RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Register Allocation Part 2

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- First homework due this Wednesday, February 9.
  Do you need an extension?

- First project will be posted this week!

Local Register Allocation (and Assignment)

Readings: EaC 13.1-13.2, Appendix A (ILOC)

Local: within single basic block
Global: across procedure/function

Review - Basic Approach of Allocators

Allocator may need to reserve physical registers to ensure feasibility

Notation:
k is the number of registers on the target machine

- Must be able to compute memory addresses
- Requires some minimal set of registers, *F*
  - → *F* depends on target architecture
- F contains registers to make spilling work
  - → set F registers "aside" for address computation & instruction execution, i.e. these are not available for register assignment
- Note: F physical registers need to be able to support the pathological case where all virtual registers are spilled

What if $k - |F| < |values| < k$?

- The allocator can either
  - → Check for this situation
  - → Accept the fact that the technique is an approximation

Top-down Versus Bottom-up Allocation

Top-down allocator

- May use notion of "live ranges" of virtual registers
- Work from "external" notion of what is important
- Assign registers in priority order
- Register assignment remains fixed for entire basic block
- Save some registers for the values relegated to memory (feasible set F)

Bottom-up allocator

- Work from detailed knowledge about problem instance
- Incorporate knowledge of partial solution at each step
- Register assignment may change across basic block
- Save some registers for the values relegated to memory (feasible set F)

Live Ranges (live on exit)

Assume i and j are two instructions in a basic block

A **value** (virtual register) is *live* between its *definition* and its *uses*

- Find definitions (x ← …) and uses (y ← … x …)
- From definition to last use is its *live range*
  - → How many (static) definitions can you have for a virtual register?
- Can represent live range as an interval [*i*,*j*] (in block)
  - → *live on exit*

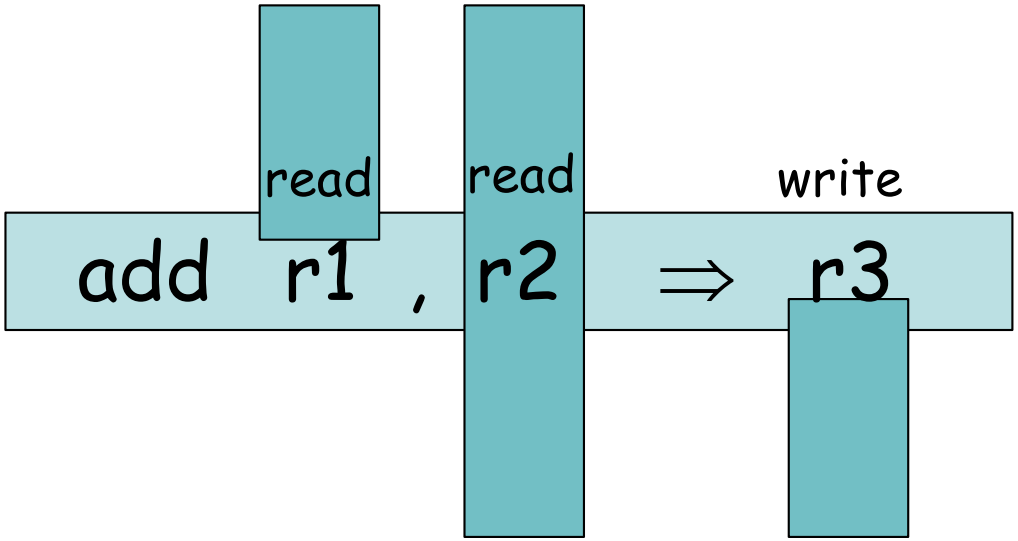
Let *MAXLIVE* be the maximum, over each instruction *i* in the block, of the number of values (virtual registers) live at *i*.

- If MAXLIVE ≤ *k*, allocation should be easy
  - → no need to reserve set of *F* registers for spilling
- If MAXLIVE > *k*, some values must be spilled to memory

Finding live ranges is harder in the global case

Live Ranges (live on exit)

