



RICE

COMP 412
FALL 2009

Local Register Allocation & Lab 1

COMP 412

Extra slides at the end
on live ranges may prove
helpful.

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.



Where are we (and why)?

Local Register Allocation

- Chapter 13 in EAC covers Register Allocation
 - Read Sections 13.1, 13.2, & 13.3
 - Look at questions 1 and 2 in the exercises for Chapter 13
- Read Chapter 1 (*for context*) , look at Appendix A (*for ILOC*)
- Lab specs are on the class website

Why are we here?

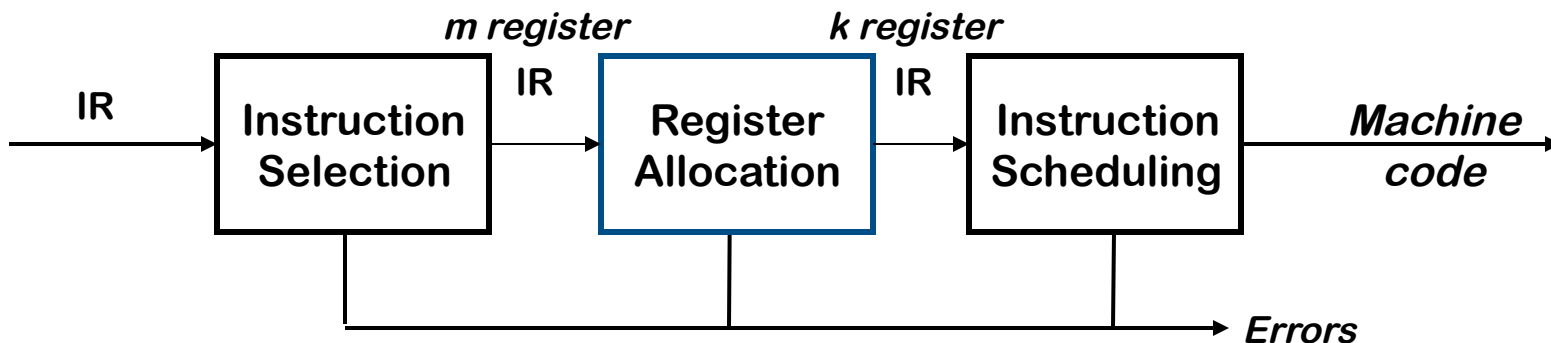
- Making time for scanning and parsing
- Providing implicit motivation for the start of the course
- And, allocation is both challenging and fun ...

Now, back to the lecture



Register Allocation

Part of the compiler's back end



Critical properties

- Produce correct code that uses no more than k registers
- Minimize added work from loads and stores that *spill* values
- Minimize space used to hold *spilled values*
- Operate efficiently
 $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

Register Allocation

r0 holds base address for local variables

@x is constant offset of x from r0



Consider a fragment of assembly code (or ILOC)

```
loadI    2      ⇒ r1    // r1 ← 2
loadAI   r0, @b ⇒ r2    // r2 ← b
mult     r1, r2 ⇒ r3    // r3 ← 2 · b
loadAI   r0, @a ⇒ r4    // r4 ← a
sub      r4, r3 ⇒ r5    // r5 ← a - (2 · b)
```

From the allocation perspective, these registers are virtual or pseudo-registers

The Problem

- At each instruction, decide which *values* to keep in registers
 - Note: each pseudo-register in the example is a value
- Simple if $|values| \leq |registers|$
- Harder if $|values| > |registers|$
- The compiler must automate this process



Register Allocation

The Task

- At each point in the code, pick the values to keep in registers
- Insert code to move values between registers & memory
 - No transformations (leave that to optimization & scheduling)
- Minimize inserted code — both dynamic & static measures
- Make good use of any *extra* registers

For straight-line code,
dynamic \cong static

Allocation versus assignment

- *Allocation* is deciding which values to keep in registers
- *Assignment* is choosing specific registers for values
- This distinction is often lost in the literature

The compiler must perform both allocation & assignment

and your lab



Basic Blocks in Assembly Code (or ILOC)

Definition

- A *basic block* is a maximal length segment of straight-line (i.e., branch free) code

