



香港中文大學

計算機科學與工程學系

CSC 3420

Computer Systems Architecture

Chapter 6 : Enhancing Performance with Pipelining

Pipelining is Natural!

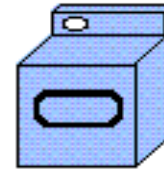
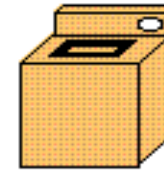
° Laundry Example

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

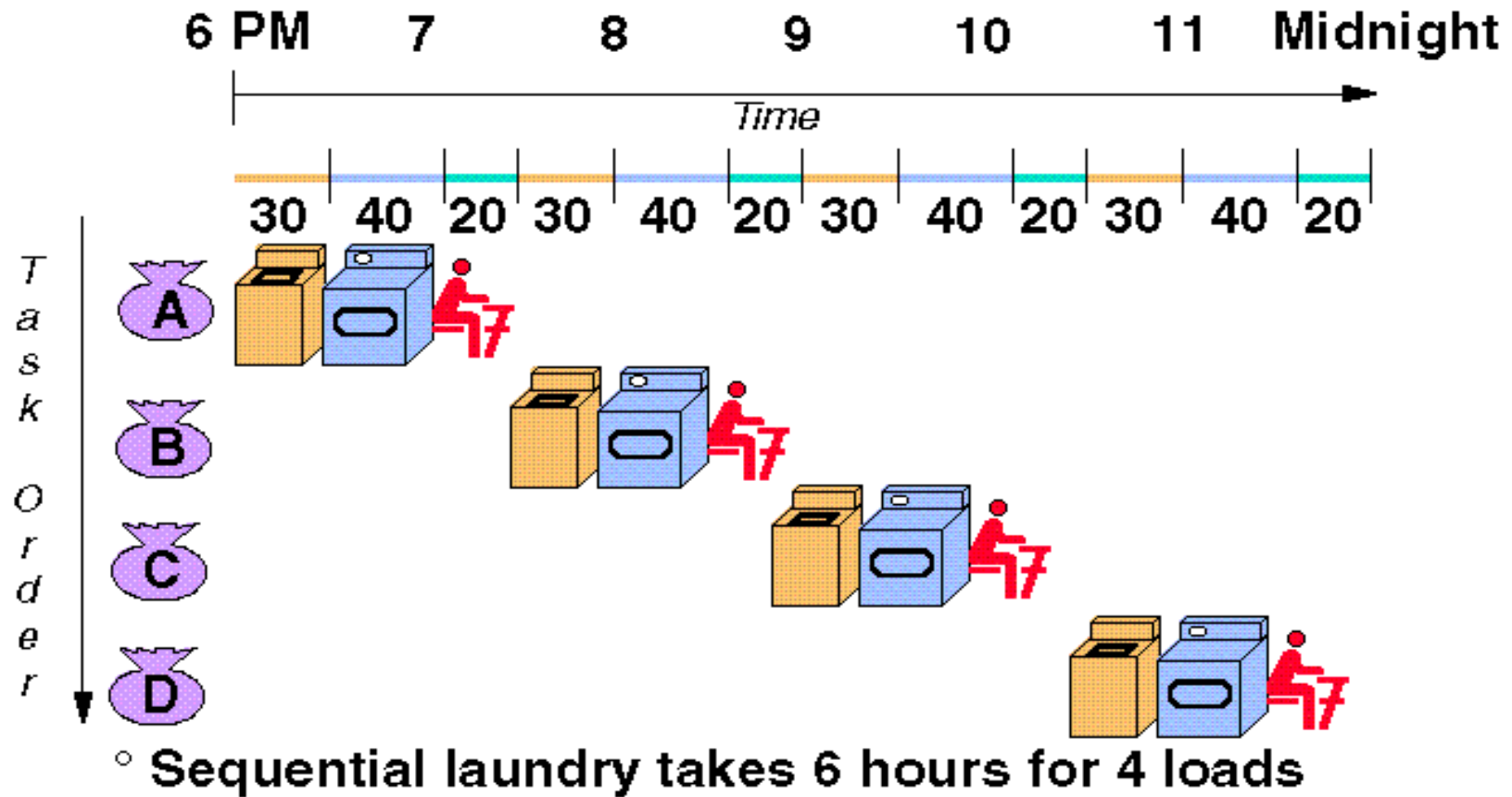
° Washer takes 30 minutes

° Dryer takes 40 minutes

° Iron/Folder takes 20 minutes

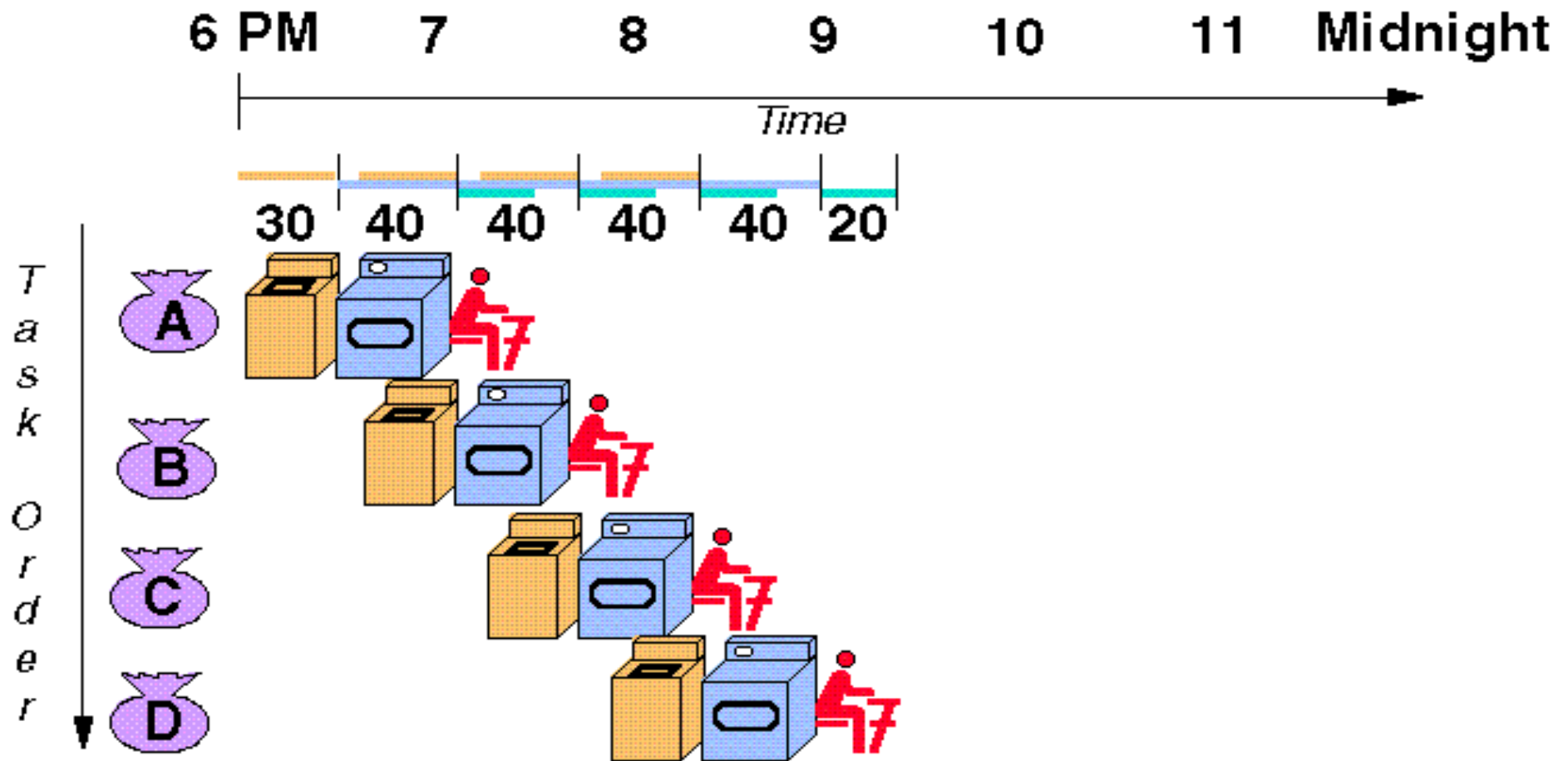


Sequential Laundry



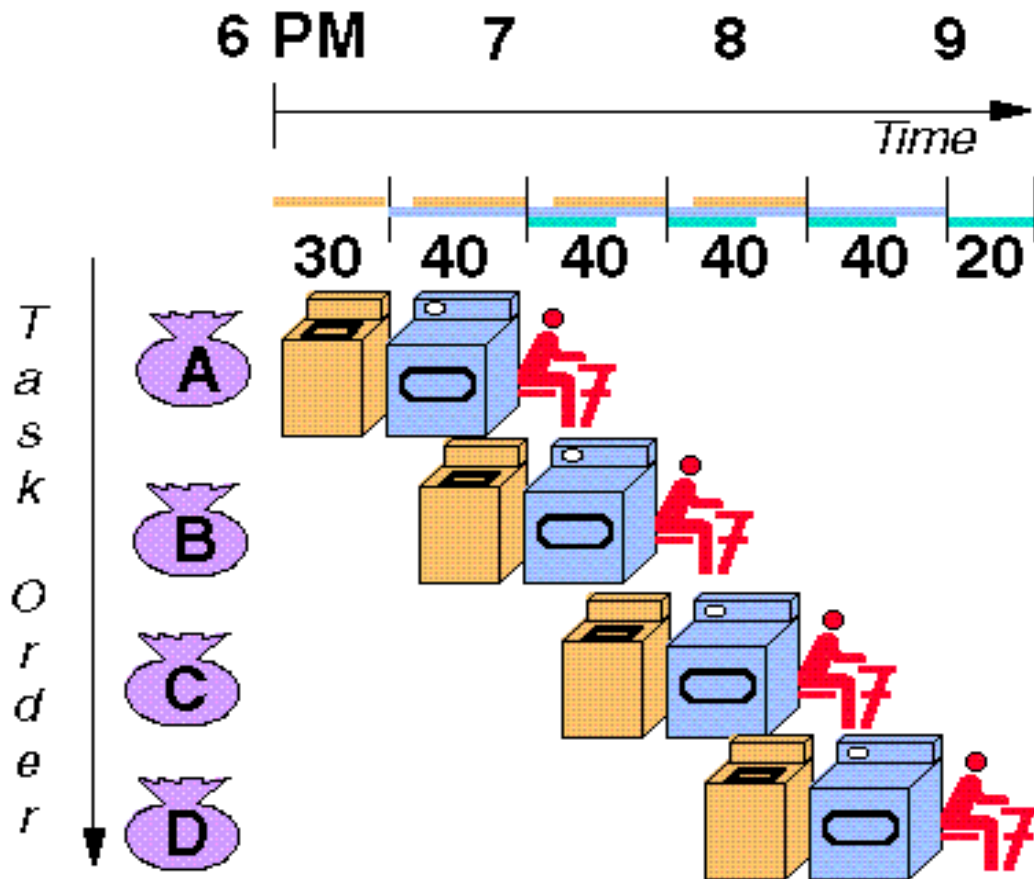
If they learned pipelining, how long would they take?

Pipelined Laundry: Start work ASAP



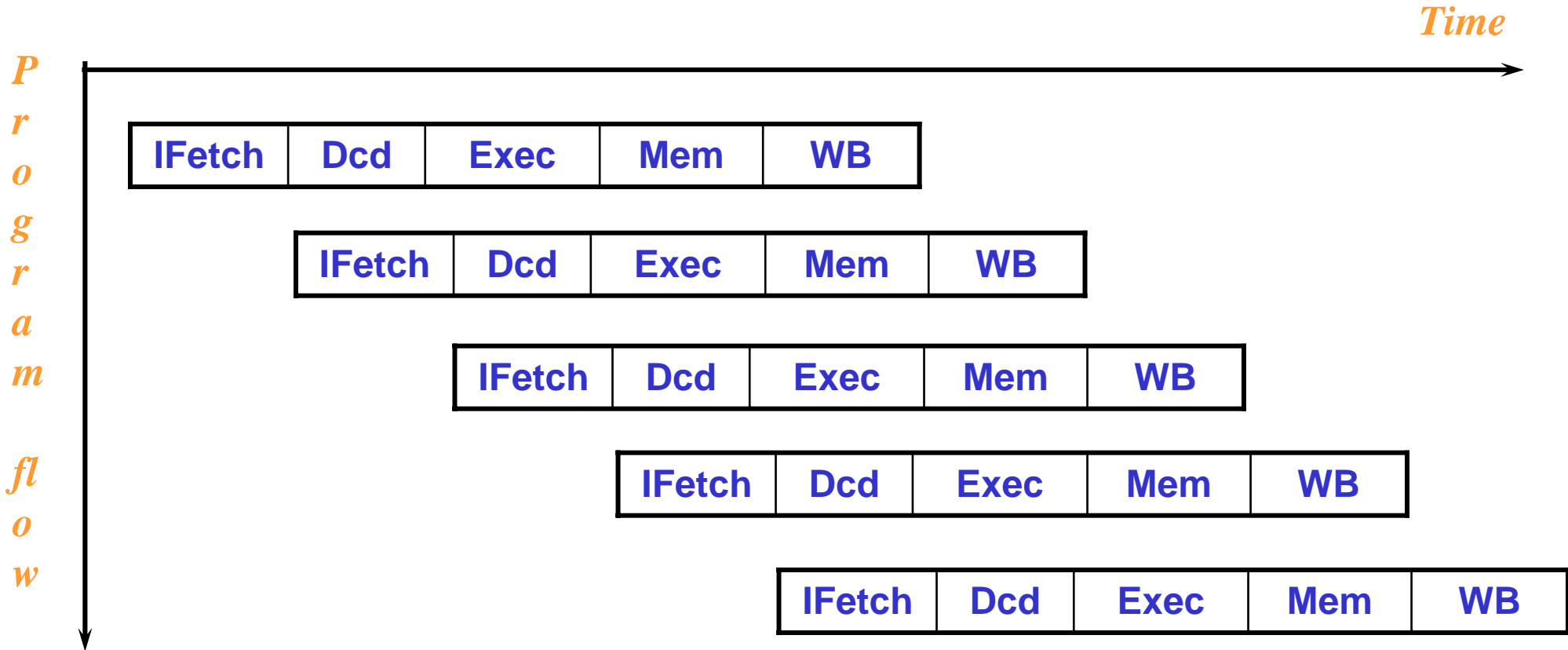
Pipelined laundry takes 3.5 hours for 4 loads

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* using different resources
- Potential speedup = *Number of pipe stages*
- Unbalanced lengths of pipe stages *reduces* speedup
- Time to “*fill*” pipeline and time to “*drain*” it reduces speedup
- **Stall for Dependencies**

Pipelined Execution: Start an Instruction Every Clock Cycle



Why Pipeline?

