, `Uq`, `Us`, `Ut`, `Uv`, `Uy`: Memory address operand. Treated the same as operand `m`, at the moment.

ARM and ARM's Thumb2 mode:

- `j`: An immediate integer between 0 and 65535 (valid for `MOVW`)
- `I`: An immediate integer valid for a data-processing instruction.
- `J`: An immediate integer between -4095 and 4095.
- `K`: An immediate integer whose bitwise inverse is valid for a data-processing instruction. (Can be used with template modifier "`B`" to print the inverted value).
- `L`: An immediate integer whose negation is valid for a data-processing instruction. (Can be used with template modifier "`n`" to print the negated value).
- `M`: A power of two or a integer between 0 and 32.
- `N`: Invalid immediate constraint.
- `O`: Invalid immediate constraint.
- `r`: A general-purpose 32-bit integer register (`r0-r15`).
- `l`: In Thumb2 mode, low 32-bit GPR registers (`r0-r7`). In ARM mode, same as `r`.
- `h`: In Thumb2 mode, a high 32-bit GPR register (`r8-r15`). In ARM mode, invalid.
- `w`: A 32, 64, or 128-bit floating-point/SIMD register: `s0-s31`, `d0-d31`, or `q0-q15`.
- `x`: A 32, 64, or 128-bit floating-point/SIMD register: `s0-s15`, `d0-d7`, or `q0-q3`.
- `t`: A floating-point/SIMD register, only supports 32-bit values: `s0-s31`.

ARM's Thumb1 mode:

- `I`: An immediate integer between 0 and 255.
- `J`: An immediate integer between -255 and -1.
- `K`: An immediate integer between 0 and 255, with optional left-shift by some amount.
- `L`: An immediate integer between -7 and 7.
- `M`: An immediate integer which is a multiple of 4 between 0 and 1020.
- `N`: An immediate integer between 0 and 31.
- `O`: An immediate integer which is a multiple of 4 between -508 and 508.
- `r`: A low 32-bit GPR register (`r0-r7`).
- `l`: A low 32-bit GPR register (`r0-r7`).
- `h`: A high GPR register (`r0-r7`).
- `w`: A 32, 64, or 128-bit floating-point/SIMD register: `s0-s31`, `d0-d31`, or `q0-q15`.
- `x`: A 32, 64, or 128-bit floating-point/SIMD register: `s0-s15`, `d0-d7`, or `q0-q3`.
- `t`: A floating-point/SIMD register, only supports 32-bit values: `s0-s31`.

Hexagon:

- `o`, `v`: A memory address operand, treated the same as constraint `m`, at the moment.
- `r`: A 32 or 64-bit register.

MSP430:

- `r`: An 8 or 16-bit register.

MIPS:

- `I`: An immediate signed 16-bit integer.
- `J`: An immediate integer zero.
- `K`: An immediate unsigned 16-bit integer.
- `L`: An immediate 32-bit integer, where the lower 16 bits are 0.
- `N`: An immediate integer between -65535 and -1.
- `O`: An immediate signed 15-bit integer.
- `P`: An immediate integer between 1 and 65535.
- `m`: A memory address operand. In MIPS-SE mode, allows a base address register plus 16-bit immediate offset. In MIPS mode, just a base register.
- `R`: A memory address operand. In MIPS-SE mode, allows a base address register plus a 9-bit signed offset. In MIPS mode, the same as constraint `m`.
- `ZC`: A memory address operand, suitable for use in a `pref`, `ll`, or `sc` instruction on the given subtarget (details vary).
- `r`, `d`, `y`: A 32 or 64-bit GPR register.
- `f`: A 32 or 64-bit FPU register (`F0-F31`), or a 128-bit MSA register (`W0-W31`). In the case of MSA registers, it is recommended to use the `w` argument modifier for compatibility with GCC.
- `c`: A 32-bit or 64-bit GPR register suitable for indirect jump (always `25`).
- `l`: The `lo` register, 32 or 64-bit.
- `x`: Invalid.

NVPTX:

- `b`: A 1-bit integer register.
- `c` or `h`: A 16-bit integer register.
- `r`: A 32-bit integer register.
- `l` or `N`: A 64-bit integer register.
- `f`: A 32-bit float register.
- `d`: A 64-bit float register.

PowerPC:

- `I`: An immediate signed 16-bit integer.
- `J`: An immediate unsigned 16-bit integer, shifted left 16 bits.
- `K`: An immediate unsigned 16-bit integer.
- `L`: An immediate signed 16-bit integer, shifted left 16 bits.
- `M`: An immediate integer greater than 31.
- `N`: An immediate integer that is an exact power of 2.
- `O`: The immediate integer constant 0.
- `P`: An immediate integer constant whose negation is a signed 16-bit constant.
- `es`, `o`, `Q`, `Z`, `Zy`: A memory address operand, currently treated the same as `m`.
- `r`: A 32 or 64-bit integer register.
- `b`: A 32 or 64-bit integer register, excluding `R0` (that is: `R1-R31`).
- `f`: A 32 or 64-bit float register (`F0-F31`), or when QPX is enabled, a 128 or 256-bit QPX register (`Q0-Q31`; aliases the `F` registers).
- `v`: For `4 x f32` or `4 x f64` types, when QPX is enabled, a 128 or 256-bit QPX register (`Q0-Q31`), otherwise a 128-bit altivec vector register (`V0-V31`).
- `y`: Condition register (`CR0-CR7`).
- `wc`: An individual CR bit in a CR register.
- `wa`, `wd`, `wf`: Any 128-bit VSX vector register, from the full VSX register set (overlapping both the floating-point and vector register files).
- `ws`: A 32 or 64-bit floating point register, from the full VSX register set.

Sparc:

- `I`: An immediate 13-bit signed integer.
- `r`: A 32-bit integer register.

SystemZ:

- `I`: An immediate unsigned 8-bit integer.
- `J`: An immediate unsigned 12-bit integer.
- `K`: An immediate signed 16-bit integer.
- `L`: An immediate signed 20-bit integer.
- `M`: An immediate integer 0x7fffffff.
- `Q`: A memory address operand with a base address and a 12-bit immediate unsigned displacement.
- `R`: A memory address operand with a base address, a 12-bit immediate unsigned displacement, and an index register.
- `S`: A memory address operand with a base address and a 20-bit immediate signed displacement.
- `T`: A memory address operand with a base address, a 20-bit immediate signed displacement, and an index register.
- `r` or `d`: A 32, 64, or 128-bit integer register.
- `a`: A 32, 64, or 128-bit integer address register (excludes R0, which in an address context evaluates as zero).
- `h`: A 32-bit value in the high part of a 64bit data register (LLVM-specific)
- `f`: A 32, 64, or 128-bit floating point register.

X86:

- `I`: An immediate integer between 0 and 31.
- `J`: An immediate integer between 0 and 64.
- `K`: An immediate signed 8-bit integer.
- `L`: An immediate integer, 0xff or 0xffff or (in 64-bit mode only) 0xffffffff.
- `M`: An immediate integer between 0 and 3.
- `N`: An immediate unsigned 8-bit integer.
- `O`: An immediate integer between 0 and 127.
- `e`: An immediate 32-bit signed integer.

- `Z`: An immediate 32-bit unsigned integer.
- `o`, `v`: Treated the same as `m`, at the moment.
- `q`: An 8, 16, 32, or 64-bit register which can be accessed as an 8-bit `l` integer register. On X86-32, this is the `a`, `b`, `c`, and `d` registers, and on X86-64, it is all of the integer registers.
- `Q`: An 8, 16, 32, or 64-bit register which can be accessed as an 8-bit `h` integer register. This is the `a`, `b`, `c`, and `d` registers.
- `r` or `l`: An 8, 16, 32, or 64-bit integer register.
- `R`: An 8, 16, 32, or 64-bit "legacy" integer register – one which has existed since i386, and can be accessed without the REX prefix.
- `f`: A 32, 64, or 80-bit '387 FPU stack pseudo-register.
- `y`: A 64-bit MMX register, if MMX is enabled.
- `x`: If SSE is enabled: a 32 or 64-bit scalar operand, or 128-bit vector operand in a SSE register. If AVX is also enabled, can also be a 256-bit vector operand in an AVX register. If AVX-512 is also enabled, can also be a 512-bit vector operand in an AVX512 register, Otherwise, an error.
- `Y`: The same as `x`, if *SSE2* is enabled, otherwise an error.
- `A`: Special case: allocates EAX first, then EDX, for a single operand (in 32-bit mode, a 64-bit integer operand will get split into two registers). It is not recommended to use this constraint, as in 64-bit mode, the 64-bit operand will get allocated only to RAX – if two 32-bit operands are needed, you're better off splitting it yourself, before passing it to the asm statement.

XCore:

- `r`: A 32-bit integer register.

### Asm template argument modifiers

In the asm template string, modifiers can be used on the operand reference, like "`${0:n}`".

The modifiers are, in general, expected to behave the same way they do in GCC. LLVM's support is often implemented on an 'as-needed' basis, to support C inline asm code which was supported by GCC. A mismatch in behavior between LLVM and GCC likely indicates a bug in LLVM.

Target-independent:

- `c`: Print an immediate integer constant unadorned, without the target-specific immediate punctuation (e.g. no `$` prefix).
- `n`: Negate and print immediate integer constant unadorned, without the target-specific immediate punctuation (e.g. no `$` prefix).
- `l`: Print as an unadorned label, without the target-specific label punctuation (e.g. no `$` prefix).

AArch64:

- `w`: Print a GPR register with a `w*` name instead of `x*` name. E.g., instead of `x30`, print `w30`.
- `x`: Print a GPR register with a `x*` name. (this is the default, anyhow).
- `b`, `h`, `s`, `d`, `q`: Print a floating-point/SIMD register with a `b*`, `h*`, `s*`, `d*`, or `q*` name, rather than the default of `v*`.

AMDGPU:

- `r`: No effect.

ARM:

- `a`: Print an operand as an address (with `[` and `]` surrounding a register).
- `P`: No effect.
- `q`: No effect.

- `y`: Print a VFP single-precision register as an indexed double (e.g. print as `d4[1]` instead of `s9`)
- `B`: Bitwise invert and print an immediate integer constant without # prefix.
- `L`: Print the low 16-bits of an immediate integer constant.
- `M`: Print as a register set suitable for ldm/stm. Also prints *all* register operands subsequent to the specified one (`!`), so use carefully.
- `Q`: Print the low-order register of a register-pair, or the low-order register of a two-register operand.
- `R`: Print the high-order register of a register-pair, or the high-order register of a two-register operand.
- `H`: Print the second register of a register-pair. (On a big-endian system, `H` is equivalent to `Q`, and on little-endian system, `H` is equivalent to `R`.)
- `e`: Print the low doubleword register of a NEON quad register.
- `f`: Print the high doubleword register of a NEON quad register.
- `m`: Print the base register of a memory operand without the `[` and `]` adornment.

Hexagon:

- `L`: Print the second register of a two-register operand. Requires that it has been allocated consecutively to the first.
- `I`: Print the letter 'i' if the operand is an integer constant, otherwise nothing. Used to print 'addi' vs 'add' instructions.

MSP430:

No additional modifiers.

MIPS:

- `X`: Print an immediate integer as hexadecimal
- `x`: Print the low 16 bits of an immediate integer as hexadecimal.
- `d`: Print an immediate integer as decimal.
- `m`: Subtract one and print an immediate integer as decimal.
- `z`: Print $0 if an immediate zero, otherwise print normally.
- `L`: Print the low-order register of a two-register operand, or prints the address of the low-order word of a double-word memory operand.
- `M`: Print the high-order register of a two-register operand, or prints the address of the high-order word of a double-word memory operand.
- `D`: Print the second register of a two-register operand, or prints the second word of a double-word memory operand. (On a big-endian system, `D` is equivalent to `L`, and on little-endian system, `D` is equivalent to `M`.)
- `w`: No effect. Provided for compatibility with GCC which requires this modifier in order to print MSA registers (`W0-W31`) with the `f` constraint.

NVPTX:

- `r`: No effect.

