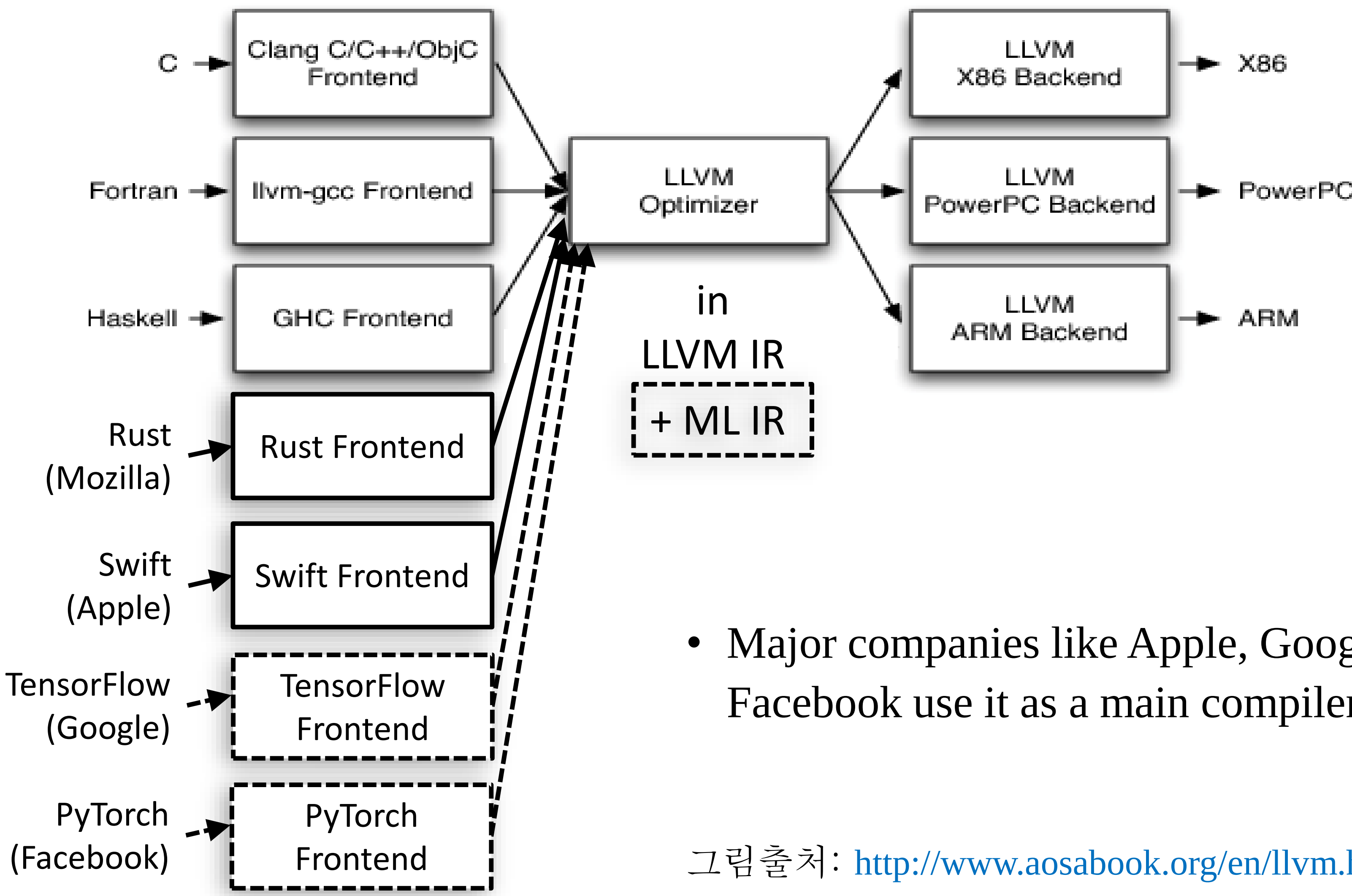
LLVM IR

Chung-Kil Hur

What is LLVM?



- Major companies like Apple, Google, Facebook use it as a main compiler.

그림출처: http://www.aosabook.org/en/llvm.html

A running example

C

LLVM IR

```
int f(int x, int y) {
  int z;
  if (x == y) {
    z = x + y;
  } else {
    z = x – y;
  }
  return z + z + z;
}
```

-> ?

