

LLVM Infrastructure

Letterio Galletta

Agenda

- LLVM architecture
- LLVM basic tools
- LLVM IR
 - Module declaration
 - Function declaration
 - Basic instruction

References

- [LLVM Language Reference Manual](#)
- [The Architecture of Open Source Applications: LLVM](#)
- [Mapping High Level Constructs to LLVM IR](#)
- [LLVM's getelementptr, by example](#)

LLVM



A collection of modular and reusable compiler and toolchain technologies

- The **LLVM Core** libraries and tools provide a source- and target-independent optimizer, along with code generation for many target CPUs
- [Clang](#) is a C/C++/Objective-C compiler. The [Clang Static Analyzer](#) and [clang-tidy](#) are tools that automatically find bugs in your code
- The [polly](#) project implements a suite of cache-locality optimizations as well as auto-parallelism and vectorization
- The [klee](#) project implements a "symbolic virtual machine"
- See the [LLVM webpage](#) for others

LLVM: Installation

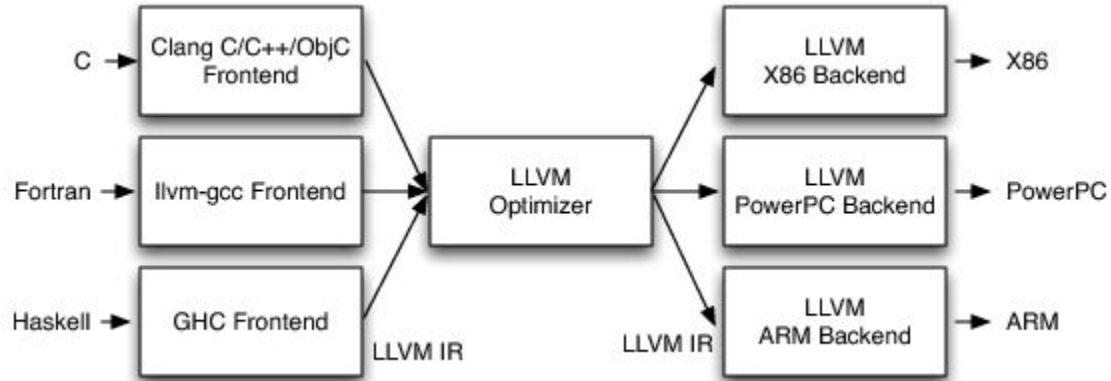
Source distribution and pre-compiled binaries at

<https://releases.llvm.org/download.html>

Many Linux distributions have it in their package system, e.g., debian, ubuntu, fedora

Note: course material tested with version 10

Architecture



- LLVM supports multiple frontends and multiple backends
- The main technology that enables this architecture is LLVM IR

Main features

LLVM is designed according to the principle that everything is a library:

- Every piece of code is not linked to a specific tool and it is reusable
- Each library concerns the generation, manipulation and analysis of program written in LLVM IR

LLVM IR

- It is the hearth of the project, usually every component of LLVM consumes LLVM IR and produces LLVM IR
 - The frontends of different language target this IR
 - The code optimizer runs of this IR to achieve better performances
 - Code generators make use of it to target specific architecture
- It aims to be at low level enough that high-level ideas can be cleanly mapped to it
- It aims to be at high-level enough that it is target-independent and can be compiled to different architecture, e.g., GPUs

LLVM IR

A Single-Static Assignment (SSA) with the following characteristics:

- Code is organized as three-address instructions
 - It is a sort of RISC-like virtual instruction set
 - Instructions mainly take some number of inputs from registers and produce a result in a different register
- It has a infinite number of registers
- It is strongly typed with a simple type system, e.g., **i32** for 32-bit integers
- It abstracts away some details of the machine: the calling convention is abstracted through **call** and **ret** instructions and explicit arguments

LLVM IR: forms

There are three equivalent forms:

- LLVM Assembly language
 - Text form saved on disk for humans to read
- LLVM Bitcode
 - Binary form saved on disk for programs to manipulate
- LLVM In-memory IR
 - Data structures used for analysis and optimization

An example of LLVM IR (1)

We consider a simple C program

See `add.c`

Compiles it with

```
$ clang -emit-llvm -c -S add.c
```

It generates a LLVM IR file in text format

See `add.ll`

An example of LLVM IR: observations (1)

- The result of compiling a compilation unit is a module that contains functions, global variables, external function prototypes, and symbol table entries
- Global variables start with the symbol `@` and are typed
- Functions are declared with the **define** keyword and we specified the return type and the list of formal parameters with their types
- Local identifiers start with the symbol `%` and usually represent registers
- We have to allocate the memory to store local variables explicitly

See [add.ll](#)

An example of LLVM IR: observations (2)

We have two categories of local variables:

- **Register allocated local variables:** These are the temporaries and allocated virtual registers, that are allocated to physical registers during the code generation phase
- **Stack allocated local variables:** These are created by allocating variables on the stack frame of a currently executing function through **alloca** instruction. It returns a pointer and an explicit use of **load** and **store** instructions is required to access and store the value

LLVM tools from the command line (1)

- **llvm-as** takes an LLVM IR in assembly form and converts it to bitcode

```
$ llvm-as add.ll -o add.bc
```

- **llvm-dis** takes a bitcode file and disassemble it

```
$ llvm-dis add.bc -o add1.ll
```

- **llvm-link** links two or more llvm bitcode files and output only one

```
$ clang -emit-llvm -c main.c
```

```
$ llvm-link main.bc add.bc -o output.bc
```

LLVM tools from the command line (2)

- **lli** directly executes program in LLVM IR bitcode using a JIT compiler or interpreter

```
$ lli output.bc
```

- **llc** compiles a LLVM IR program into assembly language for a specific architecture

```
$ llc output.bc           # generate an assembly file
```

```
$ llc -filetype=obj output.bc #generate an object file
```

```
$ clang output.o -o a.out    # invoke the linker to build the executable file
```

LLVM tools from the command line (3)

opt takes an input file and runs an analysis or an optimization on that file

```
$ opt [options] [input file]
```

Some useful analyses are

- **basicaa**: basic alias analysis
- **da**: dependence analysis

Some useful optimization pass

- **constprop**: simple constant propagation
- **licm**: loop invariant code motion

```
$ opt -print-memdeps -analyze deps.bc
```

LLVM IR at a glance

C programming language	LLVM IR
Scope: file, functions	Module, function declaration
Type: bool, char, int, struct{int, char}	i1, i8, i32, {i32, i8}
A statement with multiple expressions	A sequence of instructions each of which is a form of $x = y \text{ op } z$
Data-flow: a sequence of reads/writes on variables	<ol style="list-style-type: none">1. load the values of memory addresses (variables) to registers;2. compute the values in registers;3. store the values of registers to memory addresses;
Control-flow in a function: if, for, while, do while, switch-case,...	A set of basic blocks each of which ends with a conditional jump (or return)

Overview of LLVM IR

- Each assembly/bitcode file defines a **module**
- A module contains prototypes, global variables and function definition
- A function definition is made of a set of basic blocks
- Each basic block is made of a sequence of instructions

An example of a module

See `empty-module.ll`

```
; ModuleID = 'empty-module'
```

```
source_filename = "empty-module.c"
```

```
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
```

```
target triple = "x86_64-pc-linux-gnu"
```

- A target data layout specifies how data is to be laid out in memory
- Target triple describes the target host

See [documentation](#) for details of the available options

An example of function definition

See fun-decl.ll

```
; ModuleID = 'fun-decl.c'
```

```
source_filename = "fun-decl.c"
```

```
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
```

```
target triple = "x86_64-pc-linux-gnu"
```

```
define dso_local void @bar(i32 %0) #0 {
```

```
    %2 = alloca i32, align 4
```

```
    store i32 %0, i32* %2, align 4
```

```
    ret void
```

```
}
```

