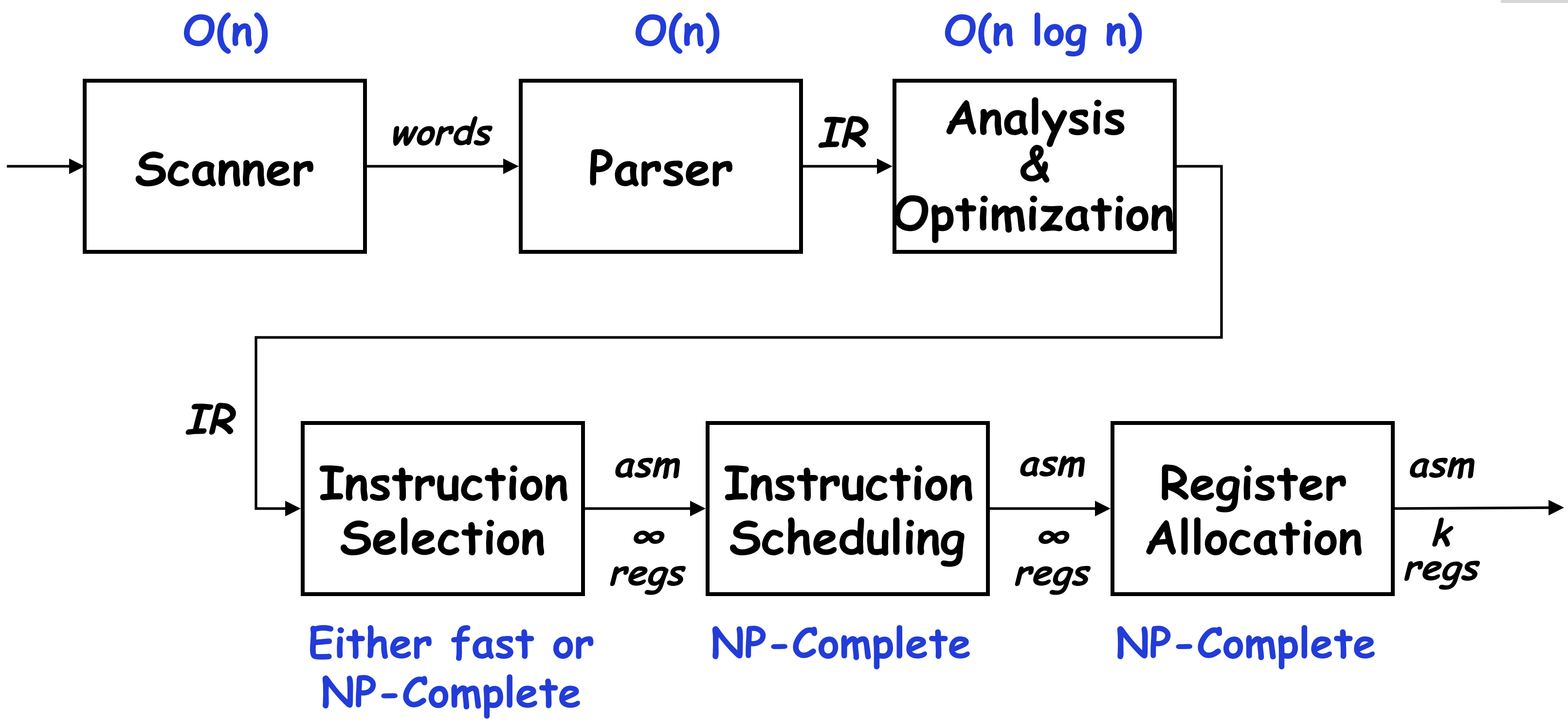
# Instruction Selection and Scheduling and Phase IV

