

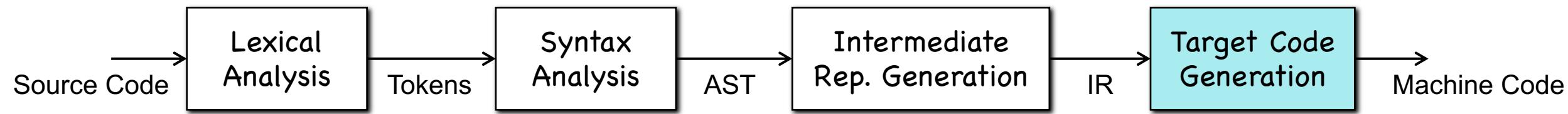
Chapter 8

Target Code Generation

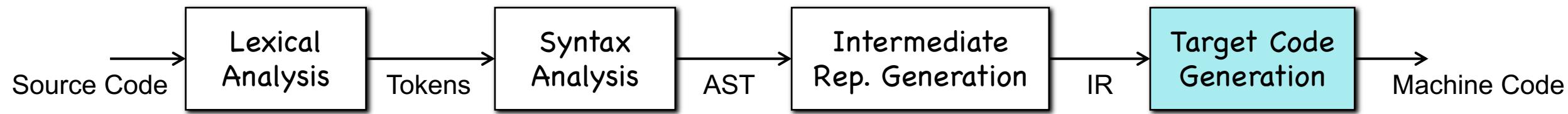
Qingkai Shi

qingkaishi@nju.edu.cn

Target Code Generation



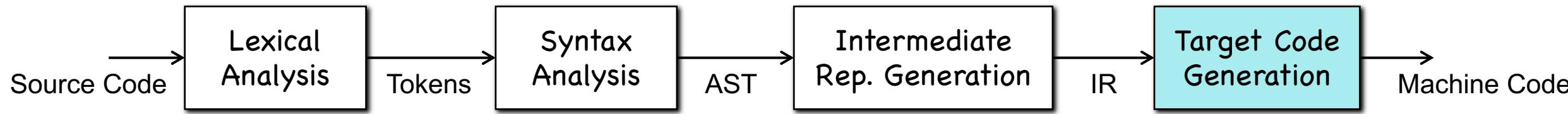
Target Code Generation



- **Code Generation/Target Code Model**
/Memory Allocation

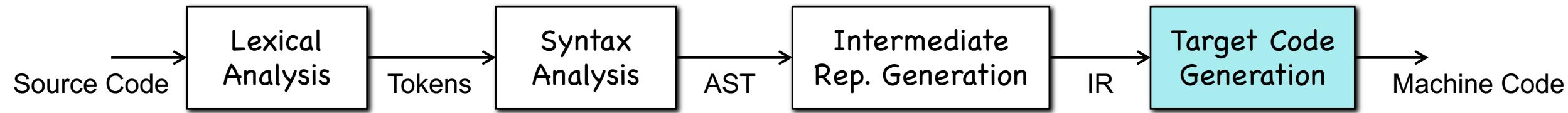


Target Code Generation



- **Code Generation/Target Code Model**
/Memory Allocation
- **Gen Better Code/Preliminaries**
/Local Optimization
/Register Allocation

Target Code Generation



- **Code Generation/Target Code Model**

/Memory Allocation

```
LD R0, a      // R0 = a
ADD R0, R0, #1 // R0 = R0 + 1
ST a, R0      // a = R0
```

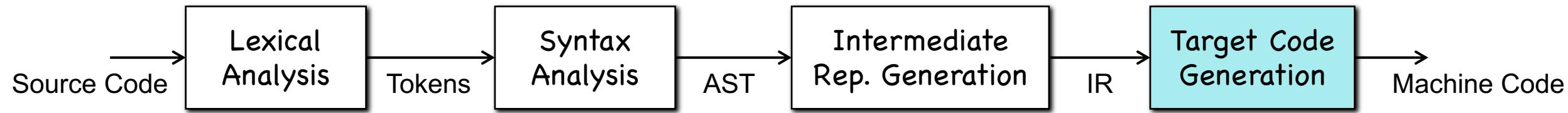
- **Gen Better Code/Preliminaries**

/Local Optimization

Possible target code for $a = a + 1$

/Register Allocation

Target Code Generation



- **Code Generation/Target Code Model**

/Memory Allocation

```

LD  R0, a      // R0 = a
ADD R0, R0, #1 // R0 = R0 + 1
ST  a, R0      // a = R0
  
```

- **Gen Better Code/Preliminaries**

/Local Optimization

Possible target code for $a = a + 1$

/Register Allocation

How about "INC a" ??

PART I: Target Code Generation

Recap: Three-Address Code

- What is three-address code? What does “address” mean?
-

Three-address code of

do i = i + 1; while (a[i + 2] < v);

```
L:   t1 = i + 1
      i = t1
      t2 = i + 2
      t3 = a [ t2 ]
      if t3 < v goto L
```

Symbolic Labels

```
100:  t1 = i + 1
    101: i = t1
    102: t2 = i + 2
    103: t3 = a [ t2 ]
    104: if t3 < v goto 100
```

Numeric Labels

Target Machine

- Memory (Heap/Stack/...). Each byte has an address.
- n Registers: R0, R1, ..., Rn-1. Each four bytes.

Target Machine

- Memory (Heap/Stack/...). Each byte has an address.
- n Registers: R0, R1, ..., Rn-1. Each four bytes.
- Load/Store/Calculation/Jump/... (Like x86 assembly)

Target Machine

- Memory (Heap/Stack/...). Each byte has an address.
 - n Registers: R0, R1, ..., Rn-1. Each four bytes.
 - Load/Store/Calculation/Jump/... (Like x86 assembly)
 - Load/Store
 - LD $R0, \text{addr}$
 - LD $R0, R1$
 - LD $R0, \#500$
 - ST $\text{addr}, R0$
- (Each LD/ST loads/stores a 4-byte integer)

Target Machine

- Memory (Heap/Stack/...). Each byte has an address.
 - n Registers: R0, R1, ..., Rn-1. Each four bytes.
 - Load/Store/Calculation/Jump/... (Like x86 assembly)
-
- Load/Store
 - LD $R0, \text{addr}$
 - LD $R0, R1$
 - LD $R0, \#500$
 - ST $\text{addr}, R0$
 - Calculation
 - OP $\text{dst}, \text{src}_1, \text{src}_2$
 - e.g., SUB $R0, R1, R2$
- (Each LD/ST loads/stores a 4-byte integer)

Target Machine

- Memory (Heap/Stack/...). Each byte has an address.
 - n Registers: R0, R1, ..., Rn-1. Each four bytes.
 - Load/Store/Calculation/Jump/... (Like x86 assembly)
-
- Load/Store
 - LD $R0, \text{addr}$
 - LD $R0, R1$
 - LD $R0, \#500$
 - ST $\text{addr}, R0$
 - Calculation
 - OP $\text{dst}, \text{src}_1, \text{src}_2$
 - e.g., SUB $R0, R1, R2$
 - Jump
 - BR L
 - Bcond R, L
 - e.g., BLTZ R, L
- (Each LD/ST loads/stores a 4-byte integer)

Target Machine

- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1
```

$$x = y - z$$

Target Machine

- Load/Store/Calculation/Jump/...

LD R1, y	// R1 = y
LD R2, z	// R2 = z
SUB R1, R1, R2	// R1 = R1 - R2
ST x, R1	// x = R1

$$x = y - z$$

Target Machine

- Load/Store/Calculation/Jump/...

LD R1, y	// R1 = y
LD R2, z	// R2 = z
SUB R1, R1, R2	// R1 = R1 - R2
ST x, R1	// x = R1

$$x = y - z$$

Target Machine

- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1         // x = R1
```

$$x = y - z$$

Target Machine

- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1
```

$$x = y - z$$

Target Machine

- Load/Store/Calculation/Jump/...

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1
```

$$x = y - z$$

Target Machine

- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1
```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

Target Machine

- Load/Store/Calculation/Jump/...

```

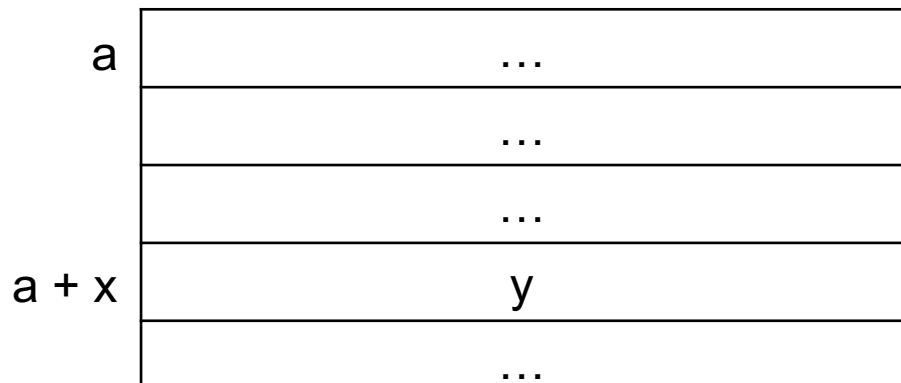
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Memory

Target Machine

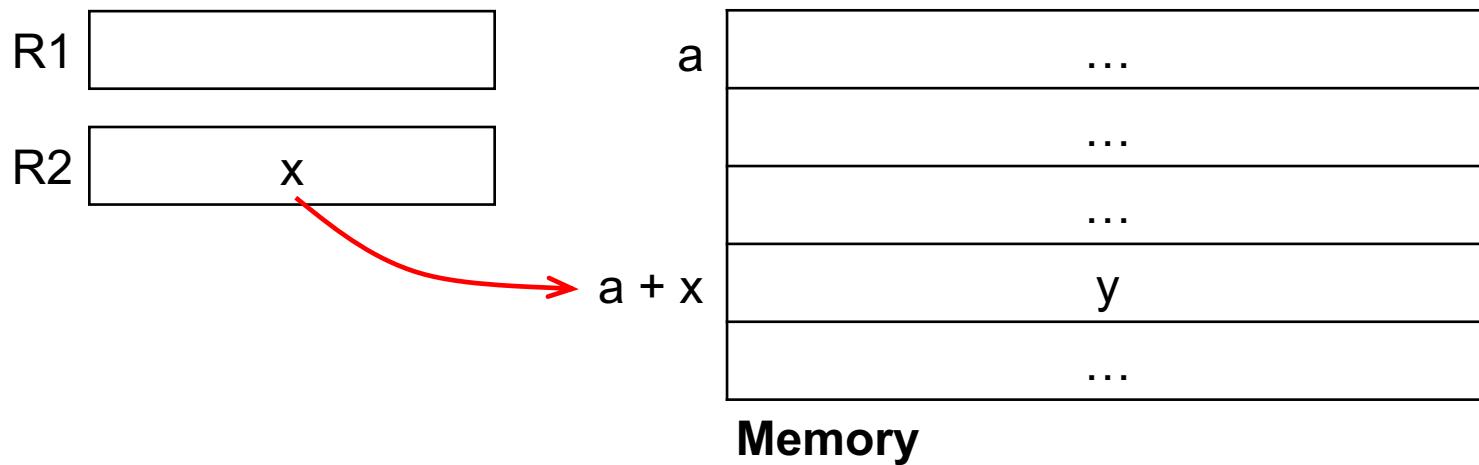
- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1
```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Target Machine

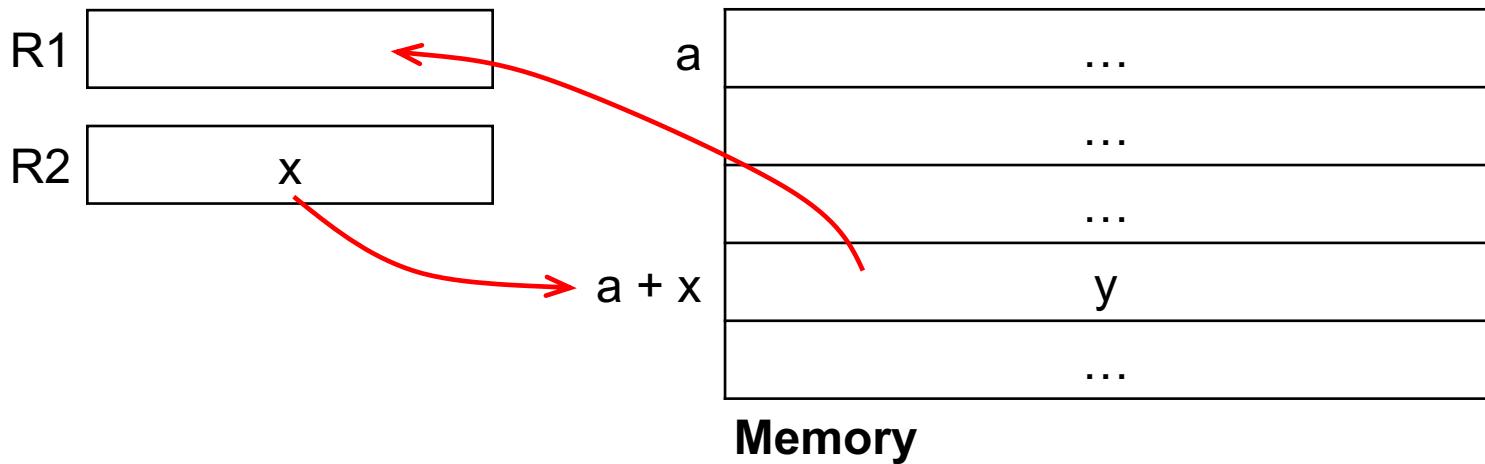
- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1
```

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

$$x = y - z$$



Target Machine

- Load/Store/Calculation/Jump/...

```

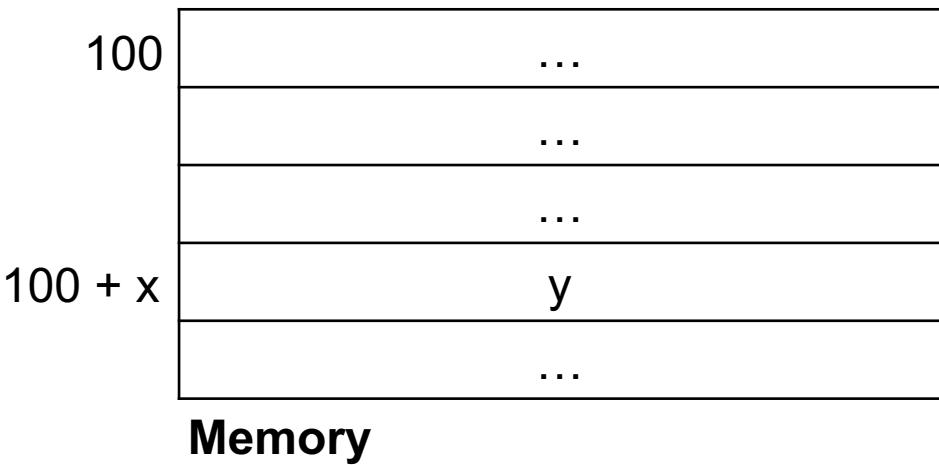
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Target Machine

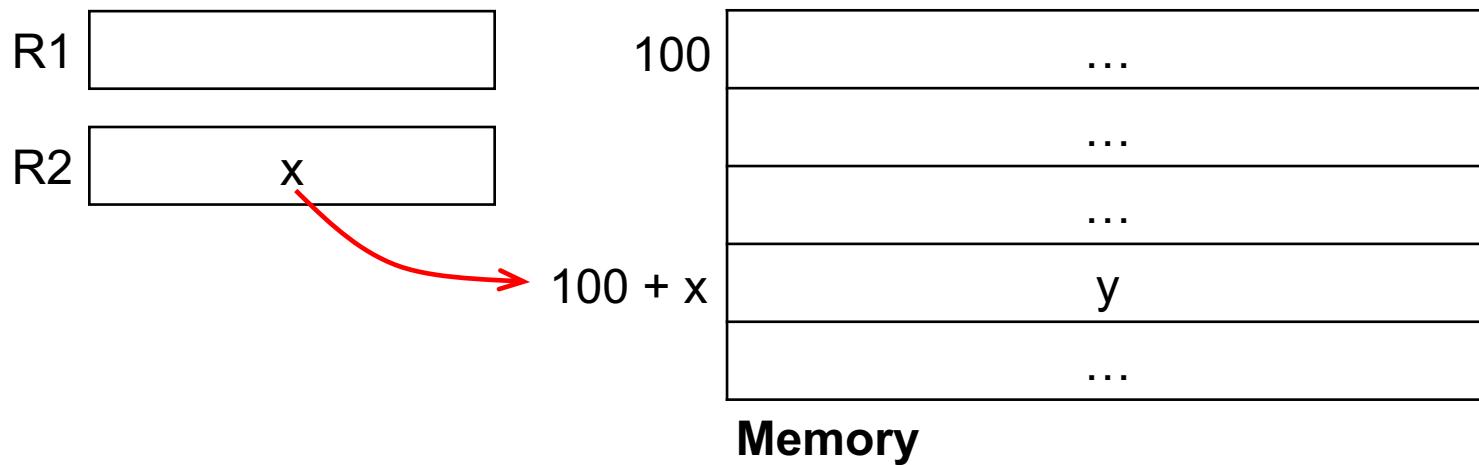
- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1
```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Target Machine

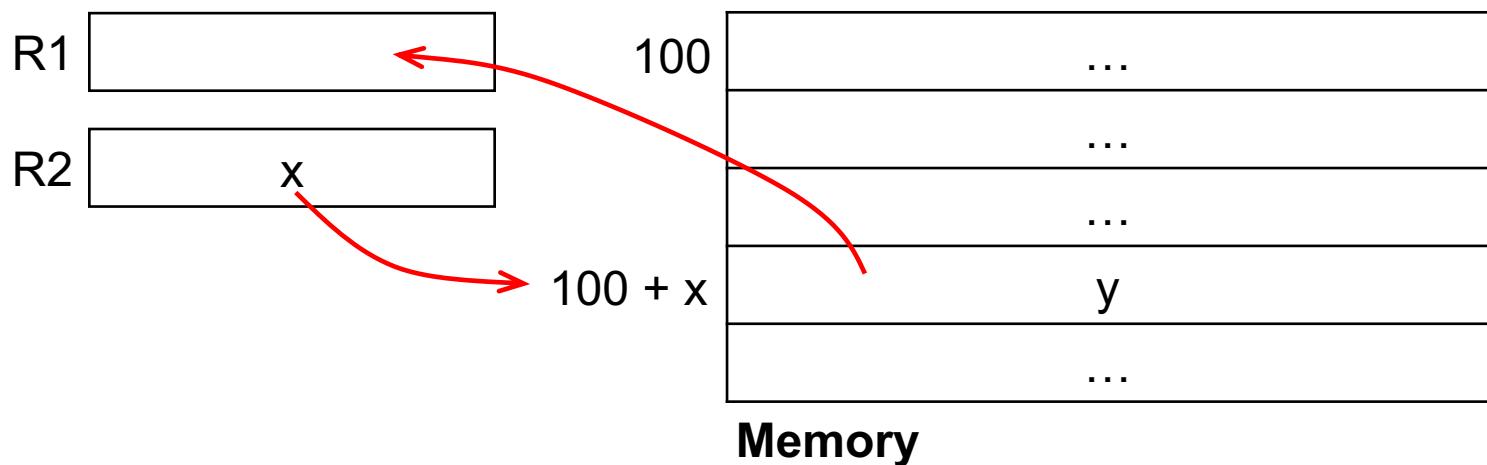
- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1
```

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

$$x = y - z$$



Target Machine

- Load/Store/Calculation/Jump/...

```

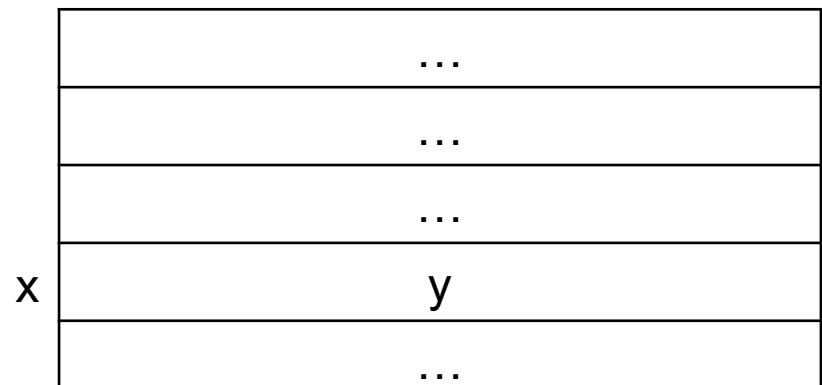
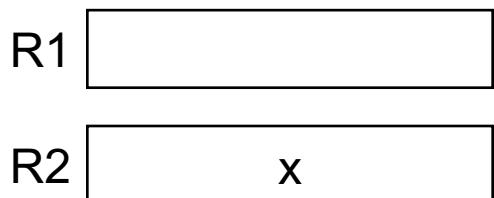
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Memory

Target Machine

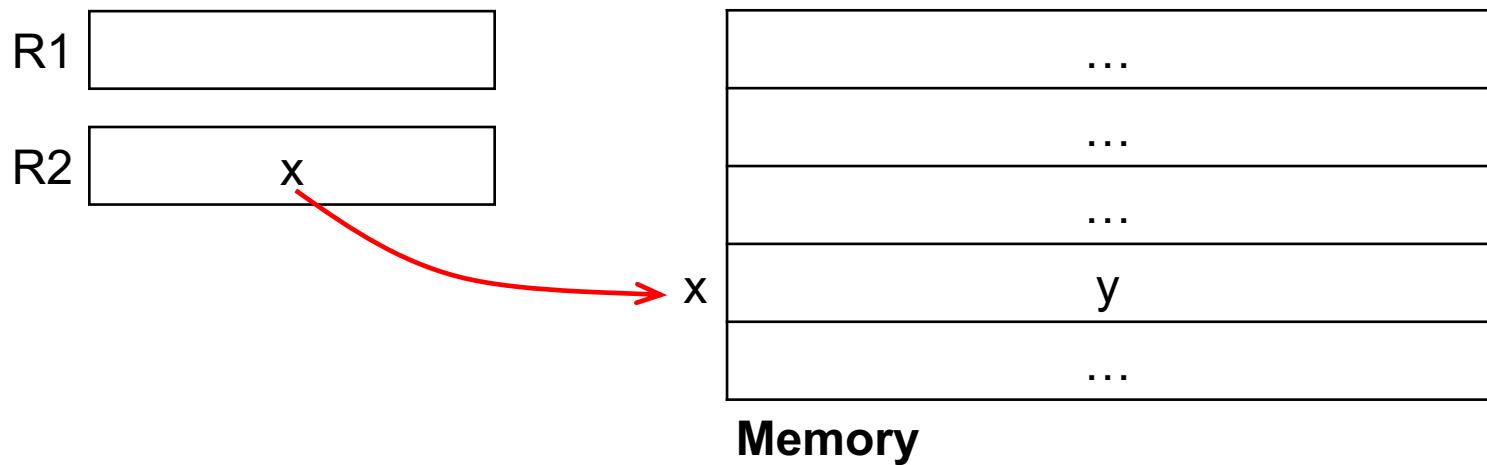
- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1
```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Target Machine

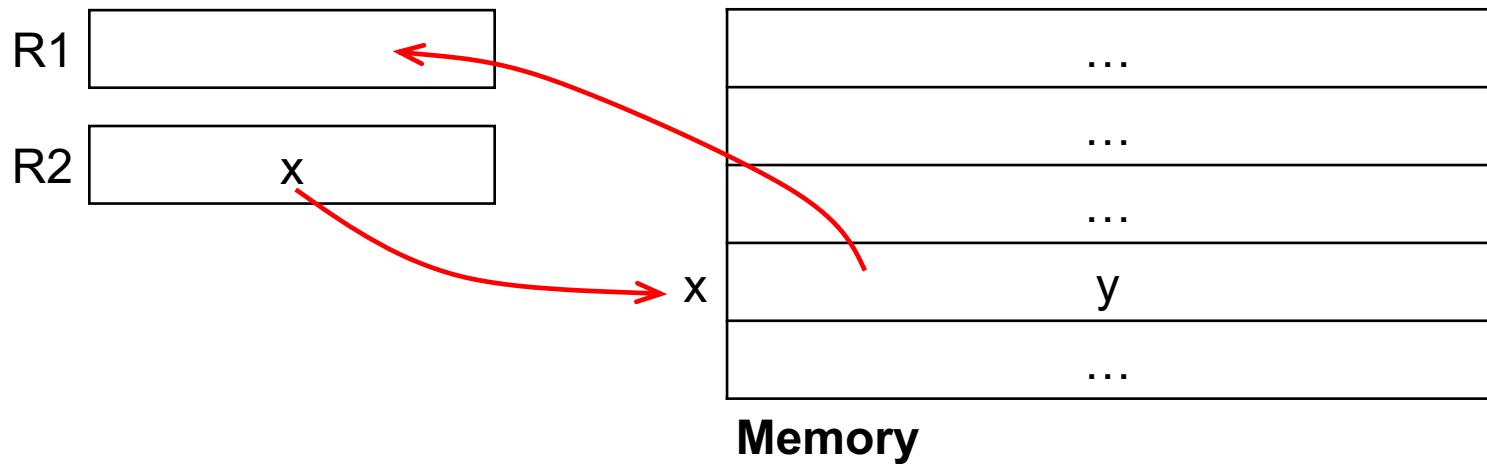
- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1
```

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

$$x = y - z$$



Target Machine

- Load/Store/Calculation/Jump/...

```

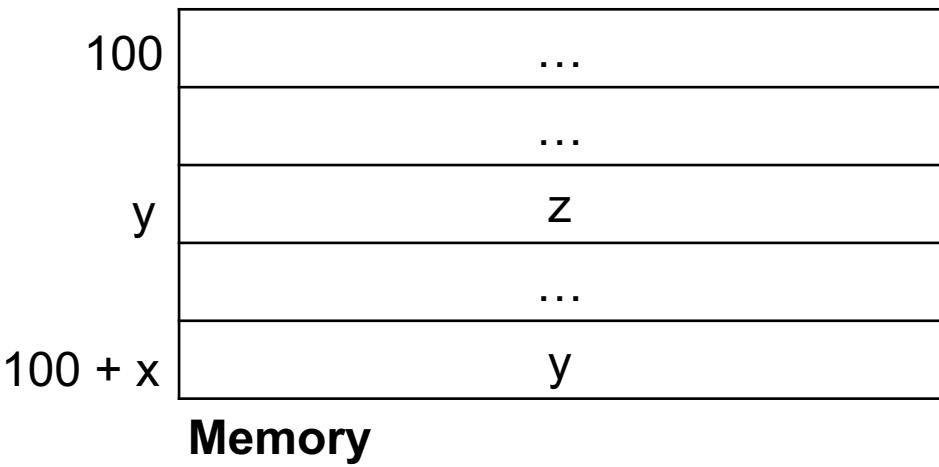
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Target Machine

- Load/Store/Calculation/Jump/...

```

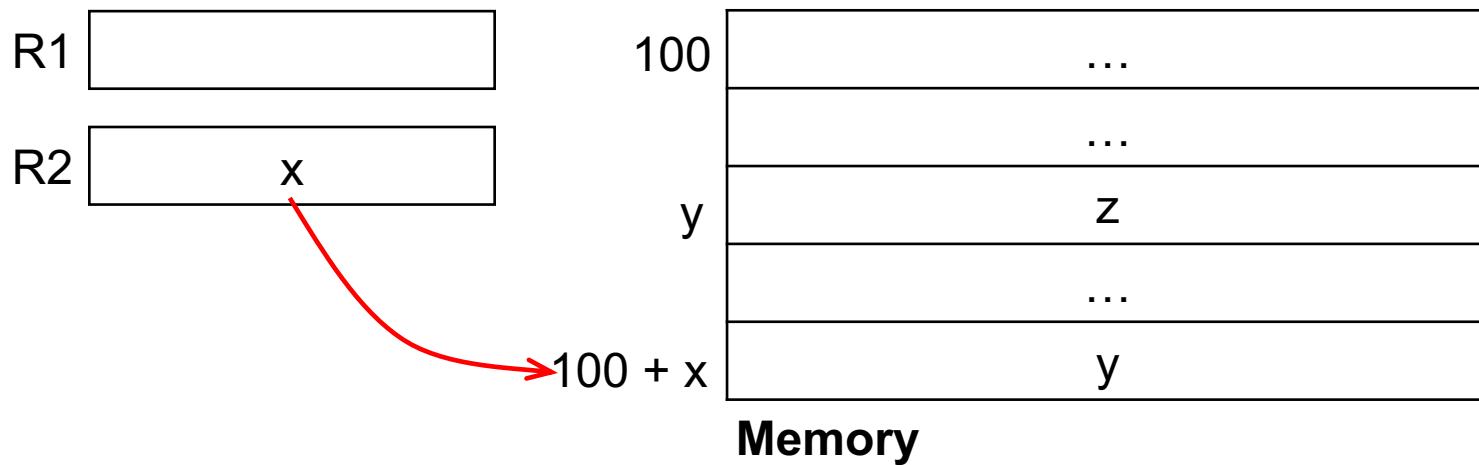
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Target Machine

- Load/Store/Calculation/Jump/...

```

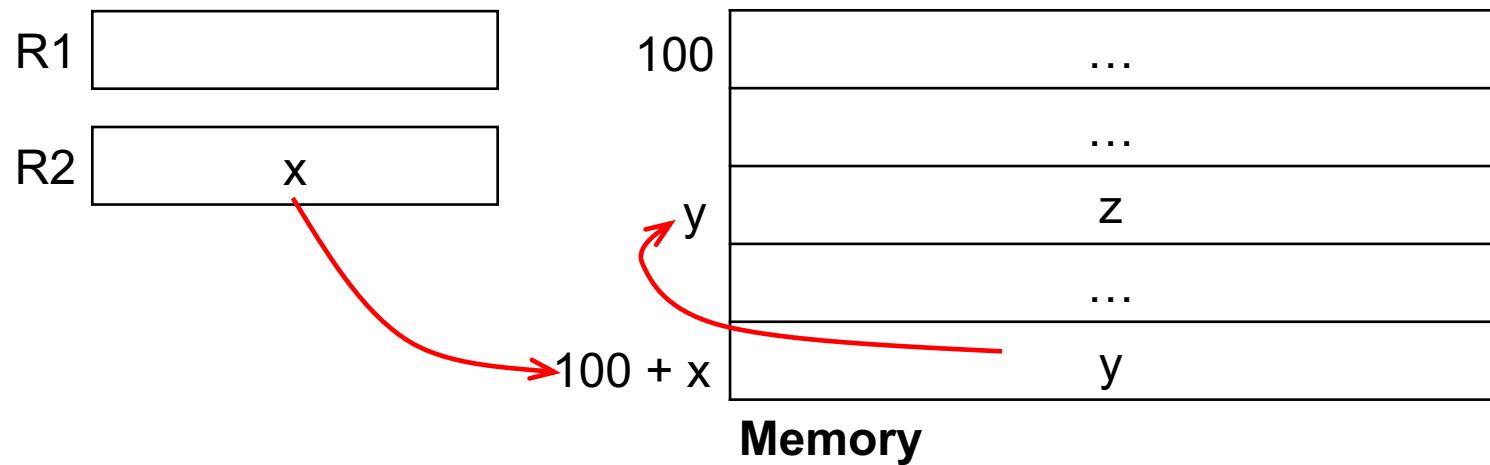
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Target Machine

- Load/Store/Calculation/Jump/...

```

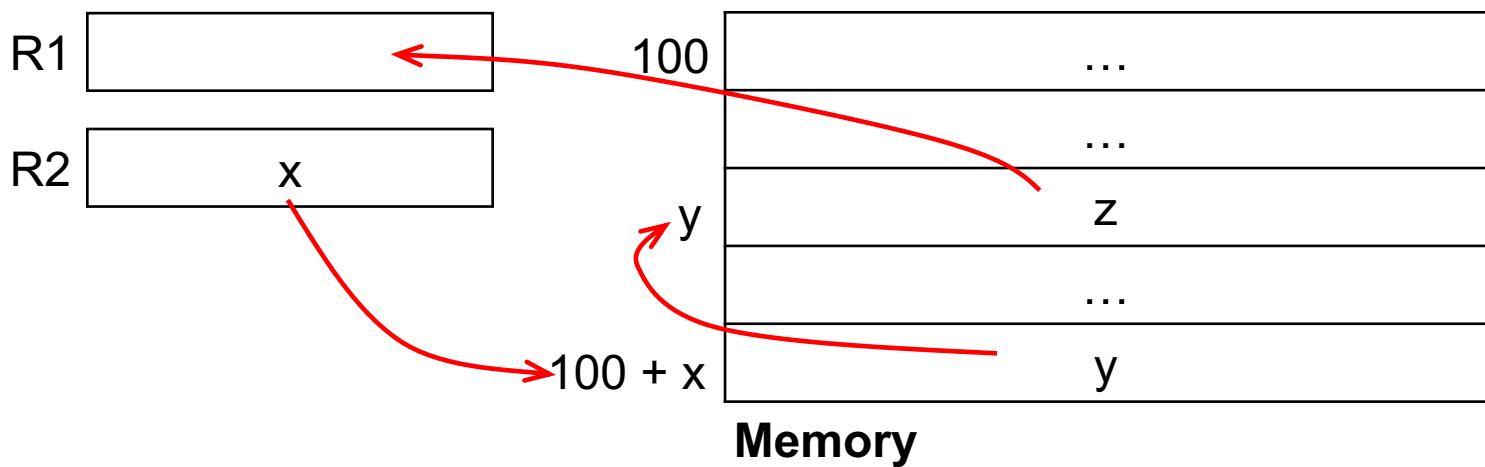
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

$$x = y - z$$



Target Machine

- Load/Store/Calculation/Jump/...

```

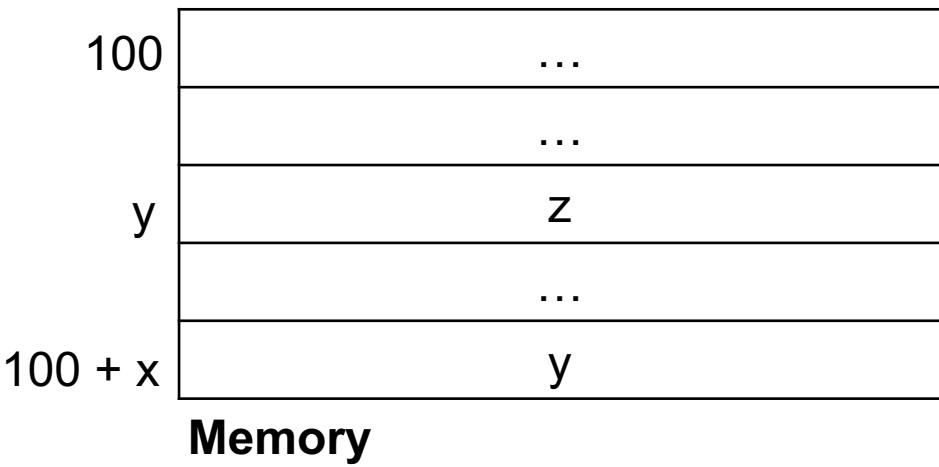
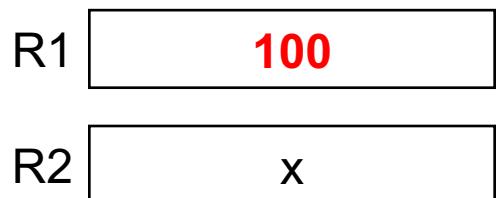
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100



Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

```

LD  R1, i          // R1 = i
MUL R1, R1, 4      // R1 = R1 * 4
LD   R2, a(R1)     // R2 = contents(a + contents(R1))
ST   b, R2         // b = R2

```

$$b = a[i] \text{ (a is an int array)}$$

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

```

LD  R1, i          // R1 = i
MUL R1, R1, 4      // R1 = R1 * 4
LD   R2, a(R1)     // R2 = contents(a + contents(R1))
ST   b, R2         // b = R2

```

$$b = a[i] \text{ (a is an int array)}$$

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

```

LD  R1, i          // R1 = i
MUL R1, R1, 4      // R1 = R1 * 4
LD  R2, a(R1)       // R2 = contents(a + contents(R1))
ST   b, R2          // b = R2

```

$$b = a[i] \text{ (a is an int array)}$$

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

```

LD  R1, i          // R1 = i
MUL R1, R1, 4      // R1 = R1 * 4
LD  R2, a(R1)       // R2 = contents(a + contents(R1))
ST   b, R2          // b = R2

```

$$b = a[i] \text{ (a is an int array)}$$

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

```

LD  R1, i          // R1 = i
MUL R1, R1, 4      // R1 = R1 * 4
LD   R2, a(R1)     // R2 = contents(a + contents(R1))
ST   b, R2         // b = R2

```

$$b = a[i] \text{ (a is an int array)}$$

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

x = y - z

```

LD  R1, x          // R1 = x
LD  R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M

```

if x < y goto M

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

$x = y - z$

```

LD  R1, x          // R1 = x
LD  R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M

```

if $x < y$ goto M

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

$x = y - z$

```

LD  R1, x          // R1 = x
LD  R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M

```

if $x < y$ goto M

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

- Addressing Modes
 - LD R1, a(R2)
 - LD R1, 100(R2)
 - LD R1, *R2
 - LD R1, *100(R2)
 - LD R1, #100

$x = y - z$

```

LD  R1, x          // R1 = x
LD  R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M

```

if $x < y$ goto M

Target Machine

- Load/Store/Calculation/Jump/...

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

$$x = y - z$$

```

LD  R1, x          // R1 = x
LD  R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M

```

if x < y goto M

Target Machine

- Load/Store/Calculation/Jump/...

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1
```

$$x = y - z$$

- Addressing Modes

- LD R1, a(R2)
- LD R1, 100(R2)
- LD R1, *R2
- LD R1, *100(R2)
- LD R1, #100

```
y = *q
q = q + 4
*p = y
p = p + 4
```

Try!

Question?

How does a machine understand
x, y, z, and M?

```
LD R1, y          // R1 = y
LD R2, z          // R2 = z
SUB R1, R1, R2   // R1 = R1 - R2
ST x, R1          // x = R1
```

$$x = y - z$$

```
LD R1, x          // R1 = x
LD R2, y          // R2 = y
SUB R1, R1, R2   // R1 = R1 - R2
BLTZ R1, M        // if R1 < 0 jump to M
```

if x < y goto M

Question?

How does a machine understand
x, y, z, and M?

Each variable or label corresponds
to a memory address

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

$x = y - z$

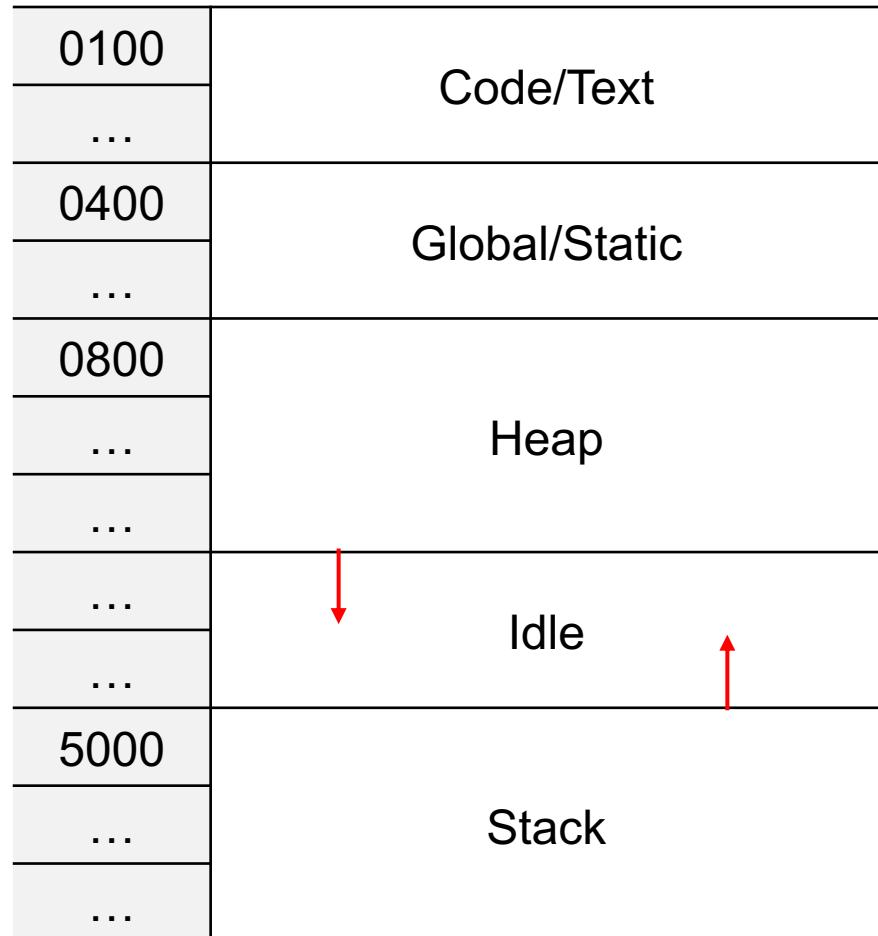
```

LD  R1, x          // R1 = x
LD  R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M

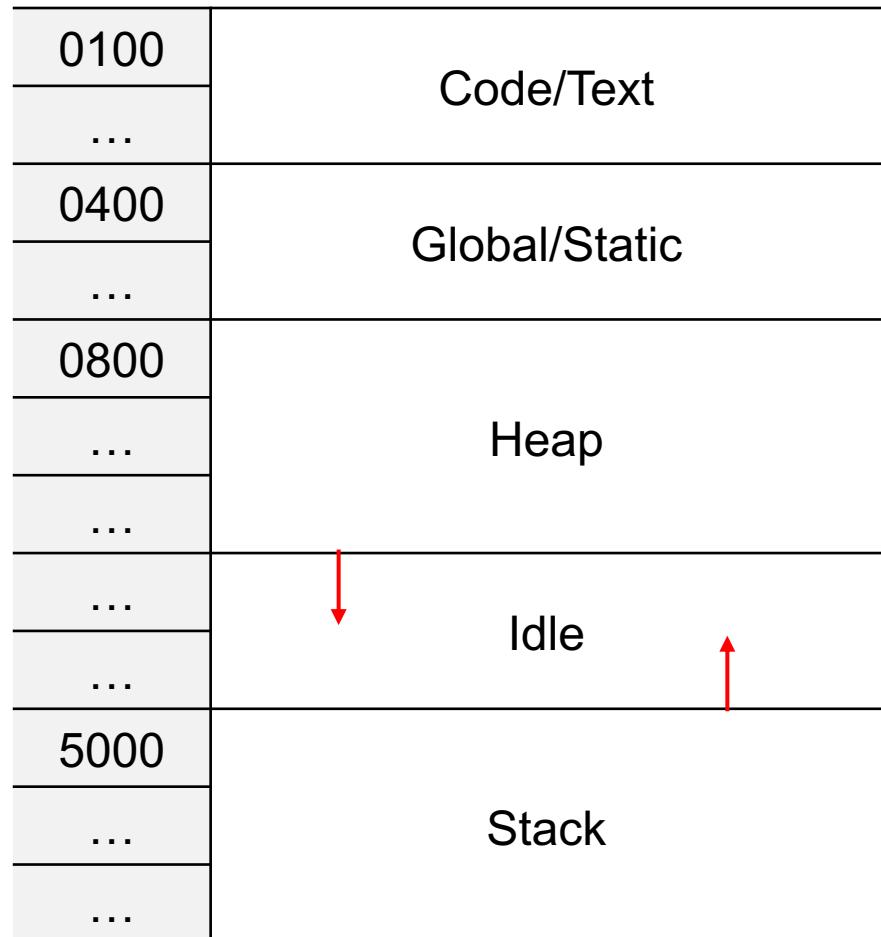
```

if $x < y$ goto M

Memory Structure



Memory Structure



```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

$$x = y - z$$

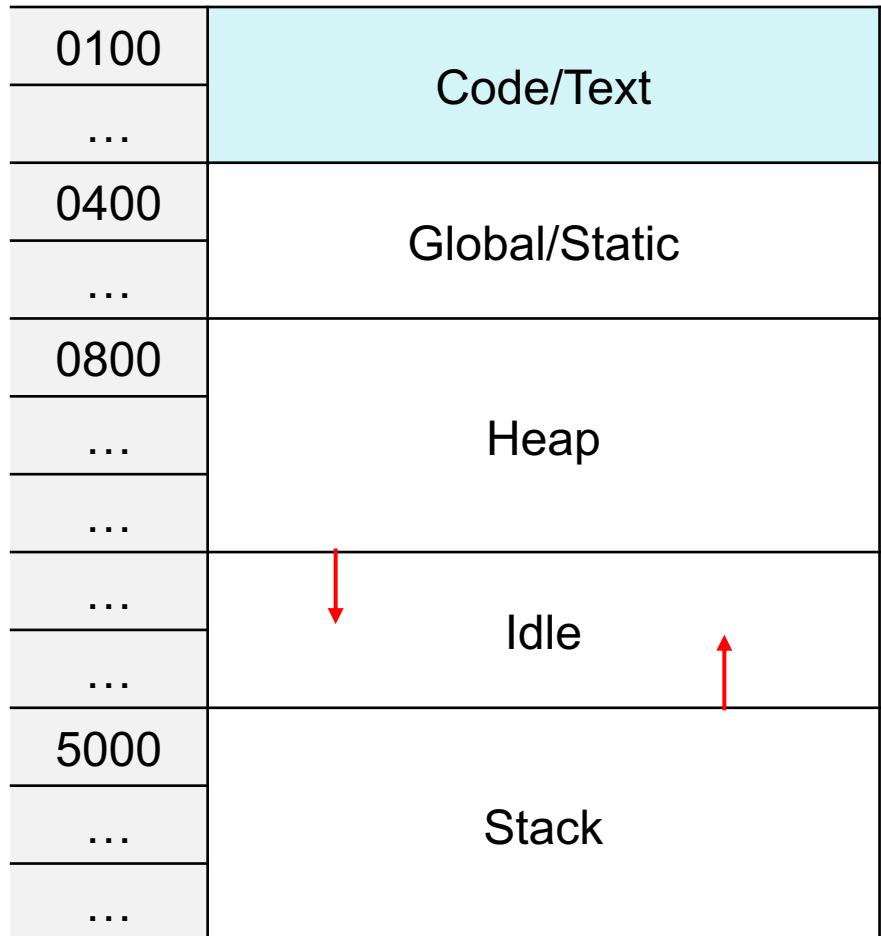
```

LD  R1, x          // R1 = x
LD  R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M

```

if $x < y$ goto M

Text Memory



```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST  x, R1          // x = R1

```

$x = y - z$

```

LD  R1, x          // R1 = x
LD  R2, y          // R2 = y
SUB R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M

```

if $x < y$ goto M

Text Memory

0100	LD R1 x
0104	LD R2 y
0108	SUB R1 R1 R2
0112	BLTZ R1 #0124
0116	...
0120	...
0124	...
...	...
	Static
	Heap
	Idle
	Stack

Load to memory

```

LD   R1, x           // R1 = x
LD   R2, y           // R2 = y
SUB  R1, R1, R2      // R1 = R1 - R2
BLTZ R1, M           // if R1 < 0 jump to M

```

if x < y goto M

Text Memory

0100	LD R1 x
0104	LD R2 y
0108	SUB R1 R1 R2
0112	BLTZ R1 #0124 ←
0116	...
0120	...
0124	...
...	...
	Static
	Heap
	Idle
	Stack

M is translated to a memory address

```

LD   R1, x      // R1 = x
LD   R2, y      // R2 = y
SUB  R1, R1, R2 // R1 = R1 - R2
BLTZ R1, M      // if R1 < 0 jump to M

```

if x < y goto M

Text Memory

0100	LD R1 x
0104	LD R2 y
0108	SUB R1 R1 R2
0112	BLTZ R1 #0124 ←
0116	...
0120	...
0124	...
...	...
	Static
	Heap
	Idle
	Stack

M is translated to a memory address

```

LD   R1, x      // R1 = x
LD   R2, y      // R2 = y
SUB  R1, R1, R2 // R1 = R1 - R2
BLTZ R1, M      // if R1 < 0 jump to M

```

if x < y goto M

Jump in X86

```
int x = 2;
int y = 4;

int main (int argc, char** argv) {
    if (argc > 0) { return x; }
    else return y;
}
```

- **clang -c hello.c –o hello.o**
- **objdump –D hello.o**

Jump

```
int x = 2;  
int y = 4;  
  
int main (int argc, char *argv)  
{  
    if (argc > 1)  
        x = atoi(argv[1]);  
    else  
        x = 2;  
    y = x + 2;  
    return 0;  
}
```

- clang -c hello.c
- objdump -D hello

Disassembly of section __TEXT,__text:

0000000000000000 <_main>:

0: 55	pushq	%rbp
1: 48 89 e5	movq	%rsp, %rbp
4: c7 45 fc 00 00 00 00	movl	\$0, -4(%rbp)
b: 89 7d f8	movl	%edi, -8(%rbp)
e: 48 89 75 f0	movq	%rsi, -16(%rbp)
12: 83 7d f8 00	cmpl	\$0, -8(%rbp)
16: 0f 8e 0e 00 00 00	jle	0x2a <_main+0x2a>
1c: 8b 05 00 00 00 00	movl	(%rip), %eax
22: 89 45 fc	movl	%eax, -4(%rbp)
25: e9 09 00 00 00	jmp	0x33 <_main+0x33>
2a: 8b 05 00 00 00 00	movl	(%rip), %eax
30: 89 45 fc	movl	%eax, -4(%rbp)
33: 8b 45 fc	movl	-4(%rbp), %eax
36: 5d	popq	%rbp
37: c3	retq	

Jump

```
int x = 2;  
int y = 4;  
  
int main (int  
          argc,  
          char *argv,  
          int  
          *envp)  
{  
    if (argc > 1)  
        y = atoi(argv[1]);  
    else  
        y = 4;  
    if (y < 0)  
        y = -y;  
    if (y > 10)  
        y = 10;  
    x = x + y;  
    if (x < 0)  
        x = -x;  
    if (x > 10)  
        x = 10;  
    printf("x = %d, y = %d\n", x, y);  
}
```

- clang -c hello.c
- objdump -D hello

Disassembly of section __TEXT,__text:

0000000000000000 <_main>:

0:	55	pushq	%rbp
1:	48 89 e5	movq	%rsp, %rbp
4:	c7 45 fc 00 00 00 00	movl	\$0, -4(%rbp)
b:	89 7d f8	movl	%edi, -8(%rbp)
e:	48 89 75 f0	movq	%rsi, -16(%rbp)
12:	83 7d f8 00	cmpl	\$0, -8(%rbp)
16:	0f 8e 0e 00 00 00	jle	0x2a <_main+0x2a>
1c:	8b 05 00 00 00 00	movl	(%rip), %eax
22:	89 45 fc	movl	%eax, -4(%rbp)
25:	e9 09 00 00 00	jmp	0x33 <_main+0x33>
2a:	8b 05 00 00 00 00	movl	(%rip), %eax
30:	89 45 fc	movl	%eax, -4(%rbp)
33:	8b 45 fc	movl	-4(%rbp), %eax
36:	5d	popq	%rbp
37:	c3	retq	

Jump

Disassembly of section __TEXT,__text:

	0000000000000000 <_main>:		
int x = 2;	0: 55	pushq	%rbp
int y = 4;	1: 48 89 e5	movq	%rsp, %rbp
	4: c7 45 fc 00 00 00 00	movl	\$0, -4(%rbp)
int main (int	b: 89 7d f8	movl	%edi, -8(%rbp)
if (arg	e: 48 89 75 f0	movq	%rsi, -16(%rbp)
else re	12: 83 7d f8 00	cmpl	\$0, -8(%rbp)
}	16: 0f 8e 0e 00 00 00	jle	0x2a <_main+0x2a>
	1c: 8b 05 00 00 00 00	movl	(%rip), %eax
• clang -c hello.	22: 89 45 fc	movl	%eax, -4(%rbp)
• objdump -D h	25: e9 09 00 00 00	jmp	0x33 <_main+0x33>
	2a: 8b 05 00 00 00 00	movl	(%rip), %eax
	30: 89 45 fc	movl	%eax, -4(%rbp)
	33: 8b 45 fc	movl	-4(%rbp), %eax
	36: 5d	popq	%rbp
	37: c3	retq	

Jump

```
int x = 2;
int y = 4;

int main (int argc, char *argv[])
{
    if (argc > 1)
        x = atoi(argv[1]);
    else
        x = 2;
    y = x + 2;
    return 0;
}
```

- **clang -c hello.c**
- **objdump -D hello.o**

Disassembly of section __TEXT,__text:

0000000000000000 <_main>:

0: 55	pushq %rbp
1: 48 89 e5	movq %rsp, %rbp
4: c7 45 fc 00 00 00 00	movl \$0, -4(%rbp)
b: 89 7d f8	movl %edi, -8(%rbp)
e: 48 89 75 f0	movq %rsi, -16(%rbp)
12: 83 7d f8 00	cmpl \$0, -8(%rbp)
16: 0f 8e 0e 00 00 00	jle 0x2a <_main+0x2a>
1c: 8b 05 00 00 00 00	movl (%rip), %eax
22: 89 45 fc	movl %eax, -4(%rbp)
25: e9 09 00 00 00	jmp 0x33 <_main+0x33>
2a: 8b 05 00 00 00 00	movl (%rip), %eax
30: 89 45 fc	movl %eax, -4(%rbp)
33: 8b 45 fc	movl -4(%rbp), %eax
36: 5d	popq %rbp
37: c3	retq

Jump

