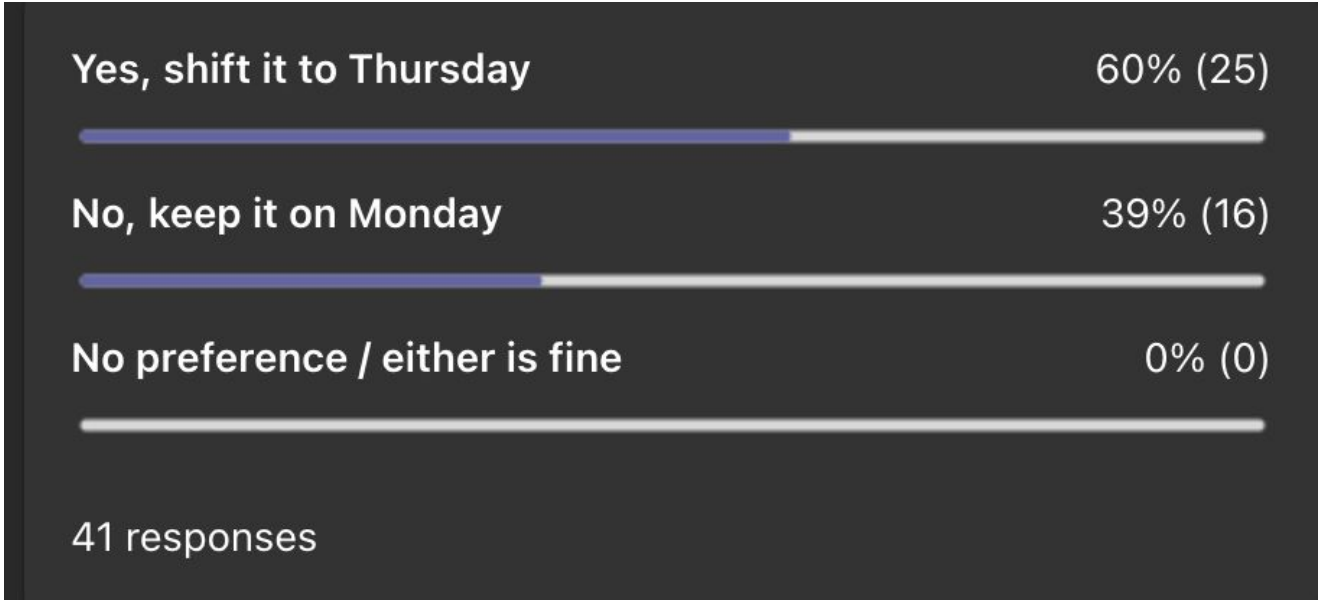
L9: LLVM IR continue..

Software Analysis

IIT Guwahati · August 2026



From next week, classes will be held on Tuesday, Wednesday and Thursday

C Program to LLVM IR

```c
void swap(char **p, char **q){
  char* t = *p;
  *p = *q;
  *q = t;
}
int main(){
  char a1;
  char *a;
  char b1;
  char *b;
  a = &a1;
  b = &b1;
  swap(&a,&b);
}
```

swap.c

Compile

```
define void @swap(ptr %p, ptr %q) #0 {
entry:
  %0 = load ptr, ptr %p, align 8
  %1 = load ptr, ptr %q, align 8
  store ptr %1, ptr %p, align 8
  store ptr %0, ptr %q, align 8
  ret void
}

define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1
  %a = alloca ptr, align 8
  %b1 = alloca i8, align 1
  %b = alloca ptr, align 8
  store ptr %a1, ptr %a, align 8
  store ptr %b1, ptr %b, align 8
  call void @swap(ptr %a, ptr %b)
  ret i32 0
}
```

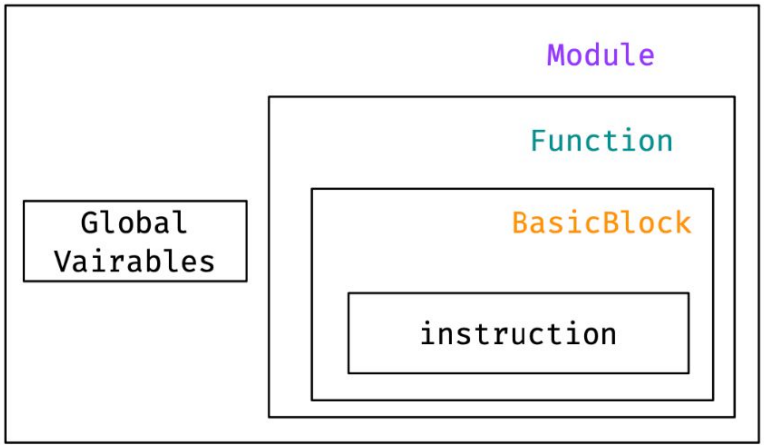
Function

BasicBlock

Instruction

swap.ll

C Program to LLVM IR



LLVM-IR Scopes

Module contains Functions and Global Variables
- Whole module is the unit of translation, analysis and optimization.

Function contains BasicBlocks and Arguments, which correspond to functions.

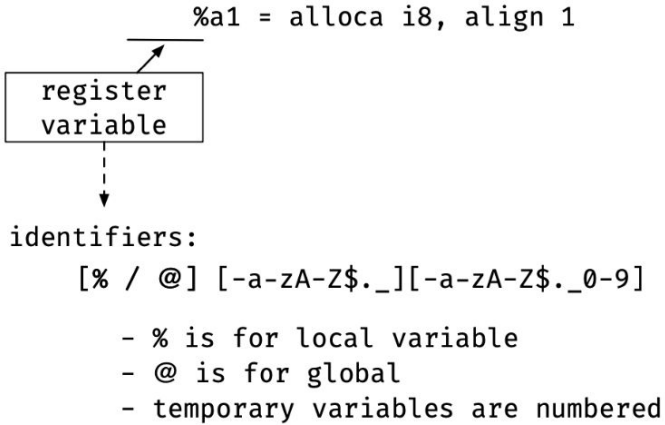
BasicBlock contains list of instructions.
- Each block is contiguous chunk of instructions

