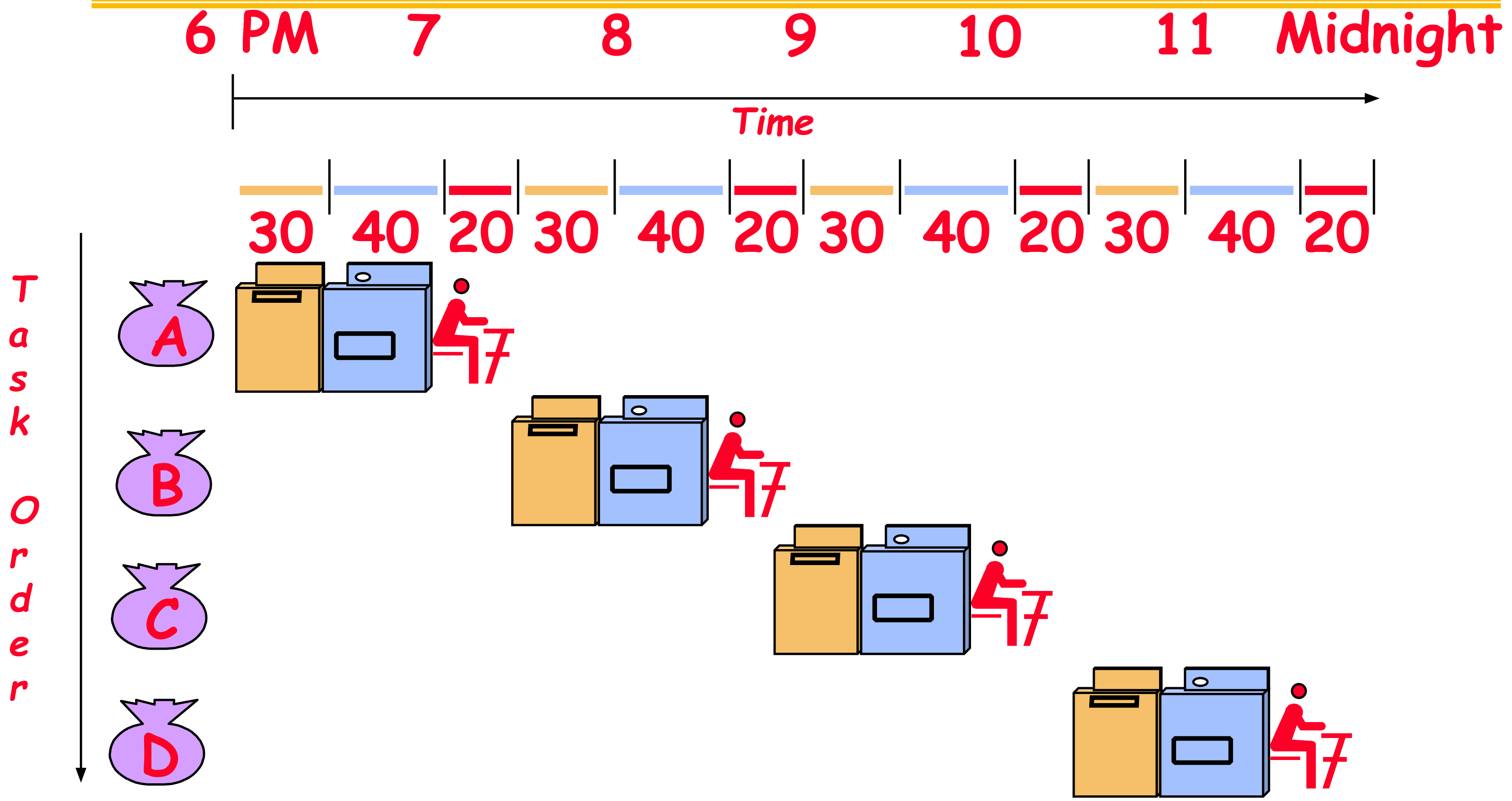
Pipelining

Outline

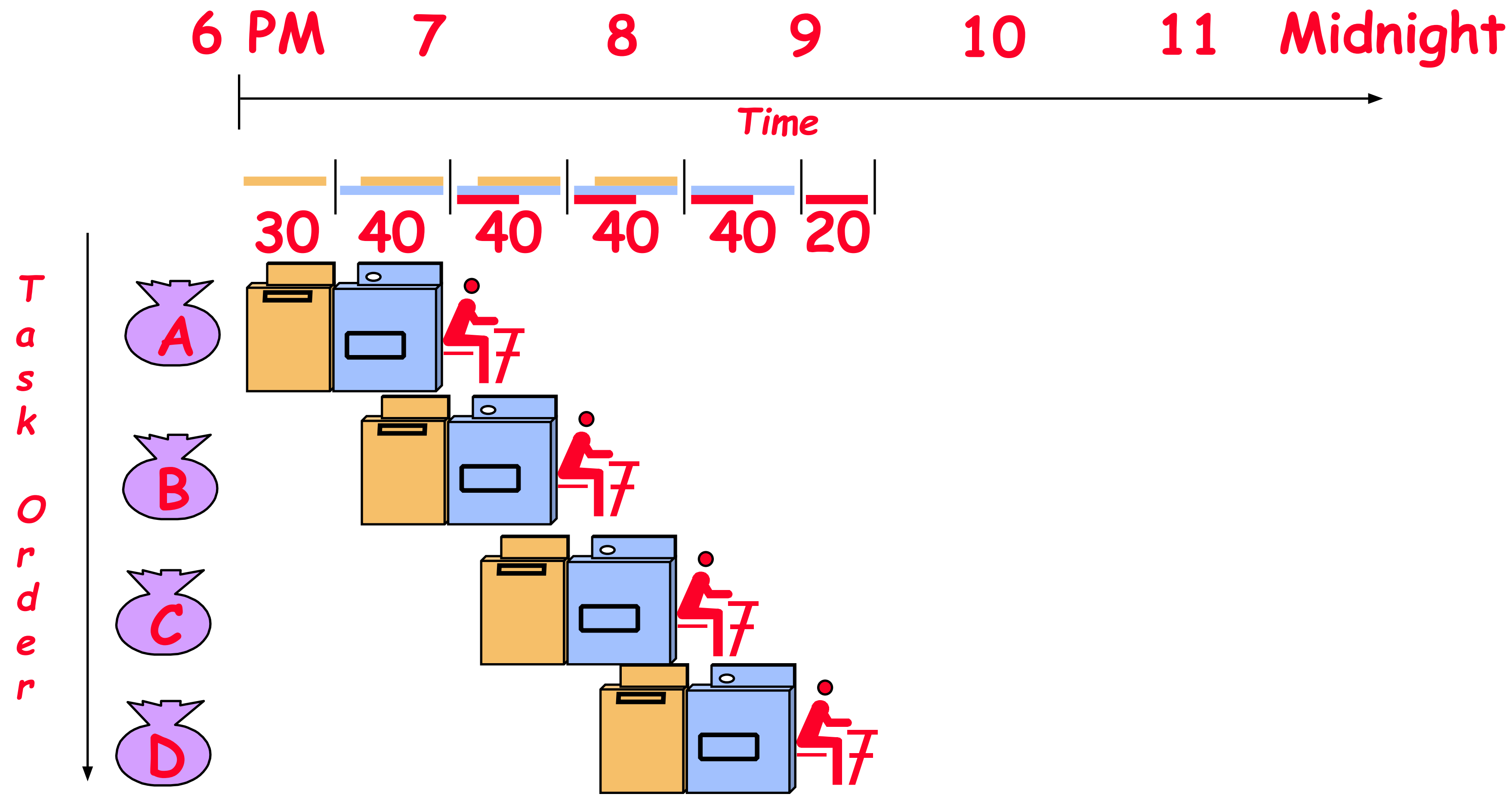
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Multi-cycle Operations
- Conclusion

