SSA Form 1

# Static Single Assignment Form

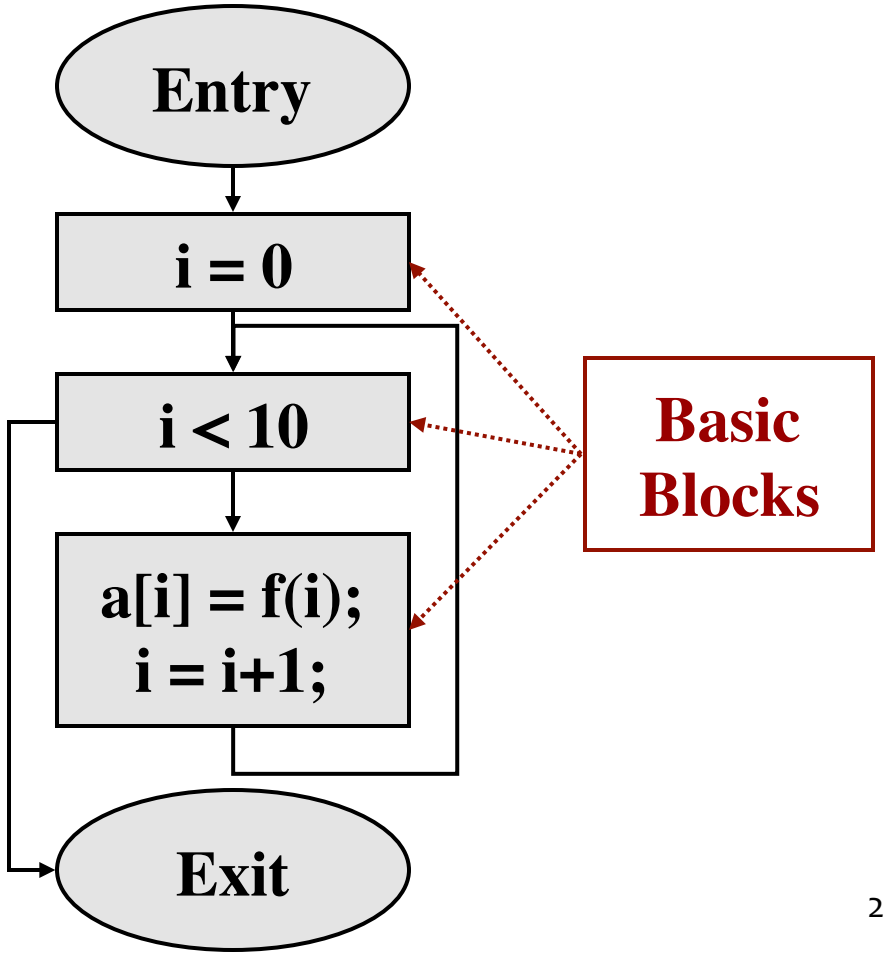
CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