# Structure of a Compiler

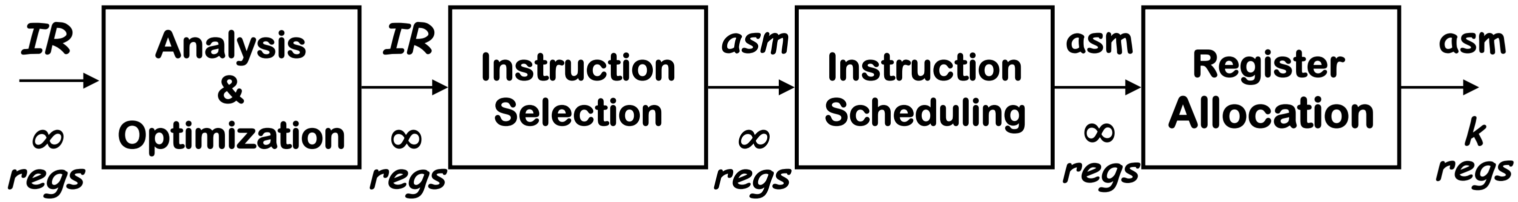



A compiler is a lot of fast stuff followed by some hard problems

- The hard stuff is mostly in optimization and code generation
- For superscalars, its allocation & scheduling that count

# Structure of a Compiler

For the rest of CISC672, we assume the following model




- Selection is fairly simple (problem of the 1980s)
- Allocation & scheduling are complex
- Operation placement is not yet critical *(unified register set)*

What about the IR ?

- Low-level, RISC-like IR called ILOC
- Has "enough" registers
- ILOC was designed for this stuff

**Branches, compares, & labels**
**Memory tags**
**Hierarchy of loads & stores**
**Provision for multiple ops/cycle**

# Definitions

Instruction selection

- Mapping *IR* into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling

- Reordering operations to hide latencies
- Assumes a fixed program *(set of operations)*
- Changes demand for registers

Register allocation

- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations

**These 3 problems are tightly coupled.**

# The Big Picture

How hard are these problems?

Instruction selection

- Can make locally optimal choices, with automated tool
- Global optimality is (undoubtedly) NP-Complete

Instruction scheduling

- Single basic block $\Rightarrow$ heuristics work well
- General problem, with control flow $\Rightarrow$ NP-Complete

Register allocation

- Single basic block, no spilling, & 1 register size $\Rightarrow$ linear time
- Whole procedure is NP-Complete

# The Big Picture

*Conventional wisdom says that we lose little by solving these problems independently*

This slide is full of "fuzzy" terms

Instruction selection
- Use some form of pattern matching
- Assume enough registers or target "important" values

Instruction scheduling
- Within a block, list scheduling is "close" to optimal
- Across blocks, build framework to apply list scheduling

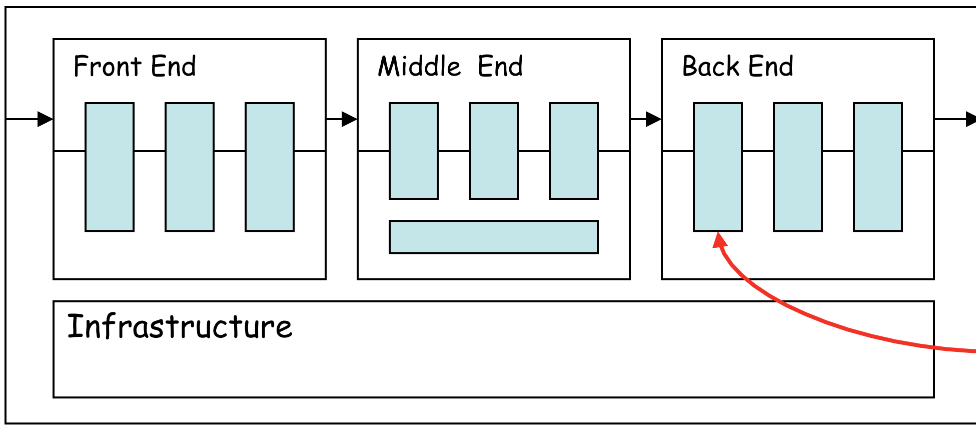
Optimal for > 85% of blocks

Register allocation
- Start from virtual registers & map "enough" into *k*
- With targeting, focus on good priority heuristic

# The Problem

Writing a compiler is a lot of work

- Would like to reuse components whenever possible
- Would like to automate construction of components




Today's lecture:

Automating Instruction Selection

- Front end construction is largely automated
- Middle is largely hand crafted
- (Parts of ) back end can be automated

# Definitions

Instruction selection
- Mapping *IR* into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling
- Reordering operations to hide latencies
- Assumes a fixed program *(set of operations)*
- Changes demand for registers