PowerPC:

- `L`: Print the second register of a two-register operand. Requires that it has been allocated consecutively to the first.
- `I`: Print the letter 'i' if the operand is an integer constant, otherwise nothing. Used to print 'addi' vs 'add' instructions.
- `y`: For a memory operand, prints formatter for a two-register X-form instruction. (Currently always prints `r0,OPERAND`).
- `U`: Prints 'u' if the memory operand is an update form, and nothing otherwise. (NOTE: LLVM does not support update form, so this will currently always print nothing)

- `X`: Prints 'x' if the memory operand is an indexed form. (NOTE: LLVM does not support indexed form, so this will currently always print nothing)

Sparc:

- `r`: No effect.

SystemZ:

SystemZ implements only `n`, and does *not* support any of the other target-independent modifiers.

X86:

- `c`: Print an unadorned integer or symbol name. (The latter is target-specific behavior for this typically target-independent modifier).
- `A`: Print a register name with a '*' before it.
- `b`: Print an 8-bit register name (e.g. `al`); do nothing on a memory operand.
- `h`: Print the upper 8-bit register name (e.g. `ah`); do nothing on a memory operand.
- `w`: Print the 16-bit register name (e.g. `ax`); do nothing on a memory operand.
- `k`: Print the 32-bit register name (e.g. `eax`); do nothing on a memory operand.
- `q`: Print the 64-bit register name (e.g. `rax`), if 64-bit registers are available, otherwise the 32-bit register name; do nothing on a memory operand.
- `n`: Negate and print an unadorned integer, or, for operands other than an immediate integer (e.g. a relocatable symbol expression), print a '-' before the operand. (The behavior for relocatable symbol expressions is a target-specific behavior for this typically target-independent modifier)
- `H`: Print a memory reference with additional offset +8.
- `P`: Print a memory reference or operand for use as the argument of a call instruction. (E.g. omit `(rip)`, even though it's PC-relative.)

XCore:

No additional modifiers.

### 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

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.

Metadata does not have a type, and is not a value. If referenced from a `call` instruction, it uses the `metadata` type.

All metadata are identified in syntax by a exclamation point ('!').

## Metadata Nodes and Metadata Strings

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:

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

Metadata nodes that aren't uniqued use the `distinct` keyword. For example:

```
!0 = distinct !{!"test\00", i32 10}
```

`distinct` nodes are useful when nodes shouldn't be merged based on their content. They can also occur when transformations cause uniquing collisions when metadata operands change.

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

```
!foo = !{!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 to an instruction. Here metadata `!21` is attached to the `add` instruction using the `!dbg` identifier:

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

Metadata can also be attached to a function definition. Here metadata `!22` is attached to the `foo` function using the `!dbg` identifier:

```
define void @foo() !dbg !22 {
  ret void
}
```

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

### Specialized Metadata Nodes

Specialized metadata nodes are custom data structures in metadata (as opposed to generic tuples). Their fields are labelled, and can be specified in any order.

These aren't inherently debug info centric, but currently all the specialized metadata nodes are related to debug info.

#### DICompileUnit

`DICompileUnit` nodes represent a compile unit. The `enums:`, `retainedTypes:`, `subprograms:`, `globals:`, `imports:` and `macros:` fields are tuples containing the debug info to be emitted along with the compile unit, regardless of code optimizations (some nodes are only emitted if there are references to them from instructions).

```
!0 = !DICompileUnit(language: DW_LANG_C99, file: !1, producer: "clang",
                    isOptimized: true, flags: "-O2", runtimeVersion: 2,
```

```
                      splitDebugFilename: "abc.debug", emissionKind: FullDebug,
                      enums: !2, retainedTypes: !3, subprograms: !4,
                      globals: !5, imports: !6, macros: !7, dwoId: 0x0abcd)
```

Compile unit descriptors provide the root scope for objects declared in a specific compilation unit. File descriptors are defined using this scope. These descriptors are collected by a named metadata `!llvm.dbg.cu`. They keep track of subprograms, global variables, type information, and imported entities (declarations and namespaces).

#### DIFile

`DIFile` nodes represent files. The `filename:` can include slashes.

```
!0 = !DIFile(filename: "path/to/file", directory: "/path/to/dir")
```

Files are sometimes used in `scope:` fields, and are the only valid target for `file:` fields.

#### DIBasicType

`DIBasicType` nodes represent primitive types, such as `int`, `bool` and `float`. `tag:` defaults to `DW_TAG_base_type`.

```
!0 = !DIBasicType(name: "unsigned char", size: 8, align: 8,
                  encoding: DW_ATE_unsigned_char)
!1 = !DIBasicType(tag: DW_TAG_unspecified_type, name: "decltype(nullptr)")
```

The `encoding:` describes the details of the type. Usually it's one of the following:

```
DW_ATE_address       = 1
DW_ATE_boolean       = 2
DW_ATE_float         = 4
DW_ATE_signed        = 5
DW_ATE_signed_char   = 6
DW_ATE_unsigned      = 7
DW_ATE_unsigned_char = 8
```

#### DISubroutineType

`DISubroutineType` nodes represent subroutine types. Their `types:` field refers to a tuple; the first operand is the return type, while the rest are the types of the formal arguments in order. If the first operand is `null`, that represents a function with no return value (such as `void foo() {}` in C++).

```
!0 = !BasicType(name: "int", size: 32, align: 32, DW_ATE_signed)
!1 = !BasicType(name: "char", size: 8, align: 8, DW_ATE_signed_char)
!2 = !DISubroutineType(types: !{null, !0, !1}) ; void (int, char)
```

#### DIDerivedType

`DIDerivedType` nodes represent types derived from other types, such as qualified types.

```
!0 = !DIBasicType(name: "unsigned char", size: 8, align: 8,
                  encoding: DW_ATE_unsigned_char)
!1 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !0, size: 32,
                    align: 32)
```

The following `tag:` values are valid:

```
DW_TAG_member             = 13
DW_TAG_pointer_type       = 15
DW_TAG_reference_type     = 16
DW_TAG_typedef            = 22
DW_TAG_inheritance        = 28
DW_TAG_ptr_to_member_type = 31
```

```
DW_TAG_const_type         = 38
DW_TAG_friend             = 42
DW_TAG_volatile_type      = 53
DW_TAG_restrict_type      = 55
```

`DW_TAG_member` is used to define a member of a *composite type*. The type of the member is the `baseType:`. The `offset:` is the member's bit offset. If the composite type has an ODR `identifier:` and does not set `flags: DIFwdDecl`, then the member is uniqued based only on its `name:` and `scope:`.

`DW_TAG_inheritance` and `DW_TAG_friend` are used in the `elements:` field of *composite types* to describe parents and friends.

`DW_TAG_typedef` is used to provide a name for the `baseType:`.

`DW_TAG_pointer_type`, `DW_TAG_reference_type`, `DW_TAG_const_type`, `DW_TAG_volatile_type` and `DW_TAG_restrict_type` are used to qualify the `baseType:`.

Note that the `void *` type is expressed as a type derived from NULL.

### DICompositeType

`DICompositeType` nodes represent types composed of other types, like structures and unions. `elements:` points to a tuple of the composed types.

If the source language supports ODR, the `identifier:` field gives the unique identifier used for type merging between modules. When specified, *subprogram declarations* and *member derived types* that reference the ODR-type in their `scope:` change uniquing rules.

For a given `identifier:`, there should only be a single composite type that does not have `flags: DIFlagFwdDecl` set. LLVM tools that link modules together will unique such definitions at parse time via the `identifier:` field, even if the nodes are `distinct`.

```
!0 = !DIEnumerator(name: "SixKind", value: 7)
!1 = !DIEnumerator(name: "SevenKind", value: 7)
!2 = !DIEnumerator(name: "NegEightKind", value: -8)
!3 = !DICompositeType(tag: DW_TAG_enumeration_type, name: "Enum", file: !12,
                      line: 2, size: 32, align: 32, identifier: "_M4Enum",
                      elements: !{!0, !1, !2})
```

The following `tag:` values are valid:

```
DW_TAG_array_type       = 1
DW_TAG_class_type       = 2
DW_TAG_enumeration_type = 4
DW_TAG_structure_type   = 19
DW_TAG_union_type       = 23
```

For `DW_TAG_array_type`, the `elements:` should be *subrange descriptors*, each representing the range of subscripts at that level of indexing. The `DIFlagVector` flag to `flags:` indicates that an array type is a native packed vector.

For `DW_TAG_enumeration_type`, the `elements:` should be *enumerator descriptors*, each representing the definition of an enumeration value for the set. All enumeration type descriptors are collected in the `enums:` field of the *compile unit*.

For `DW_TAG_structure_type`, `DW_TAG_class_type`, and `DW_TAG_union_type`, the `elements:` should be *derived types* with `tag: DW_TAG_member`, `tag: DW_TAG_inheritance`, or `tag: DW_TAG_friend`; or *subprograms* with `isDefinition: false`.

### DISubrange

`DISubrange` nodes are the elements for `DW_TAG_array_type` variants of *DICompositeType*. `count: -1` indicates an empty array.

```
!0 = !DISubrange(count: 5, lowerBound: 0) ; array counting from 0
!1 = !DISubrange(count: 5, lowerBound: 1) ; array counting from 1
!2 = !DISubrange(count: -1) ; empty array.
```

### DIEnumerator

`DIEnumerator` nodes are the elements for `DW_TAG_enumeration_type` variants of *DICompositeType*.

```
!0 = !DIEnumerator(name: "SixKind", value: 7)
!1 = !DIEnumerator(name: "SevenKind", value: 7)
!2 = !DIEnumerator(name: "NegEightKind", value: -8)
```

### DITemplateTypeParameter

`DITemplateTypeParameter` nodes represent type parameters to generic source language constructs. They are used (optionally) in *DICompositeType* and *DISubprogram* `templateParams:` fields.

```
!0 = !DITemplateTypeParameter(name: "Ty", type: !1)
```

### DITemplateValueParameter

`DITemplateValueParameter` nodes represent value parameters to generic source language constructs. `tag:` defaults to `DW_TAG_template_value_parameter`, but if specified can also be set to `DW_TAG_GNU_template_template_param` or `DW_TAG_GNU_template_param_pack`. They are used (optionally) in *DICompositeType* and *DISubprogram* `templateParams:` fields.

```
!0 = !DITemplateValueParameter(name: "Ty", type: !1, value: i32 7)
```

### DINamespace

`DINamespace` nodes represent namespaces in the source language.

```
!0 = !DINamespace(name: "myawesomeproject", scope: !1, file: !2, line: 7)
```

### DIGlobalVariable

`DIGlobalVariable` nodes represent global variables in the source language.

```
!0 = !DIGlobalVariable(name: "foo", linkageName: "foo", scope: !1,
                       file: !2, line: 7, type: !3, isLocal: true,
                       isDefinition: false, variable: i32* @foo,
                       declaration: !4)
```

All global variables should be referenced by the *`globals:`* field of a *compile unit*.

### DISubprogram

`DISubprogram` nodes represent functions from the source language. A `DISubprogram` may be attached to a function definition using `!dbg` metadata. The `variables:` field points at *variables* that must be retained, even if their IR counterparts are optimized out of the IR. The `type:` field must point at an *DISubroutineType*.

When `isDefinition: false`, subprograms describe a declaration in the type tree as opposed to a definition of a function. If the scope is a composite type with an ODR `identifier:` and that does not set `flags: DIFwdDecl`, then the subprogram declaration is uniqued based only on its `linkageName:` and `scope:`.

```
define void @_Z3foov() !dbg !0 {
  ...
}

!0 = distinct !DISubprogram(name: "foo", linkageName: "_Zfoov", scope: !1,
                            file: !2, line: 7, type: !3, isLocal: true,
                            isDefinition: true, scopeLine: 8,
                            containingType: !4,
                            virtuality: DW_VIRTUALITY_pure_virtual,
                            virtualIndex: 10, flags: DIFlagPrototyped,
                            isOptimized: true, templateParams: !5,
                            declaration: !6, variables: !7)
```

### DILexicalBlock

`DILexicalBlock` nodes describe nested blocks within a *subprogram*. The line number and column numbers are used to distinguish two lexical blocks at same depth. They are valid targets for `scope:` fields.

