Lecture 05: Pipelining: Basic/ Intermediate Concepts and Implementation

CSE 564 **Computer Architecture** Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

Contents

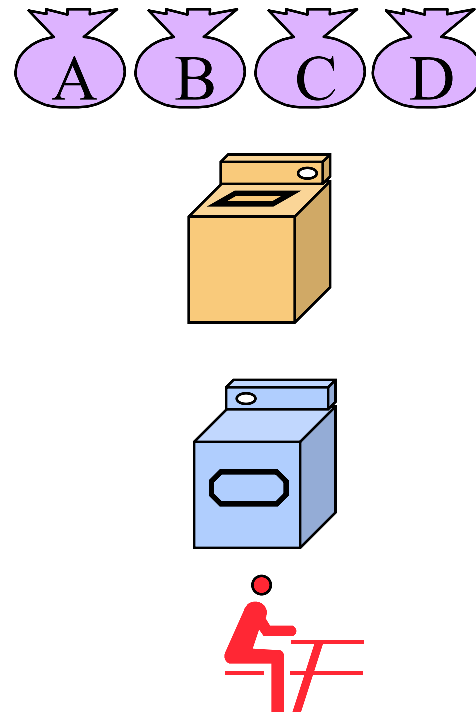
1. **Pipelining Introduction**
2. **The Major Hurdle of Pipelining—Pipeline Hazards**
3. RISC-V ISA and its Implementation

Reading:

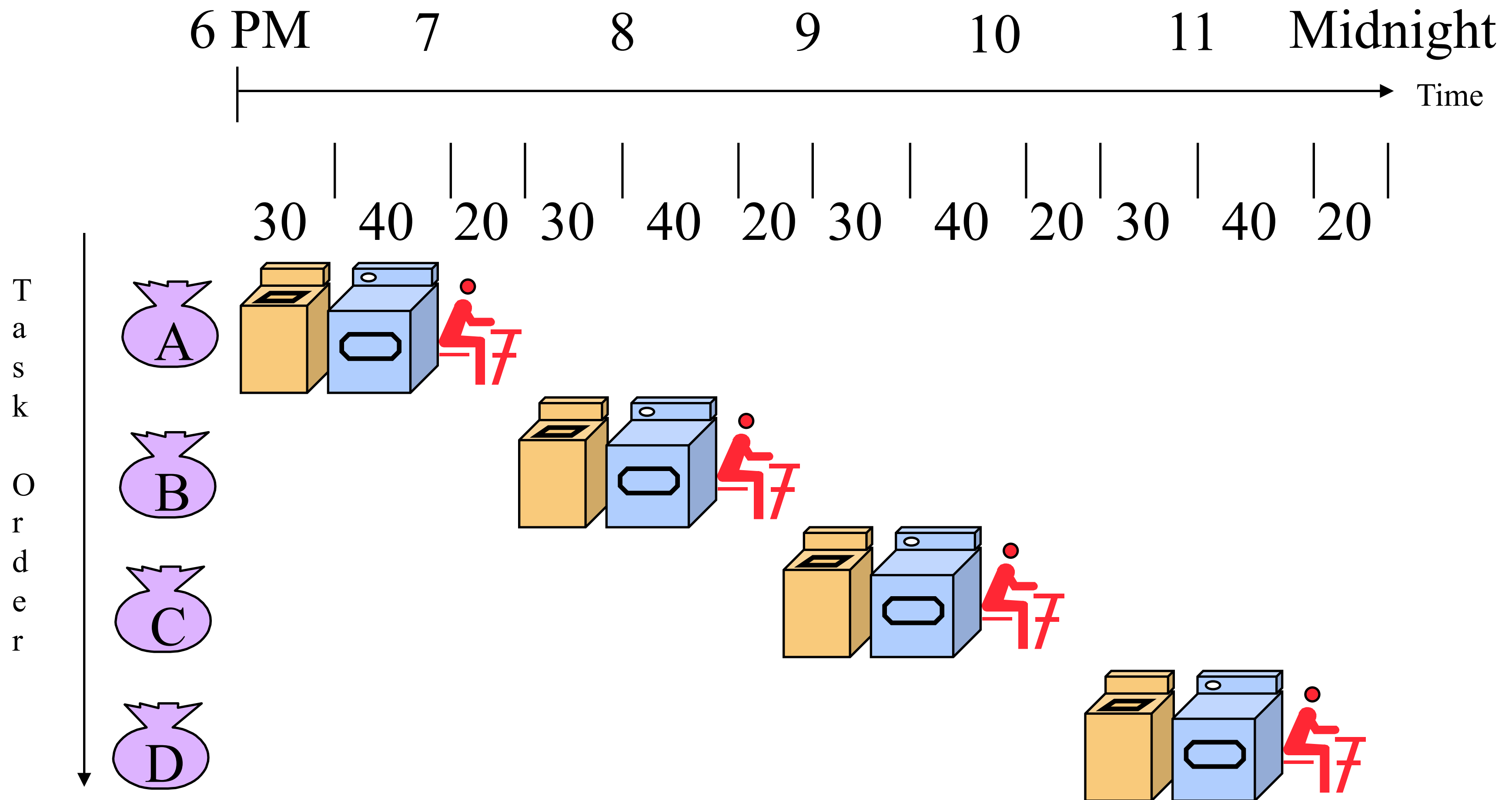
- **Textbook: Appendix C**
- **RISC-V ISA**
- **Chisel Tutorial**

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
- One load: 90 minutes

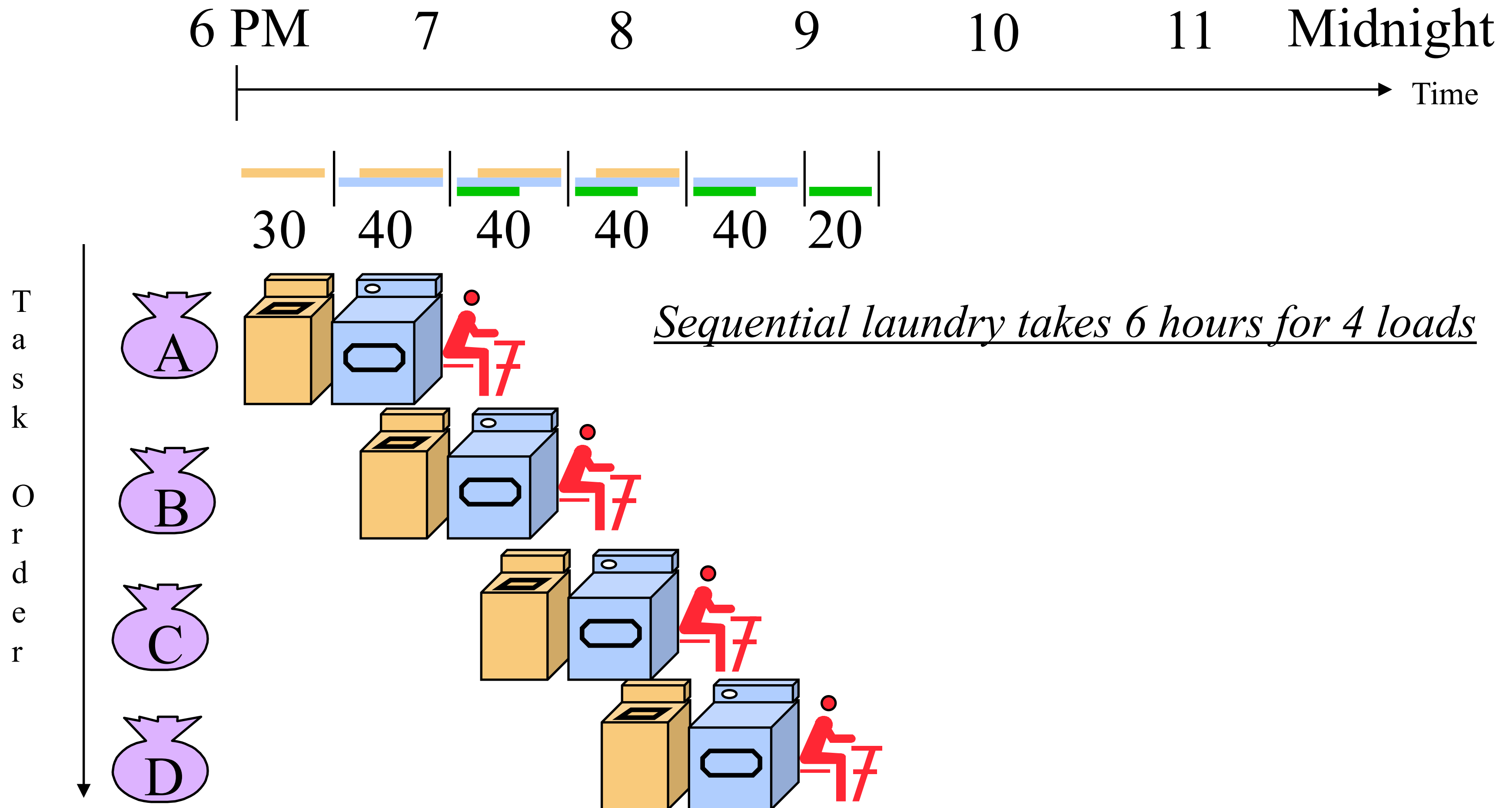


Sequential Laundry



- 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

The Basics of a RISC Instruction Set (1/2)

- *RISC* (Reduced Instruction Set Computer) architecture or *load-store* architecture:
  - All operations on data apply to data in register and typically change the entire register (32 or 64 bits per register).
  - The only operations that affect memory are load and store operation.
  - The instruction formats are few in number with all instructions typically being one size.

† These simple three properties lead to dramatic simplifications in the implementation of pipelining.

```
1          .file     "sum.c"
2          .text
3          .align    2
4          .globl    sum
5          .type     sum, @function
6  sum:
7          add       sp,sp,-48
8          sd        s0,40(sp)
9          add       s0,sp,48
10         sw        a0,-36(s0)
11         sd        a1,-48(s0)
12         fsw       fa2,-40(s0)
13         sw        zero,-24(s0)
14         sw        zero,-20(s0)
15         j         .L2
16 .L3:
17         lw        a5,-20(s0)
18         sll       a5,a5,2
19         ld        a4,-48(s0)
20         add       a5,a4,a5
21         flw       fa4,0(a5)
22         flw       fa5,-40(s0)
23         fmul.s    fa5,fa4,fa5
24         flw       fa4,-24(s0)
25         fadd.s    fa5,fa4,fa5
26         fsw       fa5,-24(s0)
27         lw        a5,-20(s0)
28         addw      a5,a5,1
29         sw        a5,-20(s0)
30 .L2:
31         lw        a4,-20(s0)
32         lw        a5,-36(s0)
33         blt       a4,a5,.L3
```

The Basics of a RISC Instruction Set (2/2)

- MIPS 64 / RISC-V
  - 32 registers, and R0 = 0;
  - Three classes of instructions
    - ALU instruction: add (DADD), subtract (DSUB), and logical operations (such as AND or OR);
    - Load and store instructions:
    - Branches and jumps:

R-format (add, sub, …)

I-format (lw, sw, …)

J-format (j)

RISC Instruction Set

- Every instruction can be implemented in at most 5 clock cycles/ stages
  - Instruction fetch cycle (IF): send PC to memory, fetch the current instruction from memory, and update PC to the next sequential PC by adding 4 to the PC.
  - Instruction decode/register fetch cycle (ID): decode the instruction, read the registers corresponding to register source specifiers from the register file.
  - Execution/effective address cycle (EX): perform Memory address calculation for Load/Store, Register-Register ALU instruction and Register-Immediate ALU instruction.
  - Memory access (MEM): Perform memory access for load/store instructions.
  - Write-back cycle (WB): Write back results to the dest operands for Register-Register ALU instruction or Load instruction.

Classic 5-Stage Pipeline for a RISC

- Each cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.
  - Simple;
  - However, be ensure that the overlap of instructions in the pipeline cannot cause such a conflict. (also called Hazard)

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	**IF**	**ID**	**EX**	**MEM**	**WB**				
Instruction *i+1*		**IF**	**ID**	**EX**	**MEM**	**WB**			
Instruction *i+2*			**IF**	**ID**	**EX**	**MEM**	**WB**		
Instruction *i+3*				**IF**	**ID**	**EX**	**MEM**	**WB**	
Instruction *i+4*					**IF**	**ID**	**EX**	**MEM**	**WB**

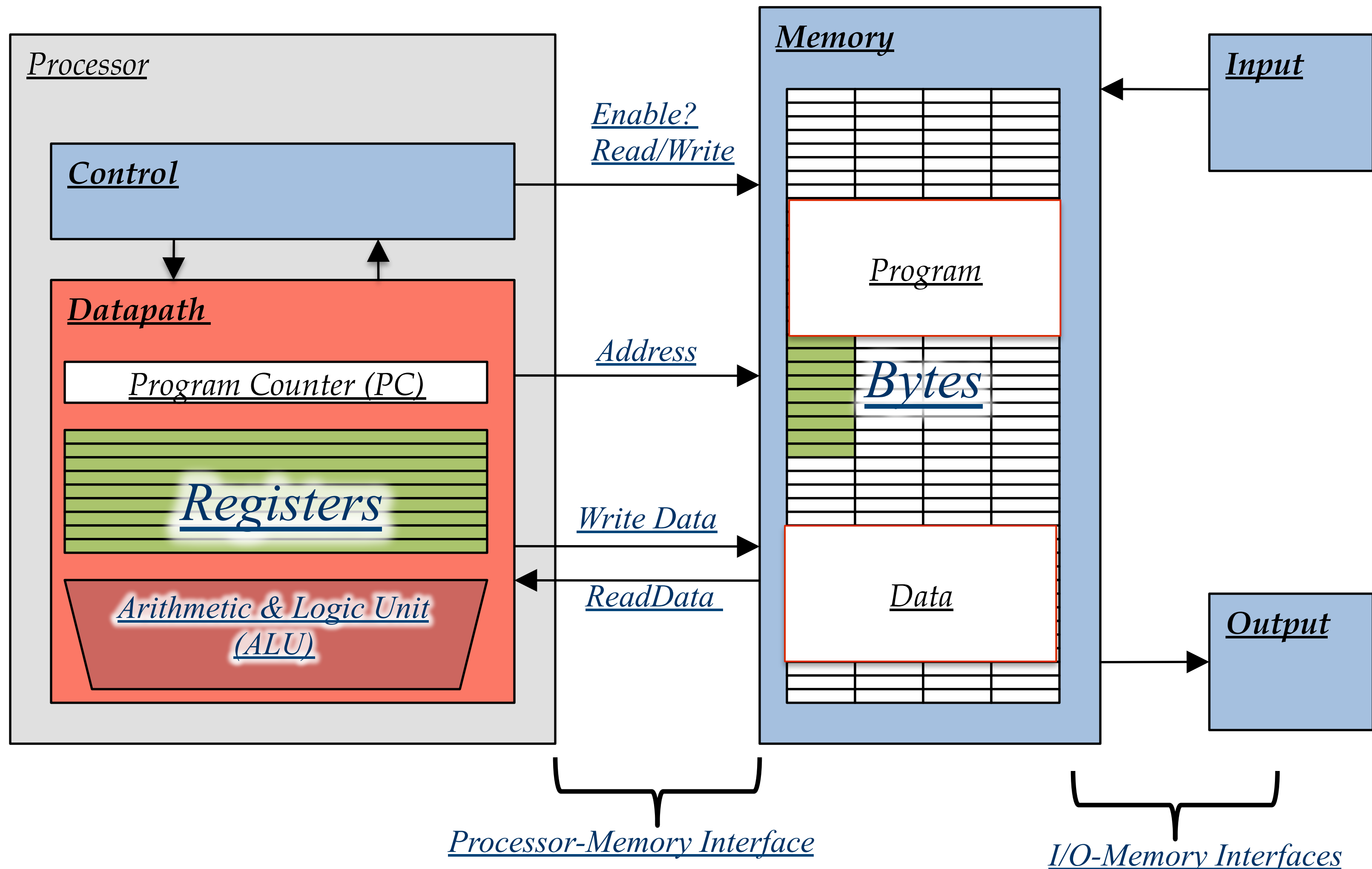
Computer Pipelines

- Pipeline properties
  - Execute billions of instructions, so throughput is what matters.
  - 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.
- The time per instruction on the pipelined processor in ideal conditions is equal to,

