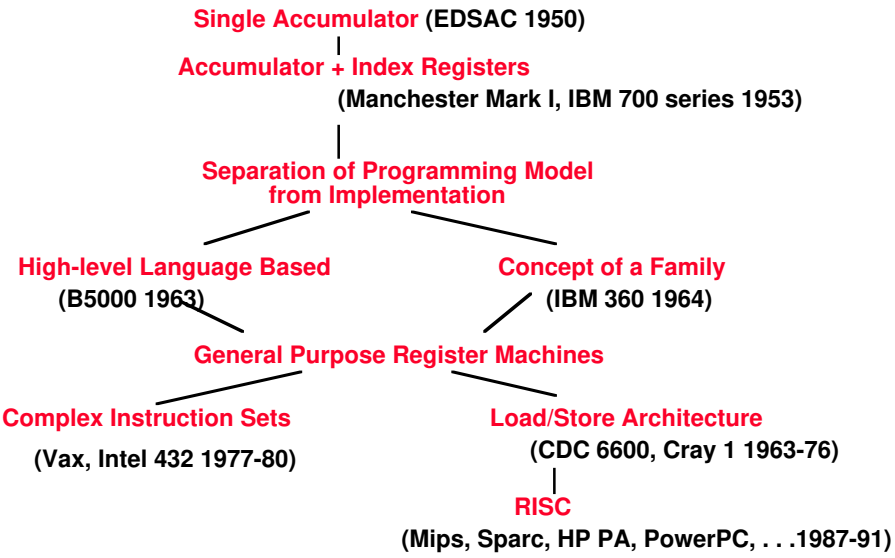
Review From Last Time: Instruction Set Evolution



© Patterson 1994

Review: 80x86 v. DLX Instruction Counts

SPEC pgm	*x86*	*DLX*	*DLX÷86*
gcc	3,771,327,742	3,892,063,460	1.03
espresso	2,216,423,413	2,801,294,286	1.26
spice	15,257,026,309	16,965,928,788	1.11
nasa7	15,603,040,963	6,118,740,321	0.39

© Patterson 1994

Review From Last Time: Intel Summary

- Archeology: history of instruction design in a single product
  - Address size: 16 bit vs. 32-bit
  - Protection: Segmentation vs. paged
  - Temp. storage: accumulator vs. stack vs. registers
- "Golden Handcuffs" of binary compatibility affect design 20 years later, as Moore predicted
- Not too difficult to make faster, as Intel has shown
- HP/Intel announcement of common future instruction set by 2000 means end of 80x86???
- "Beauty is in the eye of the beholder"
  - At 50M/year sold, it is a beautiful business

