LLVM Bitcode File Format

Written by Chris Lattner, Joshua Haberman, and Peter S. Housel.

Abstract

This document describes the LLVM bitstream file format and the encoding of the LLVM IR into it.

Overview

What is commonly known as the LLVM bitcode file format (also, sometimes anachronistically known as bytecode) is actually two things: a bitstream container format and an encoding of LLVM IR into the container format.

The bitstream format is an abstract encoding of structured data, very similar to XML in some ways. Like XML, bitstream files contain tags, and nested structures, and you can parse the file without having to understand the tags. Unlike XML, the bitstream format is a binary encoding, and unlike XML it provides a mechanism for the file to self-describe "abbreviations", which are effectively size optimizations for the content.

LLVM IR files may be optionally embedded into a wrapper structure that makes it easy to embed extra data along with LLVM IR files.

This document first describes the LLVM bitstream format, describes the wrapper format, then describes the record structure used by LLVM IR files.

Bitstream Format

The bitstream format is literally a stream of bits, with a very simple structure. This structure consists of the following concepts:

- A "magic number" that identifies the contents of the stream.
- Encoding primitives like variable bit-rate integers.
- Blocks, which define nested content.
- Data Records, which describe entities within the file.
- Abbreviations, which specify compression optimizations for the file.

Note that the llvm-bcanalyzer tool can be used to dump and inspect arbitrary bitstreams, which is very useful for understanding the encoding.

Magic Numbers

The first two bytes of a bitcode file are 'BC' (0x42, 0x43). The second two bytes are an application-specific magic number. Generic bitcode tools can look at only the first two bytes to verify the file is bitcode, while application-specific programs will want to look at all four.

Primitives

A bitstream literally consists of a stream of bits, which are read in order starting with the least significant bit of each byte. The stream is made up of a number of primitive values that encode a stream of unsigned integer values. These integers are encoded in two ways: either as Fixed Width Integers or as Variable Width Integers.

Fixed Width Integers

Fixed-width integer values have their low bits emitted directly to the file. For example, a 3-bit integer value encodes 1 as 001. Fixed width integers are used when there are a well-known number of options for a field. For example, boolean values are usually encoded with a 1-bit wide integer.

Variable Width Integers

Variable-width integer (VBR) values encode values of arbitrary size, optimizing for the case where the values are small. Given a 4-bit VBR field, any 3-bit value (0 through 7) is encoded directly, with the high bit set to zero. Values larger than N-1 bits emit their bits in a series of N-1 bit chunks, where all but the last set the high bit.

For example, the value 27 (0x1B) is encoded as 1011 0011 when emitted as a vbr4 value. The first set of four bits indicates the value 3 (011) with a continuation piece (indicated by a high bit of 1). The next word indicates a value of 24 (011 << 3) with no continuation. The sum (3+24) yields the value 27.

6-bit characters

6-bit characters encode common characters into a fixed 6-bit field. They represent the following characters with the following 6-bit values:

```
'a' .. 'z' —  0 .. 25
'A' .. 'Z' — 26 .. 51
'0' .. '9' — 52 .. 61
       '.' — 62
       '_' — 63
```

This encoding is only suitable for encoding characters and strings that consist only of the above characters. It is completely incapable of encoding characters not in the set.

Word Alignment

Occasionally, it is useful to emit zero bits until the bitstream is a multiple of 32 bits. This ensures that the bit position in the stream can be represented as a multiple of 32-bit words.

Abbreviation IDs

A bitstream is a sequential series of Blocks and Data Records. Both of these start with an abbreviation ID encoded as a fixed-bitwidth field. The width is specified by the current block, as described below. The value of the abbreviation ID specifies either a builtin ID (which have special meanings, defined below) or one of the abbreviation IDs defined for the current block by the stream itself.

The set of builtin abbrev IDs is:

- `0` - `END_BLOCK` — This abbrev ID marks the end of the current block.
- `1` - `ENTER_SUBBLOCK` — This abbrev ID marks the beginning of a new block.
- `2` - `DEFINE_ABBREV` — This defines a new abbreviation.
- `3` - `UNABBREV_RECORD` — This ID specifies the definition of an unabbreviated record.

Abbreviation IDs 4 and above are defined by the stream itself, and specify an abbreviated record encoding.