It is faster

Suppose we execute 100 instructions

◦ *Single Cycle Machine*

$$45 \text{ ns/cycle} \times 100 = 4500 \text{ ns}$$

◦ *Multicycle Machine*

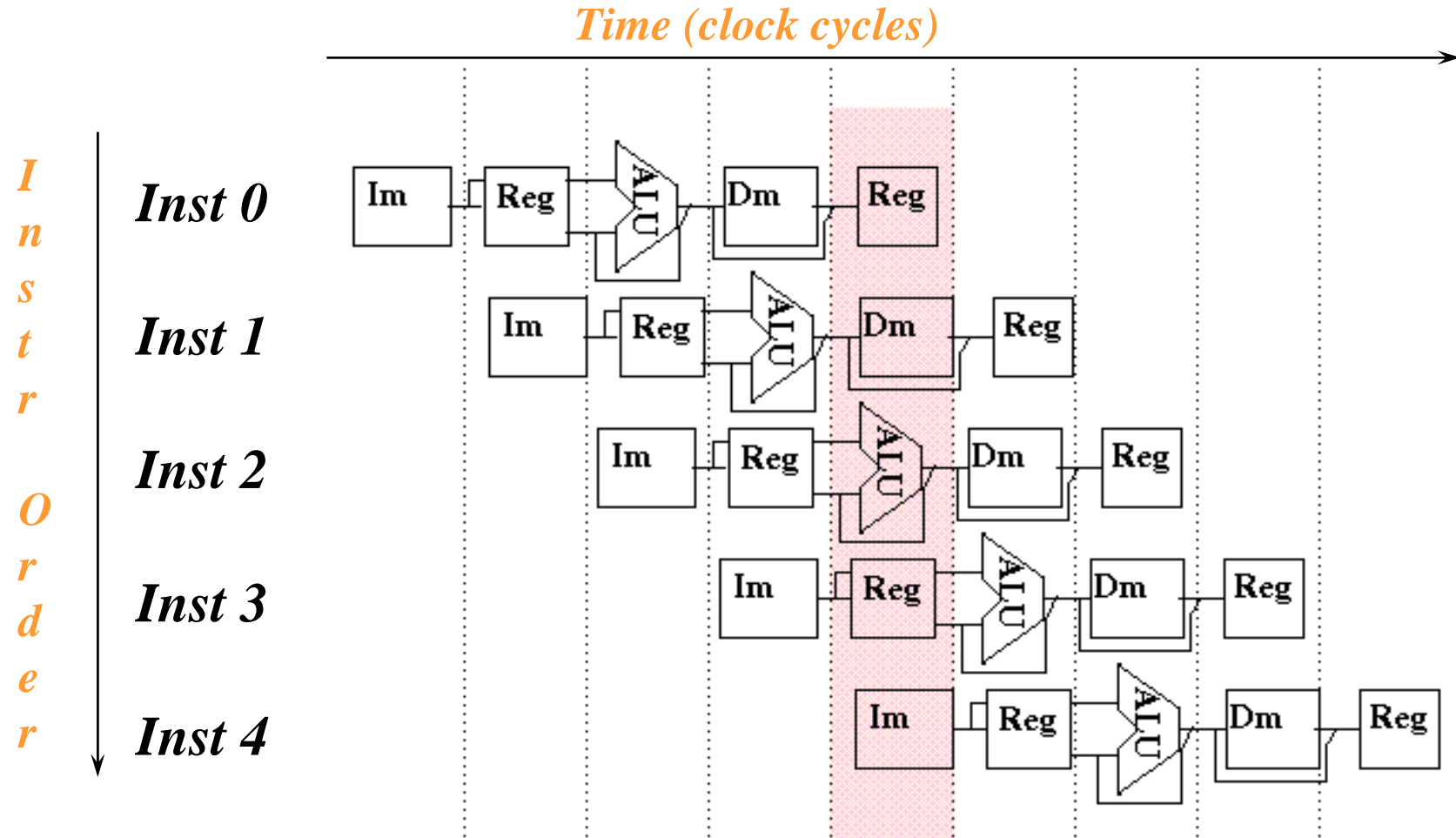
$$10 \text{ ns/cycle} \times 100 = 1000 \text{ ns}$$

◦ *Ideal pipelined machine*

$$10 \text{ ns/cycle} \times 100 = 1000 \text{ ns}$$

Is Pipelines Possible?

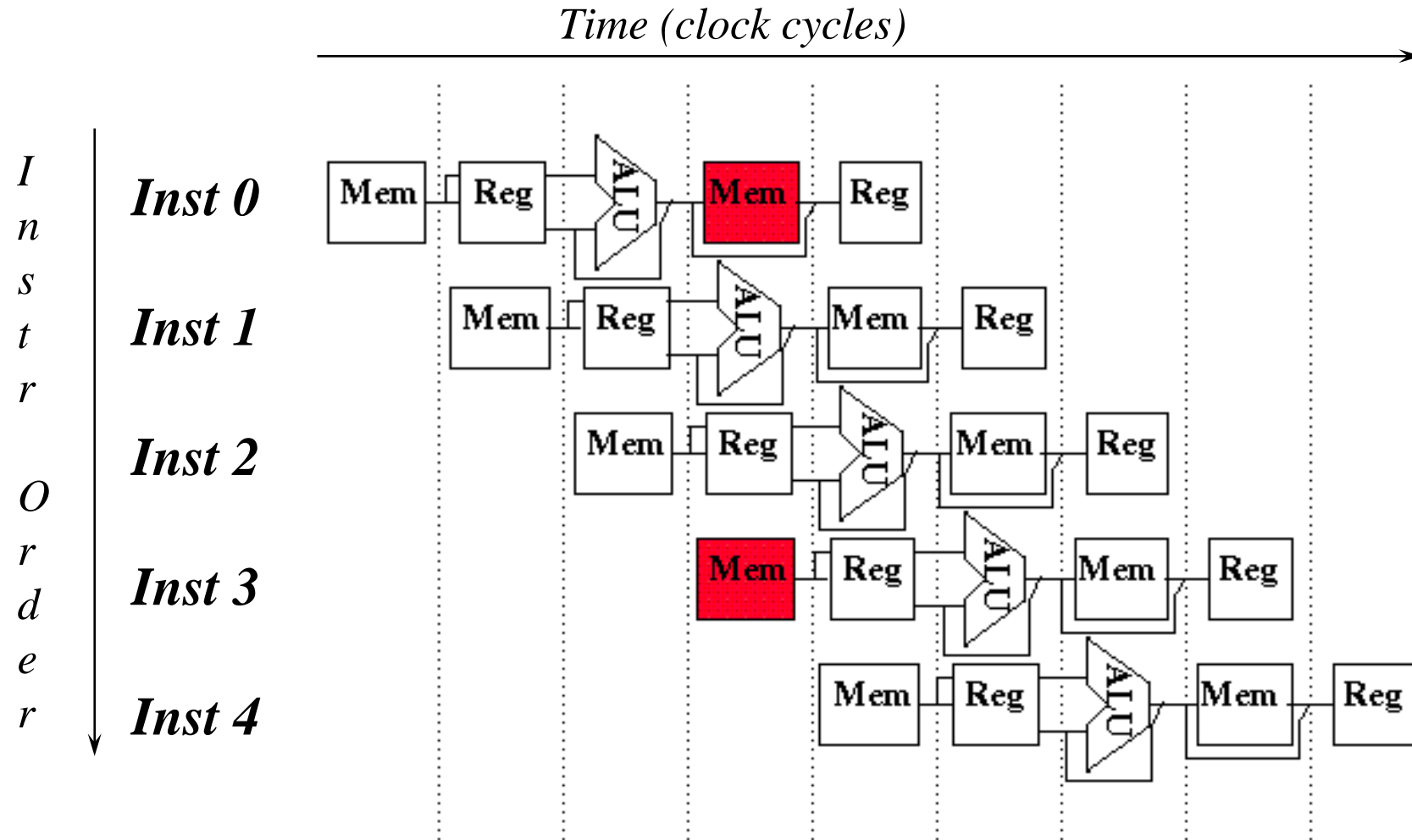
Yes. The resources are there



Pitfalls

- What makes pipelining easy?
 - all *instructions* are the *same length*
 - just a *few instruction formats*
 - memory operands appear only in *loads and stores*
- What makes pipelining hard?
 - *structural hazards*: suppose we had only one memory (see next page)
 - *control hazards*: need to worry about branch instructions
 - *data hazards*: an instruction depends on a previous instruction
- Can always resolve hazards by waiting
 - pipeline control must detect hazard
 - take action (or delay action) to resolve hazards
- We'll build a simple pipeline and look at these issues
- We'll talk about modern processors and what really makes it hard:
 - exception handling
 - trying to improve performance with out-of-order execution, etc.

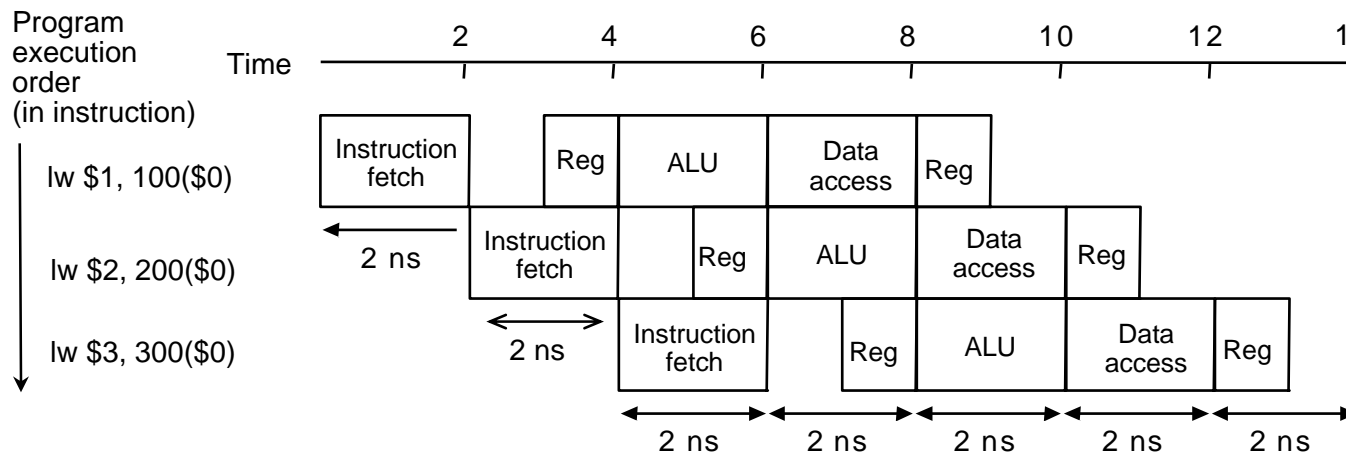
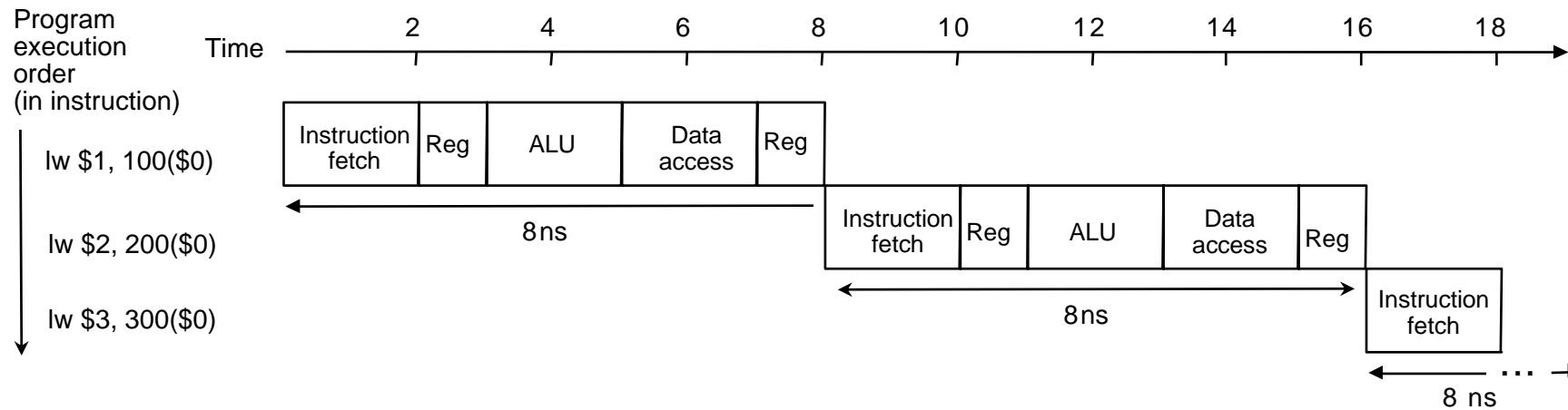
Example of a Simple Structural Hazard