```
!0 = distinct !DILexicalBlock(scope: !1, file: !2, line: 7, column: 35)
```

Usually lexical blocks are `distinct` to prevent node merging based on operands.

### DILexicalBlockFile

`DILexicalBlockFile` nodes are used to discriminate between sections of a *lexical block*. The `file:` field can be changed to indicate textual inclusion, or the `discriminator:` field can be used to discriminate between control flow within a single block in the source language.

```
!0 = !DILexicalBlock(scope: !3, file: !4, line: 7, column: 35)
!1 = !DILexicalBlockFile(scope: !0, file: !4, discriminator: 0)
!2 = !DILexicalBlockFile(scope: !0, file: !4, discriminator: 1)
```

### DILocation

`DILocation` nodes represent source debug locations. The `scope:` field is mandatory, and points at an *DILexicalBlockFile*, an *DILexicalBlock*, or an *DISubprogram*.

```
!0 = !DILocation(line: 2900, column: 42, scope: !1, inlinedAt: !2)
```

### DILocalVariable

`DILocalVariable` nodes represent local variables in the source language. If the `arg:` field is set to non-zero, then this variable is a subprogram parameter, and it will be included in the `variables:` field of its *DISubprogram*.

```
!0 = !DILocalVariable(name: "this", arg: 1, scope: !3, file: !2, line: 7,
                      type: !3, flags: DIFlagArtificial)
!1 = !DILocalVariable(name: "x", arg: 2, scope: !4, file: !2, line: 7,
                      type: !3)
!2 = !DILocalVariable(name: "y", scope: !5, file: !2, line: 7, type: !3)
```

### DIExpression

`DIExpression` nodes represent DWARF expression sequences. They are used in *debug intrinsics* (such as `llvm.dbg.declare`) to describe how the referenced LLVM variable relates to the source language variable.

The current supported vocabulary is limited:

- `DW_OP_deref` dereferences the working expression.
- `DW_OP_plus, 93` adds 93 to the working expression.

- `DW_OP_bit_piece, 16, 8` specifies the offset and size (16 and 8 here, respectively) of the variable piece from the working expression.

```
!0 = !DIExpression(DW_OP_deref)
!1 = !DIExpression(DW_OP_plus, 3)
!2 = !DIExpression(DW_OP_bit_piece, 3, 7)
!3 = !DIExpression(DW_OP_deref, DW_OP_plus, 3, DW_OP_bit_piece, 3, 7)
```

### DIObjCProperty

`DIObjCProperty` nodes represent Objective-C property nodes.

```
!3 = !DIObjCProperty(name: "foo", file: !1, line: 7, setter: "setFoo",
                     getter: "getFoo", attributes: 7, type: !2)
```

### DIImportedEntity

`DIImportedEntity` nodes represent entities (such as modules) imported into a compile unit.

```
!2 = !DIImportedEntity(tag: DW_TAG_imported_module, name: "foo", scope: !0,
                       entity: !1, line: 7)
```

### DIMacro

`DIMacro` nodes represent definition or undefinition of a macro identifiers. The `name:` field is the macro identifier, followed by macro parameters when defining a function-like macro, and the `value` field is the token-string used to expand the macro identifier.

```
!2 = !DIMacro(macinfo: DW_MACINFO_define, line: 7, name: "foo(x)",
              value: "((x) + 1)")
!3 = !DIMacro(macinfo: DW_MACINFO_undef, line: 30, name: "foo")
```

### DIMacroFile

`DIMacroFile` nodes represent inclusion of source files. The `nodes:` field is a list of `DIMacro` and `DIMacroFile` nodes that appear in the included source file.

```
!2 = !DIMacroFile(macinfo: DW_MACINFO_start_file, line: 7, file: !2,
                  nodes: !3)
```

## '**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 = !{ !"an example type tree" }
!1 = !{ !"int", !0 }
!2 = !{ !"float", !0 }
!3 = !{ !"const float", !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 = !{ i64 0, i64 4, !1, i64 8, i64 4, !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.

### '`noalias`' and '`alias.scope`' Metadata

`noalias` and `alias.scope` metadata provide the ability to specify generic noalias memory-access sets. This means that some collection of memory access instructions (loads, stores, memory-accessing calls, etc.) that carry `noalias` metadata can specifically be specified not to alias with some other collection of memory access instructions that carry `alias.scope` metadata. Each type of metadata specifies a list of scopes where each scope has an id and a domain.

When evaluating an aliasing query, if for some domain, the set of scopes with that domain in one instruction's `alias.scope` list is a subset of (or equal to) the set of scopes for that domain in another instruction's `noalias` list, then the two memory accesses are assumed not to alias.

Because scopes in one domain don't affect scopes in other domains, separate domains can be used to compose multiple independent noalias sets. This is used for example during inlining. As the noalias function parameters are turned into noalias scope metadata, a new domain is used every time the function is inlined.

The metadata identifying each domain is itself a list containing one or two entries. The first entry is the name of the domain. Note that if the name is a string then it can be combined across functions and translation units. A self-reference can be used to create globally unique domain names. A descriptive string may optionally be provided as a second list entry.

The metadata identifying each scope is also itself a list containing two or three entries. The first entry is the name of the scope. Note that if the name is a string then it can be combined across functions and translation units. A self-reference can be used to create globally unique scope names. A metadata reference to the scope's domain is the second entry. A descriptive string may optionally be provided as a third list entry.

For example,

```
; Two scope domains:
!0 = !{!0}
!1 = !{!1}
```

```
; Some scopes in these domains:
!2 = !{!2, !0}
!3 = !{!3, !0}
!4 = !{!4, !1}

; Some scope lists:
!5 = !{!4} ; A list containing only scope !4
!6 = !{!4, !3, !2}
!7 = !{!3}

; These two instructions don't alias:
%0 = load float, float* %c, align 4, !alias.scope !5
store float %0, float* %arrayidx.i, align 4, !noalias !5

; These two instructions also don't alias (for domain !1, the set of scopes
; in the !alias.scope equals that in the !noalias list):
%2 = load float, float* %c, align 4, !alias.scope !5
store float %2, float* %arrayidx.i2, align 4, !noalias !6

; These two instructions may alias (for domain !0, the set of scopes in
; the !noalias list is not a superset of, or equal to, the scopes in the
; !alias.scope list):
%2 = load float, float* %c, align 4, !alias.scope !6
store float %0, float* %arrayidx.i, align 4, !noalias !7
```

### '`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 float type number representing the maximum relative error, for example:

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

### '`range`' Metadata

`range` metadata may be attached only to `load`, `call` and `invoke` of integer types. It expresses the possible ranges the loaded value or the value returned by the called function at this call site is in. The ranges are represented with a flattened list of integers. The loaded value or the value returned 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, i8* %x, align 1, !range !0 ; Can only be 0 or 1
  %b = load i8, i8* %y, align 1, !range !1 ; Can only be 255 (-1), 0 or 1
  %c = call i8 @foo(),       !range !2 ; Can only be 0, 1, 3, 4 or 5
  %d = invoke i8 @bar() to label %cont
```

```
          unwind label %lpad, !range !3 ; Can only be -2, -1, 3, 4 or 5
...
!0 = !{ i8 0, i8 2 }
!1 = !{ i8 255, i8 2 }
!2 = !{ i8 0, i8 2, i8 3, i8 6 }
!3 = !{ i8 -2, i8 0, i8 3, i8 6 }
```

### '`unpredictable`' Metadata

`unpredictable` metadata may be attached to any branch or switch instruction. It can be used to express the unpredictability of control flow. Similar to the llvm.expect intrinsic, it may be used to alter optimizations related to compare and branch instructions. The metadata is treated as a boolean value; if it exists, it signals that the branch or switch that it is attached to is completely unpredictable.

### '`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 = !{!0}
!1 = !{!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.loop.unroll.count` suggests an unroll factor to the loop unroller:

```
  br i1 %exitcond, label %._crit_edge, label %.lr.ph, !llvm.loop !0
...
!0 = !{!0, !1}
!1 = !{!"llvm.loop.unroll.count", i32 4}
```

### '`llvm.loop.vectorize`' and '`llvm.loop.interleave`'

Metadata prefixed with `llvm.loop.vectorize` or `llvm.loop.interleave` are used to control per-loop vectorization and interleaving parameters such as vectorization width and interleave count. These metadata should be used in conjunction with `llvm.loop` loop identification metadata. The `llvm.loop.vectorize` and `llvm.loop.interleave` metadata are only optimization hints and the optimizer will only interleave and vectorize loops if it believes it is safe to do so. The `llvm.mem.parallel_loop_access` metadata which contains information about loop-carried memory dependencies can be helpful in determining the safety of these transformations.

### '`llvm.loop.interleave.count`' Metadata

This metadata suggests an interleave count to the loop interleaver. The first operand is the string `llvm.loop.interleave.count` and the second operand is an integer specifying the interleave count. For example:

```
!0 = !{!"llvm.loop.interleave.count", i32 4}
```

Note that setting `llvm.loop.interleave.count` to 1 disables interleaving multiple iterations of the loop. If `llvm.loop.interleave.count` is set to 0 then the interleave count will be determined

automatically.

#### '`llvm.loop.vectorize.enable`' Metadata

This metadata selectively enables or disables vectorization for the loop. The first operand is the string `llvm.loop.vectorize.enable` and the second operand is a bit. If the bit operand value is 1 vectorization is enabled. A value of 0 disables vectorization:

```
!0 = !{!"llvm.loop.vectorize.enable", i1 0}
!1 = !{!"llvm.loop.vectorize.enable", i1 1}
```

#### '`llvm.loop.vectorize.width`' Metadata

This metadata sets the target width of the vectorizer. The first operand is the string `llvm.loop.vectorize.width` and the second operand is an integer specifying the width. For example:

```
!0 = !{!"llvm.loop.vectorize.width", i32 4}
```

Note that setting `llvm.loop.vectorize.width` to 1 disables vectorization of the loop. If `llvm.loop.vectorize.width` is set to 0 or if the loop does not have this metadata the width will be determined automatically.

#### '`llvm.loop.unroll`'

Metadata prefixed with `llvm.loop.unroll` are loop unrolling optimization hints such as the unroll factor. `llvm.loop.unroll` metadata should be used in conjunction with `llvm.loop` loop identification metadata. The `llvm.loop.unroll` metadata are only optimization hints and the unrolling will only be performed if the optimizer believes it is safe to do so.

#### '`llvm.loop.unroll.count`' Metadata

This metadata suggests an unroll factor to the loop unroller. The first operand is the string `llvm.loop.unroll.count` and the second operand is a positive integer specifying the unroll factor. For example:

```
!0 = !{!"llvm.loop.unroll.count", i32 4}
```

If the trip count of the loop is less than the unroll count the loop will be partially unrolled.

#### '`llvm.loop.unroll.disable`' Metadata

This metadata disables loop unrolling. The metadata has a single operand which is the string `llvm.loop.unroll.disable`. For example:

```
!0 = !{!"llvm.loop.unroll.disable"}
```

#### '`llvm.loop.unroll.runtime.disable`' Metadata

This metadata disables runtime loop unrolling. The metadata has a single operand which is the string `llvm.loop.unroll.runtime.disable`. For example:

```
!0 = !{!"llvm.loop.unroll.runtime.disable"}
```

#### '`llvm.loop.unroll.enable`' Metadata

This metadata suggests that the loop should be fully unrolled if the trip count is known at compile time and partially unrolled if the trip count is not known at compile time. The metadata has a

single operand which is the string `llvm.loop.unroll.enable`. For example:

```
!0 = !{!"llvm.loop.unroll.enable"}
```

### '`llvm.loop.unroll.full`' Metadata

This metadata suggests that the loop should be unrolled fully. The metadata has a single operand which is the string `llvm.loop.unroll.full`. For example:

```
!0 = !{!"llvm.loop.unroll.full"}
```

### '`llvm.loop.licm_versioning.disable`' Metadata

This metadata indicates that the loop should not be versioned for the purpose of enabling loop-invariant code motion (LICM). The metadata has a single operand which is the string `llvm.loop.licm_versioning.disable`. For example:

```
!0 = !{!"llvm.loop.licm_versioning.disable"}
```

### '`llvm.loop.distribute.enable`' Metadata

Loop distribution allows splitting a loop into multiple loops. Currently, this is only performed if the entire loop cannot be vectorized due to unsafe memory dependencies. The transformation will attempt to isolate the unsafe dependencies into their own loop.