Instruction is opcode + vector of operands in '3-address' style
- All operands have types
- Instruction result is typed

LLVM IR

```
int main(){
  char a1;
  char *a;
  char b1;
  char *b;
  a = &a1;
  b = &b1;
  swap(&a,&b);
}
```

```
define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1
  %a = alloca ptr, align 8
  %b1 = alloca i8, align 1
  %b = alloca ptr, align 8
  store ptr %a1, ptr %a, align 8
  store ptr %b1, ptr %b, align 8
  call void @swap(ptr %a, ptr %b)
  ret i32 0
}
```

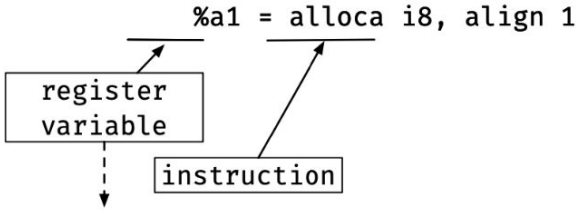


https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM IR

```
int main(){
  char a1;
  char *a;
  char b1;
  char *b;
  a = &a1;
  b = &b1;
  swap(&a,&b);
}
```

```
define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1
  %a = alloca ptr, align 8
  %b1 = alloca i8, align 1
  %b = alloca ptr, align 8
  store ptr %a1, ptr %a, align 8
  store ptr %b1, ptr %b, align 8
  call void @swap(ptr %a, ptr %b)
  ret i32 0
}
```



identifiers:

[% / @] [-a-zA-Z$._][-a-zA-Z$._0-9]

- % is for local variable
- @ is for global
- temporary variables are numbered

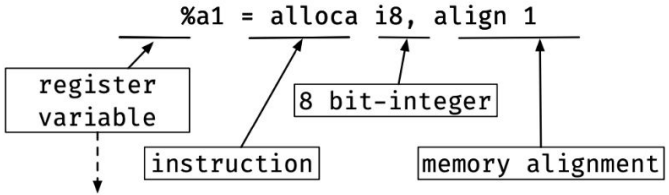
alloca: instruction allocates i8 (sizeof) bytes of memory on runtime stack

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM IR

```
int main(){
  char a1;
  char *a;
  char b1;
  char *b;
  a = &a1;
  b = &b1;
  swap(&a,&b);
}
```

```
define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1
  %a = alloca ptr, align 8
  %b1 = alloca i8, align 1
  %b = alloca ptr, align 8
  store ptr %a1, ptr %a, align 8
  store ptr %b1, ptr %b, align 8
  call void @swap(ptr %a, ptr %b)
  ret i32 0
}
```



identifiers:

[% / @] [-a-zA-Z$._][-a-zA-Z$._0-9]

- % is for local variable
- @ is for global
- temporary variables are numbered

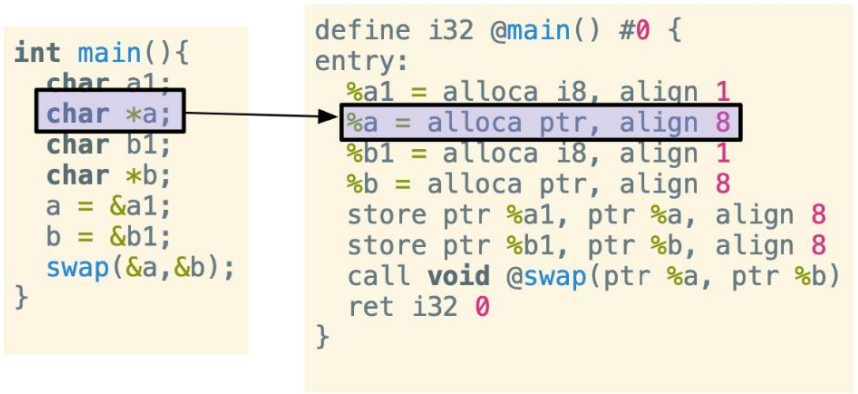
alloca: instruction allocates i8 (sizeof) bytes of memory on runtime stack

align: indicates the memory operation should be aligned to 1 byte

- `align 8` specifies that the allocated memory should be aligned on an 8-byte boundary. For example, its address could be 0x0008, 0x0010, 0x0018, 0x0020, and so on, but not 0x0001, 0x0002, or any other value that isn't a multiple of 8.

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM IR




```
%a = alloca ptr, align 8
```

↓

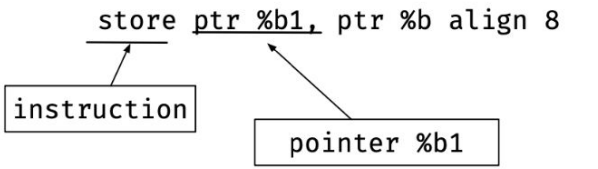
allocate 8-bit integer pointer for a

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM IR

```
int main(){
  char a1;
  char *a;
  char b1;
  char *b;
  a = &a1;
  b = &b1;
  swap(&a,&b);
}
```

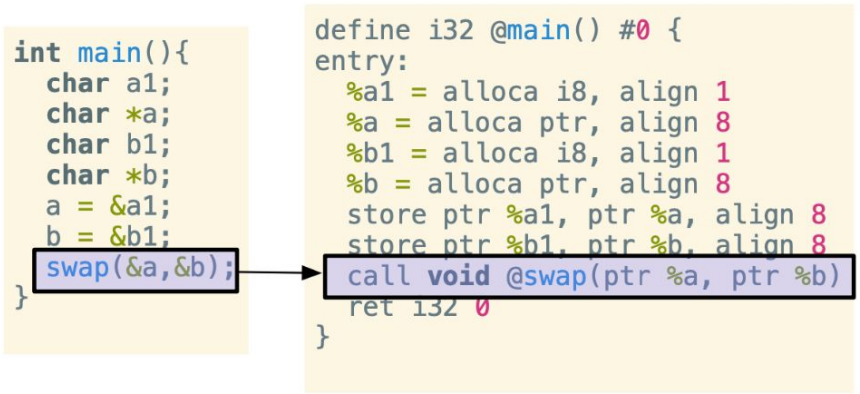
```
define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1
  %a = alloca ptr, align 8
  %b1 = alloca i8, align 1
  %b = alloca ptr, align 8
  store ptr %a1, ptr %a, align 8
  store ptr %b1, ptr %b, align 8
  call void @swap(ptr %a, ptr %b)
  ret i32 0
}
```

