
Pipelining

Dr. Soner Onder

CS 4431

Michigan Technological University

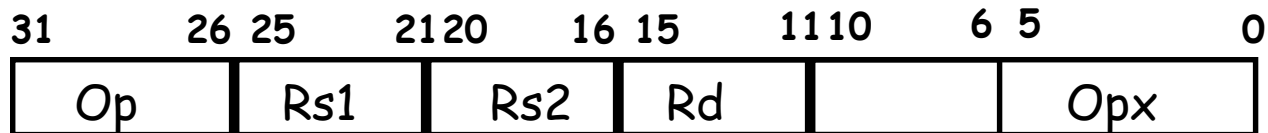
A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
base + displacement
 - no indirection
- Simple branch conditions
- Delayed branch

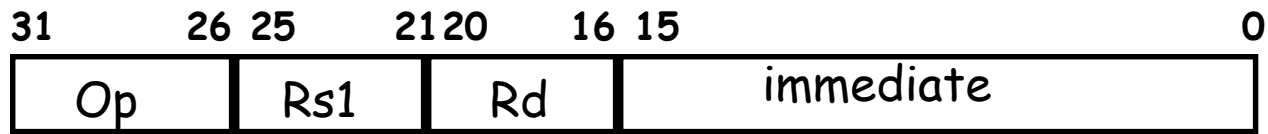
**see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3**

Example: MIPS (MIPS)

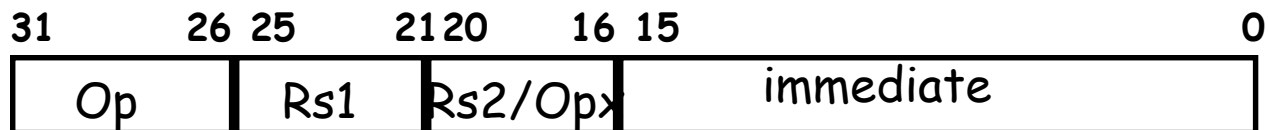
Register-Register



Register-Immediate



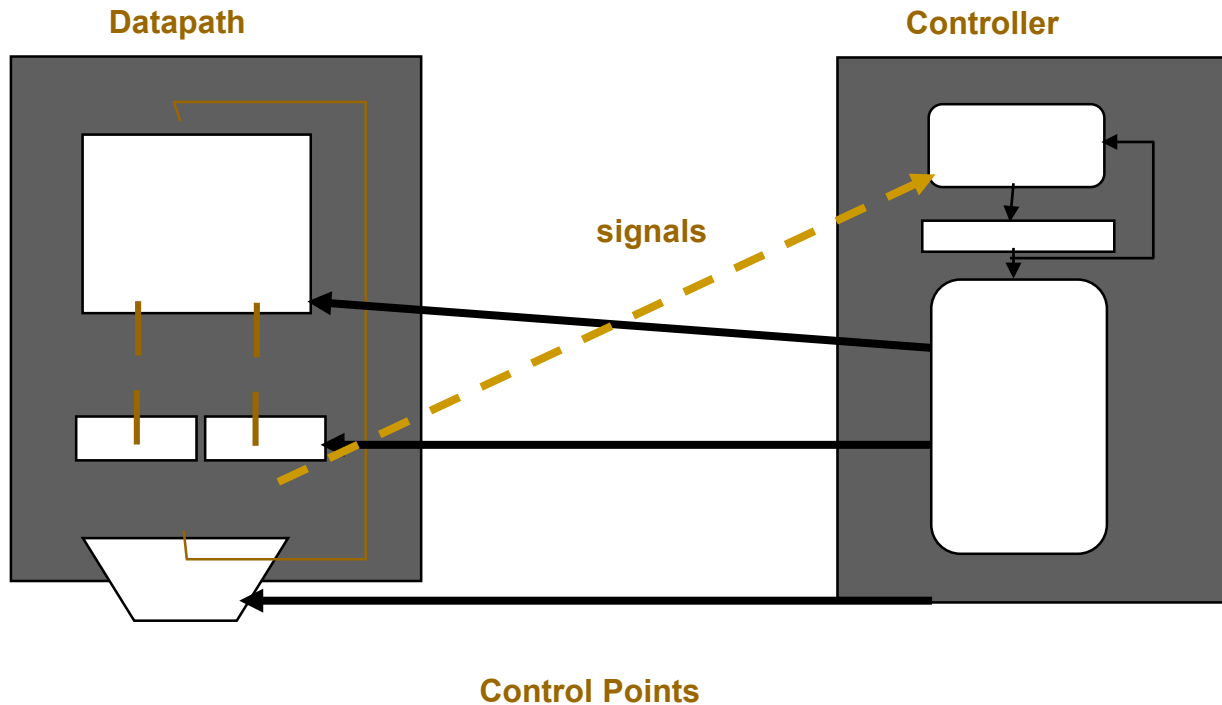
Branch



Jump / Call



Datapath vs Control



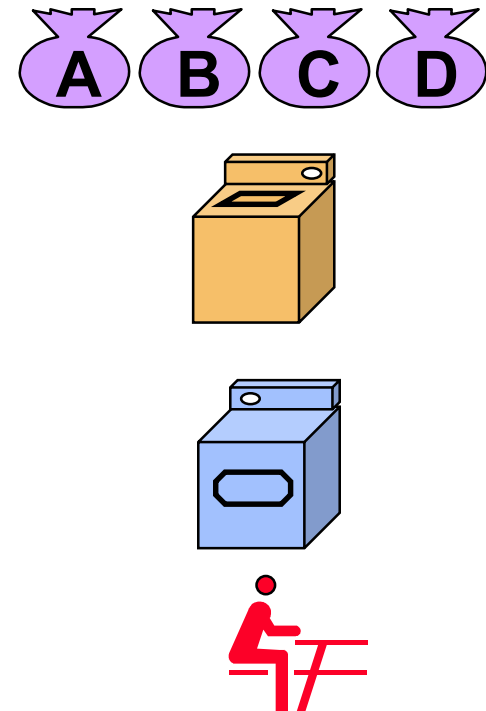
- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
 - Inputs are Control Points
 - Outputs are signals
- Controller: State machine to orchestrate operation on the data path
 - Based on desired function and signals

Approaching an ISA

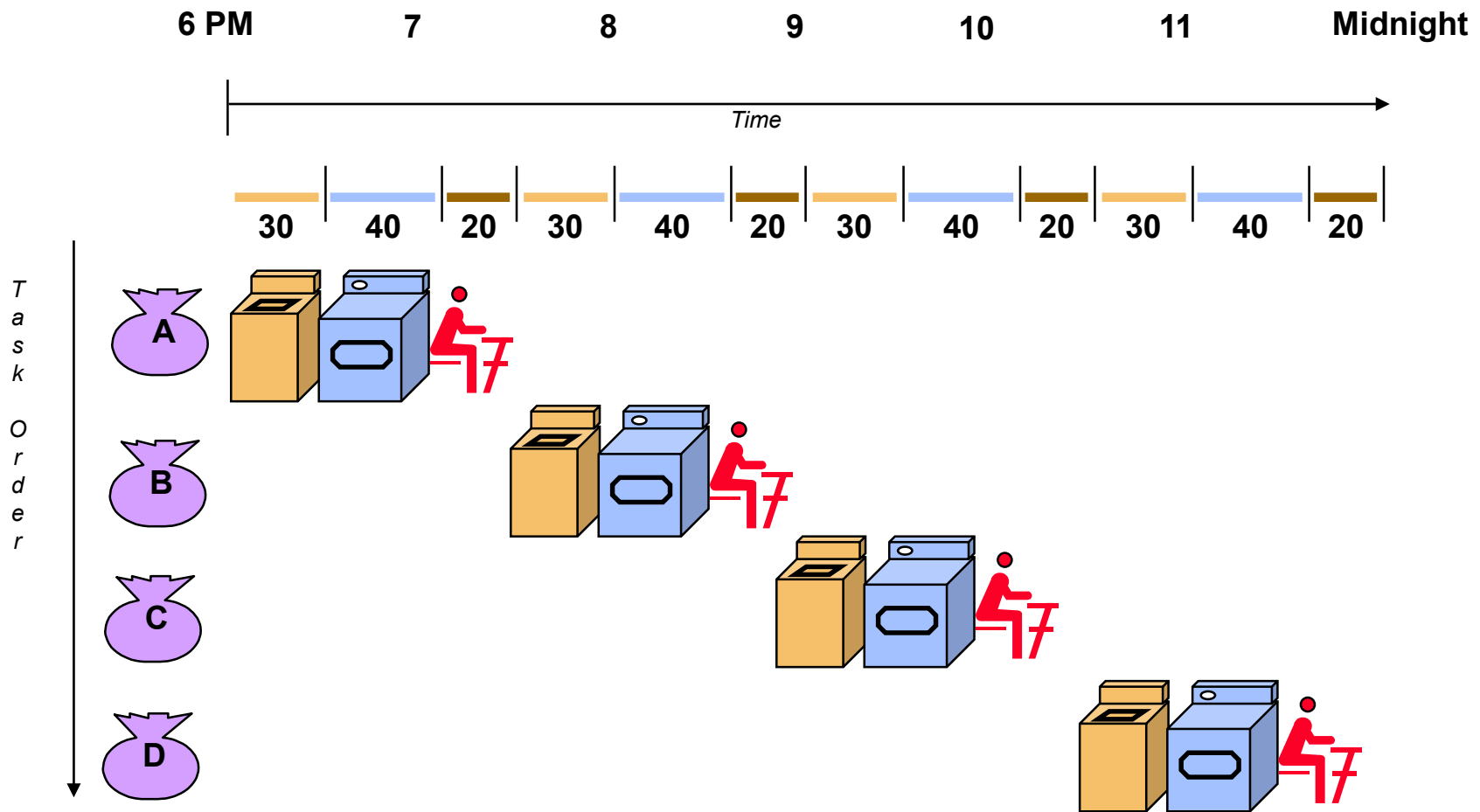
- Instruction Set Architecture
 - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on *architected registers* and memory
- Given technology constraints assemble adequate datapath
 - Architected storage mapped to actual storage
 - Function units to do all the required operations
 - Possible additional storage (eg. MAR, MBR, ...)
 - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller

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



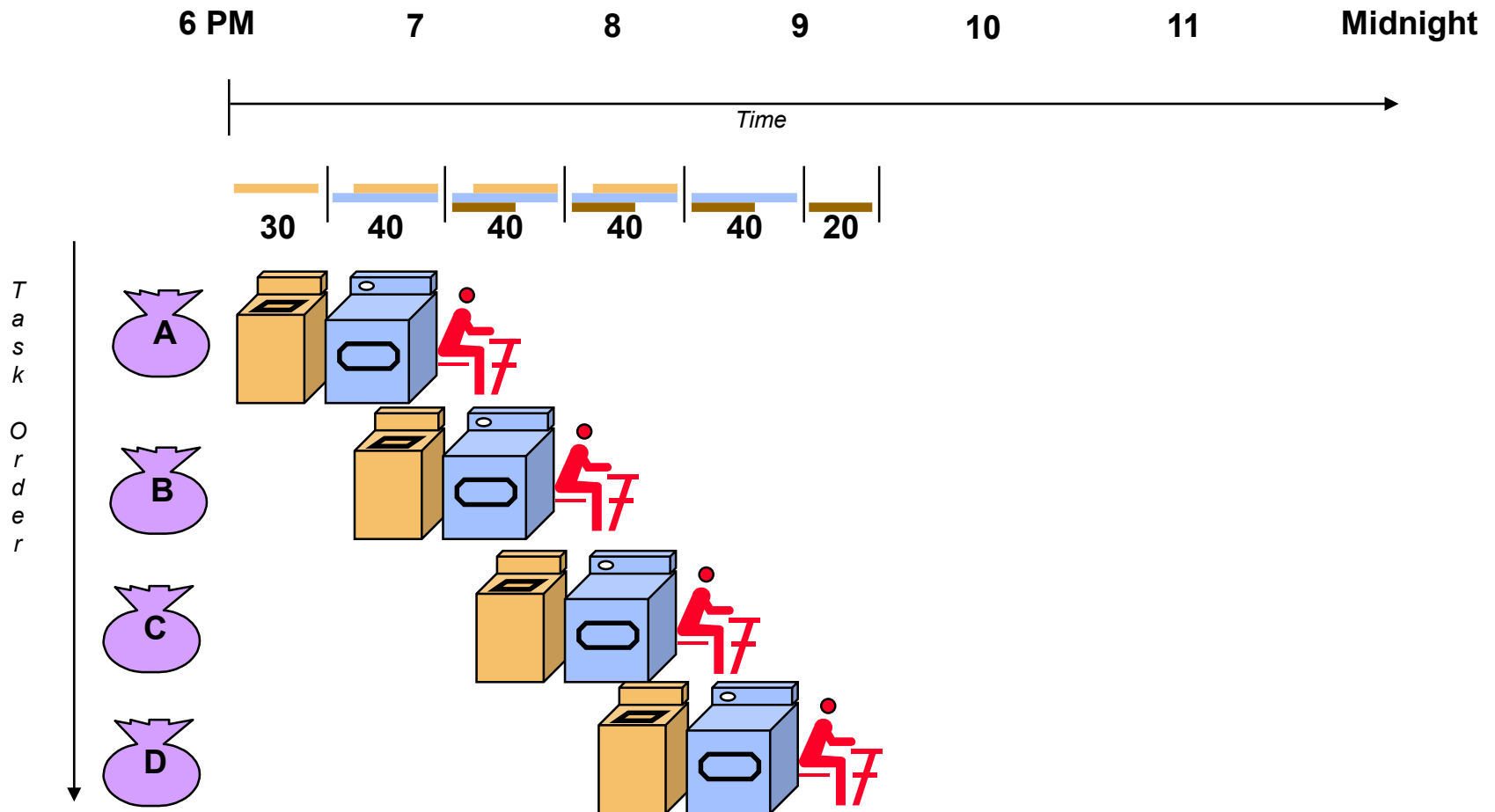
Sequential Laundry



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

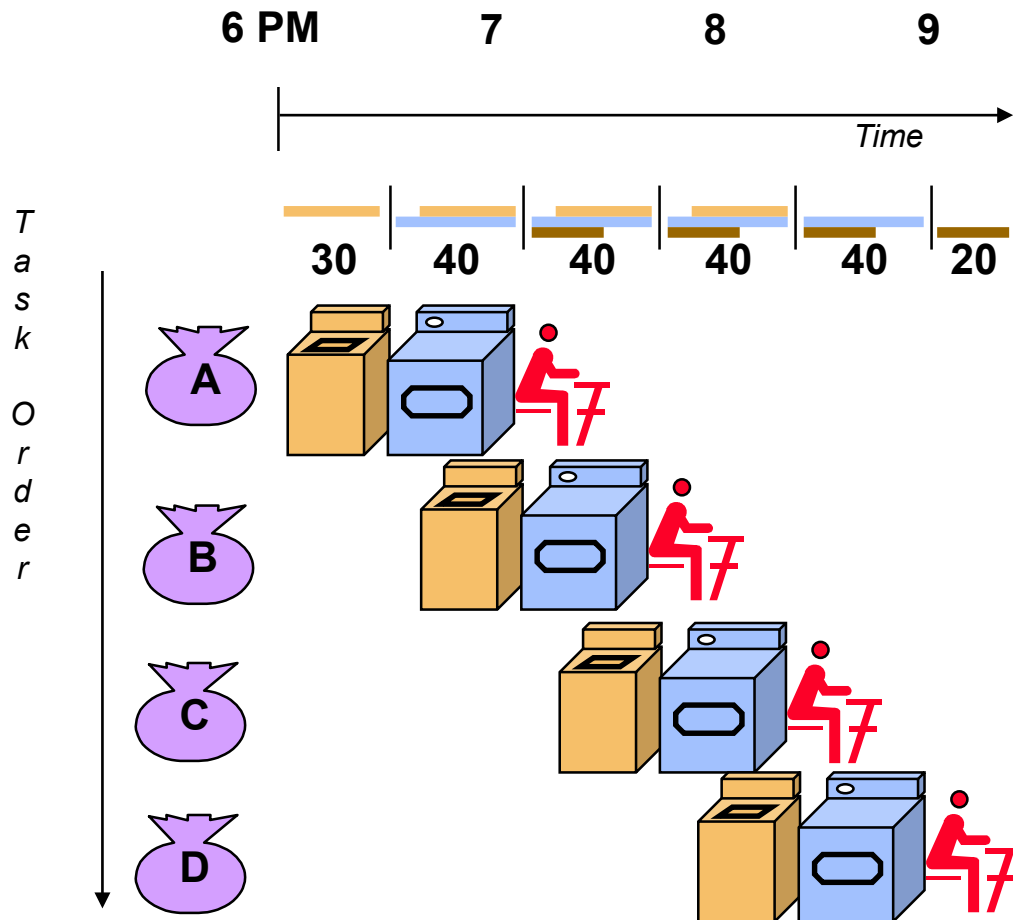
Pipelined Laundry

Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

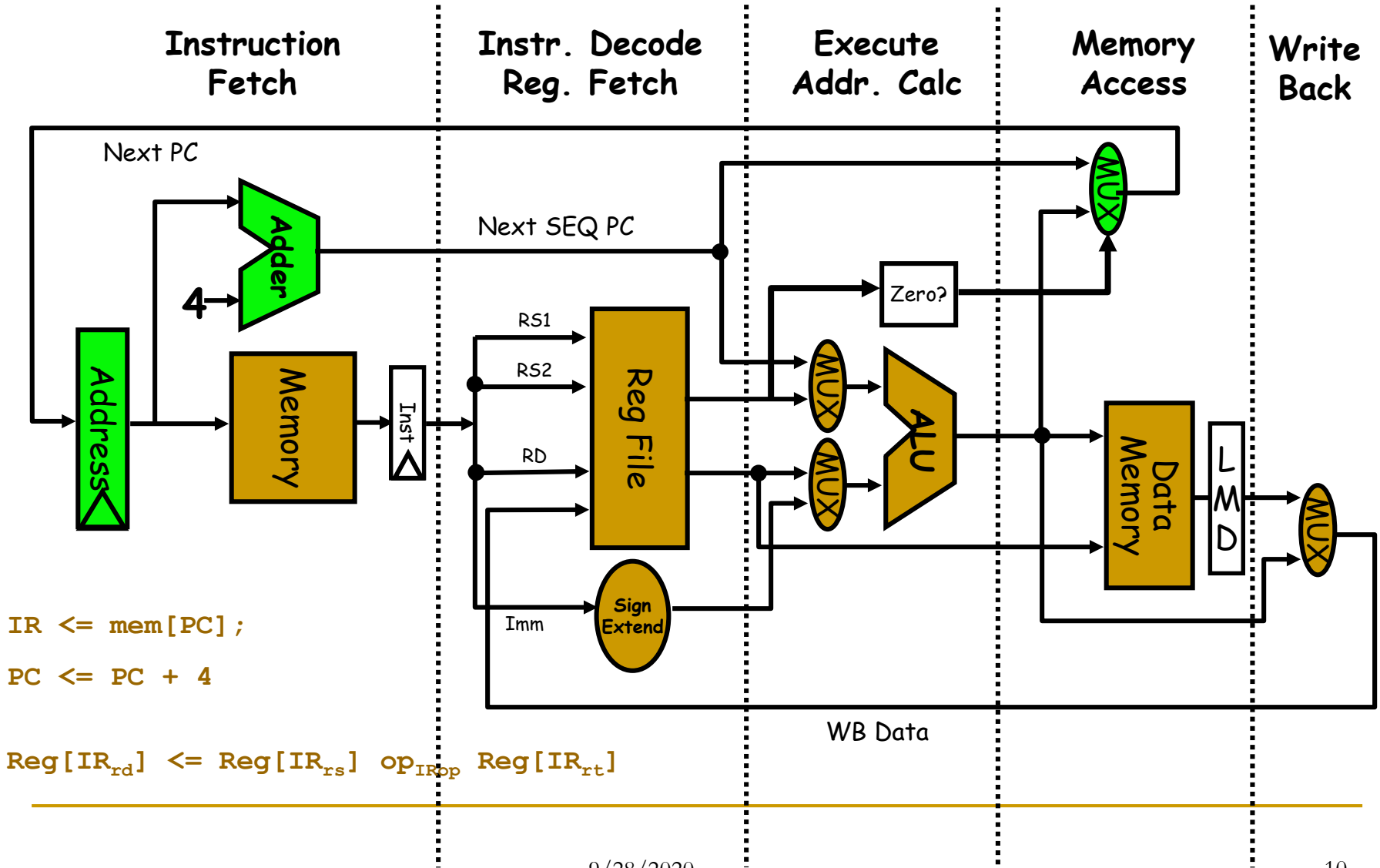
Pipelining Lessons



- 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

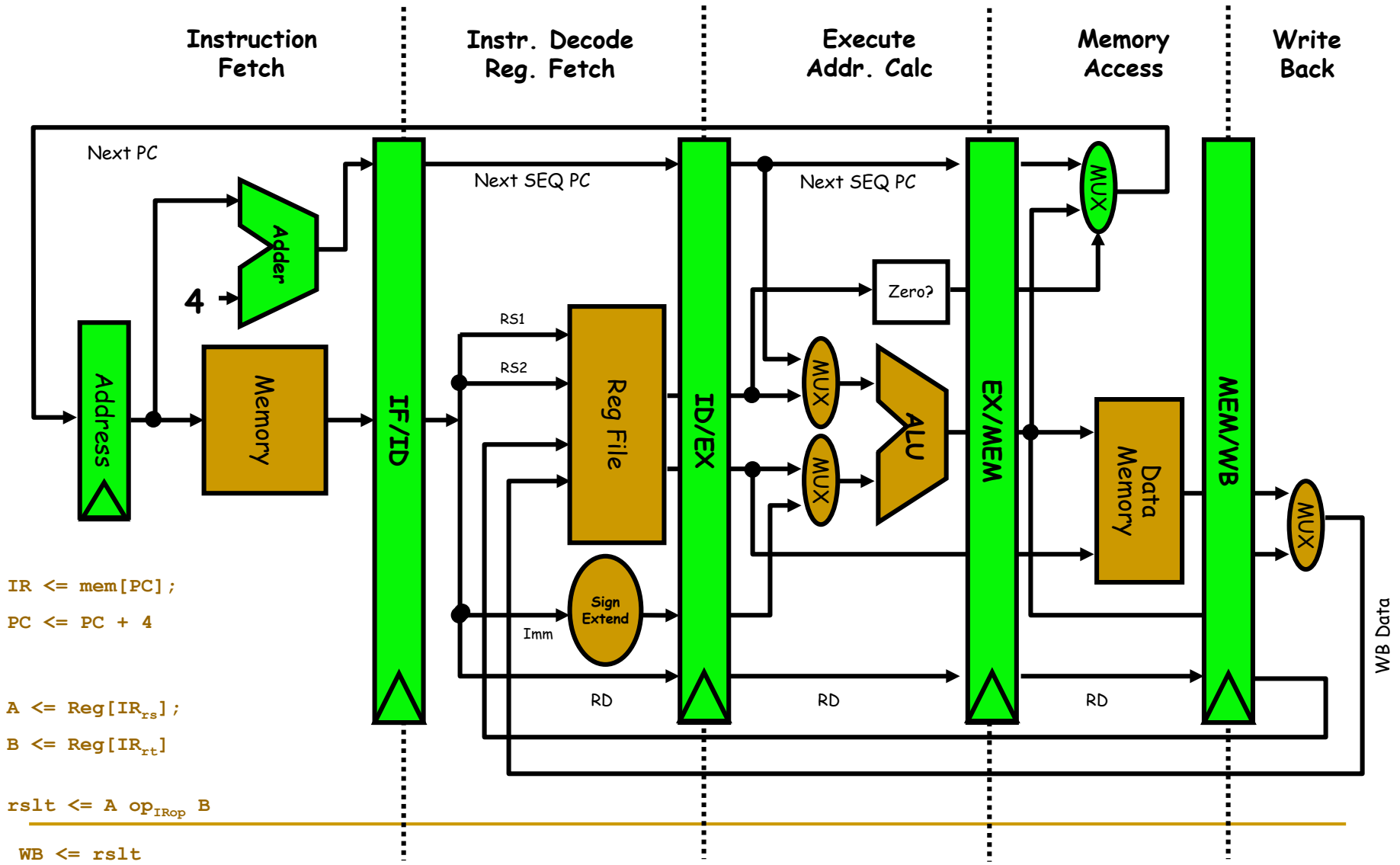
5 Steps of MIPS Datapath

Figure A.2, Page A-8



5 Steps of MIPS Datapath