Sequential Laundry



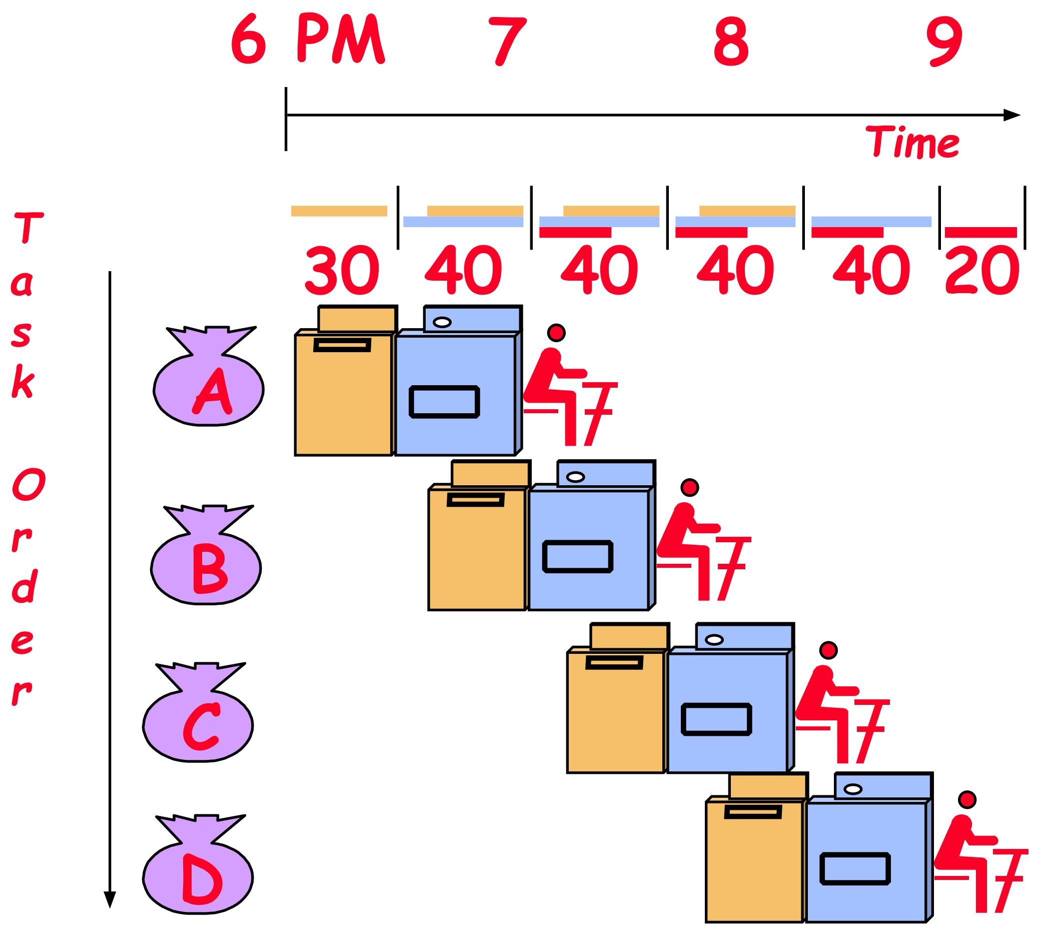
- **Sequential laundry takes 6 hours for 4 loads**
- **If they learned pipelining, how long would laundry take?**

Pipelined Laundry
Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



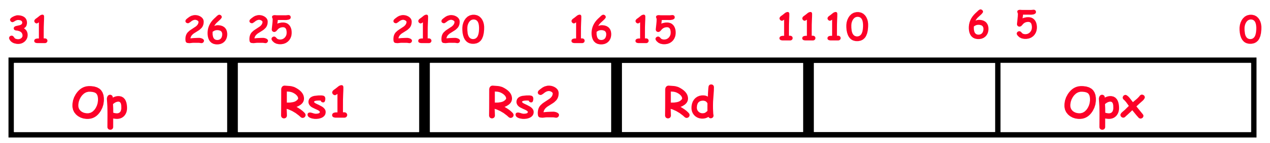
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and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

Instruction Pipelining

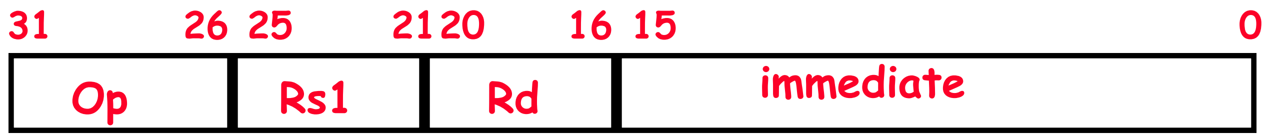
- Execute billions of instructions, so *throughput* is what matters
  - except when?
- What is desirable in instruction sets for pipelining?
  - Variable length instructions vs. all instructions same length?
  - Memory operands part of any operation vs. memory operands only in loads or stores?
  - Register operand many places in instruction format vs. registers located in same place?

Example: MIPS (Note register location)