Register allocation
- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations

## The Problem

Modern computers (still) have many ways to do anything

Consider register-to-register copy in ILOC

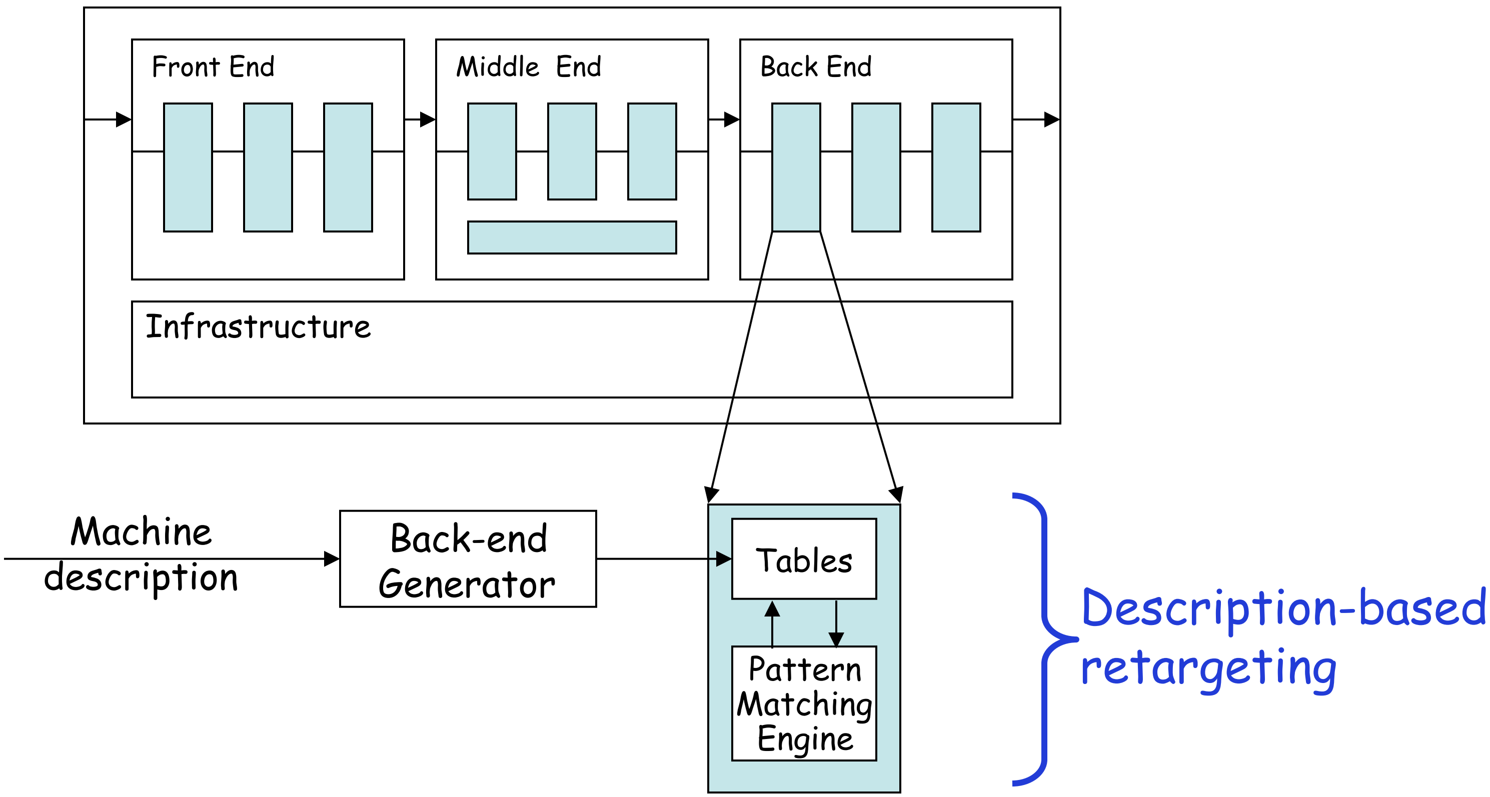
- Obvious operation is `i2i` $r_i \Rightarrow r_j$
- Many others exist

| | | |
|---|---|---|
| `addI` $r_i, 0 \Rightarrow r_j$ | `subI` $r_i, 0 \Rightarrow r_j$ | `lshiftI` $r_i, 0 \Rightarrow r_j$ |
| `multI` $r_i, 1 \Rightarrow r_j$ | `divI` $r_i, 1 \Rightarrow r_j$ | `rshiftI` $r_i, 0 \Rightarrow r_j$ |
| `orI` $r_i, 0 \Rightarrow r_j$ | `xorI` $r_i, 0 \Rightarrow r_j$ | … and others … |

