Computer Organization and Architecture

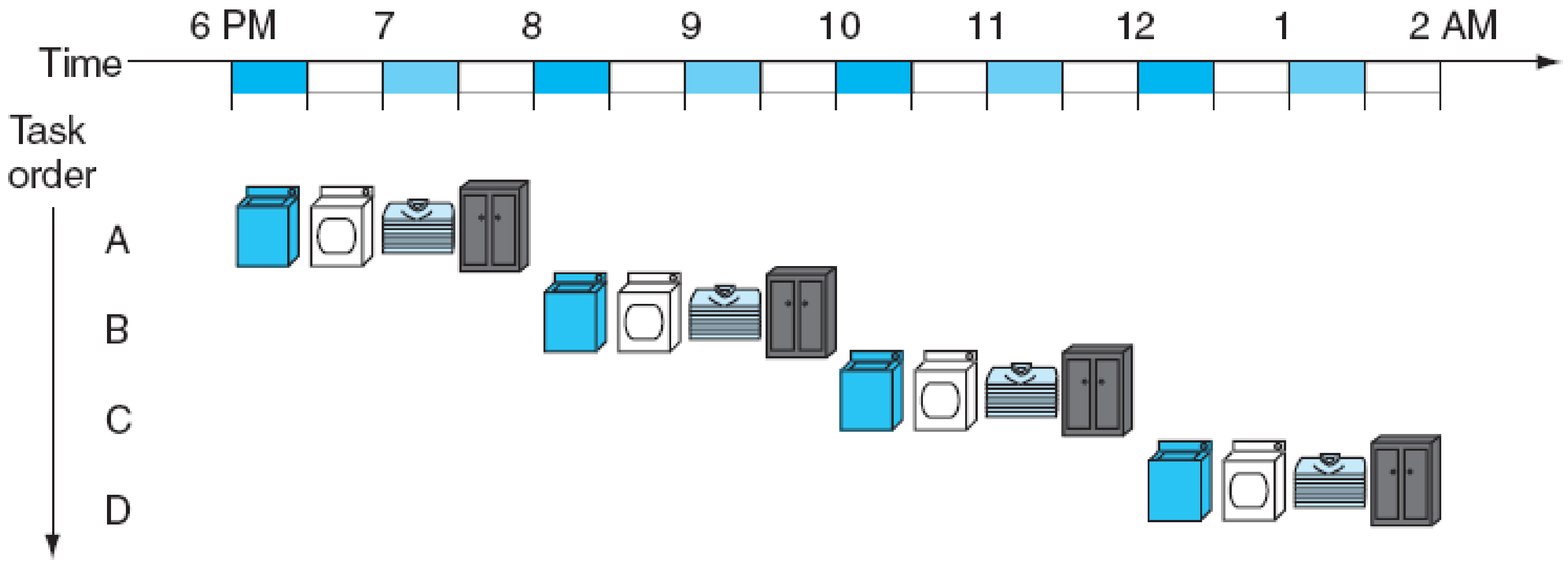
Dr. Muhammad Naeem Awais

Lecture Contents

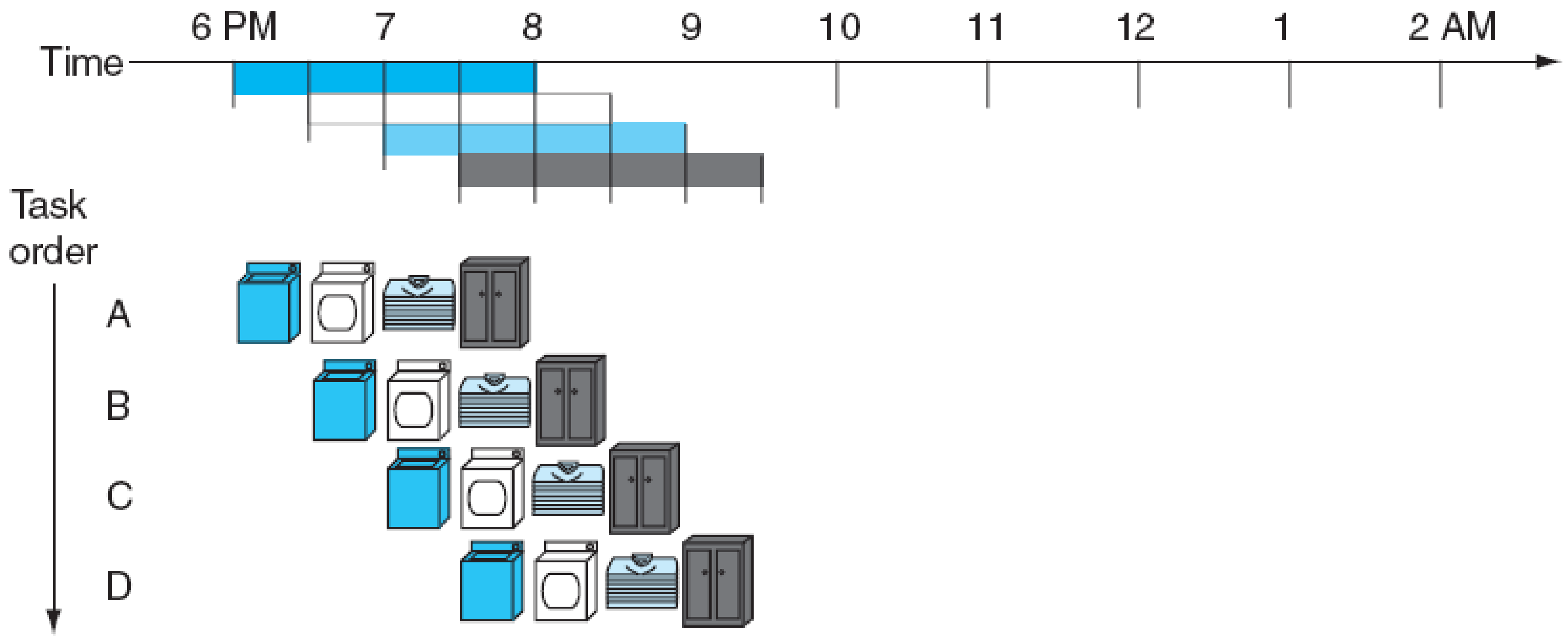
- An overview of pipelining
- MIPS pipelined datapath
- MIPS pipelined control
- Hazards & their elimination
  - Structural Hazards
  - Data Hazards
  - Control Hazards
- Multi-cycle Floating point MIPS pipeline

Introduction

- An implementation technique where multiple instructions are overlapped in execution
- It is a key factor in making processors fast



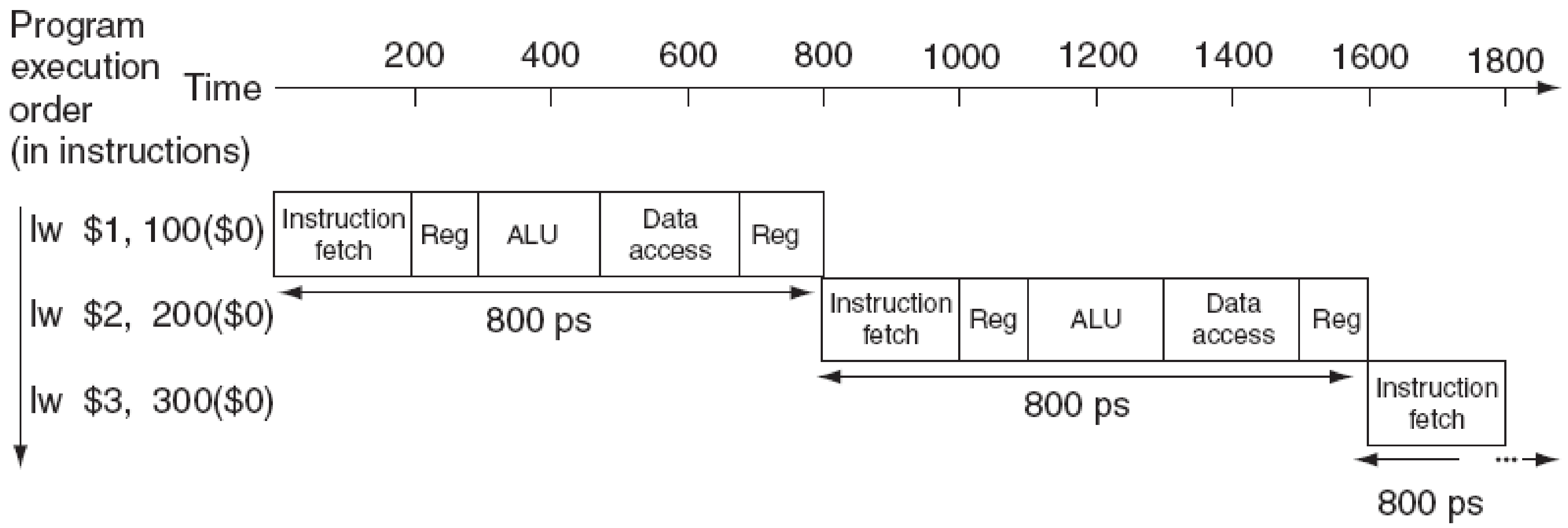
Pipelining basics



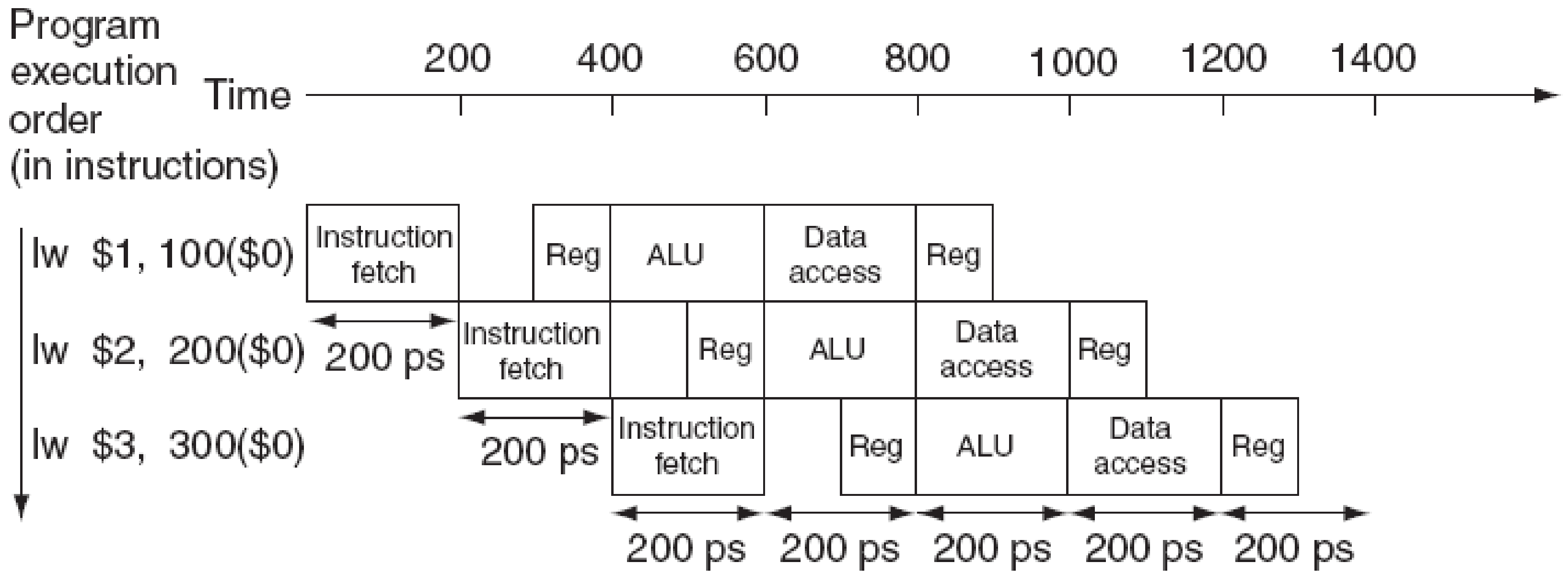
- As long as we have separate resources for each stage, we can pipeline the task
- Speedup due to pipelining is equal to the number of stages in the pipeline
- 20 loads of sequential laundry take 20 times the single load time.
- While 20 loads of pipeline take about 5 times the single load time. So the speedup is four times
- How much is it faster in the above case?

Single cycle versus pipelined performance

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps



Single cycle versus pipelined performance



- Pipeline stages are not perfectly balanced. <u>Slowest stage determines the pipeline stage time.</u> So in this case speedup is less than number of stages in the pipeline
- If the stages had been perfectly balanced, then speedup would have been equal to the *pipeline depth.*
- check yourself for 1000 instruction, speedup is well below 5 (number of pipeline stages)

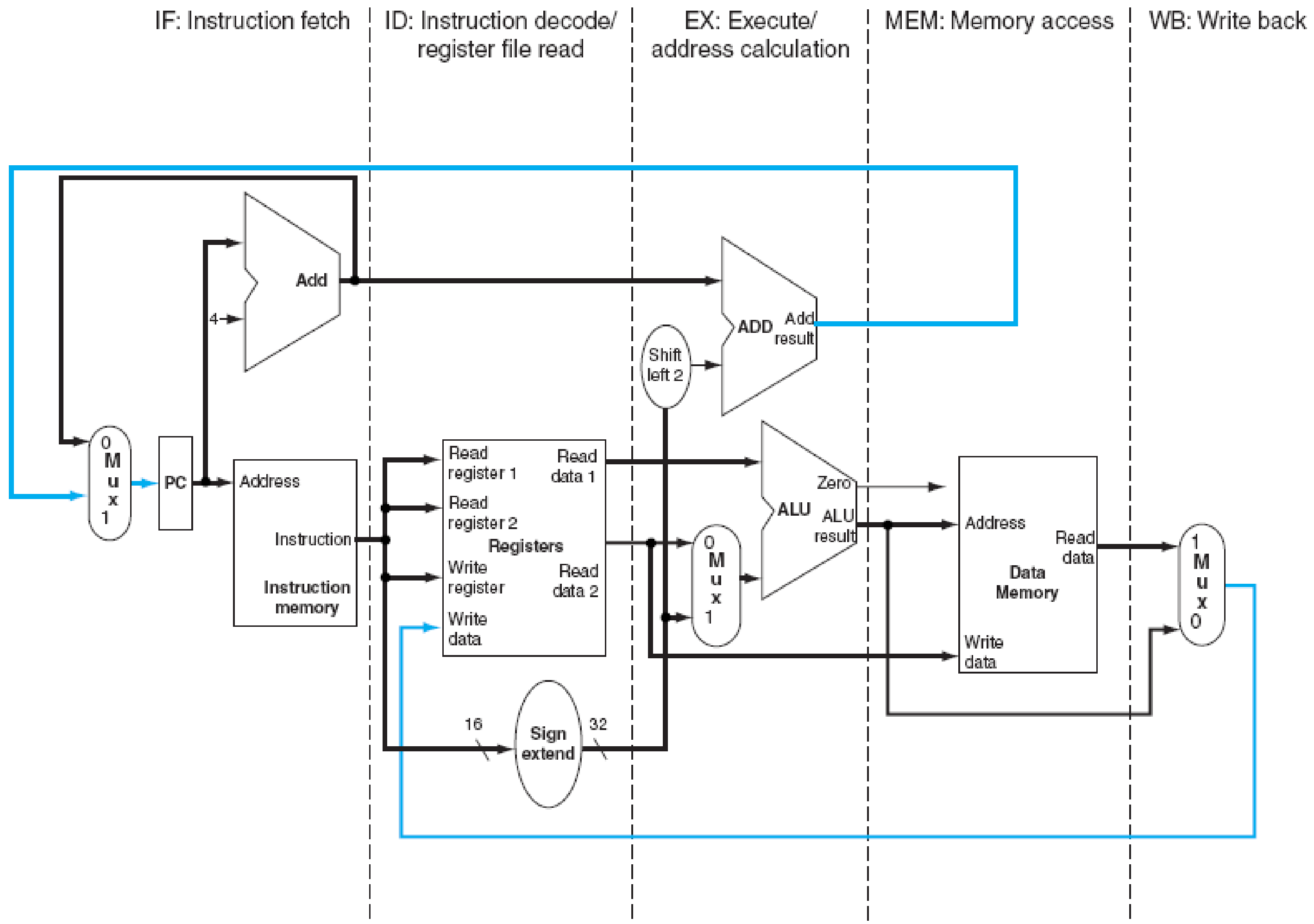
Pipelining overview summary

- Pipelining is a technique that exploits parallelism among instructions
- Principles of previous example are applied in pipelined processors
- Pipelining does not reduce execution time of individual instructions
- It only increases throughput
- Pipelining performance is affected by pipelining hazards
  - Structural hazard
  - Data hazard
  - Control hazard

A pipelined datapath

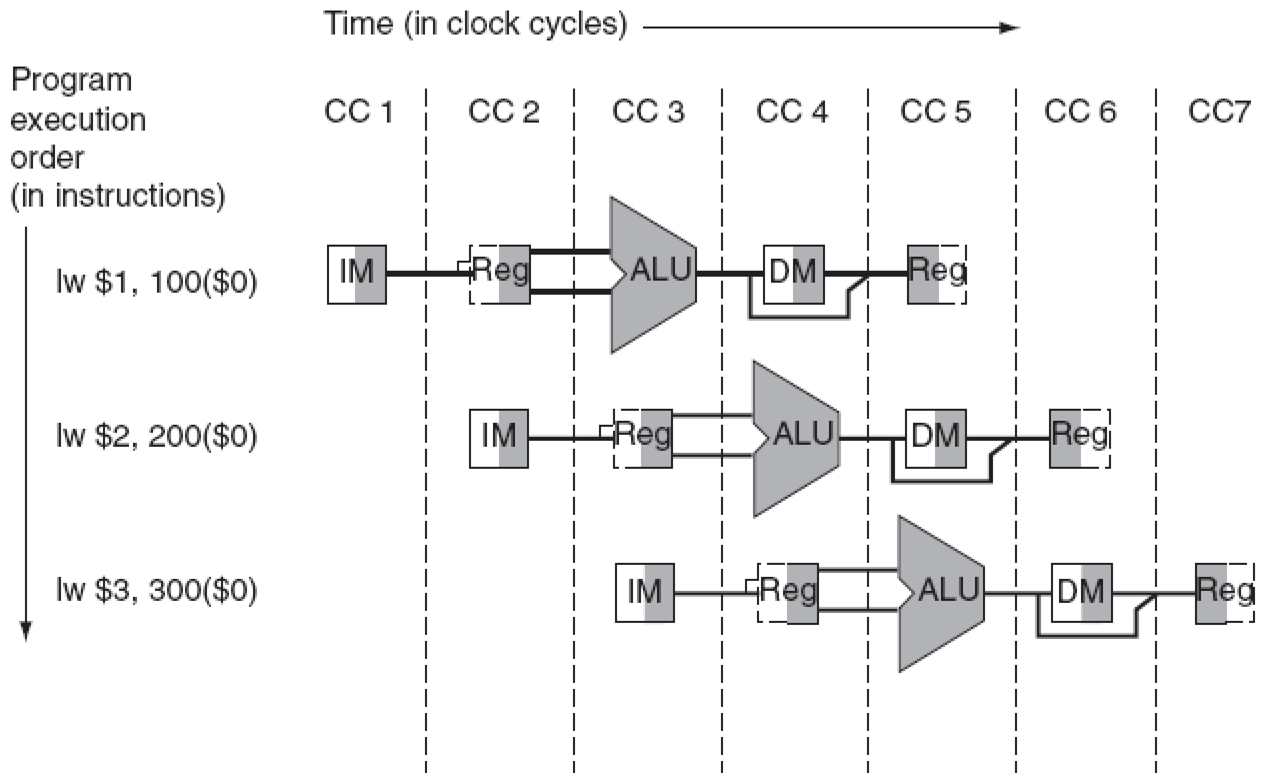
- In MIPS an instruction execution is divided into 5 stages
- So MIPS will make a five stage pipeline
- At a given cycle up to 5 instructions will be in execution during any cycle
- Five stages of pipeline are
  - IF: Instruction fetch
  - ID: Instruction decode and register file read
  - EX: Execution or address calculation
  - MEM: Data memory access
  - WB: Write back

