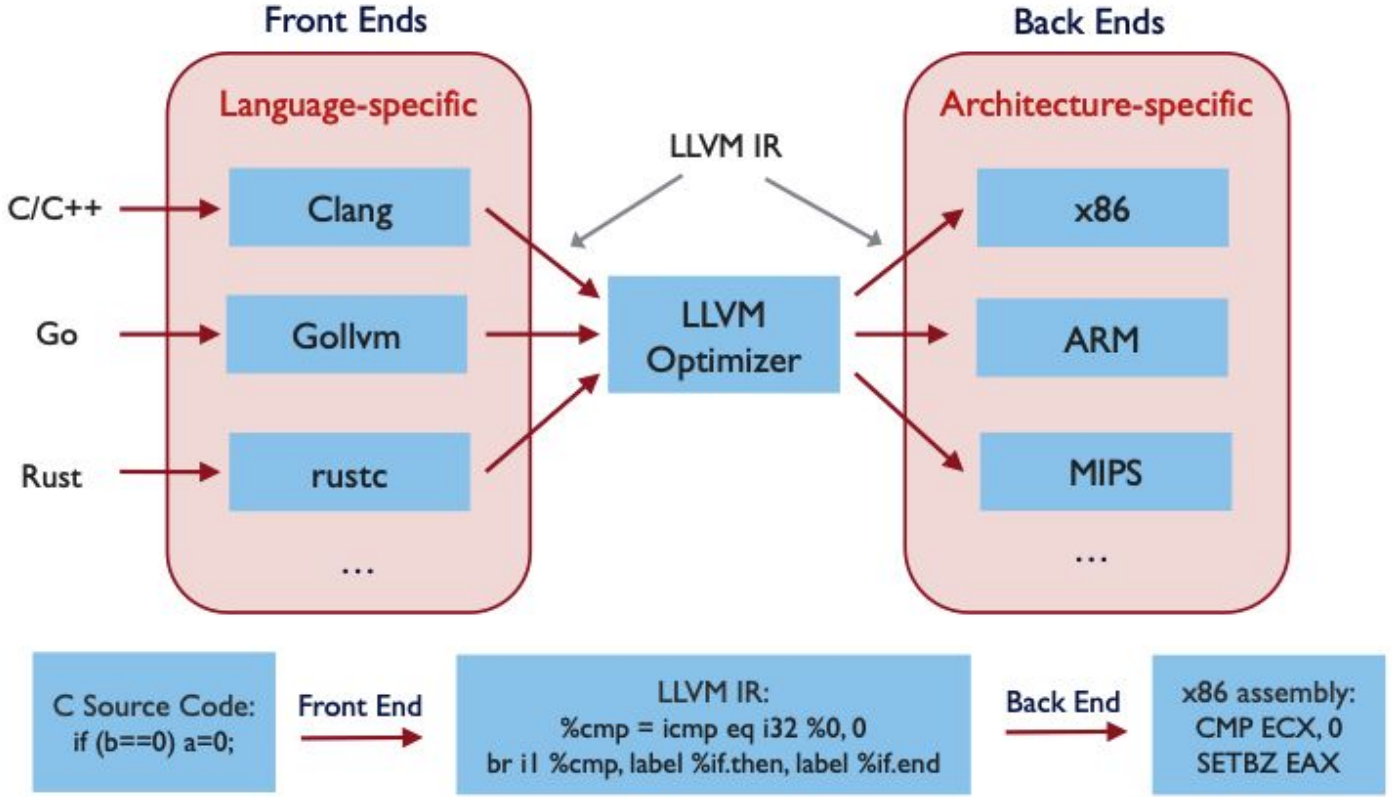
L8: LLVM IR

Software Analysis

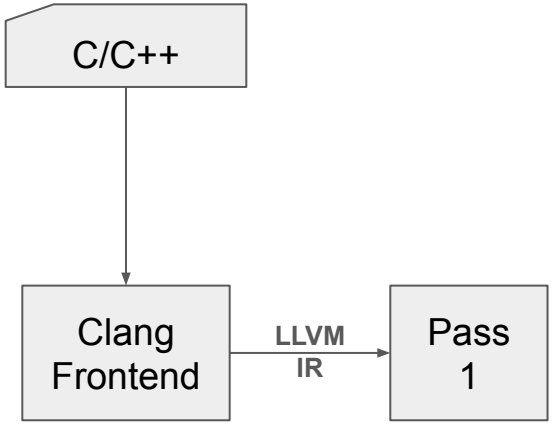
IIT Guwahati · August 2026

Overview of LLVM



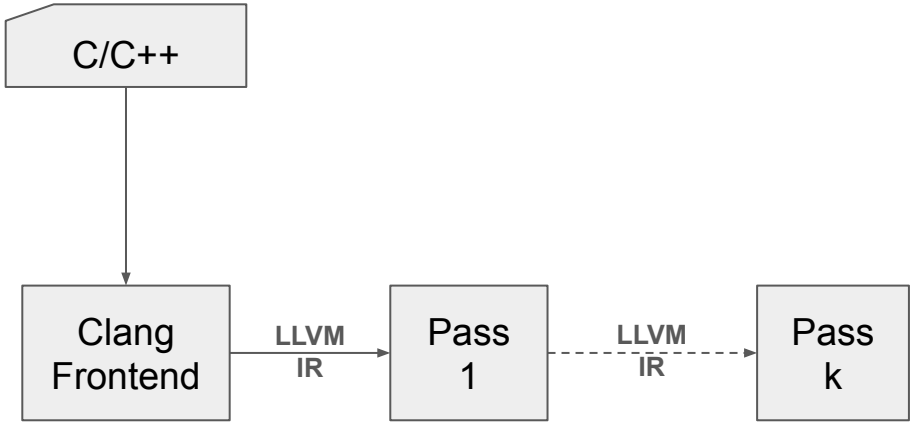
https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM Passes