LLVM IR: Control Flow Graph (CFG)

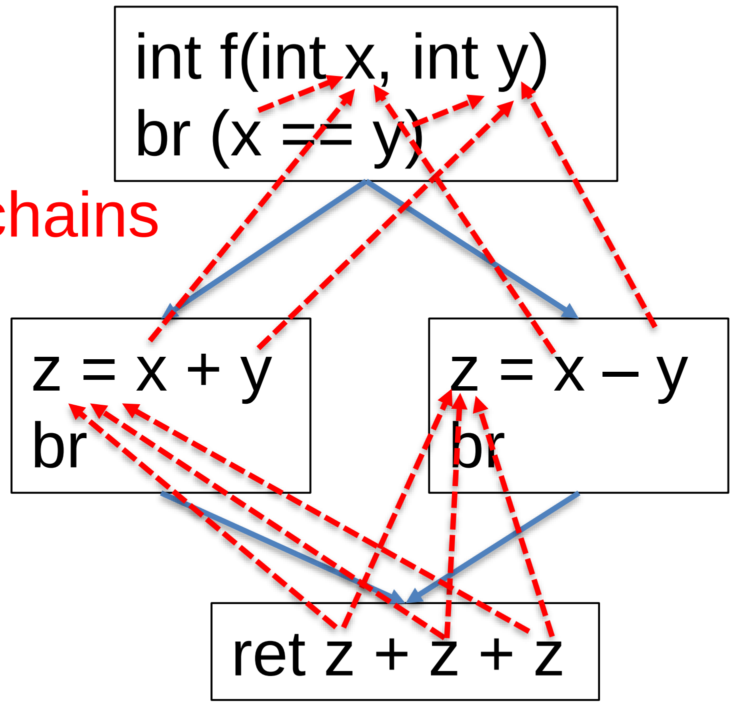
C

```
int f(int x, int y) {
  int z;
  if (x == y) {
    z = x + y;
  } else {
    z = x – y;
  }
  return z + z + z;
}
```

->

LLVM IR

use-def chains



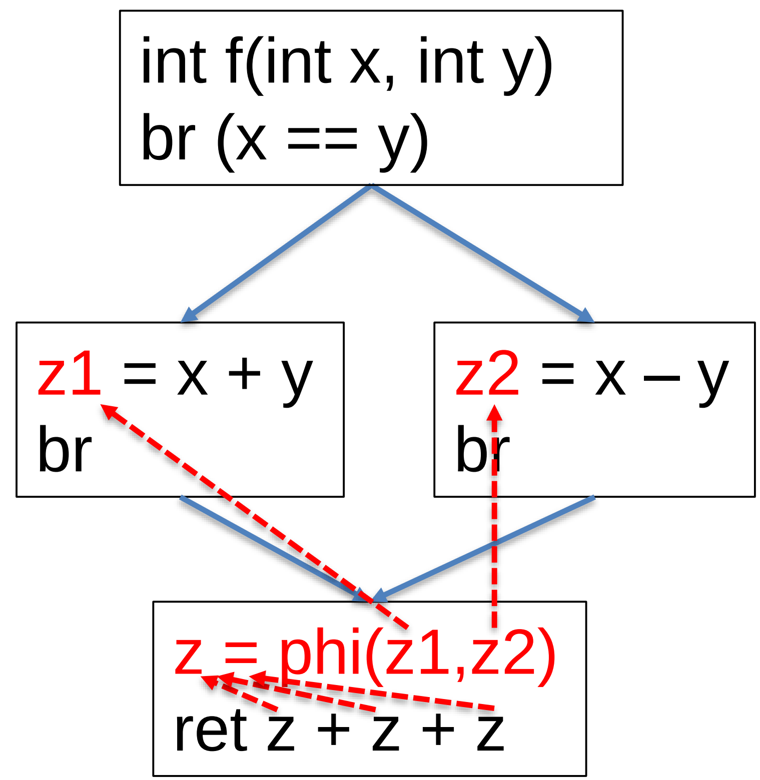
LLVM IR:
Static Single Assignment (SSA)

C

```
int f(int x, int y) {
  int z;
  if (x == y) {
    z = x + y;
  } else {
    z = x – y;
  }
  return z + z + z;
}
```

