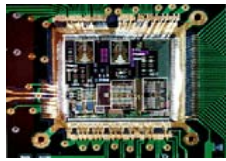
Lecture 6: Pipelining

CSCE5610 Computer System Architecture
CSCE4610 Computer Architecture

Instructor: Saraju P. Mohanty, Ph. D.

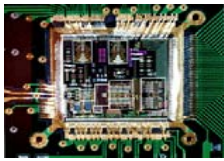
NOTE: The figures, text etc included in slides are borrowed from various books, websites, authors pages, and other sources for academic purpose only. The instructor does not claim any originality.



The Big Picture: Where are We Now?

- We know five classic components of a computer.
- We understand how the instruction set plays a key role in determining the performance, the design complexity of the datapath and controller.
- We designed a processor comprising of datapath and controller for a small set of instructions.
- Datapath:
  - Single-cycle implementation - Too slow
  - Multi-cycle implementation – Large instructions take longer time, small instructions take shorter time
- Controller: Approach I: *FSM Based Approach*
  - Structured approach to derive a circuit implementation from FSM specification
  - Implementation styles: (1) Random-logic (2) PLA (3) ROM

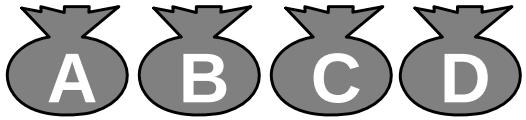
    Approach 2: *Microprogramming*
  - Control written as a program using microinstructions
  - Flexible but slower



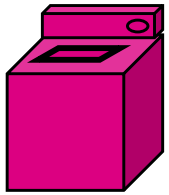
Pipelining is Natural!

Laundry Example

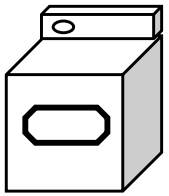
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold.



- Washing takes 30 minutes.

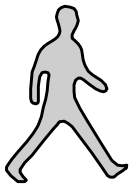


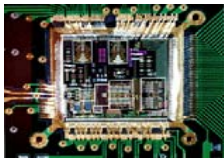
- Drying takes 30 minutes.



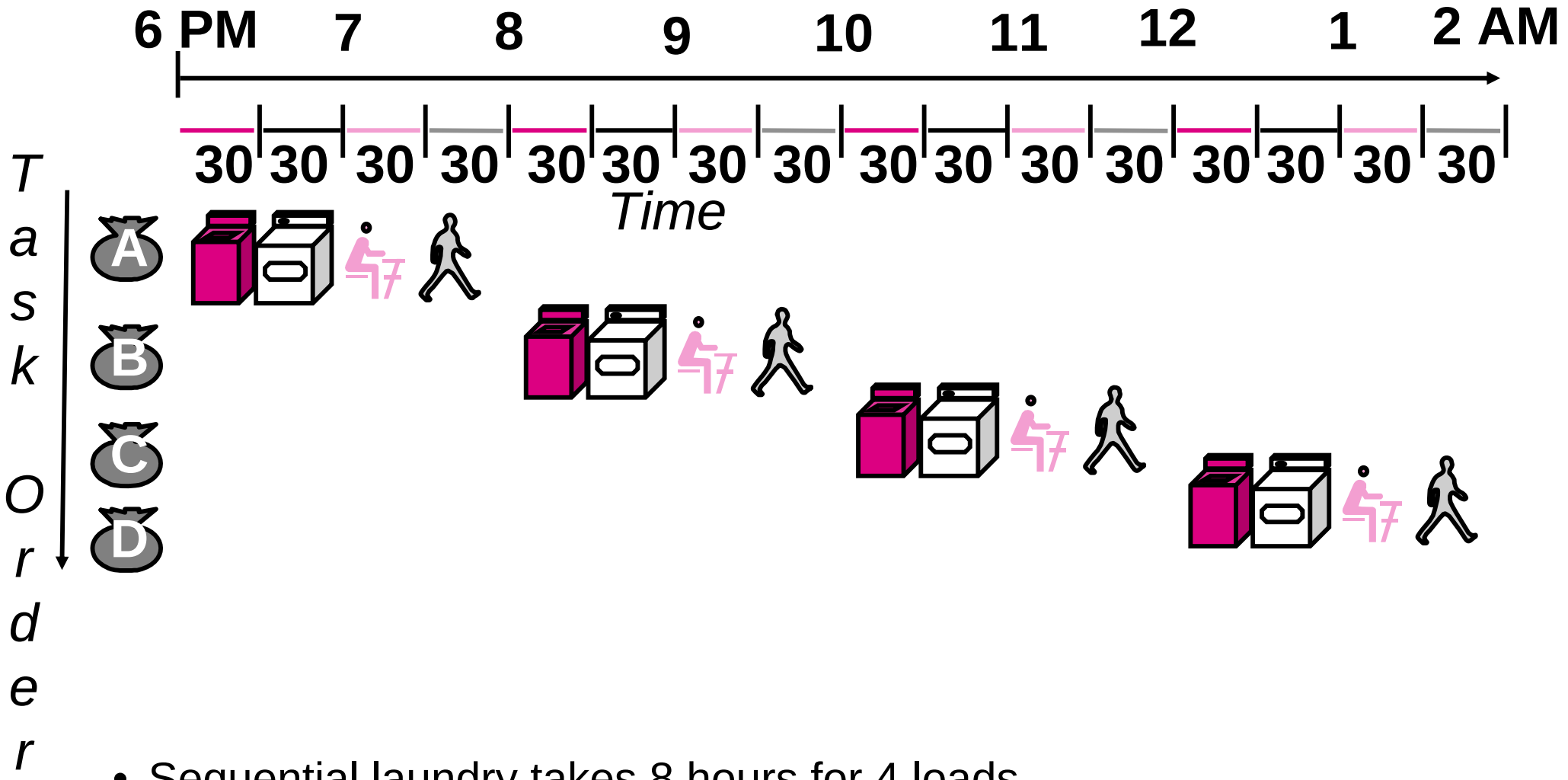
- Folding takes 30 minutes.

- Putting-away takes 30 minutes to put clothes into drawers.



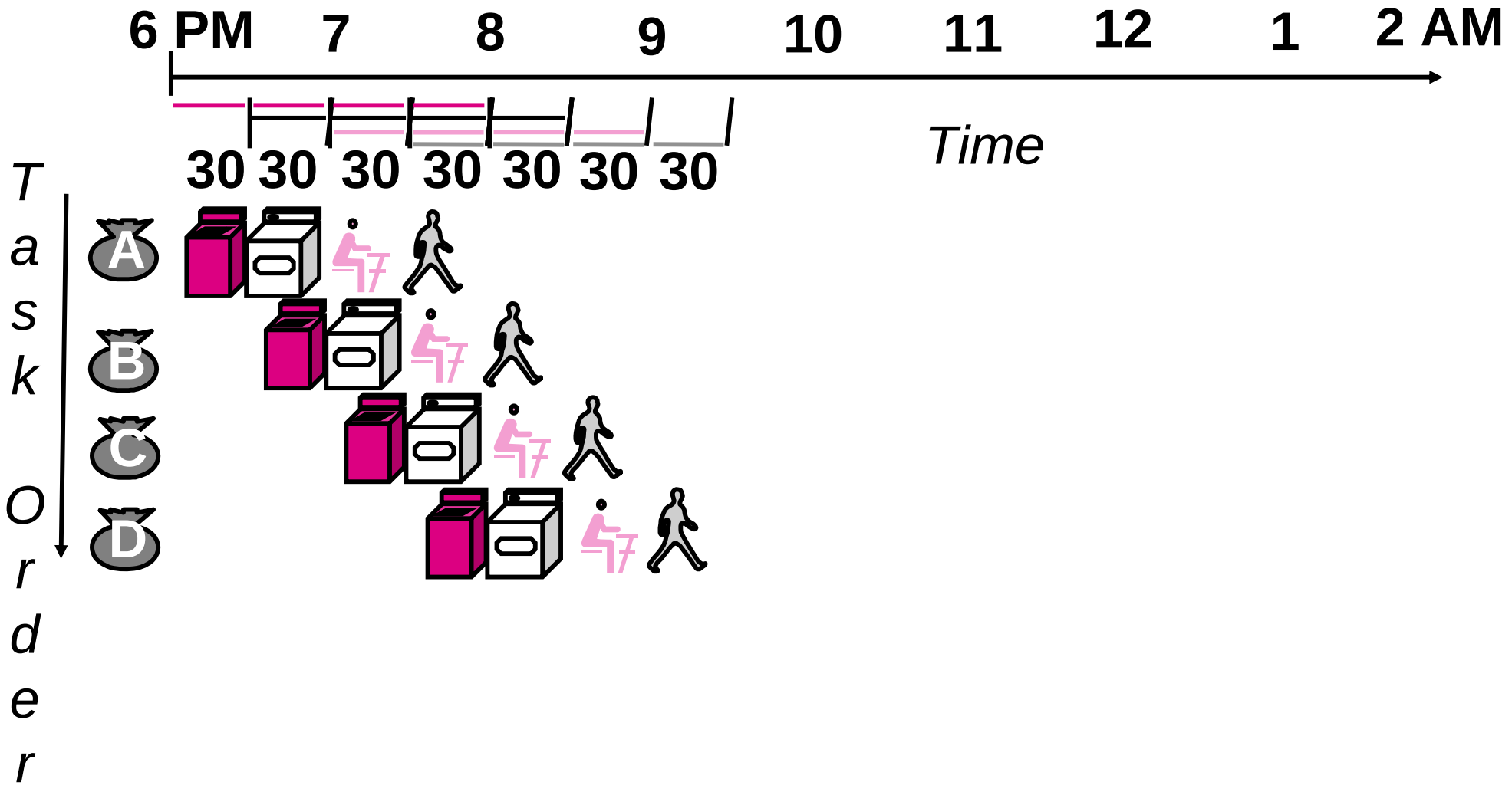


Sequential Laundry



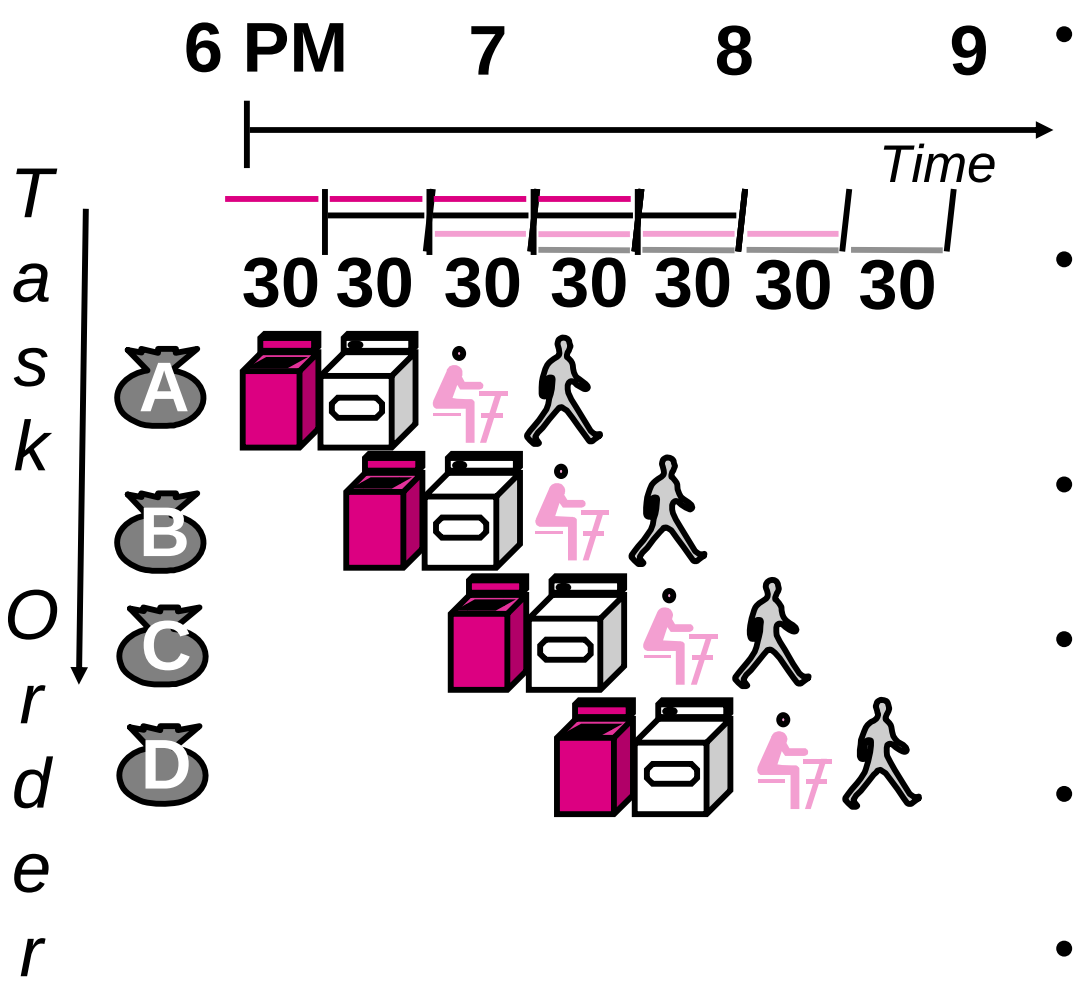
- Sequential laundry takes 8 hours for 4 loads.
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



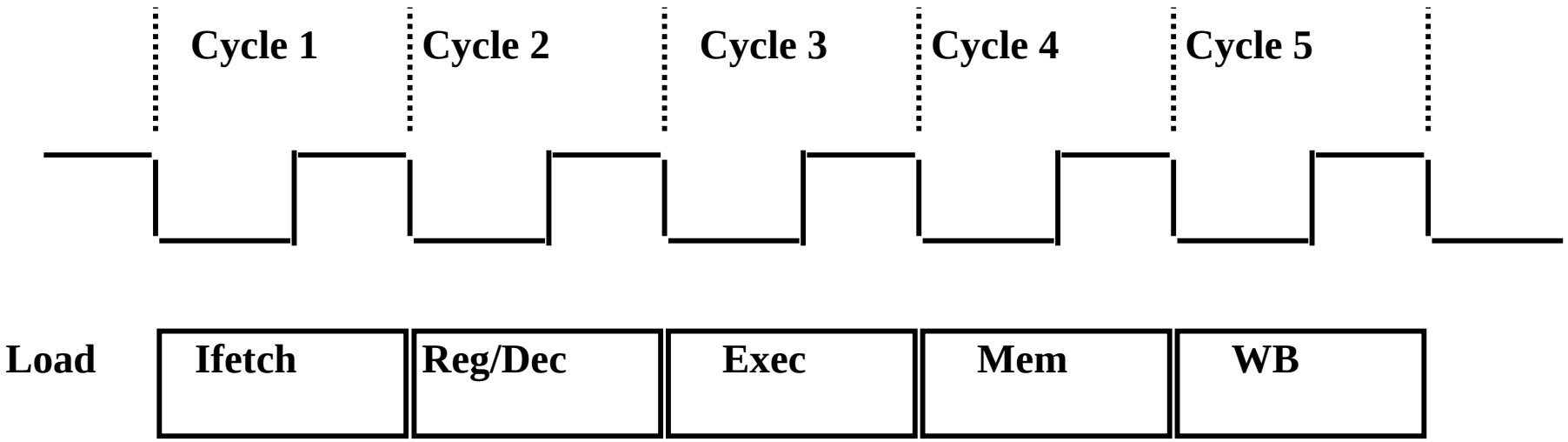
- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons



