Lecture 05 and 06: Pipeline: Basic/Intermediate Concepts and Implementation

CSCE 513 **Computer Architecture**

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
https://passlab.github.io/CSCE513

Contents

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

Reading:

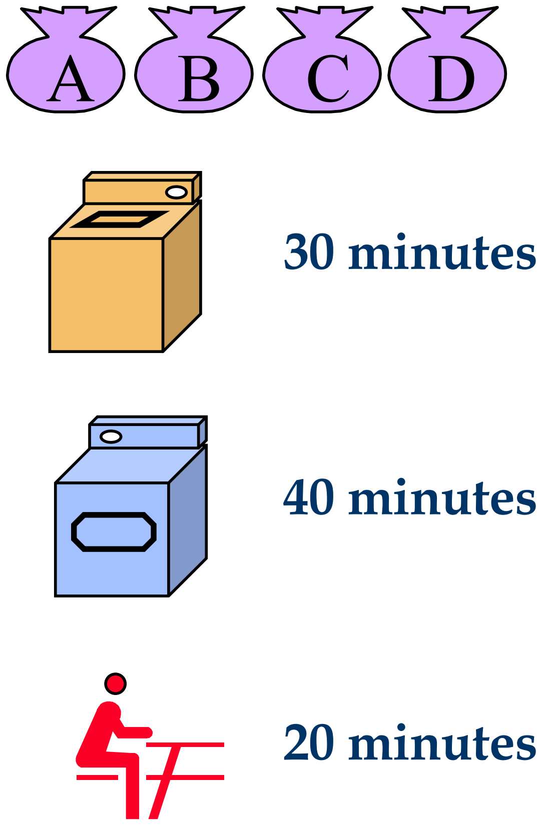
- **Textbook: Appendix C**
- **RISC-V Sodor core**
  - **Chisel: https://github.com/freechipsproject/chisel3/wiki/Short-Users-Guide-to-Chisel**
  - **https://github.com/ucb-bar/riscv-sodor**

Pipelining: Its Natural!

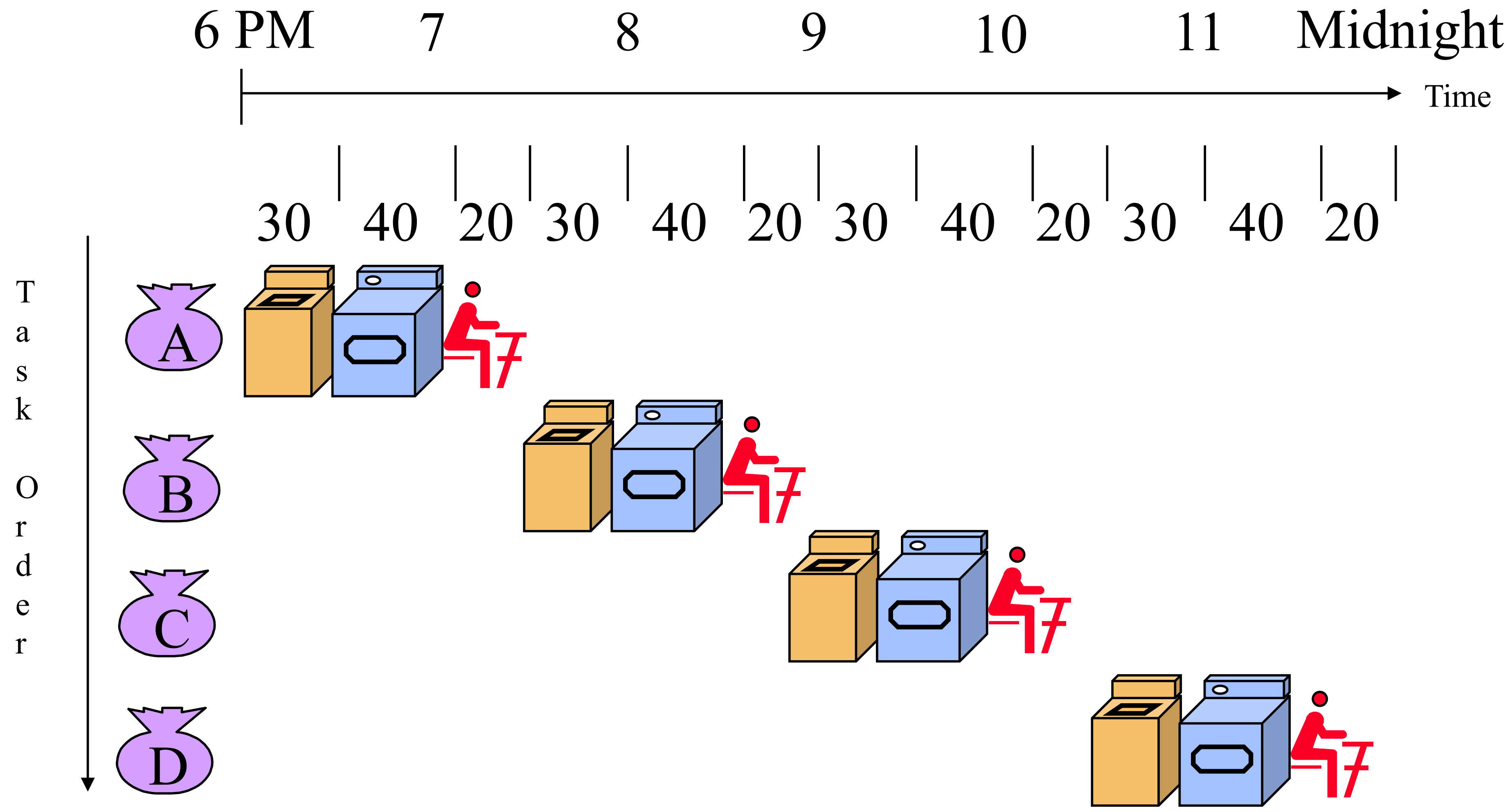
- Laundry Example

- 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

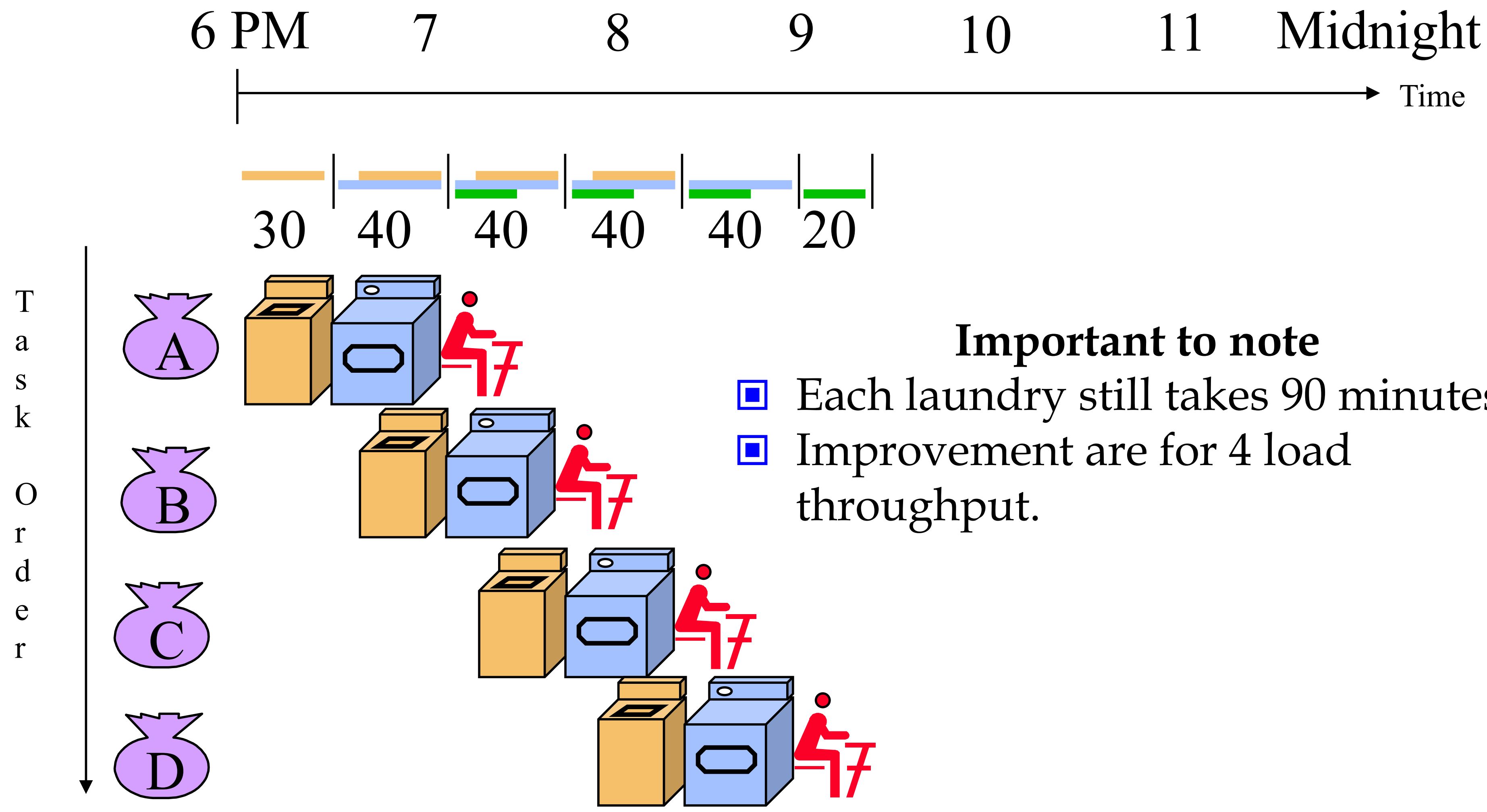


Sequential Laundry



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

Pipelined Laundry Start Work ASAP



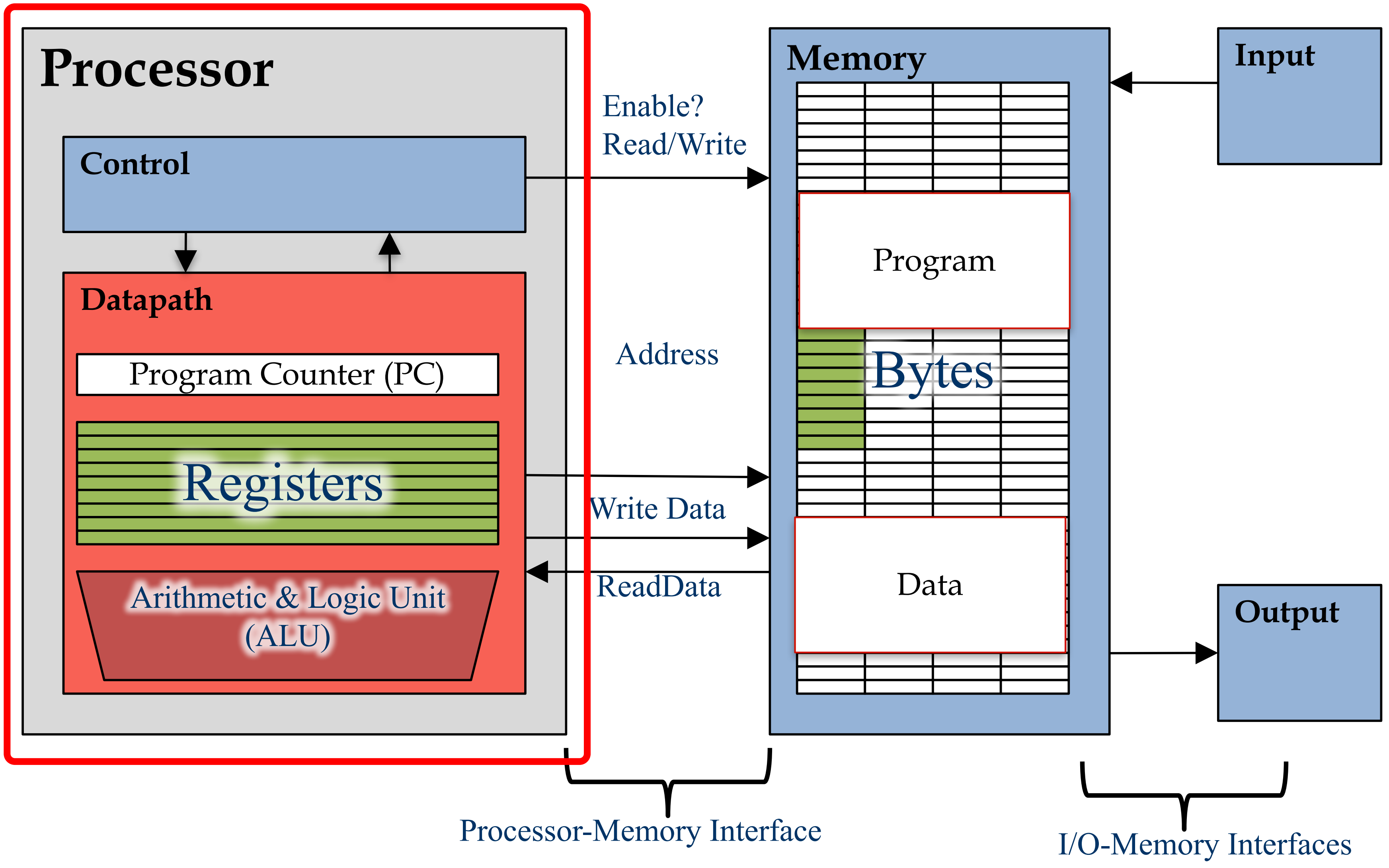
Important to note

- Each laundry still takes 90 minutes.
- Improvement are for 4 load throughput.

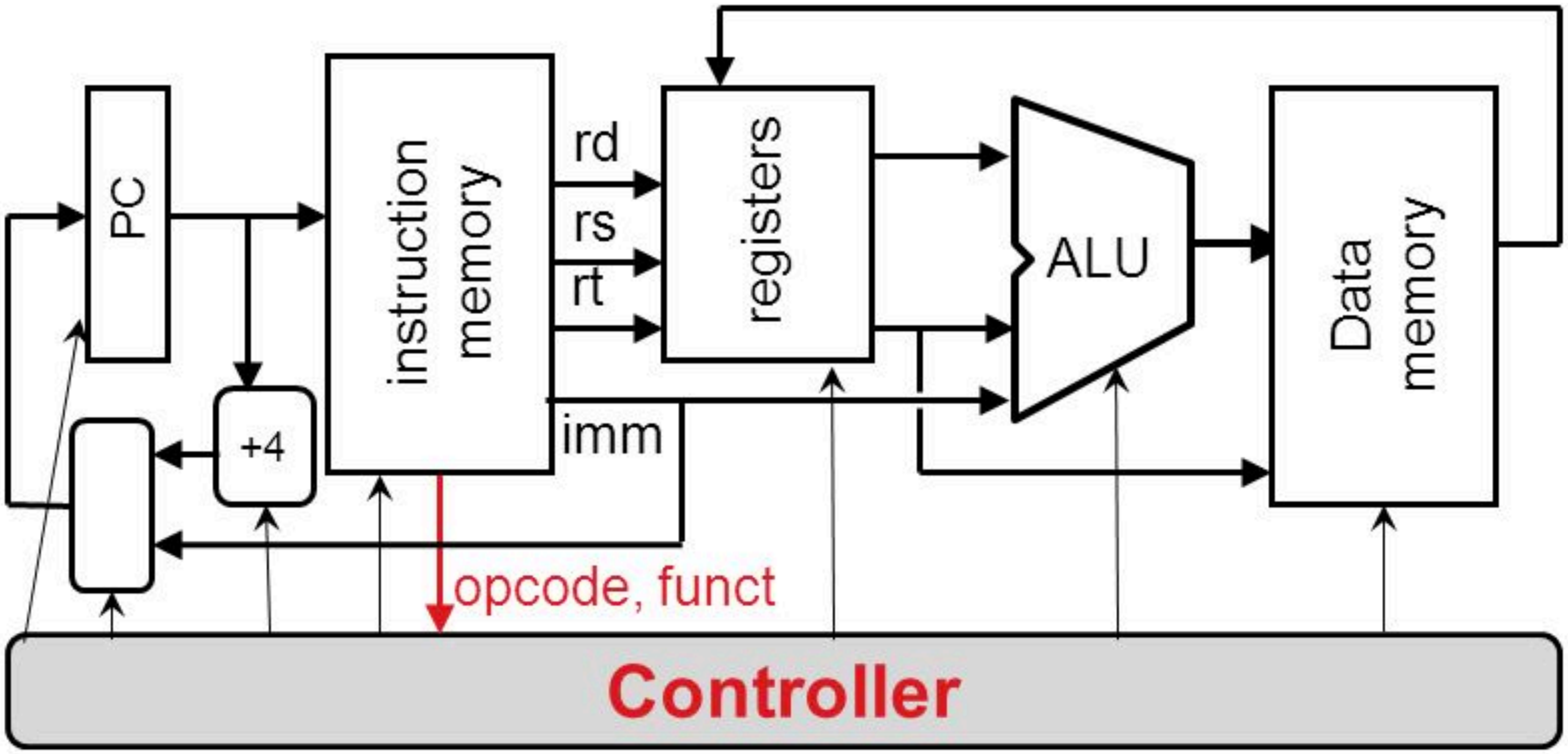
- Pipelined laundry takes 3.5 hours for 4 loads
  - 6/3.5=1.7 time speedup compared to sequential laundry