Importance

(assuming normal execution)

- Strongest facts are provable for branch-free code
- If any statement executes, they all execute
- Execution is totally ordered

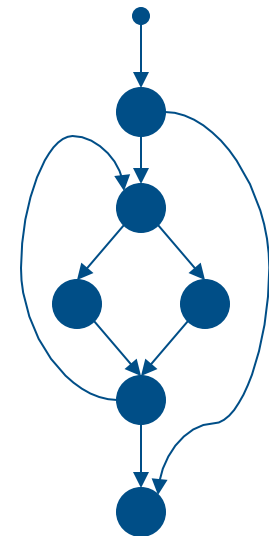
Optimization

- Many techniques for improving basic blocks
- Simplest problems
- Strongest methods

Local Register Allocation



- What is “local” ? (different from “regional” or “global”)
 - A local transformation operates on basic blocks
 - Many optimizations are done on a local scale or scope
- Does local allocation solve the problem?
 - It produces good register use inside a block
 - Inefficiencies can arise at boundaries between blocks
 - Your lab assumes that the block is the entire program
 - Assumption significantly simplifies allocation
- How many passes can the allocator make?
 - This is an off-line problem
 - As many passes as it takes, within reason
 - *You can do a fine job in a couple of passes*



Blocks in a Control-flow Graph (CFG)

ILOC



Your lab will do register allocation on basic blocks in "ILOC"

- Pseudo-code for a simple, abstracted RISC machine
 - generated by the instruction selection process
- Simple, compact data structures
- You will use a tiny subset of ILOC

| | | | |
|------------------------|---------------------|---|-------------------|
| <code>a - 2 x b</code> | <code>loadI</code> | <code>2</code> | $\Rightarrow r_1$ |
| | <code>loadAI</code> | <code>r₀, @b</code> | $\Rightarrow r_2$ |
| | <code>add</code> | <code>r₁, r₂</code> | $\Rightarrow r_3$ |
| | <code>loadAI</code> | <code>r₀, @a</code> | $\Rightarrow r_4$ |
| | <code>sub</code> | <code>r₄, r₃</code> | $\Rightarrow r_5$ |

ILOC:

- simple three-address code
- RISC-like addressing modes
 - I, AI, AO
- unlimited virtual registers

ILOC is described in Appendix A of EAC

The subset used in lab 1 & 3 is described in the lab handout



ILOC

Your lab will do register allocation on basic blocks in "ILOC"

- Pseudo-code for a simple, abstracted RISC machine
 - generated by the instruction selection process
- Simple, compact data structures
- You will use a tiny subset of ILOC

$a - 2 \times b$

| | | | |
|--------|-------|-------|-------|
| loadI | 2 | | r_1 |
| loadAI | r_0 | @b | r_2 |
| add | r_1 | r_2 | r_3 |
| loadAI | r_0 | @a | r_4 |
| sub | r_4 | r_3 | r_5 |

Quadruples:

- table of $k \times 4$ small integers
- simple record structure
- easy to reorder
- all names are explicit

ILOC is described in Appendix A of EAC

The subset used in lab 1 & 3 is described in the lab handout



Register Allocation

Optimal register allocation is hard

Local Allocation

- Simplified cases $\Rightarrow O(n)$
- Real cases \Rightarrow NP-Complete

Global Allocation

- NP-Complete for 1 register
- NP-Complete for k registers
(most sub-problems are NPC, too)

Real compilers face real problems

Local Assignment

- Single size, no spilling $\Rightarrow O(n)$
- Two sizes \Rightarrow NP-Complete

Global Assignment

- NP-Complete

Recent Results:

Optimal allocation on a procedure in SSA Form can be done in low-order polynomial time.

This result does not solve the entire problem, but it does offer insight into the structure of the problem & where the complexity lies.

We will come back to this issue later.