- Instructions and data are fetched from the same memory.
- In this case, detection is easy

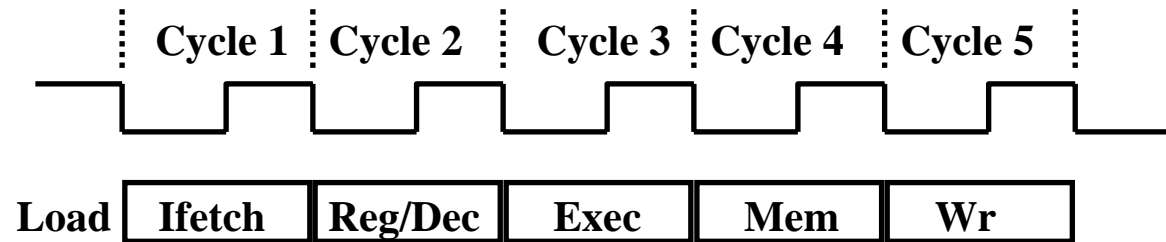
Pipelining in MIPS

- Improve performance by increasing instruction throughput



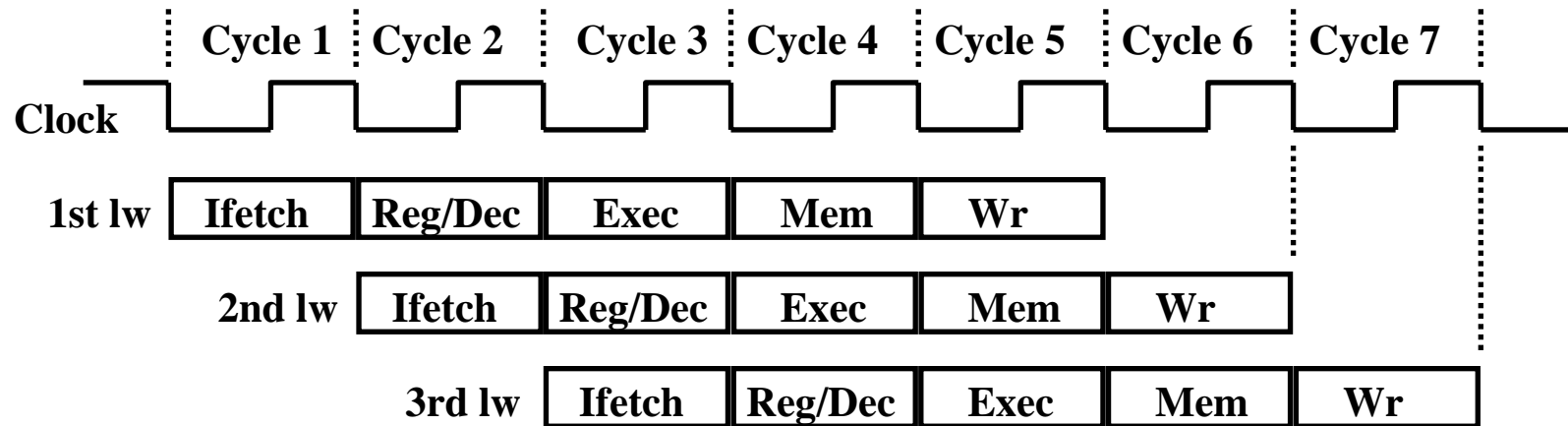
Ideal speedup is number of stages in the pipeline. Do we achieve this?

The Five Stages of Load instruction (lw)



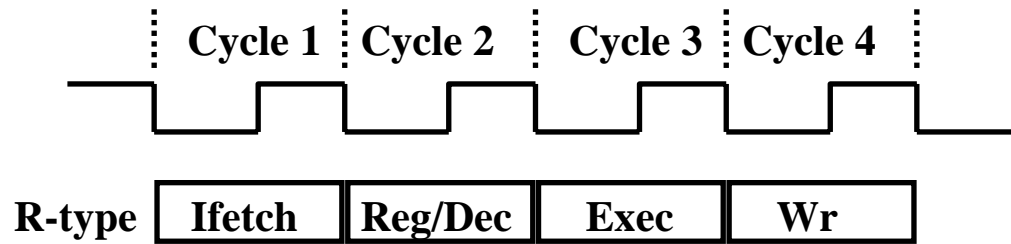
- The load instruction has 5 stages:
 - **Ifetch**: Instruction Fetch : Fetch the instruction from the Instruction Memory
 - **Reg/Dec**: Registers Fetch and Instruction Decode
 - **Exec**: Calculate the memory address
 - **Mem**: Read the data from the Data Memory
 - **Wr**: Write the data back to the register file
- Five independent functional units to work on each stage
 - Each functional unit is used only once
- The 2nd load can start as soon as the 1st finishes its Ifetch stage
- Each load instruction still takes five cycles to complete
- The throughput, however, is much higher

Pipelining the Load Instruction



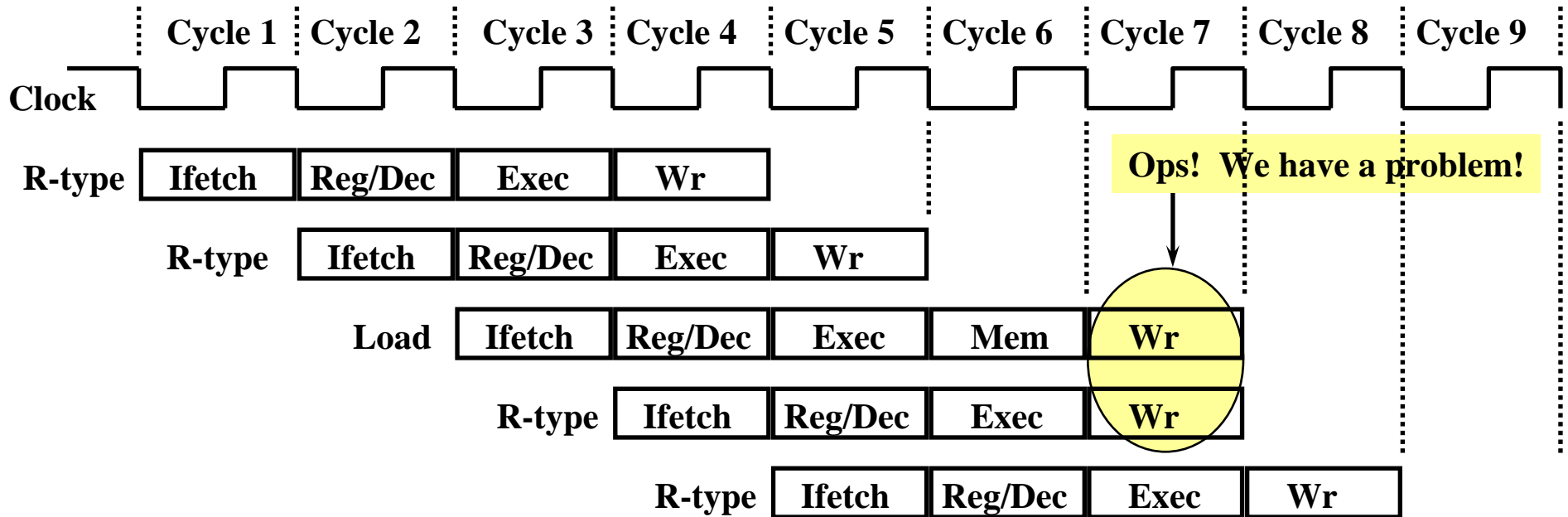
- **The five independent functional units in the pipeline datapath are:**
 - **Instruction Memory for the Ifetch stage**
 - **Register File's Read ports (bus A and busB) for the Reg/Dec stage**
 - **ALU for the Exec stage**
 - **Data Memory for the Mem stage**
 - **Register File's Write port (bus W) for the Wr stage**
- **One instruction enters the pipeline every cycle**
 - **One instruction comes out of the pipeline (complete) every cycle**
 - **The “Effective” Cycles per Instruction (CPI) is 1**

The Four Stages of R-type



- **Ifetch:** Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec:** Registers Fetch and Instruction Decode
- **Exec:** ALU operates on the two register operands
- **Wr:** Write the ALU output back to the register file

Pipelining the R-type and Load Instruction

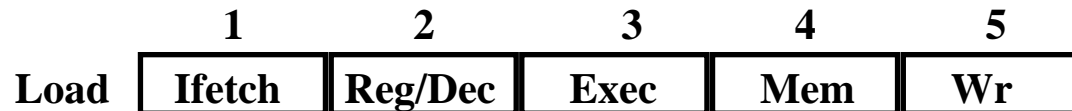


◦ We have a problem:

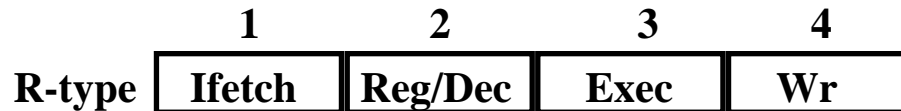
- Two instructions try to write to the register file at the same time!

Important Observation

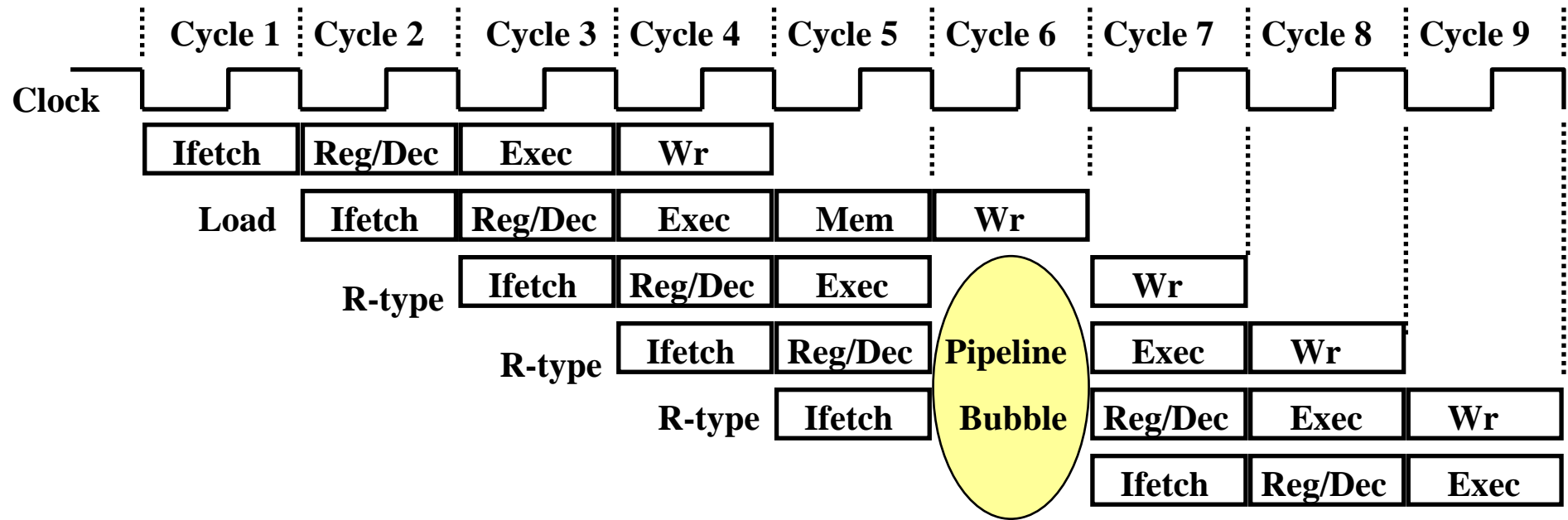
- Each functional unit can only be **used once per instruction**
- Each functional unit must be **used at the same stage for all instructions:**
 - **Load uses Register File's Write Port during its 5th stage**