Mapping Laundry Pipeline to Computer Pipeline

Components of a Computer



CPU and Datapath vs Control



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

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

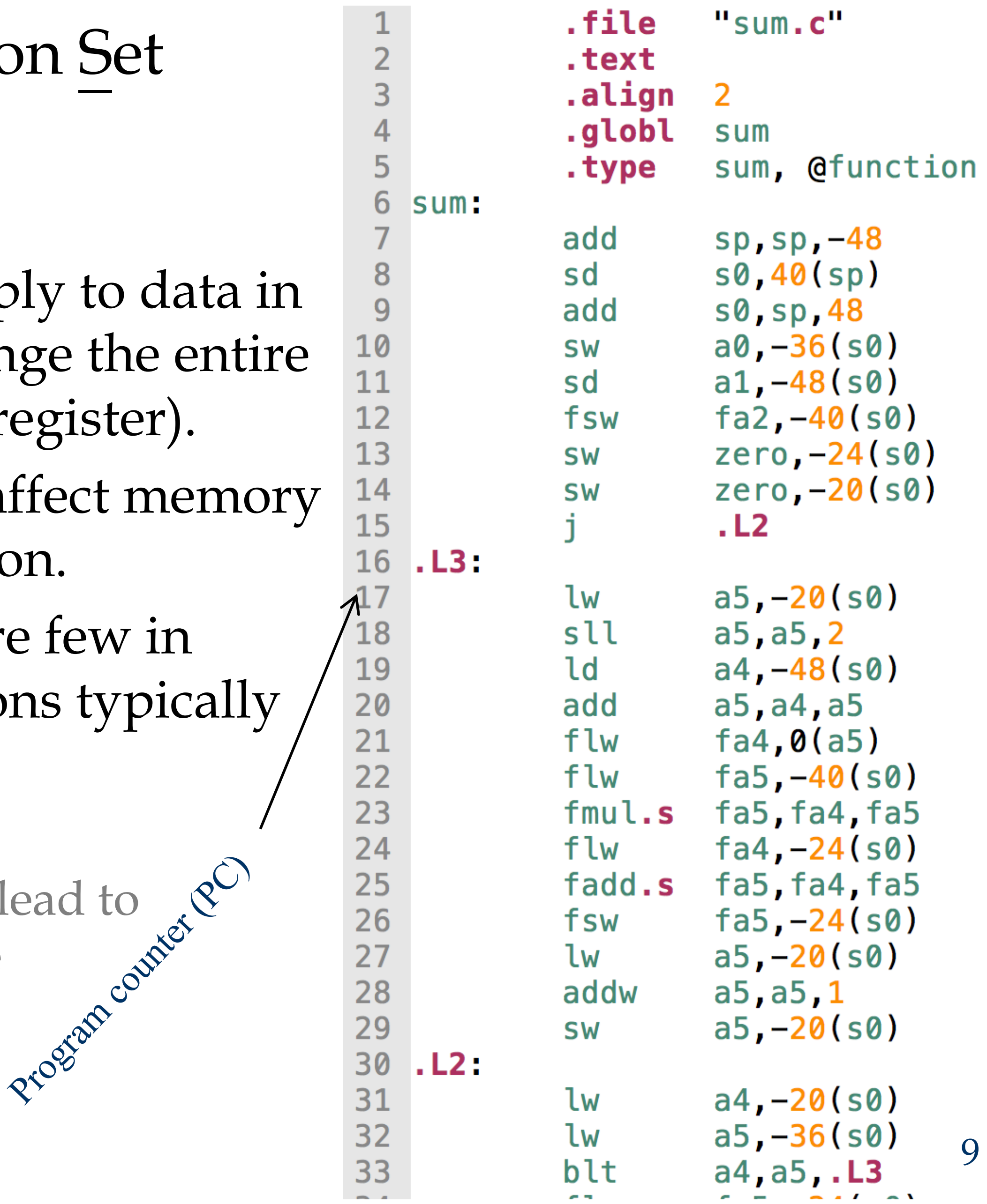
- RISC-V
  - 32 registers, and R0 = 0;
  - Three classes of instructions
    - ALU instruction: add (DADD), subtract (DSUB), and logical operations (such as AND or OR);
    - Load and store instructions:
    - Branches and jumps:

Name	Field						Comments
(Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

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

- *RISC* (Reduced Instruction Set Computer) 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.

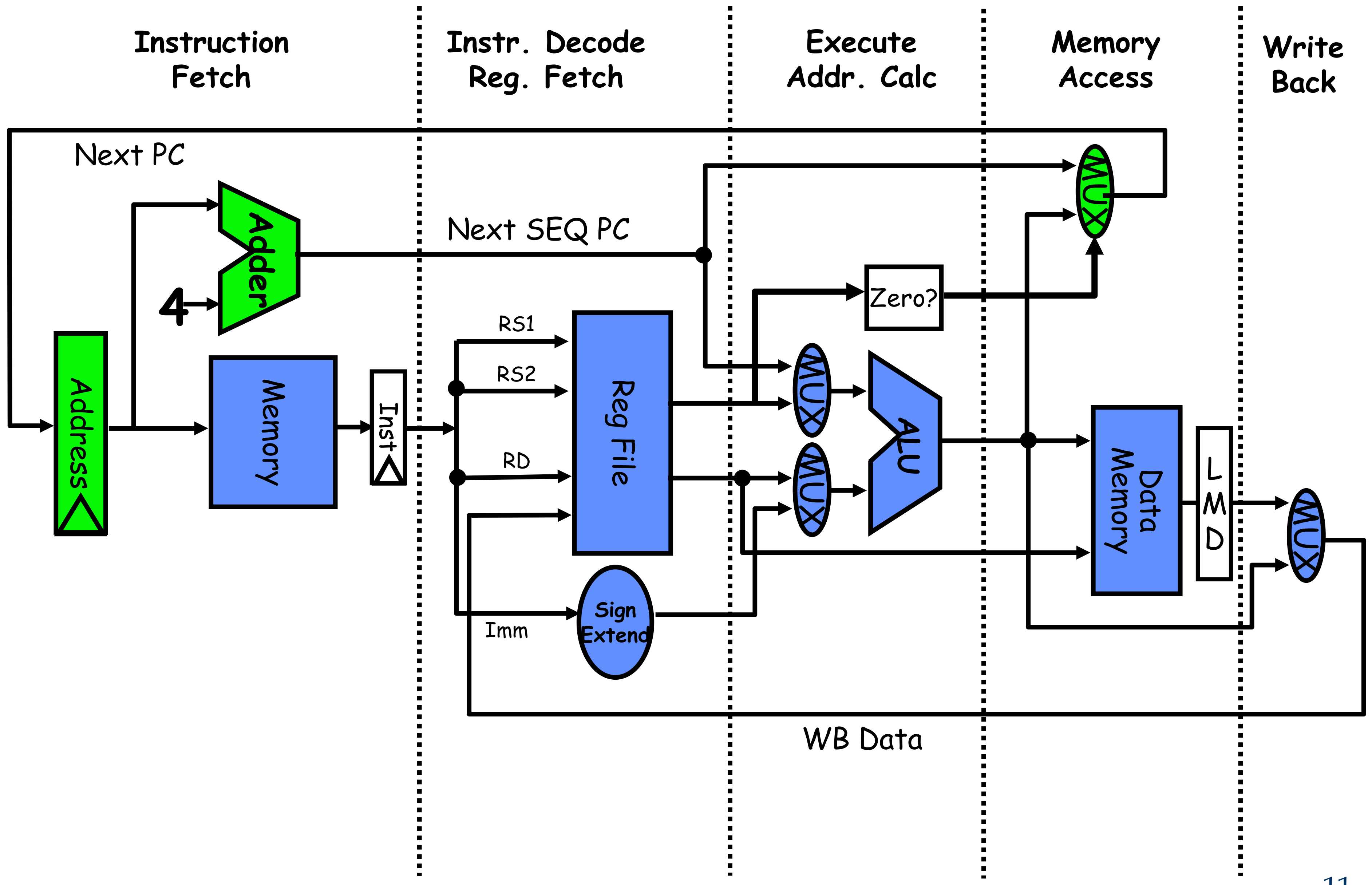


```
        .file   "sum.c"
        .text
        .align  2
        .globl  sum
        .type   sum, @function
sum:
        add     sp,sp,-48
        sd      s0,40(sp)
        add     s0,sp,48
        sw      a0,-36(s0)
        sd      a1,-48(s0)
        fsw     fa2,-40(s0)
        sw      zero,-24(s0)
        sw      zero,-20(s0)
        j       .L2
.L3:
        lw      a5,-20(s0)
        sll     a5,a5,2
        ld      a4,-48(s0)
        add     a5,a4,a5
        flw     fa4,0(a5)
        flw     fa5,-40(s0)
        fmul.s  fa5,fa4,fa5
        flw     fa4,-24(s0)
        fadd.s  fa5,fa4,fa5
        fsw     fa5,-24(s0)
        lw      a5,-20(s0)
        addw    a5,a5,1
        sw      a5,-20(s0)
.L2:
        lw      a4,-20(s0)
        lw      a5,-36(s0)
        blt     a4,a5,.L3
```

RISC Instruction Set

- Every instruction to 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 destination operands for Register-Register ALU instruction or Load instruction.

5 Stages of A Typical RISC ISA Pipeline



Classic 5-Stage Pipeline for a RISC

- In each cycle, hardware initiates a new instruction and executes some part of five different instructions:
  - Simple
  - However, be ensure that the overlap of instructions in the pipeline cannot cause 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 instruction
    - 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.**

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.
  - Register file is used in 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 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

- One register files used in ID and WB stage
  - Read from register in ID (second half of CC), and write to register in WB (first half of CC) in one cycle
- Separate IM and DM: I-cache and D-cache

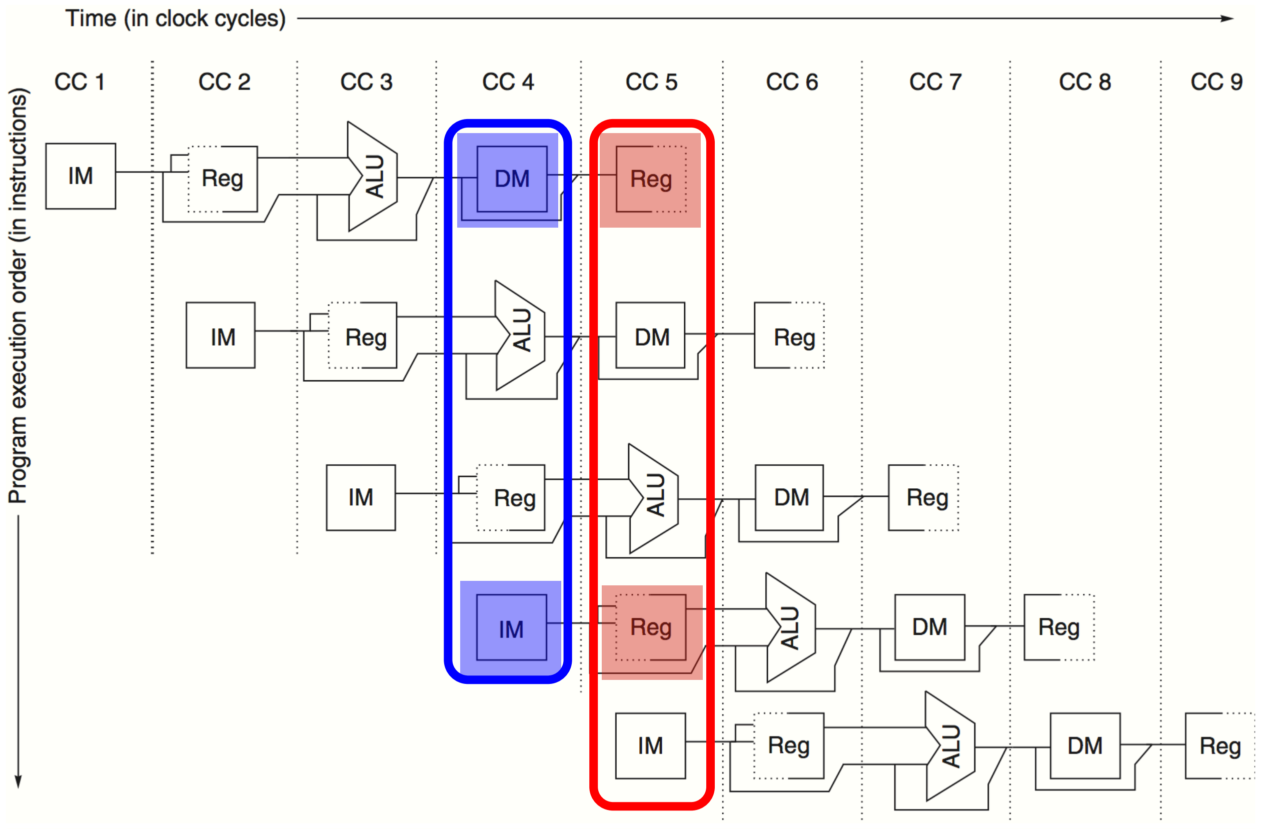
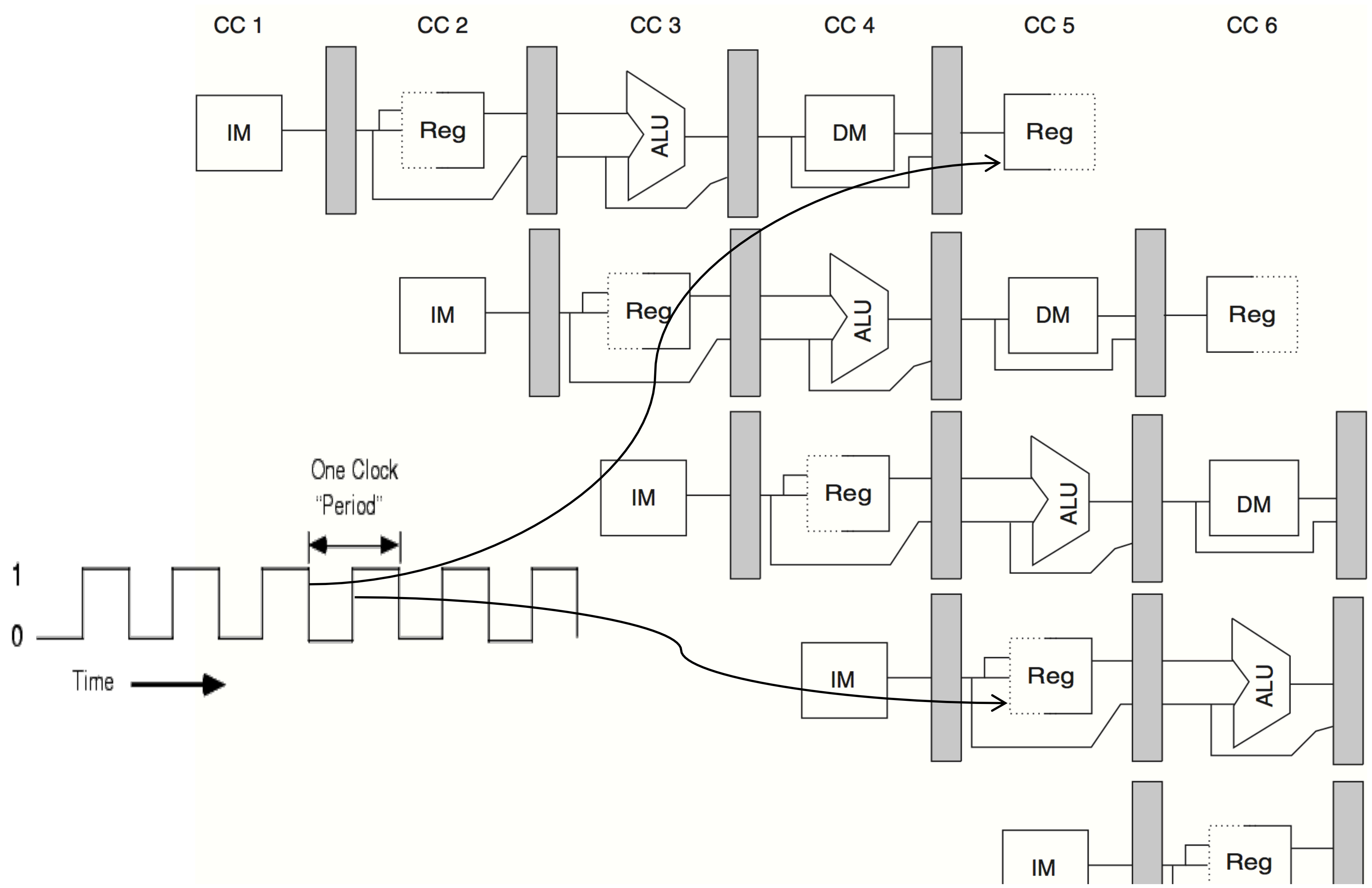