Figure A.3, Page A-9



$IR \leftarrow mem[PC];$

$PC \leftarrow PC + 4$

$A \leftarrow Reg[IR_{rs}];$

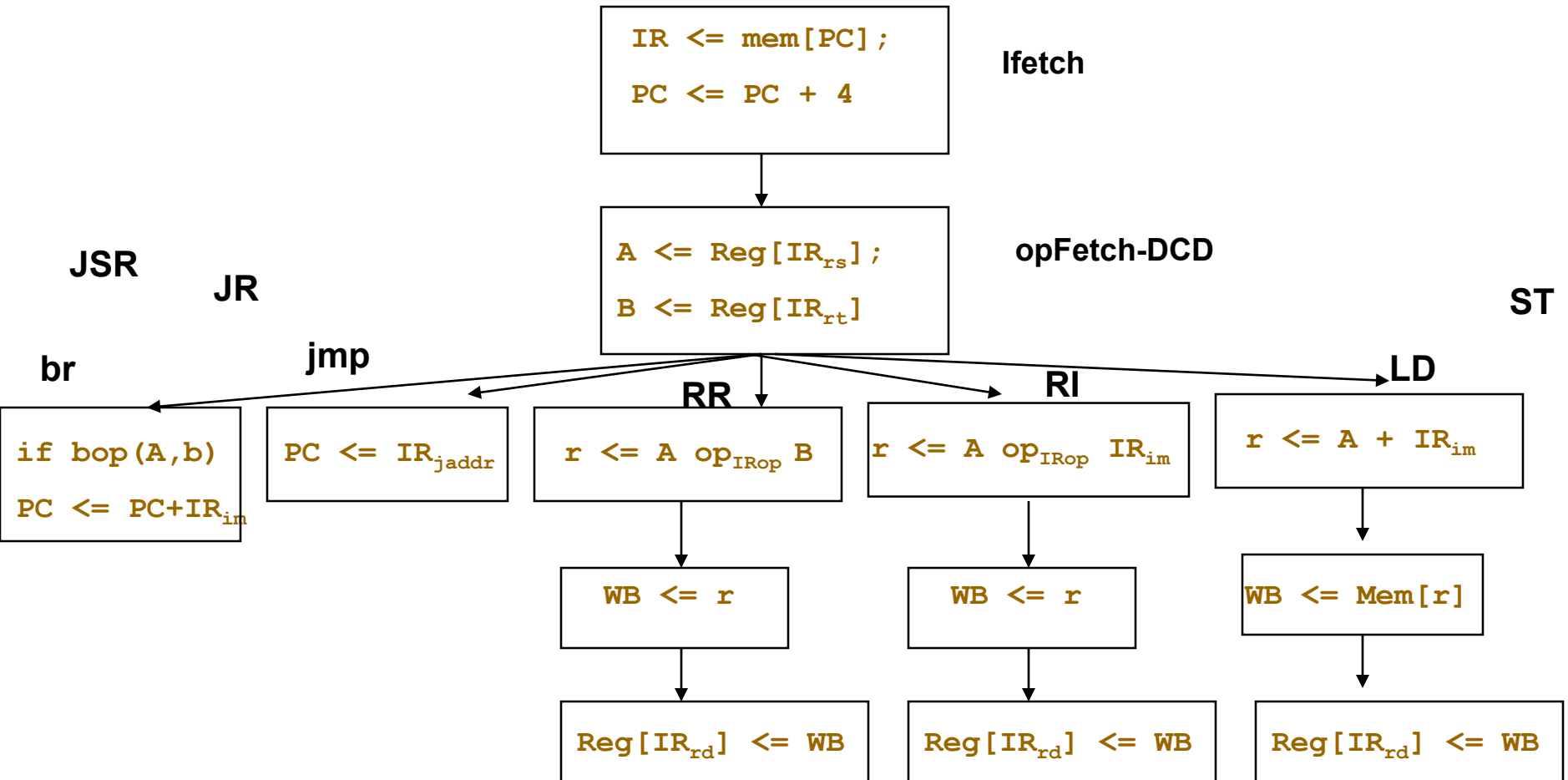
$B \leftarrow Reg[IR_{rt}]$

$rslt \leftarrow A \text{ op}_{IRop} B$

$WB \leftarrow rslt$

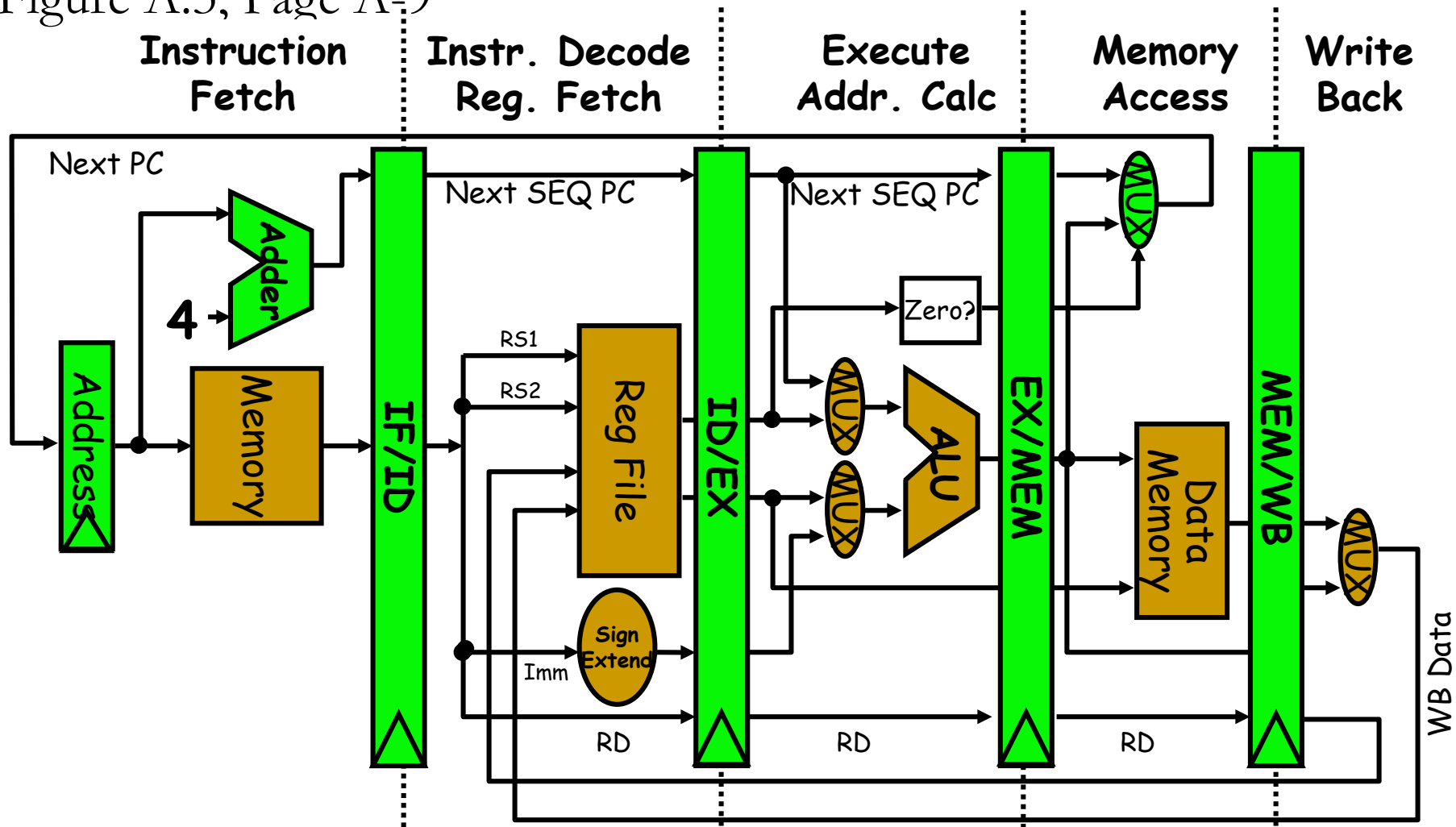
$Reg[IR_{rd}] \leftarrow WB$

Inst. Set Processor Controller



5 Steps of MIPS Datapath

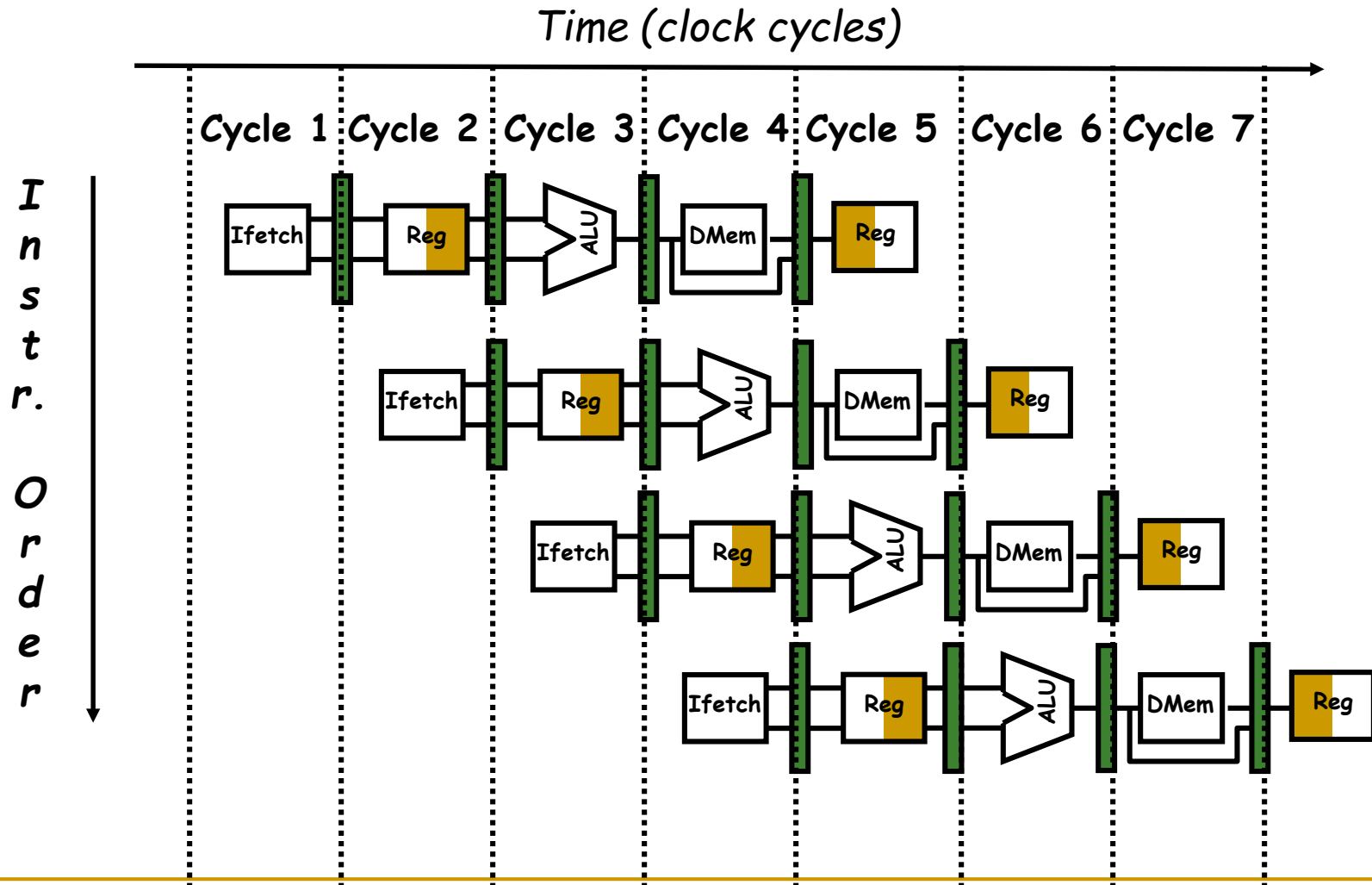
Figure A.3, Page A-9



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

Visualizing Pipelining