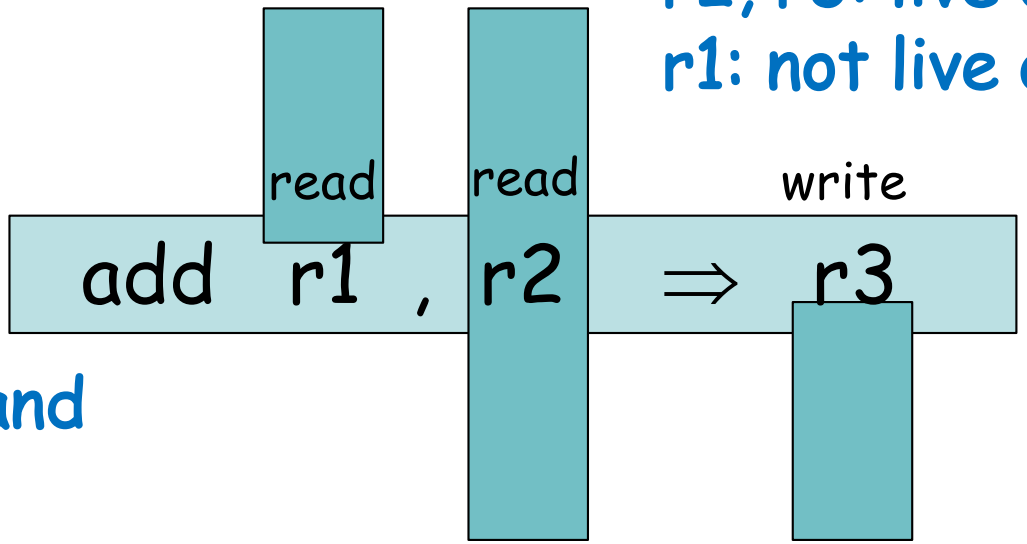
r1 does not occur after instruction;
r2 occurs before and after instr.;
r3 occurs after instr.



Live Ranges (live on exit)

r1 does not occur after instruction;
r2 occurs before and after instr.;
r3 occurs after instr.

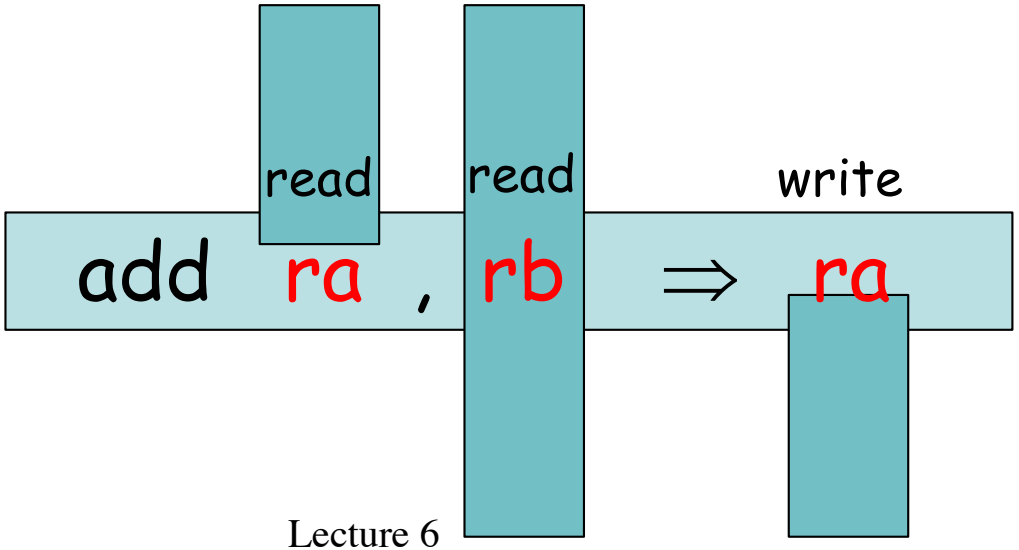
r2, r3: live on exit
r1: not live on exit



Live ranges of r1 and r3 do not overlap!

After register allocation:

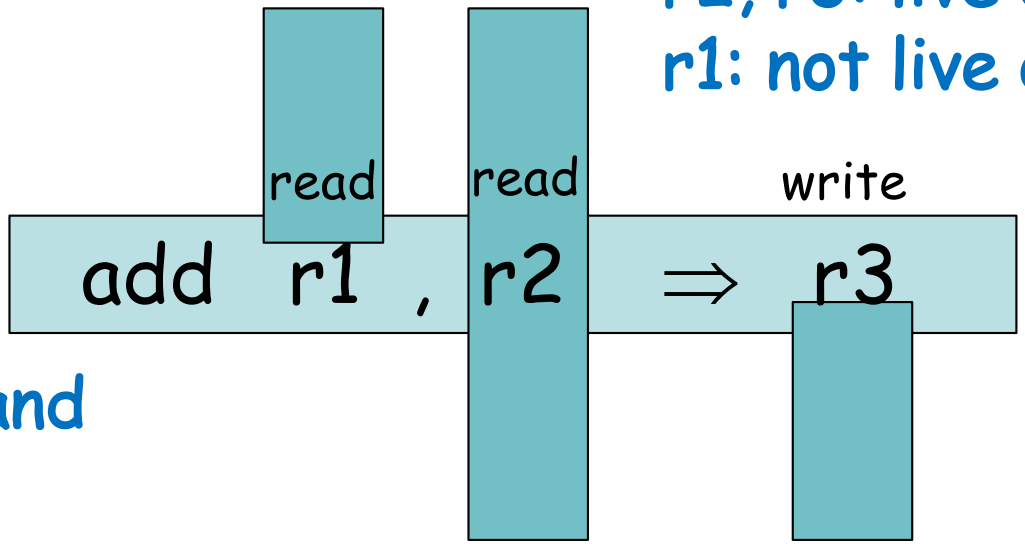
Physical register ra assigned to r1 can also be reassigned to r3



Live Ranges (live on exit)

r1 does not occur after instruction;
r2 occurs before and after instr.;
r3 occurs after instr.

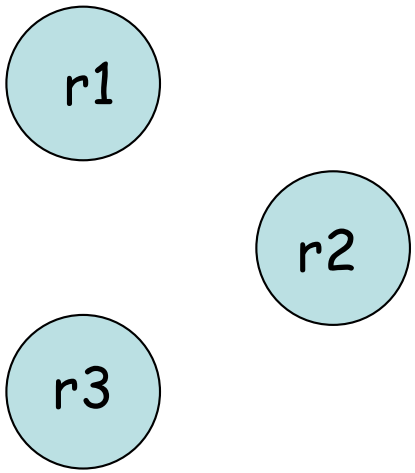
r2, r3: live on exit
r1: not live on exit



Live ranges of r1 and r3 do not overlap!

Register allocation as a graph coloring problem:

Interference Graph:
nodes: live ranges

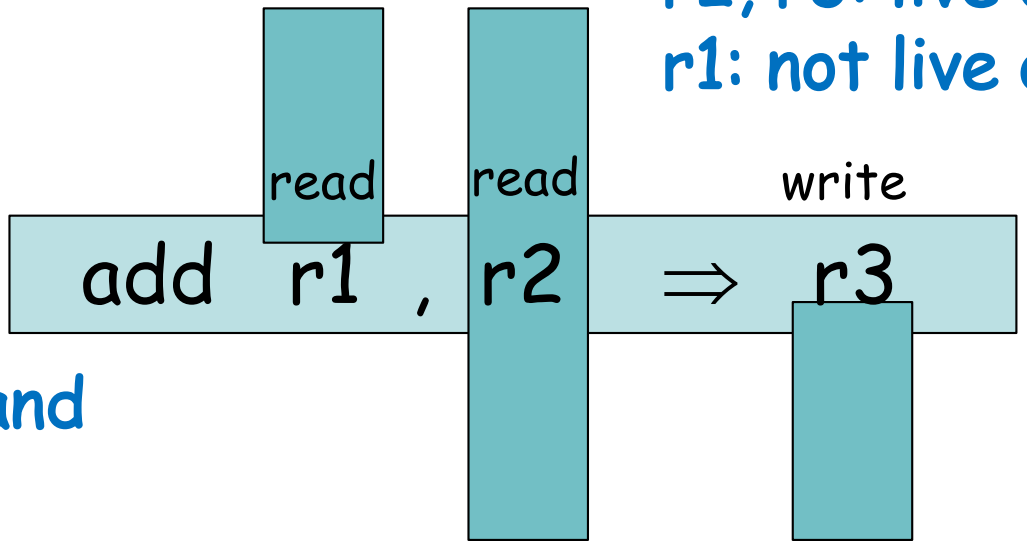


Graph coloring problem:
- Color all nodes

Live Ranges (live on exit)

r1 does not occur after instruction;
r2 occurs before and after instr.;
r3 occurs after instr.

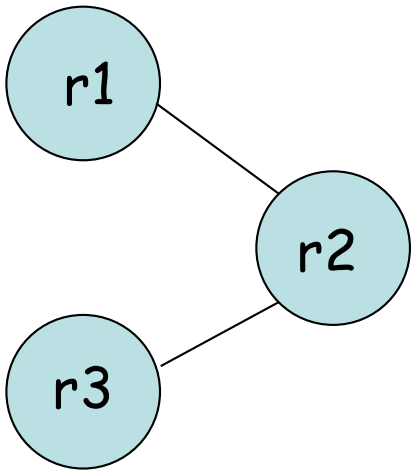
r2, r3: live on exit
r1: not live on exit



Live ranges of r1 and r3 do not overlap!

