RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers
Register Allocation and Introduction to Instruction Scheduling

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

Announcement

- **First homework out**
  - → Due tomorrow 11:59pm EST.
  - → Sakai submission window open.
  - → pdf format only.
- **Late policy**
  - → 20% penalty for first 24 hours late.
  - → Not accepted 24 hours after deadline.

If you have personal overriding reasons that prevents you from submitting homework online, please let me know in advance.

Review: ILOC

Register allocation on basic blocks in "ILOC" (local register allocation)

- Pseudo-code for a simple, abstracted RISC machine
  - → generated by the instruction selection process
- Simple, compact data structures
- Here: we only use a small subset of ILOC
  - → ILOC simulator at ~zhang/cs415_2016/ILOC_Simulator on ilab cluster

Naïve Representation:

loadI	2		r1
loadAI	r0	@y	r2
add	r1	r2	r3
loadAI	r0	@x	r4
sub	r4	r3	r5

Quadruples:

- **table of *k x 4* small integers**
- **simple record structure**
- **easy to reorder**
- **all names are explicit**

ILOC is described in Appendix A of EAC

Review: F – Set of Feasible Registers

Allocator may need to reserve registers to ensure feasibility

- Must be able to compute addresses
- Requires some minimal set of registers, *F*
  - → *F* depends on target architecture
- F contains registers to make spilling work
  (set F registers "aside", i.e., not available for register assignment)

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

Live Range of a Value (Virtual Register)

A **value** 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 does a second definition affect this?
- 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 (pseudo-registers) live at *i*.

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

Top-down Versus Bottom-up Allocation

Top-down allocator

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

Top-down Allocator

The idea:

- Keep busiest values in a register
- Use the feasible (reserved) set, *F*, for the rest

Algorithm (using a heuristic):

- Rank values by number of occurrences

  may or may not use explicit live ranges
- Allocate *the first n* values to k – F registers
- Rewrite code to reflect these choices

SPILL: Move values with no register into memory

(add LOADs & STOREs)

An Example

- Here is a sample code sequence

```
loadI   1028    ⇒ r1    // r1 ← 1028
load    r1      ⇒ r2    // r2 ← MEM(r1) == y
mult    r1, r2  ⇒ r3    // r3 ← 1028 · y
loadI   5       ⇒ r4    // r4 ← 5
sub     r4, r2  ⇒ r5    // r5 ← 5 - y
loadI   8       ⇒ r6    // r6 ← 8
mult    r5, r6  ⇒ r7    // r7 ← 8 · (5 - y)
sub     r7, r3  ⇒ r8    // r8 ← 8 · (5 - y) - (1028 · y)
store   r8      ⇒ r1    // MEM(r1) ← 8 · (5 - y) - (1028 · y)
```

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 sets on exit of each instruction

An Example : Top-Down

- Top down (3 physical registers: ra, rb, rc)

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	//					

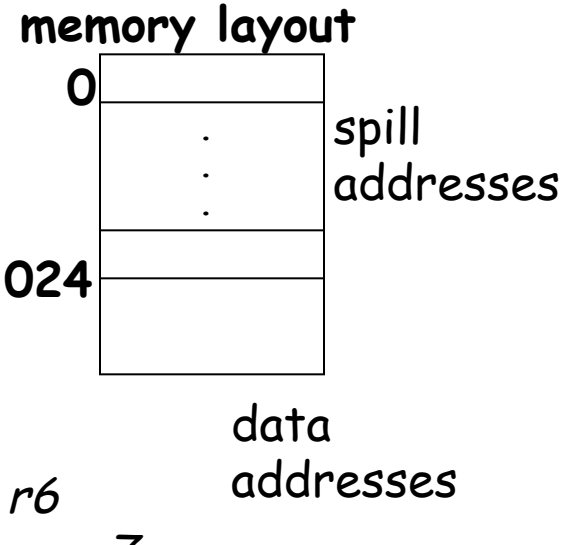
- Consider statements with MAXLIVE > (*k-F*)