A pipelined datapath



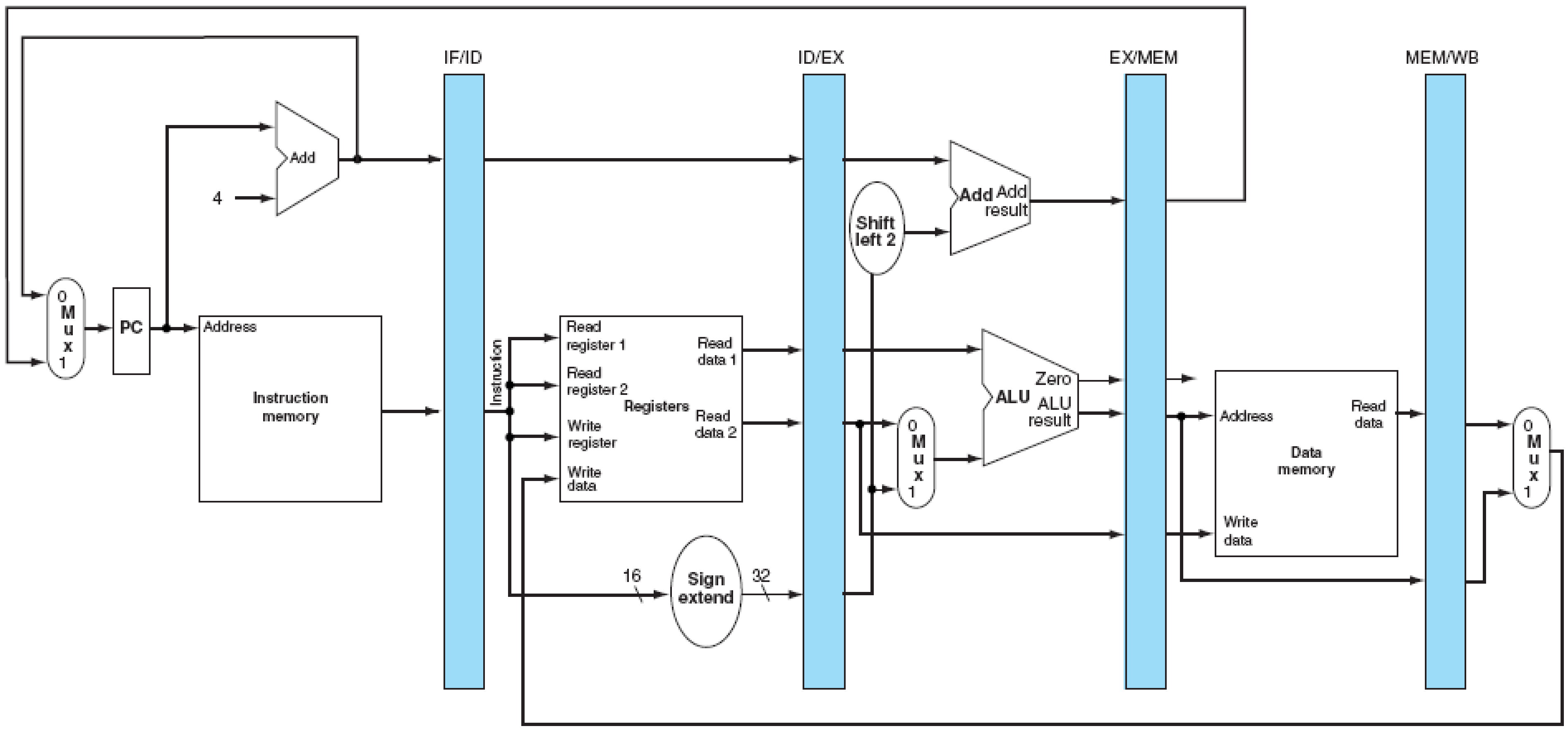
- Data flows from left to right
- There are, however, two exceptions
- These exception can cause data or control hazards

A pipelined datapath



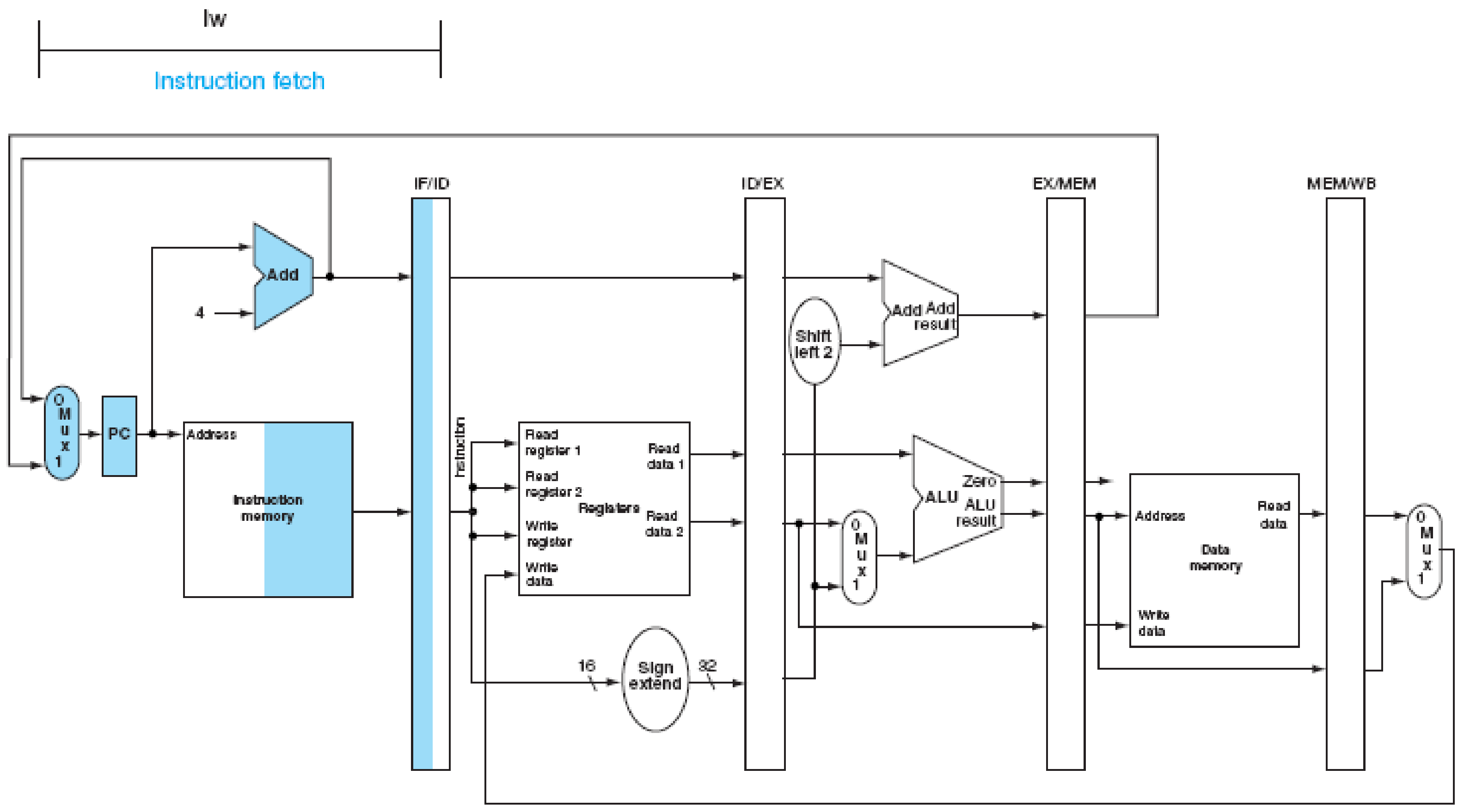
- Just for the sake of understanding, we pretend that every instruction has its own datapath
- Datapath components are used only once during execution of an instruction
- In order to have shared datapath, registers need to be added
- Registers are added between stages
- Important: Register files are written in first half of cycle and read in second half of cycle

Pipelined datapath with pipelined registers



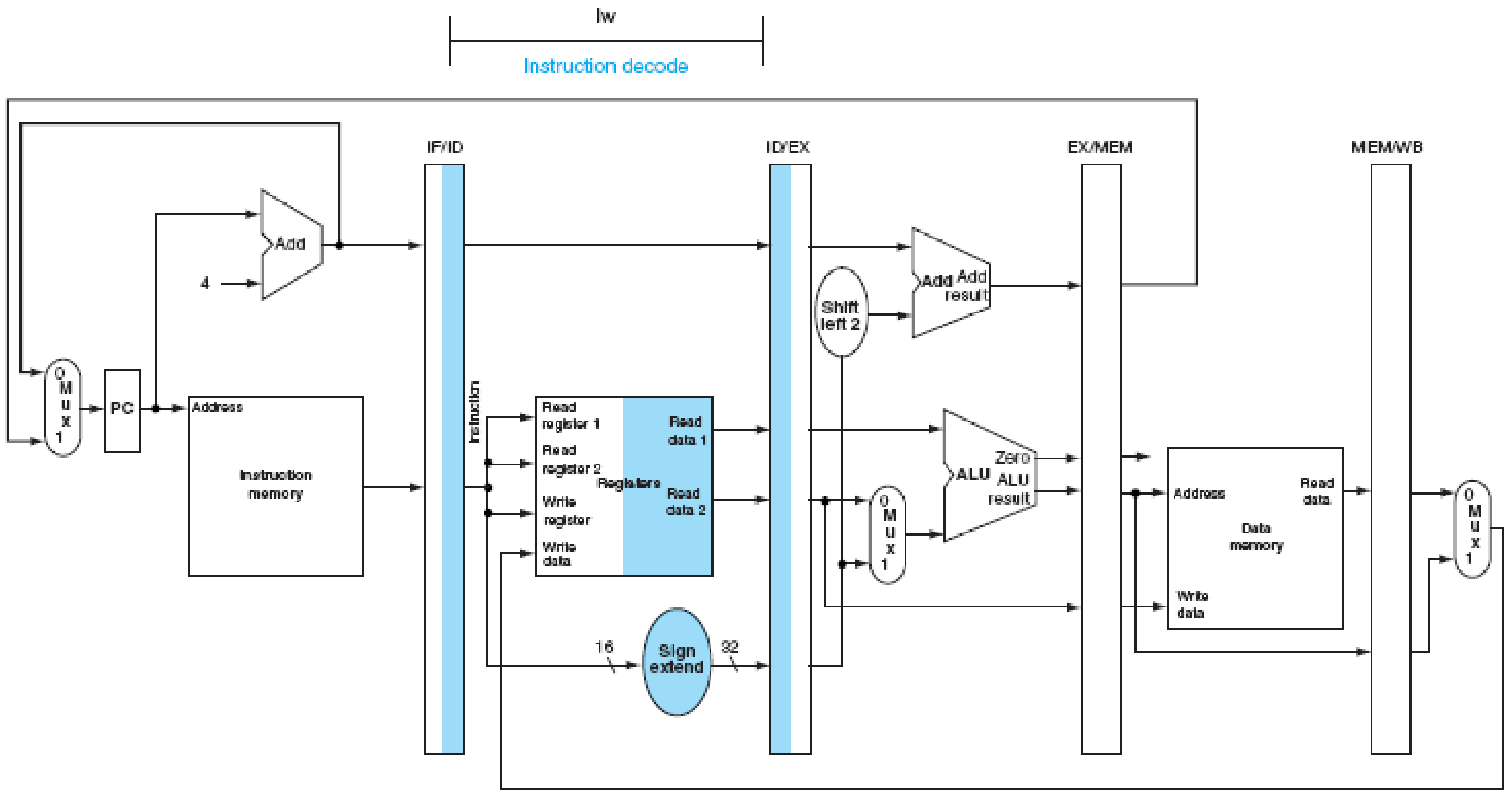
- Pipeline registers are used to hold values between the pipeline stages
- They are labeled by the stages that they separate
- Registers must be wide enough to hold all the necessary data
- Register sizes for now are 64, 128, 97, 64 bits

Five stages of pipeline for lw instruction



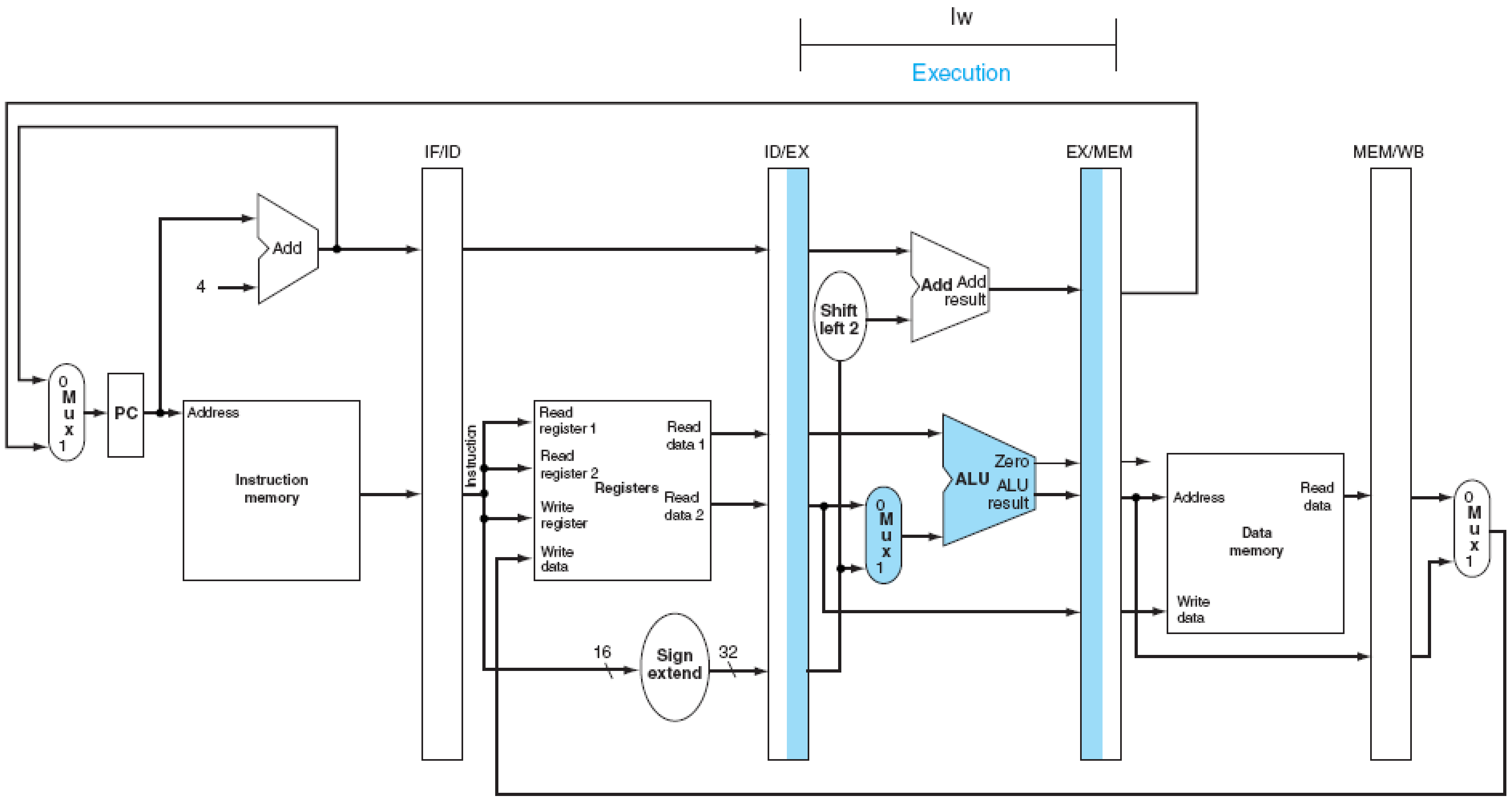
- Instruction read from memory
- IF/ID register being written
- PC is incremented by 4 and written into IF/ID

Five stages of pipeline for lw instruction



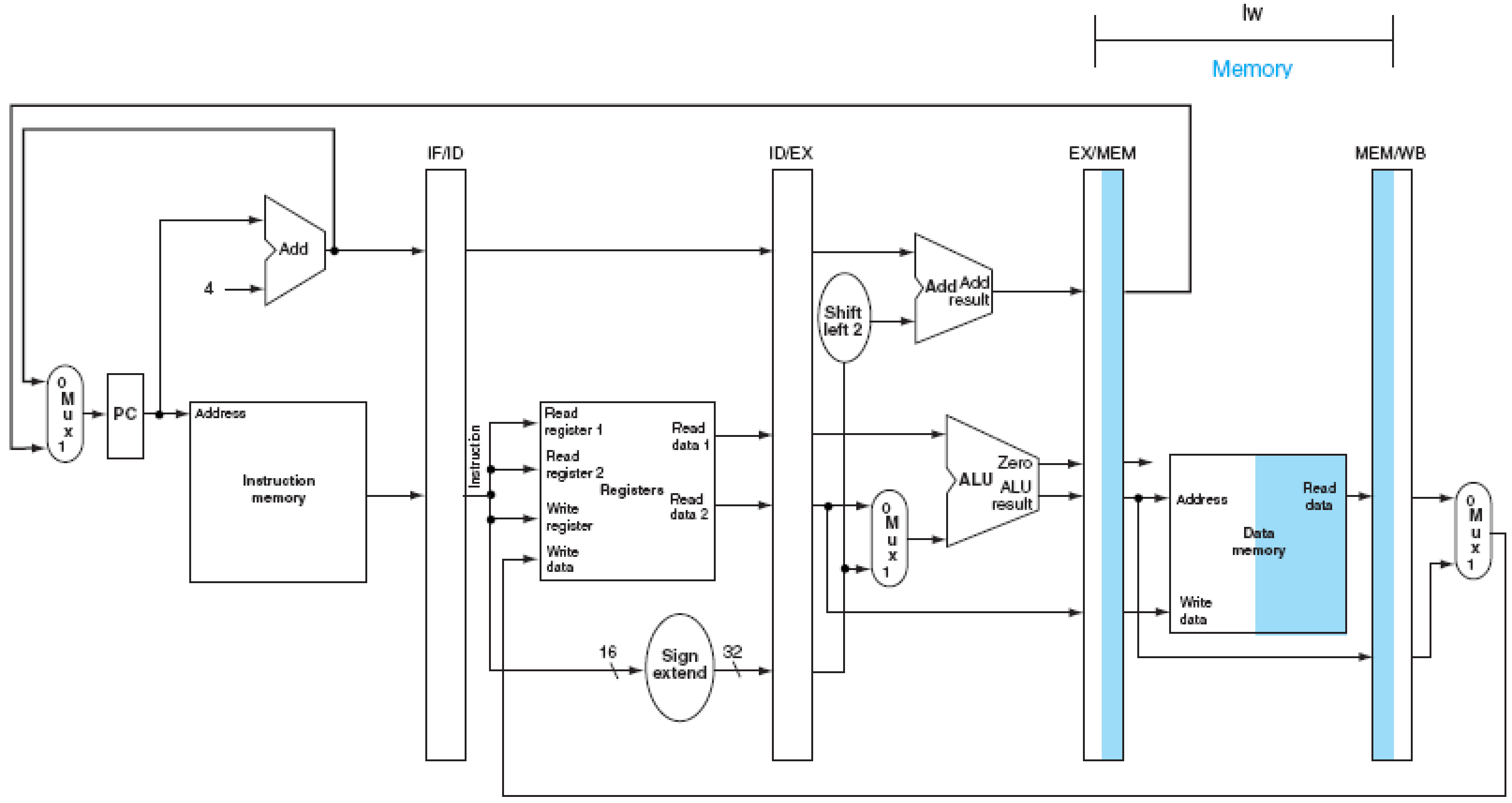
- IF/ID register being read
- Register files are read
- Sign extension is performed
- ID/EX register being written