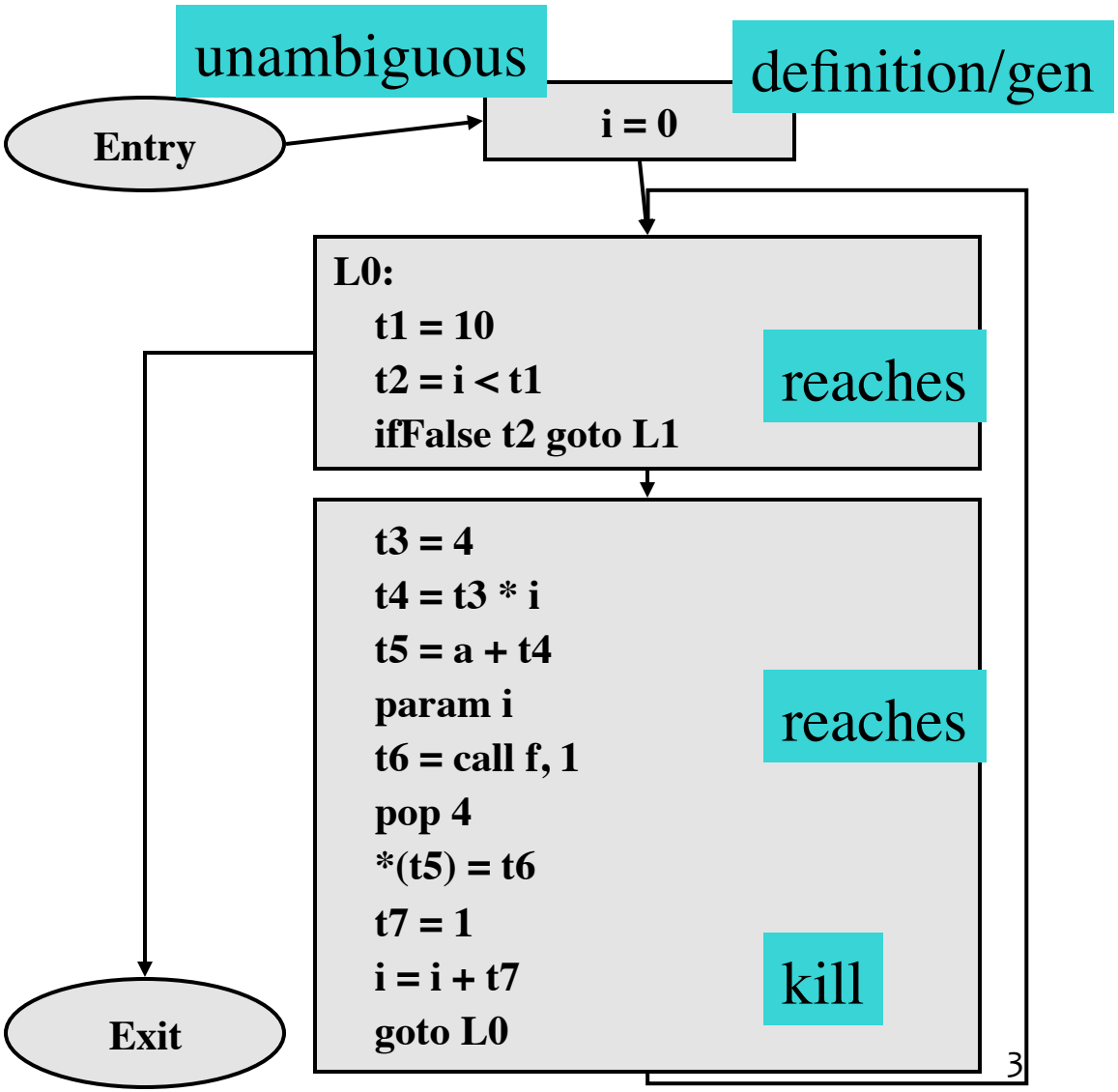
# Control Flow Graph (CFG)

```
int main() {
 extern int f(int);
 int i;
 int *a;
 for (i = 0;
        i < 10;
              i = i + 1)
    { a[i] = f(i); }
}
```



# Control Flow Graph in TAC

```
main:
   i = 0
L0:
   t1 = 10
   t2 = i < t1
   ifFalse t2 Goto L1
   t3 = 4
   t4 = t3 * i
   t5 = a + t4
   param i
   t6 = call f, 1
   pop 4
   *(t5) = t6
   t7 = 1
   i = i + t7
   goto L0
L1:
   return
```



# SSA Form

- *def-use* chains keep track of where variables were defined and where they were used
- Consider the case where each variable has only one definition in the intermediate representation
- One static definition, accessed many times
- Static Single Assignment Form (SSA)

# SSA Form

- SSA is useful because
  - Dataflow analysis and optimization is simpler when each variable has only one definition
  - If a variable has N uses and M definitions (which use N+M instructions) it takes N*M to represent def-use chains
  - Complexity is the same for SSA but in practice it is usually linear in number of definitions
  - SSA simplifies the register interference graph