This metadata can be used to selectively enable or disable distribution of the loop. The first operand is the string `llvm.loop.distribute.enable` and the second operand is a bit. If the bit operand value is 1 distribution is enabled. A value of 0 disables distribution:

```
!0 = !{!"llvm.loop.distribute.enable", i1 0}
!1 = !{!"llvm.loop.distribute.enable", i1 1}
```

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

### '`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

The `llvm.mem.parallel_loop_access` metadata refers to a loop identifier, or metadata containing a list of loop identifiers for nested loops. The metadata is attached to memory accessing instructions and denotes that no loop carried memory dependence exist between it and other instructions denoted with the same loop identifier. The metadata on memory reads also implies that if conversion (i.e. speculative execution within a loop iteration) is safe.

Precisely, given two instructions `m1` and `m2` that both have the `llvm.mem.parallel_loop_access` metadata, with `L1` and `L2` being the set of loops associated with that metadata, respectively, then there is no loop carried dependence between `m1` and `m2` for loops in both `L1` and `L2`.

As a special case, if all memory accessing instructions in a loop have `llvm.mem.parallel_loop_access` metadata that refers to that loop, then the loop has no loop carried memory dependences and is considered to be a parallel loop.

Note that if not all memory access instructions have such metadata referring to the loop, then the loop is considered not being trivially parallel. Additional memory dependence analysis is required to make that determination. As a fail safe mechanism, this causes loops that were originally

parallel to be considered sequential (if optimization passes that are unaware of the parallel semantics insert new memory instructions into 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:
  ...
  %val0 = load i32, i32* %arrayidx, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %val0, i32* %arrayidx1, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %for.end, label %for.body, !llvm.loop !0

for.end:
...
!0 = !{!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:
  ...
  %val1 = load i32, i32* %arrayidx3, !llvm.mem.parallel_loop_access !2
  ...
  br label %inner.for.body

inner.for.body:
  ...
  %val0 = load i32, i32* %arrayidx1, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %val0, i32* %arrayidx2, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %inner.for.end, label %inner.for.body, !llvm.loop !1

inner.for.end:
  ...
  store i32 %val1, i32* %arrayidx4, !llvm.mem.parallel_loop_access !2
  ...
  br i1 %exitcond, label %outer.for.end, label %outer.for.body, !llvm.loop !2

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

### '`invariant.group`' Metadata

The `invariant.group` metadata may be attached to `load`/`store` instructions. The existence of the `invariant.group` metadata on the instruction tells the optimizer that every `load` and `store` to the same pointer operand within the same invariant group can be assumed to load or store the same value (but see the `llvm.invariant.group.barrier` intrinsic which affects when two pointers are considered the same).

Examples:

```
@unknownPtr = external global i8
...
%ptr = alloca i8
store i8 42, i8* %ptr, !invariant.group !0
call void @foo(i8* %ptr)

%a = load i8, i8* %ptr, !invariant.group !0 ; Can assume that value under %ptr didn't c
call void @foo(i8* %ptr)
%b = load i8, i8* %ptr, !invariant.group !1 ; Can't assume anything, because group char

%newPtr = call i8* @getPointer(i8* %ptr)
%c = load i8, i8* %newPtr, !invariant.group !0 ; Can't assume anything, because we only

%unknownValue = load i8, i8* @unknownPtr
store i8 %unknownValue, i8* %ptr, !invariant.group !0 ; Can assume that %unknownValue =
```

```
call void @foo(i8* %ptr)
%newPtr2 = call i8* @llvm.invariant.group.barrier(i8* %ptr)
%d = load i8, i8* %newPtr2, !invariant.group !0  ; Can't step through invariant.group.b

...
declare void @foo(i8*)
declare i8* @getPointer(i8*)
declare i8* @llvm.invariant.group.barrier(i8*)

!0 = !{!"magic ptr"}
!1 = !{!"other ptr"}
```

## 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**<br>Emits an error if two values disagree, otherwise the resulting value is that of the operands. |
| 2 | **Warning**<br>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**<br>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**<br>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. |

| Value | Behavior |
|---|---|
| 5 | **Append**<br>Appends the two values, which are required to be metadata nodes. |
| 6 | **AppendUnique**<br>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 = !{ i32 1, !"foo", i32 1 }
!1 = !{ i32 4, !"bar", i32 37 }
!2 = !{ i32 2, !"qux", i32 42 }
!3 = !{ i32 3, !"qux",
  !{
    !"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:

  ```
  !{ !"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. |

| Key | Value |
| --- | --- |
| `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 = !{ i32 6, !"Linker Options",
   !{
      !{ !"-lz" },
      !{ !"-framework", !"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.

## C type width Module Flags Metadata

The ARM backend emits a section into each generated object file describing the options that it was compiled with (in a compiler-independent way) to prevent linking incompatible objects, and to allow automatic library selection. Some of these options are not visible at the IR level, namely wchar_t width and enum width.

To pass this information to the backend, these options are encoded in module flags metadata, using the following key-value pairs:

| Key | Value |
| --- | --- |
| short_wchar | • 0 — sizeof(wchar_t) == 4<br>• 1 — sizeof(wchar_t) == 2 |
| short_enum | • 0 — Enums are at least as large as an `int`.<br>• 1 — Enums are stored in the smallest integer type which can represent all of its values. |

For example, the following metadata section specifies that the module was compiled with a `wchar_t` width of 4 bytes, and the underlying type of an enum is the smallest type which can represent all of its values:

```
!llvm.module.flags = !{!0, !1}
!0 = !{i32 1, !"short_wchar", i32 1}
!1 = !{i32 1, !"short_enum", i32 0}
```

## 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 ()*, i8* }
@llvm.global_ctors = appending global [1 x %0] [%0 { i32 65535, void ()* @ctor, i8* @da
```

The `@llvm.global_ctors` array contains a list of constructor functions, priorities, and an optional associated global or function. 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.

If the third field is present, non-null, and points to a global variable or function, the initializer function will only run if the associated data from the current module is not discarded.

### The '`llvm.global_dtors`' Global Variable

```
%0 = type { i32, void ()*, i8* }
@llvm.global_dtors = appending global [1 x %0] [%0 { i32 65535, void ()* @dtor, i8* @da
```

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

If the third field is present, non-null, and points to a global variable or function, the destructor function will only run if the associated data from the current module is not discarded.

## 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*', '*catchswitch*', '*catchret*', '*cleanupret*', 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] <ty>|<fnty> <fnptrval>(<function args>) [fn attrs
              [operand bundles] 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. '`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`.
4. '`fnty`': shall be the signature of the function being invoked. The argument types must match the types implied by this signature. This type can be omitted if the function is not varargs.
5. '`fnptrval`': An LLVM value containing a pointer to a function to be invoked. In most cases, this is a direct function invocation, but indirect `invoke`'s are just as possible, calling an arbitrary pointer to function value.
6. '`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.
7. '`normal label`': the label reached when the called function executes a '`ret`' instruction.
8. '`exception label`': the label reached when a callee returns via the *resume* instruction or other exception handling mechanism.
9. The optional *function attributes* list. Only '`noreturn`', '`nounwind`', '`readonly`' and '`readnone`' attributes are valid here.
10. The optional *operand bundles* list.

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
```

### `catchswitch` Instruction

Syntax:

```
<resultval> = catchswitch within <parent> [ label <handler1>, label <handler2>, ... ] u
<resultval> = catchswitch within <parent> [ label <handler1>, label <handler2>, ... ] u
```

Overview:

The `catchswitch` instruction is used by LLVM's exception handling system to describe the set of possible catch handlers that may be executed by the *EH personality routine*.

Arguments:

The `parent` argument is the token of the funclet that contains the `catchswitch` instruction. If the `catchswitch` is not inside a funclet, this operand may be the token `none`.

The `default` argument is the label of another basic block beginning with either a `cleanuppad` or `catchswitch` instruction. This unwind destination must be a legal target with respect to the `parent` links, as described in the exception handling documentation.

The `handlers` are a nonempty list of successor blocks that each begin with a *catchpad* instruction.

Semantics:

Executing this instruction transfers control to one of the successors in `handlers`, if appropriate, or continues to unwind via the unwind label if present.

The `catchswitch` is both a terminator and a "pad" instruction, meaning that it must be both the first non-phi instruction and last instruction in the basic block. Therefore, it must be the only non-phi instruction in the block.

Example:

```
dispatch1:
  %cs1 = catchswitch within none [label %handler0, label %handler1] unwind to caller
dispatch2:
  %cs2 = catchswitch within %parenthandler [label %handler0] unwind label %cleanup
```

### `catchret` Instruction

Syntax:

```
catchret from <token> to label <normal>
```

Overview:

The `catchret` instruction is a terminator instruction that has a single successor.

Arguments:

The first argument to a `catchret` indicates which `catchpad` it exits. It must be a *catchpad*. The second argument to a `catchret` specifies where control will transfer to next.

Semantics:

The `catchret` instruction ends an existing (in-flight) exception whose unwinding was interrupted with a *catchpad* instruction. The *personality function* gets a chance to execute arbitrary code to, for example, destroy the active exception. Control then transfers to `normal`.

The `token` argument must be a token produced by a `catchpad` instruction. If the specified `catchpad` is not the most-recently-entered not-yet-exited funclet pad (as described in the EH documentation), the `catchret`'s behavior is undefined.

Example:

```
catchret from %catch label %continue
```

### `cleanupret` Instruction

Syntax:

```
cleanupret from <value> unwind label <continue>
cleanupret from <value> unwind to caller
```

Overview:

The '`cleanupret`' instruction is a terminator instruction that has an optional successor.

Arguments:

The '`cleanupret`' instruction requires one argument, which indicates which `cleanuppad` it exits, and must be a *cleanuppad*. If the specified `cleanuppad` is not the most-recently-entered not-yet-exited funclet pad (as described in the EH documentation), the `cleanupret`'s behavior is undefined.

The '`cleanupret`' instruction also has an optional successor, `continue`, which must be the label of another basic block beginning with either a `cleanuppad` or `catchswitch` instruction. This unwind destination must be a legal target with respect to the `parent` links, as described in the exception handling documentation.

Semantics:

The '`cleanupret`' instruction indicates to the *personality function* that one *cleanuppad* it transferred control to has ended. It transfers control to `continue` or unwinds out of the function.

Example:

```
cleanupret from %cleanup unwind to caller
cleanupret from %cleanup unwind label %continue
```

### '`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) 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>
```

### '`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 i3
```

### '`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 i3
```

### `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>, <ty2> <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 of any integer type.

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>, <ty2> <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 of any integer type.