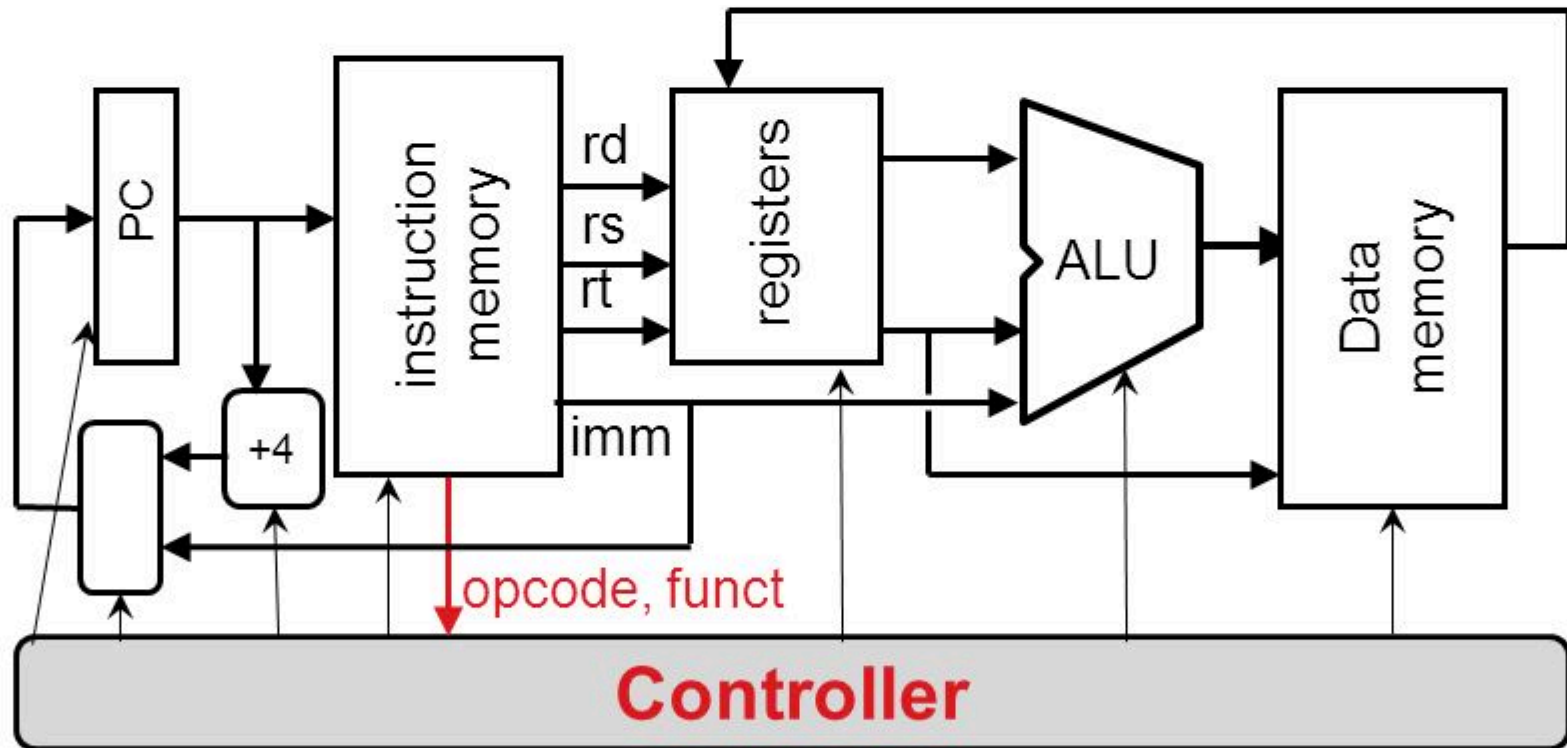
$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stage}}$$

  - However, the stages may not be perfectly balanced.
  - Pipelining yields a reduction in the average execution time per instruction.

Review: Components of a Computer

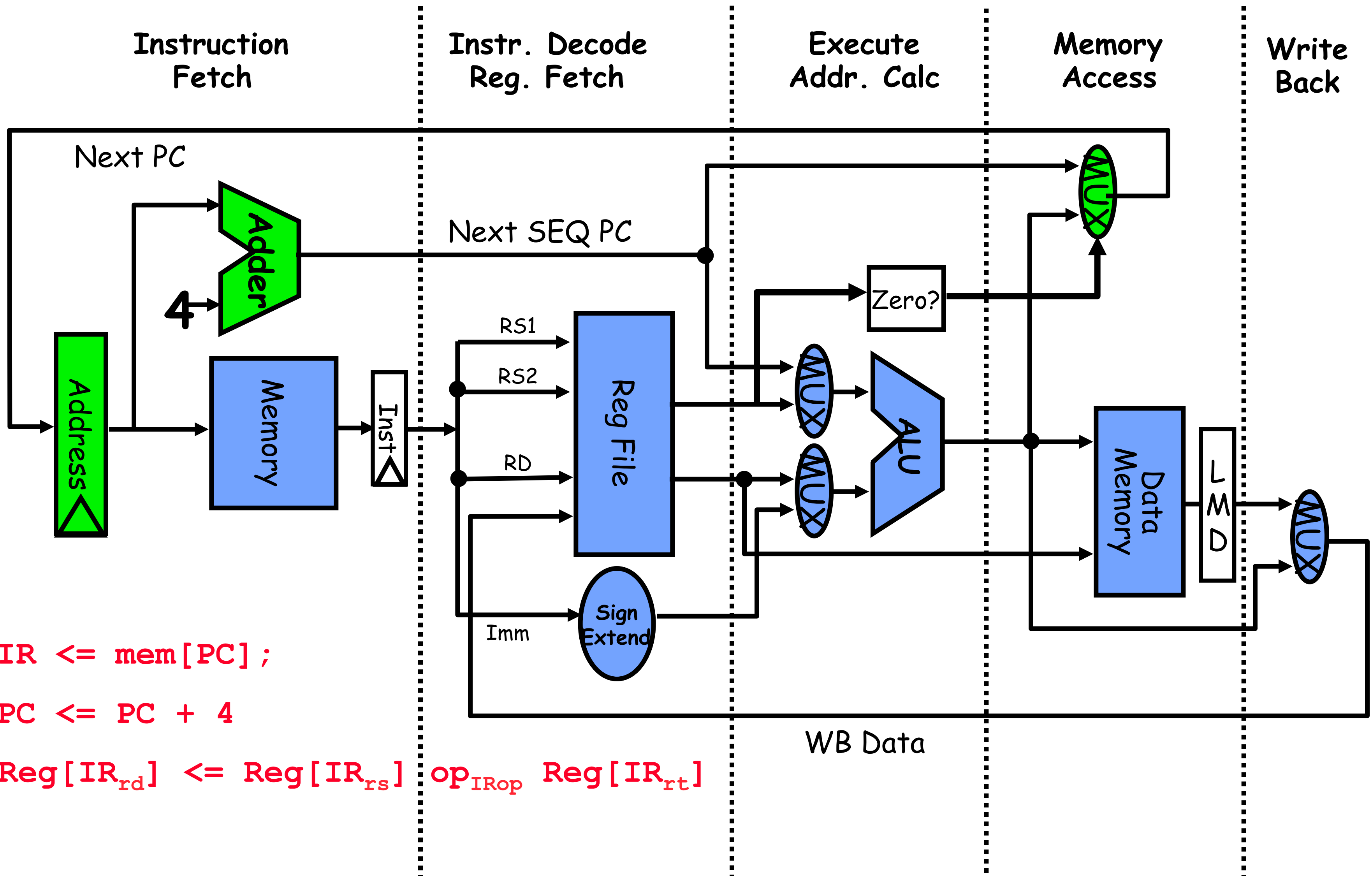


CPU and Datapath vs Control



- 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

5 Stages of MIPS Pipeline

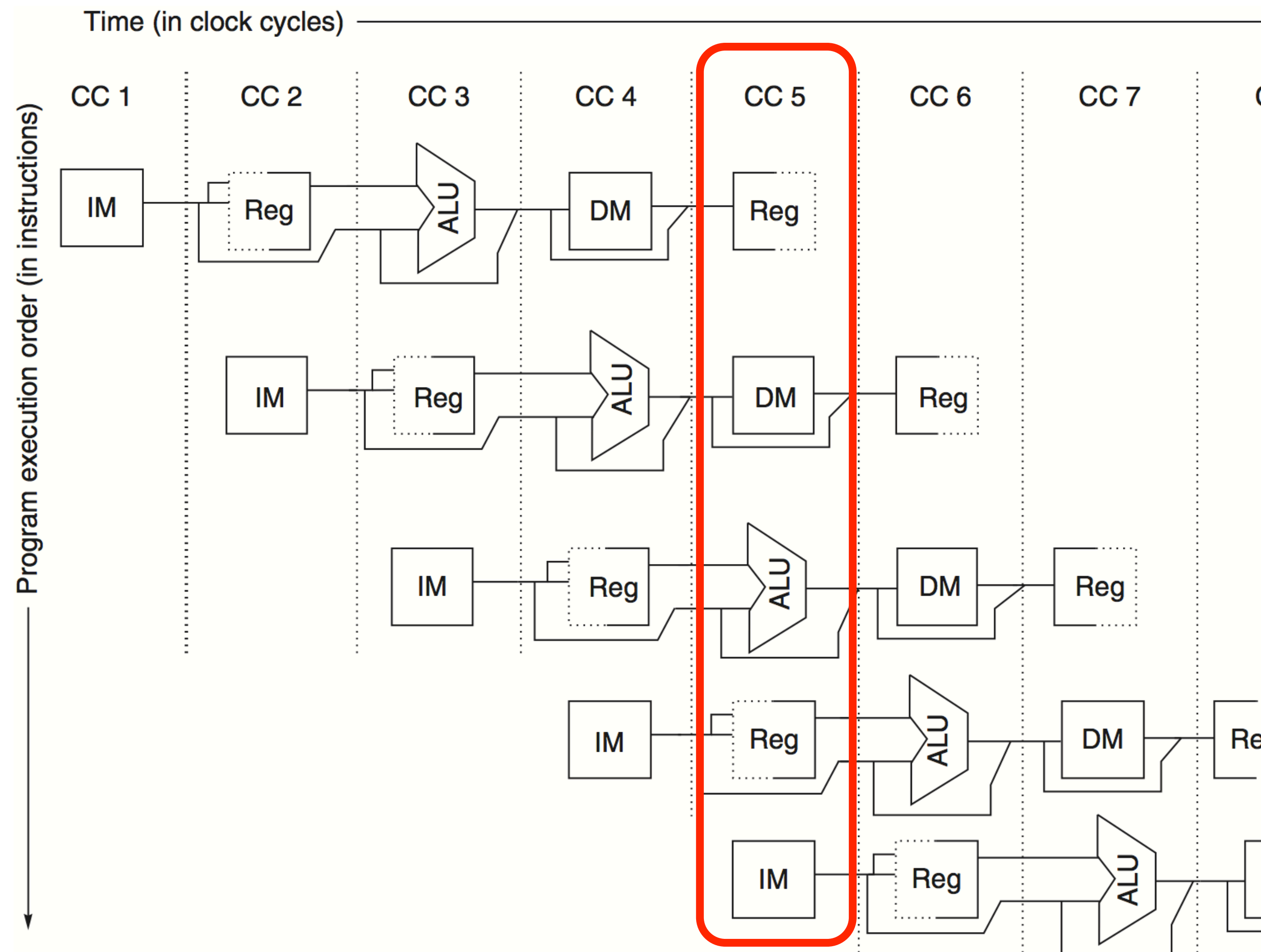


Making RISC Pipelining Real

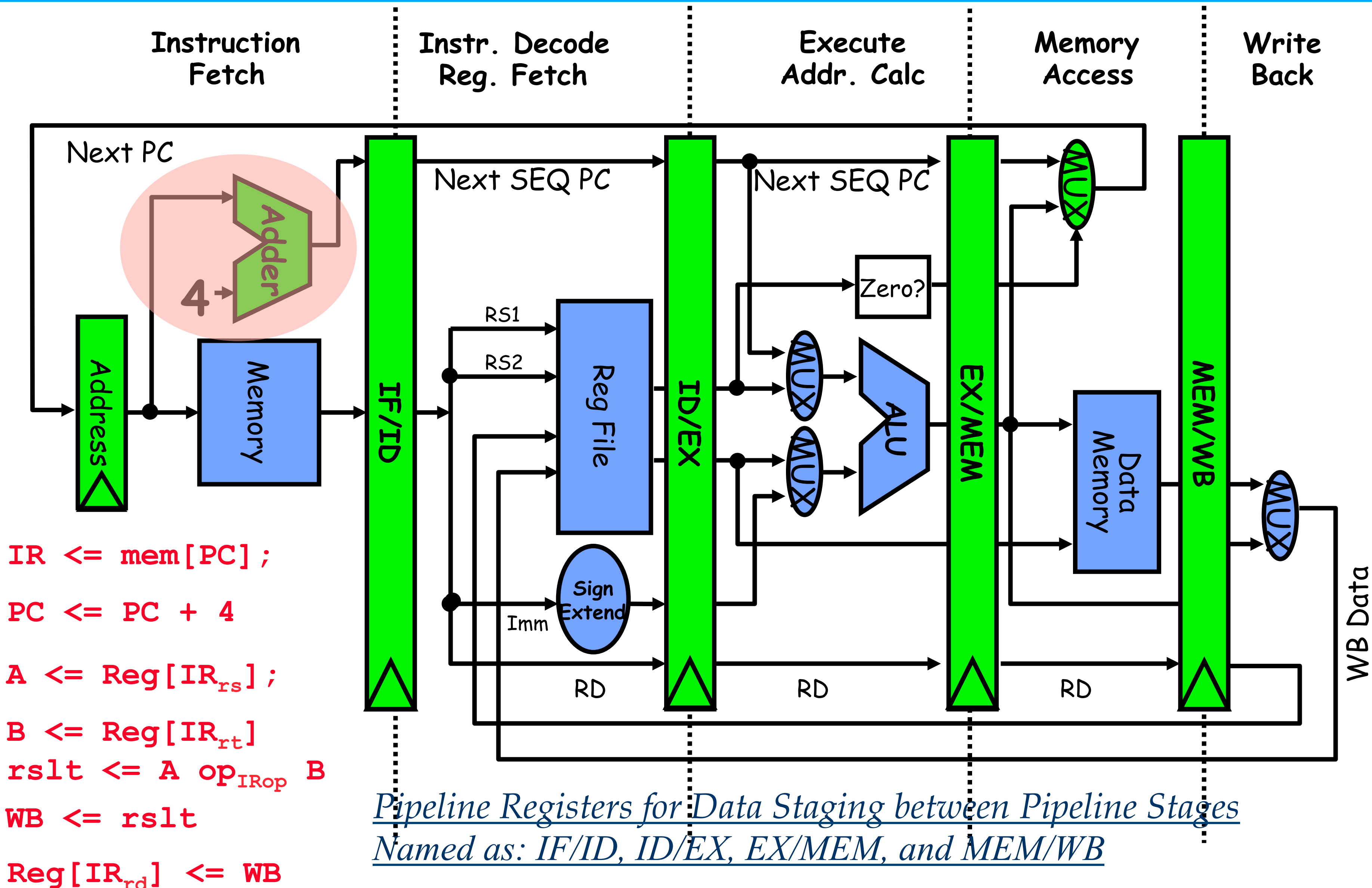
- Function units used in different cycles
  - Hence we can overlap the execution of multiple instructions
- Important things to make it real
  - Separate instruction and data memories, e.g. I-cache and D-cache, banking
    - Eliminate a conflict for accessing a single memory.
  - The Register file is used in the two stages (two R and one W every cycle)
    - Read from register in ID (second half of CC), and write to register in WB (first half of CC).
  - PC
    - Increment and store the PC every clock, and done it during the IF stage.
    - A branch does not change the PC until the ID stage (have an adder to compute the potential branch target).
  - Staging data between pipeline stages
    - Pipeline register