Observations

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
- Use these registers only for spilling

Notation:

k is the number of registers on the target machine

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

- Remember that the underlying problem is NP-Complete
- The allocator can either
 - Check for this situation
 - Adopt a more complex strategy *(iterate?)*
 - Accept the fact that the technique is an approximation



Observations

A value is *live* between its *definition* and its *uses*

- Find definitions ($x \leftarrow \dots$) and uses ($y \leftarrow \dots x \dots$)
- 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)

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

- If $\text{MAXLIVE} \leq k$, allocation should be easy
- If $\text{MAXLIVE} \leq k$, no need to reserve F registers for spilling
- If $\text{MAXLIVE} > k$, some values must be spilled to memory
- If $\text{MAXLIVE} > k$, need to reserve F registers for spilling

Finding live ranges is harder in the global case



An Example

- Here is a sample code sequence

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

Example is mildly confusing. It uses 1028 as both the address of y and as a constant in the computation.

The intent is to create a long live range for pedagogical purposes.



An Example

➤ Live ranges in the example

| | | | | | | | |
|-------|--------|------|----------------|--|--|--|--|
| loadI | 1028 | ⇒ r1 | // r1 | | | | |
| load | r1 | ⇒ r2 | // r1 r2 | | | | |
| mult | r1, r2 | ⇒ r3 | // r1 r2 r3 | | | | |
| load | x | ⇒ r4 | // r1 r2 r3 r4 | | | | |
| sub | r4, r2 | ⇒ r5 | // r1 r3 r5 | | | | |
| load | z | ⇒ r6 | // r1 r3 r5 r6 | | | | |
| mult | r5, r6 | ⇒ r7 | // r1 r3 r7 | | | | |
| sub | r7, r3 | ⇒ r8 | // r1 r8 | | | | |
| store | r8 | ⇒ r1 | // | | | | |

A pseudo-register is live if it has been defined & has a use in the future



An Example

➤ Live Ranges

```
loadI 1028 => r1 // r1
load  r1   => r2 // r1 r2
mult  r1, r2 => r3 // r1 r2 r3
load  x   => r4 // r1 r2 r3 r4
sub   r4, r2 => r5 // r1 r3 r5
load  z   => r6 // r1 r3 r5 r6
mult  r5, r6 => r7 // r1 r3 r7
sub   r7, r3 => r8 // r1 r8
store r8   => r1 //
```

MAXLIVE is 4

r1 r2 r3 r4

r1 r3 r5 r6

Register that holds the address in a store is an operand, not a target ...

Compute these "live" sets in a backward pass over the code. Start with live as the empty set. At each op, remove target & add operands



Top-down Versus Bottom-up Allocation

Top-down allocator

- Work from external notion of what is important
- Assign registers in priority order
- Save some registers for the values relegated to memory

Bottom-up allocator

- Work from detailed knowledge about problem instance
- Incorporate knowledge of partial solution at each step
- Handle all values uniformly

You will implement one of each

This terminology is nonstandard; that is okay. We are allowed to invent terms as long as they are descriptive.



Top-down Allocator

The idea:

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

Algorithm:

- Rank values by number of occurrences
- Allocate first $k - F$ values to registers
- Rewrite code to reflect these choices

Move values with no register into memory
(add LOADs & STOREs)

Seem familiar?

- C's register declaration
- Common technique of 60's and 70's

You will implement a variant (see Chapter 13, Question 1)



An Example

- Top down (3 registers; reserve 2 for operands)

| | | | | |
|-------|--------|------|--------------------|-------------------------------|
| loadI | 1028 | ⇒ r1 | // r1 | r1 is used more often than r3 |
| load | r1 | ⇒ r2 | // r1 r2 | |
| mult | r1, r2 | ⇒ r3 | // <u>r1 r2 r3</u> | spill r3 |
| load | x | ⇒ r4 | // r1 r2 r3 r4 | |
| sub | r4, r2 | ⇒ r5 | // r1 r3 r5 | |
| load | z | ⇒ r6 | // r1 r3 r5 r6 | |
| mult | r5, r6 | ⇒ r7 | // <u>r1 r3</u> r7 | |
| sub | r7, r3 | ⇒ r8 | // r1 r8 | restore r3 |
| store | r8 | ⇒ r1 | // | |

Note that this assumes that no extra register is needed for spilling (Absolute location for each register — use load and store immediate)