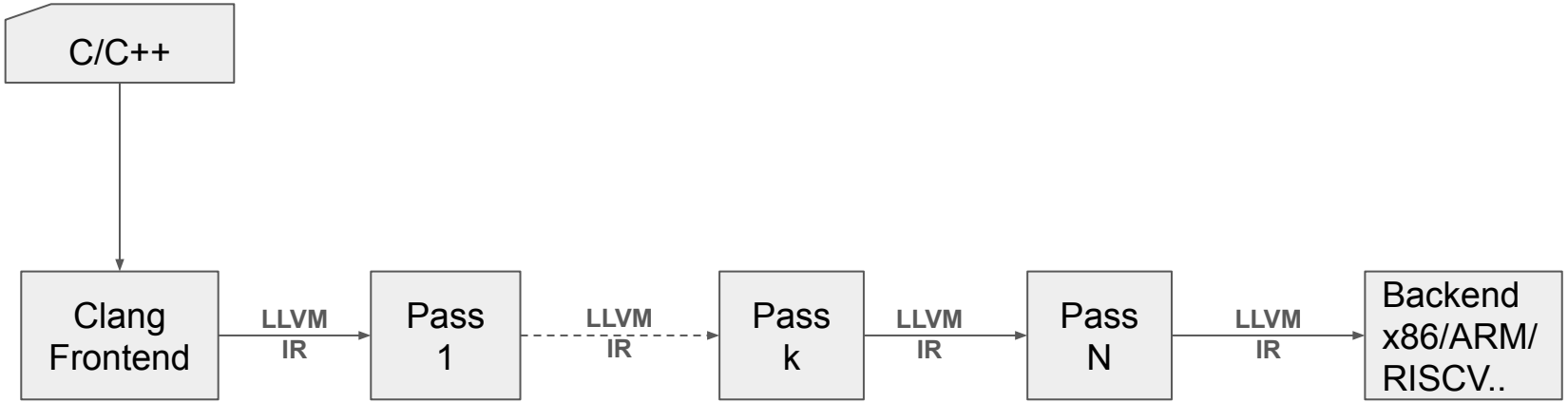
https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM Passes



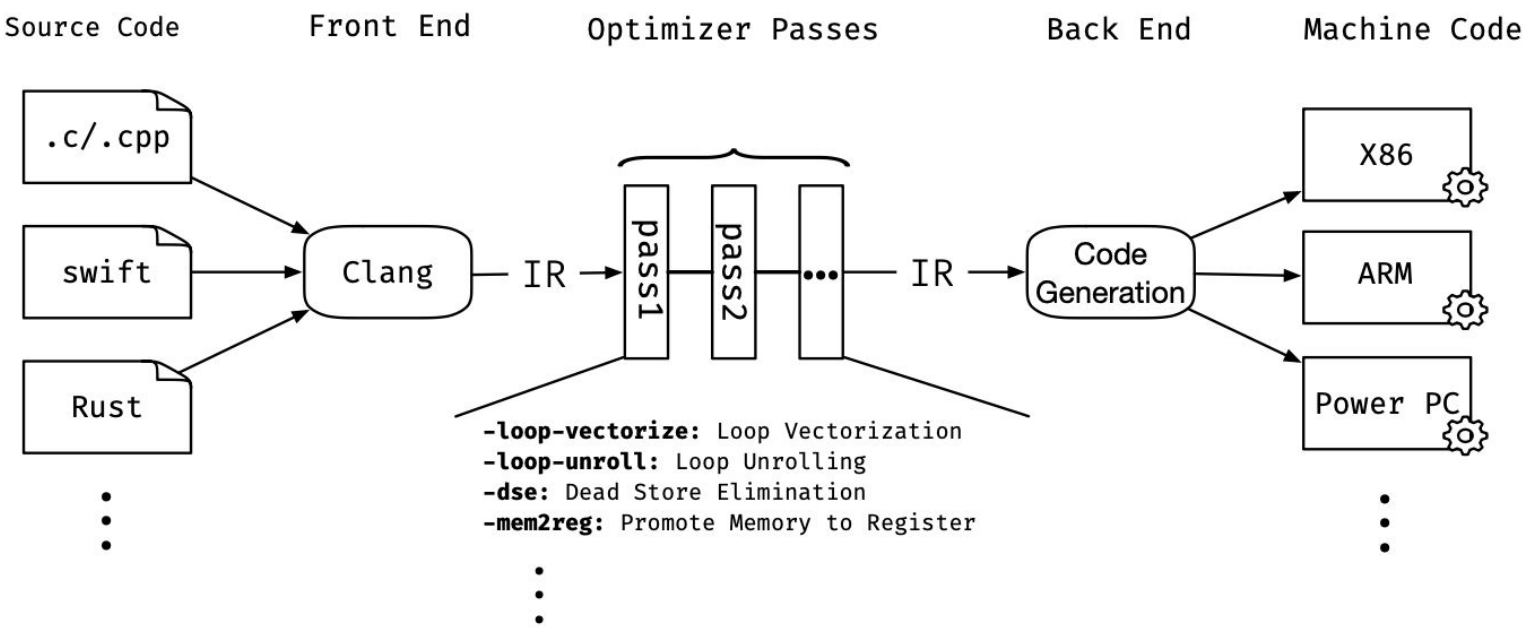
https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM Passes