Pipeline Datapath

- Register files in ID and WB stage
  - Read from register in ID (second half of CC), and write to register in WB (first half of CC).
- IM and DM

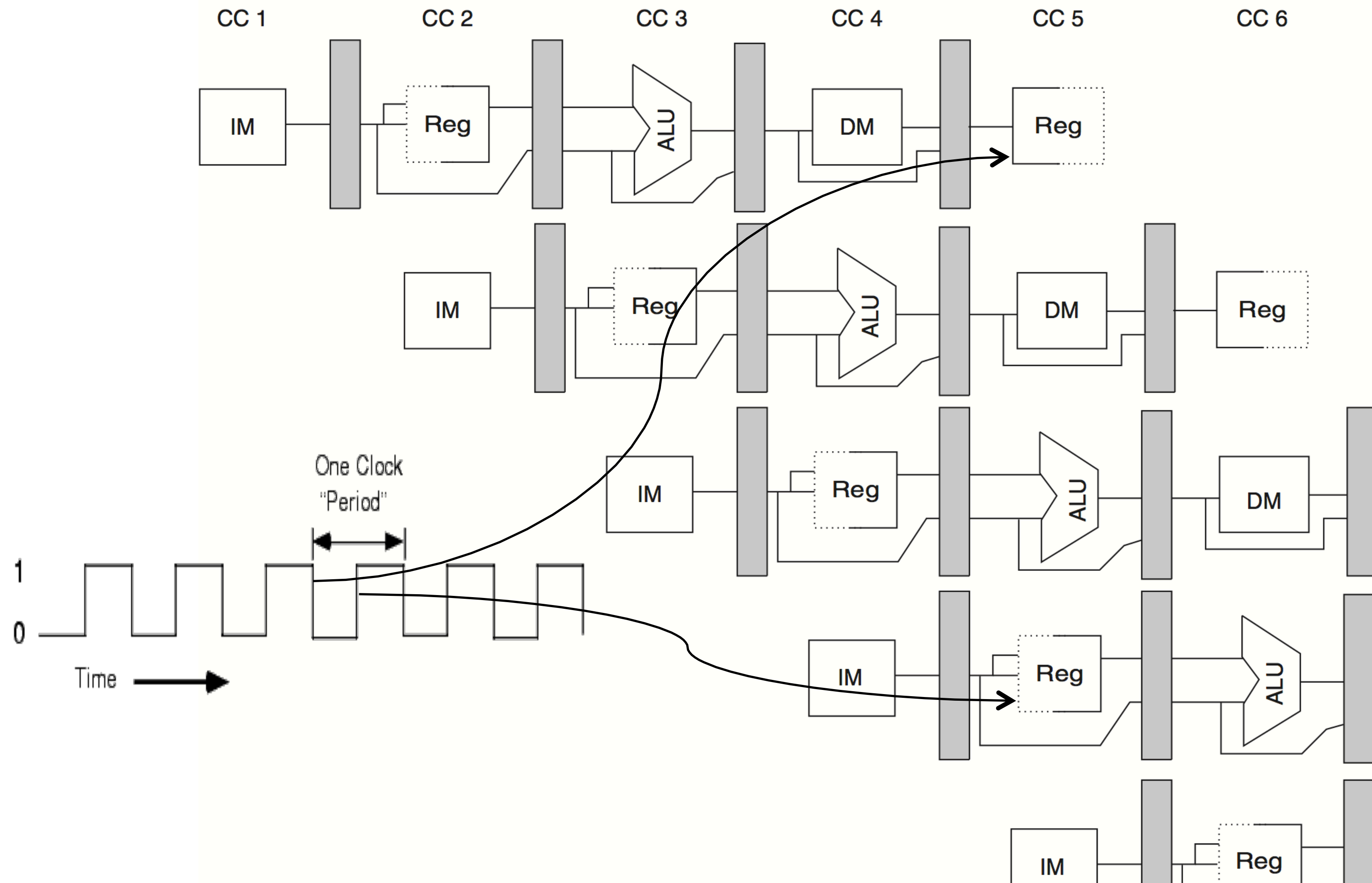


Pipeline Registers



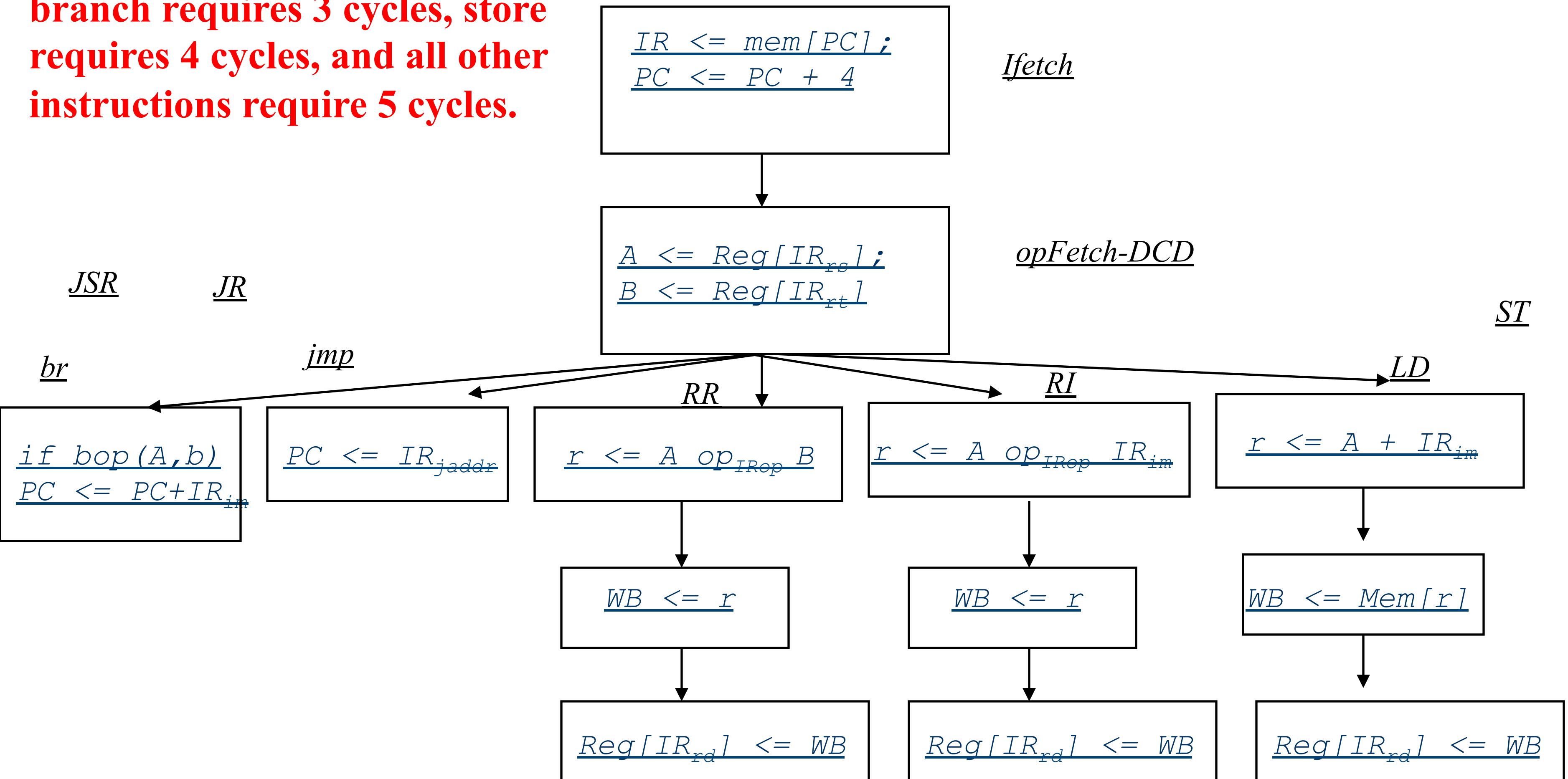
Pipeline Registers

- **Edge-triggered** property of register is critical



Inst. Set Processor Controller

branch requires 3 cycles, store requires 4 cycles, and all other instructions require 5 cycles.



Events on Every Pipeline Stage

Stage	Any Instruction		
IF	IF/ID.IR ← MEM[PC]; IF/ID.NPC ← PC+4 PC ← if ((EX/MEM.opcode=branch) & EX/MEM.cond) {EX/MEM.ALUoutput} else {PC + 4}		
ID	ID/EX.A ← Regs[IF/ID.IR[Rs]]; ID/EX.B ← Regs[IF/ID.IR[Rt]] ID/EX.NPC ← IF/ID.NPC; ID/EX.Imm ← extend(IF/ID.IR[Imm]); ID/EX.Rw ← IF/ID.IR[Rt or Rd]		
	ALU Instruction	Load / Store	Branch
EX	EX/MEM.ALUoutput ← ID/EX.A func ID/EX.B, or EX/MEM.ALUoutput ← ID/EX.A op ID/EX.Imm	EX/MEM.ALUoutput ← ID/EX.A + ID/EX.Imm EX/MEM.B ← ID/EX.B	EX/MEM.ALUoutput ← ID/EX.NPC + (ID/EX.Imm << 2) EX/MEM.cond ← br condition
MEM	MEM/WB.ALUoutput ← EX/MEM.ALUoutput	MEM/WB.LMD ← MEM[EX/MEM.ALUoutput] or MEM[EX/MEM.ALUoutput] ← EX/MEM.B	
WB	Regs[MEM/WB.Rw] ← MEM/WB.ALUOutput	For load only: Regs[MEM/WB.Rw] ← MEM/WB.LMD	

Pipelining Performance (1/2)

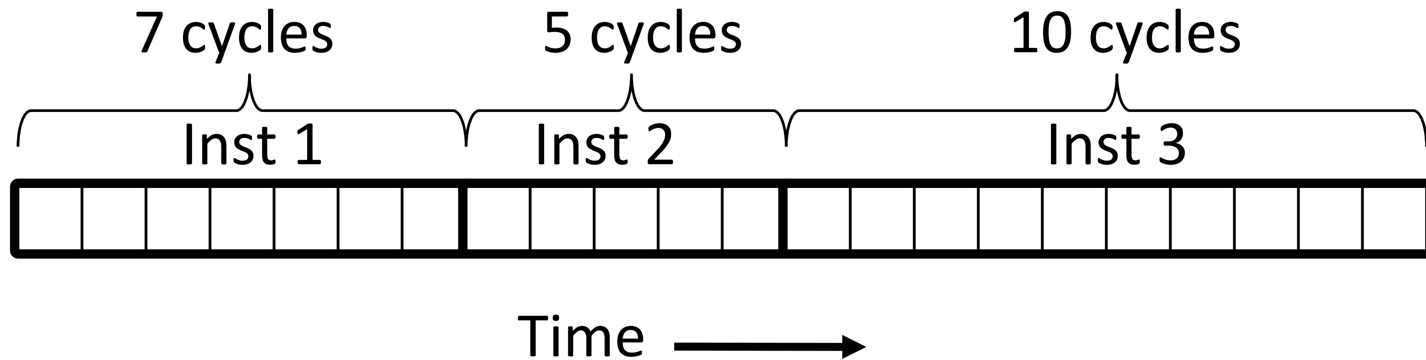
- Pipelining increases throughput, not reduce the execution time of an individual instruction.
  - In face, slightly increases the execution time (an instruction) due to overhead in the control of the pipeline.
  - Practical depth of a pipeline is limits by increasing execution time.
- Pipeline overhead
  - Unbalanced pipeline stage;
  - Pipeline stage overhead;
  - Pipeline register delay;
  - Clock skew.

Processor Performance

$$CPU\ \text{Time} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology, and ISA
- **Cycles per instructions (CPI) depends on ISA and µarchitecture**
- Time per cycle depends upon the µarchitecture and base technology

CPI for Different Instructions



Total clock cycles = 7+5+10 = 22

Total instructions = 3

CPI = 22/3 = 7.33

CPI is always an average over a large number of instructions

Pipeline Performance (2/2)

- Example 1 (p.C-10): Consider the unpipelined processor in previous section. Assume that it has a 1ns clock cycle and that it uses 4 cycles for ALU operations and branches, and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?
- Answer

The average instruction execution time on the unpipelined processor is

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI} \\ &= 1 \text{ ns} \times \left((40\% + 20\%) \times 4 + 40\% \times 5 \right) \\ &= 1 \text{ ns} \times 4.4 = 4.4 \text{ ns}\end{aligned}$$

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{4.4 \text{ ns}}{1.2 \text{ ns}} = 3.7 \text{ times}$$

† In the pipeline, the clock must run at the speed of the slowest stage plus overhead, which will be 1+0.2 ns.