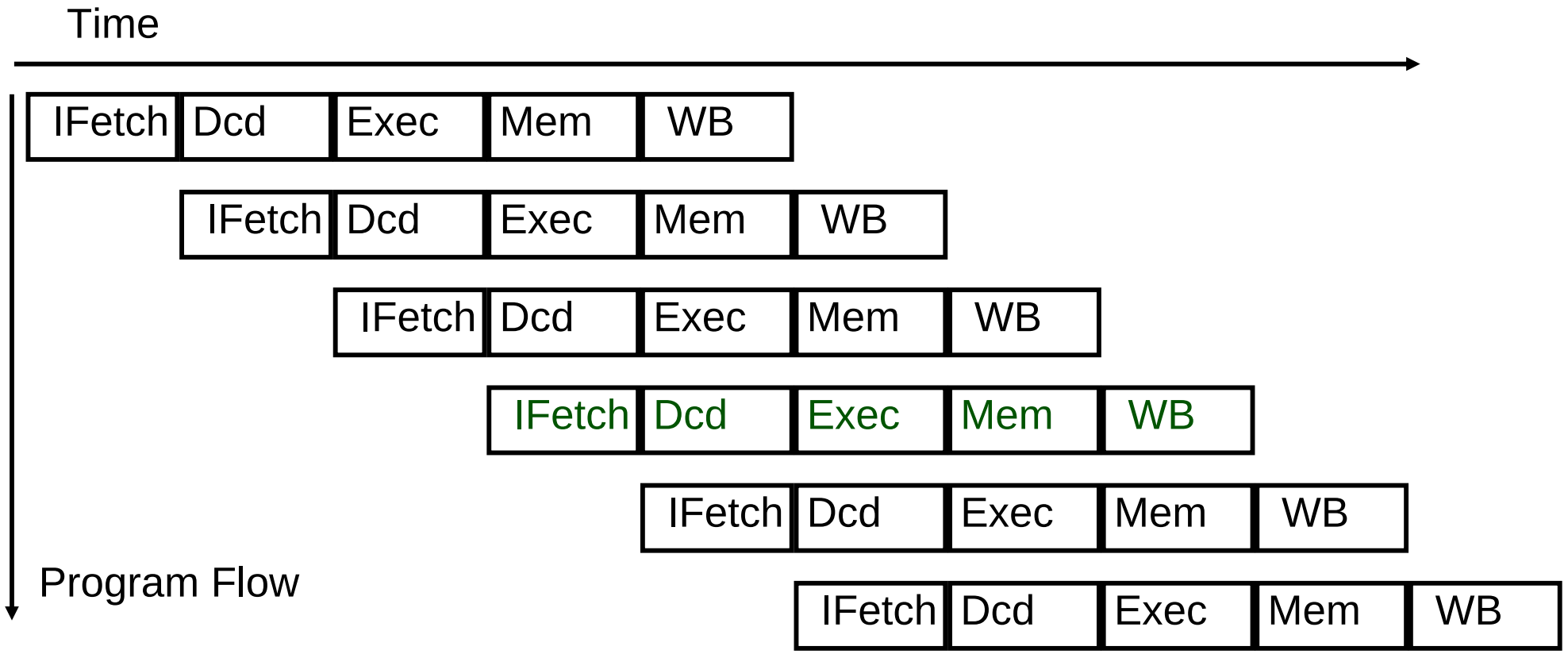
- Pipelining doesn't help latency of single task, it helps throughput of entire workload.
- Multiple tasks operating simultaneously using different resources.
- Potential speedup = Number pipeline stages.
- Pipeline rate limited by slowest pipeline stage.
- Unbalanced lengths of pipe stages reduces speedup.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

MIPS case: The Five Stages of Load

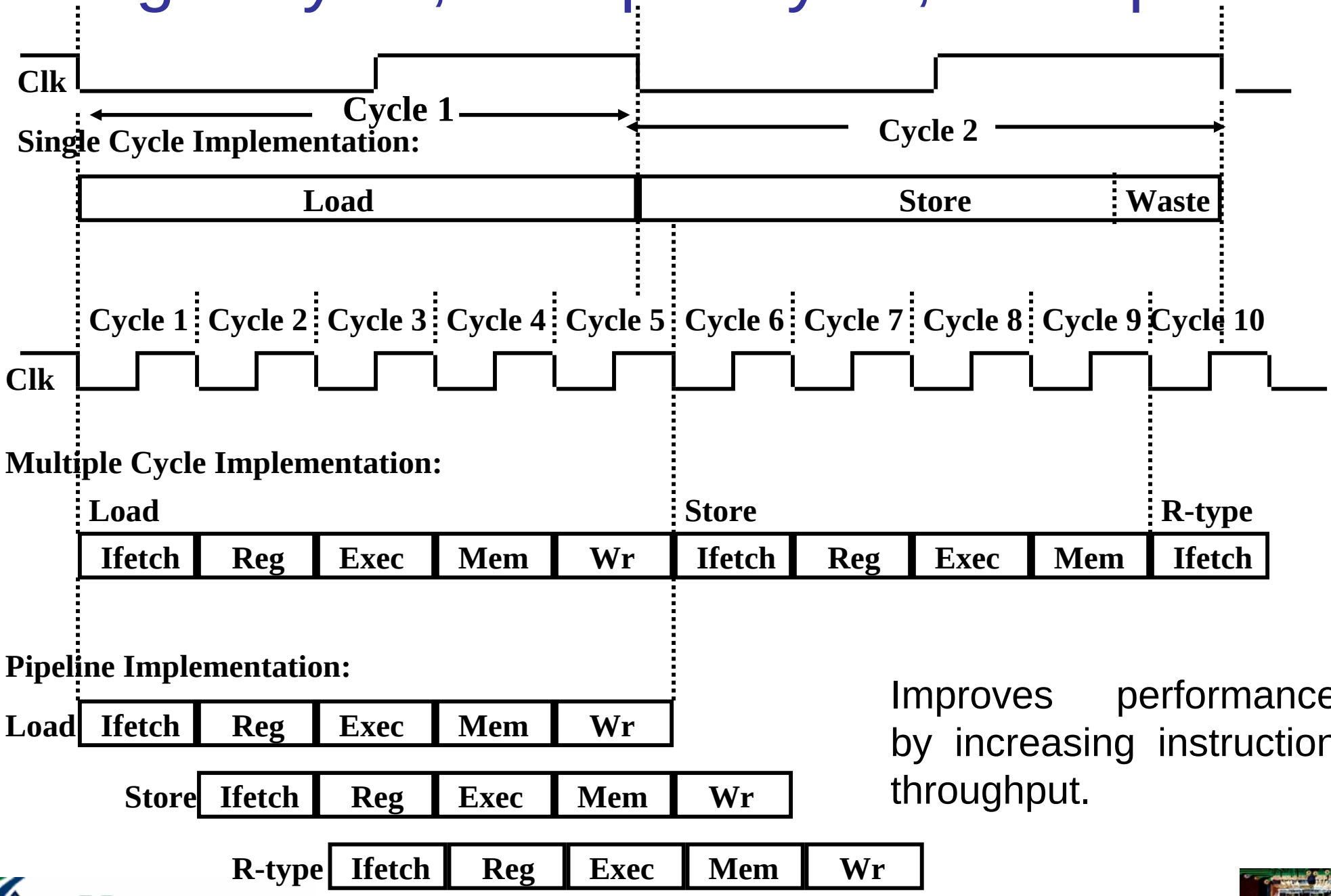


- Ifetch: Instruction Fetch
  - –Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- WB: Write the data back to the register file

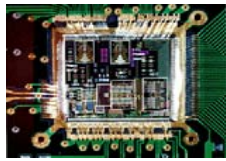
Conventional Pipelined Execution Representation



Single Cycle, Multiple Cycle, vs. Pipeline

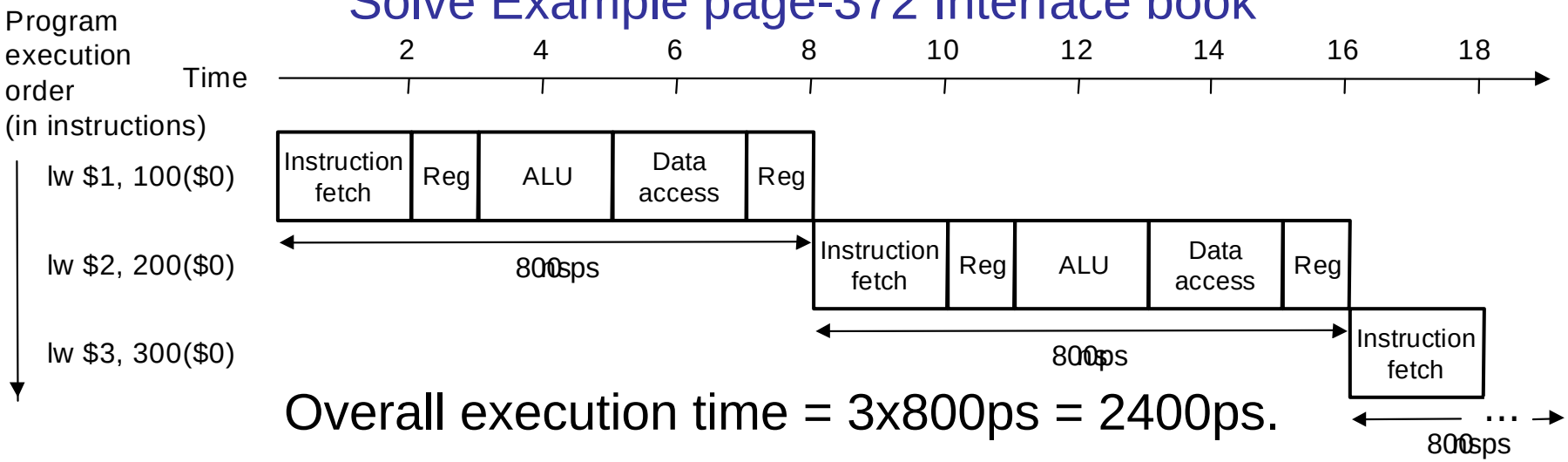


Improves performance by increasing instruction throughput.

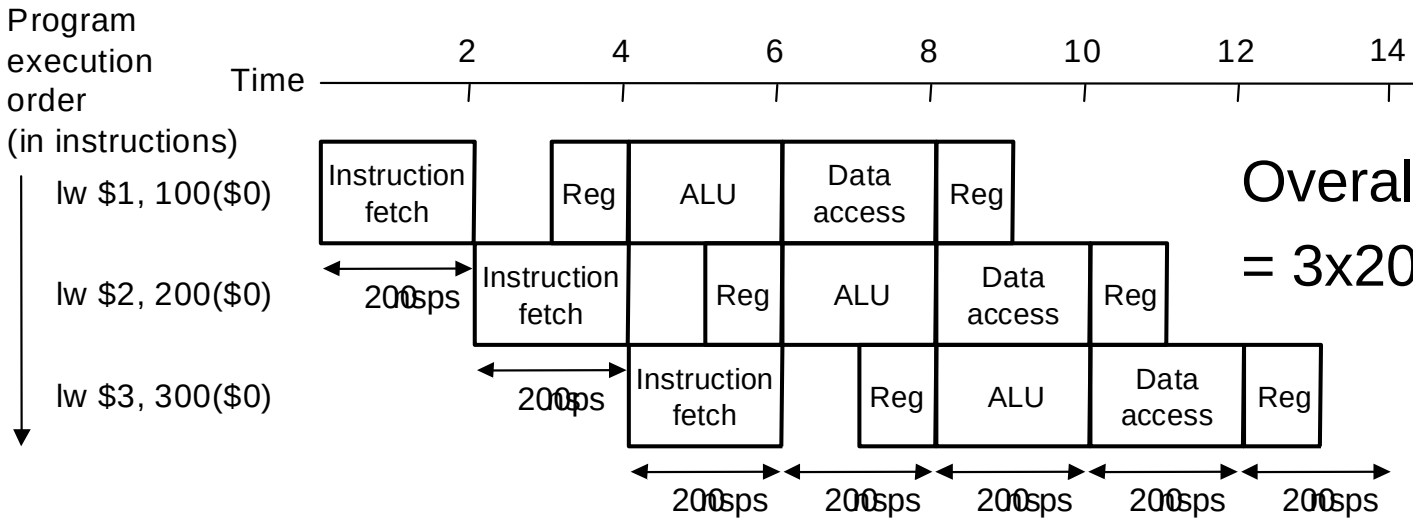


Single Cycle Vs Pipelining

Solve Example page-372 Interface book



Overall execution time = 3x800ps = 2400ps.

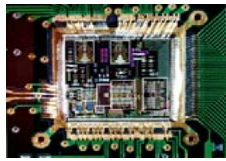


Overall execution time = 3x200ps = 600ps.

Speedup = 2400/600 = 4.

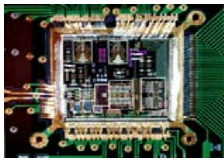
Ideal speedup is number of stages in the pipeline. Do we achieve this?

Solve Example page-A10 of Quantitative book.

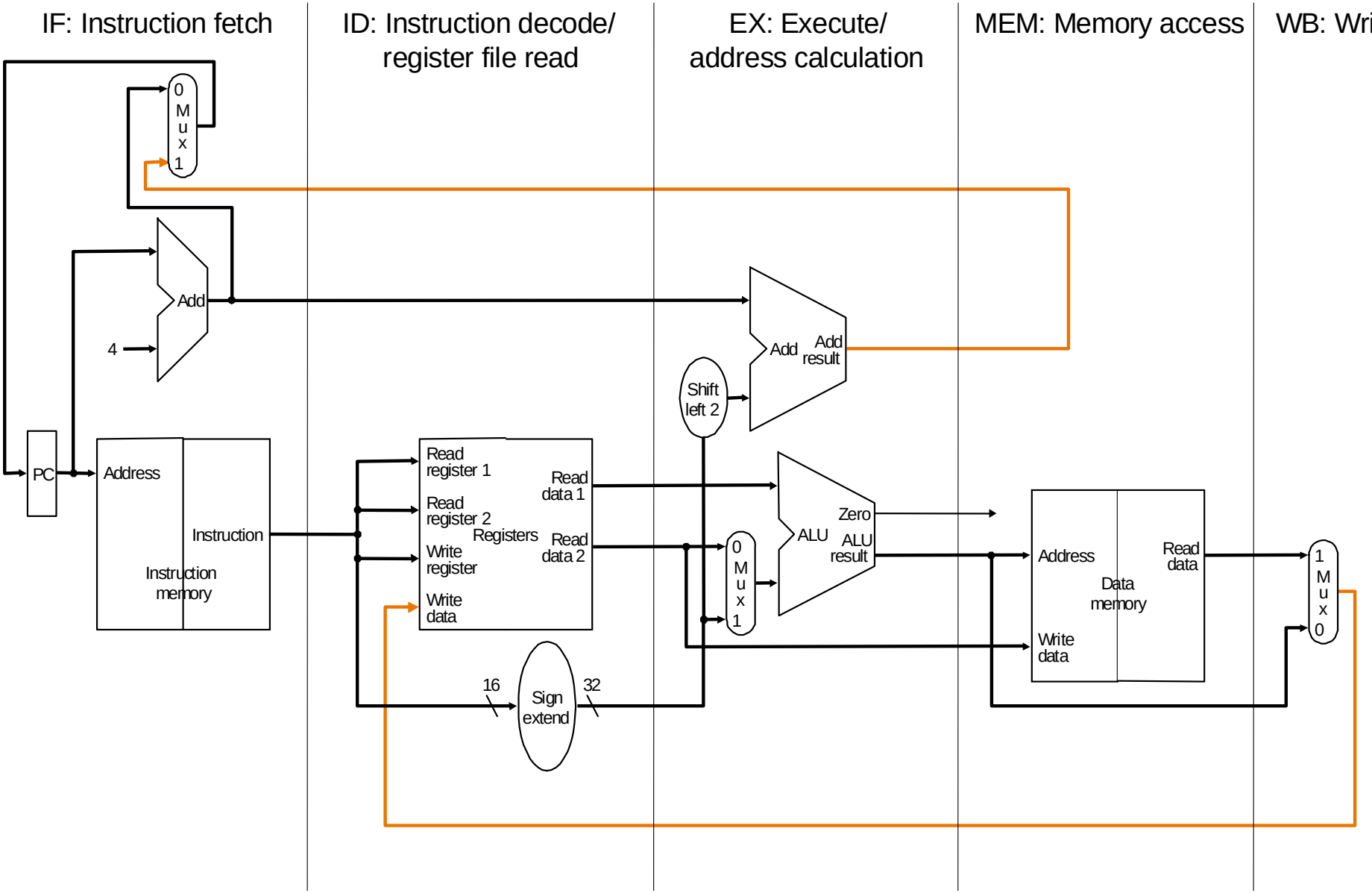


Pipelining – What makes it easy/hard?