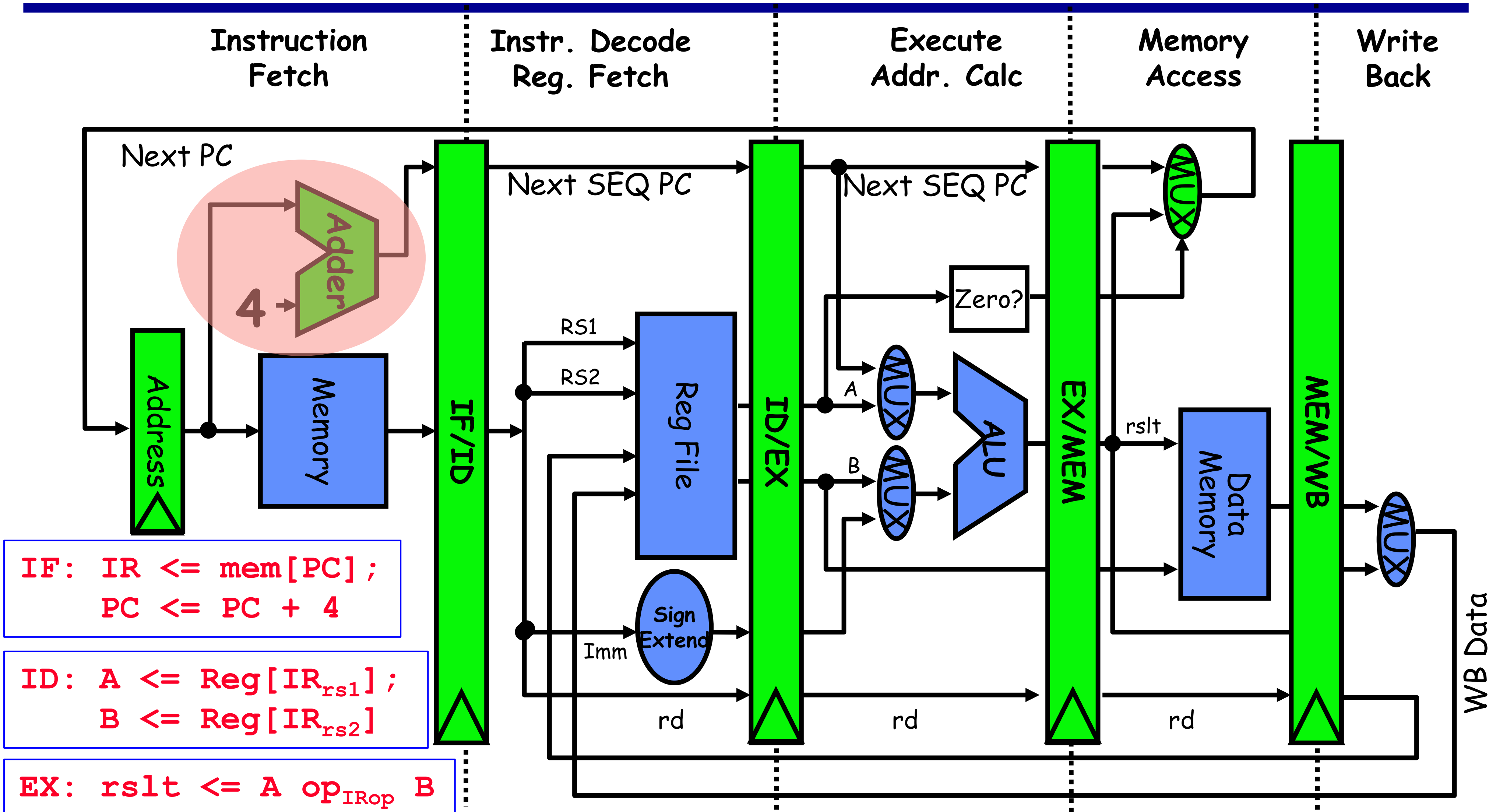
Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on

Pipeline Registers: Staging Data between Pipeline Stages

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



Main Operations in Each Pipeline Stage



```
IF:  IR <= mem[PC];
     PC <= PC + 4
```

ID: A <= Reg[IR_{rs1}];
B <= Reg[IR_{rs2}]

EX: rslt <= A op_{IRop} B

MEM: WB <= rslt

WB: Reg[IR_{rd}] <= WB

Pipeline Registers for Data Staging between Pipeline Stages
Named as: IF/ID, ID/EX, EX/MEM, and MEM/WB

Detailed Operations in 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[Rs1]]; ID/EX.B ← Regs[IF/ID.IR[Rs2]] **ID/EX.NPC ← IF/ID.NPC;** ID/EX.Imm ← extend(IF/ID.IR[Imm]); ID/EX.Rw ← IF/ID.IR[Rs2 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	

Branch requires 3 cycles; **Store requires 4 cycles,** and all other instructions require 5 cycles.

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 limited by increasing execution time.
- Pipeline overhead
  - Unbalanced pipeline stage
    - branch: 3 cycles; store: 4 cycles and else: 5 cycles
  - Pipeline stage overhead;
  - Pipeline register delay;
  - Clock skew
    - Same sourced clock signal arrives at different components at different time

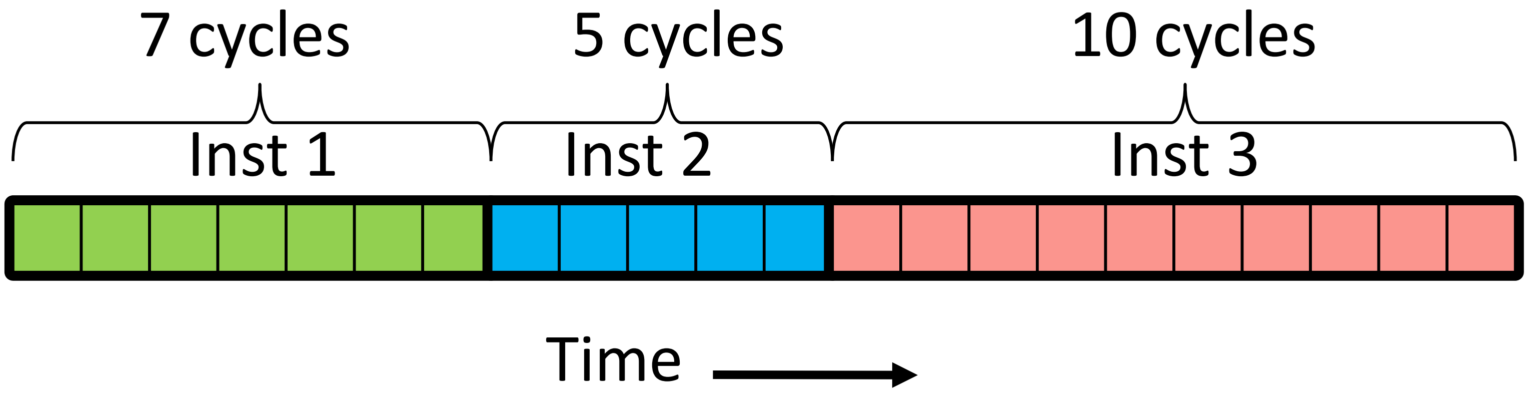
Processor Performance

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

- 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 with Different Instructions, Unpipelined

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$



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

For Questions A.3 in Assignment #1

- **60% branch instructions are taken**

Instruction	**Clock cycles**
All ALU operations	1.0
Loads	3.5
Stores	2.8
Branches	
Taken	4.0
Not taken	2.0
Jumps	2.4

Average the instruction frequencies of gobmk and mcf to obtain the instruction mix. You may assume that all other instructions (for instructions not accounted for by the types in Table A.29) require 3.0 clock cycles each.

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

CPI for Standard Pipeline CPU

- CPI = 1

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**

- Speedup: Unpipelined/pipelined = # pipelined stages
- Thus pipelined time in general:

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

Pipeline Performance (2/2)

Example Consider the unpipelined processor in the previous section. Assume that it has a [redacted] 0.5 ns clock cycle) and that it uses four cycles for ALU operations and branches and five 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 0.1 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

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI}\\ &= 0.5\,\text{ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5]\\ &= 0.5\text{ ns} \times 4.4\\ &= 2.2\text{ ns}\end{aligned}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 0.5 + 0.1 or 0.6 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}\\ &= \frac{2.2\text{ ns}}{0.6\text{ ns}} = 3.7\text{ times}\end{aligned}$$

Pipeline Hazards

- Hazards prevent the next instruction in the instruction steam from executing during its designated clock cycle
  - **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.

Performance with Pipeline Stall (1/2)

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

$$\boxed{\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}\\ &= 1 + \text{Pipelined stall clock cycles per instruction}\end{aligned}}$$

Performance with Pipeline Stall (2/2)

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

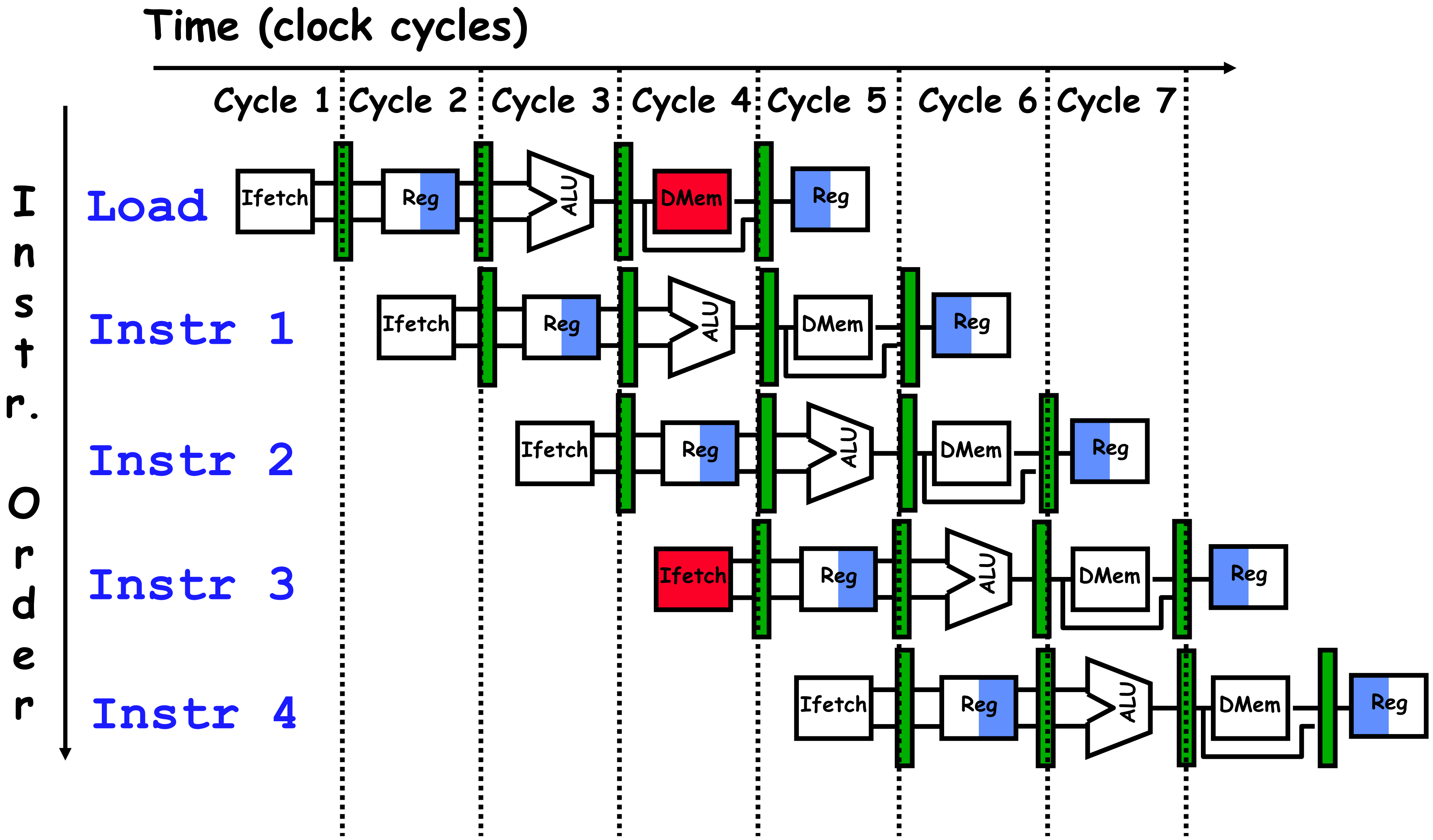
$$= \frac{1}{1+\text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

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

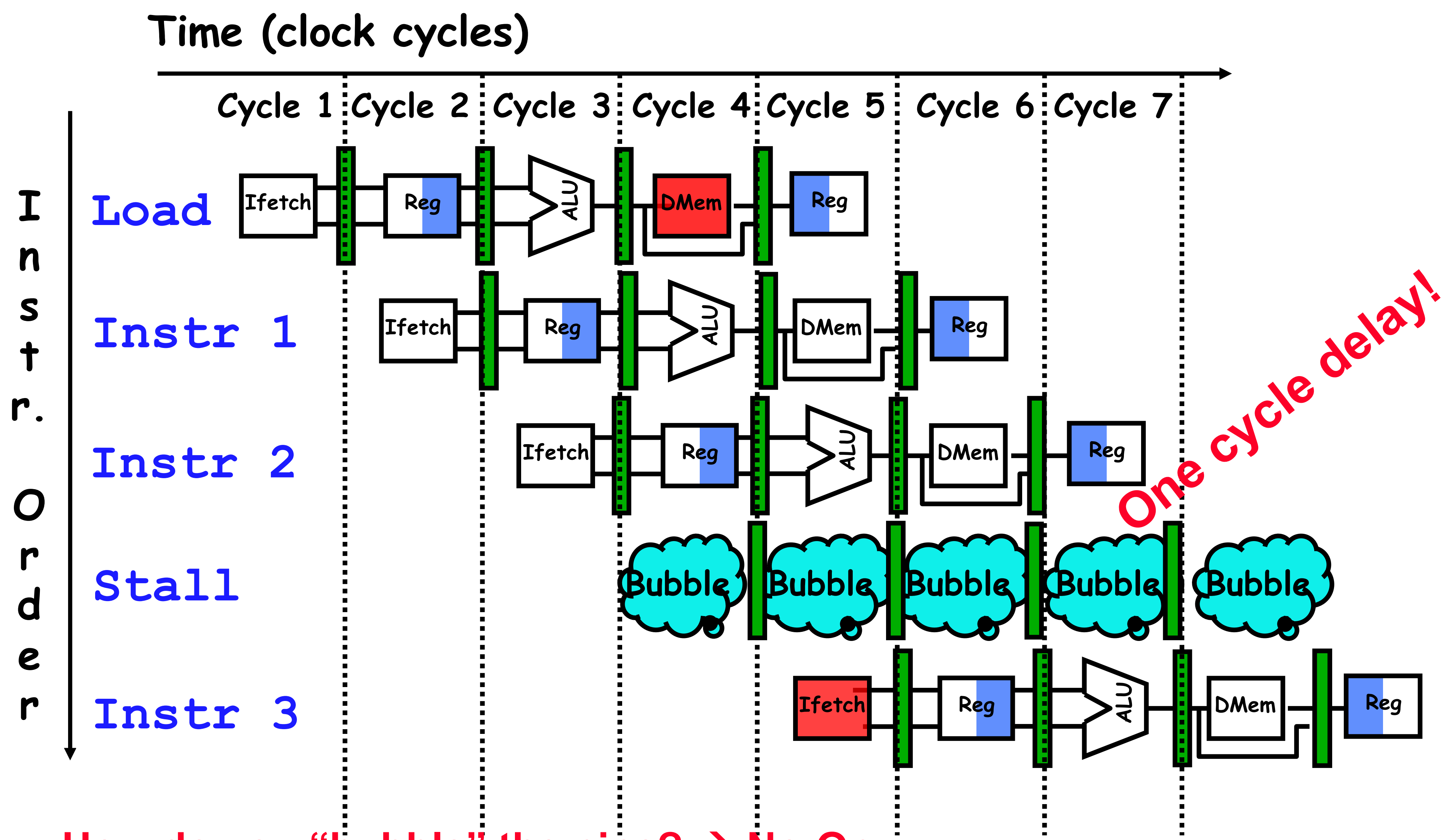
Structure Hazards

- Structure Hazards
  - If some combination of instructions cannot be accommodated because of **resource conflict**:
    - Resources: functional units, register, memory, etc.
  - Occur when
    - Some functional unit is not fully pipelined, or
    - No enough duplicated resources.
  - One example, both IF and MEM stages need to access memory

One Memory Port→Structural Hazards



One Memory Port/Structural Hazards



How do you "bubble" the pipe? → No-Op

Summary of Structure Hazard

- To address structure hazard, have separate memory access for instructions.
  - Splitting the cache into separate *instruction* and *data caches*, or
  - Use a set of buffers, e.g. *instruction buffers*, to hold instruction;
- However, it will increase cost
  - 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

- Pipelining enables overlapping execution of multiple instructions → Instruction Level Parallelism
- 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.

```
ADD  R1, R2, R3
SUB  R4, R1, R5
AND  R6, R1, R7
OR   R8, R1, R9
XOR  R10, R1, R11
```

Data Hazard on R1

Time (clock cycles)