Five stages of pipeline for lw instruction



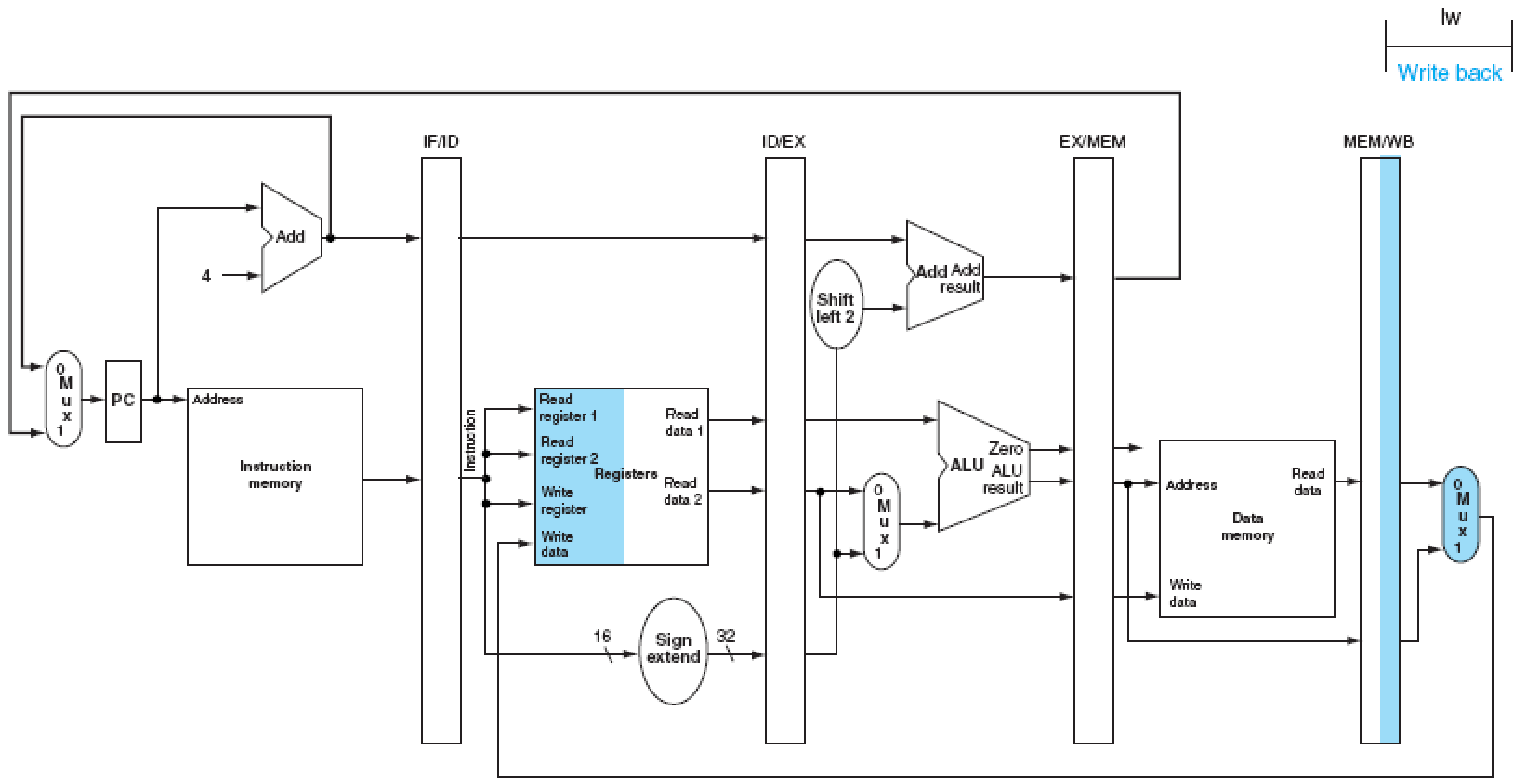
- ID/EX being read
- Address calculation being performed using ALU
- Result placed in EX/MEM register

Five stages of pipeline for lw instruction



- Address taken from EX/MEM register
- Memory being read
- MEM/WB register being written

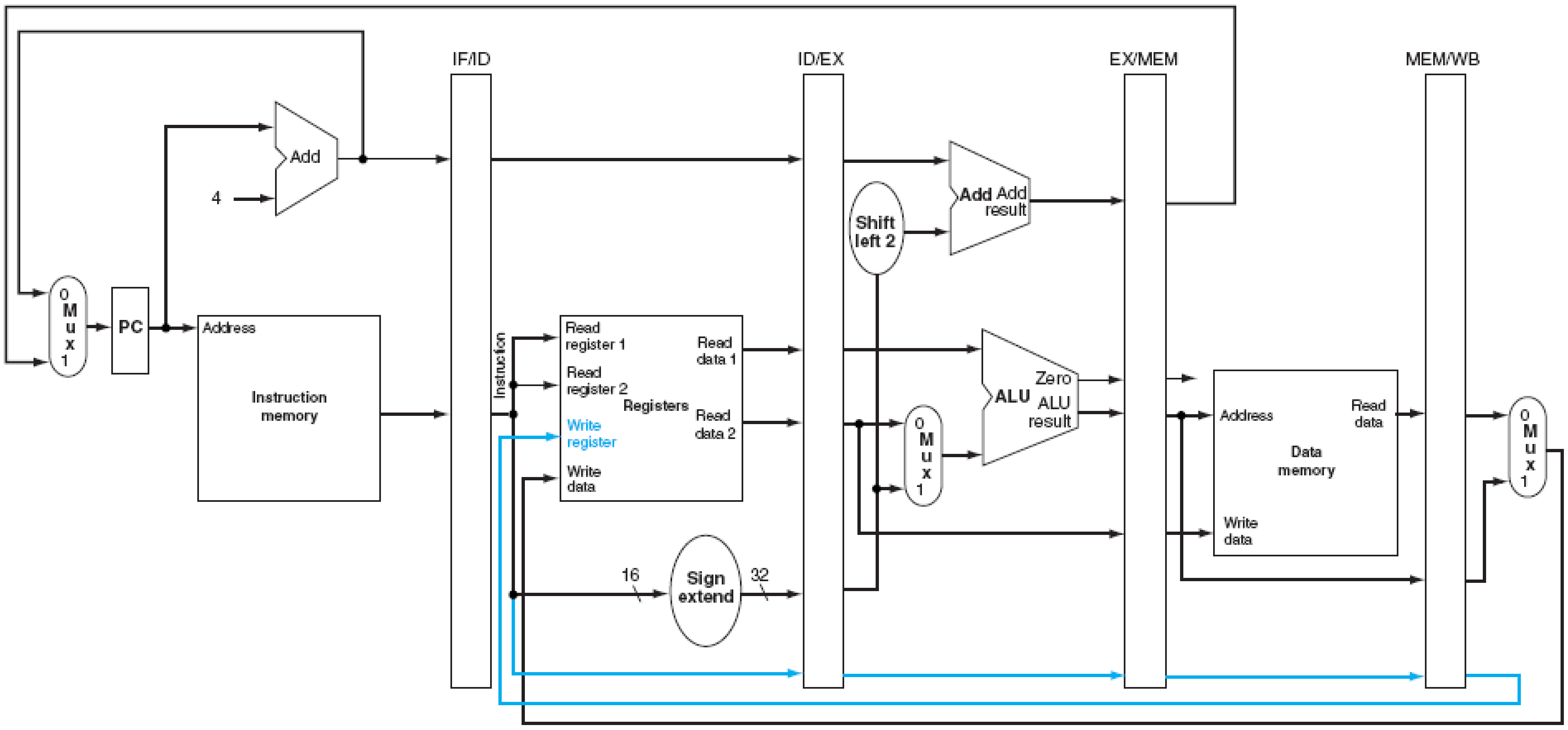
Five stages of pipeline for lw instruction



- MEM/WB register being read
- Register file being written

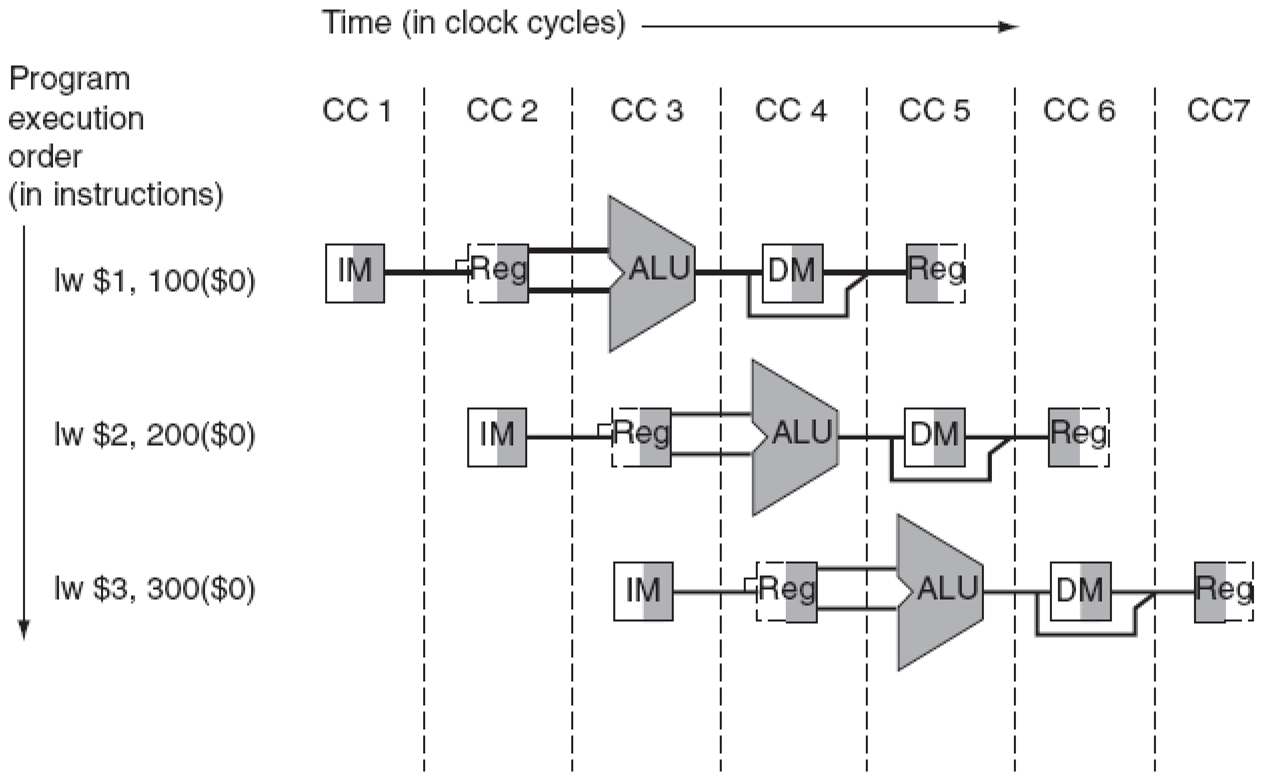
Presence of a bug and its rectification

- In the final stage: where does the write register number comes from??



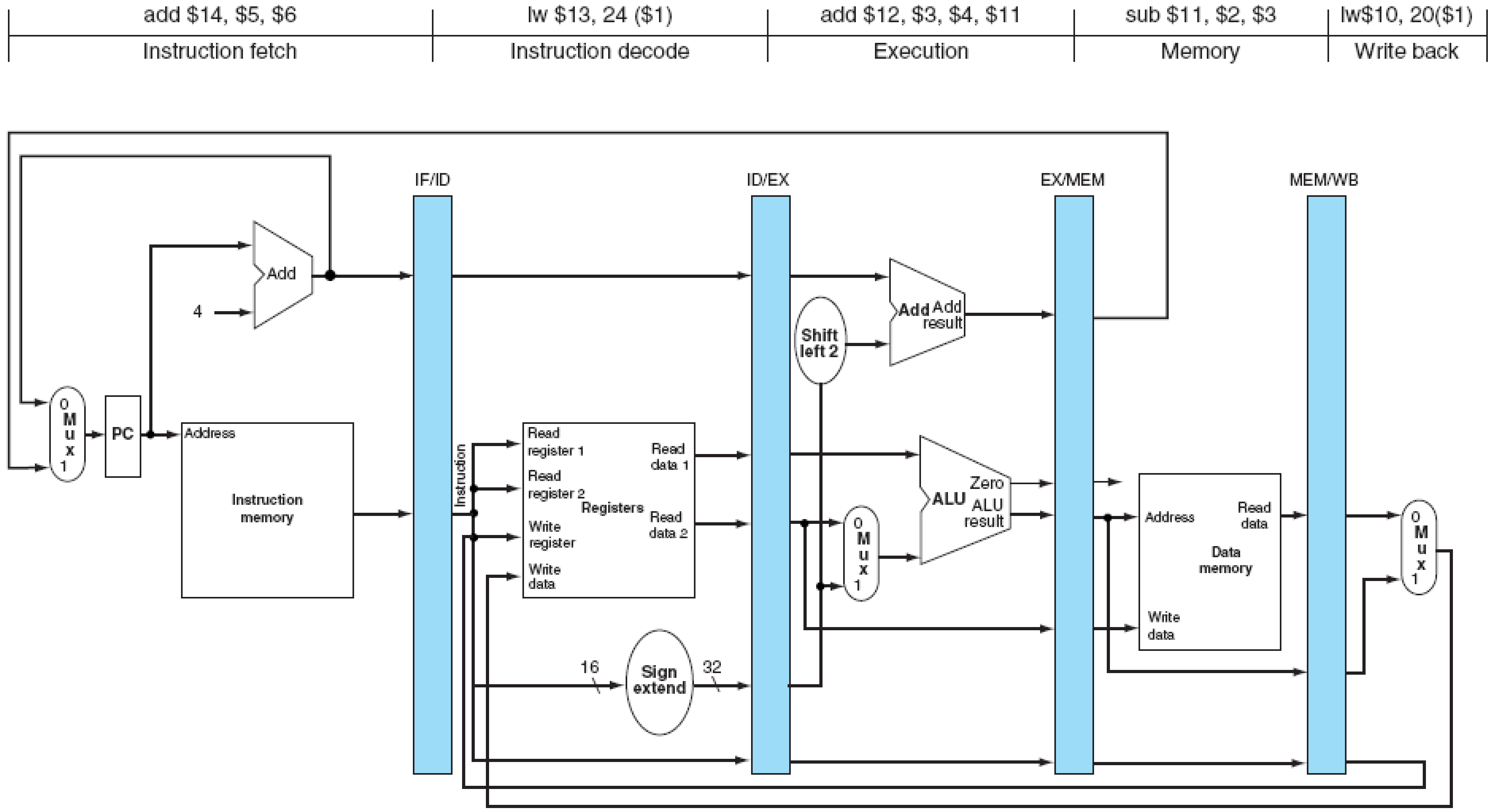
Graphical representation of pipelining

- Multiple clock cycle pipeline diagram
  - Example



Graphical representation of pipelining

- Single clock cycle pipeline diagram
  - Example



Pipelined control identification

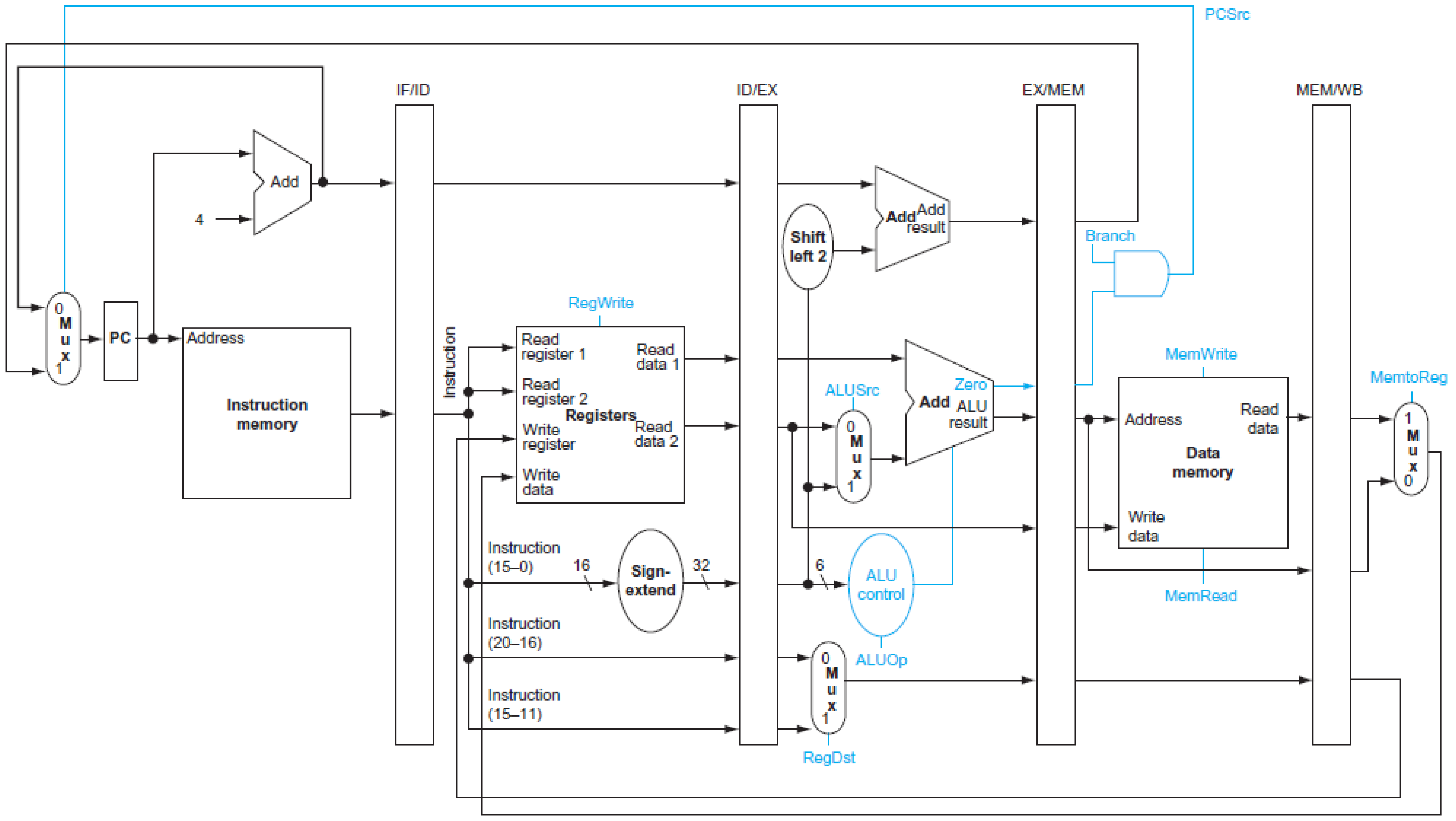


FIGURE 4.46 The pipelined datapath of Figure 4.41 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