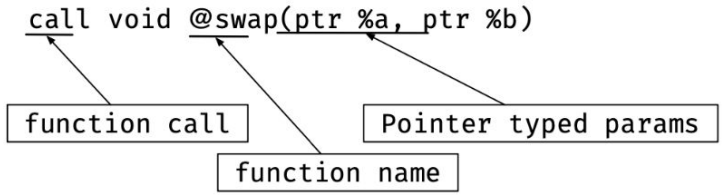


store the pointer %b1 to the memory location that %b points to

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM IR

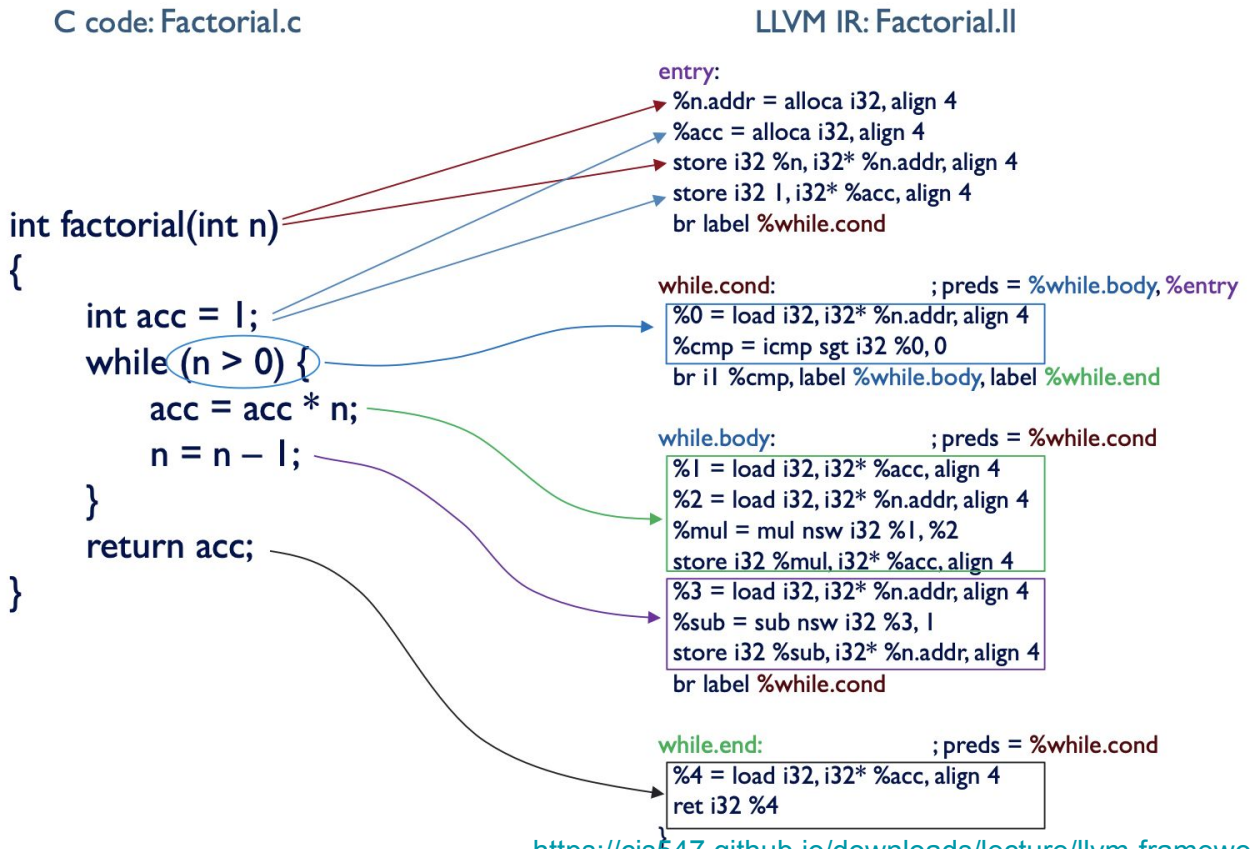




call instruction will be used to build control flow.

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

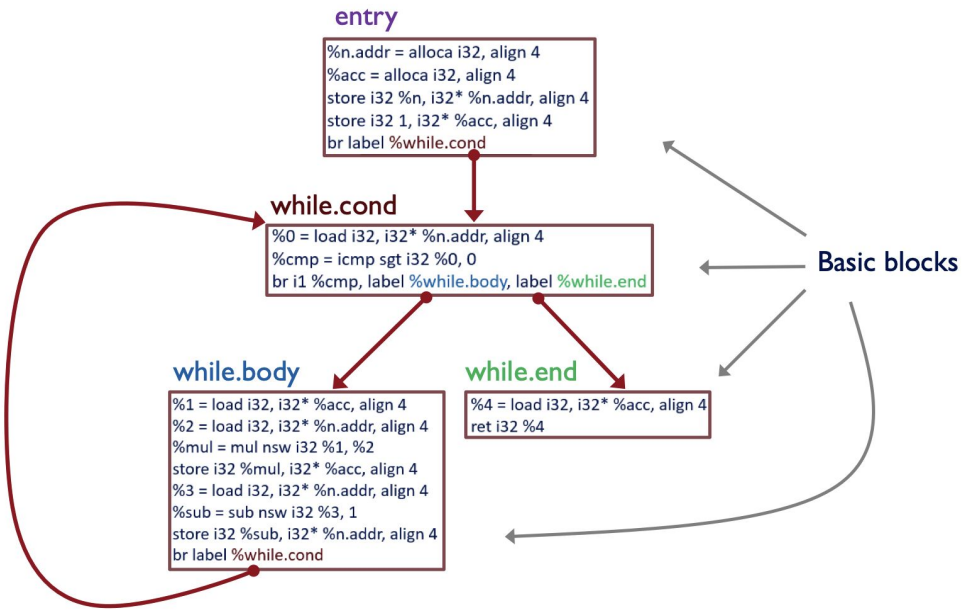
C program -> LLVM IR



https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

Basic Blocks and Control Flow Graph (CFG)

