

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* (and has been since the 60's)

Assumed latencies (conservative)

Operation	Cycles
load	3
store	3
loadl	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
- Branches typically have delay slots
 - > Fill slots with unrelated operations
 - > Percolates branch upward
- Scheduler should hide the latencies

Lab 3 will build a local scheduler

CS430

2

Example

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

Simple schedule

```

1 loadAl r0,@w ⇒ r1
4 add r1,r1 ⇒ r1
5 loadAl r0,@x ⇒ r2
8 mult r1,r2 ⇒ r1
9 loadAl r0,@y ⇒ r2
12 mult r1,r2 ⇒ r1
13 loadAl r0,@z ⇒ r2
16 mult r1,r2 ⇒ r1
18 storeAl r1 ⇒ r0,@w
21 r1 is free
    
```

2 registers, 20 cycles

Schedule loads early

```

1 loadAl r0,@w ⇒ r1
2 loadAl r0,@x ⇒ r2
3 loadAl r0,@y ⇒ r3
4 add r1,r1 ⇒ r1
5 mult r1,r2 ⇒ r1
6 loadAl r0,@z ⇒ r2
7 mult r1,r3 ⇒ r1
9 mult r1,r2 ⇒ r1
11 storeAl r1 ⇒ r0,@w
14 r1 is free
    
```

3 registers, 13 cycles

Reordering operations for speed is called instruction scheduling

CS430

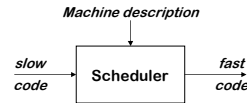
3

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

CS430

4

Instruction Scheduling (The Abstract View)

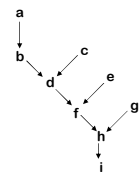
To capture properties of the code, build a **precedence graph** \mathcal{G}

- Nodes $n \in \mathcal{G}$ are operations with *type*(n) and *delay*(n)
- An edge $e = (n_1, n_2) \in \mathcal{G}$ if & only if n_2 uses the result of n_1

```

a: loadAl r0,@w ⇒ r1
b: add r1,r1 ⇒ r1
c: loadAl r0,@x ⇒ r2
d: mult r1,r2 ⇒ r1
e: loadAl r0,@y ⇒ r2
f: mult r1,r2 ⇒ r1
g: loadAl r0,@z ⇒ r2
h: mult r1,r2 ⇒ r1
i: storeAl r1 ⇒ r0,@w
    
```

The Code



The Precedence Graph

CS430

5

Instruction Scheduling (Definitions)

A **correct schedule** S maps each $n \in \mathcal{N}$ into a non-negative integer representing its cycle number, and

1. $S(n) \geq 0$, for all $n \in \mathcal{N}$, obviously
2. If $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$
3. For each type t , there are no more operations of type t in any cycle than the target machine can issue

The **length** of a schedule S , denoted $L(S)$, is

$$L(S) = \max_{n \in \mathcal{N}} (S(n) + \text{delay}(n))$$

The goal is to find the shortest possible correct schedule.

S is **time-optimal** if $L(S) \leq L(S_i)$, for all other schedules S_i . A schedule might also be optimal in terms of registers, power, or space....

CS430

6

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

CS430

7

Instruction Scheduling

The big picture

1. Build a precedence 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

CS430

8

Local List Scheduling

```

Cycle ← 1
Ready ← leaves of P
Active ← ∅

while (Ready ∪ Active ≠ ∅)
  if (Ready ≠ ∅) 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
    
```

Annotations:

- Removal in priority order (points to the 'remove an op from Ready' line)
- op has completed execution (points to the 'Active ← Active ∪ op' line)
- If successor's operands are ready, put it on Ready (points to the 'Ready ← Ready ∪ s' line)