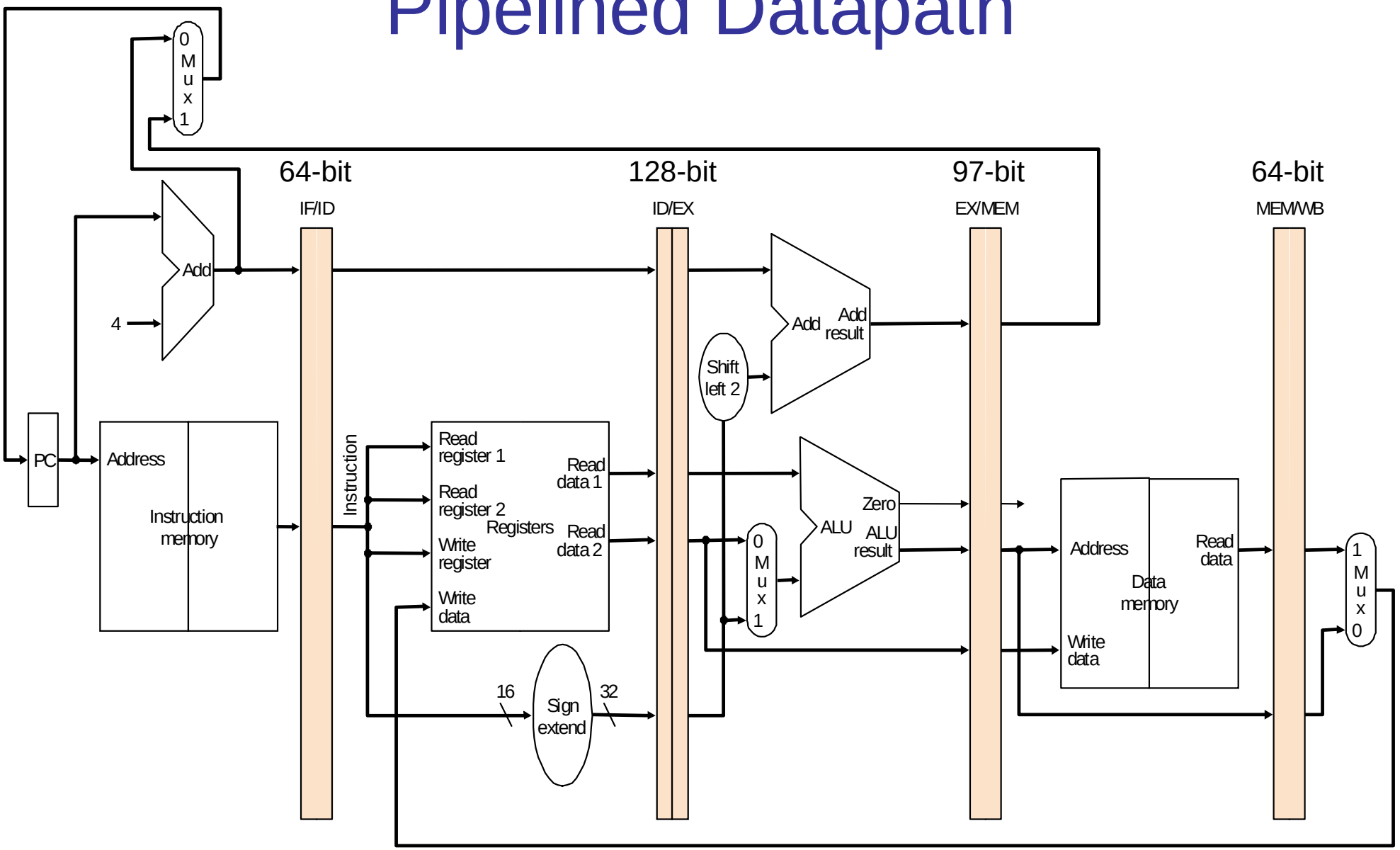
- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - data hazards: an instruction depends on a previous instruction
  - control hazards: need to worry about branch instructions
- We'll build a simple pipeline and look at these issues.



Pipelining the Datapath: Basic Idea

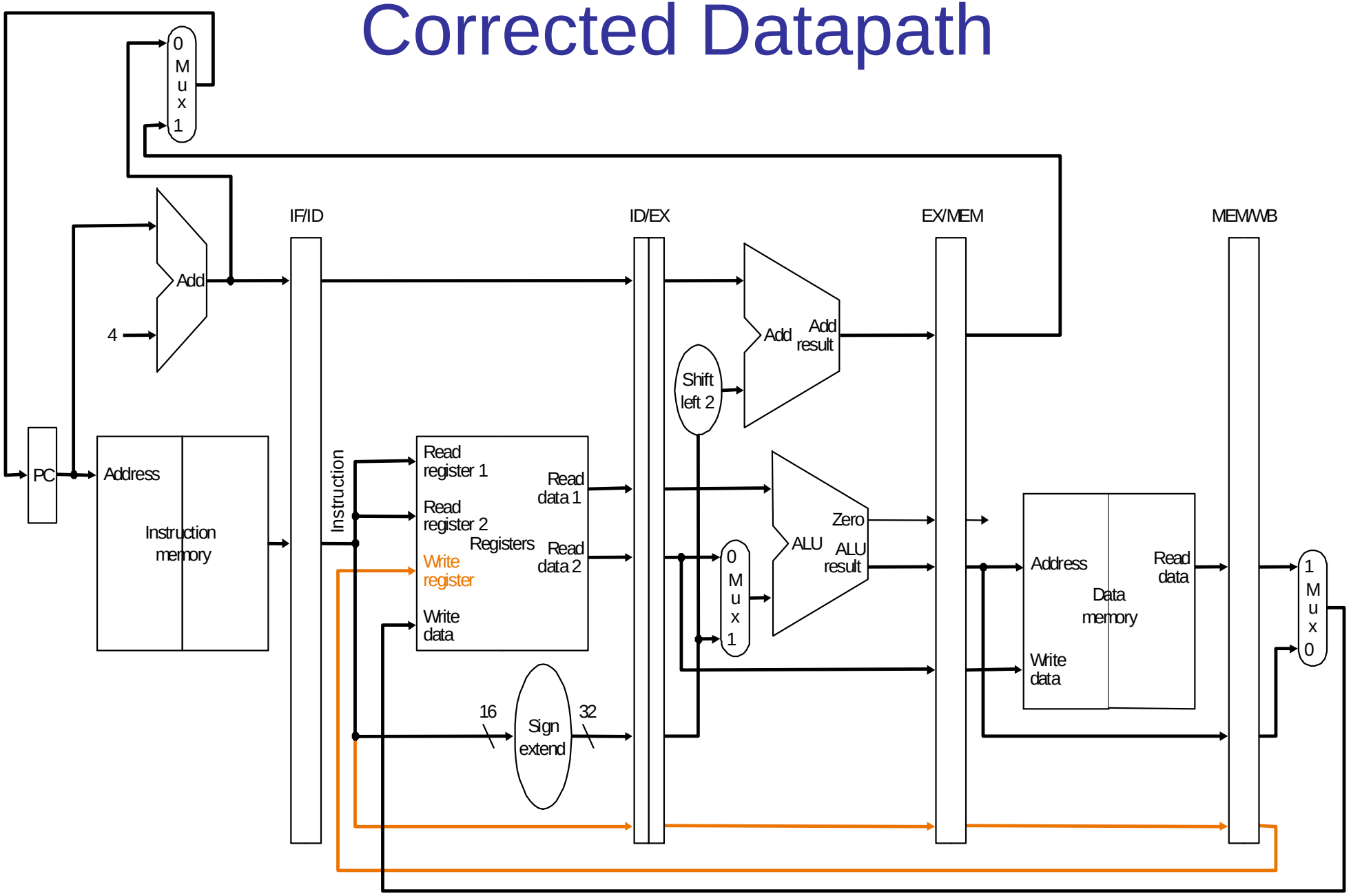


Pipelined Datapath



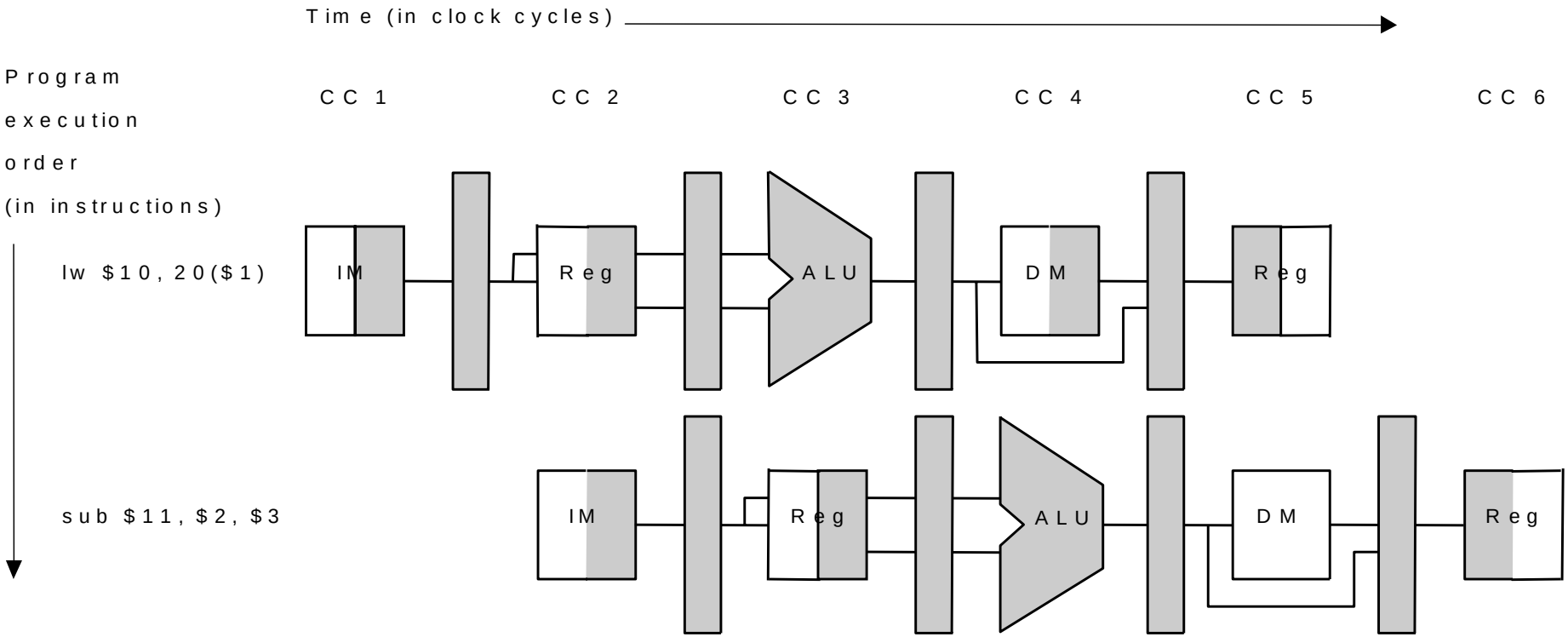
- Follow Fig. 12 (page-389) to Fig. 14 (page-391) to understand pipelined execution of *lw* instruction. Others instructions will be similar.

Corrected Datapath

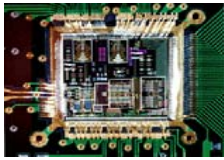


- For load instruction, register number is needed in the last stage, thus same needs to be passed along in order to be preserved.

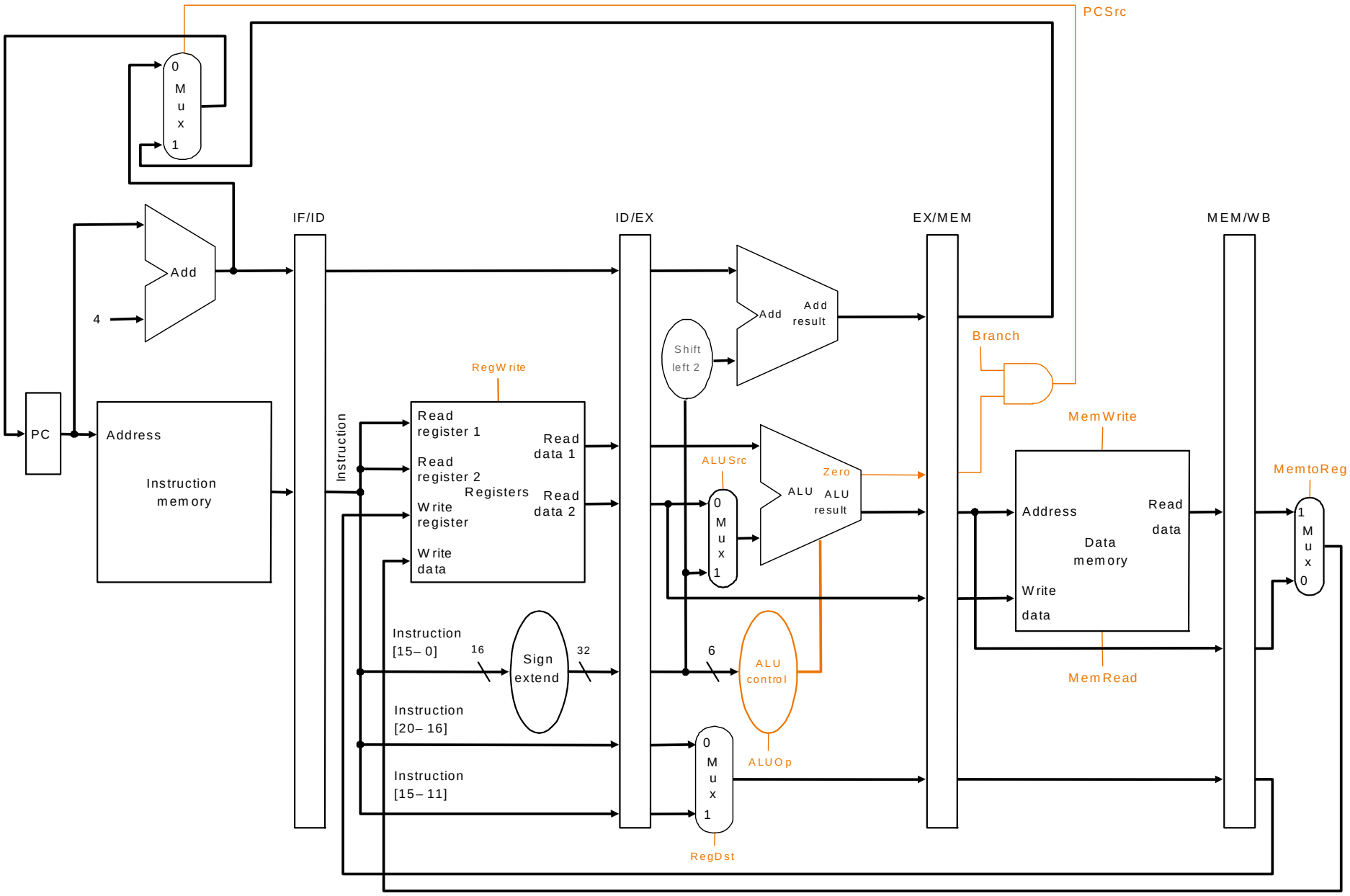
Graphically Representing Pipelines

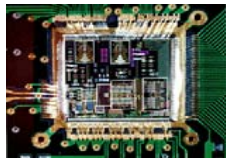


- Pipeline can be thought of as a series of datapaths shifted in time.
- The above graphics can help in answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths



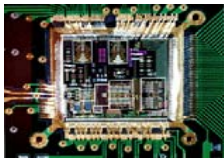
Pipeline Control





Pipeline Control

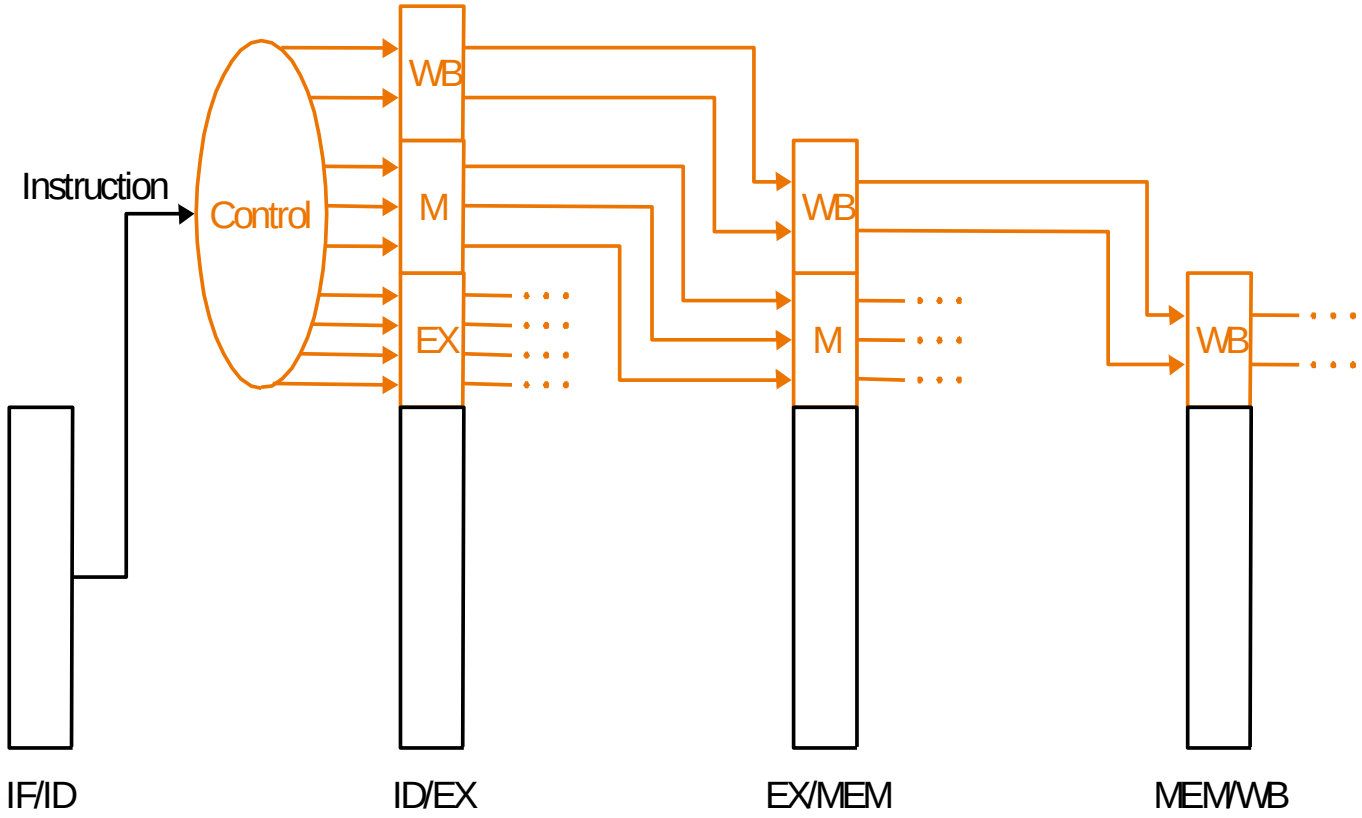
- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back