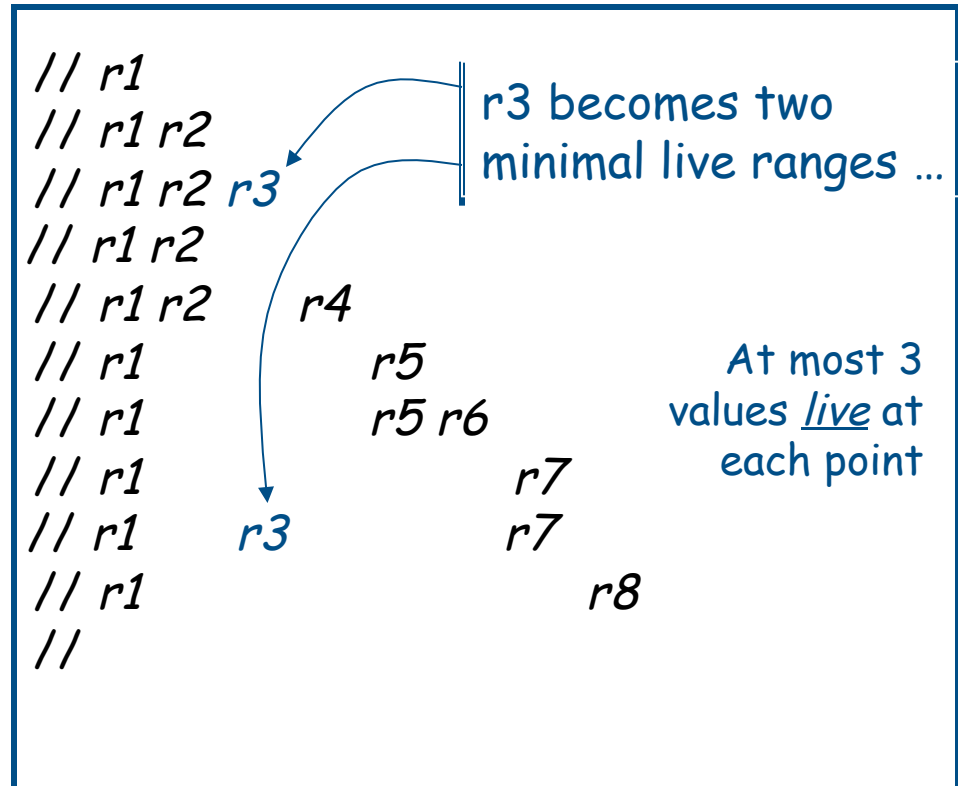
An Example

- Top down (3 registers; reserve 2 for operands)

```

loadI    1028    => r1
load     r1      => r2
mult     r1, r2  => r3
spill    r3      => 16
load     x       => r4
sub      r4, r2  => r5
load     z       => r6
mult     r5, r6  => r7
restore  16      => r3
sub      r7, r3  => r8
store    r8      => r1

```



The two short versions of r3 each overlap with fewer values, which simplifies the allocation problem. Such "spilling" will (eventually) create a code where the allocator can succeed.



An Example

- Top down (3 registers; reserve 2 for operands)

```
loadI 1028 => r1 // r1
load  r1   => r2 // r1 r2
mult  r1, r2 => r3 // r1 r2 r3
spill r3   => 16 // r1 r2
loadI x    => r4 // r1 r2 r4
sub   r4, r2 => r5 // r1 r5
loadI z    => r6 // r1 r5 r6
mult  r5, r6 => r7 // r1 r7
restore 16 => r3 // r1 r3 r7 possible delay
sub   r7, r3 => r8 // r1 r8
store r8   => r1 //
```

2 more ops

This code is slower than the original, but it works correctly on a target machine with only three (available) registers. Correctness is a virtue.



Bottom-up Allocator

The idea:

- Focus on replacement rather than allocation
- Keep values used "soon" in registers

Algorithm:

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

Replacement:

- Spill the value whose **next use is farthest in the future**
- Prefer clean value to dirty value
- Sound familiar? Think page replacement ...