Blocks

Blocks in a bitstream denote nested regions of the stream, and are identified by a content-specific id number (for example, LLVM IR uses an ID of 12 to represent function bodies). Block IDs 0-7 are reserved for standard blocks whose meaning is defined by Bitcode; block IDs 8 and greater are application specific. Nested blocks capture the hierarchical structure of the data encoded in it, and various properties are associated with blocks as the file is parsed. Block definitions allow the reader to efficiently skip blocks in constant time if the reader wants a summary of blocks, or if it wants to efficiently skip data it does not understand. The LLVM IR reader uses this mechanism to skip function bodies, lazily reading them on demand.

When reading and encoding the stream, several properties are maintained for the block. In particular, each block maintains:

1. A current abbrev id width. This value starts at 2 at the beginning of the stream, and is set every time a block record is entered. The block entry specifies the abbrev id width for the

body of the block.
2. A set of abbreviations. Abbreviations may be defined within a block, in which case they are only defined in that block (neither subblocks nor enclosing blocks see the abbreviation). Abbreviations can also be defined inside a `BLOCKINFO` block, in which case they are defined in all blocks that match the ID that the BLOCKINFO block is describing.

As sub blocks are entered, these properties are saved and the new sub-block has its own set of abbreviations, and its own abbrev id width. When a sub-block is popped, the saved values are restored.

ENTER_SUBBLOCK Encoding

`[ENTER_SUBBLOCK, blockid`$_{vbr8}$`, newabbrevlen`$_{vbr4}$`, <align32bits>, blocklen`$_{32}$`]`

The `ENTER_SUBBLOCK` abbreviation ID specifies the start of a new block record. The `blockid` value is encoded as an 8-bit VBR identifier, and indicates the type of block being entered, which can be a standard block or an application-specific block. The `newabbrevlen` value is a 4-bit VBR, which specifies the abbrev id width for the sub-block. The `blocklen` value is a 32-bit aligned value that specifies the size of the subblock in 32-bit words. This value allows the reader to skip over the entire block in one jump.

END_BLOCK Encoding

`[END_BLOCK, <align32bits>]`

The `END_BLOCK` abbreviation ID specifies the end of the current block record. Its end is aligned to 32-bits to ensure that the size of the block is an even multiple of 32-bits.

Data Records

Data records consist of a record code and a number of (up to) 64-bit integer values. The interpretation of the code and values is application specific and may vary between different block types. Records can be encoded either using an unabbrev record, or with an abbreviation. In the LLVM IR format, for example, there is a record which encodes the target triple of a module. The code is `MODULE_CODE_TRIPLE`, and the values of the record are the ASCII codes for the characters in the string.

UNABBREV_RECORD Encoding

`[UNABBREV_RECORD, code`$_{vbr6}$`, numops`$_{vbr6}$`, op0`$_{vbr6}$`, op1`$_{vbr6}$`, ...]`

An `UNABBREV_RECORD` provides a default fallback encoding, which is both completely general and extremely inefficient. It can describe an arbitrary record by emitting the code and operands as VBRs.

For example, emitting an LLVM IR target triple as an unabbreviated record requires emitting the `UNABBREV_RECORD` abbrevid, a vbr6 for the `MODULE_CODE_TRIPLE` code, a vbr6 for the length of the string, which is equal to the number of operands, and a vbr6 for each character. Because there are no letters with values less than 32, each letter would need to be emitted as at least a two-part VBR, which means that each letter would require at least 12 bits. This is not an efficient encoding, but it is fully general.

Abbreviated Record Encoding

`[<abbrevid>, fields...]`

An abbreviated record is a abbreviation id followed by a set of fields that are encoded according to the abbreviation definition. This allows records to be encoded significantly more densely than records encoded with the `UNABBREV_RECORD` type, and allows the abbreviation types to be specified in the stream itself, which allows the files to be completely self describing. The actual encoding of abbreviations is defined below.

The record code, which is the first field of an abbreviated record, may be encoded in the abbreviation definition (as a literal operand) or supplied in the abbreviated record (as a Fixed or VBR operand value).

Abbreviations

Abbreviations are an important form of compression for bitstreams. The idea is to specify a dense encoding for a class of records once, then use that encoding to emit many records. It takes space to emit the encoding into the file, but the space is recouped (hopefully plus some) when the records that use it are emitted.