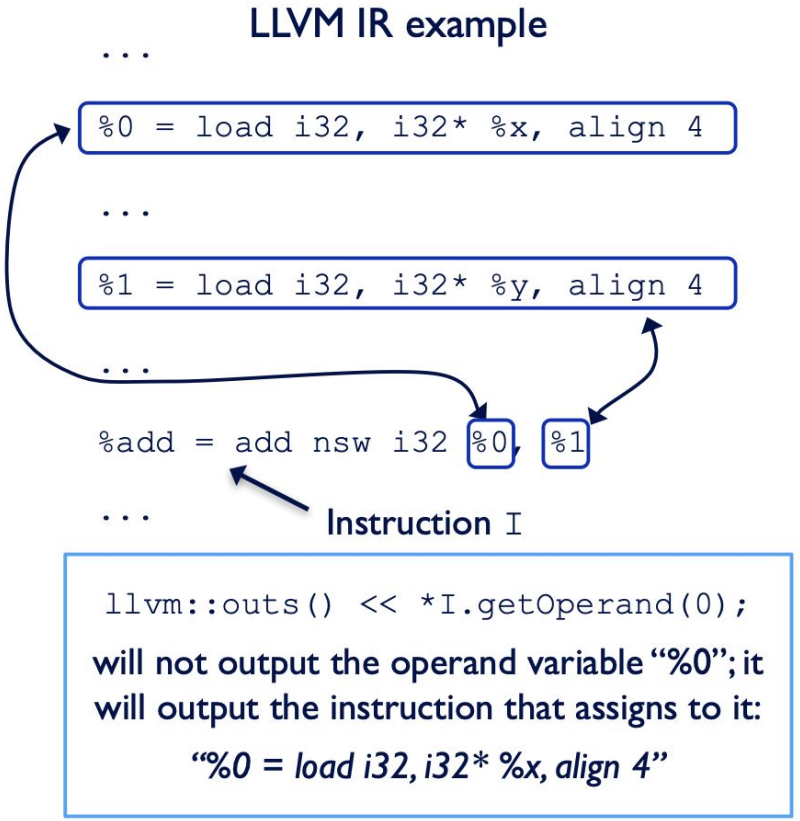
```
int factorial(int n){
    int acc = 1;
    while(n>0){
        acc = acc *n;
        n=n-1;
    }
    return acc;
}
```




```
$clang -S -fno-discard-value-names -emit-llvm factorial.c
$opt -p=dot-cfg factorial.ll
$dot -Tpng .factorial.dot -o factorial.png
```

Instructions and Variables

- Each Variable <=> the Instruction that assigns to it.
- There is a unique instruction assigning to each variable since LLVM IR uses the SSA form.
- Thus, each instruction can be viewed as the name of the assigned variable.



https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

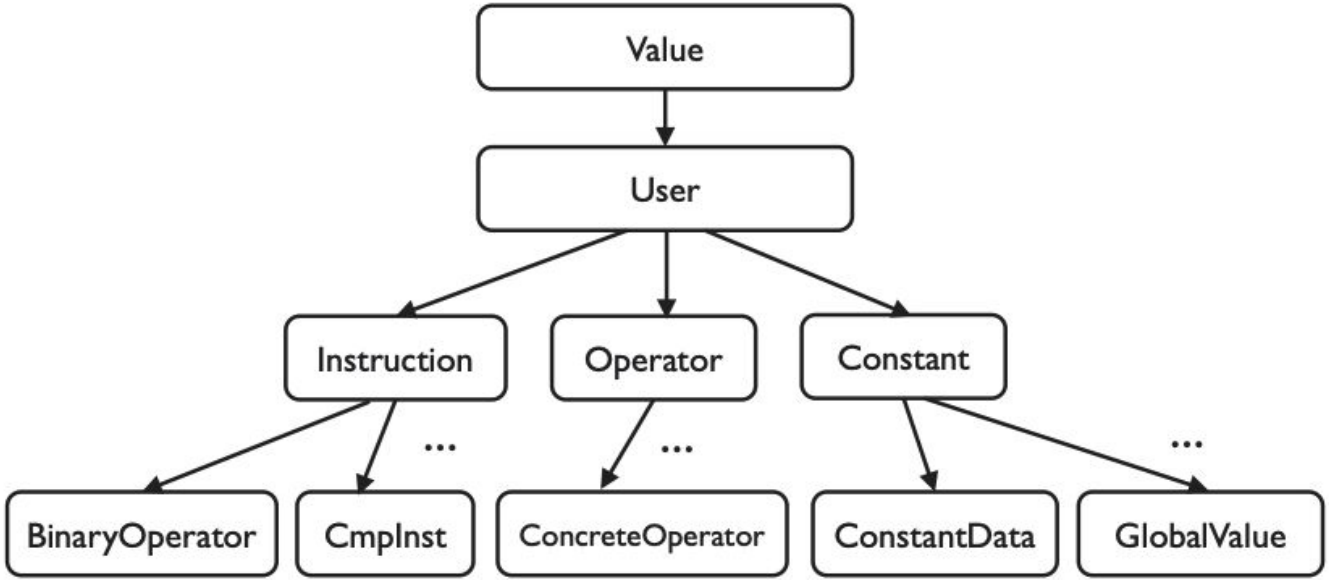
Printing Information

Use **outs()** and **errs()** to print instead of using std::cout , std::cerr, and printf.

Also, there is no equivalent of std::endl in LLVM.