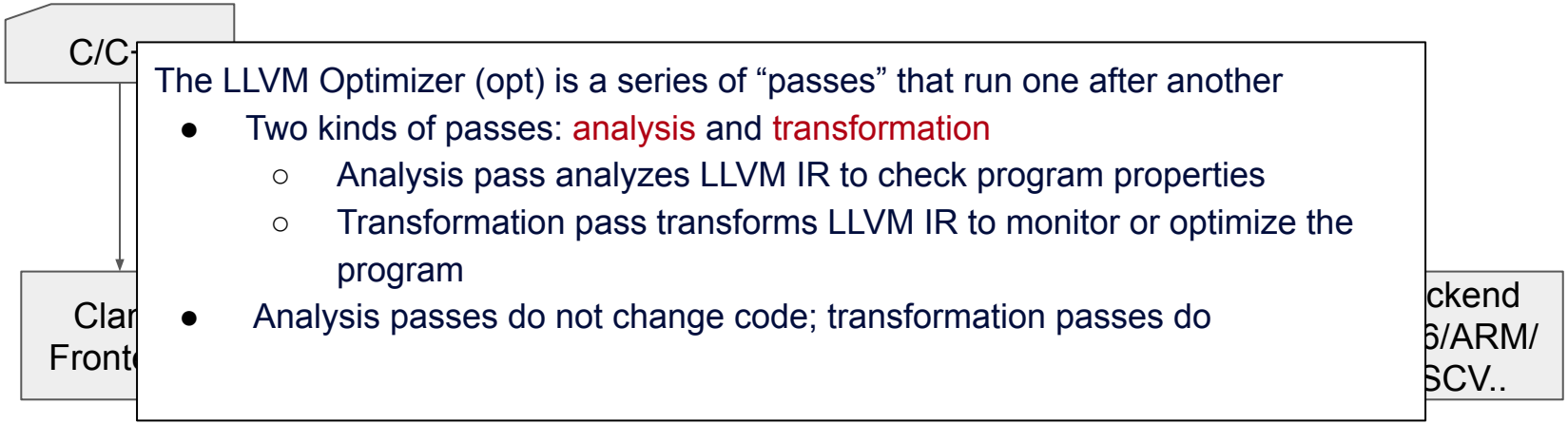
https://cis547.github.io/downloads/lecture/llvm-framework-primer.pdf

LLVM Passes



https://github.com/SVF-tools/Software-Security-Analysis/blob/slides/2.SVFIR-control-flow.pdf

LLVM Passes



- LLVM is typically extended by implementing new passes that look at and change the LLVM IR as it flows through the compilation process.

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

LLVM IR

sample.c

```
int global =0;
int sum(int x, int y){
return x+y;
}
```

$ clang -S -emit-llvm sample.c

Perform preprocessing and compilation steps only, and emit textual assembly

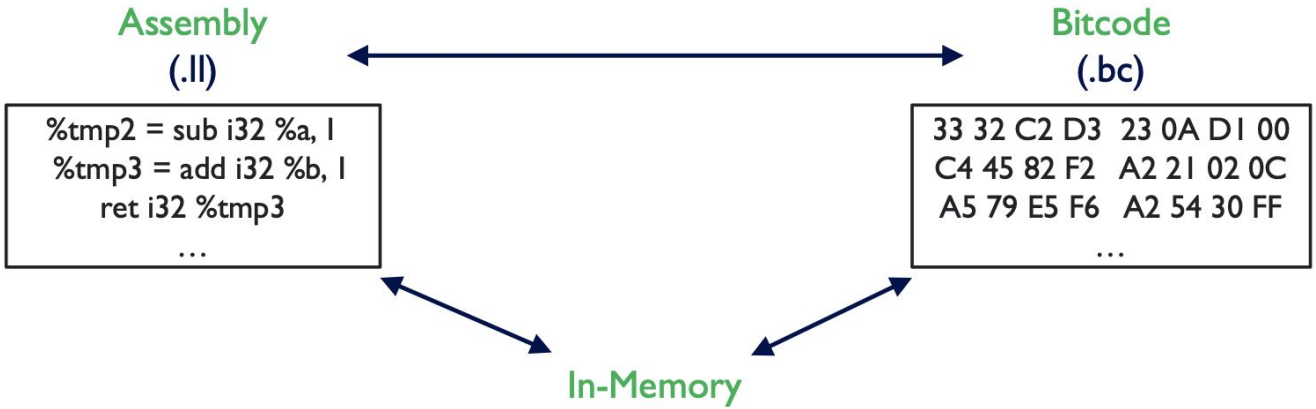
sample.ll

```
@global = global i32 0, align 4
define i32 @sum(i32 noundef %0, i32
noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
  %5 = load i32, ptr %3, align 4
  %6 = load i32, ptr %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
```

LLVM IR

The formats:

- In-memory: binary in-memory format, used during compilation process
- Bitcode: binary on-disk format, suitable for fast loading
  - (Obtained by “clang -emit-llvm -c factorial.c -o xxx.bc”)
- Assembly: human-readable format
  - (Obtained by “clang -emit-llvm -S -c factorial.c -o xxx.ll”)