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

An Example : Top-Down

- Top down (3 physical registers: ra, rb, rc)

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)

  Spill heuristic: - number of occurrences of virtual register
  - 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

- Top down (3 physical registers: ra, rb, rc)

1	loadI	1028	⇒ ra	// *r1*					
2	load	ra	⇒ rb	// *r1 r2*					
3	mult	ra, rb	⇒ f1	// *r1 r2 r3*					
	store*	f1	⇒ 10	// spill code					
4	loadI	5	⇒ rc	// *r1 r2 r3 r4*					
5	sub	rc, rb	⇒ rb	// *r1*	*r3*	*r5*			
6	loadI	8	⇒ rc	// *r1*	*r3*	*r5 r6*			
7	mult	rb, rc	⇒ rb	// *r1*	*r3*		*r7*		
	load*	10	⇒ 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

  Note that this assumes that an extra register is not needed for save/restore

Spill code

- A virtual register is spilled by using only registers from the feasible set (F), not the allocated set (k-F)
- How to insert spill code, with F = {f1, f2, … }?
  - → For the definition of the spilled value (assignment of the value to the virtual register), use a feasible register as the target register and then use an additional register to load its address in memory, and perform the store:

```
add r1, r2  ⇒ f1
loadI @f    ⇒ f2  // value lives at memory location @f
store f1    ⇒ f2
```

  - → For the use of the spilled value, load value from memory into a feasible register:

```
loadI @f    ⇒ f1
load f1     ⇒ f1
add f1, r2  ⇒ r1
```

- How many feasible registers do we need for an *add* instruction?

Bottom-up Allocator

The idea:

- Focus on replacement rather than allocation
- Keep values "used soon" in registers
- Only parts of a live range may be assigned to a physical register (≠ top-down allocation's "all-or-nothing" approach)

Algorithm:

- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement (heuristic):

- Spill the value whose next use is farthest in the future

Our Example

- Here is the sample code sequence

```
loadI   1028    ⇒ r1    // r1 ← 1028
load    r1      ⇒ r2    // r2 ← MEM(r1) == y
mult    r1, r2  ⇒ r3    // r3 ← 1028 · y
loadI   5       ⇒ r4    // r4 ← 5
sub     r4, r2  ⇒ r5    // r5 ← 5 - y
loadI   8       ⇒ r6    // r6 ← 8
mult    r5, r6  ⇒ r7    // r7 ← 8 · (5 - y)
sub     r7, r3  ⇒ r8    // r5 ← 8 · (5 - y) - (1028 · y)
store   r8      ⇒ r1    // MEM(r1) ← 8 · (5 - y) - (1028 · y)
```

Our Example: Live Ranges

- Live Ranges

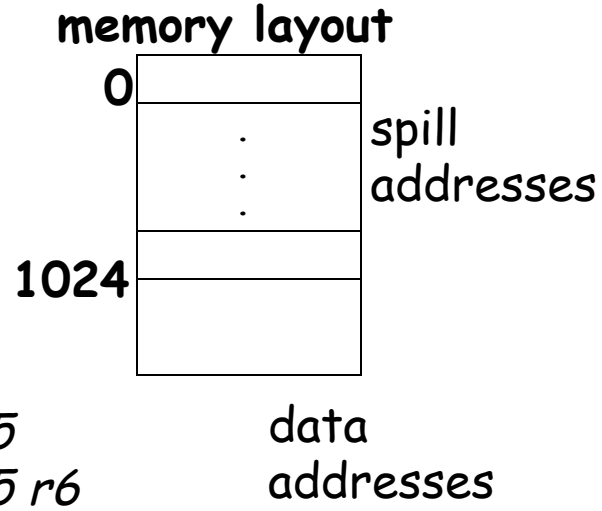
1	loadI	1028	⇒ r1	// *r1*					
2	load	r1	⇒ r2	// *r1 r2*					
3	mult	r1, r2	⇒ r3	// *r1 r2 r3*					
4	loadI	5	⇒ r4	// *r1 r2 r3 r4*					
5	sub	r4, r2	⇒ r5	// *r1*	*r3*	*r5*			
6	loadI	8	⇒ r6	// *r1*	*r3*	*r5 r6*			
7	mult	r5, r6	⇒ r7	// *r1*	*r3*		*r7*		
8	sub	r7, r3	⇒ r8	// *r1*				*r8*	
9	store	r8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