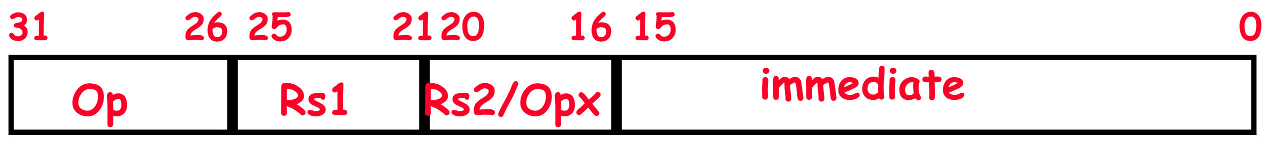
Register-Register



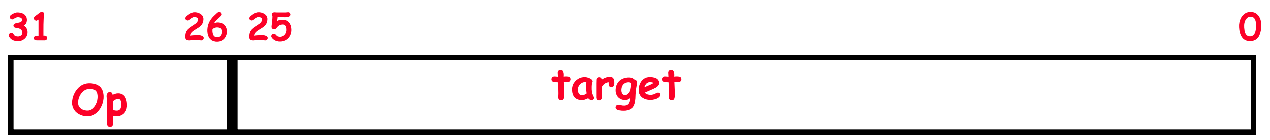
Register-Immediate



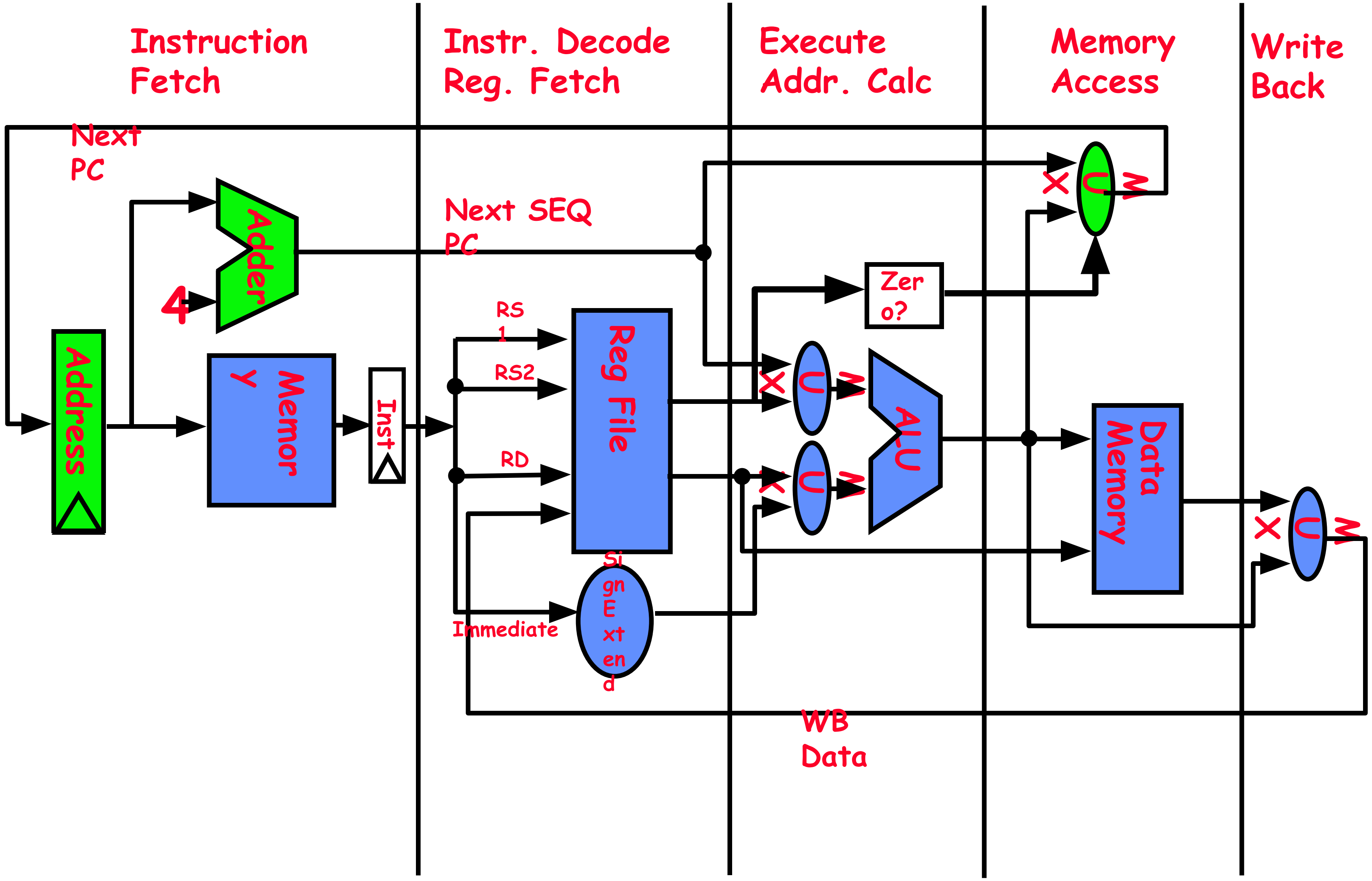
Branch



Jump / Call

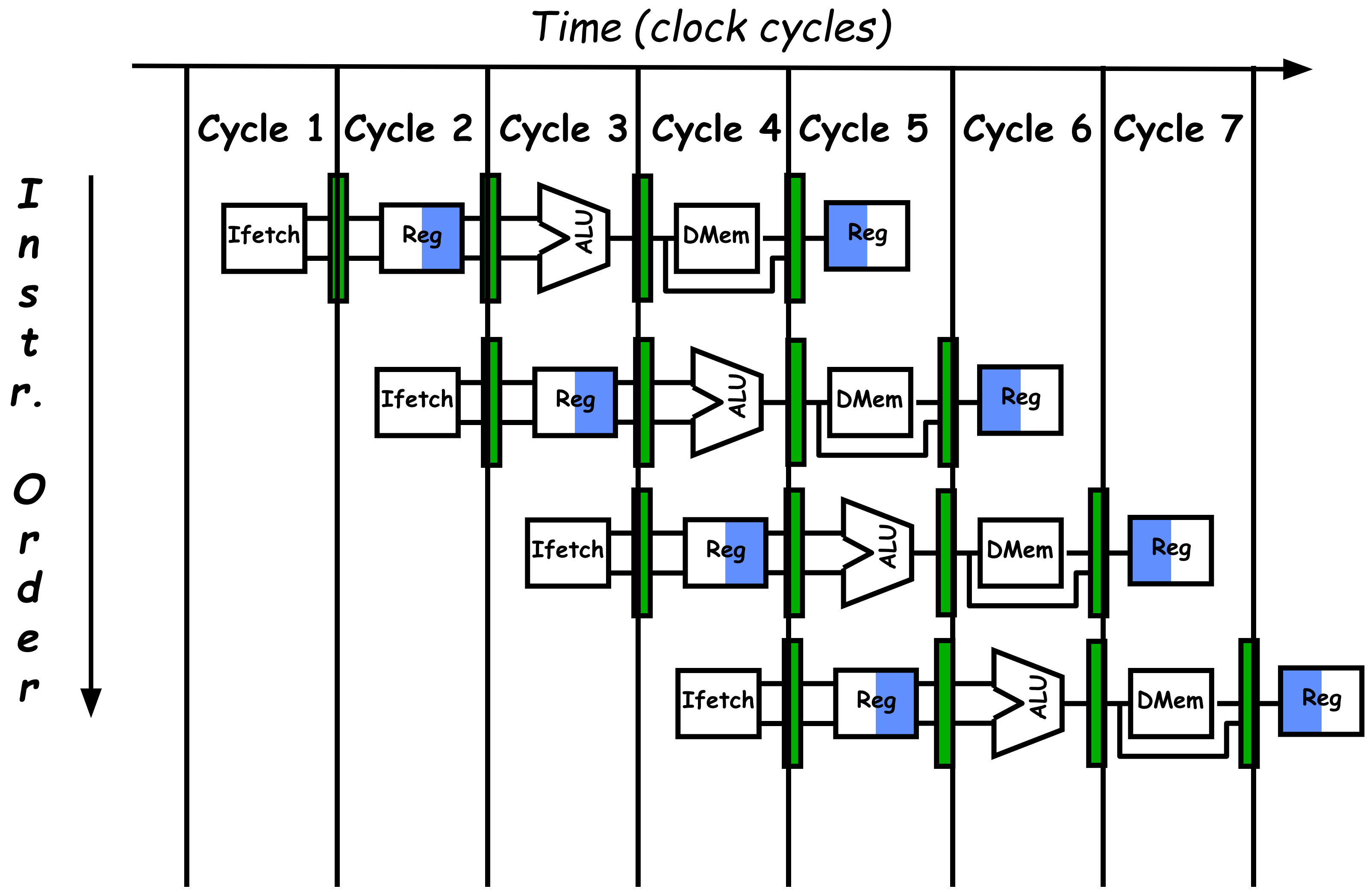


5 Steps of MIPS Datapath



Five-Stage Pipelining

Figure A.2, Page A-8

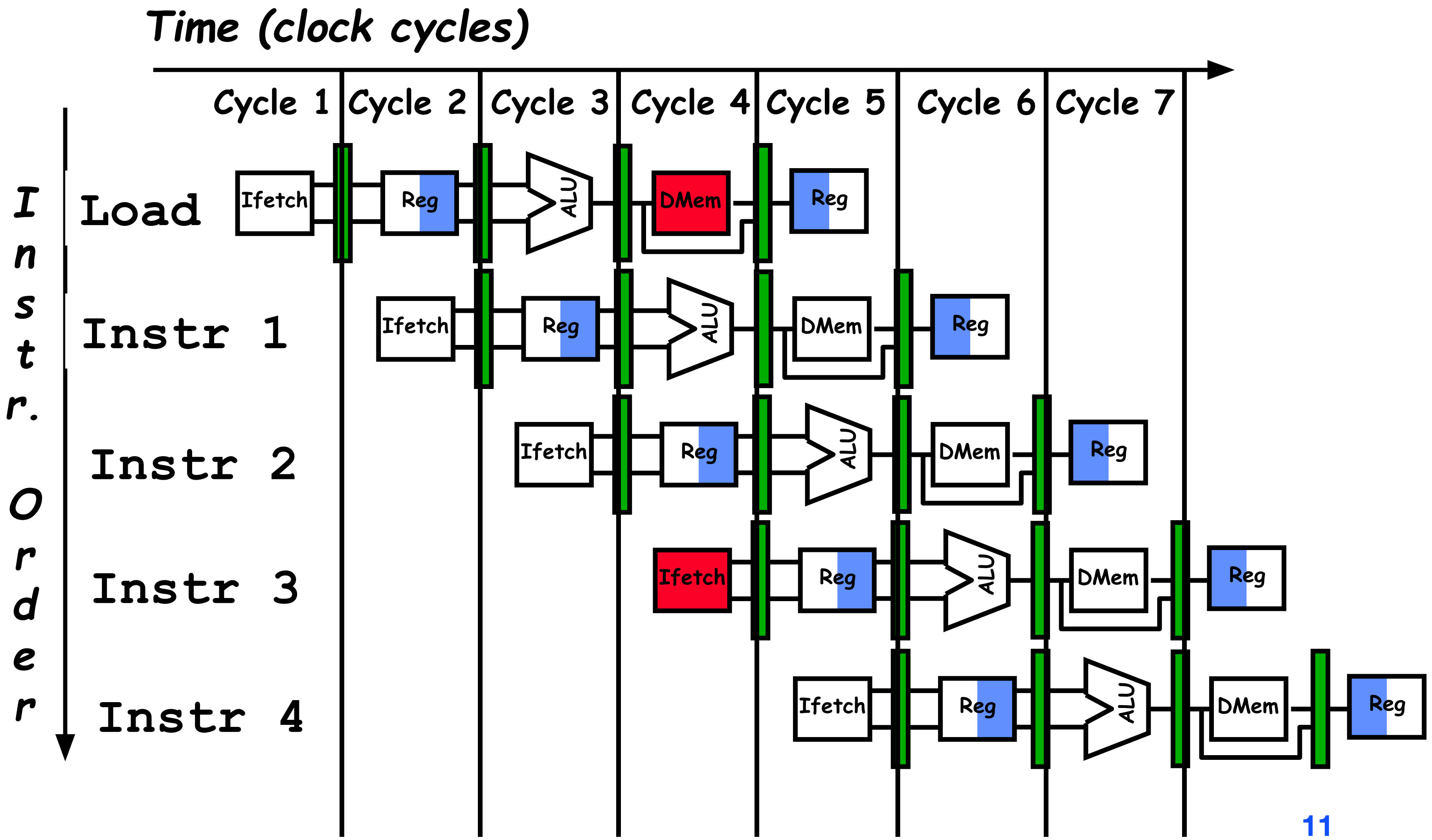


Pipelining is not quite that easy!

- 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).