Register allocation as a graph coloring problem:

Interference Graph:
nodes: live ranges
edges: live at the same time

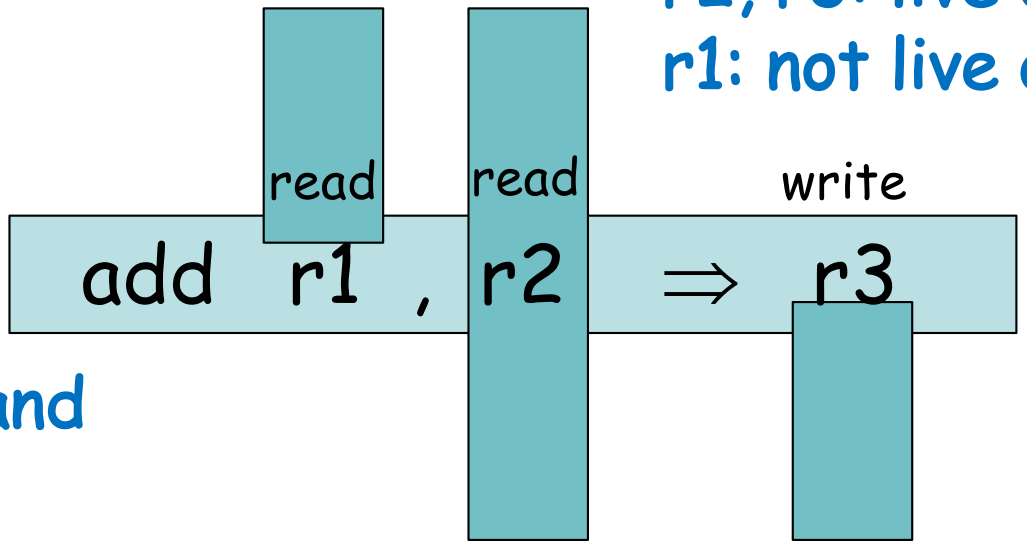


Graph coloring problem:
- Color all nodes
- Use minimal number of colors such that no adjacent nodes have the same color

Live Ranges (live on exit)

r1 does not occur after instruction;
r2 occurs before and after instr.;
r3 occurs after instr.

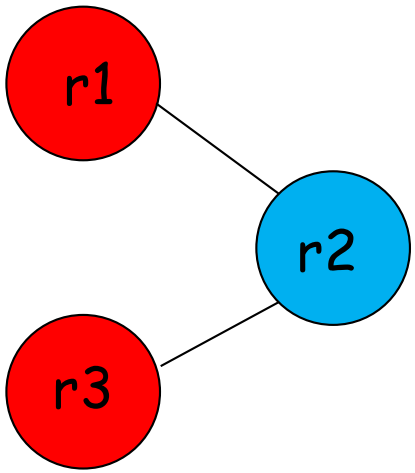
r2, r3: live on exit
r1: not live on exit



Live ranges of r1 and r3 do not overlap!

Register allocation as a graph coloring problem:

Interference Graph:
nodes: live ranges
edges: live at the same time



Graph coloring problem:
- Color all nodes
- Use minimal number of colors such that no adjacent nodes have the same color

ANSWER: Two colors

Top-down Allocator

The idea:

- Machine has k physical registers
- Keep "busiest" values in an assigned register
- Use the feasible (reserved) set, *F*, for the rest
- *F* is the minimal set of registers needed to execute any instruction with all operands in memory:
  - → Move values with no assigned register from/to memory by adding LOADs and STOREs (SPILL CODE)

Basic algorithm (**not graph coloring**!):

- Rank values by number of occurrences (or some other metric)
- Allocate first $k - F$ values to registers
- Rewrite code (with spill code) to reflect these choices

An Example : Top-Down

- Live Ranges

1	loadI	1028	⇒ r1
2	load	r1	⇒ r2
3	mult	r1, r2	⇒ r3
4	loadI	5	⇒ r4
5	sub	r4, r2	⇒ r5
6	loadI	8	⇒ r6
7	mult	r5, r6	⇒ r7
8	sub	r7, r3	⇒ r8
9	store	r8	⇒ r1

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

1	loadI	1028	⇒ r1	
2	load	r1	⇒ r2	
3	mult	r1, r2	⇒ r3	
4	loadI	5	⇒ r4	
5	sub	r4, r2	⇒ r5	
6	loadI	8	⇒ r6	
7	mult	r5, r6	⇒ r7	
8	sub	r7, r3	⇒ r8	
9	store	r8	⇒ r1	WAIT: r1 is reused as a target register?