- **R-type uses Register File's Write Port during its 4th stage**



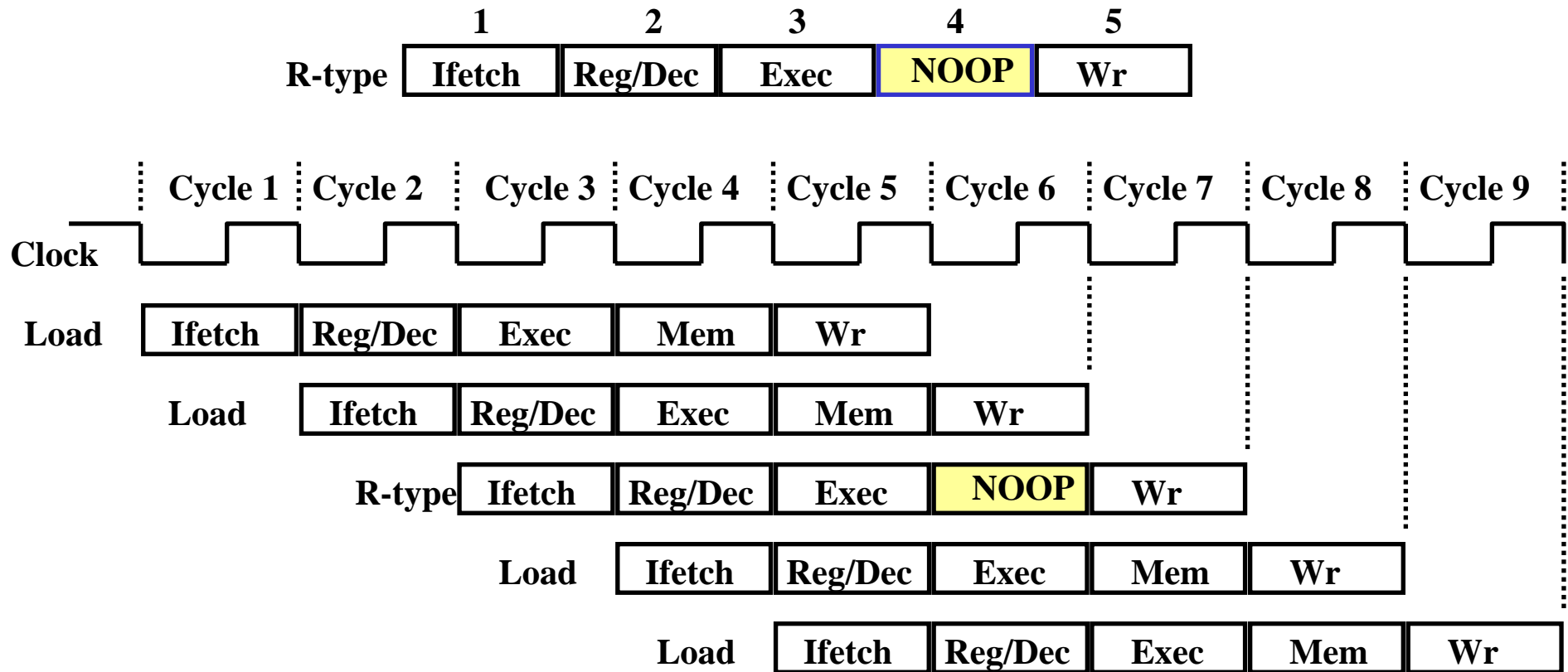
Solution 1: Insert “Bubble” into the Pipeline



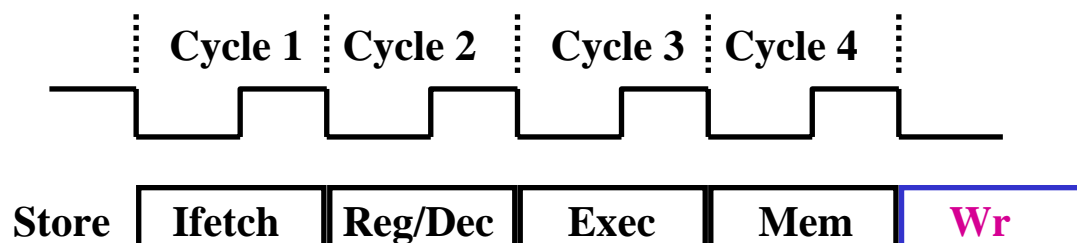
- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
 - The control logic can be complex
- No instruction is completed during Cycle 5:
 - The “Effective” CPI for load is 2

Solution 2: Delay R-type's Write by One Cycle

- **Delay R-type's register write by one cycle:**
 - Now R-type instructions also use Reg File's write port at Stage 5
 - **NOOP stage:** nothing is being done

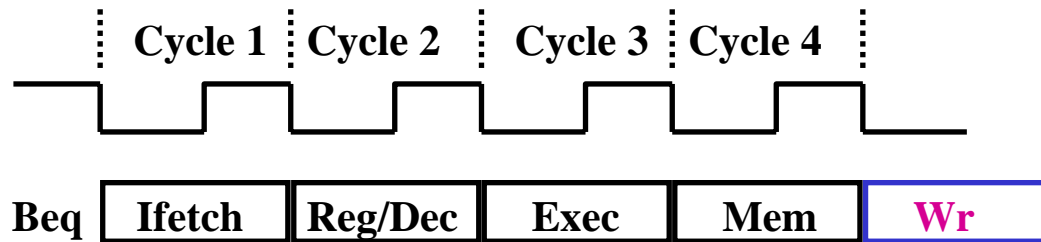


The Four Stages of Store



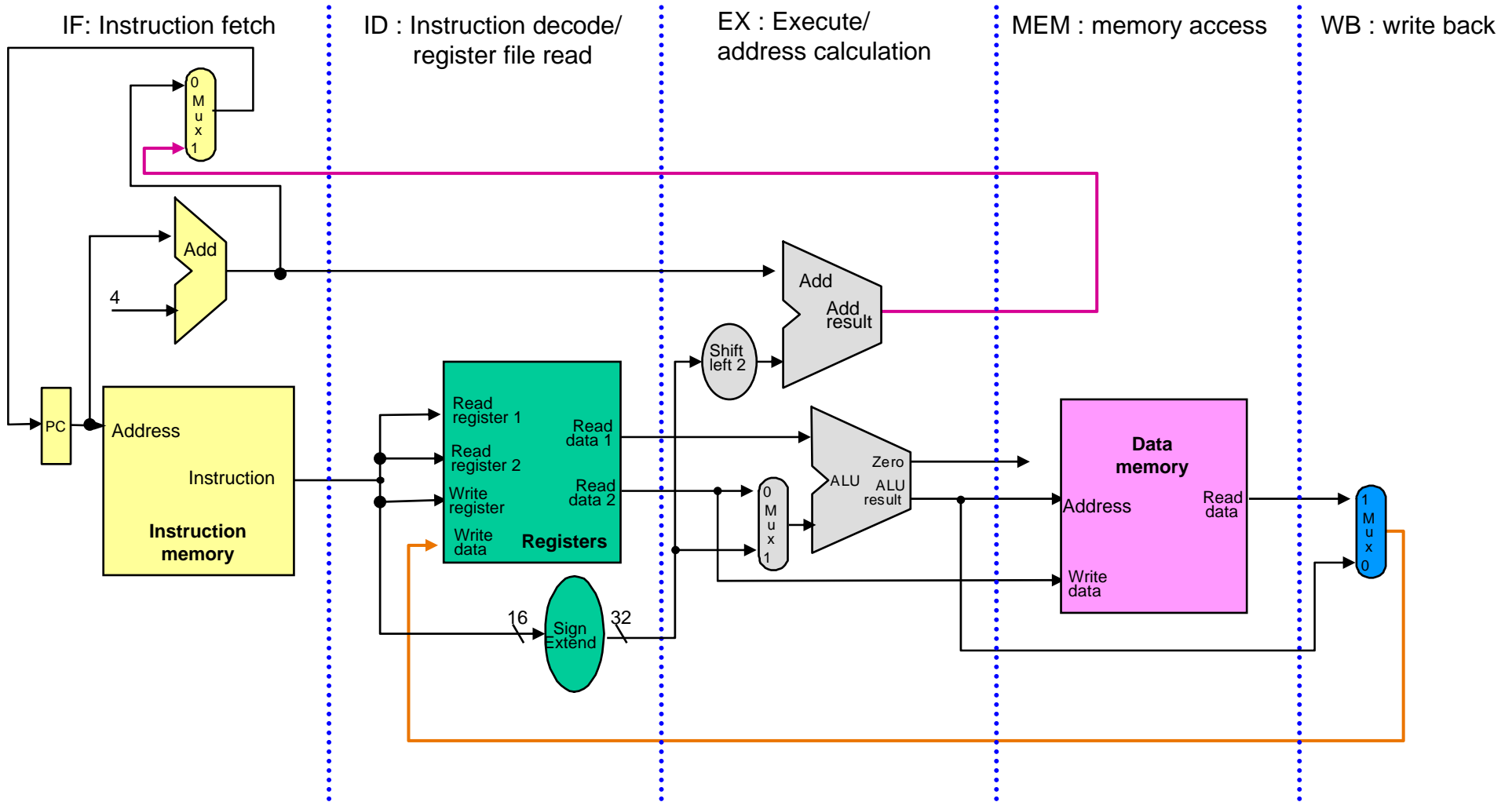
- **Ifetch:** Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec:** Registers Fetch and Instruction Decode
- **Exec:** Calculate the memory address
- **Mem:** Write the data into the Data Memory
- **Wr:** Dummy stage, not really required

The Four Stages of Beq



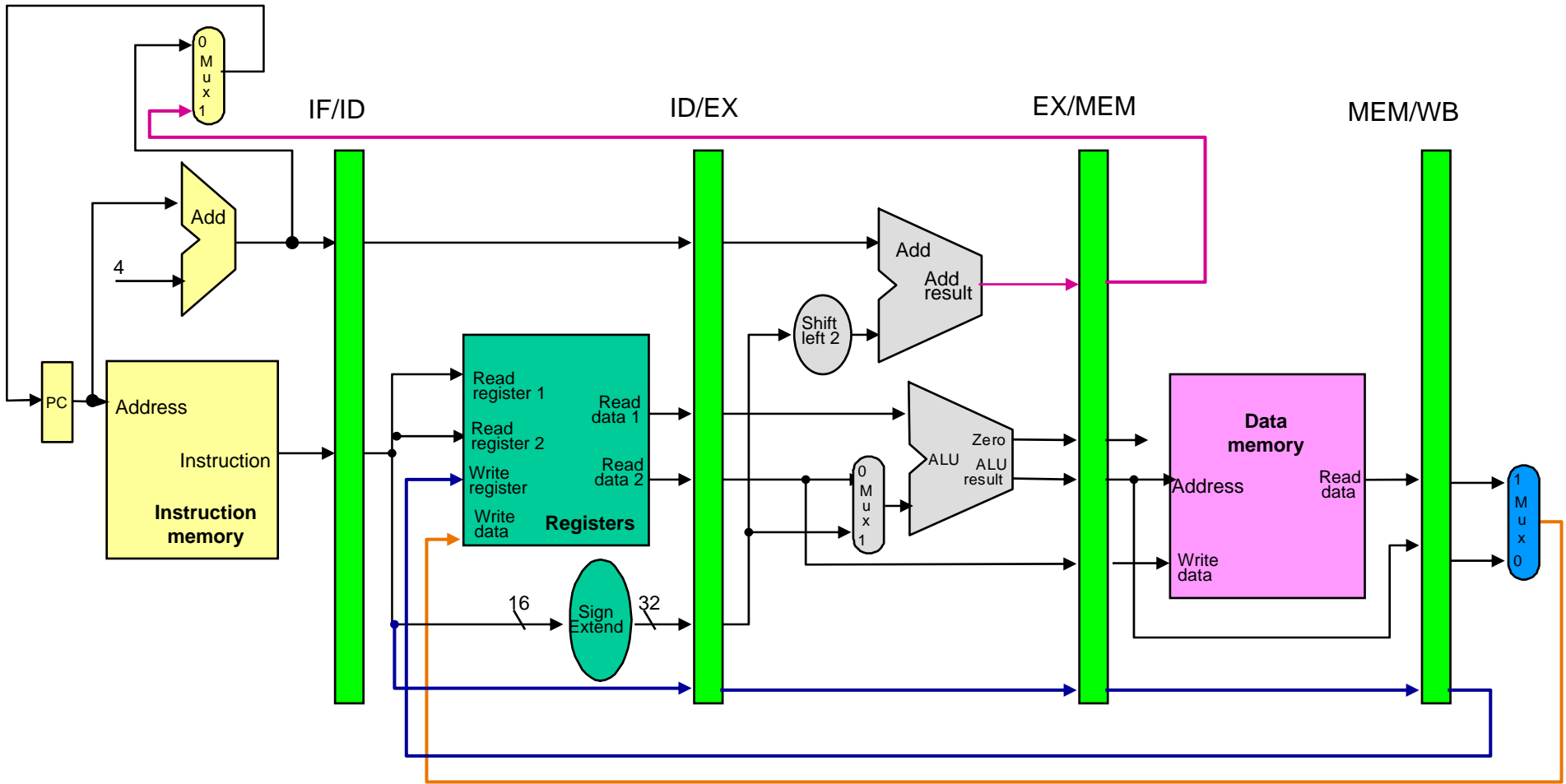
- **Ifetch:** Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec:** Registers Fetch and Instruction Decode
- **Exec:** ALU compares the two register operands
 - Adder calculates the branch target address
- **Mem:** If the registers we compared in the Exec stage are the same,
 - Write the branch target address into the PC
- **Wr:** Dummy stage, not really required

