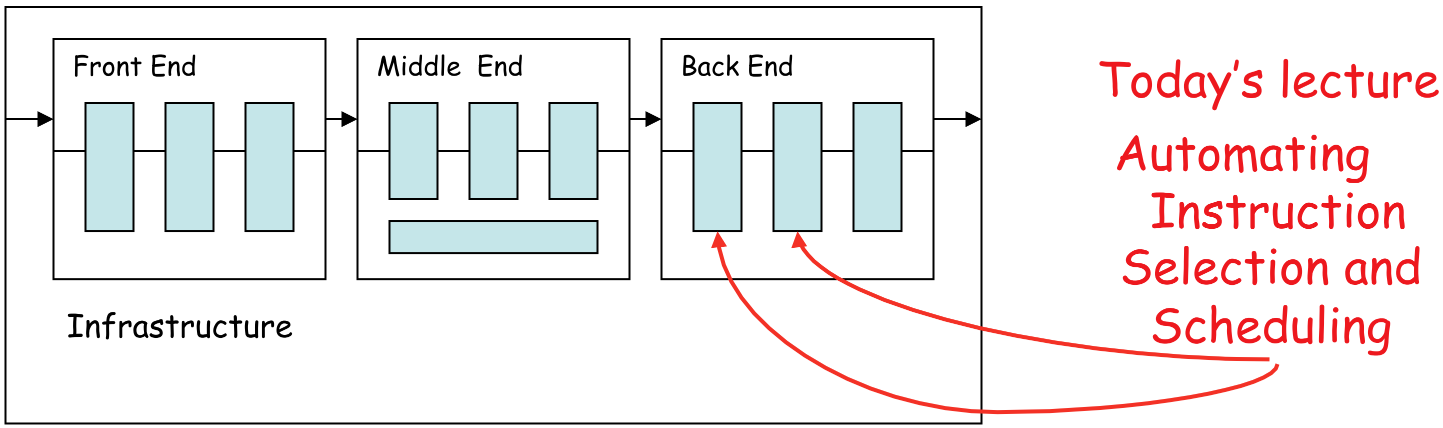
# Instruction Selection and Scheduling

# The Problem

Writing a compiler is a lot of work

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




- 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 … |

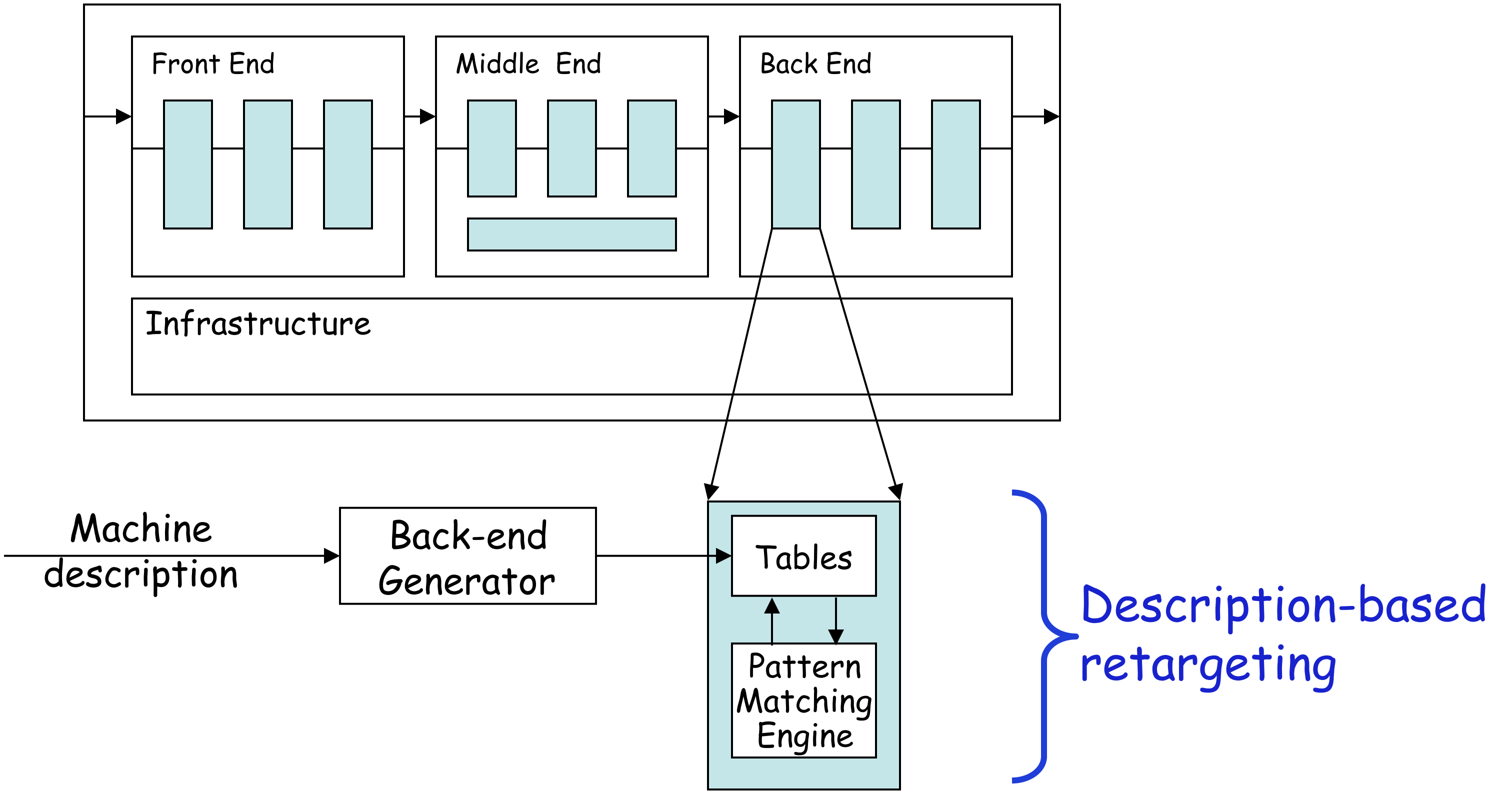
# The Problem

Modern computers (still) have many ways to do anything

- 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