Bottom-up Allocator



The algorithm should sound familiar

Decade algorithm

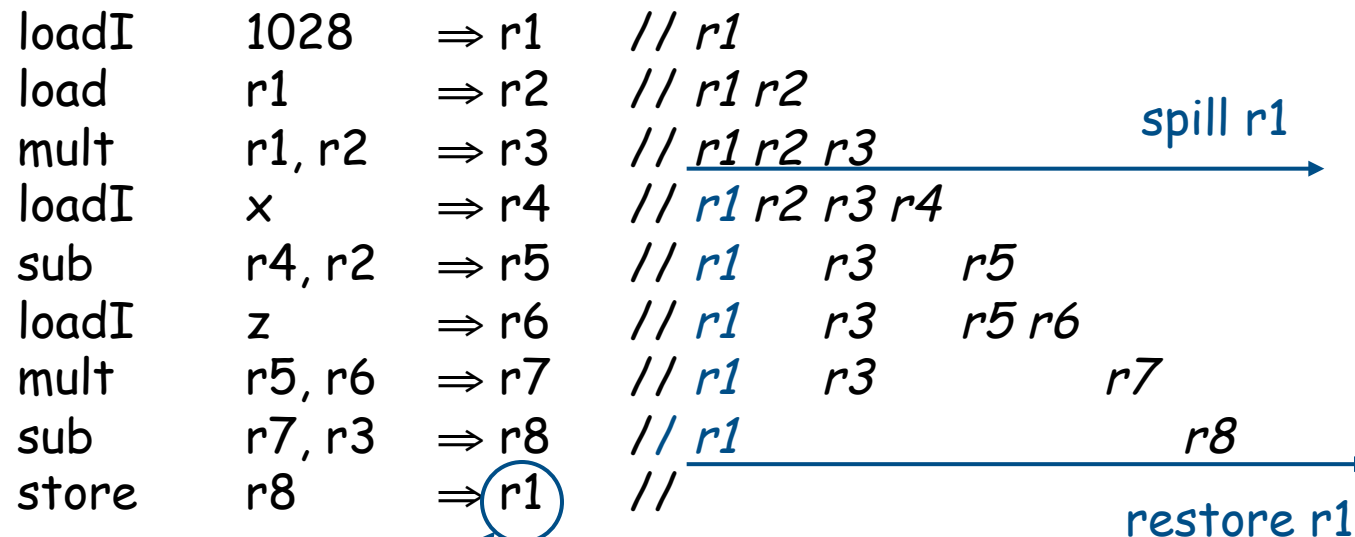
- Sheldon Best, 1955, for Fortran I
 - Laslo Belady, 1965, for paging studies
 - William Harrison, 1975, in ECS compiler work
 - Chris Fraser, 1989, in the LCC compiler
 - Forgotten student, 1995, COMP 412 at Rice
 - Vincenzo Liberatore, 1997, Rutgers
-
- It will be reinvented again
 - Many authors have argued for its optimality

(wrong)



An Example

- Bottom up (3 registers ; reserve 2 for operands)



N.B.: This slot is a use, not a definition

Note that this assumes that no extra register is needed for spilling (Absolute location for each register — use load and store immediate)



An Example

- Bottom up (3 registers ; reserve 2 for operands)

| | | | | | | | |
|---------|--------|------|-------------|--|--|--|--------------------|
| loadI | 1028 | ⇒ r1 | // r1 | | | | |
| load | r1 | ⇒ r2 | // r1 r2 | | | | |
| mult | r1, r2 | ⇒ r3 | // r1 r2 r3 | | | | |
| spill | r1 | ⇒ 20 | // r2 r3 | | | | |
| loadI | x | ⇒ r4 | // r2 r3 r4 | | | | |
| sub | r4, r2 | ⇒ r5 | // r3 r5 | | | | At most 3 |
| loadI | z | ⇒ r6 | // r3 r5 r6 | | | | values <i>live</i> |
| mult | r5, r6 | ⇒ r7 | // r3 r7 | | | | at each |
| sub | r7, r3 | ⇒ r8 | // r8 | | | | point |
| restore | 20 | ⇒ r1 | // r1 | | | | |
| store | r8 | ⇒ r1 | // r8 | | | | |

The two short versions of r1 each overlap with fewer values, which simplifies the allocation problem. Such "spilling" will (eventually) create a code where the allocator can succeed.

