

# Tutorial for LLVM Intermediate Representation

Prof. Moonzoo Kim  
CS Dept., KAIST

# Motivation for Learning LLVM Low-level Language (i.e., Handling Intermediate Representation)

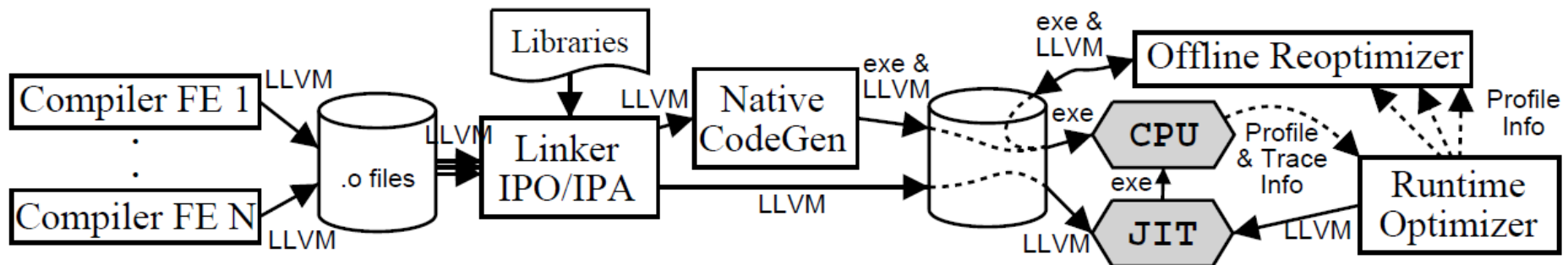
- Biologists know how to analyze laboratory mice. In addition, they know how to modify the mice by applying new medicine or artificial organ
- Mechanical engineers know how to analyze and modify mechanical products using CAD tools.
- Software engineers also have to know how to analyze and modify software code which is far more complex than any engineering product. Thus, software analysis/modification requires automated analysis tools.
  - Using source level analysis framework (e.g., Clang, C Intermediate Language (CIL), EDG parser)
  - Using low-level intermediate representation (IR) analysis framework (e.g., LLVM IR)

# LLVM is Professional Compiler

- Clang, the LLVM C/C++ front-end supports the full-features of C/C++ and compatible with GCC
- The executable compiled by Clang/LLVM is as fast as the executable by GCC

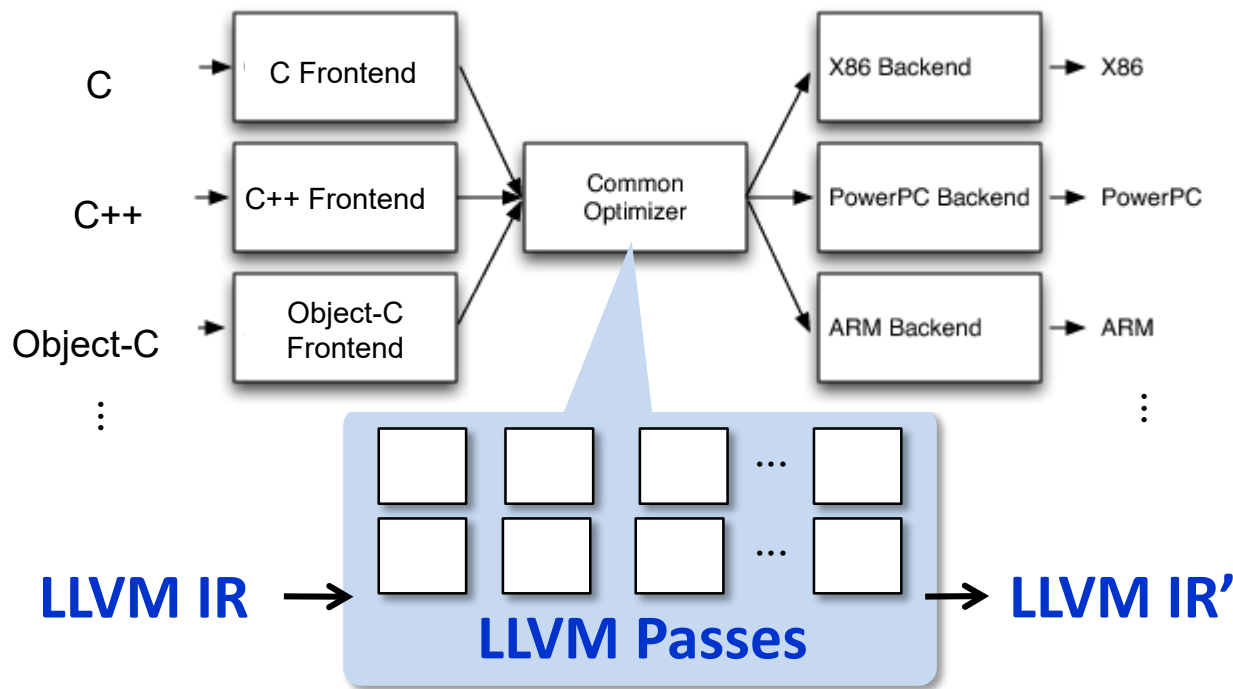
# LLVM Compiler Infrastructure (1/2)