```
int x = 2;
int y = 4;

int main (int argc, char *argv[])
{
    if (argc > 1)
        x = atoi(argv[1]);
    else
        x = 2;
    y = 4;
    return 0;
}
```

- clang -c hello.c
- objdump -D hello

Disassembly of section __TEXT,__text:

0000000000000000 <_main>:

0: 55	pushq	%rbp
1: 48 89 e5	movq	%rsp, %rbp
4: c7 45 fc 00 00 00 00	movl	\$0, -4(%rbp)
b: 89 7d f8	movl	%edi, -8(%rbp)
e: 48 89 75 f0	movq	%rsi, -16(%rbp)
12: 83 7d f8 00	cmpl	\$0, -8(%rbp)
16: 0f 8e 0e 00 00 00	jle	0x2a <_main+0x2a>
1c: 8b 05 00 00 00 00	movl	(%rip), %eax
22: 89 45 fc	movl	%eax, -4(%rbp)
25: e9 09 00 00 00	jmp	0x33 <_main+0x33>
2a: 8b 05 00 00 00 00	movl	(%rip), %eax
30: 89 45 fc	movl	%eax, -4(%rbp)
33: 8b 45 fc	movl	-4(%rbp), %eax
36: 5d	popq	%rbp
37: c3	retq	

Jump

```
int x = 2;  
int y = 4;  
  
int main (int  
    if (arg  
        else re  
    }
```

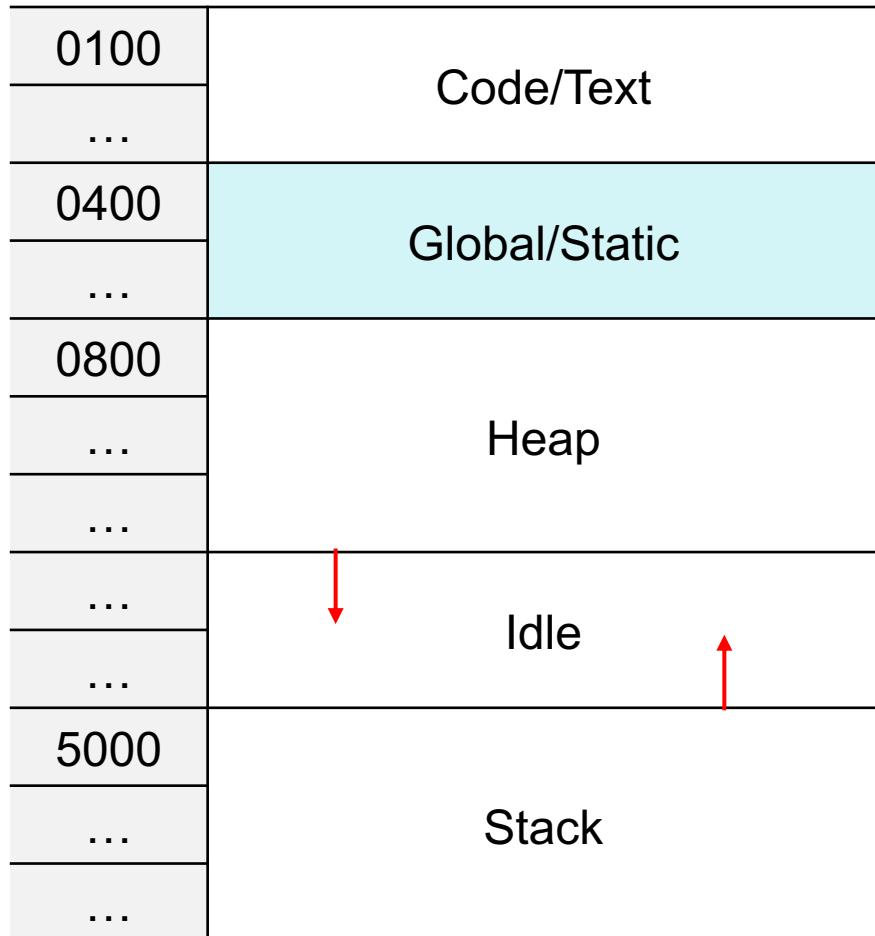
- clang -c hello.c
- objdump -D hello

Disassembly of section __TEXT,__text:

```
0000000000000000 <_main>:
```

0: 55	pushq	%rbp
1: 48 89 e5	movq	%rsp, %rbp
4: c7 45 fc 00 00 00 00	movl	\$0, -4(%rbp)
b: 89 7d f8	movl	%edi, -8(%rbp)
e: 48 89 75 f0	movq	%rsi, -16(%rbp)
12: 83 7d f8 00	cmpl	\$0, -8(%rbp)
16: 0f 8e 0e 00 00 00	jle	0x2a <_main+0x2a>
1c: 8b 05 00 00 00 00	movl	(%rip), %eax
22: 89 45 fc	movl	%eax, -4(%rbp)
25: e9 09 00 00 00	jmp	0x33 <_main+0x33>
2a: 8b 05 00 00 00 00	movl	(%rip), %eax
30: 89 45 fc	movl	%eax, -4(%rbp)
33: 8b 45 fc	movl	-4(%rbp), %eax
36: 5d	popq	%rbp
37: c3	retq	

Global Memory



Global Memory

0100	...	Text
0104	...	
0108	...	
0112	...	
...	...	
0400	2	Global
0404	3	
0408	...	
...	...	
...	...	Heap
...	...	Idle
...	...	Stack

```
int y = 2;  
int z = 3;  
  
void foo () {  
    int x;  
    x = y - z;  
}
```

Global Memory

0100	LD R1 *0400	Text
0104	LD R2 *0404	
0108	SUB R1 R1 R2	
0112	...	
...	...	
0400	2	Global
0404	3	
0408	...	
...	...	
...	Heap	
...	Idle	
...	Stack	

```

LD  R1, y           // R1 = y
LD  R2, z           // R2 = z
SUB R1, R1, R2      // R1 = R1 - R2
ST   x, R1          // x = R1

```

$x = y - z$

```

int y = 2;
int z = 3;

void foo () {
    int x;
    x = y - z;
}

```

Global Memory

0100	LD R1 *0400	Text
0104	LD R2 *0404	
0108	SUB R1 R1 R2	
0112	...	
...	...	
0400	2	Global
0404	3	
0408	...	
...	...	
...	Heap	
...	Idle	
...	Stack	

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$x = y - z$

```

int y = 2;
int z = 3;

void foo () {
    int x;
    x = y - z;
}

```

Global Memory

0100	LD R1 *0400	Text
0104	LD R2 *0404	
0108	SUB R1 R1 R2	
0112	...	
...	...	
0400	2	Global
0404	3	
0408	...	
...	...	
...	Heap	
...	Idle	
...	Stack	

```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST   x, R1         // x = R1

```

$x = y - z$

```

int y = 2;
int z = 3;

void foo () {
    int x;
    x = y - z;
}

```

Globals in x86

```
int x = 2;
int y = 4;

int main (int argc, char** argv) {
    if (argc > 0) { return x; }
    else return y;
}
```

- **clang -c hello.c –o hello.o**
- **objdump –D hello.o**

Globals in x86

```
int x = 2;  
int y = 4;  
  
int main (int argc, char** argv) {  
    if (argc > 0) { return x; }  
    else return y;  
}
```

- **clang -c hello.c –o hello.o**
- **objdump –D hello.o**

Disassembly of section __DATA,__data:

0000000000000038 <_x>:

38: 02 00
3a: 00 00

000000000000003c <_y>:

3c: 04 00
3e: 00 00

Globals in x86

```
int x = 2;  
int y = 4;  
  
int main (int argc, char** argv) {  
    if (argc > 0) { return x; }  
    else return y;  
}
```

- **clang -c hello.c –o hello.o**
- **objdump –D hello.o**

Disassembly of section __DATA,__data:

```
0000000000000038 <_x>:  
 38: 02 00  
 3a: 00 00
```

```
000000000000003c <_y>:  
 3c: 04 00  
 3e: 00 00
```

Globals in x86

```
int x = 2;  
int y = 4;  
  
int main (int argc, char** argv) {  
    if (argc > 0) { return x; }  
    else return y;  
}
```

- **clang -c hello.c –o hello.o**
- **objdump –D hello.o**

Disassembly of section __DATA,__data:

```
0000000000000038 <_x>:  
 38: 02 00  
 3a: 00 00
```

```
000000000000003c <_y>:  
 3c: 04 00  
 3e: 00 00
```

Take Away Message

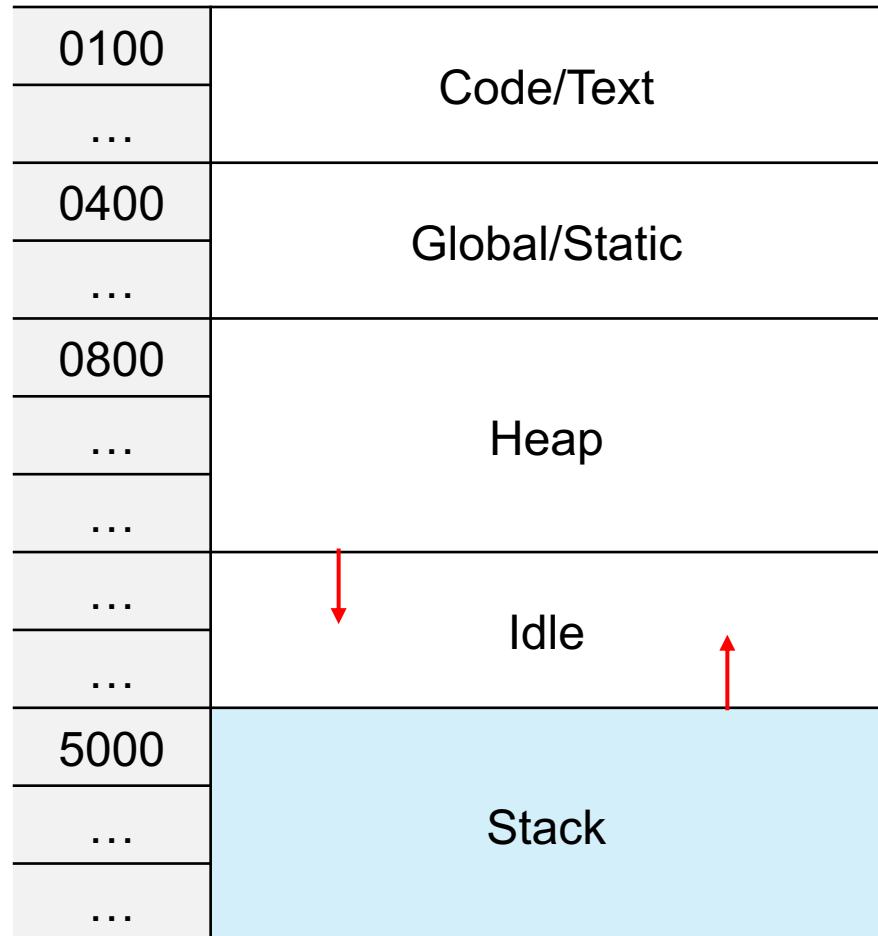
The memory locations of **the code** and **the globals** are determined by the compiler!

Take Away Message

The memory locations of **the code** and **the globals** are determined by the compiler!

The memory locations of other values, *e.g.*, **the locals**, are controlled by the code generated by the compiler and stored in heap or stack!

Stack Memory



Stack Memory

0100	LD R1, *0400	Text
0104	LD R2, *0404	
0108	SUB R1 R1 R2	
0112	ST x, R1	
...	...	
0400	2	Global
0404	3	
0408	...	
...	...	
...		Heap
...		Idle
...		Stack

```

LD R1, y           // R1 = y
LD R2, z           // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST x, R1           // x = R1

```

$$x = y - z$$

```

int y = 2;
int z = 3;

```

```

void foo () {
    int x;
    x = y - z;
}

```

Stack Memory

0100	LD R1 *0400	Text
0104	LD R2 *0404	
0108	SUB R1 R1 R2	
0112	ST x, R1	
...	...	
0400	2	Global
0404	3	
0408	...	
...	...	
...	Heap	
...	Idle	
...	Stack	

```

LD  R1, y           // R1 = y
LD  R2, z           // R2 = z
SUB R1, R1, R2      // R1 = R1 - R2
ST  x, R1         // x = R1

```

$$x = y - z$$

```

int y = 2;
int z = 3;

void foo () {
    int x;
    x = y - z;
}

```

Stack Memory

0100	LD R1 *0400	Text
0104	LD R2 *0404	
0108	SUB R1 R1 R2	
0112	ST x, R1	
...	...	
0400	2	Global
0404	3	
0408	...	
...	...	
...		Heap
...		Idle
...		Stack

```

LD  R1, y           // R1 = y
LD  R2, z           // R2 = z
SUB R1, R1, R2      // R1 = R1 - R2
ST x, R1          // x = R1

```

$$x = y - z$$

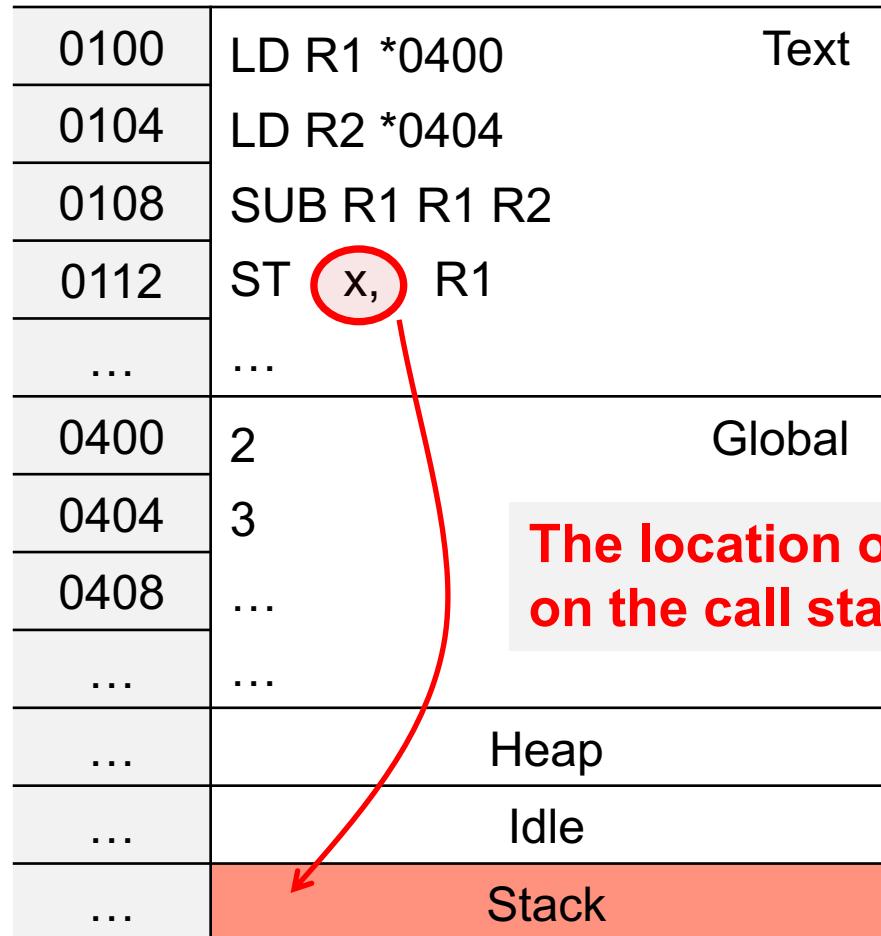
```

int y = 2;
int z = 3;

void foo () {
    int x;
    x = y - z;
}

```

Stack Memory



```

LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2    // R1 = R1 - R2
ST x, R1        // x = R1

```

$$x = y - z$$

```

= 2;
= 3;

void foo () {
int x;
x = y - z;
}

```

Stack Memory

- Call Stack

```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }  
  
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```

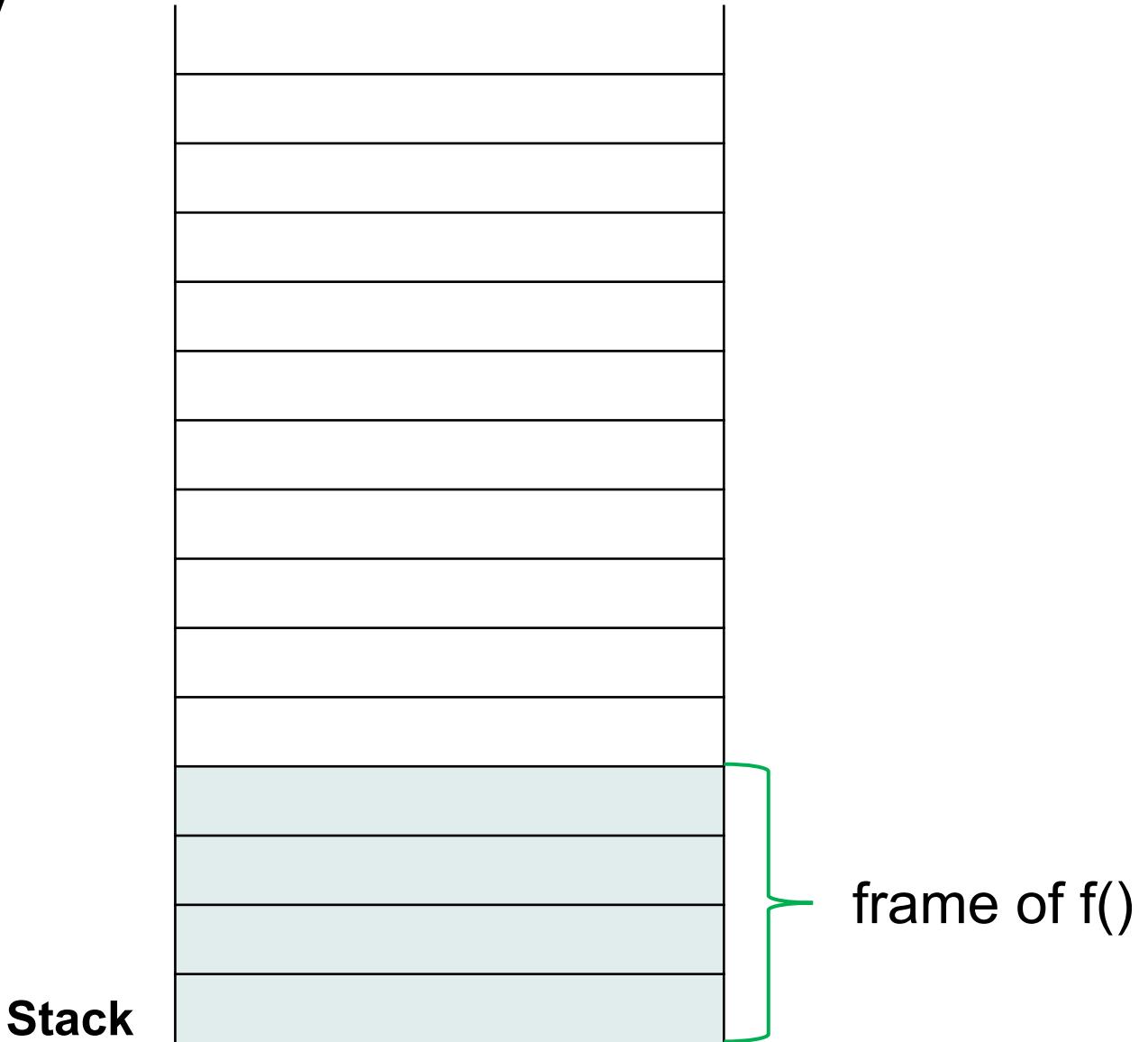
Stack



Stack Memory

- Call Stack

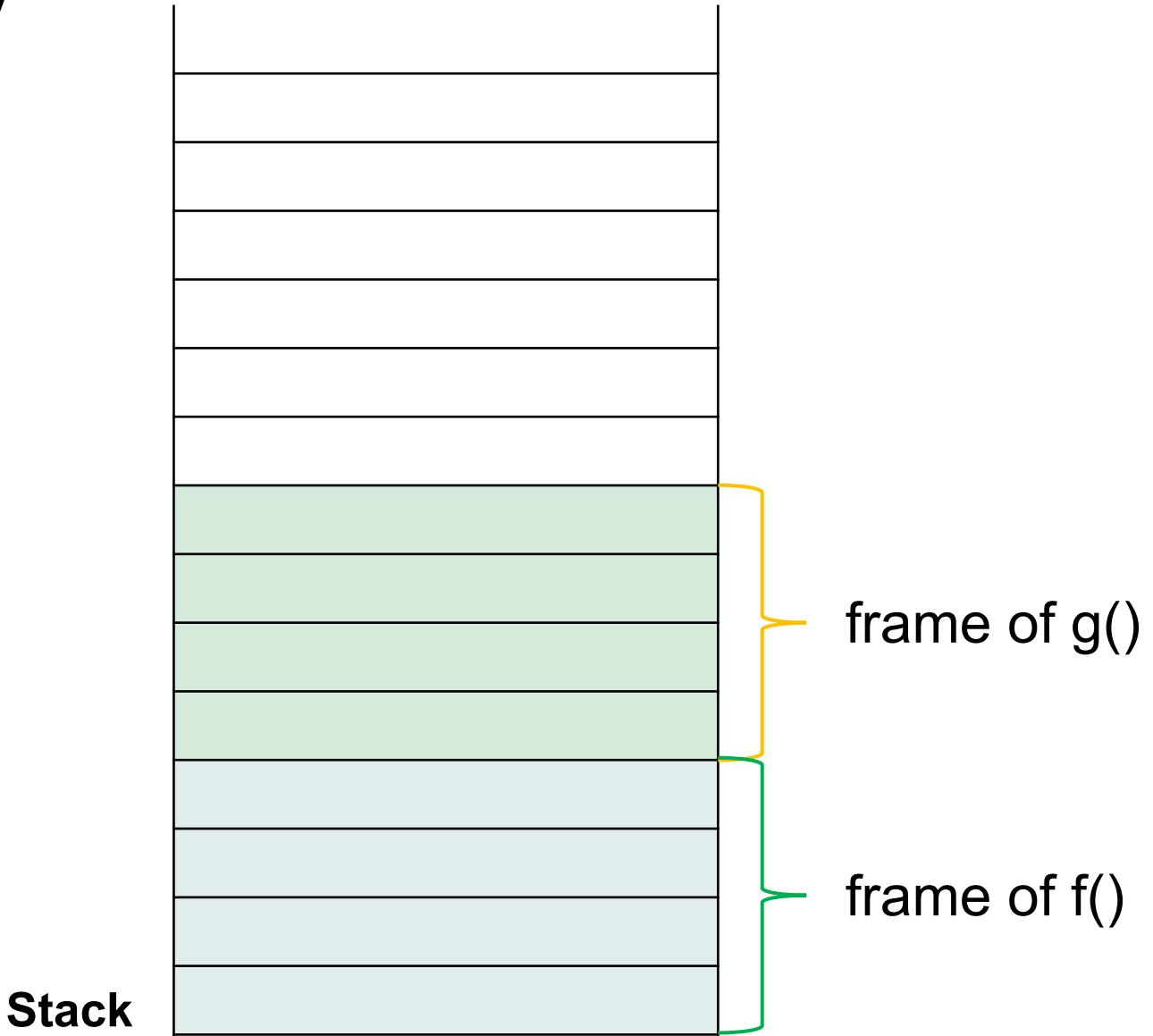
```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }  
  
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```



Stack Memory

- Call Stack

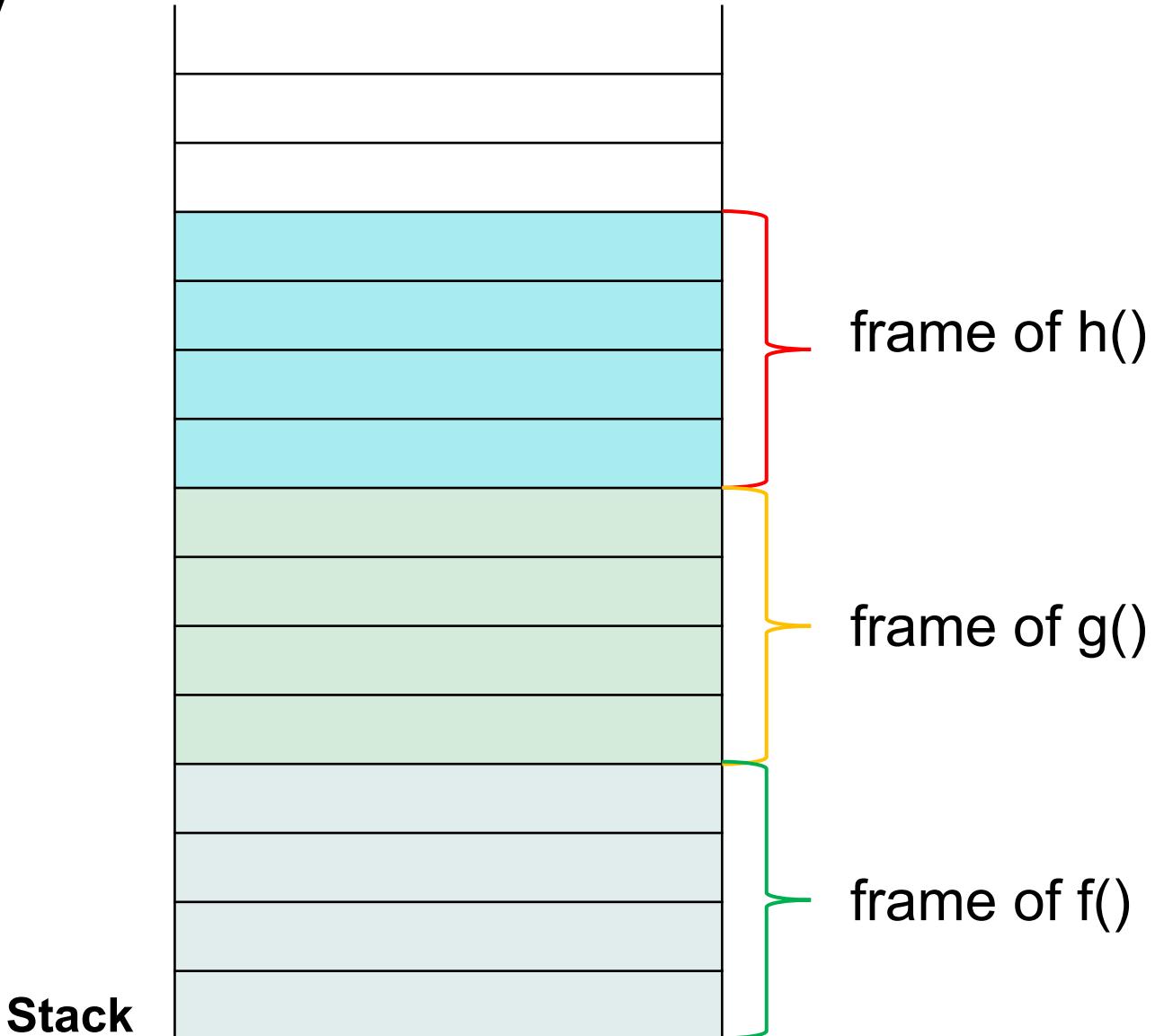
```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }  
  
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```



Stack Memory

- Call Stack

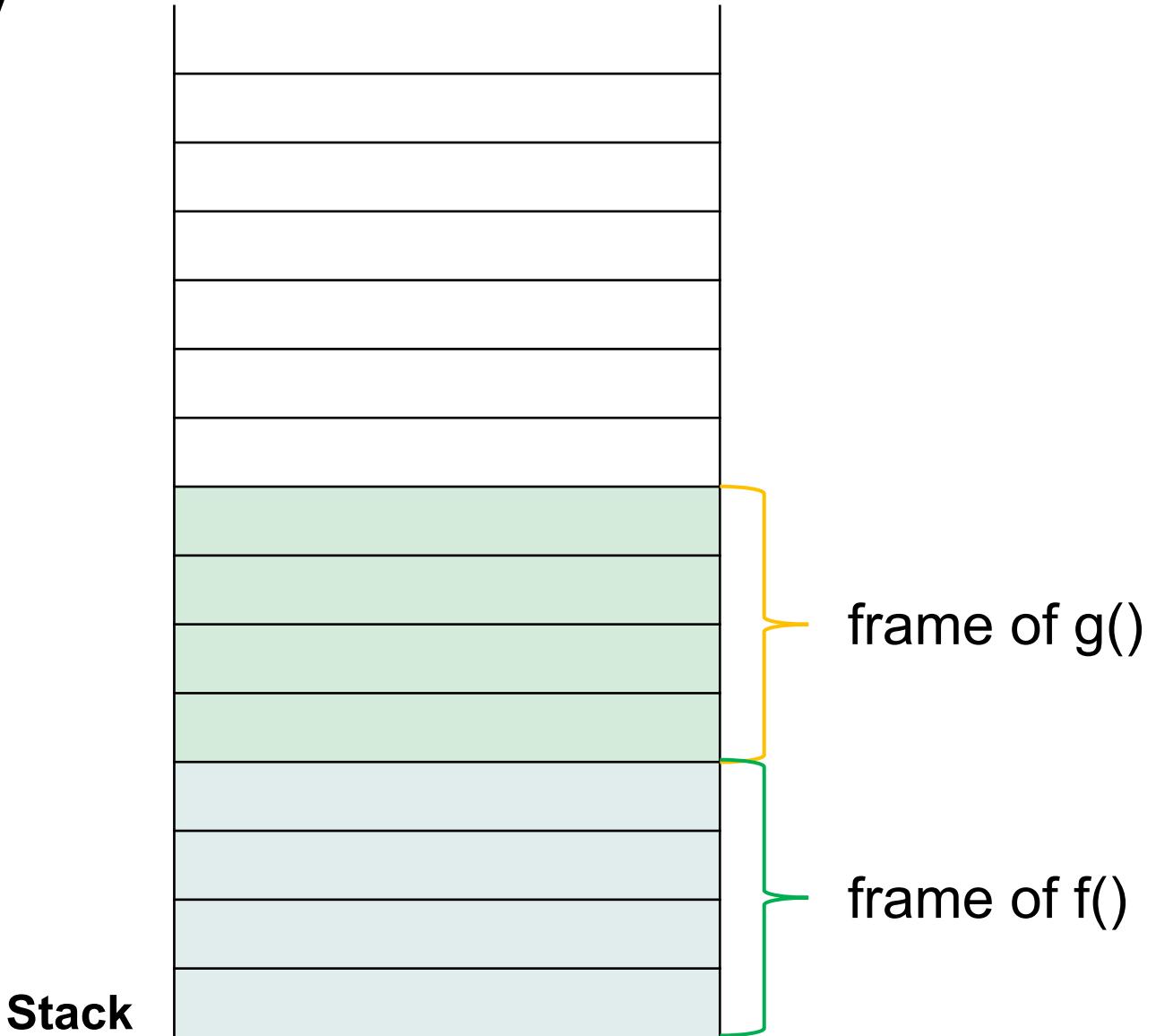
```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }  
  
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```



Stack Memory

- Call Stack

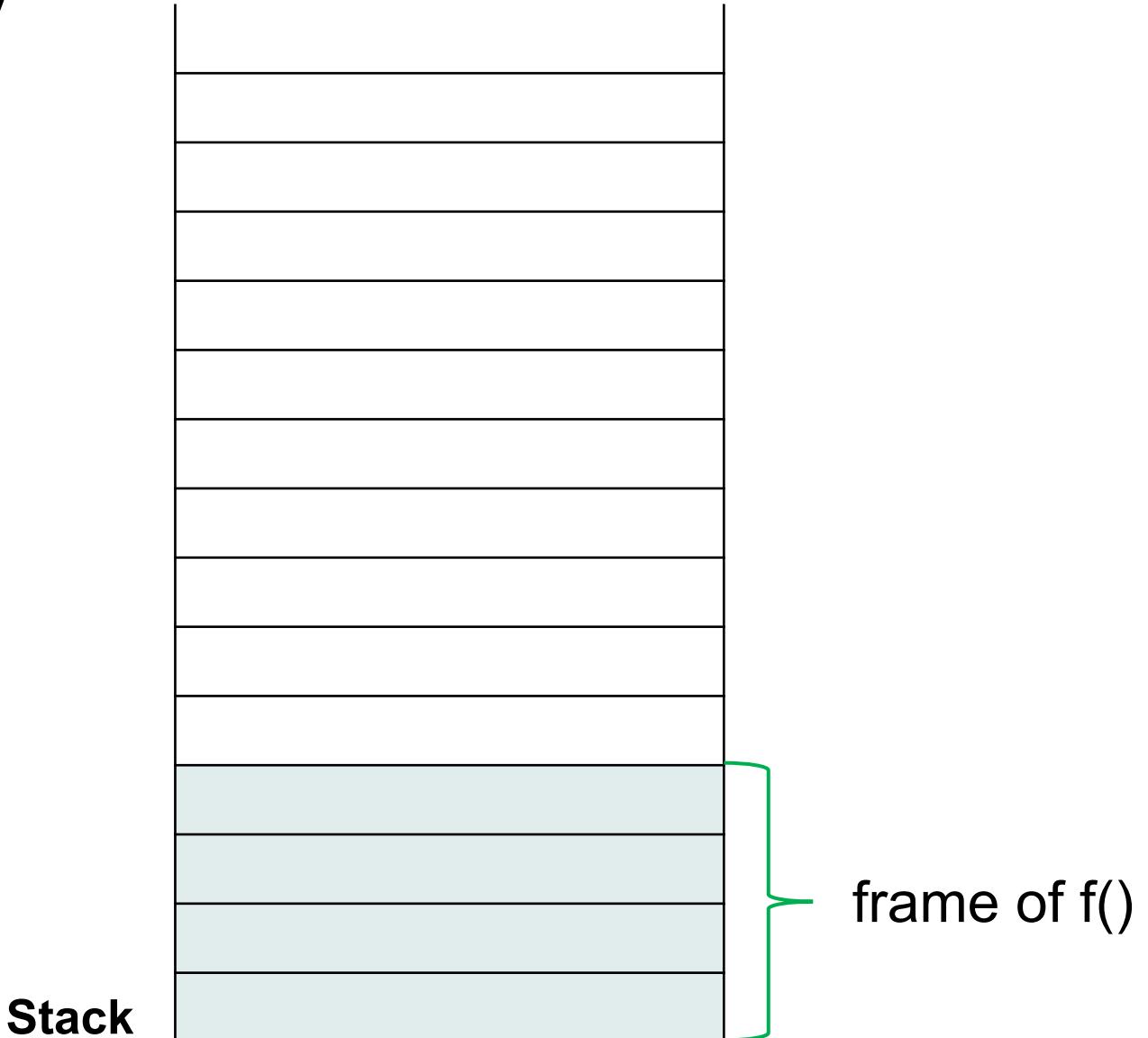
```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }  
  
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```



Stack Memory

- Call Stack

```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }  
  
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```



Stack Memory

- Call Stack

```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }
```



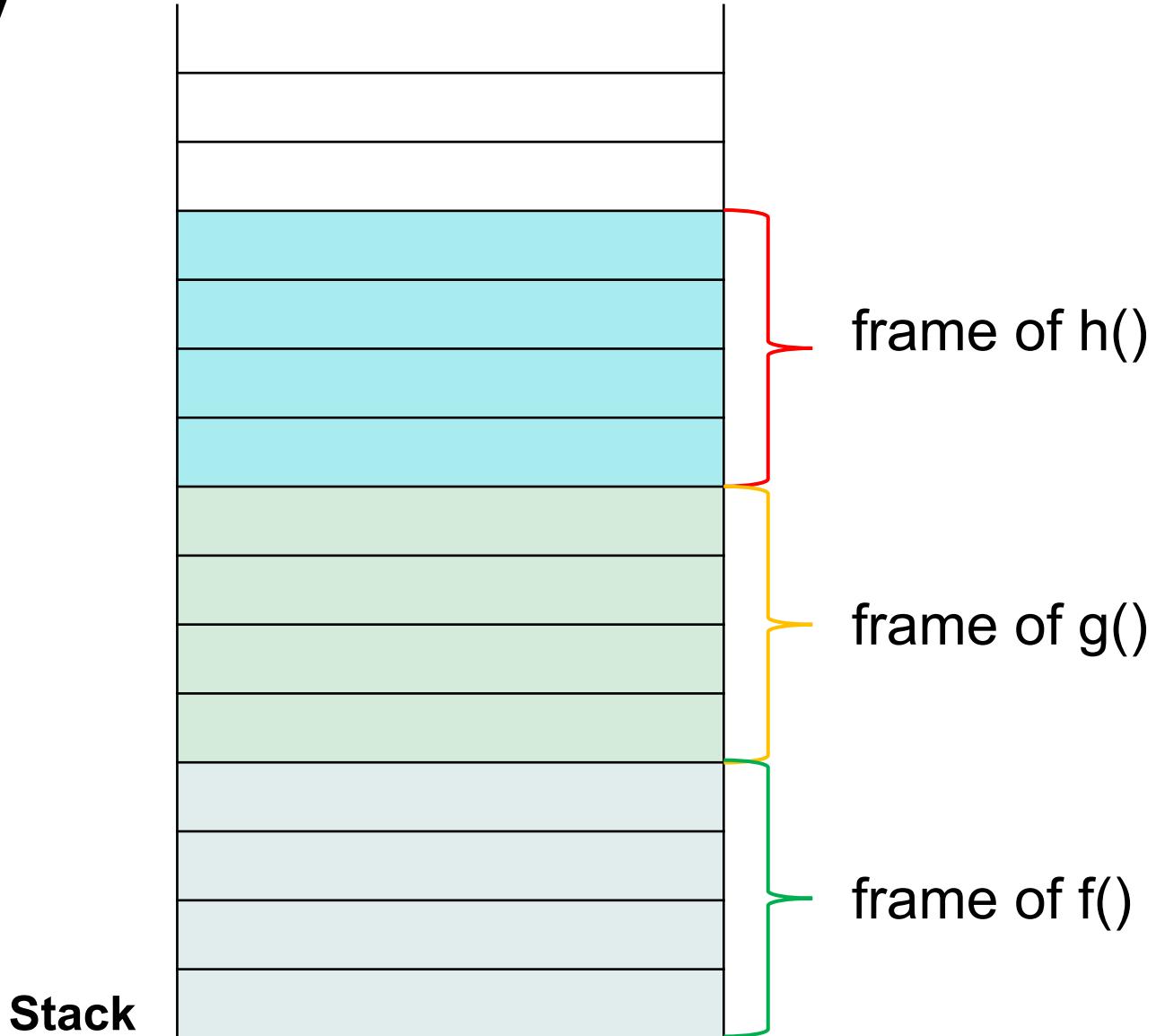
```
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```

Stack



Stack Memory

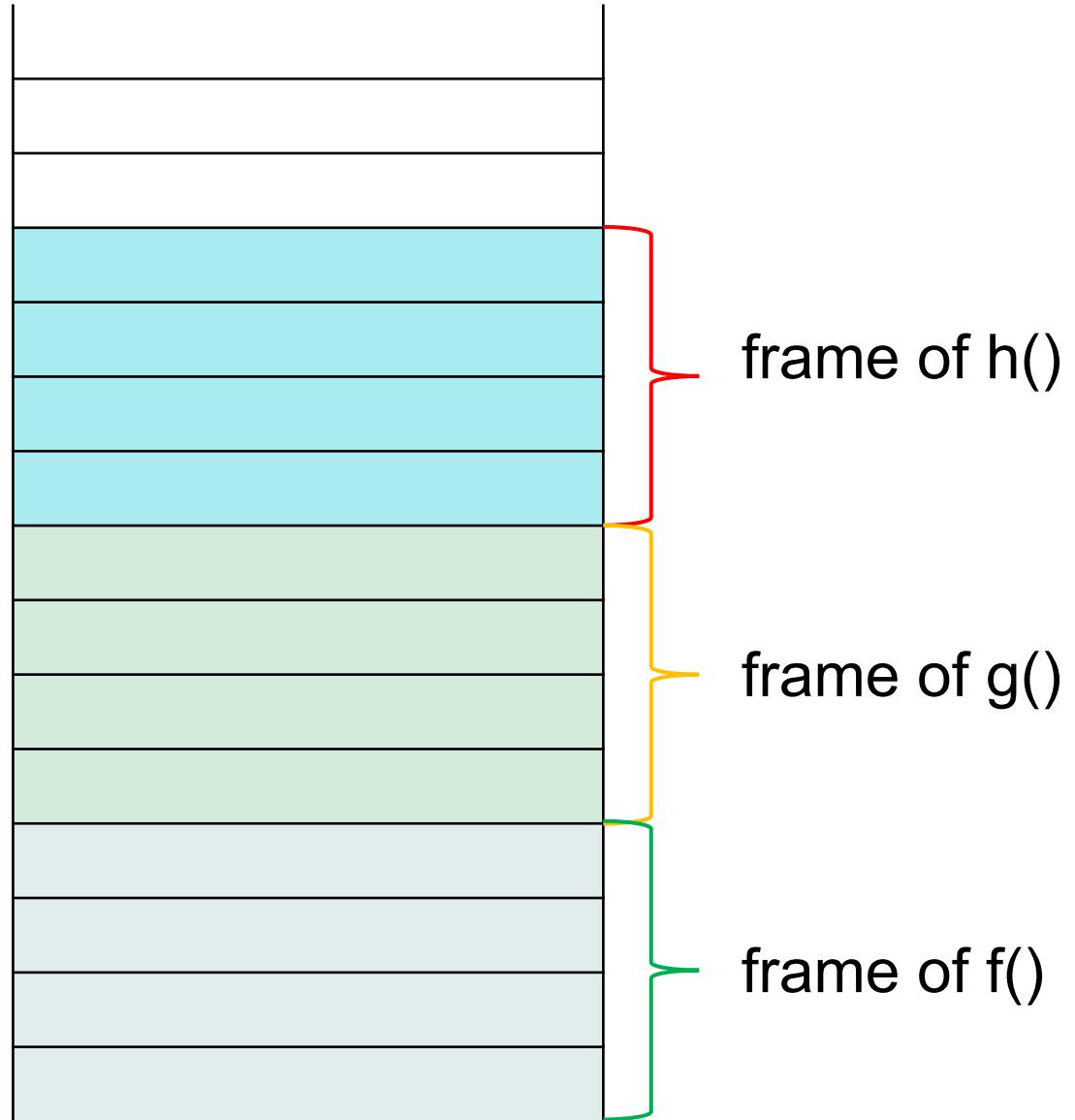
- Call Stack
- What is in a Frame?
 - Local variables
 - Return address
 - Return value, Parameters
 - Data to recover
 - ...



Stack Memory

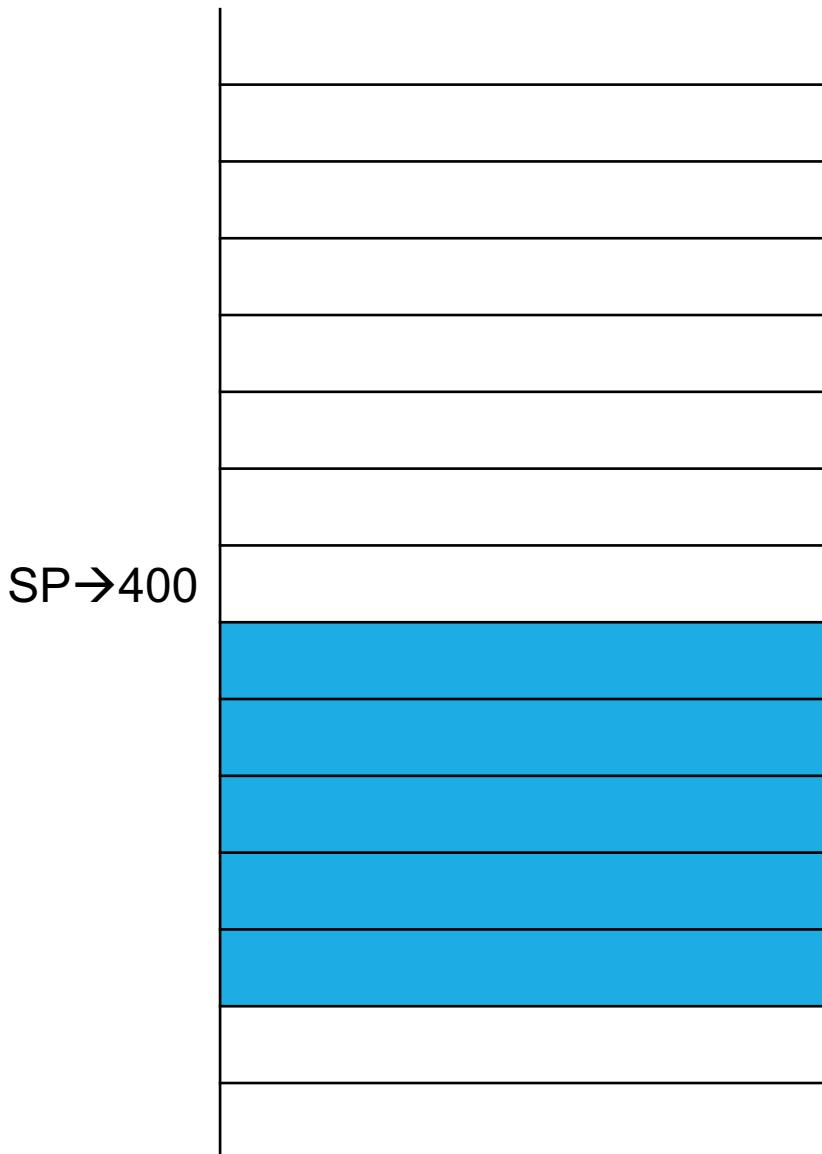
- Call Stack
- What is in a Frame?
 - Local variables
 - Return address
 - Return value, Parameters
 - Data to recover
 - ...
- A **special register**: SP
- A **convention** of using stack

Stack



Stack Memory

- Pop or Release Stack Memory



Stack Memory

- Pop or Release Stack Memory

SP → 400
pop!
ADD SP SP #20
SP → 420



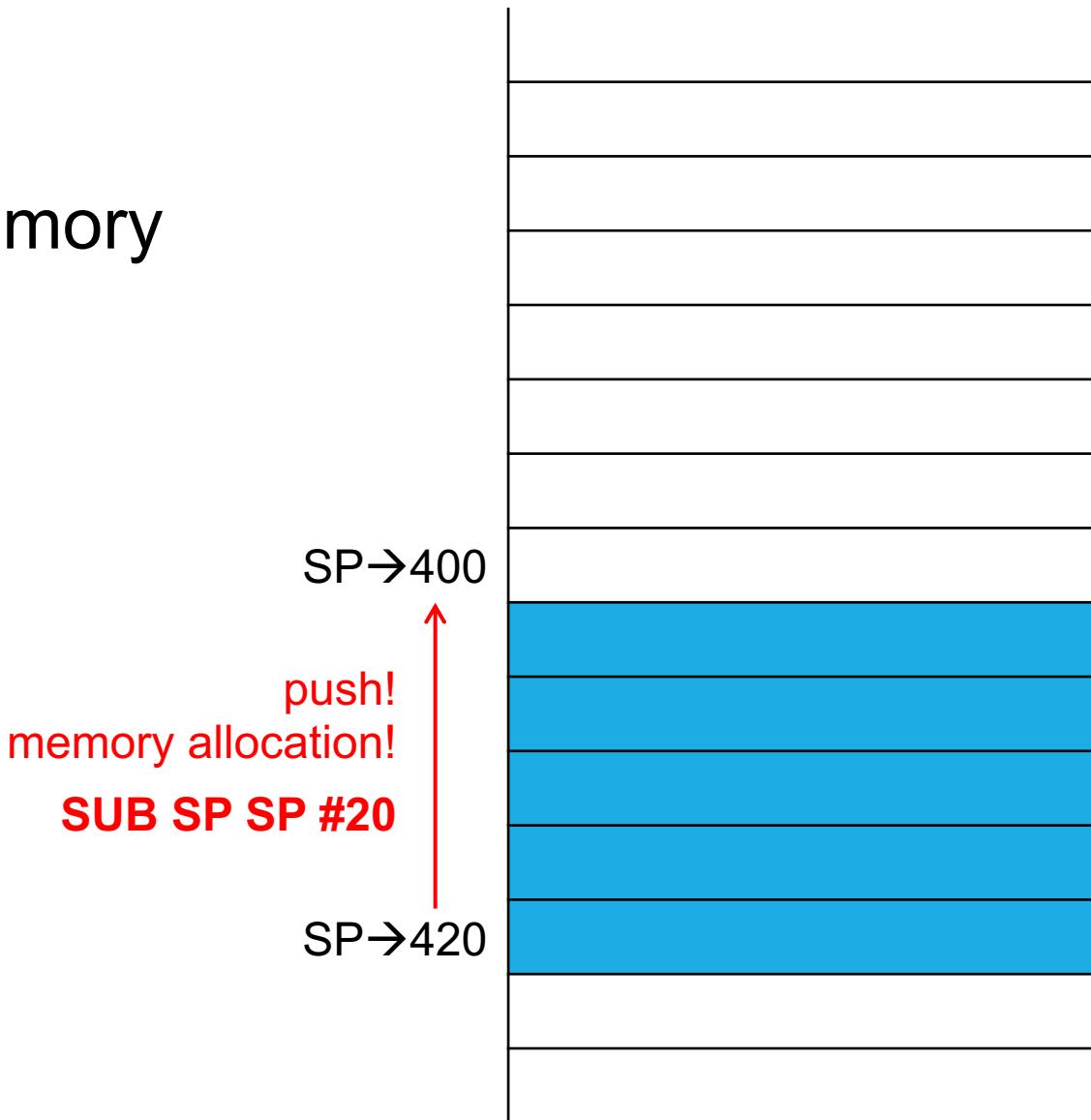
Stack Memory

- Push or Allocate Stack Memory



Stack Memory

- Push or Allocate Stack Memory



Stack Memory

- A convention of using stack

Stack Memory

- Before a call
 - If the procedure uses some registers, push them into the stack
 - Push (record) the **return address** & arguments to the stack
 - Jump to the code of callee

Stack Memory

- Before a call
 - If the procedure uses some registers, push them into the stack
 - Push (record) the **return address** & arguments to the stack
 - Jump to the code of callee
- During a call
 - Allocate memory spaces for locals; and actions based on the locals

Stack Memory

- Before a call
 - If the procedure uses some registers, push them into the stack
 - Push (record) the **return address** & arguments to the stack
 - Jump to the code of callee
- During a call
 - Allocate memory spaces for locals; and actions based on the locals
- After a call
 - Pop the frame;
 - Return to the caller according to the **return address** in the stack
 - Recover data; Get the return value if the function has a return value

Stack Memory

```
1. int y, z;  
2. int main() {  
3.     int r;  
4.     ...  
5.     r = foo();  
6.     ...  
7.     return r;  
8. }  
  
9. int foo() {  
10.    int x;  
11.    x = y - z;  
12.    return x;  
13. }
```

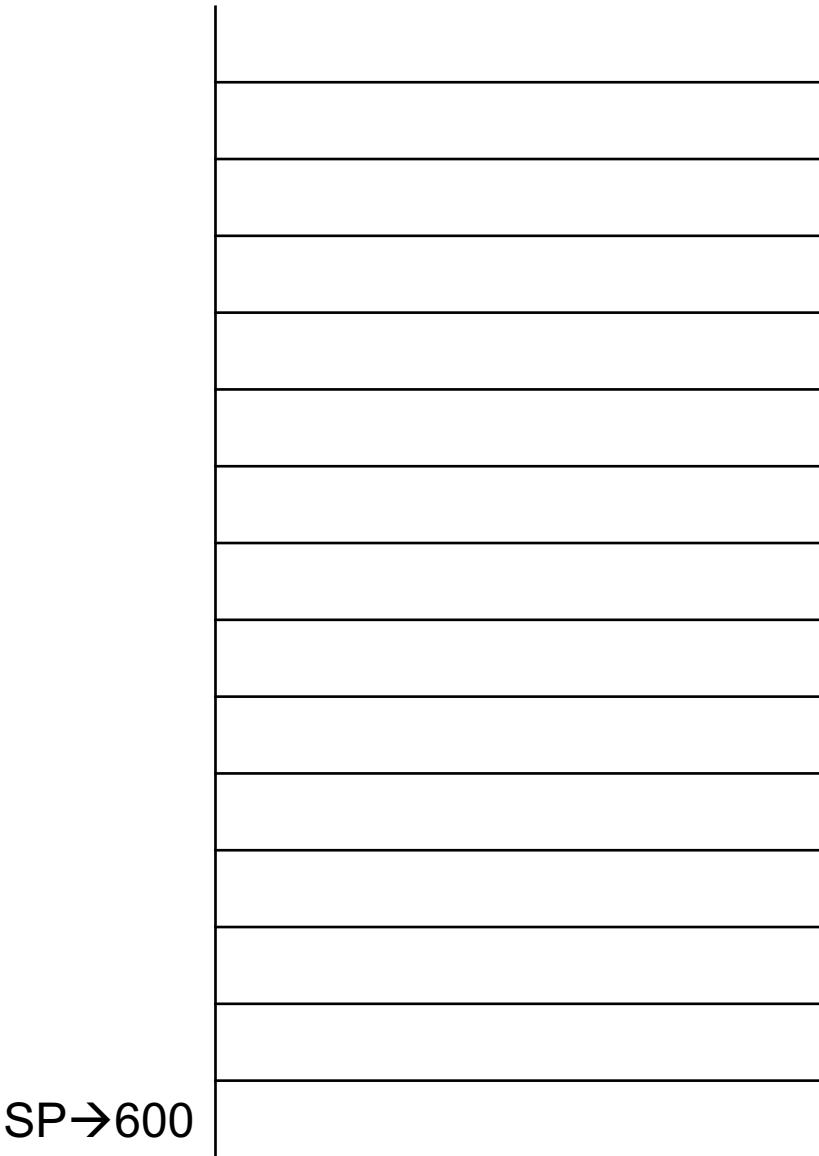
Stack Memory



```
1. int y, z;  
2. int main() {  
3.     int r;  
4.     ...  
5.     r = foo();  
6.     ...  
7.     return r;  
8. }  
  
9. int foo() {  
10.    int x;  
11.    x = y - z;  
12.    return x;  
13. }
```

