

LLVM

## Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

©Hans de Nivelles, 2025

## History

LLVM used to mean **Low Level Virtual Machine**, but now it just means ‘LLVM’.

- Introduced in the PhD thesis of Chris Lattner at University of Illinois (2000-2004).
- A **standardized** intermediate representation,
- for which tools are available in public domain,
- based on **static single assignment (SSA)** form,
- well-documented.

## Portability?

LLVM programs are not completely portable because an LLVM program may contain library calls.

Also, an LLVM program contains choices (integer sizes, calling mechanism, etc.) that are optimized for a certain processor.

Still, the language itself is portable and is intended to be usable as intermediate representation on all platforms.

LLVM is complete. It supports all high-level constructs, and the main elements of all possible instruction sets.

It became a competitor to *C*, which also can be used as portable assembly language.

## Comparison to Our Register/Stack Machine

- Register names start with % (a small syntactic difference.)
- There is no explicit memory stack. Local variables in memory are allocated with the `alloca` statement. When they are no longer needed, they are simply forgotten.  
Parameters to functions are passed explicitly.
- Arguments of operations can be constants or variables. (We allowed only variables.)
- Type information is separated between the instruction, and data size. An instruction contains just enough type information to be able to do its job. Data contains just enough type information to know its size.
- LLVM uses *C* syntax for pointers (that is: `*`)

If you want to create LLVM by yourself, use the `clang` compiler with options

```
clang++ -c -emit-llvm -S -Wall myprog.cpp -o myprog.llvm
```

Use the search function of your editor to find the function that you are interested in. Be aware of name mangling.  $C^{++}$  encodes types into the name of a function.

In  $C$ , this problem does not exist.

You can also study the effect of optimization.

## Registers in LLVM

**Named Registers:** A register name starts with a %, followed by a letter, a dollar sign, a dot, or an underscore. After that, there may be an unbounded number of additional letters, dots, underscores, dollar signs or digits.

In addition, LLVM uses registers of form %*i*, where *i* is a counter.

As far as I see, there is no difference in meaning or scope between the two types of registers.

## First Class Types

**First Class Types** are types that the LLVM virtual machine can do calculations on. These are the types of form `iN`, `void`, the types `float/double`, labels, and pointer types.

**Single Value Types** are types that can be stored in a register of the LLVM virtual machine.

I think that every first class type is also single value, but it seems that struct and array types are single value but not first class.

## General Structure of Instruction

In LLVM, most instructions have form

`%i = function arg1, ..., argn`. Each `argi` must be a constant, or a register name.

Assignments of form `%i = %j`, or `%i = constant` are not allowed.

Wherever one needs `%i`, one can use `%j` or the constant instead.

## Allocation

As said, local variables (from the source program) have to be allocated in memory by default.

```
%res = alloca T           // Allocate one T.  
%res = alloca T, n N     // Allocate N Ts.  
%res = alloca T, align A  
    // Instructs the code generator that the  
    // address of the pointer must be a multiple of  
    // A. A is typically 4,8, or 16.
```

`%res` is always of type `T*`.

It seems that allocated memory is freed automatically when the function in which it was allocated, is exited. Somewhere during code generation, this problem needs to be solved.

## Load/Store

Loading and storing passes information between LLVM registers and memory:

```
%res = load T* Pval
%res = load T* Pval, align A
      ; %res will have type T.
```

```
store T Val, T* Pval
store T Val, T* Pval, align A
      ; Val is a constant or a register.
```

Load (or store) a value from memory.

If we know that `Pval` is aligned, then loading/storing can be more efficient. Hence it is useful to include this information in the instruction.

## Integer Operations

Signed and unsigned ints are represented by the same types `iN`, where `N` is the size in bits. Some operations are different for signed and unsigned.

```
%res = add iN Val1, Val2
```

```
%res = sub iN Val1, Val2
```

```
%res = mul iN Val1, Val2
```

```
; Work correctly on signed and unsigned.
```

## Integer Operations (2)

```
%res = udiv iN Val1, Val2
```

```
%res = urem iN Val1, Val2
```

```
%res = sdiv iN Val1, Val2
```

```
%res = srem iN Val1, Val2
```

; Division and remainder for signed and unsigned.

Dividing by zero leads to undefined behaviour. If you are certain that the division fits, you can add `exact` keyword.