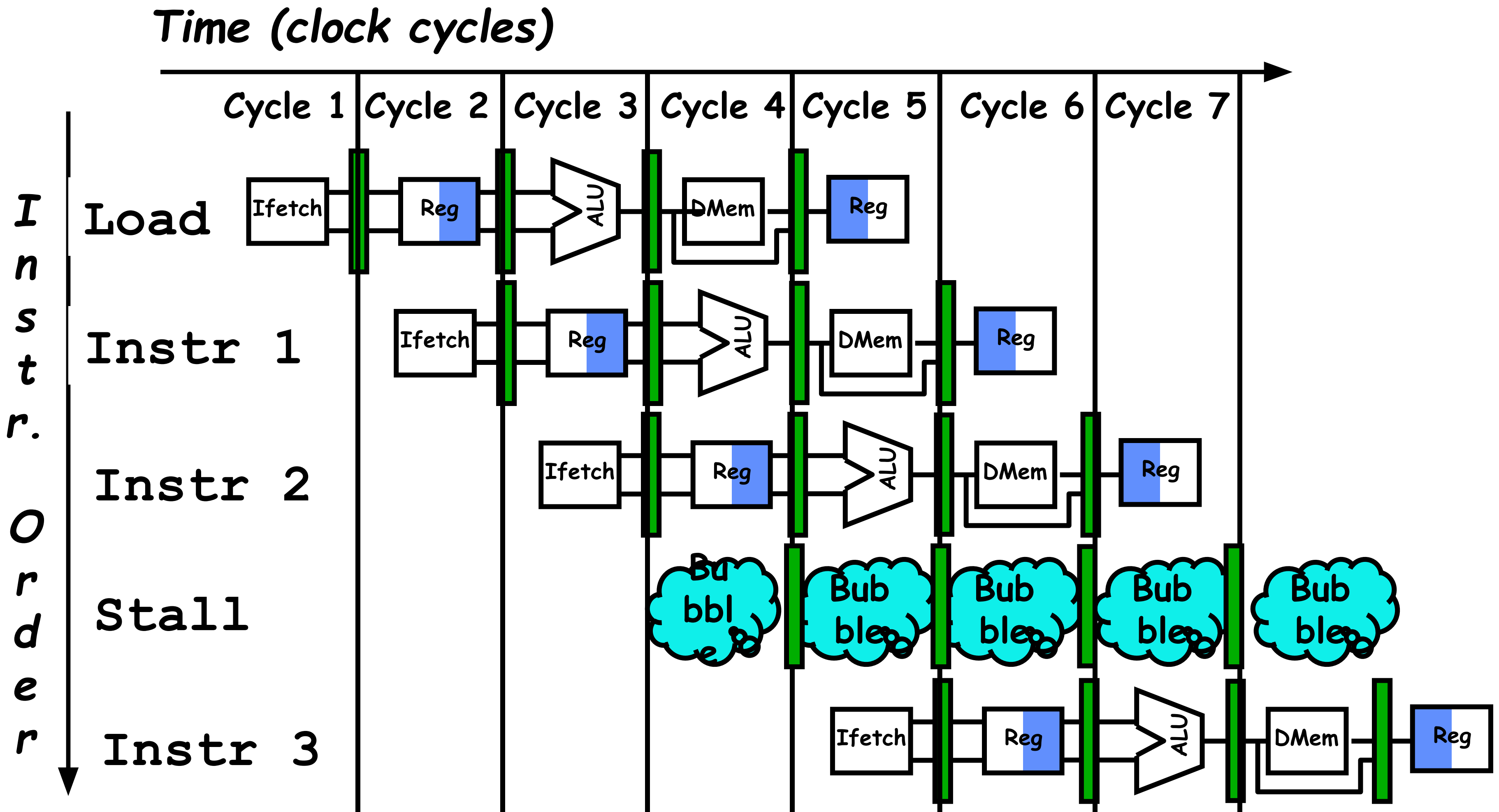
One Memory Port/Structural Hazards

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?

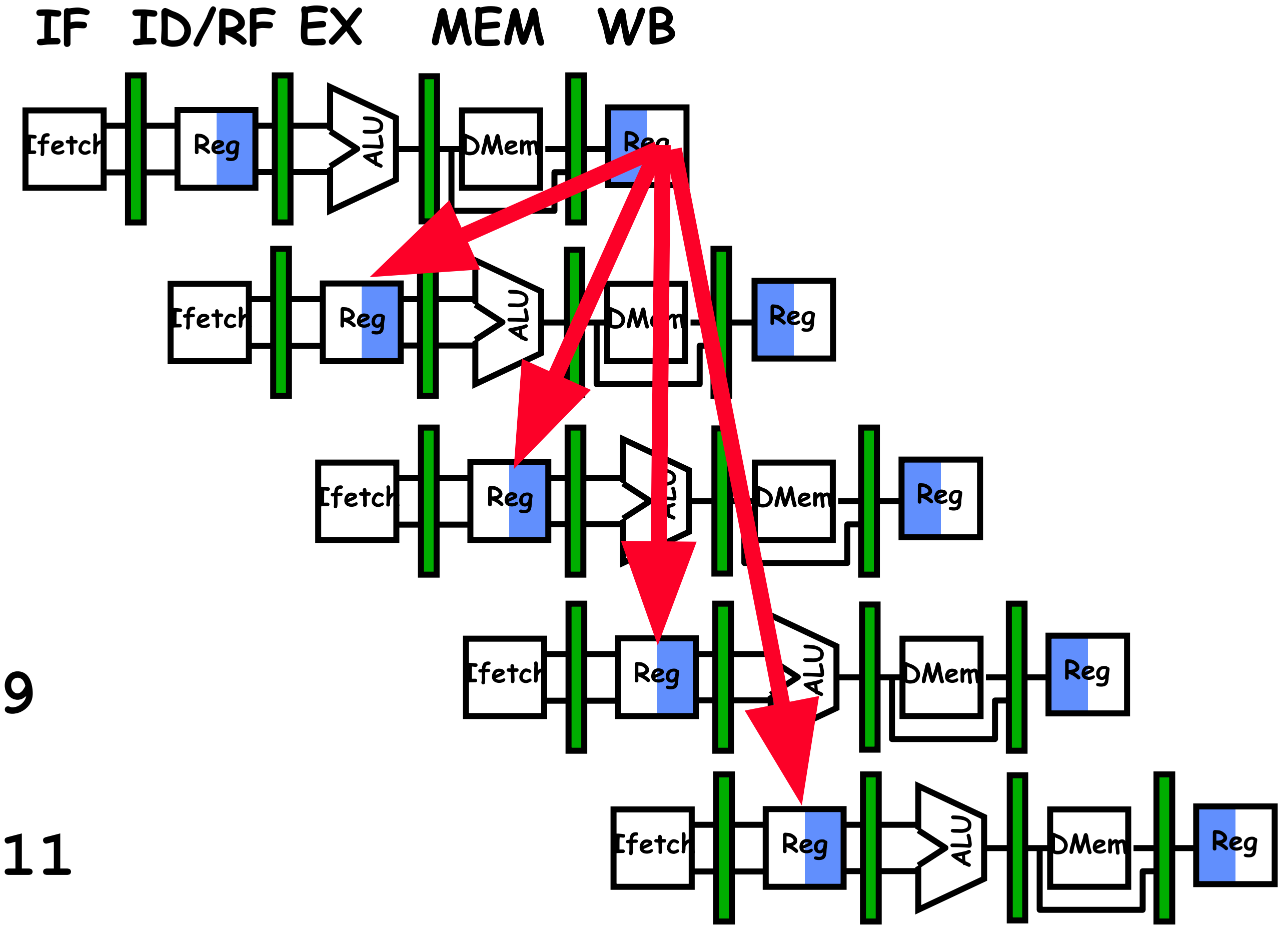
Data Hazard on R1

Figure A.6, Page A-17

Time (clock cycles)

Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or  r8,r1,r9
xor r10,r1,r11
```



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 "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

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
```