NOTE: live sets on exit of each instruction

An Example : Bottom-up

- Bottom up (3 registers)

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 that this assumes that extra registers are not needed for save/ restore

An Example : Bottom-up

- Bottom up (3 physical registers: ra, rb, rc)

	source code			life ranges	register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ r1	// *r1*	*r1*		
2	load	r1	⇒ r2	// *r1 r2*	*r1*	*r2*	
3	mult	r1, r2	⇒ r3	// *r1 r2 r3*	*r1*	*r2*	*r3*
4	loadI	5	⇒ r4	// *r1 r2 r3 r4*	*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5	// *r1 r5 r3*	*r4*	*r5*	*r3*
6	loadI	8	⇒ r6	// *r1 r5 r3 r6*	*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7	// *r1 r7 r3*	*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8	// *r1 r8*	*r6*	*r8*	*r3*
9	store	r8	⇒ r1	//	r1	r8	r3

Note: this is only one possible allocation and assignment!

An Example : Bottom-up

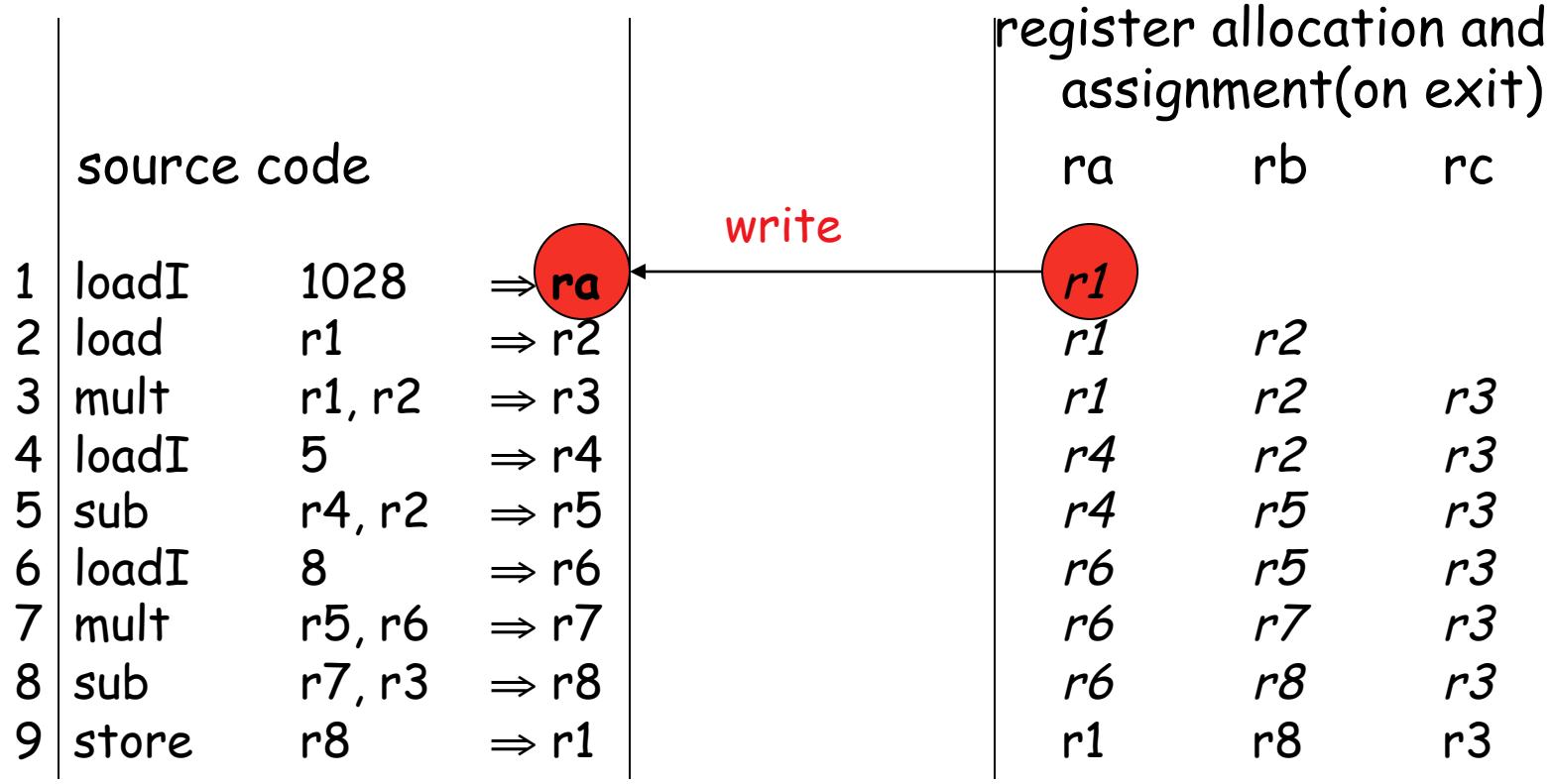
- Bottom up (3 physical registers: ra, rb, rc)