->

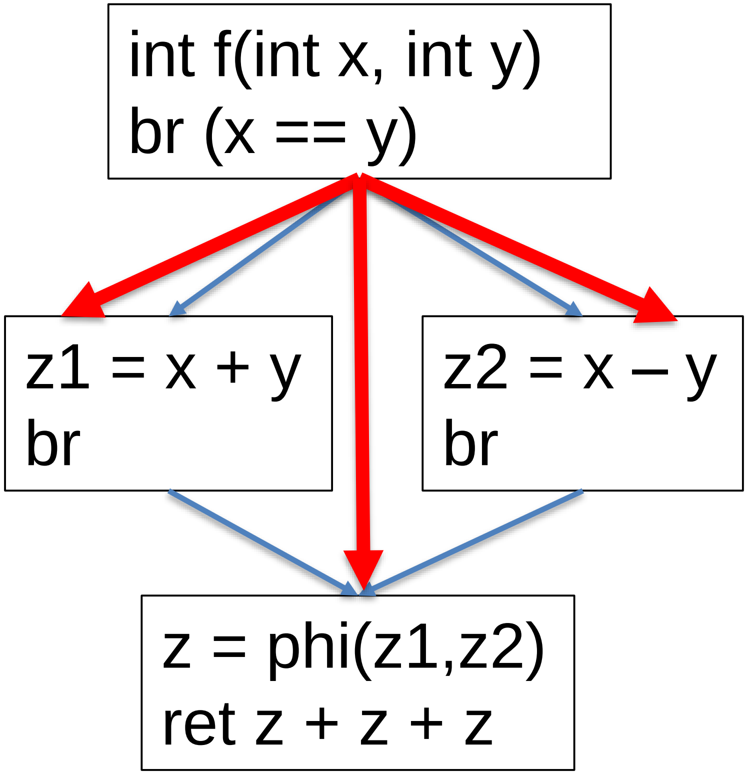
LLVM IR



LLVM IR:
Domination Tree

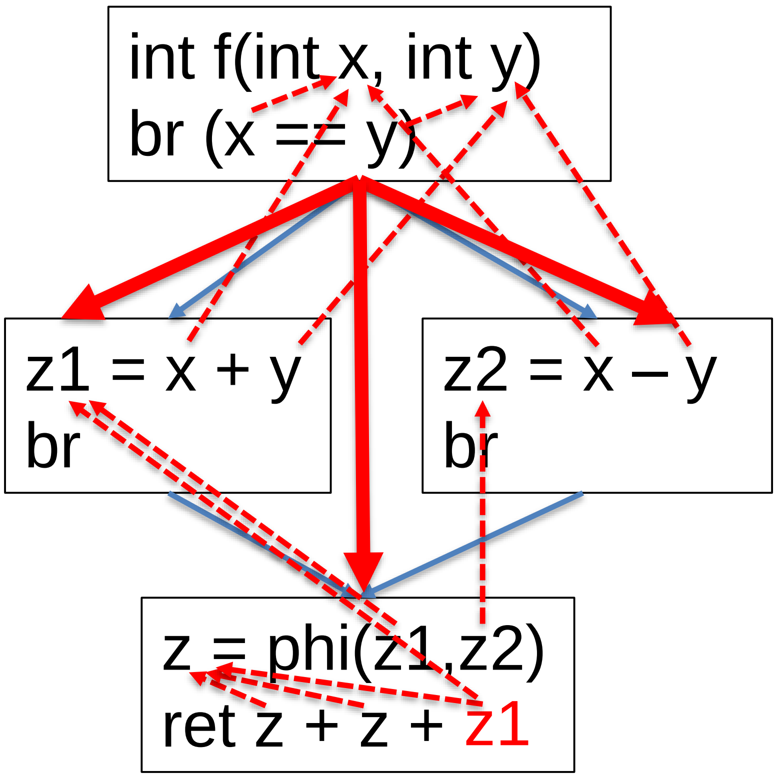
A *dominates* B if A is always visited before visiting B