- **Called an "anti-dependence" by compiler writers. 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
```

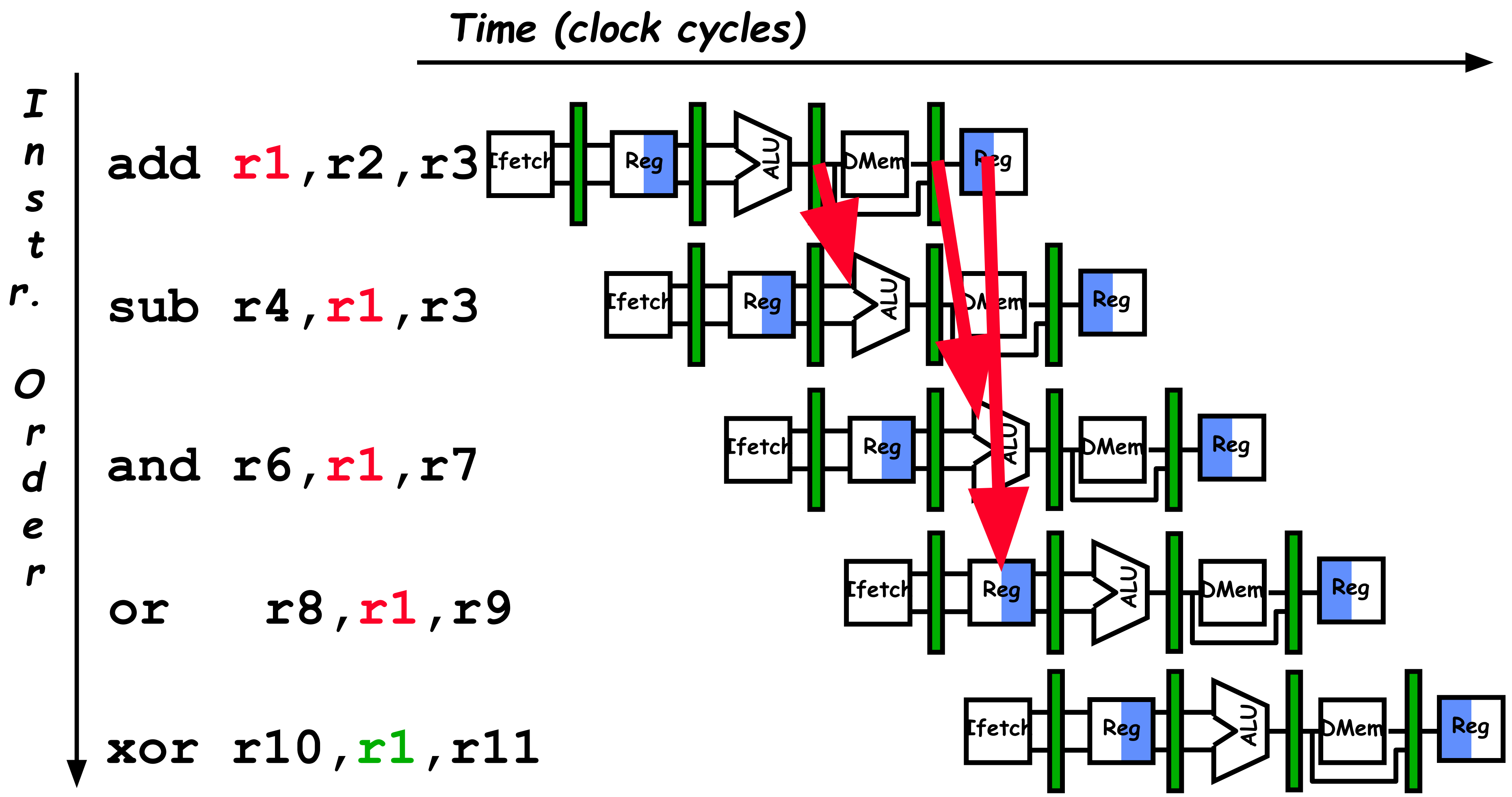
- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

Outline

- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Multi-cycle Operations
- Conclusion

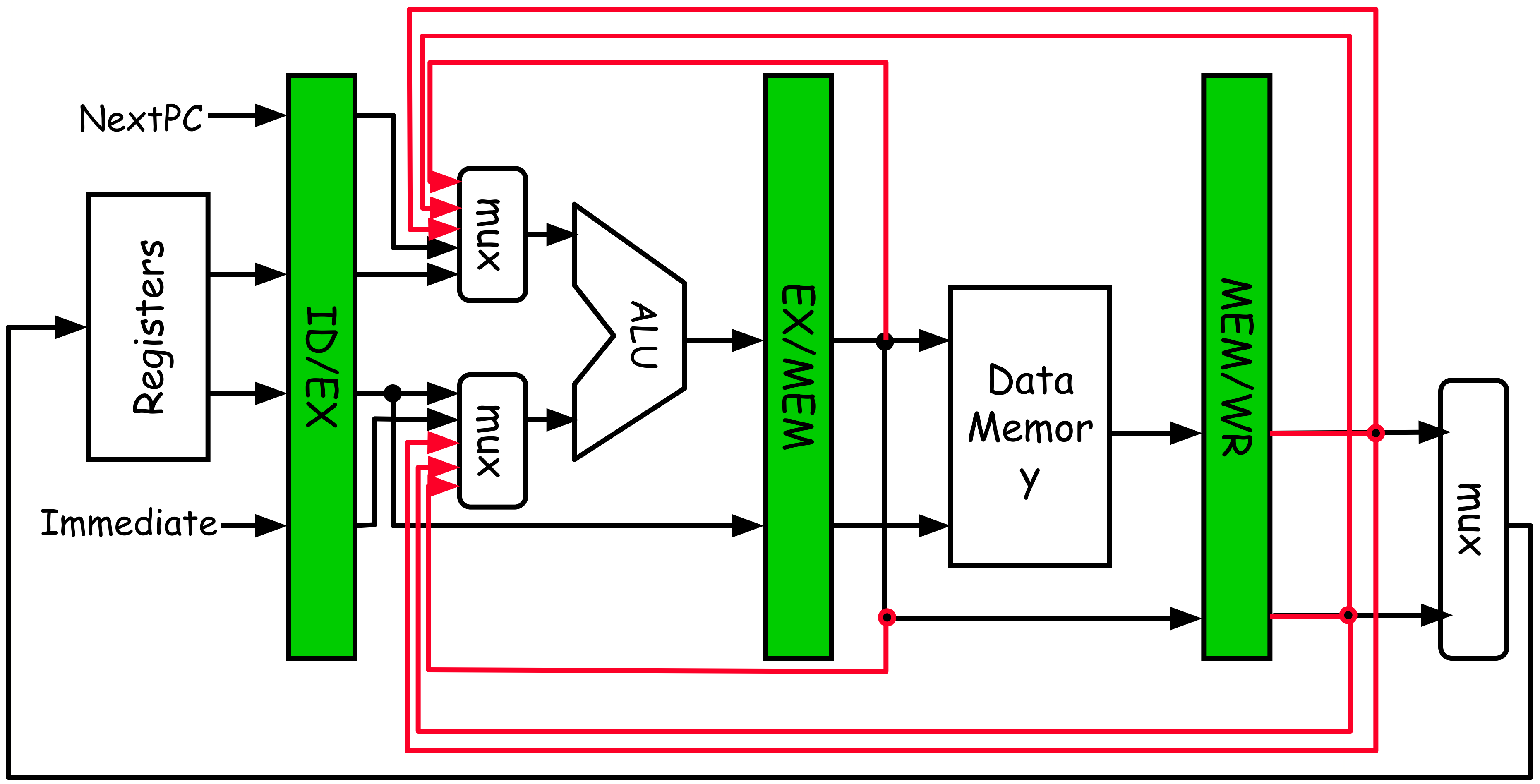
Forwarding to Avoid Data Hazard