- Human would ignore all of these
- Algorithm must look at all of them & find low-cost encoding
  - → Take context into account (*busy functional unit?*)

# The Goal

Want to automate generation of instruction selectors



# The Big Picture

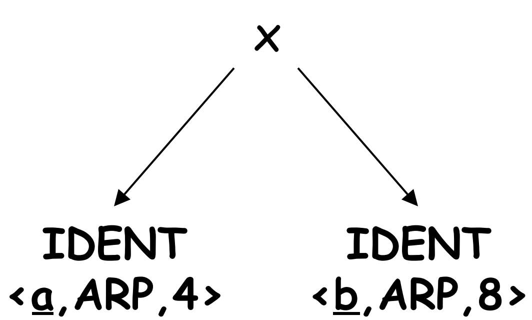
Need pattern matching techniques

- Must produce good code *(some metric for good)*
- Must run quickly

A treewalk code generator runs quickly

How good was the code?

**Tree**




**Treewalk Code**

loadI 4 $\Rightarrow r_5$
loadAO $r_{arp}, r_5 \Rightarrow r_6$

loadI 8 $\Rightarrow r_7$
loadAO $r_{arp}, r_7 \Rightarrow r_8$
mult $r_6, r_8 \Rightarrow r_9$

**Desired Code**

loadAI $r_{arp}, 4 \Rightarrow r_5$
loadAI $r_{arp}, 8 \Rightarrow r_6$
mult $r_5, r_6 \Rightarrow r_7$

# The Big Picture

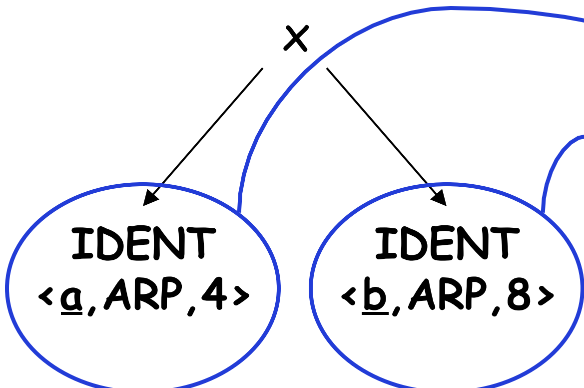
Need pattern matching techniques

- Must produce good code *(some metric for good)*
- Must run quickly

A treewalk code generator runs quickly

How good was the code?

**Tree**




**Treewalk Code**

loadI $4 \Rightarrow r_5$
loadAO $r_{arp}, r_5 \Rightarrow r_6$
loadI $8 \Rightarrow r_7$
loadAO $r_{arp}, r_7 \Rightarrow r_8$
mult $r_6, r_8 \Rightarrow r_9$

**Desired Code**

loadAI $r_{arp}, 4 \Rightarrow r_5$
loadAI $r_{arp}, 8 \Rightarrow r_6$
mult $r_5, r_6 \Rightarrow r_7$

Pretty easy to fix. See 1st digression in Ch. 7

# The Big Picture

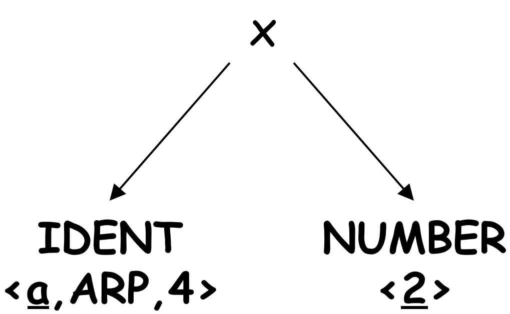
Need pattern matching techniques

- Must produce good code *(some metric for good)*
- Must run quickly

A treewalk code generator runs quickly

How good was the code?