Stack Memory

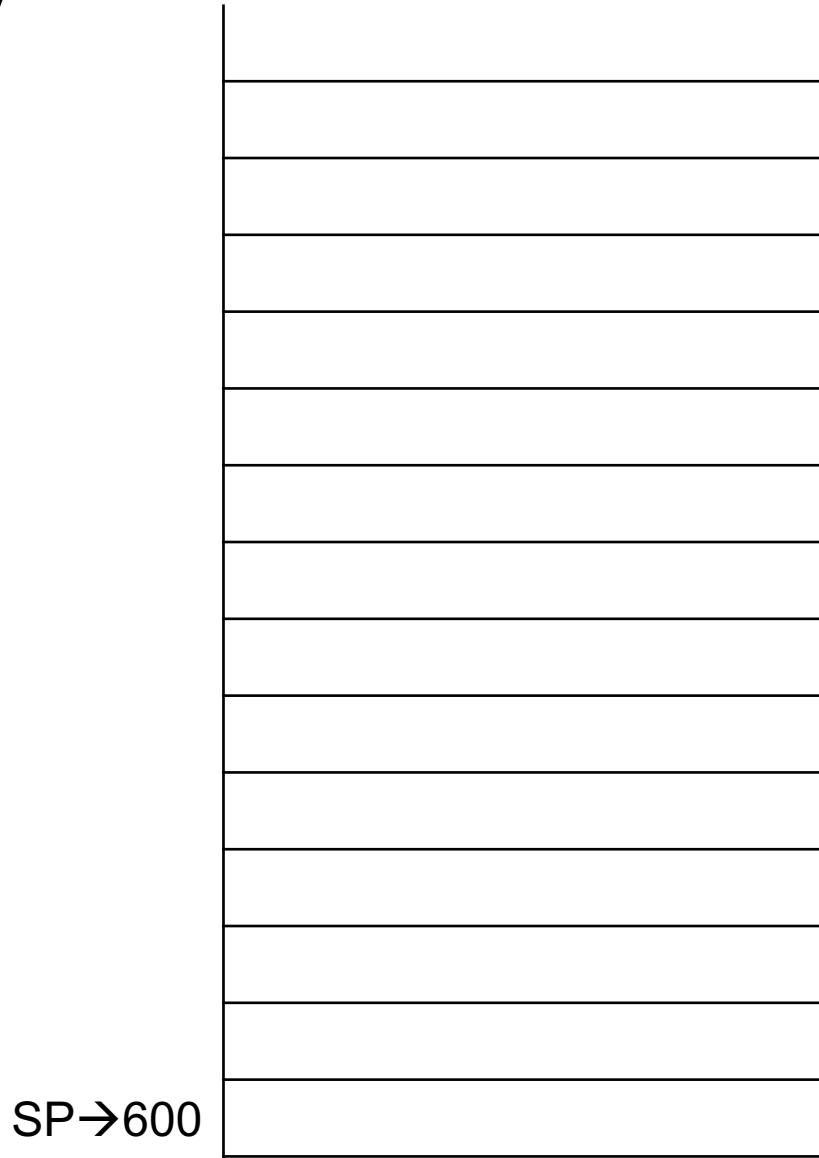
LD SP, #600 // initialize the stack



```
1. int y, z;  
2. int main() {  
3.     int r;  
4.     ...  
5.     r = foo();  
6.     ...  
7.     return r;  
8. }  
  
9. int foo() {  
10.    int x;  
11.    x = y - z;  
12.    return x;  
13. }
```

Stack Memory

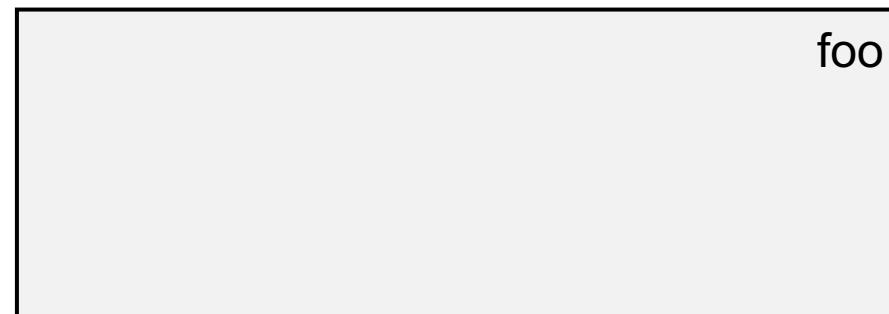
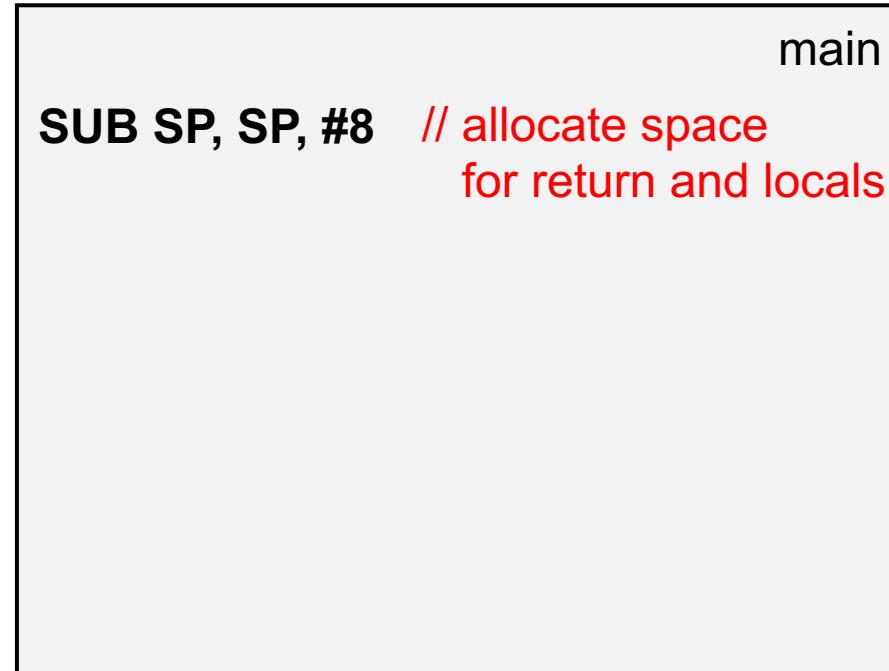
LD SP, #600 // initialize the stack



```
1. int y, z;  
2. int main() {  
3.     int r;  
4.     ...  
5.     r = foo();  
6.     ...  
7.     return r;  
8. }  
  
9. int foo() {  
10.    int x;  
11.    x = y - z;  
12.    return x;  
13. }
```

Stack Memory

LD SP, #600 // initialize the stack



SP → 592
↑
SP → 600



```

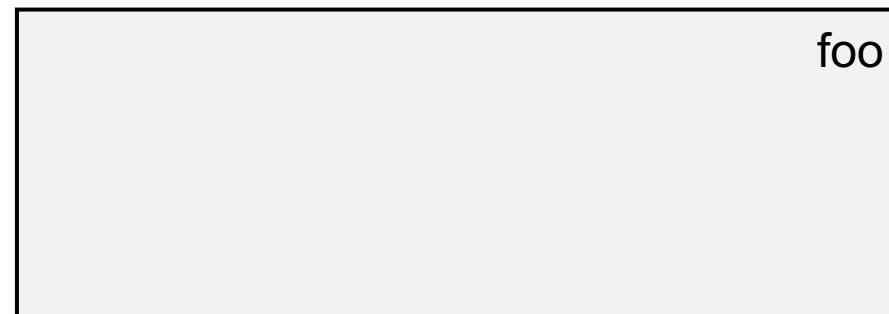
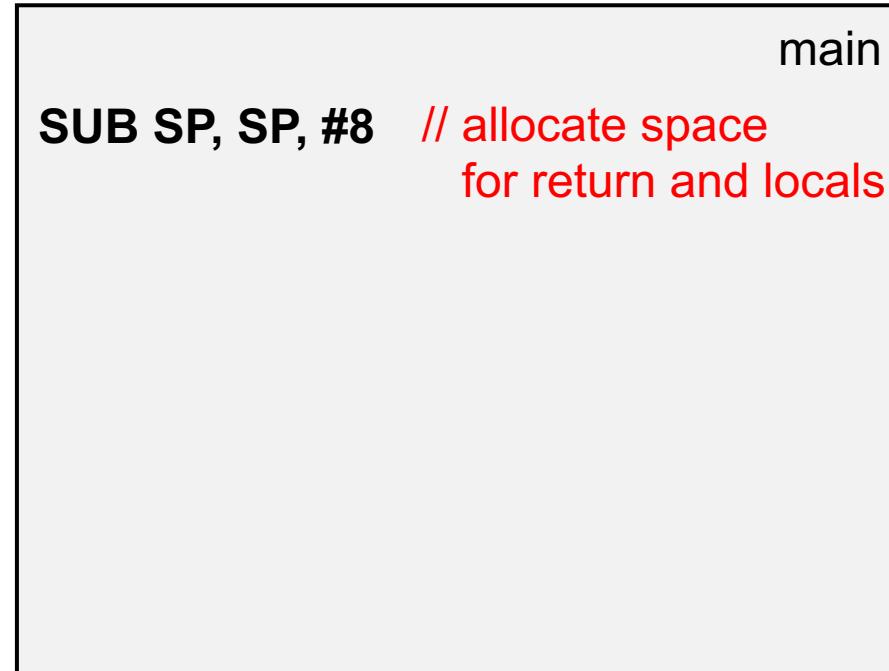
1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack



SP→592



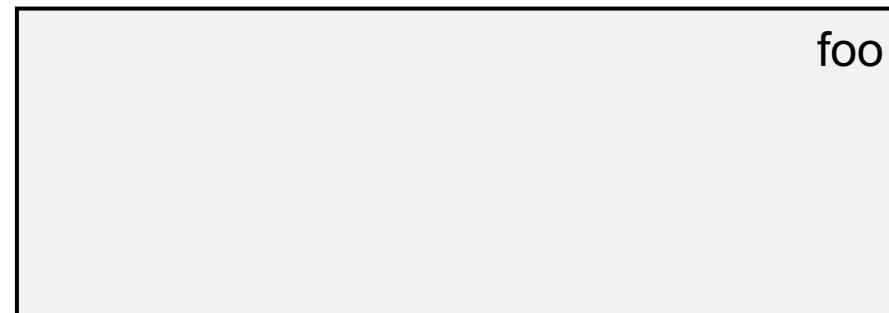
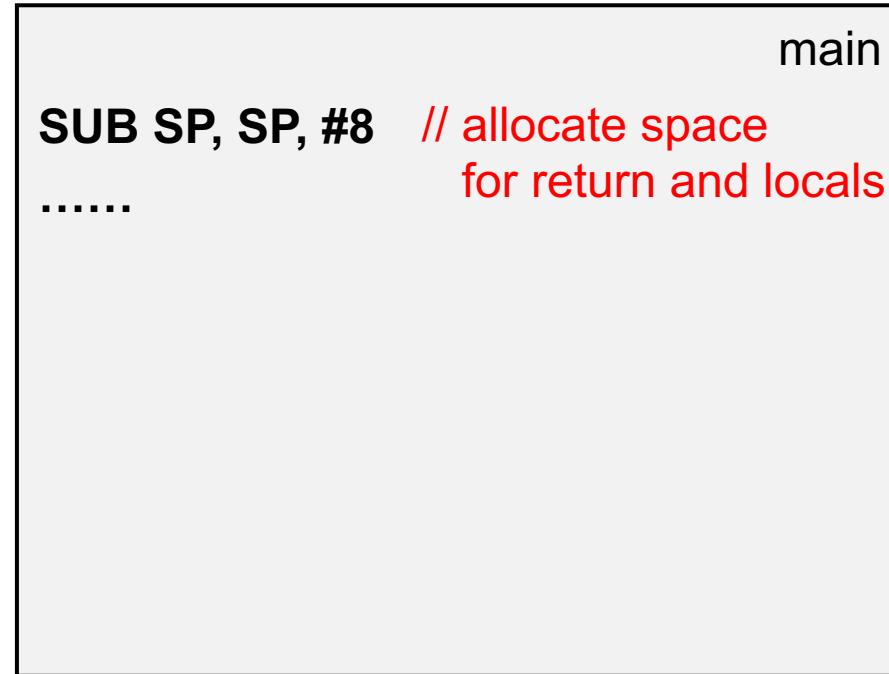
```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }
```

Stack Memory

LD SP, #600 // initialize the stack



SP→592



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }
```

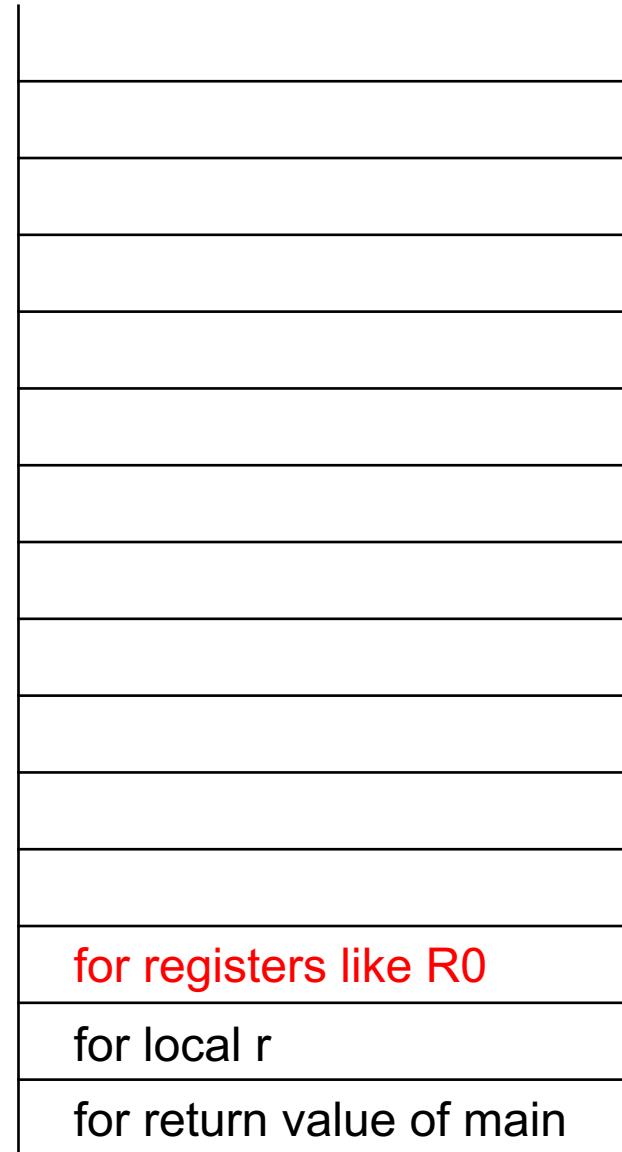
Stack Memory

LD SP, #600 // initialize the stack

```
main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
```

```
foo
```

SP→588



```
1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }
```

Stack Memory

LD SP, #600 // initialize the stack

```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
L1 .....

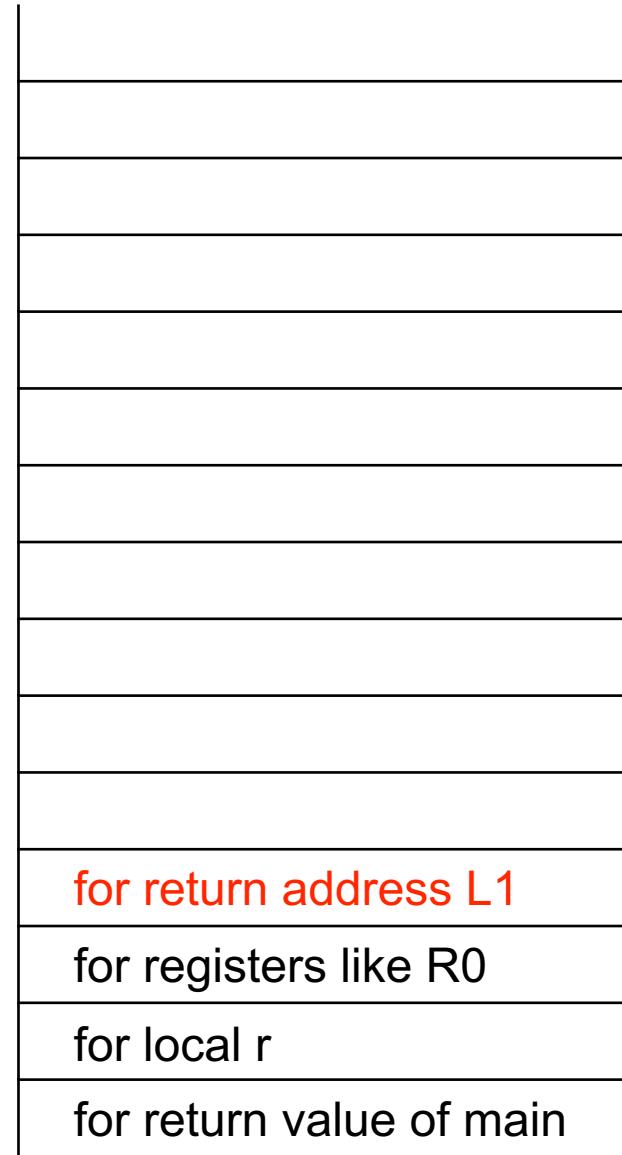
```

```

L2 .....
foo
BR .....

```

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
L1
.....

```

```

L2
.....
foo
BR .....

```

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }
9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....
```

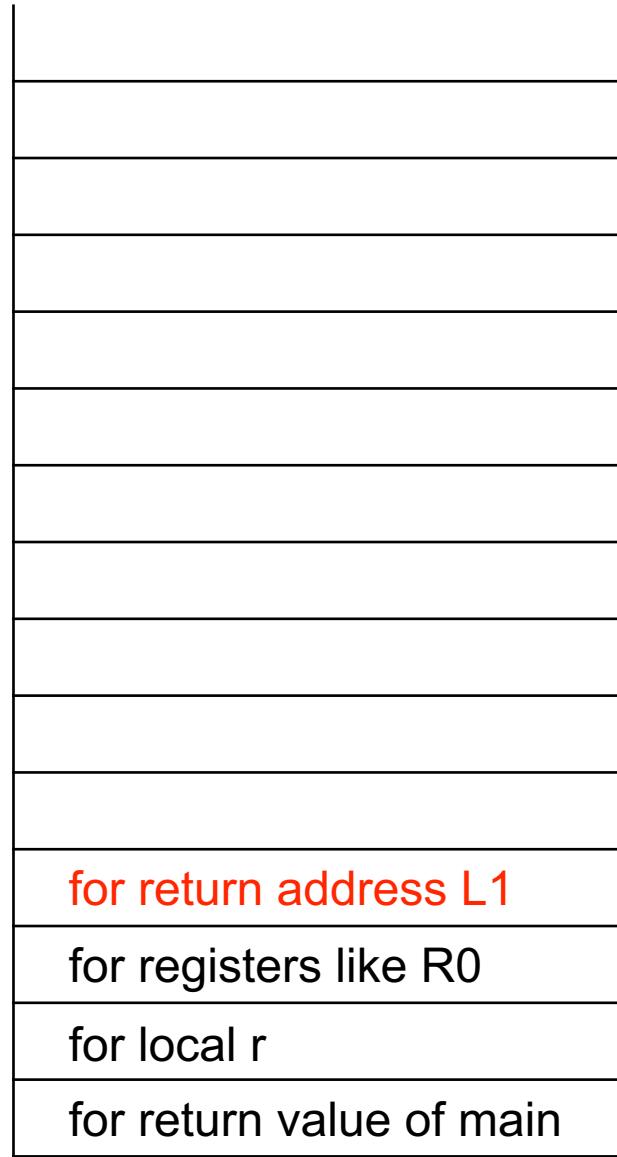
```

L1
.....
L2
.....
```

foo

BR

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }
```

Stack Memory

LD SP, #600 // initialize the stack

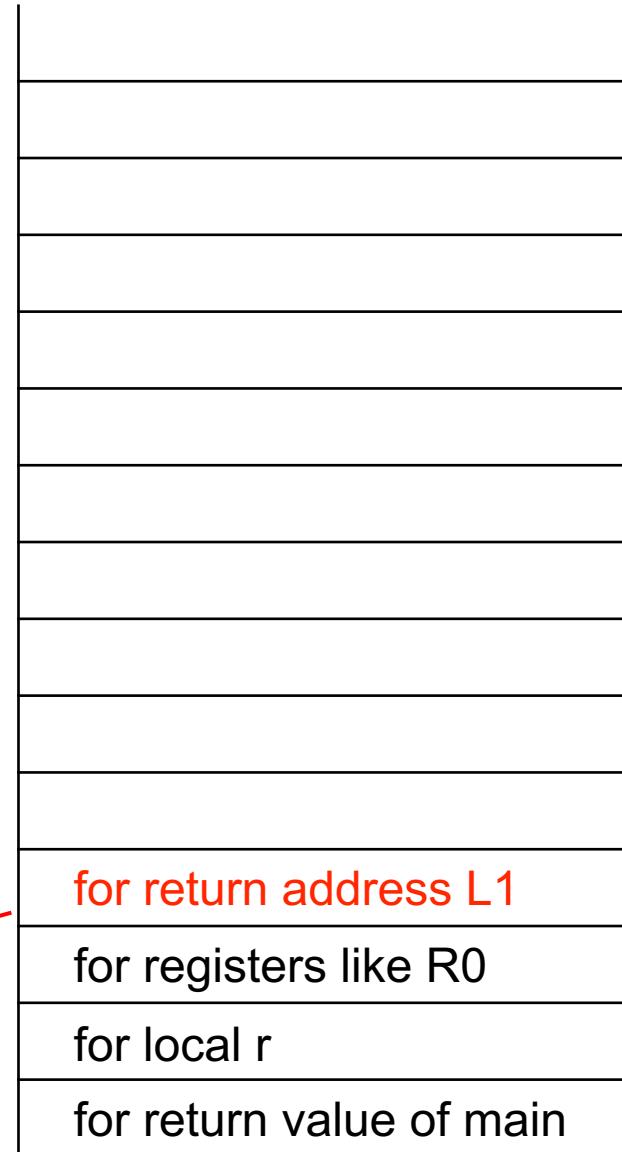
```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....
L1
```

```

L2
foo
.....
BR .....
```

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }
9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }
```

Stack Memory

LD SP, #600 // initialize the stack

```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....
```

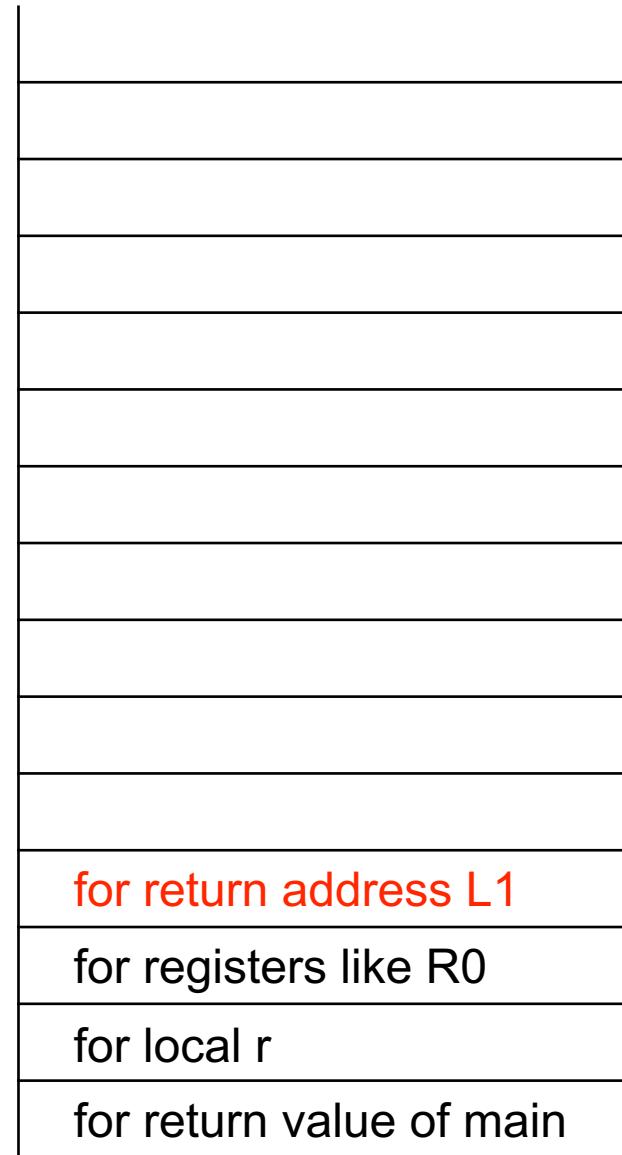
L₁

```

foo
.....
```

BR

SP→584



for return address L₁

for registers like R₀

for local r

for return value of main

```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }
```

Stack Memory

LD SP, #600 // initialize the stack

```

main

SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0

SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....

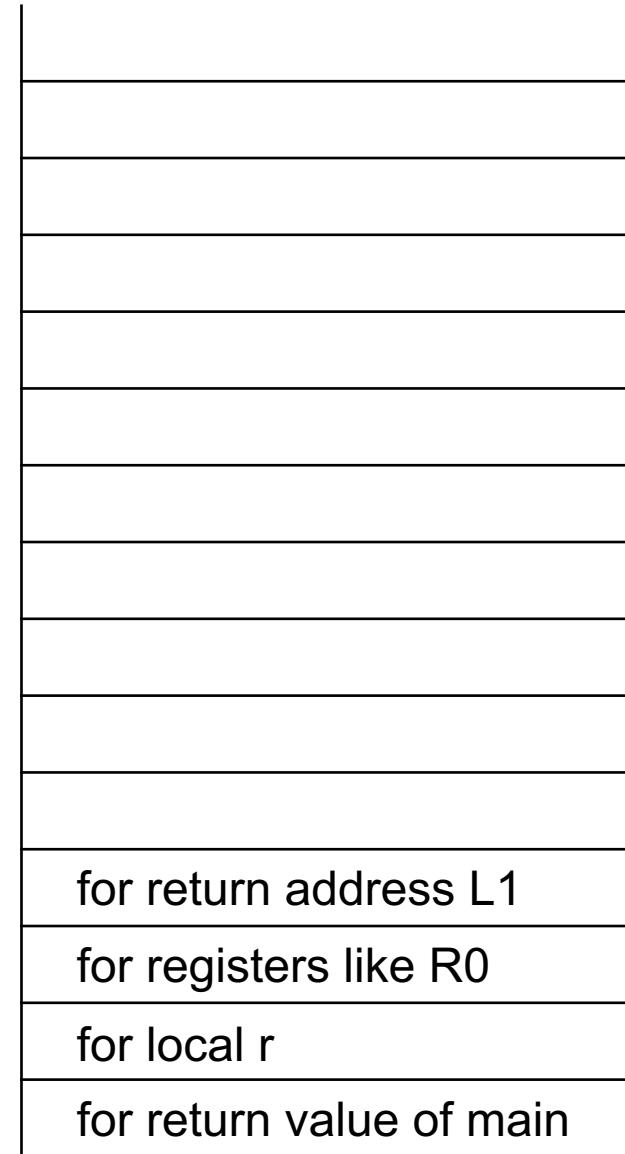
```

L₁

L₂

BR

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....

```

L₁

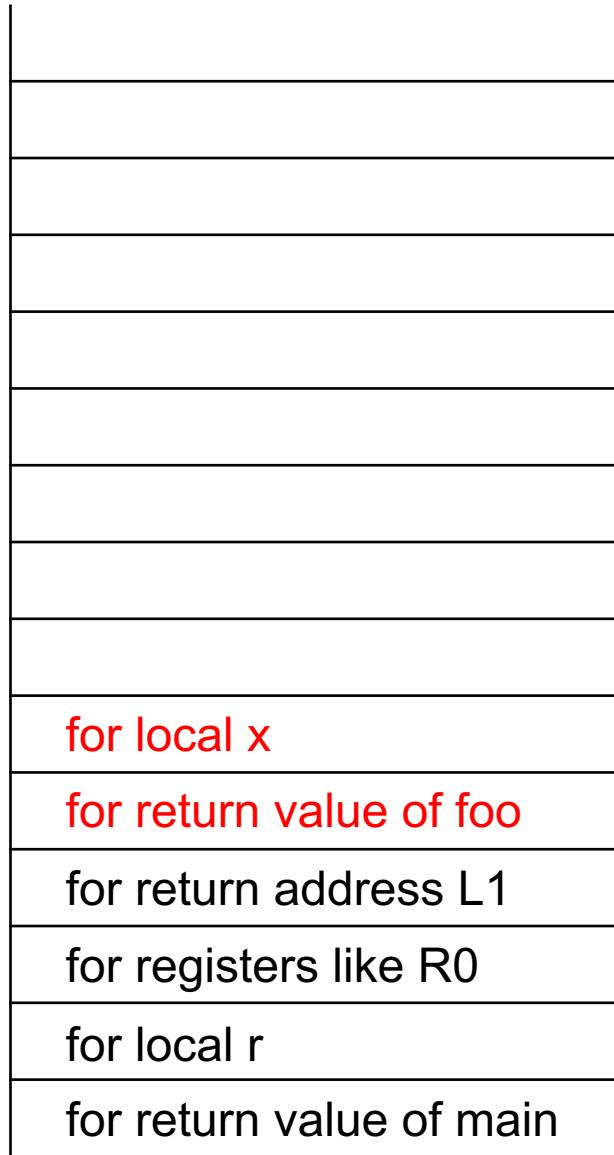
```

SUB SP, SP, #8
..... // x = y - z

```

BR

SP→576



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....

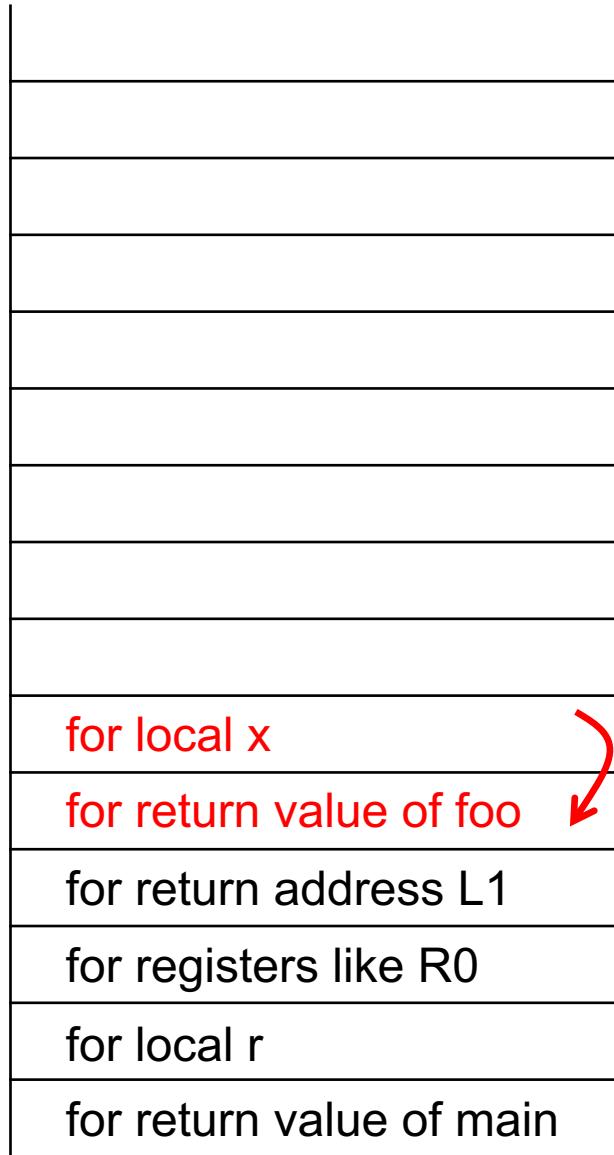
```

```

L2
SUB SP, SP, #8
.....
LD R0, 4(SP) ST 8(SP), R0
BR .....

```

SP→576



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }
9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

```

        main
LD SP, #600 // initialize the stack

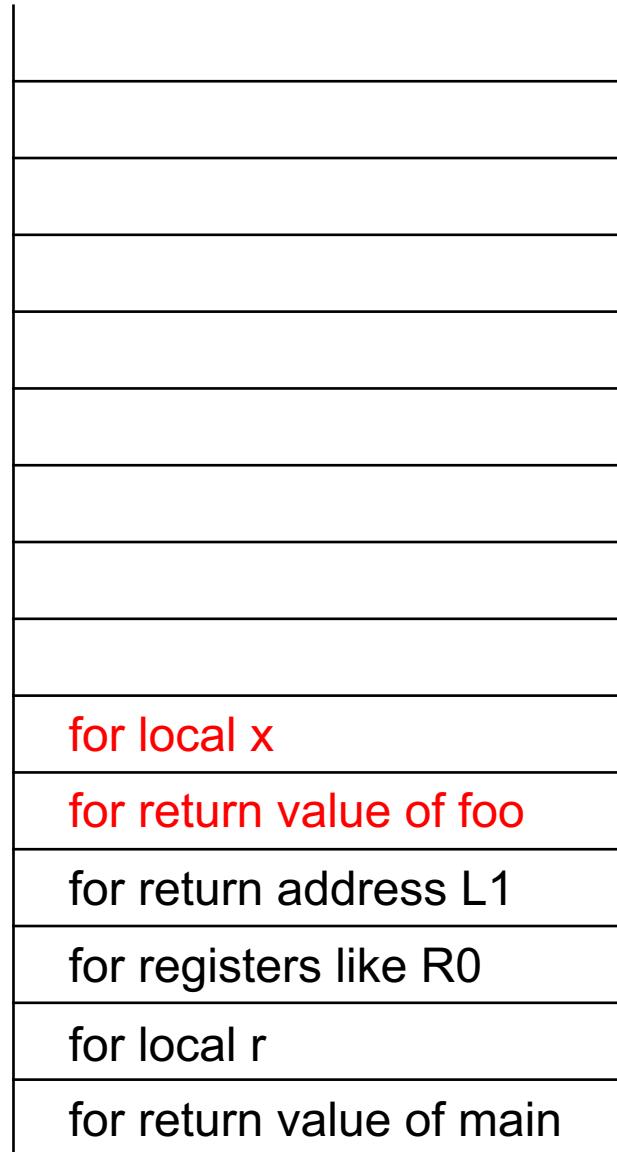
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0

SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....
L1
```

```

L2
SUB SP, SP, #8
.....
LD R0, 4(SP) ST 8(SP), R0
ADD SP, SP, #8
BR .....
```

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }
```

Stack Memory

LD SP, #600 // initialize the stack

```

        main
LD SP, #600 // initialize the stack

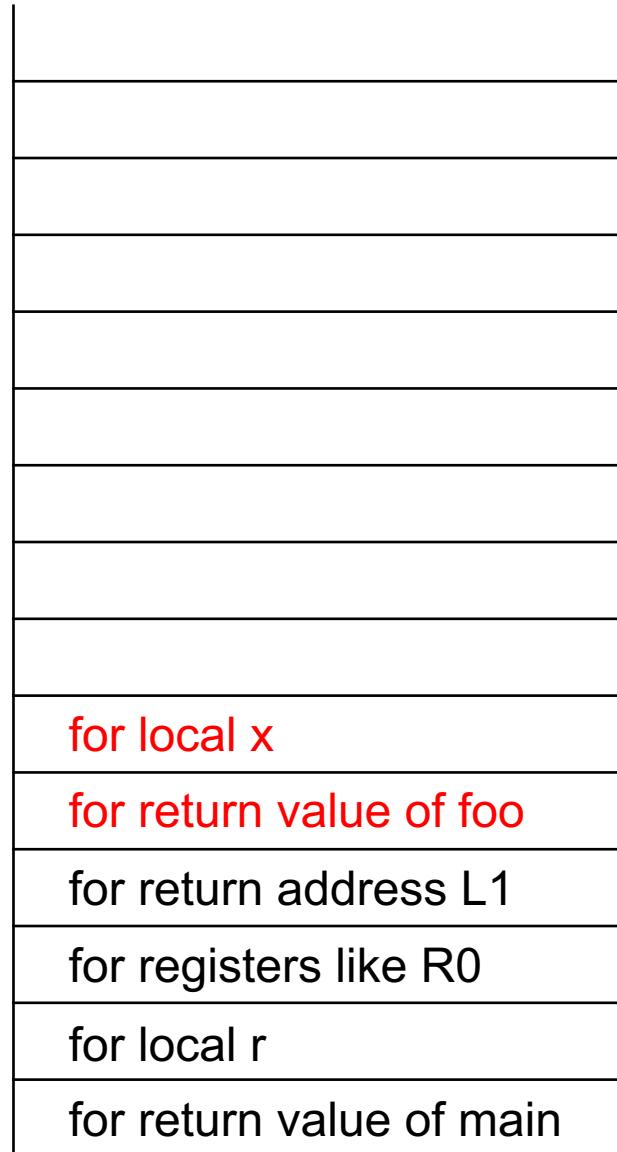
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0

SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....
L1
```

```

L2
SUB SP, SP, #8
.....
LD R0, 4(SP) ST 8(SP), R0
ADD SP, SP, #8
BR 4(SP)
```

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }
```

Stack Memory

LD SP, #600 // initialize the stack

```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....

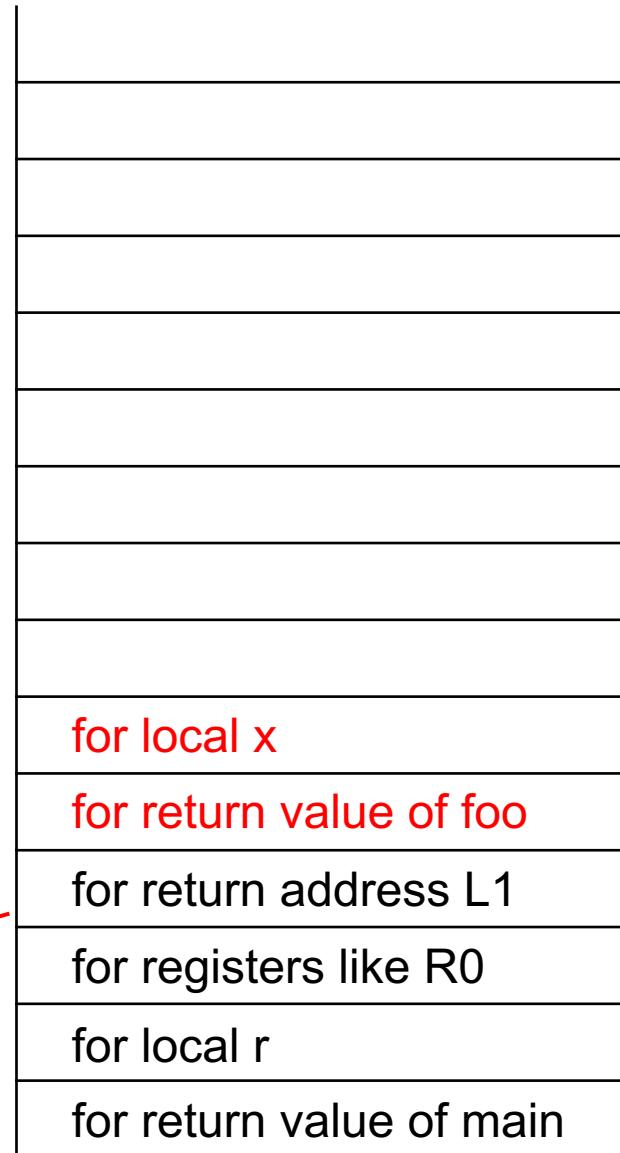
```

```

L1
SUB SP, SP, #8
.....
LD R0, 4(SP) ST 8(SP), R0
ADD SP, SP, #8
BR 4(SP) ←

```

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }
9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

```

main
SUB SP, SP, #8 // allocate space
for return and locals
.....
SUB SP, SP, #4 // record registers
ST 4(SP), R0
SUB SP, SP, #4 // record return addr
ST 4(SP), L1
BR L2
.....

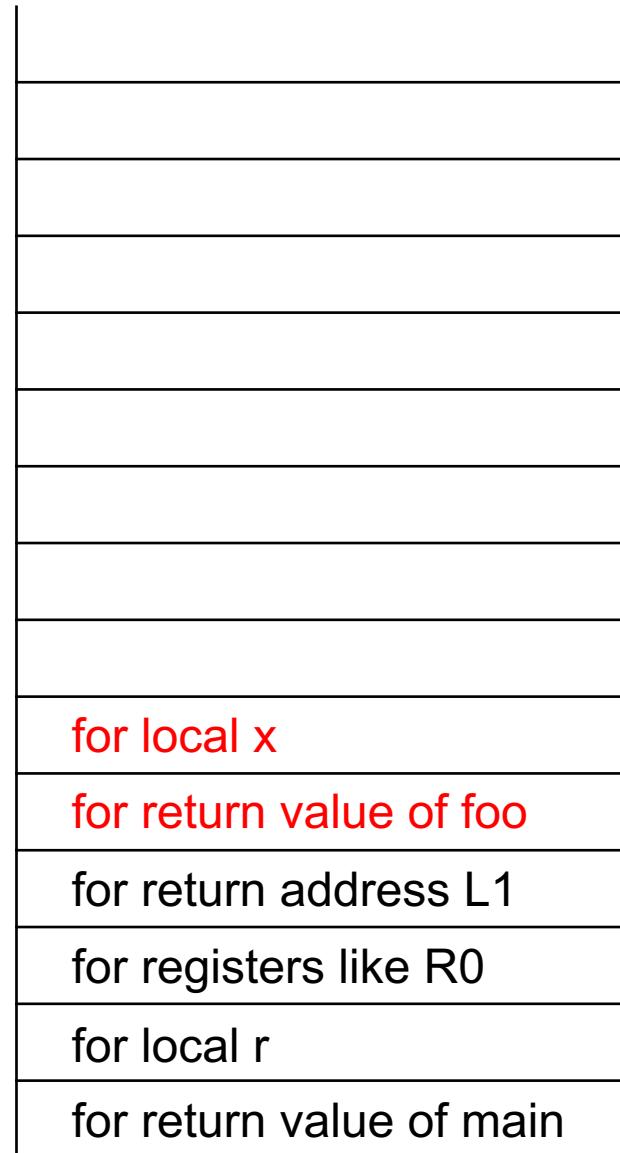
```

```

L1
SUB SP, SP, #8
.....
LD R0, 4(SP) ST 8(SP), R0
ADD SP, SP, #8
BR 4(SP)

```

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }
9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

main

SUB SP, SP, #8 // allocate space
for return and locals
.....

SUB SP, SP, #4 // record registers
ST 4(SP), R0

SUB SP, SP, #4 // record return addr
ST 4(SP), L₁
BR L₂

L₁ LD R0, 8(SP) // recover R0

SP→584

for local x

for return value of foo

for return address L₁

for registers like R0

for local r

for return value of main

```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

main

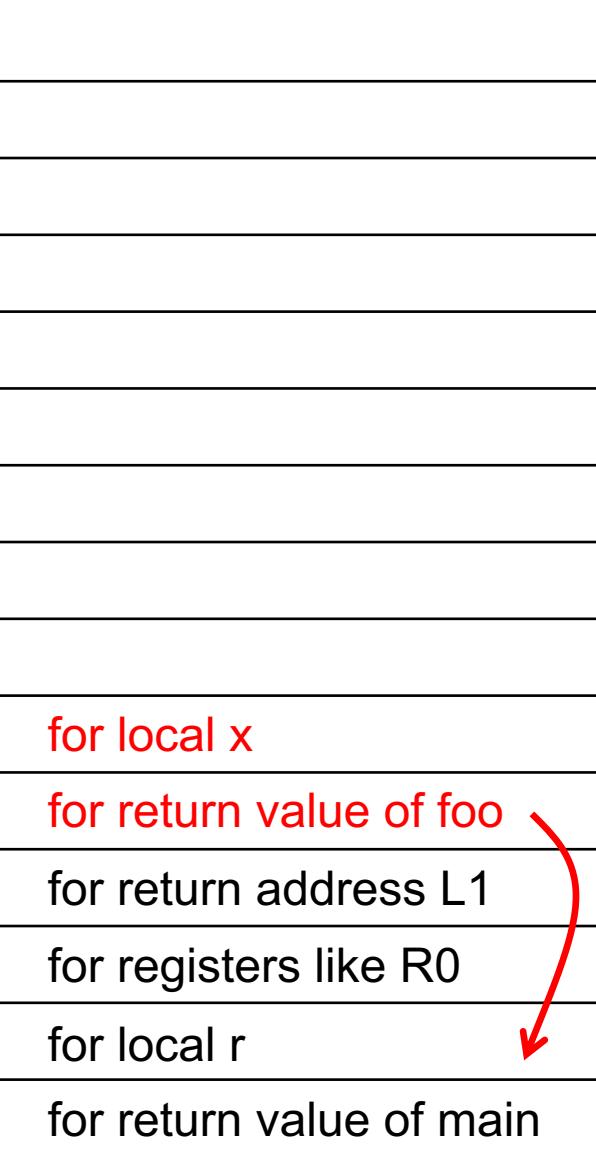
SUB SP, SP, #8 // allocate space
for return and locals
.....

SUB SP, SP, #4 // record registers
ST 4(SP), R0

SUB SP, SP, #4 // record return addr
ST 4(SP), L₁
BR L₂

L₁ LD R0, 8(SP) // recover R0
LD R1, 0(SP) ST 12(SP), R1

SP→584



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

main

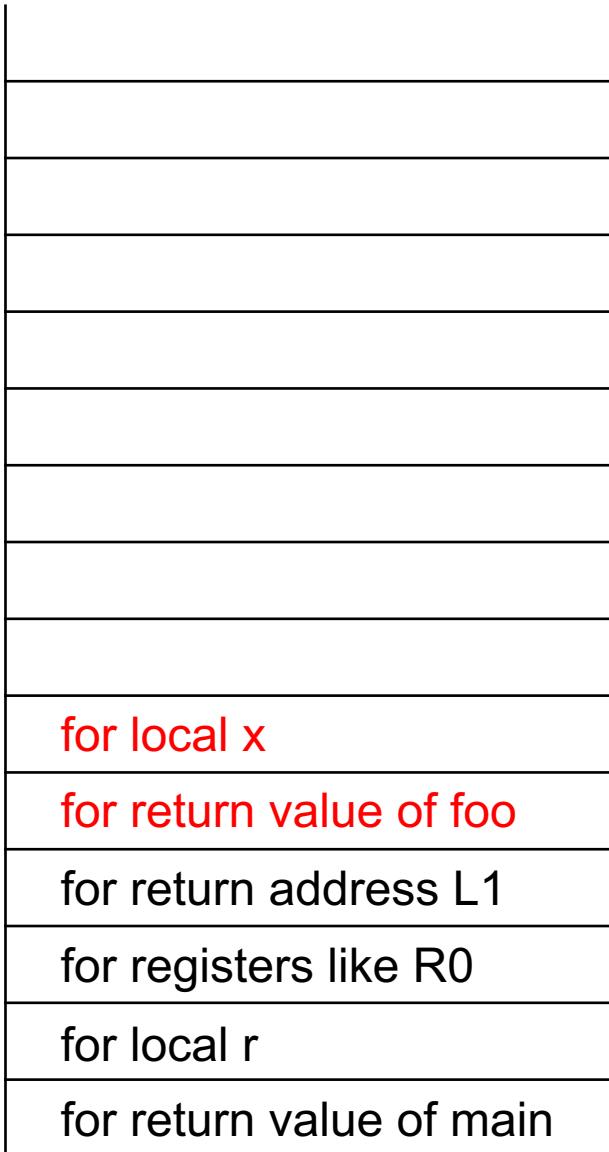
SUB SP, SP, #8 // allocate space
for return and locals
.....

SUB SP, SP, #4 // record registers
ST 4(SP), R0

SUB SP, SP, #4 // record return addr
ST 4(SP), L₁
BR L₂

L₁ LD R0, 8(SP) // recover R0
LD R1, 0(SP) ST 12(SP), R1
ADD SP, SP, #8 // pop the space

SP → 592



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

main

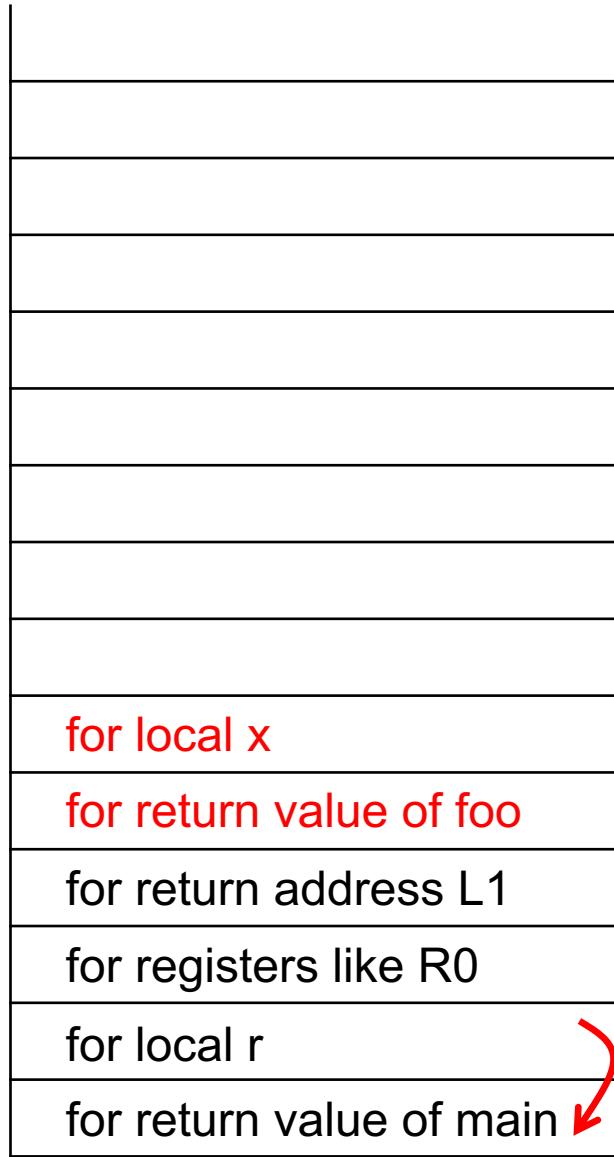
SUB SP, SP, #8 // allocate space
for return and locals
.....

SUB SP, SP, #4 // record registers
ST 4(SP), R0

SUB SP, SP, #4 // record return addr
ST 4(SP), L₁
BR L₂

L₁ LD R0, 8(SP) // recover R0
LD R1, 0(SP) ST 12(SP), R1
ADD SP, SP, #8 // pop the space
LD R0, 4(SP) ST 8(SP), R0

SP → 592



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Stack Memory

LD SP, #600 // initialize the stack

main

SUB SP, SP, #8 // allocate space
for return and locals
.....

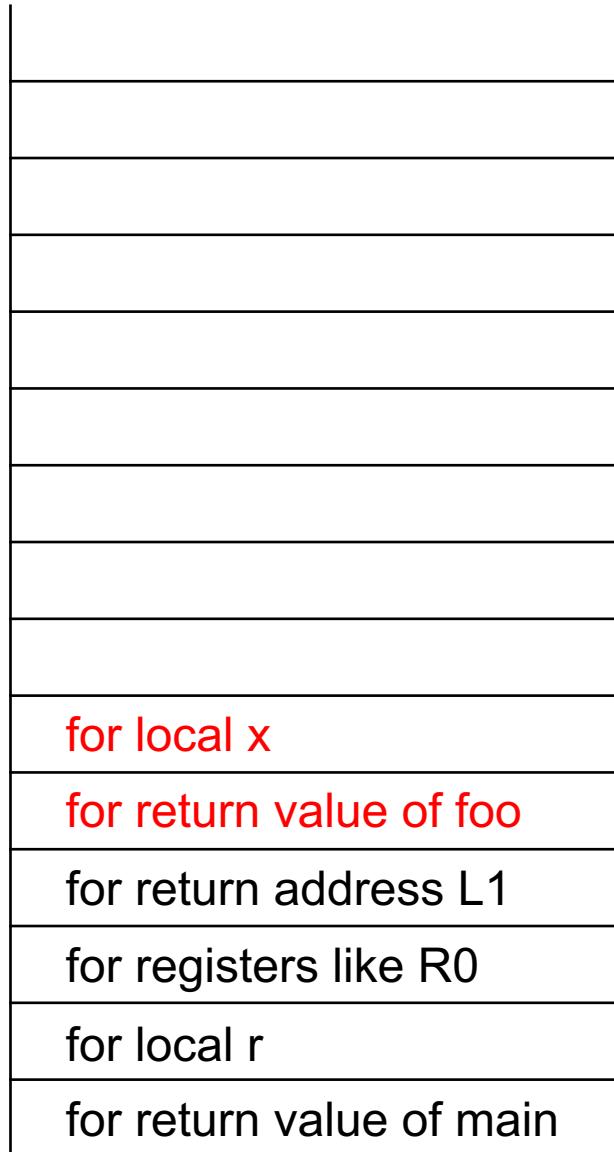
SUB SP, SP, #4 // record registers
ST 4(SP), R0

SUB SP, SP, #4 // record return addr
ST 4(SP), L₁
BR L₂

L₁ LD R0, 8(SP) // recover R0
LD R1, 0(SP) ST 12(SP), R1
ADD SP, SP, #8 // pop the space
LD R0, 4(SP) ST 8(SP), R0

ADD SP, SP, #8

SP → 600



```

1. int y, z;
2. int main() {
3.     int r;
4.     ...
5.     r = foo();
6.     ...
7.     return r;
8. }

