RUTGERS

THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Register Allocation

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

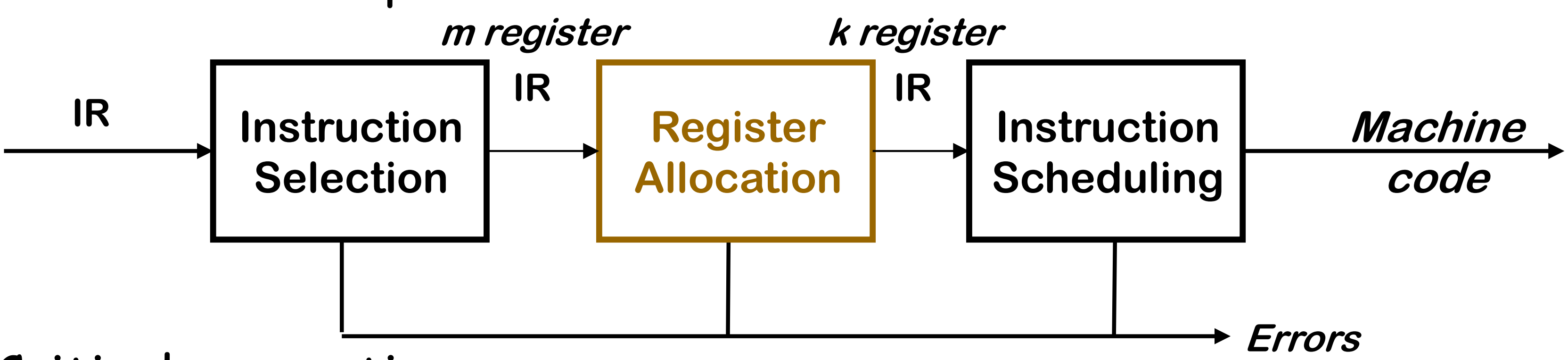
Local Register Allocation (and Assignment)

Readings: EaC 13.1-13.2, Appendix A (ILOC)

Local: within single basic block
Global: across procedure/function

Register Allocation

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 $O(2^n)$

Memory Model / Code Shape

- register-register model ← Will use this one from now on
  - → Values that may safely reside in registers are assigned to a unique virtual register (alias analysis)
  - → 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 levels of program abstraction
  - → Even without register allocation/assignment, code will "work"

Memory Model / Code Shape

- register-register model
  - → Values that may safely reside in registers are assigned to a unique virtual register (alias analysis; unambiguous values); there are different “flavors”
- memory-memory model
  - → All program-named values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result back into memory)

```
a := 1
b := 2
c := a + b + 3
```

assumption: no aliasing

register-register

all in registers

```
loadI 1 ⇒ r1
loadI 2 ⇒ r2
add r1, r2 ⇒ r3
loadI 3 ⇒ r4
add r3, r4 ⇒ r5
```

preserve memory view (memory consistency)

```
loadI 1 ⇒ r1
storeAI r1 ⇒ r0,@a
loadI 2 ⇒ r2
storeAI r2 ⇒ r0,@b
add r1, r2 ⇒ r3
loadI 3 ⇒ r4
add r3, r4 ⇒ r5
storeAI r5 ⇒ r0,@c
```

memory-memory

```
loadI 1 ⇒ r1
storeAI r1 ⇒ r0,@a
loadI 2 ⇒ r2
storeAI r2 ⇒ r0,@b
loadAI r0,@a ⇒ r3
loadAI r0,@b ⇒ r4
add r3, r4 ⇒ r5
loadI 3 ⇒ r7
add r5, r7 ⇒ r8
storeAI r8 ⇒ r0,@c
```

Register Allocation

Consider a fragment of assembly code (or ILOC)

loadI	2	$\Rightarrow$ r1	// $r1 \leftarrow 2$
loadAI	r0, @y	$\Rightarrow$ r2	// $r2 \leftarrow y$
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

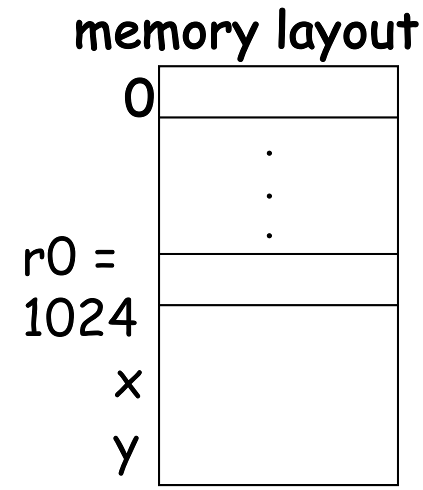
- At each instruction, decide which *values* to keep in registers
  - → Note: a value is a *pseudo-register (virtual register)*
  - → Simple if $|values| \leq |registers|$
- Harder if $|values| > |registers|$
- The compiler must automate this process

Register Allocation

Consider a fragment of assembly code (or ILOC)

address immediate

```
loadI    2       ⇒ r1    // r1 ← 2
loadAI   r0, 8   ⇒ r2    // r2 ← y
mult     r1, r2  ⇒ r3    // r3 ← 2 · y
loadAI   r0, 4   ⇒ 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 (virtual register)*
  - → 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 reordering 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

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
- How many passes can the allocator make?
  - → This is a compile-time ("off-line") problem (not done during program execution); typically, as many passes as it takes
- memory-to-memory vs. register-to-register model
  - → code shape and safety issues

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

Basic Approach of Allocators

Allocator may need to reserve physical registers to ensure feasibility

- Must be able to compute memory addresses
- Requires some minimal set of registers, *F*
  - → *F* depends on target architecture
- F contains registers to make spilling work
  - → set F registers "aside" for address computation & instruction execution, i.e. these are not available for register assignment
- Note: F physical registers need to be able to support the pathological case where all virtual registers are spilled

Notation:

k is the number of registers on the target machine

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

- The allocator can either
  - → Check for this situation
  - → Accept the fact that the technique is an approximation

Top-down Versus Bottom-up Allocation

Top-down allocator

- May use notion of "live ranges" of virtual registers
- 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
- Save some registers for the values relegated to memory (feasible set F)

Live Ranges (live on exit)

Assume i and j are two instructions in a basic block

A **value** (virtual register) 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 many (static) definitions can you have for a virtual register?
- 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 (virtual registers) live at *i*.

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

Finding live ranges is harder in the global case

Next topic

More Register Allocation EaC 13.1 – 13.3
(Top-down and Bottom-Up Allocation)