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

```
1  loadI   1028     ⇒ r1   // r1
2  load    r1       ⇒ r2   // r1
3  mult    r1, r2   ⇒ r3   // r1
4  loadI   5        ⇒ r4   // r1
5  sub     r4, r2   ⇒ r5   // r1
6  loadI   8        ⇒ r6   // r1
7  mult    r5, r6   ⇒ r7   // r1
8  sub     r7, r3   ⇒ r8   // r1
9  store   r8       ⇒ r1   //
```

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

```
1  loadI   1028     ⇒ r1    // r1
2  load    r1       ⇒ r2    // r1 r2
3  mult    r1, r2   ⇒ r3    // r1 r2
4  loadI   5        ⇒ r4    // r1 r2
5  sub     r4, r2   ⇒ r5    // r1
6  loadI   8        ⇒ r6    // r1
7  mult    r5, r6   ⇒ r7    // r1
8  sub     r7, r3   ⇒ r8    // r1
9  store   r8       ⇒ r1    //
```

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

```
1  loadI   1028     ⇒ r1   // r1
2  load    r1       ⇒ r2   // r1 r2
3  mult    r1, r2   ⇒ r3   // r1 r2 r3
4  loadI   5        ⇒ r4   // r1 r2 r3
5  sub     r4, r2   ⇒ r5   // r1    r3
6  loadI   8        ⇒ r6   // r1    r3
7  mult    r5, r6   ⇒ r7   // r1    r3
8  sub     r7, r3   ⇒ r8   // r1
9  store   r8       ⇒ r1   //
```

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- Live Ranges

1	loadI	1028	⇒ r1	// *r1*					
2	load	r1	⇒ r2	// *r1 r2*					
3	mult	r1, r2	⇒ r3	// *r1 r2 r3*					
4	loadI	5	⇒ r4	// *r1 r2 r3 r4*					
5	sub	r4, r2	⇒ r5	// *r1*	*r3*	*r5*			
6	loadI	8	⇒ r6	// *r1*	*r3*	*r5 r6*			
7	mult	r5, r6	⇒ r7	// *r1*	*r3*		*r7*		
8	sub	r7, r3	⇒ r8	// *r1*				*r8*	
9	store	r8	⇒ r1	//					

NOTE: live ranges (on exit) of each instruction

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - $k = 4$, $F = 1$, $(k-F) = 3$

Note: ILOC needs larger F set -> homework

```
1  loadI   1028     ⇒ r1   // r1
2  load    r1       ⇒ r2   // r1 r2
3  mult    r1, r2   ⇒ r3   // r1 r2 r3
4  loadI   5        ⇒ r4   // r1 r2 r3 r4
5  sub     r4, r2   ⇒ r5   // r1    r3    r5
6  loadI   8        ⇒ r6   // r1    r3    r5 r6
7  mult    r5, r6   ⇒ r7   // r1    r3          r7
8  sub     r7, r3   ⇒ r8   // r1                      r8
9  store   r8       ⇒ r1   //
```

- Consider statements with MAXLIVE > (k-F) *basic algorithm*

Spill heuristic: - 1. number of occurrences of virtual register
- 2. length of live range (tie breaker)

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - *k* = 4, *F* = 1, (*k*–*F*) = 3

```
1  loadI   1028     ⇒ r1    // r1
2  load    r1       ⇒ r2    // r1 r2
3  mult    r1, r2   ⇒ r3    // r1 r2 r3
4  loadI   5        ⇒ r4    // r1 r2 r3 r4          -- MAXLIVE = 4
5  sub     r4, r2   ⇒ r5    // r1    r3    r5
6  loadI   8        ⇒ r6    // r1    r3    r5 r6  -- MAXLIVE = 4
7  mult    r5, r6   ⇒ r7    // r1    r3          r7
8  sub     r7, r3   ⇒ r8    // r1                   r8
9  store   r8       ⇒ r1    //
```

- Consider statements with MAXLIVE > (*k*-*F*) *basic algorithm*

  Spill heuristic: - 1. number of occurrences of virtual register
  - 2. length of live range (tie breaker)

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - $k = 4$, $F = 1$, $(k-F) = 3$