**Tree**




**Treewalk Code**

| | | |
|---|---|---|
| loadI | 4 | $\Rightarrow r_5$ |
| loadAO | $r_{arp}, r_5$ | $\Rightarrow r_6$ |
| loadI | 2 | $\Rightarrow r_7$ |
| mult | $r_6, r_7$ | $\Rightarrow r_8$ |

**Desired Code**

| | | |
|---|---|---|
| loadAI | $r_{arp}, 4$ | $\Rightarrow r_5$ |
| multI | $r_5, 2$ | $\Rightarrow r_7$ |

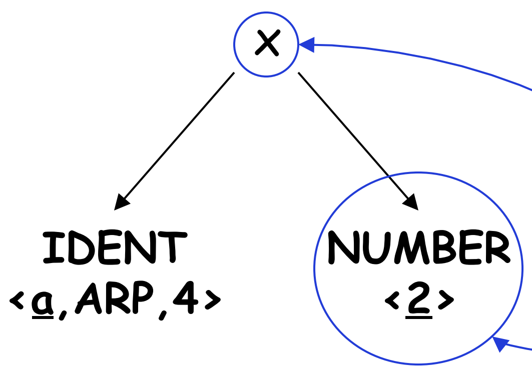
# The Big Picture

Need pattern matching techniques

- Must produce good code *(some metric for good)*
- Must run quickly

A treewalk code generator runs quickly

How good was the code?

| Tree | Treewalk Code | Desired Code |
|---|---|---|
|  | loadI 4 $\Rightarrow r_5$<br>loadAO $r_{arp}, r_5 \Rightarrow r_6$<br>loadI 2 $\Rightarrow r_7$<br>mult $r_6, r_7 \Rightarrow r_8$ | loadAI $r_{arp}, 4 \Rightarrow r_5$<br>multI $r_5, 2 \Rightarrow r_7$ |

Must combine these

This is a nonlocal problem

# The Big Picture

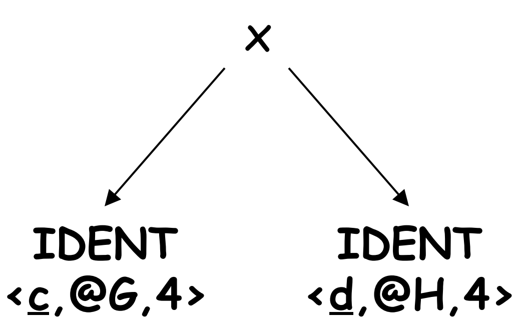
Need pattern matching techniques

- Must produce good code *(some metric for good)*
- Must run quickly

A treewalk code generator runs quickly

How good was the code?

**Tree**




**Treewalk Code**

| | | |
|---|---|---|
| loadI | @G | $\Rightarrow r_5$ |
| loadI | 4 | $\Rightarrow r_6$ |
| loadAO | $r_5, r_6$ | $\Rightarrow r_7$ |
| loadI | @H | $\Rightarrow r_7$ |
| loadI | 4 | $\Rightarrow r_8$ |
| loadAO | $r_8, r_9$ | $\Rightarrow r_{10}$ |
| mult | $r_7, r_{10}$ | $\Rightarrow r_{11}$ |

**Desired Code**

| | | |
|---|---|---|
| loadI | 4 | $\Rightarrow r_5$ |
| loadAI | $r_5$,@G | $\Rightarrow r_6$ |
| loadAI | $r_5$,@H | $\Rightarrow r_7$ |
| mult | $r_6, r_7$ | $\Rightarrow r_8$ |

# The Big Picture

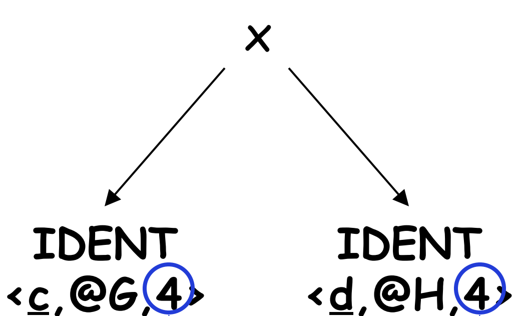
Need pattern matching techniques

- Must produce good code *(some metric for good)*
- Must run quickly

A treewalk code generator can meet the second criteria

How did it do on the first ?