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
```

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> - I
<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, i3
```

## 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 other 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
```

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

## 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 [inalloca] <type> [, <ty> <NumElements>] [, align <alignment>]     ;
```

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. The alignment may not be greater than `1 << 29`. 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>, <ty>* <pointer>[, align <alignment>][, !nontemporal !<
<result> = load atomic [volatile] <ty>, <ty>* <pointer> [singlethread] <ordering>, alig
!<index> = !{ i32 1 }
!<deref_bytes_node> = !{i64 <dereferenceable_bytes>}
!<align_node> = !{ i64 <value_alignment> }
```

Overview:

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

Arguments:

The argument to the `load` instruction specifies the memory address from which to load. The type specified must be a *first class* type of known size (i.e. not containing an *opaque structural 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, pointer, or floating-point 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 maximum possible alignment is `1 << 29`. An alignment value higher than the size of the loaded type implies memory up to the alignment value bytes can be safely loaded without trapping in the default address space. Access of the high bytes can interfere with debugging tools, so should not be accessed if the function has the `sanitize_thread` or `sanitize_address` attributes.

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 the address operand to this load points to memory which can be assumed unchanged. Being invariant does not imply that a location is dereferenceable, but it does imply that once the location is known dereferenceable its value is henceforth unchanging.

The optional `!invariant.group` metadata must reference a single metadata name `<index>` corresponding to a metadata node. See `invariant.group` metadata.

The optional `!nonnull` metadata must reference a single metadata name `<index>` corresponding to a metadata node with no entries. The existence of the `!nonnull` metadata on the instruction tells the optimizer that the value loaded is known to never be null. This is analogous to the `nonnull` attribute on parameters and return values. This metadata can only be applied to loads of a pointer type.

The optional `!dereferenceable` metadata must reference a single metadata name `<deref_bytes_node>` corresponding to a metadata node with one `i64` entry. The existence of the `!dereferenceable` metadata on the instruction tells the optimizer that the value loaded is known to be dereferenceable. The number of bytes known to be dereferenceable is specified by the integer value in the metadata node. This is analogous to the ''dereferenceable'' attribute on parameters and return values. This metadata can only be applied to loads of a pointer type.

The optional `!dereferenceable_or_null` metadata must reference a single metadata name `<deref_bytes_node>` corresponding to a metadata node with one `i64` entry. The existence of the `!dereferenceable_or_null` metadata on the instruction tells the optimizer that the value loaded is known to be either dereferenceable or null. The number of bytes known to be dereferenceable is specified by the integer value in the metadata node. This is analogous to the ''dereferenceable_or_null'' attribute on parameters and return values. This metadata can only be applied to loads of a pointer type.

The optional `!align` metadata must reference a single metadata name `<align_node>` corresponding to a metadata node with one `i64` entry. The existence of the `!align` metadata on the instruction tells the optimizer that the value loaded is known to be aligned to a boundary specified by the integer value in the metadata node. The alignment must be a power of 2. This is analogous to the ''align'' attribute on parameters and return values. This metadata can only be applied to loads of a pointer type.

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, i32* %ptr                      ; yields i32:val = i32 3
```

### '`store`' Instruction

Syntax:

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

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*. Only values of *first class* types of known size (i.e. not containing an *opaque structural type*) can be stored.

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, pointer, or floating-point 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 maximum possible alignment is `1 << 29`. An alignment value higher than the size of the stored type implies memory up to the alignment value bytes can be stored to without trapping in the default address space. Storing to the higher bytes however may result in data races if another thread can access the same address. Introducing a data race is not allowed. Storing to the extra bytes is not allowed even in situations where a data race is known to not exist if the function has the `sanitize_address` attribute.

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.group` metadata must reference a single metadata name `<index>`. See `invariant.group` metadata.

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, 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 [weak] [volatile] <ty>* <pointer>, <ty> <cmp>, <ty> <new> [singlethread] <succe
```

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 tries to store 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 or pointer 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 success and failure *ordering* arguments specify how this `cmpxchg` synchronizes with other atomic operations. Both ordering parameters must be at least `monotonic`, the ordering constraint on failure must be no stronger than that on success, and the failure ordering cannot be either `release` or `acq_rel`.

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, the '`<new>`' is written. The original value at the location is returned, together with a flag indicating success (true) or failure (false).

If the cmpxchg operation is marked as `weak` then a spurious failure is permitted: the operation may not write `<new>` even if the comparison matched.

If the cmpxchg operation is strong (the default), the `i1` value is 1 if and only if the value loaded equals `cmp`.

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 the second ordering parameter.

Example:

```
entry:
  %orig = load atomic i32, i32* %ptr unordered, align 4                        ; yields i
  br label %loop

loop:
  %cmp = phi i32 [ %orig, %entry ], [%value_loaded, %loop]
  %squared = mul i32 %cmp, %cmp
  %val_success = cmpxchg i32* %ptr, i32 %cmp, i32 %squared acq_rel monotonic ; yields
  %value_loaded = extractvalue { i32, i1 } %val_success, 0
  %success = extractvalue { i32, i1 } %val_success, 1
  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 <ty>, <ty>* <ptrval>{, <ty> <idx>}*
<result> = getelementptr inbounds <ty>, <ty>* <ptrval>{, <ty> <idx>}*
<result> = getelementptr <ty>, <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. The instruction can also be used to calculate a vector of such addresses.

Arguments:

The first argument is always a type used as the basis for the calculations. The second argument is always a pointer or a vector of pointers, and is the base address to start from. 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, %struct.ST* %s, i64 1, i32 2, i32 1, i
  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, %struct.ST* %s, i32 1                        ; yields
  %t2 = getelementptr %struct.ST, %struct.ST* %t1, i32 0, i32 2                ; yields
  %t3 = getelementptr %struct.RT, %struct.RT* %t2, i32 0, i32 1                ; yields
  %t4 = getelementptr [10 x [20 x i32]], [10 x [20 x i32]]* %t3, i32 0, i32 5  ; yields
  %t5 = getelementptr [20 x i32], [20 x i32]* %t4, i32 0, i32 13               ; yields
  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]}, {i32, [12 x i8]}* %saptr, i64 0, i32 1
; yields i8*:vptr
%vptr = getelementptr {i32, <2 x i8>}, {i32, <2 x i8>}* %svptr, i64 0, i32 1, i32 1
; yields i8*:eptr
%eptr = getelementptr [12 x i8], [12 x i8]* %aptr, i64 0, i32 1
; yields i32*:iptr
%iptr = getelementptr [10 x i32], [10 x i32]* @arr, i16 0, i16 0
```

Vector of pointers:

The `getelementptr` returns a vector of pointers, instead of a single address, when one or more of its arguments is a vector. In such cases, all vector arguments should have the same number of elements, and every scalar argument will be effectively broadcast into a vector during address calculation.

```
; All arguments are vectors:
;   A[i] = ptrs[i] + offsets[i]*sizeof(i8)
%A = getelementptr i8, <4 x i8*> %ptrs, <4 x i64> %offsets

; Add the same scalar offset to each pointer of a vector:
;   A[i] = ptrs[i] + offset*sizeof(i8)
%A = getelementptr i8, <4 x i8*> %ptrs, i64 %offset

; Add distinct offsets to the same pointer:
;   A[i] = ptr + offsets[i]*sizeof(i8)
%A = getelementptr i8, i8* %ptr, <4 x i64> %offsets

; In all cases described above the type of the result is <4 x i8*>
```

The two following instructions are equivalent:

```
getelementptr  %struct.ST, <4 x %struct.ST*> %s, <4 x i64> %ind1,
  <4 x i32> <i32 2, i32 2, i32 2, i32 2>,
  <4 x i32> <i32 1, i32 1, i32 1, i32 1>,
  <4 x i32> %ind4,
  <4 x i64> <i64 13, i64 13, i64 13, i64 13>

getelementptr  %struct.ST, <4 x %struct.ST*> %s, <4 x i64> %ind1,
  i32 2, i32 1, <4 x i32> %ind4, i64 13
```

Let's look at the C code, where the vector version of `getelementptr` makes sense:

```
// Let's assume that we vectorize the following loop:
double *A, B; int *C;
for (int i = 0; i < size; ++i) {
  A[i] = B[C[i]];
}
```

```
; get pointers for 8 elements from array B
%ptrs = getelementptr double, double* %B, <8 x i32> %C
; load 8 elements from array B into A
%A = call <8 x double> @llvm.masked.gather.v8f64(<8 x double*> %ptrs,
     i32 8, <8 x i1> %mask, <8 x double> %passthru)
```

## 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 casts a `value` from a larger *floating point* type to a smaller *floating point* type. If the value cannot fit (i.e. overflows) within the destination type, `ty2`, then the results are undefined. If the cast produces an inexact result, how rounding is performed (e.g. truncation, also known as round to zero) is 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 fp128:0xL00000000000000004000900000000
```

### '`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 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 archi
%Y = ptrtoint i32* %P to i64                       ; yields zero extension on 32-bit a
%Z = ptrtoint <4 x i32*> %P to <4 x i64>; yields vector zero extension for a vector of
```

### `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
%Z = addrspacecast <4 x i32*> %z to <4 x float addrspace(3)*>   ; yields <4 x float ad
```

## 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 codes 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
```

### '`fcmp`' Instruction

Syntax:

```
<result> = fcmp [fast-math flags]* <cond> <ty> <op1>, <op2>     ; yields i1 or <N x i1>
```

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 codes 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.

The `fcmp` instruction can also optionally take any number of *fast-math flags*, which are optimization hints to enable otherwise unsafe floating point optimizations.

Any set of fast-math flags are legal on an `fcmp` instruction, but the only flags that have any effect on its semantics are those that allow assumptions to be made about the values of input arguments; namely `nnan`, `ninf`, and `nsz`. See *Fast-Math Flags* for more information.

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
```

### '`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 IR-level 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.

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.

If the condition is an i1 and the value arguments are vectors of the same size, then an entire vector is selected.

Example:

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

### '`call`' Instruction

Syntax:

```
<result> = [tail | musttail | notail ] call [fast-math flags] [cconv] [ret attrs] <ty>
            [ operand bundles ]
```

Overview:

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

Arguments:

This instruction requires several arguments:

1. The optional `tail` and `musttail` markers indicate that the optimizers should perform tail call optimization. The `tail` marker is a hint that can be ignored. The `musttail` marker means that the call must be tail call optimized in order for the program to be correct. The `musttail` marker provides these guarantees:
    1. The call will not cause unbounded stack growth if it is part of a recursive cycle in the call graph.
    2. Arguments with the *inalloca* attribute are forwarded in place.

Both markers imply that the callee does not access allocas or varargs from the caller. Calls marked `musttail` must obey the following additional rules:

- The call must immediately precede a *ret* instruction, or a pointer bitcast followed by a ret instruction.
- The ret instruction must return the (possibly bitcasted) value produced by the call or void.
- The caller and callee prototypes must match. Pointer types of parameters or return types may differ in pointee type, but not in address space.
- The calling conventions of the caller and callee must match.
- All ABI-impacting function attributes, such as sret, byval, inreg, returned, and inalloca, must match.
- The callee must be varargs iff the caller is varargs. Bitcasting a non-varargs function to the appropriate varargs type is legal so long as the non-varargs prefixes obey the other rules.

Tail call optimization for calls marked `tail` is guaranteed to occur if the following conditions 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 `notail` marker indicates that the optimizers should not add `tail` or `musttail` markers to the call. It is used to prevent tail call optimization from being performed on the call.
3. The optional `fast-math flags` marker indicates that the call has one or more *fast-math flags*, which are optimization hints to enable otherwise unsafe floating-point optimizations. Fast-math flags are only valid for calls that return a floating-point scalar or vector type.
4. 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.
5. The optional *Parameter Attributes* list for return values. Only `zeroext`, `signext`, and `inreg` attributes are valid here.
6. `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`.
7. `fnty`: shall be the signature of the function being called. The argument types must match the types implied by this signature. This type can be omitted if the function is not varargs.
8. `fnptrval`: An LLVM value containing a pointer to a function to be called. In most cases, this is a direct function call, but indirect `call`s are just as possible, calling an arbitrary pointer to function value.
9. `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.
10. The optional *function attributes* list. Only `noreturn`, `nounwind`, `readonly` , `readnone`, and `convergent` attributes are valid here.
11. The optional *operand bundles* list.

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 { i32, 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
%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> <clause>+
<resultval> = landingpad <resultty> 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* upon re-entry to the function. The `resultval` has the type `resultty`.

Arguments:

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* 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.

Example:

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

### '`catchpad`' Instruction

Syntax:

```
<resultval> = catchpad within <catchswitch> [<args>*]
```

Overview:

The `catchpad` instruction is used by LLVM's exception handling system to specify that a basic block begins a catch handler — one where a personality routine attempts to transfer control to catch an exception.

Arguments:

The `catchswitch` operand must always be a token produced by a *catchswitch* instruction in a predecessor block. This ensures that each `catchpad` has exactly one predecessor block, and it always terminates in a `catchswitch`.

The `args` correspond to whatever information the personality routine requires to know if this is an appropriate handler for the exception. Control will transfer to the `catchpad` if this is the first appropriate handler for the exception.

The `resultval` has the type *token* and is used to match the `catchpad` to corresponding *catchrets* and other nested EH pads.

Semantics:

When the call stack is being unwound due to an exception being thrown, the exception is compared against the `args`. If it doesn't match, control will not reach the `catchpad` instruction. The representation of `args` is entirely target and personality function-specific.

Like the *landingpad* instruction, the `catchpad` instruction must be the first non-phi of its parent basic block.

The meaning of the tokens produced and consumed by `catchpad` and other "pad" instructions is described in the Windows exception handling documentation.