Pipelined control

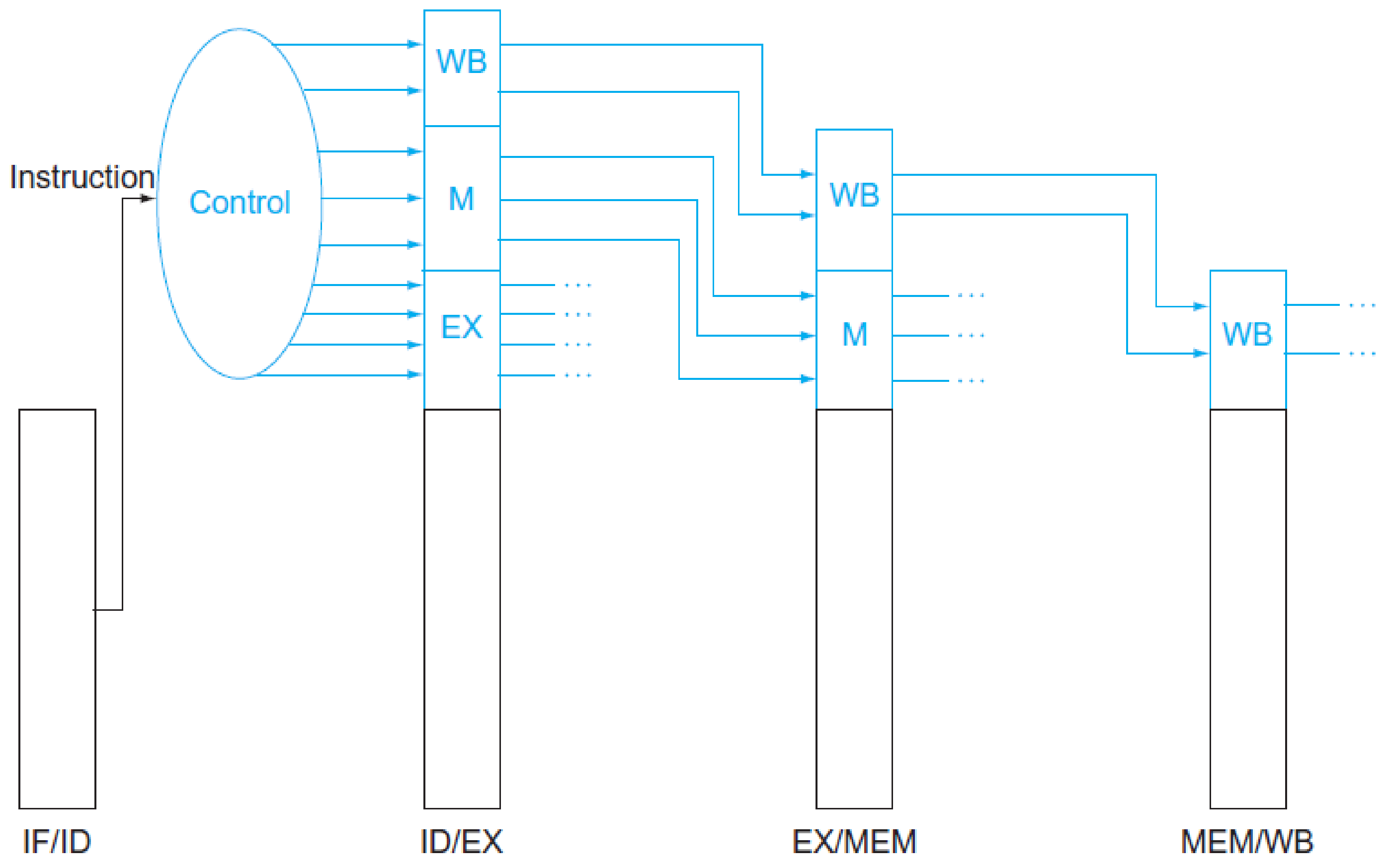


FIGURE 4.50 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

Cont...Pipelined control

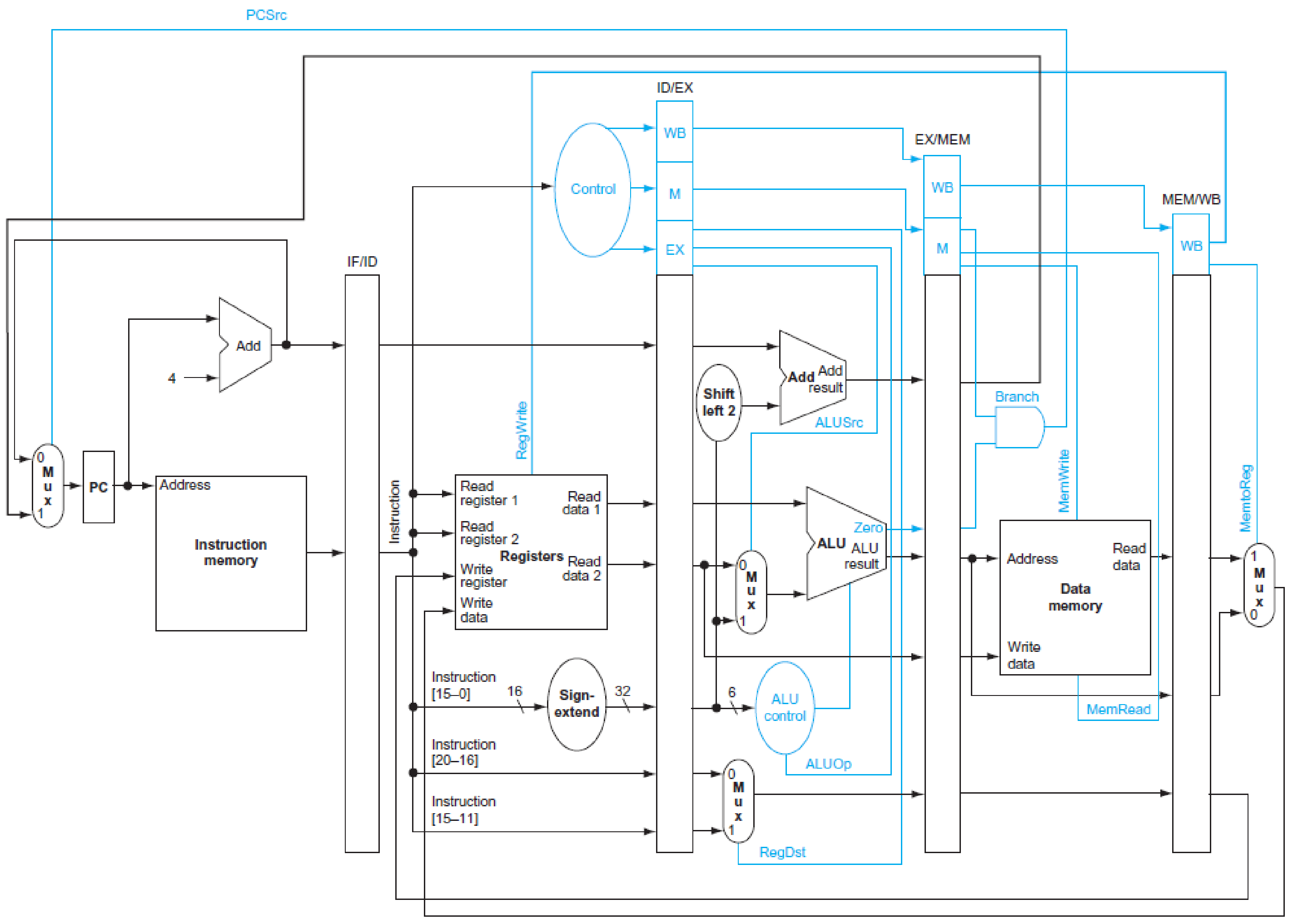


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Pipeline Hazards

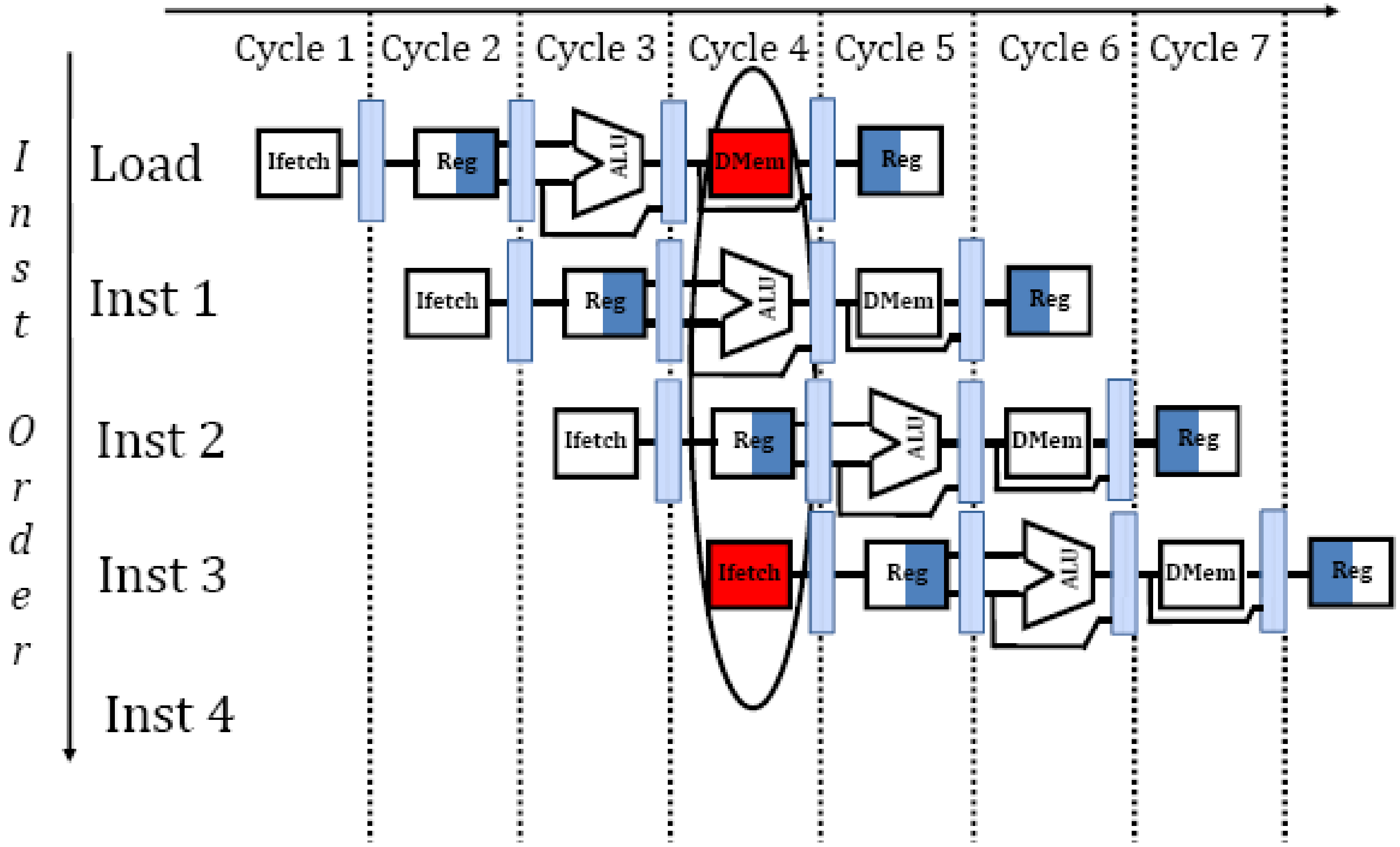
- A simple pipeline would work just fine if
  - All the instructions were independent of each other
  - In reality, it is not the case
- Hazards occur in a pipelined processor when the execution of next instruction is prevented in its designated clock cycle
  - **Structural hazard:** arise from resource conflict. Attempt to use same hardware to do two different things at once
  - **Data hazards:** arise when an instruction depends on the results of a previous instruction which is still in pipeline
  - **Control hazard:** arise from pipelining of branches that change program counter

Structural Hazard

- Occur when some combination of instructions cannot be accommodated because of the resource conflicts

- Examples
  - When some functional unit is not fully pipelined
  - If a resource has not been duplicated enough, such as reg-file has only one read port but pipeline needs two reads in one cycle
  - A single shared memory for instructions and data

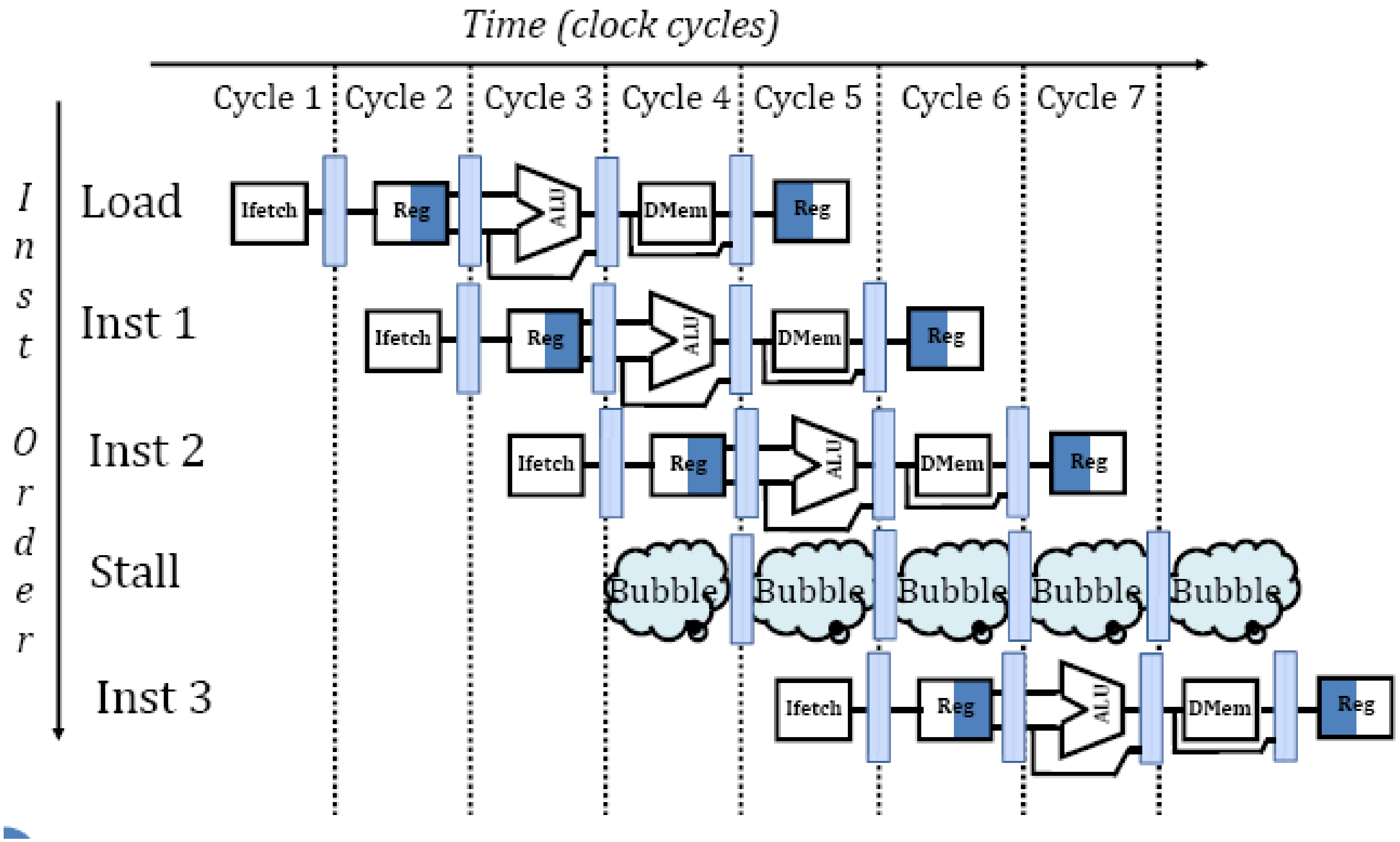
Structural Hazard#1: Memory



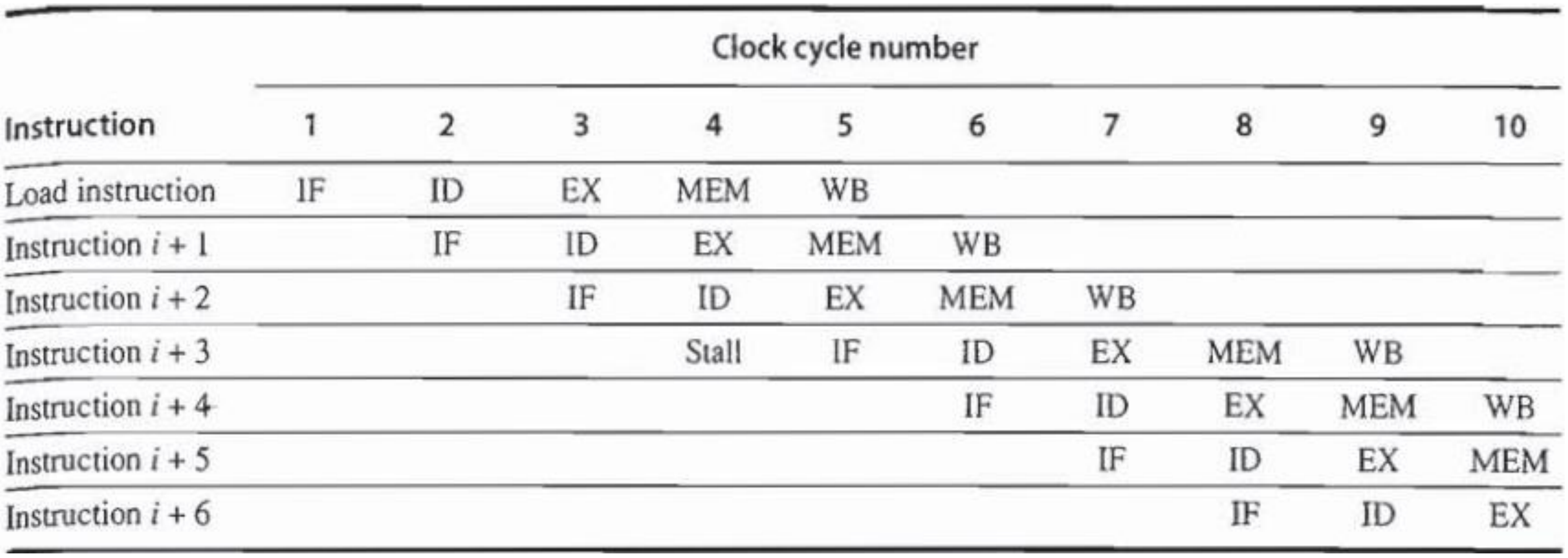
Resolving Structural Hazard

- Eliminate the use of same hardware for two different things at the same time
- **Solution 1:** Wait
  - Must detect the hazard
  - Must have mechanism to stall
- **Solution 2:** Duplicate hardware
  - Multiple such units will help both instructions to progress

