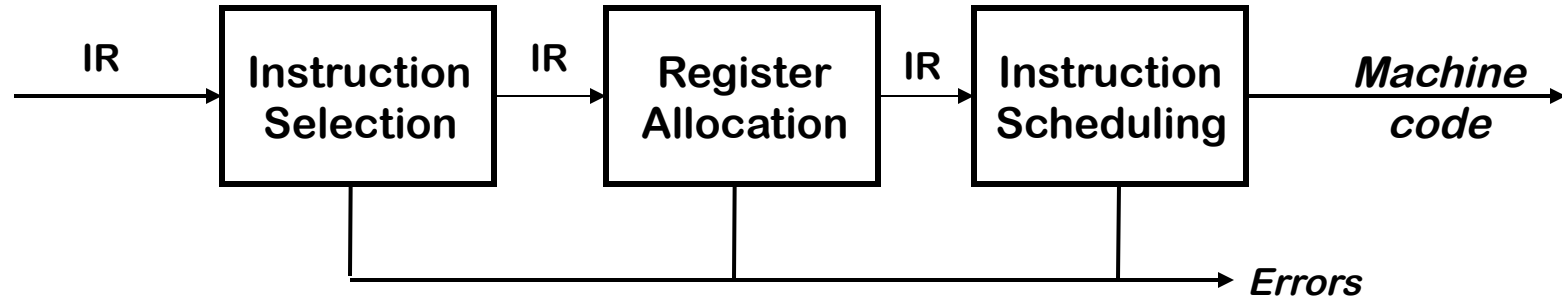


CS415 Compilers Register Allocation

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

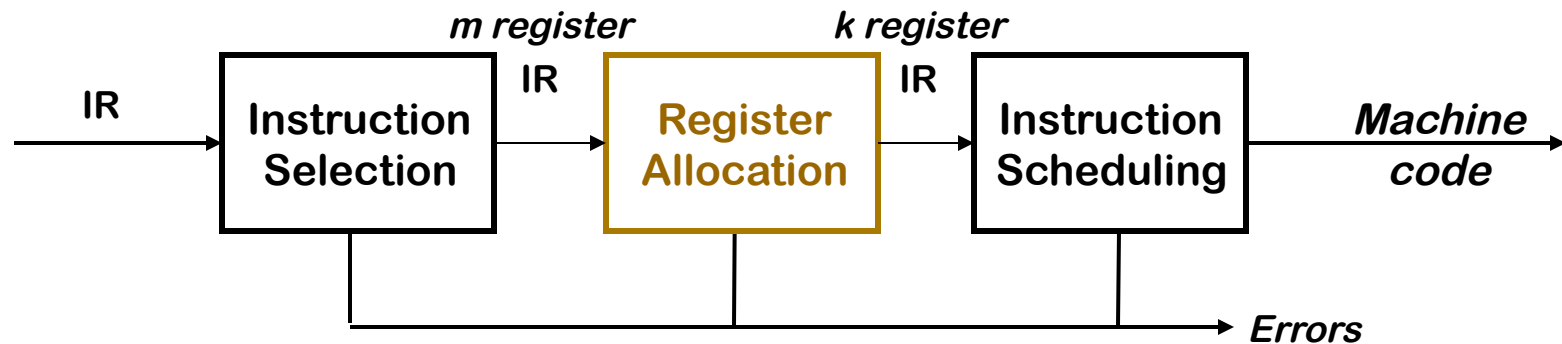


Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been *less* successful in the back end

Part of the compiler's back end



Critical properties

- Produce correct code that uses k (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold *spilled values*
- Operate efficiently
 $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

Readings: EaC 13.1-13.3, Appendix A (ILOP)

Local: within single basic block

Global: across procedure/function

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
- Here: we only use a small subset of ILOC

Naïve Representation:

loadl	2		r1
loadAl	r0	@y	r2
add	r1	r2	r3
store	r0		r4
sub	r4	r3	r5

Quadruples:

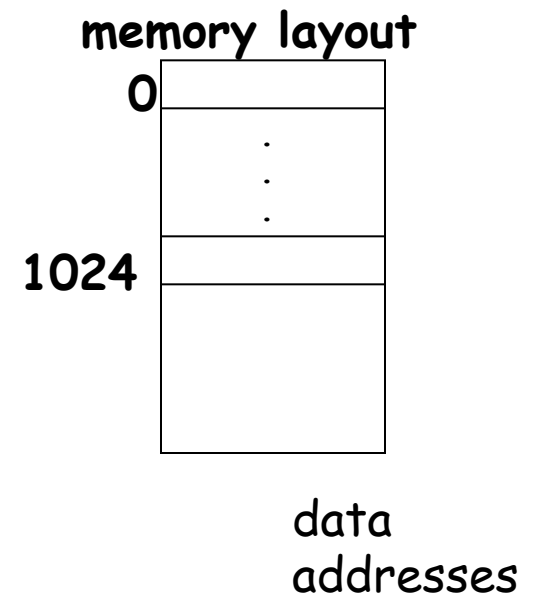
- table of $k \times 4$ small integers
- simple record structure
- easy to reorder
- all names are explicit

