Local Register Allocation & Lab Exercise 1

Extended version

COMP 412

Fall 2005

Where are we (and why)?

Register Allocation

- Chapter 13 in EAC
  - 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

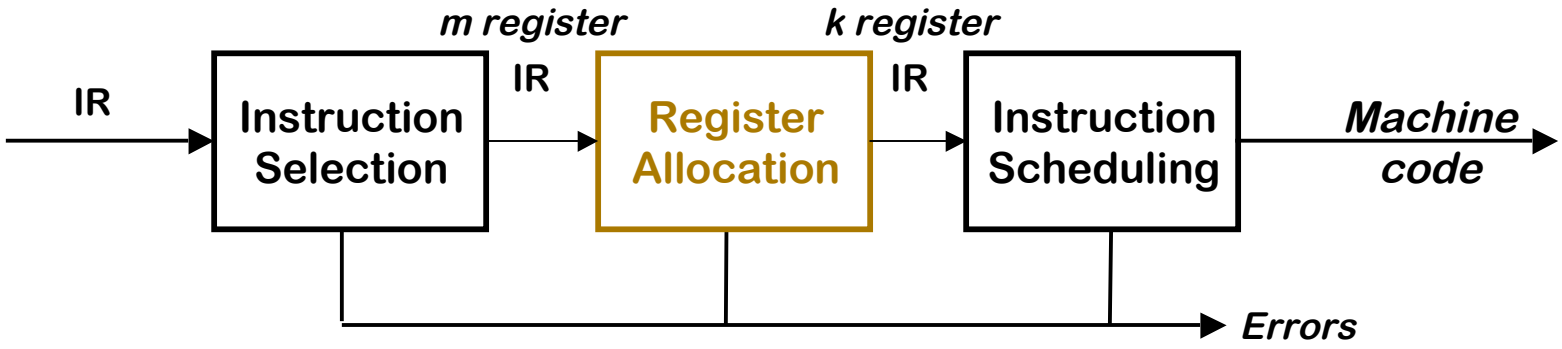
Why are we here?

- 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 *k* (or fewer) 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

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

Consider a fragment of assembly code (or ILOC)

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

The Problem

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

Register allocation is described in Chapters 1 & 13 of EAC

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 scheduling)
- Minimize inserted code — both dynamic & static measures
- Make good use of any *extra* registers

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

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's "local" ? (as opposed to "global")
  - A local transformation operates on basic blocks
  - Many optimizations are done locally
- Does local allocation solve the problem?
  - It produces decent register use inside a block
  - Inefficiencies can arise at boundaries between blocks
  - Your lab assumes that the block is the entire program
- How many passes can the allocator make?
  - This is an off-line problem
  - As many passes as it takes

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

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

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

Register Allocation

Can we do this optimally? (on real code?)

Local Allocation

- Simplified cases ⇒ O(n)
- Real cases ⇒ NP-Complete

Local Assignment

- Single size, no spilling ⇒ O(n)
- Two sizes ⇒ NP-Complete

Global Allocation

- NP-Complete for 1 register
- NP-Complete for *k* registers

(most sub-problems are NPC, too)

Global Assignment

- NP-Complete

Real compilers face real problems

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 ← …) 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)