Figure A.2, Page A-8

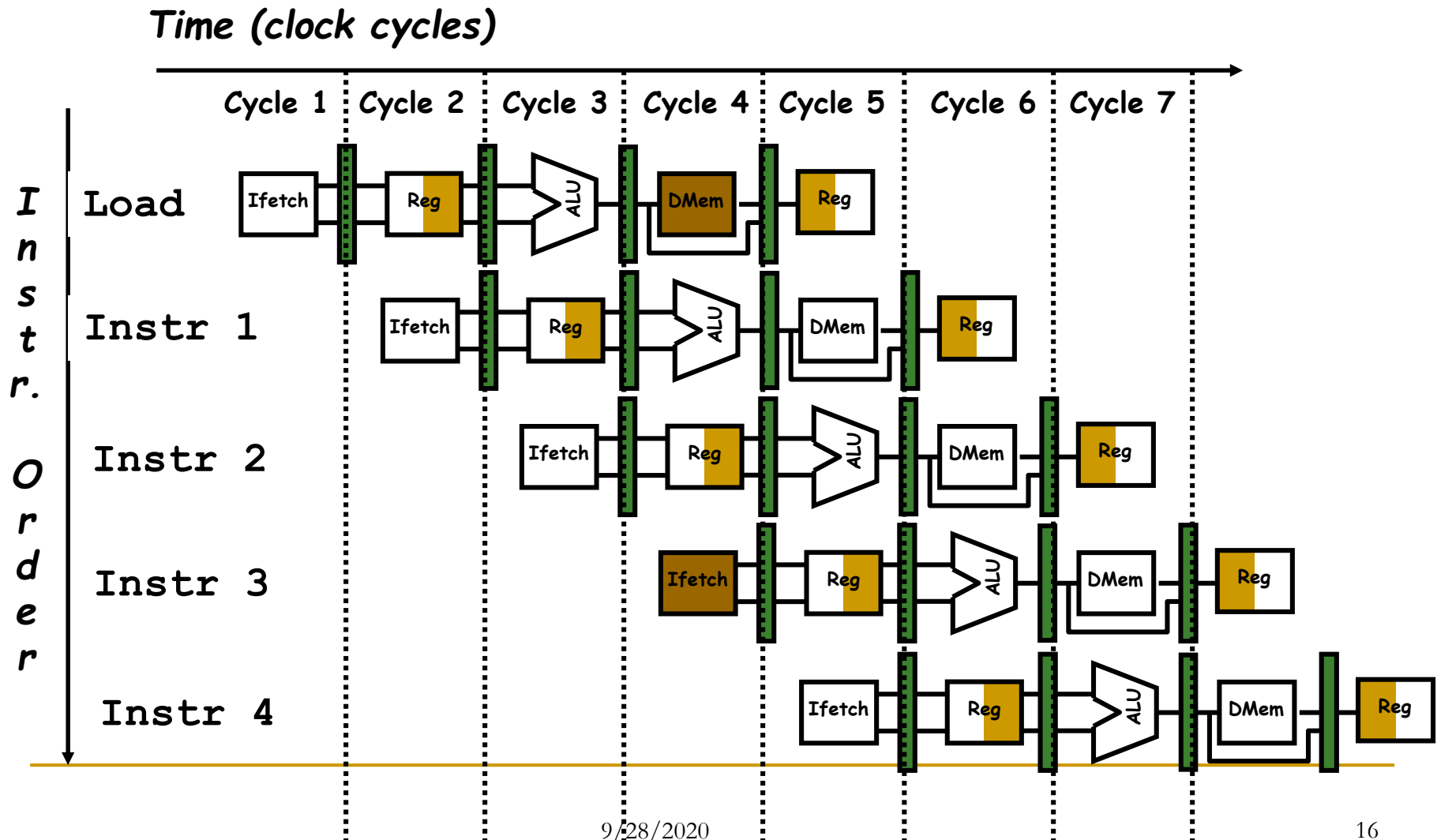


Pipelining is not quite that easy!

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
 - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
 - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

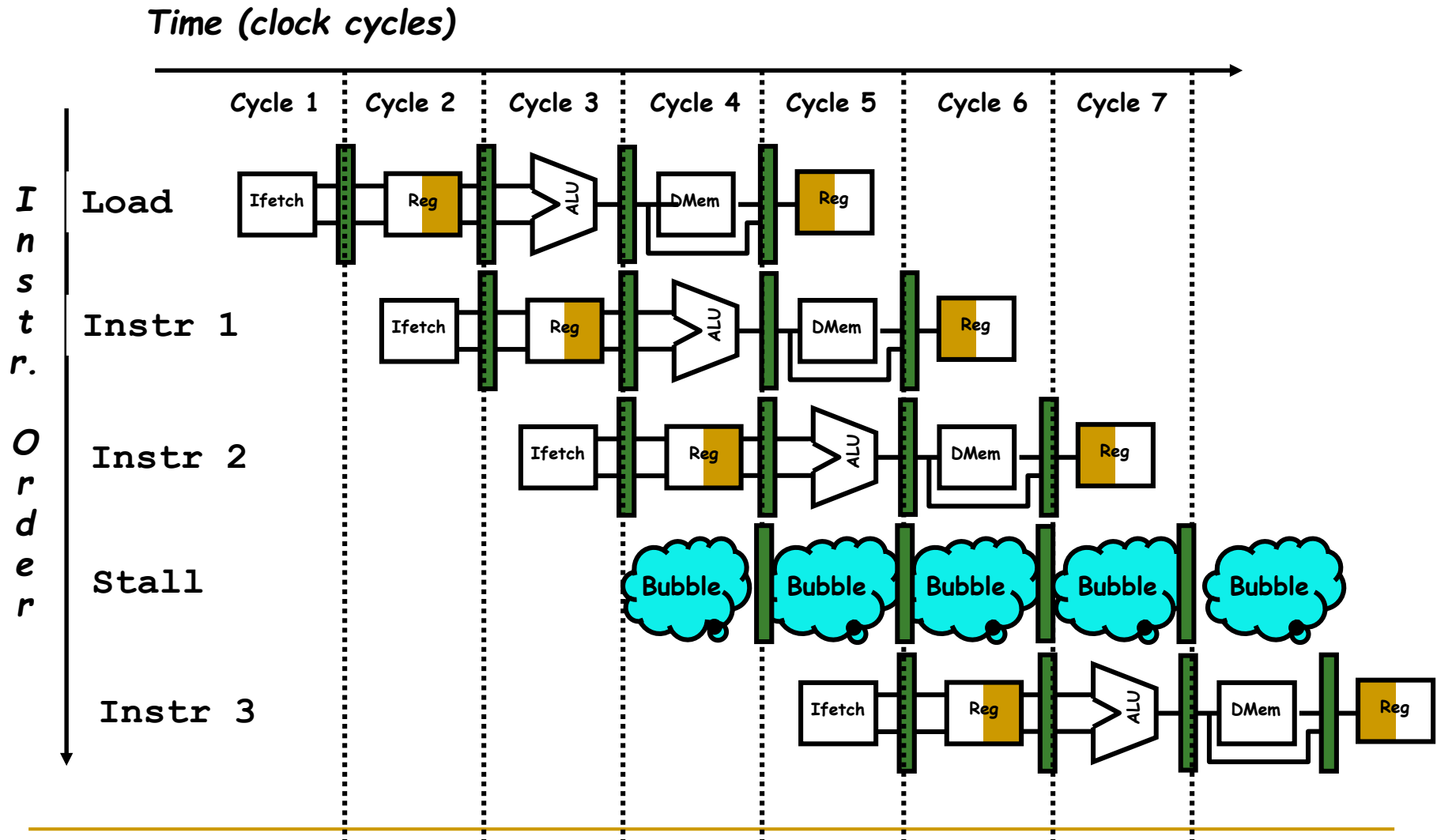
One Memory Port/Structural Hazards

Figure A.4, Page A-14



One Memory Port/Structural Hazards

(Similar to Figure A.5, Page A-15)

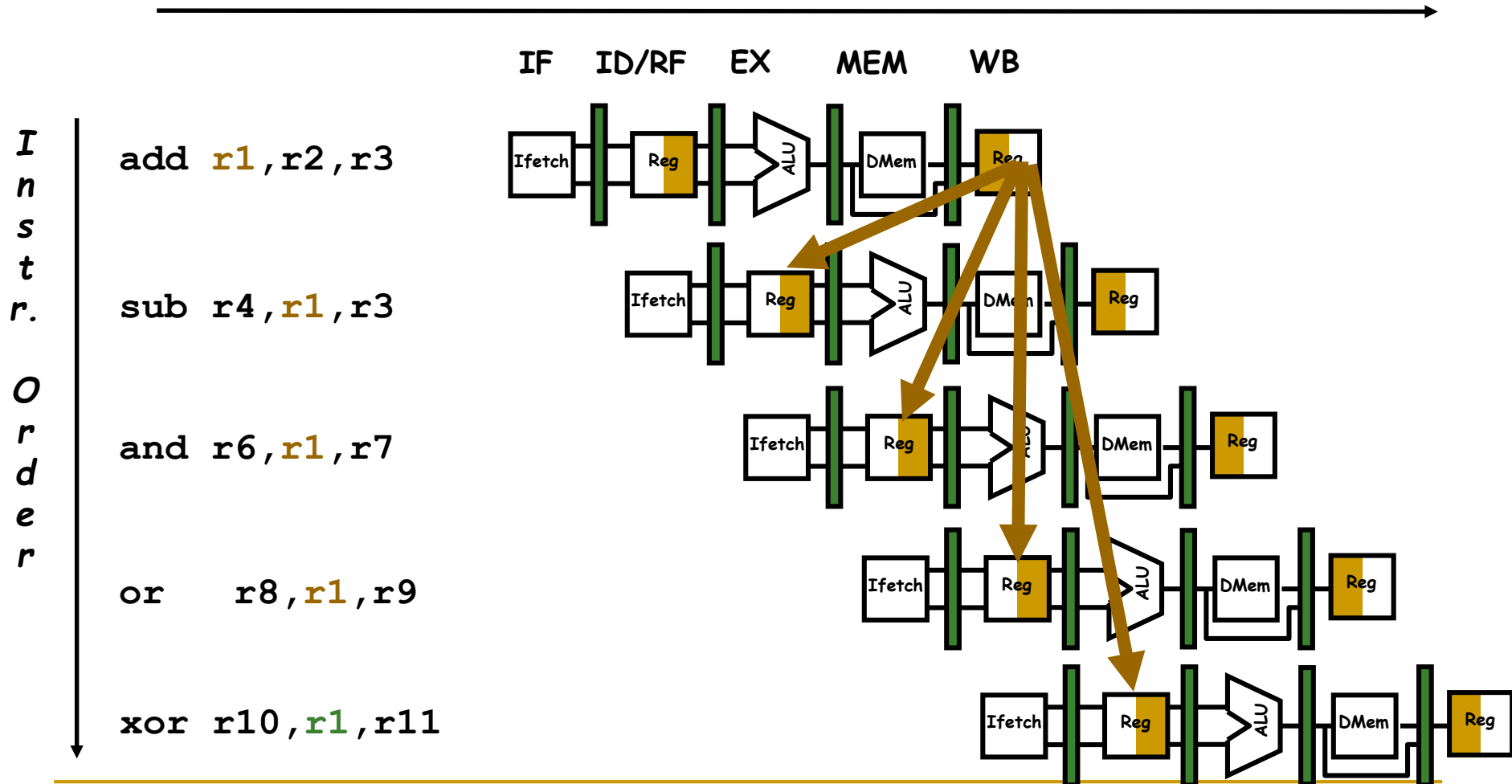


How do you “bubble” the pipe?