	source code			register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ r1	*r1*		
2	load	r1	⇒ r2	*r1*	*r2*	
3	mult	r1, r2	⇒ r3	*r1*	*r2*	*r3*
4	loadI	5	⇒ r4	*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5	*r4*	*r5*	*r3*
6	loadI	8	⇒ r6	*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7	*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8	*r6*	*r8*	*r3*
9	store	r8	⇒ r1	r1	r8	r3

Let's generate code now!

An Example : Bottom-up

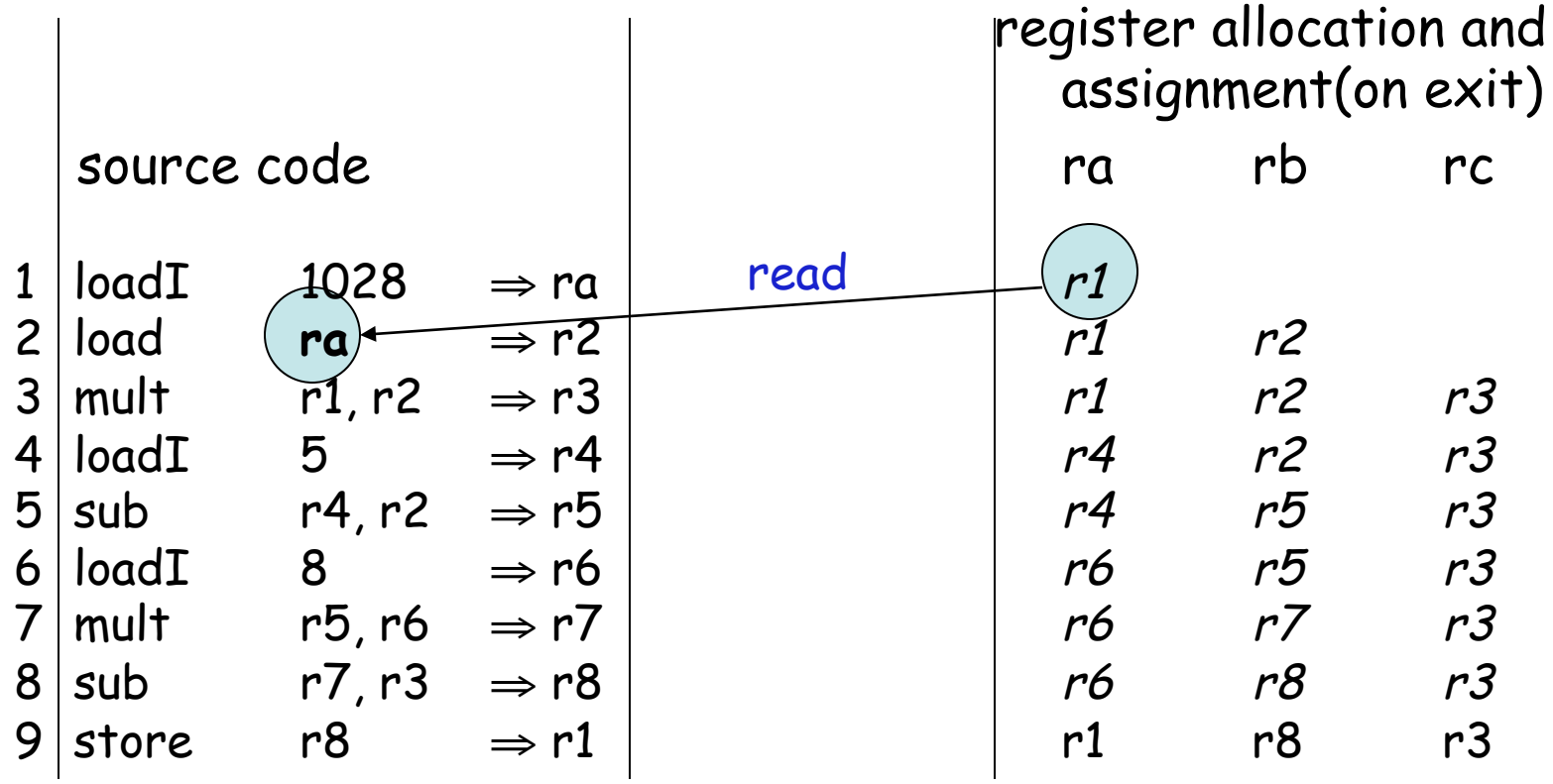
- Bottom up (3 physical registers: ra, rb, rc)