Domination Tree



LLVM IR:
Checking use-after-def

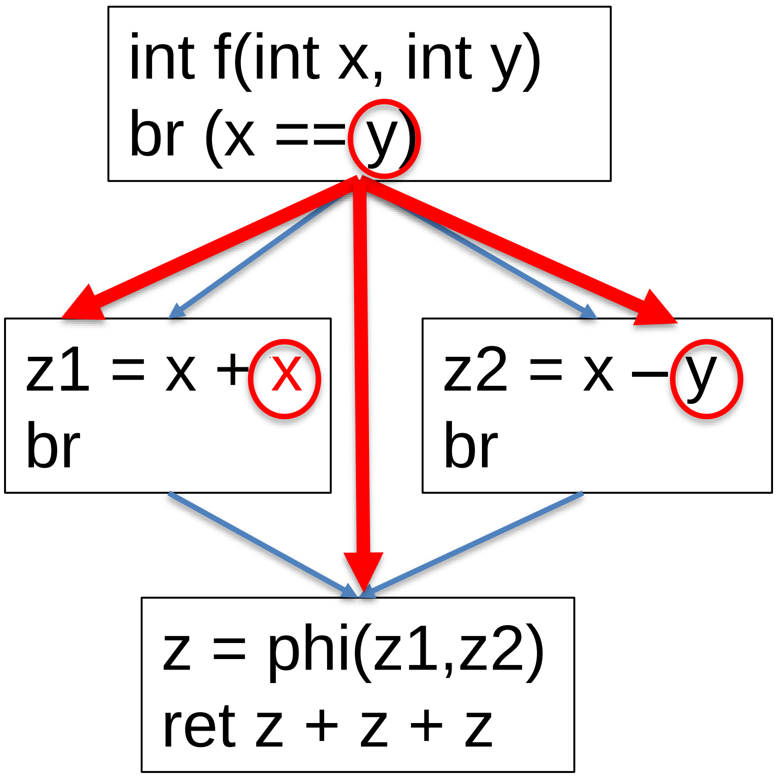
We can check whether
a variable is used
after it is defined
using use-def and dom tree.



An example optimization (Try)

Goal
- replace y by x when x == y

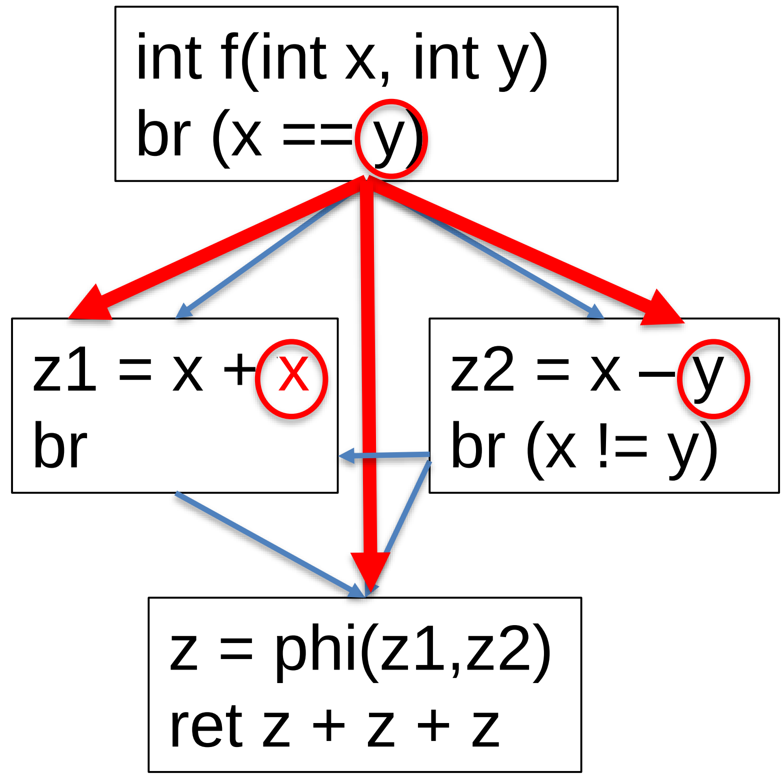
Algorithm
- Check if x == y dominates its left successor
- If not, do nothing
- If so, replace all uses of y that are dominated by the left successor