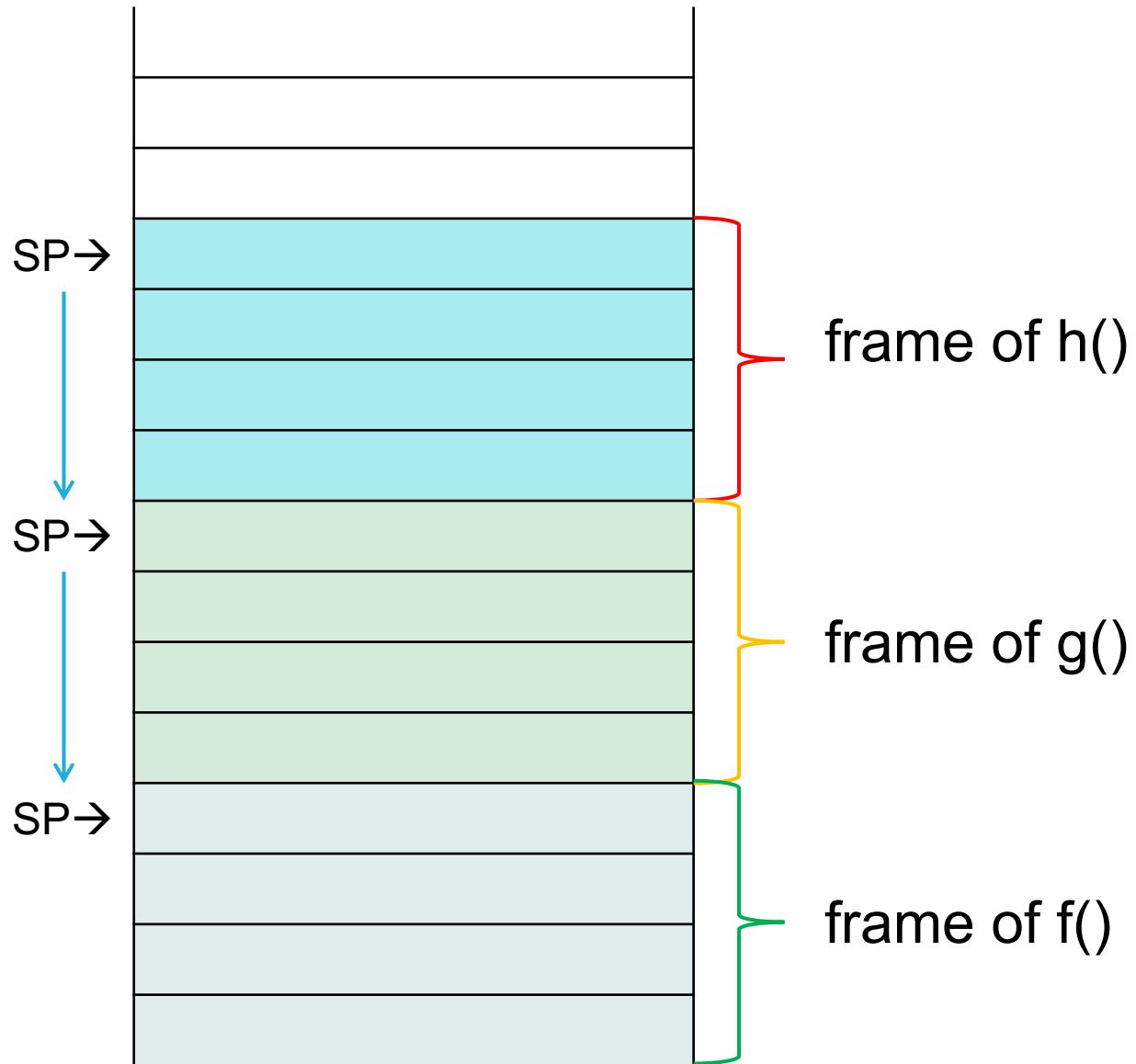
9. int foo() {
10.    int x;
11.    x = y - z;
12.    return x;
13. }

```

Jump via Return

- Call Stack

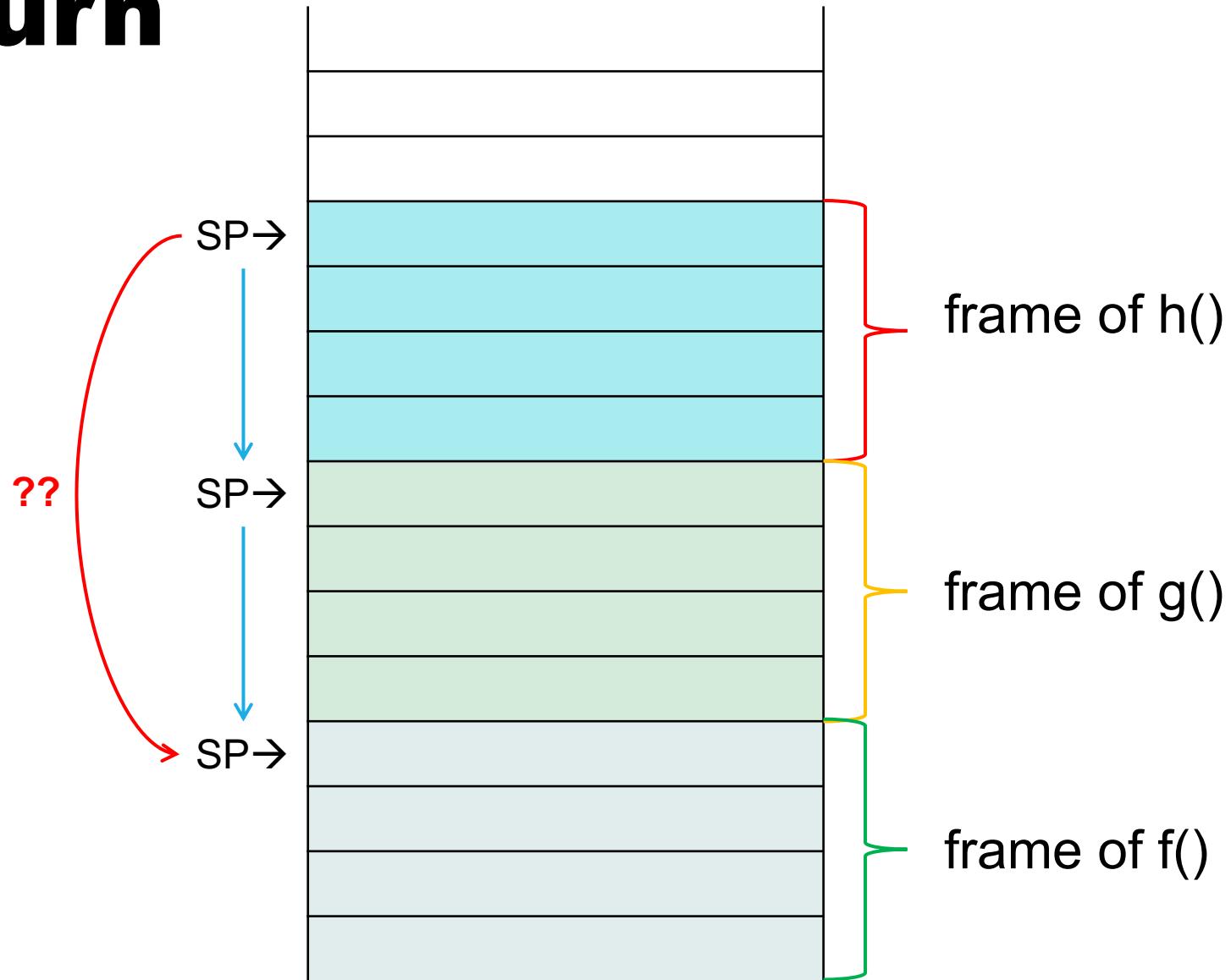
```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }  
  
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```



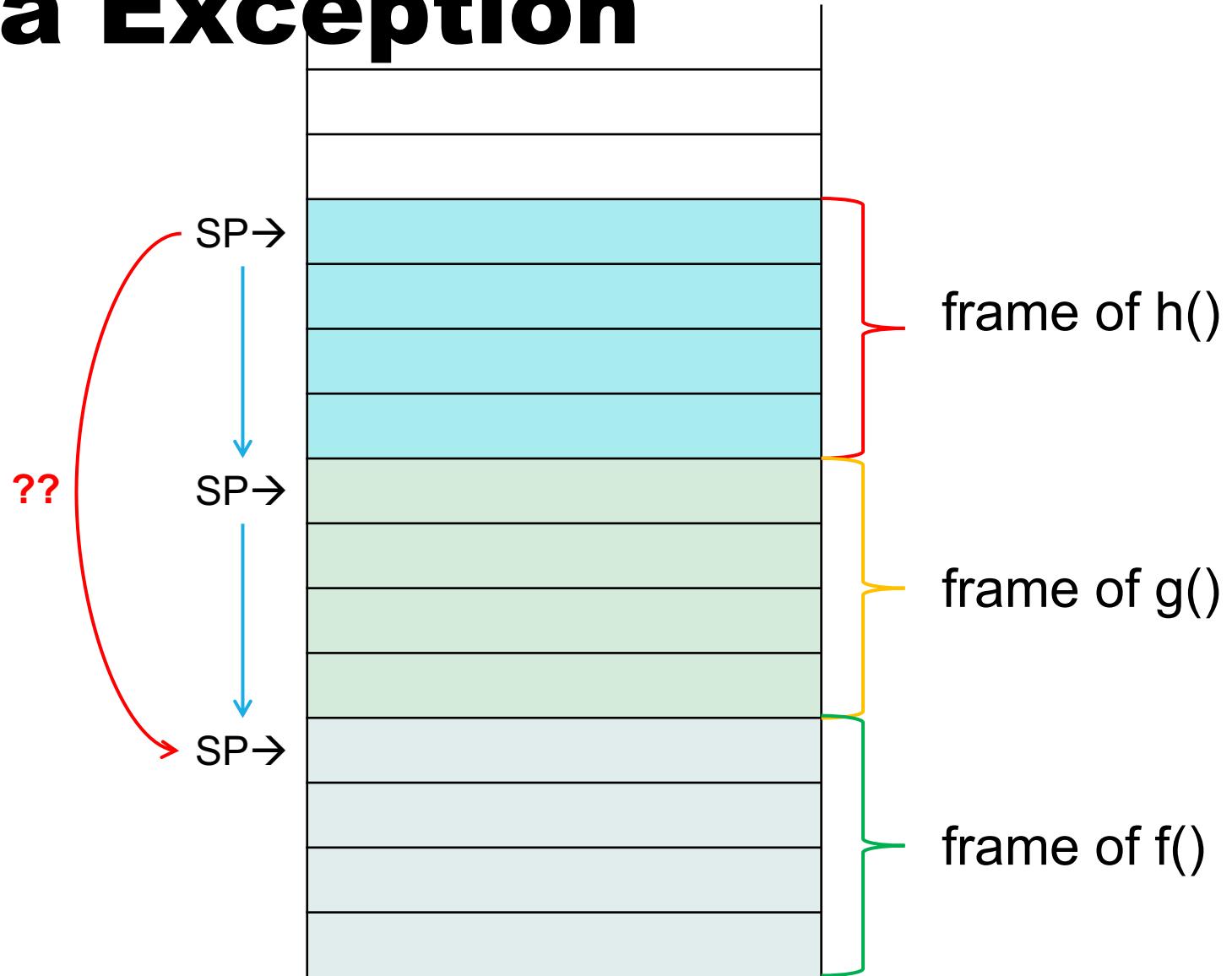
Jump via Return

- Call Stack

```
1. void f() {  
2.     ...  
3.     g();  
4.     ...  
5. }  
  
6. void g() {  
7.     ...  
8.     h();  
9.     ...  
10. }
```



Long Jump via Exception



Long Jump via Exception

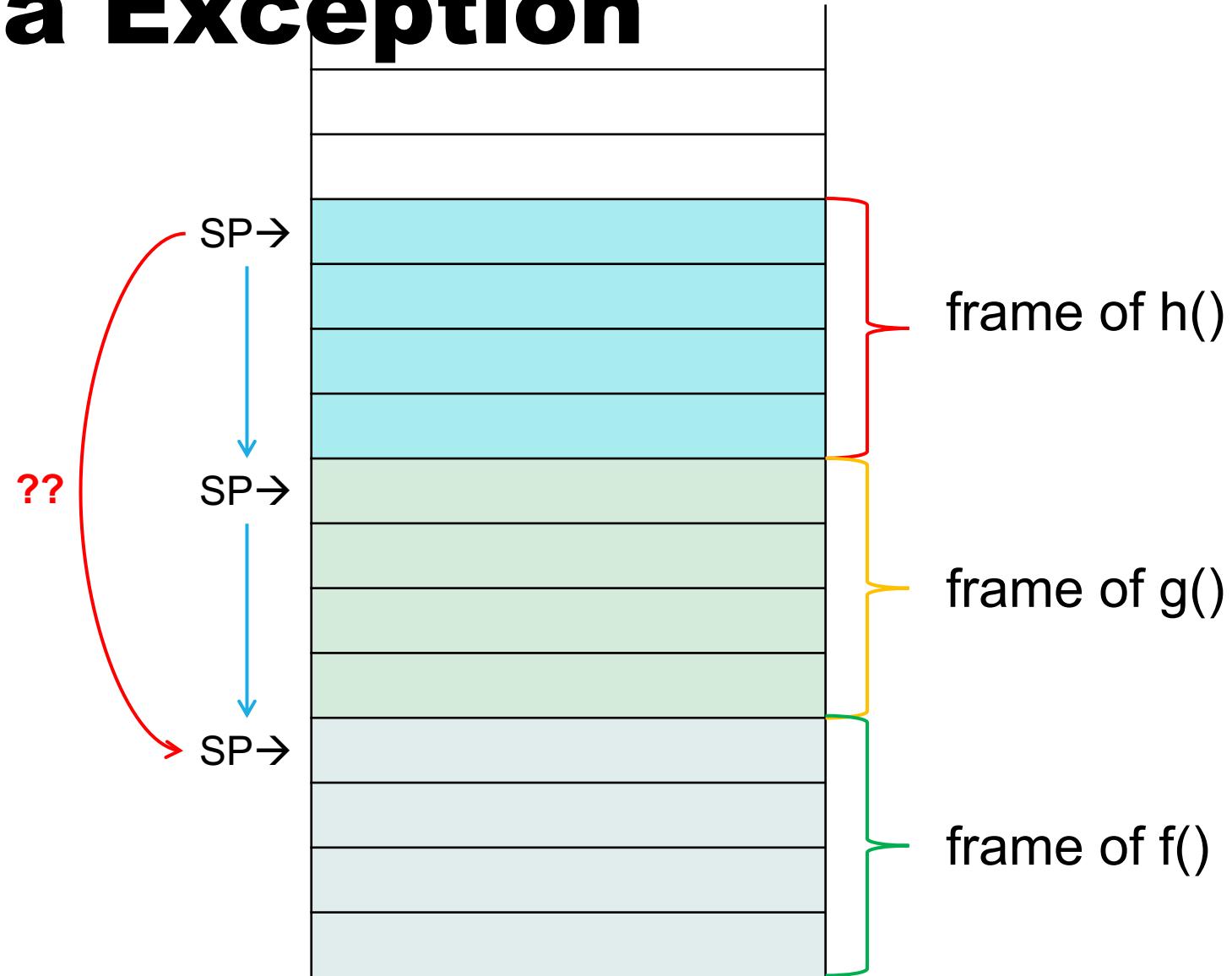
```

void h() throws Exception {
    throw new Exception();
}

void g () throws Exception {
    h();
}

void f() {
    try { g(); }
    catch (...) { ... }
}

```



Long Jump via Exception

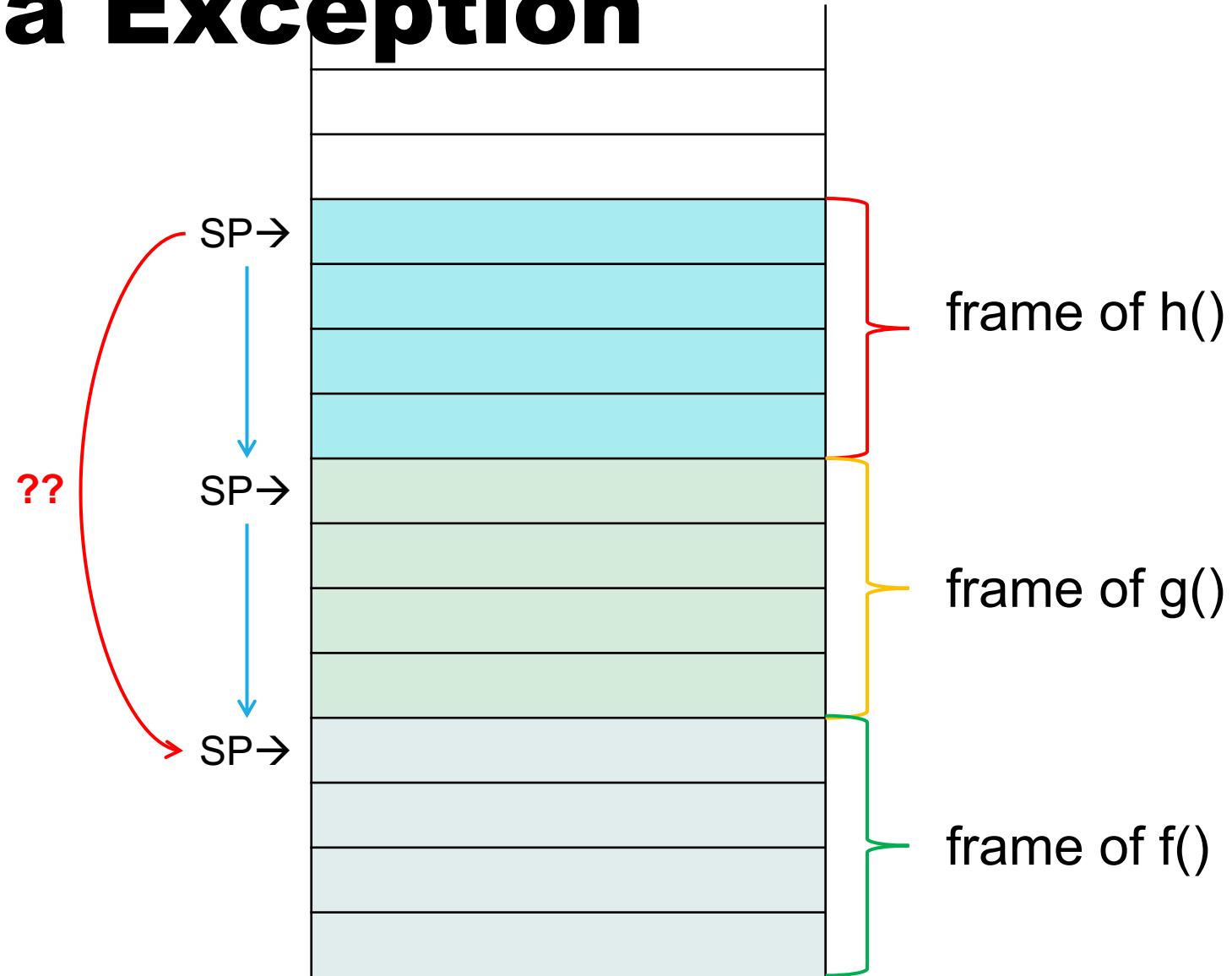
```

void h() throws Exception {
    throw new Exception();
}

void g () throws Exception {
    h();
}

void f() {
    try { g(); }
    catch (...) { ... }
}

```



Long Jump via Exception

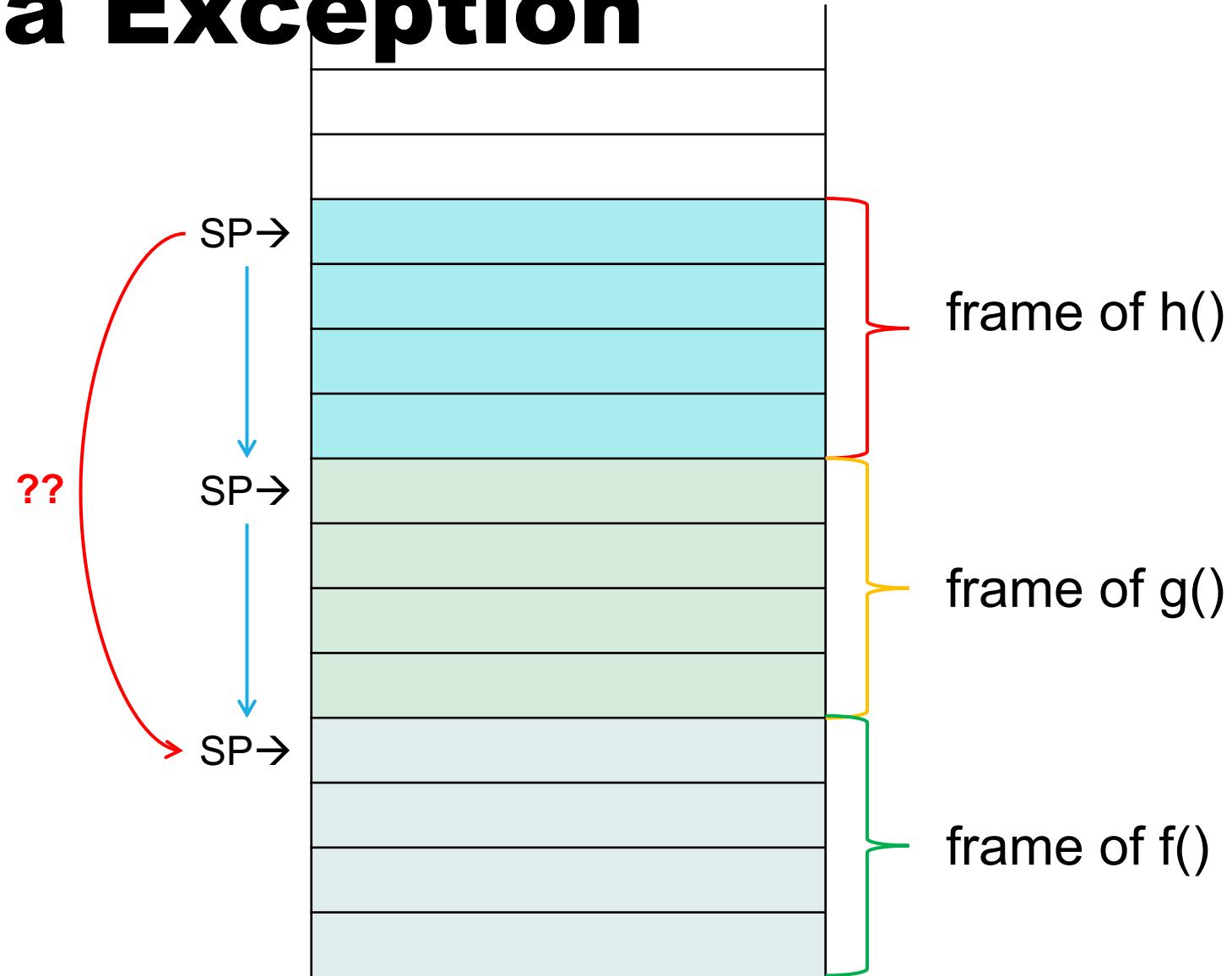
```

void h() throws Exception {
    throw new Exception();
}

void g () throws Exception {
    h();
}

void f() {
    try { g(); }
    catch (...) { ... }
}

```



Long Jump via Exception

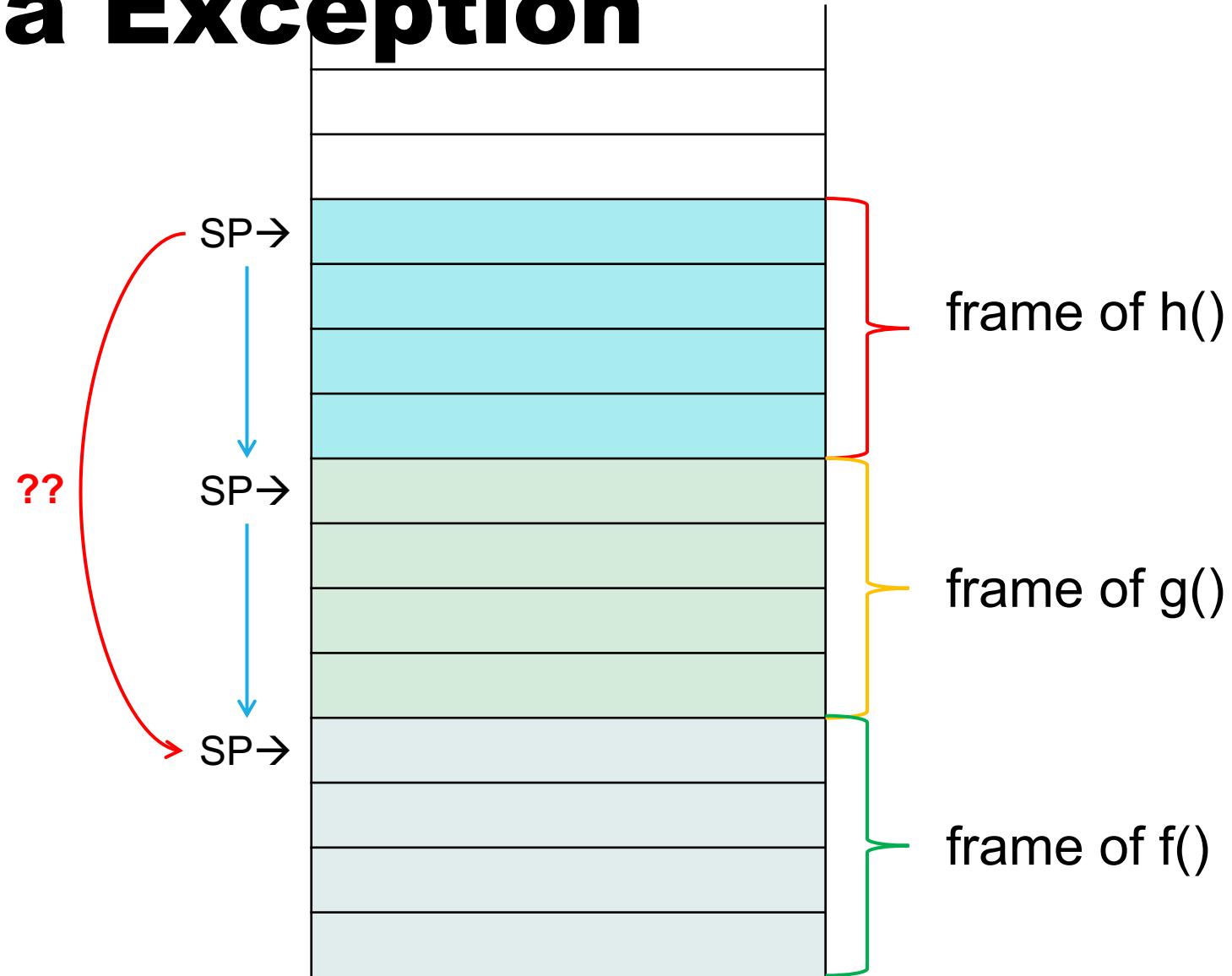
```

void h() throws Exception {
    throw new Exception();
}

void g () throws Exception {
    h();
}

void f() {
    try { g(); }
    catch (...) { ... }
}

```



Long Jump via Exception

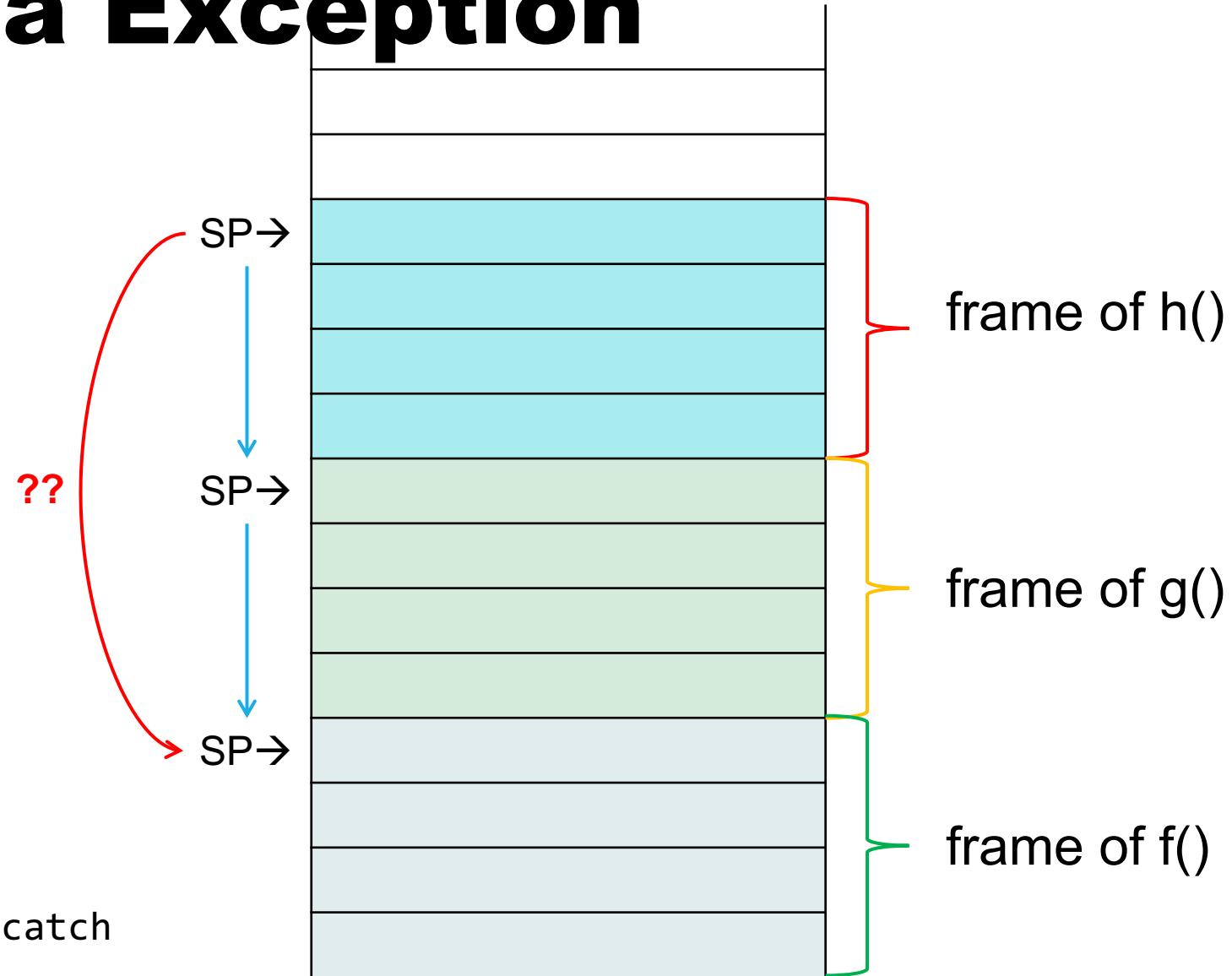
```

void h() throws Exception {
    throw new Exception();
}

void g () throws Exception {
    h();
}

void f() {
    try { g(); }
    catch (...) { ... }
}

```



Code generation:

- **try**: record the address of catch
- **throw**: jump to the address of catch

Heap Memory

- Dynamically allocate memory in heap
 - C: `void *malloc(size_t)`
 - Java/C++: the `new` operator

Heap Memory

- Dynamically allocate memory in heap
 - C: `void *malloc(size_t)`
 - Java/C++: the `new` operator
- Focus on the efficiency and minimizing segmentation

Heap Memory

- Dynamically allocate memory in heap
 - C: `void *malloc(size_t)`
 - Java/C++: the `new` operator
- Focus on the efficiency and minimizing segmentation
- Manual management (C/C++) or garbage collection (Java)

Heap Memory

- Dynamically allocate memory in heap
 - C: `void *malloc(size_t)`
 - Java/C++: the `new` operator
- Focus on the efficiency and minimizing segmentation
- Manual management (C/C++) or garbage collection (Java)
- *Refer to Chapter 7, the Dragon book.*

Summary

- Generate target code from three-address code containing
 - basic instructions
 - global variables
 - local variables
 - functions
 - function calls
 - ...

PART II-1: Control Flow Graph

Basic Block

- Partition three-address code to basic blocks
 - A basic block is a sequence of consecutive instructions

Basic Block

- Partition three-address code to basic blocks
 - A basic block is a sequence of consecutive instructions
 - The first instruction of each basic block is a leader
 - [1] The first instruction of a function is a leader
 - [2]
 - [3]

Basic Block

- Partition three-address code to basic blocks
 - A basic block is a sequence of consecutive instructions
 - The first instruction of each basic block is a leader
 - [1] The first instruction of a function is a leader
 - [2] Any instruction that is the target of a jump is a leader
 - [3]

Basic Block

- Partition three-address code to basic blocks
 - A basic block is a sequence of consecutive instructions
 - The first instruction of each basic block is a leader
 - [1] The first instruction of a function is a leader
 - [2] Any instruction that is the target of a jump is a leader
 - [3] Any instruction that follows a jump is a leader

Basic Block

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Basic Block

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Basic Block

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Basic Block

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Basic Block

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Basic Block

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Basic Block

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Basic Block

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Basic Block

1)	i = 1
2)	j = 1
3)	t1 = 10 * i
4)	t2 = t1 + j
5)	t3 = 8 * t2
6)	t4 = t3 - 88
7)	a[t4] = 0.0
8)	j = j + 1
9)	if j <= 10 goto (3)
10)	i = i + 1
11)	if i <= 10 goto (2)
12)	i = 1
13)	t5 = i - 1
14)	t6 = 88 * t5
15)	a[t6] = 1.0
16)	i = i + 1
17)	if i <= 10 goto (13)

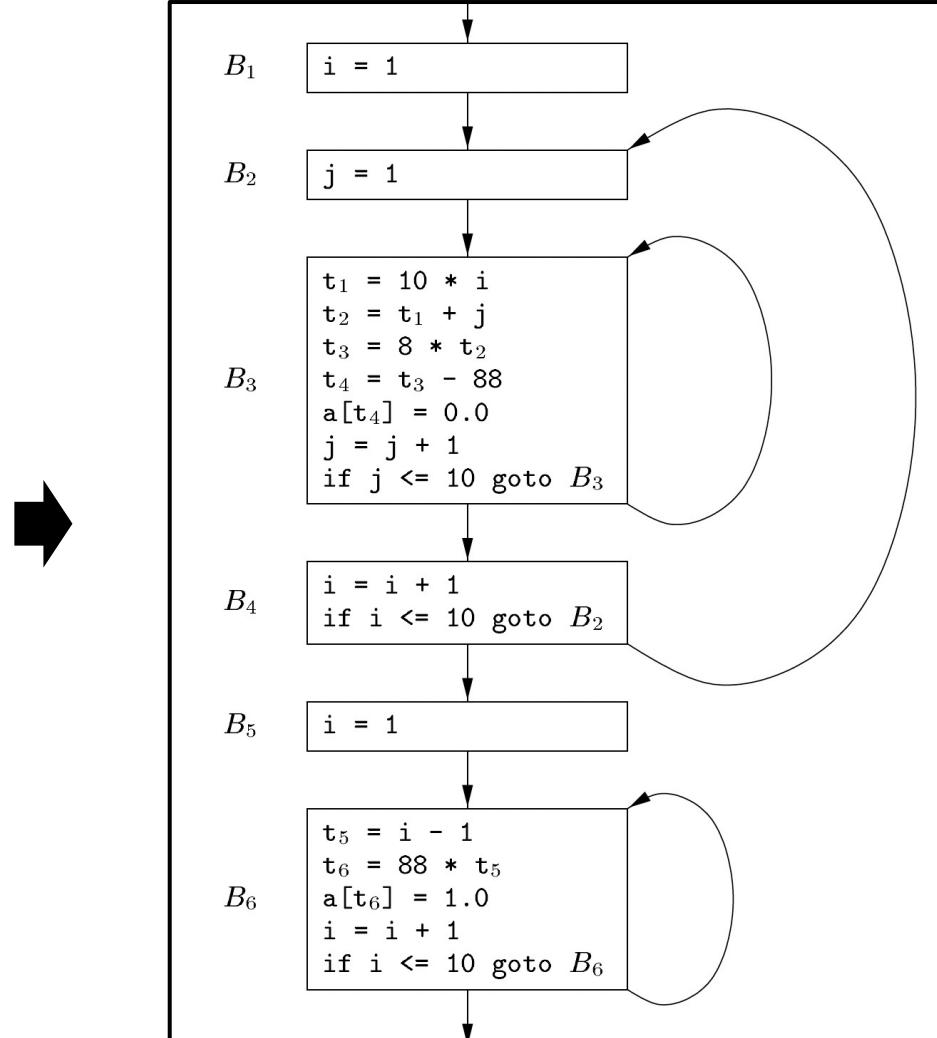
Basic Block

1)	i = 1
2)	j = 1
3)	t1 = 10 * i
4)	t2 = t1 + j
5)	t3 = 8 * t2
6)	t4 = t3 - 88
7)	a[t4] = 0.0
8)	j = j + 1
9)	if j <= 10 goto (3)
10)	i = i + 1
11)	if i <= 10 goto (2)
12)	i = 1
13)	t5 = i - 1
14)	t6 = 88 * t5
15)	a[t6] = 1.0
16)	i = i + 1
17)	if i <= 10 goto (13)

Control Flow Graph

```

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
    
```

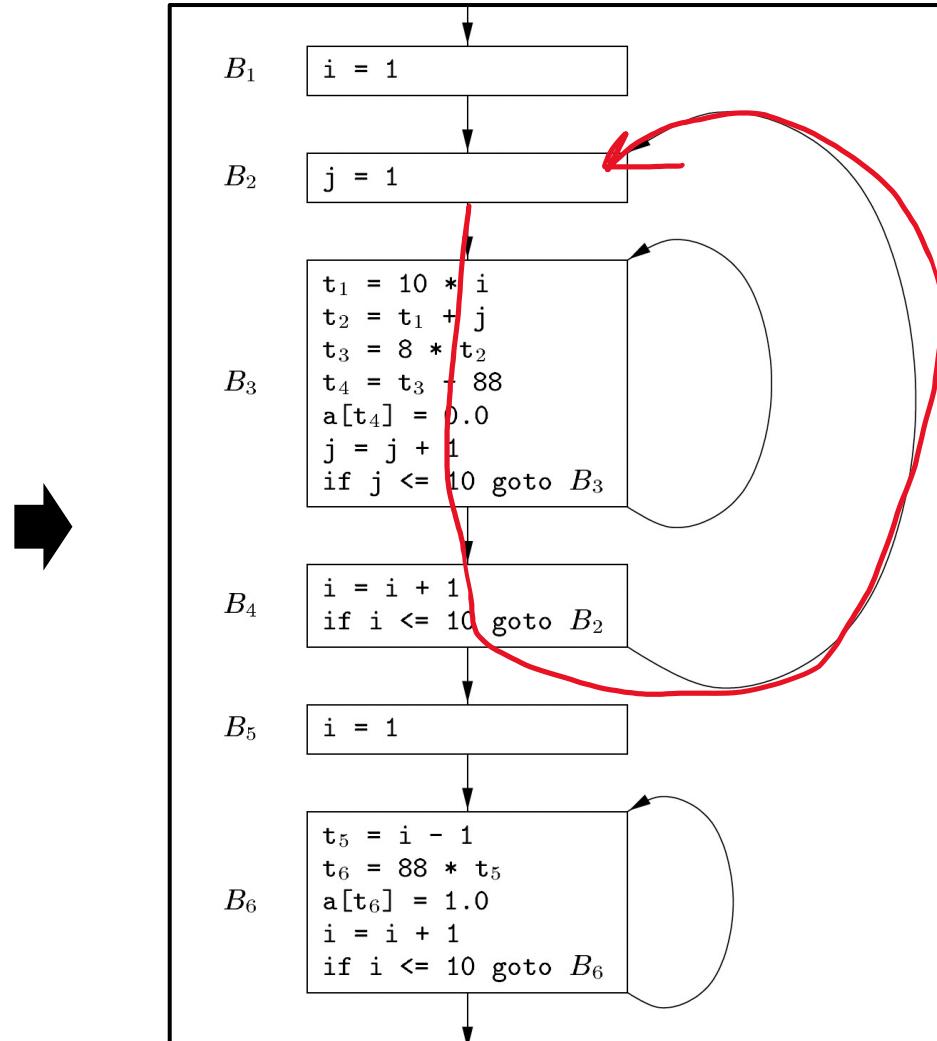


Loop

```

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

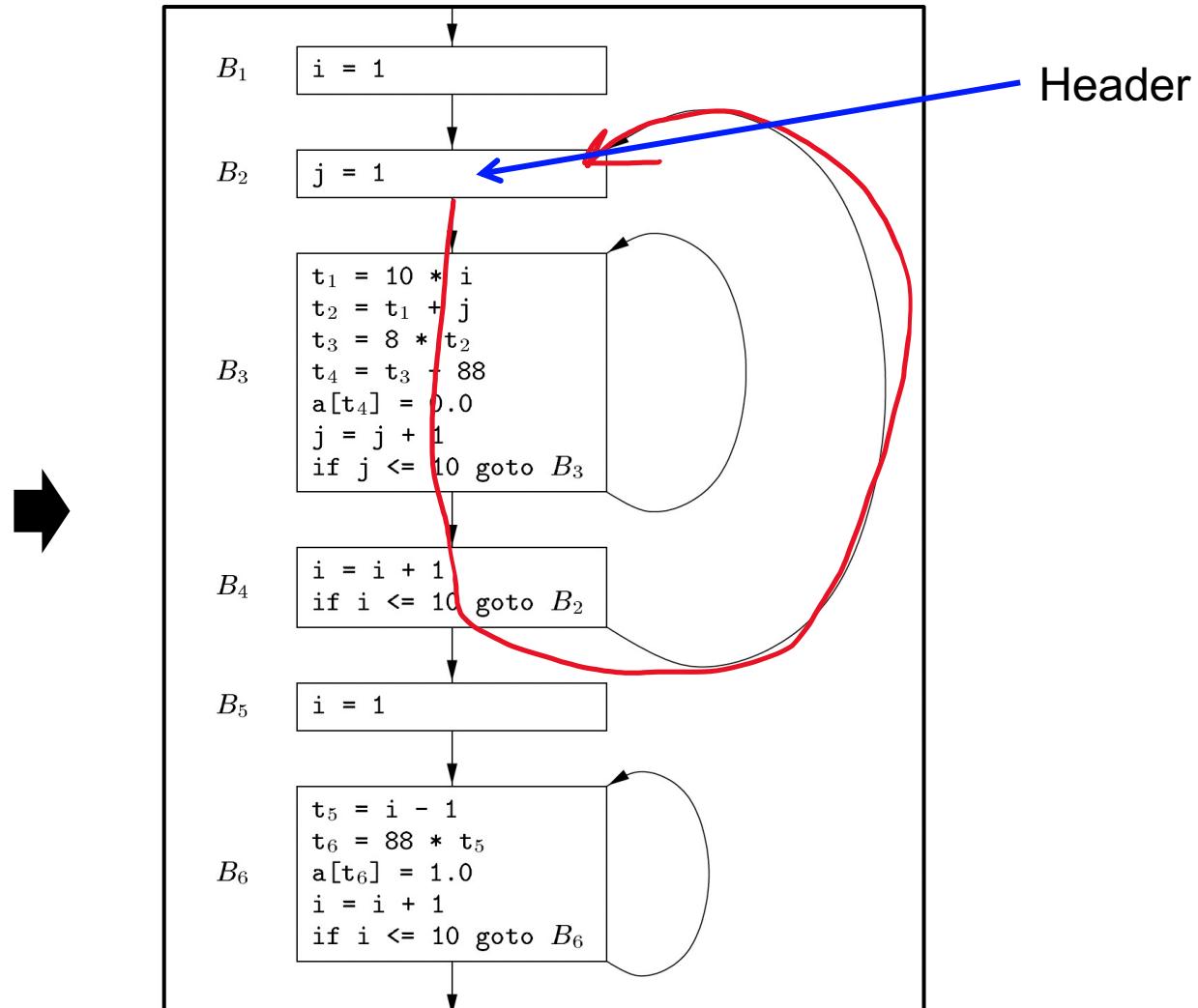
```



Loop

```

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
    
```

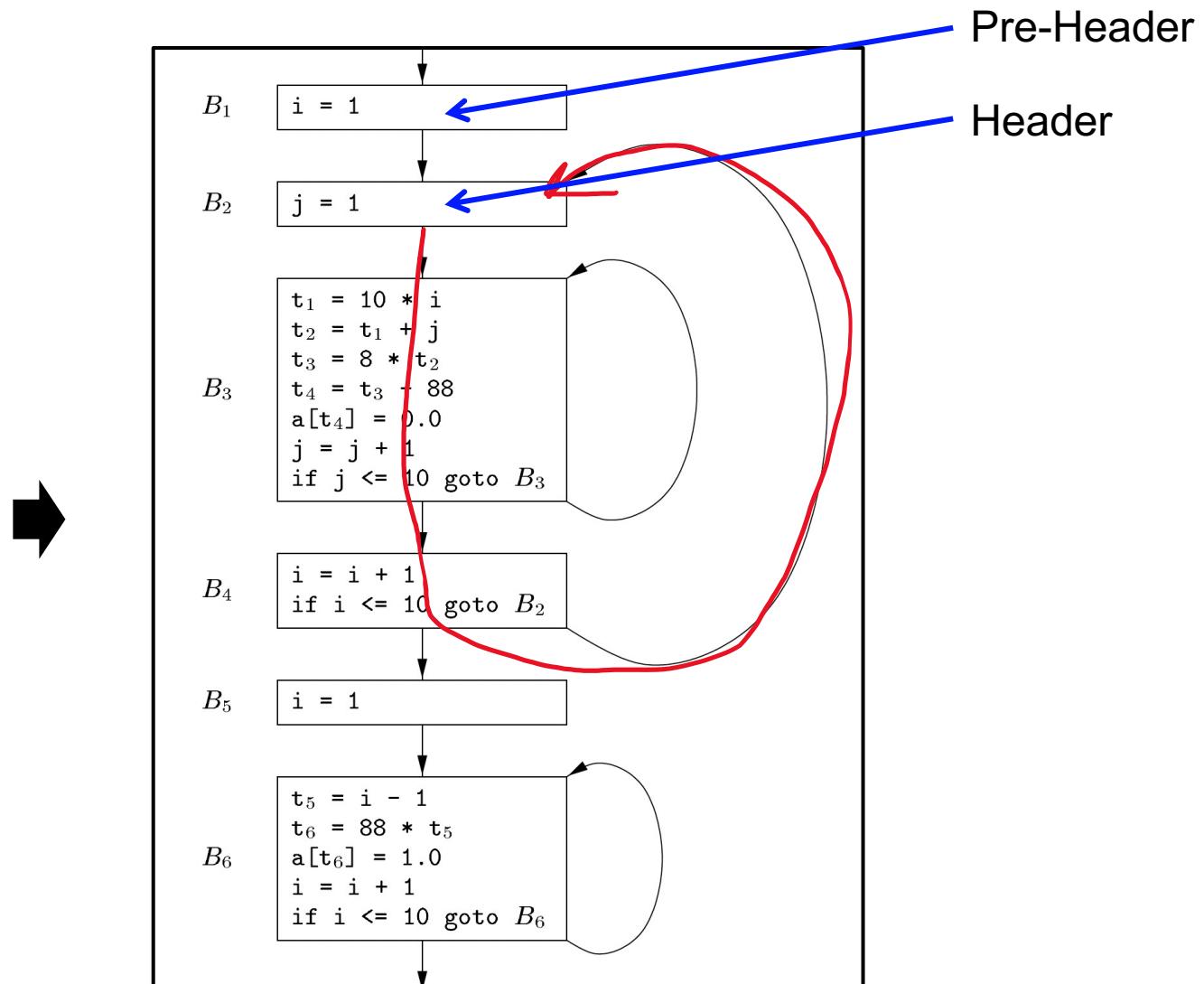


Loop

```

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

```

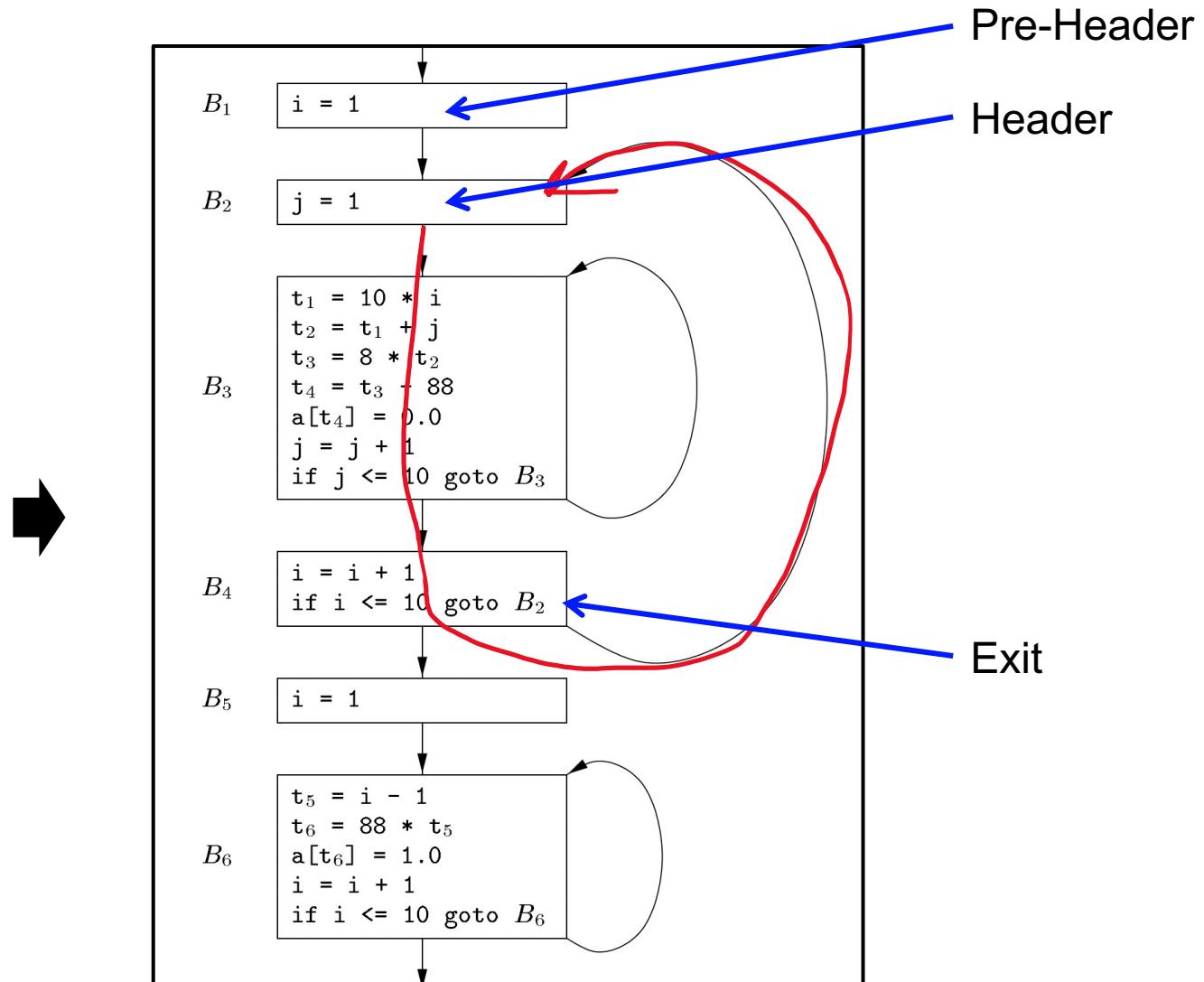


Loop

```

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

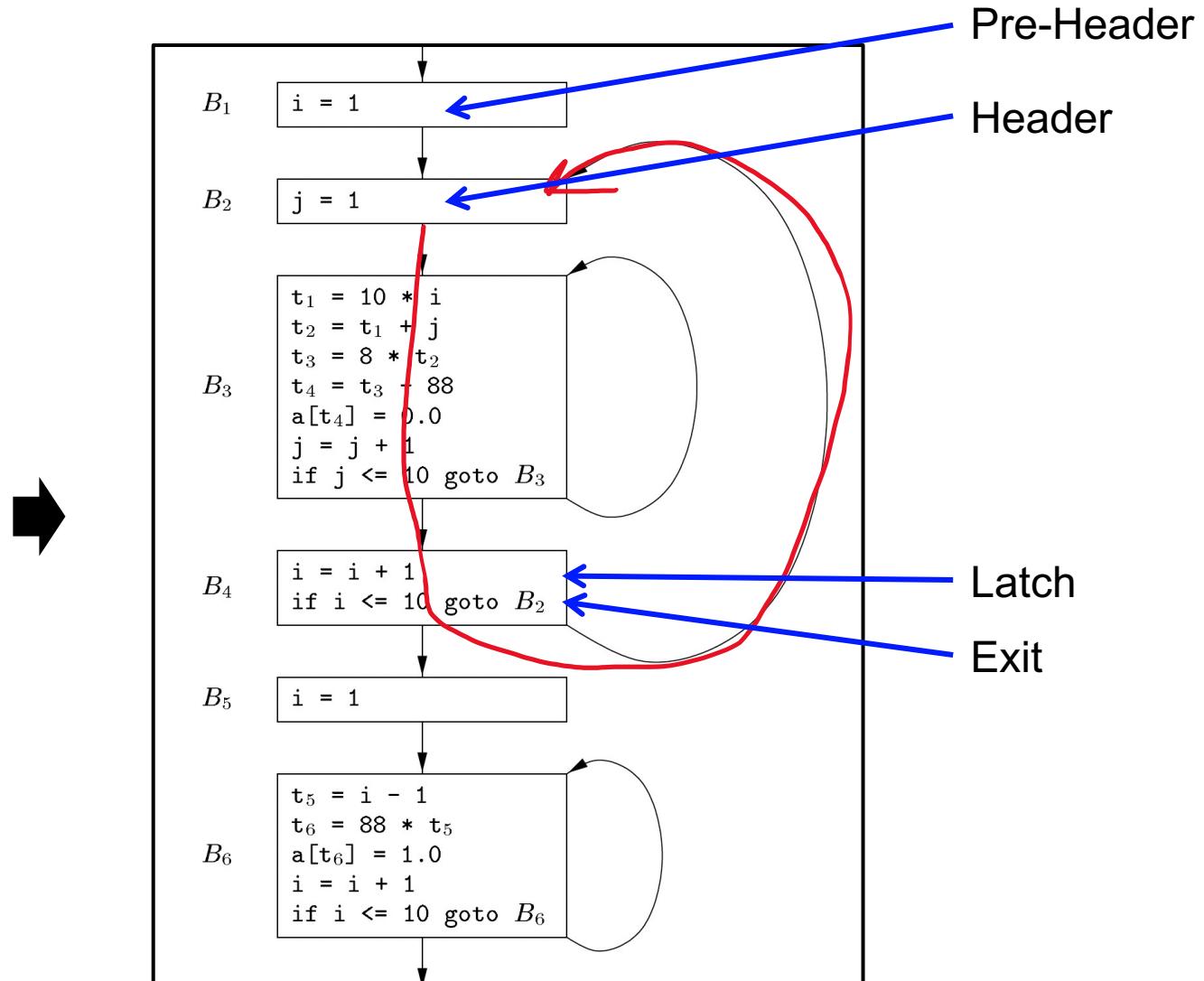
```



