COMP 412
FALL 2010

Local Register Allocation & Lab 1
COMP 412

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

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
  - http://www.clear.rice.edu/comp412

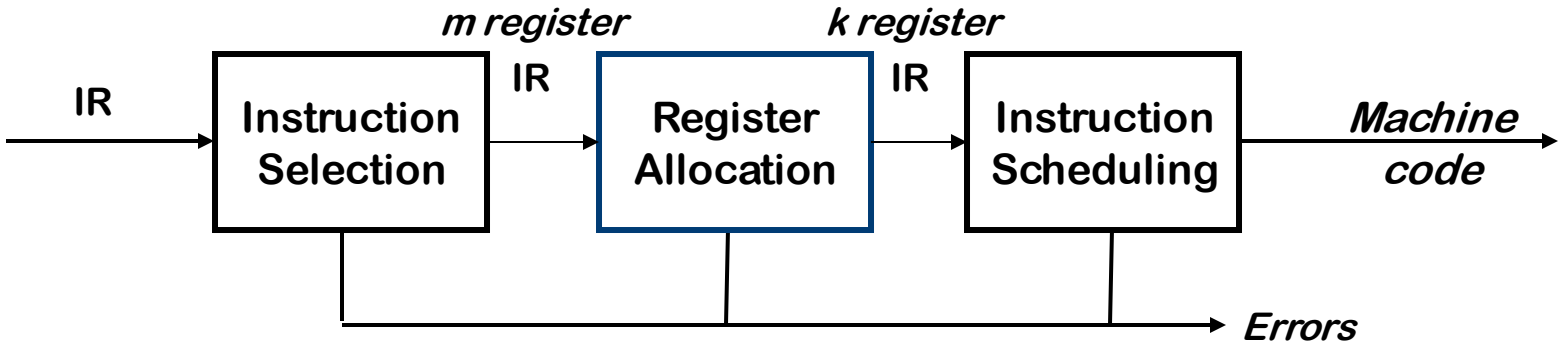
Why are we here?

- 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 $O(2^n)$

Notation: The literature on register allocation consistently uses *k* as the number of registers available on the target system.

Register Allocation

r0 holds base address for local variables

@x is constant offset of x from r0

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, @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)
```

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 ≅ 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

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

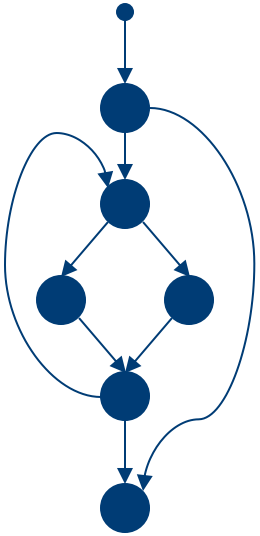
Ignore, for the moment, exceptions

Role of Basic Blocks in 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)

Register Allocation

Optimal register allocation is hard

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

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.

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 x b

```
loadI    2        ⇒ r1
loadAI   r0, @b   ⇒ r2
add      r1, r2   ⇒ r3
loadAI   r0, @a   ⇒ r4
sub      r4, r3   ⇒ r5
```

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 and on the last slide of this lecture

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 x 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 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

Observations

The Register Allocator does not need to "understand" the code

- It needs to distinguish definitions from uses
  - Definitions might need to store a spilled value
  - Uses might need to load a spilled value
- ILOC makes definitions and uses pretty clear
  - The assignment arrow, ⇒, separates uses from definitions
    - *Except on the store operation, which uses all its register operands*
  - That is the point of the arrow!
- Your allocator needs to know, by opcode, how many definitions and how many uses it should see
  - Beyond that, the meaning of the ILOC is somewhat irrelevant to the allocator

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
- If MAXLIVE > *k*, need to reserve *F* registers for spilling

Finding live ranges is harder in the global case

Concrete Example of MAXLIVE

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)
```

Store uses this register & defines a memory location.

The code 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. Remember, the allocator does not need to understand the computation. It just needs to preserve the computation.

Concrete Example of MAXLIVE

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* after an operation if it has been defined & has a use in the future

Concrete Example of MAXLIVE

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     //
```

MAXLIVE is 4

Remember, r1 is a use, not a definition

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
- Reserve registers for use in spills, say *r* registers

Algorithm

- Rank values by number of occurrences
- Allocate first *k* – *r* values to registers
- Rewrite code to reflect these choices

Move values with no register into memory

(add LOADs & STOREs)

Programmers applied this idea by hand in the 70's & early 80's

- C's register declaration

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

Top-down Allocator

How many registers must the allocator reserve?

- Need registers to compute spill addresses & load values
- Number depends on target architecture
  - Typically, must be able to load 2 values
- Reserve these registers for spilling

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

- 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

Back to the Example

Top down (3 registers, need 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*					
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	//					

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

Back to the Example

Top down (3 registers, need 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*	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	// *r1 r3 r7*	
sub	r7, r3	⇒ r8	// *r1*	restore r3 → *r8*
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, need 2 for operands)

not an ILOC op.