Abbreviations can be determined dynamically per client, per file. Because the abbreviations are stored in the bitstream itself, different streams of the same format can contain different sets of abbreviations according to the needs of the specific stream. As a concrete example, LLVM IR files usually emit an abbreviation for binary operators. If a specific LLVM module contained no or few binary operators, the abbreviation does not need to be emitted.

DEFINE_ABBREV Encoding

`[DEFINE_ABBREV, numabbrevops`$_{\texttt{vbr5}}$`, abbrevop0, abbrevop1, ...]`

A `DEFINE_ABBREV` record adds an abbreviation to the list of currently defined abbreviations in the scope of this block. This definition only exists inside this immediate block — it is not visible in subblocks or enclosing blocks. Abbreviations are implicitly assigned IDs sequentially starting from 4 (the first application-defined abbreviation ID). Any abbreviations defined in a `BLOCKINFO` record for the particular block type receive IDs first, in order, followed by any abbreviations defined within the block itself. Abbreviated data records reference this ID to indicate what abbreviation they are invoking.

An abbreviation definition consists of the `DEFINE_ABBREV` abbrevid followed by a VBR that specifies the number of abbrev operands, then the abbrev operands themselves. Abbreviation operands come in three forms. They all start with a single bit that indicates whether the abbrev operand is a literal operand (when the bit is 1) or an encoding operand (when the bit is 0).

1. Literal operands — `[1`$_1$`, litvalue`$_{\texttt{vbr8}}$`]` — Literal operands specify that the value in the result is always a single specific value. This specific value is emitted as a vbr8 after the bit indicating that it is a literal operand.
2. Encoding info without data — `[0`$_1$`, encoding`$_3$`]` — Operand encodings that do not have extra data are just emitted as their code.
3. Encoding info with data — `[0`$_1$`, encoding`$_3$`, value`$_{\texttt{vbr5}}$`]` — Operand encodings that do have extra data are emitted as their code, followed by the extra data.

The possible operand encodings are:

- Fixed (code 1): The field should be emitted as a fixed-width value, whose width is specified by the operand's extra data.
- VBR (code 2): The field should be emitted as a variable-width value, whose width is specified by the operand's extra data.
- Array (code 3): This field is an array of values. The array operand has no extra data, but expects another operand to follow it, indicating the element type of the array. When reading an array in an abbreviated record, the first integer is a vbr6 that indicates the array length, followed by the encoded elements of the array. An array may only occur as the last operand of an abbreviation (except for the one final operand that gives the array's type).
- Char6 (code 4): This field should be emitted as a char6-encoded value. This operand type takes no extra data. Char6 encoding is normally used as an array element type.
- Blob (code 5): This field is emitted as a vbr6, followed by padding to a 32-bit boundary (for alignment) and an array of 8-bit objects. The array of bytes is further followed by tail padding to ensure that its total length is a multiple of 4 bytes. This makes it very efficient for the reader to decode the data without having to make a copy of it: it can use a pointer to the data in the mapped in file and poke directly at it. A blob may only occur as the last operand of an abbreviation.

For example, target triples in LLVM modules are encoded as a record of the form `[TRIPLE, 'a', 'b', 'c', 'd']`. Consider if the bitstream emitted the following abbrev entry:

```
[0, Fixed, 4]
[0, Array]
[0, Char6]
```

When emitting a record with this abbreviation, the above entry would be emitted as:

$$[4_{\texttt{abbrevwidth}}, 2_4, 4_{\texttt{vbr6}}, 0_6, 1_6, 2_6, 3_6]$$

These values are:

1. The first value, 4, is the abbreviation ID for this abbreviation.
2. The second value, 2, is the record code for `TRIPLE` records within LLVM IR file `MODULE_BLOCK` blocks.
3. The third value, 4, is the length of the array.
4. The rest of the values are the char6 encoded values for `"abcd"`.

With this abbreviation, the triple is emitted with only 37 bits (assuming a abbrev id width of 3). Without the abbreviation, significantly more space would be required to emit the target triple. Also, because the `TRIPLE` value is not emitted as a literal in the abbreviation, the abbreviation can also be used for any other string value.

Standard Blocks

In addition to the basic block structure and record encodings, the bitstream also defines specific built-in block types. These block types specify how the stream is to be decoded or other metadata. In the future, new standard blocks may be added. Block IDs 0-7 are reserved for standard blocks.