Data Hazard on R1

Figure A.6, Page A-17

Time (clock cycles)



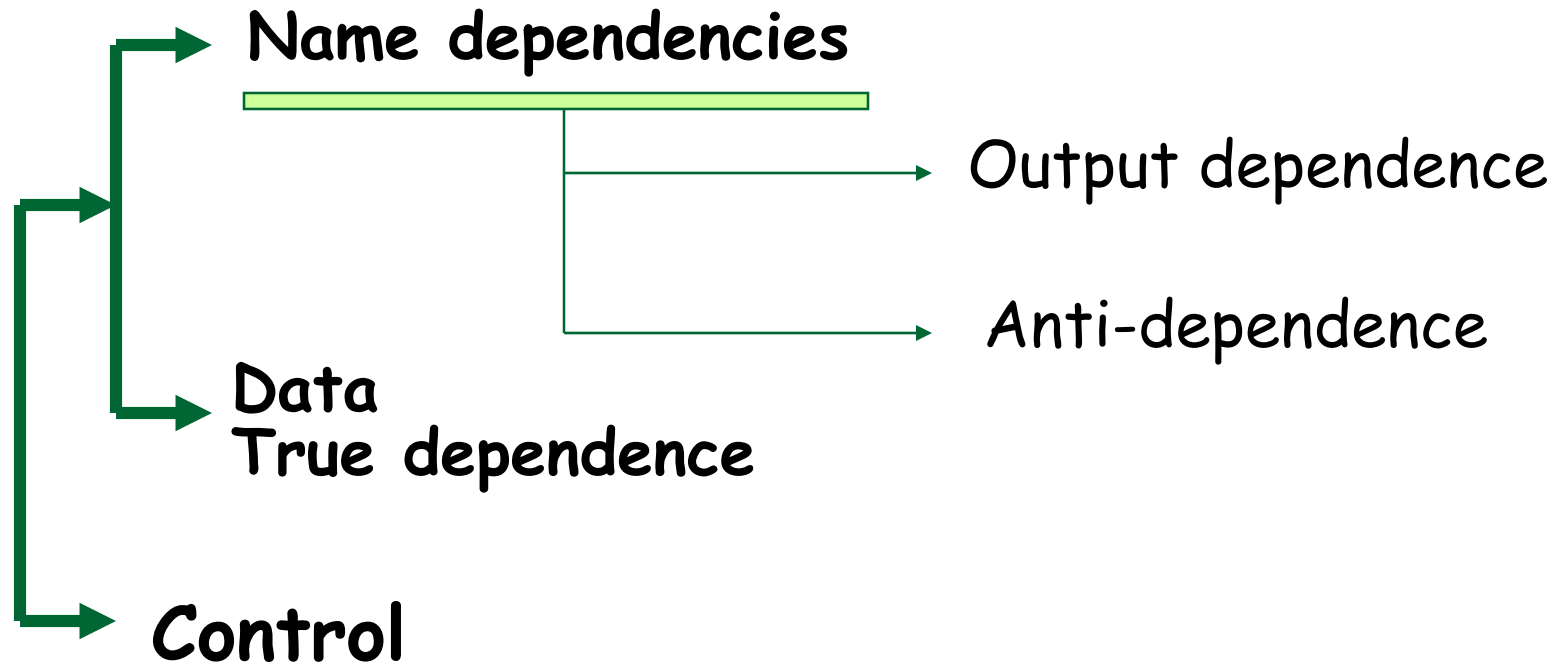
Dependences and hazards

- Dependences are a program property:
 - If two instructions are data dependent they cannot execute simultaneously.
 - Existence of control-dependences means serialization.
 - Whether a dependence results in a hazard and whether that hazard actually causes a stall are properties of the pipeline organization.
 - Data dependences may occur through registers or memory.

Dependences and hazards

- The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. A data dependence:
 - Indicates that there is a possibility of a hazard.
 - Determines the order in which results must be calculated, and
 - Sets an upper bound on the amount of parallelism that can be exploited.

Dependencies



Data dependences

- Data dependence, true dependence, and true data dependence are terms used to mean the same thing :
- An instruction j is data dependent on instruction i if either of the following holds:
 - instruction i produces a result that may be used by instruction j , or
 - instruction j is data dependent on instruction k , and instruction k is data dependent on instruction i .
- Chains of dependent instructions.

Name dependences

- Output dependence :
 - When instruction I and j write the same register or memory location. The ordering must be preserved to leave the correct value in the register:
 - add r7,r4,r3
 - div r7,r2,r8
- Antidependence :
 - When instruction j writes a register or memory location that instruction i reads :
 - i: add r6,r5,r4
 - j: sub r5,r8,r11

Data Dependences through registers/memory

- Dependences through registers are easy :
 - `lw r10,10(r11)`
 - `add r12,r10,r8`
 - just compare register names.

- Dependences through memory are harder :
 - `sw r10,4 (r2)`
 - `lw r6,0(r4)`
 - is $r2+4 = r4+0$? If so they are dependent, if not, they are not.

Control dependences

- An instruction j is control dependent on i if the execution of j is controlled by instruction i .

l: If $a < b$


j: $a = a + 1$; j is control dependent on l.

- 1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
- 2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

Three Generic Data Hazards

- Read After Write (RAW)

Instr_j tries to read operand before Instr_i writes it

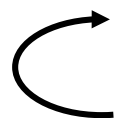
 I: add **r1**, r2, r3
J: sub r4, **r1**, r3

- Caused by a true dependence in the program.

Three Generic Data Hazards

- Write After Read (WAR)

Instr_j writes operand *before* Instr_i reads it



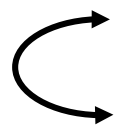
```
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- Caused by an “anti-dependence” in the program. This results from reuse of the name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
 - ❑ All instructions take 5 stages, and
 - ❑ Reads are always in stage 2, and
 - ❑ Writes are always in stage 5

Three Generic Data Hazards

- Write After Write (WAW)

Instr_j writes operand *before* Instr_i writes it.

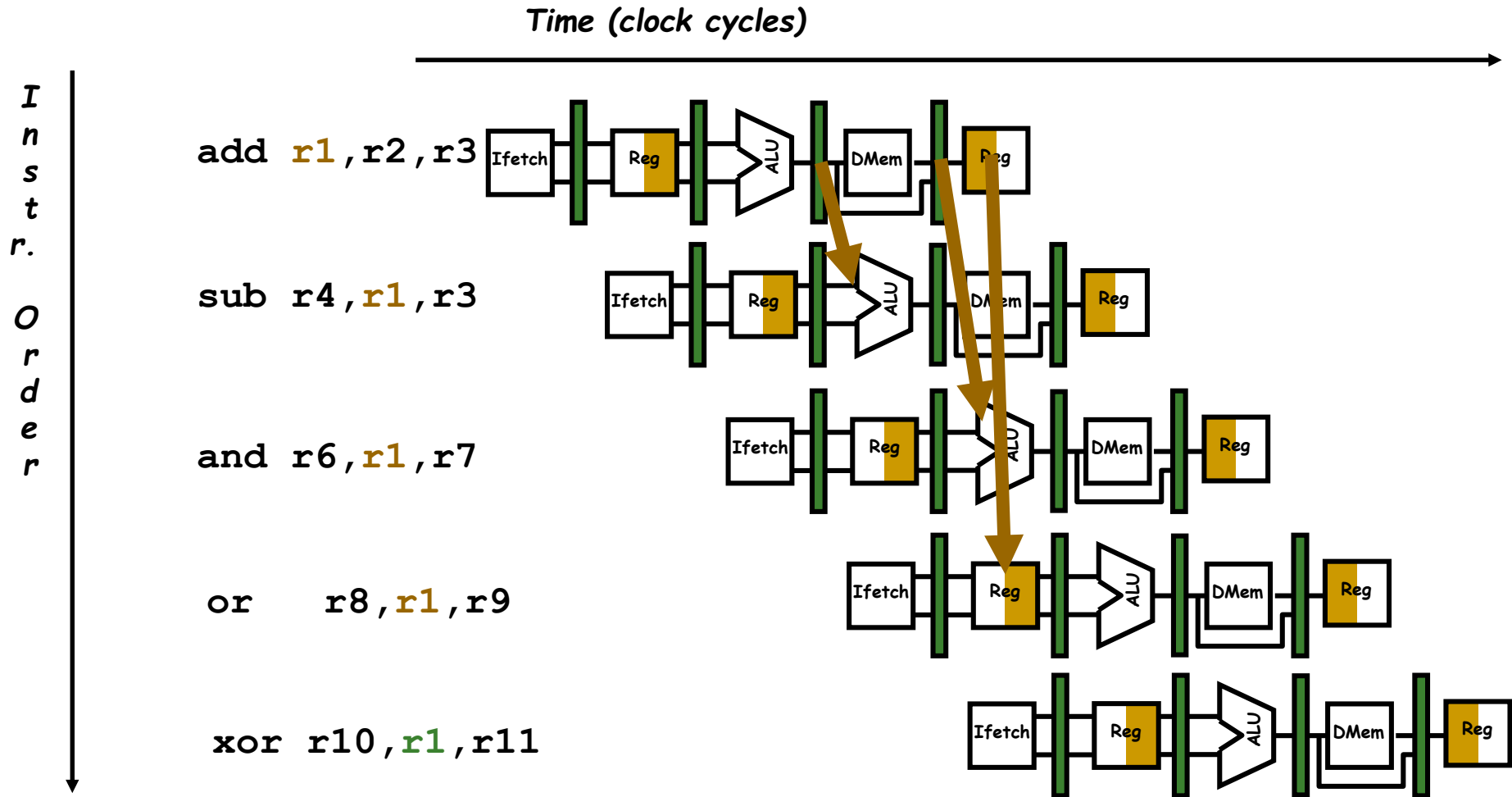


```
I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- Caused by an “output dependence” in the program. This also results from the reuse of name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