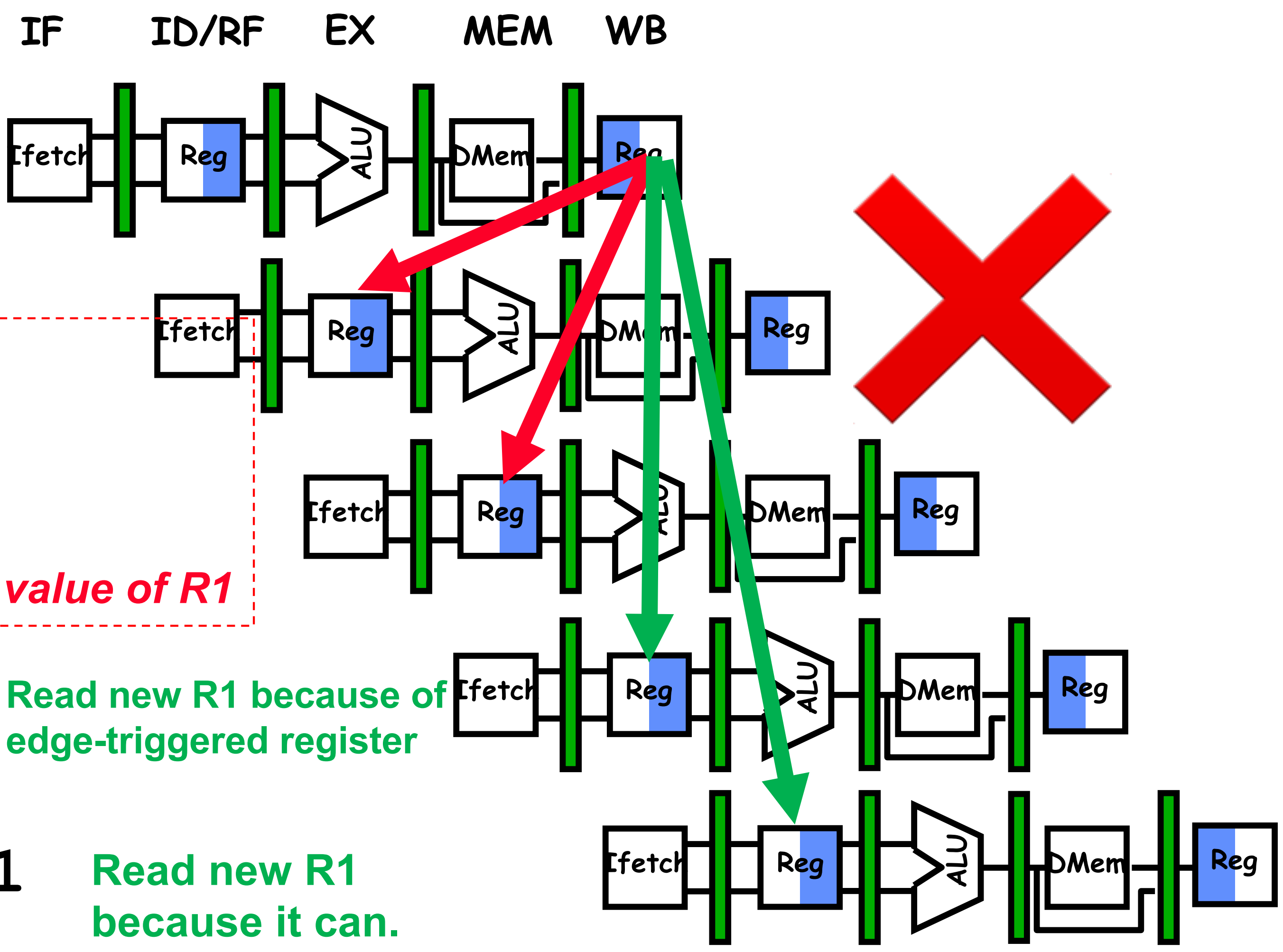
Instr. Order

```
ADD  R1,R2,R3
SUB  R4,R1,R5
AND  R6,R1,R7
OR   R8,R1,R9
XOR  R10,R1,R11
```

Read old value of R1

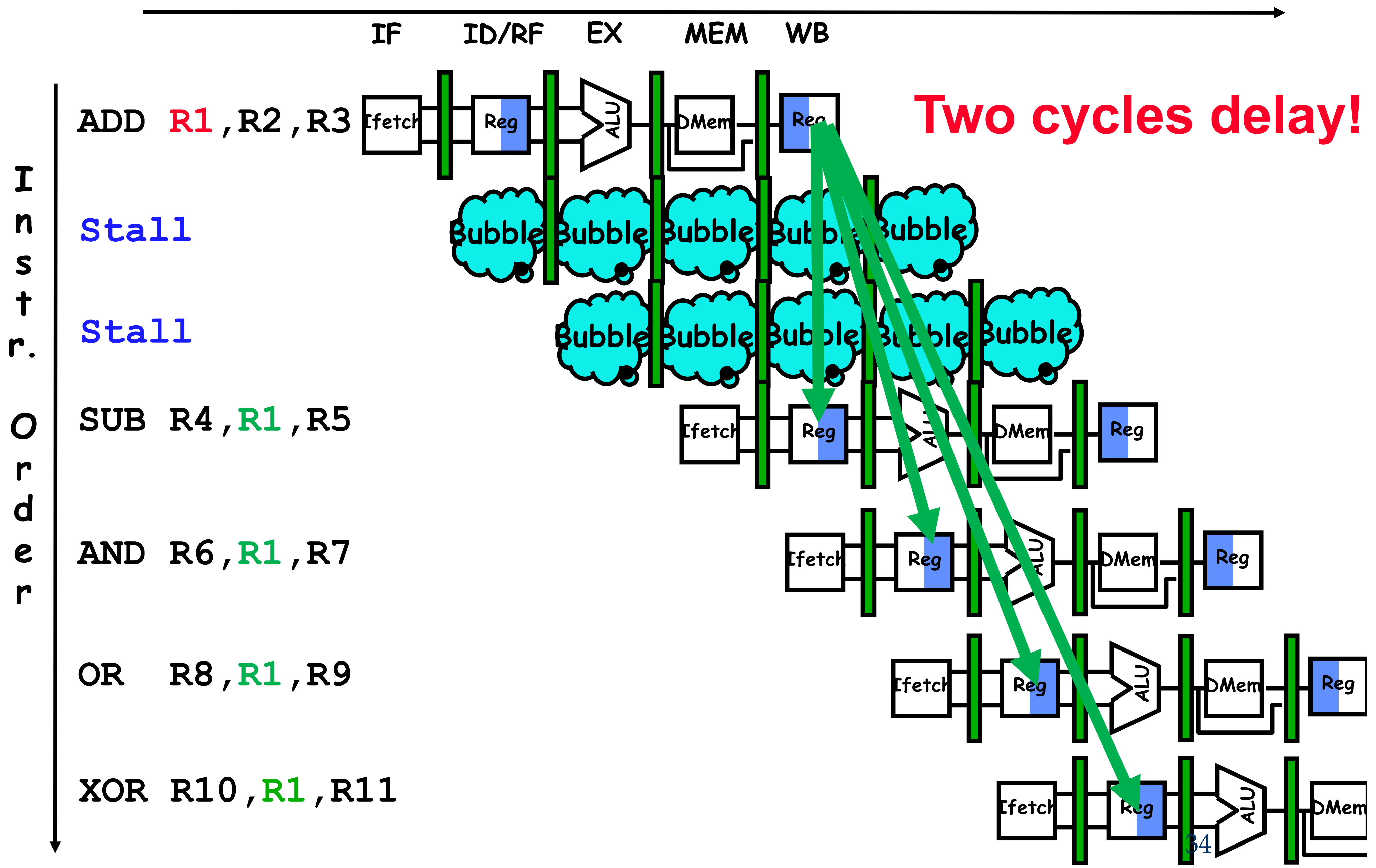
Read new R1 because of edge-triggered register

Read new R1 because it can.



Solution #1: Insert stalls

Time (clock cycles)



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

Time (clock cycles)

Pipeline register: Staging data along the pipeline

No cycle delay!

I
n
s
t
r.

O
r
d
e
r

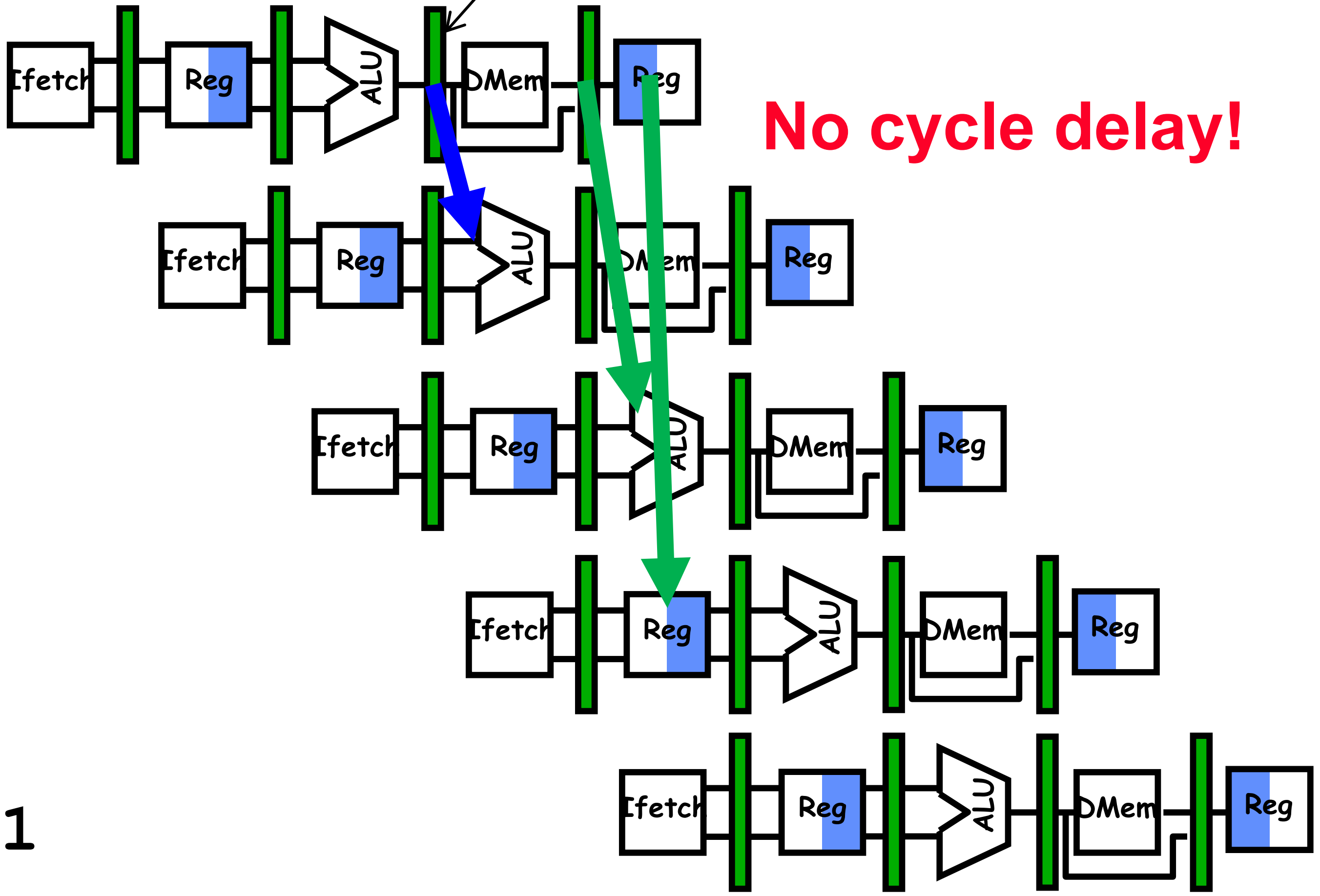
ADD R1,R2,R3

SUB R4,R1,R5

AND R6,R1,R7

OR R8,R1,R9

XOR R10,R1,R11



Three Generic Data Hazards (1/3)

- **Read After Write (RAW)**
  - Instr$_J$ tries to **READ** operand **AFTER** Instr$_I$ **WRITES** it

```
I:  ADD R1,R2,R3
J:  SUB R4,R1,R3
```

- Caused by a "**true dependence**" (in compiler nomenclature). This hazard results from an actual need for communication.
  - SUB needs value produced by ADD
- **Nature of the computation and we cannot avoid.**

Three Generic Data Hazards (2/3)

- **Write After Read (WAR)**
  - Instr_J **WRITEs** operand **AFTER** 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", not the value stored in R1.
- **Can't happen in 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 **AFTER** Instr$_I$ **WRITEs** it.

```
I: SUB R1,R4,R3
J: ADD R1,R2,R3
K: MUL R6,R1,R7
```

- 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:**
  - **All writes are ordered and take place in stage 5**
- WAR and WAW hazards occur in complex pipelines
- Read After Read – RAR is NOT a hazard

Double/Triple Data Hazards

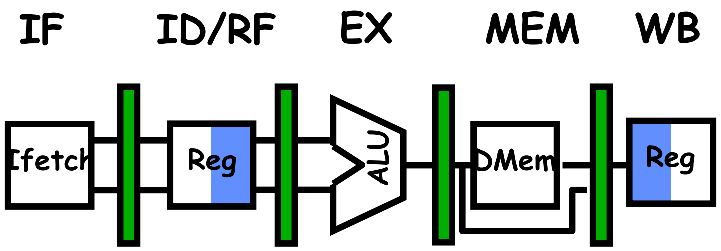
- Consider the sequence:

```
ADD R1,R1,R2
SUB R1,R1,R3
AND R1,R1,r4
```

- Hazards occur
  - ADD → SUB: RAW and WAW on R1
  - SUB → SUB: WAR on R1
  - SUB → AND: RAW on R1
  - …

RAW Hazards with Load/Store

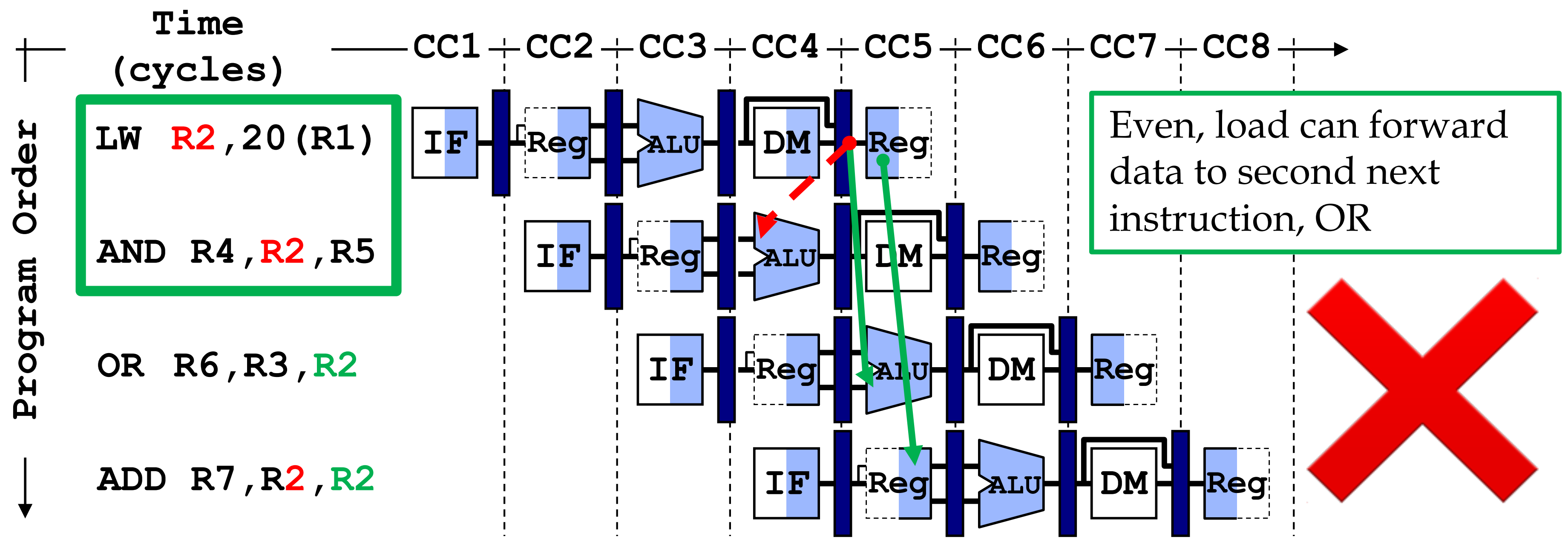
- **lw R2, 20(R1): load a word from memory @ [R1]+20 into R2**
  - ID/RF: Read register R1: [R1] (rs1 register)
  - EX: Calculate effective address: [R1] + 20
  - **MEM: Memory read from [R1]+20**
    - **Data is available in MEM|WB**
    - **Unlike ALU: data is available in EX|MEM**
  - WB: data write back to R2 (rd register)



- **sw R4,12(R1): store a word in R4 in the memory @ [R1]+12**
  - ID/RF: Read register R1 and R4 (rs1 and rs2 register, no rd register)
    - R1 is needed in EX, and R4 is needed in MEM
  - EX: Calculate effective address: [R1] + 12
  - **MEM: Memory write to [R1]+12**
    - **Need R4 to be available**
    - **Unlike ALU, data needs to be available in ID|EX**
  - No need WB