LLVM Instruction set

RISC-like architecture:

- Only 31 types of instructions exist
- They are three-address form: one or two operands, and one result
- Load/store architecture
 - Memory can be accessed via **load/store** instruction
 - Computational instructions operate on registers
- Infinite and typed virtual registers
 - It is possible to declare a new register any point (the backend maps virtual registers to physical ones)
 - A register is declared with a primitive type (boolean, integer, float, pointer)

Data representations

- Primitive types
- Constants
- Registers
- Variables
- Local variables, heap variables, global variables
- Load and store instructions
- Aggregated types

Primitive Types

Language independent primitive types with predefined sizes

- void: **void**
- bool: **i1**
- integers: **i[N]** where **N** is, e.g., **i8**, **i16**, **i32**, **i64**, ...
- floating-point types:
 - half** (16-bit floating point value)
 - float** (32-bit floating point value)
 - double** (64-bit floating point value)
- Pointer type is a form of **<type>*** (e.g. **i32***, **(i32*)***)

Constants

- Boolean (**i1**): **true** and **false**
- Integers: standard integers including negative numbers
- Floating point: decimal notation, exponential notation, or hexadecimal notation (IEEE754 Std.)
- Pointer: **null** is treated as a special value

Variables

- All addressable objects (“lvalues”) are explicitly allocated
- **Globals:** `[@][a-zA-Z$. _][a-zA-Z$. _0-9]*`
 - Each variable has a global scope symbol that points to the memory address of the object
- **Locals**
 - The `alloca` instruction allocates memory in the **stack frame**
 - Deallocated automatically if the function returns.
- **Heap variables**
 - No special instructions: use functions `malloc/free` as usual

Registers

- Identifier syntax
 - Named registers: `[%][a-zA-Z$. _][a-zA-Z$. _0-9]*`
 - Unnamed registers: `[%][0-9][0-9]*`
- A register has a function-level scope
 - Two registers in different functions may have the same identifier
- A register is assigned for a particular type and a value at its first (and the only) definition

An example of globals and registers

See `global-local.ll`

```
; ModuleID = 'global.c'
```

```
; ....
```

```
@foo = global i32 42
```

```
define dso_local void @bar(i32 %0) #0 {
```

```
    %2 = load i32, i32* @foo
```

```
    %3 = add i32 %2, 7
```

```
    ret void
```

```
}
```

Memory instructions

Memory access

- Load instructions read memory
- Store instructions write memory
- Atomic compare and exchange
- Atomic read/modify/write

Memory allocation

- Stack allocation (**alloca**)
- Calls to heap-allocation functions
 - malloc (no special instruction)
- Global variable declarations
 - Not really instructions, allocate memory
 - All globals are pointers to memory objects