Figure A.7, Page A-19



HW Change for Forwarding

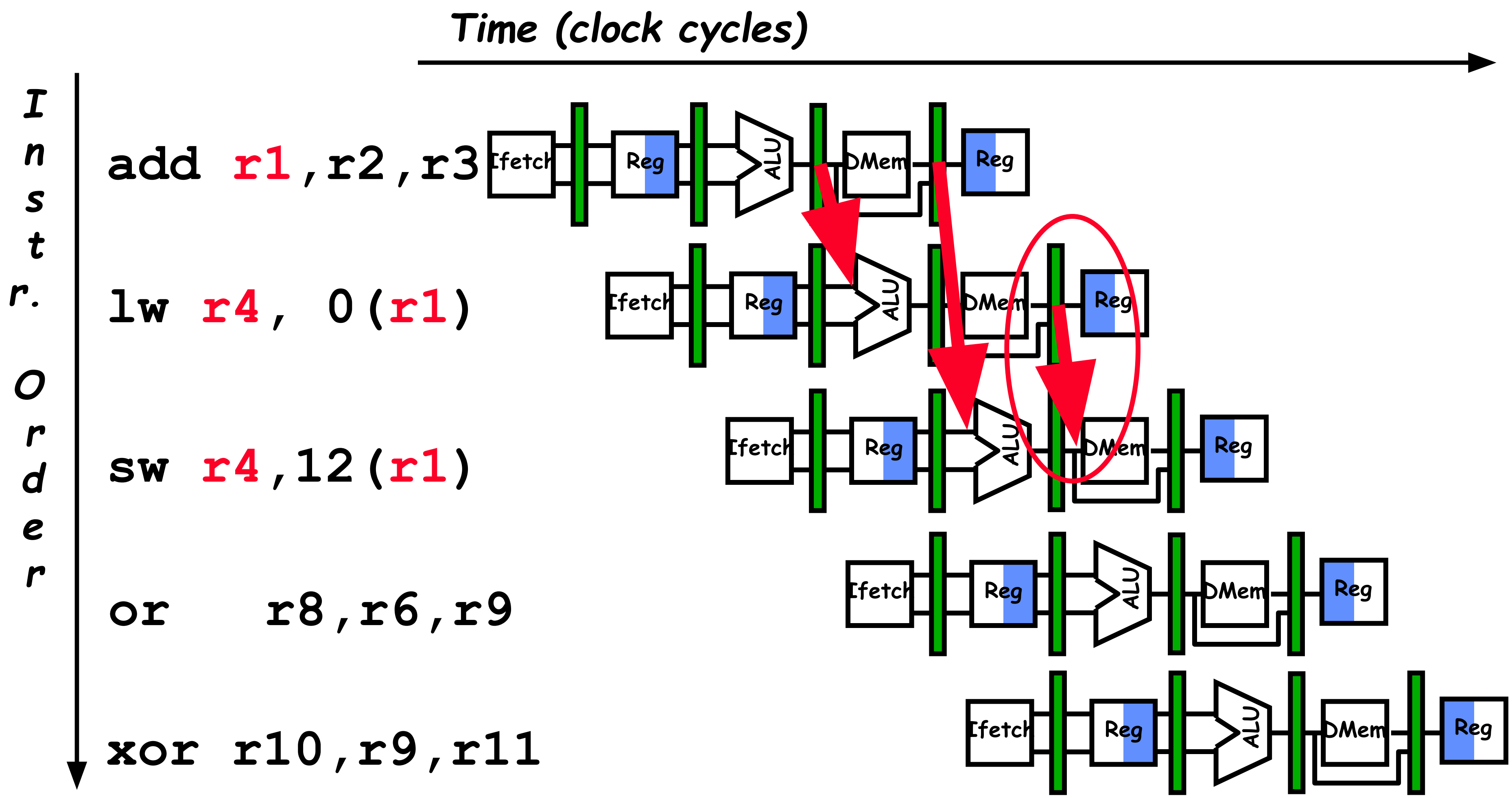
Figure A.23, Page A-37



What circuit detects and resolves this hazard?

Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-20



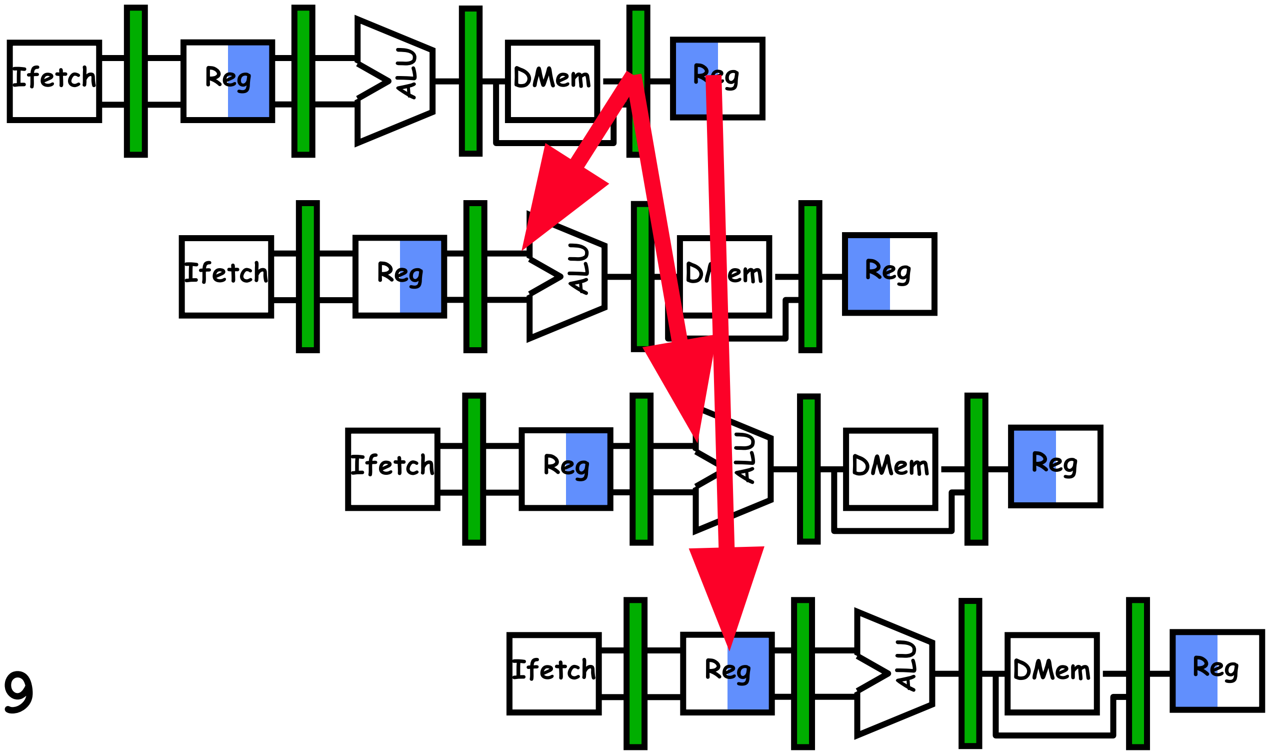
Data Hazard Even with Forwarding

Figure A.9, Page A-21

Time (clock cycles)

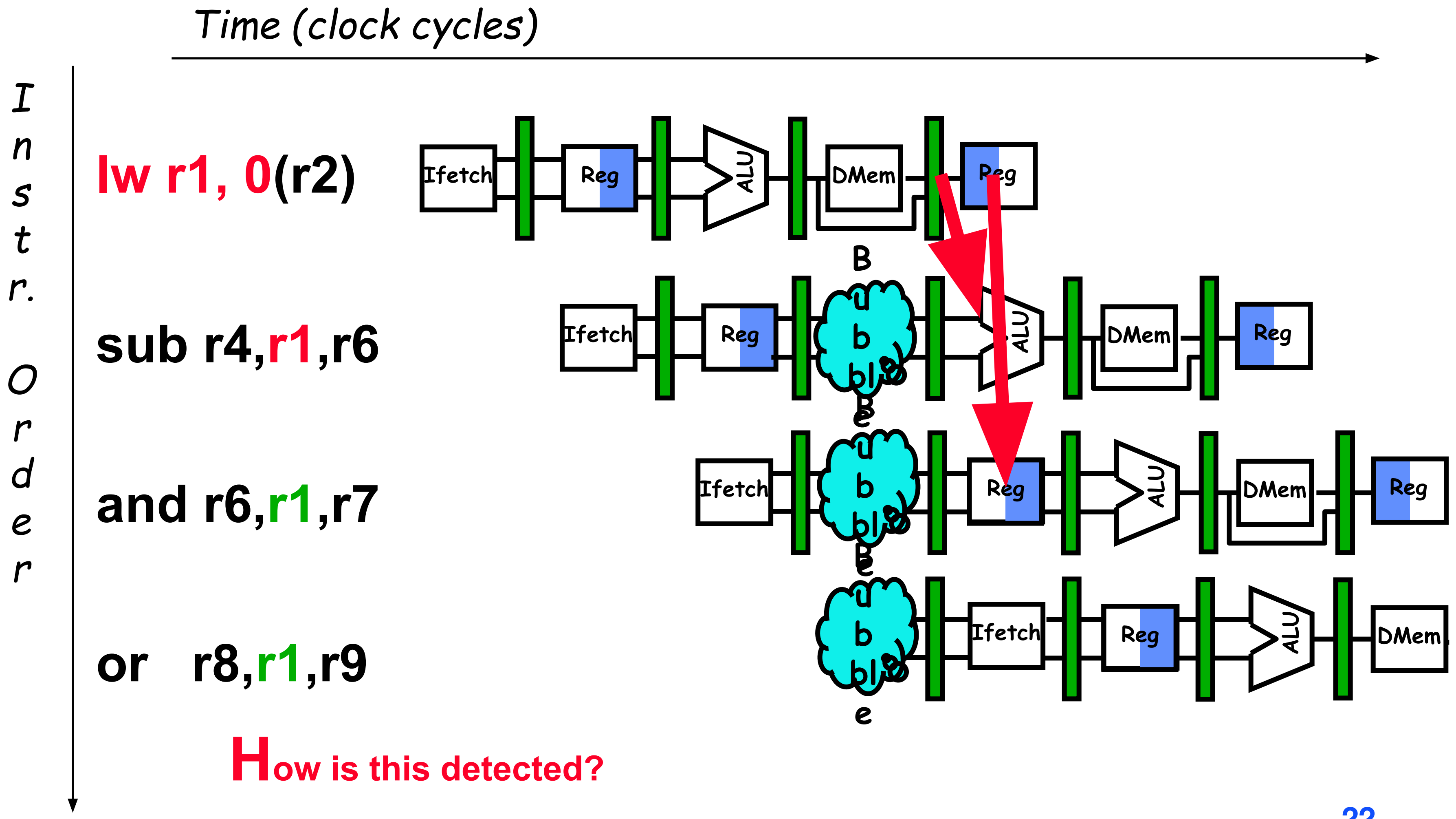
Instr. Order