© Patterson 1994

Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold

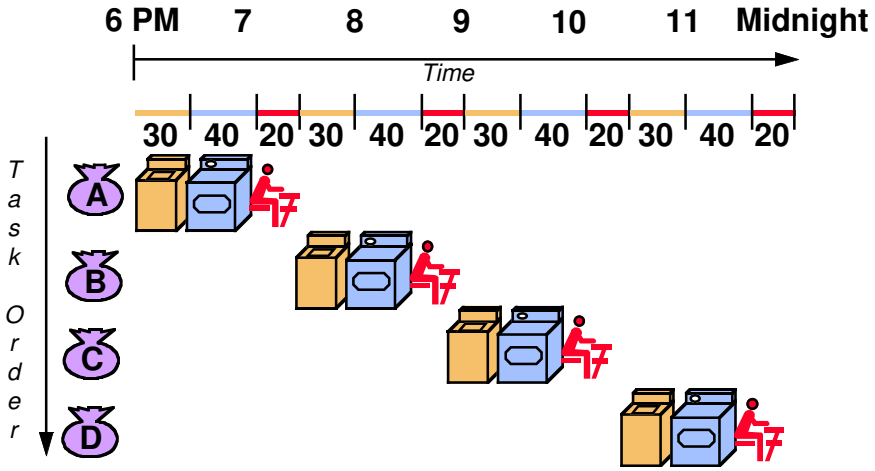
- Washer takes 30 minutes

- Dryer takes 40 minutes

- "Folder" takes 20 minutes

© Patterson 1994

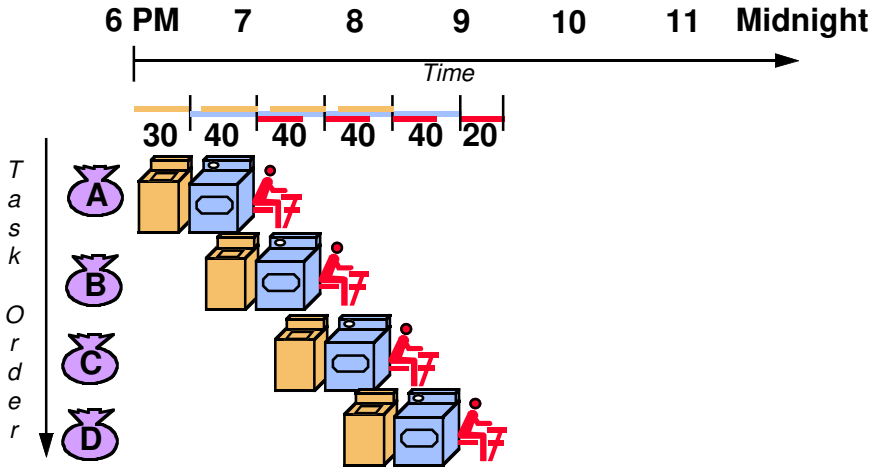
Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

© Patterson 1994

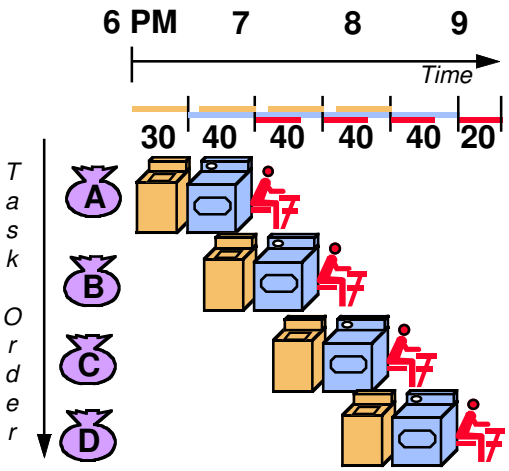
Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