#0 - BLOCKINFO Block

The BLOCKINFO block allows the description of metadata for other blocks. The currently specified records are:

```
[SETBID (#1), blockid]
[DEFINE_ABBREV, ...]
[BLOCKNAME, ...name...]
[SETRECORDNAME, RecordID, ...name...]
```

The SETBID record (code 1) indicates which block ID is being described. SETBID records can occur multiple times throughout the block to change which block ID is being described. There must be a SETBID record prior to any other records.

Standard DEFINE_ABBREV records can occur inside BLOCKINFO blocks, but unlike their occurrence in normal blocks, the abbreviation is defined for blocks matching the block ID we are describing, *not* the BLOCKINFO block itself. The abbreviations defined in BLOCKINFO blocks receive abbreviation IDs as described in DEFINE_ABBREV.

The BLOCKNAME record (code 2) can optionally occur in this block. The elements of the record are the bytes of the string name of the block. llvm-bcanalyzer can use this to dump out bitcode files symbolically.

The SETRECORDNAME record (code 3) can also optionally occur in this block. The first operand value is a record ID number, and the rest of the elements of the record are the bytes for the string name of the record. llvm-bcanalyzer can use this to dump out bitcode files symbolically.

Note that although the data in BLOCKINFO blocks is described as "metadata," the abbreviations they contain are essential for parsing records from the corresponding blocks. It is not safe to skip them.

Bitcode Wrapper Format

Bitcode files for LLVM IR may optionally be wrapped in a simple wrapper structure. This structure contains a simple header that indicates the offset and size of the embedded BC file. This allows additional information to be stored alongside the BC file. The structure of this file header is:

```
[Magic32, Version32, Offset32, Size32, CPUType32]
```

Each of the fields are 32-bit fields stored in little endian form (as with the rest of the bitcode file fields). The Magic number is always 0x0B17C0DE and the version is currently always 0. The Offset field is the offset in bytes to the start of the bitcode stream in the file, and the Size field is the size in bytes of the stream. CPUType is a target-specific value that can be used to encode the CPU of the target.

LLVM IR Encoding

LLVM IR is encoded into a bitstream by defining blocks and records. It uses blocks for things like constant pools, functions, symbol tables, etc. It uses records for things like instructions, global variable descriptors, type descriptions, etc. This document does not describe the set of abbreviations that the writer uses, as these are fully self-described in the file, and the reader is not allowed to build in any knowledge of this.

Basics

LLVM IR Magic Number

The magic number for LLVM IR files is:

```
[0x0₄, 0xC₄, 0xE₄, 0xD₄]
```

When combined with the bitcode magic number and viewed as bytes, this is `"BC 0xC0DE"`.

Signed VBRs

Variable Width Integer encoding is an efficient way to encode arbitrary sized unsigned values, but is an extremely inefficient for encoding signed values, as signed values are otherwise treated as maximally large unsigned values.

As such, signed VBR values of a specific width are emitted as follows:

- Positive values are emitted as VBRs of the specified width, but with their value shifted left by one.
- Negative values are emitted as VBRs of the specified width, but the negated value is shifted left by one, and the low bit is set.

With this encoding, small positive and small negative values can both be emitted efficiently. Signed VBR encoding is used in `CST_CODE_INTEGER` and `CST_CODE_WIDE_INTEGER` records within `CONSTANTS_BLOCK` blocks.

LLVM IR Blocks

LLVM IR is defined with the following blocks:

- 8 — `MODULE_BLOCK` — This is the top-level block that contains the entire module, and describes a variety of per-module information.
- 9 — `PARAMATTR_BLOCK` — This enumerates the parameter attributes.
- 10 — `TYPE_BLOCK` — This describes all of the types in the module.
- 11 — `CONSTANTS_BLOCK` — This describes constants for a module or function.
- 12 — `FUNCTION_BLOCK` — This describes a function body.
- 13 — `TYPE_SYMTAB_BLOCK` — This describes the type symbol table.
- 14 — `VALUE_SYMTAB_BLOCK` — This describes a value symbol table.
- 15 — `METADATA_BLOCK` — This describes metadata items.
- 16 — `METADATA_ATTACHMENT` — This contains records associating metadata with function instruction values.