Performance with Pipeline Stall (1/2)

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

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

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

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

$$= 1 + \text{Pipelined stall clock cycles per instruction}$$

Performance with Pipeline Stall (2/2)

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

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

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\Rightarrow \text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

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

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

Pipelining speedup is proportional to the pipeline depth and 1/(1+ stall cycles)

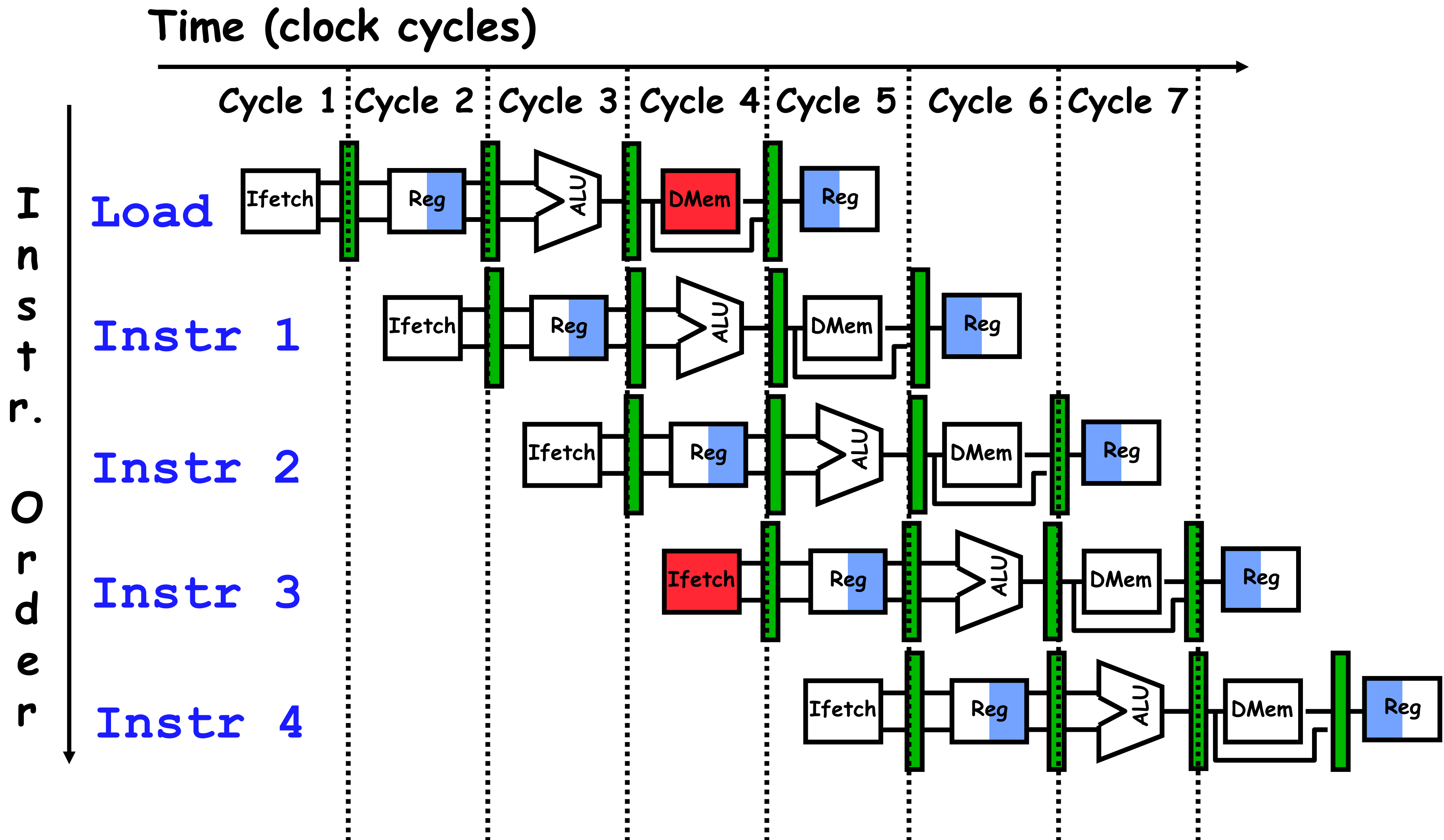
Pipeline Hazards

- Hazard, that prevent the next instruction in the instruction steam.
  - **Structural hazards: resource conflict, e.g. using the same unit**
  - **Data hazards: an instruction depends on the results of a previous instruction**
  - **Control hazards: arise from the pipelining of branches and other instructions that change the PC.**
- Hazards in pipelines can make it necessary to *stall* the pipeline.
  - Stall will reduce pipeline performance.

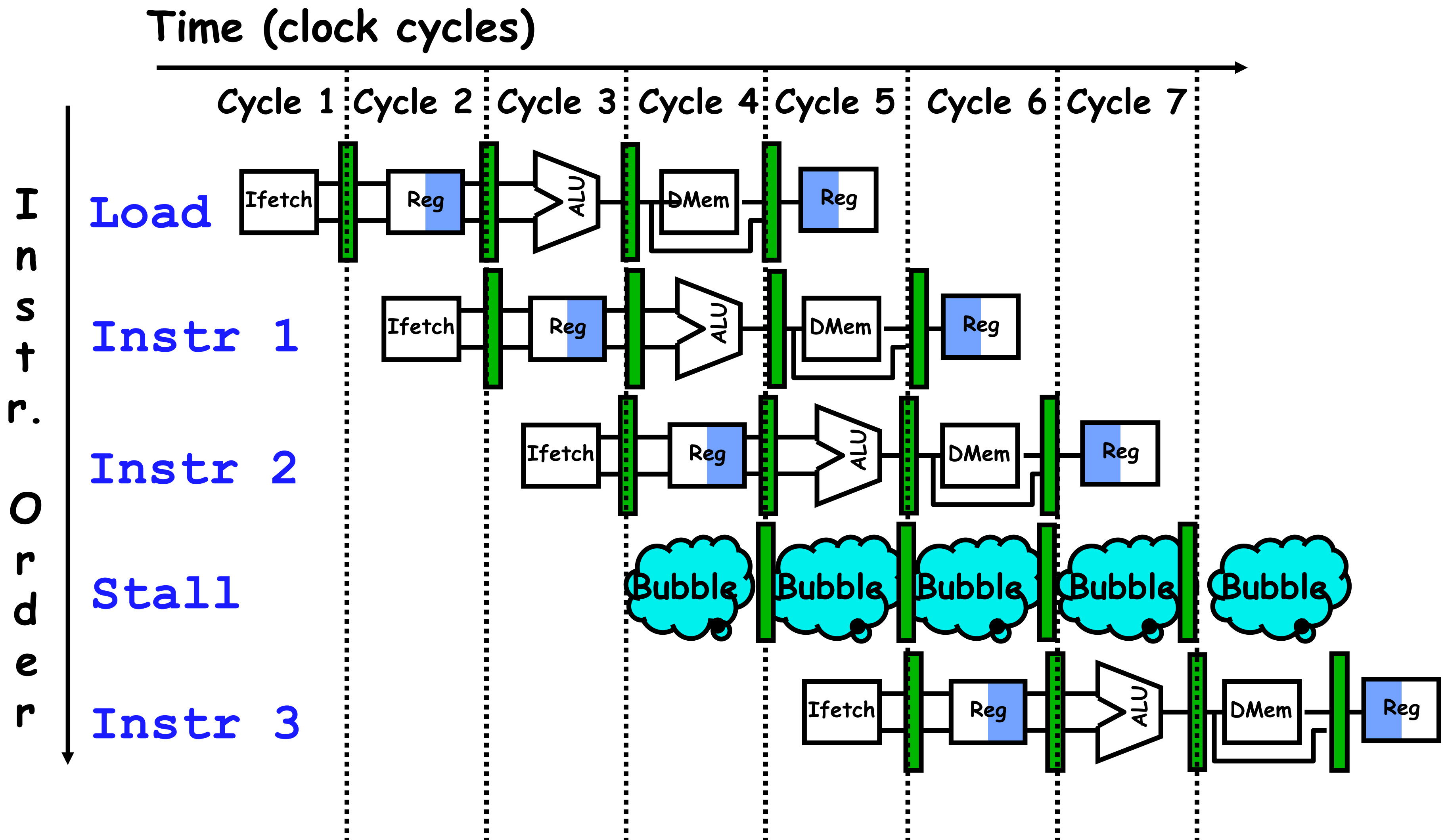
Structure Hazards

- Structure Hazards
  - If some combination of instructions cannot be accommodated because of resource conflict (resources are pipelining of functional units and duplication of resources).
    - » Occur when some functional unit is not fully pipelined, or
    - » No enough duplicated resources.

One Memory Port/Structural Hazards



One Memory Port/Structural Hazards



How do you “bubble” the pipe?

Performance on Structure Hazard

- Example 2 (p.C-14): Let's see how much the **load structure hazard** might cost. Suppose that data reference constitute 40% of the mix, and that the ideal CPI of the pipelined processor, ignoring the structure hazard, is 1. Assume that the processor with the structure hazard has a clock rate that is 1.05 times higher than the clock rate of processor without the hazard. Disregarding any other performance losses, is the pipeline with or without the structure hazard faster, and by how much?
- Answer

The average instruction execution time on the unpipelined processor is

$$\begin{aligned}\text{Average instruction time}_{\text{ideal}} &= \text{CPI} \times \text{Clock cycle time}_{\text{ideal}} \\ &= 1 \times \text{Clock cycle time}_{\text{ideal}}\end{aligned}$$

$$\begin{aligned}\text{Average instruction time}_{\text{structure hazard}} &= \text{CPI} \times \text{Clock cycle time} \\ &= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05} \\ &= 1.3 \times \text{Clock cycle time}_{\text{ideal}}\end{aligned}$$

How to solve structure hazard? (Next slide)

Summary of Structure Hazard

- An alternative to this structure hazard, designer could provide a separate memory access for instructions.
  - Splitting the cache into separate *instruction* and *data caches*, or
  - Use a set of buffers, usually called *instruction buffers*, to hold instruction;
- However, it will increase cost overhead.
  - Ex1: pipelining function units or duplicated resources is a high cost;
  - Ex2: require twice bandwidth and often have higher bandwidth at the pins to support both an instruction and a data cache access every cycle;
  - Ex3: a floating-point multiplier consumes lots of gates.

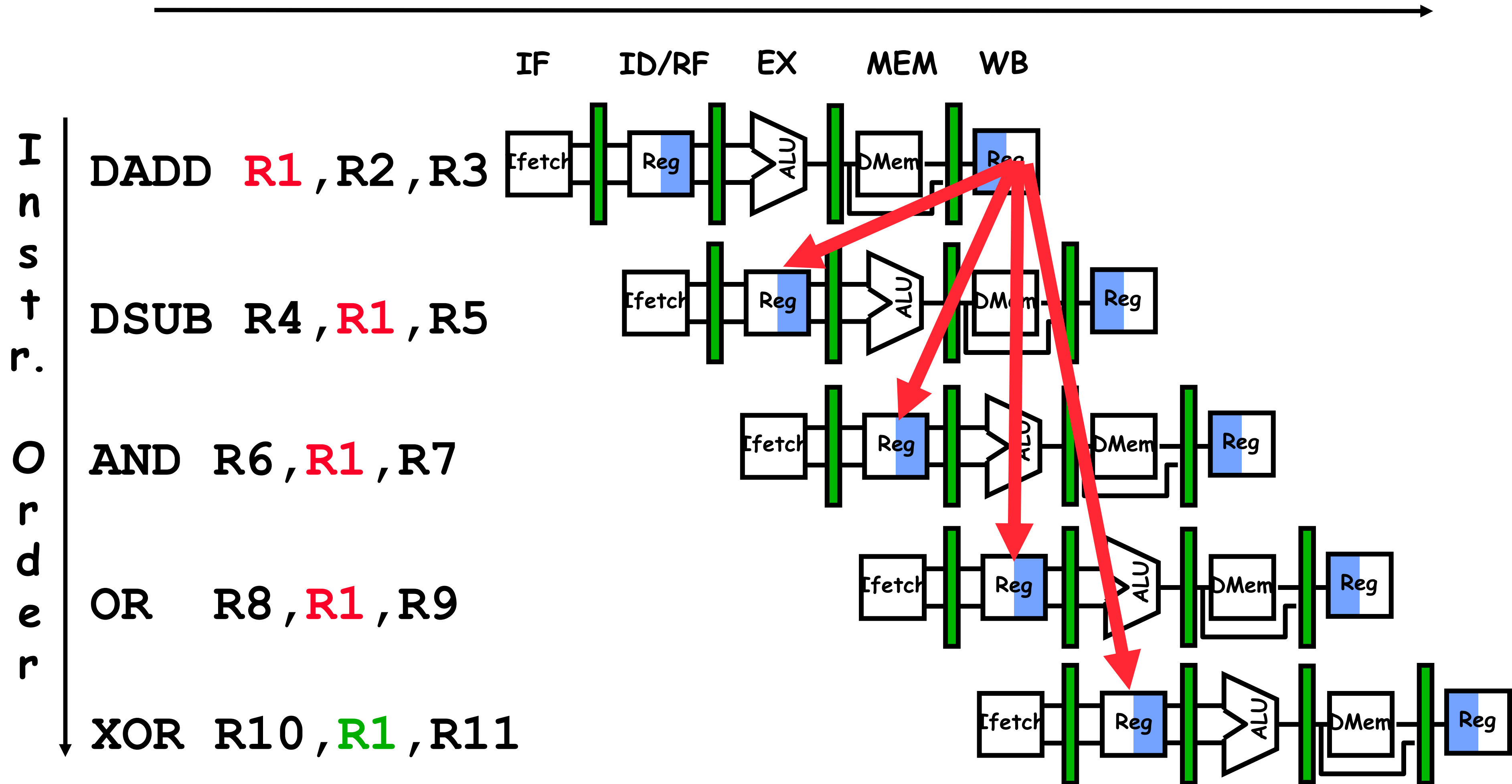
† If the structure hazard is rare, it may not be worth the cost to avoid it.

Data Hazards

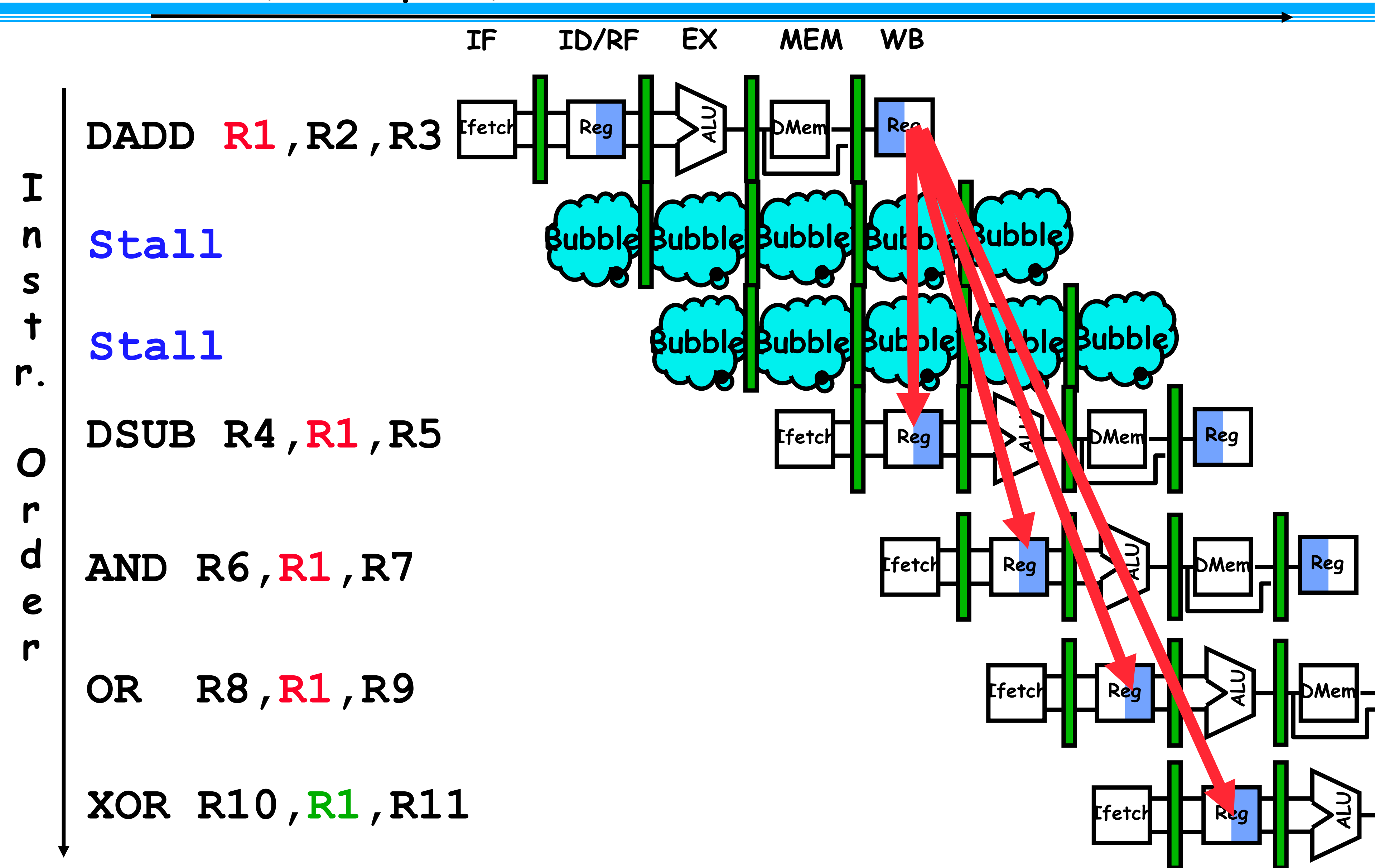
- Data Hazards
  - Occur when the **pipeline changes the order of read/write accesses to operands s**o that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.
    - Occur when some functional unit is not fully pipelined, or
    - No enough duplicated resources.
  - A example of pipelined execution

DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11

Data Hazard on R1



Solution #1: Insert stalls



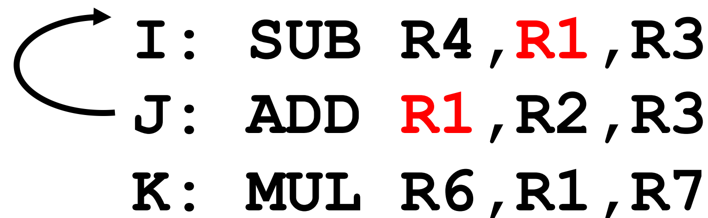
Three Generic Data Hazards (1/3)

- Read After Write (RAW)
  - $Instr_J$ tries to read operand *before* $Instr_I$ writes it

- Caused by a "true dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards (2/3)

- Write After Read (WAR)
  - $Instr_J$ writes operand *before* $Instr_I$ reads it



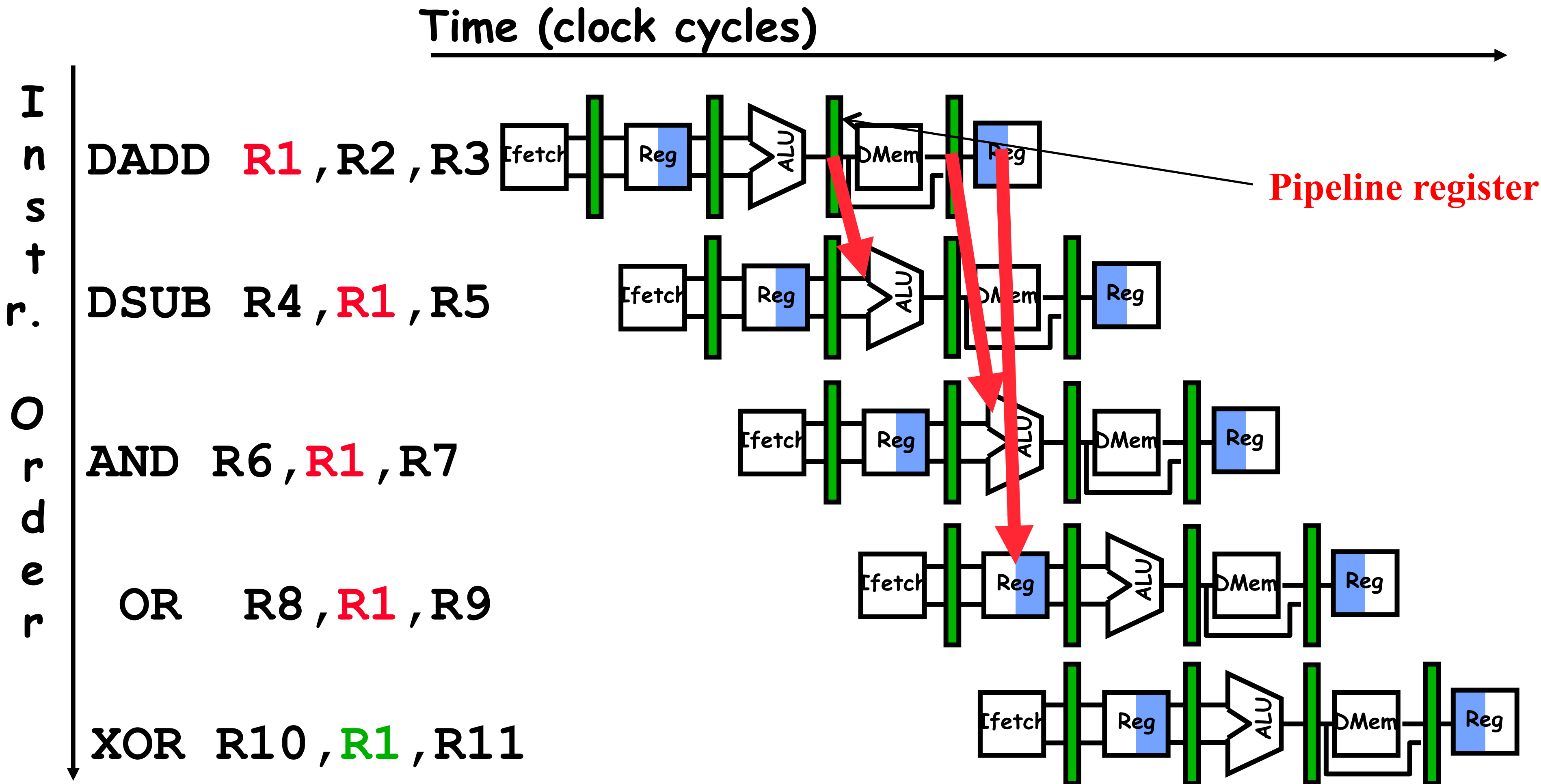
- Called an "anti-dependence" by compiler writers. This results from reuse of the name "R1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Three Generic Data Hazards (3/3)

- Write After Write (WAW)
  - $Instr_J$ writes operand *before* $Instr_I$ writes it.

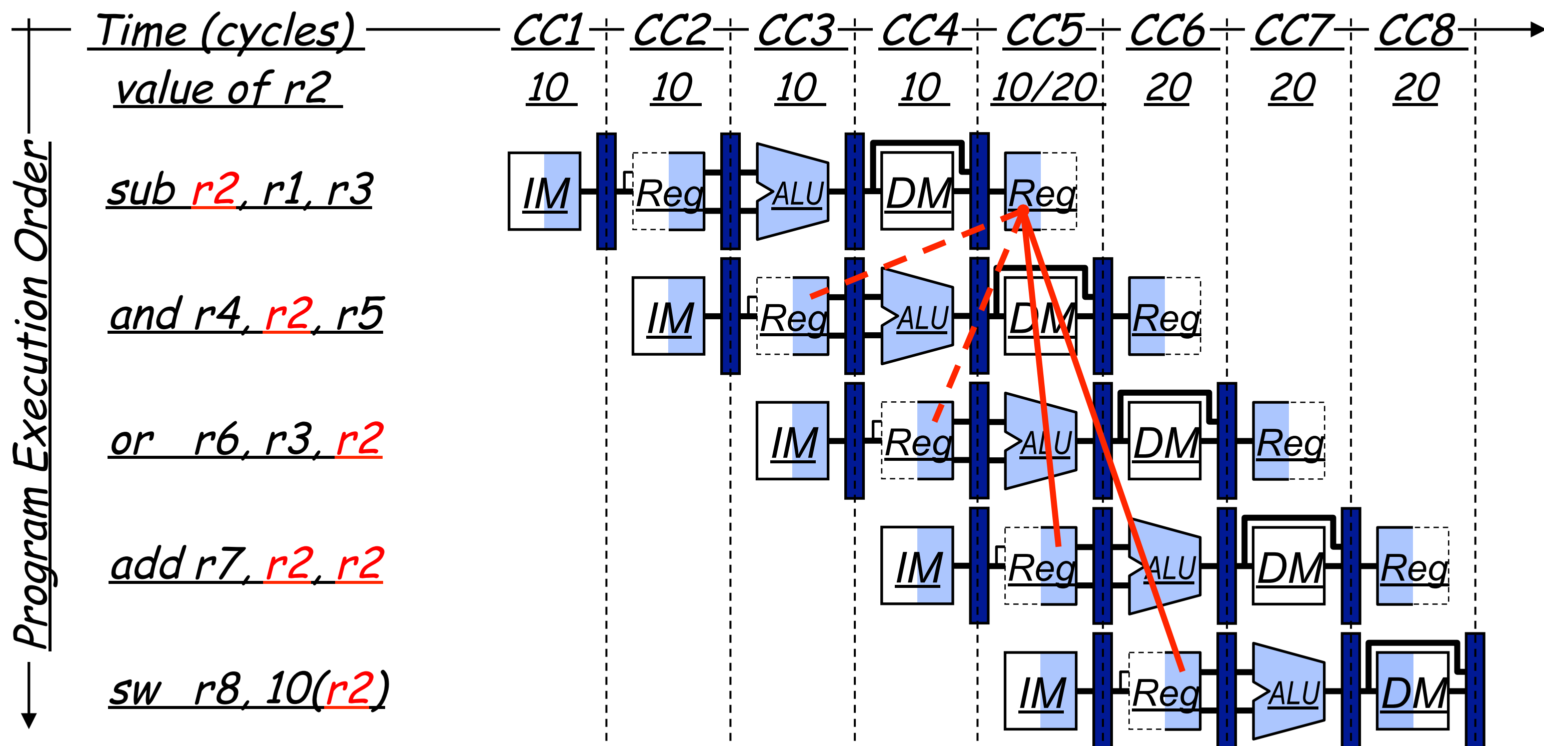
- This hazard also results from the reuse of name r1
- Hazard when writes occur in the wrong order
- Can't happen in our basic 5-stage pipeline because:
  - All writes are ordered and take place in stage 5
- WAR and WAW hazards occur in complex pipelines
- Notice that Read After Read – RAR is NOT a hazard

#2: Forwarding (aka bypassing) to Avoid Data Hazard



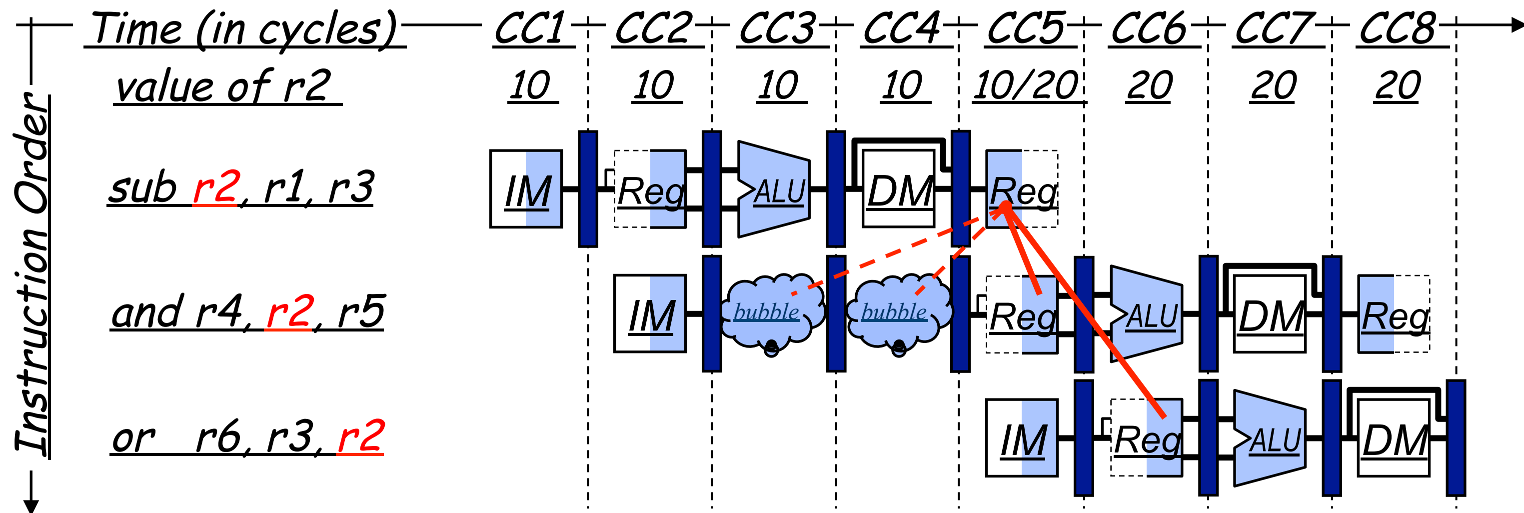
Another Example of a RAW Data Hazard

- Result of sub is needed by and, or, add, & sw instructions
- Instructions and & or will read old value of r2 from reg file
- During CC5, r2 is written and read – new value is read



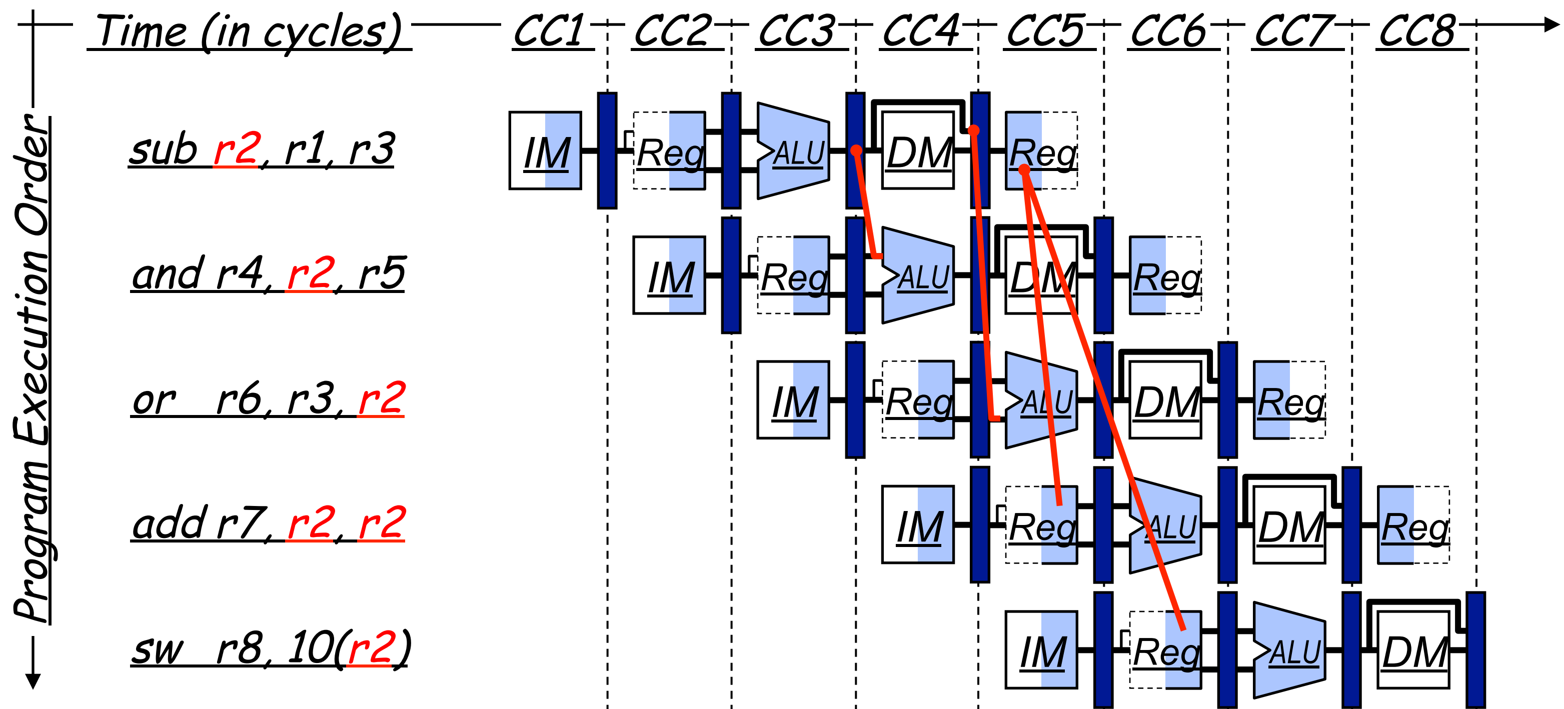
Solution #1: Stalling the Pipeline

- The and instruction cannot fetch r2 until CC5
  - The and instruction remains in the IF/ID register until CC5
- Two bubbles are inserted into ID/EX at end of CC3 & CC4
  - Bubbles are NOP instructions: do not modify registers or memory
  - Bubbles delay instruction execution and waste clock cycles