**Tree**




Common offset

**Treewalk Code**

| | | |
|---|---|---|
| loadI | @G | $\Rightarrow r_5$ |
| loadI | 4 | $\Rightarrow r_6$ |
| loadAO | $r_5, r_6$ | $\Rightarrow r_7$ |
| loadI | @H | $\Rightarrow r_7$ |
| loadI | 4 | $\Rightarrow r_8$ |
| loadAO | $r_8, r_9$ | $\Rightarrow r_{10}$ |
| mult | $r_7, r_{10}$ | $\Rightarrow r_{11}$ |

**Desired Code**

| | | |
|---|---|---|
| loadI | 4 | $\Rightarrow r_5$ |
| loadAI | $r_5$,@G | $\Rightarrow r_6$ |
| loadAI | $r_5$,@H | $\Rightarrow r_7$ |
| mult | $r_6, r_7$ | $\Rightarrow r_8$ |

Again, a nonlocal problem

## How do we perform this kind of matching ?

Tree-oriented IR suggests pattern matching on trees

- Tree-patterns as input, matcher as output
- Each pattern maps to a target-machine instruction sequence
- Use dynamic programming or bottom-up rewrite systems

Linear IR suggests using some sort of string matching

- Strings as input, matcher as output
- Each string maps to a target-machine instruction sequence
- Use text matching or peephole matching

In practice, both work well; matchers are quite different

# Phase IV Discussion

You are given a working coolc binary that works, so use it!

- Compile some simple coolc programs and look at assembly output. Reverse Engineering is the key!
- Main goal of project is to generate code for expressions.

Example: $X \leftarrow E_1 + E_2$

- Compute $E_1$ (e.g., result in $a0)
- Store result on stack (e.g., store $a0)
- Compute $E_2$
- Load $E_1$ (into a temporary)
- Perform Add
- Store result on stack

**Need to figure out how many temporaries you need.**
**Naïve approach: Use a temporary for every lefthand side of an operator.**

# Instruction Scheduling

# What Makes Code Run Fast?

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Execution time is *order-dependent*

Assumed latencies *(conservative)*

| Operation | Cycles |
|---|---|
| load | 3 |
| store | 3 |
| loadI | 1 |
| add | 1 |
| mult | 2 |
| fadd | 1 |
| fmult | 2 |
| shift | 1 |
| branch | 0 to 8 |

- **Loads & stores may or may not block**
  - **Non-blocking ⇒fill those issue slots**
- **Instructions can have different latencies and use different resources**
- **Branches typically have delay slots**
  - **Fill slots with unrelated operations**
  - **Percolates branch upward**
- **Scheduler should hide the latencies**

# Example

$$w \leftarrow w * 2 * x * y * z$$

| *Cycles* | *Simple schedule* | | |
|---|---|---|---|
| 1 | loadAI | r0,@w | ⇒ r1 |
| 4 | add | r1,r1 | ⇒ r1 |
| 5 | loadAI | r0,@x | ⇒ r2 |
| 8 | mult | r1,r2 | ⇒ r1 |
| 9 | loadAI | r0,@y | ⇒ r2 |
| 12 | mult | r1,r2 | ⇒ r1 |
| 13 | loadAI | r0,@z | ⇒ r2 |
| 16 | mult | r1,r2 | ⇒ r1 |
| 18 | storeAI | r1 | ⇒ r0,@w |
| 21 | r1 is free | | |

**2 registers, 20 cycles**

| *Cycles* | *Schedule loads early* | | |
|---|---|---|---|
| 1 | loadAI | r0,@w | ⇒ r1 |
| 2 | loadAI | r0,@x | ⇒ r2 |
| 3 | loadAI | r0,@y | ⇒ r3 |
| 4 | add | r1,r1 | ⇒ r1 |
| 5 | mult | r1,r2 | ⇒ r1 |
| 6 | loadAI | r0,@z | ⇒ r2 |
| 7 | mult | r1,r3 | ⇒ r1 |
| 9 | mult | r1,r2 | ⇒ r1 |
| 11 | storeAI | r1 | ⇒ r0,@w |
| 14 | r1 is free | | |