When a `catchpad` has been "entered" but not yet "exited" (as described in the EH documentation), it is undefined behavior to execute a *call* or *invoke* that does not carry an appropriate *"funclet" bundle*.

Example:

```
dispatch:
  %cs = catchswitch within none [label %handler0] unwind to caller
  ;; A catch block which can catch an integer.
handler0:
  %tok = catchpad within %cs [i8** @_ZTIi]
```

### `cleanuppad` Instruction

Syntax:

```
<resultval> = cleanuppad within <parent> [<args>*]
```

Overview:

The `cleanuppad` instruction is used by LLVM's exception handling system to specify that a basic block is a cleanup block — one where a personality routine attempts to transfer control to run cleanup actions. The `args` correspond to whatever additional information the *personality function* requires to execute the cleanup. The `resultval` has the type *token* and is used to match the `cleanuppad` to corresponding *cleanuprets*. The `parent` argument is the token of the funclet that

contains the `cleanuppad` instruction. If the `cleanuppad` is not inside a funclet, this operand may be the token `none`.

Arguments:

The instruction takes a list of arbitrary values which are interpreted by the *personality function*.

Semantics:

When the call stack is being unwound due to an exception being thrown, the *personality function* transfers control to the `cleanuppad` with the aid of the personality-specific arguments. As with calling conventions, how the personality function results are represented in LLVM IR is target specific.

The `cleanuppad` instruction has several restrictions:

- A cleanup block is a basic block which is the unwind destination of an exceptional instruction.
- A cleanup block must have a '`cleanuppad`' instruction as its first non-PHI instruction.
- There can be only one '`cleanuppad`' instruction within the cleanup block.
- A basic block that is not a cleanup block may not include a '`cleanuppad`' instruction.

When a `cleanuppad` has been "entered" but not yet "exited" (as described in the EH documentation), it is undefined behavior to execute a *call* or *invoke* that does not carry an appropriate *"funclet" bundle*.

Example:

```
%tok = cleanuppad within %cs []
```

# 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.

```
; This struct is different for every platform. For most platforms,
; it is merely an i8*.
%struct.va_list = type { i8* }

; For Unix x86_64 platforms, va_list is the following struct:
; %struct.va_list = type { i32, i32, i8*, i8* }

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

  ; Read a single integer argument
  %tmp = va_arg i8* %ap2, 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's support for Accurate Garbage Collection (GC) requires the frontend to generate code containing appropriate intrinsic calls and select an appropriate GC strategy which knows how to lower these intrinsics in a manner which is appropriate for the target collector.

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

### Experimental Statepoint Intrinsics

LLVM provides an second experimental set of intrinsics for describing garbage collection safepoints in compiled code. These intrinsics are an alternative to the `llvm.gcroot` intrinsics, but are compatible with the ones for *read* and *write* barriers. The differences in approach are covered in the Garbage Collection with LLVM documentation. The intrinsics themselves are described in *Garbage Collection Safepoints in LLVM*.

#### '`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.localescape`' and '`llvm.localrecover`' Intrinsics

Syntax:

```
declare void @llvm.localescape(...)
declare i8* @llvm.localrecover(i8* %func, i8* %fp, i32 %idx)
```

Overview:

The '`llvm.localescape`' intrinsic escapes offsets of a collection of static allocas, and the '`llvm.localrecover`' intrinsic applies those offsets to a live frame pointer to recover the address of the allocation. The offset is computed during frame layout of the caller of `llvm.localescape`.

Arguments:

All arguments to '`llvm.localescape`' must be pointers to static allocas or casts of static allocas. Each function can only call '`llvm.localescape`' once, and it can only do so from the entry block.

The `func` argument to '`llvm.localrecover`' must be a constant bitcasted pointer to a function defined in the current module. The code generator cannot determine the frame allocation offset of functions defined in other modules.

The `fp` argument to '`llvm.localrecover`' must be a frame pointer of a call frame that is currently live. The return value of '`llvm.localaddress`' is one way to produce such a value, but various runtimes also expose a suitable pointer in platform-specific ways.

The `idx` argument to '`llvm.localrecover`' indicates which alloca passed to '`llvm.localescape`' to recover. It is zero-indexed.

Semantics:

These intrinsics allow a group of functions to share access to a set of local stack allocations of a one parent function. The parent function may call the `llvm.localescape` intrinsic once from the function entry block, and the child functions can use `llvm.localrecover` to access the escaped allocas. The `llvm.localescape` intrinsic blocks inlining, as inlining changes where the escaped allocas are allocated, which would break attempts to use `llvm.localrecover`.

### '`llvm.read_register`' and '`llvm.write_register`' Intrinsics

Syntax:

```
declare i32 @llvm.read_register.i32(metadata)
declare i64 @llvm.read_register.i64(metadata)
declare void @llvm.write_register.i32(metadata, i32 @value)
declare void @llvm.write_register.i64(metadata, i64 @value)
!0 = !{!"sp\00"}
```

Overview:

The `llvm.read_register` and `llvm.write_register` intrinsics provides access to the named register. The register must be valid on the architecture being compiled to. The type needs to be compatible with the register being read.

Semantics:

The `llvm.read_register` intrinsic returns the current value of the register, where possible. The `llvm.write_register` intrinsic sets the current value of the register, where possible.

This is useful to implement named register global variables that need to always be mapped to a specific register, as is common practice on bare-metal programs including OS kernels.

The compiler doesn't check for register availability or use of the used register in surrounding code, including inline assembly. Because of that, allocatable registers are not supported.

Warning: So far it only works with the stack pointer on selected architectures (ARM, AArch64, PowerPC and x86_64). Significant amount of work is needed to support other registers and even more so, allocatable registers.

### '`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.get.dynamic.area.offset`' Intrinsic

Syntax:

```
declare i32 @llvm.get.dynamic.area.offset.i32()
declare i64 @llvm.get.dynamic.area.offset.i64()

Overview:
"""""""""

The '``llvm.get.dynamic.area.offset.*``' intrinsic family is used to
get the offset from native stack pointer to the address of the most
recent dynamic alloca on the caller's stack. These intrinsics are
intendend for use in combination with
:ref:`llvm.stacksave <int_stacksave>` to get a
pointer to the most recent dynamic alloca. This is useful, for example,
for AddressSanitizer's stack unpoisoning routines.
```

Semantics:

These intrinsics return a non-negative integer value that can be used to get the address of the most recent dynamic alloca, allocated by *alloca* on the caller's stack. In particular, for targets where stack grows downwards, adding this offset to the native stack pointer would get the address of the most recent dynamic alloca. For targets where stack grows upwards, the situation is a bit more complicated, because substracting this value from stack pointer would get the address one past the end of the most recent dynamic alloca.

Although for most targets *llvm.get.dynamic.area.offset <int_get_dynamic_area_offset>* returns just a zero, for others, such as PowerPC and PowerPC64, it returns a compile-time-known constant value.

The return value type of *llvm.get.dynamic.area.offset* must match the target's generic address space's (address space 0) pointer type.

### '`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.

### '`llvm.clear_cache`' Intrinsic

Syntax:

```
declare void @llvm.clear_cache(i8*, i8*)
```

Overview:

The '`llvm.clear_cache`' intrinsic ensures visibility of modifications in the specified range to the execution unit of the processor. On targets with non-unified instruction and data cache, the implementation flushes the instruction cache.

Semantics:

On platforms with coherent instruction and data caches (e.g. x86), this intrinsic is a nop. On platforms with non-coherent instruction and data cache (e.g. ARM, MIPS), the intrinsic is lowered either to appropriate instructions or a system call, if cache flushing requires special privileges.

The default behavior is to emit a call to `__clear_cache` from the run time library.

This instrinsic does *not* empty the instruction pipeline. Modifications of the current function are outside the scope of the intrinsic.

### '`llvm.instrprof_increment`' Intrinsic

Syntax:

```
declare void @llvm.instrprof_increment(i8* <name>, i64 <hash>,
                                       i32 <num-counters>, i32 <index>)
```

Overview:

The '`llvm.instrprof_increment`' intrinsic can be emitted by a frontend for use with instrumentation based profiling. These will be lowered by the `-instrprof` pass to generate execution counts of a program at runtime.

Arguments:

The first argument is a pointer to a global variable containing the name of the entity being instrumented. This should generally be the (mangled) function name for a set of counters.

The second argument is a hash value that can be used by the consumer of the profile data to detect changes to the instrumented source, and the third is the number of counters associated with `name`. It is an error if `hash` or `num-counters` differ between two instances of `instrprof_increment` that refer to the same name.

The last argument refers to which of the counters for `name` should be incremented. It should be a value between 0 and `num-counters`.

Semantics:

This intrinsic represents an increment of a profiling counter. It will cause the `-instrprof` pass to generate the appropriate data structures and the code to increment the appropriate value, in a format that can be written out by a compiler runtime and consumed via the `llvm-profdata` tool.

### '`llvm.instrprof_value_profile`' Intrinsic

Syntax:

```
declare void @llvm.instrprof_value_profile(i8* <name>, i64 <hash>,
                                           i64 <value>, i32 <value_kind>,
                                           i32 <index>)
```

Overview:

The '`llvm.instrprof_value_profile`' intrinsic can be emitted by a frontend for use with instrumentation based profiling. This will be lowered by the `-instrprof` pass to find out the target values, instrumented expressions take in a program at runtime.

Arguments:

The first argument is a pointer to a global variable containing the name of the entity being instrumented. `name` should generally be the (mangled) function name for a set of counters.

The second argument is a hash value that can be used by the consumer of the profile data to detect changes to the instrumented source. It is an error if `hash` differs between two instances of `llvm.instrprof_*` that refer to the same name.

The third argument is the value of the expression being profiled. The profiled expression's value should be representable as an unsigned 64-bit value. The fourth argument represents the kind of value profiling that is being done. The supported value profiling kinds are enumerated through the `InstrProfValueKind` type declared in the `<include/llvm/ProfileData/InstrProf.h>` header file. The last argument is the index of the instrumented expression within `name`. It should be >= 0.

Semantics:

This intrinsic represents the point where a call to a runtime routine should be inserted for value profiling of target expressions. `-instrprof` pass will generate the appropriate data structures and replace the `llvm.instrprof_value_profile` intrinsic with the call to the profile runtime library with proper arguments.

### '`llvm.thread.pointer`' Intrinsic

Syntax:

```
declare i8* @llvm.thread.pointer()
```

Overview:

The '`llvm.thread.pointer`' intrinsic returns the value of the thread pointer.

Semantics:

The '`llvm.thread.pointer`' intrinsic returns a pointer to the TLS area for the current thread. The exact semantics of this value are target specific: it may point to the start of TLS area, to the end, or somewhere in the middle. Depending on the target, this intrinsic may read a register, call a helper function, read from an alternate memory space, or perform other operations necessary to locate the TLS area. Not all targets support this intrinsic.

## 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, and does not set errno.

### '`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.minnum.*`' Intrinsic

Syntax:

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

```
declare float     @llvm.minnum.f32(float %Val0, float %Val1)
declare double    @llvm.minnum.f64(double %Val0, double %Val1)
declare x86_fp80  @llvm.minnum.f80(x86_fp80 %Val0, x86_fp80 %Val1)
declare fp128     @llvm.minnum.f128(fp128 %Val0, fp128 %Val1)
declare ppc_fp128 @llvm.minnum.ppcf128(ppc_fp128 %Val0, ppc_fp128 %Val1)
```

Overview:

The '`llvm.minnum.*`' intrinsics return the minimum of the two arguments.

Arguments:

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

Semantics:

Follows the IEEE-754 semantics for minNum, which also match for libm's fmin.

If either operand is a NaN, returns the other non-NaN operand. Returns NaN only if both operands are NaN. If the operands compare equal, returns a value that compares equal to both operands. This means that fmin(+/-0.0, +/-0.0) could return either -0.0 or 0.0.

### '`llvm.maxnum.*`' Intrinsic

Syntax:

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

```
declare float     @llvm.maxnum.f32(float  %Val0, float  %Val1l)
declare double    @llvm.maxnum.f64(double %Val0, double %Val1)
declare x86_fp80  @llvm.maxnum.f80(x86_fp80  %Val0, x86_fp80  %Val1)
```