Solution #2: Forwarding ALU Result

- The ALU result is forwarded (fed back) to the ALU input
  - No bubbles are inserted into the pipeline and no cycles are wasted
- ALU result exists in either EX/MEM or MEM/WB register



Double Data Hazard

- Consider the sequence:

```
add r1,r1,r2
sub r1,r1,r3
and r1,r1,r4
```

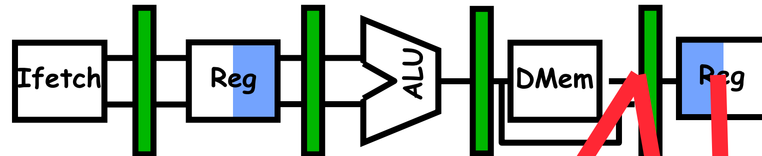
- Both hazards occur
  - Want to use the most recent
  - When executing AND, forward result of SUB
    - ForwardA = 01 (from the EX/MEM pipe stage)

Data Hazard Even with Forwarding

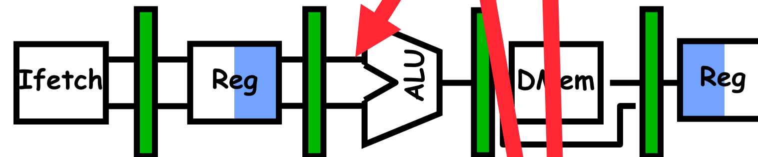
Time (clock cycles)

I n s t r. O r d e r

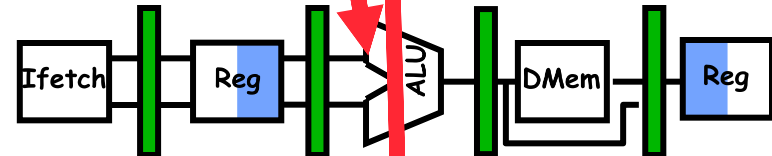
LD R1,0(R2)



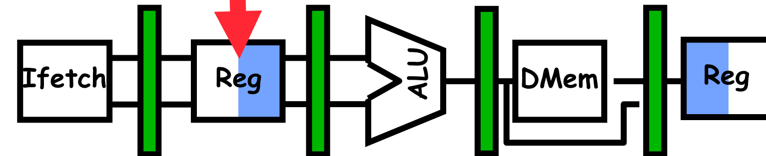
DSUB R4,R1,R6

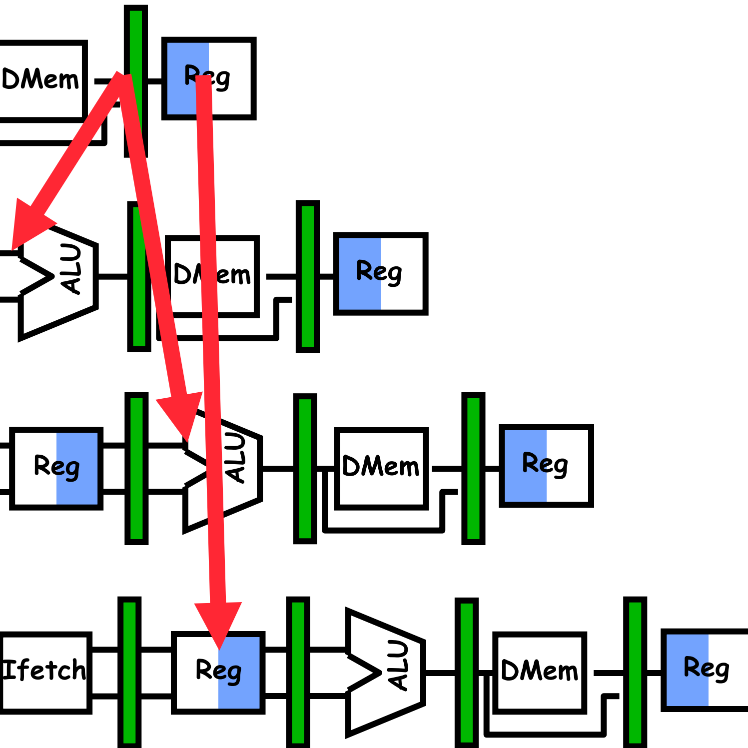


DAND R6,R1,R7

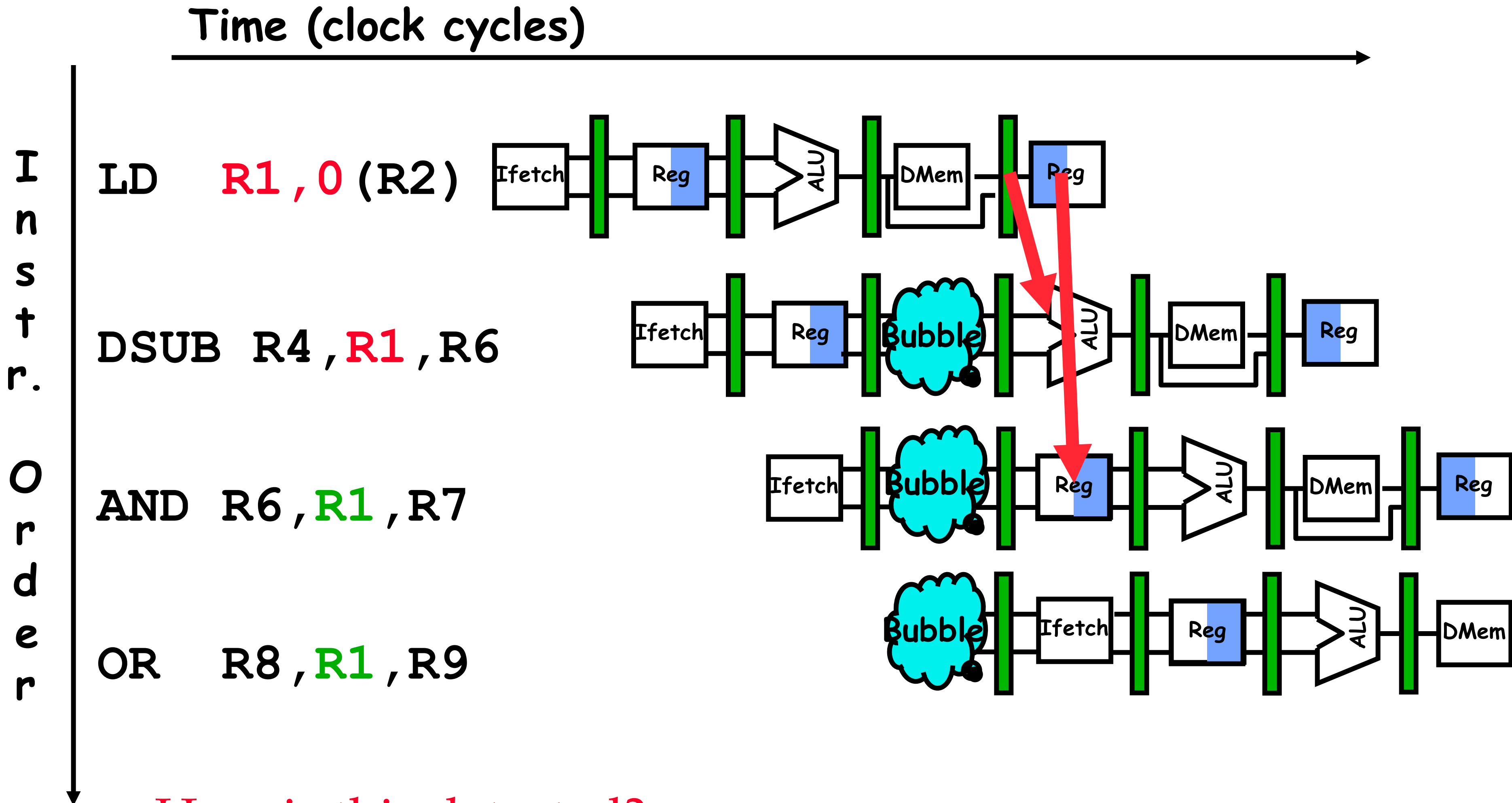


OR R8,R1,R9





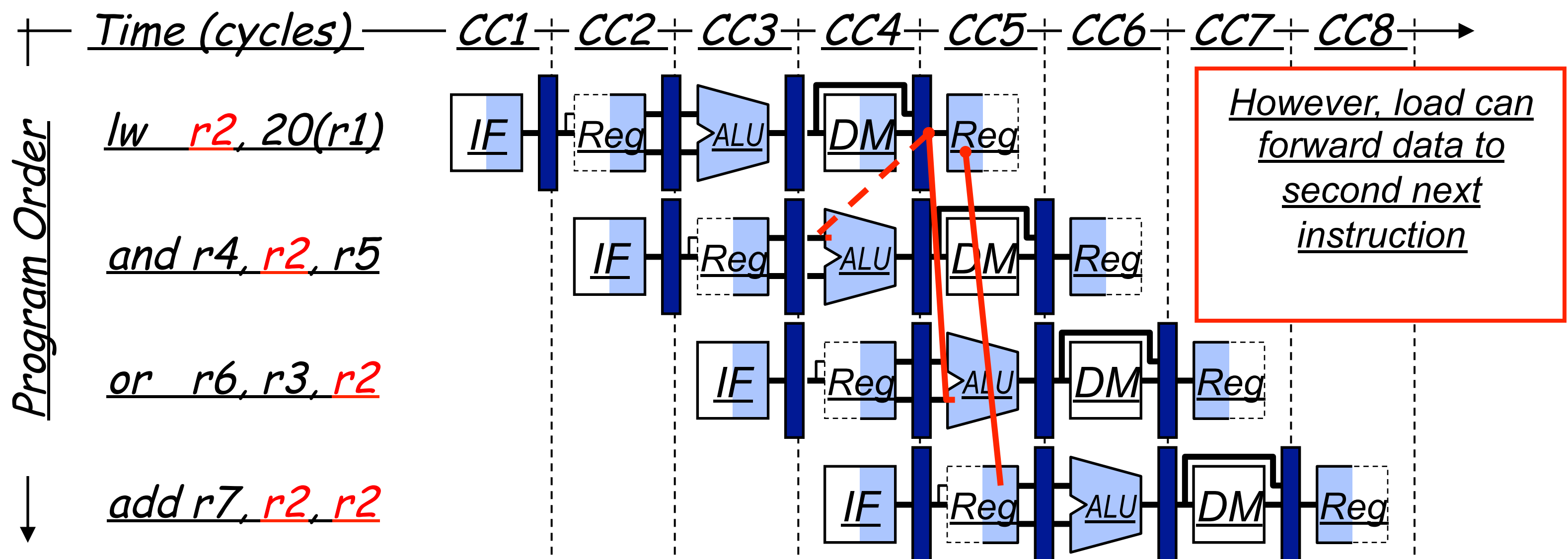
Data Hazard Even with Forwarding



How is this detected?

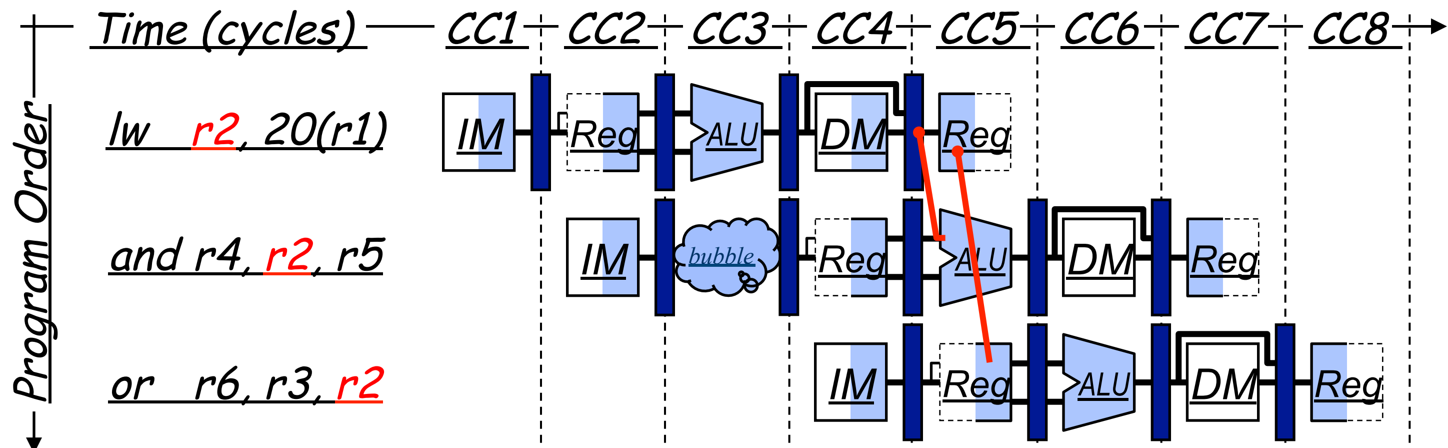
Load Delay

- Not all RAW data hazards can be forwarded
  - **Load has a delay that cannot be eliminated by forwardin**
- In the example shown below …
  - The LW instruction does not have data until end of CC4
  - AND wants data at beginning of CC4 - NOT possible

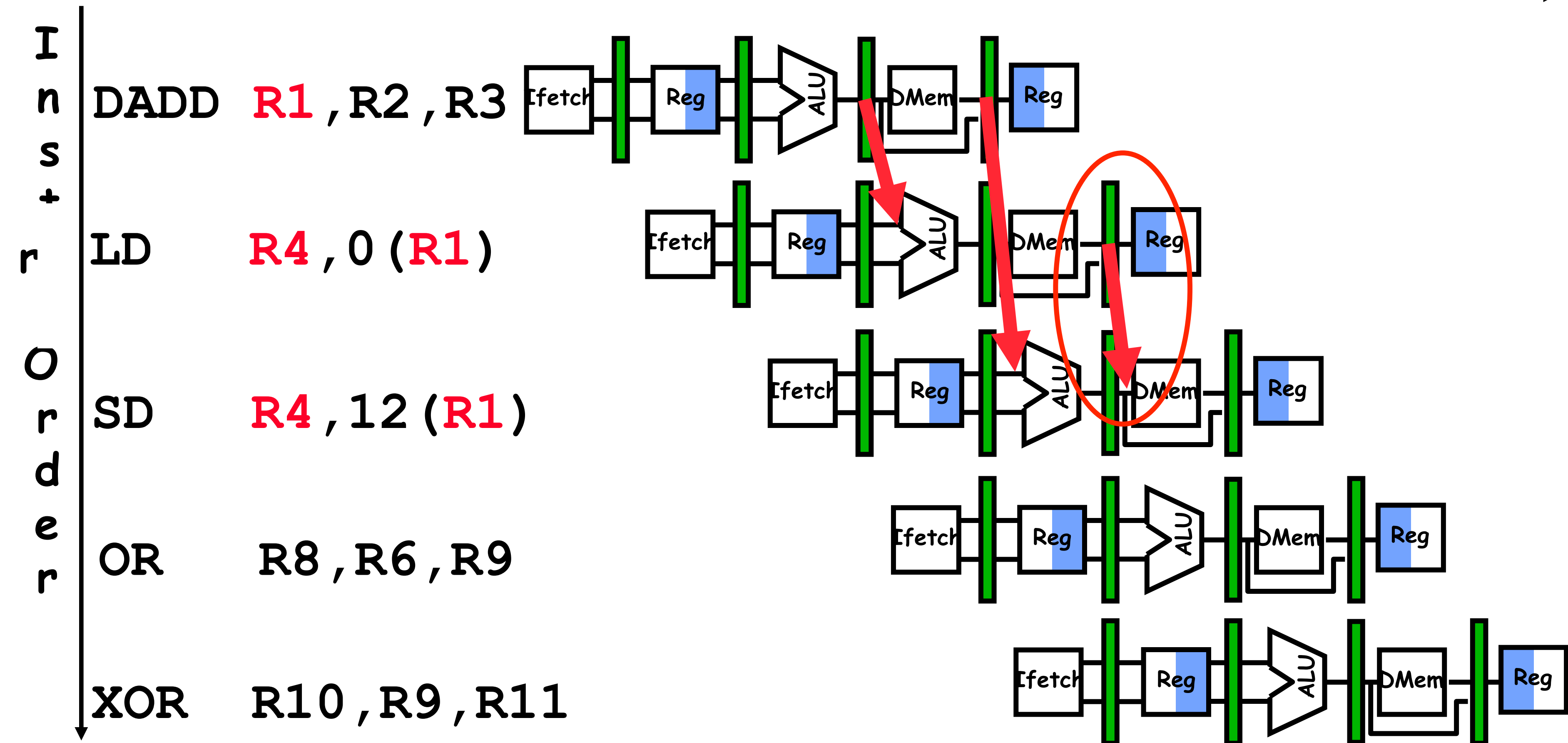


Stall the Pipeline for one Cycle

- Freeze the PC and the IF/ID registers
  - No new instruction is fetched and instruction after load is stalled
- Allow the Load in ID/EX register to proceed
- Introduce a bubble into the ID/EX register
- Load can forward data after stalling next instruction