- Example 1 - printing a function name (Function* F) :
  - ~~`std::cout << F->getName().str() << std::endl;`~~
  - `outs() <<F->getName() << "\n";`
- Example 2 - printing an instruction (Instruction *I):
  - `I->dump() or outs() << *I << "\n";`
- Example 3 - printing a basic block (BasicBlock* BB):
  - `BB->dump() or outs() << *BB << "\n";`

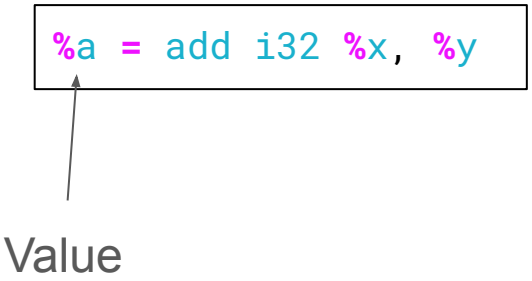
LLVM IR Class Hierarchy



https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

llvm::Value

- It is the base class of all values computed by a program that may be used as operands to other values.
- **Value** is the super class of other important classes such as **Instruction** and **Function**.



https://llvm.org/doxygen/classllvm_1_1Value.html#details

llvm::Value

- It is the base class of all values computed by a program that may be used as operands to other values.
- **Value** is the super class of other important classes such as **Instruction** and **Function**.

```
%a = add i32 %x, %y
```

Value

```
Value *V;

V->getType();
V->getName();
V->getNumUses();
V->dump();
```

Useful API

https://llvm.org/doxygen/classllvm_1_1Value.html#details

llvm::Value

- It is the base class of all values computed by a program that may be used as operands to other values.
- **Value** is the super class of other important classes such as **Instruction** and **Function**.

```
%a = add i32 %x, %y
```

Value

```
Value *V;

V->getType();
V->getName();
V->getNumUses();
V->dump();
```

Useful API

```
for (Value *V : Values) {
    errs() << "Name: " << V->getName() << "\n";
    errs() << "Uses: " << V->getNumUses() << "\n";
    V->dump();
}
```

https://llvm.org/doxygen/classllvm_1_1Value.html#details

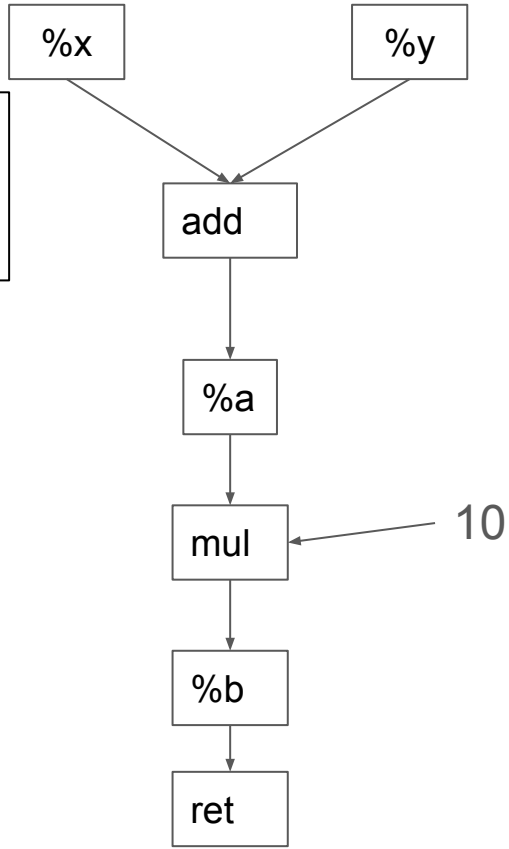
llvm::Value

```
%a = add i32 %x, %y
%b = mul i32 %a, 10
ret i32 %b
```

https://llvm.org/doxygen/classllvm_1_1Value.html#details

llvm::Value

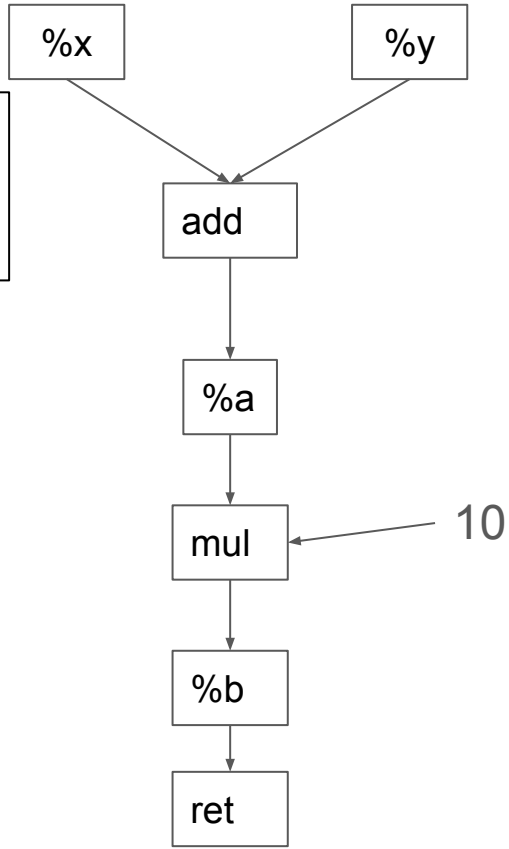
```
%a = add i32 %x, %y
%b = mul i32 %a, 10
ret i32 %b
```



https://llvm.org/doxygen/classllvm_1_1Value.html#details

llvm::Value

```
%a = add i32 %x, %y
%b = mul i32 %a, 10
ret i32 %b
```



Find who uses %a

```
Value *A = ...;

for (User *U : A->users())
{
    U->dump();
}
```

%b = mul i32 %a, 10

https://llvm.org/doxygen/classllvm_1_1Value.html#details

llvm::User

- Any **`Value`** that consumes other **`Values`** as its inputs (operands).