MODULE_BLOCK Contents

The `MODULE_BLOCK` block (id 8) is the top-level block for LLVM bitcode files, and each bitcode file must contain exactly one. In addition to records (described below) containing information about the module, a `MODULE_BLOCK` block may contain the following sub-blocks:

- BLOCKINFO
- PARAMATTR_BLOCK
- TYPE_BLOCK
- TYPE_SYMTAB_BLOCK
- VALUE_SYMTAB_BLOCK
- CONSTANTS_BLOCK
- FUNCTION_BLOCK
- METADATA_BLOCK

MODULE_CODE_VERSION Record

```
[VERSION, version#]
```

The VERSION record (code 1) contains a single value indicating the format version. Only version 0 is supported at this time.

MODULE_CODE_TRIPLE Record

```
[TRIPLE, ...string...]
```

The TRIPLE record (code 2) contains a variable number of values representing the bytes of the `target triple` specification string.

MODULE_CODE_DATALAYOUT Record

```
[DATALAYOUT, ...string...]
```

The DATALAYOUT record (code 3) contains a variable number of values representing the bytes of the `target datalayout` specification string.

MODULE_CODE_ASM Record

```
[ASM, ...string...]
```

The ASM record (code 4) contains a variable number of values representing the bytes of `module asm` strings, with individual assembly blocks separated by newline (ASCII 10) characters.

MODULE_CODE_SECTIONNAME Record

```
[SECTIONNAME, ...string...]
```

The SECTIONNAME record (code 5) contains a variable number of values representing the bytes of a single section name string. There should be one SECTIONNAME record for each section name referenced (e.g., in global variable or function `section` attributes) within the module. These records can be referenced by the 1-based index in the *section* fields of GLOBALVAR or FUNCTION records.

MODULE_CODE_DEPLIB Record

```
[DEPLIB, ...string...]
```

The `DEPLIB` record (code 6) contains a variable number of values representing the bytes of a single dependent library name string, one of the libraries mentioned in a `deplibs` declaration. There should be one `DEPLIB` record for each library name referenced.

MODULE_CODE_GLOBALVAR Record

`[GLOBALVAR, pointer type, isconst, initid, linkage, alignment, section, visibility, threadlocal]`

The `GLOBALVAR` record (code 7) marks the declaration or definition of a global variable. The operand fields are:

- *pointer type*: The type index of the pointer type used to point to this global variable
- *isconst*: Non-zero if the variable is treated as constant within the module, or zero if it is not
- *initid*: If non-zero, the value index of the initializer for this variable, plus 1.
- *linkage*: An encoding of the linkage type for this variable:
  - `external`: code 0
  - `weak`: code 1
  - `appending`: code 2
  - `internal`: code 3
  - `linkonce`: code 4
  - `dllimport`: code 5
  - `dllexport`: code 6
  - `extern_weak`: code 7
  - `common`: code 8
  - `private`: code 9
  - `weak_odr`: code 10
  - `linkonce_odr`: code 11
  - `available_externally`: code 12
  - `linker_private`: code 13
- *alignment*: The logarithm base 2 of the variable's requested alignment, plus 1
- *section*: If non-zero, the 1-based section index in the table of MODULE_CODE_SECTIONNAME entries.
- *visibility*: If present, an encoding of the visibility of this variable:
  - `default`: code 0
  - `hidden`: code 1
  - `protected`: code 2
- *threadlocal*: If present and non-zero, indicates that the variable is `thread_local`
- *unnamed_addr*: If present and non-zero, indicates that the variable has `unnamed_addr`

MODULE_CODE_FUNCTION Record

`[FUNCTION, type, callingconv, isproto, linkage, paramattr, alignment, section, visibility, gc]`

The `FUNCTION` record (code 8) marks the declaration or definition of a function. The operand fields are:

- *type*: The type index of the function type describing this function
- *callingconv*: The calling convention number:
  - `ccc`: code 0
  - `fastcc`: code 8