© Patterson 1994

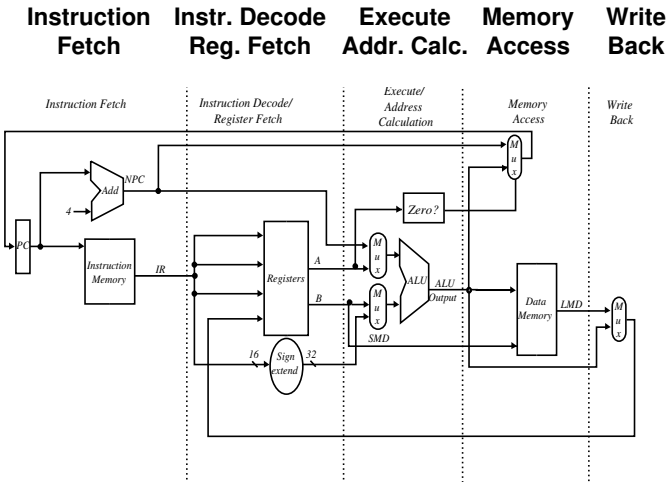
Pipelining Lessons



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

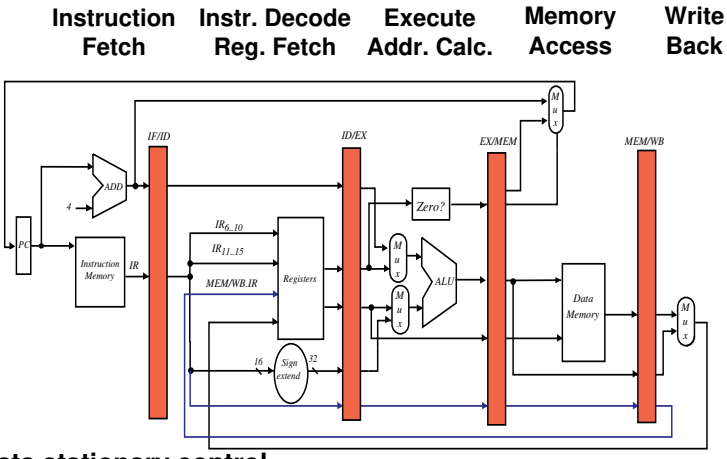
© Patterson 1994

5 Steps of DLX Datapath *(Figure 4.1, page 168, Beta Edition of CA:AQA,2/e)*



© Patterson 1994

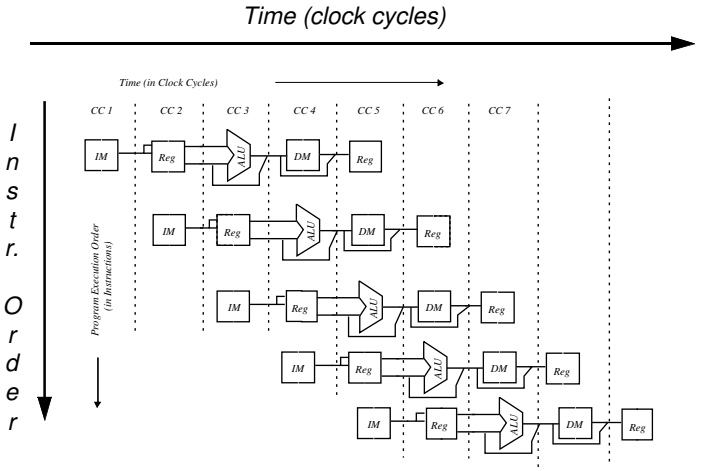
Pipelined DLX Datapath *(Figure 4.4, page 171 Beta Edition of CA:AQA,2/e)*



- **Data stationary control**
  - **–local decode for each instruction phase / pipeline stage**

© Patterson 1994

Visualizing Pipelining *(Figure 4.3, page 170 Beta Edition of CA:AQA,2/e)*



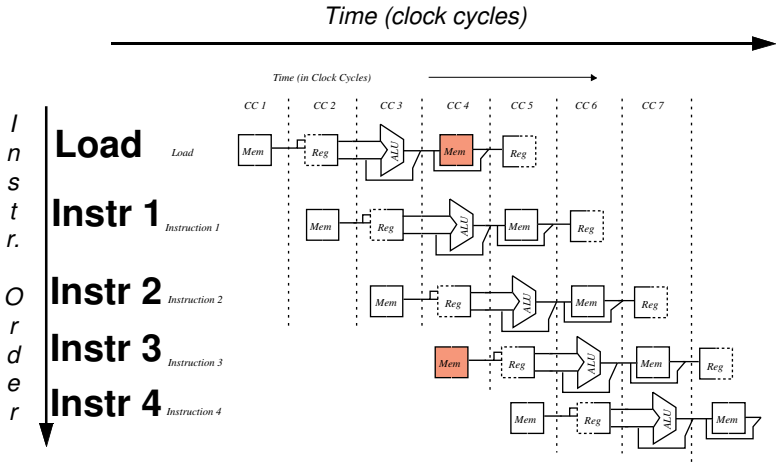
© Patterson 1994

Its not that easy for computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - structural hazards: HW cannot support this combination of instructions
  - data hazards: instruction depends on result of prior instruction still inthe pipeline
  - control hazards: pipelining of branches & other instructions that change the PC
- Common solution is to stall the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline

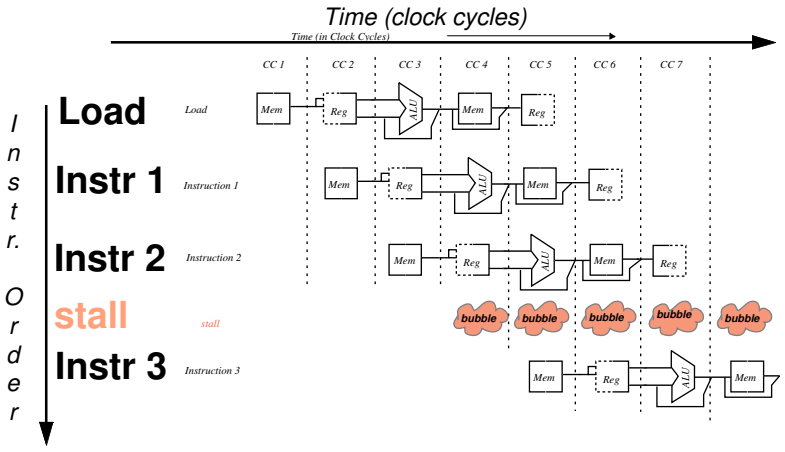
© Patterson 1994

1 Memory is Structural Hazard: *(Figure 4.7, page 178 Beta Edition)*



© Patterson 1994

1 Memory is Structural Hazard: *(Figure 4.8, page 179 Beta Edition)*



© Patterson 1994

Speed Up Equation for Pipelining

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\text{Ideal CPI} = \frac{\text{CPI unpipelined}}{\text{Pipeline depth}}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

© Patterson 1994

Example: Dual-port vs. single port

- Machine A: Dual ported memory
- Machine B: Single ported memory, 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$SpeedUp_A$ = Pipeline Depth/(1+ 0) x ($clock_{unpipe}/clock_{unpipe}$)
= Pipeline Depth