Solution 1: Wait



Structural hazard

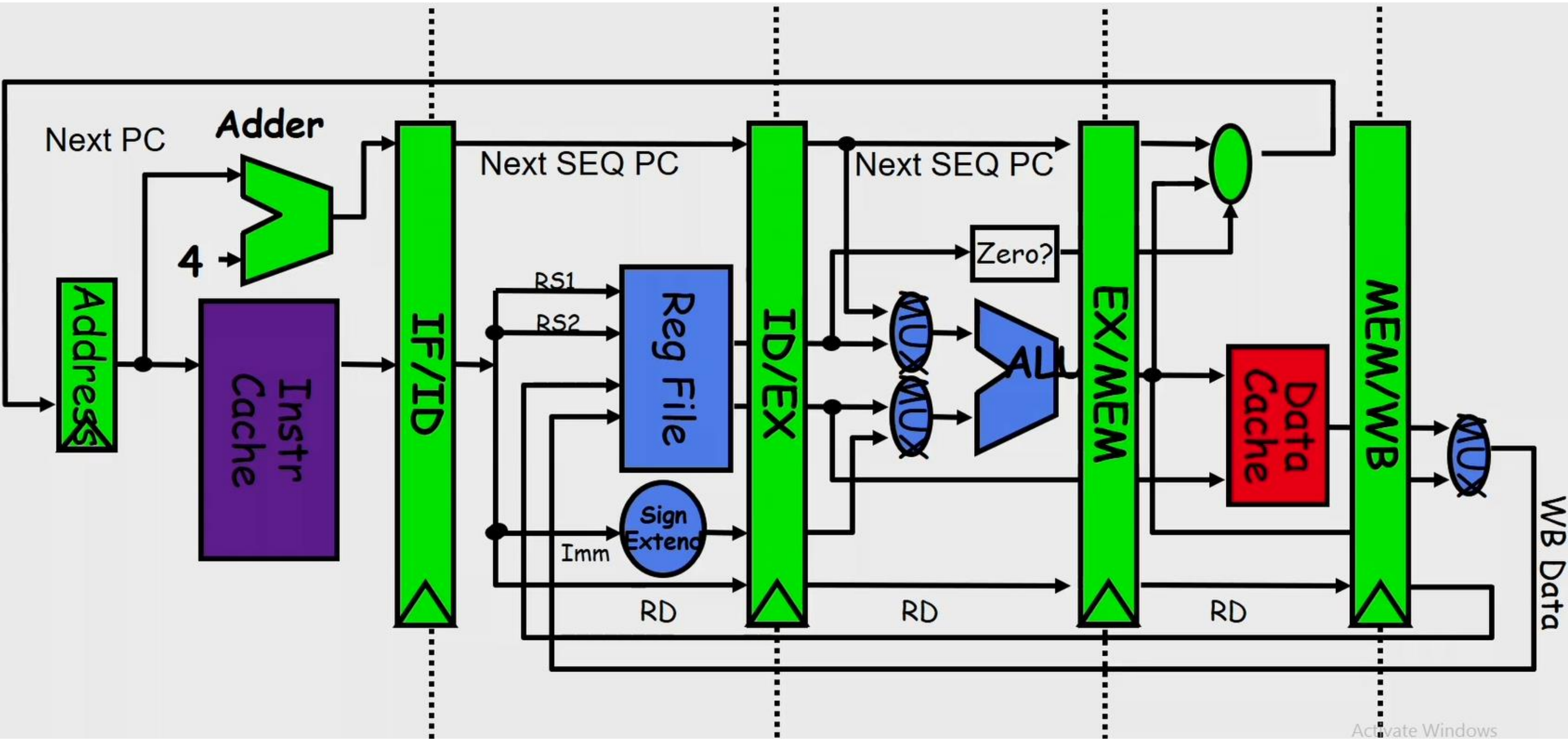


	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction *i* + 1		IF	ID	EX	MEM	WB				
Instruction *i* + 2			IF	ID	EX	MEM	WB			
Instruction *i* + 3				Stall	IF	ID	EX	MEM	WB	
Instruction *i* + 4						IF	ID	EX	MEM	WB
Instruction *i* + 5							IF	ID	EX	MEM
Instruction *i* + 6								IF	ID	EX

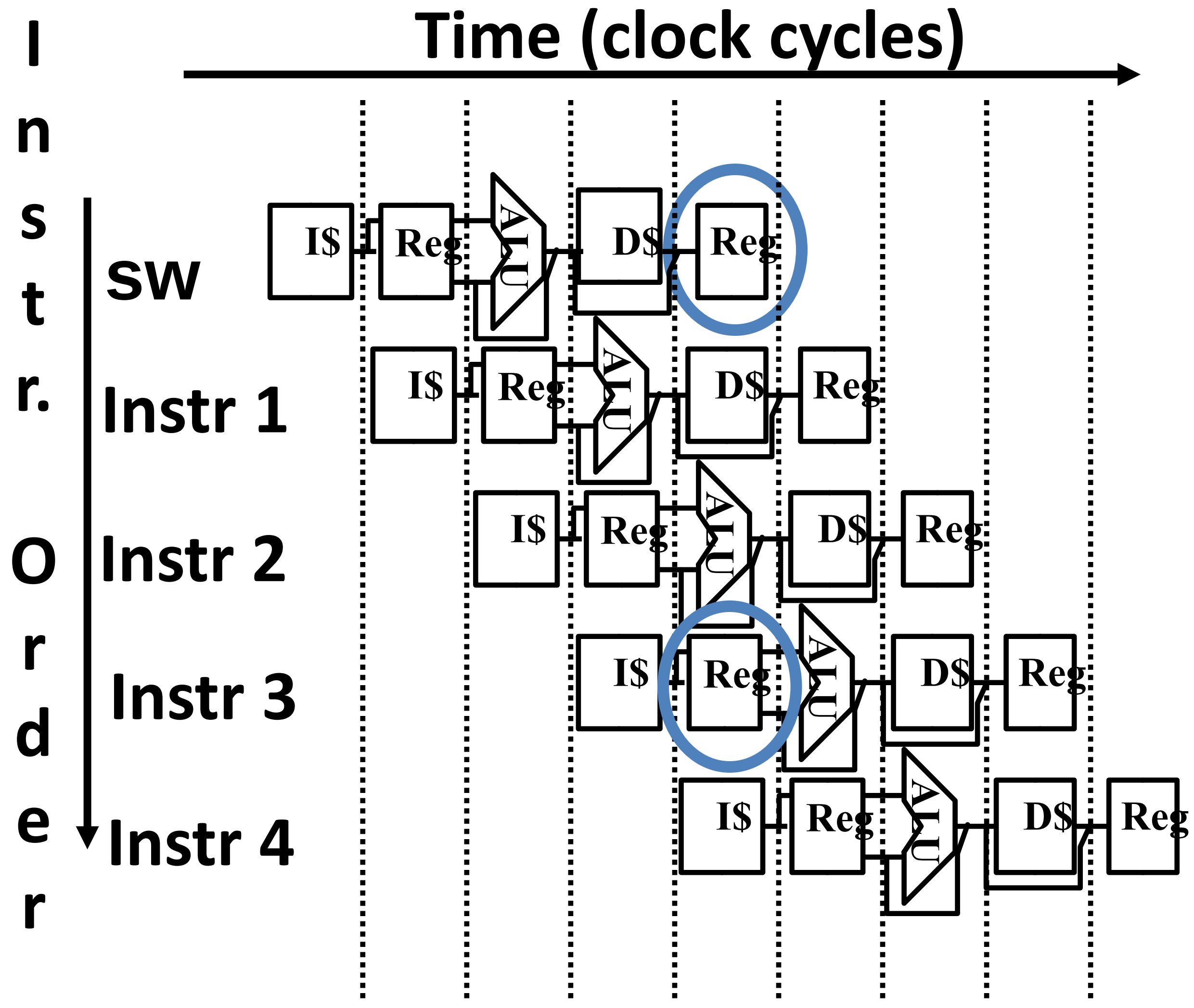
Solution 2: Duplicate hardware

- Infeasible and inefficient to create second memory
- So simulate this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)
- Have both an L1 Instruction Cache and an L1 Data Cache
- Need more complex hardware to control when both caches miss

Solution 2: Duplicate hardware



Structural Hazard #2: Registers



Can we read and write to registers simultaneously?

Structural Hazard #2: Registers

Two different solutions have been used:

1) RegFile access is *VERY* fast: takes less than half the time of ALU stage
   - Write to Registers during first half of each clock cycle
   - Read from Registers during second half of each clock cycle

2) Build RegFile with independent read and write ports

Result: can perform Read and Write during same clock cycle

Eliminating Structural Hazards

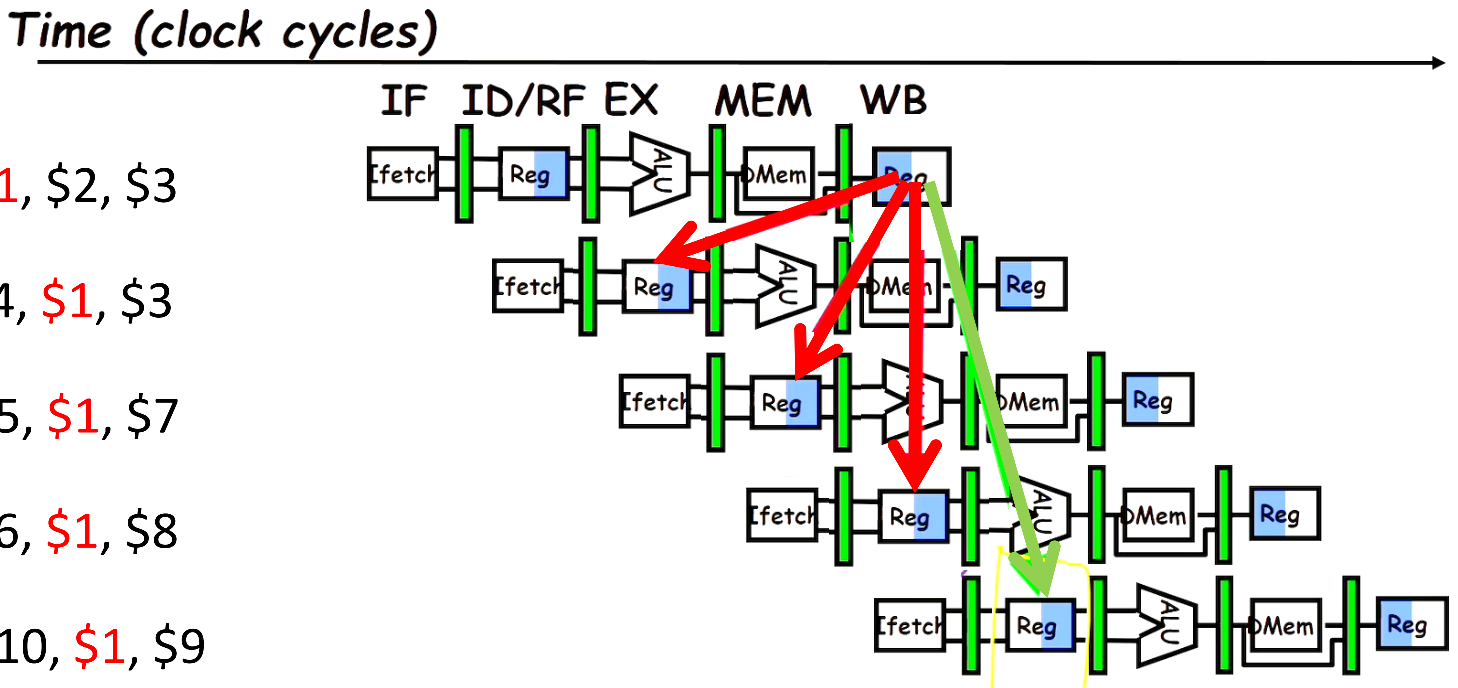
- Duplicate resource
- Pipeline the resource
- Reorder the instructions
- Stall → if its is too expensive to eliminate a structural hazard
- No new instruction is issued until the hazard has been resolved

Effects of Structural hazard

- A processor with no structural hazards will have less CPI as compared to a processor with structural hazards
- Still, sometimes designers allow structural hazards because
  - Reduction of hardware cost
  - Remember a 1-port memory is much cheaper as compared to 2-port memory (from previous example)
  - Specially if structural hazards are not that common, then no point in adding extra hardware and increasing the overall cost

Data Hazards

- Data hazards occur when pipeline changes the order of read/write access to operands as compared to unpipelined execution
- Example



add $1, $2, $3

sub $4, $1, $3

mul $5, $1, $7

and $6, $1, $8

or $10, $1, $9

Three Generic Data Hazards

- **Read After Write (RAW)**
- Instr$_J$ tries to read operand before Instr$_I$ writes it

I: add $1, $2, $3

J: sub $4, $1, $3

- Caused by a data dependence
- This hazard results from an actual need for communication.

Three Generic Data Hazards

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

I: add \$4, \$1, \$3
J: sub \$1, \$2, \$3
K: mul \$4, \$1, \$3

- Called an anti-dependence by compiler writes.
- This hazard results from reuse of the name r1
- Can’t happen in MIPS 5-stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Three Generic Data Hazards

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

I: add \$1, \$4, \$3
J: sub \$1, \$2, \$3
K: mul \$6, \$1, \$7

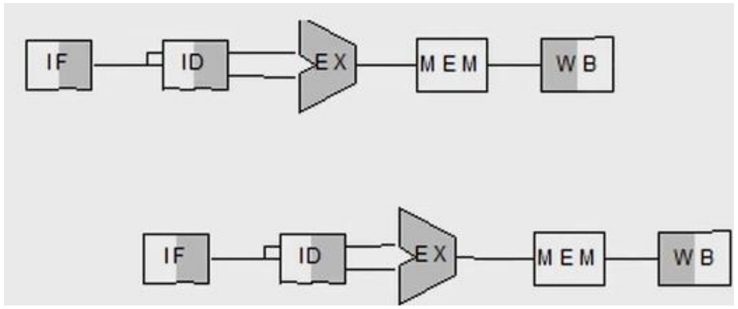
- Called an output dependence
- This also results from the reuse of name r1
- Can’t happen in MIPS 5-stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- WAR and WAW happen in out of order pipes

How to Handle RAW Data Hazard?

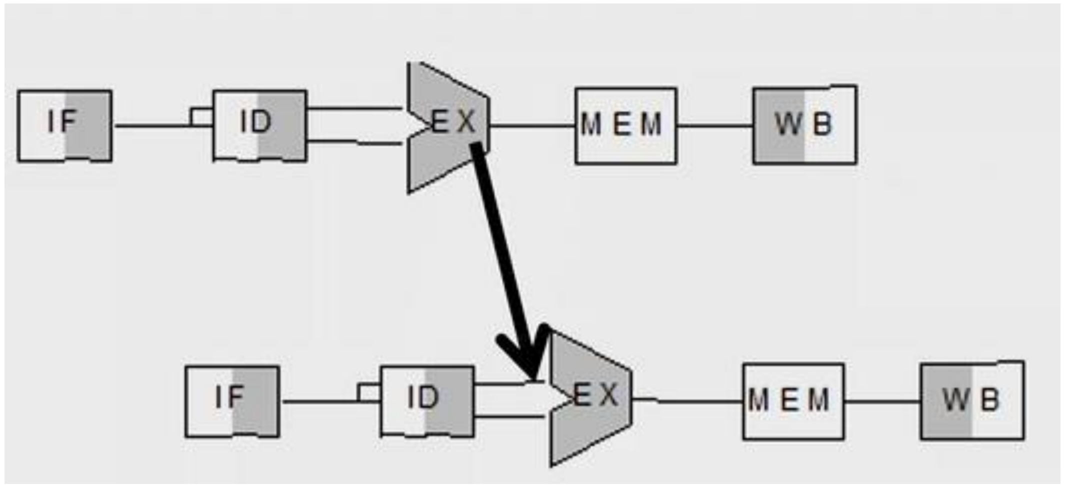
- Data Hazard: instruction needs data from the result of a previous instruction still executing in pipeline
- **Solution:** Forward data if possible