Incorporating the pipelining idea into our MIPS CPU



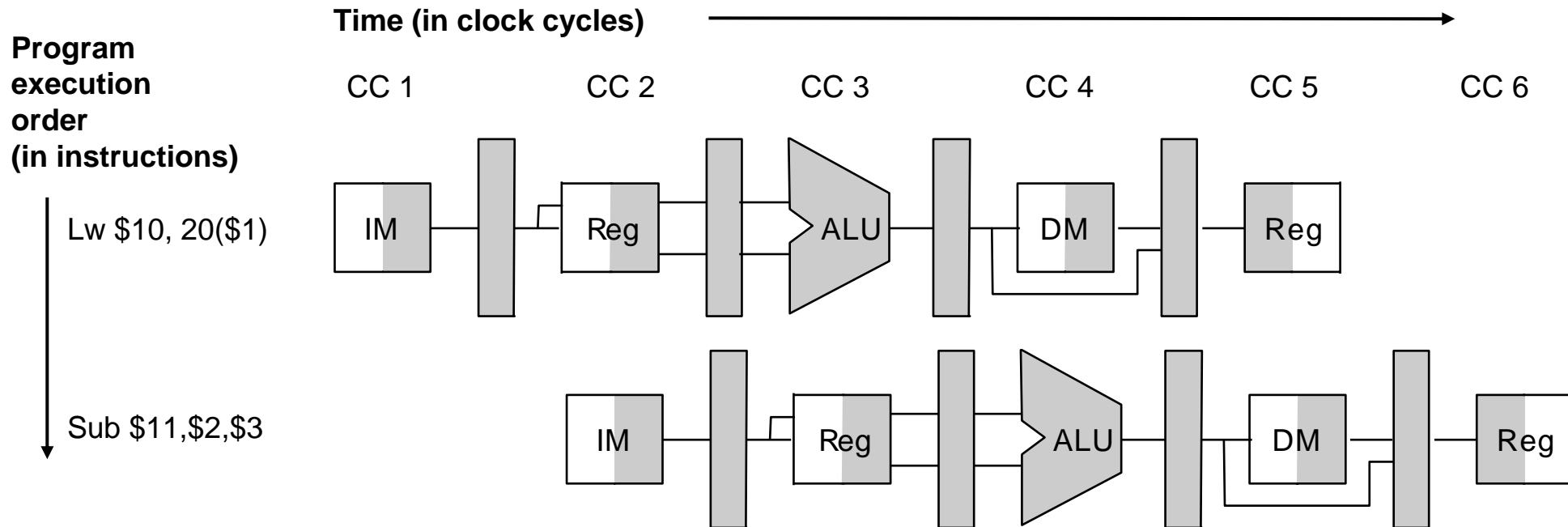
What do we need to add to actually split the datapath into stages?

Corrected Datapath



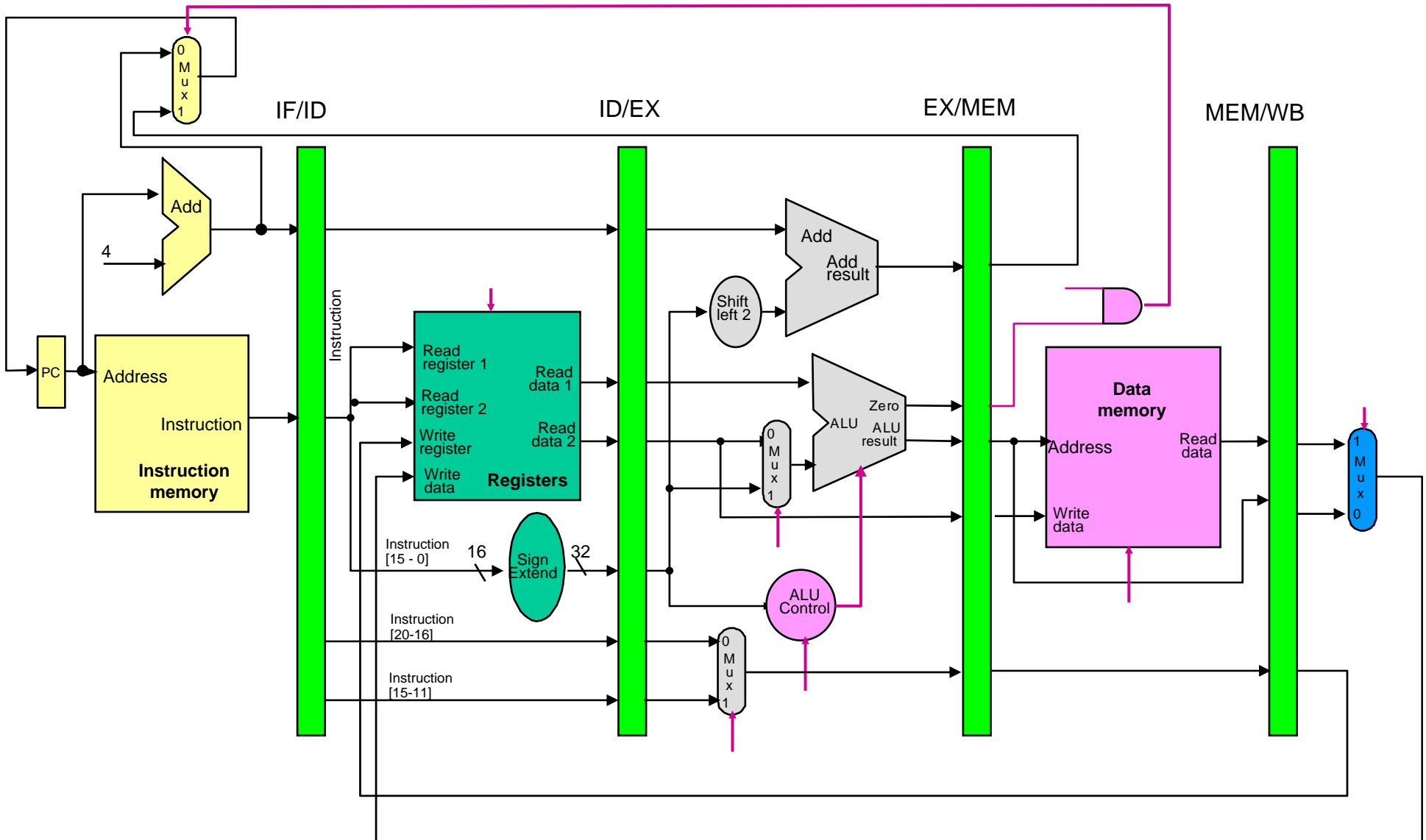
- *Add buffers for register address*

Graphically Representing Pipelines



- **Can help with answering questions like:**
 - how many cycles does it take to execute this code?
 - what is the ALU doing during cycle 4?
 - use this representation to help understand datapaths

Pipeline Control



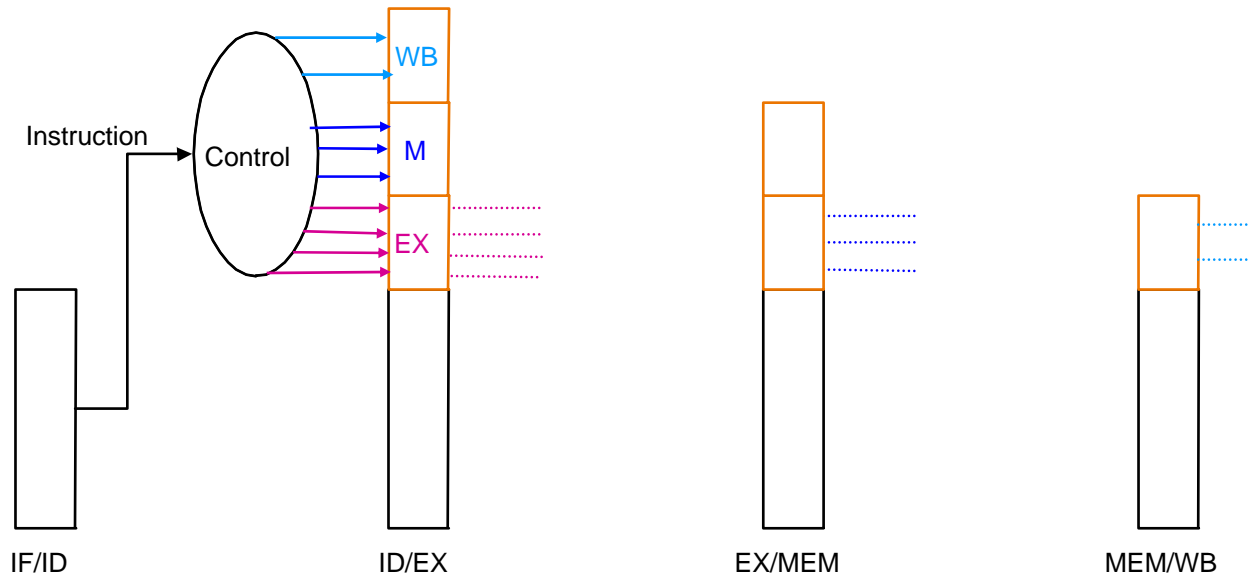
Pipeline control

- **We have 5 stages. What needs to be controlled in each stage?**
 - **Instruction Fetch and PC Increment**
 - **Instruction Decode / Register Fetch**
 - **Execution**
 - **Memory Stage**
 - **Write Back**
- **How would control be handled in an automobile plant?**
 - **a fancy control center telling everyone what to do?**
 - **should we use a finite state machine?**
- **Key Observation: Control Signals at Stage N = Func (Instr. at Stage N)**
 - **N = Exec, Mem, or Wr**
 - **Example: Controls Signals at Exec Stage = Func(Load's Exec)**
- **The Main Control generates the control signals during Reg/Dec**
 - **Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later**
 - **Control signals for Mem (MemWr Branch) are used 2 cycles later**
 - **Control signals for Wr (MemtoReg MemWr) are used 3 cycles later**