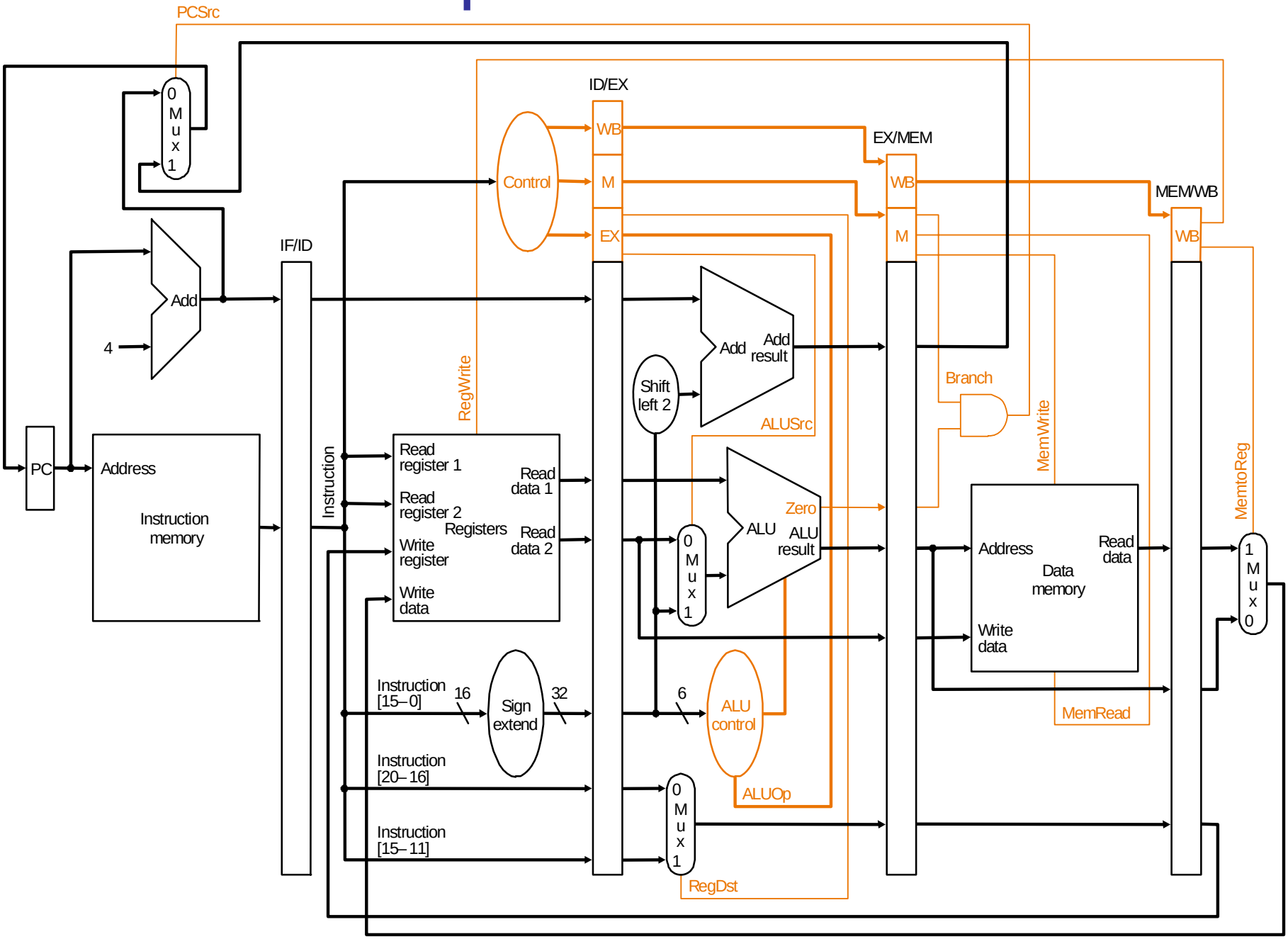
Pipeline Control

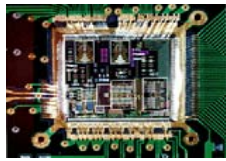
- Pass control signals along just like the data.

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X



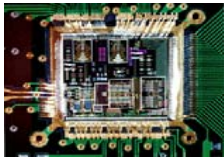
Datapath with Control



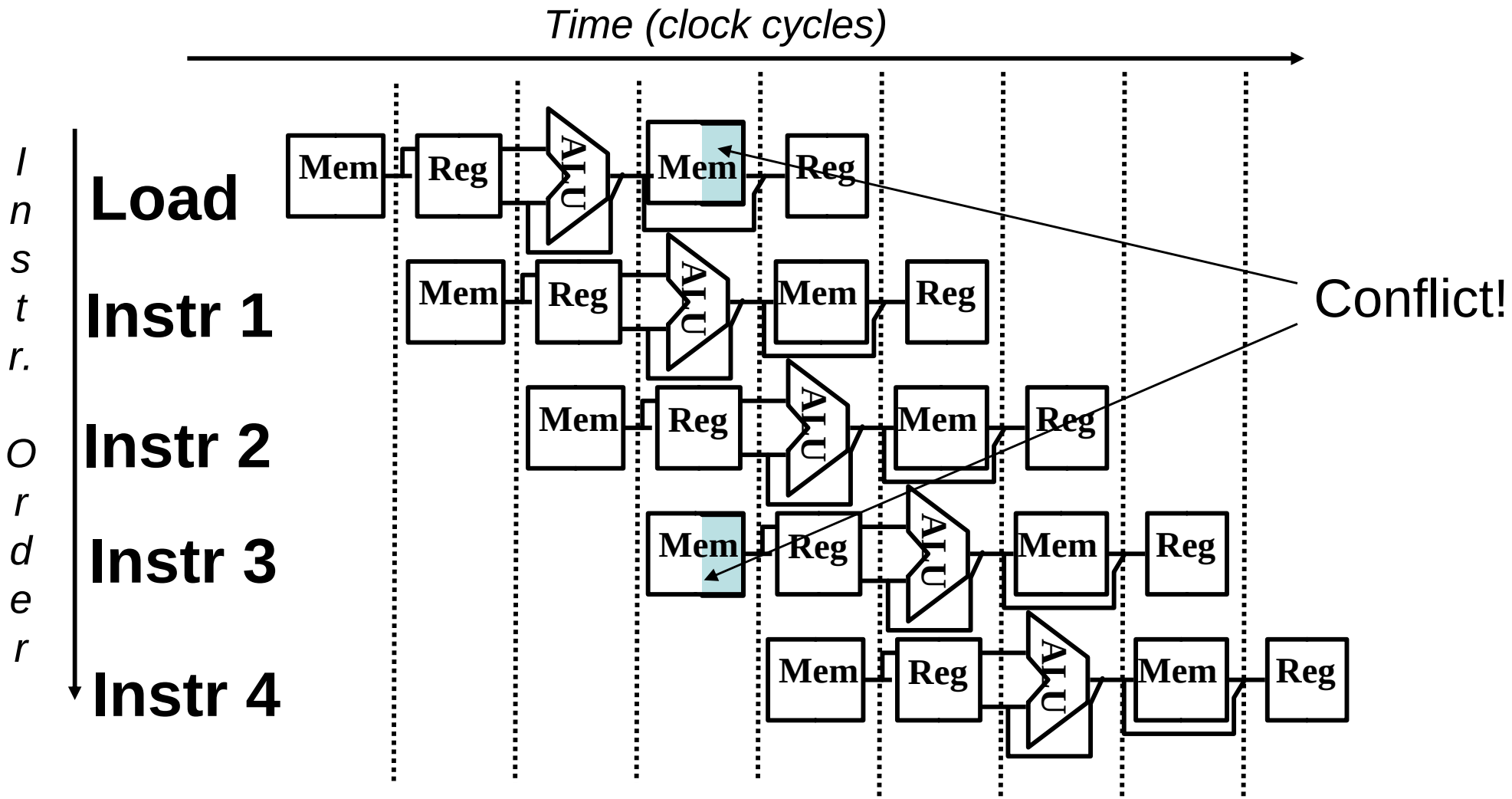


Major Hurdles of Pipelining

- Pipeline Hazards
  - Dictionary meaning of hazard: “a source of danger”
  - structural hazards: attempt to use the same resource two different ways at the same time
    - e.g., combined washer/dryer would be a structural hazard
  - data hazards: attempt to use item before it is ready
    - Instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaluated
    - Branch instructions
- One Solution: Wait until dependencies are resolved
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards



Single Memory is a Structural Hazard



- One memory port generates conflict whenever memory reference occurs.
- Detection is easy in this case! (right half highlight means read, left half write).
- Solutions: Stall the pipeline or use split cache.

NOTE: Refer Fig A.4 page-A-14 of Quantitative book for more details.

Performance of Pipeline with Stalls

- Speed up from pipelining = $\frac{CPIunpipelined}{CPIpipelined} \times \frac{ClockCycleUnpipelined}{ClockcyclesPipelined}$

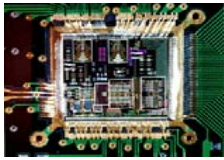
- Speed up from pipelining =

$$\frac{1}{1 + PipelineStallCyclesPerInstruction} \times \frac{ClockCycleUnpipelined}{ClockCyclesPipelined}$$

- Speed up from pipelining =

$$\frac{1}{1 + PipelineStallCyclesPerInstruction} \times PipelineDepth$$

Solve Example page-A13 of Quantitative book.



Data Hazard on r1

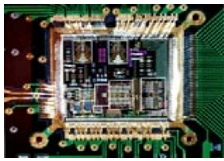
add r1 ,r2,r3

sub r4, r1 ,r3

and r6, r1 ,r7

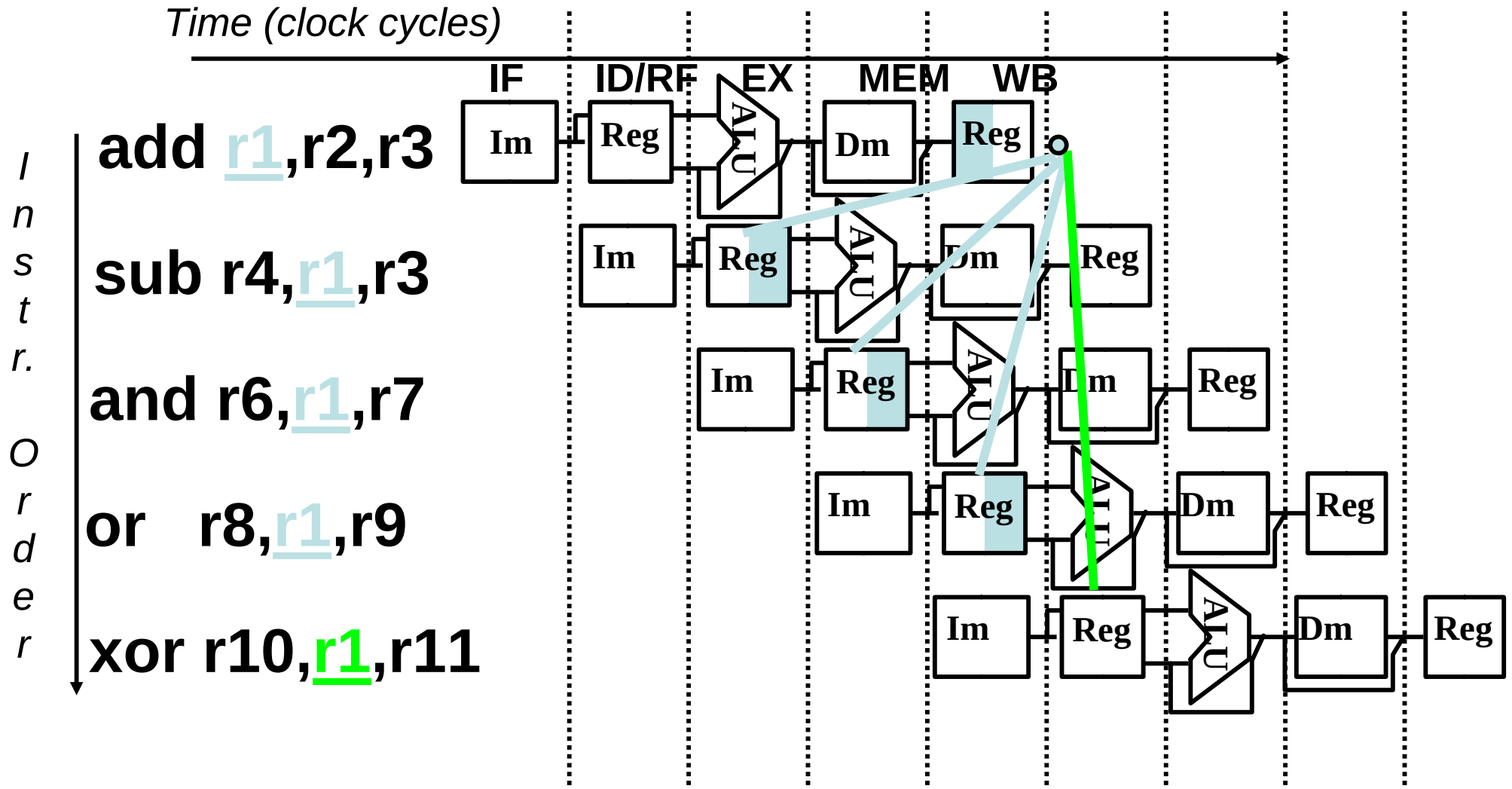
or r8, r1 ,r9

xor r10, r1 ,r11



Data Hazard on r1:

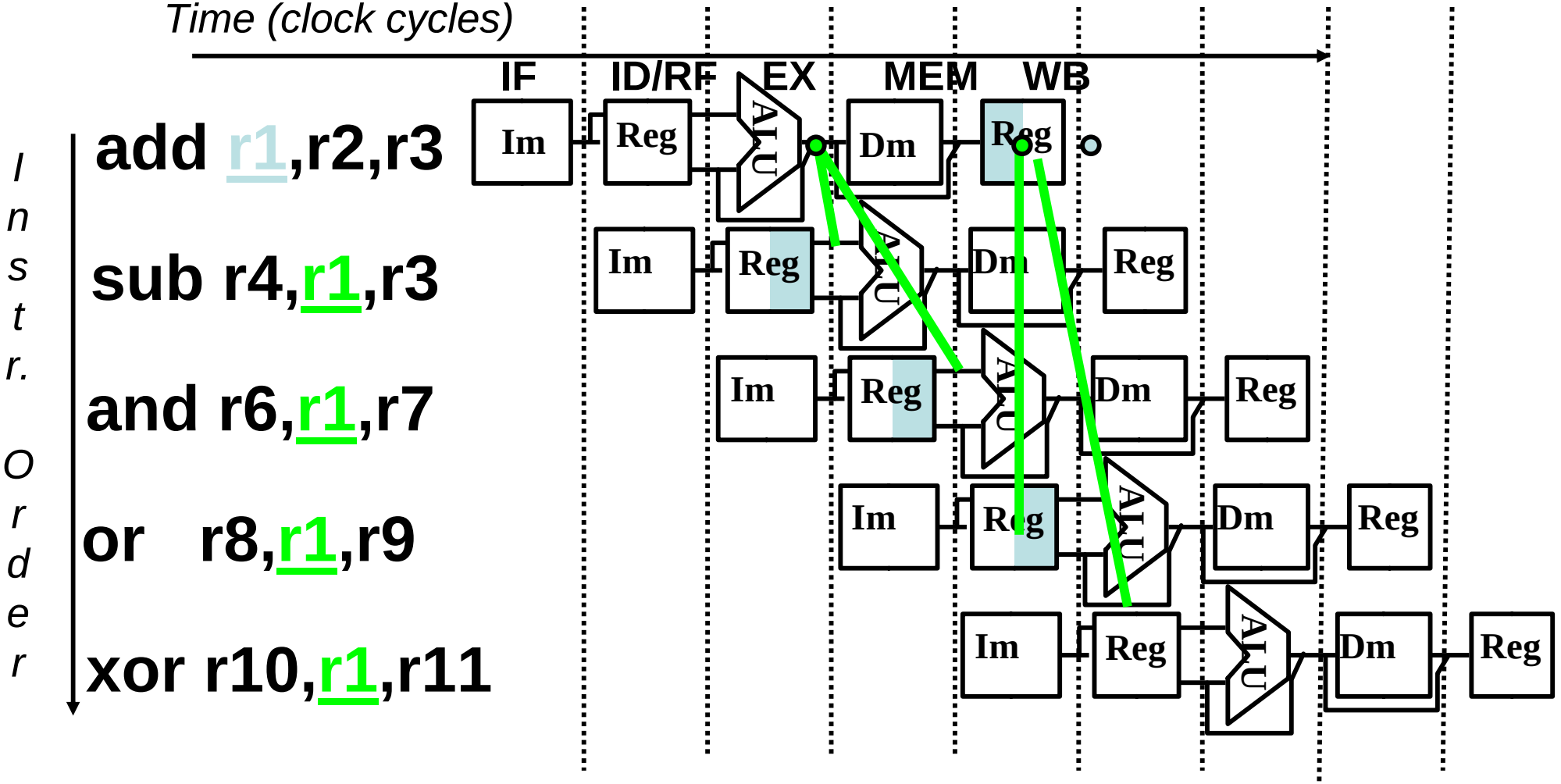
- Dependencies backwards in time are hazards.