add $1, $2, $3

sub $4, $1, $3



Cont…



Operand Forwarding to Avoid Data Hazard

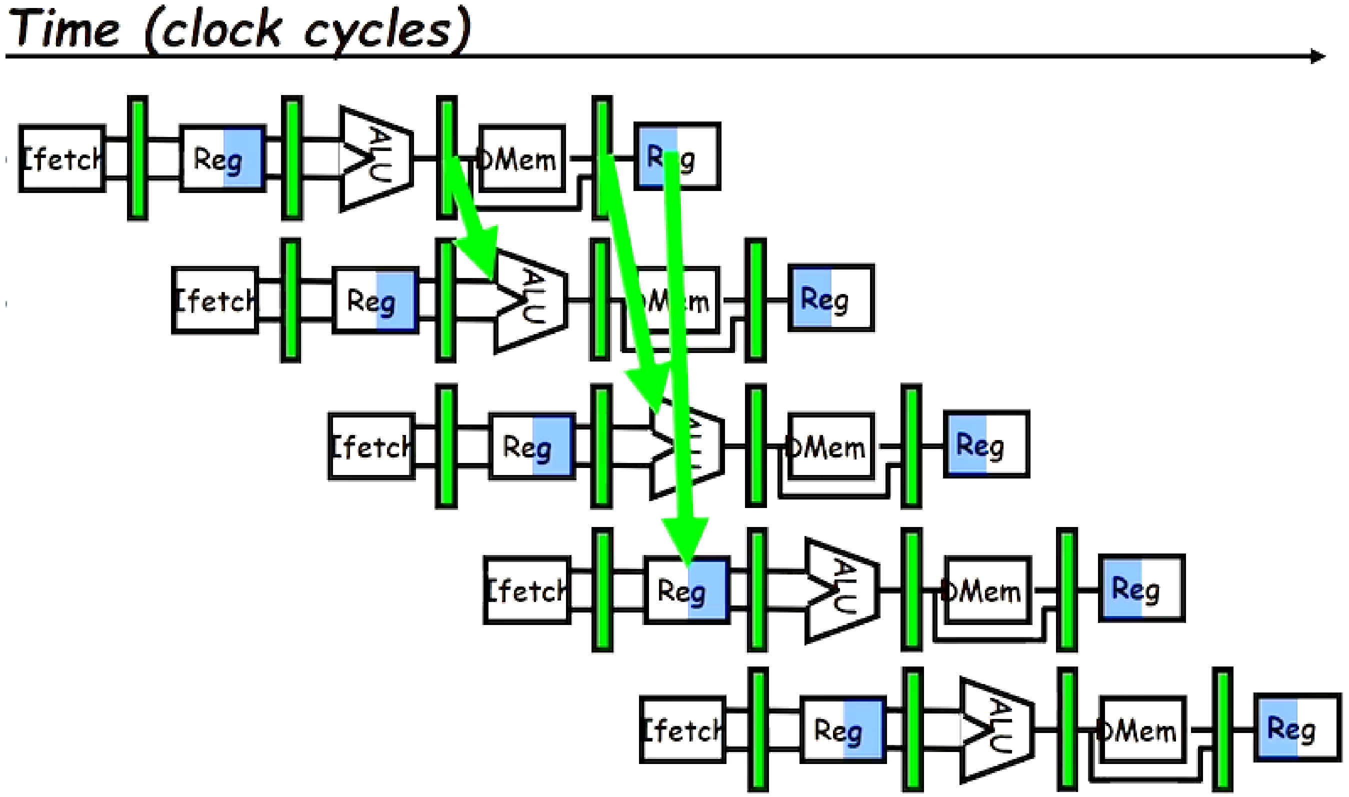
add $1, $2, $3

sub $4, $1, $3

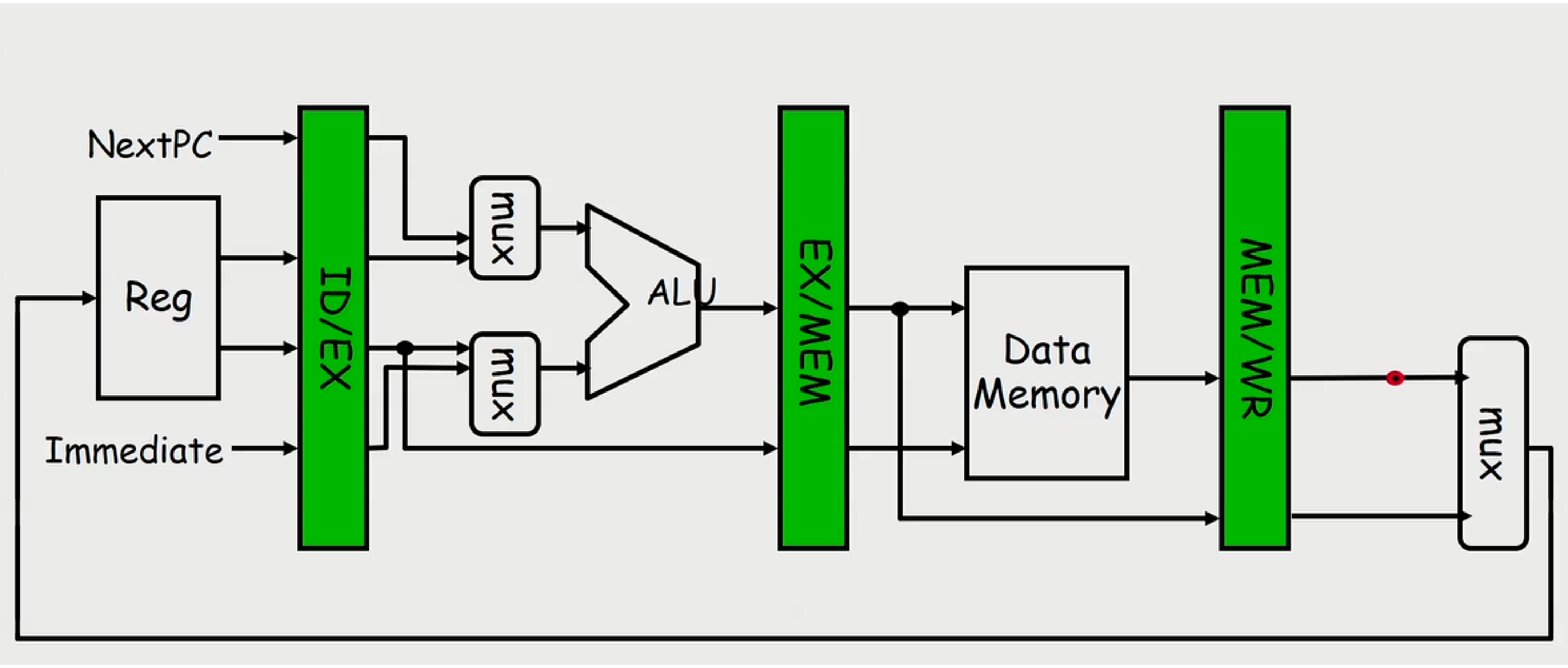
mul $5, $1, $7

and $6, $1, $8

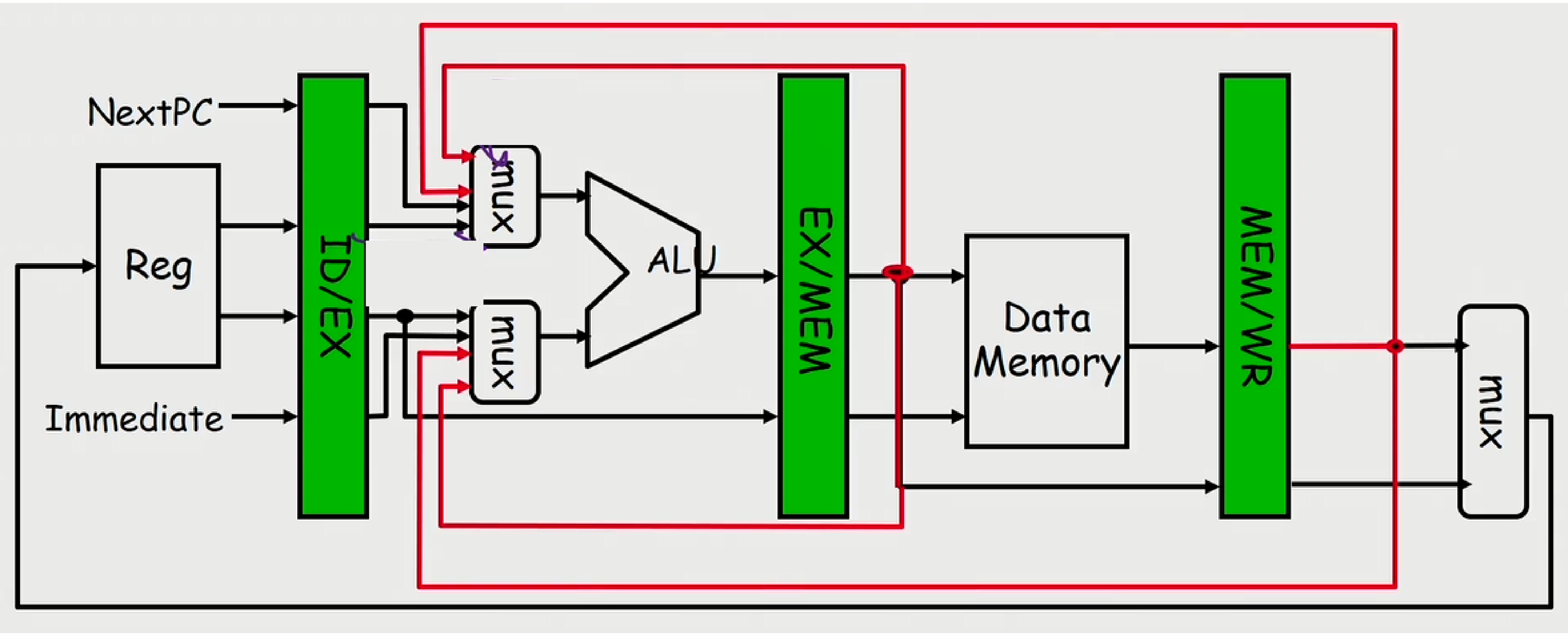
or $10, $1, $9



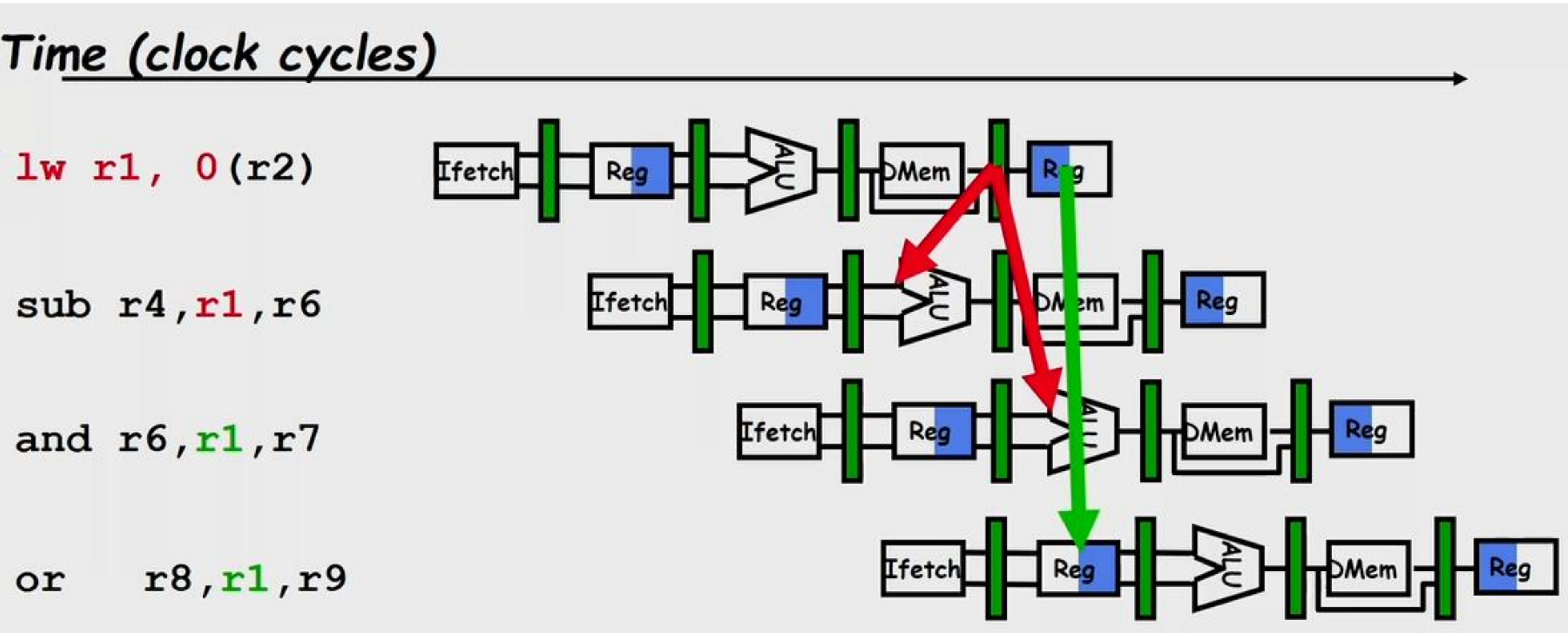
Hardware Change for Forwarding



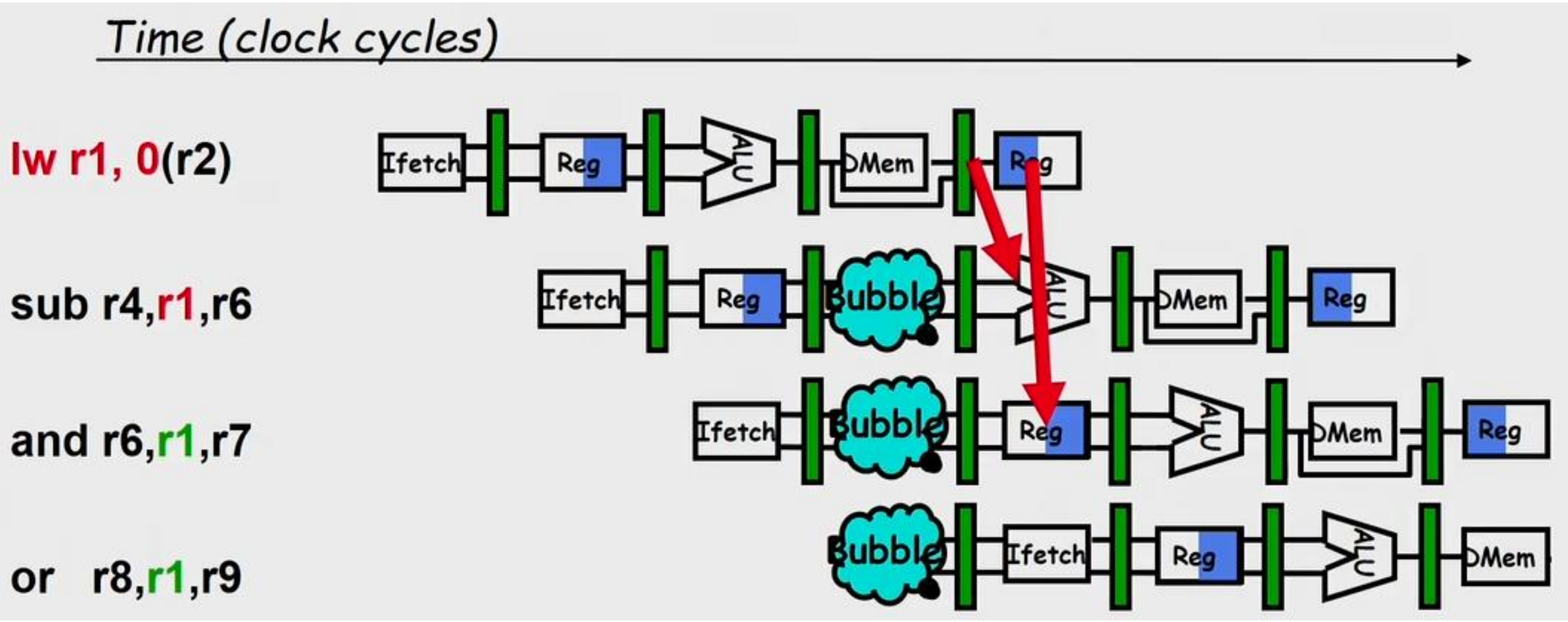
Cont…


NextPC
Reg
Immediate
ID/EX
mux
mux
ALU
EX/MEM
Data Memory
MEM/WR
mux

Data Hazard even with Operand Forwarding



Resolving the Load-ALU Hazard



Static/Dynamic Scheduling…CPE440(Computer Architecture)

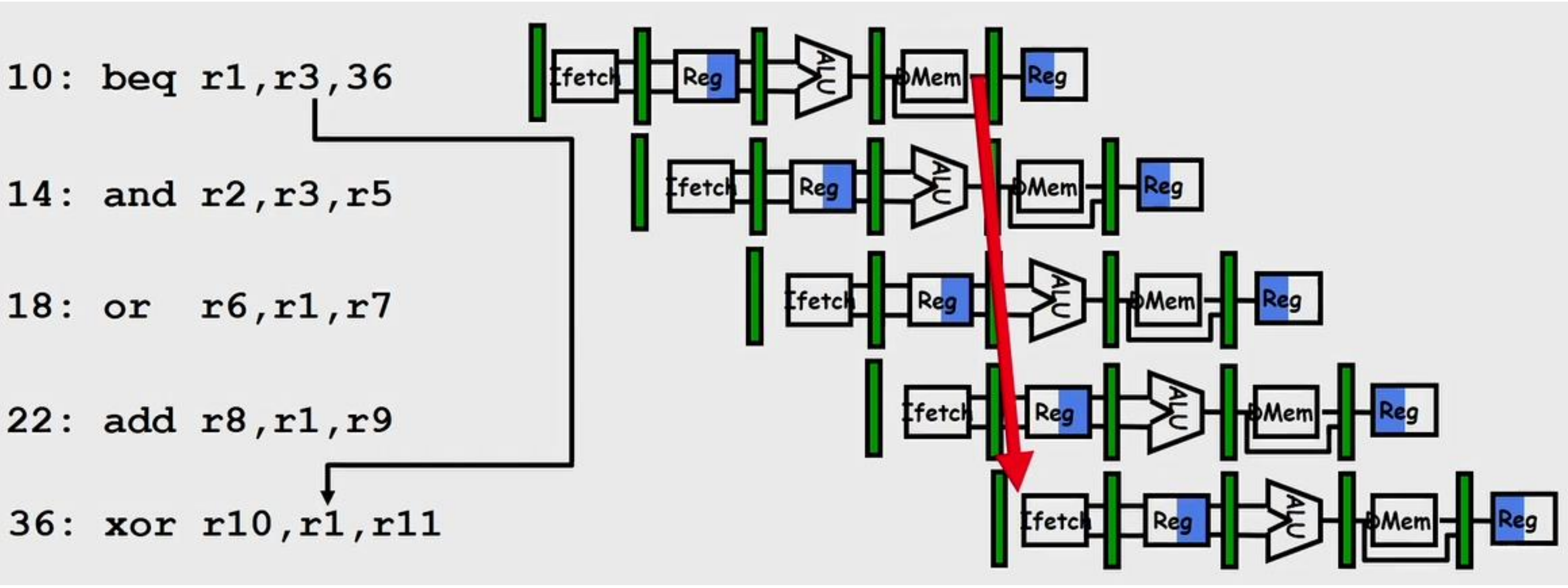
- Instruction Level Parallelism must be used by finding unrelated instructions that can be overlapped in a pipeline

- To avoid stalls, dependent instructions must be separated from source instructions by latency of source instructions in clock cycles

- If it is done by compiler → Static scheduling

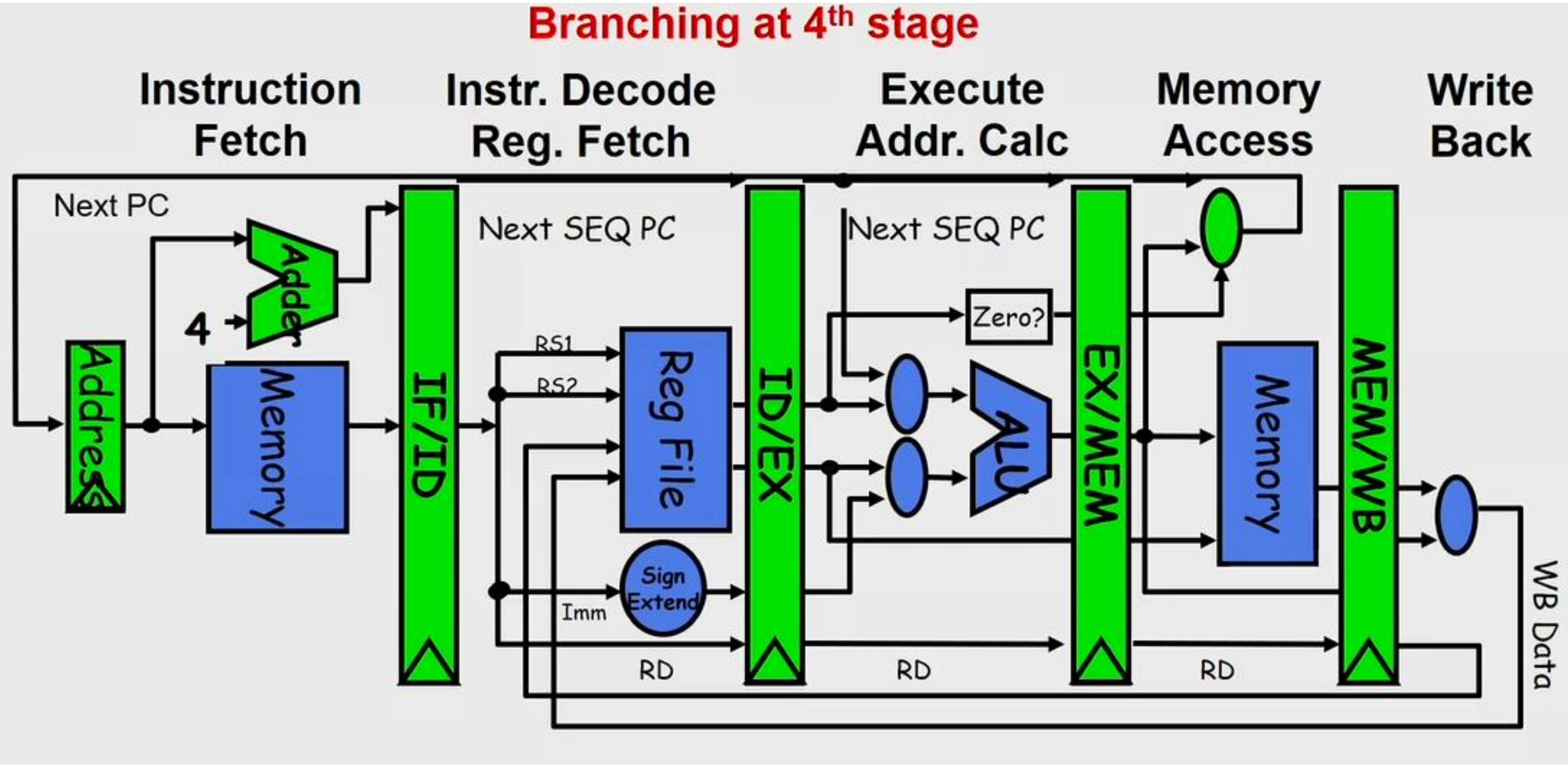
- If done by hardware at run time → Dynamic scheduling