ILOC is described in Appendix A of EAC

Simulator at ~zz124/cs415_2014/ILOC_Simulator on ilab cluster

Source code

```
A = 5;  
B = 6;  
A = A + B;
```



Source code

```

A = 5;
B = 6;
A = A + B;

```

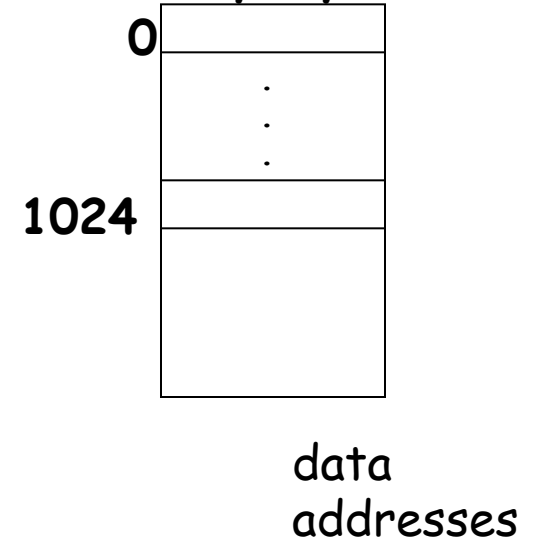
ILOC code

```

loadI 5  ⇒ r1
// compute address of A in r2
...
store r1 ⇒ r2 // content(A) = r1
loadI 6  ⇒ r3
// compute address of B in r4
...
store r3 ⇒ r4 // content(B) = r3
add r1, r3 ⇒ r5
// compute address of A in r6
...
store r5 ⇒ r6 // content(A) = r1 + r3

```

memory layout



Is this code correct?

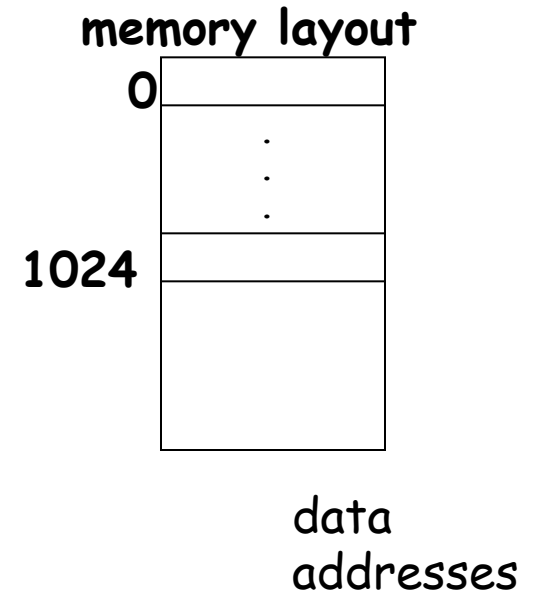
Source code
foo (var A, B)

```
A = 5;
B = 6;
A = A + B;
end foo;
```

```
call foo(x,x);
print x;
```

ILOC code

```
loadI 5  ⇒ r1
// compute address of A in r2
...
store r1 ⇒ r2 // content(A) = r1
loadI 6  ⇒ r3
// compute address of B in r4
...
store r3 ⇒ r4 // content(B) = r3
add r1, r3 ⇒ r5
// compute address of C in r6
...
store r5 ⇒ r6 // content(C) = r1 + r3
```



Is this code correct?

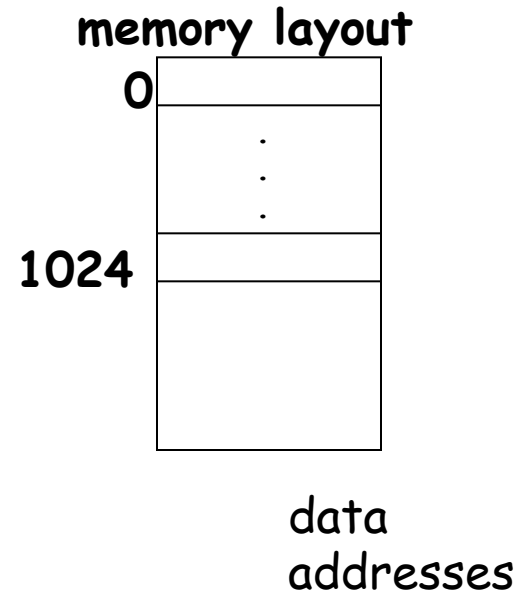
Source code
foo (var A, B)

```
A = 5;
B = 6;
A = A + B;
end foo;
```

```
call foo(x,x);
print x;
```


ILOC code

```
loadI 5  ⇒ r1
// compute address of A in r2
...
store r1 ⇒ r2 // content(A) = r1
loadI 6  ⇒ r3
// compute address of B in r4
...
store r3 ⇒ r4 // content(B) = r3
add r1, r3 ⇒ r5
// compute address of C in r6
...
store r5 ⇒ r6 // content(C) = r1 + r3
```



Incorrect for
call-by-reference! Is this code correct?