For written registers, use current register assignment.

An Example : Bottom-up

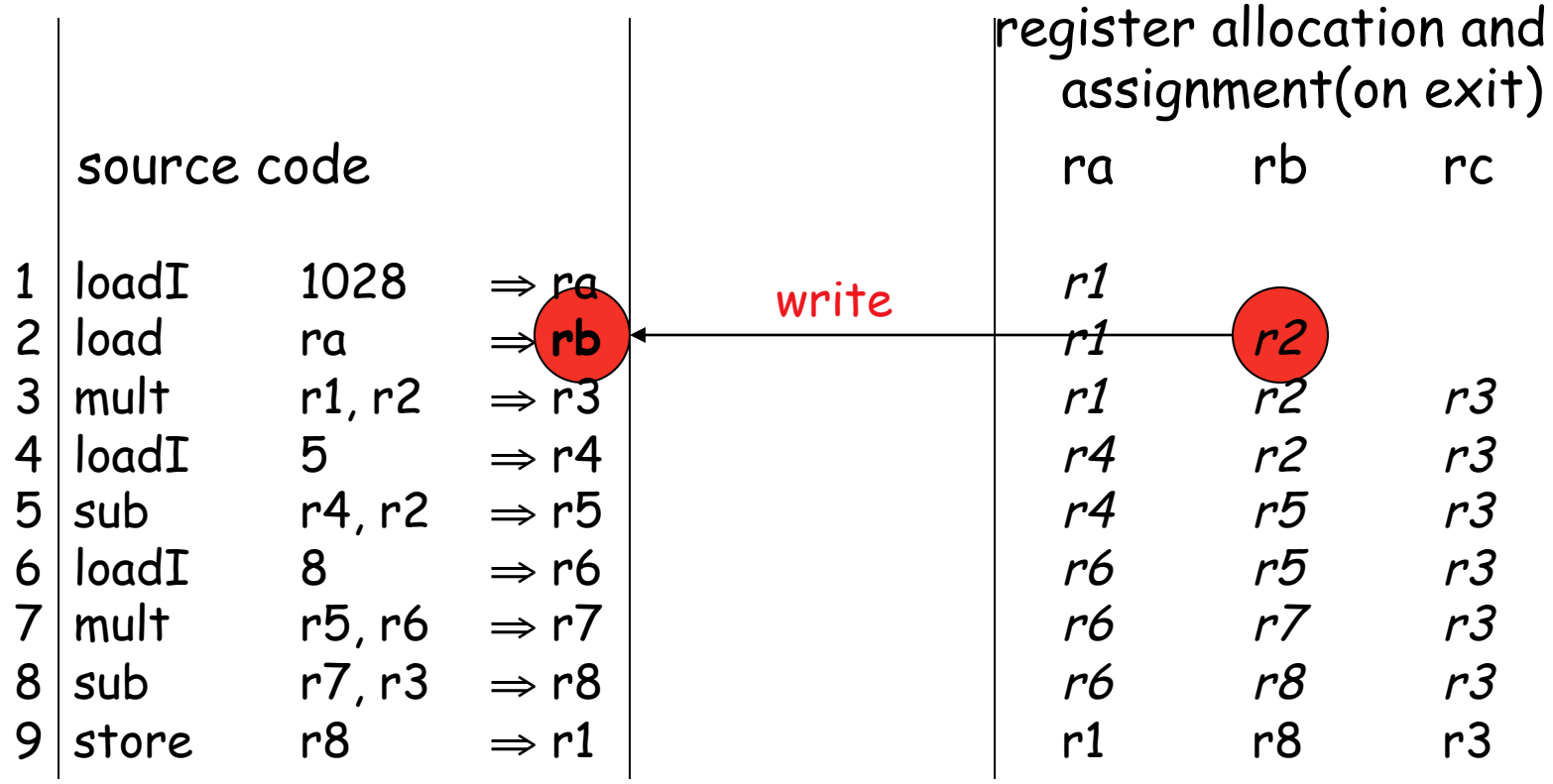
- Bottom up (3 physical registers: ra, rb, rc)