- Clang, the LLVM C/C++ front-end supports the full-features of C/C++ and compatible with GCC
- The executable compiled by Clang/LLVM is as fast as the executable by GCC



# LLVM Compiler Infrastructure (2/2)

- A collection of modular compilers and analyzers written in C++ with STL.
- LLVM provides 108+ Passes <http://llvm.org/docs/Passes.html>
  - Analyzers (41): alias analysis, call graph constructions, dependence analysis, etc.
  - Transformers (57): dead code elimination, function inlining, constant propagation, loop unrolling, etc.
  - Utilities (10): CFG viewer, basic block extractor, etc.



# LLVM IR as Analysis Target

- The LLVM IR of a program is a *better target for analysis and engineering* than the program source code.
  - Language-independent
    - Able to represent C/C++/Object-C programs
  - Simple
    - register machine
      - Infinite set of typed virtual registers
      - 3-address form instruction
      - Only 31 instruction opcodes
    - static single assignment (SSA)
    - composed as basic blocks
  - Informative
    - typed language
    - control-flow
- LLVM IR is also called as LLVM language, assembly, bitcode, bytecode, code representation

# LLVM IR At a Glance

## *C program language*

## *LLVM IR*

- |  |  |
|--|--|
| • Scope: <i>file, function</i>   | <i>module, function</i>  |
| • Type: <i>bool, char, int, struct{int, char}</i>                          | <i>i1, i8, i32, {i32, i8}</i>  |
| • A statement with multiple expressions                                    | A sequence of instructions each of which is in a form of “ <i>x = y op z</i> ”.  |
| • Data-flow:<br>a sequence of reads/writes on variables                    | <ol style="list-style-type: none"><li>1. load the values of memory addresses (variables) to registers;</li><li>2. compute the values in registers;</li><li>3. store the values of registers to memory addresses</li></ol> <p>* each register must be assigned exactly once (SSA)</p> |
| • Control-flow in a function:<br>if, for, while, do while, switch-case,... | A set of basic blocks each of which ends with a conditional jump (or return)   |

# Example

*simple.c*

```
1  #include <stdio.h>
2  int x, y ;
3
4  int main() {
5      int t ;
6      scanf("%d %d",&x,&y);
7      t = x - y ;
8      if (t > 0)
9          printf("x > y") ;
10     return 0 ;
11 }
```

```
$ clang -S -emit-llvm
-fno-discard-value-names simple.c
```

*simple.ll* (simplified)

```
...
2  6 @x = common global i32 0, align 4
   7 @y = common global i32 0, align 4
...
4  11 define i32 @main() #0 {
   12     entry:
...
5  14 %t = alloca i32, align 4
...
6  16 %call = call i32 @__isoc99_scanf(...i32* @x,i32* @y)
...
7  17 %0 = load i32* @x, align 4
   18 %1 = load i32* @y, align 4
   19 %sub = sub nsw i32 %0, %1
   20 store i32 %sub, i32* %t, align 4
...
8  21 %2 = load i32* %t, align 4
   22 %cmp = icmp sgt i32 %2, 0
   23 br i1 %cmp, label %if.then,
        label %if.end
...
9  24 if.then:
   25     %call1 = call i32 @printf(...)
   26     br label %if.end
...
10 27 if.end:
    28     ret i32 0
```



# Contents

- LLVM IR Instruction
  - architecture, static single assignment
- Data representation ([page 13](#))
  - types, constants, registers, variables
  - load/store instructions, cast instructions
  - computational instructions
- Control representation ([page 32](#))
  - control flow (basic block)
  - control instructions