Lab One



The task:

- Implement a version of the top-down allocator
(See EaC Section 13.3.1 & Exercise 2a, Section 13.3)
- Implement a version of the bottom-up allocator
(See EaC Section 13.3.2)
- Run them on a collection of test blocks
- Write up a report
 - Describe the experience
 - Compare the two allocators

Due date: Wednesday, September 16, 2007, 11:59 PM

Documentation: Friday, September 18, 2007, 11:59 PM

Test & report blocks are available on the web site

Live Ranges

(a somewhat subtle point)



Your allocator is not bound by the names used in its input

- Every computed value is part of some live range
 - Even if it has no name in the source code (e.g., $2 * y$ in $x - 2 * y$)
- A live range usually has a single name, such as r_{17}
- A single name with multiple values can be renamed into distinct live ranges



Live Ranges

(From Figure 13.3 in EaC)

| Operation | | | Live Ranges |
|-----------|---------------|---------------------------|------------------------------------|
| loadI | @base | $\Rightarrow r_{arp}$ | <i>none</i> |
| loadAI | $r_{arp}, @a$ | $\Rightarrow r_a$ | r_{arp}, r_a |
| loadI | 2 | $\Rightarrow r_2$ | r_{arp}, r_a, r_2 |
| loadAI | $r_{arp}, @b$ | $\Rightarrow r_b$ | r_{arp}, r_a, r_2 |
| loadAI | $r_{arp}, @c$ | $\Rightarrow r_c$ | r_{arp}, r_a, r_b, r_2 |
| loadAI | $r_{arp}, @d$ | $\Rightarrow r_d$ | $r_{arp}, r_a, r_c, r_b, r_2$ |
| mult | r_a, r_2 | $\Rightarrow r_a$ | $r_{arp}, r_a, r_d, r_c, r_b, r_2$ |
| mult | r_a, r_b | $\Rightarrow r_a$ | $r_{arp}, r_a, r_d, r_c, r_b$ |
| mult | r_a, r_c | $\Rightarrow r_a$ | r_{arp}, r_a, r_d, r_c |
| mult | r_a, r_d | $\Rightarrow r_a$ | r_{arp}, r_a, r_d |
| storeAI | r_a | $\Rightarrow r_{arp}, @w$ | |

There are five distinct values, or live ranges, named r_a

The last 4 fit in F .

Live Ranges

(a somewhat subtle point)



Your allocator is not bound by the names used in its input

- Every computed value is part of some live range
 - Even if it has no name in the source code (e.g., $2 * y$ in $x - 2 * y$)
- A live range usually has a single name, such as r_{17}
- A single name with multiple values can be renamed into distinct live ranges

- Renaming distinct live ranges with distinct names can simplify the implementation of the allocator
 - It can also help with debugging the allocator

Next class: Finish the View from 35,000 Feet.

Lab One



ILOC subset:

| <i>Operation</i> | <i>Meaning</i> | <i>Latency</i> |
|--------------------------------|------------------------------|----------------|
| load r1 \Rightarrow r2 | MEM(r1) \rightarrow r2 | 2 |
| store r1 \Rightarrow r2 | r1 \rightarrow MEM(r2) | 2 |
| loadI x \Rightarrow r1 | x \rightarrow r1 | 1 |
| add r1, r2 \Rightarrow r3 | r1 + r2 \rightarrow r3 | 1 |
| sub r1, r2 \Rightarrow r3 | r1 - r2 \rightarrow r3 | 1 |
| mult r1, r2 \Rightarrow r3 | r1 x r2 \rightarrow r3 | 1 |
| lshift r1, r2 \Rightarrow r3 | r1 \ll r2 \rightarrow r3 | 1 |
| rshift r1, r2 \Rightarrow r3 | r1 \gg r2 \rightarrow r3 | 1 |
| output x | print out MEM(x) | 1 |

*These same operations, with different latencies, will appear in lab 3
Assume a register-to-register memory model, with 1 class of registers*