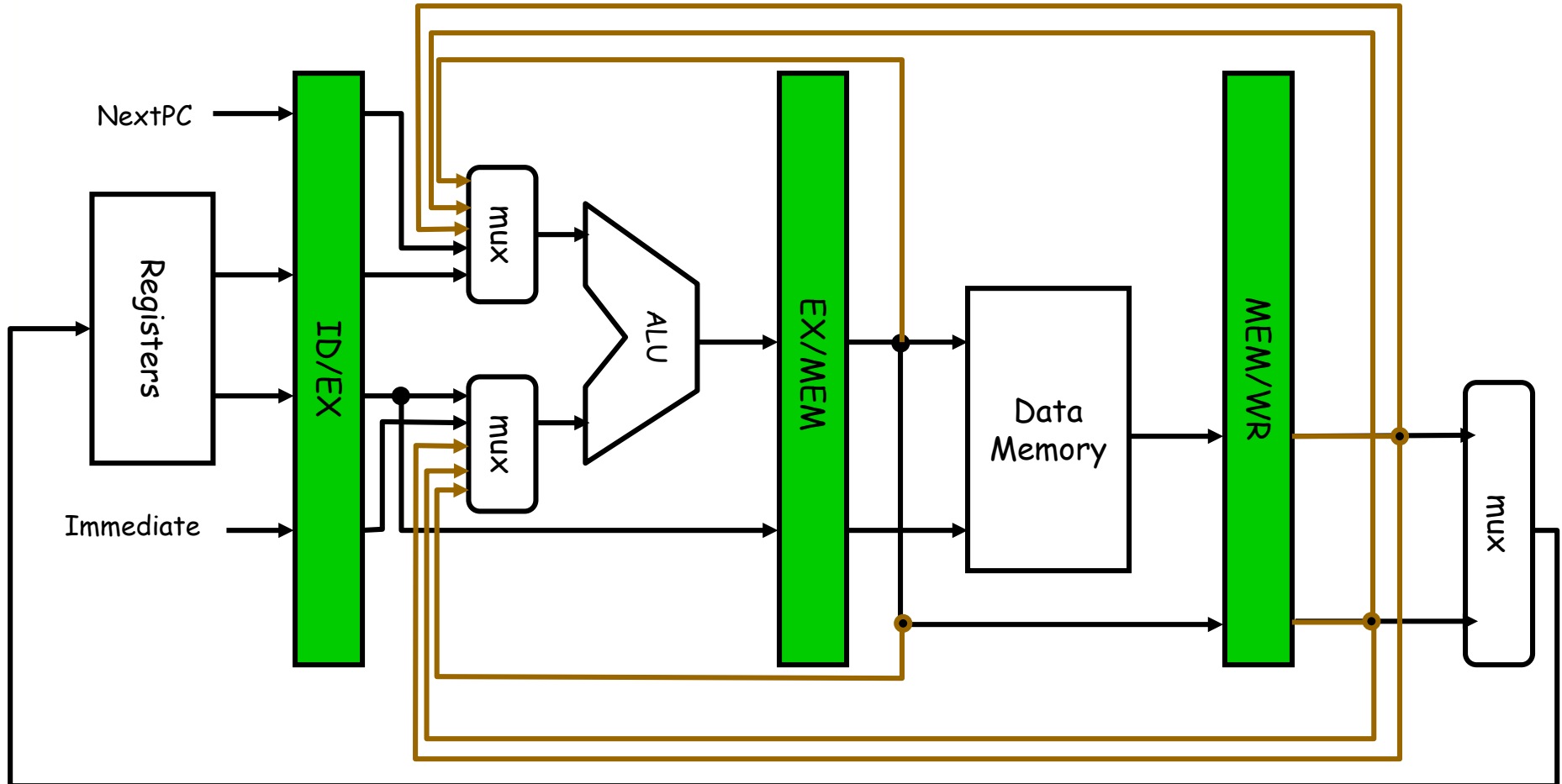
Forwarding to Avoid Data Hazard

Figure A.7, Page A-19



HW Change for Forwarding

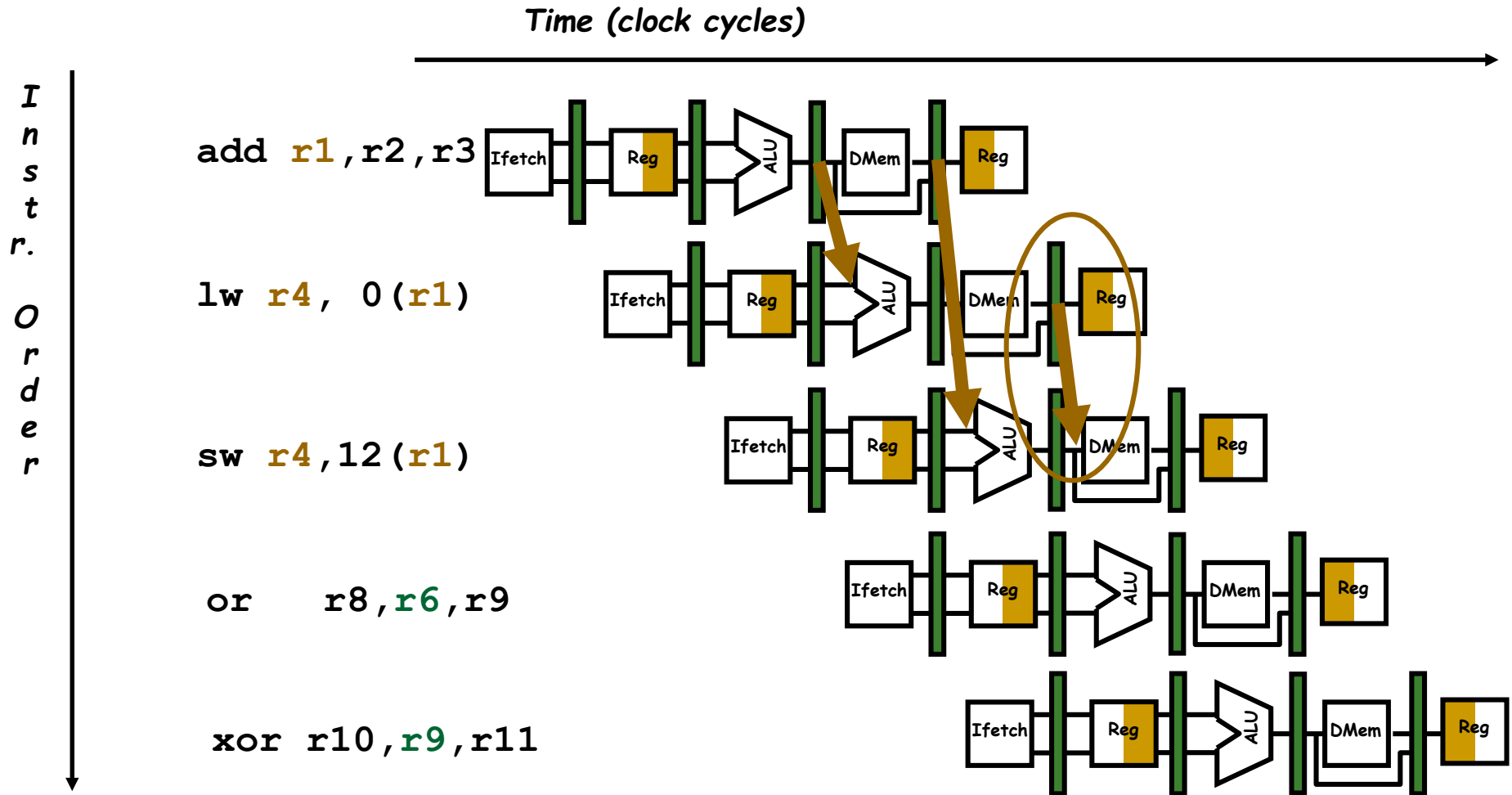
Figure A.23, Page A-37



What circuit detects and resolves this hazard?

Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-20



Data Hazard Even with Forwarding

Figure A.9, Page A-21

Time (clock cycles)

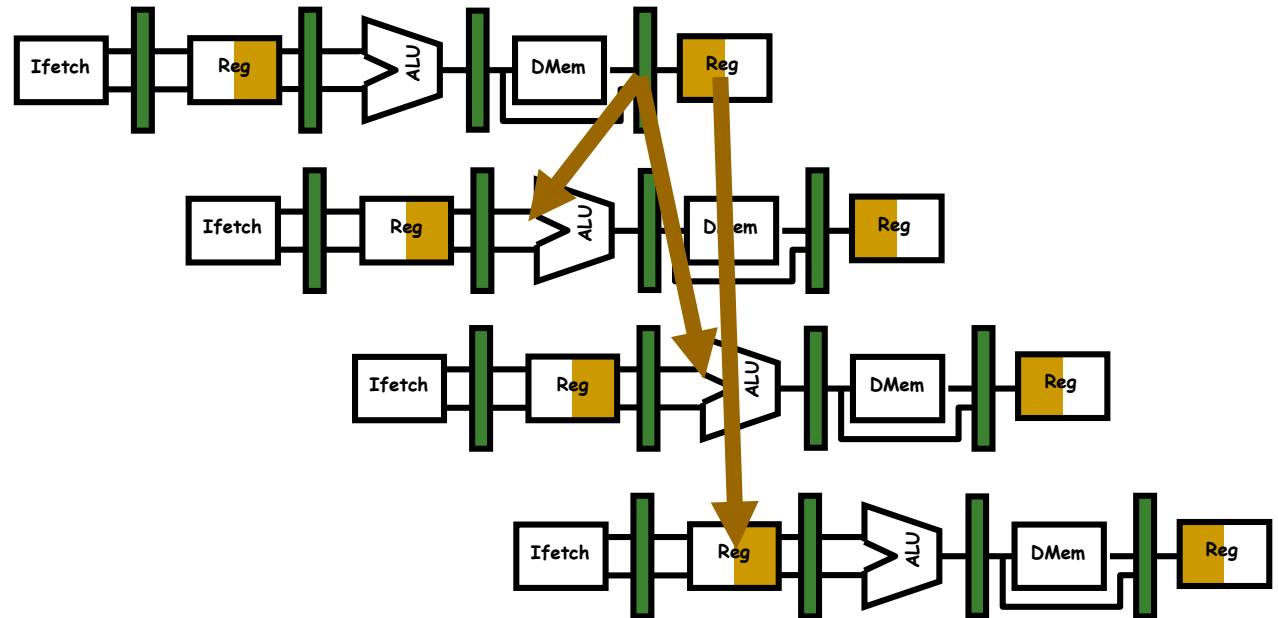
Inst.
Order

lw r1, 0(r2)

sub r4, r1, r6

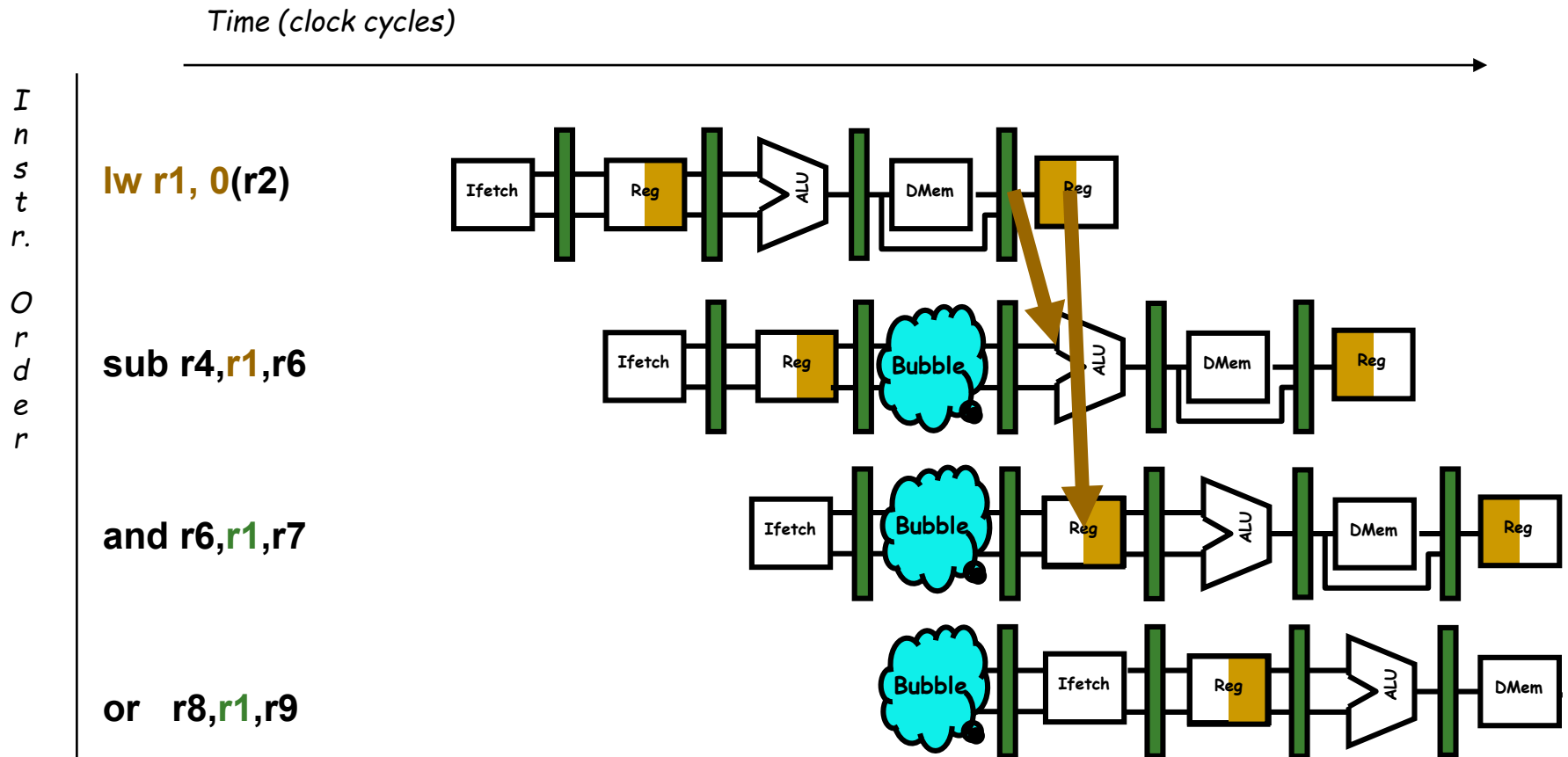
and r6, r1, r7

or r8, r1, r9



Data Hazard Even with Forwarding

(Similar to Figure A.10, Page A-21)



How is this detected?

Software Scheduling to Avoid Load Hazards

Try producing fast code for

$a = b + c;$

$d = e - f;$

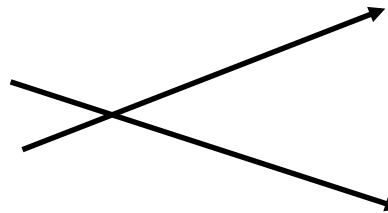
assuming $a, b, c, d, e,$ and f in memory.

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:

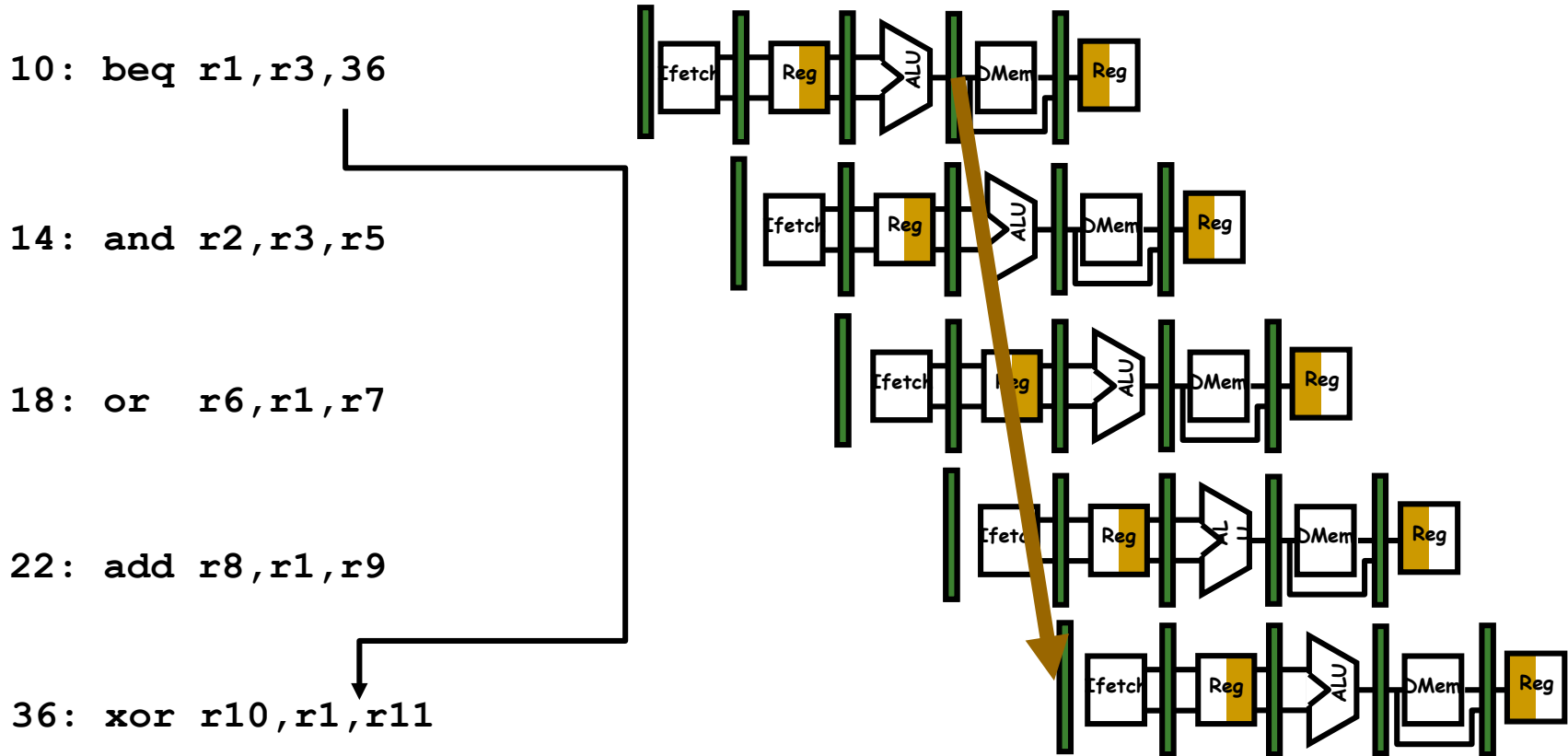
```
LW      Rb,b
LW      Rc,c
LW      Re,e
ADD     Ra,Rb,Rc
LW      Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW      d,Rd
```



Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches

Three Stage Stall



What do you do with the 3 instructions in between?

How do you do it?

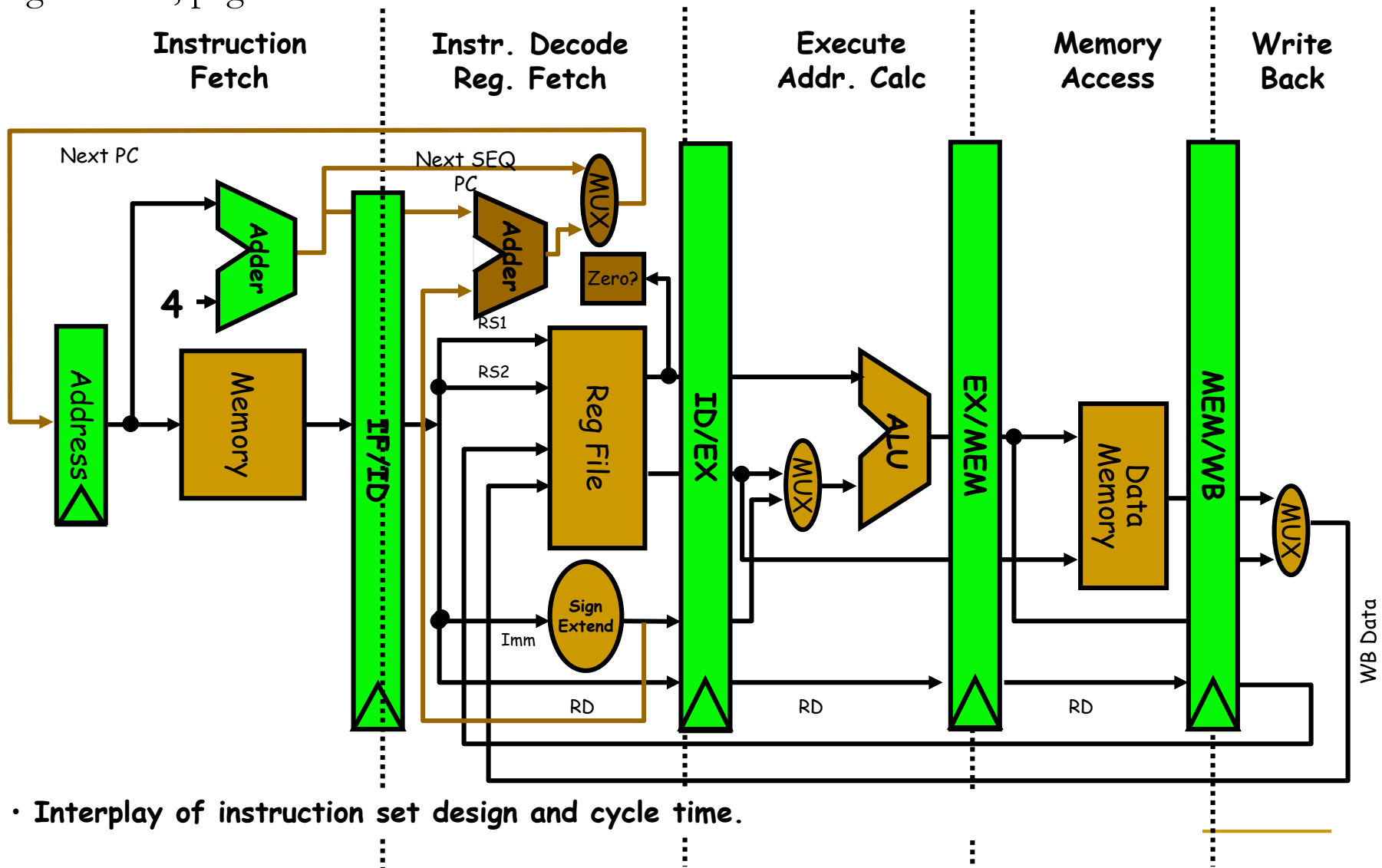
Where is the "commit"?

Branch Stall Impact

- If $CPI = 1$, 30% branch,
Stall 3 cycles \Rightarrow new $CPI = 1.9!$
- Two part solution:
 - Determine branch taken or not sooner, AND
 - Compute taken branch address earlier
- MIPS branch tests if register = 0 or $\neq 0$
- MIPS Solution:
 - Move Zero test to ID/RF stage
 - Adder to calculate new PC in ID/RF stage
 - 1 clock cycle penalty for branch versus 3

Pipelined MIPS Datapath

Figure A.24, page A-38



- Interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- ❑ Execute successor instructions in sequence
- ❑ “Squash” instructions in pipeline if branch actually taken
- ❑ Advantage of late pipeline state update
- ❑ 47% MIPS branches not taken on average
- ❑ PC+4 already calculated, so use it to get next instruction

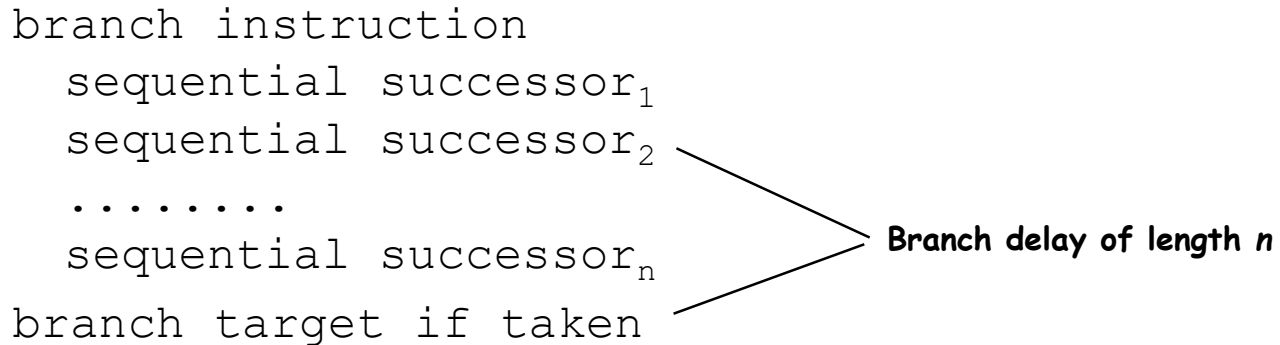
#3: Predict Branch Taken

- ❑ 53% MIPS branches taken on average
- ❑ But haven't calculated branch target address in MIPS
 - MIPS still incurs 1 cycle branch penalty
 - Other machines: branch target known before outcome

Four Branch Hazard Alternatives

#4: Delayed Branch

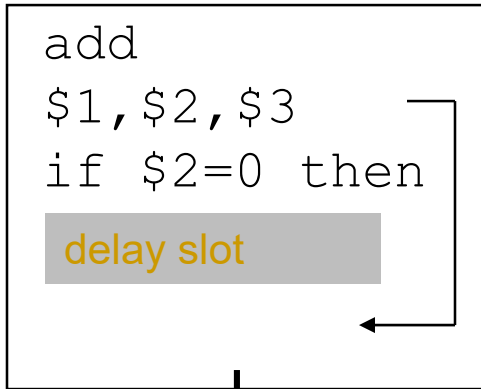
- Define branch to take place **AFTER** a following instruction



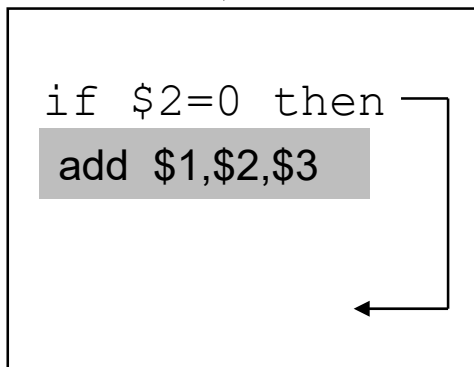
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots (Fig A.14)

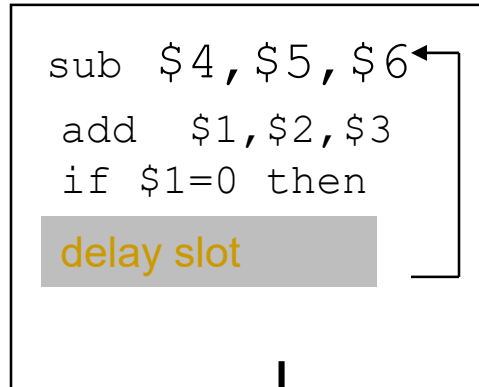
A. From before branch



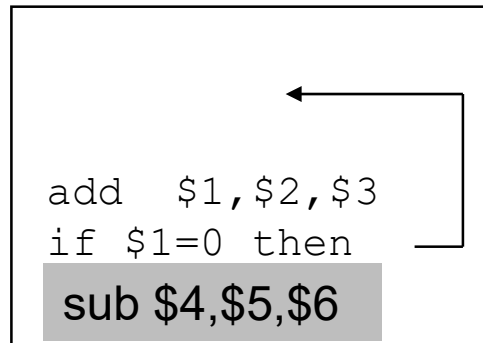
becomes



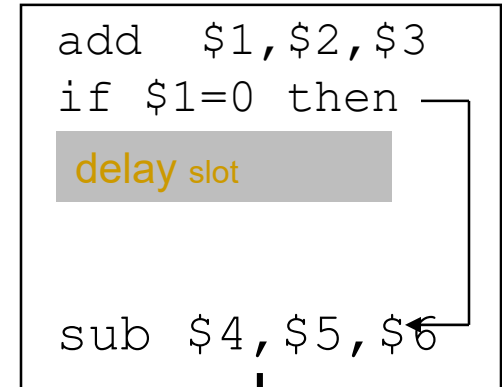
B. From branch target



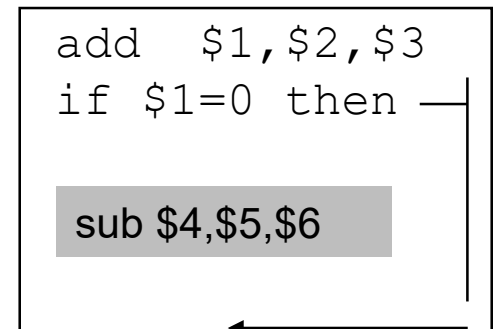
becomes



C. From fall through



becomes



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Delayed Branch

Compiler effectiveness for single branch delay slot:

- ❑ Fills about 60% of branch delay slots
- ❑ About 80% of instructions executed in branch delay slots useful in computation
- ❑ About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
 - ❑ Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
 - ❑ Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