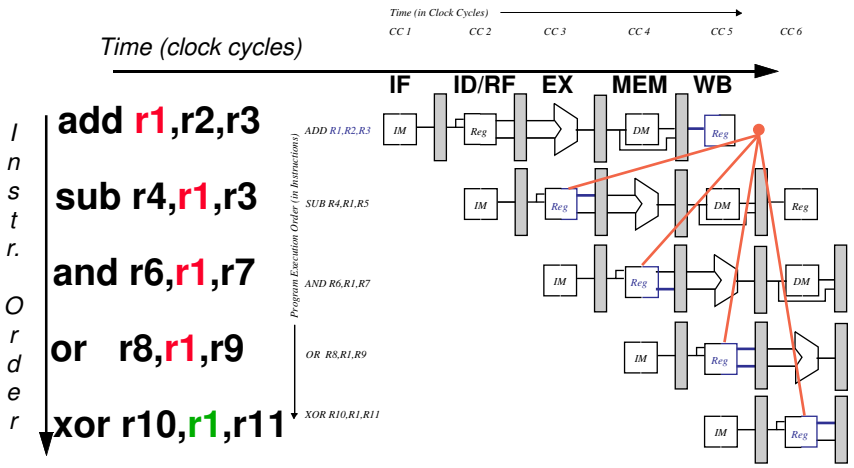
$SpeedUp_B$ = Pipeline Depth/(1+ 0.4 x 1)
x ($clock_{unpipe}/(clock_{unpipe}$ / 1.05)
= (Pipeline Depth/1.4) x 1.05
= 0.75 x Pipeline Depth

$SpeedUp_A$ / $SpeedUp_B$ = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33

- Machine A is 1.33 times faster

© Patterson 1994

Data Hazard on r1: *(Figure 4.10, page 182 Beta Edition)*



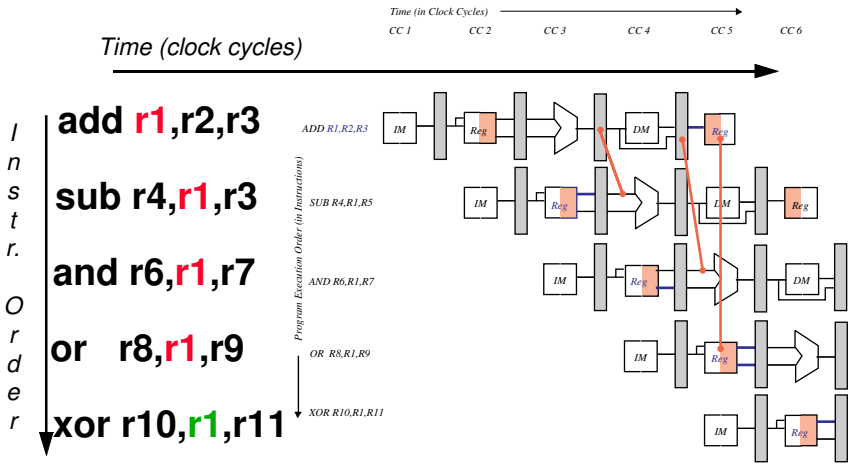
© Patterson 1994

3 Generic Data Hazards

- $Instr_I$ followed be $Instr_J$
- **Read After Write (RAW)**
  $Instr_J$ tries to read operand before $Instr_I$ writes it
- **Write After Read (WAR)**
  $Instr_J$ tries to write operand before $Instr_I$ reads it
- Can't happen in DLX 5 stage pipeline because
  all instructions take 5 stages,
  reads are always in stage 2, and
  writes are always in stage 5
- **Write After Write (WAW)**
  $Instr_J$ tries to write operand before $Instr_I$ writes it
  – Leaves wrong result ($Instr_I$ not $Instr_J$)
- Can't happen in DLX 5 stage pipeline because
  all instructions take 5 stages, and
  writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