NOTE: Refer Fig A.6 page-A-16 of Quantitative book for more details.

Data Hazard Solution

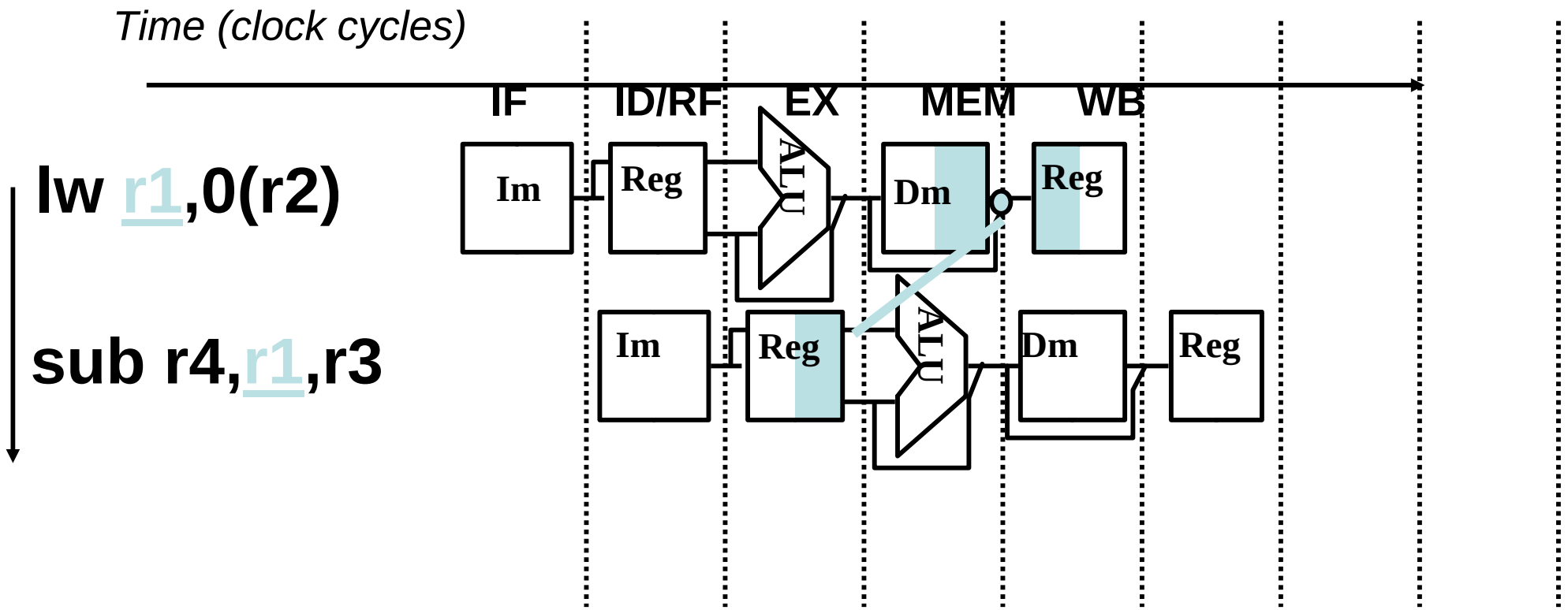
- "Forward" result from one stage to another.
- "or" OK if define read/write properly.



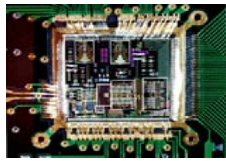
NOTE: Refer Fig A.7 page-A-18 of Quantitative book for more details.

Forwarding (or Bypassing): What about Loads?

- Can't solve with forwarding:
  - Must delay/stall instruction dependent on load.
  - A hardware called "pipeline interlock" detects hazard and stalls.



NOTE: Refer Fig A.9 page-A-20 of Quantitative book for more details.

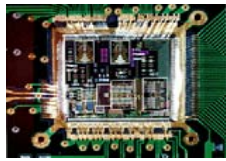


Forwarding Unit

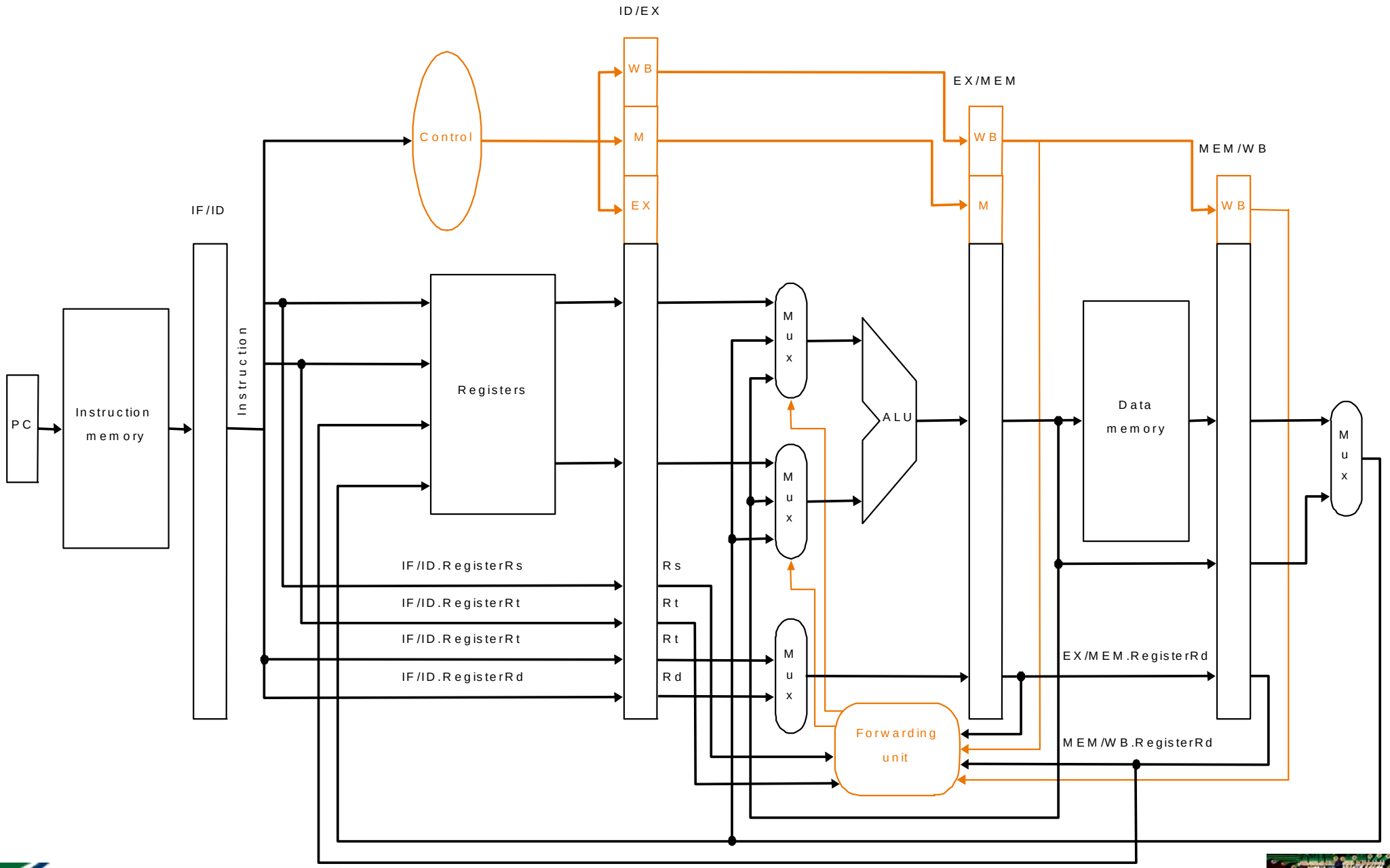
- Need to detect a hazard and then forward the proper value to resolve the hazard.
- When an instruction tried to read a register in its EX stage that an earlier instruction intends to write in its WB stage, then we need the values as inputs to the ALU.
- Notation: “ID/EX.RegisterRs”

Name of the pipeline register *name of the field in ID/EX register*

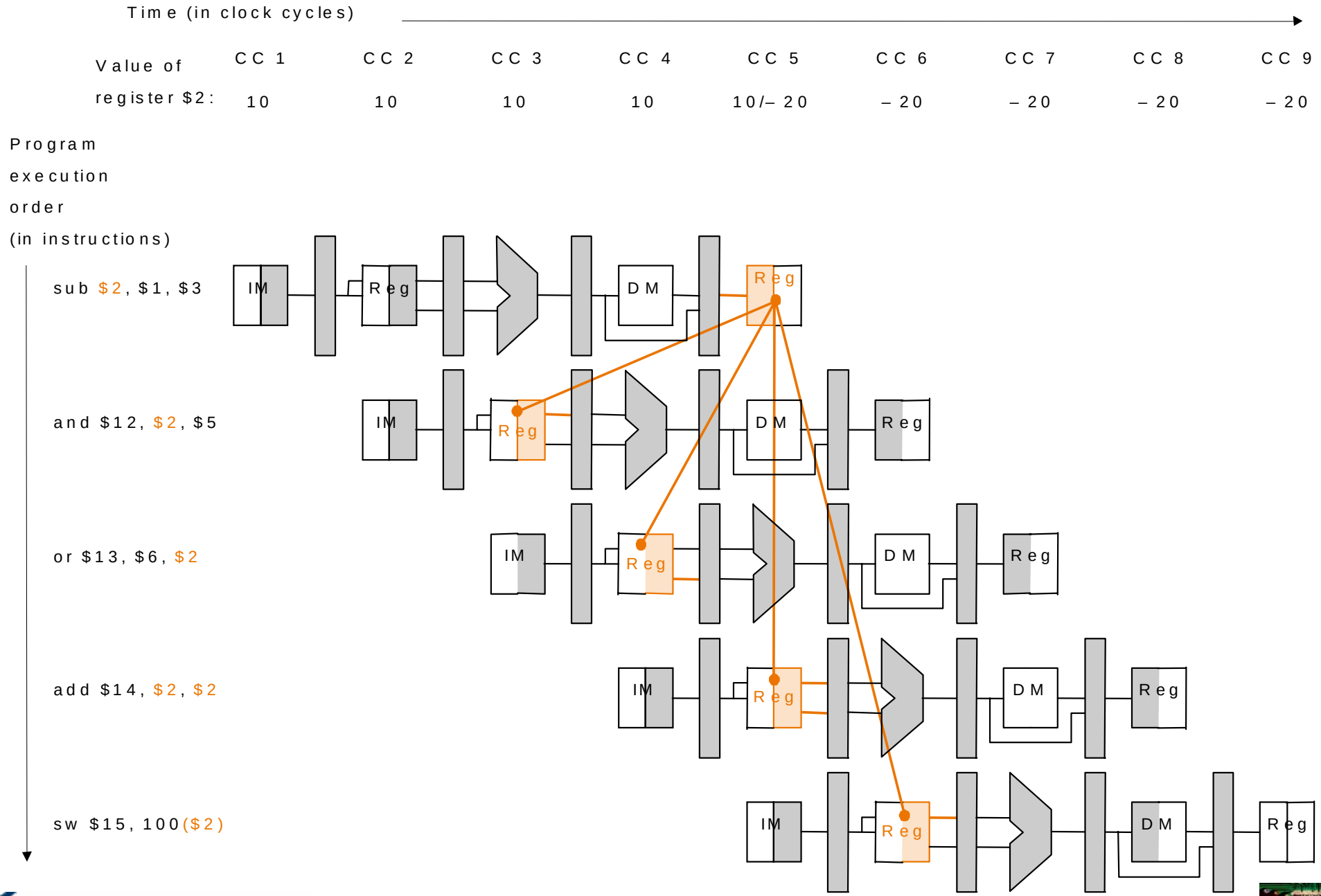
- Two pairs of hazard conditions:
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRd



Hardware with Forwarding Unit

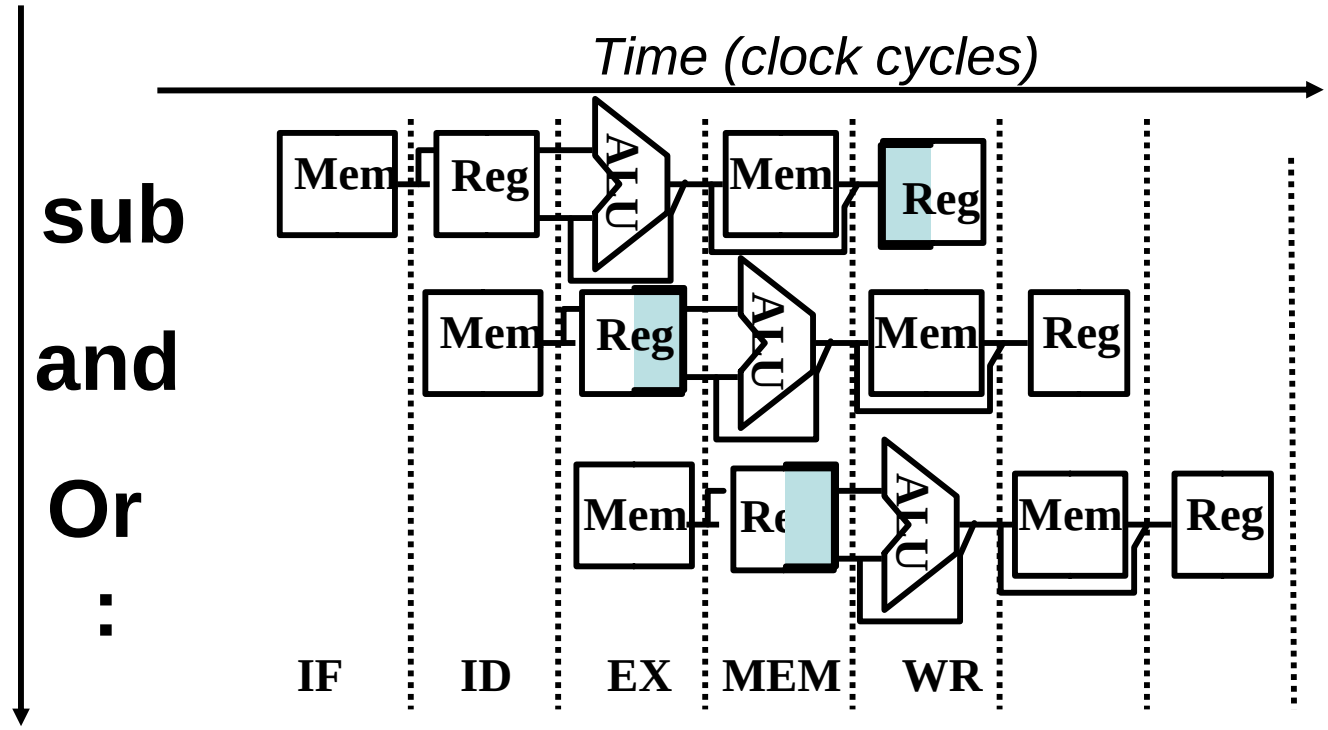


Dependence Detection: An Example



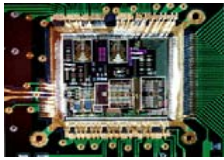
Data Hazard Detection by Forwarding Unit

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



- The first hazard is on register $2, between the result of *sub $2, $1, $3* and the first read operand of *and $12, $2, $5.*
- This hazard is of type 1a; can be detected by observing that

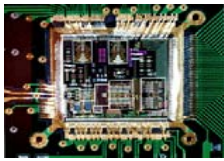
  EX/MEM.RegisterRd = ID/EX.RegisterRs = $2