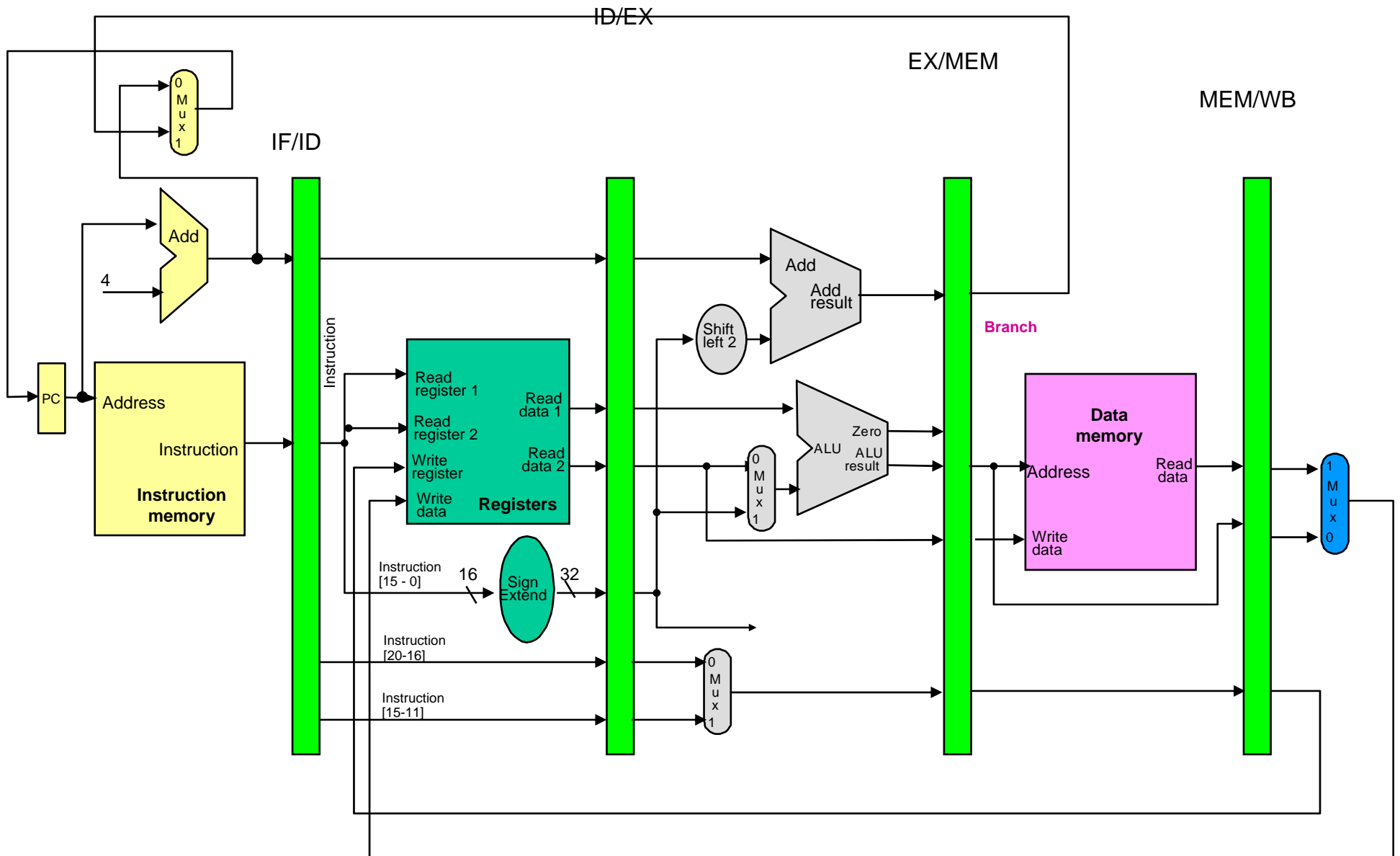
Pipeline Control

- Pass control signals along just like the data

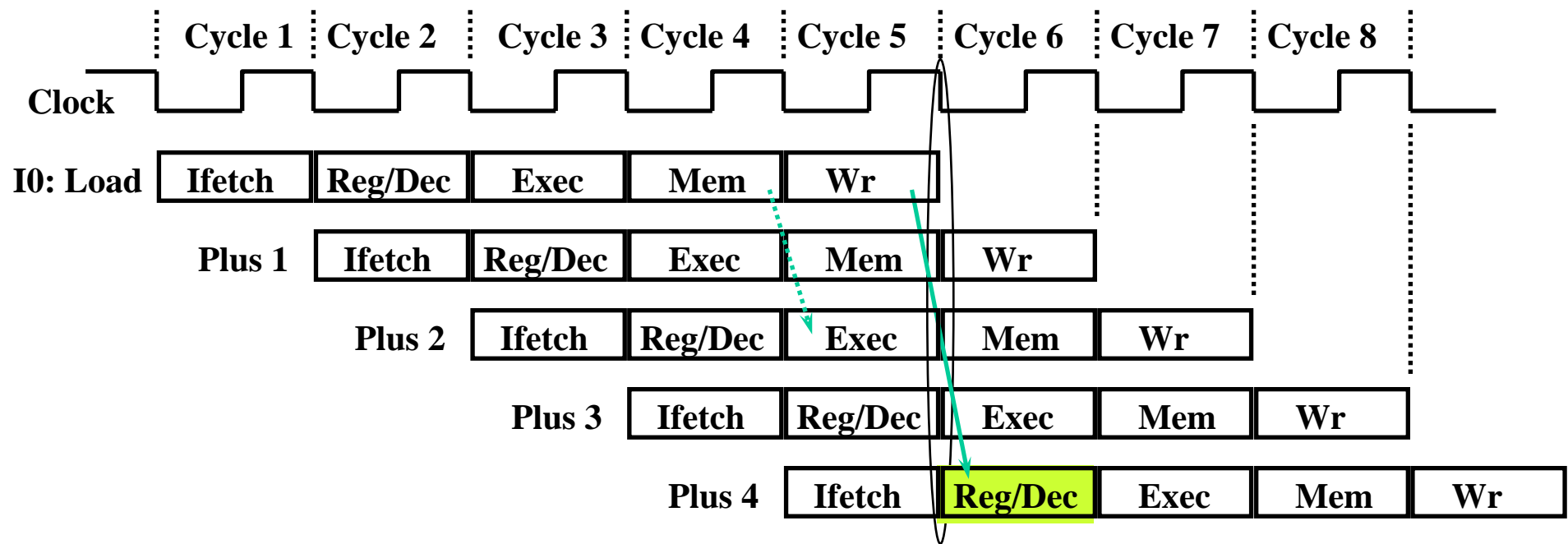
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format									
lw									
sw									
beq									



Datapath with Control



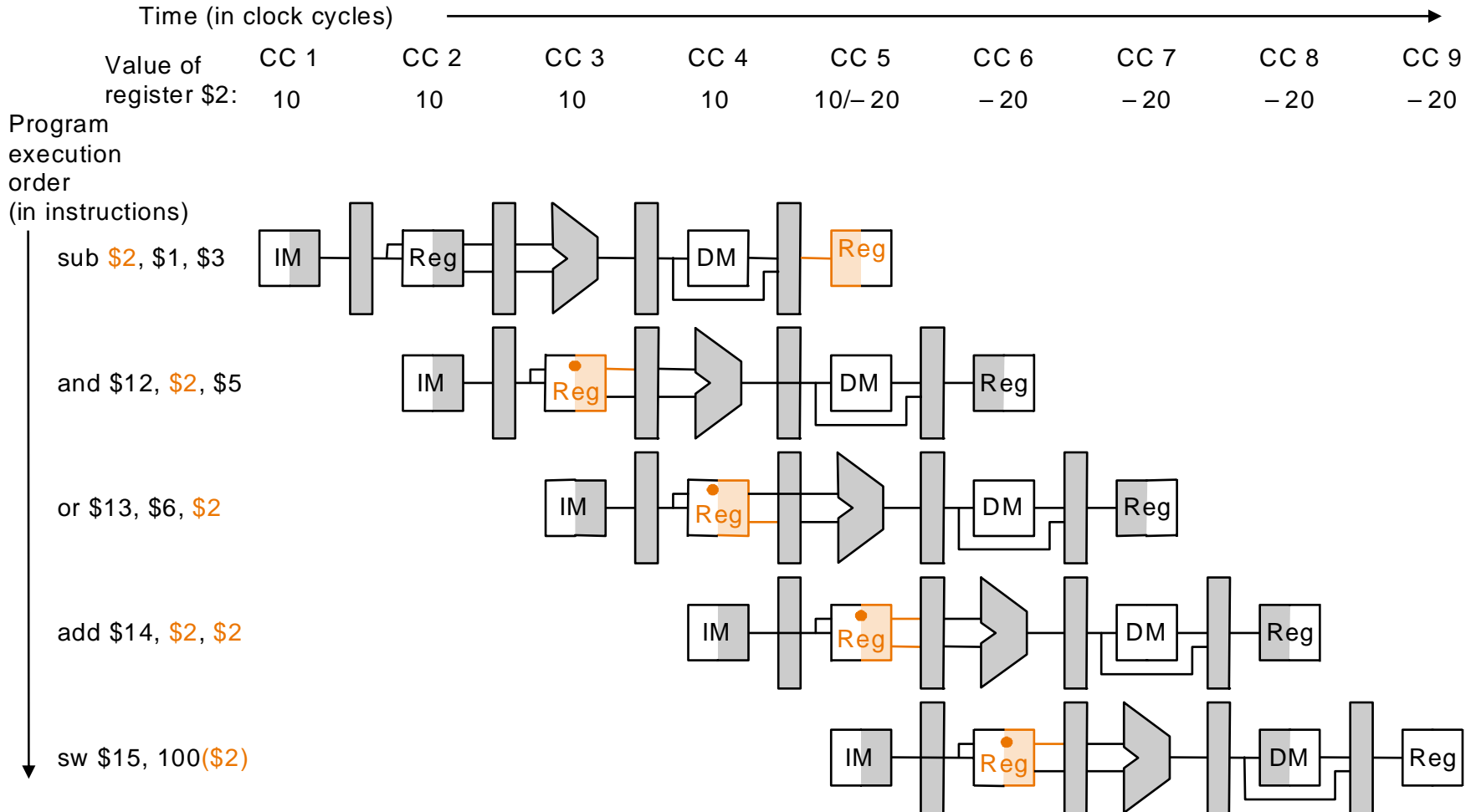
The Delay Load Phenomenon



- Although Load is fetched during Cycle 1:
 - The data is NOT written into the Reg File until the end of Cycle 5
 - We cannot read this value from the Reg File until Cycle 6
 - 3-instruction delay before the load take effect
- Clever design techniques can reduce the delay to ONE instruction

Dependencies

- Problem with starting next instruction before first is finished
- **Data hazards** : dependencies that “go backward in time”
 - In our example: \$1=0, \$3= 20



Software Solution

- Use compiler to guarantee no hazards
- Where do we insert the “nops” ?

```
sub    $2, $1, $3
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)
```

```
sub    $2, $1, $3
```

```
sw     $15, 100($2)
```

- **Problem:** this really slows us down!

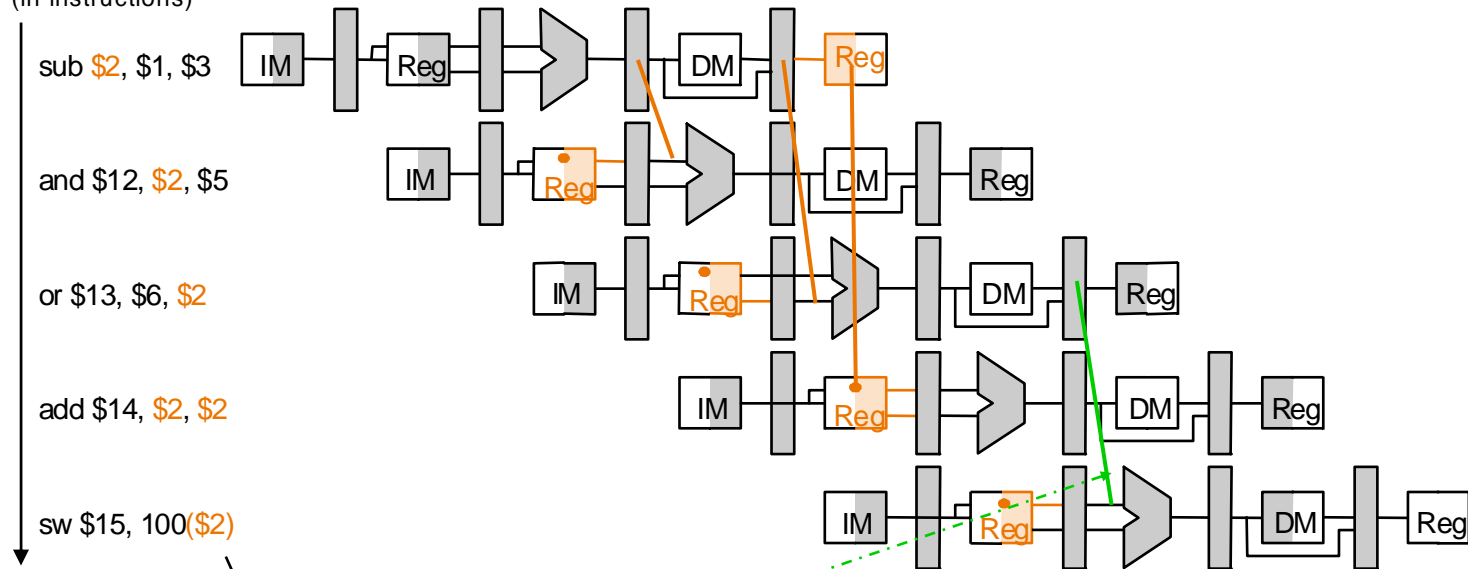
Forwarding

- Use temporary results, don't wait for them to be written
 - register file forwarding to handle read/write to same register
 - ALU forwarding

Time (in clock cycles) →

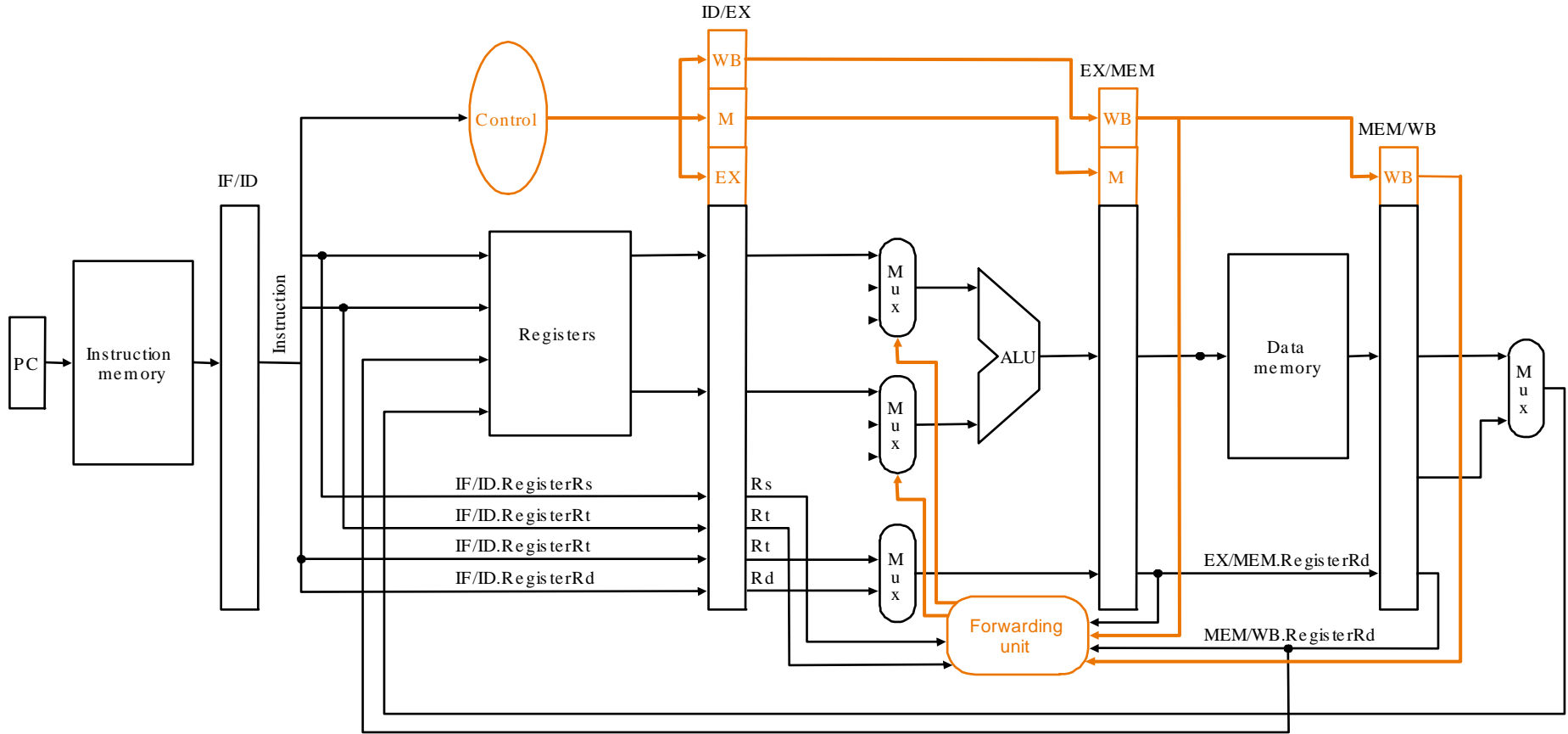
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register \$2 :	10	10	10	10	10/-20	-20	-20	-20	-20
Value of EX/MEM :	X	X	X	-20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	-20	X	X	X	X