Machine description can help with scheduling & allocation

# The Big Picture

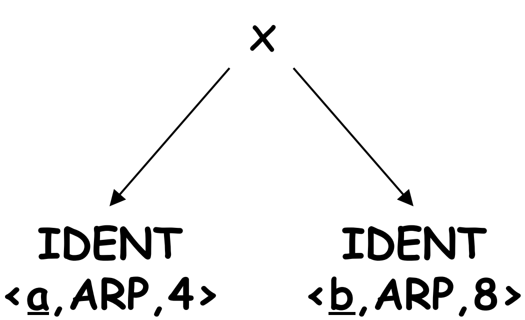
Need pattern matching techniques

- 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

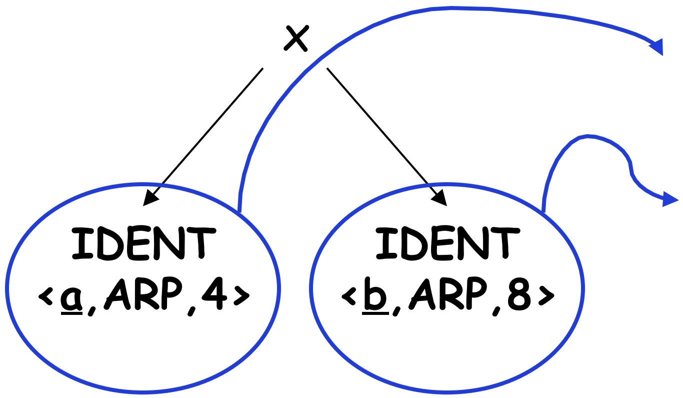
Need pattern matching techniques

- 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 | $r_5, r_6$ | $\Rightarrow r_7$ |