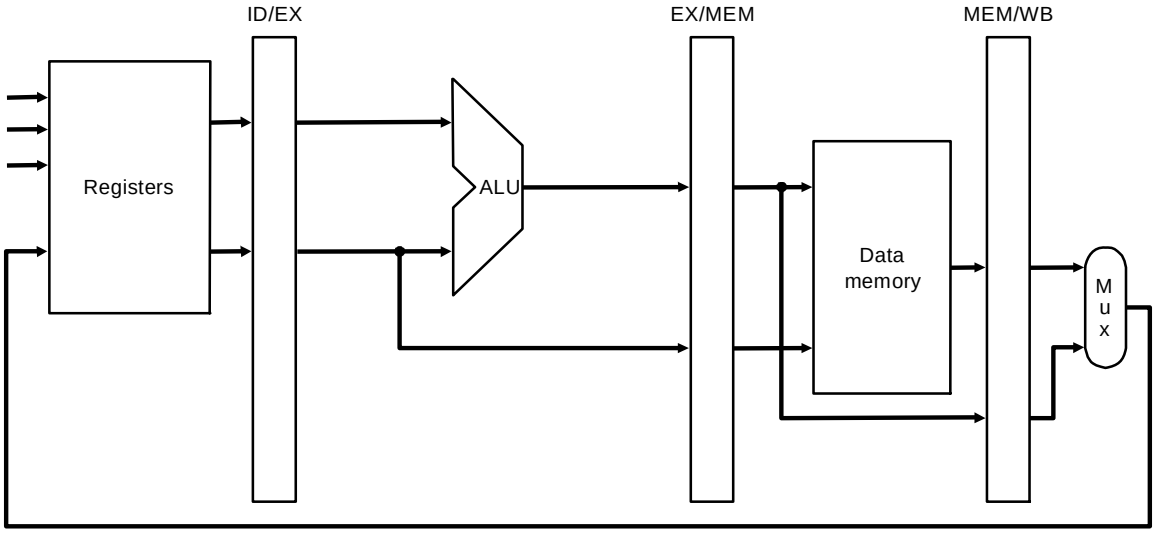
Refining Hazard Detection Conditions

- Some instructions do not write registers, therefore the following conditions are inaccurate => some times data is forward unnecessarily.
- Two Pairs of hazard conditions:
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRd
- Solution: check whether the RegWrite signal is active or not

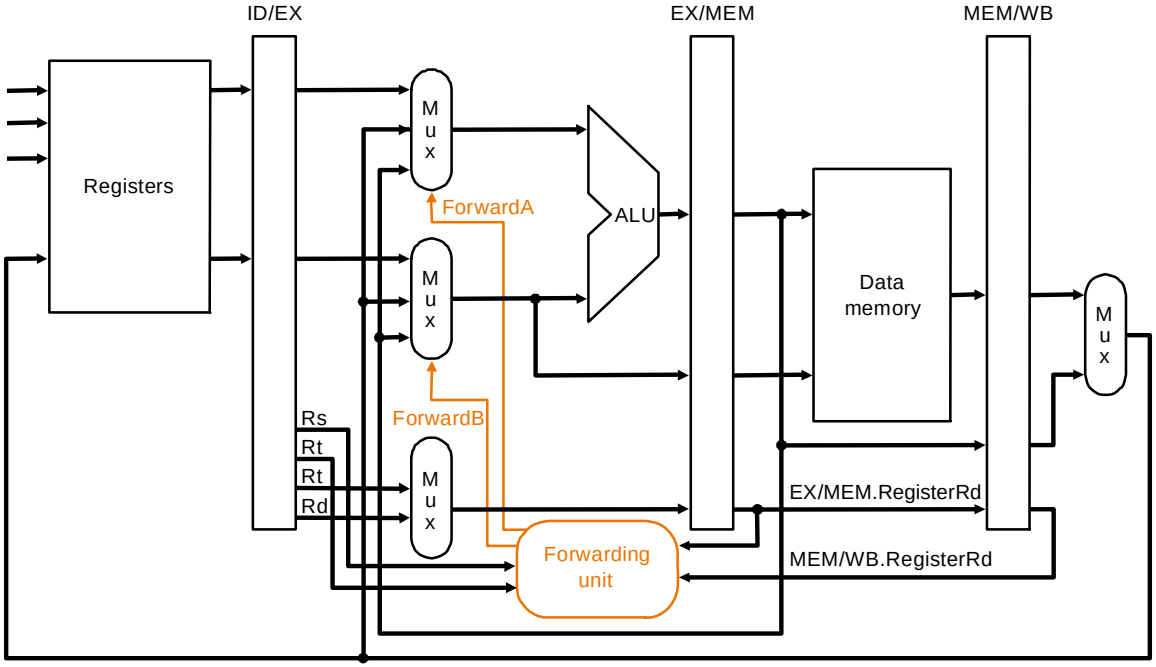
  i.e., by checking WB field in EX and MEM stages will be enough



How to forward the data?



a. No forwarding

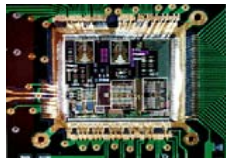


b. With forwarding

NOTE: R_t is one field, but shown twice.

Assumption: Only instructions we need to forward are the four R-type instructions: *add, sub, and,* and *or.*

Mux Control	***Source***	***Explanation***
Forward A = 00	ID/EX	The first ALU operand comes from the register file.
Forward A = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
Forward A = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result
Forward B = 00	ID/EX	The second ALU operand comes from the register file.
Forward B = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
Forward B = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.



Data Forwarding Unit – Final conditions

- EX hazard:

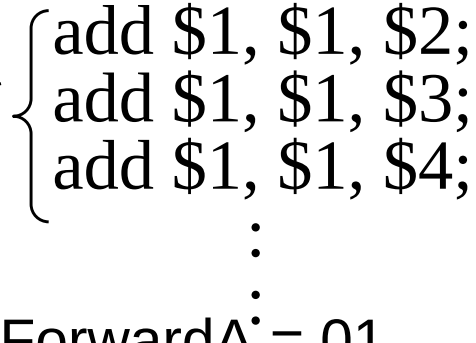
  If (EX/MEM.RegWrite
  and (EX/MEM. RegisterRd != 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

  If (EX/MEM.RegWrite
  And (EX/MEM. RegisterRd != 0)
  And (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

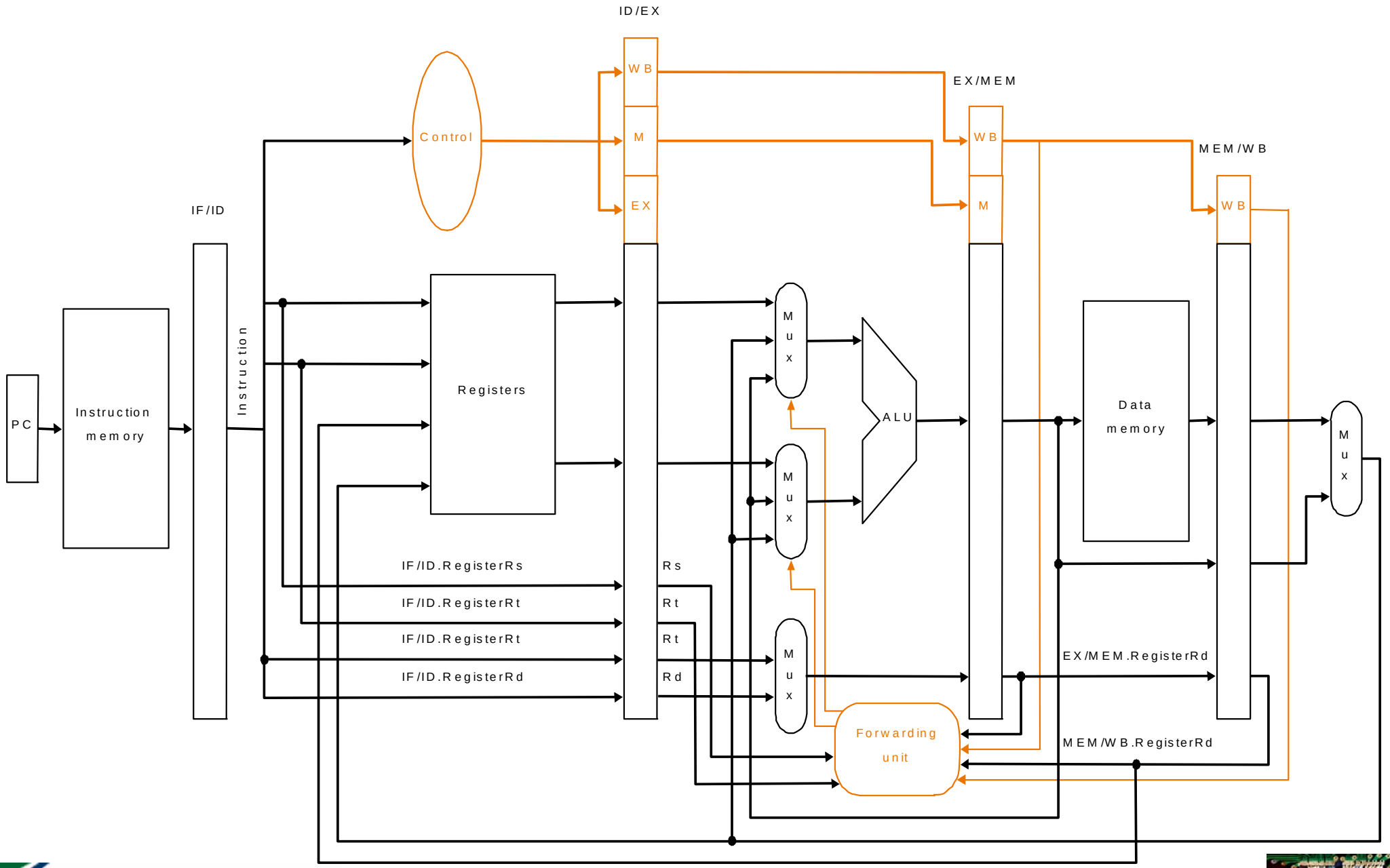
- MEM hazard:

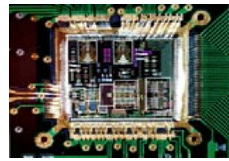
  If (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd != 0)
  and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

  If (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd != 0)
  and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01



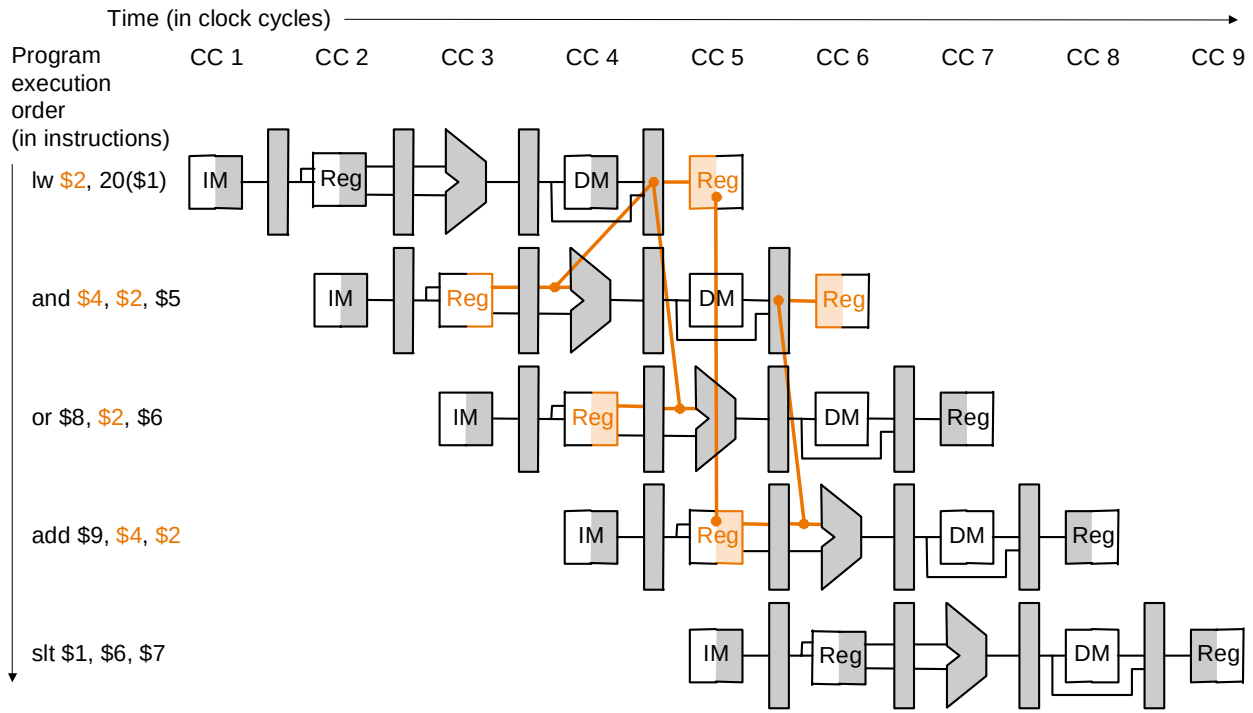
Hardware with Forwarding Unit





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” the load instruction.

Hazard Detection Unit

- When forwarding unit fails to resolve, the hazard then we need to resort to a hazard detection unit.
- Operates during ID stage so that it can insert the stall between load and the instruction that immediately uses the load results.
- Thus the hazard detection unit checks for the load instructions:

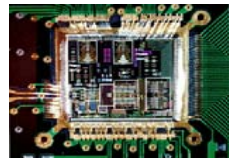
If (ID/EX. MemRead *-- checks to see if it's a load*

and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or

(ID/EX.RegisterRt = IF/ID.RegisterRt)))

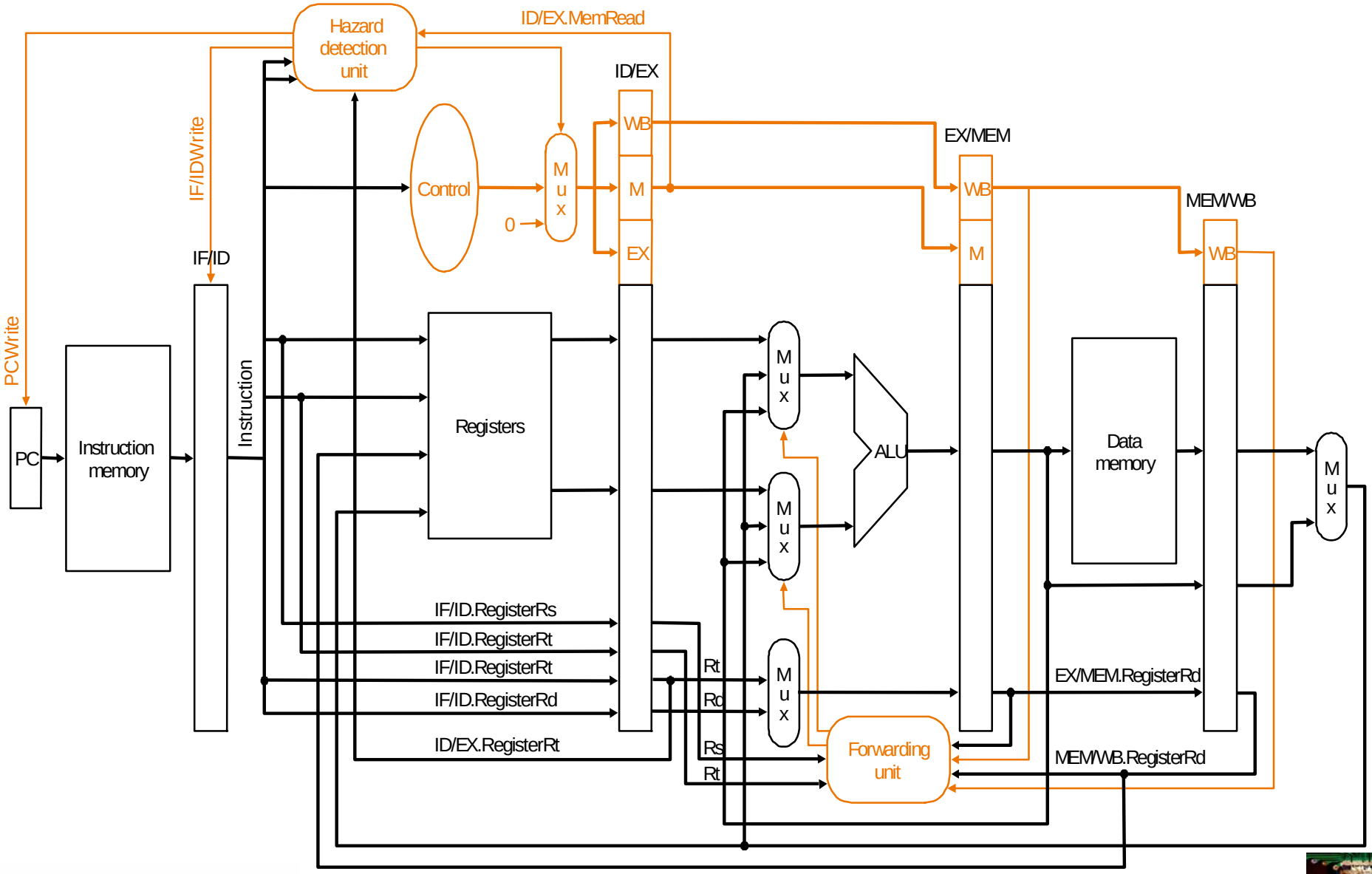
stall the pipeline

-- check if the destination register of the load matches either source register of the instruction in the ID stage.