```
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      Ra,a
LW  Re,e
LW  Rf,f
SUB     Rd,Re,Rf
SW  Rd,d
```

Fast code:

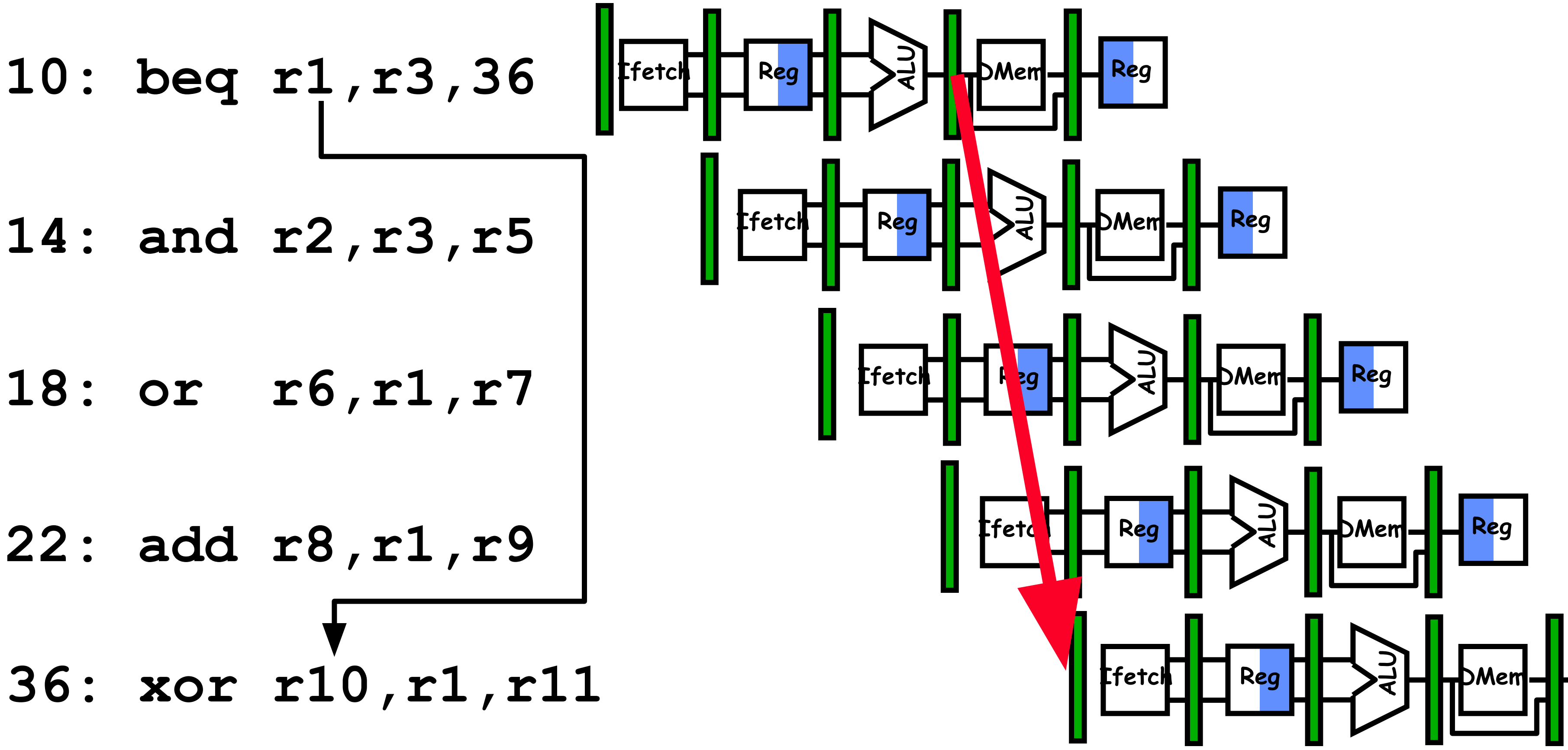
```
LW  Rb,b
LW  Rc,c
LW  Re,e
ADD     Ra,Rb,Rc
LW  Rf,f
SW      Ra,a
SUB     Rd,Re,Rf
SW  Rd,d
```

Compiler optimizes for performance. Hardware checks for safety.

Outline

- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Multi-cycle Operations
- Conclusion

Control Hazard on Branches
Three Stage Stall



What do you do with the 3 instructions in between?

How do you do it?

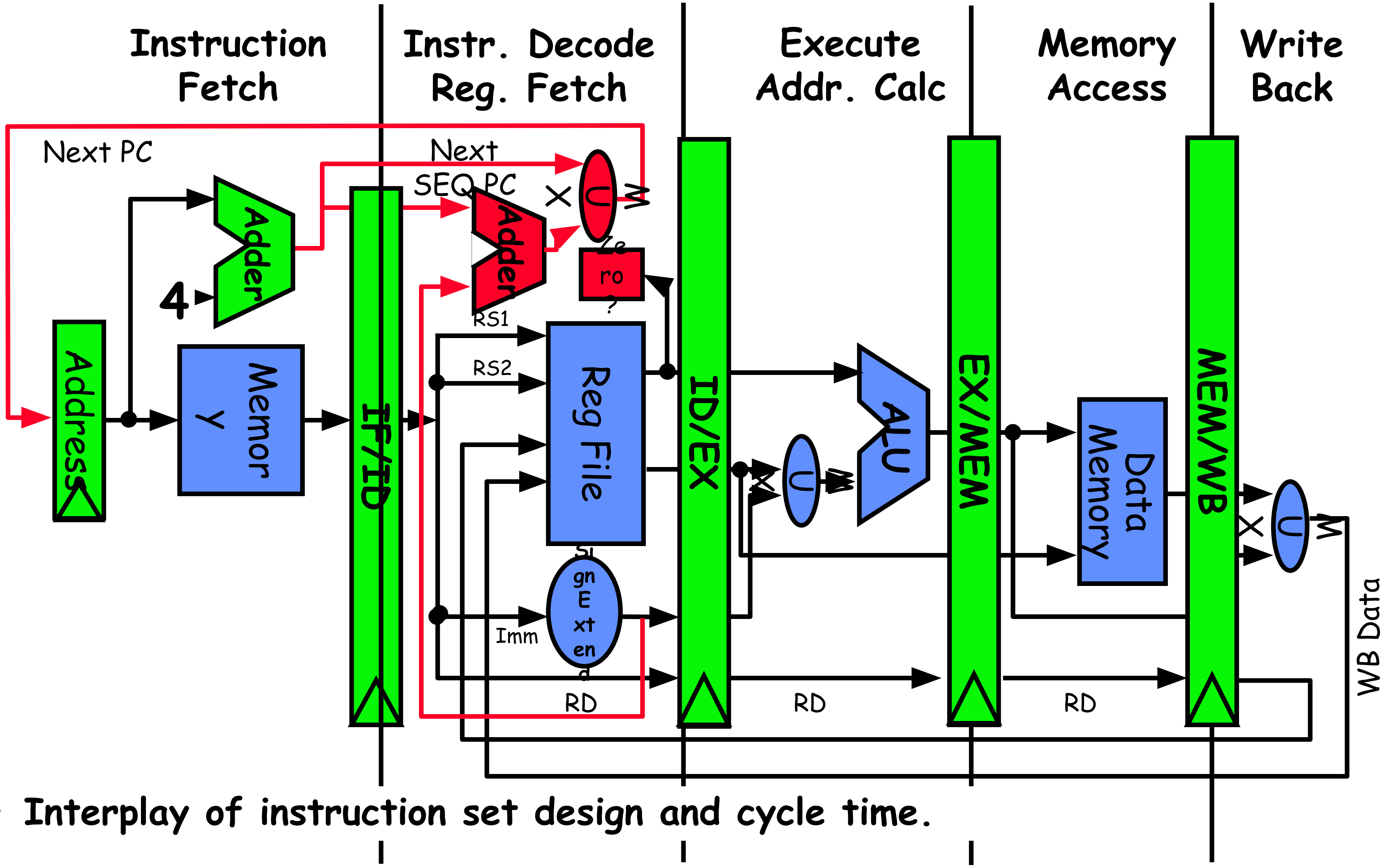
Where is the "commit"?

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => 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 ≠ 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

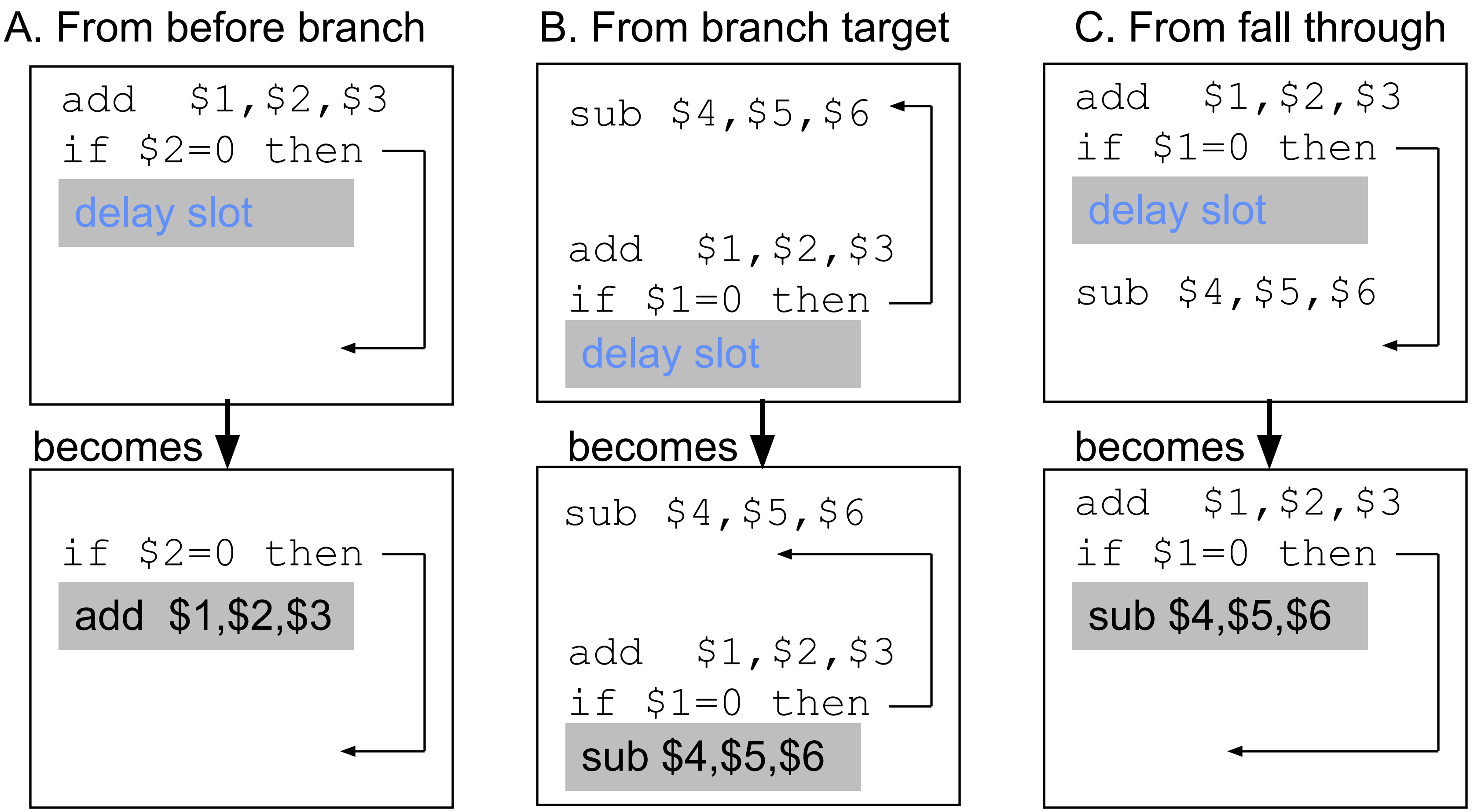
- Define branch to take place **AFTER** a following instruction

`branch instruction`
`sequential successor`$_1$
`sequential successor`$_2$
`........`
`sequential successor`$_n$
`branch target if taken`

Branch delay of length *n*

- 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 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**

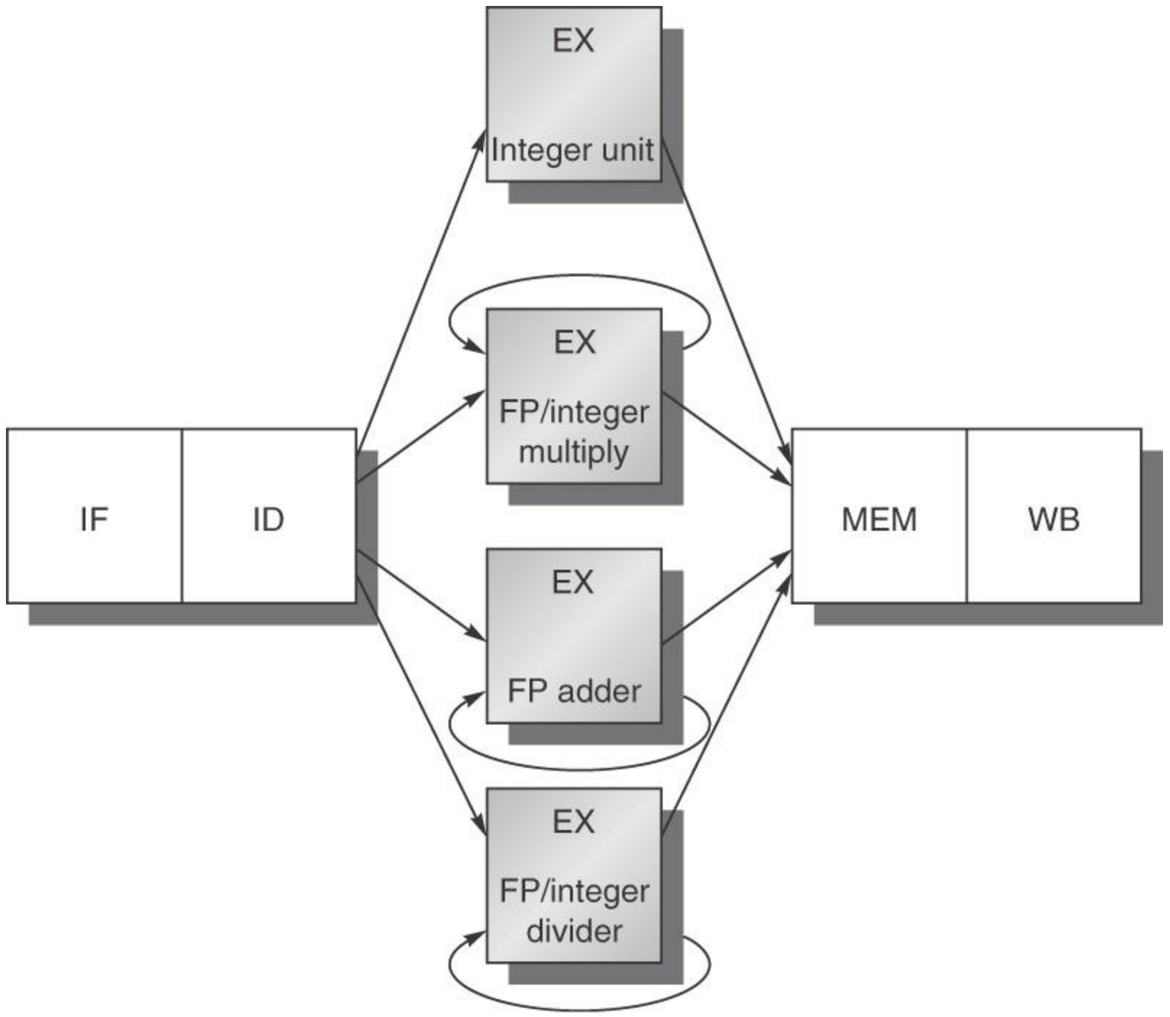
Delayed Branch

- **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

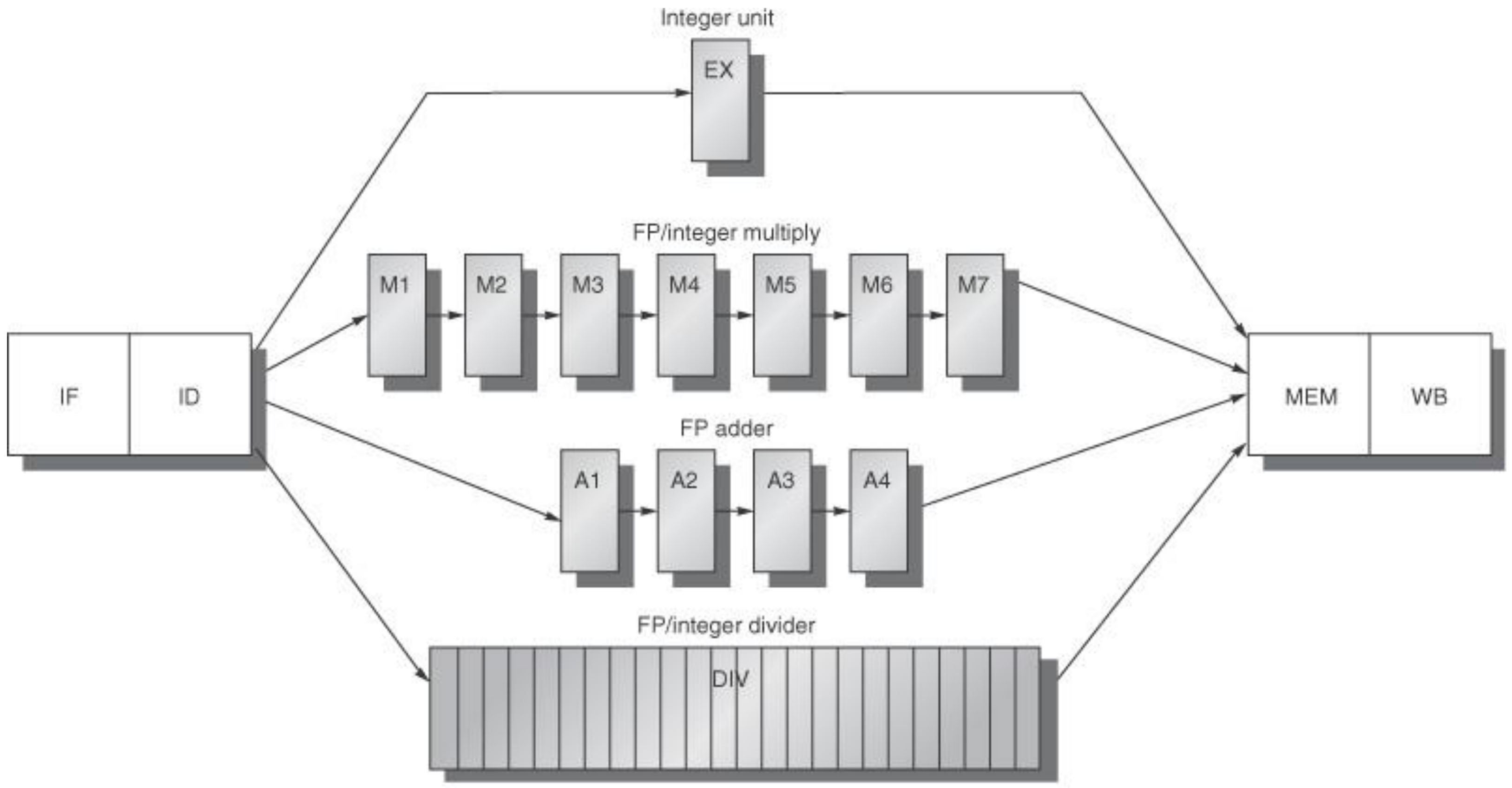
Outline

- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- **Multi-cycle Operations**
- **Conclusion**

Multi-cycle Operations



Multi-cycle Operations - Example



Multi-cycle Operations - Example

Unit	Latency	Initiation Interval
Integer ALU	0	1
Memory	1	1
FP Adder	3	1
FP Multiplier	6	1
FP Divider	24	25

Problems 1/5