```
loadI    1028      ⇒ r1     // r1
load     r1        ⇒ r2     // r1 r2
mult     r1, r2    ⇒ r3     // r1 r2 r3
spill    r3        ⇒ 16     // r1 r2
load     x         ⇒ r4     // r1 r2    r4
sub      r4, r2    ⇒ r5     // r1          r5
load     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     //
```

r3 becomes two minimal live ranges …

"spill" and "restore" become stores and loads

- In practice, relative to r_{arp} (r_0 in this lecture)
- In lab 1, to absolute addresses between 0 and 1023

An Example

Top down (3 registers, need 2 for operands)

loadI	1028	⇒ r1	// r1	
load	r1	⇒ r2	// r1 r2	
mult	r1, r2	⇒ r3	// r1 r2 r3	
spill	r3	⇒ 16	// r1 r2	
load	x	⇒ r4	// r1 r2 r4	
sub	r4, r2	⇒ r5	// r1 r5	
load	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	//	

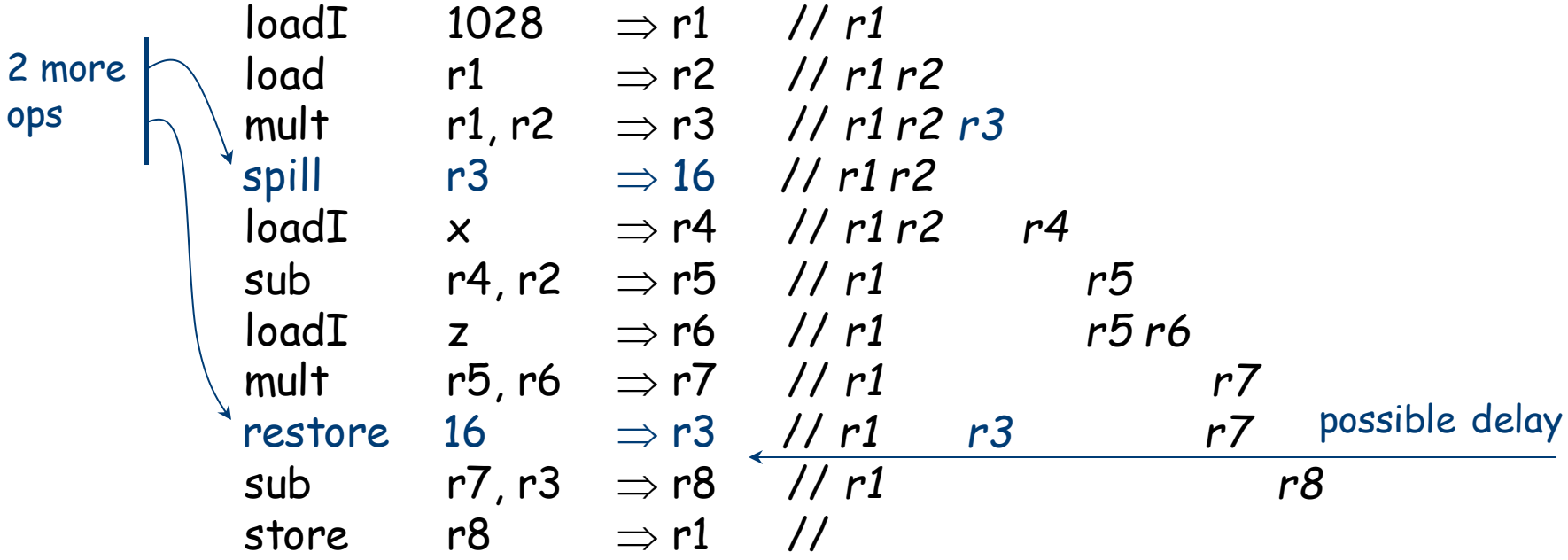
r3 becomes two minimal live ranges …

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, need 2 for operands)



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; need 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    //
```

All registers are used at this point

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; need 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

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; need 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

Your 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 15, 2010, 11:59 PM

Documentation: Friday, September 19, 2010, 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	⇒	r_{arp}	*none*
loadAI	r_{arp}, @a	⇒	r_a	r_{arp},r_a
loadI	2	⇒	r_2	r_{arp},r_a ,r_2
loadAI	r_{arp}, @b	⇒	r_b	r_{arp},r_a ,r_2
loadAI	r_{arp}, @c	⇒	r_c	r_{arp},r_a ,r_b ,r_2
loadAI	r_{arp}, @d	⇒	r_d	r_{arp},r_a ,r_c ,r_b ,r_2
mult	r_a, r_2	⇒	r_a	r_{arp},r_a ,r_d ,r_c ,r_b,r_2
mult	r_a, r_b	⇒	r_a	r_{arp},r_a ,r_d ,r_c,r_b
mult	r_a, r_c	⇒	r_a	r_{arp},r_a,r_d,r_c
mult	r_a, r_d	⇒	r_a	r_{arp},r_a,r_d
storeAI	r_d	⇒	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

Lab One

ILOC subset:

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