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

LLVM Passes

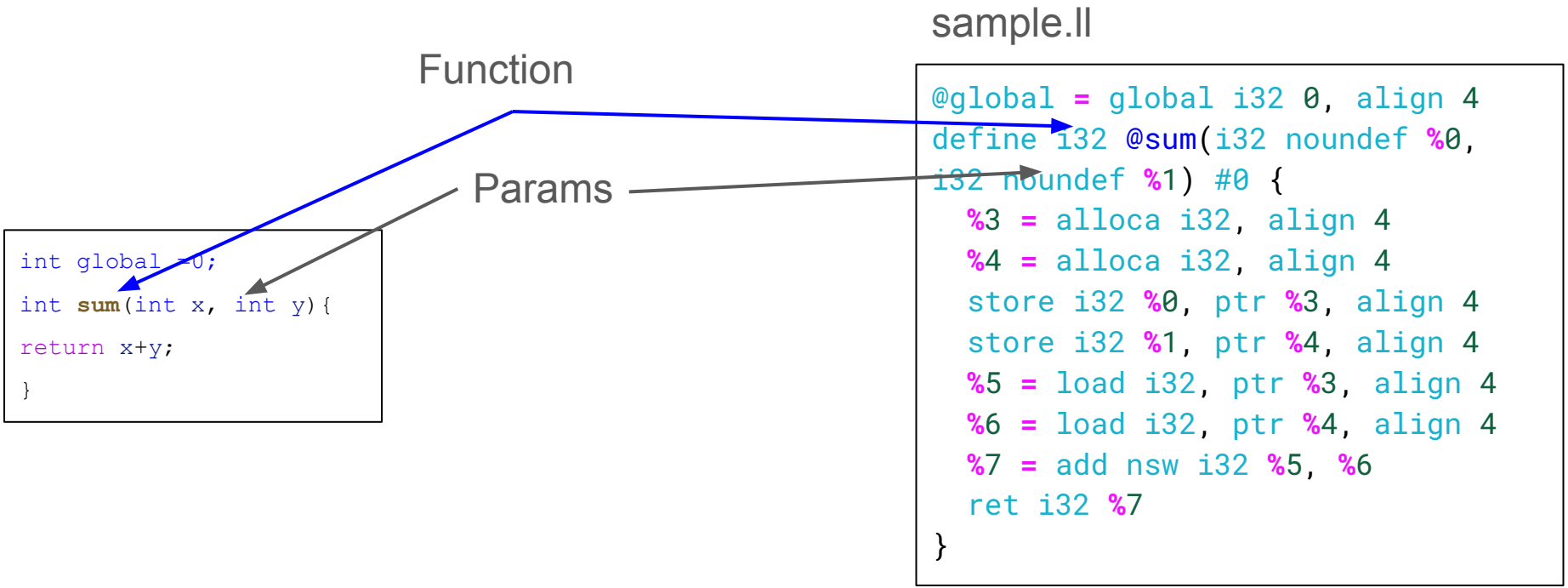
Function

```
int global = 0;
int sum(int x, int y){
return x+y;
}
```

sample.ll

```
@global = global i32 0, align 4
define i32 @sum(i32 noundef %0, i32
noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
  %5 = load i32, ptr %3, align 4
  %6 = load i32, ptr %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
```

LLVM Passes


Function
Params
int global =0;
int sum(int x, int y){
return x+y;
}
sample.ll
@global = global i32 0, align 4
define i32 @sum(i32 noundef %0,
i32 noundef %1) #0 {
%3 = alloca i32, align 4
%4 = alloca i32, align 4
store i32 %0, ptr %3, align 4
store i32 %1, ptr %4, align 4
%5 = load i32, ptr %3, align 4
%6 = load i32, ptr %4, align 4
%7 = add nsw i32 %5, %6
ret i32 %7
}

LLVM Passes

```
int global =0;
int sum(int x, int y){
return x+y;
}
```

Function

Params

`alloca` allocates memory on the **stack** for the current function invocation.

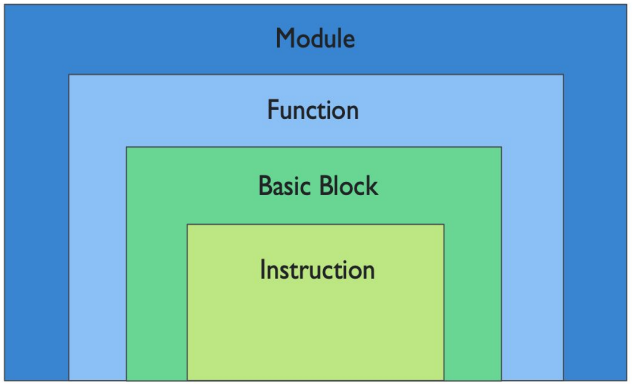
sample.ll

```
@global = global i32 0, align 4
define i32 @sum(i32 noundef %0, i32
noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
  %5 = load i32, ptr %3, align 4
  %6 = load i32, ptr %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
```