1	loadI	1028	⇒ r1	// *r1*				
2	load	r1	⇒ r2	// *r1 r2*				
3	mult	r1, r2	⇒ r3	// *r1 r2 r3*				
4	loadI	5	⇒ r4	// *r1 r2 r3 r4*				*-- MAXLIVE = 4*
5	sub	r4, r2	⇒ r5	// *r1*	*r3*	*r5*		
6	loadI	8	⇒ r6	// *r1*	*r3*	*r5 r6*		*-- MAXLIVE = 4*
7	mult	r5, r6	⇒ r7	// *r1*	*r3*		*r7*	
8	sub	r7, r3	⇒ r8	// *r1*			*r8*	
9	store	r8	⇒ r1	//				

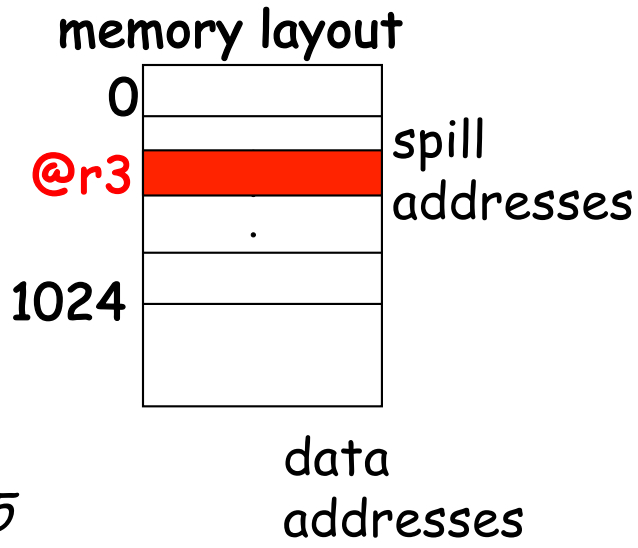
#occ.	length
r1: 4	8
r2: 3	3
r3: 2	5
r4: 2	1
r5: 2	2
r6,r7,r8: 2.	1

- Consider statements with MAXLIVE > $(k-F)$ *basic algorithm*

Spill heuristic: - 1. number of occurrences of virtual register
- 2. length of live range (tie breaker)

An Example : Top-Down

- 3 physical registers to allocate: ra, rb, rc
- 1 selected register: f1 (feasible set)
  - $k = 4$, $F = 1$, $(k-F) = 3$




```
1  loadI   1028      ⇒ r1    // r1
2  load    r1        ⇒ r2    // r1 r2
3  mult    r1, r2    ⇒ r3    // r1 r2 r3
4  loadI   5         ⇒ r4    // r1 r2 r3 r4
5  sub     r4, r2    ⇒ r5    // r1    r3    r5
6  loadI   8         ⇒ r6    // r1    r3    r5 r6
7  mult    r5, r6    ⇒ r7    // r1    r3          r7
8  sub     r7, r3    ⇒ r8    // r1                   r8
9  store   r8        ⇒ r1    //
```

- Consider statements with MAXLIVE > $(k-F)$ *basic algorithm*

Spill heuristic: - 1. number of occurrences of virtual register
- 2. length of live range (tie breaker)

Note: EAC Top down algorithm does not look at live ranges and MAXLIVE, but counts overall occurrences across entire basic block

An Example : Top-Down

- 3 physical registers for allocation: ra, rb, rc
- 1 physical register <u>designated</u> to be in the feasible set *F*

```
1  loadI      1028       ⇒ ra    // r1
2  load       ra         ⇒ rb    // r1 r2
3  mult       ra, rb     ⇒ f1    // r1 r2
   storeAI    f1 ⇒ r0, @r3       // spill code
4  loadI      5          ⇒ rc    // r1 r2    r4              -- MAXLIVE = 3
5  sub        rc, rb     ⇒ rb    // r1          r5
6  loadI      8          ⇒ rc    // r1          r5 r6.       -- MAXLIVE = 3
7  mult       rb, rc     ⇒ rb    // r1                  r7
   loadAI r0, @r3        ⇒ f1    // spill code
8  sub        rb, f1     ⇒ rb    // r1                       r8
9  store      rb         ⇒ ra    //
```

- Insert spill code for every occurrence of spilled virtual register in basic block using feasible register **f1**;
  Remove spilled register from consideration for allocation

Next topic

Bottom-up register allocation

Lexical Analysis

Read EaC: Chapters 2.1 – 2.5;