```
declare fp128     @llvm.maxnum.f128(fp128 %Val0, fp128 %Val1)
declare ppc_fp128 @llvm.maxnum.ppcf128(ppc_fp128  %Val0, ppc_fp128  %Val1)
```

Overview:

The '`llvm.maxnum.*`' intrinsics return the maximum of the two arguments.

Arguments:

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

Semantics:

Follows the IEEE-754 semantics for maxNum, which also match for libm's fmax.

If either operand is a NaN, returns the other non-NaN operand. Returns NaN only if both operands are NaN. If the operands compare equal, returns a value that compares equal to both operands. This means that fmax(+/-0.0, +/-0.0) could return either -0.0 or 0.0.

### '`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.bitreverse.*`' Intrinsics

Syntax:

This is an overloaded intrinsic function. You can use bitreverse on any integer type.

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

Overview:

The '`llvm.bitreverse`' family of intrinsics is used to reverse the bitpattern of an integer value; for example `0b10110110` becomes `0b01101101`.

Semantics:

The `llvm.bitreverse.iN` intrinsic returns an i16 value that has bit `M` in the input moved to bit `N-M` in the output.

### '`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>)
declare <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 vector 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>)
declare <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 vector 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 fast arithmetic overflow checking.

Each of these intrinsics returns a two-element struct. The first element of this struct contains the result of the corresponding arithmetic operation modulo $2^n$, where n is the bit width of the result. Therefore, for example, the first element of the struct returned by `llvm.sadd.with.overflow.i32` is always the same as the result of a 32-bit `add` instruction with the same operands, where the `add` is *not* modified by an `nsw` or `nuw` flag.

The second element of the result is an `i1` that is 1 if the arithmetic operation overflowed and 0 otherwise. An operation overflows if, for any values of its operands `A` and `B` and for any `N` larger than the operands' width, `ext(A op B) to iN` is not equal to `(ext(A) to iN) op (ext(B) to iN)` where `ext` is `sext` for signed overflow and `zext` for unsigned overflow, and `op` is the underlying arithmetic operation.

The behavior of these intrinsics is well-defined for all argument values.

### '`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.canonicalize.*`' Intrinsic

Syntax:

```
declare float @llvm.canonicalize.f32(float %a)
declare double @llvm.canonicalize.f64(double %b)
```

Overview:

The '`llvm.canonicalize.*`' intrinsic returns the platform specific canonical encoding of a floating point number. This canonicalization is useful for implementing certain numeric primitives such as frexp. The canonical encoding is defined by IEEE-754-2008 to be:

```
2.1.8 canonical encoding: The preferred encoding of a floating-point
representation in a format. Applied to declets, significands of finite
numbers, infinities, and NaNs, especially in decimal formats.
```

This operation can also be considered equivalent to the IEEE-754-2008 conversion of a floating-point value to the same format. NaNs are handled according to section 6.2.

Examples of non-canonical encodings:

- x87 pseudo denormals, pseudo NaNs, pseudo Infinity, Unnormals. These are converted to a canonical representation per hardware-specific protocol.
- Many normal decimal floating point numbers have non-canonical alternative encodings.
- Some machines, like GPUs or ARMv7 NEON, do not support subnormal values. These are treated as non-canonical encodings of zero and will be flushed to a zero of the same sign by this operation.

Note that per IEEE-754-2008 6.2, systems that support signaling NaNs with default exception handling must signal an invalid exception, and produce a quiet NaN result.

This function should always be implementable as multiplication by 1.0, provided that the compiler does not constant fold the operation. Likewise, division by 1.0 and `llvm.minnum(x, x)` are possible implementations. Addition with -0.0 is also sufficient provided that the rounding mode is not -Infinity.

`@llvm.canonicalize` must preserve the equality relation. That is:

- `(@llvm.canonicalize(x) == x)` is equivalent to `(x == x)`
- `(@llvm.canonicalize(x) == @llvm.canonicalize(y))` is equivalent to to `(x == y)`

Additionally, the sign of zero must be conserved: `@llvm.canonicalize(-0.0) = -0.0` and `@llvm.canonicalize(+0.0) = +0.0`

The payload bits of a NaN must be conserved, with two exceptions. First, environments which use only a single canonical representation of NaN must perform said canonicalization. Second, SNaNs must be quieted per the usual methods.

The canonicalization operation may be optimized away if:

- The input is known to be canonical. For example, it was produced by a floating-point operation that is required by the standard to be canonical.
- The result is consumed only by (or fused with) other floating-point operations. That is, the bits of the floating point value are not examined.

### '`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. This never sets errno, just as `llvm.fma.*`.

Examples:

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

## 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(float %a)
declare i16 @llvm.convert.to.fp16.f64(double %a)
```

Overview:

The `llvm.convert.to.fp16` intrinsic function performs a conversion from a conventional floating point type 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 a conventional 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(float %a)
store i16 %res, i16* @x, align 2
```

### `llvm.convert.from.fp16` Intrinsic

Syntax:

```
declare float @llvm.convert.from.fp16.f32(i16 %a)
declare double @llvm.convert.from.fp16.f64(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, i16* @x, align 2
%res = call float @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], [10 x i8]* %tramp, i32 0, i32 0
call i8* @llvm.init.trampoline(i8* %tramp1, i8* bitcast (i32 (i8*, i32, i32)* @f to 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 than 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*.

## Masked Vector Load and Store Intrinsics

LLVM provides intrinsics for predicated vector load and store operations. The predicate is specified by a mask operand, which holds one bit per vector element, switching the associated vector lane on or off. The memory addresses corresponding to the "off" lanes are not accessed. When all bits of the mask are on, the intrinsic is identical to a regular vector load or store. When all bits are off, no memory is accessed.

### '`llvm.masked.load.*`' Intrinsics

Syntax:

This is an overloaded intrinsic. The loaded data is a vector of any integer, floating point or pointer data type.

```
declare <16 x float>  @llvm.masked.load.v16f32.p0v16f32 (<16 x float>* <ptr>, i32 <alig
declare <2 x double>  @llvm.masked.load.v2f64.p0v2f64  (<2 x double>* <ptr>, i32 <align
;; The data is a vector of pointers to double
declare <8 x double*> @llvm.masked.load.v8p0f64.p0v8p0f64    (<8 x double*>* <ptr>, i32
;; The data is a vector of function pointers
declare <8 x i32 ()*> @llvm.masked.load.v8p0f_i32f.p0v8p0f_i32f (<8 x i32 ()*>* <ptr>,
```

Overview:

Reads a vector from memory according to the provided mask. The mask holds a bit for each vector lane, and is used to prevent memory accesses to the masked-off lanes. The masked-off lanes in the result vector are taken from the corresponding lanes of the `passthru` operand.

Arguments:

The first operand is the base pointer for the load. The second operand is the alignment of the source location. It must be a constant integer value. The third operand, mask, is a vector of boolean values with the same number of elements as the return type. The fourth is a pass-through value that is used to fill the masked-off lanes of the result. The return type, underlying type of the base pointer and the type of the `passthru` operand are the same vector types.

Semantics:

The `llvm.masked.load` intrinsic is designed for conditional reading of selected vector elements in a single IR operation. It is useful for targets that support vector masked loads and allows vectorizing predicated basic blocks on these targets. Other targets may support this intrinsic differently, for example by lowering it into a sequence of branches that guard scalar load operations. The result of this operation is equivalent to a regular vector load instruction followed by a 'select' between the loaded and the passthru values, predicated on the same mask. However, using this intrinsic prevents exceptions on memory access to masked-off lanes.

```
%res = call <16 x float> @llvm.masked.load.v16f32.p0v16f32 (<16 x float>* %ptr, i32 4,

;; The result of the two following instructions is identical aside from potential memor
%loadlal = load <16 x float>, <16 x float>* %ptr, align 4
%res = select <16 x i1> %mask, <16 x float> %loadlal, <16 x float> %passthru
```

### `llvm.masked.store.*` Intrinsics

Syntax:

This is an overloaded intrinsic. The data stored in memory is a vector of any integer, floating point or pointer data type.

```
declare void @llvm.masked.store.v8i32.p0v8i32  (<8  x i32>   <value>, <8  x i32>*   <pt
declare void @llvm.masked.store.v16f32.p0v16f32 (<16 x float> <value>, <16 x float>* <p
;; The data is a vector of pointers to double
declare void @llvm.masked.store.v8p0f64.p0v8p0f64    (<8 x double*> <value>, <8 x doubl
;; The data is a vector of function pointers
declare void @llvm.masked.store.v4p0f_i32f.p0v4p0f_i32f (<4 x i32 ()*> <value>, <4 x i3
```

Overview:

Writes a vector to memory according to the provided mask. The mask holds a bit for each vector lane, and is used to prevent memory accesses to the masked-off lanes.

Arguments:

The first operand is the vector value to be written to memory. The second operand is the base pointer for the store, it has the same underlying type as the value operand. The third operand is the alignment of the destination location. The fourth operand, mask, is a vector of boolean values. The types of the mask and the value operand must have the same number of vector elements.

Semantics:

The '`llvm.masked.store`' intrinsics is designed for conditional writing of selected vector elements in a single IR operation. It is useful for targets that support vector masked store and allows vectorizing predicated basic blocks on these targets. Other targets may support this intrinsic differently, for example by lowering it into a sequence of branches that guard scalar store operations. The result of this operation is equivalent to a load-modify-store sequence. However, using this intrinsic prevents exceptions and data races on memory access to masked-off lanes.

```
call void @llvm.masked.store.v16f32.p0v16f32(<16 x float> %value, <16 x float>* %ptr, i

;; The result of the following instructions is identical aside from potential data race
%oldval = load <16 x float>, <16 x float>* %ptr, align 4
%res = select <16 x i1> %mask, <16 x float> %value, <16 x float> %oldval
store <16 x float> %res, <16 x float>* %ptr, align 4
```

## Masked Vector Gather and Scatter Intrinsics

LLVM provides intrinsics for vector gather and scatter operations. They are similar to *Masked Vector Load and Store*, except they are designed for arbitrary memory accesses, rather than sequential memory accesses. Gather and scatter also employ a mask operand, which holds one bit per vector element, switching the associated vector lane on or off. The memory addresses corresponding to the "off" lanes are not accessed. When all bits are off, no memory is accessed.

### '`llvm.masked.gather.*`' Intrinsics

Syntax:

This is an overloaded intrinsic. The loaded data are multiple scalar values of any integer, floating point or pointer data type gathered together into one vector.

```
declare <16 x float> @llvm.masked.gather.v16f32   (<16 x float*> <ptrs>, i32 <alignment
declare <2 x double> @llvm.masked.gather.v2f64    (<2 x double*> <ptrs>, i32 <alignment
declare <8 x float*> @llvm.masked.gather.v8p0f32  (<8 x float**> <ptrs>, i32 <alignment
```

Overview:

Reads scalar values from arbitrary memory locations and gathers them into one vector. The memory locations are provided in the vector of pointers '`ptrs`'. The memory is accessed according to the provided mask. The mask holds a bit for each vector lane, and is used to prevent memory accesses to the masked-off lanes. The masked-off lanes in the result vector are taken from the corresponding lanes of the '`passthru`' operand.

Arguments:

The first operand is a vector of pointers which holds all memory addresses to read. The second operand is an alignment of the source addresses. It must be a constant integer value. The third operand, mask, is a vector of boolean values with the same number of elements as the return type. The fourth is a pass-through value that is used to fill the masked-off lanes of the result. The return type, underlying type of the vector of pointers and the type of the '`passthru`' operand are the same vector types.