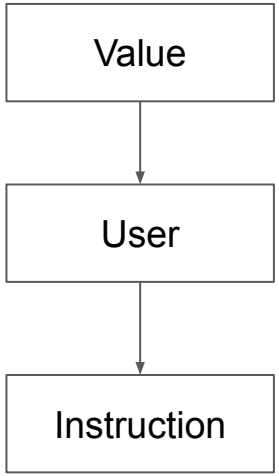
```
%z = add i32 %x, %y
```

the add instruction is a User of: %x and %y

```
User *U;

unsigned N = U->getNumOperands();

for (unsigned i = 0; i < N; ++i)
{
    Value *V = U->getOperand(i);
    V->dump();
}
```

https://llvm.org/doxygen/classllvm_1_1User.html

llvm::Instruction

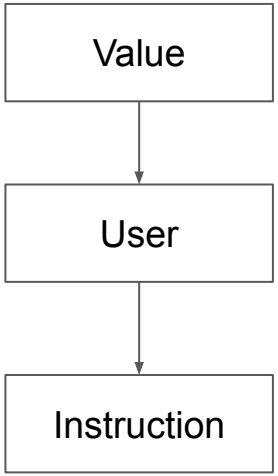


- A Value that uses other Values and represents an operation in a BasicBlock.

```
%a = add i32 %x, %y
```

https://llvm.org/doxygen/IR_2Instruction_8h_source.html

llvm::Instruction



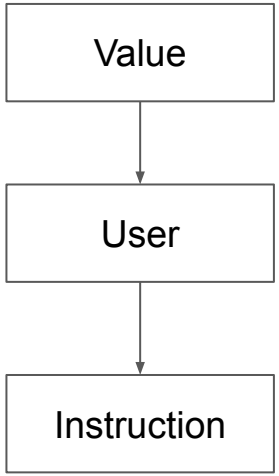
- A Value that uses other Values and represents an operation in a BasicBlock.

```
%a = add i32 %x, %y
```

```
Instruction *I = ...;

I->getOpcode();
I->getParent();
I->getNumOperands();
```

https://llvm.org/doxygen/IR_2Instruction_8h_source.html

llvm::Instruction



- A Value that uses other Values and represents an operation in a BasicBlock.

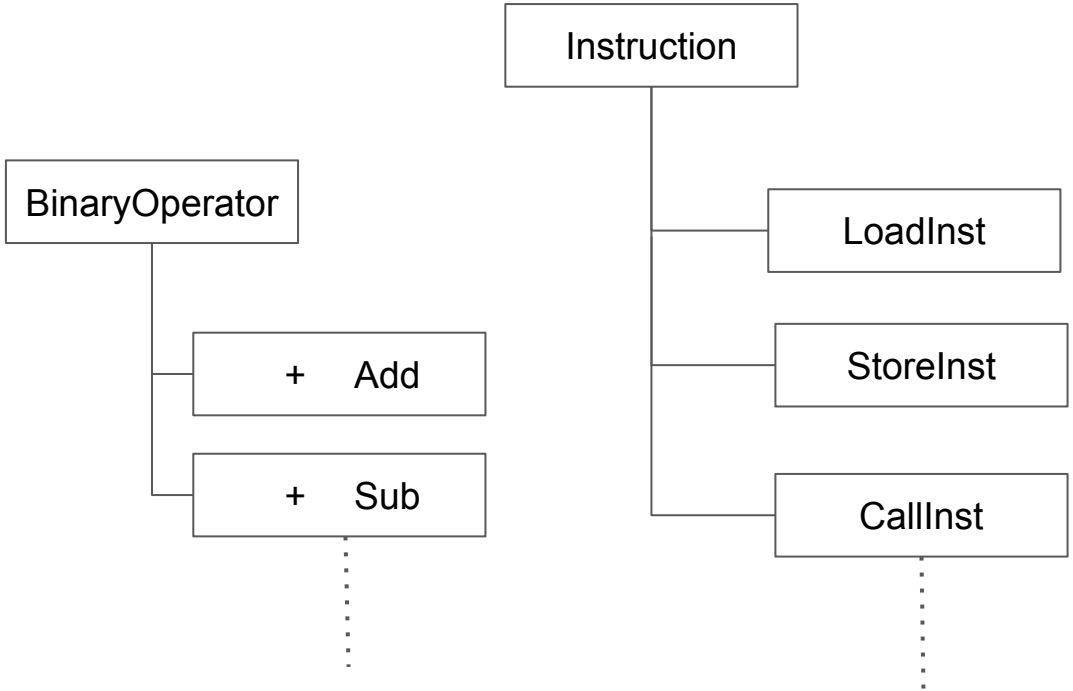
```
%a = add i32 %x, %y
```

```
Instruction *I = ...;

I->getOpcode();
I->getParent();
I->getNumOperands();
```

```
for (Instruction &I : BB) {
    errs() << "Opcode: " <<
I.getOpcodeName() << "\n";

    for (Value *V : I.operands()) {
        V->dump();
    }
}
```

https://llvm.org/doxygen/IR_2Instruction_8h_source.html

llvm::Instruction



https://llvm.org/doxygen/IR_2Instruction_8h_source.html

llvm::Instruction

```
for (Instruction &I : BB) {

    if (auto *BO = dyn_cast<BinaryOperator>(&I)) {

        if (BO->getOpcode() == Instruction::Add) {
            errs() << "Found ADD\n";
        }
    }
}
```

https://llvm.org/doxygen/IR_2Instruction_8h_source.html

Checking Types

```
isa<T>(V)
cast<T>(V)
dyn_cast<T>(V)
```

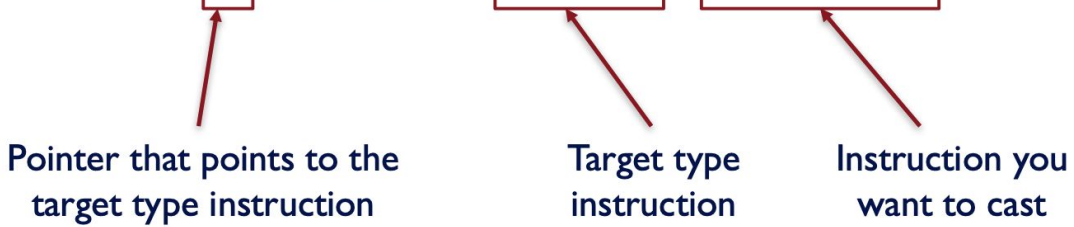
```
if (auto *CI =
dyn_cast<CallInst>(&I)) {
    ...
}
```

https://llvm.org/doxygen/IR_2Instruction_8h_source.html

Checking Instruction Type

A dynamic cast converts an instruction to a more specific type in its class hierarchy at runtime:

```
auto *AI = dyn_cast<CastInst>(Instruction)
```



If target type is not in original instruction's class hierarchy, `AI` will point to NULL. This property can be used to check if an instruction is of a particular type:

```
if (LoadInst *LI = dyn_cast<LoadInst>(I)) {
    // if I can be converted to LoadInst, do something
}
```

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

Instruction: StoreInst

An instruction for storing to memory.