## 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 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

## Peephole Matching

- Basic idea
- Compiler can discover local improvements locally
  - → Look at a small set of adjacent operations
  - → Move a "peephole" over code & search for improvement
- Classic example: store followed by load

| Original code | | | | Improved code | | | |
|---|---|---|---|---|---|---|---|
| storeAI | $r_1$ | $\Rightarrow$ | $r_{arp}, 8$ | storeAI | $r_1$ | $\Rightarrow$ | $r_{arp}, 8$ |
| loadAI | $r_{arp}, 8$ | $\Rightarrow$ | $r_{15}$ | i2i | $r_1$ | $\Rightarrow$ | $r_{15}$ |

## Peephole Matching

- Basic idea
- Compiler can discover local improvements locally
  - → Look at a small set of adjacent operations
  - → Move a "peephole" over code & search for improvement
- Classic example: store followed by load
- Simple algebraic identities

**Original code**

addI $r_2, 0 \Rightarrow r_7$
mult $r_4, r_7 \Rightarrow r_{10}$

**Improved code**

mult $r_4, r_2 \Rightarrow r_{10}$

## Peephole Matching

- Basic idea
- Compiler can discover local improvements locally
  - → Look at a small set of adjacent operations
  - → Move a "peephole" over code & search for improvement
- Classic example: store followed by load
- Simple algebraic identities
- Jump to a jump

| Original code | Improved code |
| --- | --- |
| jumpI → $L_{10}$ | $L_{10}$: jumpI → $L_{11}$ |
| $L_{10}$: jumpI → $L_{11}$ | |

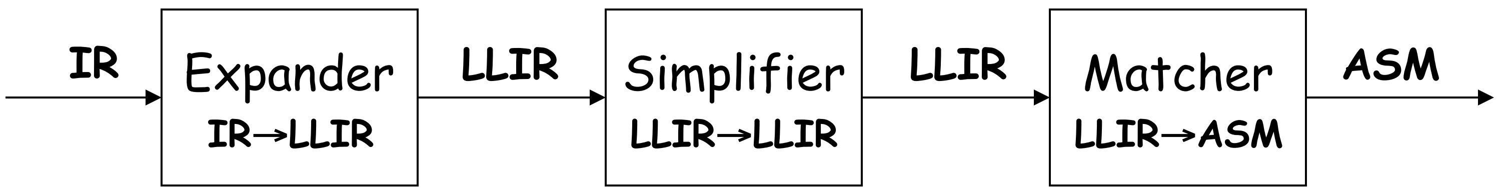
# Peephole Matching

Implementing it

- Early systems used limited set of hand-coded patterns
- Window size ensured quick processing

Modern peephole instruction selectors

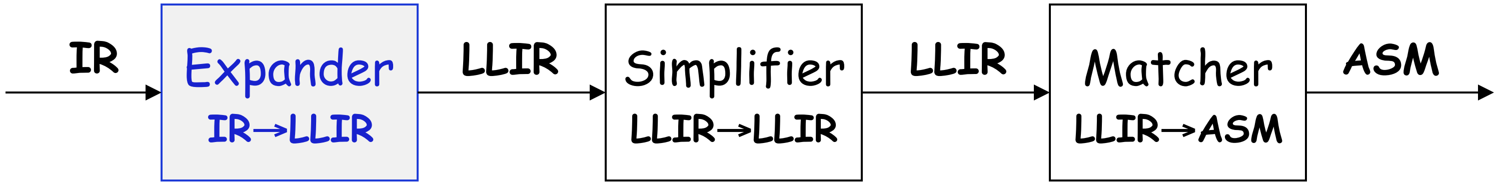
- Break problem into three tasks



# Peephole Matching

Expander

- Turns IR code into a low-level IR (LLIR)
- Operation-by-operation, template-driven rewriting
- Significant, albeit constant, expansion of size



# Peephole Matching

Simplifier

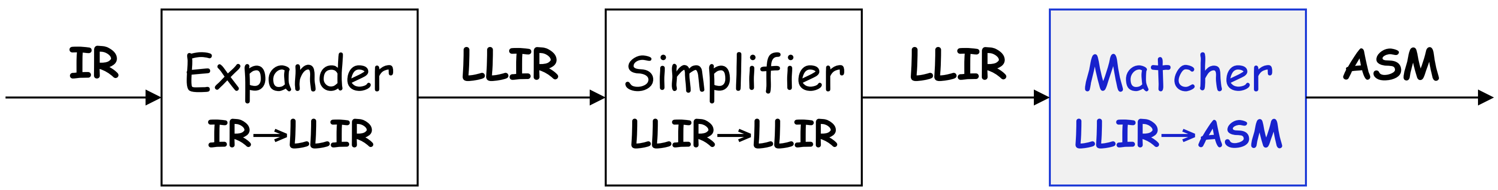
- Looks at LLIR through window and rewrites is
- Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
- Performs local optimization within window