Program Structure in LLVM IR

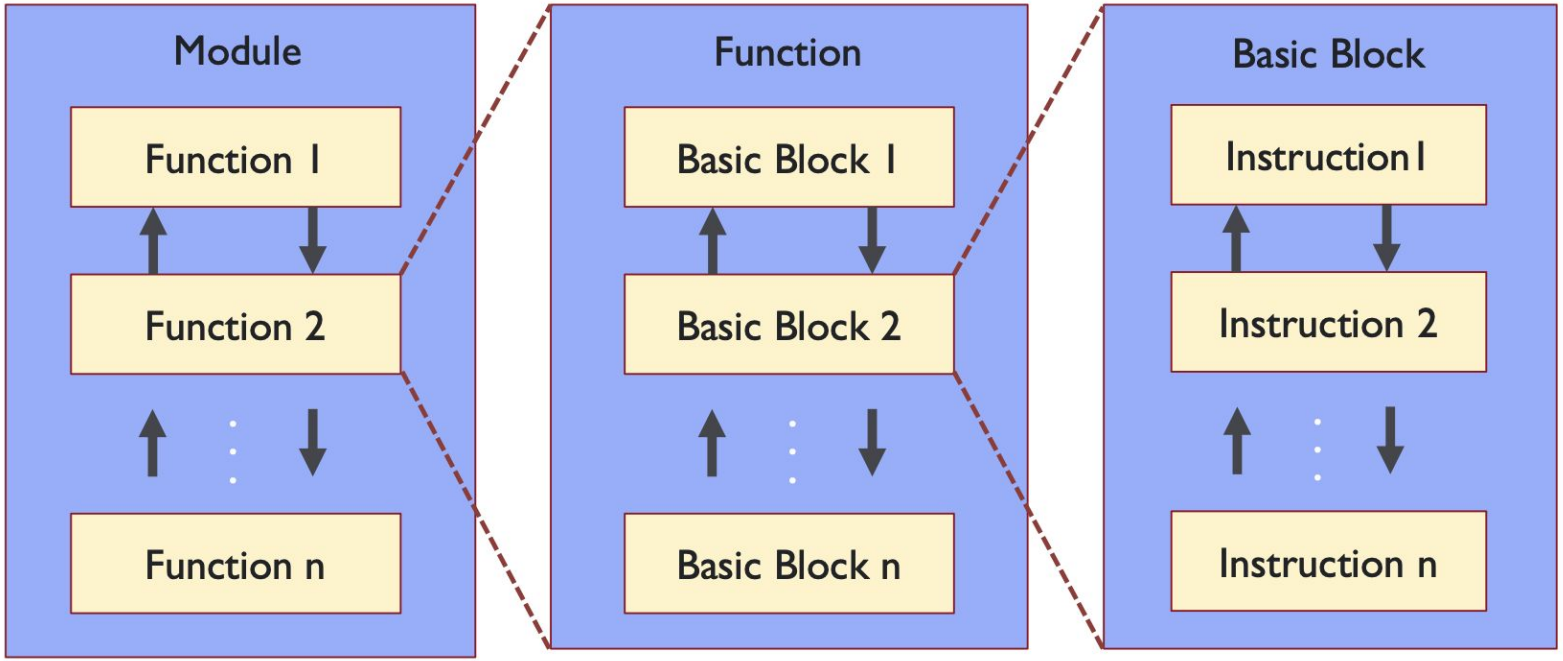
- **Module** is a top-level container of LLVM IR, corresponding to each translation unit of the front-end compiler.
- **Function** is a function in a programming language, including a function signature and several basic blocks. The first basic block in a function is called an entry basic block.
- **Basic Block** is a set of instructions that are executed sequentially, with only one entry and one exit, and non-head and tail instructions will not jump to other instructions in the order they are executed.
- **Instruction** is the smallest executable unit in LLVM IR; each instruction occupies a single line.