    - coldcc: code 9
    - x86_stdcallcc: code 64
    - x86_fastcallcc: code 65
    - arm_apcscc: code 66
    - arm_aapcscc: code 67
    - arm_aapcs_vfpcc: code 68
- *isproto*: Non-zero if this entry represents a declaration rather than a definition
- *linkage*: An encoding of the linkage type for this function
- *paramattr*: If nonzero, the 1-based parameter attribute index into the table of PARAMATTR_CODE_ENTRY entries.
- *alignment*: The logarithm base 2 of the function's requested alignment, plus 1
- *section*: If non-zero, the 1-based section index in the table of MODULE_CODE_SECTIONNAME entries.
- *visibility*: An encoding of the visibility of this function
- *gc*: If present and nonzero, the 1-based garbage collector index in the table of MODULE_CODE_GCNAME entries.
- *unnamed_addr*: If present and non-zero, indicates that the function has unnamed_addr

MODULE_CODE_ALIAS Record

```
[ALIAS, alias type, aliasee val#, linkage, visibility]
```

The ALIAS record (code 9) marks the definition of an alias. The operand fields are

- *alias type*: The type index of the alias
- *aliasee val#*: The value index of the aliased value
- *linkage*: An encoding of the linkage type for this alias
- *visibility*: If present, an encoding of the visibility of the alias

MODULE_CODE_PURGEVALS Record

```
[PURGEVALS, numvals]
```

The PURGEVALS record (code 10) resets the module-level value list to the size given by the single operand value. Module-level value list items are added by GLOBALVAR, FUNCTION, and ALIAS records. After a PURGEVALS record is seen, new value indices will start from the given *numvals* value.

MODULE_CODE_GCNAME Record

```
[GCNAME, ...string...]
```

The GCNAME record (code 11) contains a variable number of values representing the bytes of a single garbage collector name string. There should be one GCNAME record for each garbage collector name referenced in function gc attributes within the module. These records can be referenced by 1-based index in the *gc* fields of FUNCTION records.

PARAMATTR_BLOCK Contents

The PARAMATTR_BLOCK block (id 9) contains a table of entries describing the attributes of function parameters. These entries are referenced by 1-based index in the *paramattr* field of module block

FUNCTION records, or within the *attr* field of function block INST_INVOKE and INST_CALL records.

Entries within PARAMATTR_BLOCK are constructed to ensure that each is unique (i.e., no two indicies represent equivalent attribute lists).

PARAMATTR_CODE_ENTRY Record

```
[ENTRY, paramidx0, attr0, paramidx1, attr1...]
```

The ENTRY record (code 1) contains an even number of values describing a unique set of function parameter attributes. Each *paramidx* value indicates which set of attributes is represented, with 0 representing the return value attributes, 0xFFFFFFFF representing function attributes, and other values representing 1-based function parameters. Each *attr* value is a bitmap with the following interpretation:

- bit 0: `zeroext`
- bit 1: `signext`
- bit 2: `noreturn`
- bit 3: `inreg`
- bit 4: `sret`
- bit 5: `nounwind`
- bit 6: `noalias`
- bit 7: `byval`
- bit 8: `nest`
- bit 9: `readnone`
- bit 10: `readonly`
- bit 11: `noinline`
- bit 12: `alwaysinline`
- bit 13: `optsize`
- bit 14: `ssp`
- bit 15: `sspreq`
- bits 16–31: `align` *n*
- bit 32: `nocapture`
- bit 33: `noredzone`
- bit 34: `noimplicitfloat`
- bit 35: `naked`
- bit 36: `inlinehint`
- bits 37–39: `alignstack` *n*, represented as the logarithm base 2 of the requested alignment, plus 1

TYPE_BLOCK Contents

The TYPE_BLOCK block (id 10) contains records which constitute a table of type operator entries used to represent types referenced within an LLVM module. Each record (with the exception of NUMENTRY) generates a single type table entry, which may be referenced by 0-based index from instructions, constants, metadata, type symbol table entries, or other type operator records.

Entries within TYPE_BLOCK are constructed to ensure that each entry is unique (i.e., no two indicies represent structurally equivalent types).

TYPE_CODE_NUMENTRY Record

```
[NUMENTRY, numentries]
```

The NUMENTRY record (code 1) contains a single value which indicates the total number of type code entries in the type table of the module. If present, NUMENTRY should be the first record in the block.

TYPE_CODE_VOID Record

```
[VOID]
```

The VOID record (code 2) adds a `void` type to the type table.

TYPE_CODE_FLOAT Record

```
[FLOAT]
```

The FLOAT record (code 3) adds a `float` (32-bit floating point) type to the type table.