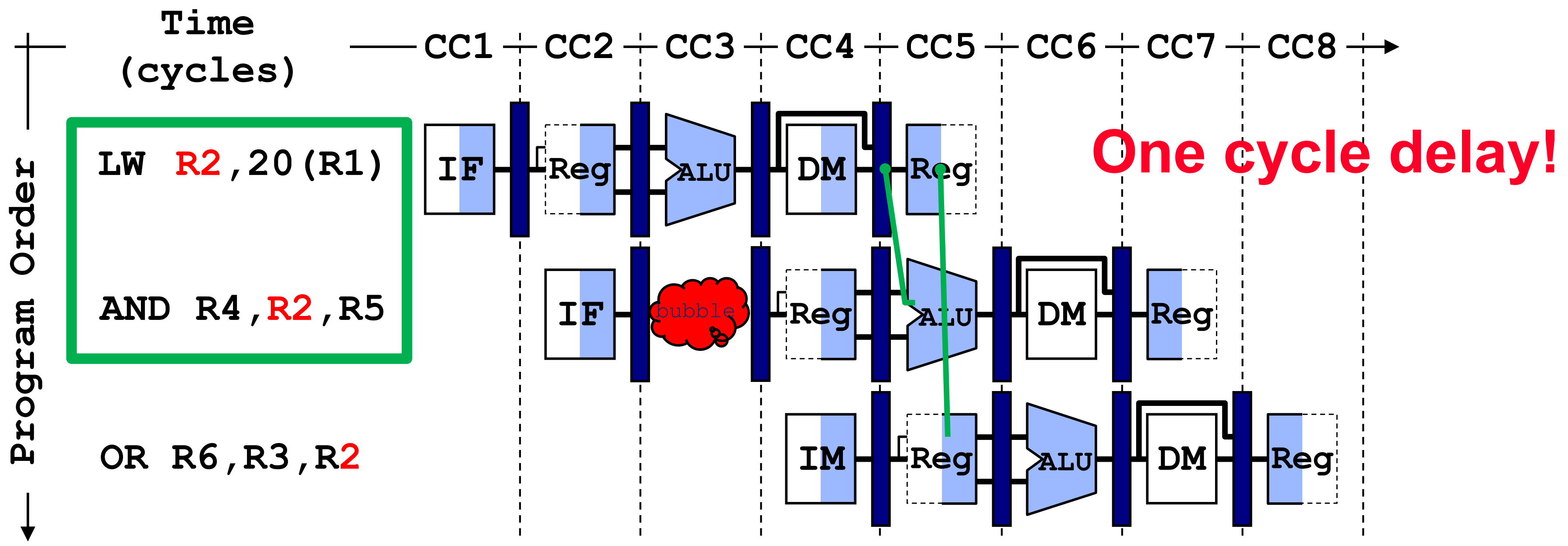
Load Delay (Load→EXE-Use RAW Hazard)

- **Not all RAW data hazards can be forwarded**
  - **Load has a delay that cannot be eliminated by forwarding**
  - **lw R2, 20(R1): load data from memory [R1]+20 into R2**
- In the example shown below …
  - **Unlike ALU intrus, LW does not have data until CC4 end**
  - AND wants data at beginning of CC4 - NOT possible



Solution: Stall the Pipeline for One Cycle

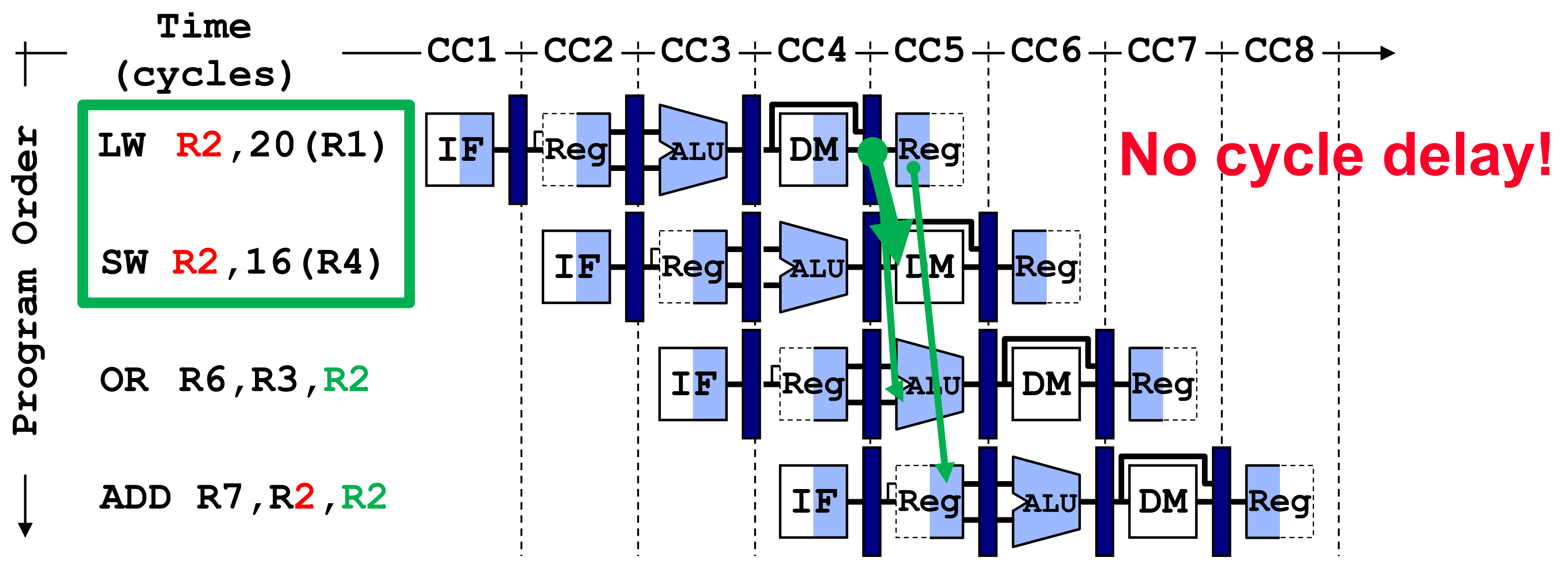
- Freeze the PC and the IF/ID registers
  - No new instruction is fetched and instruction after load is stalled
- Allow the Load in ID/EX register to proceed
- Introduce a bubble into the ID/EX register
- Load can forward data after stalling next instruction



Forwarding CAN Avoid Delay in Load→ MEM-USE RAW Hazard

```
LW R2,20(R1)
SW R2,16(R4)
```

- Load a word from memory to R2 and then store R2 to a different memory location. **R2 is rd for LW and Rs2 for SW.**
- Because Store only needs the Rs2 data in the MEM stage, not in the EX stag as for ALU instructions: **Only Store is this special!**

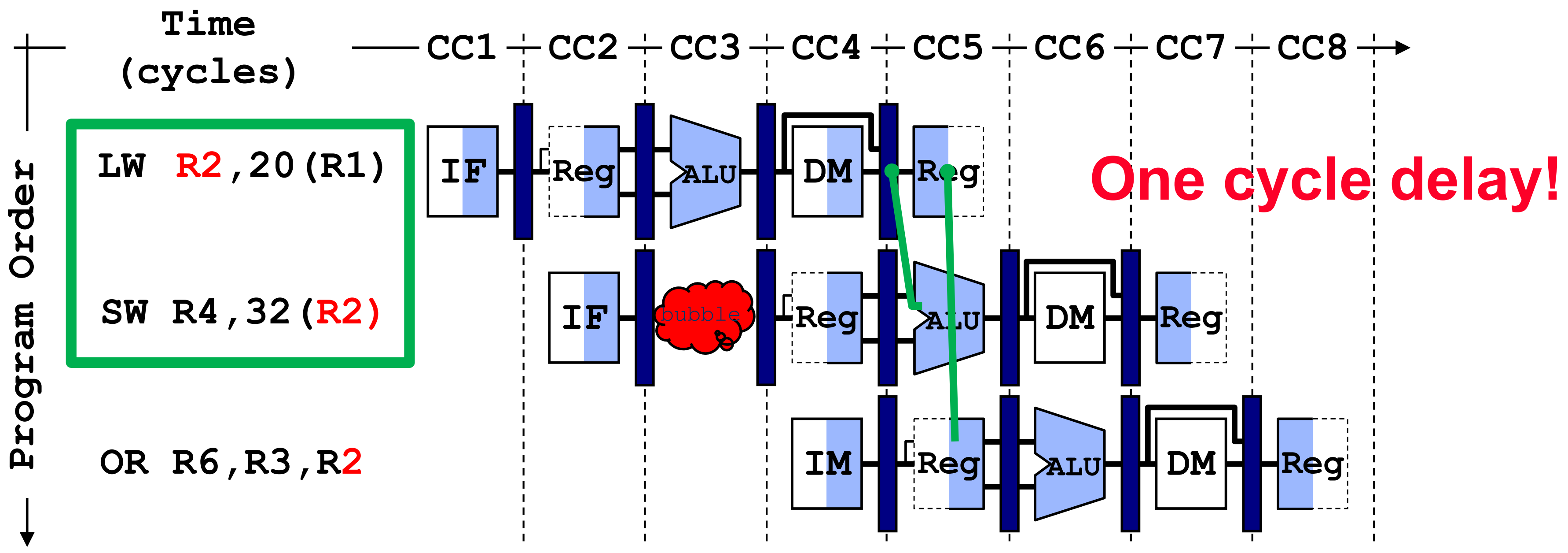


Load→EXE-Use RAW Hazard by Store

- The same as Load→EXE-Use by ALU
  - R2 is Rs1 for SW

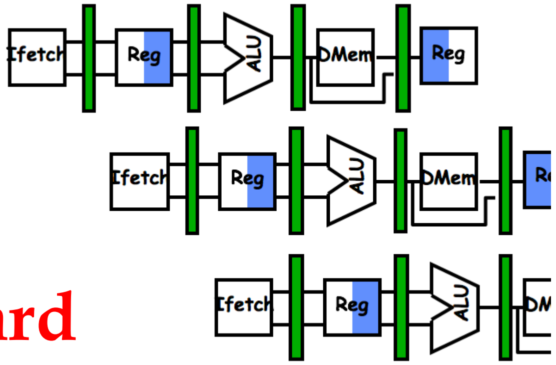
```
LW  R2,20(R1)
SW R4,32(R2)
```

- Introduce a bubble into the ID/EX register



Summary 1/2: RAW Hazards and NO-Forwarding

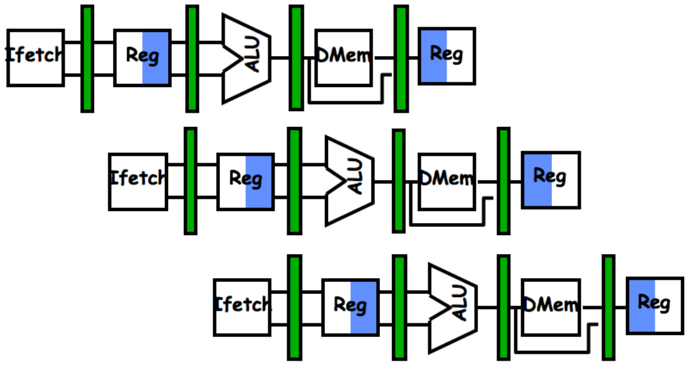
- Stages for **Input (Rs1 and Rs2)** and **Output (Rd):**
  - **ALU and LD need Rs when entering EX**
  - **Store needs Rs1 at EX for calculating EA and then Rs2 at MEM for storing data**
  - **ALU instructions produce Rd at the end of EX stage**
  - **LD produce Rd at the end of MEM stage**



- **If NO forwarding (interlocking):**
  - **2 cycles bubble delay for any RAW hazard**
    - » **Input and Output have to go through register file**

Summary 2/2: RAW Hazards and Forwarding

- Stages for **Input (Rs1 and Rs2)** and **Output (Rd):**
  - **ALU and LD need Rs when entering EX**
  - **Store needs Rs1 at EX for calculating EA and then Rs2 at MEM for storing data**
  - **ALU instructions produce Rd at the end of EX stage**
  - **LD produce Rd at the end of MEM stage**
- **If NO Forwarding (interlocking): 2 cycles delay**



- **Full forwarding (Full Bypassing):**
  - **0 cycle delay for RAW hazard between ALU instructions**
  - **1 cycle delay for Load→EXE-Use RAW hazard**
    - » **EXE-use for both ALU and store**
  - **0 cycle delay for Load→ MEM-Use RAW hazard**
    - » **MEM-Use: Store**
- Two instructions that has RAW data dependency
  - If R and W for the shared register are in the same stage, forwarding will incur no cycle delay

Compiler Scheduling

- Compilers can schedule code in a way to avoid load → ALU-use stalls

  a = b + c; d = e – f;