Instruction ⊂ Basic Block ⊂ Function ⊂ Module



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

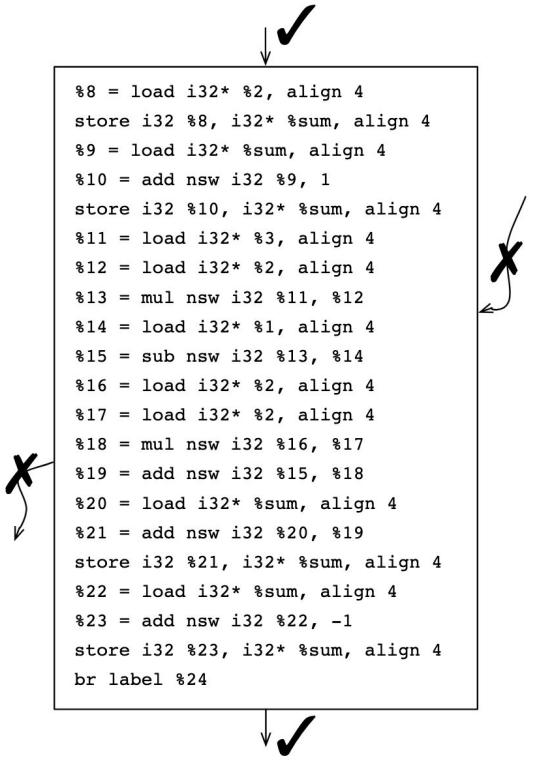
LLVM IR Interators



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

Basic Blocks

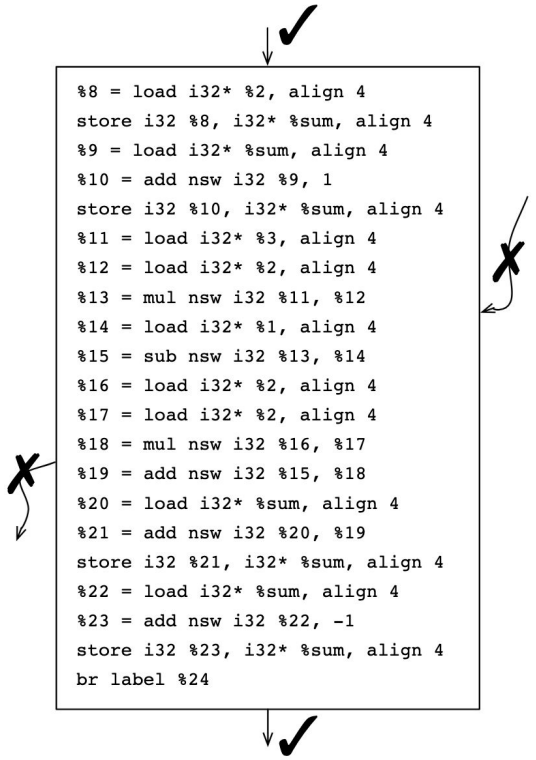
- Basic blocks are maximal sequences of consecutive instructions with the following properties:
  - The flow of control can only enter the basic block through the first instruction in the block.
  - There are no jumps into the middle of the block.
  - Control will leave the block without halting or branching, except possibly at the last instruction in the block.



https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/ControlFlowGraphs.pdf

Identifying Basic Blocks

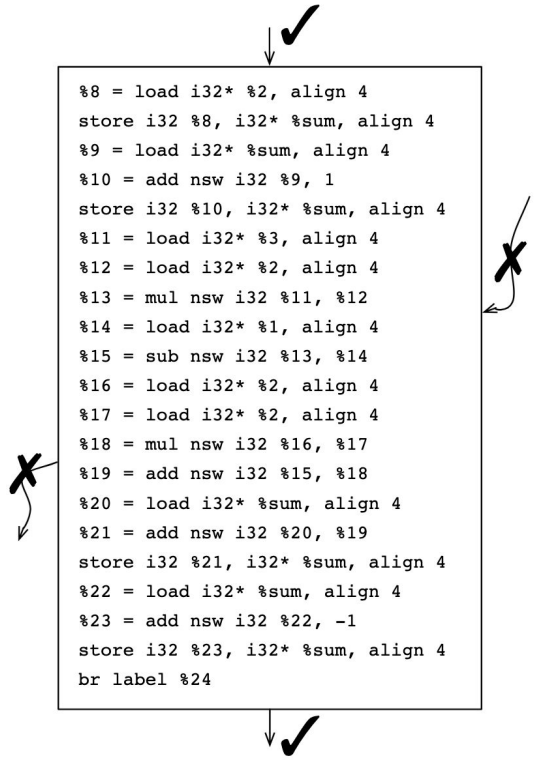
- The first instruction of a basic block is called a leader
- We can identity leaders via these three properties:
  - The first instruction in the intermediate code is a leader.
  - Any instruction that is the target of a conditional or unconditional jump is a leader.
  - Any instruction that immediately follows a conditional or unconditional jump is a leader.



https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/ControlFlowGraphs.pdf

Identifying Basic Blocks

- Once we have found the leaders, it is straightforward to find the basic blocks:
  - For each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.



https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/ControlFlowGraphs.pdf

Variables and Types

Two kinds of variables: local and global

- "%" indicates local variables Example: %1 = add nsw i32 %a, %tmp
- "@" indicates global variables: Example: @g = global i32 20, align 4

Two kinds of types: primitive (e.g. integer, floating-point) and derived (e.g. pointer, struct)

- Integer type is used to specify an integer of desired bit width:
  - i1 A single-bit integer
  - i32 A 32-bit integer

Pointer type is used to specify memory locations:

- i32** A pointer to a pointer to an integer.
- i32 (i32*) * A pointer to a function that takes as argument a pointer to an integer, and returns an integer as result.

https://llvm.org/docs/LangRef.html#type-system

LLVM IR Interators

```
Approach 1 (using STL iterator):
for (Function::iterator FI = F->begin(); FI != F->end(); FI++) {
for (BasicBlock::iterator BI = FI->begin(); BI != FI->end(); BI++) {
// some operations
} }
```

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

LLVM IR Interators

```
Approach 1 (using STL iterator):
for (Function::iterator FI = F->begin(); FI != F->end(); FI++) {
for (BasicBlock::iterator BI = FI->begin(); BI != FI->end(); BI++) {
// some operations
} }
```

```
Approach 2 (using auto keyword):
for (auto FI = F->begin(); FI != F->end(); FI++) {
for (auto BI = FI->begin(); BI != FI->end(); BI++) {
// some operations
}}
```

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

LLVM IR Interators

```
Approach 1 (using STL iterator):
for (Function::iterator FI = F->begin(); FI != F->end(); FI++) {
for (BasicBlock::iterator BI = FI->begin(); BI != FI->end(); BI++) {
// some operations
} }
```

```
Approach 2 (using auto keyword):
for (auto FI = F->begin(); FI != F->end(); FI++) {
for (auto BI = FI->begin(); BI != FI->end(); BI++) {
// some operations
}}
```

```
Approach 3 (using InstIterator):
#include llvm/IR/InstIterator.h
for (inst_iterator It = inst_begin(F), E = inst_end(F); It != E;
++It){
// some operations
}
```

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

The SSA Form

- The Static Single Assignment (SSA) form requires that every variable be defined only **once**, but may be used multiple times.
- SSA was proposed in 1988 and an efficient algorithm was developed in IBM, which is still in use in many compilers.

https://llvm.org/docs/LangRef.html#type-system

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

Why SSA?

```
int f(int a) {
    int x = 10;

    if (a > 0)
        x = 20;

    return x;
}
```

At `return x`, which value of `x` are we talking about?