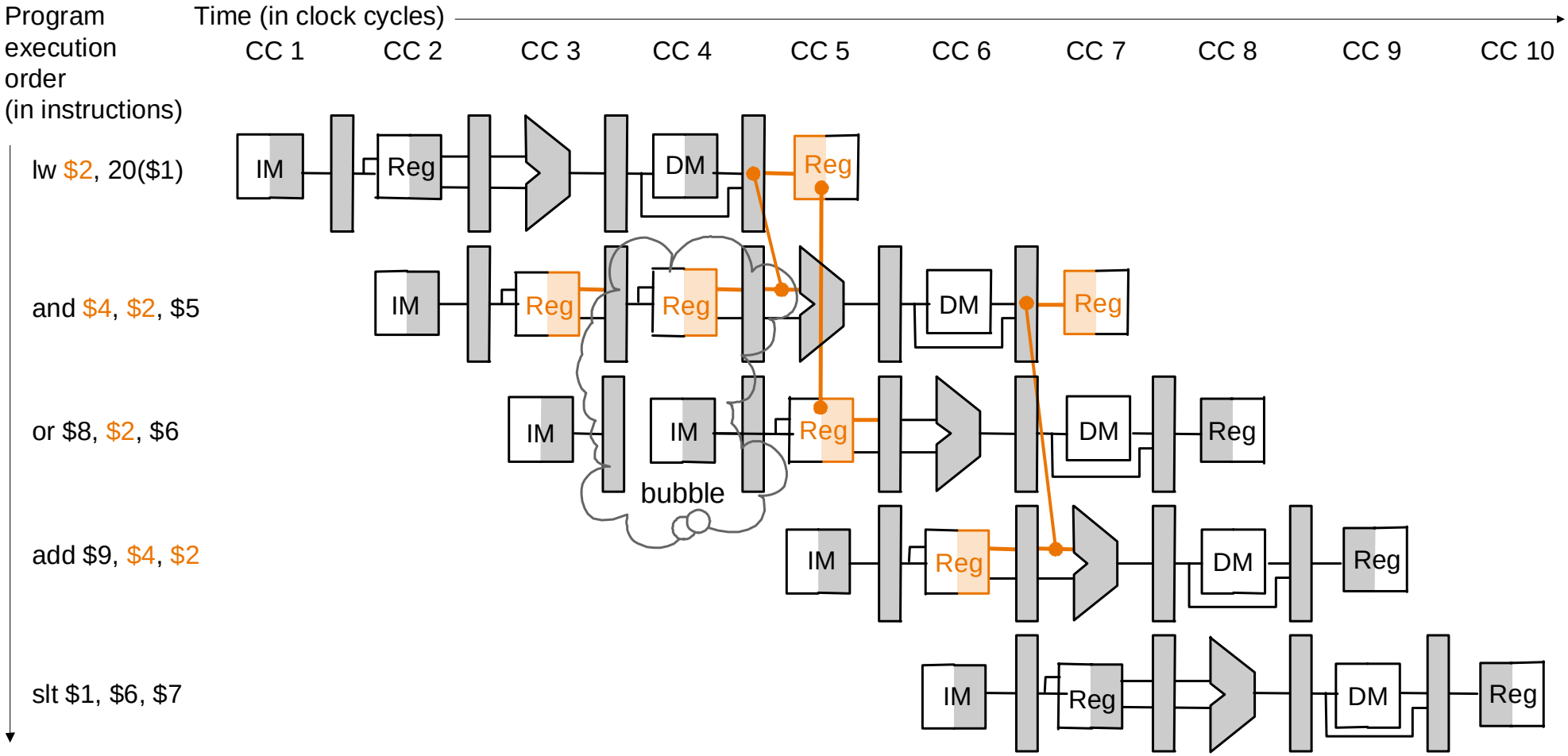


Hazard Detection Unit

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



Stall Insertion



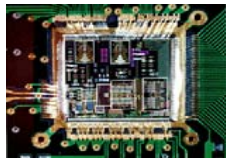
- Since the dependencies go forward in time, there are no data hazards!

Another Solution: A Software Solution

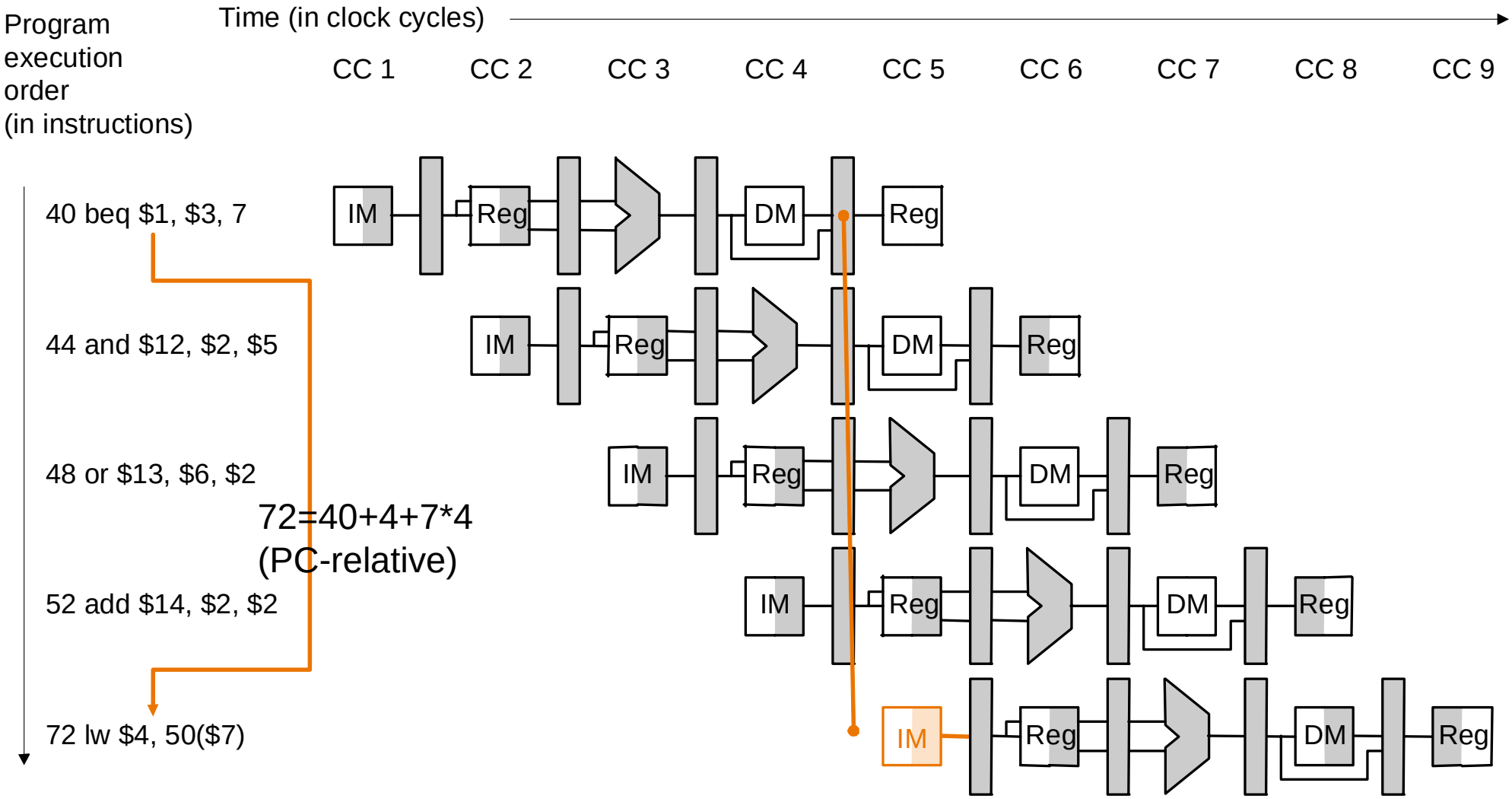
- Have compiler guarantee no hazards.
- Where do we insert the “nops” ?

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

- Problem: this really slows us down!



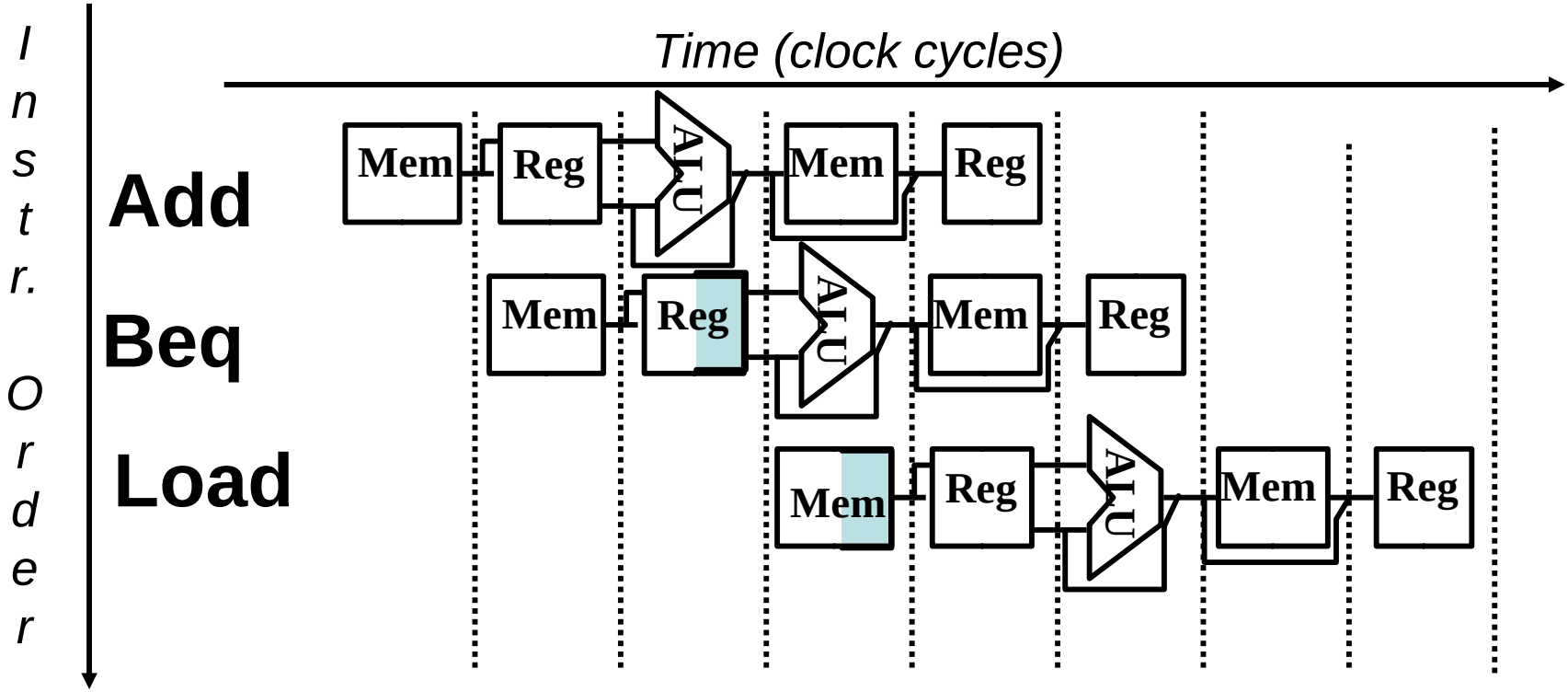
Branch / Control Hazards



- Control hazards are also known as branch hazards.
- Numbers to the instruction are addresses of the instructions.
- Branch instruction decides only in MEM stage (CC4).

Control Hazards – Solution I (Stall)

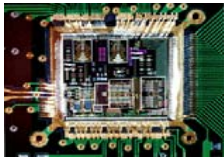
- Stall: wait until decision is clear
  - Its possible to move up decision to 2nd stage by adding hardware to check registers as being read.



- Impact: 2 clock cycles per branch instruction => slow

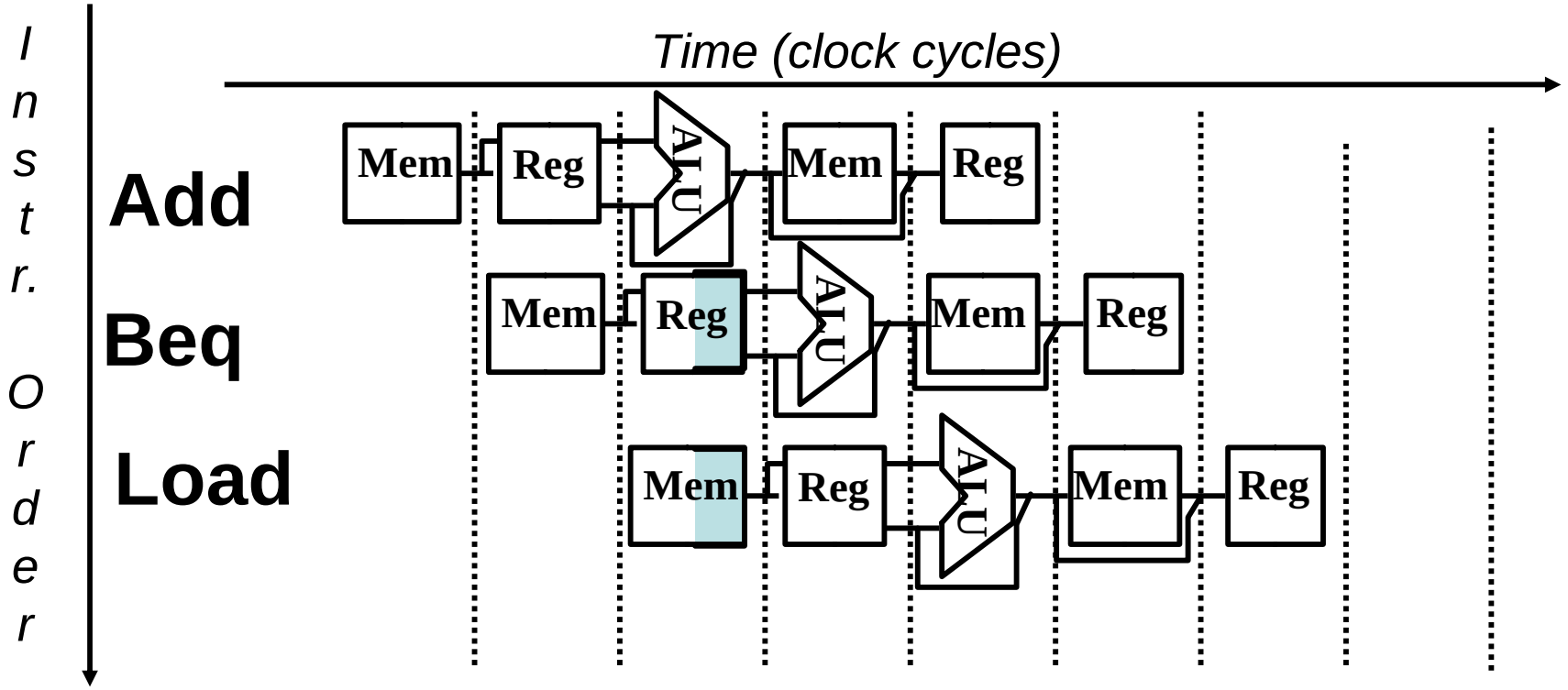
Control Hazards - Solution II (Branch Prediction)

- Branch stalling is too slow.
- Assume that the branch will not be taken and thus continue execution down the sequential instruction stream.
- If branch is taken,
  - The instructions that are being fetched and decoded must be discarded.
  - Execution must continue at the target.
- If branches are not taken half the time, and if it costs little to discard the instructions, then this solution halves the cost of control hazards!
- To discard instructions, we change the original control values to 0s (Just as in Branch Stall case)
- But there is more it: we need to flush instructions in IF, ID, and EX stages of the pipeline!