- This is the heart of the peephole system
  - → Benefit of peephole optimization shows up in this step

# Peephole Matching

Matcher

- Compares simplified LLIR against a library of patterns
- Picks low-cost pattern that captures effects
- Generates the assembly code output



# Example

**Original IR Code**

| OP | $Arg_1$ | $Arg_2$ | Result |
|---|---|---|---|
| mult | 2 | y | $t_1$ |
| sub | x | $t_1$ | w |

$t_1 = r_{14}$
$w = r_{20}$

**Expand**

**LLIR Code**

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_{arp} + r_{11}$
$r_{13} \leftarrow MEM(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_{arp} + r_{15}$
$r_{17} \leftarrow MEM(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_{arp} + r_{19}$
$MEM(r_{20}) \leftarrow r_{18}$

# Example

Original IR Code

| OP | $Arg_1$ | $Arg_2$ | Result |
|---|---|---|---|
| mult | 2 | y | $t_1$ |
| sub | x | $t_1$ | w |

Expand

$t_1 = r_{14}$
$w = r_{20}$

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_{arp} + r_{11}$
$r_{13} \leftarrow MEM(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_{arp} + r_{15}$
$r_{17} \leftarrow MEM(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_{arp} + r_{19}$
$MEM(r_{20}) \leftarrow r_{18}$

# Example

Original IR Code

| OP | $Arg_1$ | $Arg_2$ | Result |
|---|---|---|---|
| mult | 2 | y | $t_1$ |
| sub | x | $t_1$ | w |

Expand

$t_1 = r_{14}$
$w = r_{20}$

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_{arp} + r_{11}$
$r_{13} \leftarrow MEM(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_{arp} + r_{15}$
$r_{17} \leftarrow MEM(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_{arp} + r_{19}$
$MEM(r_{20}) \leftarrow r_{18}$

# Example

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_{arp} + r_{11}$
$r_{13} \leftarrow MEM(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_{arp} + r_{15}$
$r_{17} \leftarrow MEM(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_{arp} + r_{19}$
$MEM(r_{20}) \leftarrow r_{18}$

Simplify ⇒

LLIR Code

$r_{13} \leftarrow MEM(r_{arp} + @y)$
$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow MEM(r_{arp} + @x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$MEM(r_{arp} + @w) \leftarrow r_{18}$

# Example

**LLIR Code**

$r_{13} \leftarrow \text{MEM}(r_{arp} + @y)$
$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow \text{MEM}(r_{arp} + @x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$\text{MEM}(r_{arp} + @w) \leftarrow r_{18}$

**Match**

**ILOC (Assembly) Code**

| | | |
|---|---|---|
| loadAI | $r_{arp}, @y$ | $\Rightarrow r_{13}$ |
| multI | $2 \times r_{13}$ | $\Rightarrow r_{14}$ |
| loadAI | $r_{arp}, @x$ | $\Rightarrow r_{17}$ |
| sub | $r_{17} - r_{14}$ | $\Rightarrow r_{18}$ |
| storeAI | $r_{18}$ | $\Rightarrow r_{arp}, @w$ |

- Introduced all memory operations & temporary names
- Turned out pretty good code

# Making It All Work

Details

- LLIR is largely machine independent
- Target machine described as LLIR → ASM pattern
- Actual pattern matching
  - → Use a hand-coded pattern matcher (gcc)
- Several important compilers use this technology
- It seems to produce good portable instruction selectors

Key strength appears to be late low-level optimization

# 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

# What Makes Code Run Fast?

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

# What Makes Code Run Fast?

## 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
- Branch costs vary with path taken
- Scheduler should hide the latencies