Semantics:

The '`llvm.masked.gather`' intrinsic is designed for conditional reading of multiple scalar values from arbitrary memory locations in a single IR operation. It is useful for targets that support vector masked gathers and allows vectorizing basic blocks with data and control divergence. Other targets may support this intrinsic differently, for example by lowering it into a sequence of scalar load operations. The semantics of this operation are equivalent to a sequence of conditional scalar loads with subsequent gathering all loaded values into a single vector. The mask restricts memory access to certain lanes and facilitates vectorization of predicated basic blocks.

```
%res = call <4 x double> @llvm.masked.gather.v4f64 (<4 x double*> %ptrs, i32 8, <4 x i

;; The gather with all-true mask is equivalent to the following instruction sequence
%ptr0 = extractelement <4 x double*> %ptrs, i32 0
%ptr1 = extractelement <4 x double*> %ptrs, i32 1
%ptr2 = extractelement <4 x double*> %ptrs, i32 2
%ptr3 = extractelement <4 x double*> %ptrs, i32 3

%val0 = load double, double* %ptr0, align 8
%val1 = load double, double* %ptr1, align 8
%val2 = load double, double* %ptr2, align 8
%val3 = load double, double* %ptr3, align 8

%vec0    = insertelement <4 x double>undef, %val0, 0
%vec01   = insertelement <4 x double>%vec0, %val1, 1
%vec012  = insertelement <4 x double>%vec01, %val2, 2
%vec0123 = insertelement <4 x double>%vec012, %val3, 3
```

### '`llvm.masked.scatter.*`' Intrinsics

Syntax:

This is an overloaded intrinsic. The data stored in memory is a vector of any integer, floating point or pointer data type. Each vector element is stored in an arbitrary memory address. Scatter with overlapping addresses is guaranteed to be ordered from least-significant to most-significant element.

```
declare void @llvm.masked.scatter.v8i32   (<8 x i32>     <value>, <8 x i32*>     <ptrs>
declare void @llvm.masked.scatter.v16f32  (<16 x float>  <value>, <16 x float*>  <ptrs>
declare void @llvm.masked.scatter.v4p0f64 (<4 x double*> <value>, <4 x double**> <ptrs>
```

Overview:

Writes each element from the value vector to the corresponding memory address. The memory addresses are represented as a vector of pointers. Writing is done according to the provided mask. The mask holds a bit for each vector lane, and is used to prevent memory accesses to the masked-off lanes.

Arguments:

The first operand is a vector value to be written to memory. The second operand is a vector of pointers, pointing to where the value elements should be stored. It has the same underlying type as the value operand. The third operand is an alignment of the destination addresses. The fourth operand, mask, is a vector of boolean values. The types of the mask and the value operand must have the same number of vector elements.

Semantics:

The '`llvm.masked.scatter`' intrinsics is designed for writing selected vector elements to arbitrary memory addresses in a single IR operation. The operation may be conditional, when not all bits in the mask are switched on. It is useful for targets that support vector masked scatter and allows

vectorizing basic blocks with data and control divergence. Other targets may support this intrinsic differently, for example by lowering it into a sequence of branches that guard scalar store operations.

```
;; This instruction unconditionally stores data vector in multiple addresses
call @llvm.masked.scatter.v8i32 (<8 x i32> %value, <8 x i32*> %ptrs, i32 4,  <8 x i1>

;; It is equivalent to a list of scalar stores
%val0 = extractelement <8 x i32> %value, i32 0
%val1 = extractelement <8 x i32> %value, i32 1
..
%val7 = extractelement <8 x i32> %value, i32 7
%ptr0 = extractelement <8 x i32*> %ptrs, i32 0
%ptr1 = extractelement <8 x i32*> %ptrs, i32 1
..
%ptr7 = extractelement <8 x i32*> %ptrs, i32 7
;; Note: the order of the following stores is important when they overlap:
store i32 %val0, i32* %ptr0, align 4
store i32 %val1, i32* %ptr1, align 4
..
store i32 %val7, i32* %ptr7, align 4
```

## Memory Use Markers

This class of intrinsics provides 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.

### '`llvm.invariant.group.barrier`' Intrinsic

Syntax:

```
declare i8* @llvm.invariant.group.barrier(i8* <ptr>)
```

Overview:

The '`llvm.invariant.group.barrier`' intrinsic can be used when an invariant established by invariant.group metadata no longer holds, to obtain a new pointer value that does not carry the invariant information.

Arguments:

The `llvm.invariant.group.barrier` takes only one argument, which is the pointer to the memory for which the `invariant.group` no longer holds.

Semantics:

Returns another pointer that aliases its argument but which is considered different for the purposes of `load`/`store` `invariant.group` metadata.

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

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.stackguard`' Intrinsic

Syntax:

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

Overview:

The `llvm.stackguard` intrinsic returns the system stack guard value.

It should not be generated by frontends, since it is only for internal usage. The reason why we create this intrinsic is that we still support IR form Stack Protector in FastISel.

Arguments:

None.

Semantics:

On some platforms, the value returned by this intrinsic remains unchanged between loads in the same thread. On other platforms, it returns the same global variable value, if any, e.g. `@__stack_chk_guard`.

Currently some platforms have IR-level customized stack guard loading (e.g. X86 Linux) that is not handled by `llvm.stackguard()`, while they should be in the future.

### '`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:

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

```
declare i1 @llvm.expect.i1(i1 <val>, i1 <expected_val>)
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.assume`' Intrinsic

Syntax:

```
declare void @llvm.assume(i1 %cond)
```

Overview:

The `llvm.assume` allows the optimizer to assume that the provided condition is true. This information can then be used in simplifying other parts of the code.

Arguments:

The condition which the optimizer may assume is always true.

Semantics:

The intrinsic allows the optimizer to assume that the provided condition is always true whenever the control flow reaches the intrinsic call. No code is generated for this intrinsic, and instructions that contribute only to the provided condition are not used for code generation. If the condition is violated during execution, the behavior is undefined.

Note that the optimizer might limit the transformations performed on values used by the `llvm.assume` intrinsic in order to preserve the instructions only used to form the intrinsic's input argument. This might prove undesirable if the extra information provided by the `llvm.assume` intrinsic does not cause sufficient overall improvement in code quality. For this reason, `llvm.assume` should not be used to document basic mathematical invariants that the optimizer can otherwise deduce or facts that are of little use to the optimizer.

### '`llvm.type.test`' Intrinsic

Syntax:

```
declare i1 @llvm.type.test(i8* %ptr, metadata %type) nounwind readnone
```

Arguments:

The first argument is a pointer to be tested. The second argument is a metadata object representing a *type identifier*.

Overview:

The `llvm.type.test` intrinsic tests whether the given pointer is associated with the given type identifier.

### '`llvm.type.checked.load`' Intrinsic

Syntax:

```
declare {i8*, i1} @llvm.type.checked.load(i8* %ptr, i32 %offset, metadata %type) argmer
```

Arguments:

The first argument is a pointer from which to load a function pointer. The second argument is the byte offset from which to load the function pointer. The third argument is a metadata object representing a *type identifier*.

Overview:

The `llvm.type.checked.load` intrinsic safely loads a function pointer from a virtual table pointer using type metadata. This intrinsic is used to implement control flow integrity in conjunction with virtual call optimization. The virtual call optimization pass will optimize away `llvm.type.checked.load` intrinsics associated with devirtualized calls, thereby removing the type check in cases where it is not needed to enforce the control flow integrity constraint.

If the given pointer is associated with a type metadata identifier, this function returns true as the second element of its return value. (Note that the function may also return true if the given pointer is not associated with a type metadata identifier.) If the function's return value's second element is true, the following rules apply to the first element:

- If the given pointer is associated with the given type metadata identifier, it is the function pointer loaded from the given byte offset from the given pointer.
- If the given pointer is not associated with the given type metadata identifier, it is one of the following (the choice of which is unspecified):
  1. The function pointer that would have been loaded from an arbitrarily chosen (through an unspecified mechanism) pointer associated with the type metadata.
  2. If the function has a non-void return type, a pointer to a function that returns an unspecified value without causing side effects.

If the function's return value's second element is false, the value of the first element is undefined.

### '`llvm.donothing`' Intrinsic

Syntax:

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

Overview:

The `llvm.donothing` intrinsic doesn't perform any operation. It's one of only three intrinsics (besides `llvm.experimental.patchpoint` and `llvm.experimental.gc.statepoint`) 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.

### '`llvm.experimental.deoptimize`' Intrinsic

Syntax:

```
declare type @llvm.experimental.deoptimize(...) [ "deopt"(...) ]
```

Overview:

This intrinsic, together with *deoptimization operand bundles*, allow frontends to express transfer of control and frame-local state from the currently executing (typically more specialized, hence faster) version of a function into another (typically more generic, hence slower) version.

In languages with a fully integrated managed runtime like Java and JavaScript this intrinsic can be used to implement "uncommon trap" or "side exit" like functionality. In unmanaged languages like C and C++, this intrinsic can be used to represent the slow paths of specialized functions.

Arguments:

The intrinsic takes an arbitrary number of arguments, whose meaning is decided by the *lowering strategy*.

Semantics:

The `@llvm.experimental.deoptimize` intrinsic executes an attached deoptimization continuation (denoted using a *deoptimization operand bundle*) and returns the value returned by the deoptimization continuation. Defining the semantic properties of the continuation itself is out of scope of the language reference – as far as LLVM is concerned, the deoptimization continuation can invoke arbitrary side effects, including reading from and writing to the entire heap.

Deoptimization continuations expressed using `"deopt"` operand bundles always continue execution to the end of the physical frame containing them, so all calls to `@llvm.experimental.deoptimize` must be in "tail position":

- `@llvm.experimental.deoptimize` cannot be invoked.
- The call must immediately precede a *ret* instruction.
- The `ret` instruction must return the value produced by the `@llvm.experimental.deoptimize` call if there is one, or void.

Note that the above restrictions imply that the return type for a call to `@llvm.experimental.deoptimize` will match the return type of its immediate caller.

The inliner composes the `"deopt"` continuations of the caller into the `"deopt"` continuations present in the inlinee, and also updates calls to this intrinsic to return directly from the frame of the function it inlined into.

All declarations of `@llvm.experimental.deoptimize` must share the same calling convention.

Lowering:

Calls to `@llvm.experimental.deoptimize` are lowered to calls to the symbol `__llvm_deoptimize` (it is the frontend's responsibility to ensure that this symbol is defined). The call arguments to `@llvm.experimental.deoptimize` are lowered as if they were formal arguments of the specified types, and not as varargs.

### '`llvm.experimental.guard`' Intrinsic

Syntax:

```
declare void @llvm.experimental.guard(i1, ...) [ "deopt"(...) ]
```

Overview:

This intrinsic, together with *deoptimization operand bundles*, allows frontends to express guards or checks on optimistic assumptions made during compilation. The semantics of `@llvm.experimental.guard` is defined in terms of `@llvm.experimental.deoptimize` – its body is defined to be equivalent to:

```
define void @llvm.experimental.guard(i1 %pred, <args...>) {
  %realPred = and i1 %pred, undef
  br i1 %realPred, label %continue, label %leave [, !make.implicit !{}]

leave:
  call void @llvm.experimental.deoptimize(<args...>) [ "deopt"() ]
  ret void

continue:
  ret void
}
```

with the optional `[, !make.implicit !{}]` present if and only if it is present on the call site. For more details on `!make.implicit`, see *FaultMaps and implicit checks*.

In words, `@llvm.experimental.guard` executes the attached `"deopt"` continuation if (but **not** only if) its first argument is `false`. Since the optimizer is allowed to replace the `undef` with an arbitrary value, it can optimize guard to fail "spuriously", i.e. without the original condition being false (hence the "not only if"); and this allows for "check widening" type optimizations.

`@llvm.experimental.guard` cannot be invoked.

### '`llvm.load.relative`' Intrinsic

Syntax:

```
declare i8* @llvm.load.relative.iN(i8* %ptr, iN %offset) argmemonly nounwind readonly
```

Overview:

This intrinsic loads a 32-bit value from the address `%ptr + %offset`, adds `%ptr` to that value and returns it. The constant folder specifically recognizes the form of this intrinsic and the constant initializers it may load from; if a loaded constant initializer is known to have the form `i32 trunc(x - %ptr)`, the intrinsic call is folded to `x`.

LLVM provides that the calculation of such a constant initializer will not overflow at link time under the medium code model if `x` is an `unnamed_addr` function. However, it does not provide this guarantee for a constant initializer folded into a function body. This intrinsic can be used to avoid the possibility of overflows when loading from such a constant.

## Stack Map Intrinsics

LLVM provides experimental intrinsics to support runtime patching mechanisms commonly desired in dynamic language JITs. These intrinsics are described in *Stack maps and patch points in LLVM*.