For read registers, use previous register assignment.

An Example : Bottom-up

- Bottom up (3 physical registers: ra, rb, rc)



An Example : Bottom-up

- Bottom up (3 physical registers: ra, rb, rc)

	source code			register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra	*r1*		
2	load	ra	⇒ rb	*r1*	*r2*	
3	mult	ra, rb	⇒ rc	*r1*	*r2*	*r3*
4	loadI	5	⇒ r4	*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5	*r4*	*r5*	*r3*
6	loadI	8	⇒ r6	*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7	*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8	*r6*	*r8*	*r3*
9	store	r8	⇒ r1	r1	r8	r3

Insert spill code.

An Example : Bottom-up

- Bottom up (3 physical registers: ra, rb, rc)

	source code				register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
	store*	ra	⇒ 10	spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ r4		*r4*	*r2*	*r3*
5	sub	r4, r2	⇒ r5		*r4*	*r5*	*r3*
6	loadI	8	⇒ r6		*r6*	*r5*	*r3*
7	mult	r5, r6	⇒ r7		*r6*	*r7*	*r3*
8	sub	r7, r3	⇒ r8		*r6*	*r8*	*r3*
9	store	r8	⇒ r1		r1	r8	r3

Insert spill code.

An Example : Bottom-up

- Bottom up (3 physical registers: ra, rb, rc)

	source code				register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
	store	ra	⇒ 10	spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ ra		*r4*	*r2*	*r3*
5	sub	ra, rb	⇒ rb		*r4*	*r5*	*r3*
6	loadI	8	⇒ ra		*r6*	*r5*	*r3*
7	mult	rb, ra	⇒ rb		*r6*	*r7*	*r3*
8	sub	rb, rc	⇒ rb		*r6*	*r8*	*r3*
9	store	r8	⇒ r1		r1	r8	r3

An Example : Bottom-up

- Bottom up (3 physical registers: ra, rb, rc)

	source code				register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
	store*	ra	⇒ 10	spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ ra		*r4*	*r2*	*r3*
5	sub	ra, rb	⇒ rb		*r4*	*r5*	*r3*
6	loadI	8	⇒ ra		*r6*	*r5*	*r3*
7	mult	rb, ra	⇒ rb		*r6*	*r7*	*r3*
8	sub	rb, rc	⇒ rb		*r6*	*r8*	*r3*
	load*	10	⇒ ra	spill code	*r1*	*r8*	*r3*
9	store	r8	⇒ r1	*NOT ILOC*	r1	r8	r3

Insert spill code.