# Example: w ← w * 2 * x * y * z

Load causes add to stall

| 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**

# Example: w ← 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 | | |

Load causes mult to stall

**2 registers, 20 cycles**

# Example: w ← 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 | | |

Load causes mult to stall

**2 registers, 20 cycles**

# 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 | | |

Load causes mult to stall

**2 registers, 20 cycles**

# Example: w ← w * 2 * x * y * z

Schedule loads early

| 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 to improve some metric is called instruction scheduling**

# Example: w ← w * 2 * x * y * z

*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 | | |

Load takes 3 cycles

Add can execute, does not stall

3 registers, 13 cycles

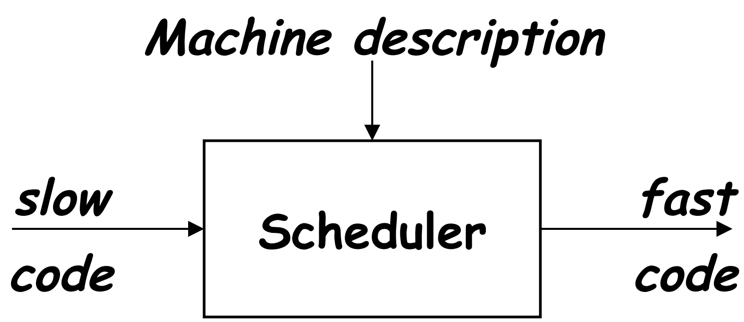
**Reordering operations to improve some metric is called instruction scheduling**

# Instruction Scheduling (Engineer's View)

The Problem

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

The Concept




**The task**

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

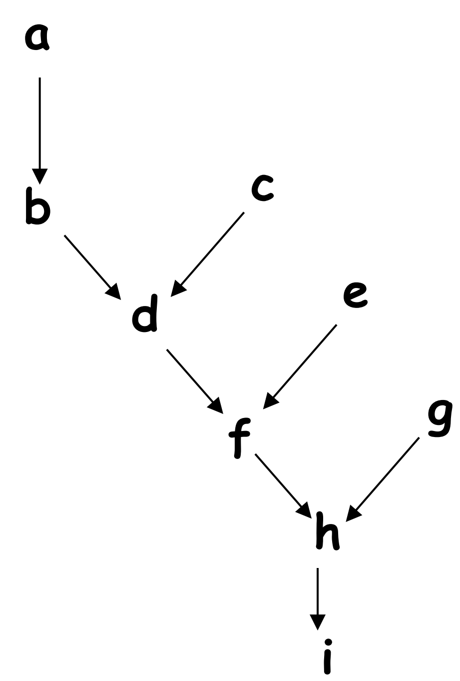
# Instruction Scheduling (The Abstract View)

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

- Nodes $n \in G$ are operations
- An edge $e = (n_1,n_2) \in G$ if $n_2$ uses the result of $n_1$

| | | | |
|---|---|---|---|
| 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?)

- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
  - → Increases register pressure
- Operands can have multiple predecessors

Together, these issues make scheduling *hard*

(NP-complete)

## Instruction Scheduling (Local list scheduling)

1. Build a dependence graph, *P*
2. Compute a <u>*priority function*</u> 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

```
Cycle ← 1
Active ← Ø
Ready ← roots of P
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 successor s of op in P
        if (s is ready) then
          Ready ← Ready ∪ s
```

Initialize and set roots as ready to schedule.

# Local List Scheduling

```
Cycle ← 1
Active ← Ø
Ready ← roots of P

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 successor s of op in P
            if (s is ready) then
               Ready ← Ready ∪ s
```

Loop while ready queue is not empty. Remove an op from ready queue to schedule (move to active)

# Local List Scheduling

```
Cycle ← 1
Active ← Ø
Ready ← roots of P

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 successor s of op in P
            if (s is ready) then
               Ready ← Ready ∪ s
```

*Simulating architecture; increment cycle count*

# Local List Scheduling

```
Cycle ← 1
Active ← Ø
Ready ← roots of P

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 successor s of op in P
            if (s is ready) then
               Ready ← Ready ∪ s
```

Check if operations in Active queue should be removed based on cycle count.