- Slow code: 2 stall cycles

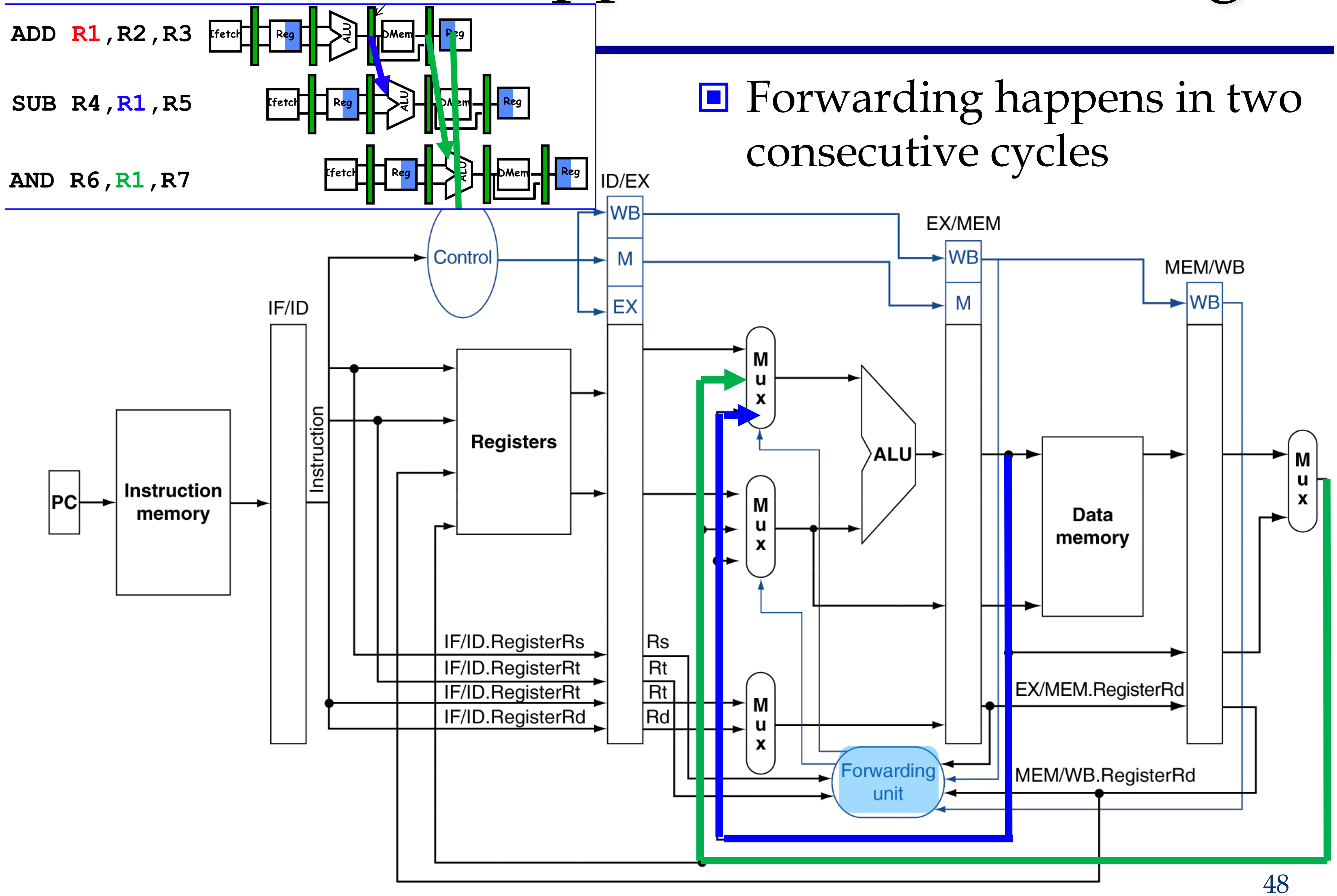
```
lw    r10, (r1)       # r1 = addr b
lw    r11, (r2)       # r2 = addr c
                      # stall
add   r12, r10, r11   # b + c
sw     r12, (r3)      # r3 = addr a
lw     r13, (r4)      # r4 = addr e
lw      r14, (r5)     # r5 = addr f
                      # stall
sub   r15, r13, r14   # e -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

Hardware Support for Forwarding

- Forwarding happens in two consecutive cycles



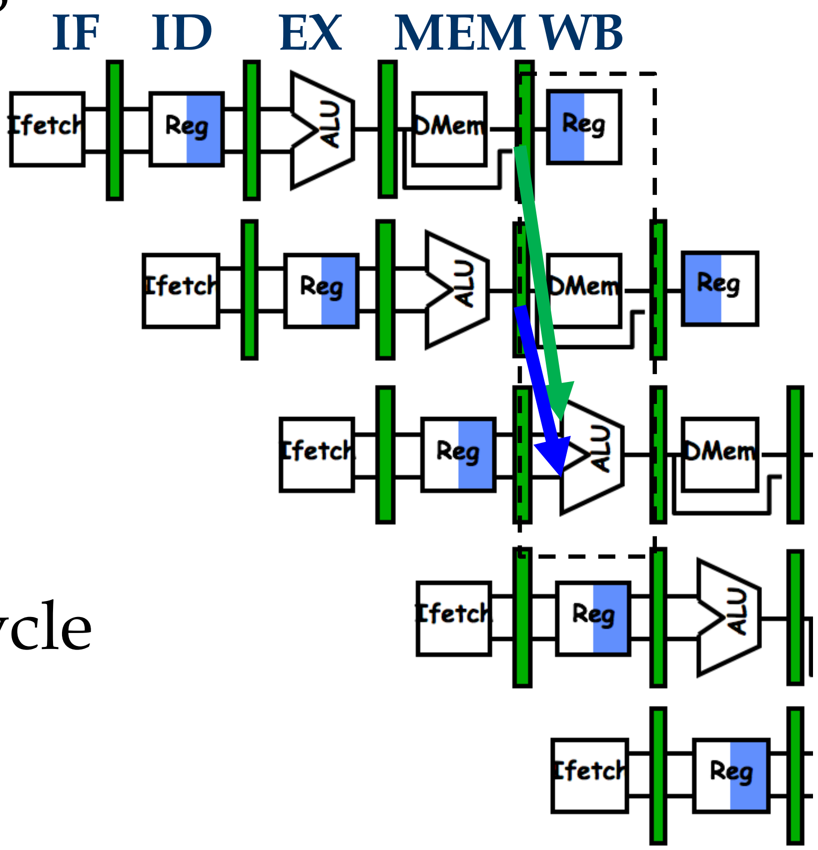
Detecting RAW Hazards

- Current instruction being executed in ID/EX register
- Previous instruction is in the EX/MEM register
- 2nd Previous is in the MEM/WB register

`ADD R1, R2, R3` **#2nd Previous in MEM/WB**

`SUB R6, R4, R5` **#Previous in EX/MEM**

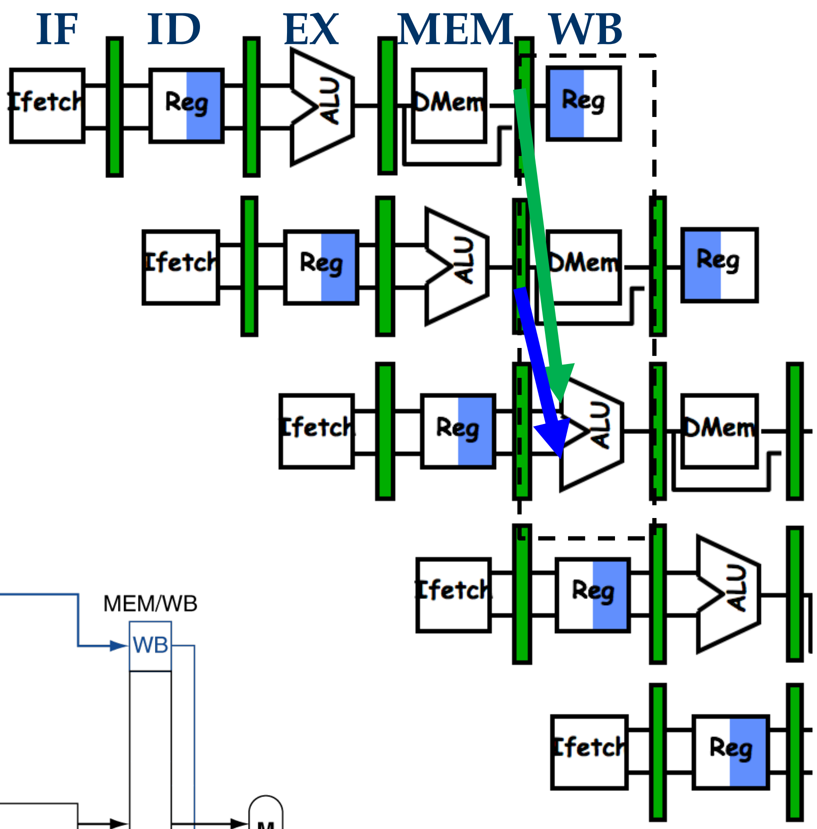
`AND R7, R1, R6` **#Current in ID/EX**

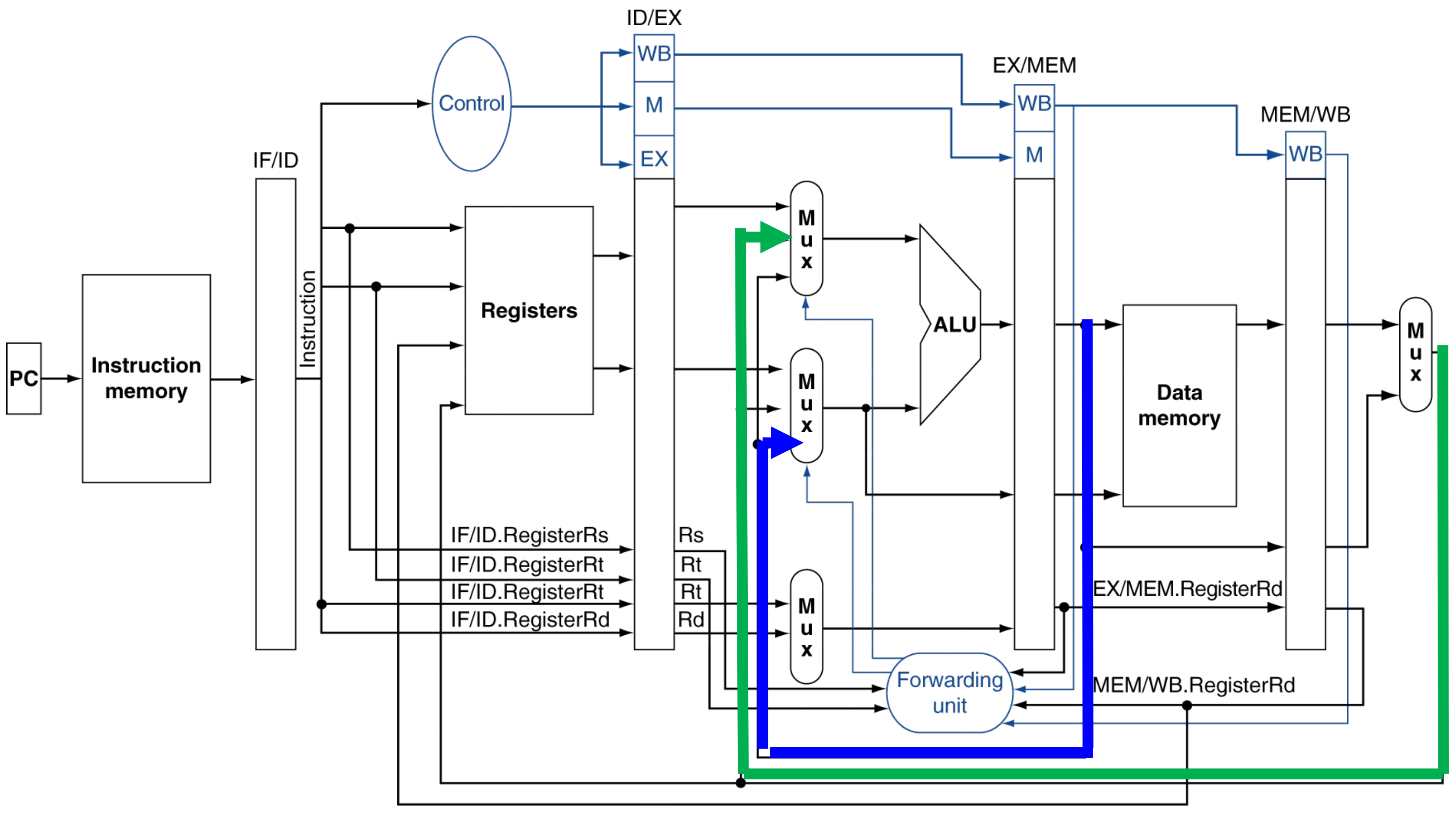


- Forwarding happens in the same cycle

Detecting RAW Hazards

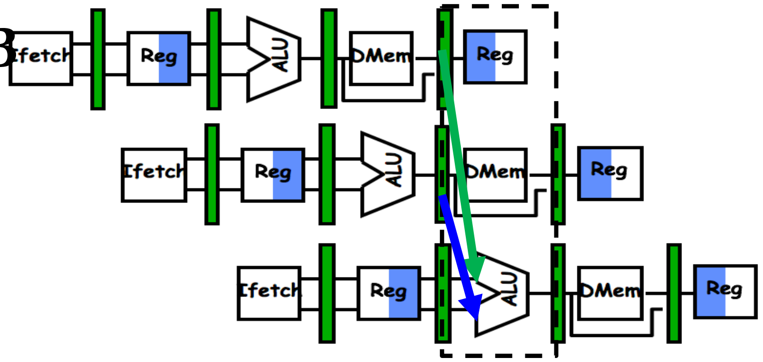
```
ADD R1, R2, R3 #2nd Previous in MEM/WB
SUB R6, R4, R5 #Previous in EX/MEM
AND R7, R1, R6 #Current in ID/EX
```





Detecting RAW Hazards