Forwarding to Avoid LW-SW Data Hazard



Compiler Scheduling

- Compilers can schedule code in a way to avoid load stalls
- Consider the following statements:

  a = b + c; d = e – f;

- Slow code: 2 stall cycles

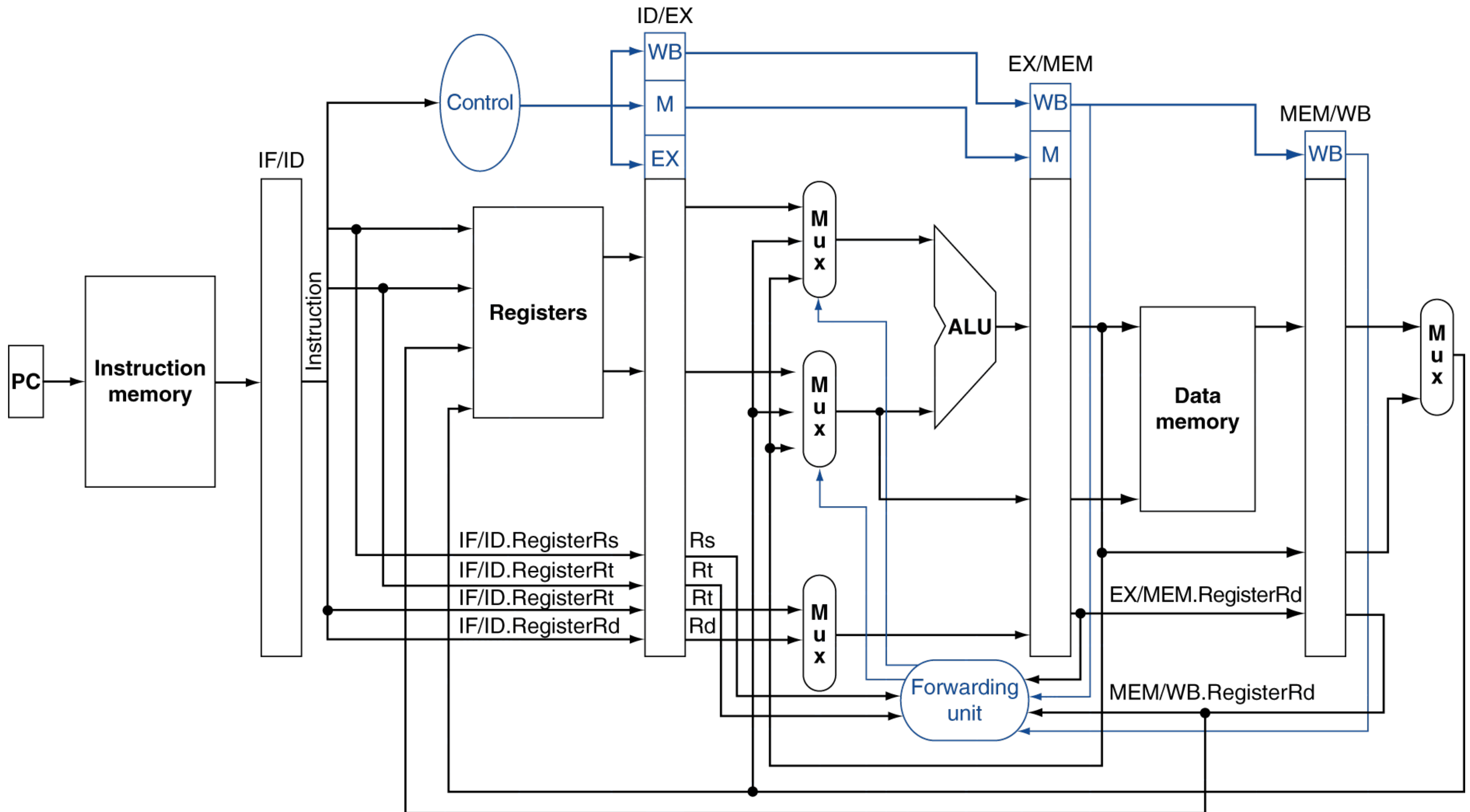
```
lw      r10, (r1)          # r1 = addr b
lw      r11, (r2)          # r2 = addr c
add     r12, r10, $11      # stall
sw      r12, (r3)          # r3 = addr a
lw r13, (r4)   # r4 = addr e
lw      r14, (r5)          # r5 = addr f
sub     r15, r13, r14      # stall
sw      r15, (r6)          # r6 = addr d
```

Fast code: No Stalls

```
lw   r10, 0(r1)
lw   r11, 0(r2)
lw   r13, 0(r4)
lw   r14, 0(r5)
add  r12, r10, r11
sw   r12, 0(r3)
sub  r15, r13, r14
sw   r14, 0(r6)
```

Compiler optimizes for performance. Hardware checks for safety.

Hardware Support for Forwarding



Detecting RAW Hazards

- Pass register numbers along pipeline
  - ID/EX.RegisterRs = register number for Rs in ID/EX
  - ID/EX.RegisterRt = register number for Rt in ID/EX
  - ID/EX.RegisterRd = register number for Rd in ID/EX
- Current instruction being executed in ID/EX register
- Previous instruction is in the EX/MEM register
- Second previous is in the MEM/WB register
- RAW Data hazards when

  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

  Fwd from EX/MEM pipeline reg

  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

  Fwd from MEM/WB pipeline reg

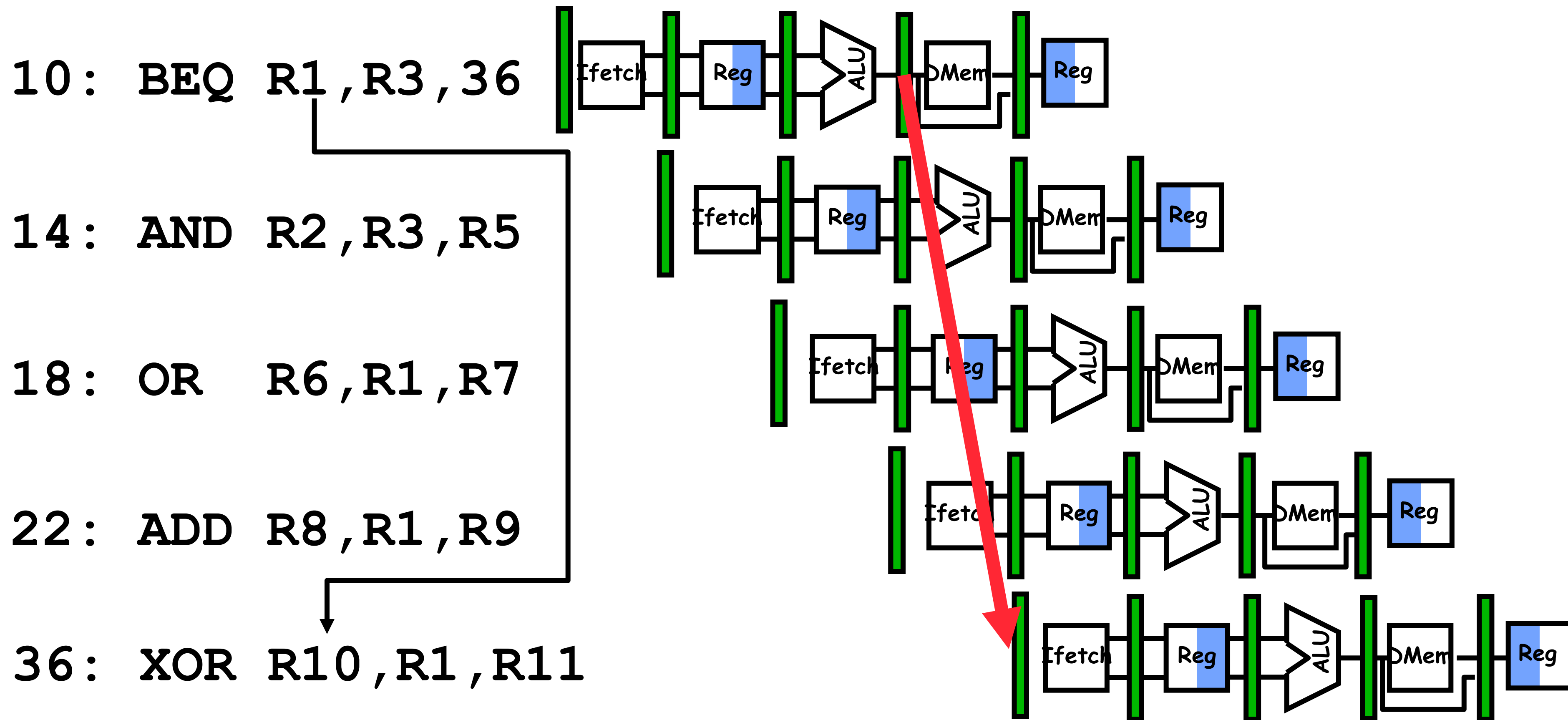
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not R0
  - EX/MEM.RegisterRd ≠ 0
  - MEM/WB.RegisterRd ≠ 0

Forwarding Conditions

- Detecting RAW hazard with Previous Instruction
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01 (Forward from EX/MEM pipe stage)
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01 (Forward from EX/MEM pipe stage)
- Detecting RAW hazard with Second Previous
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10 (Forward from MEM/WB pipe stage)
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10 (Forward from MEM/WB pipe stage)

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"?

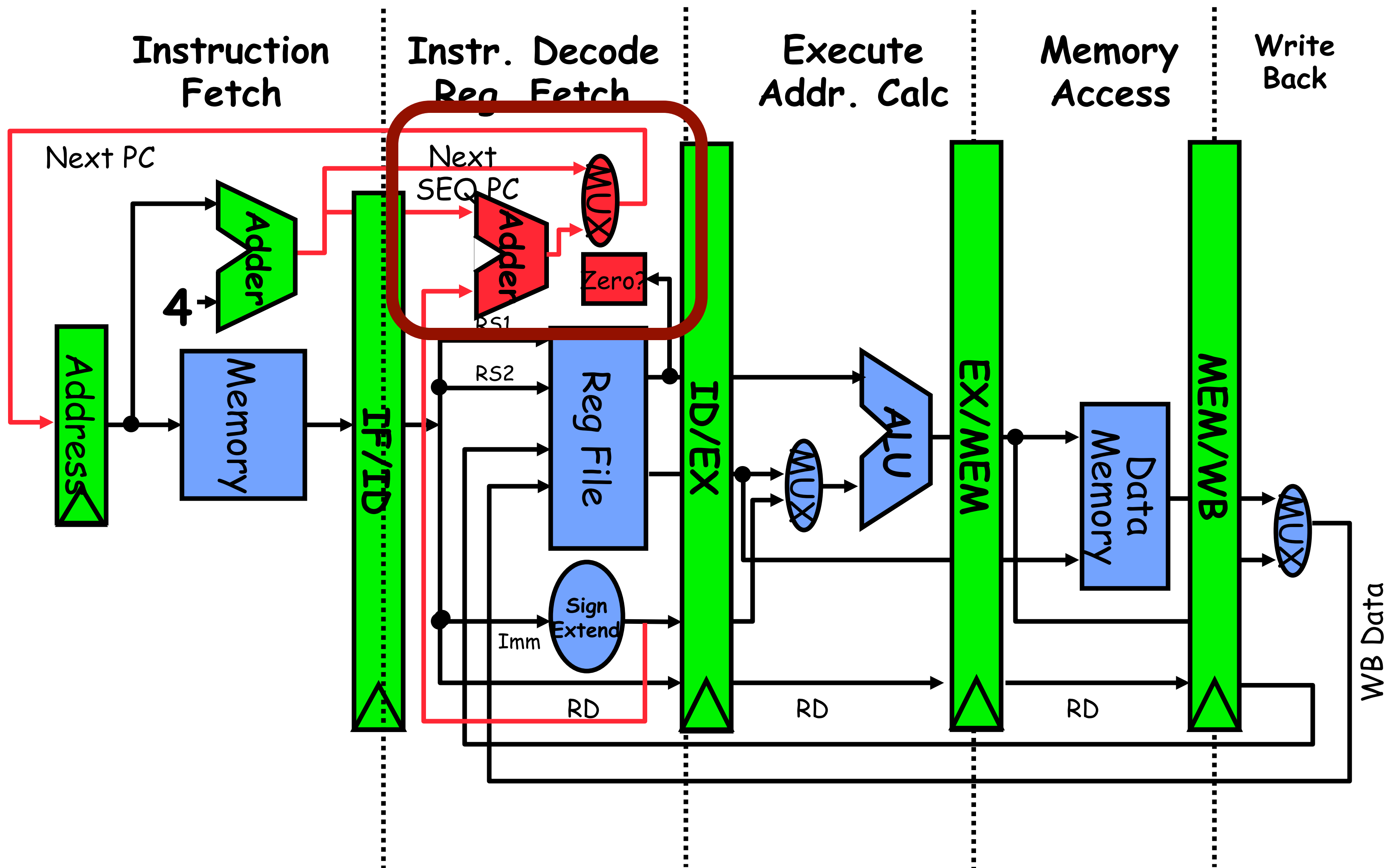
Branch/Control Hazards

- Branch instructions can cause great performance loss
- Branch instructions need two things:
  - Branch Result — Taken or Not Taken
  - Branch Target
    - PC + 4 — If Branch is NOT taken
    - PC + 4 + 4 × imm — If Branch is Taken
- For our pipeline: 3-cycle branch delay
  - PC is updated 3 cycles after fetching branch instruction
  - Branch target address is calculated in the ALU stage
  - Branch result is also computed in the ALU stage
  - What to do with the next 3 instructions after branch?