# SSA Form

- Original Program

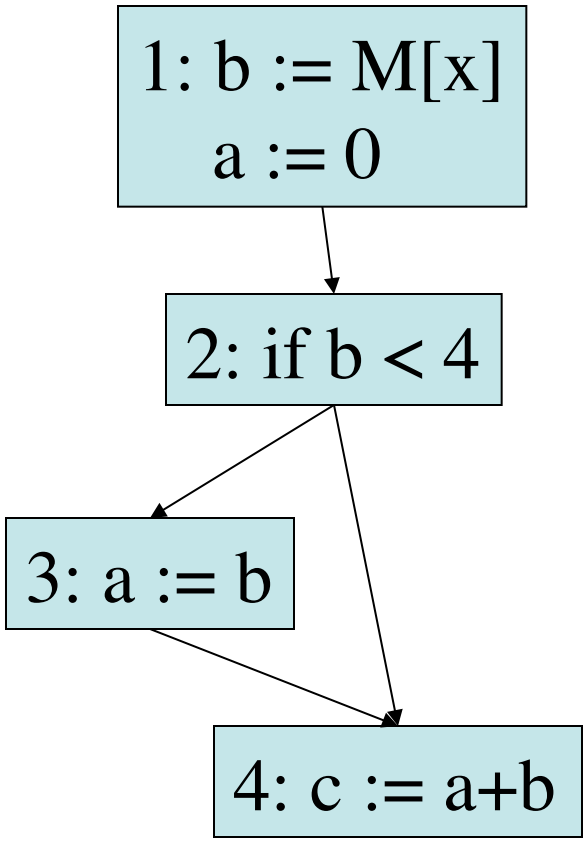
```
a := x + y
b := a - 1
a := y + b
b := x * 4
a := a + b
```

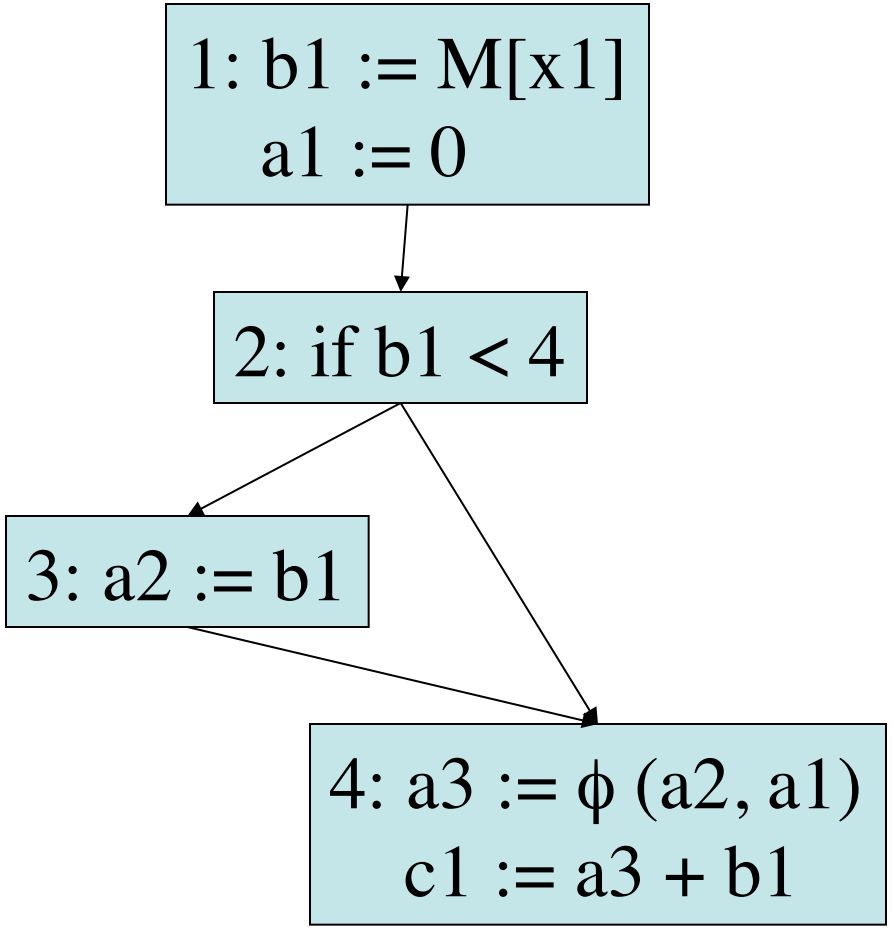
- SSA Form

```
a1 := x + y
b1 := a1 - 1
a2 := y + b1
b2 := x * 4
a3 := a2 + b2
```

*what about conditional branches?*

# SSA Form


1: b := M[x]
a := 0
2: if b < 4
3: a := b
4: c := a+b


1: b1 := M[x1]
a1 := 0
2: if b1 < 4
3: a2 := b1
4: a3 := ϕ (a2, a1)
c1 := a3 + b1

# Edge-split SSA Form

Unique Successor & Unique Predecessor