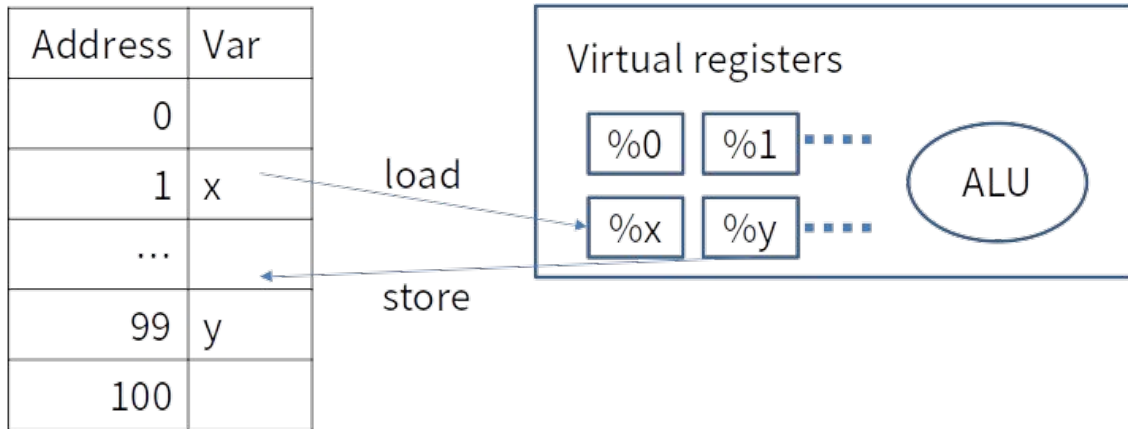
Load and store instructions

$\text{<result>=load <type>*, <ptr>}$

- **result**: the target register
- **type**: the type of the data (a pointer type)
- **ptr**: the register that has the address of the data

$\text{store <type> <value>, <type>* <ptr>}$

- **type**: the type of the value
- **value**: either a constant or a register that holds the value
- **ptr**: the register that has the address where the data should be stored



An example of local variables

See `local-vars.ll`

```
; Function Attrs: noinline nounwind optnone uwtable
```

```
define dso_local void @foo(i32 %0) #0 {
```

```
    %2 = alloca i32, align 4
```

```
    %3 = alloca i32, align 4
```

```
    store i32 %0, i32* %2, align 4
```

```
    %4 = load i32, i32* %2, align 4
```

```
    %5 = mul nsw i32 %4, 2
```

```
    store i32 %5, i32* %3, align 4
```

```
    ret void
```

```
}
```

Instructions for computations

- Arithmetic and binary operators
 - Two's complement arithmetic (**add**, **sub**, **multiply**, etc)
 - Bit-shifting and bit-masking
- Pointer arithmetic (**getelementptr** or “GEP”)
- Comparison instructions (**icmp**, **fcmp**)
 - Generates a boolean result

Arithmetic instructions

- Binary operations:
 - Add: **add**, **sub**, **fsub**
 - Multiplication: **mul**, **fmul**
 - Division: **udiv**, **sdiv**, **fdiv**
 - Remainder: **urem**, **srem**, **frem**
- Bitwise binary operations
 - shift operations: **shl**, **lshl**, **ashr**
 - logical operations: **and**, **or**, **xor**

See [documentation](#) for a complete list

Example of arithmetic instruction

`<res> = add [nuw][nsw] <iN> <op1>, <op2>`

- **nuw** (no unsigned wrap): if unsigned overflow occurs, the result value becomes a poison value (undefined)

add nuw i8 255, i8 1

- **nsw** (no signed wrap): if signed overflow occurs, the result value becomes a poison value

add nsw i8 127, i8 1

Example

See arithmetic.ll

; Function Attrs: noinline nounwind optnone uwtable

```
define dso_local void @foo(i32 %0) #0 {  
    %2 = alloca i32, align 4  
    %3 = alloca i32, align 4  
    store i32 %0, i32* %2, align 4  
    %4 = load i32, i32* %2, align 4  
    %5 = mul nsw i32 %4, 2  
    store i32 %5, i32* %3, align 4  
    %6 = load i32, i32* %2, align 4  
    %7 = shl i32 %6, 10  
    ret void  
}
```

Control-flow representation

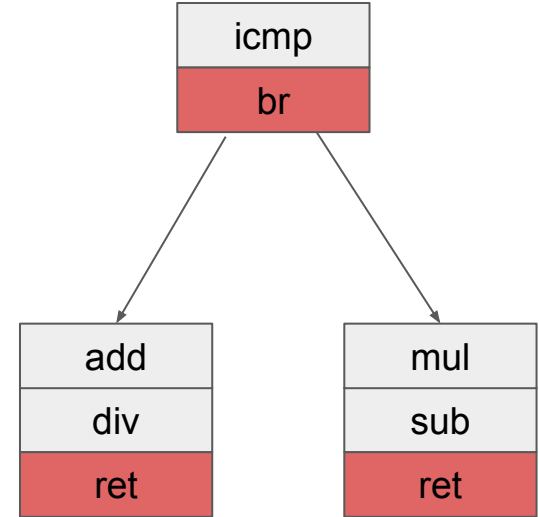
We express the control flow graph (CFG) of every function explicitly:

- A function has a set of basic blocks each of which is a sequence of instructions
- A function has **exactly one** entry basic block
- Every basic block is ended with **exactly one terminator** instruction which explicitly specifies its successor basic blocks if there exist

Terminator instructions: branches (conditional, unconditional), return, etc.

Control-flow instructions

- Terminator instructions which indicate where to jump next
 - It cannot occur inside a basic block
 - Conditional branch, unconditional branch, switch
 - return instruction
- Call instruction calls a function
 - It can occur inside a basic block



Label, return, and unconditional branch

- A label is located at the start of a basic block
 - Each basic block is addressed as the start label
 - A label `x` is referenced as register `%x` whose type is label
 - The label of the entry block of a function is “entry”
- Return `ret <type> <value> | ret void`
- Unconditional branch `br label <dest>`
 - At the end of a basic block, this instruction makes a transition to the basic block starting with label `<dest>`

`br label %entry`

Conditional Branch (1)

`<res> = icmp <cmp> <ty> <op1>, <op2>`

- Returns either `true` or `false` (`i1`) based on comparison of two variables (`op1` and `op2`) of the same type (`ty`)
- `cmp`: comparison option
 - `eq` (equal), `ne` (not equal), `ugt` (unsigned greater than),
 - `uge` (unsigned greater or equal), `ult` (unsigned less than),
 - For other see [documentation](#)

Conditional Branch (2)

br i1 <cond>, label <thenbb>, label <elsebb>

- Causes the current execution to transfer to the basic block <thenbb> if the value of <cond> is true; to the basic block <elsebb> otherwise
- Usually, we perform the comparison using **icmp** and then we jump to the correct block using **br**

Example of conditionals

See conditional.ll

```
define dso_local i32 @abs(i32 %0) #0 {
```

```
    %2 = alloca i32, align 4
```

```
    %3 = alloca i32, align 4
```

```
    store i32 %0, i32* %3, align 4
```

```
    %4 = load i32, i32* %3, align 4
```

```
    %5 = icmp slt i32 %4, 0
```

```
    br i1 %5, label %6, label %9
```

```
6:                                     ; preds = %1
```

```
    %7 = load i32, i32* %3, align 4
```

```
    %8 = sub nsw i32 0, %7
```

```
    store i32 %8, i32* %2, align 4
```

```
    br label %11
```

```
9:                                     ; preds = %1
```

```
    %10 = load i32, i32* %3, align 4
```

```
    store i32 %10, i32* %2, align 4
```

```
    br label %11
```

```
11:                                    ; preds = %9, %6
```

```
    %12 = load i32, i32* %2, align 4
```

```
    ret i32 %12
```

```
}
```

Phi instruction

$\langle \text{res} \rangle = \mathbf{phi} \langle t \rangle [\langle \text{val_0} \rangle, \langle \text{label_0} \rangle], [\langle \text{val_1} \rangle, \langle \text{label_1} \rangle], \dots$

- The Φ function of SSA: it manages control-flow merge
- Return a value val_i of type t such that the basic block executed right before the current one is of label_i