- **register-register model**
 - Values that may safely reside in registers are assigned to a unique virtual register
 - Register allocation/assignment maps virtual registers to limited set of physical registers
 - Register allocation/assignment pass needed to make code “work”
- **memory-memory model**
 - All values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result into memory)
 - Register allocation/assignment has to try to identify cases where values can be safely kept in registers
 - Safety verification is hard at the low level or program abstraction
 - Even without register allocation/assignment, code will “work”

- **register-register model**  Will use this one from now on
 - Values that may safely reside in registers are assigned to a unique virtual register
 - Register allocation/assignment maps virtual registers to limited set of physical registers
 - Register allocation/assignment pass needed to make code “work”
- **memory-memory model**
 - All values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result into memory)
 - Register allocation/assignment has to try to identify cases where values can be safely kept in registers
 - Safety verification is hard at the low level or program abstraction
 - Even without register allocation/assignment, code will “work”

Consider a fragment of assembly code (or ILOC)

```
loadI    1024    ⇒ r0    // r0 ← 1024
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: a value is a *pseudo-register*
- 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
ILOC is discussed in Appendix A of EAC

Consider a fragment of assembly code (or ILOC)

	loadI	1024	\Rightarrow r0	// $r0 \leftarrow 1024$
	loadI	2	\Rightarrow r1	// $r1 \leftarrow 2$
address	loadAI	r0, @y	\Rightarrow r2	// $r2 \leftarrow y$
immediate	mult	r1, r2	\Rightarrow r3	// $r3 \leftarrow 2 \cdot y$
	loadAI	r0, @x	\Rightarrow r4	// $r4 \leftarrow x$
	sub	r4, r3	\Rightarrow r5	// $r5 \leftarrow x - (2 \cdot y)$

The Problem

- At each instruction, decide which *values* to keep in registers
 → Note: a value is a *pseudo-register*
- 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
ILOC is discussed in Appendix A of EAC

The General Task

- At each point in the code, pick the values to keep in registers
- Insert code to move values between registers & memory
 - No instruction reordering (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

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

- 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
 - The first project (instruction scheduling) 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
- memory-to-memory vs. register-to-register model
 - code shape and safety issues

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
- Here: we only use a small subset of ILOC

Naïve Representation:

loadl	2		r1
loadAl	r0	@y	r2
add	r1	r2	r3
storeAl	r0	@x	r4
sub	r4	r3	r5

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

Simulator at ~zz124/cs415_2014/ILOC_Simulator on ilab cluster

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

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)

Local Assignment

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

Global Assignment

- NP-Complete

Real compilers face real problems

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

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)
 - *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 $\text{MAXLIVE} \leq k$, 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

Finding live ranges is harder in the global (non local) case

Top-down allocator

- 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 (different register assignments for different parts of live range)
- Save some registers for the values relegated to memory (feasible set F)

The idea:

- Keep busiest values in a register
- Use the feasible (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
- 
- SPILL: Move values with no register into memory (add LOADs & STOREs)

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**
- Sound familiar? Think page replacement ...

➤ Here is a sample code sequence

loadI	1028	⇒ r1	// $r1 \leftarrow 1028$
load	r1	⇒ r2	// $r2 \leftarrow \text{MEM}(r1) == y$
mult	r1, r2	⇒ r3	// $r3 \leftarrow 1028 \cdot y$
loadI	5	⇒ r4	// $r4 \leftarrow 5$
sub	r4, r2	⇒ r5	// $r5 \leftarrow 5 - y$
loadI	8	⇒ r6	// $r6 \leftarrow 8$
mult	r5, r6	⇒ r7	// $r7 \leftarrow 8 \cdot (5 - y)$
sub	r7, r3	⇒ r8	// $r5 \leftarrow 8 \cdot (5 - y) - (1028 \cdot y)$
store	r8	⇒ r1	// $\text{MEM}(r1) \leftarrow 8 \cdot (5 - y) - (1028 \cdot y)$

➤ 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

- Top down (3 physical registers (k-F): 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 $\text{MAXLIVE} > (k-F)$
- number of occurrences of virtual register
 - length of live range

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

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

- Consider statements with MAXLIVE > (k-F)

- number of occurrences of virtual register (most important)
- length of live range (tie breaker)
- Note: This is different from the algorithm discussed in EAC!

- 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

- 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 $F = \{f1, f2, \dots\}$?
 - 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?

More Register Allocation Instruction Scheduling

Read EaC: Chapters 12.1 - 12.3

Recitation starts TODAY!

First homework will be posted tomorrow.

http://www.cs.rutgers.edu/~zz124/cs415_spring2014/