An Example : Bottom-up

- Bottom up (3 physical registers: ra, rb, rc)

	source code				register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
	store*	ra	⇒ 10	spill code	*r1*	*r2*	*r3*
4	loadI	5	⇒ ra		*r4*	*r2*	*r3*
5	sub	ra, rb	⇒ rb		*r4*	*r5*	*r3*
6	loadI	8	⇒ ra		*r6*	*r5*	*r3*
7	mult	rb, ra	⇒ rb		*r6*	*r7*	*r3*
8	sub	rb, rc	⇒ rb		*r6*	*r8*	*r3*
	load*	10	⇒ ra	spill code	*r1*	*r8*	*r3*
9	store	rb	⇒ ra		r1	r8	r3

Done.

Spilling revisited

- Bottom up (3 physical registers: ra, rb, rc)

	source code				register allocation and assignment(on exit) ra	rb	rc
1	loadI	1028	⇒ ra		*r1*		
2	load	ra	⇒ rb		*r1*	*r2*	
3	mult	ra, rb	⇒ rc		*r1*	*r2*	*r3*
				no spill code			
4	loadI	5	⇒ ra		*r4*	*r2*	*r3*
5	sub	ra, rb	⇒ rb		*r4*	*r5*	*r3*
6	loadI	8	⇒ ra		*r6*	*r5*	*r3*
7	mult	rb, ra	⇒ rb		*r6*	*r7*	*r3*
8	sub	rb, rc	⇒ rb		*r6*	*r8*	*r3*
	loadI	1028	⇒ ra	spill code	*r1*	*r8*	*r3*
9	store	rb	⇒ ra		r1	r8	r3

Rematerialization: Re-computation is cheaper than store/load to memory

Introduction to Instruction Scheduling

Data Dependences (stmt./instr. level)

Dependences ⇒ defined on memory locations / registers and not values

Statement/instruction b depends on statement/instruction a if there exists:

- true of flow dependence
  a writes a location/register that b later reads (RAW conflict)

- anti dependence
  a reads a location/register that b later writes (WAR conflict)

- output dependence
  a writes a location/register that b later writes (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

true	anti	output
a =	= a	a =
= a	a =	a =

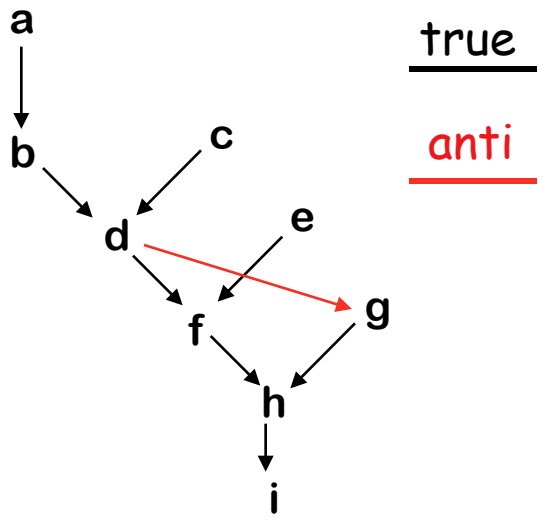
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a precedence graph *G*

- Nodes $n \in G$ are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1, n_2) \in G$ if & only if n_2 uses the result of n_1

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Precedence Graph

(all output dependences are covered, i.e., are satisfied through other dependences)

Next class

Instruction Scheduling and Lexical Analysis

Read EaC: Chapter 12

Read EaC: Chapters 2.1 – 2.5