Example of control-flow merge

See `phi-instruction.ll`

```
define dso_local i32 @abs(i32 %0) #0 {
```

```
    %2 = alloca i32, align 4
```

```
    store i32 %0, i32* %2, align 4
```

```
    %3 = load i32, i32* %2, align 4
```

```
    %4 = icmp slt i32 %3, 0
```

```
    br i1 %4, label %5, label %8
```

```
5:                                     ; preds = %1
```

```
    %6 = load i32, i32* %2, align 4
```

```
    %7 = sub nsw i32 0, %6
```

```
    br label %10
```

```
8:                                     ; preds = %1
```

```
    %9 = load i32, i32* %2, align 4
```

```
    br label %10
```

```
10:                                    ; preds = %8, %5
```

```
    %11 = phi i32 [ %7, %5 ], [ %9, %8 ]
```

```
    ret i32 %11
```

```
}
```

Function call

$\langle \text{res} \rangle = \text{call } \langle t \rangle [\langle \text{fnty} \rangle^*] \langle \text{fnptrval} \rangle (\langle \text{fn args} \rangle)$

- t : the type of the call return value
- fnty : the signature of the pointer to the target function (optional)
- fnptrval : an LLVM value containing a pointer to a target function
- fn args : argument list whose types match the function signature

Example of function call

See `call.ll`

```
; Function Attrs: noinline nounwind optnone uwtable
```

```
define dso_local i32 @bar(i32 %0) #0 {  
    %2 = alloca i32, align 4  
    store i32 %0, i32* %2, align 4  
    %3 = load i32, i32* %2, align 4  
    %4 = call i32 @abs(i32 %3) #2  
    %5 = mul nsw i32 2, %4  
    ret i32 %5  
}
```

Aggregate Types and Function Type

- Arrays: [`<# of elements>` x `<type>`]
 - Single dimensional array: `[40 x i32]`, `[4 x i8]`
 - Multi dimensional array: `[3 x [4 x i8]]`, `[12 x [10 x float]]`
- Structures: `type {<a list of types>}`
 - `type{ i32, i32, i32 }`, `type{ i8, i32 }`
- Functions: `<return type>` (a list of parameter types)
 - `i32 (i32)`, `float (i16, i32*)*`

Example of arrays and structs

See array-struct.ll

```
%struct.person = type { i32, [25 x i8], i32 }
```

```
@numbers = dso_local global [10 x i32] zeroinitializer, align 16
```

```
@.str = private unnamed_addr constant [25 x i8] c"This is a literal string\00", align 1
```

```
@bob = dso_local global %struct.person { i32 1, [25 x i8] c"Bob  
Smith\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00", i32 20 }, align 4
```

```
@matrix = common dso_local global [4 x [4 x i32]] zeroinitializer, align 16
```

Getelementptr instruction

It is used to get the address of a subelement of an aggregate data structure, e.g., arrays, structs. It performs address calculation only and does not access memory:

$\langle res \rangle = \text{getelementptr} [\text{inbounds}] \langle ty \rangle, \langle ty \rangle^* \langle ptrval \rangle \{, [\text{inrange}] \langle ty' \rangle \langle idx \rangle \}^*$

- **res**: the target register
- **ty**: the type used as the basis for the calculations
- **ptrval**: the base address to start from
- **ty'**: the type of index
- **idx**: (a pointer to) the index value

Example of pointer access

See gep.c

```
int *ptr;
```

```
int get_elem_ptr(int i) {  
    return ptr[i];  
}
```

We calculate the address of an i32 (the first argument), using:

- %3 as our base address (the second argument)
- %5 as our index (the third argument)

...and we store our calculated address into %6

See gep.ll

```
@ptr = dso_local global i32* @inttoptr (i64 1 to i32*), align 8
```

```
define dso_local i32 @get_elem_ptr(i32 %0) #0 {  
    %2 = alloca i32, align 4  
    store i32 %0, i32* %2, align 4  
    %3 = load i32*, i32** @ptr, align 8  
    %4 = load i32, i32* %2, align 4  
    %5 = sext i32 %4 to i64  
    %6 = getelementptr inbounds i32, i32* %3, i64 %5  
    %7 = load i32, i32* %6, align 4  
    ret i32 %7  
}
```