TYPE_CODE_DOUBLE Record

```
[DOUBLE]
```

The DOUBLE record (code 4) adds a `double` (64-bit floating point) type to the type table.

TYPE_CODE_LABEL Record

```
[LABEL]
```

The LABEL record (code 5) adds a `label` type to the type table.

TYPE_CODE_OPAQUE Record

```
[OPAQUE]
```

The OPAQUE record (code 6) adds an `opaque` type to the type table. Note that distinct `opaque` types are not unified.

TYPE_CODE_INTEGER Record

```
[INTEGER, width]
```

The INTEGER record (code 7) adds an integer type to the type table. The single *width* field indicates the width of the integer type.

TYPE_CODE_POINTER Record

```
[POINTER, pointee type, address space]
```

The POINTER record (code 8) adds a pointer type to the type table. The operand fields are

- *pointee type*: The type index of the pointed-to type
- *address space*: If supplied, the target-specific numbered address space where the pointed-to object resides. Otherwise, the default address space is zero.

TYPE_CODE_FUNCTION Record

```
[FUNCTION, vararg, ignored, retty, ...paramty... ]
```

The `FUNCTION` record (code 9) adds a function type to the type table. The operand fields are

- *vararg*: Non-zero if the type represents a varargs function
- *ignored*: This value field is present for backward compatibility only, and is ignored
- *retty*: The type index of the function's return type
- *paramty*: Zero or more type indices representing the parameter types of the function

TYPE_CODE_STRUCT Record

```
[STRUCT, ispacked, ...eltty...]
```

The `STRUCT` record (code 10) adds a struct type to the type table. The operand fields are

- *ispacked*: Non-zero if the type represents a packed structure
- *eltty*: Zero or more type indices representing the element types of the structure

TYPE_CODE_ARRAY Record

```
[ARRAY, numelts, eltty]
```

The `ARRAY` record (code 11) adds an array type to the type table. The operand fields are

- *numelts*: The number of elements in arrays of this type
- *eltty*: The type index of the array element type

TYPE_CODE_VECTOR Record

```
[VECTOR, numelts, eltty]
```

The `VECTOR` record (code 12) adds a vector type to the type table. The operand fields are

- *numelts*: The number of elements in vectors of this type
- *eltty*: The type index of the vector element type

TYPE_CODE_X86_FP80 Record

```
[X86_FP80]
```

The `X86_FP80` record (code 13) adds an `x86_fp80` (80-bit floating point) type to the type table.

TYPE_CODE_FP128 Record

```
[FP128]
```

The FP128 record (code 14) adds an fp128 (128-bit floating point) type to the type table.

TYPE_CODE_PPC_FP128 Record

[PPC_FP128]

The PPC_FP128 record (code 15) adds a ppc_fp128 (128-bit floating point) type to the type table.

TYPE_CODE_METADATA Record

[METADATA]

The METADATA record (code 16) adds a metadata type to the type table.

CONSTANTS_BLOCK Contents

The CONSTANTS_BLOCK block (id 11) ...

FUNCTION_BLOCK Contents

The FUNCTION_BLOCK block (id 12) ...

In addition to the record types described below, a FUNCTION_BLOCK block may contain the following sub-blocks:

- CONSTANTS_BLOCK
- VALUE_SYMTAB_BLOCK
- METADATA_ATTACHMENT

TYPE_SYMTAB_BLOCK Contents

The TYPE_SYMTAB_BLOCK block (id 13) contains entries which map between module-level named types and their corresponding type indices.

TST_CODE_ENTRY Record

[ENTRY, typeid, ...string...]

The ENTRY record (code 1) contains a variable number of values, with the first giving the type index of the designated type, and the remaining values giving the character codes of the type name. Each entry corresponds to a single named type.

VALUE_SYMTAB_BLOCK Contents

The VALUE_SYMTAB_BLOCK block (id 14) ...

METADATA_BLOCK Contents

The `METADATA_BLOCK` block (id 15) ...

METADATA_ATTACHMENT Contents

The `METADATA_ATTACHMENT` block (id 16) ...

[Chris Lattner](#)
[The LLVM Compiler Infrastructure](#)
Last modified: $Date: 2011-04-22 19:30:22 -0500 (Fri, 22 Apr 2011) $