1. **Structural hazard on divide**
   - <u>Solution</u>: Detect at ID stage and stall

<u>Example</u>:

`mul.d f0, f4, f6`	F	D	M1	M2	M3	M4	M5	M6	M7	M	W
`...`		F	D	E	M	W					
`...`			F	D	E	M	W				
`add.d f2, f4, f6`				F	D	A1	A2	A3	A4	M	W
`...`					F	D	E	M	W		
`...`						F	D	E	M	W	
`ld.d  f2, 0(r2)`							F	D	E	M	W

Problems 2/5

2. **May have multiple register writes**

   Solution:

   Use shift register to reserve when need to write and stall on conflict

Problems 3/5

3. **There are long stalls due to RAW dependencies**

Example:

```
ld.d    f4, 0(r2)
mul.d   f0, f4, f6      1-cycle stall
add.d   f2, f0, f8      long stall
s.D     f2, 0(r2)       long stall
```

Solution:

Mark registers pending as destinations, then in ID stage stall any instruction that uses the marked registers.

Problems 4/5

4. **WAW hazards are possible**

Example:

```
mul.d  f0, f4, f6
add.d  f0, f6, f8
```

Solution:

Stall the second instruction if its destination register is pending as a destination.

Problems 5/5

5. **Out-of-order completion creates problems in restarting after exceptions**

   Example:

```
div.d   f0, f4, f6  Assume a div by 0 exception
ld.d    f2, 0(r2)   Finishes before div
```

   Solution:

   1) Accept imprecise exceptions

   2) Buffer results and write-back in order.

Outline

- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Multi-cycle Operations
- Conclusion

And In Conclusion: Control and Pipelining

- **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**
- **Multi-cycle operations introduce several problems**

Acknowledgements

• ***Adapted from the slides of Prof. David Patterson and Prof. Gheith Abandah.***