\* *LLVM Language Reference Manual* <http://llvm.org/docs/LangRef.html>

\* *Mapping High-Level Constructs to LLVM IR*

<http://llvm.lyngvig.org/Articles/Mapping-High-Level-Constructs-to-LLVM-IR>

# LLVM IR Architecture

- RISC-like instruction set
  - Only 31 op-codes (types of instructions) exist
  - Most instructions (e.g. computational instructions) are in 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, int, float, pointer)

# Static Single Assignment (1/2)

- In SSA, each variable is assigned exactly once, and every variable is defined before its uses.
- Conversion
  - For each definition, create a new version of the target variable (left-hand side) and replace the target variable with the new variable.
  - For each use, replace the original referred variable with the versioned variable reaching the use point.

```
1  x = y + x ;
2  y = x + y ;
3  if (y > 0)
4      x = y ;
5  else
6      x = y + 1 ;
```



```
11 x1 = y0 + x0 ;
12 y1 = x1 + y0 ;
13 if (y1 > 0)
14     x2 = y1 ;
15 else
16     x3 = y1 + 1 ;
```

# Static Single Assignment (2/2)

- Use  $\phi$  function if two versions of a variable are reaching one use point at a joining basic block
  - $\phi(x_1, x_2)$  returns either  $x_1$  or  $x_2$  depending on which block was executed

```
1  x = y + x ;
2  y = x + y ;
3  if (y > 0)
4      x = y ;
5  else
6      x = y + 1 ;
7  y = x - y ;
```



```
11 x1 = y0 + x0 ;
12 y1 = x1 + y0 ;
13 if (y1 > 0)
14     x2 = y1 ;
15 else
16     x3 = y1 + 1 ;
17 x4 =  $\phi(x2, x3)$  ;
18 y2 = x4 - y1 ;
```

# Data Representations

- Primitive types
- Constants
- Registers (virtual 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 **1** to  $2^{23}-1$   
e.g. **i8**, **i16**, **i32**, **i1942652**
  - 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**
- Integer: standard integers including negative numbers
- Floating point: decimal notation, exponential notation, or hexadecimal notation (IEEE754 Std.)
- Pointer: **null** is treated as a special value

# 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



# Variables

- In LLVM, all addressable objects (“lvalues”) are explicitly allocated.
- Global variables
  - Each variable has a global scope symbol that points to the memory address of the object
  - Variable identifier: `[@] [a-zA-Z$. _] [a-zA-Z$. _0-9] *`
- Local variables
  - The **alloca** instruction allocates memory in the stack frame.
  - Deallocated automatically if the function returns.
- Heap variables
  - The **malloc** function call allocates memory on the heap.
  - The **free** function call frees the memory allocated by **malloc**.