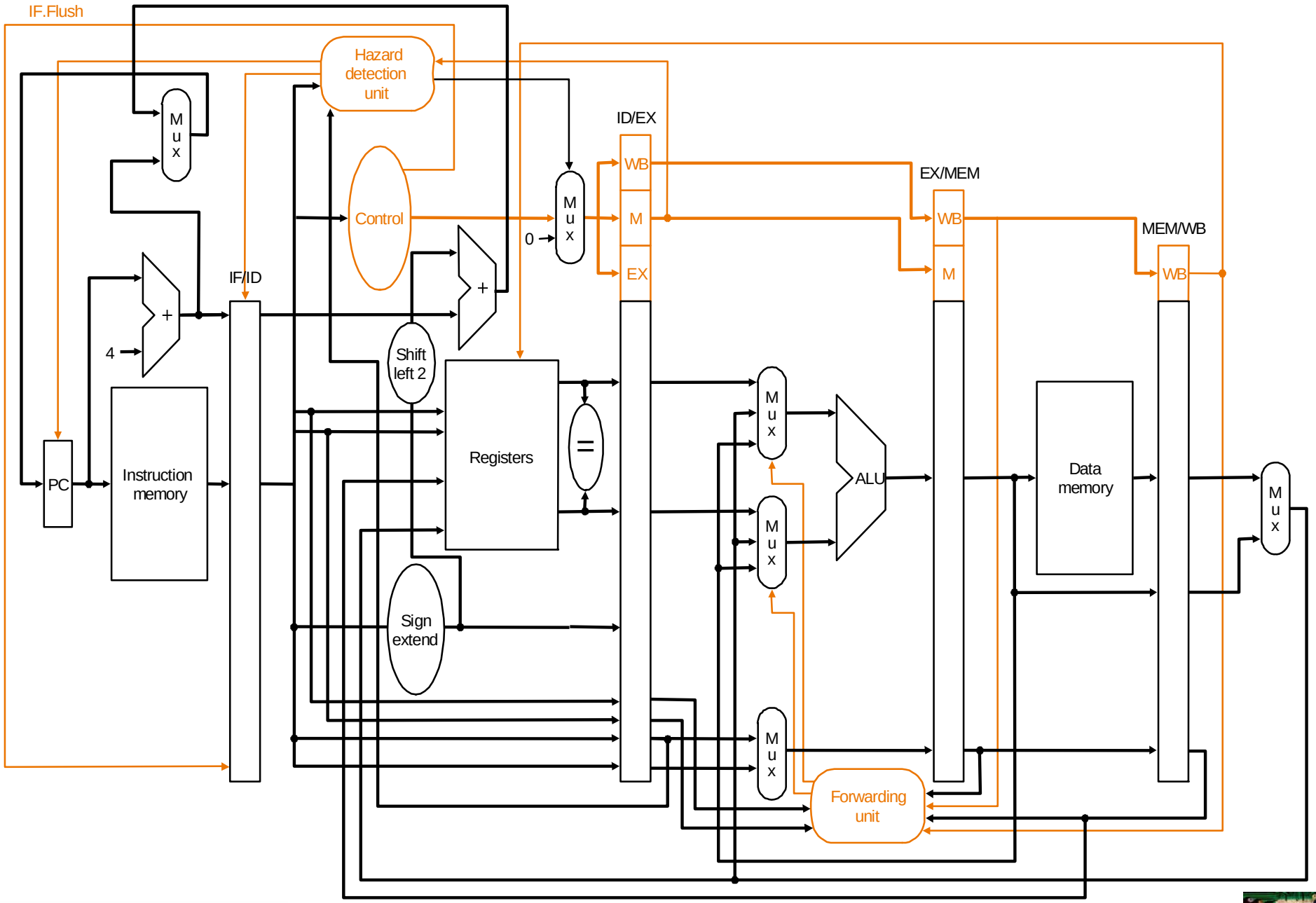
Control Hazards - Solution II (Branch Prediction)

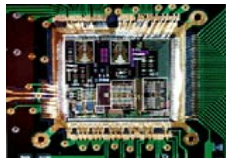
- Predict: guess one direction then back up if wrong
  - Predict not taken



Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)
More dynamic scheme: history of 1 branch (- 90%)

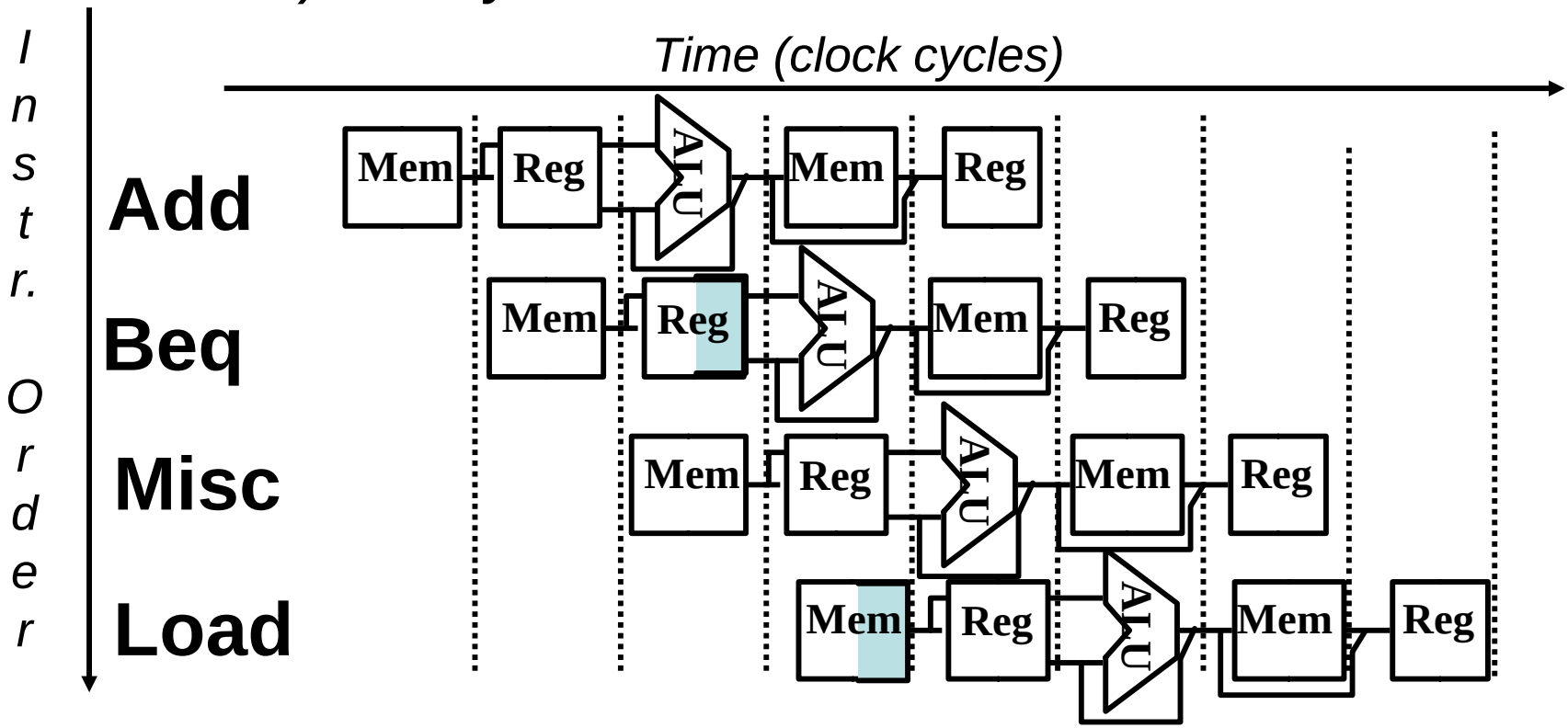
Flushing Instructions





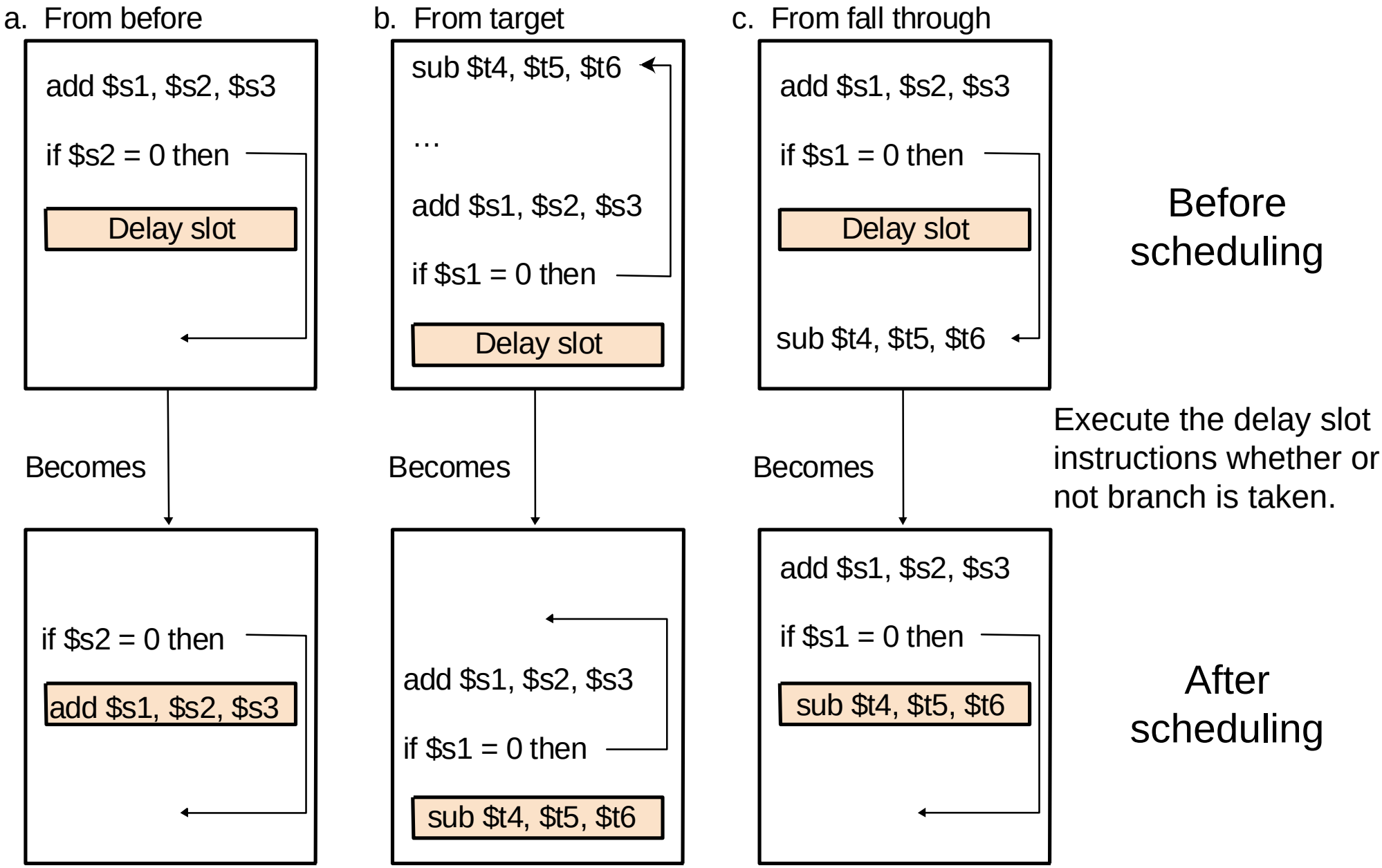
Control Hazards – Solution III (Delayed Branch)

- Redefine branch behavior (takes place after next instruction) “delayed branch”.

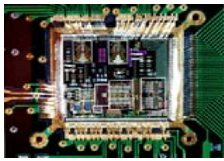


- Impact: 0 clock cycles per branch instruction if we can find instruction to put in “slot” (50% of time).
- As launching more instruction per clock cycle, less useful.

Scheduling the branch delay slot



NOTE: Refer Fig A.14 page-A-24 of Quantitative book for more details.



Performance of Pipeline with Branch Schemes

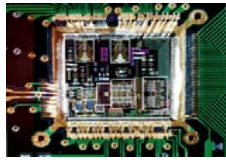
- Speed up from pipelining =

$$\frac{PipelineDepth}{1 + PipelineStallCyclesFromBranches}$$

- Speed up from pipelining =

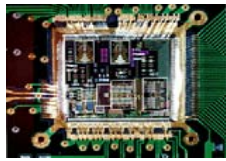
$$\frac{PipelineDepth}{1 + BranchFrequency \times BranchPenalty}$$

Solve Example page-A25 of Quantitative book.



Dynamic Branch Prediction

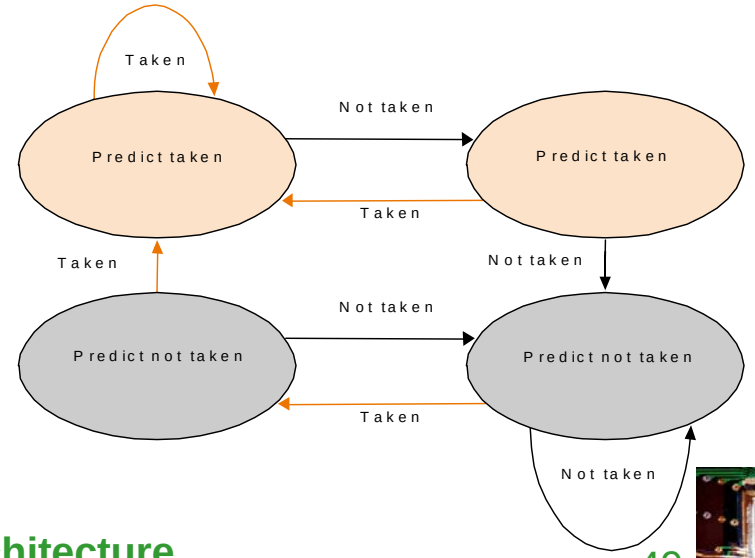
- Assuming always that a branch is not taken is known as static branch prediction. We can do better than this!
- Main Idea:
  - Look up the address of the instruction to see if a branch was taken the last time this instruction was executed.
  - If so, begin fetching new instructions from the same place as the last time!
  - Need: *branch prediction buffer* or *branch history table*
- *Branch* prediction buffer (1-bit prediction scheme)
  - It is a small memory indexed by the lower portion of the address of the branch instruction.
  - The memory contains a bit that says whether the branch was recently taken or not.
  - This may not work all the time!
  - If the prediction is false, then prediction bit is inverted and stored back.



2-bit Prediction Scheme

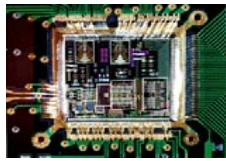
- 1-bit prediction scheme*:*
  - The bit indexed to may not be the right bit i.e., the indexed bit may have been written by a branch instruction whose lower bits match with this branch instruction.
  - Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken!

    Example: *Consider a loop branch that branches nine times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?*
- 2-bit Prediction Scheme: *A branch that strongly favors taken or not taken (typical behavior) will be mispredicted only once. The two bits are used to encode the four states of the system.*



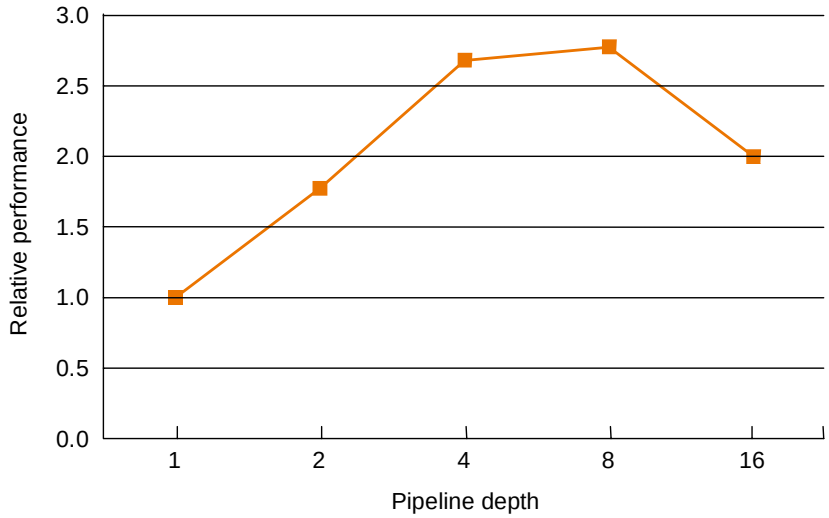
Dynamic Scheduling

- 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



Pipelining – Fallacies & Pitfalls

- Pipelining is easy.
- Pipelining ideas can be implemented independent of technology.
- Failure to consider instruction set design can adversely impact pipelining
  - Widely variable instruction lengths and running times can lead to imbalance among pipeline stages; complicate hazard detection.
  - Sophisticated addressing modes.
- Increasing the depth of pipelining always increases performance (see the table below from S.R. Kunkel & J.E. Smith, "Optimal pipelining in super computers," in *Proc 13th Symp. On Computer Architecture* (June 1986), pages 404-414.)



Pipelining Summary

- Pipelining doesn't help latency of single task, it helps throughput of entire workload.
- Multiple tasks operating simultaneously using different resources.
- Potential speedup = Number pipe stages.
- Pipeline rate limited by slowest pipeline stage.
- Unbalanced lengths of pipe stages reduces speedup.
- Time to "fill" pipeline and time to "drain" it reduces speedup.
- Three types of pipeline hazards: structural, data, and control/branch
- Stalling helps any kind of hazard.
- Data hazard solutions: Stalling, Data forwarding, and Hazard detection
- Control or Branch Hazard solutions: Stalling, Delayed Branching, and Branch Prediction.