## Integer Comparisons

`%res = icmp C iN Val1, Val2 ; %res is i1.` *C* must be one of:

**eq:** equal

**ne:** not equal

**ugt:** unsigned greater than

**uge:** unsigned greater or equal

**ult:** unsigned less than

**ule:** unsigned less or equal

**sgt:** signed greater than

**sge:** signed greater or equal

**slt:** signed less than

**sle:** signed less or equal

## Floating Point Operations

```
%res = fadd double/float Val1, Val2
```

```
%res = fsub double/float Val1, Val2
```

```
%res = fmul double/float Val1, Val2
```

```
%res = fdiv double/float Val1, Val2
```

## Floating Point Comparison

```
res = fcmp C double Val1, Val2 ; result is i1.
```

Comparison *C* is one of the following values:

**false:** no comparison, always returns false

**oeq:** ordered and equal

**ogt:** ordered and greater than

**oge:** ordered and ( greater than or equal )

**olt:** ordered and less than

**ole:** ordered and ( less than or equal )

**one:** ordered and not equal

**ord:** ordered (Neither of arguments is NaN.)

'ordered' means : Both arguments are defined (Not NaN).

The **unordered** comparisons accept NaN (not a number) values:

**ueq:** unordered or equal

**ugt:** unordered or greater than

**uge:** unordered or greater than or equal

**ult:** unordered or less than

**ule:** unordered or less than or equal

**une:** unordered or not equal

**uno:** unordered (either nans)

**true:** no comparison, always returns true

‘unordered’ means : One of the sides is undefined (NaN).

## Jumping and Branching

LLVM code is structured into **blocks**. A block has to be entered at the beginning, and exited at the end.

A block starts with a label, which has form `i:`

A block ends with a branch `br` statement, or a switch:

```
br label %L    ; even when there is only one choice.
```

```
br i1 C, label %L1, label %L2
```

```
switch T v, label %Def,
```

```
    [ T v1, label L1 ],
```

```
    [ T v2, label L2 ],
```

```
    ...
```

```
    [ T vn, label Ln ]
```

```
    ; Def is default branch.
```

## $\Phi$ -Functions

$\Phi$ -functions are part of the SSA normal form. In SSA, every register has a unique assignment point. (This implies that every assignment to a register is also an initialization.)

This causes a problem when a register has assignments in different branches that merge.

The  $\Phi$ -functions are needed to merge LLVM registers that had to be distinct because of SSA, into a single LLVM register again.

```
%val = phi T, [ Val1, Lab1 ], ..., [ Valn, Labn ]
```

The `Labi` are labels. If we came from the block starting with `Labi`, then choose `Vali`.

Phi functions must at the beginning of a block.

## $\Phi$ -Functions (2)

Note that the semantics of  $\Phi$  functions can differ. In LLVM, the meaning is simple: Pick the proper value, dependent on where you come from.

In scientific papers, you will find another semantics: The  $\Phi$  function magically knows which is the correct value to pick.

It does this by taking the value with the latest time stamp.

$v = \Phi(v_1, \dots, v_n)$  chooses the  $v_i$  that was assigned last.

It is tricky to get  $\Phi$ -functions in `clang` output, because local variables of the program are always stored in memory. You can get a  $\Phi$ -function in unoptimized code by using `Cond ? E1 : E2`.

## Converting Integers to Floating Point

```
%res = uitofp iN Val to double/float
```

```
%res = sitofp iN Val to double/float
```

We see (again) how integers are treated in the type system. The type of the data specifies only the size needed to store it. The rest of the type (which operations are meaningful) is part of the instruction.

## Converting Floating Point to Integers

```
%res = fptoui double/float Val to iN
```

```
%res = fptosi double/float Val to iN
```

```
// Fractions are truncated (not rounded!).
```

```
// Undefined if result does not fit.
```

## Converting between Integer Types

```
%res = trunc iN Val to iM
    // Instruction requires M < N.

%res = zext iN Val to iM
%res = sext iN Val to iM
    // Require M >= N. Extend either by 0-s,
    // or by leftmost bit (sign bit).
```

## Converting between Floating Point Types

```
%res = float/double Val to float/double
// It is not really defined how rounding
// takes place. If Val does not fit,
// result is undefined.

%res = fpext float/double Val to float/double.
```

## Compound Types

LLVM has three types of compound types:

- Array types have form  $[ N \times T ]$ , where  $N$  is the size of the array, and  $T$  its type. Indexing does not check bounds by default. If you don't know the size, you can use  $N = 0$ .
- Structure types have form  $\{ T_1, T_2, \dots, T_n \}$ , where  $T_i$  are the field types, and the field names do not matter.
- Vector types have form  $\langle n \times T \rangle$ , where  $N$  is the size of vector. The difference with arrays is, that there exist instructions that operate directly on vector types.

## Accessing Elements of Compound Types

Elements of arrays and structs are accessed with `getelementptr`.

The `getelementptr` command is complicated because it handles **all forms** of pointer arithmetic in a single instruction.

It is important to note that `getelementptr` never accesses memory. It only handles pointer additions, and pointer type casts.