# Load and Store Instructions

- Load

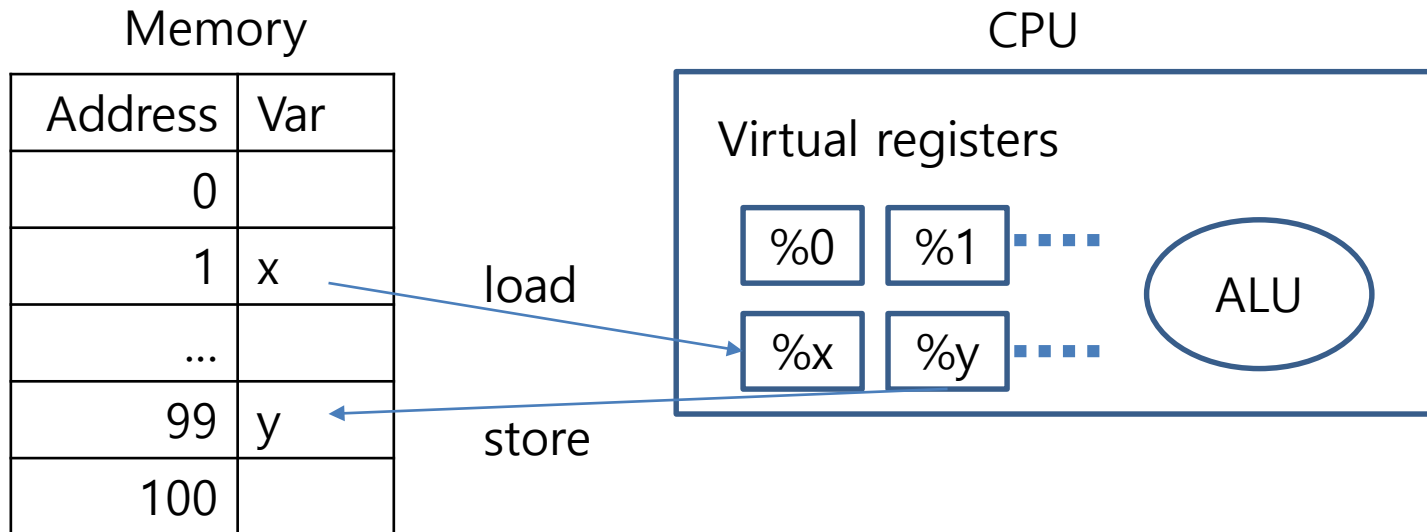
`<result>=load <type>* <ptr>`

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

- Store

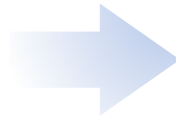
`store <type> <value>, <type>* <ptr>`

- 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



# Variable Example

```
1 #include <stdlib.h>
2
3 int g = 0 ;
4
5 int main() {
6     int t = 0;
7     int * p;
8     p=malloc(sizeof(int));
9     free(p);
10 }
```



```
1 @g = global i32 0, align 4
...
8 define i32 @main() #0 {
...
10 %t = alloca i32, align 4
11 store i32 0, i32* %t, align 4
12 %p = alloca i32*, align 8
13 %call = call noalias i8*
    @malloc(i64 4) #2
14 %0 = bitcast i8* %call to i32*
15 store i32* %0, i32** %p,
    align 8
16 %1 = load i32** %p, align 8
...

```

# Aggregate Types and Function Type

- Array: `[<# of elements> x <type>]`
  - Single dimensional array ex: `[40 x i32]`, `[4 x i8]`
  - Multi dimensional array ex: `[3 x [4 x i8]]`, `[12 x [10 x float]]`
- Structure: `type {<a list of types>}`
  - E.g. `type{ i32, i32, i32 }`, `type{ i8, i32 }`
- Function: `<return type> (a list of parameter types)`
  - E.g. `i32 (i32)`, `float (i16, i32*)`

# Getelementptr Instruction

- A memory in an aggregate type variable can be accessed by **load/store** instruction and **getelementptr** instruction that obtains the pointer to the element.

- Syntax:

**<res> = getelementptr <pty>\* <ptrval>{,<t> <idx>}\***

- res: the target register
- pty: the register that defines the aggregate type
- ptrval: the register that points to the data variable
- t: the type of index
- idx: the index value

# Aggregate Type Example 1 (1/2)

```
1 struct pair {  
2     int first;  
3     int second;  
4 };
```

```
5 int main() {  
6     int arr[10];  
7     struct pair a;
```

```
8     a.first = arr[1];
```

...

```
11 %struct.pair = type{ i32, i32 }
```

```
12 define i32 @main() {  
13     entry:
```

```
14     %arr = alloca [10 x i32]
```

```
15     %a = alloca %struct.pair
```

```
16     %arrayidx = getelementptr  
                    [10 x i32]* %arr, i32 0, i64 1
```

```
17     %0 = load i32* %arrayidx
```

```
18     %first = getelementptr  
                %struct.pair* %a, i32 0, i32 0
```

```
19     store i32 %0, i32* %first
```

# Aggregate Type Example 1 (2/2)

```
%first=getelementptr %struct.pair* %a, i32_0, i32 0
```

1. The first operand of `getelementptr` (e.g., `%a`) is a pointer to a data structure.
2. An element of an aggregate data structure must be accessed by `getelementptr` with a pointer (e.g., `%a`) and an offset index (`a.first == (&a)[0].first`, and/or `(&a)->first == (&a)[0].first`)