Why SSA?

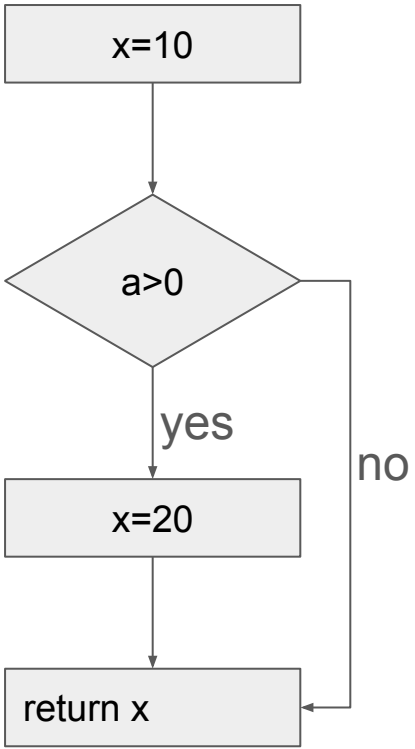
```
int f(int a) {
    int x = 10; ← Definition 1

    if (a > 0)
        x = 20; ← Definition 2

    return x; ← Use
}
```

At `return x`, which value of `x` are we talking about?

Why SSA?

```
int f(int a) {
    int x = 10; ← Definition 1

    if (a > 0)
        x = 20; ← Definition 2

    return x; ← Use
}
```

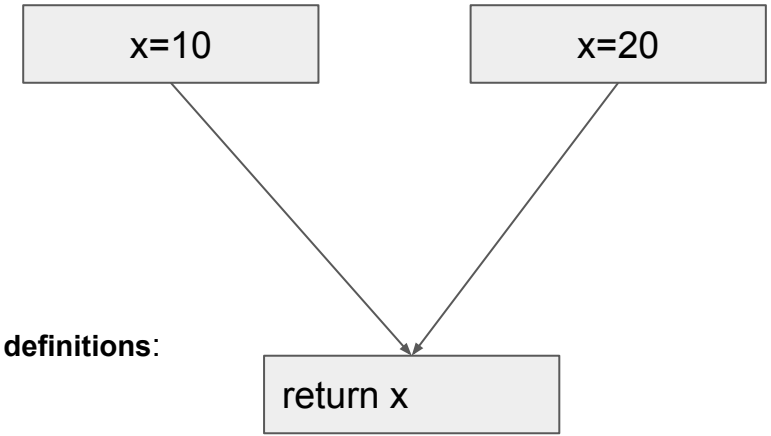
At `return x`, which value of `x` are we talking about?



Why SSA?

```
int f(int a) {
    int x = 10; ← Definition 1

    if (a > 0)
        x = 20; ← Definition 2

    return x; ← Use
}
```

At `return x`, which value of `x` are we talking about?



At `return x`, there are **two possible definitions**:

Why SSA?

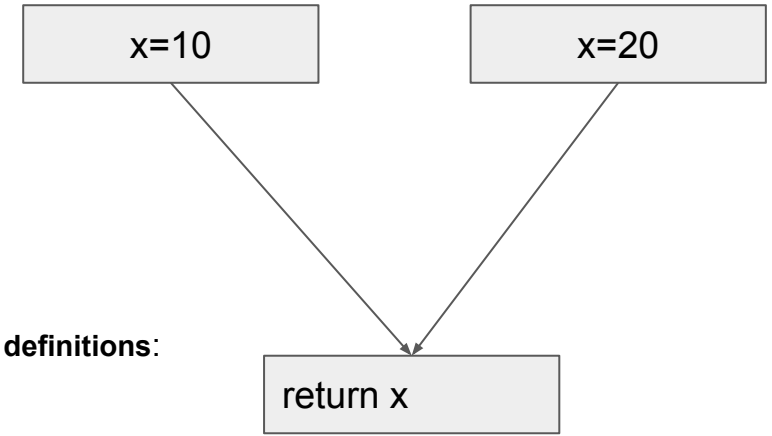
```
int f(int a) {
    int x = 10; ← Definition 1

    if (a > 0)
        x = 20; ← Definition 2

    return x; ← Use
}
```

Analysis has to determine:

Can `x = 10` reach this use? yes

Can `x = 20` reach this use? yes

x at return ∈ {10, 20}



At `return x`, which value of `x` are we talking about?