# Local List Scheduling

```
Cycle ← 1
Active ← Ø
Ready ← roots of P

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 successor s of op in P
            if (s is ready) then
               Ready ← Ready ∪ s
```

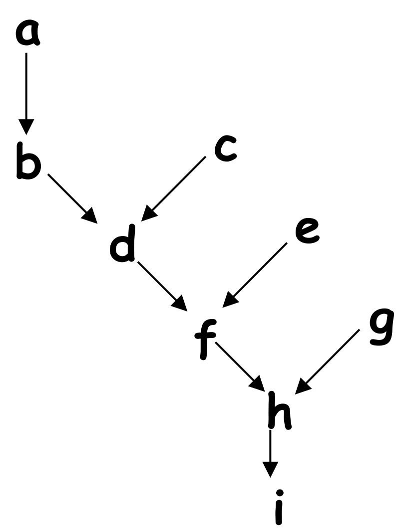
**If successor's operands are ready, put it on Ready queue**

# 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,@y | ⇒ r2 |
| f: | mult | r1,r2 | ⇒ r1 |
| g: | loadAI | r0,@z | ⇒ r2 |
| h: | mult | r1,r2 | ⇒ r1 |
| i: | storeAI | r1 | ⇒ r0,@w |

**The Code**



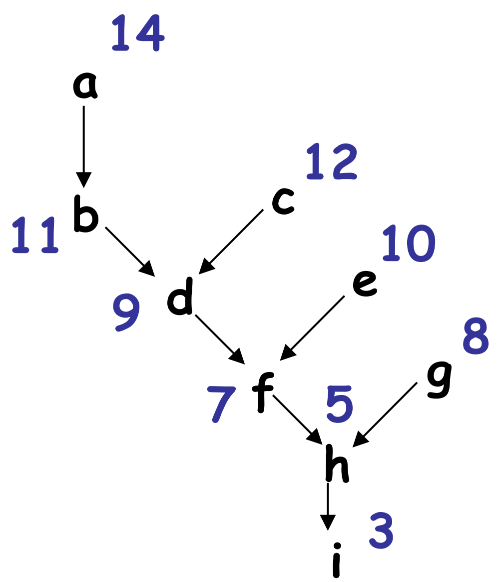

**The Dependence Graph**

# 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,@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**

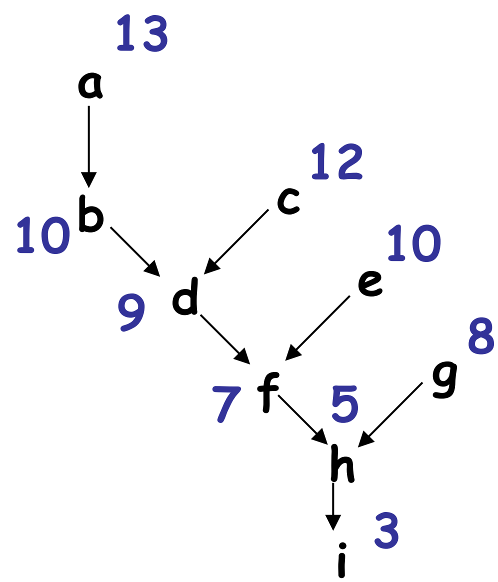
# Scheduling Example

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

New register name used

```
1)  a:  loadAI   r0,@w   ⇒ r1
2)  c:  loadAI   r0,@x   ⇒ r2
3)  e:  loadAI   r0,@y   ⇒ r3
4)  b:  add      r1,r1   ⇒ r1
5)  d:  mult     r1,r2   ⇒ r1
6)  g:  loadAI   r0,@z   ⇒ r2
7)  f:  mult     r1,r3   ⇒ r1
9)  h:  mult     r1,r2   ⇒ r1
11) i:  storeAI  r1      ⇒ r0,@w
```

The Code




The Dependence Graph

# More List Scheduling

List scheduling breaks down into two distinct classes

| Forward list scheduling | Backward list scheduling |
| --- | --- |
| • Start with available operations | • Start with no successors |
| • Work forward in time | • Work backward in time |
| • Ready $\Rightarrow$ all operands available | • Ready $\Rightarrow$ result >= all uses |