Program execution order (in instructions)



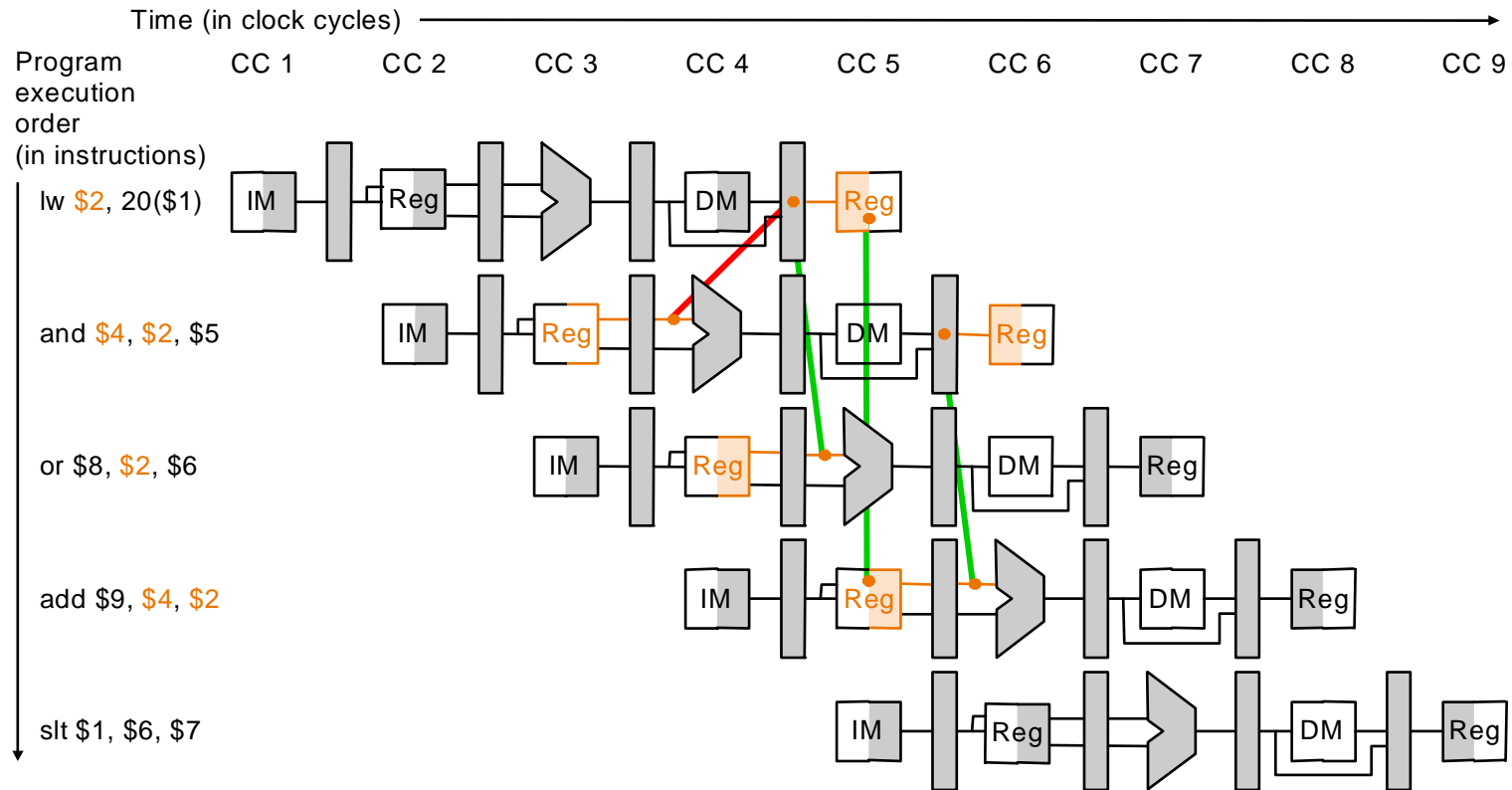
what if this \$2 was \$13?

Forwarding



Can't always forward

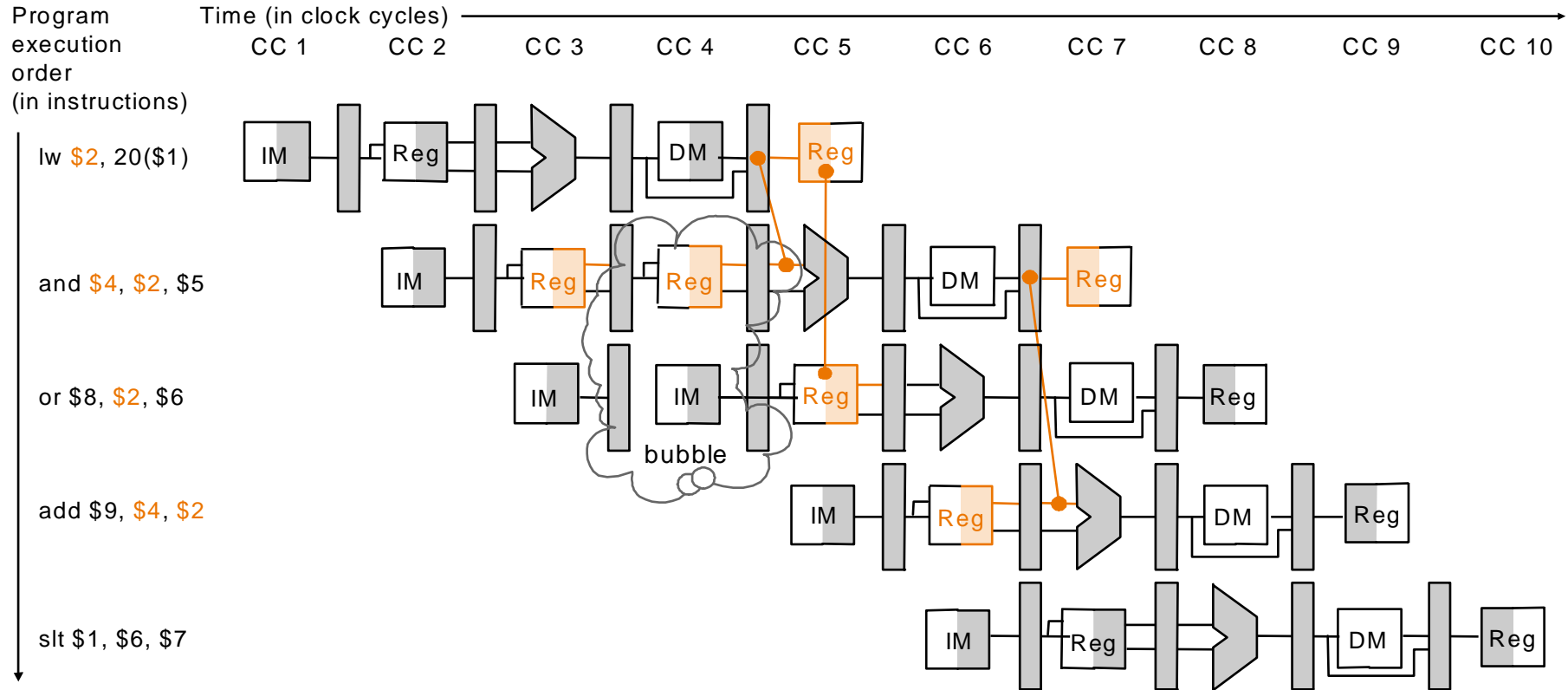
- **Load word can still cause a hazard:**
 - **an instruction tries to read a register following a load instruction that writes to the same register.**



- **Thus, we need a hazard detection unit to “stall” after the load instruction**