In addition, you will have to wash the *C/C++* conditioning out of your brain.

Repeat 10 times: A pointer to an array is not the same as pointer to its first element.

## Forms of Pointer Arithmetic

Assume that  $p$  has type  $T^*$ , i.e. pointer to  $T$ . The following operations are possible on  $p$ :

1. If  $p$  points to an array element, then  $p$  can be moved over  $n$  elements:  $q = p + n$ . The types of  $p$  and  $q$  are the same.
2. If  $T$  is a `struct`, then one can move into a field of  $T$ :  
 $q = \&(p \rightarrow f)$ . Pointer  $q$  has type  $F^*$ , where  $F$  is the type of field  $f$ .
3. If  $T$  is an array  $[n \times U]$ , then  $p$  can be advanced to the  $i$ -th element of the array. This operation cannot be expressed in  $C$ , unless one defines `array<U,n> = struct{ U val[n]; }` Then it means:  $q = \&(p \rightarrow \text{values} + i)$ . The type of  $q$  is  $U^*$ .

The first argument of `getelementptr` is the starting pointer, the second argument always does (1). If there are more arguments, they do (2) or (3).

## Accessing Elements

The general form of `getelementptr` is:

```
%q = getelementptr T* p, i1 Val1, i2 Val2, ..., in Valn
```

`T*` is the type of `p`, the `ii` must be integer types, and `Vali` are the indices.

Accessing elements:

```
struct xxxx { char f0[10]; double f1; };
```

```
xxxx p = { "pi equals", 3.1415 };
```

```
%1 = gep { [ 10 x i8 ], double }* p, i32 0, i32 0  
// Pointer to [ 10 x i8 ].
```

```
%2 = gep [ 10 x i8 ]* %1, i32 0, i32 4  
// Pointer to character 'q'.
```

```
%3 = gep { [ 10 x i8 ], double }* p, i32 0, i32 0, i32 4  
// Same as %2.
```

```
%4 = gep { [ 10 x i8 ], double }* p, i32 0, i32 1  
// Pointer to 3.1415.
```

## Casts between Pointers and Ints

```
%res = ptrtoint T* p to int  
      ; Casts p to int of length n.
```

```
%res = inttoptr in I to T*  
      ; Casts int of length n to ptr to T.
```

Don't use this for address calculations. The first conversion is used for pointer subtraction.

## Pointer Subtraction

```
X* p1;  
X* p2;  
size_t i = ( p2 - p1 );
```

In LLVM:

```
%X = type { double, double };  
  
%1 = ptrtoint %X* %p1 to i64  
%2 = ptrtoint %X* %p2 to i64  
%3 = isub i64 %1, %2  
%4 = sdiv exact i64 %3, i64 16  
; Assuming sizeof(X) = 16.
```

To me it seems inconsistent to have an specialized instruction for typed pointer addition (`getelementptr`) but no specialized instruction for pointer subtraction.

## Casting Between Pointer Types

Clang likes to cast between pointer types: (1) When a library function is called with a pointer, it is cast to pointer to u8. (2) Pointers to defined structs are cast to pointers to their definitions in order to load the struct as a single value type.

```
%res = bitcast T* p1 to i8*  
    // Casts T* to i8*.
```

```
%res2 = bitcast X* %res to { double, double } * ;
```

```
%res3 = load { double, double } * %res2  
    ; Maybe X is not Single Value?
```

Clang uses this when small structs are passed to a function, or returned by a function.

## Structs/Arrays in a Register

When a struct/array is in a register, its fields can be accessed as follows:

```
%res = extractvalue { F1, F2, ....., Fn } %v, i
```

Difference with `getelementptr` is that the struct/array is in a register, and there is no first argument that possibly skips the struct/array.

```
%res = insertvalue { F1, F2, ....., Fn } %v,  
                U %Newval, i
```

`%v` is the struct/array in which we want to replace, `%Newval` is the new value of the field that is being replaced, `i` is the relative position of the field `%res` that we want to replace.

## Calling a Function

Function names in  $C^{++}$  are **mangled**. This means that overloading is resolved by encoding type information into the function name.

```
%val = call T mangledname( T1 Val1, ..., Tn Valn ).
```

The function definition has the following form:

```
define T mangledname( T1 %Reg1, ..., Tn %Regn )  
(store Reg1 ... Regn in local variables.)  
....  
  
ret T Val  
ret void ; void has no Val.
```

## Passing and Returning Values

Simple parameters (first class types and small struct types) are passed directly.

Non-simple parameters are passed indirectly: The value is constructed in memory, and a pointer to it is passed to the function.

A simple return value (first class types and simple struct types) are returned by `ret`.

For a non-simple return value (the rest), a pointer is passed to the function, to which the function writes its result.