Counterexample

Goal
- replace y by x when x == y

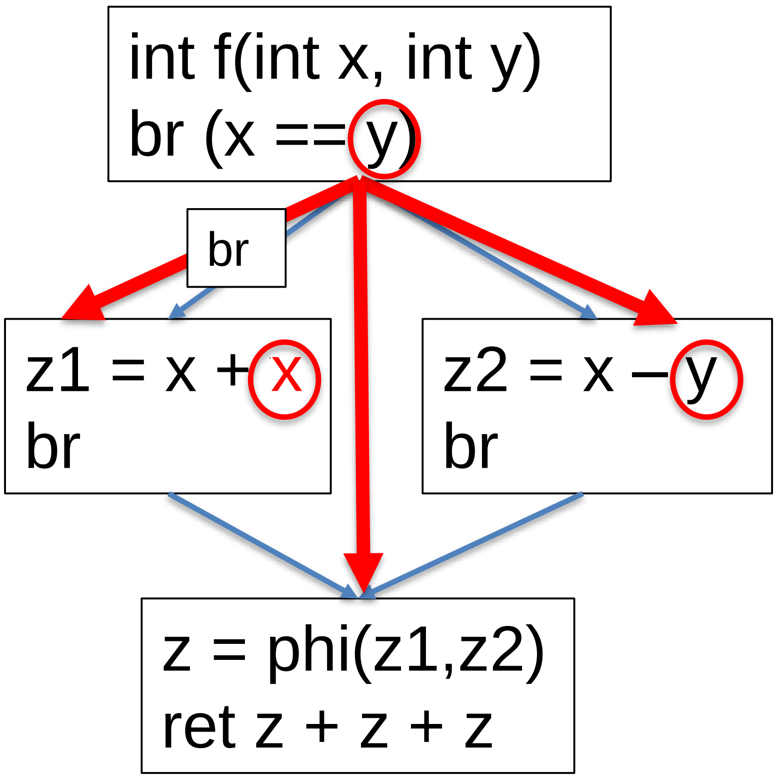
Algorithm
- Check if x == y dominates its left successor
- If not, do nothing
- If so, replace all uses of y that are dominated by the left successor



An example optimization (Correct)

Goal
- replace y by x when x == y

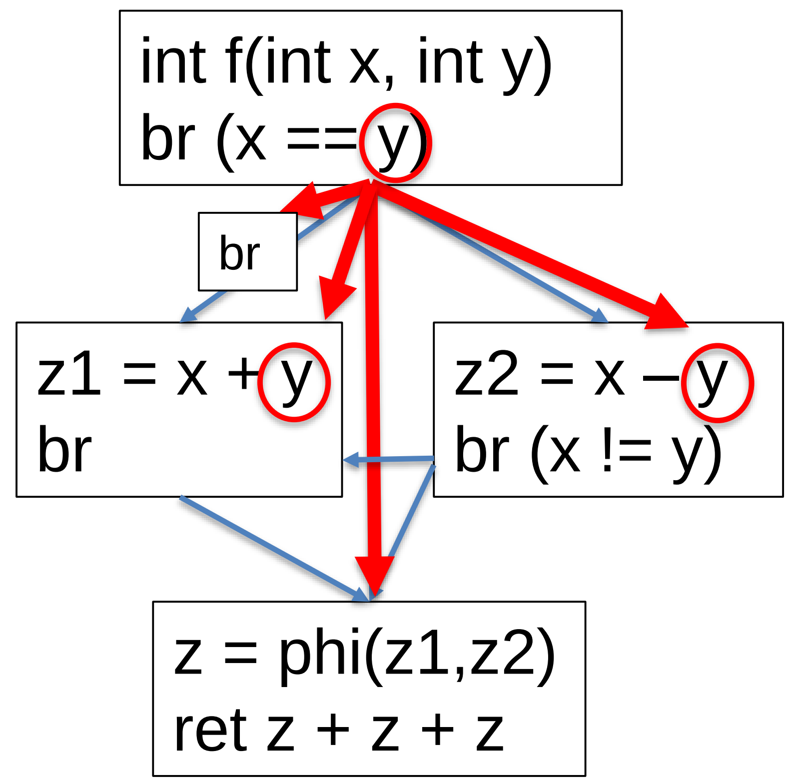
Algorithm
- Insert a dummy block between x == y and its left successor (called critical edge splitting)
- Replace all uses of y that are dominated by the dummy block