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

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) == y
mult     r1, r2    ⇒ r3    // r3 ← 1028 · y
loadI    x         ⇒ r4    // r4 ← x
sub      r4, r2    ⇒ r5    // r5 ← x - y
loadI    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)
```

An Example

- Live Ranges

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

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

Top-down Allocator

The idea:

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

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)

r1 is used more often than r3

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

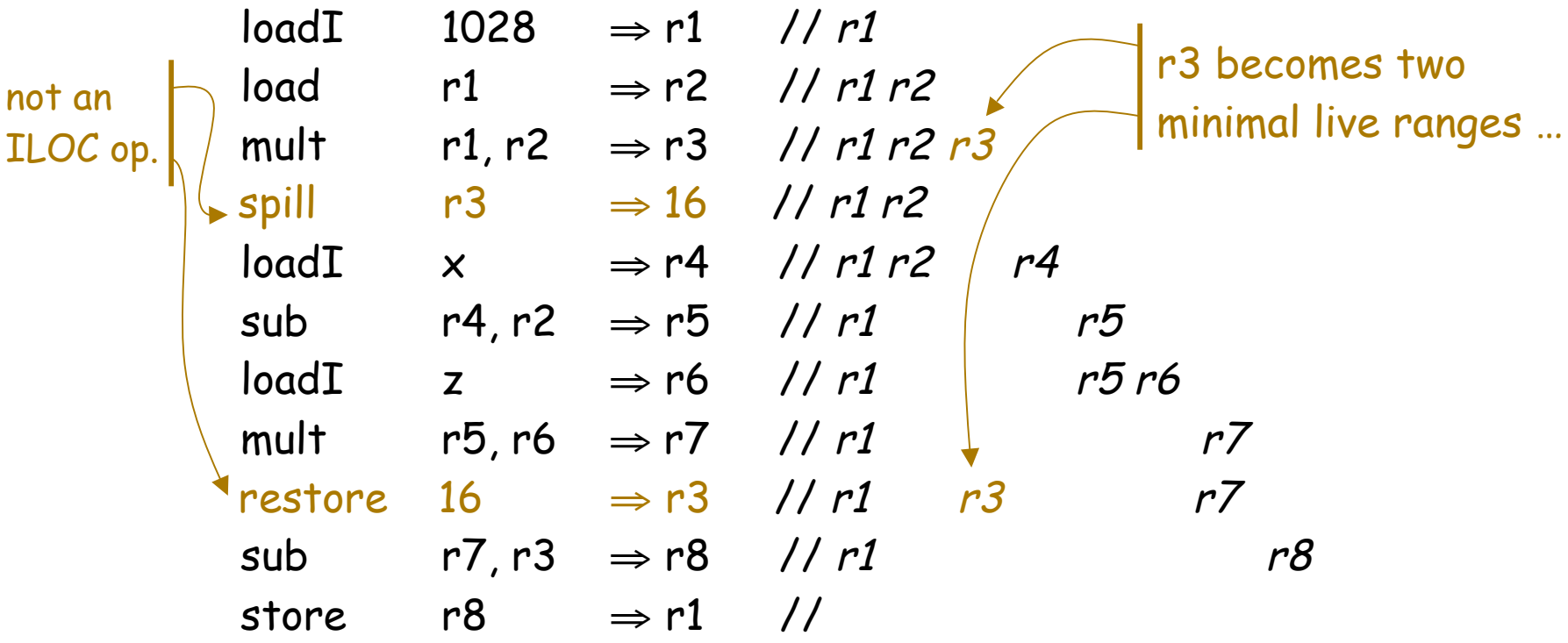
spill r3

restore r3

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)



"spill" and "restore" become stores and loads

→ In practice, relative to r_{arp}

→ In lab 1, to absolute addresses between 0 and 1023

An Example

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

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

```
// r1
// r1 r2
// r1 r2 r3
// r1 r2
// r1 r2     r4
// r1             r5
// r1             r5 r6
// r1                    r7
// r1        r3          r7
// r1                        r8
//
```

r3 becomes two minimal live ranges …

At most 3 values *live* at each point

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
sub       r7, r3   ⇒ r8     // r1                   r8
store     r8       ⇒ r1     //
```

possible delay

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

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
loadI    x        ⇒ r4    // r1 r2 r3 r4
sub      r4, r2   ⇒ r5    // r1    r3    r5
loadI    z        ⇒ r6    // r1    r3    r5 r6
mult     r5, r6   ⇒ r7    // r1    r3          r7
sub      r7, r3   ⇒ r8    // r1                r8
store    r8       ⇒ r1    //
```

spill r1

restore r1

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*
loadI	z	⇒ r6	// *r3 r5 r6*
mult	r5, r6	⇒ r7	// *r3 r7*
sub	r7, r3	⇒ r8	// *r8*
restore	20	⇒ r1	// *r1 r8*
store	r8	⇒ r1	//

At most 3 values *live* at each point

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 Section 13.3.1 & Exercise 2a, Section 13.3)
- Implement a version of the bottom-up allocator
  (See 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: Thursday, September 15, 2005, 11:59 PM

Documentation: Friday, September 16, 2005, 11:59 PM

Test & report blocks will be available from 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	⇒	r_{arp}	*none*
loadAI	r_{arp}, @w	⇒	r_w	r_{arp}, r_w
loadI	2	⇒	r_2	r_{arp}, r_w, r_2
loadAI	r_{arp}, @x	⇒	r_x	r_{arp}, r_w, r_2
loadAI	r_{arp}, @y	⇒	r_y	r_{arp}, r_w, r_x, r_2
loadAI	r_{arp}, @z	⇒	r_z	$r_{arp}, r_w, r_y, r_x, r_2$
mult	r_w, r_2	⇒	r_w	$r_{arp}, r_w, r_z, r_y, r_x, r_2$
mult	r_w, r_x	⇒	r_w	$r_{arp}, r_w, r_z, r_y, r_x$
mult	r_w, r_y	⇒	r_w	r_{arp}, r_w, r_z, r_y
mult	r_w, r_z	⇒	r_w	r_{arp}, r_w, r_z
storeAI	r_w	⇒	r_{arp}, @w	

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

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: Introduction to scanning (Lexical Analysis, Ch. 2)

Lab One

ILOC subset:

Operation				*Meaning*	*Latency*
load	r1	⇒	r2	MEM(r1) → r2	2
store	r1	⇒	r2	r1 → MEM(r2)	2
loadI	x	⇒	r1	x → r1	1
add	r1, r2	⇒	r3	r1 + r2 →r3	1
sub	r1, r2	⇒	r3	r1 - r2 → r3	1
mult	r1, r2	⇒	r3	r1 x r2 → r3	1
lshift	r1, r2	⇒	r3	r1 << r2 → r3	1
rshift	r1, r2	⇒	r3	r1 >> r2 → 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