At `return x`, there are **two possible definitions**:

Why SSA?

```
int f(int a) {
    int x = 10; ← Definition 1

    if (a > 0)
        x = 20; ← Definition 2


}
```

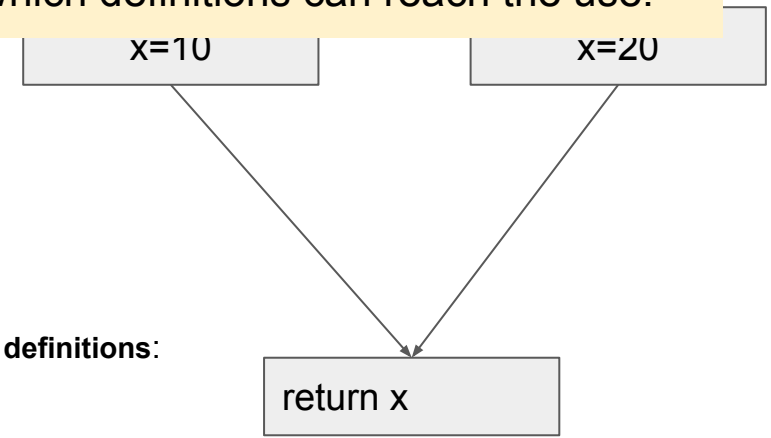
Analysis has to determine:

Can `x = 10` reach this use? yes

Can `x = 20` reach this use? yes

x at return ∈ {10, 20}

You need to analyze the CFG and determine which definitions can reach the use.



At `return x`, which value of `x` are we talking about?

At `return x`, there are **two possible definitions**:

Why SSA?

```
int f(int a) {
    int x1 = 10;

    if (a > 0)
        x2 = 20;

     x3=phi(x2,x1)

    return x3;
}
```

SSA form

Without SSA:

return x;

Which definition of x reaches here?

With SSA:
There is exactly **one definition of x3**:

x3 = phi(x2, x1)

It has been made explicit in the PHI node.

Why SSA?

```
int f(int a) {
    int x1 = 10;

    if (a > 0)

    return x;
}
```

Without SSA:

return x;

Which definition of x reaches here?

The problem isn't multiple assignments to x; it's knowing which definition reaches each use. SSA gives every value a unique identity, and PHI makes merges from different control-flow paths explicit.

With SSA:

There is exactly **one definition of x3**:

x3 = phi(x2, x1)

It has been made explicit in the PHI node.

Exercise

```
int f(int a) {
    int x = 10;
    int y = 20;
    int z = 30;
    int w = 40;

    if (a > 0) {
        x = x + 1;
        y = y + 2;
        z = z + 3;
        w = w + 4;
    }

    return x + y + z + w;
}
```

Exercise

```
int f(int a) {
    int x = 10;
    int y = 20;
    int z = 30;
    int w = 40;

    if (a > 0) {
        x = x + 1;
        y = y + 2;
        z = z + 3;
        w = w + 4;
    }

    return x + y + z + w;
}
```

non-SSA

```
x1 = 10
y1 = 20
z1 = 30
w1 = 40

if (a > 0) {
    x2 = x1 + 1
    y2 = y1 + 2
    z2 = z1 + 3
    w2 = w1 + 4
}

x3 = phi(x2, x1)
y3 = phi(y2, y1)
z3 = phi(z2, z1)
w3 = phi(w2, w1)

return x3 + y3 + z3 + w3
```

SSA

Exercise

C Code

```
x = 0;
if (y < 1) {
    x++;
} else {
    x--;
}
return x;
```

SSA Code

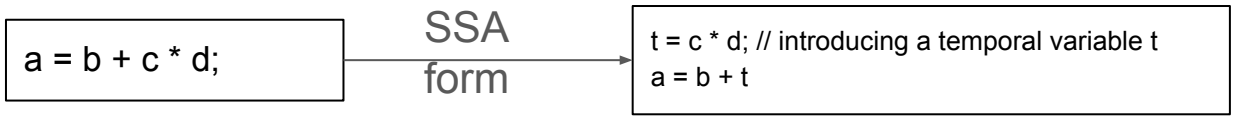
```
x_1 := 0;
if (y_1 < 1) {
    x_2 := x_1 + 1;
} else {
    x_3 := x_1 - 1;
}
// do I return x_2 or x_3?

x_4 := phi(x_2, x_3);
// return x_4 instead

return x_4;
```

LLVM Intermediate Representation (IR)

3-address code style

- Three addresses and one operator. Each instruction typically has at most three operands, aligning with the principles of SSA form, where each variable is assigned exactly once and allows for efficient analyses and optimizations.
- For example, **‘a = b op c‘**, where **‘a‘, ‘b‘, ‘c‘** are either programmer defined variables (e.g., heap, global or stack), constants or compiler-generated temporary names.
- ‘op‘ stands for an operation which is applied on ‘a‘ and ‘b‘.

```
@global = global i32 0, align 4
define i32 @sum(i32 noundef %0,
i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
  %5 = load i32, ptr %3, align 4
  %6 = load i32, ptr %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
```



Compiling a C Program to LLVM IR

Check your clang version

- clang -v

Keep the variable names.

- clang -S -fno-discard-value-names -emit-llvm sample.c -o sample.ll

```
int sum(int x, int y){
    return x+y;
}
```

```
define i32 @sum(i32 noundef %x, i32 noundef
%y) #0 {
entry:
  %x.addr = alloca i32, align 4
  %y.addr = alloca i32, align 4
  store i32 %x, ptr %x.addr, align 4
  store i32 %y, ptr %y.addr, align 4
  %0 = load i32, ptr %x.addr, align 4
  %1 = load i32, ptr %y.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

Compiling a C Program to LLVM IR

Check your clang version

- clang -v

Convert the LLVM IR to more compact static single assignment form.

```
opt -p=mem2reg -S test.ll -o test.ll
```

```
int sum(int x, int y){
    return x+y;
}
```

```
define i32 @sum(i32 noundef %x, i32 noundef
%y) #0 {
entry:
  %add = add nsw i32 %x, %y
  ret i32 %add
}
```