Control Hazard on Branch

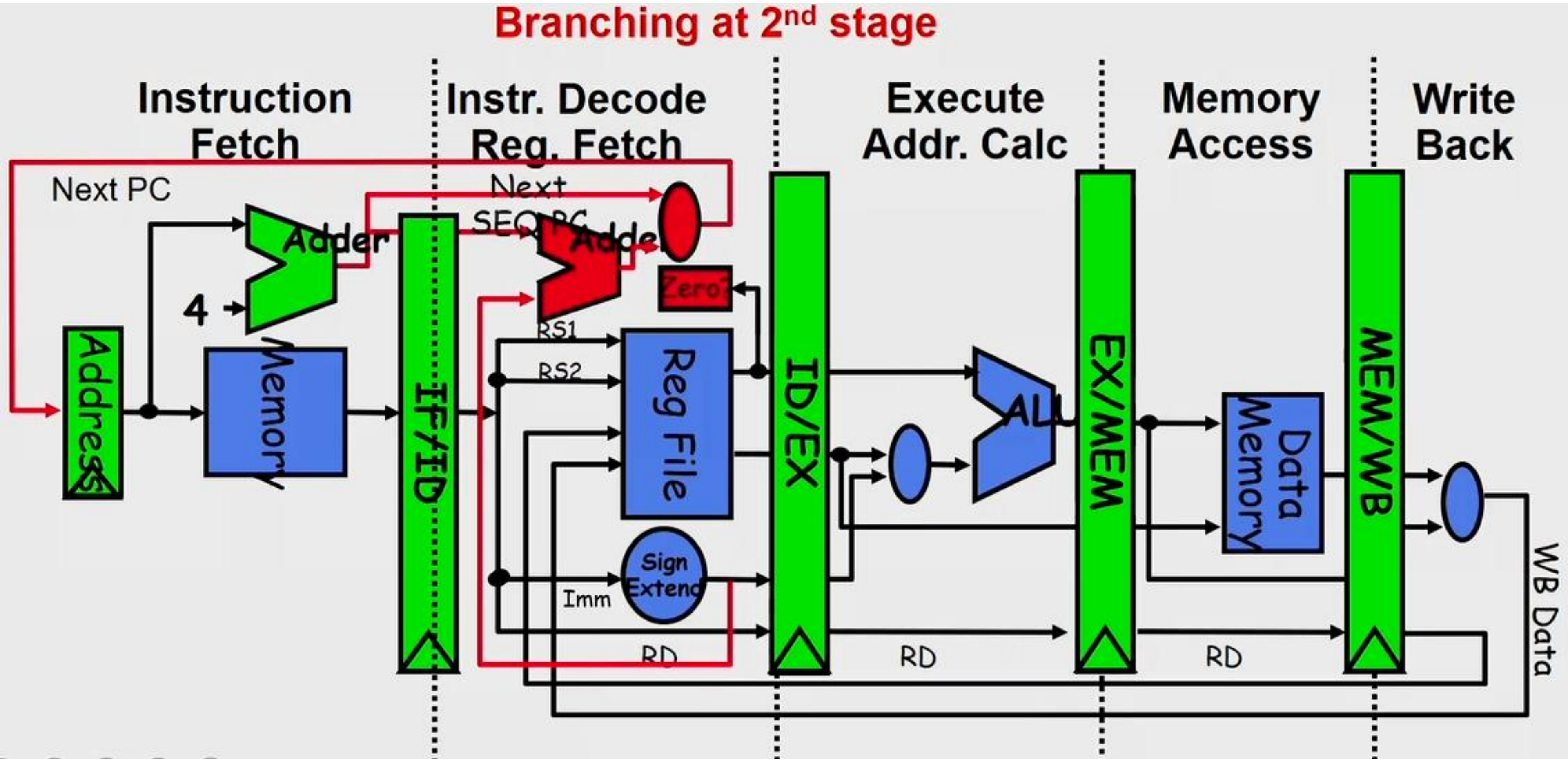


```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```

Conventional MIPS Pipeline



Branch Optimized MIPS Pipeline



Conditional Branches

- When do you know you have a branch?
  - During ID cycle (could you know before that?)
- When do you know if the branch is Taken or Not-Taken
  - During EXE cycle/ID stage depending on the design
- We need sophisticated solutions for following cases
  - Modern pipelines are deep (10+ stages)
  - Several instructions issued/cycle
  - Several predicted branches in- flight at the same time

Solution…Branch Prediction

- Static Branch Prediction
- Dynamic Branch Prediction

Four Branch Hazard Alternatives

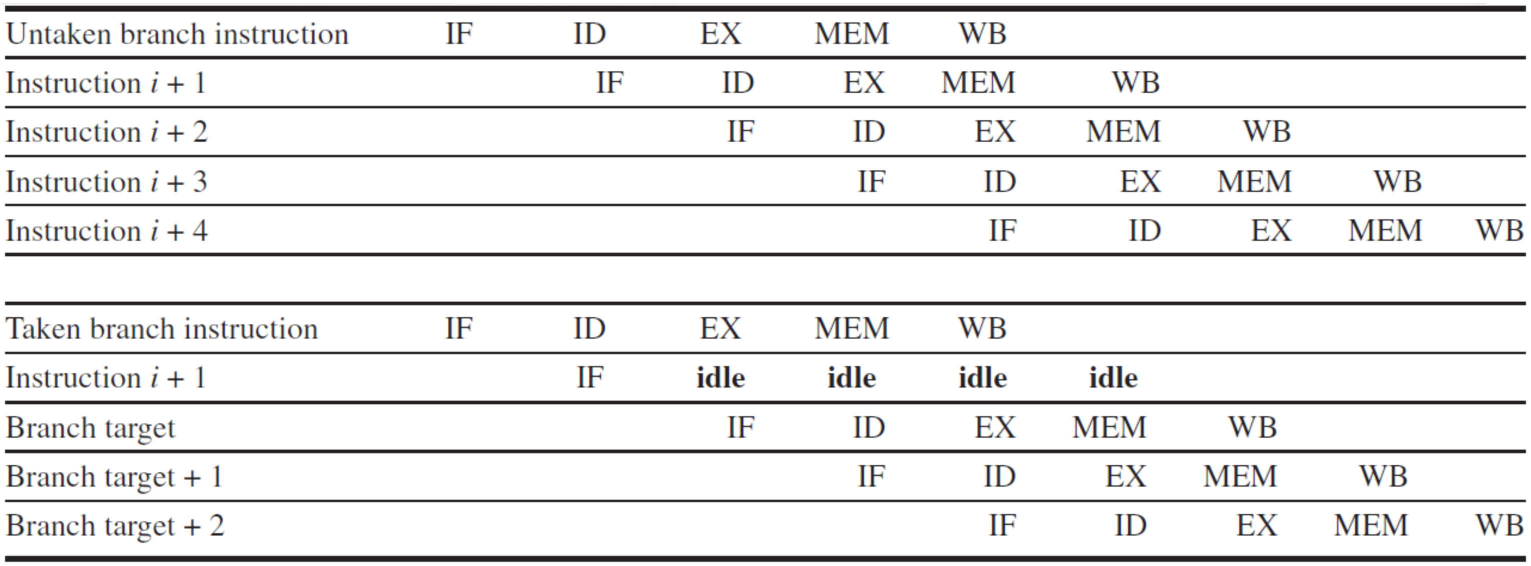
- #1: Stall until branch direction is clear
- #2: Predict Branch Not Taken
- #3: Predict Branch Taken
- #4: Delayed Branch

Static Prediction

- Predicted – NT
  - 30 – 40% accuracy (not so good)
- Predicted – T
  - 60 – 70% accuracy
- BTFN (Backward T, Forward NT)
  - Quite accurate in case of loops

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

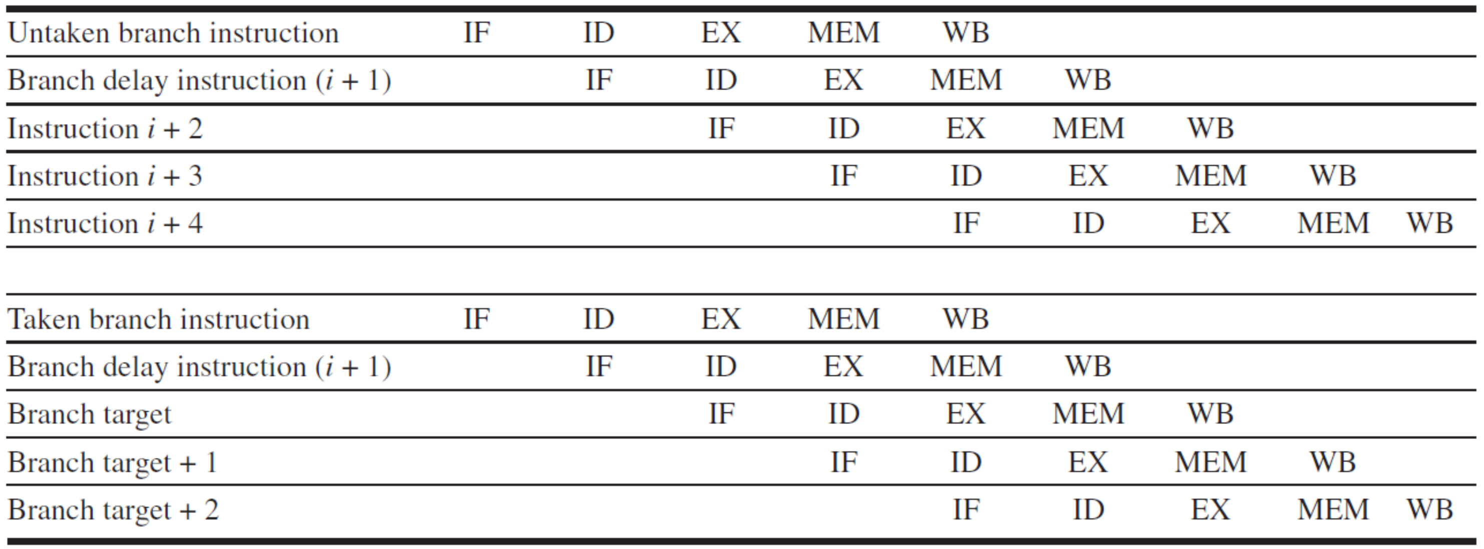


Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

- #3: Predict Branch Taken
  - But branch target address is not know by IF stage
  - Target is known at same time as branch outcome (ID stage)
  - MIPS still incurs 1 cycle branch penalty

- #4: Delayed Branch
  - Define branch to take place AFTER one instruction following the branch instruction
  - 1 slot delay allows proper decision and branch target address in 5 stage pipeline (MIPS uses this approach)
  - Where to get instructions to fill branch delay slot?



Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay 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

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i + 1$)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

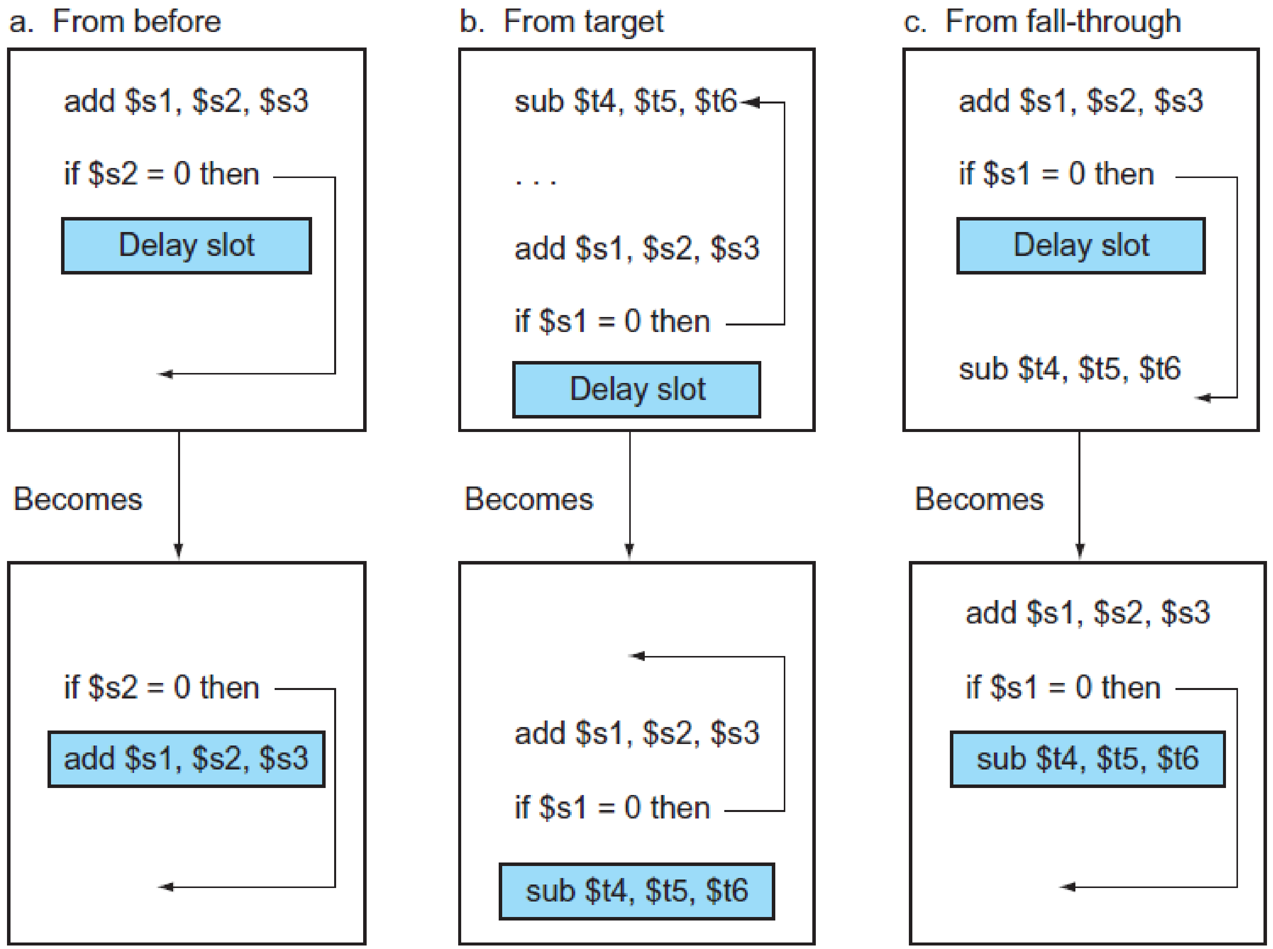
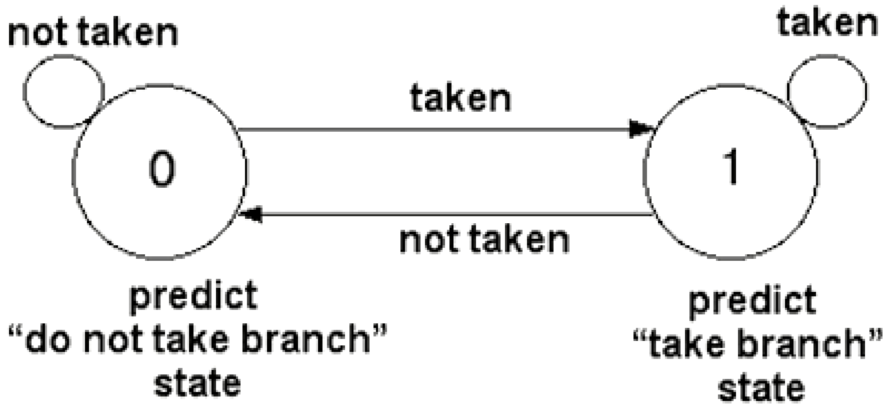


FIGURE 4.64 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of `$s1` in the branch condition prevents the `add` instruction (whose destination is `$s1`) from being moved into the branch delay slot. In (b) the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the `sub` instruction when the branch goes in the unexpected direction. By "OK" we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if `$t4` were an unused temporary register when the branch goes in the unexpected direction.

Dynamic Branch Prediction

- Prediction Scheme with
  - 1 bit FSM
  - 2 bit FSM



One bit is not enough

```
For (i=0; i<8; i = i + 1 )
        x[i] = x[i] + s
```

TTTTTTTNTTT

- Example: short loop (8 iterations)
  - Taken 7 times then not taken
  - Not taken mis-predicted as it was taken previously
- Execute the same loop again
  - First always mispredicted
  - Then 6 predicted correctly
  - Then last one mispredicted again
  - Misprediction rate = 2/8 = 25%

One bit is not enough

DC08: TTTTTTTTTTT ... TTTTTTTTTTTNTTTTTTTTT ...

100,000 iterations

NT

TN

How often is branch outcome != previous outcome?

2 / 100,000 ← **99.998% Prediction Rate**

DC44: TTTTT ... TNTTTTT ... TNTTTTT ...

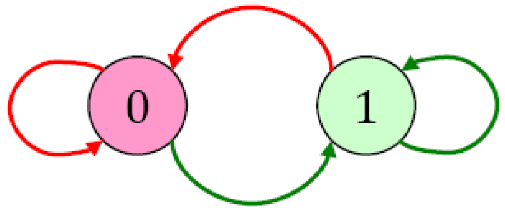
2 / 100 ← **98.0%**

DC52: TNTNTNTNTNTNTNTNTNTNTNTNTNTNT ...

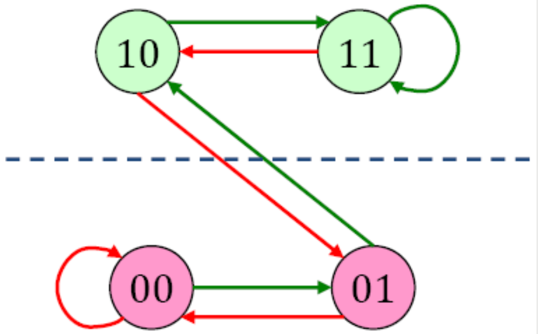
2 / 2 ← **0.0%**

Prediction Scheme with 1 or 2 bit FSM

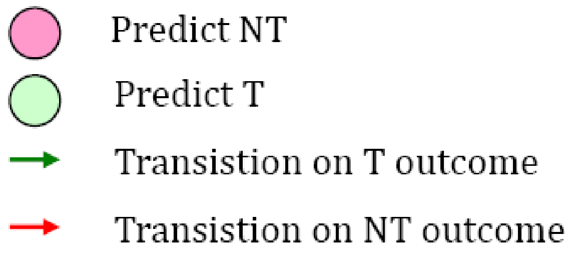
- The use of a 2-bit predictor will allow branches that favor taken (or not taken) to be mispredicted less often than the one-bit case. (reinforcement learning)



FSM for 1-bit Predictor



FSM for
2-bit Predictor



Branch Prediction In Hardware

- Branch prediction is extremely useful in loops.
- A simple branch prediction can be implemented using a small amount of memory indexed by lower order bits of the address of the branch instruction. **(branch prediction buffer)**
- One bit stores whether the branch was taken or not.
- The next time the branch instruction is fetched refer this bit

Dynamic branch prediction

- Use a Branch Prediction Buffer (BPB)
  - Also called Branch Prediction Table (BPT), Branch History Table (BHT)
  - Records previous outcomes of the branch instruction.
  - How to index into the table is an issue.
- A prediction using BPB is attempted when the branch instruction is fetched (IF stage or equivalent)
- It is acted upon during ID stage (when we know we have a branch)

Advanced Branch Prediction Techniques…CPE440(Computer Architecture)

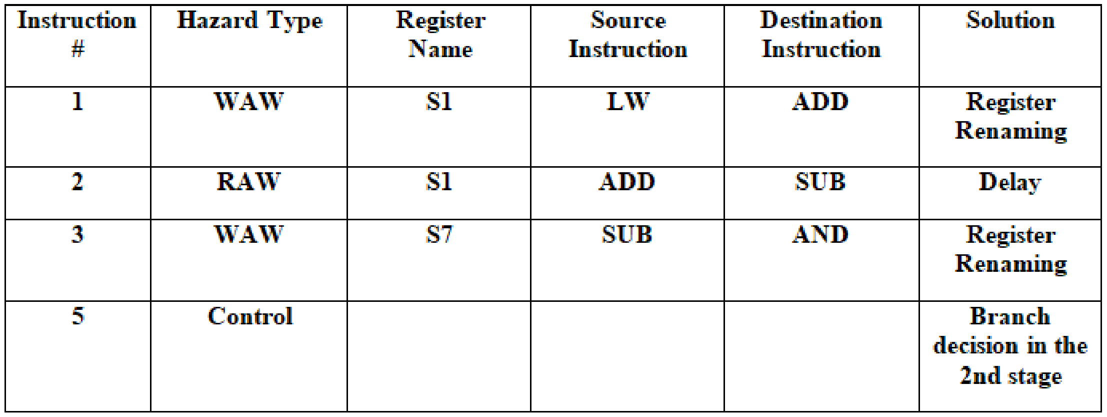
- Correlating Branch Prediction
- Tournament Branch Prediction

Example

- In a 5-stage pipelined MIPS architecture with branch decision in the 4th stage and register file with 1 reading port, ***discover*** different hazards found in the given sequence of code:

```
LW      s1, 75(s4)
ADD     s1, s2, s3
SUB     s7, s1, s6
AND     s7, s4, s5
BEQ     s8, s9, 75
        ……
```

Fill the table.



Instruction #	Hazard Type	Register Name	Source Instruction	Destination Instruction	Solution
1	WAW	S1	LW	ADD	Register Renaming
2	RAW	S1	ADD	SUB	Delay
3	WAW	S7	SUB	AND	Register Renaming
5	Control				Branch decision in the 2nd stage

Moreover, there will be a structural hazard whenever an instruction tries to read 2 registers in one cycle because register file has only one reading port. Therefore, only one register will be read in a cycle. So, a delay will be inserted between the reading operations of the both registers.

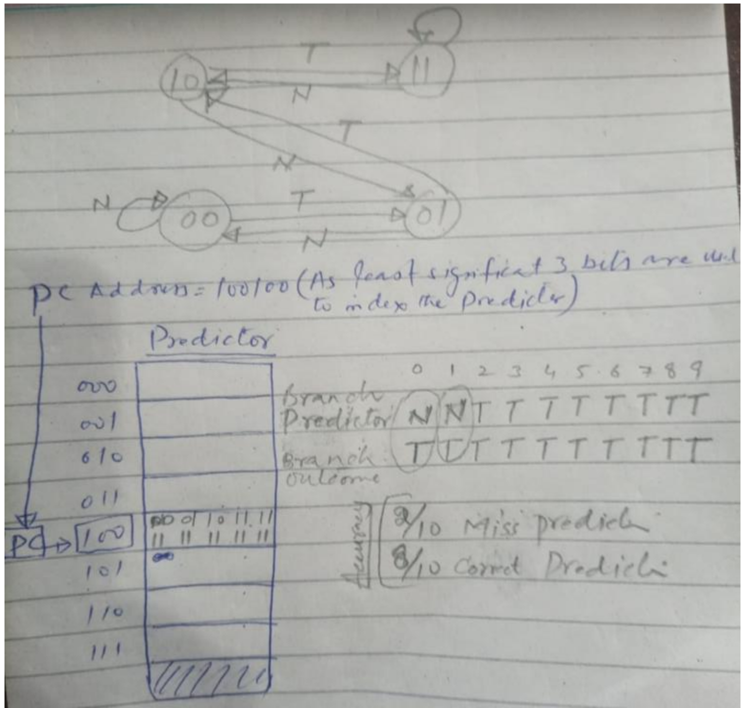
Example

- For the given piece of C code:

  For (i = 0; i =< 10; i = i + 1)

  W[i] = X[i]+ s*Y[i];

- Design a dynamic branch predictor that ***predicts*** the branches outcome for the given code efficiently. If the Program Counter (PC) address of branch is 100100 and least significant 3 bits of the PC are used to index the predictor. Initialize the predictor with zeros and fill the entries for 10 iterations. Calculate the accuracy of the branch predictor for first 10 iterations.



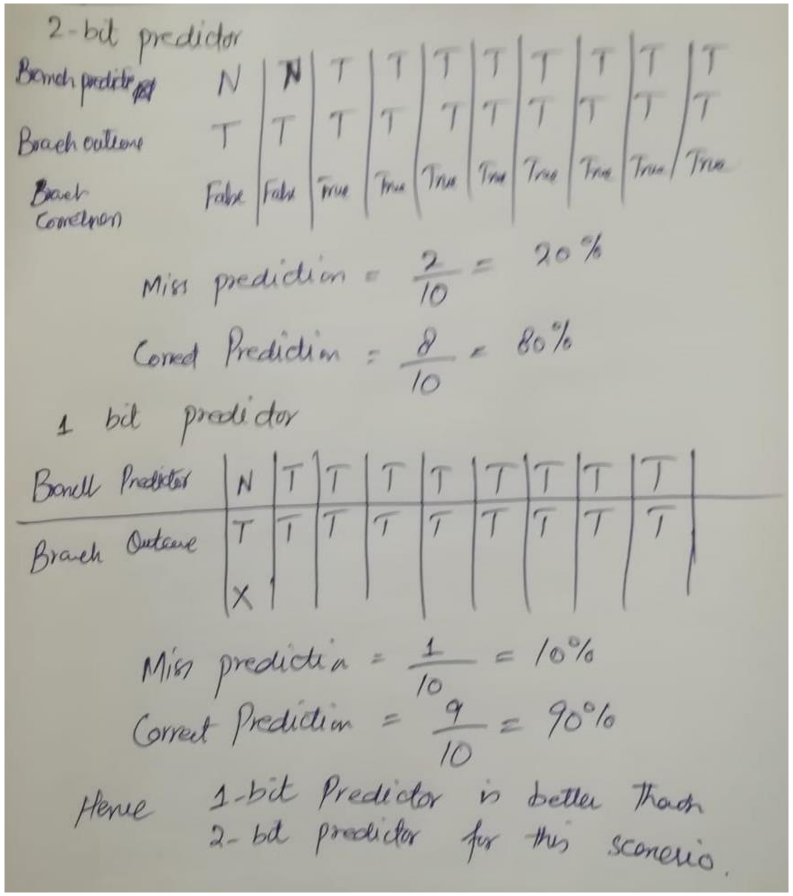
PC Address = 100100 (As least significant 3 bits are used to index the Predictor)

	0	1	2	3	4	5	6	7	8	9
Branch Predictor	N	N	T	T	T	T	T	T	T	T
Branch outcome	T	T	T	T	T	T	T	T	T	T

Accuracy
2/10 Miss predicti
8/10 Correct Predicti

Example

- Design two dynamic branch predictors (1-bit & 2-bit) that ***predict*** the branch outcome that is always taken. The Program Counter (PC) address of the branch is 100100 and least significant 3 bits of the PC are used to index the predictors. Initialize the predictor with zeros (Not Taken) and fill the entries for 10 iterations. Compare the efficiency of both predictors by calculating the accuracy of the branch predictors for first 10 iterations.



2-bit predictor

Branch predictor	N	N	T	T	T	T	T	T	T	T
Brach outcome	T	T	T	T	T	T	T	T	T	T
Brach Correlation	False	False	True	True	True	True	True	True	True	True

Miss prediction = $\frac{2}{10}$ = 20%

Correct Prediction = $\frac{8}{10}$ = 80%

1 bit predictor

Branch Predictor	N	T	T	T	T	T	T	T	T
Brach Outcome	T	T	T	T	T	T	T	T	T
	X								

Miss prediction = $\frac{1}{10}$ = 10%

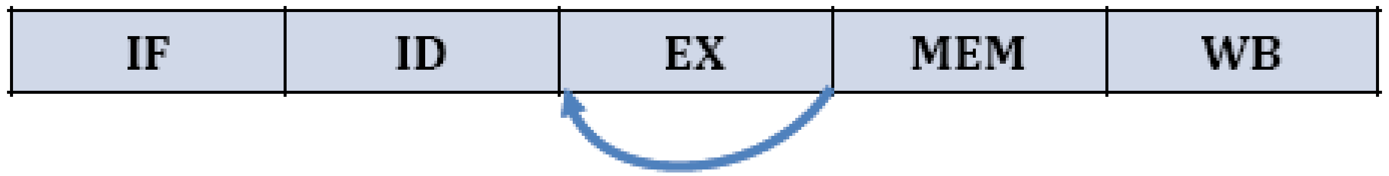
Correct Prediction = $\frac{9}{10}$ = 90%

Hence 1-bit Predictor is better than 2-bit predictor for this scenario.

Multi-cycle Floating point MIPS pipeline

- MIPS FP operation can not be completed in one clock cycle or even two
- Doing so would mean accepting a slow clock or enormous amount of logic in FP units
- Instead FP pipeline allows longer latency for operation
- Same pipeline as integer instructions with two changes
  - EX cycle may be repeated as many times as needed– number of repetitions can vary for different operations
  - There may be multiple FP functional units– A stall will occur if there is a structural hazard or data hazard

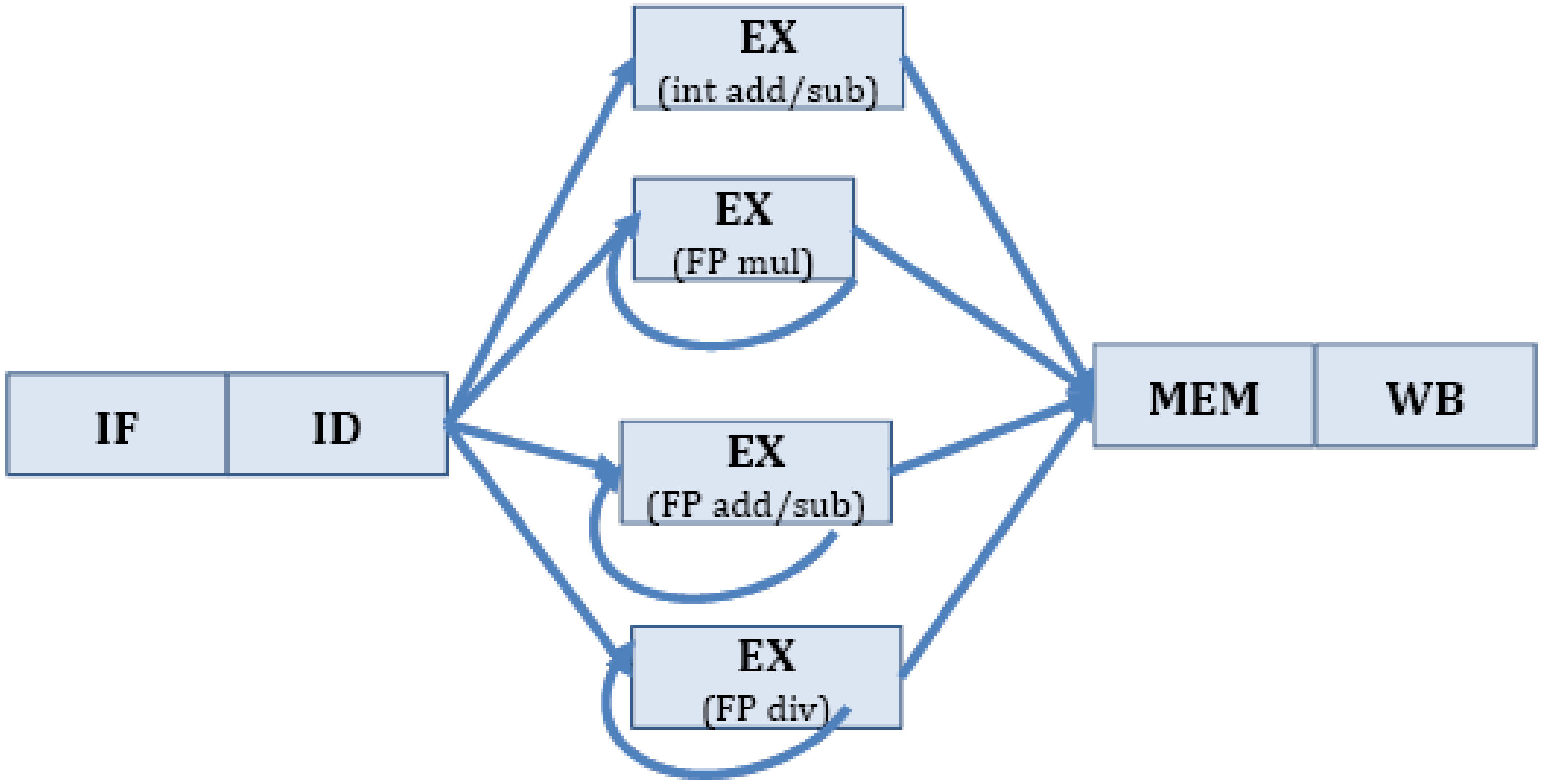
Multi-cycle Floating point MIPS pipeline



- Let’s assume in FP MIPS pipeline there are four functional units

1. Main integer unit handles load/store, integer ALU operation and branches
2. FP and integer multiplier
3. FP adder that handles FP add, subtract
4. FP and integer divider

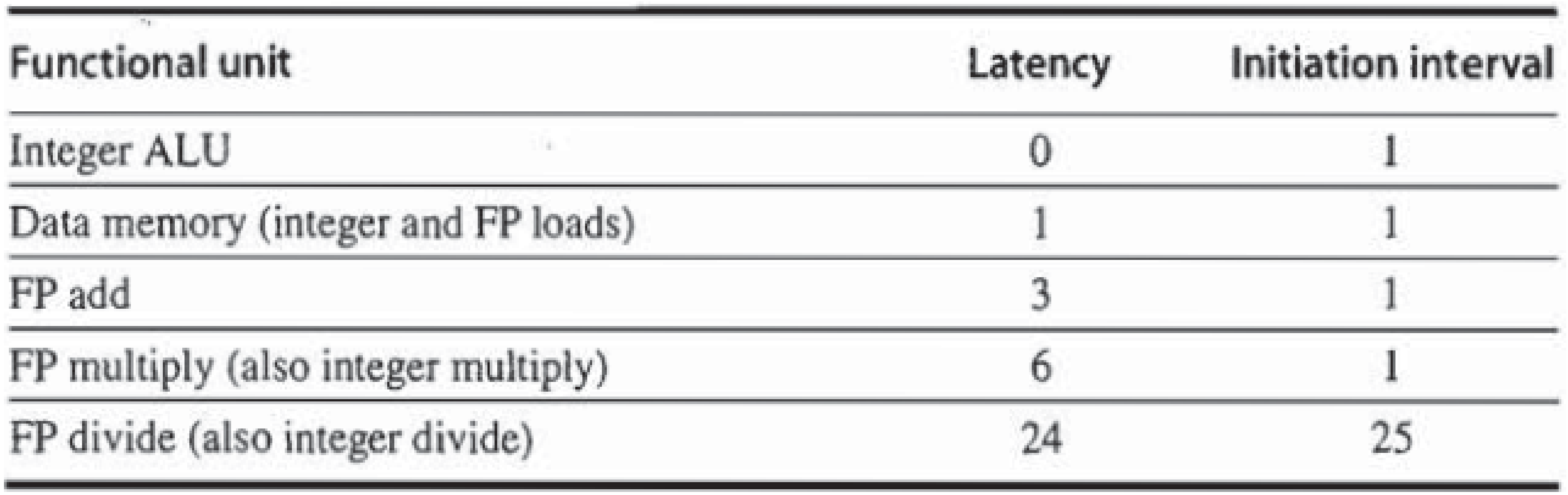
MIPS pipeline with three additional unpipelined FP



- Functional units are not pipelined, so no other instruction requiring that functional unit may issue until the previous instruction leaves (structural hazard)
- Moreover, if an instruction can not proceed to EX unit, entire pipeline stalls.
- Solution can be found by introducing pipelining in the functional units

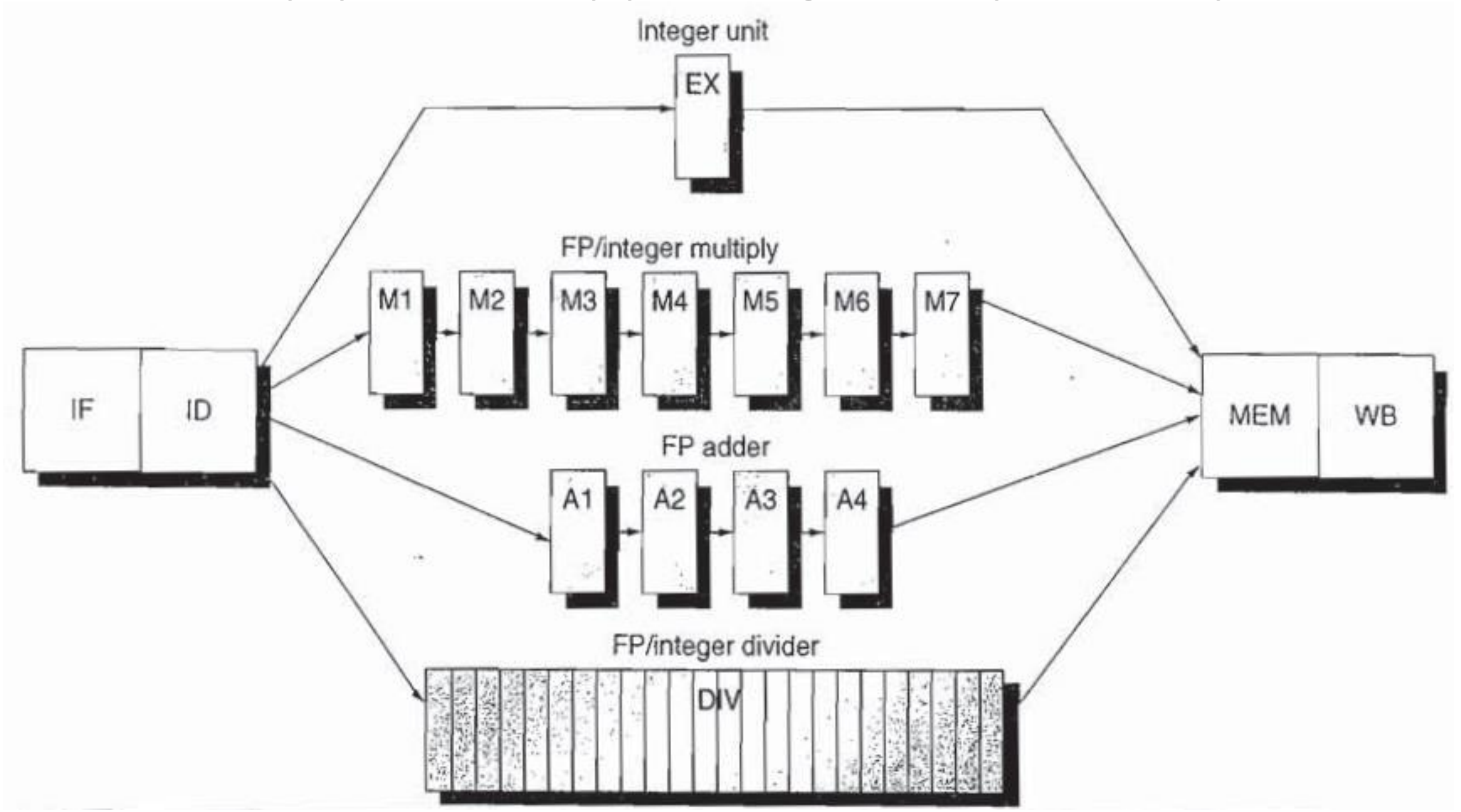
MIPS FP pipeline

- In order to understand FP pipeline, first define
- Latency
  - Number of intervening cycles between an instruction that produces a result and an instruction that uses the result
  - Normally it is EX pipeline depth minus 1
- Initiation interval or repeat interval
  - Number of cycles that must elapse between issuing two operations of same type



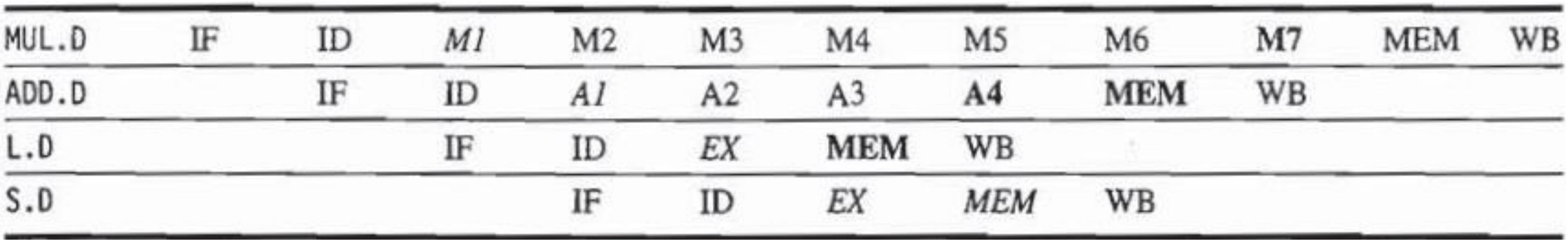
Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

MIPS FP pipeline supporting multiple FP operation



- Can support up to four FP adds, seven FP mults, and one FP divide
- To achieve higher clock speeds, fewer logic levels are put in each pipe stage which makes number of pipe stages required for more complex operations larger.
- New registers are required to be added between different FP pipeline stages

MIPS FP pipeline timing of a set of independent FP operations



MUL.D	IF	ID	*M1*	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D		IF	ID	*A1*	A2	A3	A4	MEM	WB		
L.D			IF	ID	*EX*	MEM	WB				
S.D				IF	ID	*EX*	*MEM*	WB			

- Longer latency of FP operations increases the frequency of RAW hazards