E.g. store **T** v, **T*** %y
Store value v of type **T** into location pointed to by register %y.
The value may be a constant or a register.

T = i32

```
int x = 5;
```

```
%x = alloca i32, align 4
store i32 5, i32* %x, align 4
```

store: Set value of integer pointed to by register %x to 5

T = i32*

```
int* x = 0;
```

```
%x = alloca i32*, align 8
store i32* null, i32** %x, align 8
```

store: Set value of pointer pointed to by register %x to NULL

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

Instruction: LoadInst

An instruction for reading from memory.

E.g. %x = load **T**, **T*** %y

Load value of type **T** into register %x from location pointed to by register %y.

T = i32

```
int x = …;
… = 1 / x;
```

```
%x = alloca i32, align 4
%1 = load i32, i32* %x
%div = sdiv i32 1, %1
```

load: Load integer value into register %1 from location pointed to by register %x

T = i32*

```
int *x  = …;
if (x) …
```

```
%x = alloca i32*, align 8
%1 = load i32*, i32** %x
%tobool = icmp ne i32* %1, null
```

load: Load pointer value into register %1 from location pointed to by register %x

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

Instruction: BinaryOperator

An instruction for binary operations.

```
int x = 0;
int y = 2;
z = y + x;
```

Could be +, -, *, /

```
%1 = load i32, i32* %y, align 4
%2 = load i32, i32* %x, align 4
%z = add nsw i32 %1, %2
```

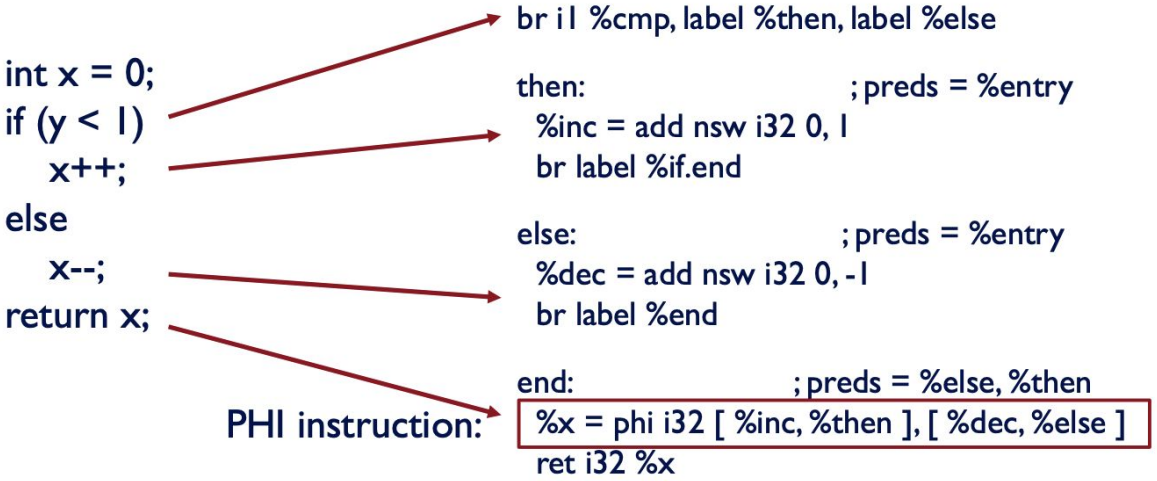
Could be add, sub, mul, udiv, sdiv

add: Store the sum of %1and %2 in %z
(nsw: no signed wrap)

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

Instruction: PhiNode

The 'phi' instruction is used to implement the 'phi' node in the SSA form.

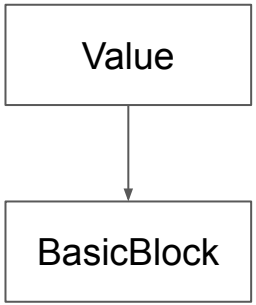


phi: Assign to %x the value of:

- %inc if predecessor basic block is %then, and
- %dec if predecessor basic block is %else

https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

llvm::BasicBlock

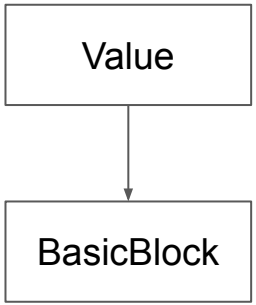



```
entry:
    %a = add i32 %x, %y
    %b = mul i32 %a, 10
    ret i32 %b
```

- A sequence of instructions with one entry and a terminating instruction.

https://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html

llvm::BasicBlock




```
entry:
    %a = add i32 %x, %y
    %b = mul i32 %a, 10
    ret i32 %b
```

- A sequence of instructions with one entry and a terminating instruction.

Iterate instructions

```
BasicBlock *BB = ...;

for (Instruction &I : *BB)
{
    I.dump();
}
```