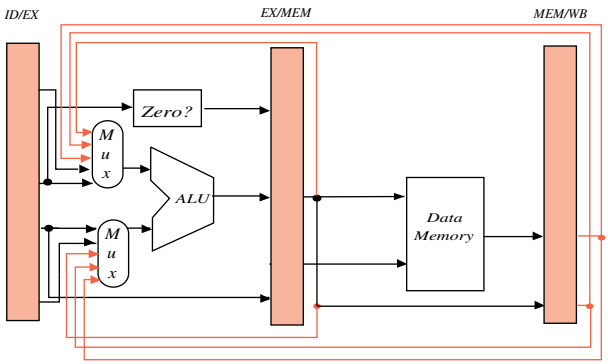
© Patterson 1994

Forwarding to Avoid Data Hazard: *(Figure 4.11, page 184 Beta)*



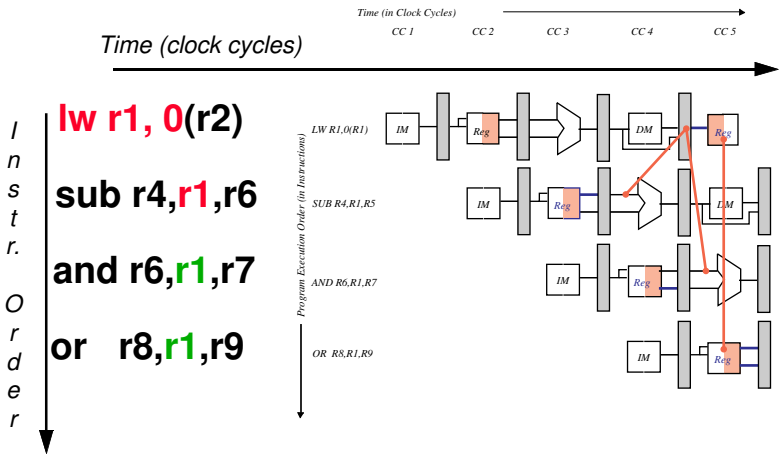
© Patterson 1994

HW Change for Forwarding: *(Figure 4.21, page 196, Beta Edition)*



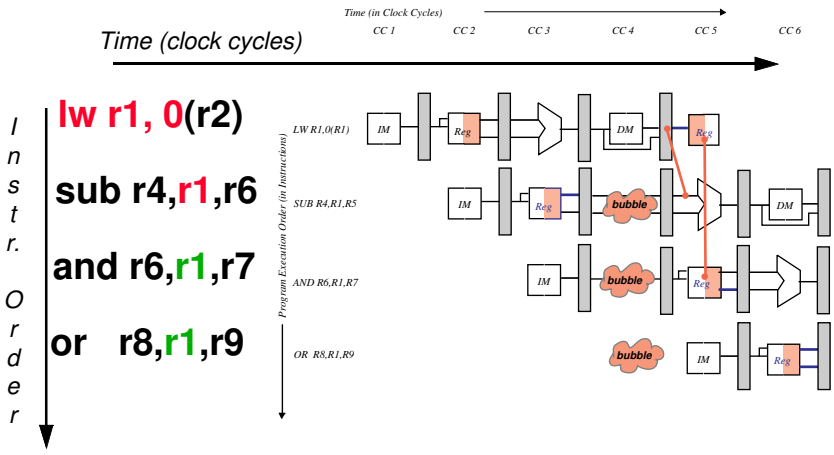
© Patterson 1994

Data Hazard even with Forwarding: *(Figure 4.13, page 188 Beta)*



© Patterson 1994

Data Hazard even with Forwarding: *(Figure 4.14, page 189 Beta)*



© Patterson 1994

Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

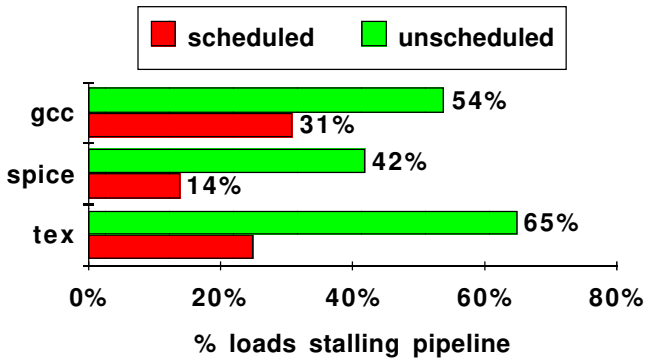
LW	Rb,b
LW	Rc,c
ADD	Ra,Rb,Rc
SW	a,Ra
LW	Re,e
LW	Rf,f
SUB	Rd,Re,Rf
SW	d,Rd

Fast code:

LW	Rb,b
LW	Rc,c
LW	Re,e
ADD	Ra,Rb,Rc
LW	Rf,f
SW	a,Ra
SUB	Rd,Re,Rf
SW	d,Rd

© Patterson 1994

Compiler Avoiding Load Stalls: *(Figure 4.17, page 192 Beta)*



© Patterson 1994

Pipelining Summary

- **Just overlap tasks, and easy if tasks are independent**
- **Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:**

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- **Hazards limit performance on computers:**
  - **structural: need more HW resources**
  - **data: need forwarding, compiler scheduling**
  - **control: discuss next time**

© Patterson 1994