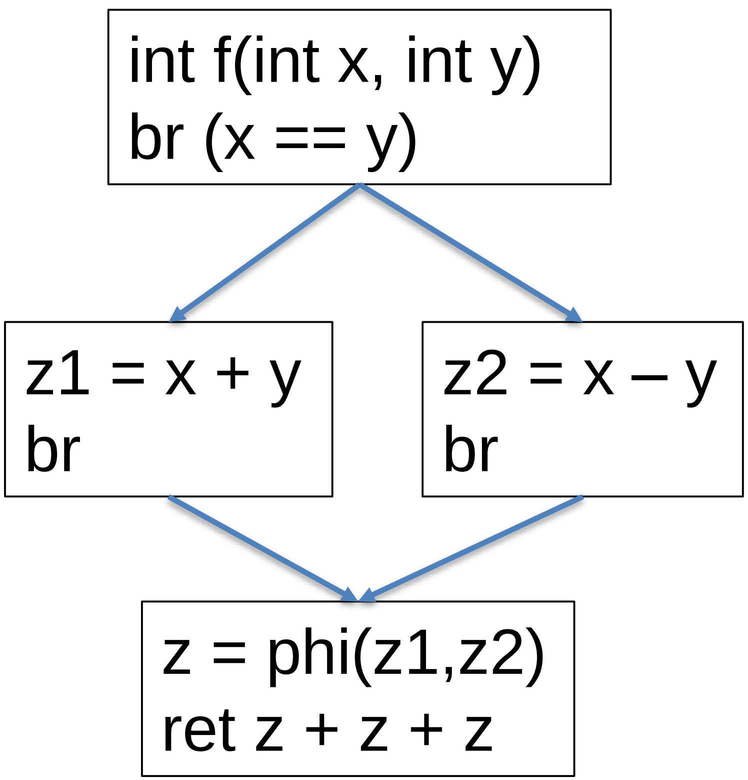
Counterexample Fixed

Goal
- replace y by x when x == y

Algorithm
- Check if x == y dominates its left successor
- If not, do nothing
- If so, replace all uses of y that are dominated by the left successor



LLVM Syntax




```
define i32 @f(i32 %x, i32 %y) {
entry:
   %0 = icmp eq i32 %x, %y
   br i1 %0, label %btrue, label %bfalse

btrue:
   %z1 = add nsw i32 %x, %y
   br label %end

bfalse:
   %z2 = sub nsw i32 %x, %y
   br label %end

end:
   %z = phi i32 [ %z1, %btrue ],
                [ %z2, %bfalse ]
   %1 = add nsw i32 %z, %z
   %2 = add nsw i32 %1, %z
   ret i32 %2
}
```

Registers

- Registers
  - Syntax: %0, %x, %gil, …
  - Can use an unbounded number of registers (ie, infinitely many)
  - Each register has a type: i1, i32, i64, f32, f64, i32*, …
  - Define: write a value (of the right type) into a register
  - Use: read the stored value from it
  - Q: Can a register be updated multiple times?

    A: Yes, in the presence of loop (ie, cycle in CFG)

```
i1 = 0
```

```
i2 = phi(i1,i3)
i3 = i2 + 1
```

  - Q: What’s difference between LLVM registers and C variables?

    A: C variables reside in memory (ie, have addresses, sharable)
    but registers do not (ie, have no addresses, unsharable)

Memory: Stack vs. Heap

```
// Stack Allocation
int f() {
    int x;
    x = 42;   //*(&x) = 42        =>
    g(&x);
    return x; // *(&x)
}
```

```
define i32 @f() {
entry:
    %xptr = alloca i32
    store i32 42, i32* %xptr
    call void @g(i32* %xptr)
    %0 = load i32, i32* %xptr
    ret i32 %0
}
```

```
// Heap Allocation
int f() {
    int* xptr;
    xptr = malloc(4);             =>
    *xptr = 42;
    g(xptr);
    int r = *xptr;
    free(xptr);
    return r;
}
```

```
define i32 @f() {
entry:
    %xptr = call i32* @malloc(i64 4)
    store i32 42, i32* %xptr
    call void @g(i32* %xptr)
    %0 = load i32, i32* %xptr
    call void @free(%xptr)
    ret i32 %0
}
```