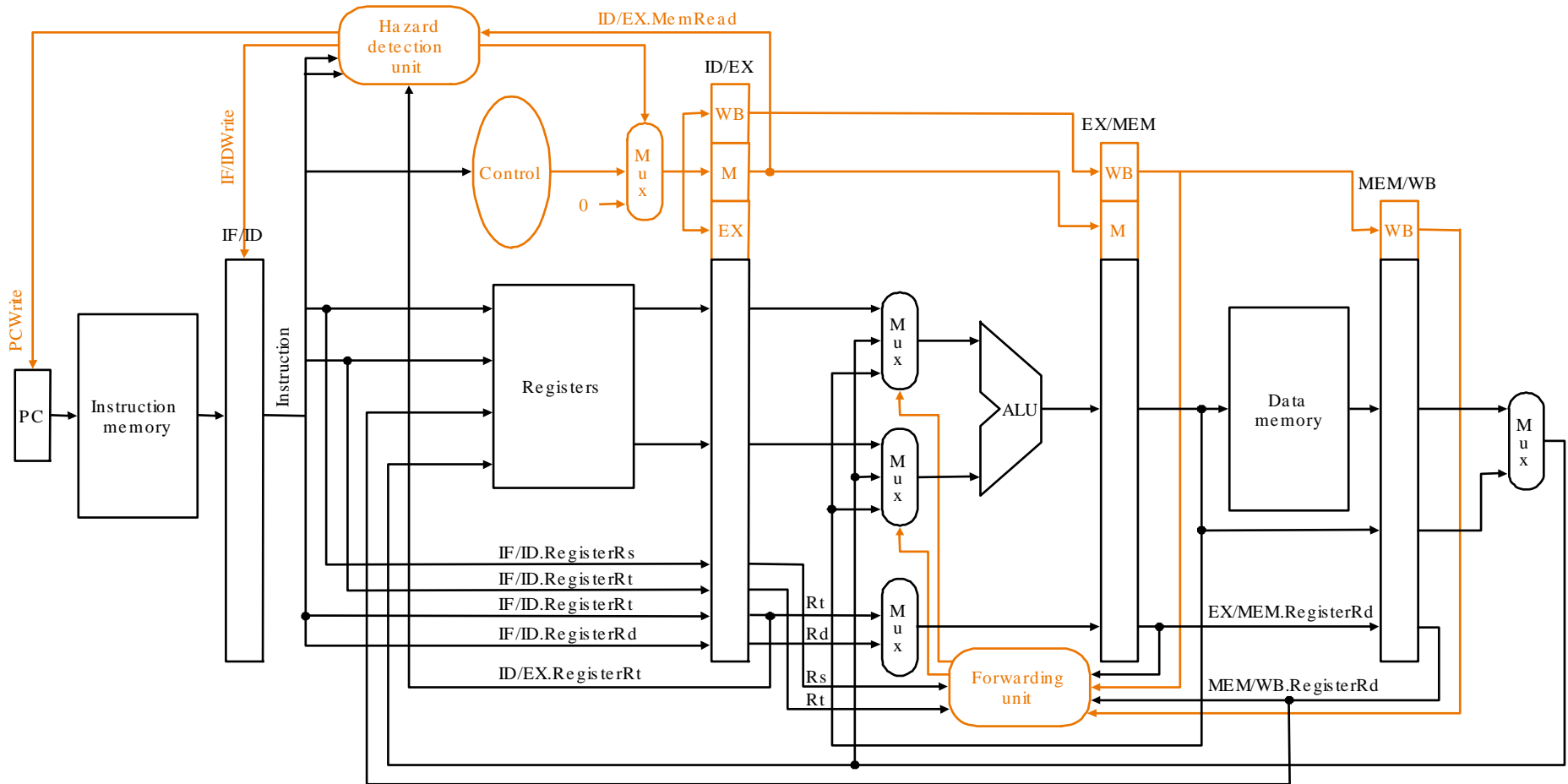
Stalling

- We can stall the pipeline by keeping an instruction in the same stage

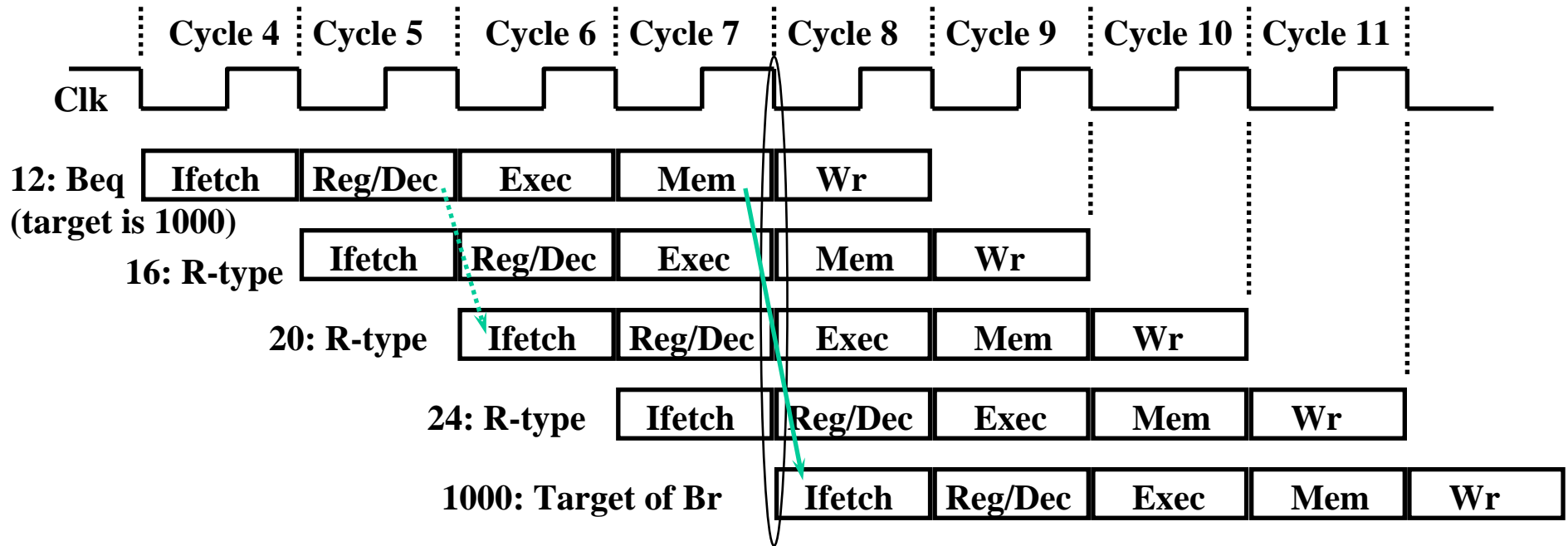


Hazard Detection Unit

- Stall by letting an instruction that won't write anything go forward



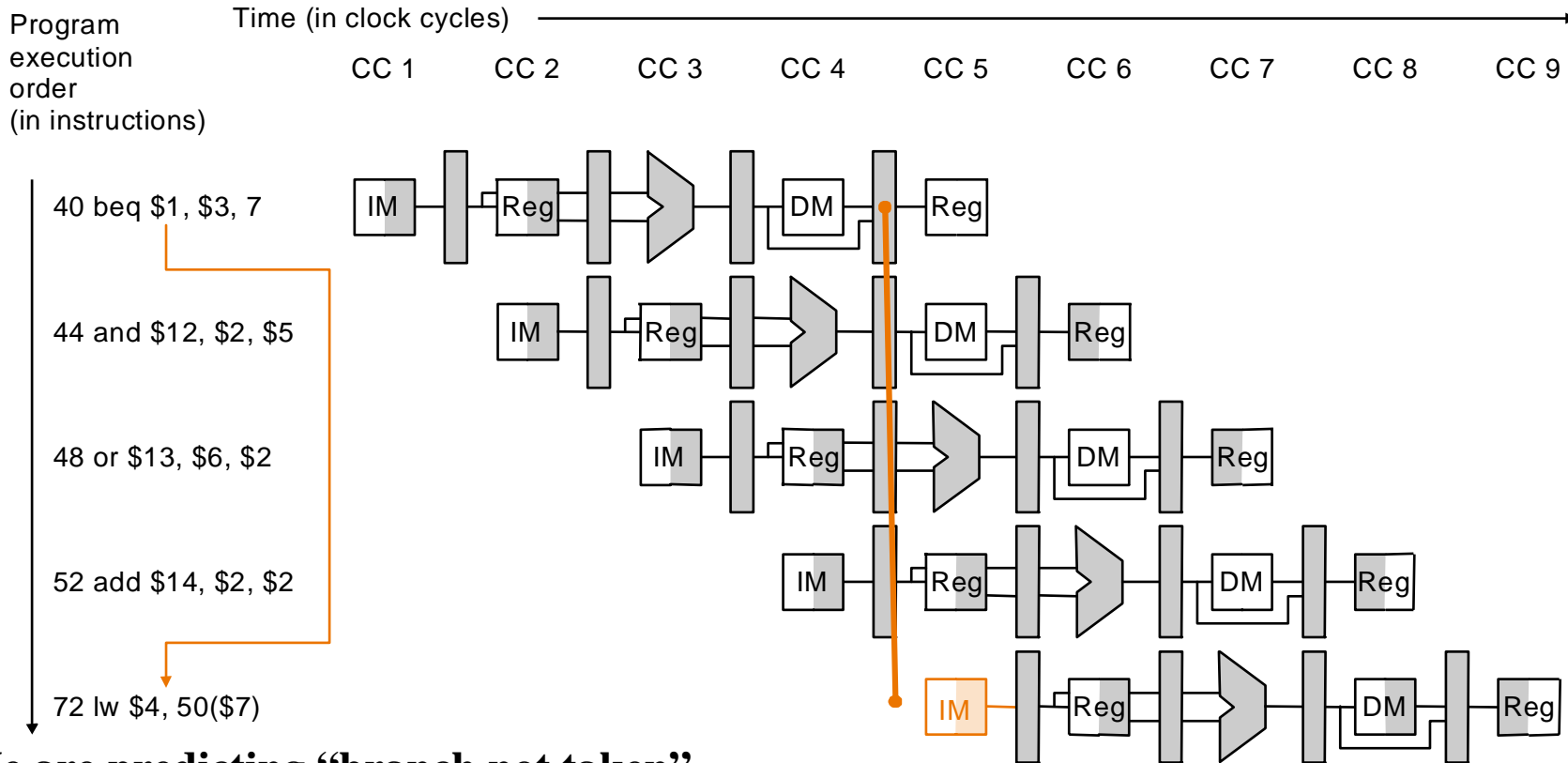
The Delay Branch Phenomenon



- **Although Beq is fetched during Cycle 4:**
 - Target address is **NOT** written into the PC until the end of Cycle 7
 - Branch's target is **NOT** fetched until Cycle 8
 - 3-instruction delay before the branch take effect
- **This is referred to as Branch Hazard:**
 - Clever design techniques can reduce the delay to **ONE** instruction

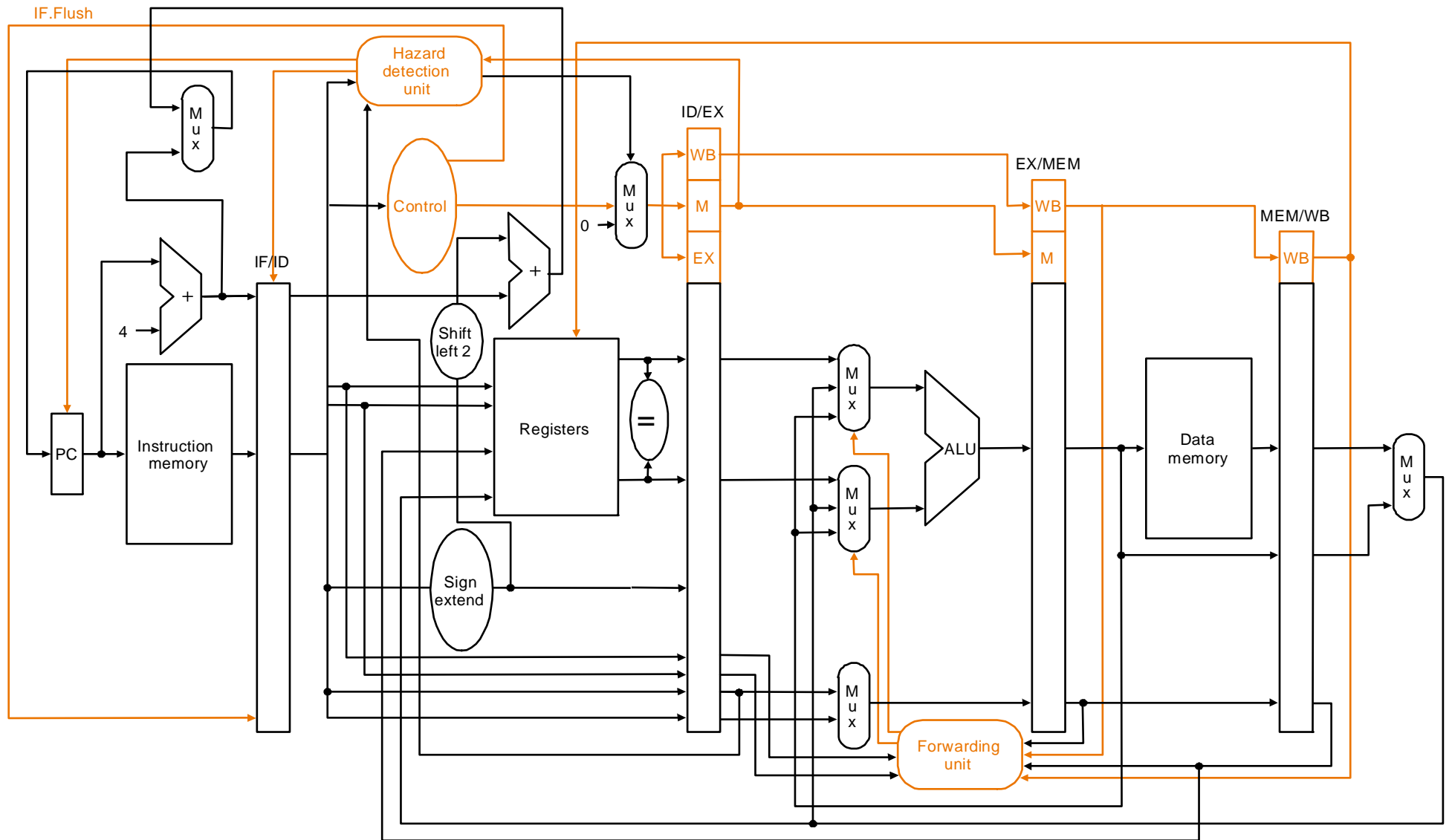
Branch Hazards

- When we decide to branch, other instructions are in the pipeline!



- We are predicting “branch not taken”
 - need to add hardware for flushing instructions if we are wrong

Flushing Instructions



Improving Performance

- **Try and avoid stalls!** e.g. reorder these instructions:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

- **Add a “branch delay slot”**
 - the next instruction after a branch is always executed
 - rely on compiler to “fill” the slot with something useful (not always possible)
- **Branch Prediction :**
 - **Static**
 - always taken, or always untaken
 - mixture : loop handling
 - **Dynamic**
 - Based on history
- **Superscalar:** start more than one instruction in the same cycle

Dynamic Scheduling

- **The hardware performs the “scheduling”**
 - hardware tries to find instructions to execute
 - out of order execution is possible
 - speculative execution and dynamic branch prediction
- **All modern processors are very complicated**
 - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
 - PowerPC and Pentium: branch history table
 - Compiler technology important

Summary

- **Disadvantages of the Single Cycle Processor**
 - Long cycle time based on the slowest Load
 - Cycle time is too long for all instructions except the Load
- **Multiple Clock Cycle Processor:**
 - Divide the instructions into smaller steps
 - Execute each step (instead of the entire instruction) in one cycle
- **Pipeline Processor:**
 - Natural enhancement of the multiple clock cycle processor
 - Each functional unit can only be used once per instruction
 - If a instruction is going to use a functional unit:
 - it must use it at the same stage as all other instructions
 - Pipeline Control:
 - Each stage's control signal depends ONLY on the instruction that is currently in that stage
- This class has given you the background you need to learn more

Where to get more information?

- **Chapter 6 of your text book**
- **Everything You Need to know about Pipeline Computers:**
 - **Peter Kogge, “The Architecture of Pipeline Computers,” McGraw Hill Book Company, 1981**
- **Some Classic References on RISC Pipelines:**
 - **Manolis Katevenis, “Reduced Instruction Set Computer Architectures for VLSI,” PhD Thesis, UC Berkeley, 1984.**
- **More from UC Berkeley:**
 - **David. A Patterson, “Reduced Instruction Set Computers,” Communications of the ACM, January 1985.**
 - **Shing Kong, “Performance, Resources, and Complexity,” PhD Thesis, UC Berkeley, 1989.**