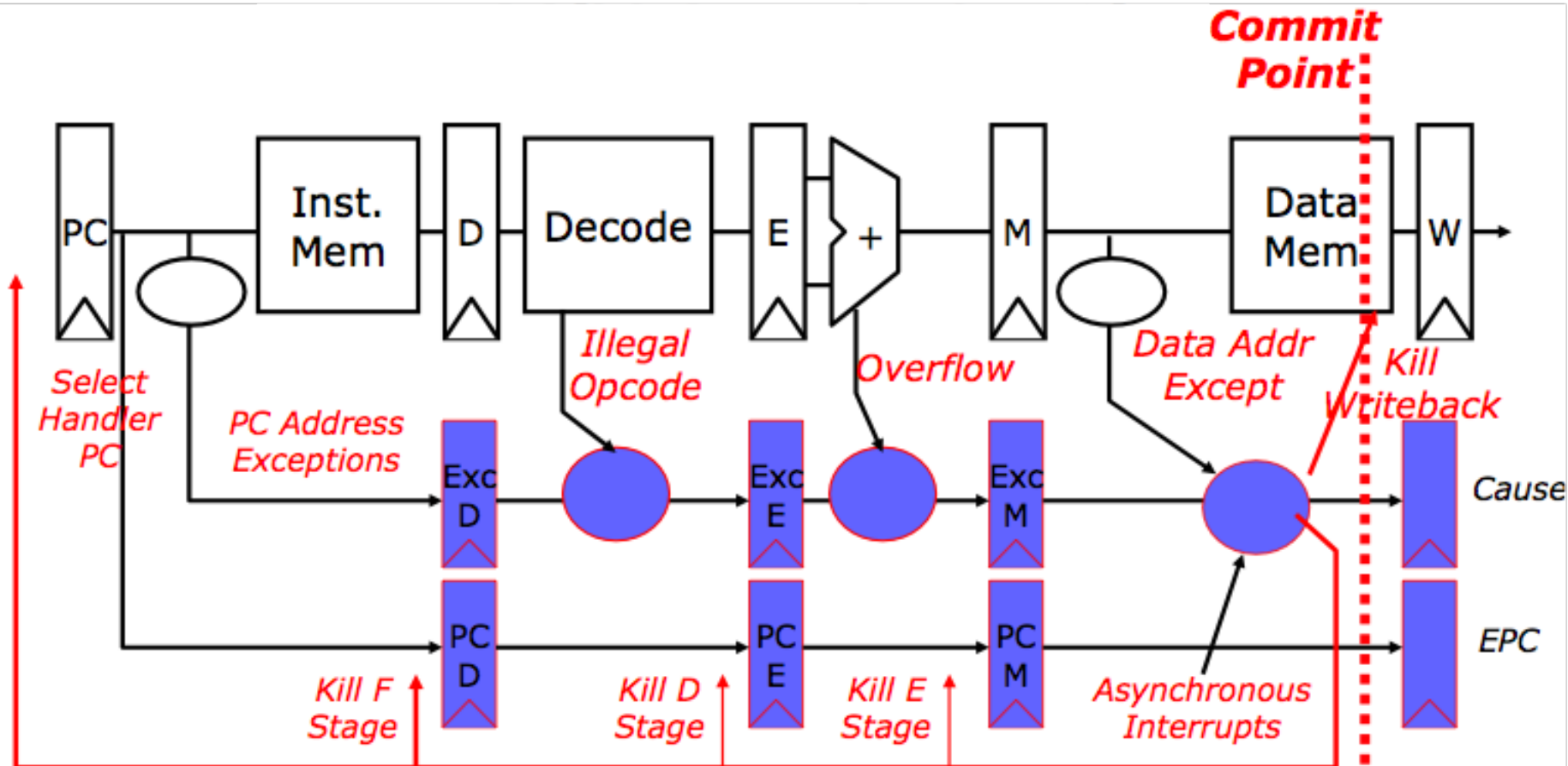
Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

<i>Scheduling scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. unpipelined</i>	<i>speedup v. stall</i>
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

Problems with Pipelining

- **Exception:** An unusual event happens to an instruction during its execution
 - Examples: divide by zero, undefined opcode
- **Interrupt:** Hardware signal to switch the processor to a new instruction stream
 - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- **Problem:** It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
 - The effect of all instructions up to and including I_i is totalling complete
 - No effect of any instruction after I_i can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.

And In Conclusion: Control and Pipelining

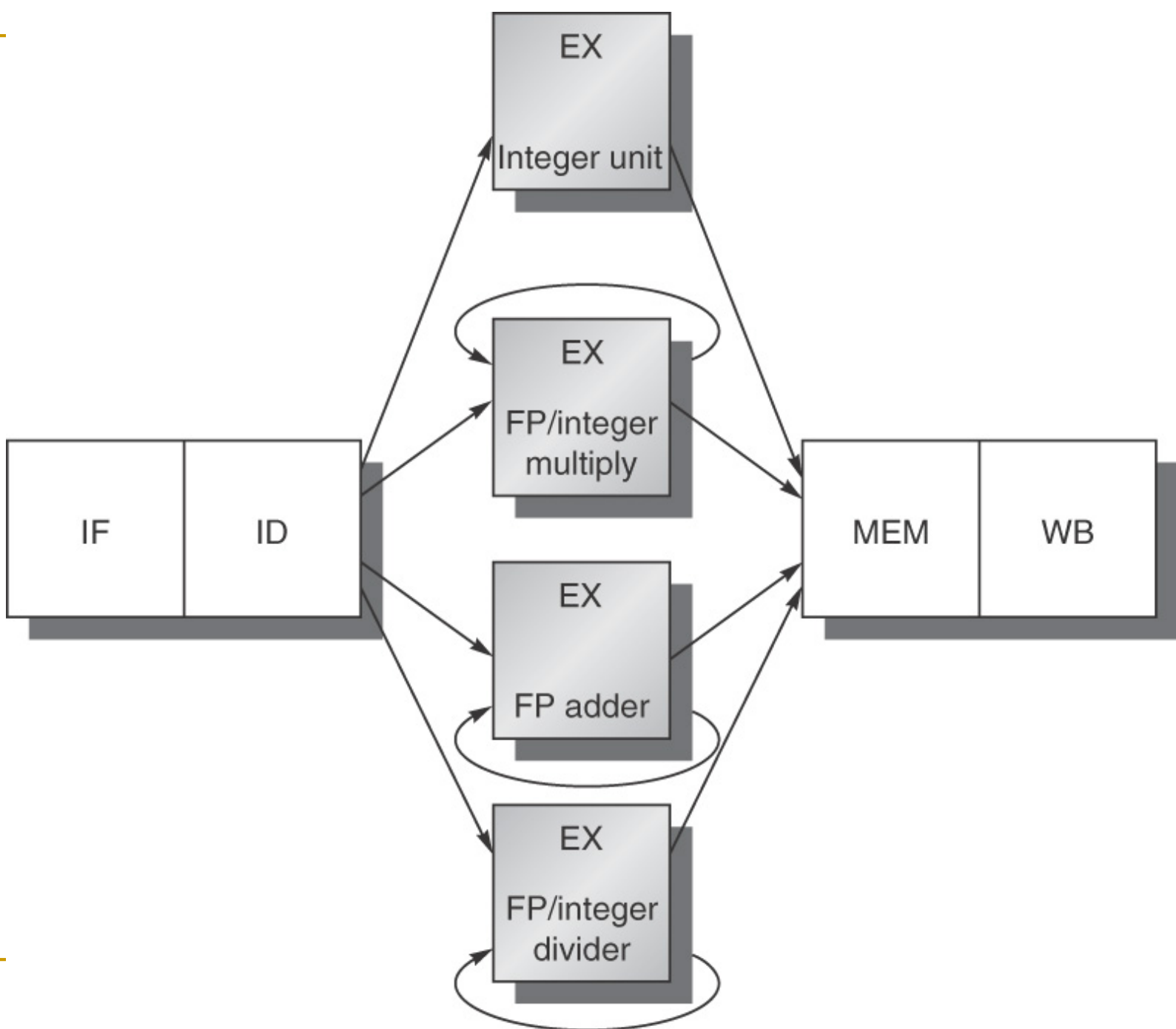
- Control VIA **State Machines** and **Microprogramming**
- Just overlap tasks; easy if tasks are independent
- Speed Up \leq Pipeline Depth; if ideal CPI is 1, then:

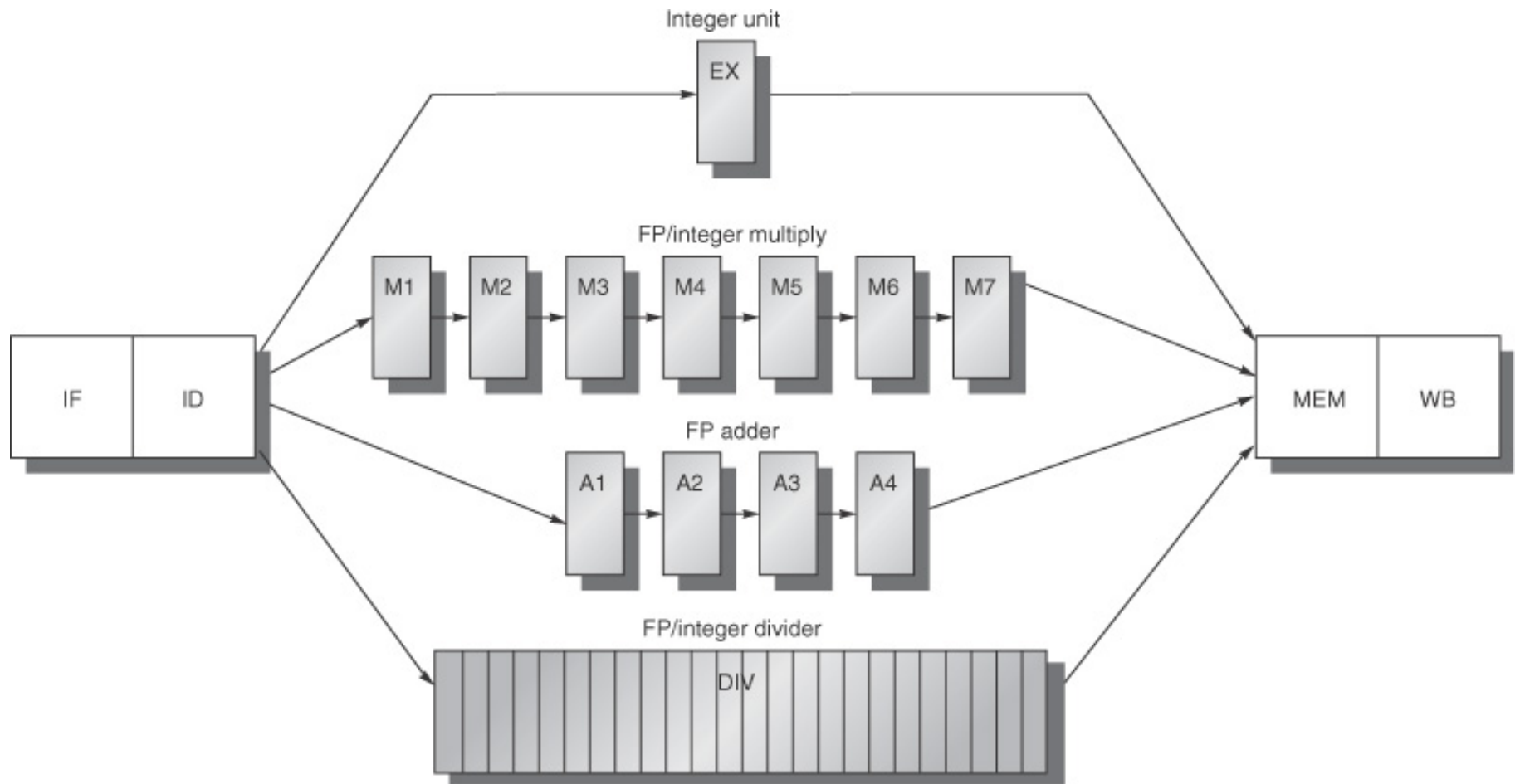
$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- Hazards limit performance on computers:
 - Structural: need more HW resources
 - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
 - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity

Handling multi-cycle operations

- How would the pipeline should be changed if some instructions need more than a single cycle to complete their execution?
- What are the consequences in terms of hazards?





© 2007 Elsevier, Inc. All rights reserved.

Speed Up Equation for Pipelining

$$CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

For simple RISC pipeline, $CPI = 1$:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$\begin{aligned}\text{SpeedUp}_A &= \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}}) \\ &= \text{Pipeline Depth}\end{aligned}$$

$$\begin{aligned}\text{SpeedUp}_B &= \text{Pipeline Depth} / (1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipeline Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipeline Depth}\end{aligned}$$

$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$$

- Machine A is 1.33 times faster