```
ADD R1, R2, R3  #2nd Previous in MEM/WB
SUB R6, R4, R5  #Previous in EX/MEM
AND R7, R1, R6  #Current in ID/EX
```



- Pass register numbers along pipelin
  - ID/EX.RegisterRs = register number for Rs in ID/EX (Rs1)
  - ID/EX.RegisterRt = register number for Rt in ID/EX (Rs2)
  - ID/EX.RegisterRd = register number for Rd in ID/EX

- RAW Data hazards when
  - **1a. EX/MEM.RegisterRd = ID/EX.RegisterRs**
  - **1b. EX/MEM.RegisterRd = ID/EX.RegisterRt**

  } **Fwd from EX/MEM pipeline reg**

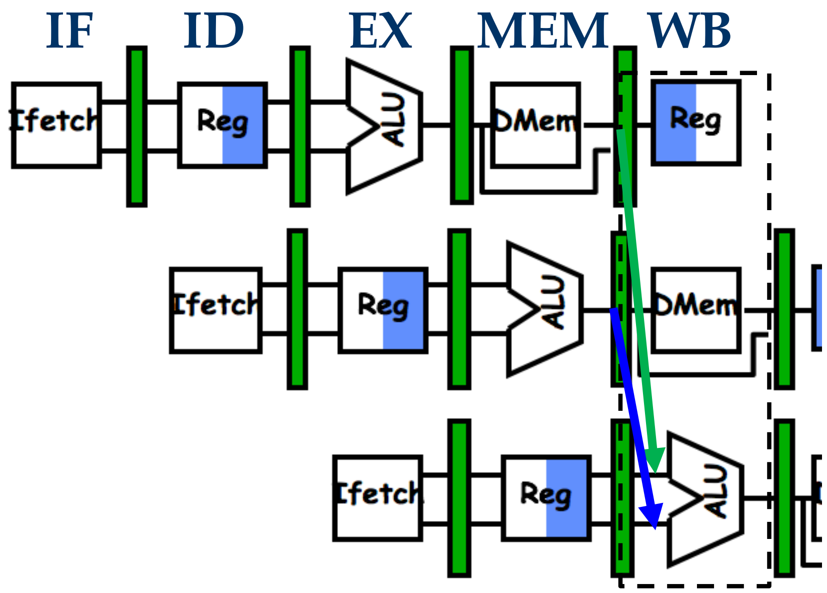
  - **2a. MEM/WB.RegisterRd = ID/EX.RegisterRs**
  - **2b. MEM/WB.RegisterRd = ID/EX.RegisterRt**

  } **Fwd from MEM/WB pipeline reg**

Detecting the Need to Forward

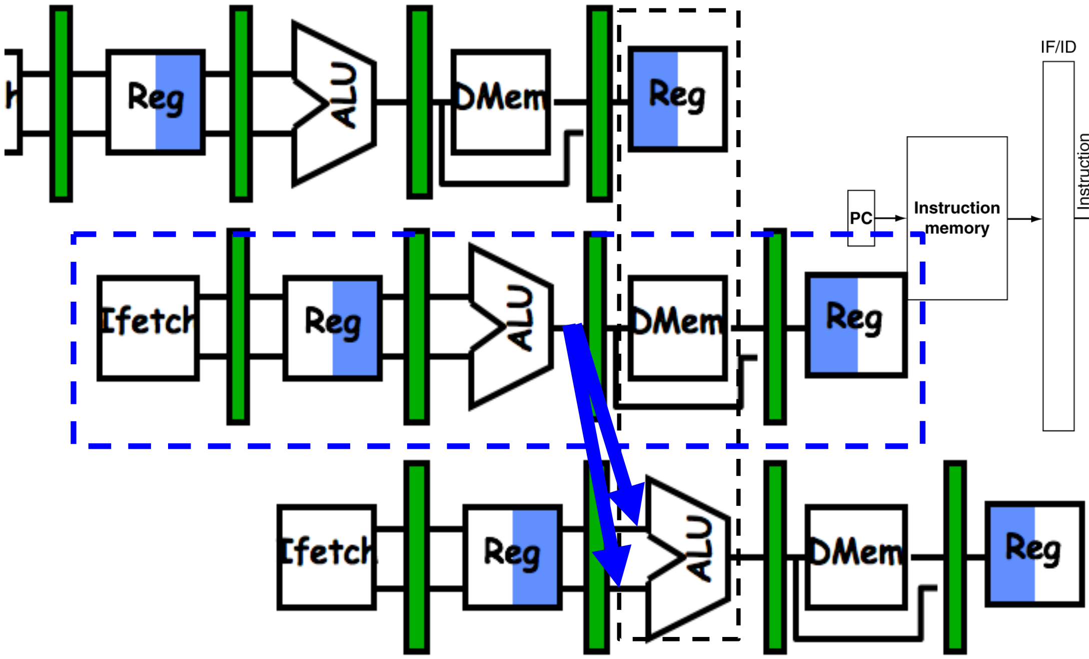
- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not R0
  - EX/MEM.RegisterRd ≠ 0
  - MEM/WB.RegisterRd ≠ 0

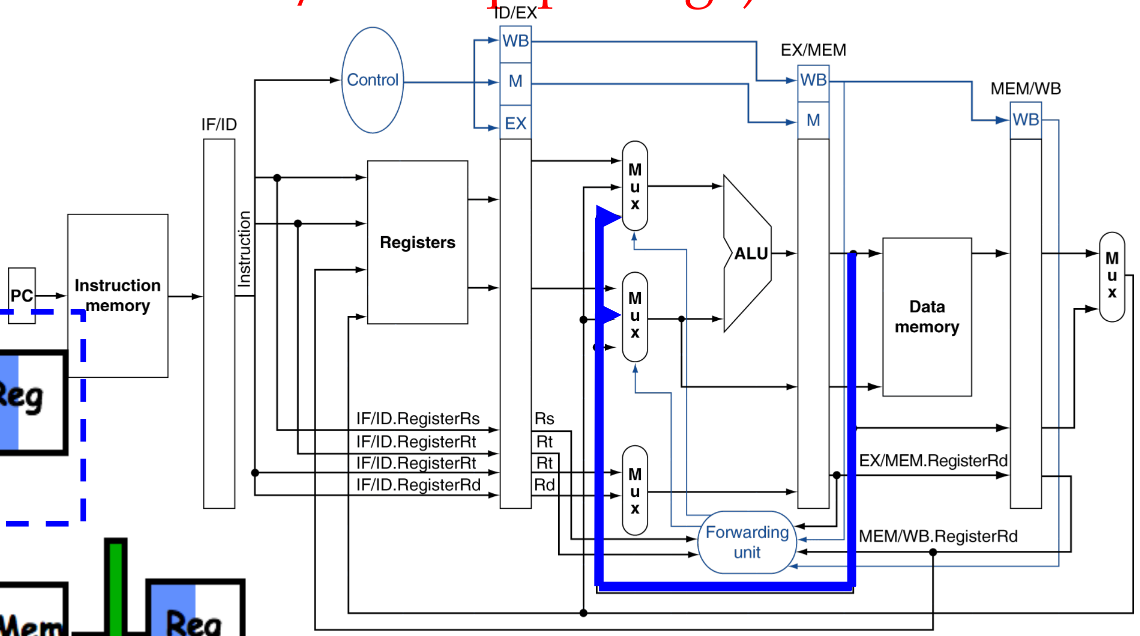
```
ADD R1, R2, R3  #2nd Previous in MEM/WB
SUB R6, R4, R5  #Previous in EX/MEM
AND R7, R1, R6  #Current in ID/EX
```



Forwarding Conditions

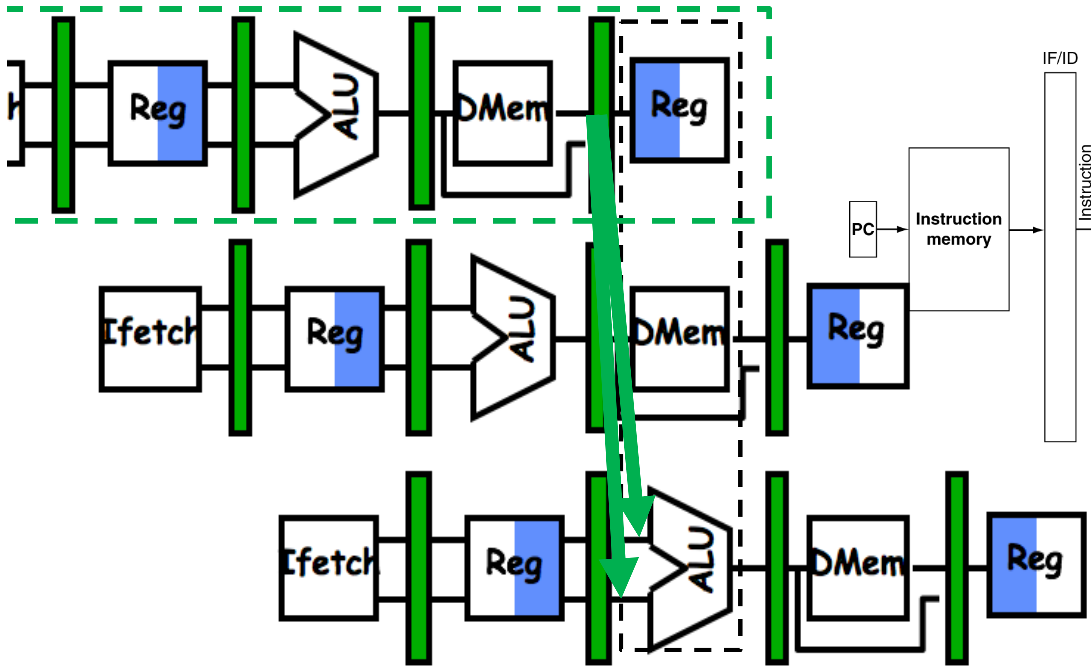
- Detecting RAW hazard with Previous Instruction
  - **if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))**
    ForwardA = 01 (Forward from EX/MEM pipe stage)
  - **if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))**
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

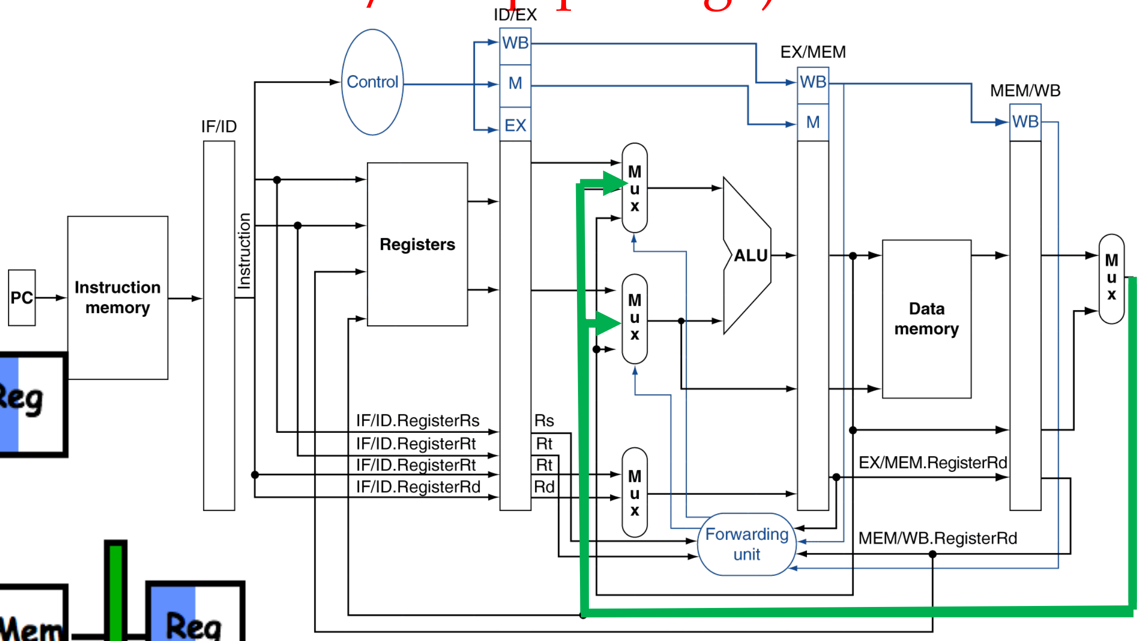




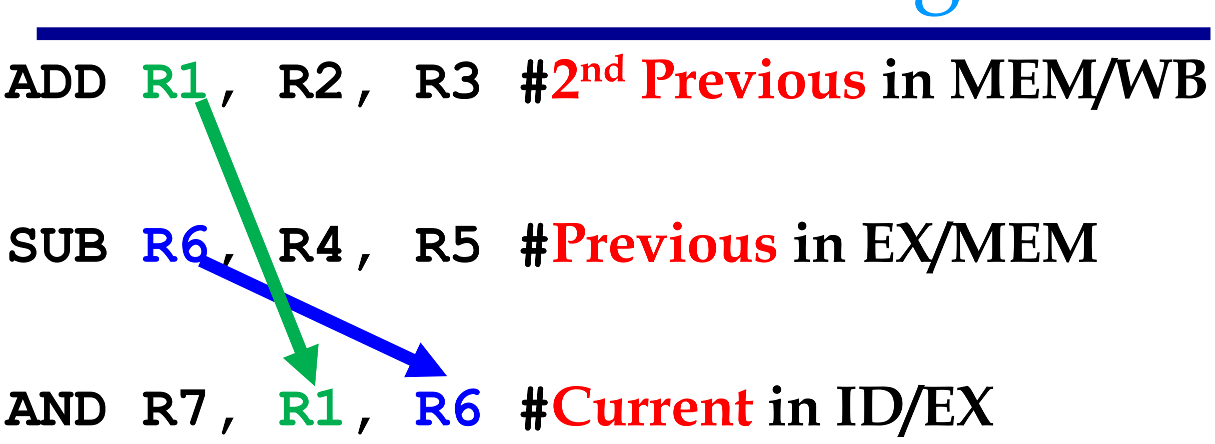
Forwarding Conditions

- 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)

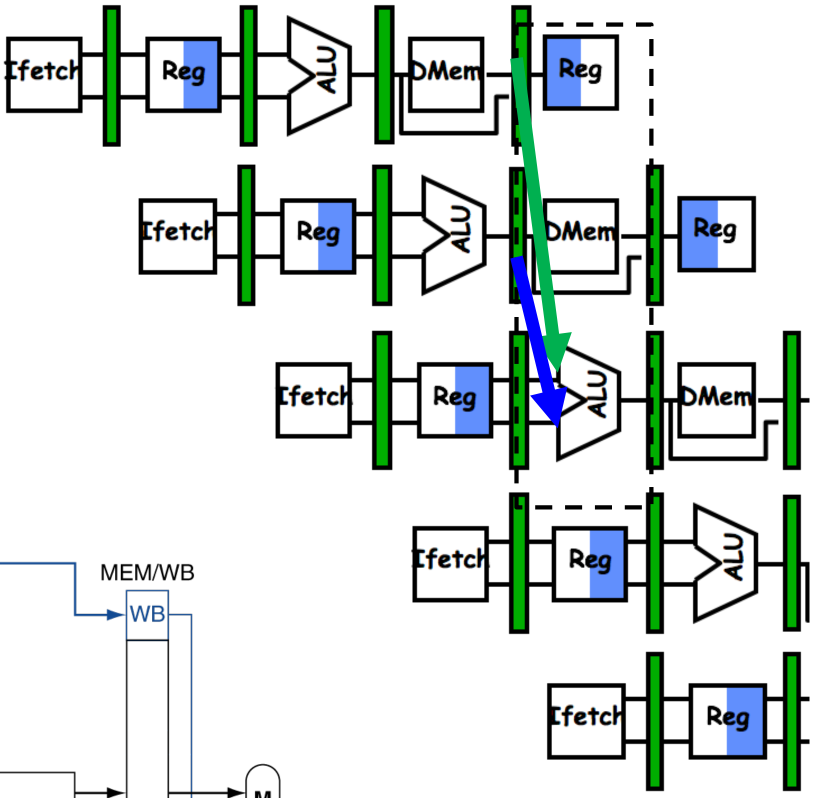


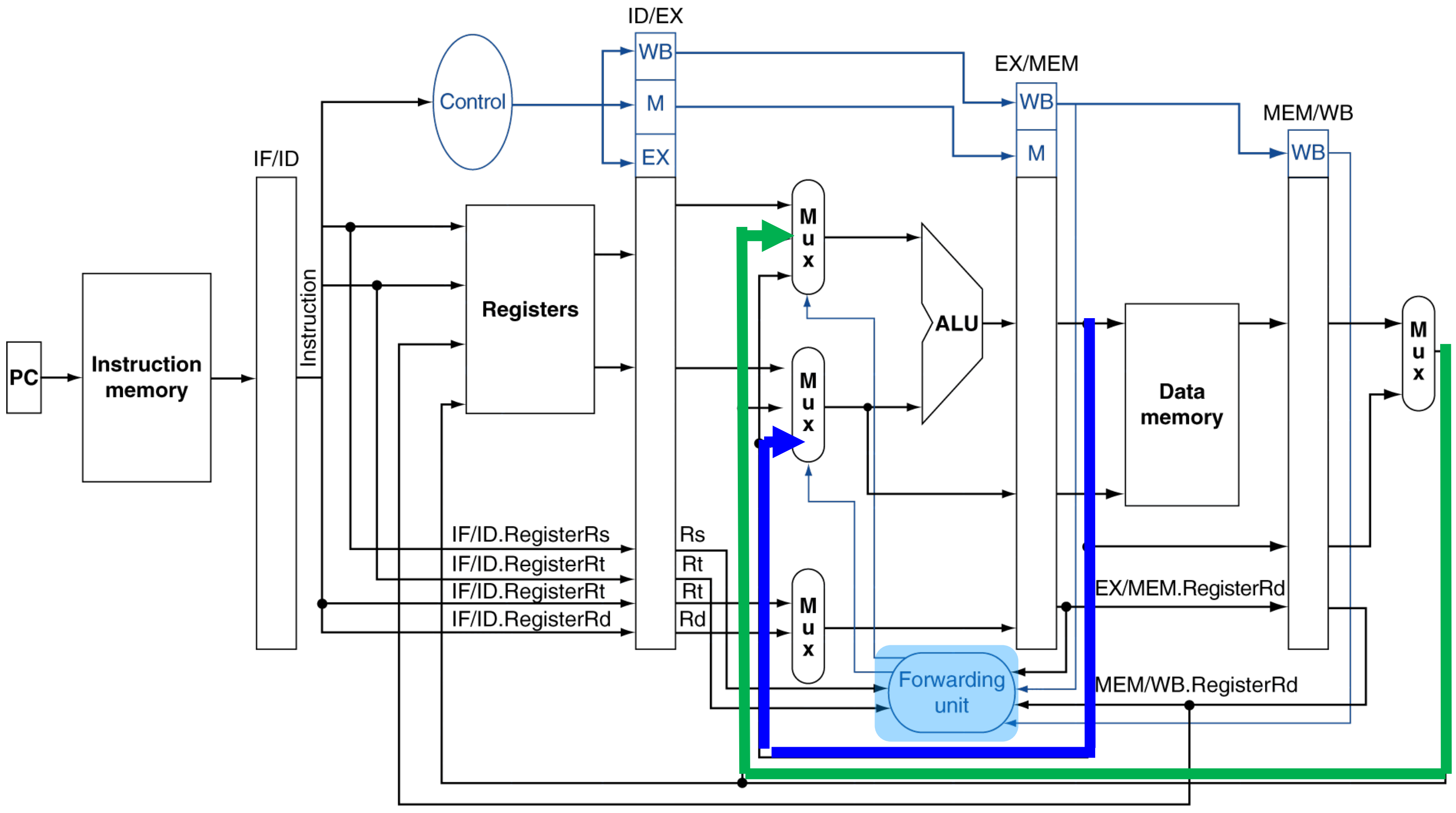


Control Signals During Forwarding: Those Light Blue lines



ADD R1, R2, R3 #2nd Previous in MEM/WB
SUB R6, R4, R5 #Previous in EX/MEM
AND R7, R1, R6 #Current in ID/EX





Control Hazard on Branches: Three Stage Stall

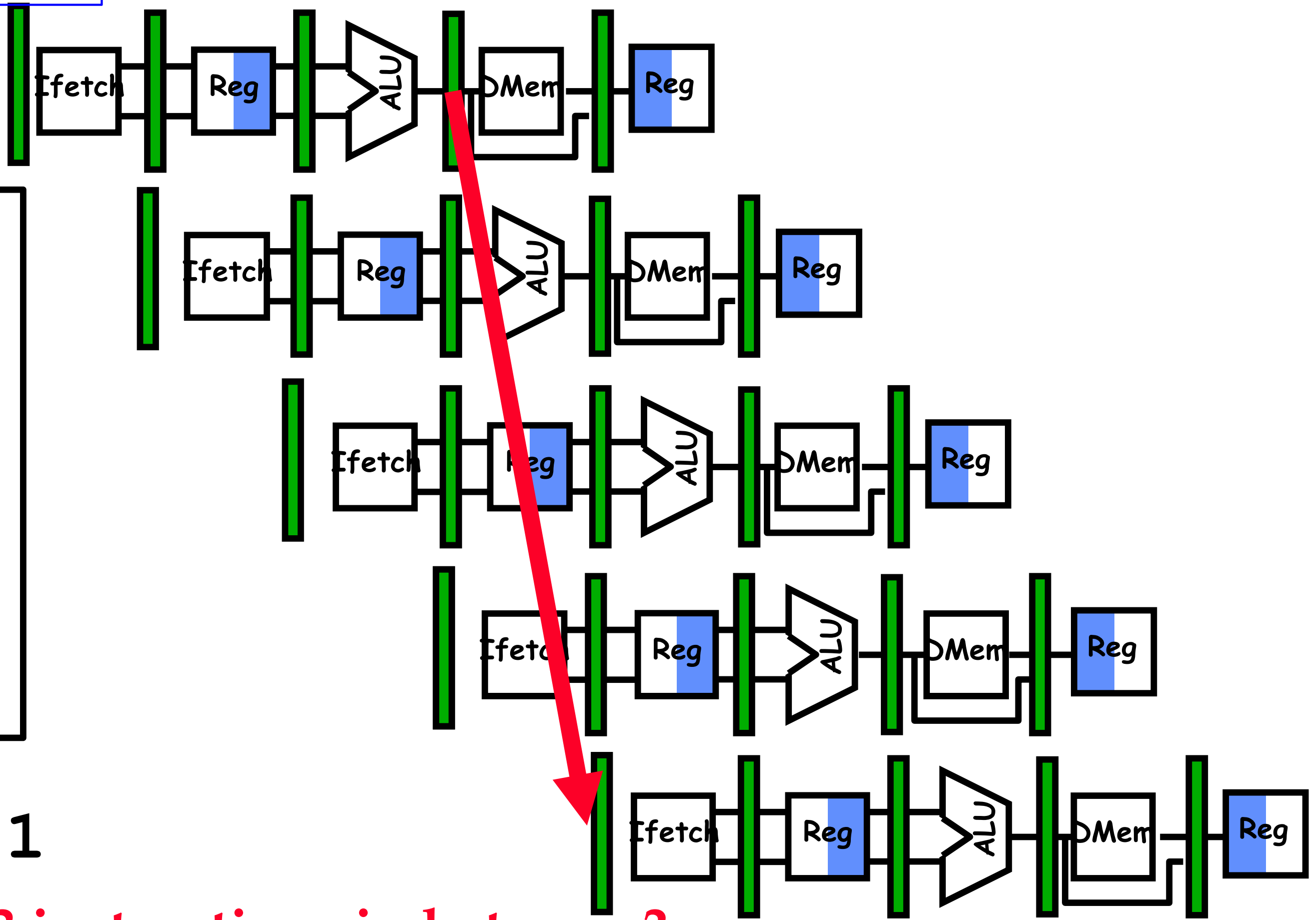
BEQ: If ([R1] == [R3]) Branch to PC + 36