Loop

```

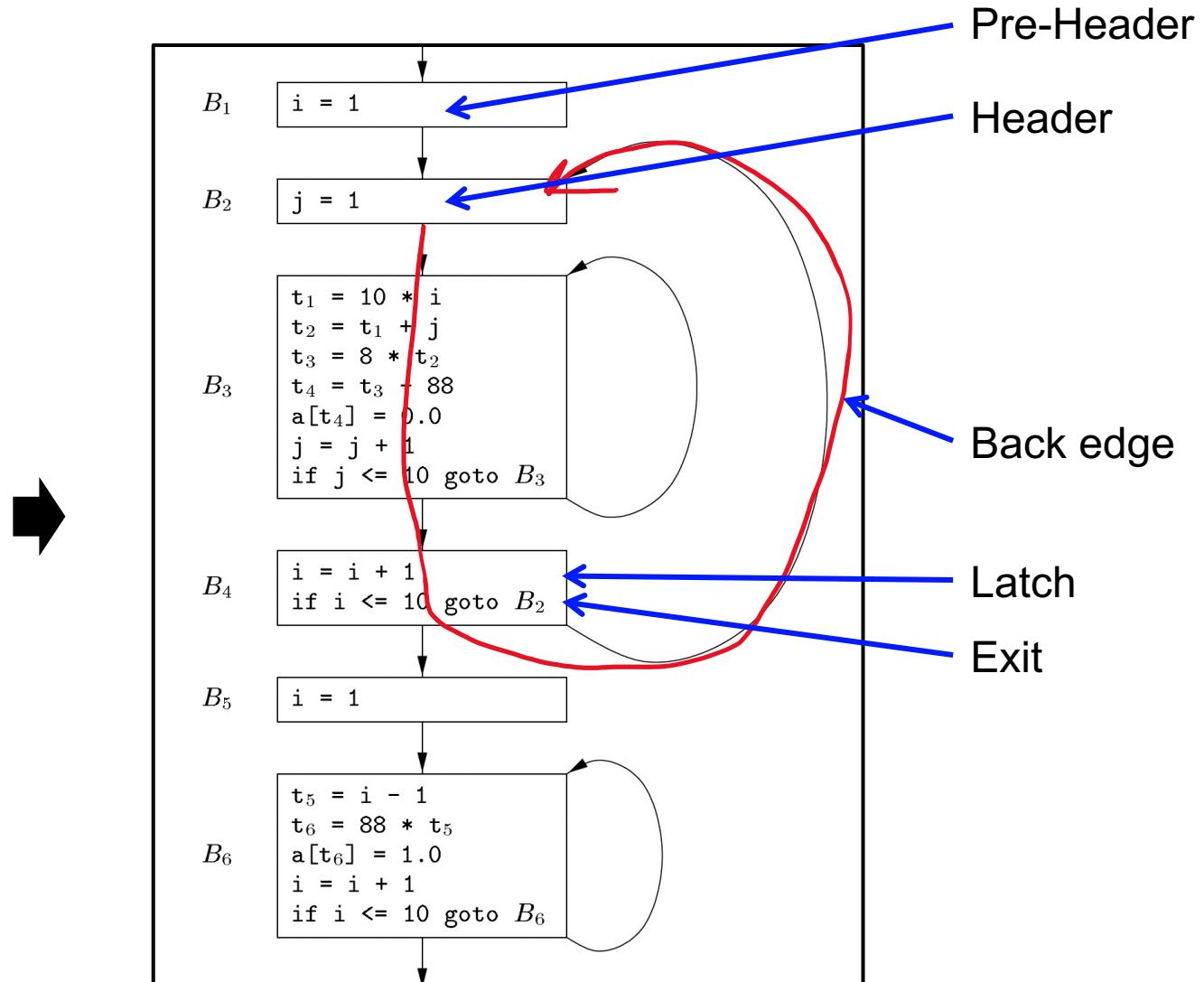
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
    
```



Loop

```

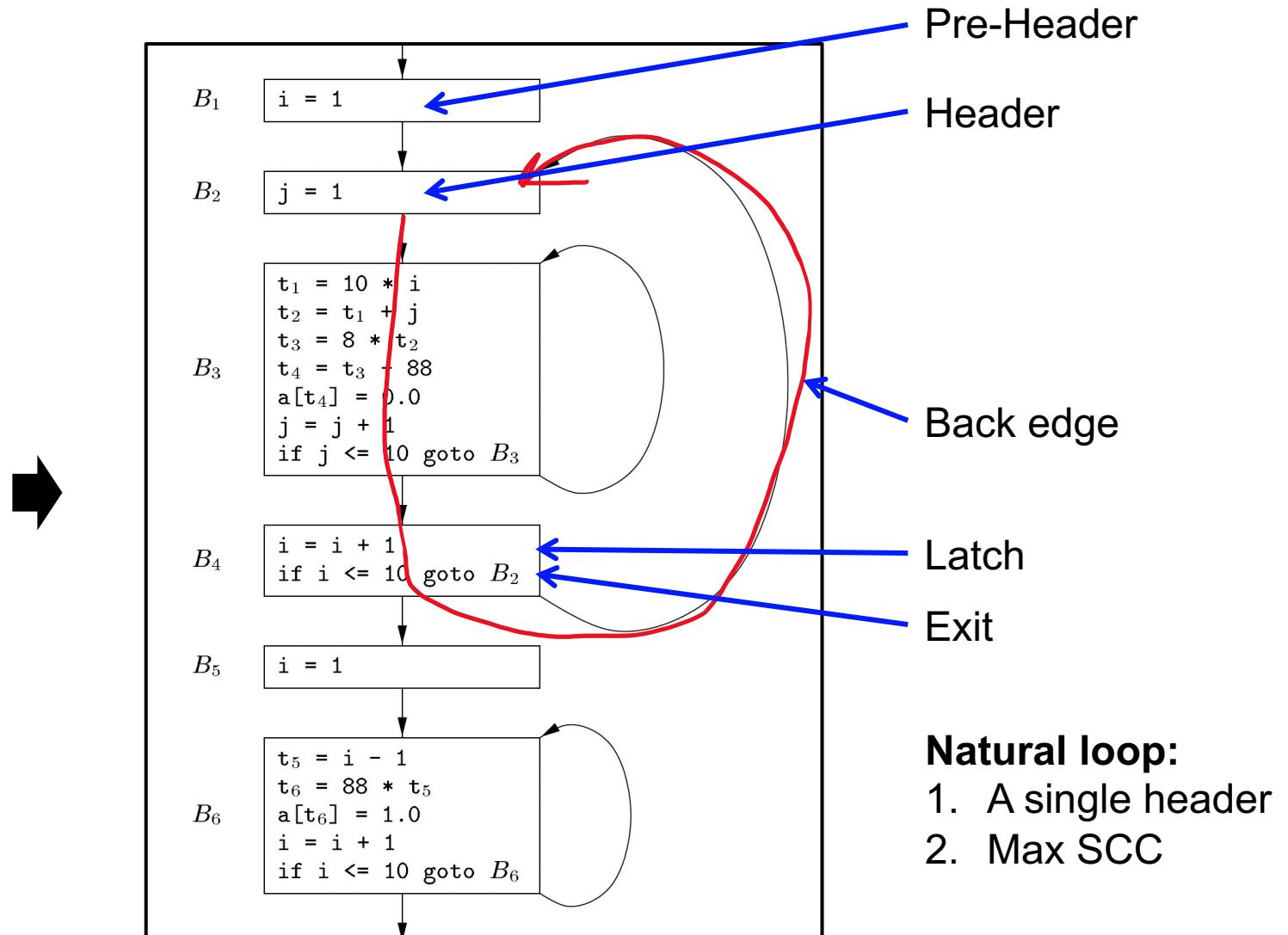
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
    
```



Loop

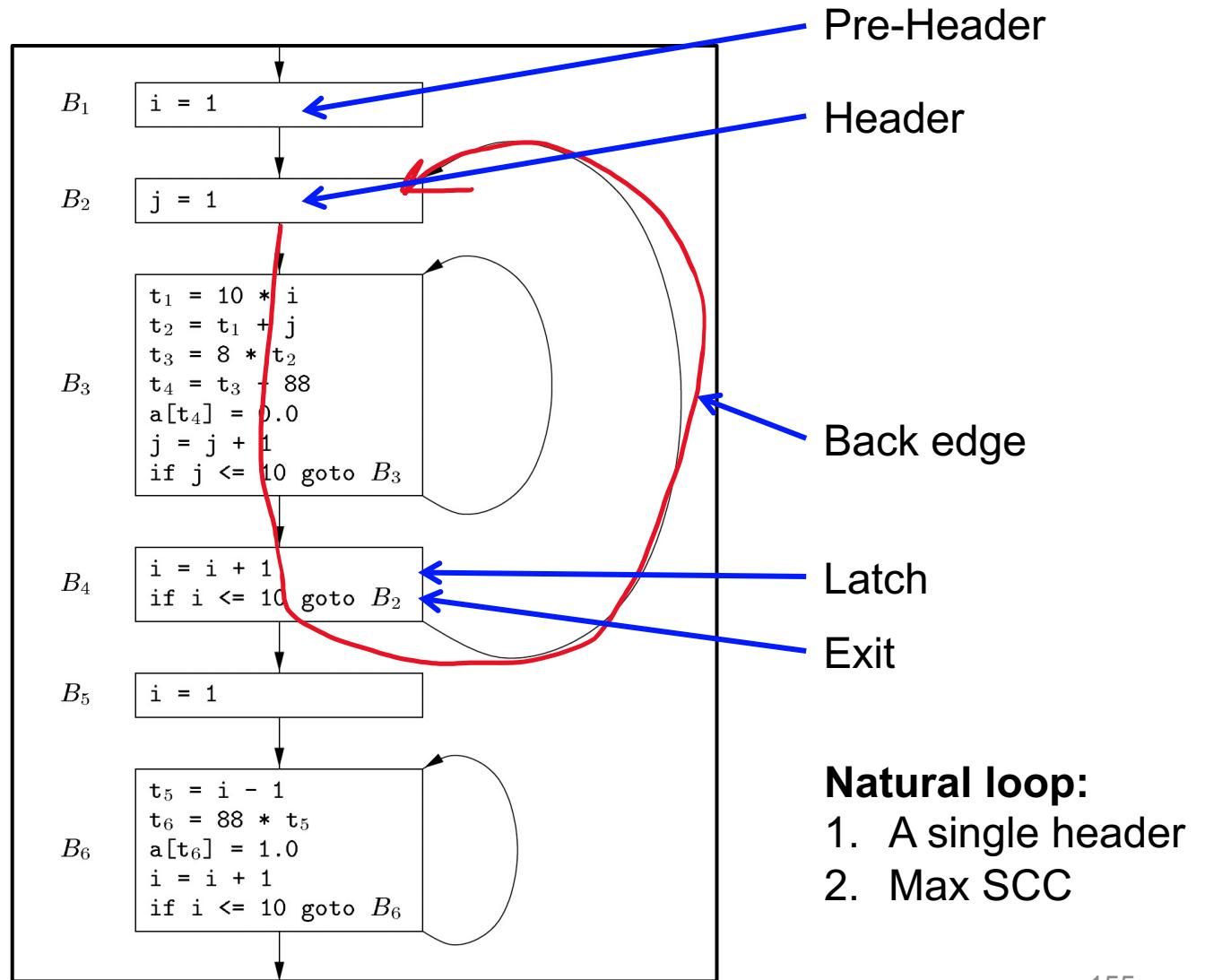
```

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
    
```



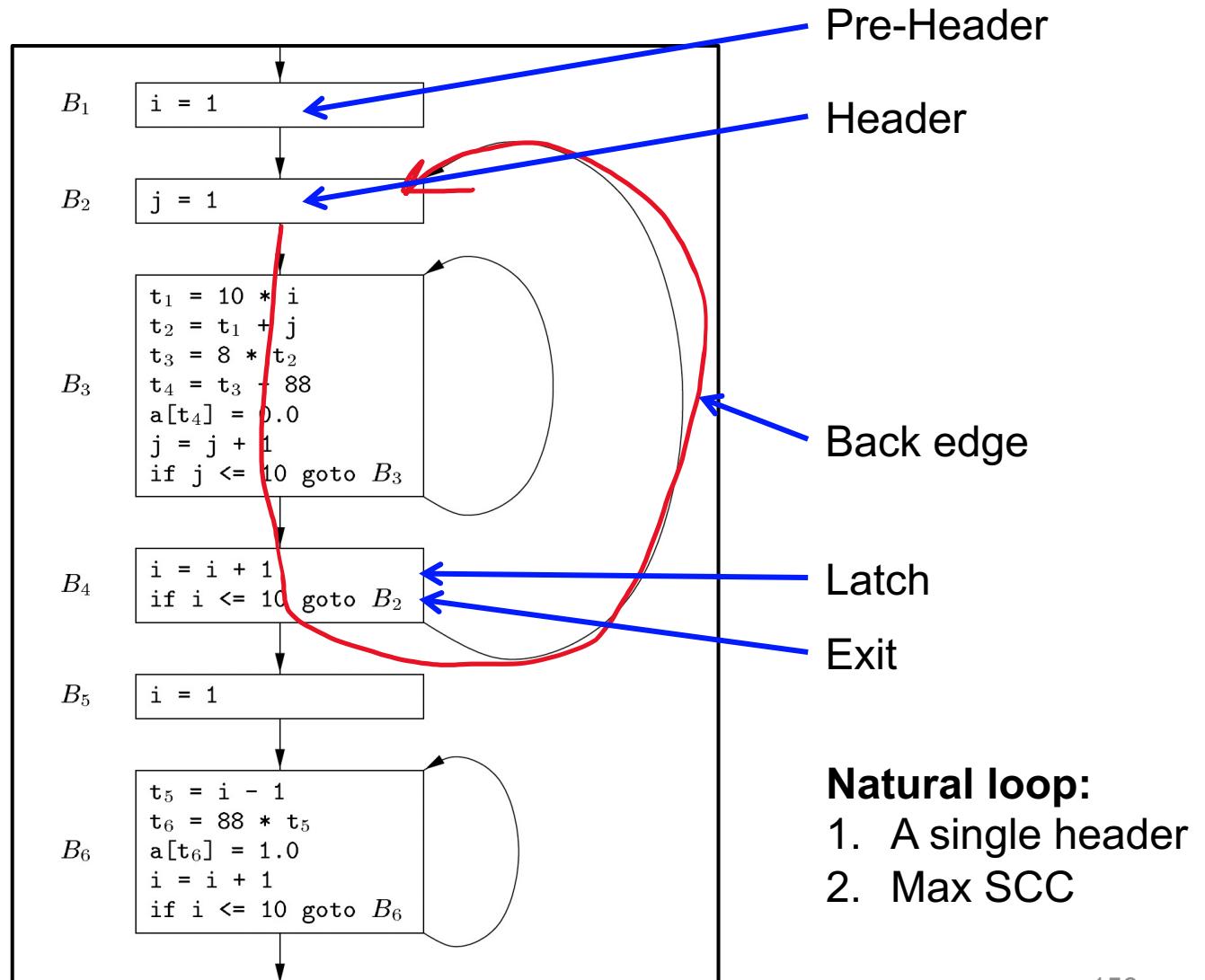
Dominance

- What does that mean for a single header?



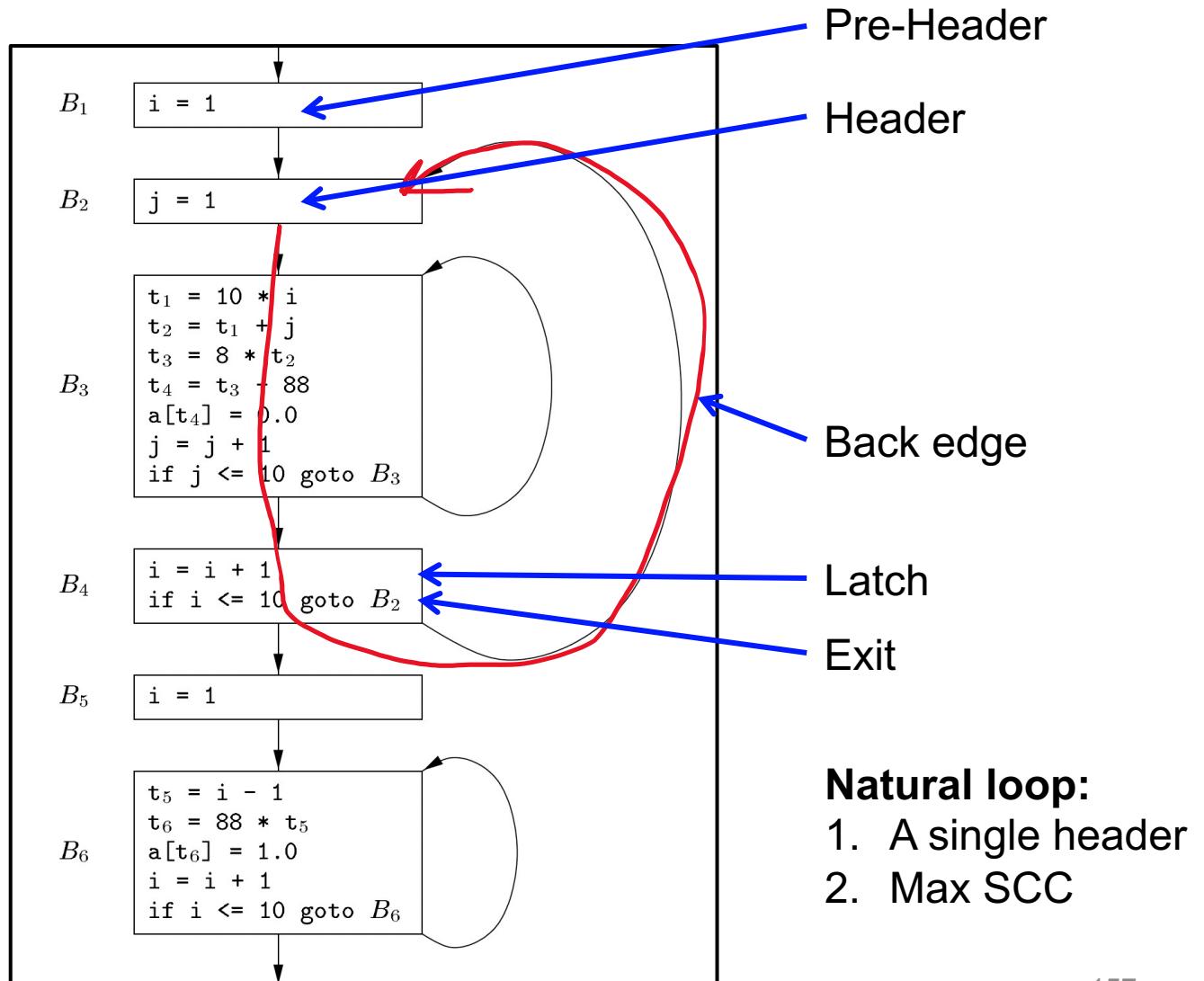
Dominance

- What does that mean for a single header?
- The header **dominates** all blocks in the loop.



Dominance

- What does that mean for a single header?
- The header **dominates** all blocks in the loop.
 - A **dom** B
 - if all paths from Entry to B goes through A



Post-Dominance

- A **dom** B
 - if all paths from Entry to B goes through A
- A **post-dom** B
 - if all paths from B to Exit goes through A

Post-Dominance

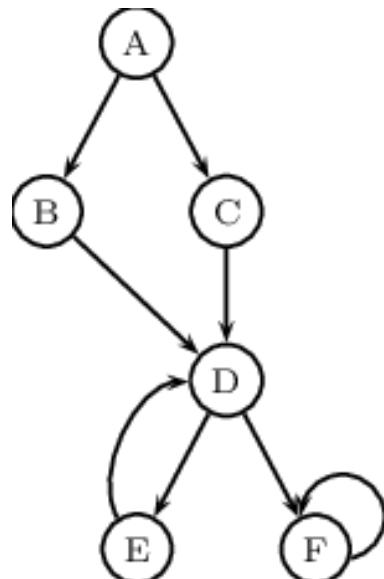
- A **dom** B
 - if all paths from Entry to B goes through A
- A **post-dom** B
 - if all paths from B to Exit goes through A
- **Strict** (post-)dominance – A (post-)**dom** B **but** $A \neq B$

Post-Dominance

- A **dom** B
 - if all paths from Entry to B goes through A
- A **post-dom** B
 - if all paths from B to Exit goes through A
- **Strict** (post-)dominance – A (post-)**dom** B **but** $A \neq B$
- **Immediate** dominance – A strict-dom B, but there's no C, such that A strict-dom C, C strict-dom B

Dominator Tree

- Almost linear time to build a dominator tree.
 - Node: Block
 - Edge: Immediate dom relation

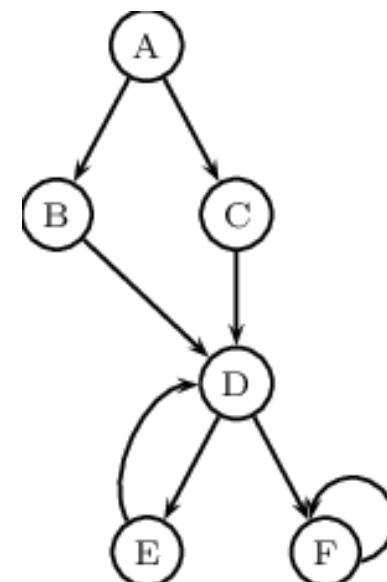


Try!

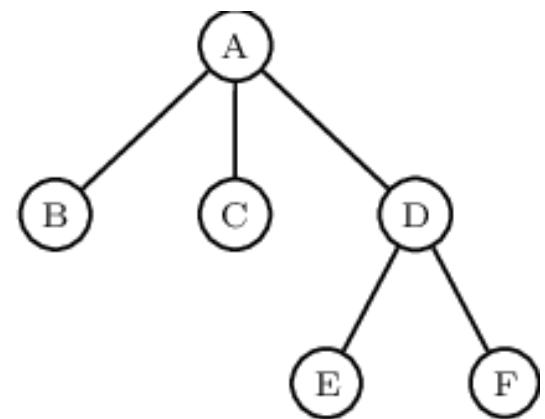
Flow Graph

Dominator Tree

- Almost linear time to build a dominator tree.
 - Node: Block
 - Edge: Immediate dom relation



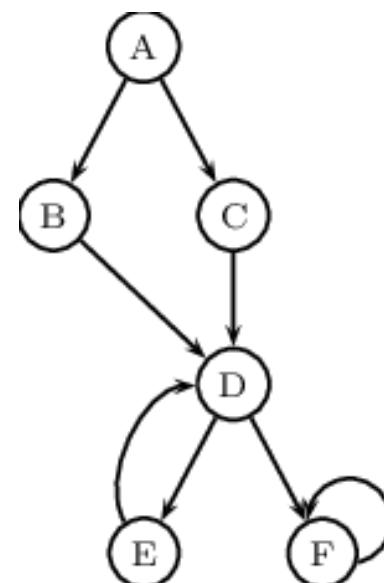
Flow Graph



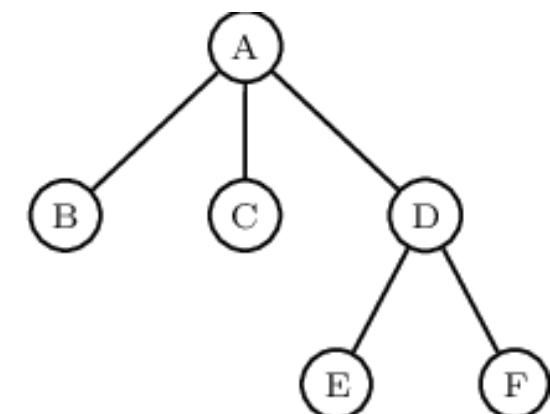
Dominator Tree

Dominator Tree

- Almost linear time to build a dominator tree.
 - Node: Block
 - Edge: Immediate dom relation
- (Nearest) common dominator
 - LCA – lowest common ancestor
 - Almost linear preprocessing time
 - Constant query time



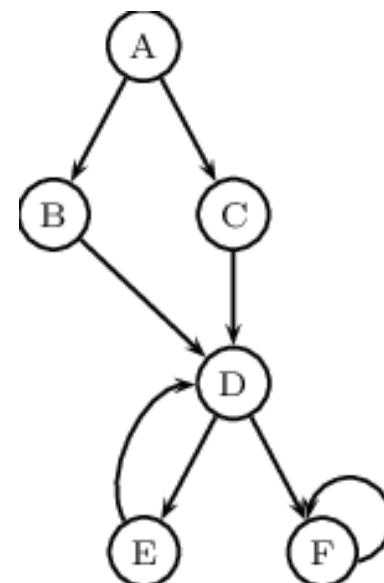
Flow Graph



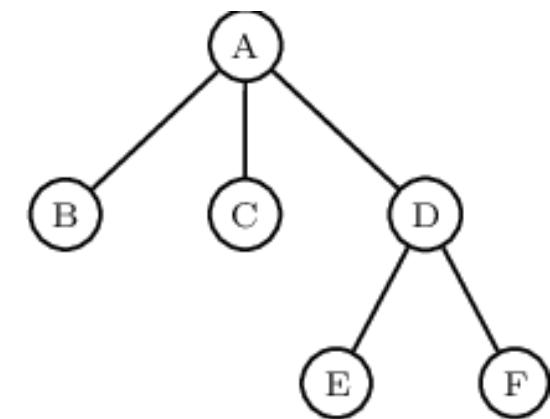
Dominator Tree

Dominator Tree

- Almost linear time to build a dominator tree.
 - Node: Block
 - Edge: Immediate dom relation
- (Nearest) common dominator
 - LCA – lowest common ancestor
 - Almost linear preprocessing time
 - Constant query time
 - **Application:** placing variable definition



Flow Graph

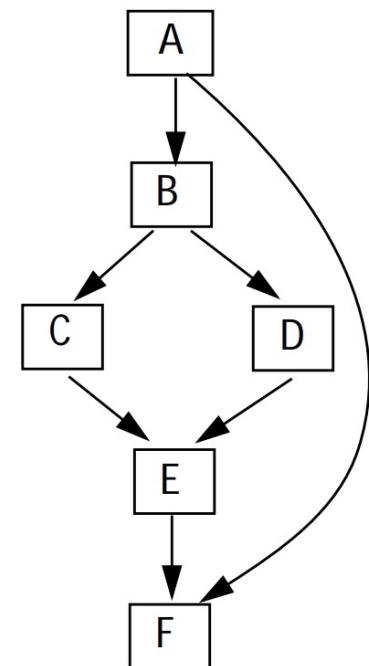


Dominator Tree

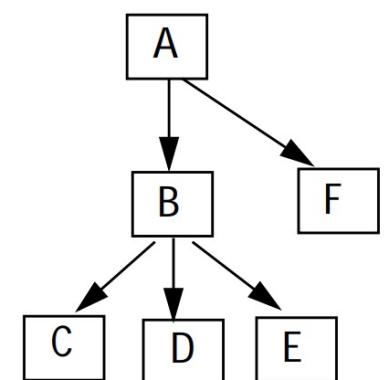
Dominance Frontier

- $DF(B) = \{ \dots \}$
 - The immediate successors of the blocks dominated by B
 - Not strictly dominated by B

Block	A	B	C	D	E	F
Dominance Frontier						



Control Flow Graph

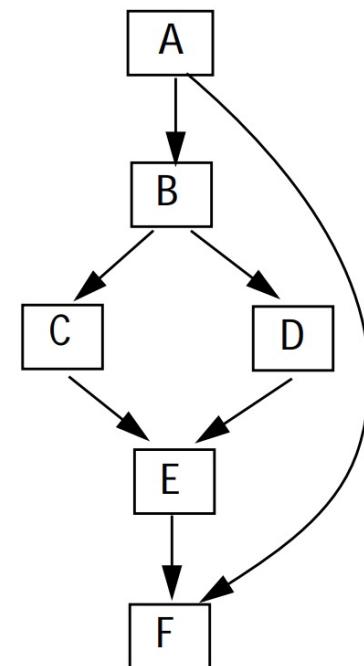


Dominator Tree

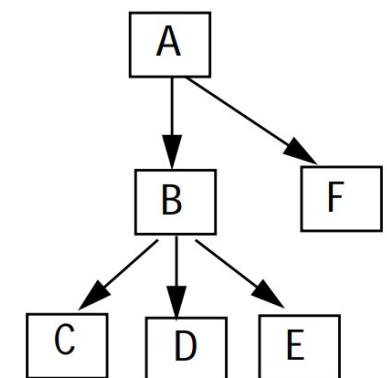
Dominance Frontier

- $DF(B) = \{ \dots \}$
 - The immediate successors of the blocks dominated by B
 - Not strictly dominated by B

Block	A	B	C	D	E	F
Dominance Frontier	{ }					



Control Flow Graph

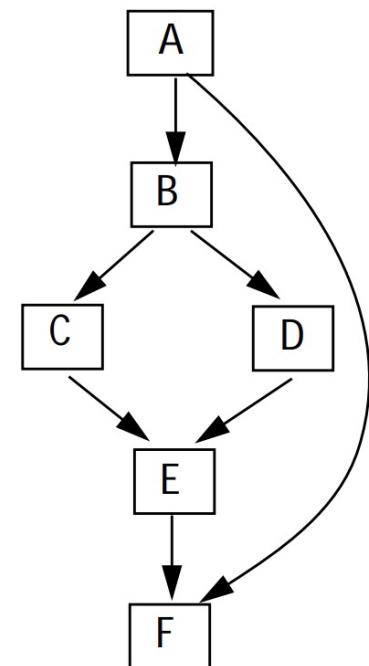


Dominator Tree

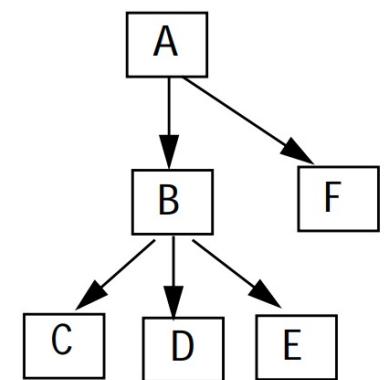
Dominance Frontier

- $DF(B) = \{ \dots \}$
 - The immediate successors of the blocks dominated by B
 - Not strictly dominated by B

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}				



Control Flow Graph

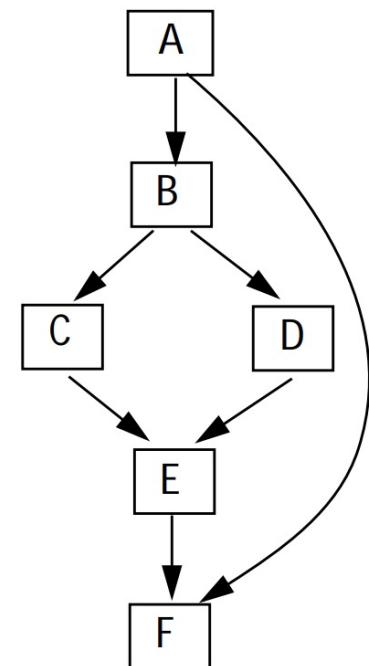


Dominator Tree

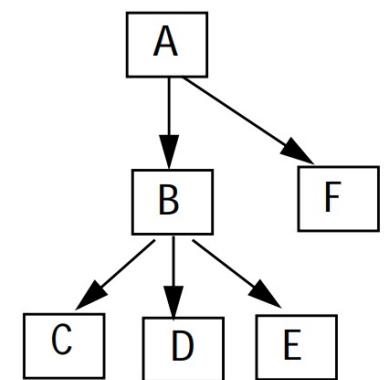
Dominance Frontier

- $DF(B) = \{ \dots \}$
 - The immediate successors of the blocks dominated by B
 - Not strictly dominated by B

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}	{E}			



Control Flow Graph

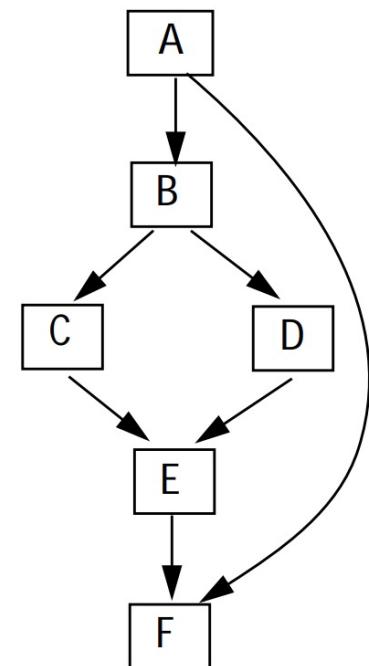


Dominator Tree

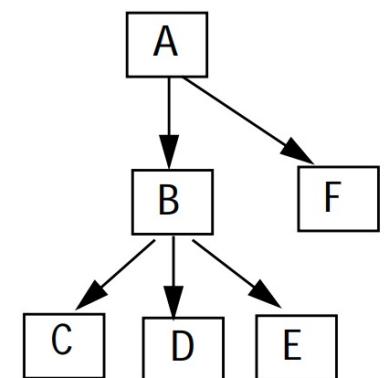
Dominance Frontier

- $DF(B) = \{ \dots \}$
 - The immediate successors of the blocks dominated by B
 - Not strictly dominated by B

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}	{E}	{E}		



Control Flow Graph

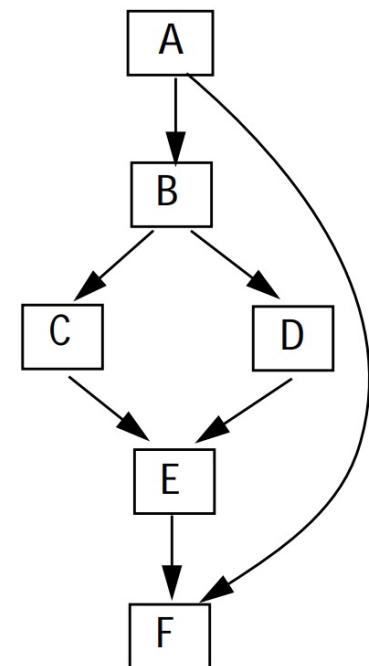


Dominator Tree

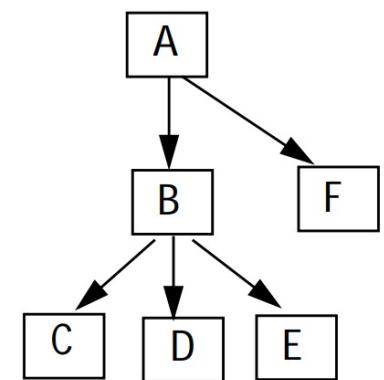
Dominance Frontier

- $DF(B) = \{ \dots \}$
 - The immediate successors of the blocks dominated by B
 - Not strictly dominated by B

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}	{E}	{E}	{F}	



Control Flow Graph

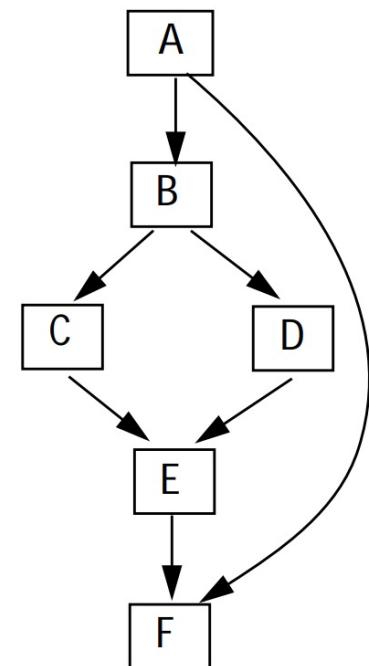


Dominator Tree

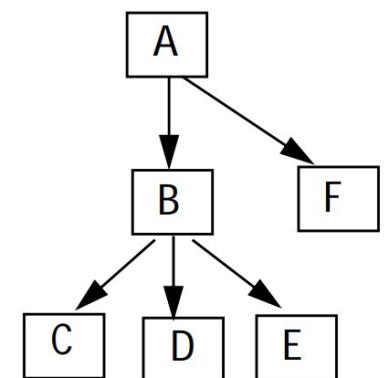
Dominance Frontier

- $DF(B) = \{ \dots \}$
 - The immediate successors of the blocks dominated by B
 - Not strictly dominated by B

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}	{E}	{E}	{F}	{ }



Control Flow Graph



Dominator Tree

Dominance Frontier

- Extend the definition of DF to a set of blocks
 - $\text{DF}(\mathbf{Bset}) = \bigcup_{B \in \text{Bset}} \text{DF}(B)$

Iterated Dominance Frontier (IDF)

- Iterated DF of Bset
 - $DF_1 = DF(Bset); Bset = Bset \cup DF_1$
 - $DF_2 = DF(Bset); Bset = Bset \cup DF_2$
 -

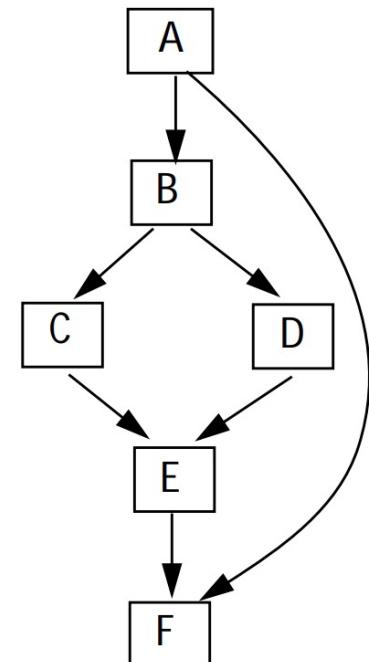
Iterated Dominance Frontier (IDF)

- Iterated DF of Bset
 - $DF_1 = DF(Bset); Bset = Bset \cup DF_1$
 - $DF_2 = DF(Bset); Bset = Bset \cup DF_2$
 -
 - until fixed point!

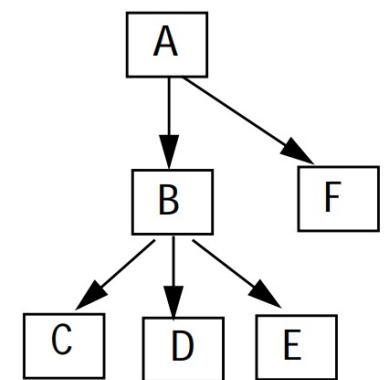
Iterated Dominance Frontier (IDF)

- Iterated DF of Bset
 - $DF_1 = DF(Bset); Bset = Bset \cup DF_1$
 - $DF_2 = DF(Bset); Bset = Bset \cup DF_2$
 -
 - until fixed point!
- $IDF(\{A, B, C\}) = \{ ? \}$

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}	{E}	{E}	{F}	{ }



Control Flow Graph



Dominator Tree

IDF vs. SSA

- What is SSA?

IDF vs. SSA

- What is SSA?
- Where should we insert φ to translate IR to SSA?

IDF vs. SSA

- Where should we insert φ to translate IR to SSA?
- By definition, merge point of multiple paths for a variable

```
if ( flag ) x = -1; else x = 1;  
y = x * a;
```

```
if ( flag ) x1 = -1; else x2 = 1;  
x3 =  $\varphi$  (x1, x2);  
y = x3 * a
```

IDF vs. SSA

- Where should we insert φ to translate IR to SSA?
- By definition, merge point of multiple paths for a variable
 - **Too complex, path explosion**

```
if ( flag ) x = -1; else x = 1;  
y = x * a;
```

```
if ( flag ) x1 = -1; else x2 = 1;  
x3 =  $\varphi$  (x1, x2);  
y = x3 * a
```

IDF vs. SSA

- Where should we insert φ to translate IR to SSA?
- By definition, merge point of multiple paths for a variable
- **The merge point of multiple definitions of a variable**

```
if ( flag ) x = -1; else x = 1;  
y = x * a;
```

```
if ( flag ) x1 = -1; else x2 = 1;  
x3 =  $\varphi$  (x1, x2);  
y = x3 * a
```

IDF vs. SSA

- Definition of a variable can be passed to blocks

IDF vs. SSA

- Definition of a variable can be passed to blocks
 - dominated by the definition or

IDF vs. SSA

- Definition of a variable can be passed to blocks
 - dominated by the definition or
 - in the IDF, *e.g.*, $\text{IDF}(\{A, B, C\}) = \{ E, F \}$

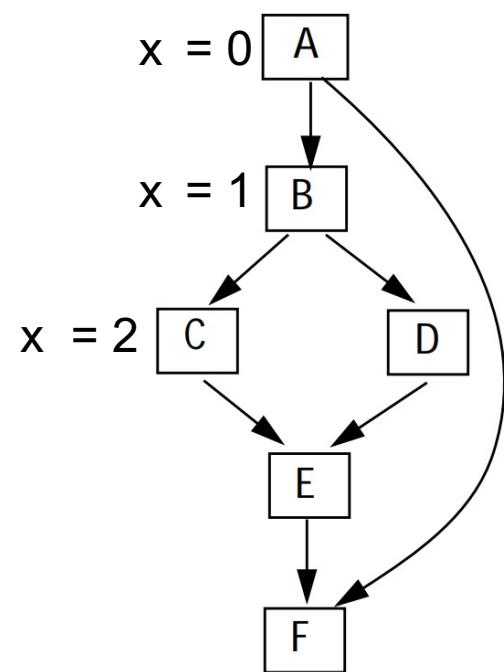
IDF vs. SSA

- Definition of a variable can be passed to blocks
 - dominated by the definition or
 - in the IDF, *e.g.*, $\text{IDF}(\{A, B, C\}) = \{ E, F \}$
 - where φ are inserted

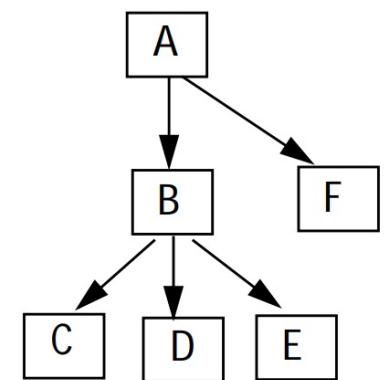
IDF vs. SSA

- Definition of a variable can be passed to blocks
 - dominated by the definition or
 - in the IDF, e.g., $\text{IDF}(\{A, B, C\}) = \{ E, F \}$
 - where φ are inserted

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}	{E}	{E}	{F}	{}



Control Flow Graph

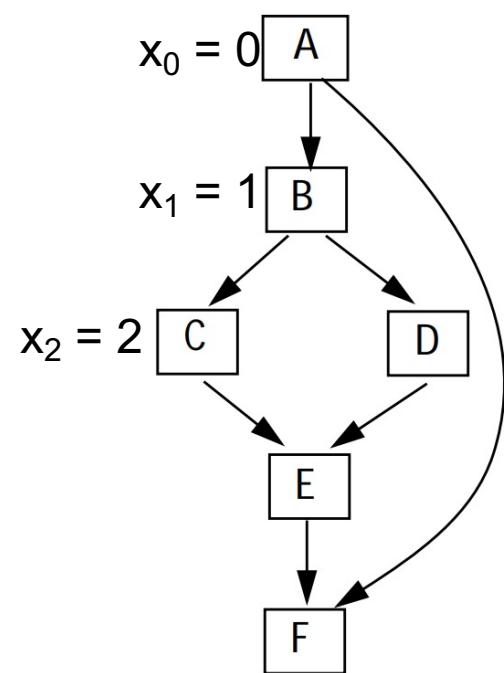


Dominator Tree

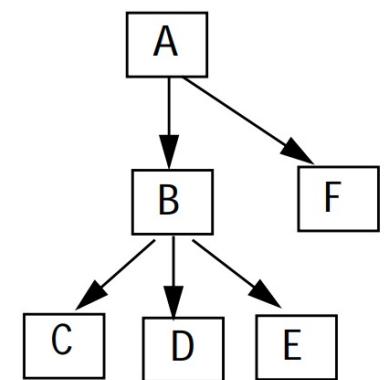
IDF vs. SSA

- Definition of a variable can be passed to blocks
 - dominated by the definition or
 - in the IDF, e.g., $\text{IDF}(\{A, B, C\}) = \{ E, F \}$
 - where φ are inserted

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}	{E}	{E}	{F}	{}



Control Flow Graph

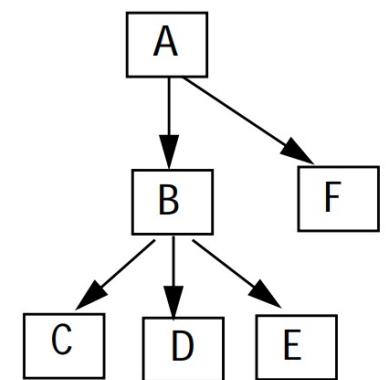
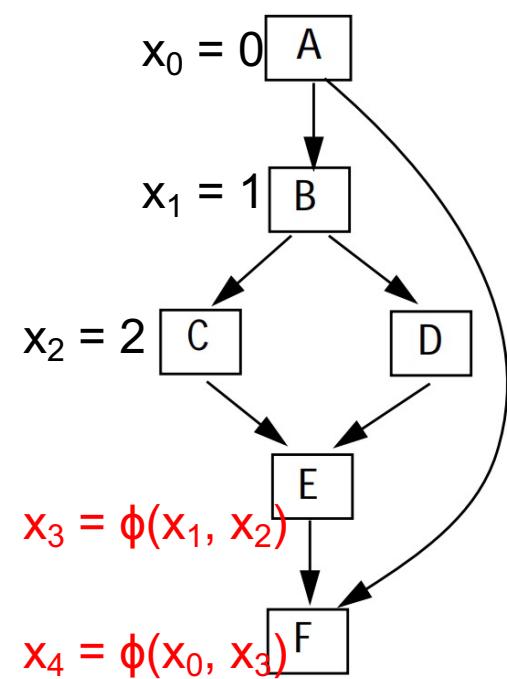


Dominator Tree

IDF vs. SSA

- Definition of a variable can be passed to blocks
 - dominated by the definition or
 - in the IDF, e.g., $\text{IDF}(\{A, B, C\}) = \{ E, F \}$
 - where ϕ are inserted

Block	A	B	C	D	E	F
Dominance Frontier	{ }	{F}	{E}	{E}	{F}	{ }



Dominator Tree

Control Flow Graph

Dominance vs. Ctrl Dependence

Dominance vs. Ctrl Dependence

- Block B is control dependent on Block A if and only if
 - The execution result of A determines if B will be executed

Dominance vs. Ctrl Dependence

- Block B is control dependent on Block A if and only if
 - The execution result of A determines if B will be executed
- Block B is control dependent on Block A if and only if
 - A has multiple successors
 - Not all successors can reach B

Dominance vs. Ctrl Dependence

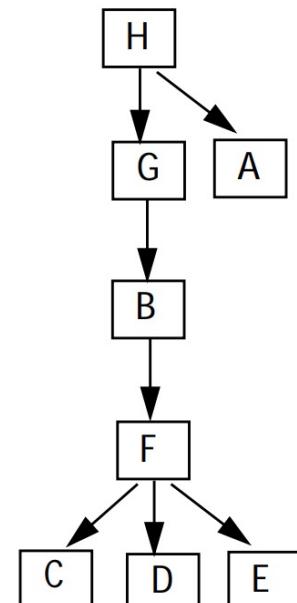
- Block B is control dependent on Block A if and only if
 - The execution result of A determines if B will be executed
- Block B is control dependent on Block A if and only if
 - A has multiple successors
 - Not all successors can reach B
- By definition, we always need a forward traversal from all successors of B to test if A ctrl-depends on B.

Dominance vs. Ctrl Dependence

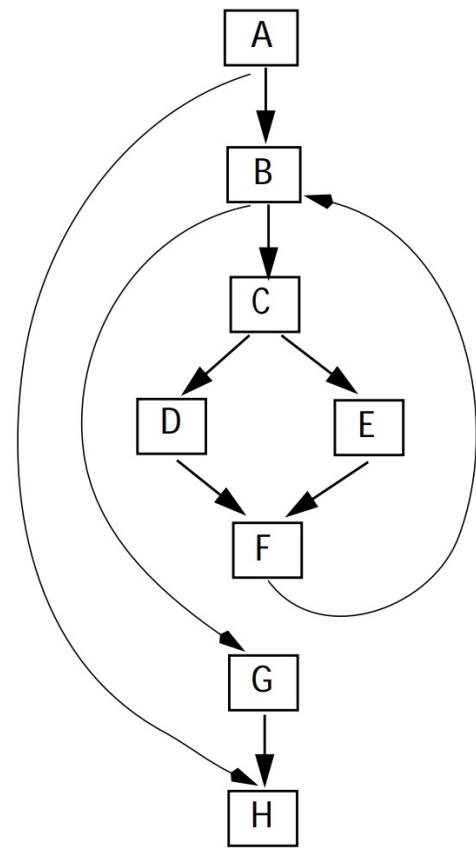
- Block B is control dependent on Block A if and only if
 - The execution result of A determines if B will be executed
- Block B is control dependent on Block A if and only if
 - A has multiple successors
 - Not all successors can reach B
- By definition, we always need a forward traversal from all successors of B to test if A ctrl-depends on B. **Too expensive!!**

Dominance vs. Ctrl Dependence

- Block B is control dependent on Block A if and only if
 - B post-dominates a successor of A
 - B does not post-dominates all successors of A



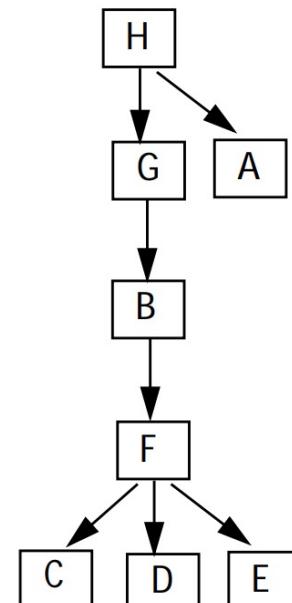
Postominator Tree



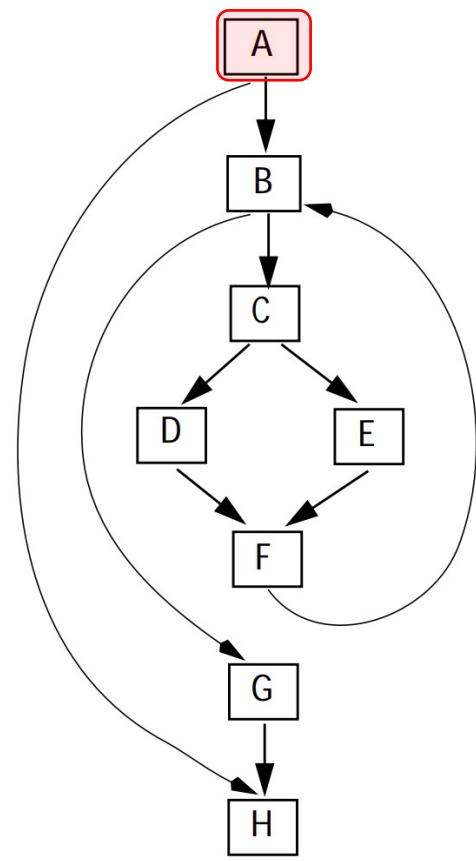
Control Flow Graph

Dominance vs. Ctrl Dependence

- Block B is control dependent on Block A if and only if
 - B post-dominates a successor of A
 - B does not post-dominates all successors of A



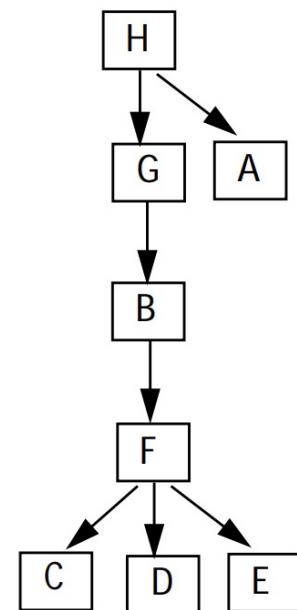
Postominator Tree



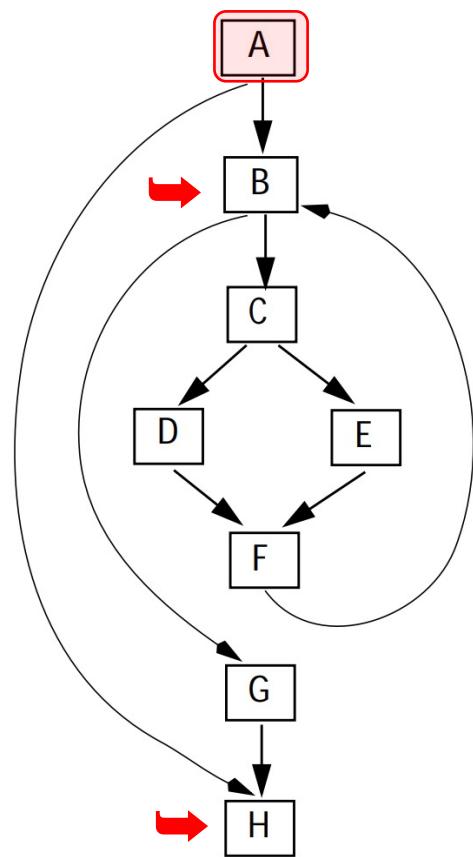
Control Flow Graph

Dominance vs. Ctrl Dependence

- Block B is control dependent on Block A if and only if
 - B post-dominates a successor of A
 - B does not post-dominates all successors of A



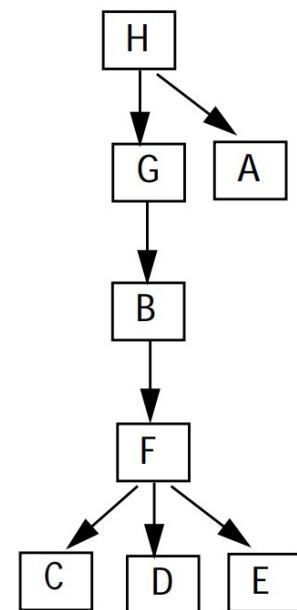
Postominator Tree



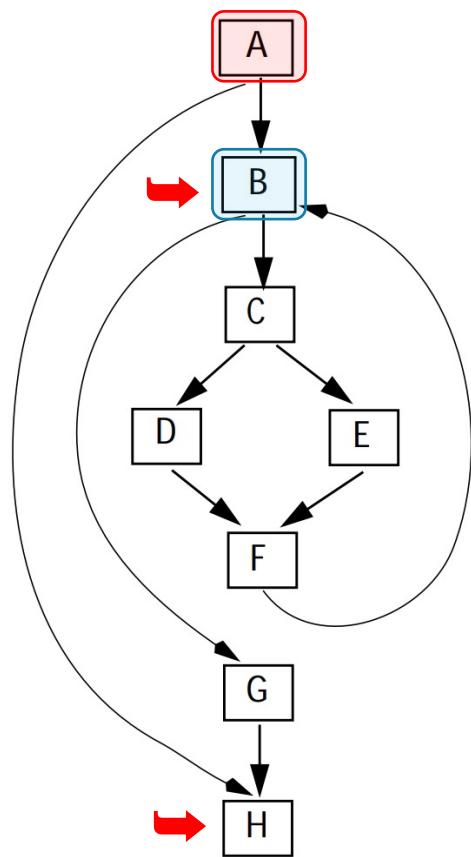
Control Flow Graph

Dominance vs. Ctrl Dependence

- Block B is control dependent on Block A if and only if
 - B post-dominates a successor of A
 - B does not post-dominates all successors of A



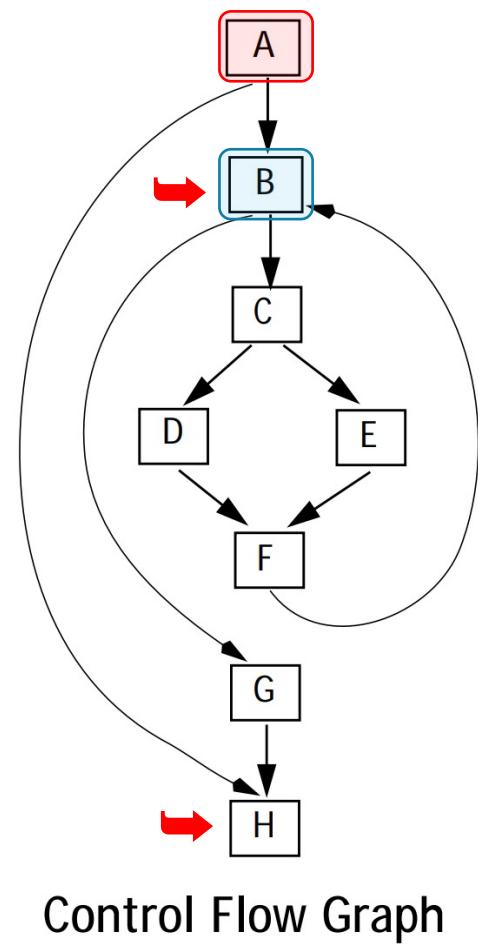
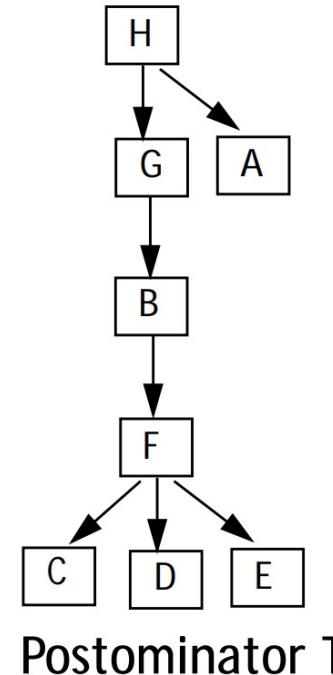
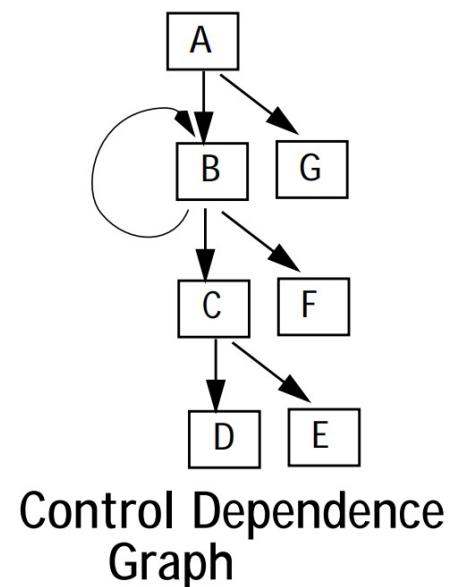
Postominator Tree



Control Flow Graph

Dominance vs. Ctrl Dependence

- Block B is control dependent on Block A if and only if
 - B post-dominates a successor of A
 - B does not post-dominates all successors of A



PART II-2: In-Block Optimization

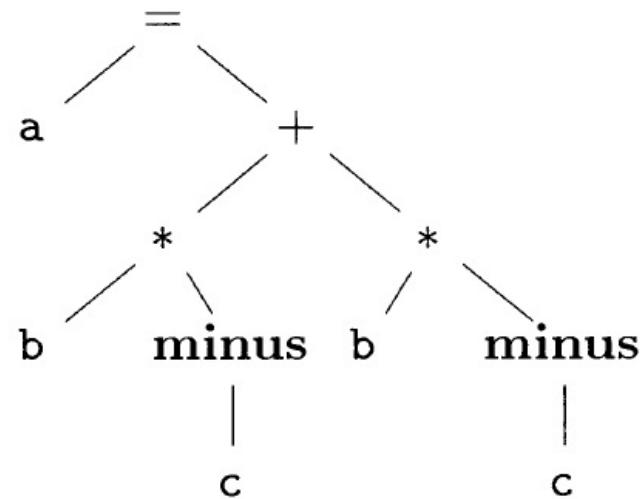
Recap: Syntax Tree and DAG

- What is the syntax tree and DAG of $a = b * (-c) + b * (-c)$

Try!