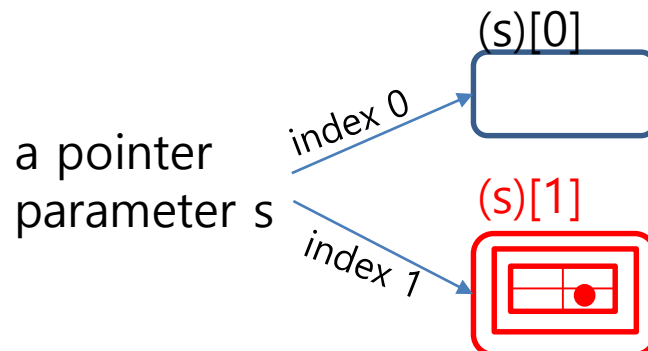
```
struct point {  
    int x;  
    struct point* p;};  
  
int main() {  
    struct point s, s1;  
    s.p=&s1; // (&s)[0].p=&s1;  
    (s.p)->x = 0;  
}
```

```
%struct.point = type { i32, %struct.point* }  
define i32 @main() #0 {  
entry:  
    %s = alloca %struct.point, align 8  
    %s1 = alloca %struct.point, align 8  
    // s.p=&s1;  
    %p = getelementptr %struct.point* %s, i32 0, i32 1  
    store %struct.point* %s1, %struct.point** %p, align 8  
    // (s.p)->x = 0  
    %p1 = getelementptr %struct.point* %s, i32 0, i32 1  
    %0 = load %struct.point** %p1, align 8  
    %x = getelementptr %struct.point* %0, i32 0, i32 0  
    store i32 0, i32* %x, align 4  
  
    ret i32 0  
}
```

# Aggregate Type Example 2

```
1 struct RT {  
2     char A;  
3     int B[10][20];  
4     char C;  
5 };  
6 struct ST {  
7     int X;  
8     double Y;  
9     struct RT Z;  
10 };  
11  
12 int *foo(struct ST *s) {  
13     return &s[1].Z.B[5][13];  
14 }
```

```
5 %struct.RT = type { i8, [10 x [20 x i32]  
    ], i8 }  
6 %struct.ST = type { i32, double, %struct  
    .RT }  
7  
8 define i32* @foo(%struct.ST* %s)  
    nounwind uwtable readnone optsize  
    ssp {  
9 entry:  
10     %arrayidx = getelementptr inbounds  
        %struct.ST* %s, i64 1, i32 2,  
        i32 1, i64 5,  
        i64 13  
11     ret i32* %arrayidx  
12 }
```





# Aggregate Type Example 3

```
%MyVar = global { [10 x i32] }  
%idx1=getelementptr {[10xi32]}, {[10xi32]}* %MyVar, i64 0, i32 0, i64 1  
%idx2=getelementptr {[10xi32]}, {[10xi32]}* %MyVar, i64 1
```

- `idx1` computes the address of the 2<sup>nd</sup> integer in the array that is in the structure in `%MyVar`, that is `MyVar+4`.
  - The type of `idx1` is `i32*`.
- `idx2` computes the address of the next structure after `%MyVar`. The value of `idx2` is `MyVar + 40` because it indexes past the ten 4-byte integers in `MyVar`.
  - The type of `idx2` is `{ [10 x i32] }*`

# Aggregate Type Example 4

```
%MyVar = global { [10 x i32] }  
%idx1=getelementptr {[10xi32]}, {[10xi32]}* %MyVar, i64 1, i32 0, i64 0  
%idx2=getelementptr {[10xi32]}, {[10xi32]}* %MyVar, i64 1
```

- The value of `%idx1` is `%MyVar+40`
  - its type is `i32*`
- The value of `%idx2` is also `%MyVar+40`
  - but its type is `{ [10 x i32] }*`.

# Integer Conversion (1/2)

- Truncate

- Syntax: `<res> = trunc <iN1> <value> to <iN2>`  
where `iN1` and `iN2` are of integer type, and `N1 > N2`

- Examples

- `%X = trunc i32 257 to i8 ; %X becomes i8:1`
    - `%Y = trunc i32 123 to i1 ; %Y becomes i1:true`
    - `%Z = trunc i32 122 to i1 ; %Z becomes i1:false`

# Integer Conversion (2/2)

- Zero extension

- `<res> = zext <iN1> <value> to <iN2>` where `iN1` and `iN2` are of integer type, and `N1 < N2`
- Fill the remaining bits with zero
- Examples
  - `%X = zext i32 257 to i64 ; %X becomes i64:257`
  - `%Y = zext i1 true to i32 ; %Y becomes i32:1`

- Sign extension

- `<res> = sext <iN1> <value> to <iN2>` where `iN1` and `iN2` are of integer type, and `N1 < N2`
- Fill the remaining bits with the sign bit (the highest order bit) of `value`
- Examples
  - `%X = sext i8 -1 to i16 ; %X becomes i16:65535`
  - `%Y = sext i1 true to i32 ; %Y becomes i32:232-1`