```
10: BEQ R1,R3,36

14: AND R2,R3,R5

18: OR  R6,R1,R7

22: ADD R8,R1,R9
      .........
46: XOR R10,R1,R11
```



What do you do with the 3 instructions in between?

How do you do it?

Where is the "commit"?

Branch/Control Hazards

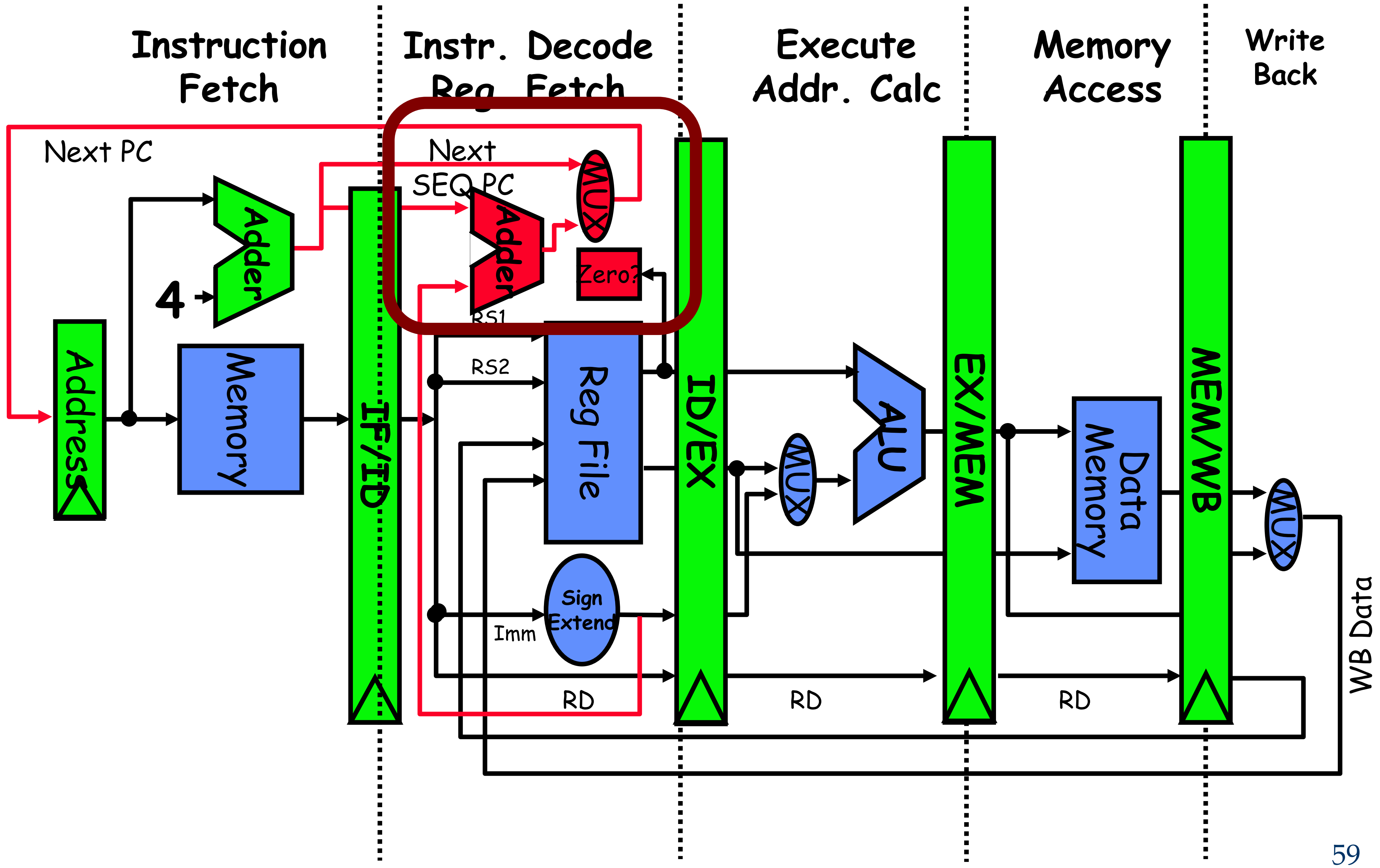
- Branch instructions can cause great performance loss
- Branch instructions need two things:
  - Branch Result — Taken or Not Taken
  - Branch Target
    - PC + 4 — If Branch is NOT taken
    - PC + 4 + 4 $\times$ imm — If Branch is Taken
- **For our pipeline: 3-cycle branch delay**
  - PC is updated 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

- CPI = 1 if without branch stalls, and 30% branch
- If stalling 3 cycles per branch
  - => new CPI = $1+0.3\times 3 = 1.9$

- 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

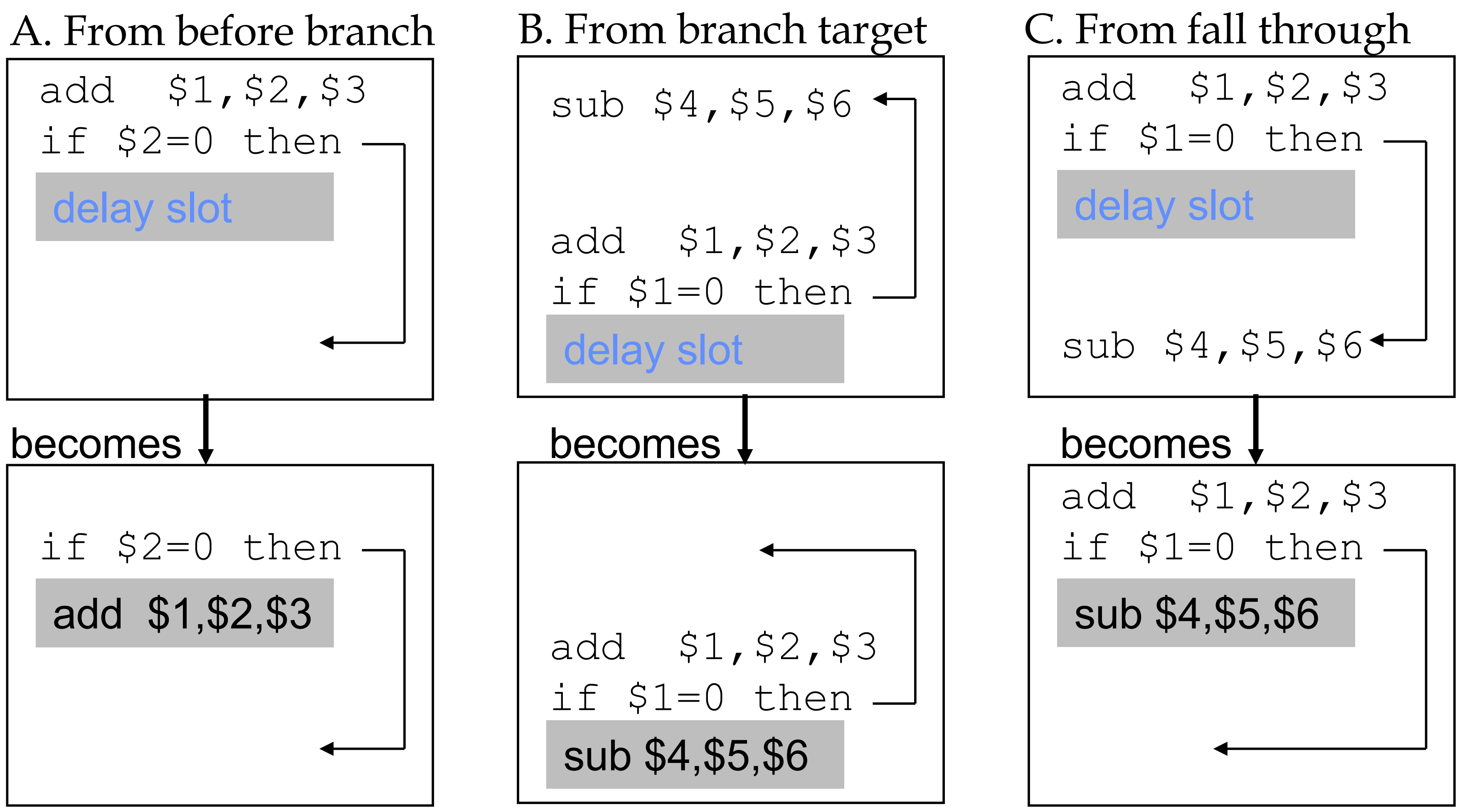
Four Branch Hazard Alternatives

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

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

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

Scheduling Branch Delay Slots



- 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

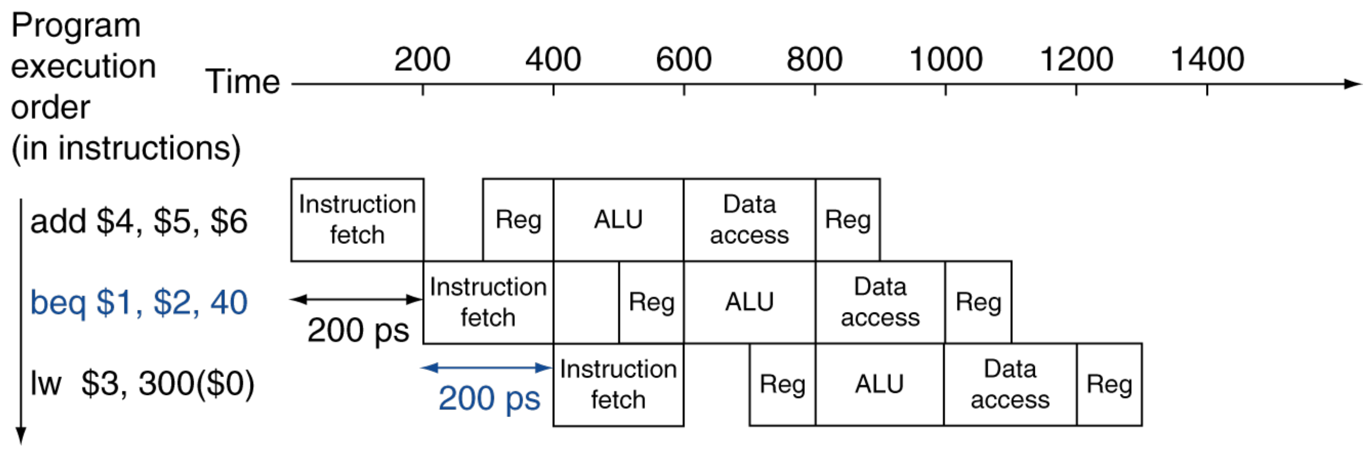
Branch scheme	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Branch Prediction

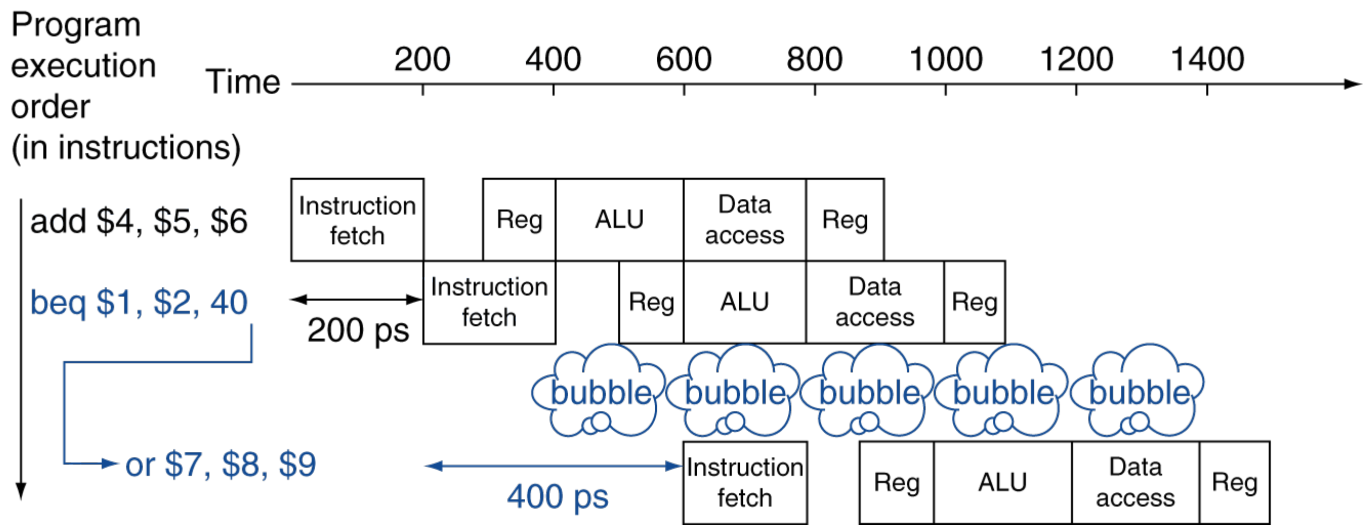
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

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

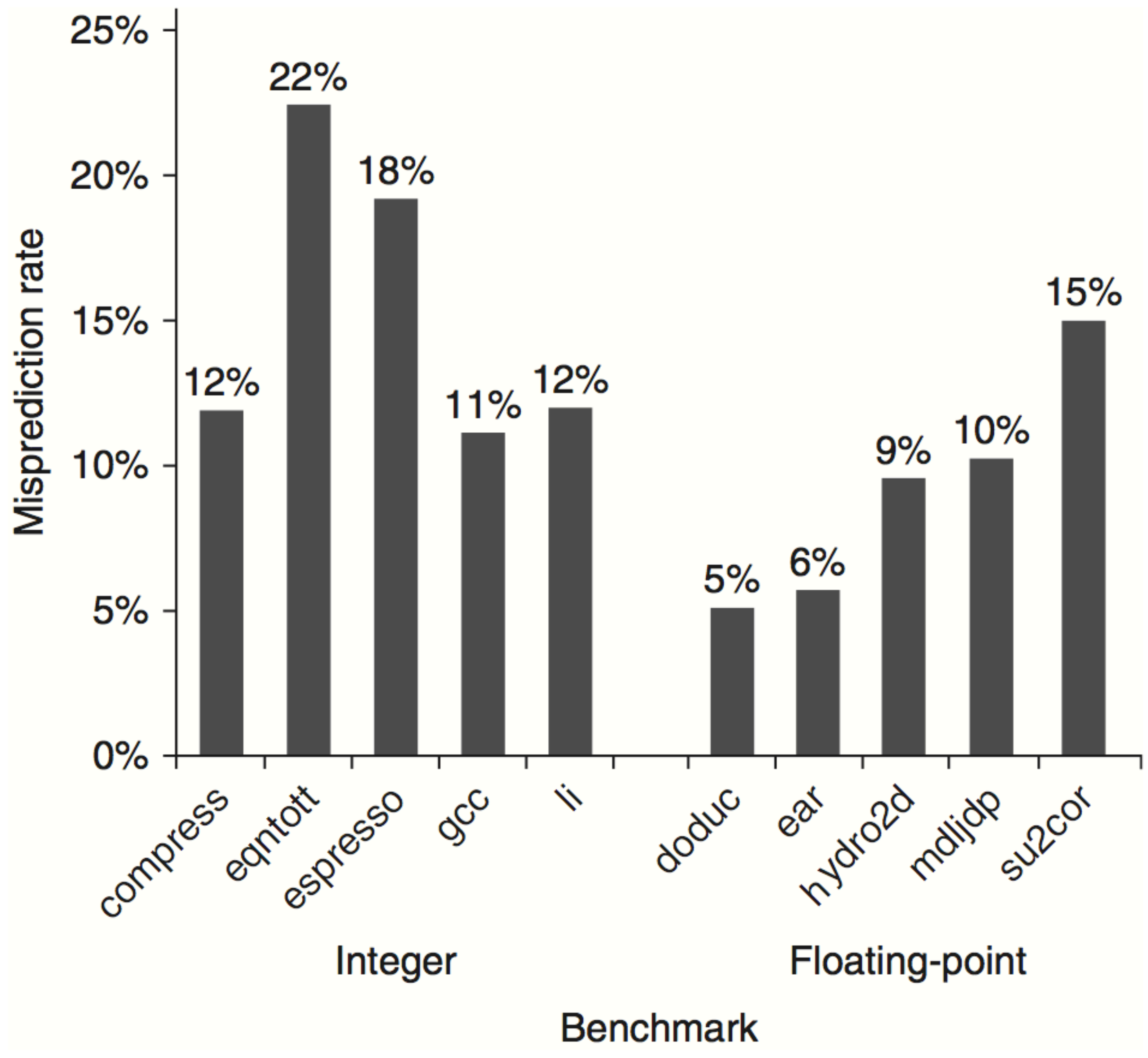


Figure C.17 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an aver-

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

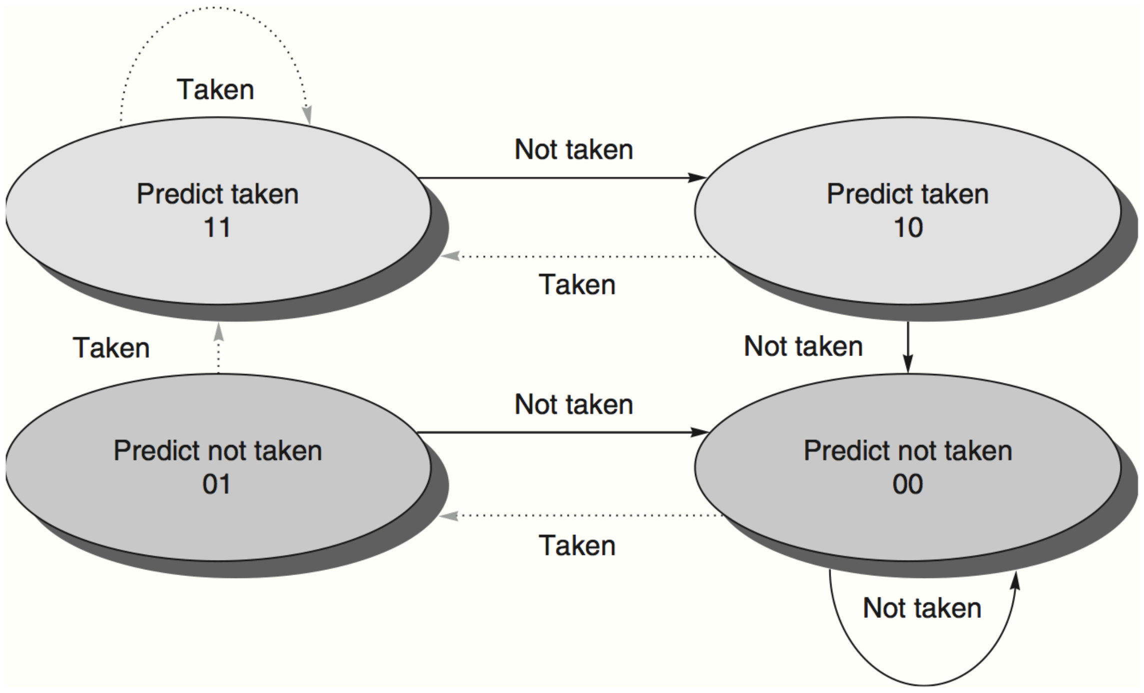


Figure C.18 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually 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 counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is pre-

Contents

Reading:

- **Textbook: Appendix C**
- **RISC-V Sodor core**
  - **Chisel: https://github.com/freechipsproject/chisel3/wiki/Short-Users-Guide-to-Chisel**
  - **https://github.com/ucb-bar/riscv-sodor**