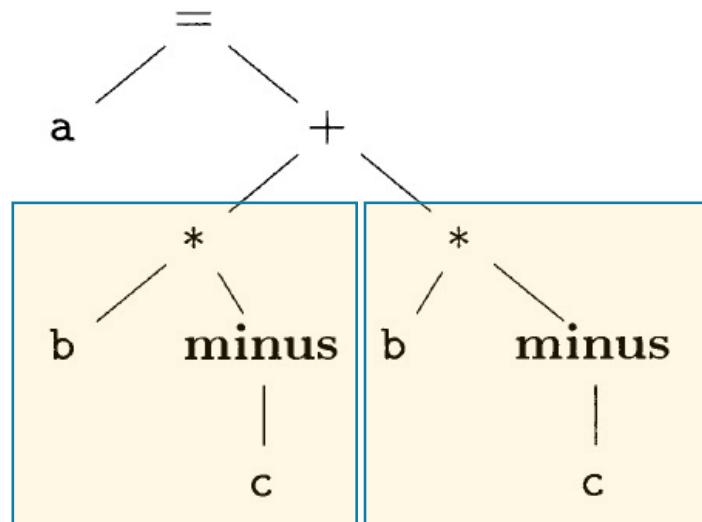
Recap: Syntax Tree and DAG

- What is the syntax tree and DAG of $a = b * (-c) + b * (-c)$



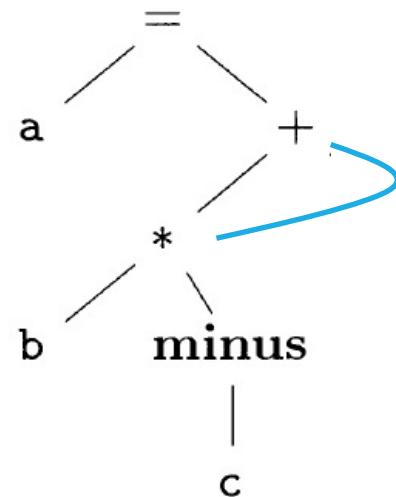
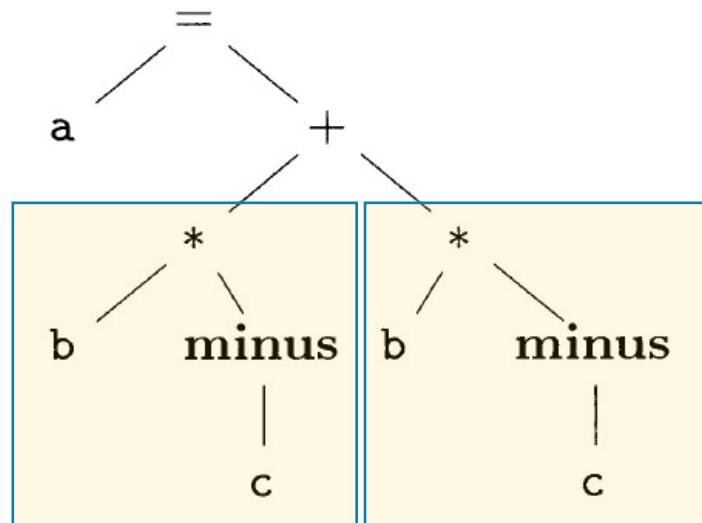
Recap: Syntax Tree and DAG

- What is the syntax tree and DAG of $a = b * (-c) + b * (-c)$



Recap: Syntax Tree and DAG

- What is the syntax tree and DAG of $a = b * (-c) + b * (-c)$

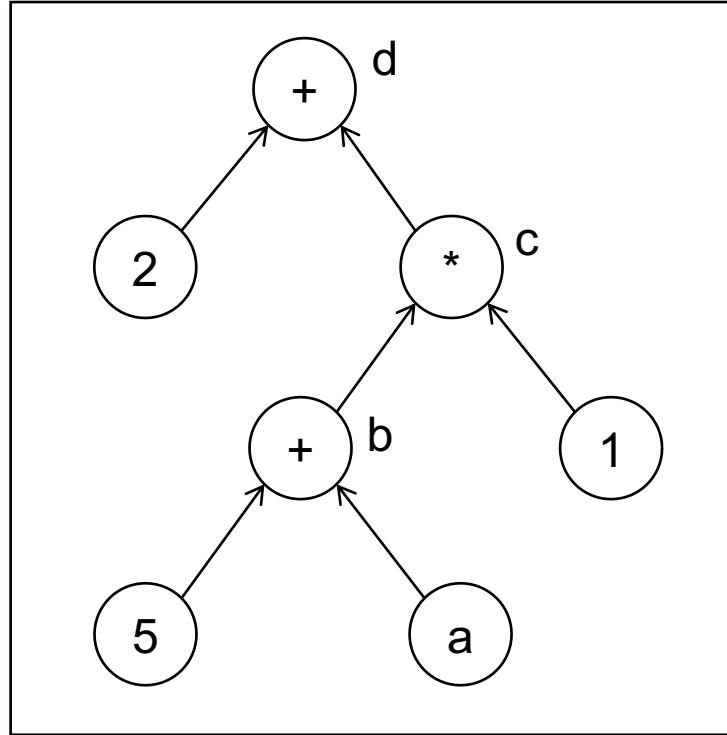


Tree of a Basic Block

```
b = a + 5
c = b * 1
d = 2 + c
```

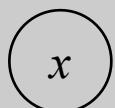
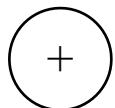
Tree of a Basic Block

```
b = a + 5  
c = b * 1  
d = 2 + c
```



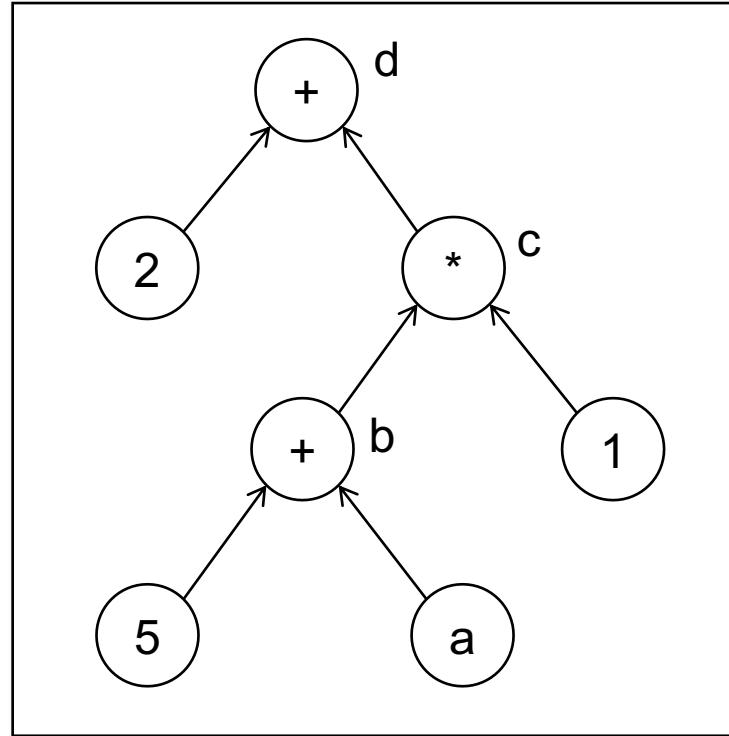
Tree-Based Instruction Selection

- Define a series of rules for selecting instructions
 - Pattern → Instructions

Patterns	Instructions
 a leaf node	LD R, x // R is a new register
 an operator node	ADD R _n , R _{o1} , R _{o2} ST x, R _n // R _n is a new register; // R _o are previous registers

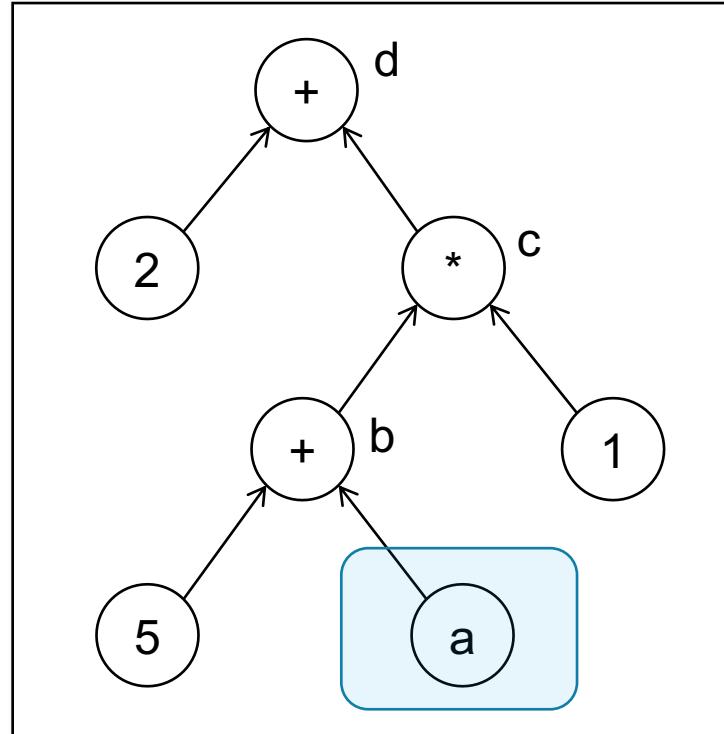
Tree-Based Instruction Selection

```
b = a + 5  
c = b * 1  
d = 2 + c
```



Tree-Based Instruction Selection

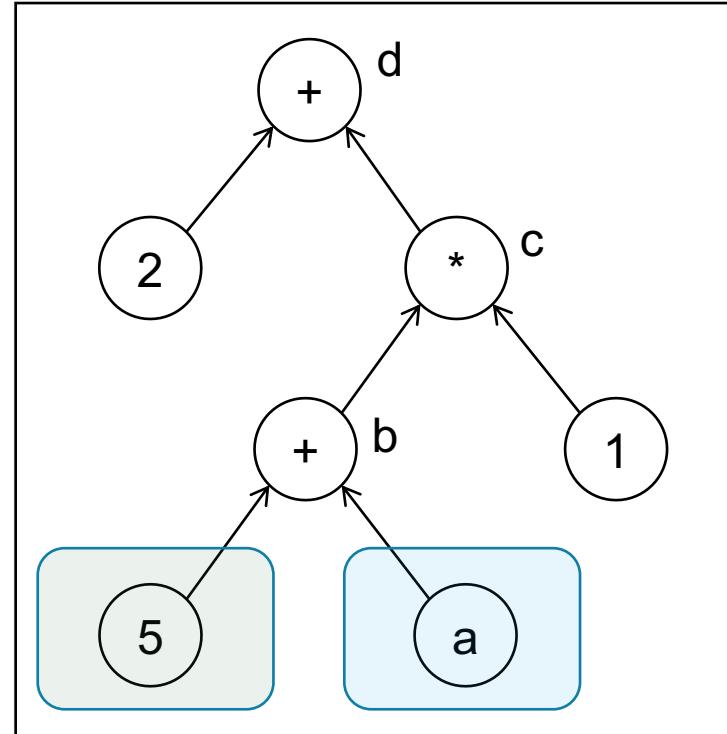
```
b = a + 5  
c = b * 1  
d = 2 + c
```



```
LD R0, a
```

Tree-Based Instruction Selection

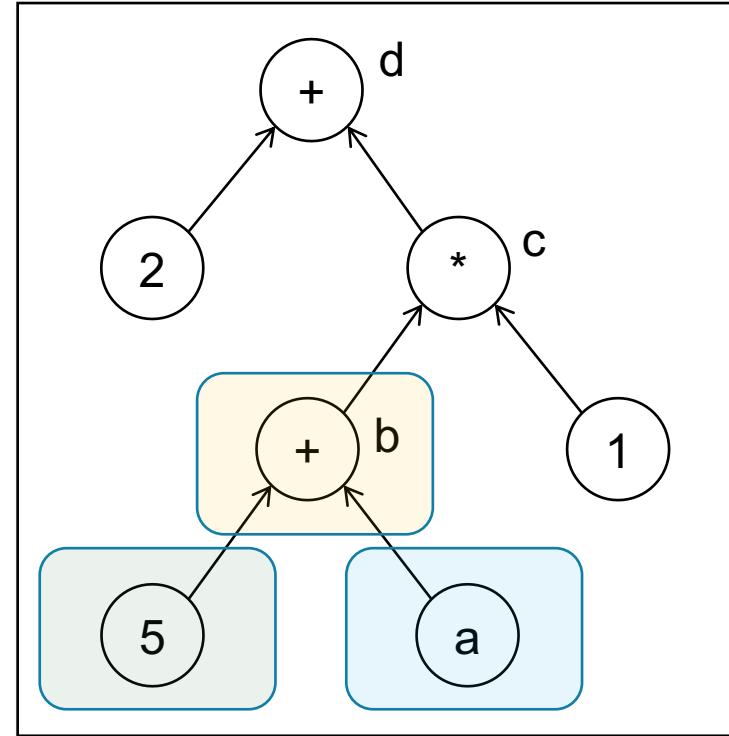
```
b = a + 5  
c = b * 1  
d = 2 + c
```



LD R0, a
LD R1, 5

Tree-Based Instruction Selection

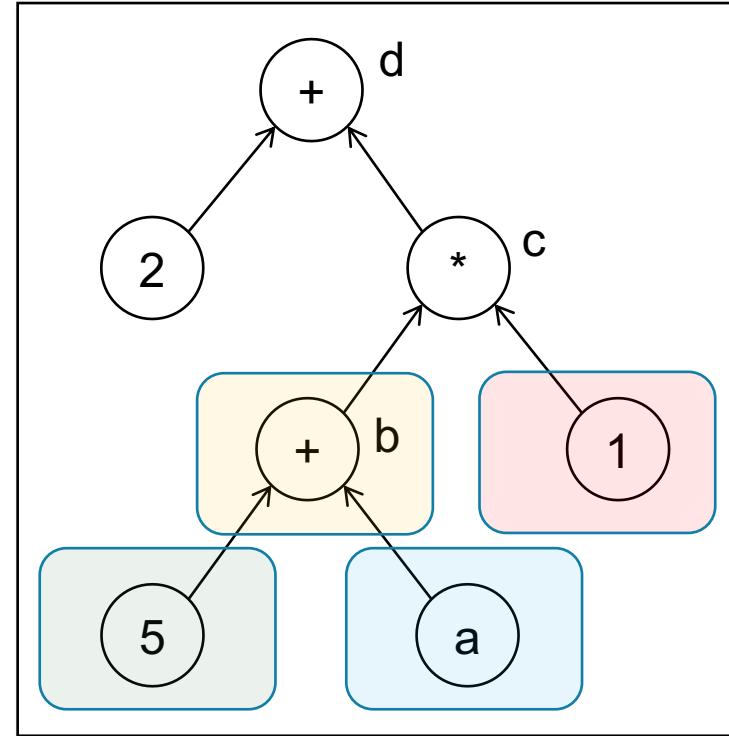
```
b = a + 5  
c = b * 1  
d = 2 + c
```



LD R0, a
LD R1, 5
ADD R2, R0, R1
ST b R2

Tree-Based Instruction Selection

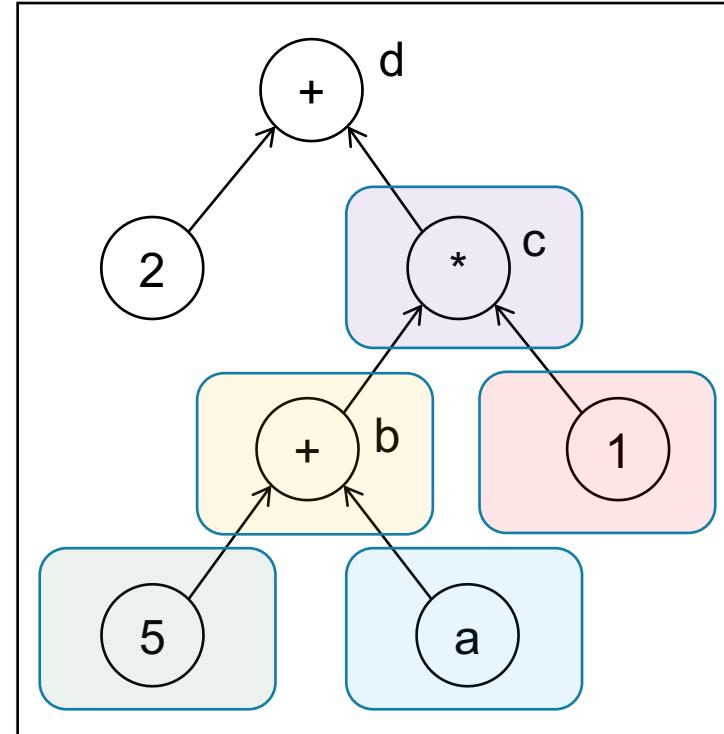
```
b = a + 5  
c = b * 1  
d = 2 + c
```



LD R0, a
LD R1, 5
ADD R2, R0, R1
ST b R2
LD R3, 1

Tree-Based Instruction Selection

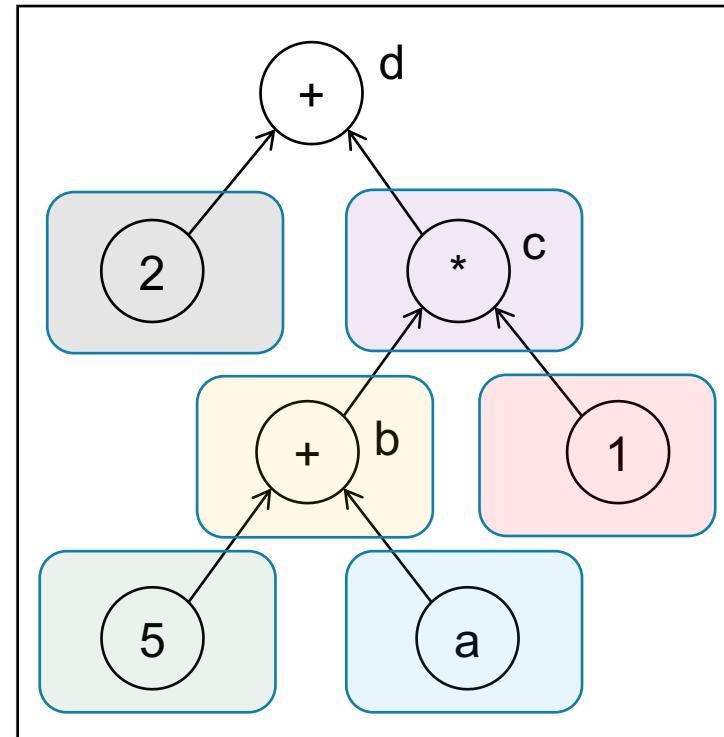
```
b = a + 5
c = b * 1
d = 2 + c
```



LD	R0,	a
LD	R1,	5
ADD	R2,	R0, R1
ST	b	R2
LD	R3,	1
MUL	R4,	R2, R3
ST	c	R4

Tree-Based Instruction Selection

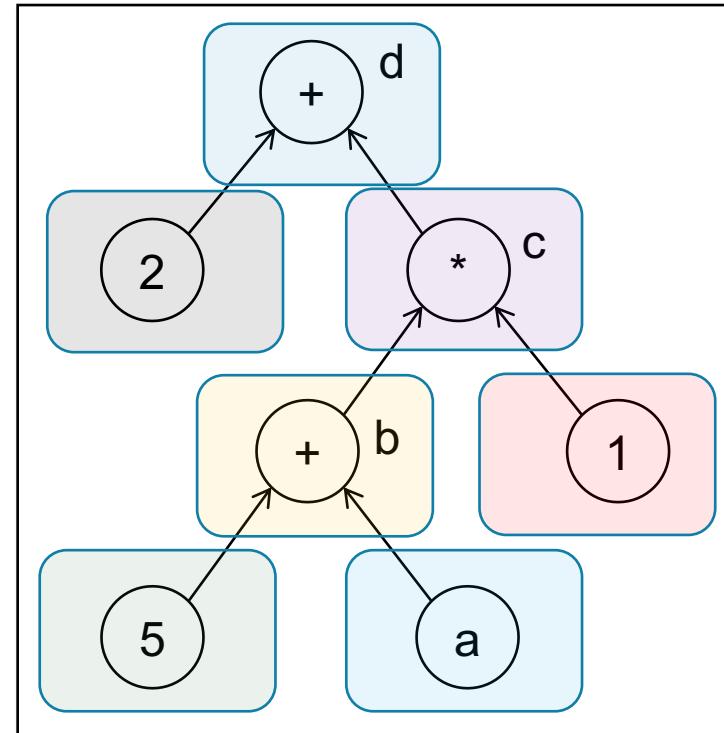
```
b = a + 5
c = b * 1
d = 2 + c
```



LD	R0,	a
LD	R1,	5
ADD	R2,	R0, R1
ST	b	R2
LD	R3,	1
MUL	R4,	R2, R3
ST	c	R4
LD	R5,	2

Tree-Based Instruction Selection

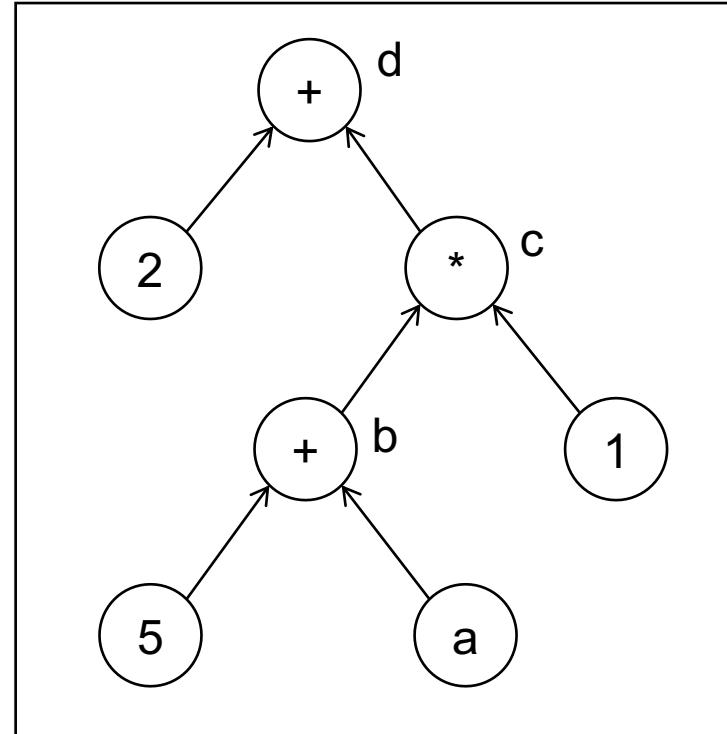
```
b = a + 5
c = b * 1
d = 2 + c
```



LD	R0,	a
LD	R1,	5
ADD	R2,	R0, R1
ST	b	R2
LD	R3,	1
MUL	R4,	R2, R3
ST	c	R4
LD	R5,	2
ADD	R6,	R4, R5
ST	d,	R6

Tree-Based Instruction Selection

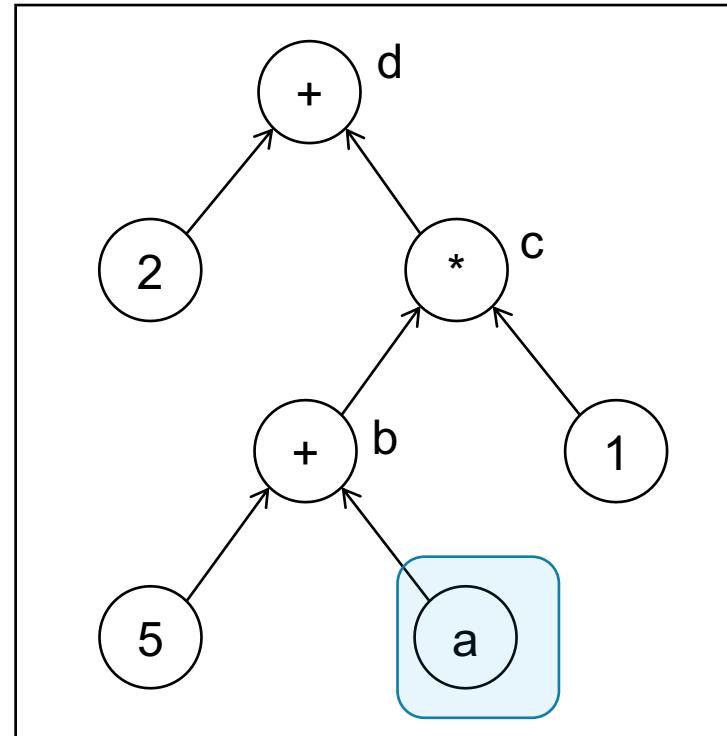
```
b = a + 5
c = b * 1
d = 2 + c
```



LD R0, a
LD R1, 5
ADD R2, R0, R1
ST b R2
LD R3, 1
MUL R4, R2, R3
ST c R4
LD R5, 2
ADD R6, R4, R5
ST d, R6

Tree-Based Instruction Selection

```
b = a + 5
c = b * 1
d = 2 + c
```

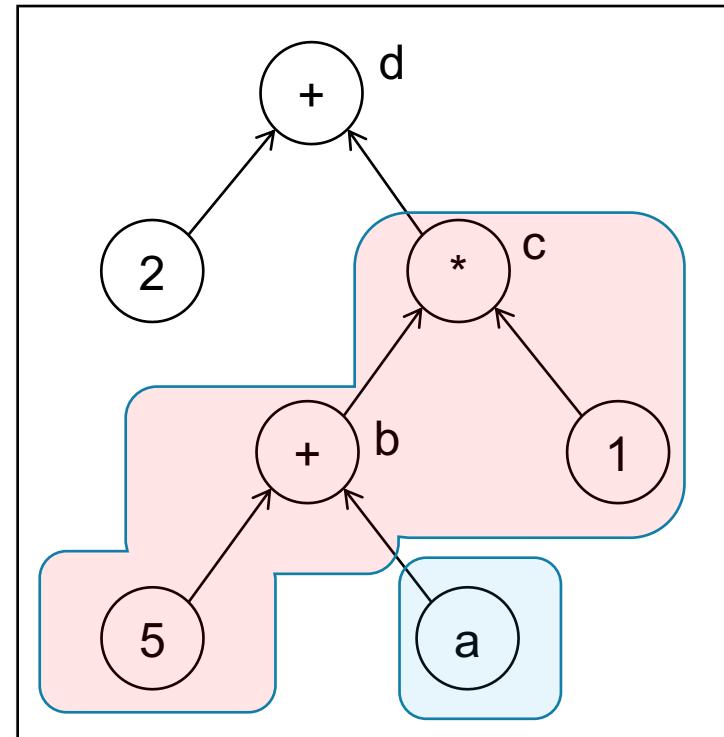


LD R0, a
LD R1, 5
ADD R2, R0, R1
ST b R2
LD R3, 1
MUL R4, R2, R3
ST c R4
LD R5, 2
ADD R6, R4, R5
ST d, R6

LD R0, a

Tree-Based Instruction Selection

```
b = a + 5
c = b * 1
d = 2 + c
```

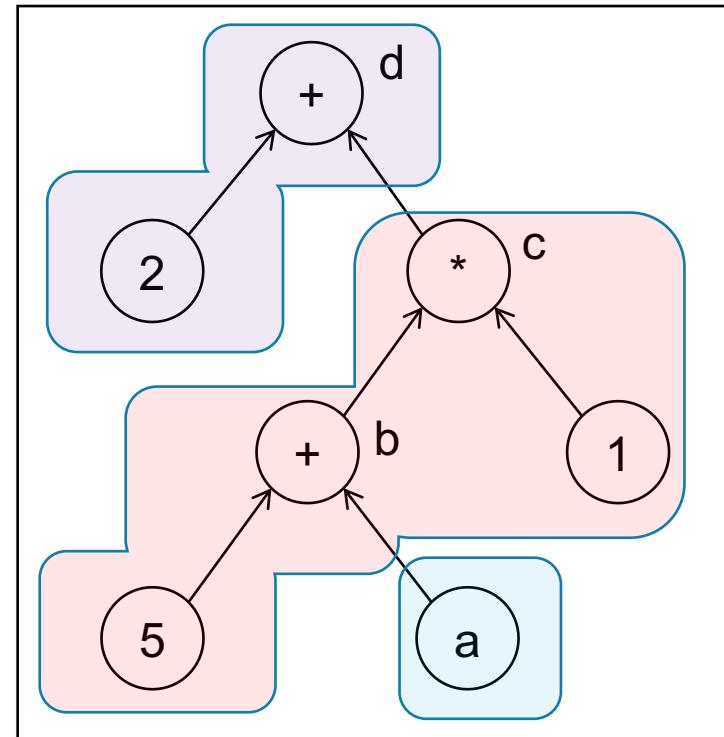


LD R0, a
LD R1, 5
ADD R2, R0, R1
ST b R2
LD R3, 1
MUL R4, R2, R3
ST c R4
LD R5, 2
ADD R6, R4, R5
ST d, R6

LD R0, a
ADD R1, R0, 5
ST b, R1
ST c, R1

Tree-Based Instruction Selection

```
b = a + 5
c = b * 1
d = 2 + c
```



LD R0, a
LD R1, 5
ADD R2, R0, R1
ST b R2
LD R3, 1
MUL R4, R2, R3
ST c R4
LD R5, 2
ADD R6, R4, R5
ST d, R6

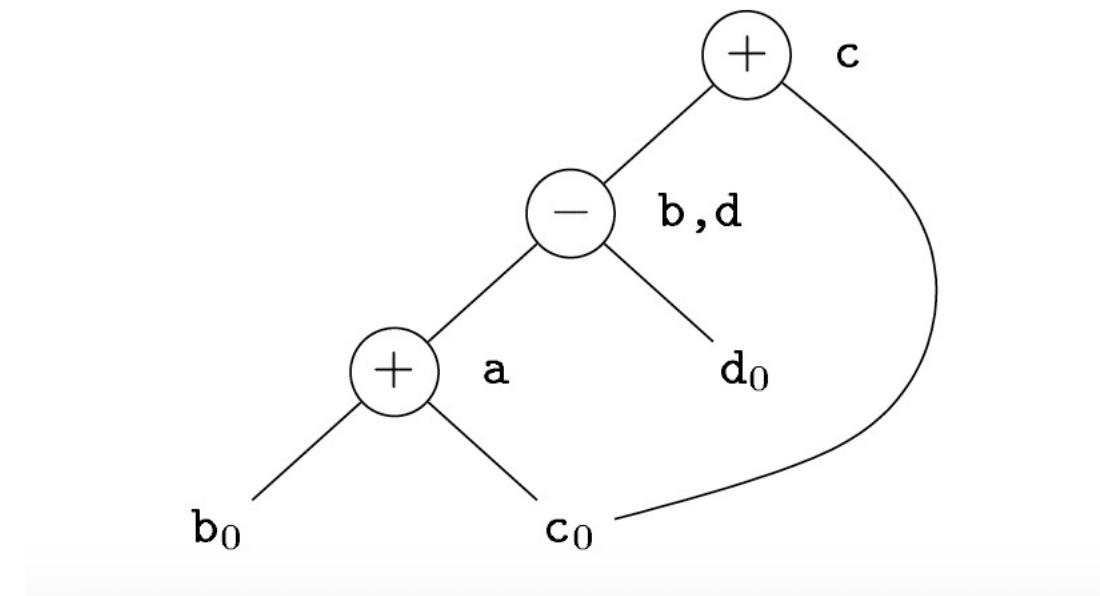
LD R0, a
ADD R1, R0, 5
ST b, R1
ST c, R1
ADD R2, R1, 2
ST d, R2

Tree-Based Instruction Selection

- Define a series of rules for selecting instructions
 - Pattern → Instructions
- **The effectiveness depends on the patterns predefined.**

From Tree to DAG

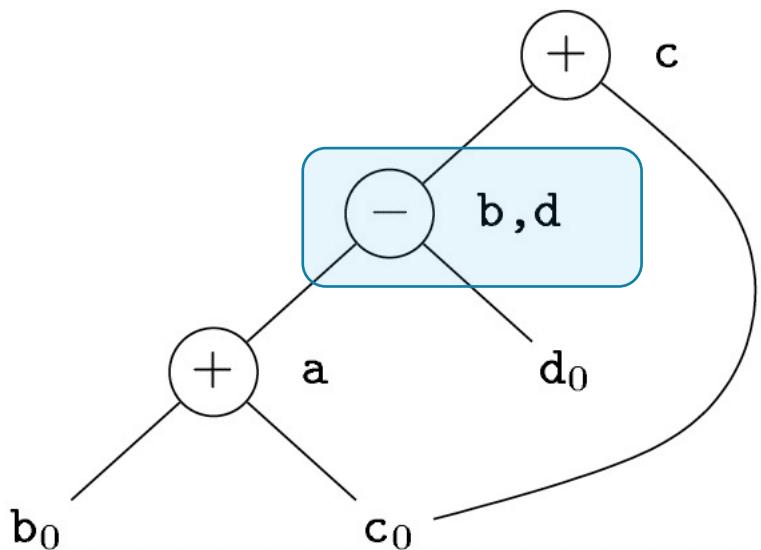
$$\begin{aligned}a &= b + c \\b &= a - d \\c &= b + c \\d &= a - d\end{aligned}$$



Local Common Sub-Expression

```
a = b + c  
b = a - d  
c = b + c  
d = a - d
```

It can be replaced by $d = b$.



Local Common Sub-Expression

- $a = x - y$
- if $x > y$ goto L

Local Common Sub-Expression

- $a = x - y$
- if $x > y$ goto L
- =>
- $a = x - y$
- if $x - y > 0$ goto L

Local Common Sub-Expression

- $a = x - y$
- if $x > y$ goto L
- =>
- $a = x - y$
- if $x - y > 0$ goto L
- =>
- $a = x - y$
- if $a > 0$ goto L

Local Common Sub-Expression

- $a = x - y$
- if $x > y$ goto L
- =>
- $a = x - y$
- if $x - y > 0$ goto L
- =>
- $a = x - y$
- if $a > 0$ goto L



Is it always correct?

Algebraic Identities

Algebraic Identities

$$x + 0 = 0 + x = x$$

$$x \times 1 = 1 \times x = x$$

$$x - 0 = x$$

$$x/1 = x$$

Algebraic Identities

$$x + 0 = 0 + x = x$$

$$x \times 1 = 1 \times x = x$$

$$x - 0 = x$$

$$x/1 = x$$

EXPENSIVE

$$x^2 = x \times x$$

$$2 \times x = x + x$$

$$x/2 = x \times 0.5$$

CHEAPER

Algebraic Identities

$$x + 0 = 0 + x = x$$

$$x \times 1 = 1 \times x = x$$

$$x - 0 = x$$

$$x/1 = x$$

EXPENSIVE

$$x^2$$

$$2 \times x$$

$$x/2$$

CHEAPER

$$= x \times x$$

$$= x + x$$

$$= x \times 0.5$$

$$2 * 3.14 = 6.28$$

Machine Idioms

```
LD  R0, a      // R0 = a
ADD R0, R0, #1 // R0 = R0 + 1
ST  a, R0      // a = R0
```

Possible target code for $a = a + 1$

How about "INC a" ??

Machine Idioms

```
LD  R0, a      // R0 = a
ADD R0, R0, #1 // R0 = R0 + 1
ST  a, R0      // a = R0
```

How about “INC a” ??

Possible target code for $a = a + 1$

Redundant Loads/Stores

```
LD  R0, a
ST  a, R0
```

PART II-3: Peephole Optimization

Peephole Optimization

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Peephole Optimization

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Peephole Optimization

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Peephole Optimization

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Peephole Optimization

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Peephole Optimization

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

- Optimize the code in a small window
- Work on IR or the target code
- **Cross-block optimization**

Peephole Optimization

```
if 0 != 1 goto L2
print debugging information
L2:
```

Peephole Optimization

```
if 0 != 1 goto L2
print debugging information
```

L2:

Peephole Optimization

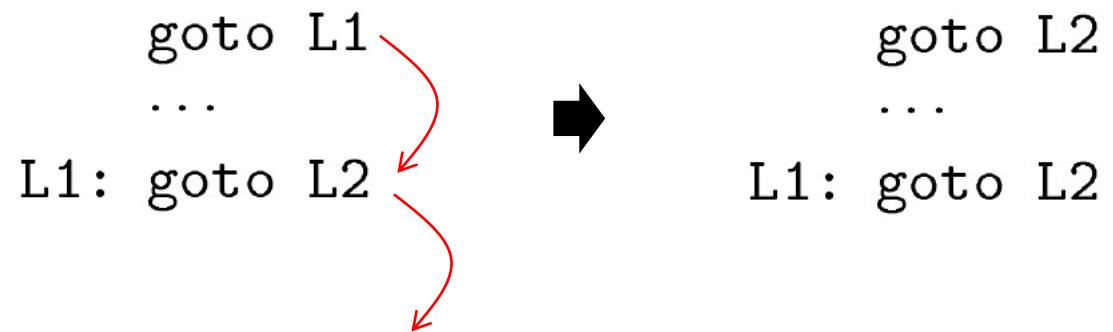
```
if 0 != 1 goto L2
print debugging information
```

L2:

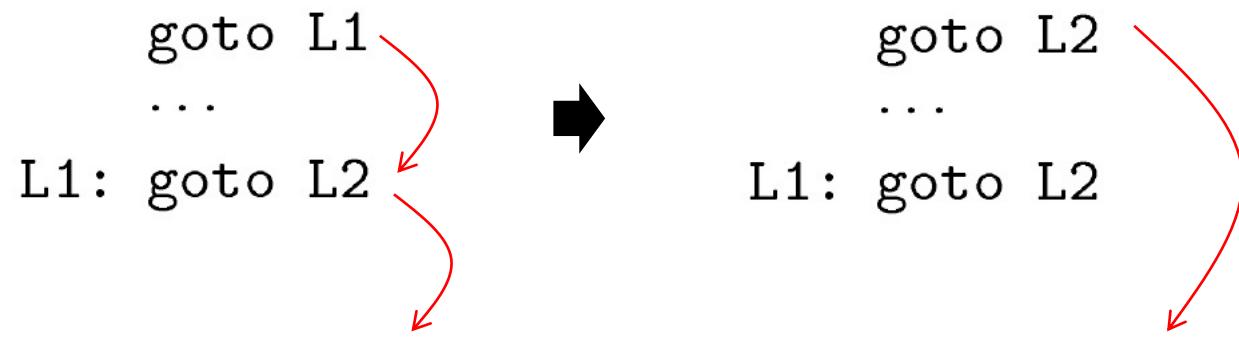
Peephole Optimization

```
        goto L1          goto L2
        ...
        L1: goto L2      ...  
    ➔           L1: goto L2
```

Peephole Optimization



Peephole Optimization



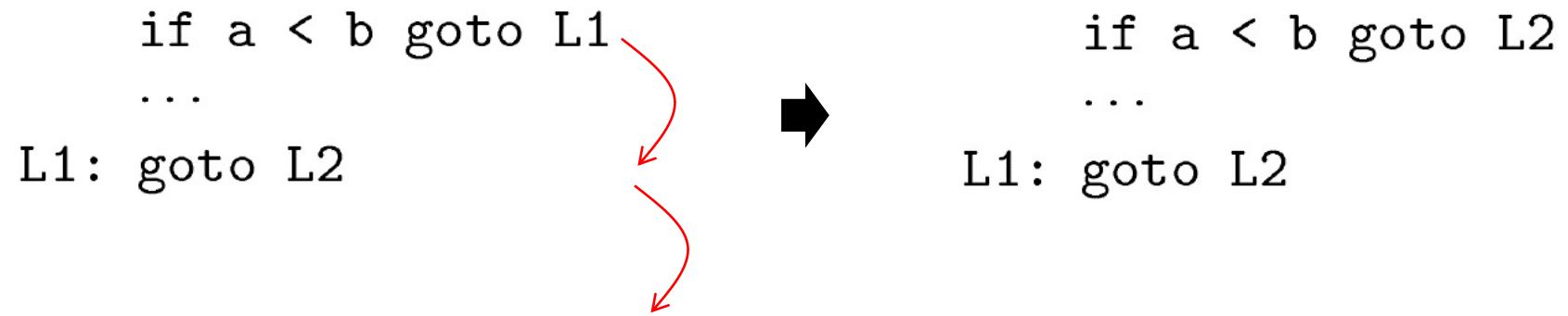
Peephole Optimization

```
if a < b goto L1
...
L1: goto L2
```

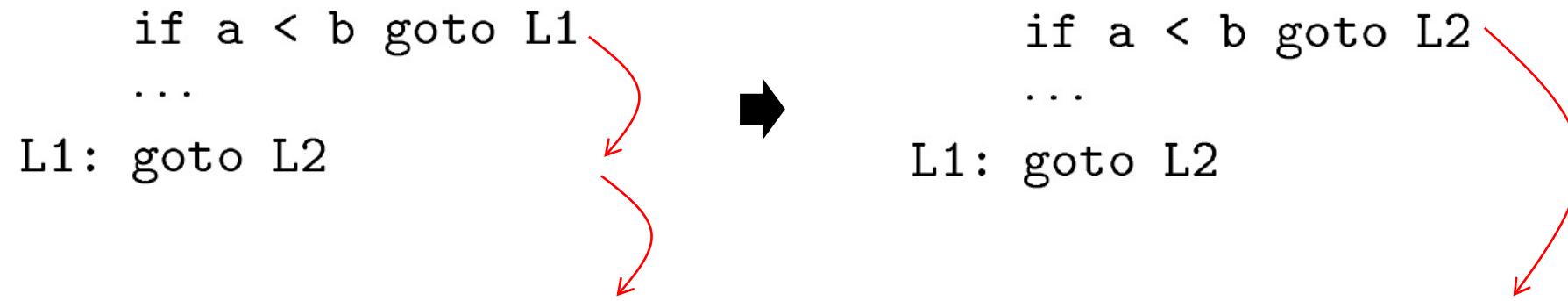


```
if a < b goto L2
...
L1: goto L2
```

Peephole Optimization



Peephole Optimization



Peephole Optimization

```
if debug == 1 goto L1
goto L2
L1: print debugging information
L2:
```



```
if debug != 1 goto L2
print debugging information
L2:
```

Peephole Optimization

We can define many many such rules for peephole optimization!

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R2,	#12
ADD	R3,	R1, R2
LD	R4,	*R3
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R2,	#12
ADD	R3,	R1,
LD	R4,	*R3
MUL	R5,	R0,
LD	R6,	#16
ADD	R7,	R1,
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9,
ST	y,	R10

- First window
- No optimization available
- Advance the window

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R2,	#12
ADD	R3,	R1, R2
LD	R4,	*R3
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R2,	#12
ADD	R3,	R1, #12R2
LD	R4,	*R3
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R2,	#12
ADD	R3,	R1, #12R2
LD	R4,	*R3
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
ADD	R3,	R1,
LD	R4,	*R3
MUL	R5,	R0,
LD	R6,	#16
ADD	R7,	R1,
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9,
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
ADD	R3,	R1,
LD	R4,	*R3
MUL	R5,	R0,
MUL	R5,	R4
LD	R6,	#16
ADD	R7,	R1,
ADD	R7,	R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9,
SUB	R10,	R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
ADD	R3,	R1,
LD	R4,	12(R1)*R3
MUL	R5,	R0,
LD	R6,	#16
ADD	R7,	R1,
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9,
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
ADD	R3,	R1, #12
LD	R4,	12(R1)*R3
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

- No more optimization available
- Advance the window

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

- No more optimization available
- Advance the window

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, #16R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10



#16R6

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R6,	#16
ADD	R7,	R1, #16R6
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
ADD	R7,	R1, #16
LD	R8,	*R7
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
ADD	R7,	R1, #16
LD	R8,	*R716(R1)
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R8,	16(R1)
LD	R9,	*R8
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R8,	16(R1)
LD	R9,	* 16(R1) ^{R8}
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R9,	*16(R1)
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R9,	*16(R1)
SUB	R10,	R9, R5
ST	y,	R10

- No more optimization available
- Advance the window

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R9,	*16(R1)
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R9,	*16(R1)
SUB	R10,	R9, R5
ST	y,	R10

- No more optimization available
- Reach the end; Terminate

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R9,	*16(R1)
SUB	R10,	R9, R5
ST	y,	R10

Peephole Optimization

LD	R0,	#2
LD	R1,	x
LD	R4,	12(R1)
MUL	R5,	R0, R4
LD	R9,	*16(R1)
SUB	R10,	R9, R5
ST	y,	R10

- No more optimization available
- Reach the end; Terminate
- **In total:**
- Four unnecessary instructions are deleted via the peephole optimization

PART II-4: Local Register Allocation

Register Allocation

- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation:** ∞ virtual registers \rightarrow k physical registers

Register Allocation

- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation:** ∞ virtual registers \rightarrow k physical registers
- **Requirement:**
 - Produce correct code using k or fewer registers

Register Allocation

- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation:** ∞ virtual registers \rightarrow k physical registers
- **Requirement:**
 - Produce correct code using k or fewer registers
 - Minimize loads, stores, and space to hold spilled values

Register Allocation

- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation:** ∞ virtual registers \rightarrow k physical registers
- **Requirement:**
 - Produce correct code using k or fewer registers
 - Minimize loads, stores, and space to hold spilled values
 - Efficient register allocation ($O(n)$ or $O(n \log n)$)

Register Allocation

- Two values CANNOT be mapped to the same register wherever they are both live (before their last use)
- **Spilling:**
 - saves a value from a register to memory

Register Allocation

- Two values CANNOT be mapped to the same register wherever they are both live (before their last use)
- **Spilling:**
 - saves a value from a register to memory
 - the register is then free for other values

Register Allocation

- Two values CANNOT be mapped to the same register wherever they are both live (before their last use)
- **Spilling:**
 - saves a value from a register to memory
 - the register is then free for other values
 - we can load the spilled value from memory to register when necessary

Local Register Allocation

- Register allocation in a block

Local Register Allocation

- Register allocation in a block
- **MAXLIVE**: the maximum of values live at each instruction
 - **MAXLIVE $\leq k$** --- Allocation is trivial
 - **MAXLIVE $> k$** --- Values must be spilled to the memory

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028	
2.	LD	R2	*R1	// R2 <- contents(R1), assume y	
3.	MUL	R3	R1	R2	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x	
5.	SUB	R5	R4	R2	// R5 <- x-y
6.	LD	R6	z	// R6 <- z	
7.	MUL	R7	R5	R6	// R7 <- z * (x-y)
8.	SUB	R8	R7	R3	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y	

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028	// R1 <- 1028
2.	LD	R2	*R1	// R2 <- contents(R1), assume y
3.	MUL	R3	R1	// R3 <- 1028 * y
4.	LD	R4	x	// R4 <- x
5.	SUB	R5	R4	// R5 <- x-y
6.	LD	R6	z	// R6 <- z
7.	MUL	R7	R5	// R7 <- z * (x-y)
8.	SUB	R8	R7	// R8 <- z * (x-y) - 1028 * y
9.	ST	*R1	R8	// contents(R1) <- z * (x-y) - 1028 * y

MAXLIVE

1.	LD	R1	#1028		//	R1
2.	LD	R2	*R1		//	R1 R2
3.	MUL	R3	R1	R2	//	R1 R2 R3
4.	LD	R4	x		//	R1 R2 R3 R4
5.	SUB	R5	R4	R2	//	R1 R3 R5
6.	LD	R6	z		//	R1 R3 R5 R6
7.	MUL	R7	R5	R6	//	R1 R3 R7
8.	SUB	R8	R7	R3	//	R1 R8
9.	ST	*R1	R8		//	

MAXLIVE

1.	LD	R1	#1028		// R1
2.	LD	R2	*R1		// R1 R2
3.	MUL	R3	R1	R2	// R1 R2 R3
4.	LD	R4	x		// R1 R2 R3 R4
5.	SUB	R5	R4	R2	// R1 R3 R5
6.	LD	R6	z		// R1 R3 R5 R6
7.	MUL	R7	R5	R6	// R1 R3 R7
8.	SUB	R8	R7	R3	// R1 R8
9.	ST	*R1	R8		//

MAXLIVE

1.	LD	R1	#1028	// R1	
2.	LD	R2	*R1	// R1 R2	
3.	MUL	R3	R1	// R1 R2 R3	
4.	LD	R4	x	// R1 R2 R3 R4	
5.	SUB	R5	R4	R2	// R1 R3 R5
6.	LD	R6	z	// R1 R3 R5 R6	
7.	MUL	R7	R5	R6	// R1 R3 R7
8.	SUB	R8	R7	R3	// R1 R8
9.	ST	*R1	R8		//

MAXLIVE = 4

Local Register Allocation

1.	LD	R1	#1028	// R1	
2.	LD	R2	*R1	// R1 R2	
3.	MUL	R3	R1 R2	// R1 R2 R3	
4.	LD	R4	X	// R1 R2 R3 R4	MAXLIVE = 4
5.	SUB	R5	R4 R2	// R1 R3 R5	
6.	LD	R6	Z	// R1 R3 R5 R6	if k ≥ 4
7.	MUL	R7	R5 R6	// R1 R3 R7	
8.	SUB	R8	R7 R3	// R1 R8	
9.	ST	*R1	R8	//	

Local Register Allocation

1.	LD	R1	#1028	// R1	
2.	LD	R2	*R1	// R1 R2	
3.	MUL	R3	R1 R2	// R1 R2 R3	
4.	LD	R4	X	// R1 R2 R3 R4	
5.	SUB	R2	R4 R2	// R1 R3 R2	MAXLIVE = 4
6.	LD	R6	z	// R1 R3 R2 R6	if k ≥ 4
7.	MUL	R7	R2 R6	// R1 R3 R7	
8.	SUB	R8	R7 R3	// R1 R8	
9.	ST	*R1	R8	//	

Local Register Allocation

1.	LD	R1	#1028	// R1	
2.	LD	R2	*R1	// R1 R2	
3.	MUL	R3	R1 R2	// R1 R2 R3	
4.	LD	R4	x	// R1 R2 R3 R4	MAXLIVE = 4
5.	SUB	R2	R4 R2	// R1 R3 R2	
6.	LD	R4	z	// R1 R3 R2 R4	if k ≥ 4
7.	MUL	R7	R2 R4	// R1 R3 R7	
8.	SUB	R8	R7 R3	// R1 R8	
9.	ST	*R1	R8	//	

Local Register Allocation

1.	LD	R1	#1028	// R1	
2.	LD	R2	*R1	// R1 R2	
3.	MUL	R3	R1 R2	// R1 R2 R3	
4.	LD	R4	x	// R1 R2 R3 R4	MAXLIVE = 4
5.	SUB	R2	R4 R2	// R1 R3 R2	
6.	LD	R4	z	// R1 R3 R2 R4	if k ≥ 4
7.	MUL	R2	R2 R4	// R1 R3	R2
8.	SUB	R8	R2 R3	// R1	R8
9.	ST	*R1	R8	//	

Local Register Allocation

1.	LD	R1	#1028	// R1	
2.	LD	R2	*R1	// R1 R2	
3.	MUL	R3	R1 R2	// R1 R2 R3	
4.	LD	R4	x	// R1 R2 R3 R4	MAXLIVE = 4
5.	SUB	R2	R4 R2	// R1 R3 R2	
6.	LD	R4	z	// R1 R3 R2 R4	if k ≥ 4
7.	MUL	R2	R2 R4	// R1 R3 R2	
8.	SUB	R2	R2 R3	// R1 R2	
9.	ST	*R1	R2	//	

Local Register Allocation

MAXLIVE = 4

if $k < 4$, e.g., 3

Local Register Allocation

1.	LD	R1	#1028	// R1	
2.	LD	R2	*R1	// R1 R2	
3.	MUL	R3	R1 R2	// R1 R2 R3	
4.	LD	R4	x	// R1 R2 R3 R4	
5.	SUB	R5	R4 R2	// R1 R3 R5	MAXLIVE = 4
6.	LD	R6	z	// R1 R3 R5 R6	<i>if k < 4, e.g., 3</i>
7.	MUL	R7	R5 R6	// R1 R3 R7	
8.	SUB	R8	R7 R3	// R1 R8	
9.	ST	*R1	R8	//	

Local Register Allocation

1.	LD	R1	#1028	// R1		
2.	LD	R2	*R1	// R1 R2		
3.	MUL	R3	R1 R2	// R1 R2 R3	Spill R3 by ST v R3	
4.	LD	R4	x	// R1 R2 R3 R4		MAXLIVE = 4
5.	SUB	R5	R4 R2	// R1 R3 R5		
6.	LD	R6	z	// R1 R3 R5 R6		if k < 4, e.g., 3
7.	MUL	R7	R5 R6	// R1 R3	R7	
8.	SUB	R8	R7 R3	// R1	R8	
9.	ST	*R1	R8	//		

Local Register Allocation

1.	LD	R1	#1028	// R1		
2.	LD	R2	*R1	// R1 R2		
3.	MUL	R3	R1	R2	// R1 R2 R3	Spill R3 by ST v R3
4.	LD	R4	x		// R1 R2 R4	
5.	SUB	R5	R4	R2	// R1 R5	MAXLIVE = 4
6.	LD	R6	z		// R1 R5 R6	if k < 4, e.g., 3
7.	MUL	R7	R5	R6	// R1 R7	Reload R3 by LD R3 v
8.	SUB	R8	R7	R3	// R1 R8	
9.	ST	*R1	R8		//	

Local Register Allocation

1.	LD	R1	#1028	// R1		
2.	LD	R2	*R1	// R1 R2		
3.	MUL	R3	R1 R2	// R1 R2 R3	Spill R3 by ST v R3	
4.	LD	R3	x	// R1 R2 R3		MAXLIVE = 4
5.	SUB	R5	R3 R2	// R1 R5		
6.	LD	R6	z	// R1 R5 R6		if k < 4, e.g., 3
7.	MUL	R7	R5 R6	// R1 R7	Reload R3 by LD R3 v	
8.	SUB	R8	R7 R3	// R1 R8		
9.	ST	*R1	R8	//		

Why R3? Usage Counts, etc.