**3 registers, 13 cycles**

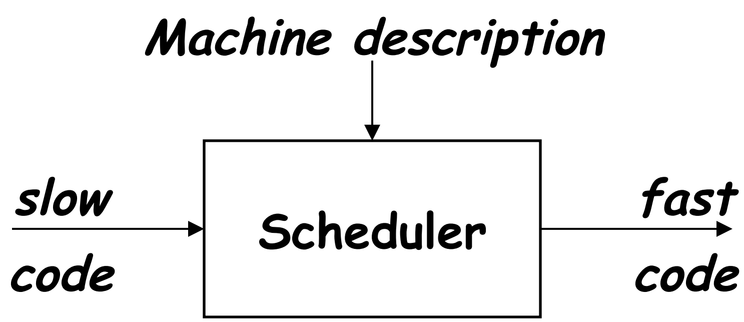
**Reordering operations for speed is called instruction scheduling**

# Instruction Scheduling (Engineer's View)

## The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

## The Concept




**The task**

- **Produce correct code**
- **Minimize wasted cycles**
- **Avoid spilling registers**
- **Operate efficiently**

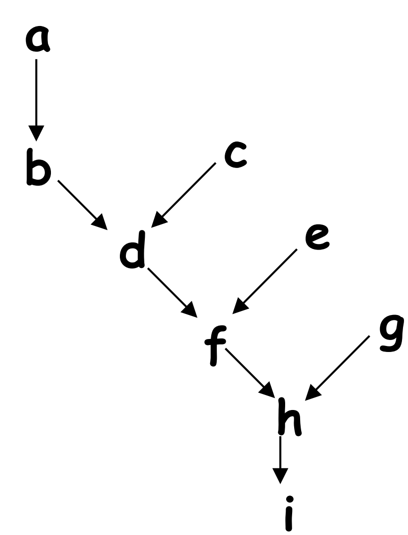
# Instruction Scheduling (The Abstract View)

To capture properties of the code, build a <u>dependence graph</u> $G$

- Nodes $n \in G$ are instructions
- An edge $e = (n_1,n_2) \in G$ if a "data hazard" between them
  - → $n_2$ uses the result of $n_1$ *(raw)*
  - → $n_2$ writes to same location as $n_1$ *(waw)*
  - → $n_2$ writes to a location used by $n_1$ *(war)*

| | | | |
|---|---|---|---|
| a: | loadAI | r0,@w | ⇒ r1 |
| b: | add | r1,r1 | ⇒ r1 |
| c: | loadAI | r0,@x | ⇒ r2 |
| d: | mult | r1,r2 | ⇒ r1 |
| e: | loadAI | r0,@y | ⇒ r2 |
| f: | mult | r1,r2 | ⇒ r1 |
| g: | loadAI | r0,@z | ⇒ r2 |
| h: | mult | r1,r2 | ⇒ r1 |
| i: | storeAI | r1 | ⇒ r0,@w |

The Code




The Dependence Graph

# Instruction Scheduling (What's so difficult?)

Critical Points

- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling *hard* (NP-complete)

Local scheduling is the simple case

- Restricted to straight-line code
- Consistent and predictable latencies

# Instruction Scheduling

The big picture

1. Build a dependence graph, *P*
2. Compute a *priority function* over the nodes in *P*
3. Use list scheduling to construct a schedule, one cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Update the ready queue