# Other Conversions

- Float-to-float
  - `fptrunc .. to, fpext .. to`
- Float-to-integer (vice versa)
  - `fptoui .. to, fptosi .. to, uitofp .. to, sitofp .. to`
- Pointer-to-integer
  - `ptrtoint .. to, inttoptr .. to`
- Bitcast
  - `<res> = bitcast <t1> <value> to <t2>`  
where `t1` and `t2` should be different types and have the same size
  - bitcast does not change any bit (i.e., no-op cast)
  - `%z = bitcast <2 x int> %v to i64`
  - `%z = bitcast <2 x i32*> %v to <2 x i64*>`

# Computational Instructions

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

# Add Instruction

- `<res> = add [nuw][nsw] <iN> <op1>, <op2>`
  - nuw (no unsigned wrap): if unsigned overflow occurs, the result value becomes a **poison value (undefined)**
    - E.g., `add nuw i8 255, i8 1`
  - nsw (no signed wrap): if signed overflow occurs, the result value becomes a poison value
    - E.g., `add nsw i8 127, i8 1`

# Control Representation

- The LLVM front-end constructs the control flow graph (CFG) of every function explicitly in LLVM IR
  - 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.
- Due to its simple control flow structure, it is convenient to analyze, transform the target program in LLVM IR




# 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>`
  - E.g: `br label %entry`

# Conditional Branch

- `<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),  
`ule` (unsigned less or equal), `sgt` (signed greater than),  
`sge` (signed greater or equal), `slt` (signed less than), `sle` (signed less or equal)
- `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.

- Example:

<pre>1  if (x &gt; y) 2      return 1 ; 3  return 0 ;</pre>		<pre>11 %0 = load i32*, %x 12 %1 = load i32*, %y 13 %cmp = icmp sgt i32 %0, %1 14 br i1 %cmp, label %if.then, label %if.end  15 if.then:     ...</pre>
---	---	--

# Switch

- **switch** <iN> <value>, label <defaultdest>  
[<iN> <val>, label <dest> ...]
- Transfer control flow to one of many possible destinations
- If the value is found (*val*), control flow is transferred to the corresponding destination (*dest*); or to the default destination (*defaultdest*)
- Examples:

```
1  switch(x) {  
2      case 1:  
3          break ;  
4      case 2:  
5          break ;  
6      default:  
7          break ;  
8  }
```



```
11  %0 = load i32*, %x  
12  switch i32 %0, label %sw.default [  
13      i32 1, label %sw.bb  
14      i32 2, label %sw.bb1]  
15  sw.bb:  
16      br label %sw.epilog  
17  sw.bb1:  
18      br label %sw.epilog  
19  sw.default:  
20      br label %sw.epilog  
21  sw.epilog:  
    ...
```

# PHI ( $\Phi$ ) instruction

- `<res> = phi <t> [ <val_0>, <label_0>],`  
`[ <val_1>, <label_1>], ...`
  - Return a value `val_i` of type `t` such that the basic block executed right before the current one is of `label_i`

- Example

```
1  y = (x > 0) ? x : 0 ;
```

```

11 %0 = load i32* %x
12 %c = icmp sgt i32 %0, 0
13 br i1 %c, label %c.t, %c.f

14 c.t:
15 %1 = load i32* %x
16 br label %c.end

17 c.f:
18 br label %c.end

19 c.end:
20 %res = phi i32 [%1, %c.t], [0, %c.f]
21 store i32 %res, i32* %y

```

# Select

- `<result> = select <selty> <cond>, <ty> <val1>, <ty> <val2> ;`
  - `<selty>` is either `i1` or `{<N x i1>}`
  - Ex> `%X = select i1 true, i8 17, i8 42 ;` yields `i8:17`
- The ‘select’ instruction is used to choose one value based on a condition, without IR-level branching.
  - If `<cond> == 1`, the instruction returns the first value argument; the second value argument otherwise
  - 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.

# Function Call

- `<res> = call <t> [<fnty>*] <fnptrval>(<fn args>)`
  - `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
- Examples:

```
1 printf("%d", abs(x));
```



```
11 @.str = constant [3 x i8] c"%d\00"
12 %0 = load i32* %x
13 %abs_x = call i32 @abs(i32 %0)

14 %printf = call i32 (i8*, ...) *
    @printf(i8*
        getelementptr ([3 x i8]* @.str,
            i32 0, i32 0),
            i32 %abs_x)
```

# Unaddressed Issues

- Many options/attributes of instructions
- Vector data type (SIMD style)
- Exception handling
- Object-oriented programming specific features
- Concurrency issues
  - Memory model, synchronization, atomic instructions

\* *<http://llvm.org/docs/LangRef.html>*