1.	LD	R1	#1028	// R1	
2.	LD	R2	*R1	// R1 R2	
3.	MUL	R3	R1	R2 // R1 R2 R3	Spill R3 by <u>ST v R3</u>
4.	LD	R4	x	// R1 R2 R3 R4	
5.	SUB	R5	R4	R2 // R1 R3 R5	MAXLIVE = 4
6.	LD	R6	z	// R1 R3 R5 R6	if k < 4, e.g., 3
7.	MUL	R7	R5	R6 // R1 R3 R7	
8.	SUB	R8	R7	R3 // R1 R8	
9.	ST	*R1	R8	//	

Spill Value, Next Use, Farthest

1.	LD	R1	#1028		// R1	
2.	LD	R2	*R1		// R1 R2	
3.	MUL	R3	R1	R2	// R1 R2 R3	Spill R1 by ST v R1
4.	LD	R4	x		// R1 R2 R3 R4	
5.	SUB	R5	R4	R2	// R1 R3 R5	
6.	LD	R6	z		// R1 R3 R5 R6	if k < 4, e.g., 3
7.	MUL	R7	R5	R6	// R1 R3	R7
8.	SUB	R8	R7	R3	// R1	R8
9.	ST	*R1	R8		//	

MAXLIVE = 4

←

Spill R1 by ST v R1

if k < 4, e.g., 3

↓

PART II-5: Global Register Allocation

Global Register Allocation

- Local register allocation does not capture reuse of values across multiple basic blocks
- Global allocation often uses the **graph-coloring** paradigm

Global Register Allocation

- Local register allocation does not capture reuse of values across multiple basic blocks
- Global allocation often uses the **graph-coloring** paradigm
 - Build a **conflict/interference graph**

Global Register Allocation

- Local register allocation does not capture reuse of values across multiple basic blocks
- Global allocation often uses the **graph-coloring** paradigm
 - Build a **conflict/interference graph**
 - Find a **k-coloring** for the graph, or change the code to a nearby problem that it can color

Global Register Allocation

- Local register allocation does not capture reuse of values across multiple basic blocks
- Global allocation often uses the **graph-coloring** paradigm
 - Build a **conflict/interference graph**
 - Find a **k-coloring** for the graph, or change the code to a nearby problem that it can color
 - **NP-complete** under nearly all assumptions, so heuristics are needed

K-Coloring

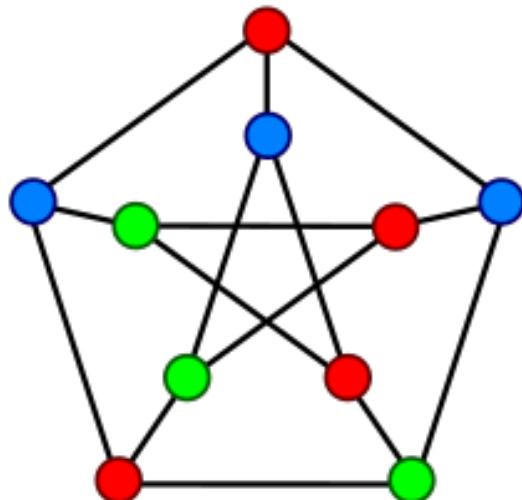
- **Vertex Coloring:** assign a color to each vertex such that no edge connects vertices with the same color.

K-Coloring

- **Vertex Coloring:** assign a color to each vertex such that no edge connects vertices with the same color.
- **K-Coloring:** a coloring using at most k colors

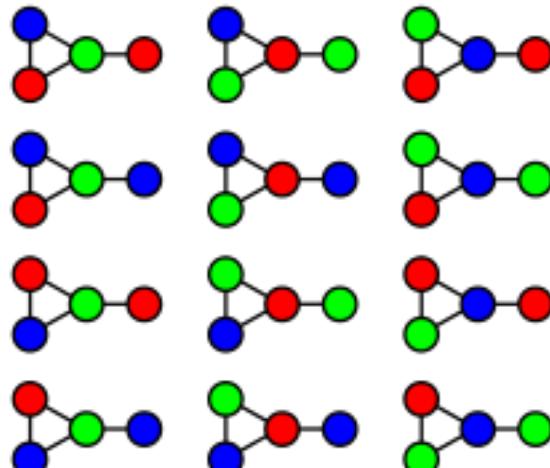
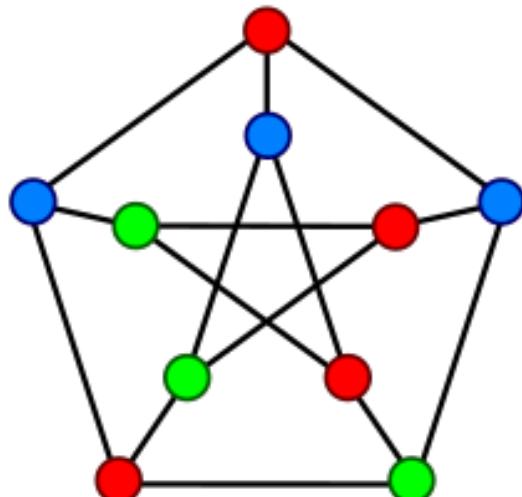
K-Coloring

- **Vertex Coloring:** assign a color to each vertex such that no edge connects vertices with the same color.
- **K-Coloring:** a coloring using at most k colors

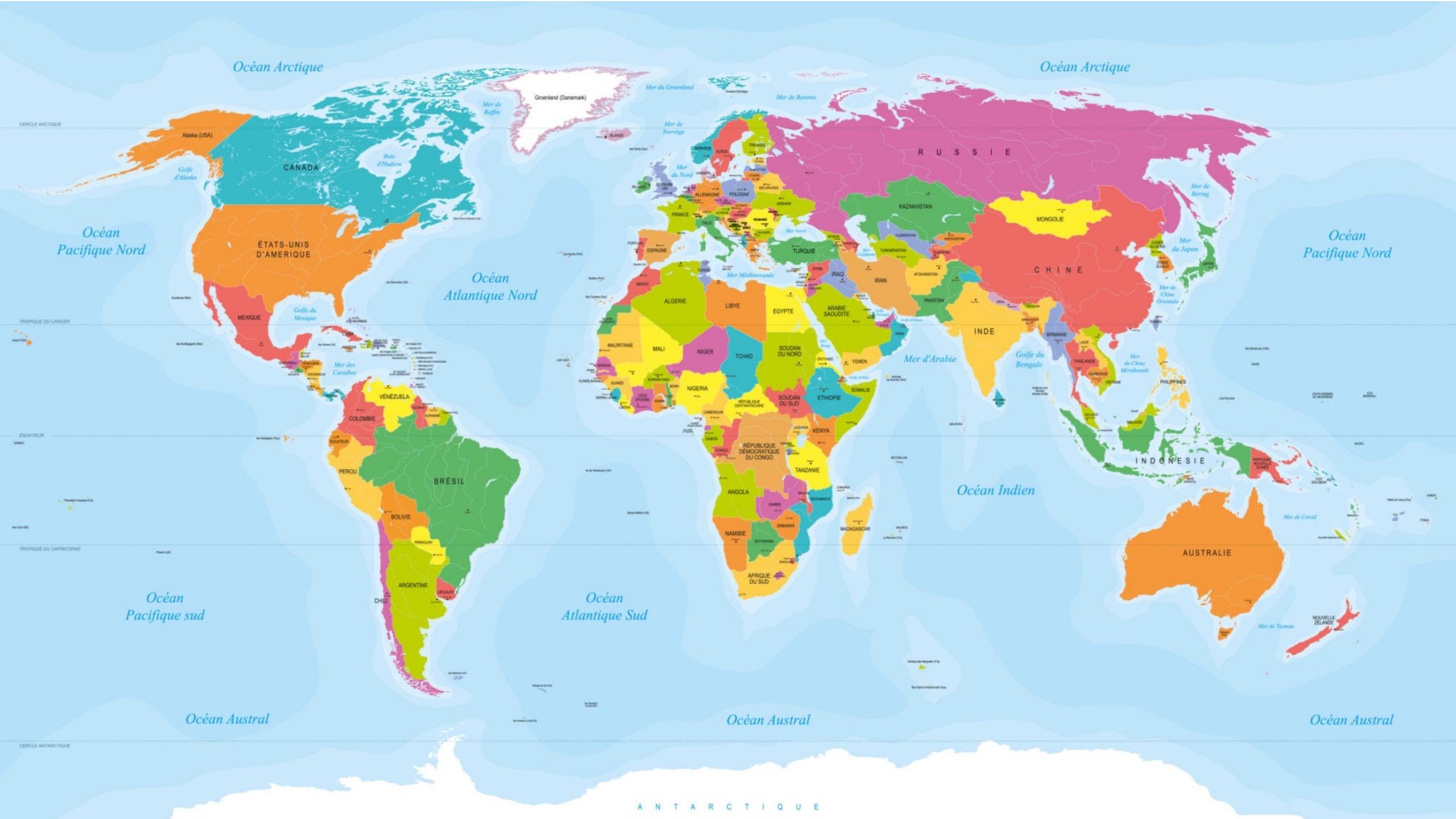


K-Coloring

- **Vertex Coloring:** assign a color to each vertex such that no edge connects vertices with the same color.
- **K-Coloring:** a coloring using at most k colors



3-coloring in 12 ways



K-Coloring for Register Allocation

- Map graph vertices onto virtual registers
- Map colors onto physical registers

K-Coloring for Register Allocation

- Map graph vertices onto virtual registers
- Map colors onto physical registers
- From live ranges construct a **conflict graph**

K-Coloring for Register Allocation

- Map graph vertices onto virtual registers
- Map colors onto physical registers
- From live ranges construct a **conflict graph**
- Color the graph so that no two neighbors have the same color

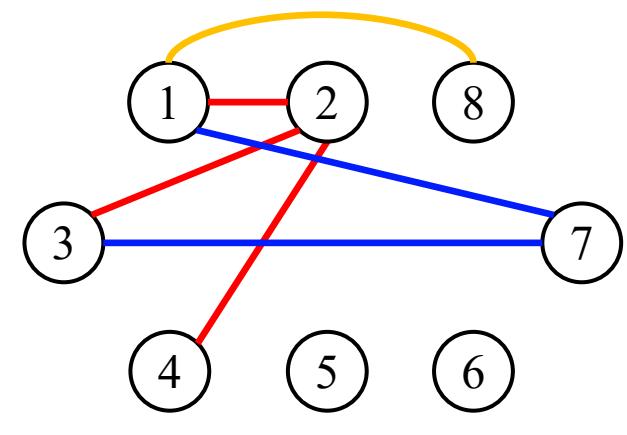
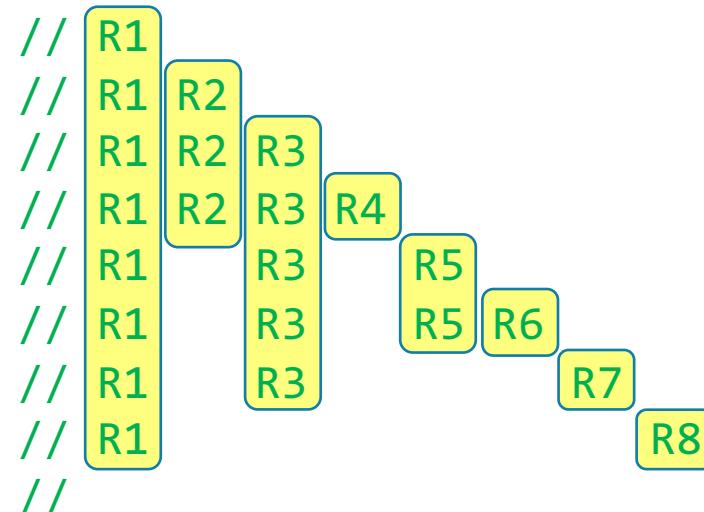
K-Coloring for Register Allocation

- Map graph vertices onto virtual registers
- Map colors onto physical registers
- From live ranges construct a **conflict graph**
- Color the graph so that no two neighbors have the same color
- If graph needs more than k colors – Spilling

Conflict Graph

- **Vertex:** virtual registers;
- **Edges:** virtual registers that have overlapping live range

1.	LD	R1	#1028
2.	LD	R2	*R1
3.	MUL	R3	R1 R2
4.	LD	R4	x
5.	SUB	R5	R4 R2
6.	LD	R6	z
7.	MUL	R7	R5 R6
8.	SUB	R8	R7 R3
9.	ST	*R1	R8

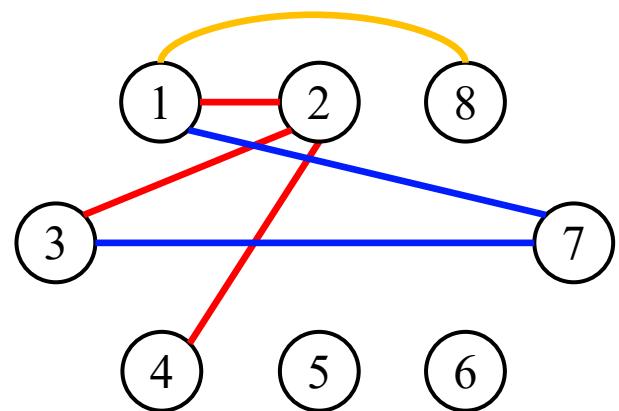


Conflict Graph

Conflict Graph

- **Vertex:** virtual registers;
- **Edges:** virtual registers that have overlapping live range

If we can use k , e.g., 4, colors, to color the graph, the 8 virtual registers can be replaced by 4 physical registers.



Conflict Graph

Coloring the Conflict Graph

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable

Coloring the Conflict Graph

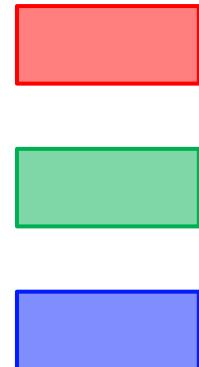
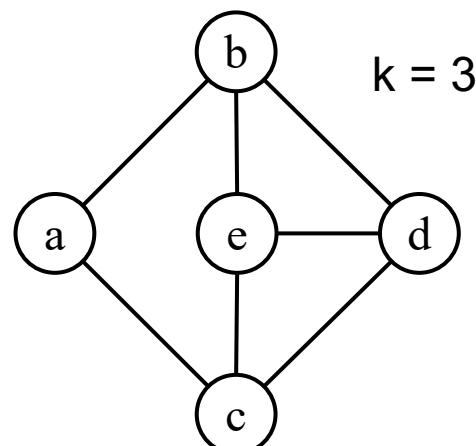
- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Chaitin's Algorithm
- ACM SIGPLAN Symposium on Compiler Construction (1982)

Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty

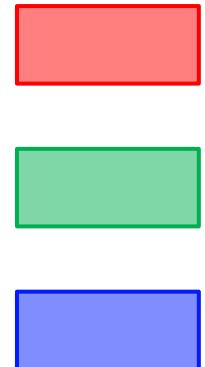
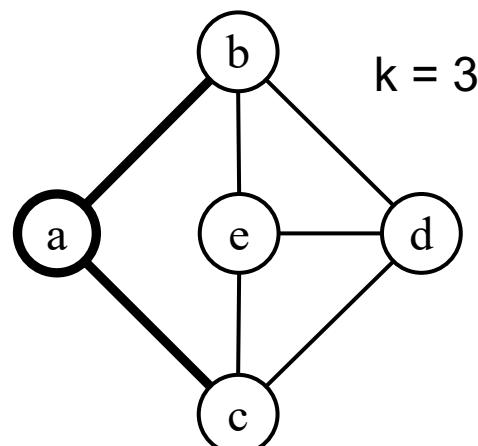
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



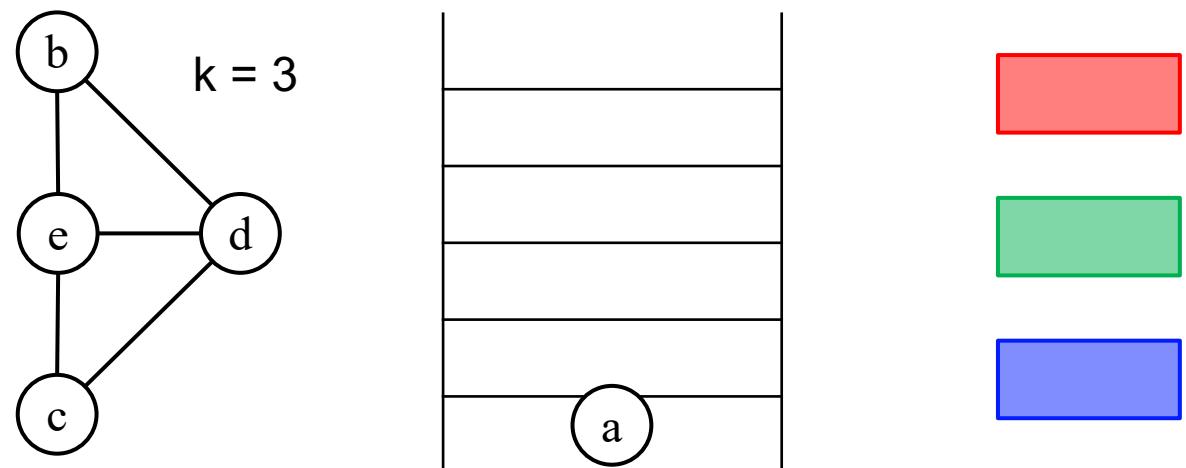
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



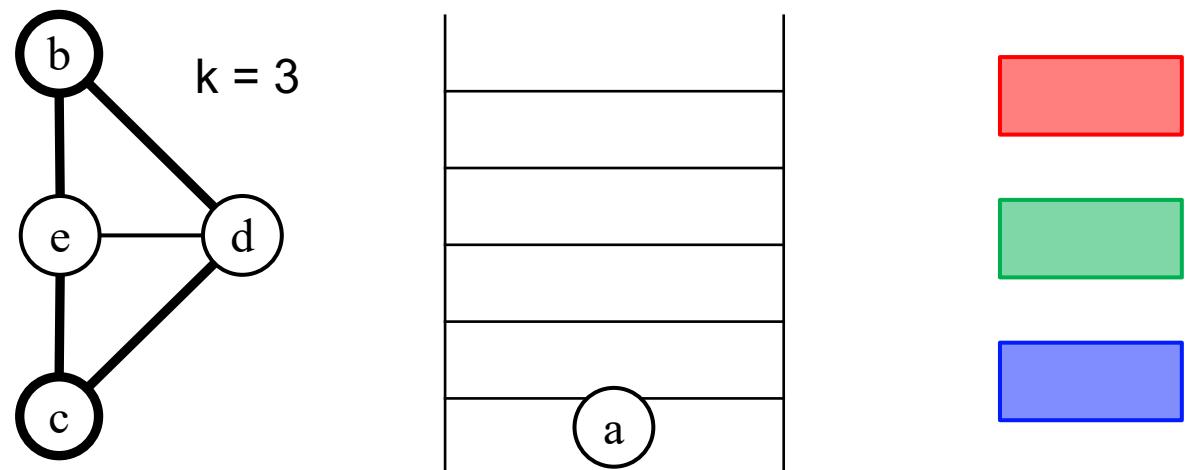
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



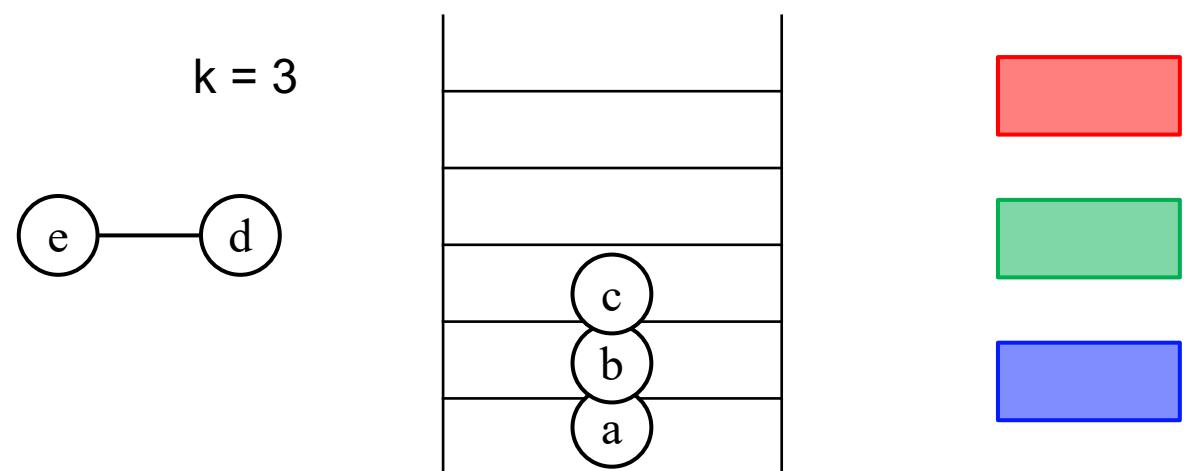
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



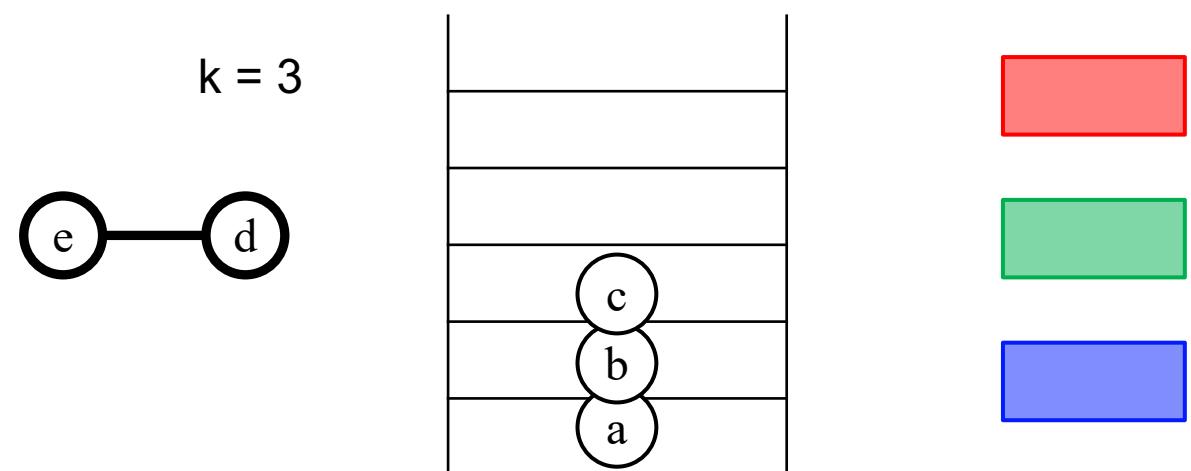
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



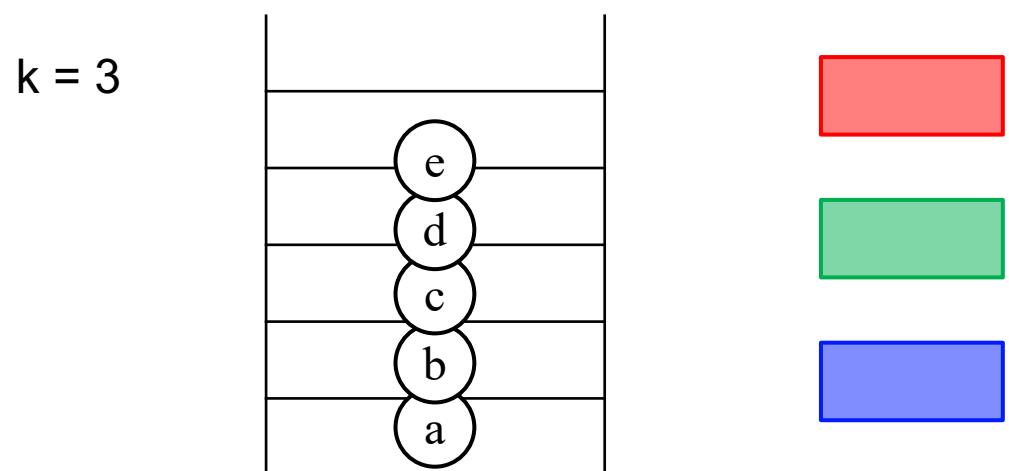
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



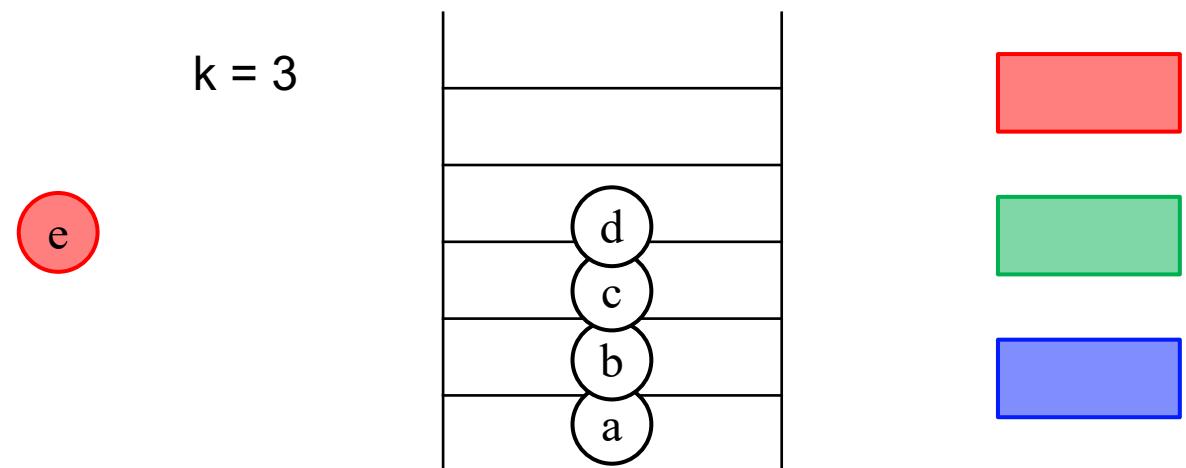
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



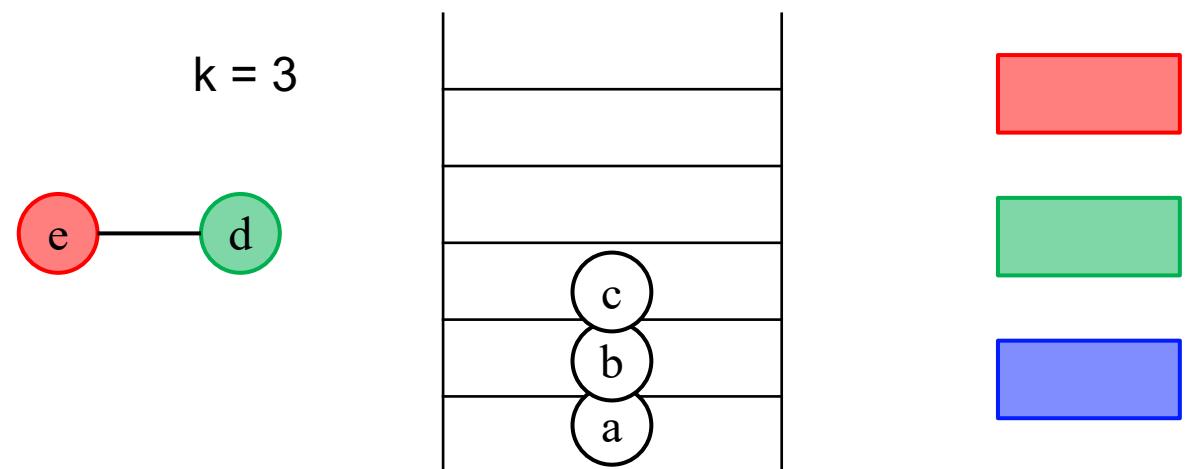
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



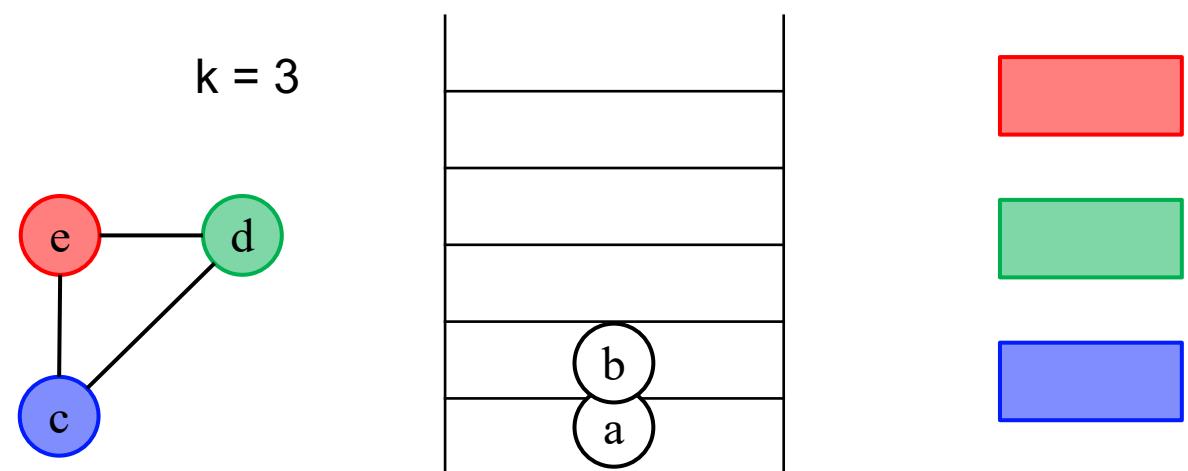
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



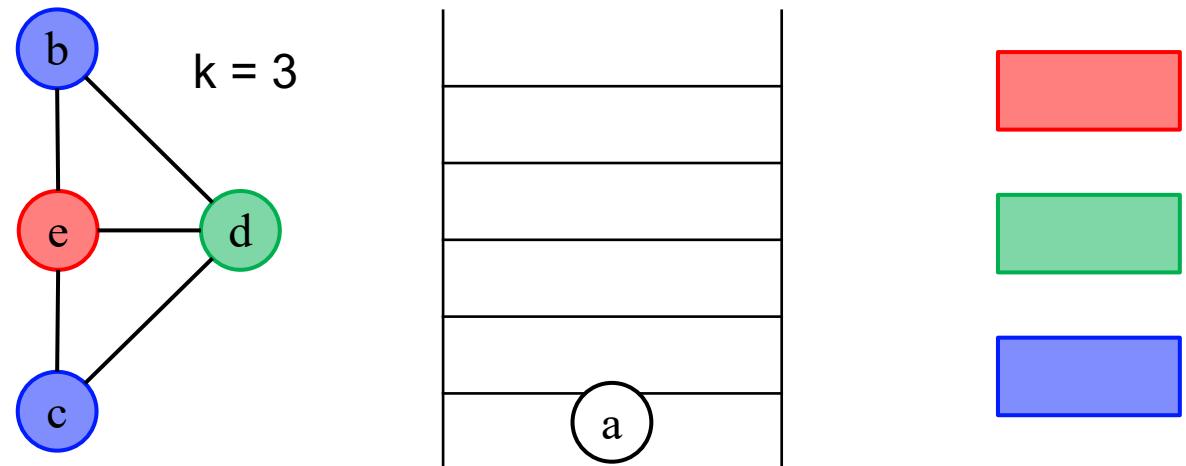
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



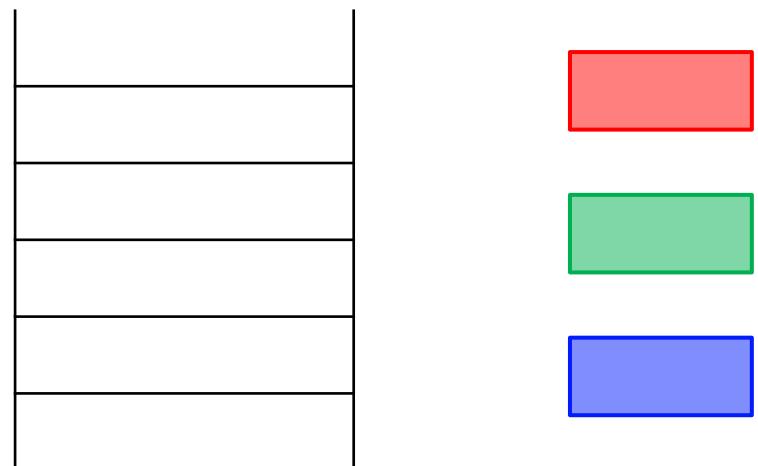
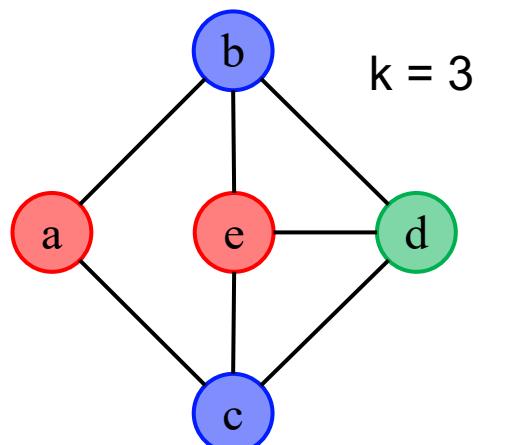
Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex such that its degree $< k$ is always k -colorable
- Remove such vertices and push them to a stack until the graph becomes empty



Chaitin's Algorithm

- When the algorithm reaches a state where every vertex has a degree $\geq k$

Chaitin's Algorithm

- When the algorithm reaches a state where every vertex has a degree $\geq k$
- Choose a vertex immediately to spill
 - put the vertex into a spill list

Chaitin's Algorithm

- When the algorithm reaches a state where every vertex has a degree $\geq k$
- Choose a vertex immediately to spill
 - put the vertex into a spill list
 - remove the vertex from the graph

Chaitin's Algorithm

- When the algorithm reaches a state where every vertex has a degree $\geq k$
- Choose a vertex immediately to spill
 - put the vertex into a spill list
 - remove the vertex from the graph
 - continue moving vertices $< k$ degree from the graph to the stack

Chaitin's Algorithm

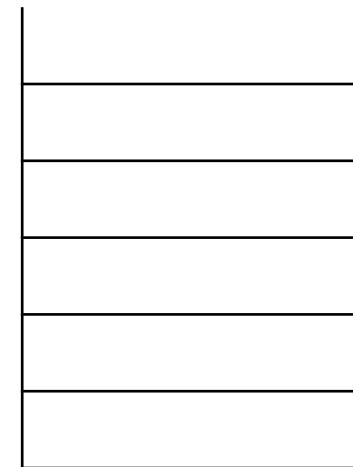
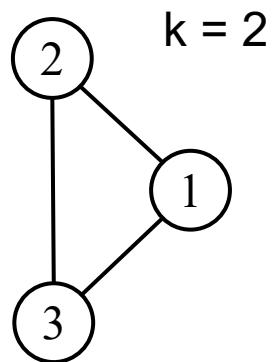
- When the algorithm reaches a state where every vertex has a degree $\geq k$
- Choose a vertex immediately to spill
 - put the vertex into a spill list
 - remove the vertex from the graph
 - continue moving vertices $< k$ degree from the graph to the stack
- If the spill list is not empty, insert spill code, rebuild conflict graph, and retry allocation.

Chaitin's Algorithm

```

1. LD   R1  #1028      // R1
2. LD   R2  *R1        // R1 R2
3. MUL  R3  R1  R2    // R1 R2 R3
4. ST   x   R3        // R1 R2
5. ST   y   R2        // R1
6. ST   z   R1        //

```



Stack



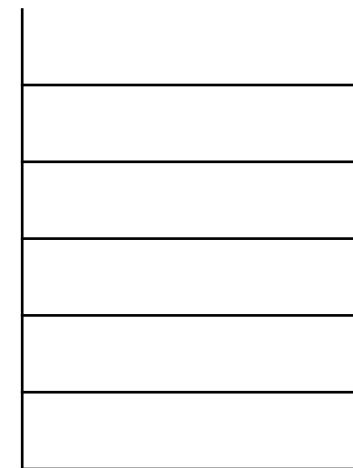
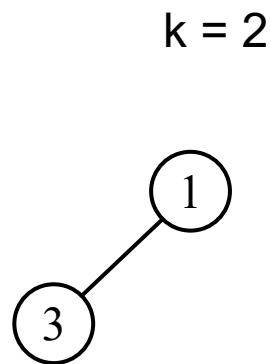
Spill List

Chaitin's Algorithm

```

1. LD   R1  #1028    // R1
2. LD   R2  *R1     // R1 R2
3. MUL  R3  R1      R2    // R1 R2 R3
4. ST   x   R3      // R1 R2
5. ST   y   R2      // R1
6. ST   z   R1      //

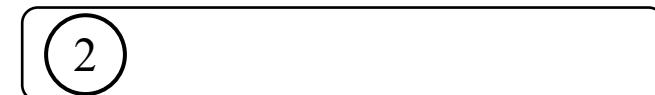
```



Stack



Spill List



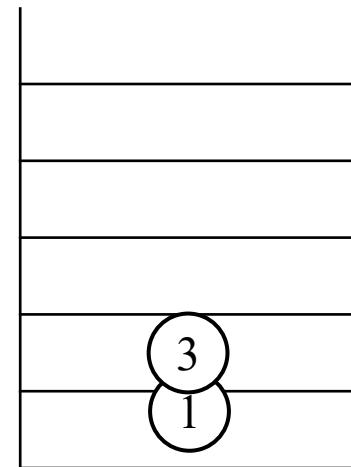
Chaitin's Algorithm

```

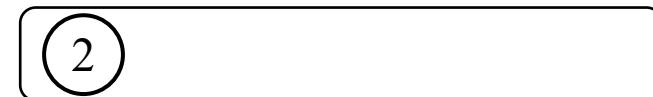
1. LD   R1  #1028    // R1
2. LD   R2  *R1      // R1 R2
3. MUL  R3  R1      // R1 R2 R3
4. ST   x   R3      // R1 R2
5. ST   y   R2      // R1
6. ST   z   R1      //

```

$k = 2$



Stack



Spill List

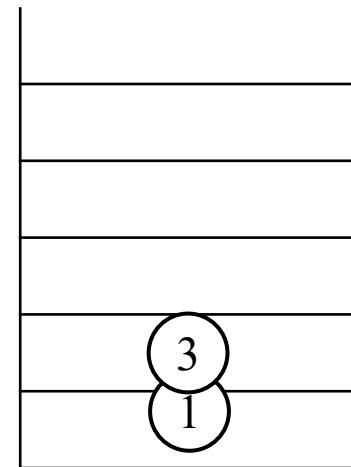
Chaitin's Algorithm

```

1. LD   R1  #1028    // R1
2. LD   R2  *R1      // R1 R2
3. MUL  R3  R1      // R1 R2 R3
4. ST   x   R3      // R1 R2
5. ST   y   R2      // R1
6. ST   z   R1      //

```

$k = 2$



Stack



Spill List

Spill R2!

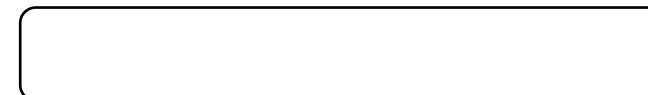
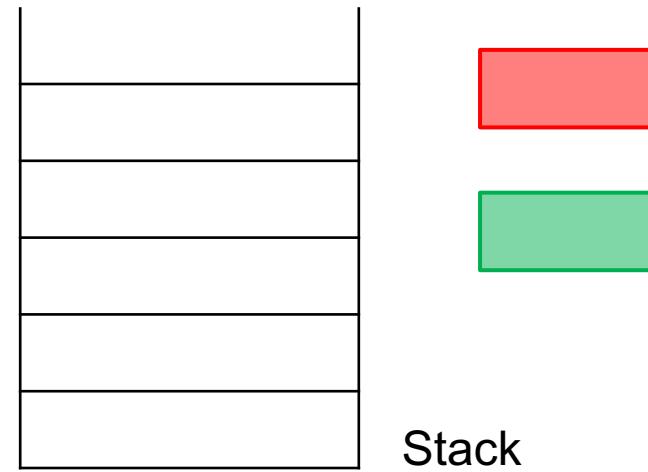
Chaitin's Algorithm

```

1. LD  R1 #1028    // R1
2. LD  R2 *R1      // R1 R2
3. ST  a  R2      // R1
4. MUL R3 R1      R1 // R1      R3
5. ST  x  R3      // R1
6. LD  R2 a        // R1 R2
7. ST  y  R2      // R1
8. ST  z  R1      //

```

k = 2

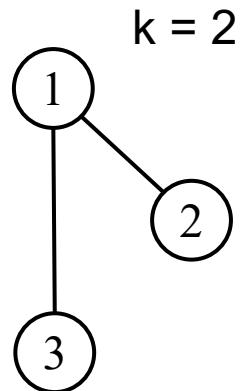


Chaitin's Algorithm

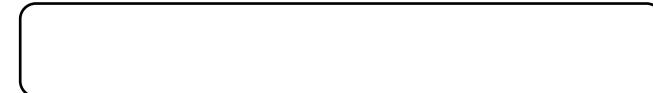
```

1. LD  R1 #1028    // R1
2. LD  R2 *R1      // R1 R2
3. ST  a  R2      // R1
4. MUL R3 R1      R1 // R1      R3
5. ST  x  R3      // R1
6. LD  R2 a        // R1 R2
7. ST  y  R2      // R1
8. ST  z  R1      //

```



Stack



Spill List

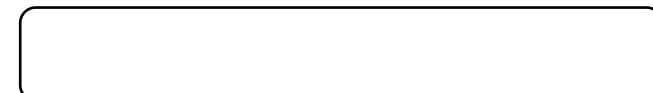
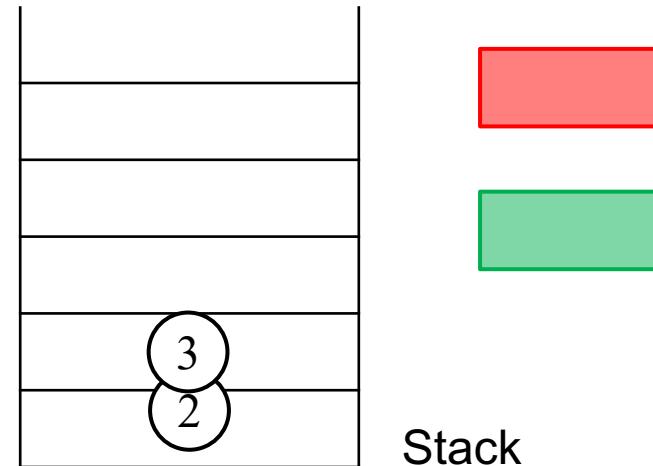
Chaitin's Algorithm

```

1. LD  R1 #1028    // R1
2. LD  R2 *R1      // R1 R2
3. ST  a  R2      // R1
4. MUL R3 R1      R1 // R1     R3
5. ST  x  R3      // R1
6. LD  R2 a        // R1 R2
7. ST  y  R2      // R1
8. ST  z  R1      //

```

① k = 2



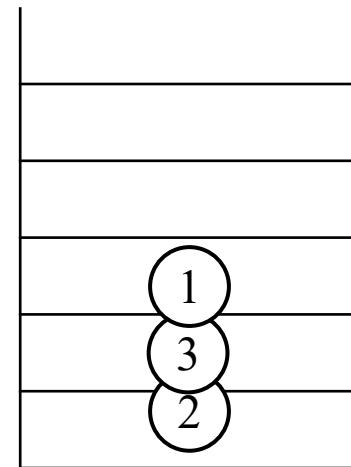
Chaitin's Algorithm

```

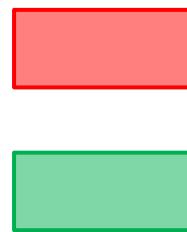
1. LD  R1 #1028    // R1
2. LD  R2 *R1      // R1 R2
3. ST  a  R2      // R1
4. MUL R3 R1      R1 // R1      R3
5. ST  x  R3      // R1
6. LD  R2 a        // R1 R2
7. ST  y  R2      // R1
8. ST  z  R1      //

```

$k = 2$



Stack



Spill List

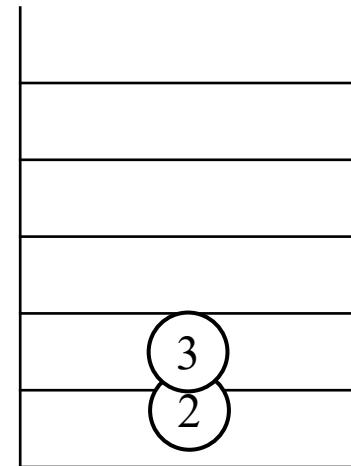
Chaitin's Algorithm

```

1. LD  R1 #1028    // R1
2. LD  R2 *R1      // R1 R2
3. ST  a  R2      // R1
4. MUL R3 R1      R1 // R1     R3
5. ST  x  R3      // R1
6. LD  R2 a        // R1 R2
7. ST  y  R2      // R1
8. ST  z  R1      //

```

k = 2
1



Stack



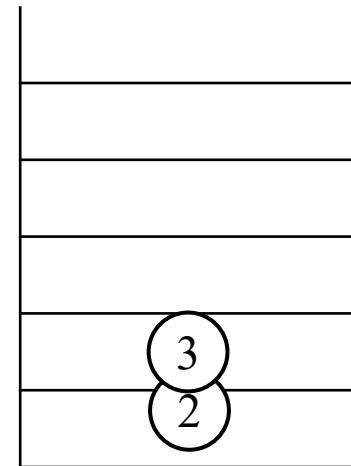
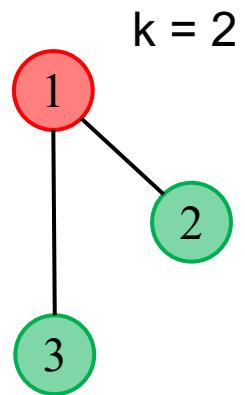
Spill List

Chaitin's Algorithm

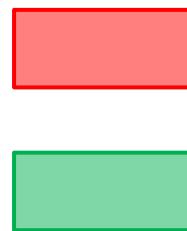
```

1. LD  R1  #1028    // R1
2. LD  R2  *R1    // R1 R2
3. ST  a   R2    // R1
4. MUL R3  R1      R1    // R1      R3
5. ST  x   R3    // R1
6. LD  R2  a    // R1 R2
7. ST  y   R2    // R1
8. ST  z   R1    //

```



Stack



Spill List

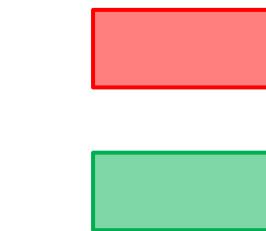
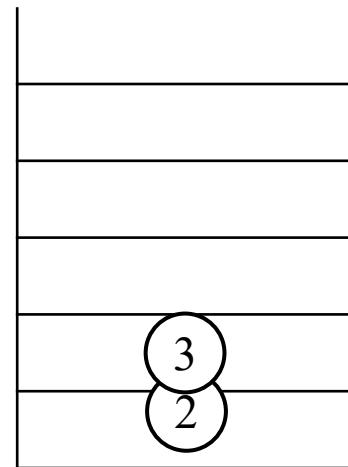
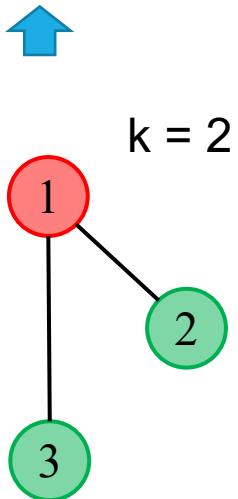
Chaitin's Algorithm

```

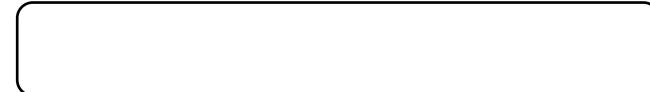
1. LD  R1  #1028    // R1
2. LD  R2  *R1     // R1 R2
3. ST  a   R2      // R1
4. MUL R3  R1      R1 // R1      R3
5. ST  x   R3      // R1
6. LD  R2  a       // R1 R2
7. ST  y   R2      // R1
8. ST  z   R1      //

```

Replace R1 with R_{red}
 Replace R2/R3 with R_{green}



Stack

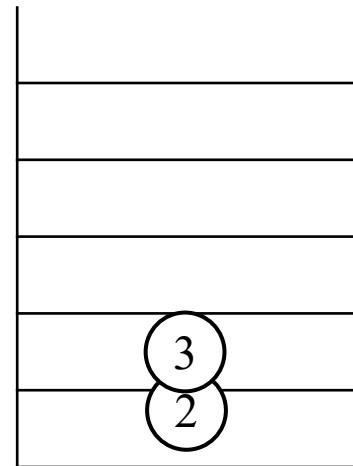
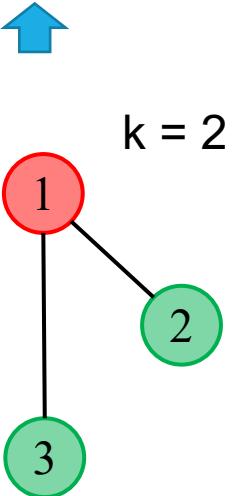


Spill List

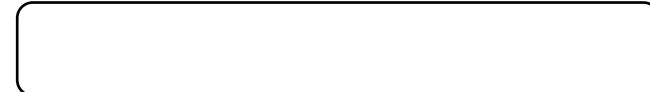
Chaitin's Algorithm

1.	LD	R_{red}	#1028
2.	LD	R_{green}	$*R_{\text{red}}$
3.	ST	a	R_{green}
4.	MUL	R_{green}	R_{red} R_{red}
5.	ST	x	R_{green}
6.	LD	R_{green}	a
7.	ST	y	R_{green}
8.	ST	z	R_{red}

Replace R1 with R_{red}
 Replace R2/R3 with R_{green}



Stack



Spill List

Chaitin-Briggs Algorithm

- When Chaitin's algorithm reaches a state where every vertex has a degree $\geq k$
- Choose a vertex immediately to spill
 - put the vertex into a spill list
 - ...