Example of array access

See gep.c

```
int array[3] = {1, 2, 3};
```

```
int get_elem_ar(int i){  
    return array[i];  
}
```

We calculate the address of an $[3 \times i32]$ (the first argument), using:

- @array as our base address (the second argument)
- **Two** indexes
- i64 0 (the third argument) to get the piece the pointer;
- %4 (the four argument) to index the array

See gep.ll

```
@array = dso_local global [3 x i32] [i32 1, i32 2, i32 3], align 4
```

```
; Function Attrs: noinline nounwind optnone uwtable
```

```
define dso_local i32 @get_elem_ar(i32 %0) #0 {
```

```
    %2 = alloca i32, align 4
```

```
    store i32 %0, i32* %2, align 4
```

```
    %3 = load i32, i32* %2, align 4
```

```
    %4 = sext i32 %3 to i64
```

```
    %5 = getelementptr inbounds [3 x i32], [3 x i32]* @array, i64 0, i64 %4
```

```
    %6 = load i32, i32* %5, align 4
```

```
    ret i32 %6
```

```
}
```


Example of matrix access

See `gep.c`

```
int matrix[3][3] = {0};

int get_elem_m(size_t i, size_t j){
    return matrix[i][j];
}
```

See `gep.ll` (no optimization)

```
define dso_local i32 @get_elem_m(i64 %0, i64 %1) #0 {
    %3 = alloca i64, align 8
    %4 = alloca i64, align 8
    store i64 %0, i64* %3, align 8
    store i64 %1, i64* %4, align 8
    %5 = load i64, i64* %3, align 8

    %6 = getelementptr inbounds [3 x [3 x i32]], [3 x [3 x i32]]* @matrix, i64
0, i64 %5

    %7 = load i64, i64* %4, align 8
    %8 = getelementptr inbounds [3 x i32], [3 x i32]* %6, i64 0, i64 %7
    %9 = load i32, i32* %8, align 4

    ret i32 %9
}
```

Example of matrix access

See `gep.c`

```
int matrix[3][3] = {0};

int get_elem_m(size_t i, size_t j){
    return matrix[i][j];
}
```

See `gep.ll` (optimization -O1)

```
; Function Attrs: norecurse nounwind readonly uwtable

define dso_local i32 @get_elem_m(i64 %0, i64 %1) local_unnamed_addr #0 {

    %3 = getelementptr inbounds [3 x [3 x i32]], [3 x [3 x i32]]* @matrix, i64
0, i64 %0, i64 %1

    %4 = load i32, i32* %3, align 4

    ret i32 %4

}
```

Example of struct field access

See gep.c

```
struct person {  
    char name[20];  
    unsigned int age;  
}  
p;  
  
int get_age() {  
    return p.age;  
}
```

See gep.ll

```
%struct.person = type { [20 x i8], i32 }  
  
@p = common dso_local local_unnamed_addr @global %struct.person  
zeroinitializer, align 4  
  
define dso_local i32 @get_age() local_unnamed_addr #0 {  
    %1 = load i32, i32* @getelementptr inbounds (%struct.person,  
    %struct.person* @p, i64 0, i32 1), align 4  
  
    ret i32 %1  
}
```

Other instructions

- **switch** instruction to transfer control flow to one of many possible blocks
- Integer conversions: **trunc**, **zext**, **sext**, **bitcast**, etc.
- Vector data type (SIMD style)
- Exception handling
- Atomic instructions
- Features for OO languages

See [documentation](#)

Conclusion

- LLVM architecture
- LLVM basic tools
- LLVM IR
 - Module declaration
 - Function declaration
 - Basic instruction

References

- [LLVM Language Reference Manual](#)
- [The Architecture of Open Source Applications: LLVM](#)
- [Mapping High Level Constructs to LLVM IR](#)
- [LLVM's getelementptr, by example](#)