Local list scheduling

- The dominant algorithm for twenty years
- A greedy, heuristic, local technique

# Local List Scheduling

```
Cycle ← 1
Ready ← nodes of P with no predecessors
Active ← Ø

while (Ready ∪ Active ≠ Ø)
  if (Ready ≠ Ø) then
    remove an op from Ready
    S(op) ← Cycle
    Active ← Active ∪ op

  Cycle ← Cycle + 1

  for each op ∈ Active
     if (S(op) + delay(op) ≤ Cycle) then
       remove op from Active
       for each predecessor s of op in P
          if (s is ready) then
            Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

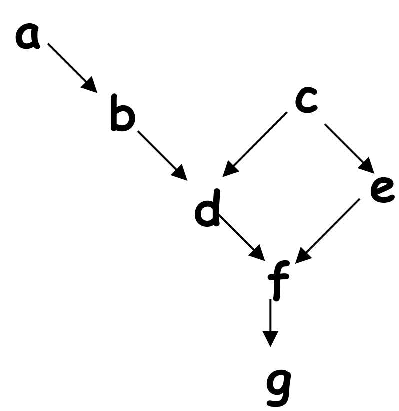
If successor's operands are ready, put it on Ready

# Scheduling Example

1. Build the dependence graph

| | | | |
|---|---|---|---|
| a: | loadAI | r0,@w | ⇒ r1 |
| b: | add | r1,r1 | ⇒ r1 |
| c: | loadAI | r0,@x | ⇒ r2 |
| d: | mult | r1,r2 | ⇒ r1 |
| e: | loadAI | r0,@z | ⇒ r2 |
| f: | mult | r1,r2 | ⇒ r1 |
| g: | storeAI | r1 | ⇒ r0,@w |

The Code



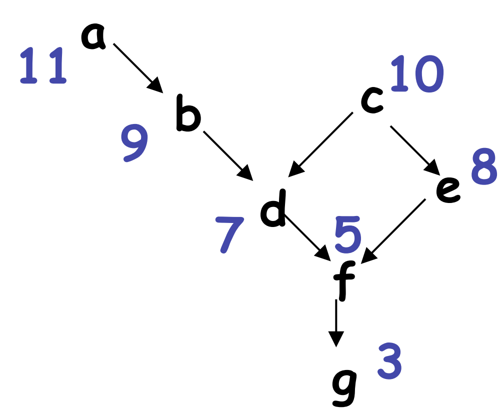

The Dependence Graph

# Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

| | | | |
|---|---|---|---|
| a: | loadAI | r0,@w | ⇒ r1 |
| b: | add | r1,r1 | ⇒ r1 |
| c: | loadAI | r0,@x | ⇒ r2 |
| d: | mult | r1,r2 | ⇒ r1 |
| e: | loadAI | r0,@z | ⇒ r2 |
| f: | mult | r1,r2 | ⇒ r1 |
| g: | storeAI | r1 | ⇒ r0,@w |

**The Code**



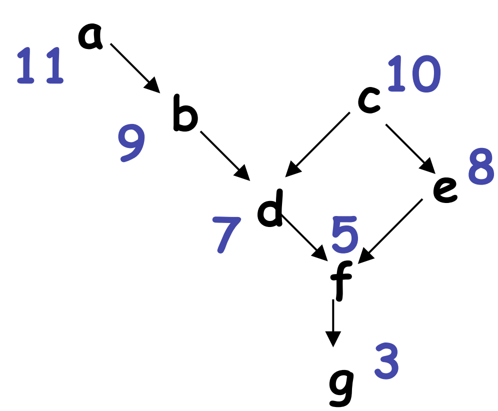

**The Dependence Graph**

# Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

| | | | |
|---|---|---|---|
| a: | loadAI | r0,@w | ⇒ r1 |
| c: | loadAI | r0,@x | ⇒ r2 |
| b: | add | r1,r1 | ⇒ r1 |
| d: | mult | r1,r2 | ⇒ r1 |
| e: | loadAI | r0,@z | ⇒ r2 |
| f: | mult | r1,r2 | ⇒ r1 |
| g: | storeAI | r1 | ⇒ r0,@w |

**The Code**




**The Dependence Graph**

# Phase IV

1. Compute the inheritance graph
2. Assign tags to all classes in depth-first order
3. Determine the layout of attributes, temporaries, and dispatch tables for each class
4. Generate code for global data: constants, dispatch tables
5. Generate code for each feature

## Revised Phase IV

1. ~~Compute the inheritance graph~~
2. ~~Assign tags to all classes in depth-first order~~
3. Determine the layout of attributes, temporaries, and ~~dispatch tables for each class~~
4. Generate code for global data: constants, ~~dispatch tables~~
5. Generate code for each feature

**Also forget about Garbage Collection.**

**Code will be given to you!**
**Do a checkout again Friday afternoon.**

## Revised Phase IV

1. ~~Compute the inheritance graph~~
2. ~~Assign tags to all classes in depth-first order~~
3. Determine the layout of attributes, temporaries, and ~~dispatch tables for each class~~
4. Generate code for global data: constants, ~~dispatch tables~~
5. Generate code for each feature

**Also forget about Garbage Collection.**

**Code will be given to you!**
**Do a checkout again Friday afternoon.**

# Phase IV Notes

1. Need to create space on stack for all local variables; this is the simplest register allocation scheme
2. A bunch of free temporary registers are available (see SPIM manual)
3. Need to write code for cgen for all tree nodes. E.g., dispatch tables (if function call is in expression) is given to you.
4. Again, look at MIPS code generated. Try to emulate what is generated!

REVERSE ENGINEERING IS THE KEY!