CS430

9

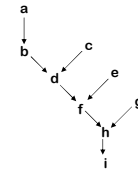
Scheduling Example

1. Build the precedence graph

```

a: loadAl r0,@w ⇒ r1
b: add r1,r1 ⇒ r1
c: loadAl r0,@x ⇒ r2
d: mult r1,r2 ⇒ r1
e: loadAl r0,@y ⇒ r2
f: mult r1,r2 ⇒ r1
g: loadAl r0,@z ⇒ r2
h: mult r1,r2 ⇒ r1
i: storeAl r1 ⇒ r0,@w
    
```

The Code



The Precedence Graph

CS430

10

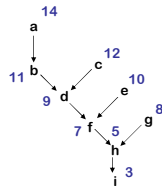
Scheduling Example

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

```

a: loadAl r0,@w ⇒ r1
b: add r1,r1 ⇒ r1
c: loadAl r0,@x ⇒ r2
d: mult r1,r2 ⇒ r1
e: loadAl r0,@y ⇒ r2
f: mult r1,r2 ⇒ r1
g: loadAl r0,@z ⇒ r2
h: mult r1,r2 ⇒ r1
i: storeAl r1 ⇒ r0,@w
    
```

The Code



The Precedence Graph

CS430

11

Scheduling Example

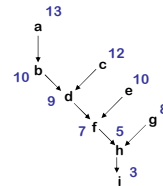
1. Build the precedence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

```

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

The Code

New register name used



The Precedence Graph

CS430

12

Detailed Scheduling Algorithm I

Idea: Keep a collection of worklists $W[c]$, one per cycle
 → We need $MaxC = \text{max delay} + 1$ such worklists

Code:

```

for each  $n \in N$  do begin count[n] := 0; earliest[n] = 0 end
for each  $(n1, n2) \in E$  do begin
    count[n2] := count[n2] + 1;
    successors[n1] := successors[n1]  $\cup$  {n2};
end
for  $i := 0$  to  $MaxC - 1$  do  $W[i] := \emptyset$ ;
Wcount := 0;
for each  $n \in N$  do
    if count[n] = 0 then begin
         $W[0] := W[0] \cup \{n\}$ ; Wcount := Wcount + 1;
    end
end
 $c := 0$ ; //  $c$  is the cycle number
 $cW := 0$ ; //  $cW$  is the number of the worklist for cycle  $c$ 
instr[c] :=  $\emptyset$ ;
    
```

CS430

13

Detailed Scheduling Algorithm II

```

while Wcount > 0 do begin
    while  $W[cW] = \emptyset$  do begin
         $c := c + 1$ ; instr[c] :=  $\emptyset$ ;  $cW := \text{mod}(cW + 1, MaxC)$ ;
    end
    nextc :=  $\text{mod}(c + 1, MaxC)$ ;
    while  $W[cW] \neq \emptyset$  do begin
        Priority → select and remove an arbitrary instruction  $x$  from  $W[cW]$ ;
        if  $\exists$  free issue units of type(x) on cycle  $c$  then begin
            instr[c] := instr[c]  $\cup$  {x}; Wcount := Wcount - 1;
            for each  $y \in \text{successors}[x]$  do begin
                count[y] := count[y] - 1;
                earliest[y] :=  $\max(\text{earliest}[y], c + \text{delay}(x))$ ;
                if count[y] = 0 then begin
                    loc :=  $\text{mod}(\text{earliest}[y], MaxC)$ ;
                     $W[loc] := W[loc] \cup \{y\}$ ; Wcount := Wcount + 1;
                end
            end
        end
        else  $W[\text{nextc}] := W[\text{nextc}] \cup \{x\}$ ;
    end
end
    
```

CS430

More List Scheduling

List scheduling breaks down into two distinct classes

Forward list scheduling	Backward list scheduling
<ul style="list-style-type: none"> Start with available operations Work forward in time Ready \Rightarrow all operands available 	<ul style="list-style-type: none"> Start with no successors Work backward in time Ready \Rightarrow latency covers uses

Variations on list scheduling

- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in precedence graph (minimize registers)
- Breadth first in precedence graph (minimize interlocks)
- Prefer operation with most successors

CS430

15