Chaitin-Briggs Algorithm

- When Chaitin's algorithm reaches a state where every vertex has a degree $\geq k$
- Choose a vertex immediately to spill
 - put the vertex into a spill list
 - ...
- **Briggs:**
 - Degree of a vertex is a **loose** upper bound on colorability
 - A vertex such that its degree $< k$ is **always** k -colorable

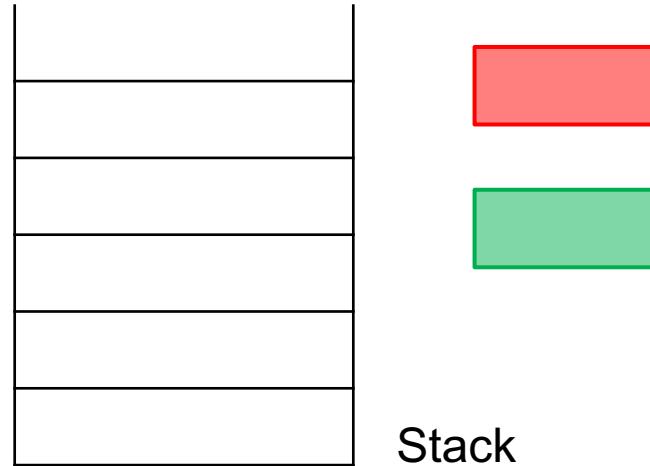
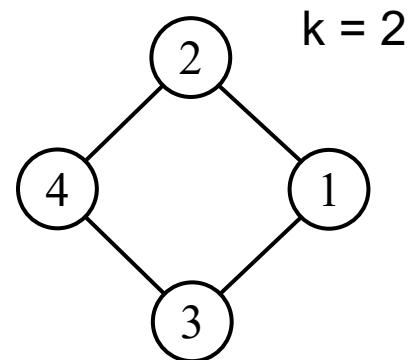
Chaitin-Briggs Algorithm

- When Chaitin's algorithm reaches a state where every vertex has a degree $\geq k$
- Choose a vertex immediately to spill
 - put the vertex into a spill list
 - ...
- **Briggs:**
 - Degree of a vertex is a **loose** upper bound on colorability
 - A vertex such that its degree $< k$ is **always** k -colorable
 - A vertex such that its degree $\geq k$ **may also be** k -colorable

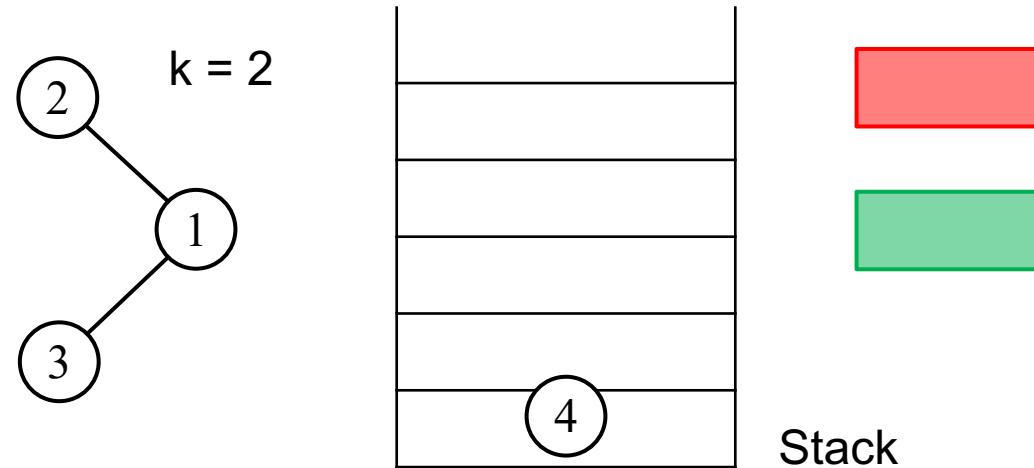
Chaitin-Briggs Algorithm

- When Chaitin's algorithm reaches a state where every vertex has a degree $\geq k$
- Choose a vertex immediately to spill
 - ~~put the vertex into a spill list~~
 - ...
- **Briggs:**
 - Push the vertex (to spill) to the stack.
 - We may have a color available when it is popped.

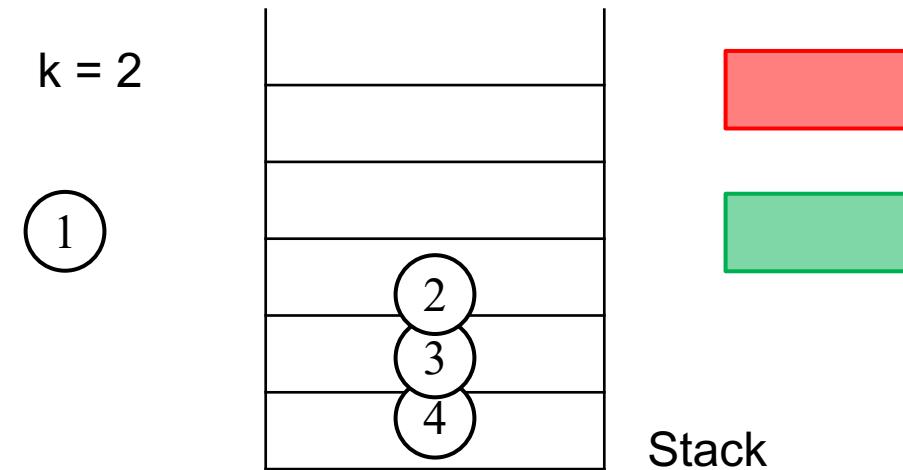
Chaitin-Briggs Algorithm



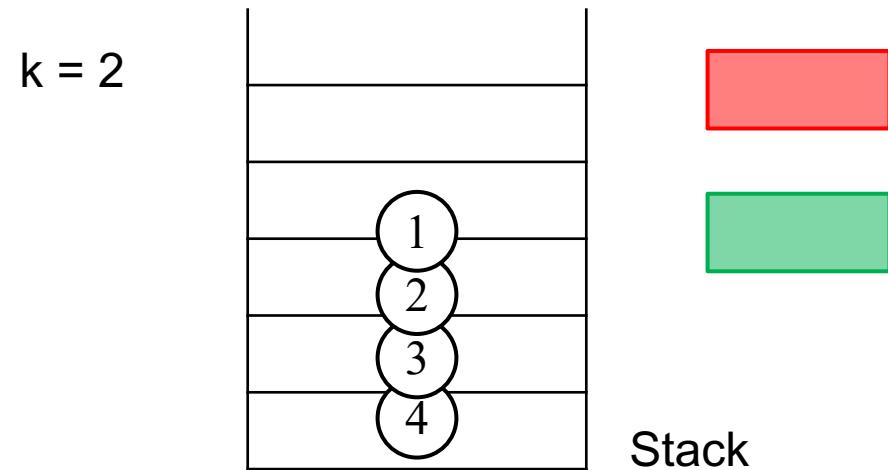
Chaitin-Briggs Algorithm



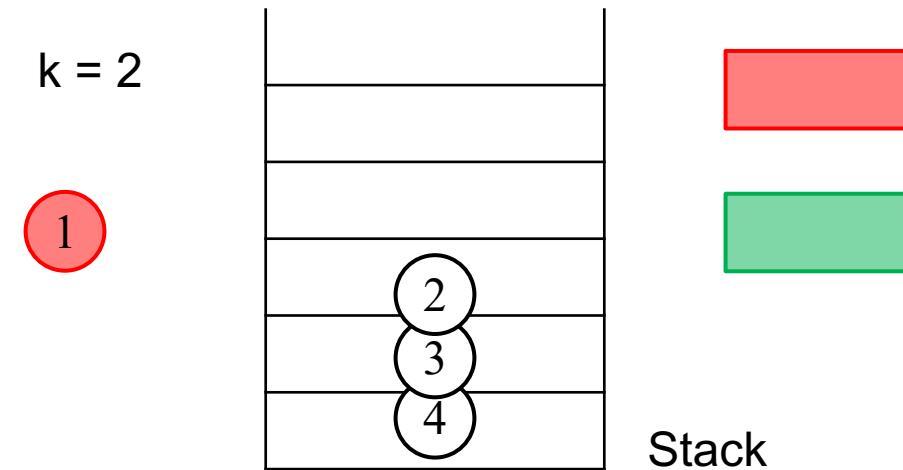
Chaitin-Briggs Algorithm



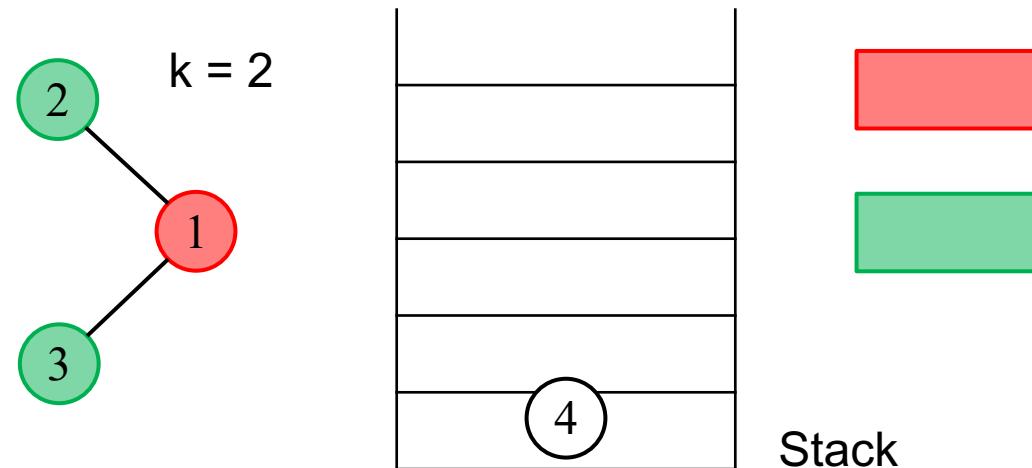
Chaitin-Briggs Algorithm



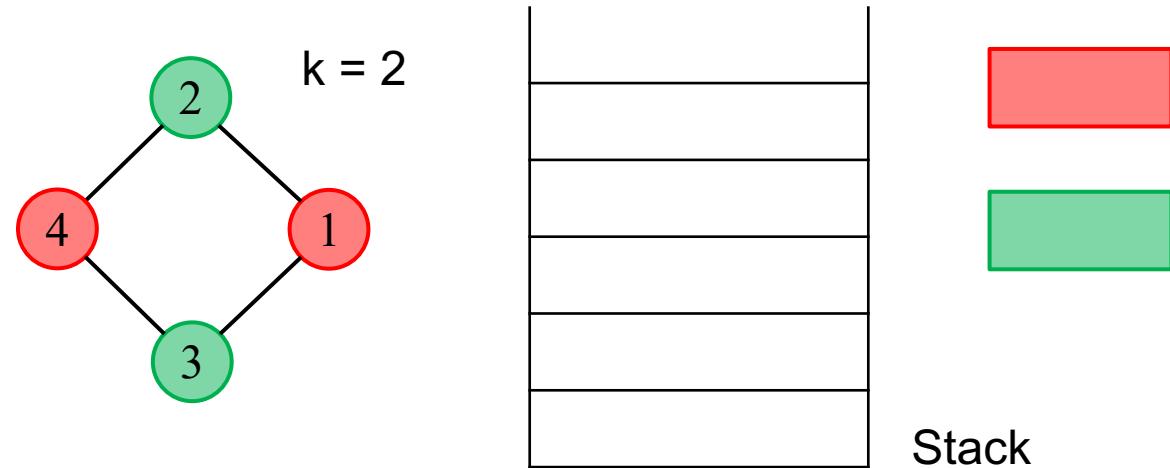
Chaitin-Briggs Algorithm



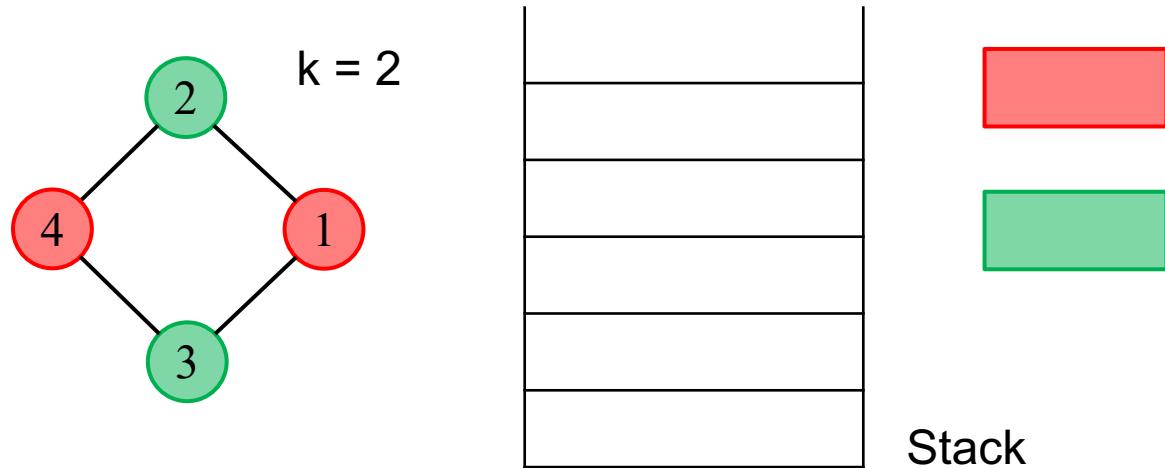
Chaitin-Briggs Algorithm



Chaitin-Briggs Algorithm

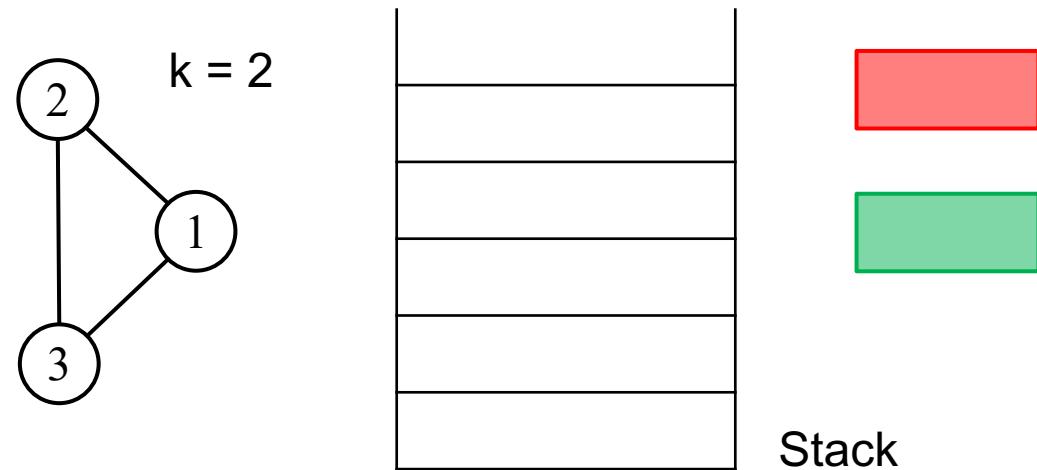


Chaitin-Briggs Algorithm

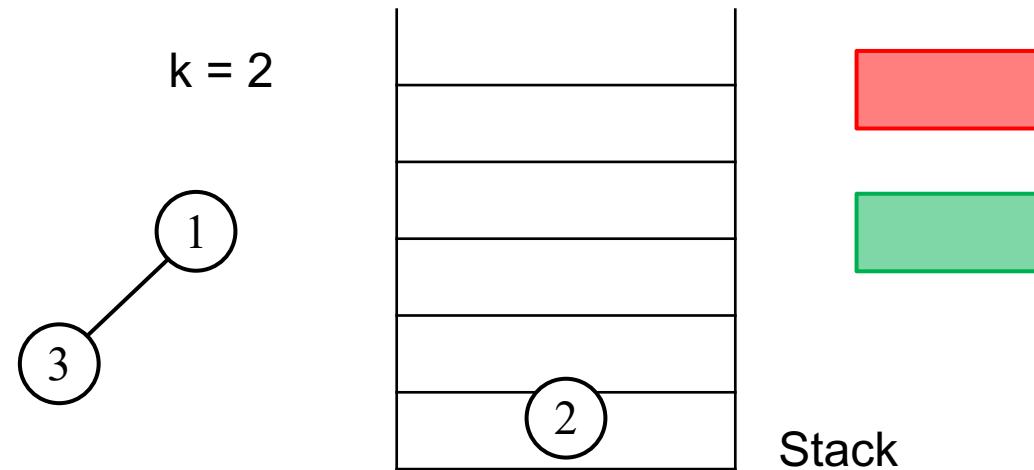


Sometimes, it is not necessary to spill!

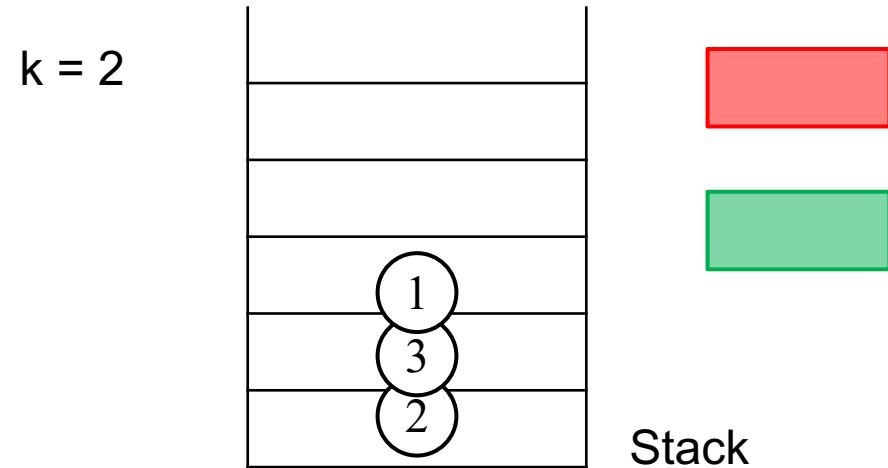
Chaitin-Briggs Algorithm



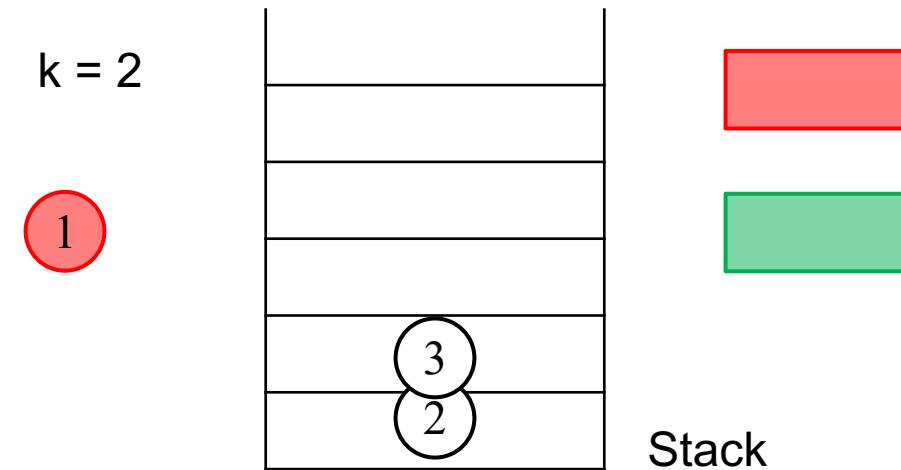
Chaitin-Briggs Algorithm



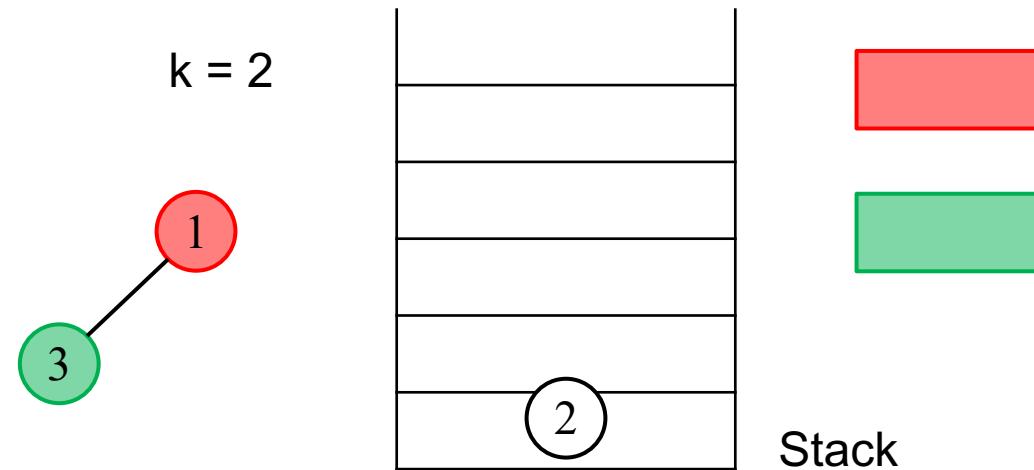
Chaitin-Briggs Algorithm



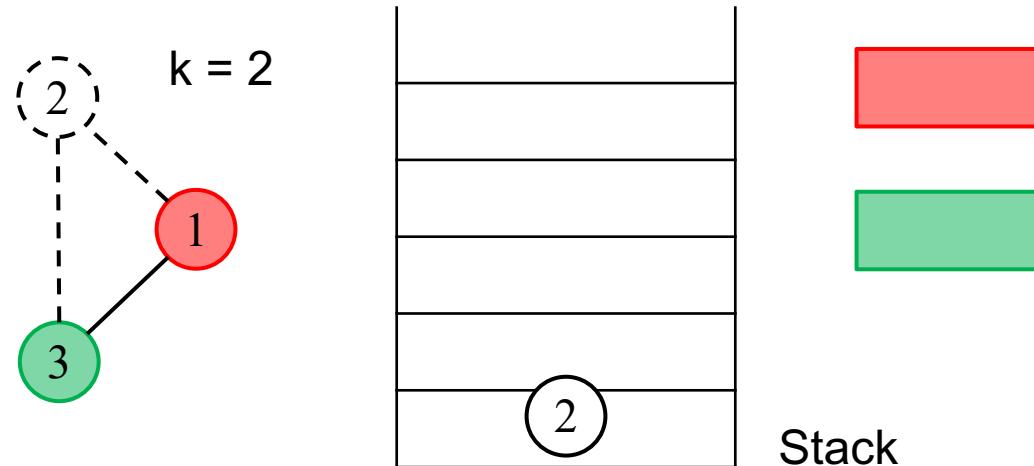
Chaitin-Briggs Algorithm



Chaitin-Briggs Algorithm



Chaitin-Briggs Algorithm



When there is not any available color, spill it.

Chaitin-Briggs Algorithm

- Compared to Chaitin's algorithm, Chaitin-Briggs algorithm delays the spill operation
 - saves a spill list
 - reduces spill code

Spill Candidates

- Minimize spill cost, *i.e.*, the loads/stores needed
 - *e.g.*, avoid inner loops

Spill Candidates

- Minimize spill cost, *i.e.*, the loads/stores needed
 - *e.g.*, avoid inner loops
- The higher the degree of a vertex
 - The greater the chance that spilling it will help coloring

Spill Candidates

- Minimize spill cost, *i.e.*, the loads/stores needed
 - *e.g.*, avoid inner loops
- The higher the degree of a vertex
 - The greater the chance that spilling it will help coloring
- Don't spill a value which is defined immediately followed by use
 - Splitting does not decrease live range

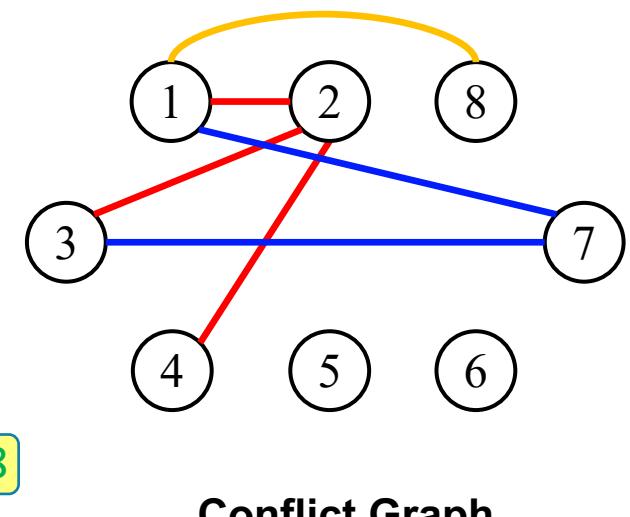
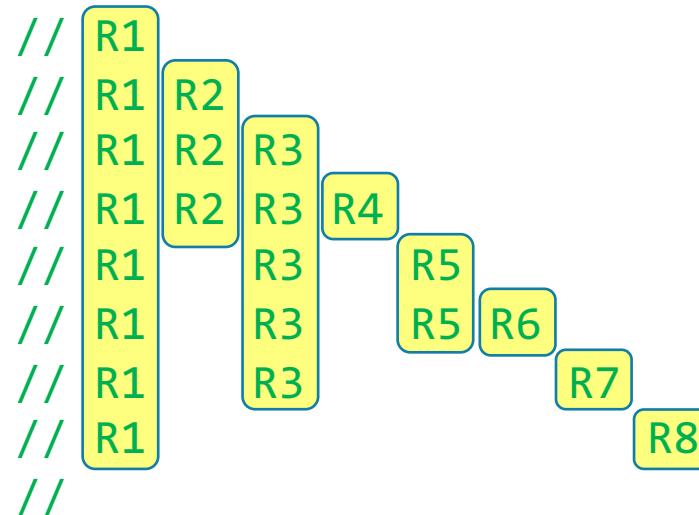
Alternative Spilling

- Splitting Live Ranges, Coalescing Virtual Registers, etc.

Alternative Spilling

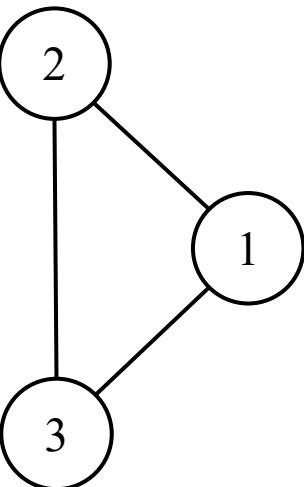
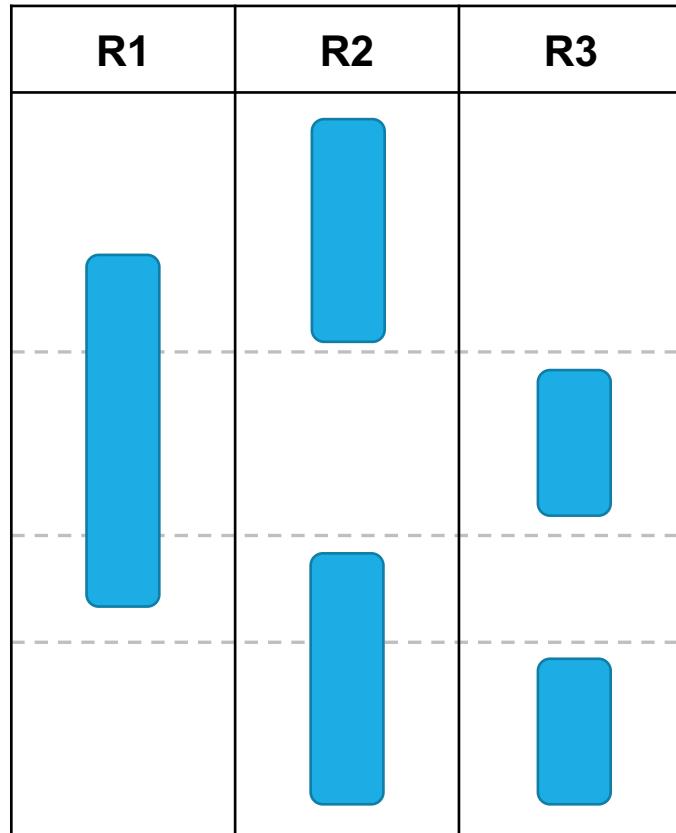
- Splitting Live Ranges, Coalescing Virtual Registers, etc.
- Recall the **conflict graph**:

1. LD	R1	#1028	
2. LD	R2	*R1	
3. MUL	R3	R1	R2
4. LD	R4	x	
5. SUB	R5	R4	R2
6. LD	R6	z	
7. MUL	R7	R5	R6
8. SUB	R8	R7	R3
9. ST	*R1	R8	



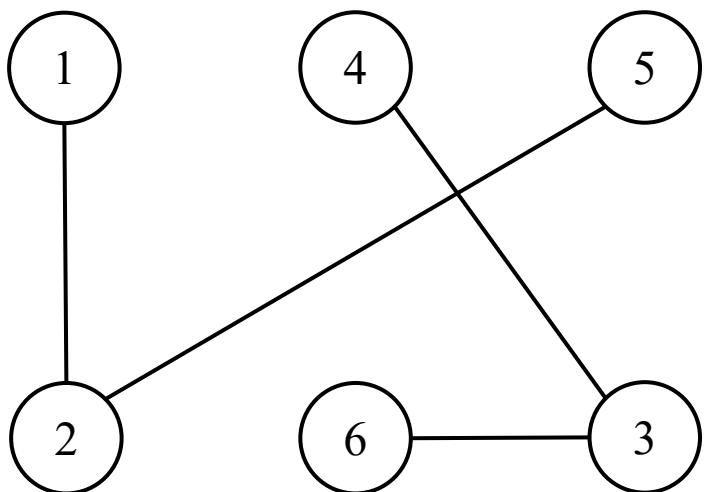
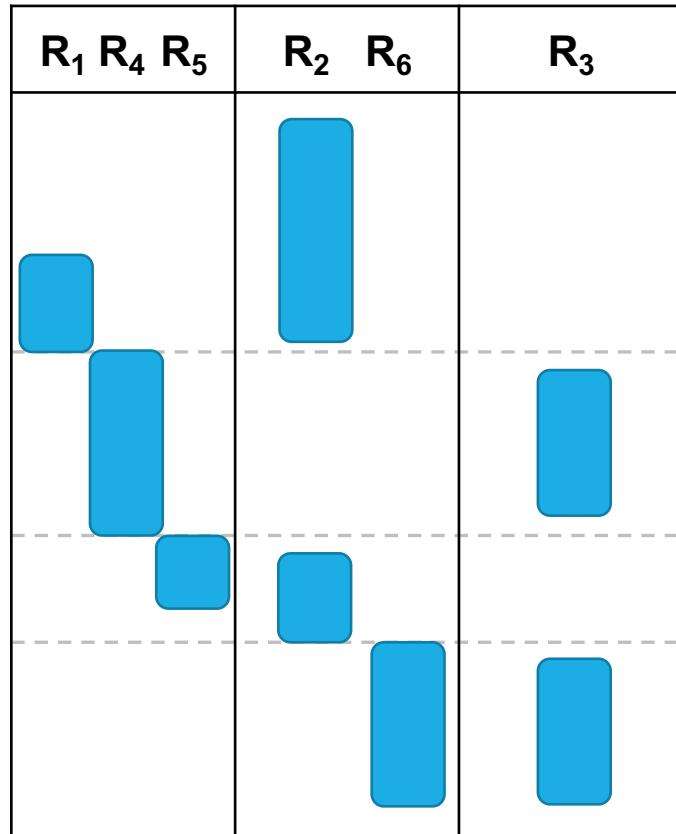
Conflict Graph

Splitting Live Range



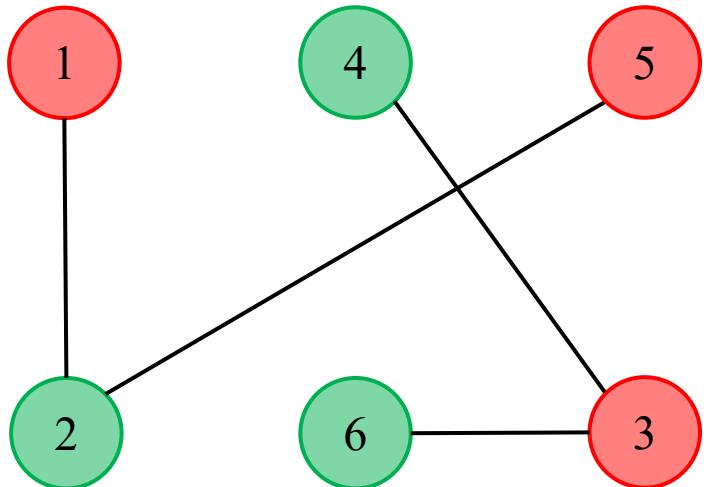
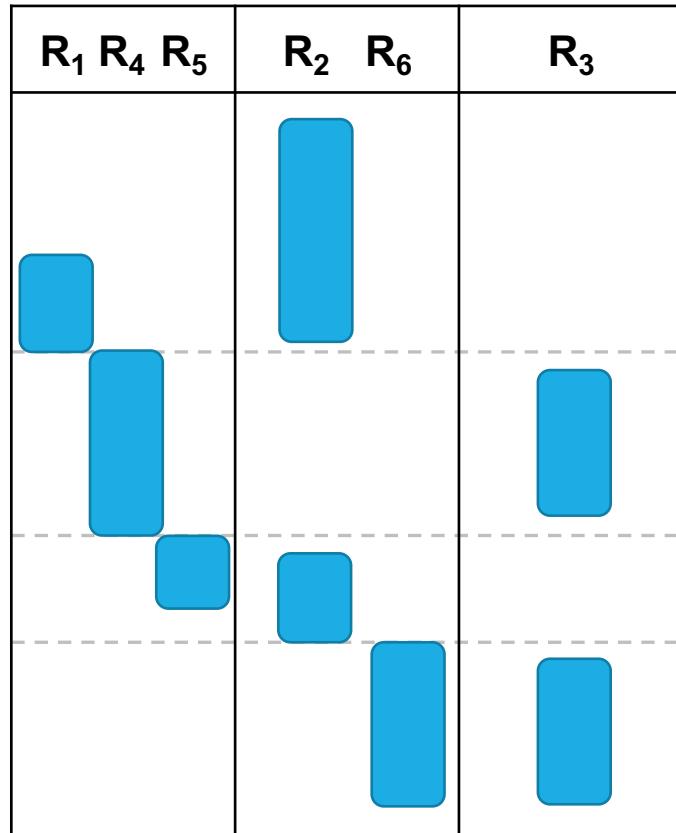
The conflict graph is not 2-colorable!

Splitting Live Range



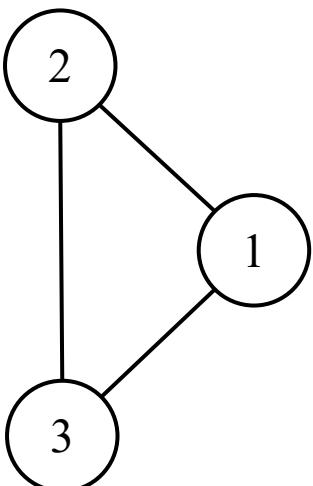
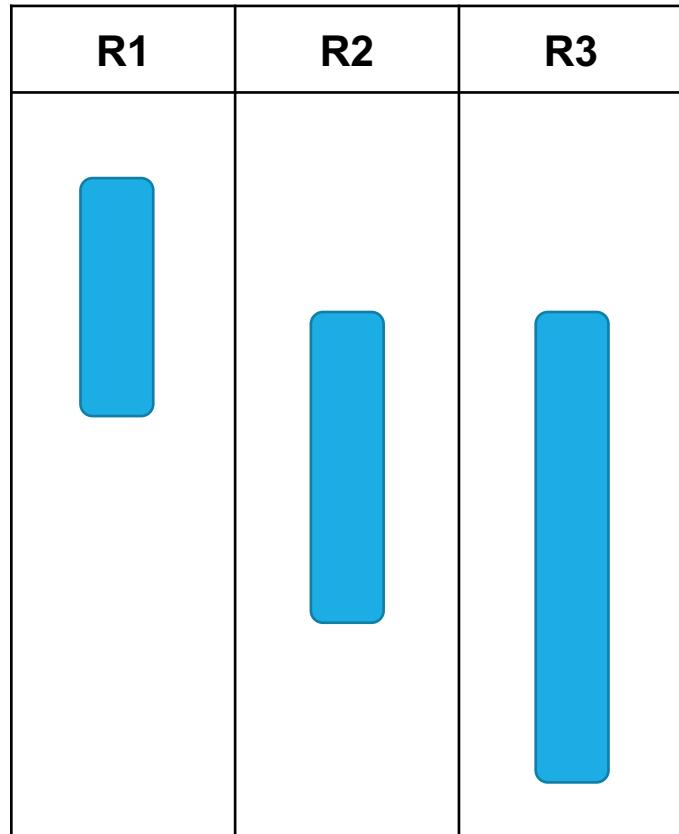
The conflict graph is 2-colorable!

Splitting Live Range

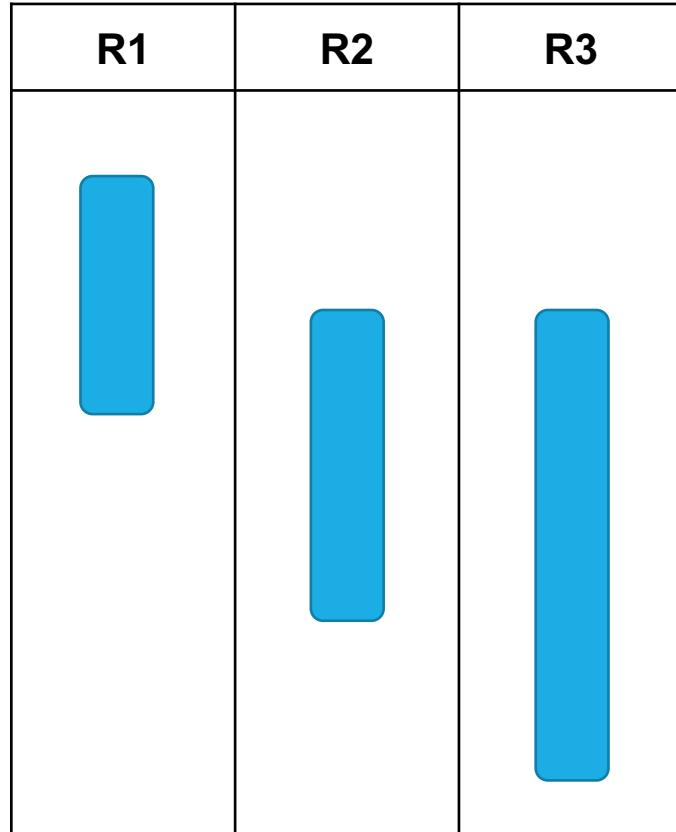


The conflict graph is 2-colorable!

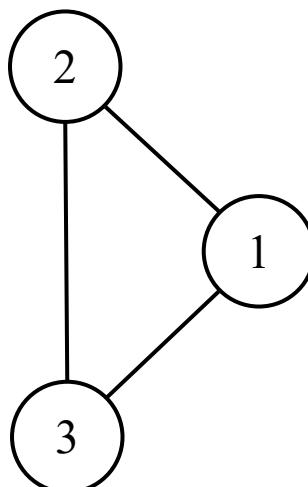
Coalescing Virtual Registers



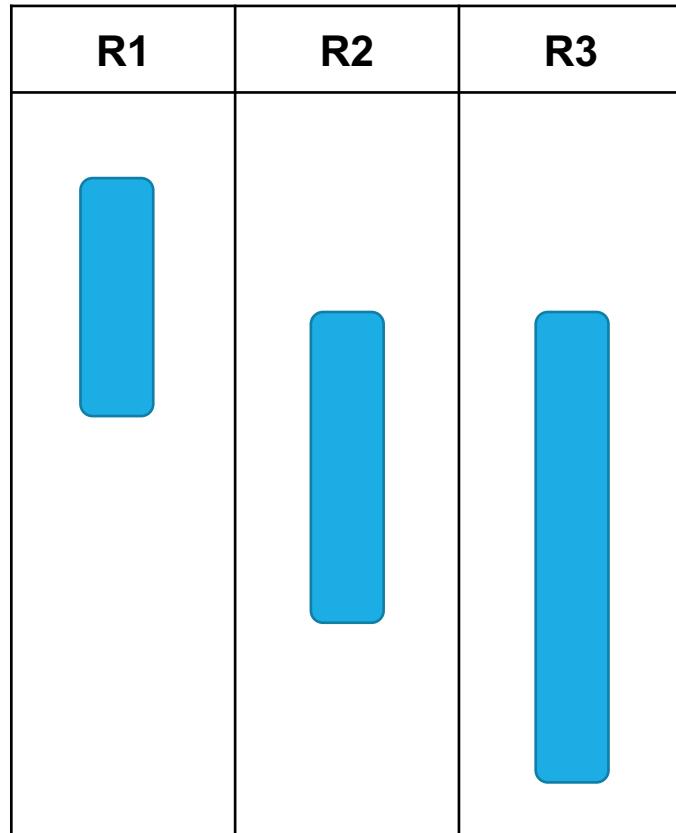
Coalescing Virtual Registers



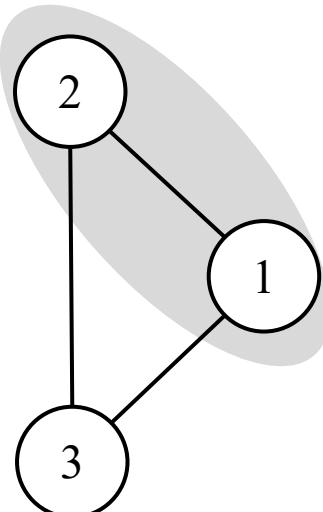
Assume $R1 = R2$ by $LD\ R2\ R1$



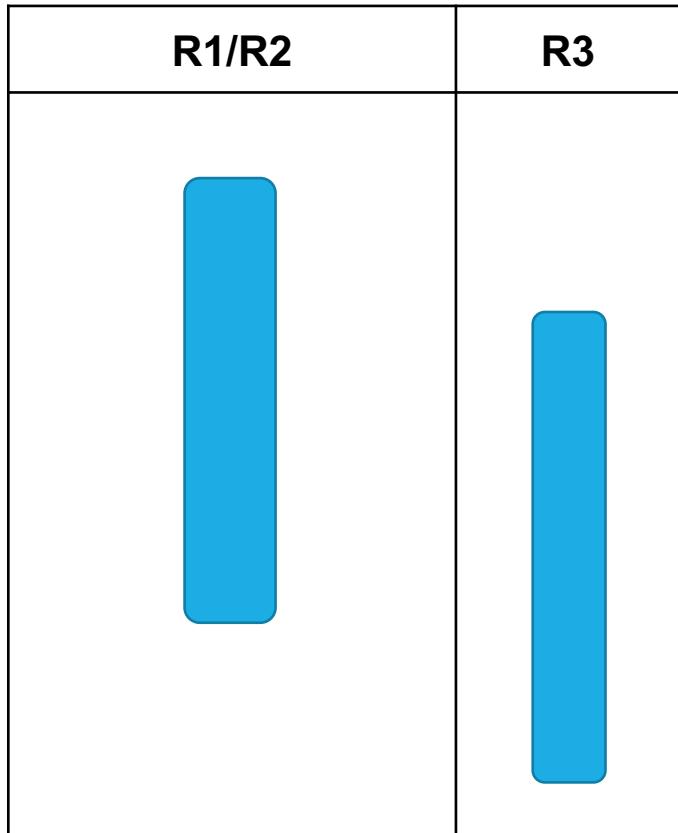
Coalescing Virtual Registers



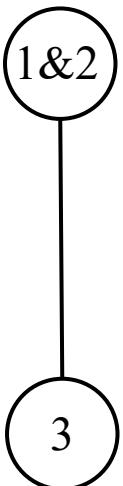
Assume $R1 = R2$ by $LD\ R2\ R1$



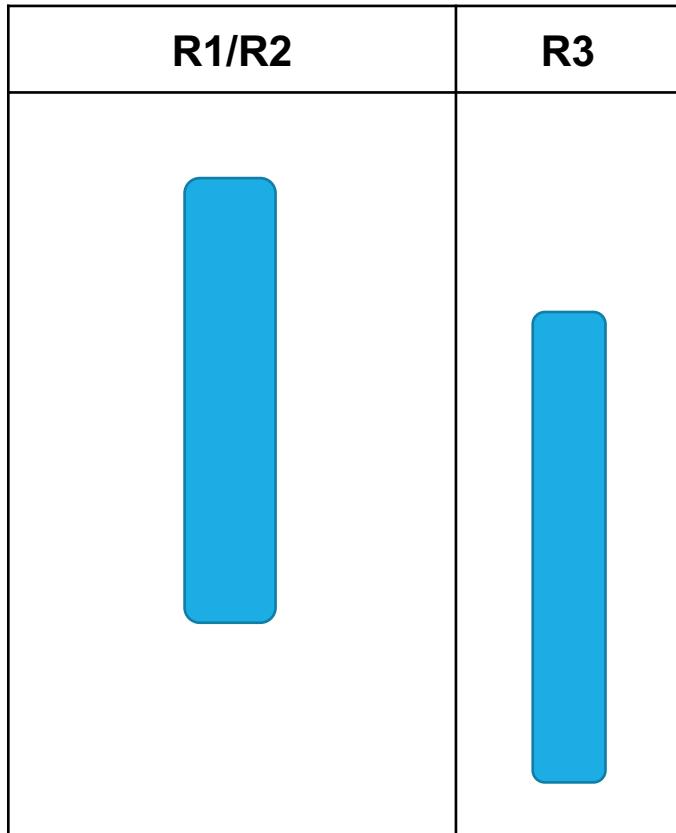
Coalescing Virtual Registers



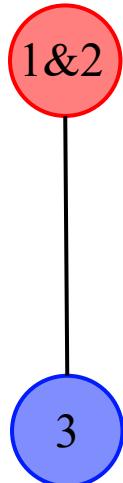
Assume R1 = R2 by LD R2 R1



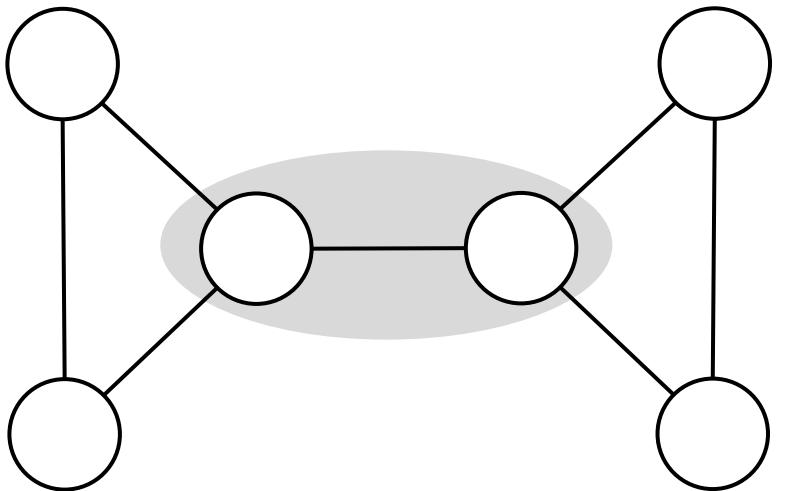
Coalescing Virtual Registers



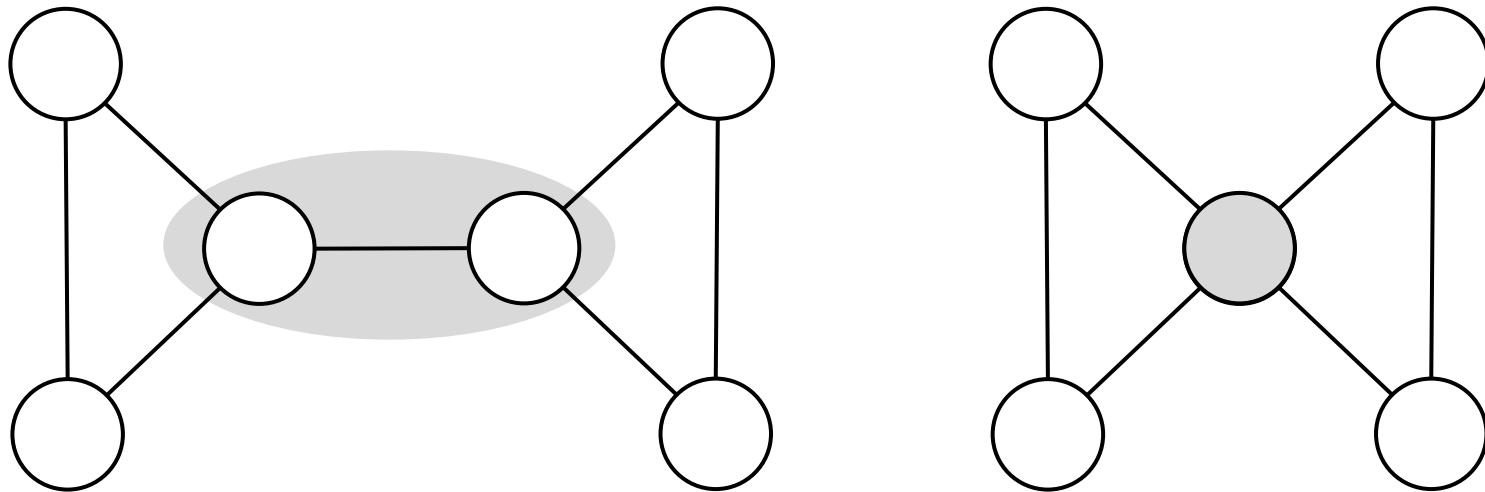
Assume $R1 = R2$ by LD R2 R1



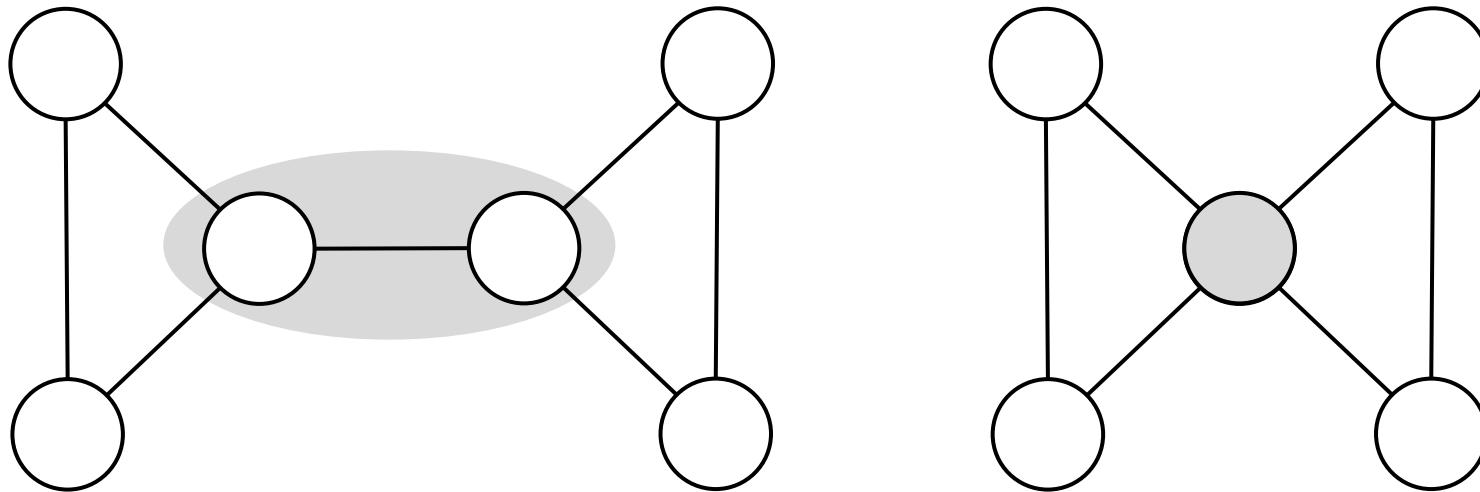
Coalescing Not Always Work



Coalescing Not Always Work



Coalescing Not Always Work

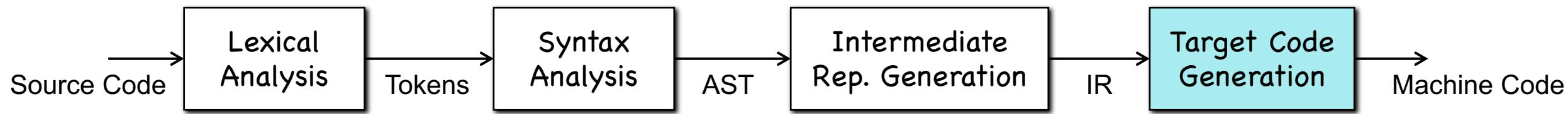


A vertex such that its degree $< k$ is always k -colorable



Degree: $3 \rightarrow 4$
Harder to color

Summary



- **Code Generation**/Target Code Model
/Memory Allocation
- **Gen Better Code**/Preliminaries (basic block, dominance frontier, ...)
/Local Optimization (tree-based selection, peephole, ...)
/Register Allocation (Chaitin's algorithm)

THANKS!