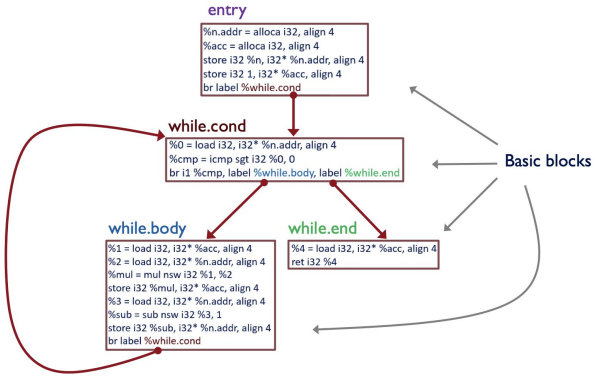
Get parent function

```
Function *F = BB->getParent();
```

https://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html

llvm::BasicBlock & CFG

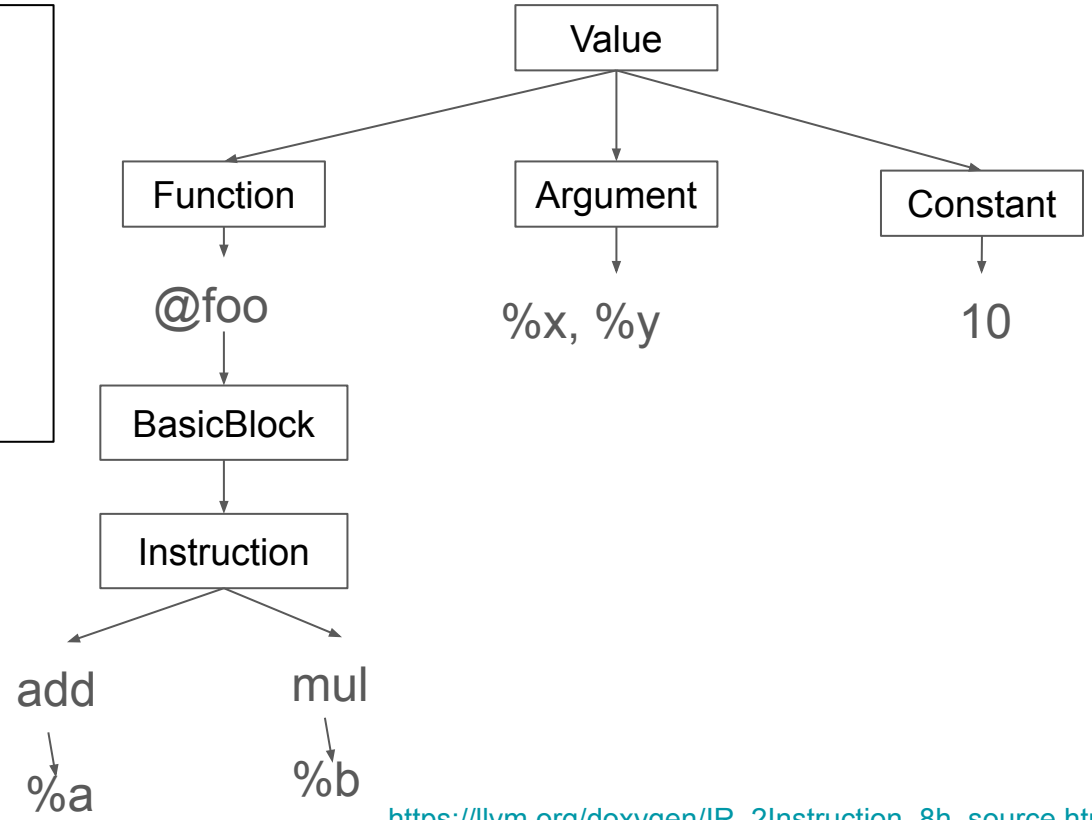
```
for (BasicBlock &BB : F) {

    errs() << "BB: "
           << BB.getName()
           << "\n";

    for (BasicBlock *Succ : successors(&BB)) {
        errs() << "  -> "
               << Succ->getName()
               << "\n";
    }
}
```



https://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html

llvm::Function

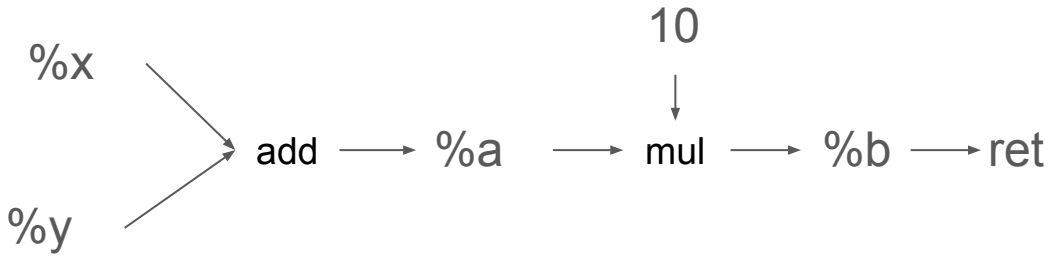
```
define i32 @foo(i32 %x,
i32 %y) {

entry:
    %a = add i32 %x, %y
    %b = mul i32 %a, 10
    ret i32 %b
}
```



https://llvm.org/doxygen/IR_2Instruction_8h_source.html

Use-def Graph

```
define i32 @foo(i32 %x, i32
%y) {

entry:
    %a = add i32 %x, %y
    %b = mul i32 %a, 10
    ret i32 %b
}
```



https://llvm.org/doxygen/IR_2Instruction_8h_source.html

Playground

https://godbolt.org/

LLVM Pass: In-Tree

- Pass is implemented **inside `llvm-project`**
- Integrated directly with LLVM's source and build system
- Requires building LLVM after adding/modifying the pass
- Best when modifying or extending LLVM itself

```
llvm-project/
└── llvm/
    ├── include/llvm/Transforms/MyPass.h
    └── lib/Transforms/MyPass.cpp
```

LLVM Pass: In-Tree

- LLVM compiler development
- New optimization passes
- Modifying LLVM internals
- Upstream LLVM contributions
- Passes requiring tight integration with LLVM

LLVM Pass: In-Tree

- LLVM compiler development
- New optimization passes
- Modifying LLVM internals
- Upstream LLVM contributions
- Passes requiring tight integration with LLVM

LLVM Pass: Out-Of-Tree

- Pass is maintained **outside `llvm-project`**
- Uses an existing LLVM installation
- Compiled as a **shared library/plugin**
- Loaded dynamically by **`opt`**
- Faster iteration and easier experimentation

```
my-pass/
├── CMakeLists.txt
└── MyPass.cpp
```

https://github.com/banach-space/llvm-tutor

```
$clang -S -emit-llvm test.c -o test.ll
$opt   -load-pass-plugin=./build/libMyPass.so -passes="my-pass"  test.ll
```