Branch Stall Impact

- If CPI = 1 without branch stalls, and 30% branch
- If stalling 3 cycles per branch
  - => new CPI = $1+0.3\times3 = 1.9$
- Two part solution:
  - Determine branch taken or not sooner, and
  - Compute taken branch address earlier
- MIPS Solution:
  - Move branch test to ID stage (second stage)
  - Adder to calculate new PC in ID stage
  - Branch delay is reduced from 3 to just 1 clock cycle

Pipelined MIPS Datapath



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

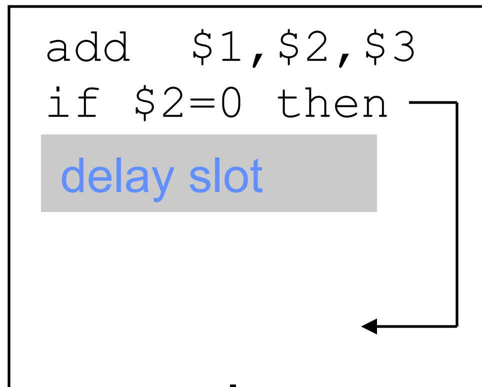
- #4: Delayed Branch
  - Define branch to take place AFTER a following instruction

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

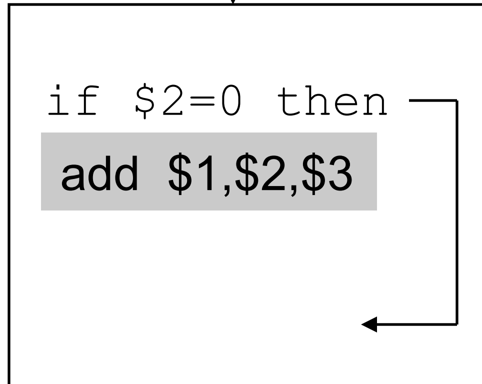
  - 1 slot delay allows proper decision and branch target address in 5 stage pipeline
  - MIPS uses this

Scheduling Branch Delay Slots

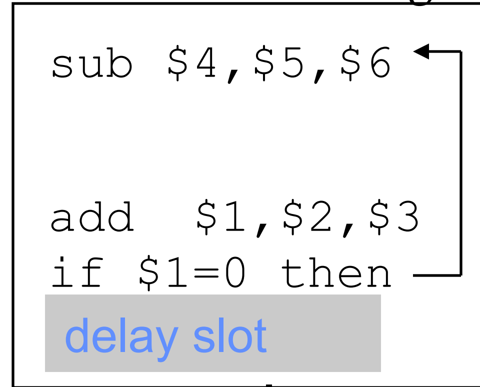
A. From before branch



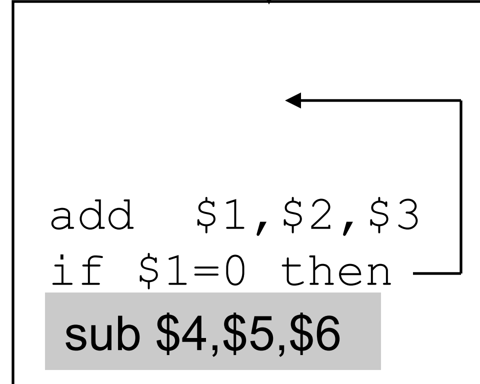
becomes



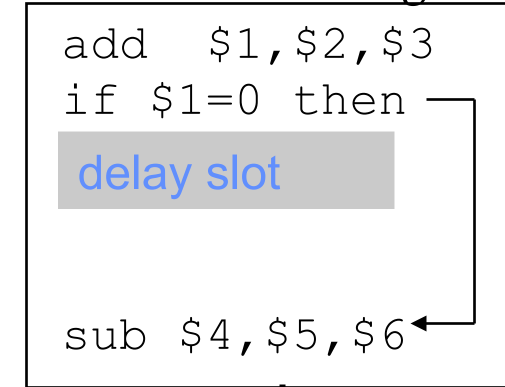
B. From branch target



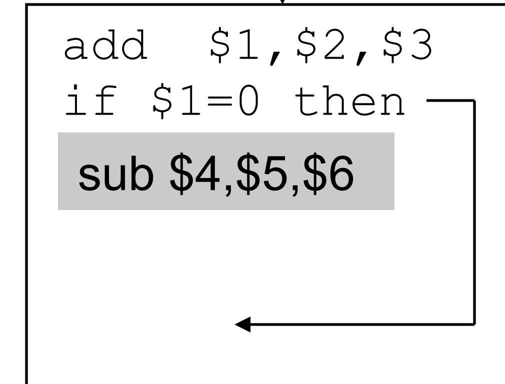
becomes



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

- The effective pipeline speedup with branch penalties, assuming an ideal CPI of 1, is

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Because of the following:

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{Branch penalty}$$

We obtain

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

Performance on Control Hazard (1/2)

- Example 3 (pA-25): for a deeper pipeline, such as that in a MIPS R4000, it takes at least three pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparison, A three-stage delay leads to the branch penalties for the three simplest prediction schemes listed in the following Figure A.15. Find the effective additional to the CPI arising from branches for this pipeline, assuming the following frequencies:

Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

Figure A.15

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted un taken	2	0	3

Performance on Control Hazard (2/2)

- Answer

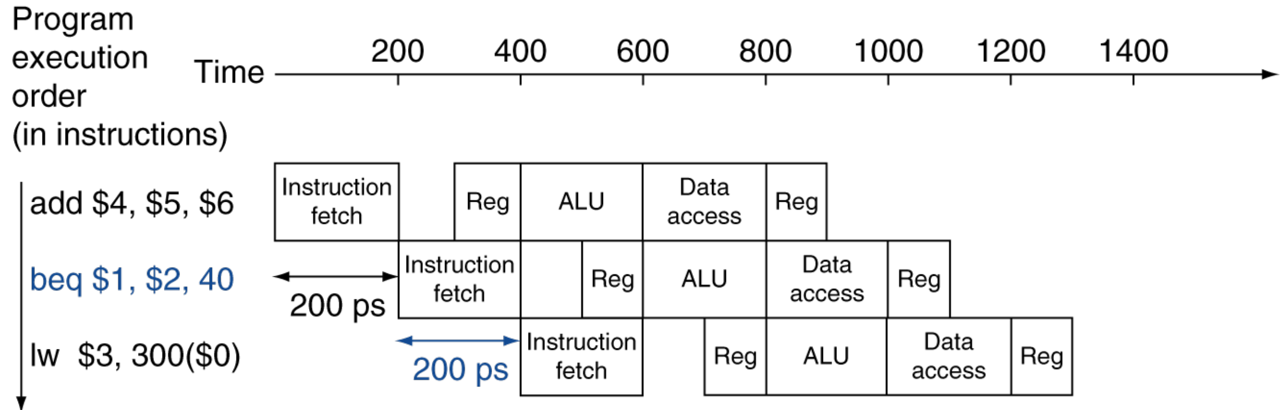
Additions to the CPI from branch cost				
Branch scheme	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Branch Prediction

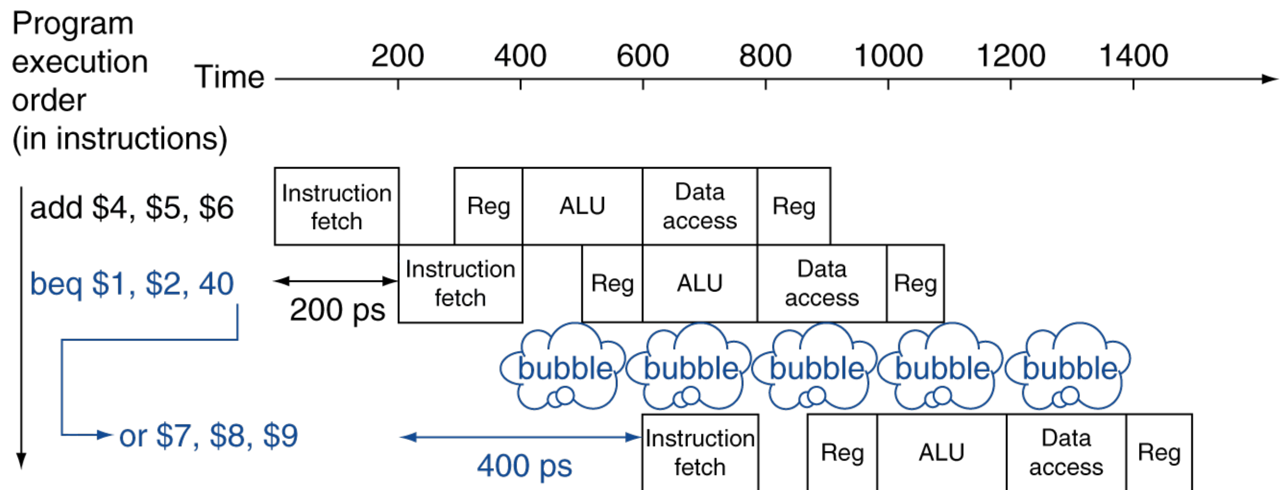
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch (BPB or BHT)
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Static Branch Prediction

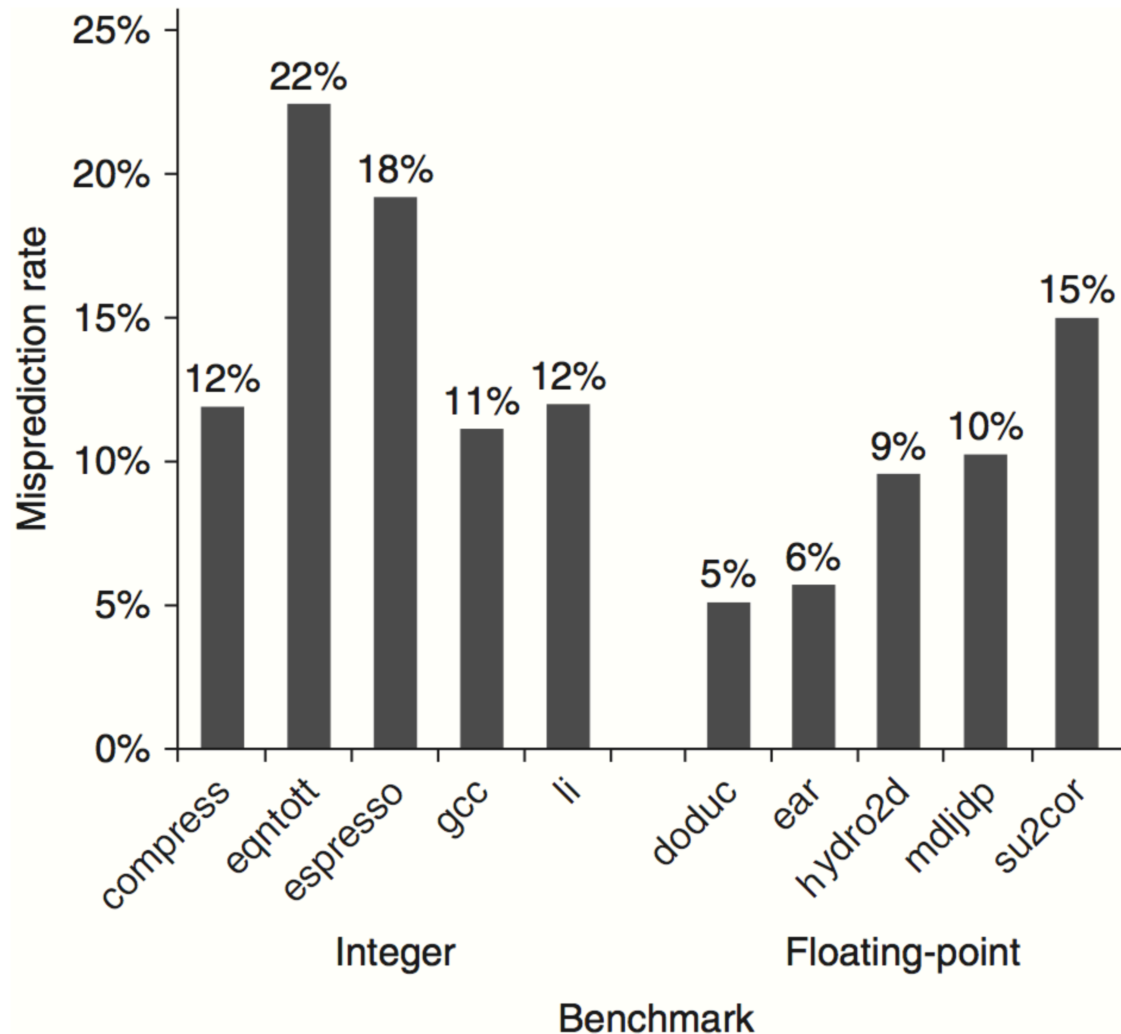


Figure C.17 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an aver-

Dynamic Branch Prediction

- 1-bit prediction scheme
  - Low-portion address as address for a one-bit flag for Taken or NotTaken historically
  - Simple
- 2-bit prediction
  - Miss twice to change

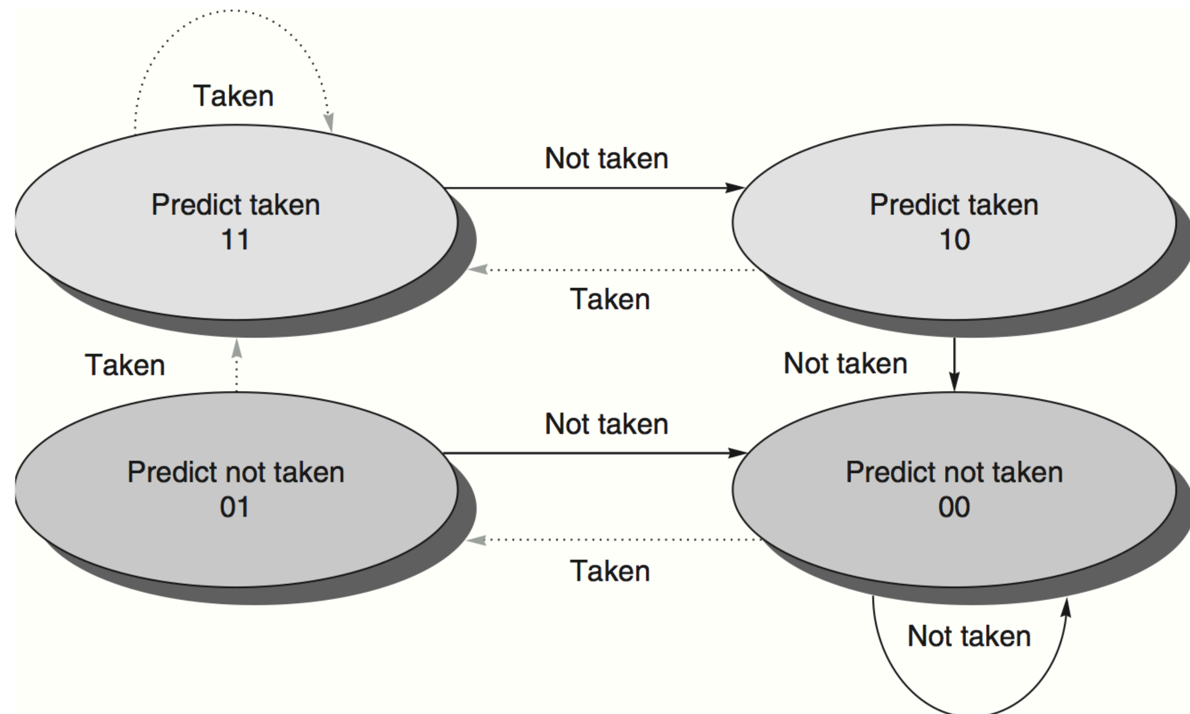


Figure C.18 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and $2^n - 1$: When the coun-

Contents

1. Pipelining Introduction
2. The Major Hurdle of Pipelining—Pipeline Hazards
3. **RISC-V ISA and its Implementation**

Reading:

- **Textbook: Appendix C**
- **RISC-V ISA**
- **Chisel Tutorial**