EE2011 Computer Organization

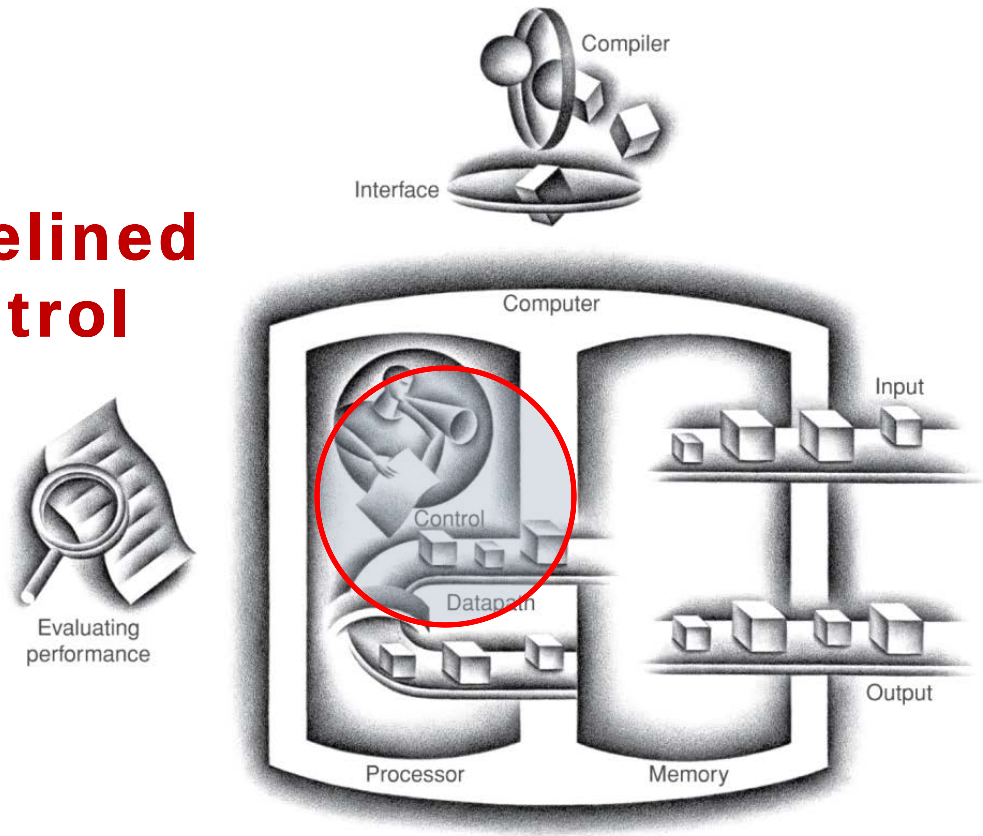
Lecture 11: Enhancing Performance with Pipelining ~ Pipelined Control

Wen-Yen Lin, Ph.D.
Department of Electrical Engineering
Chang Gung University
Email: wylin@mail.cgu.edu.tw
June 2022

PPSoC
Parallel Processing and SoC Lab
Chang Gung University

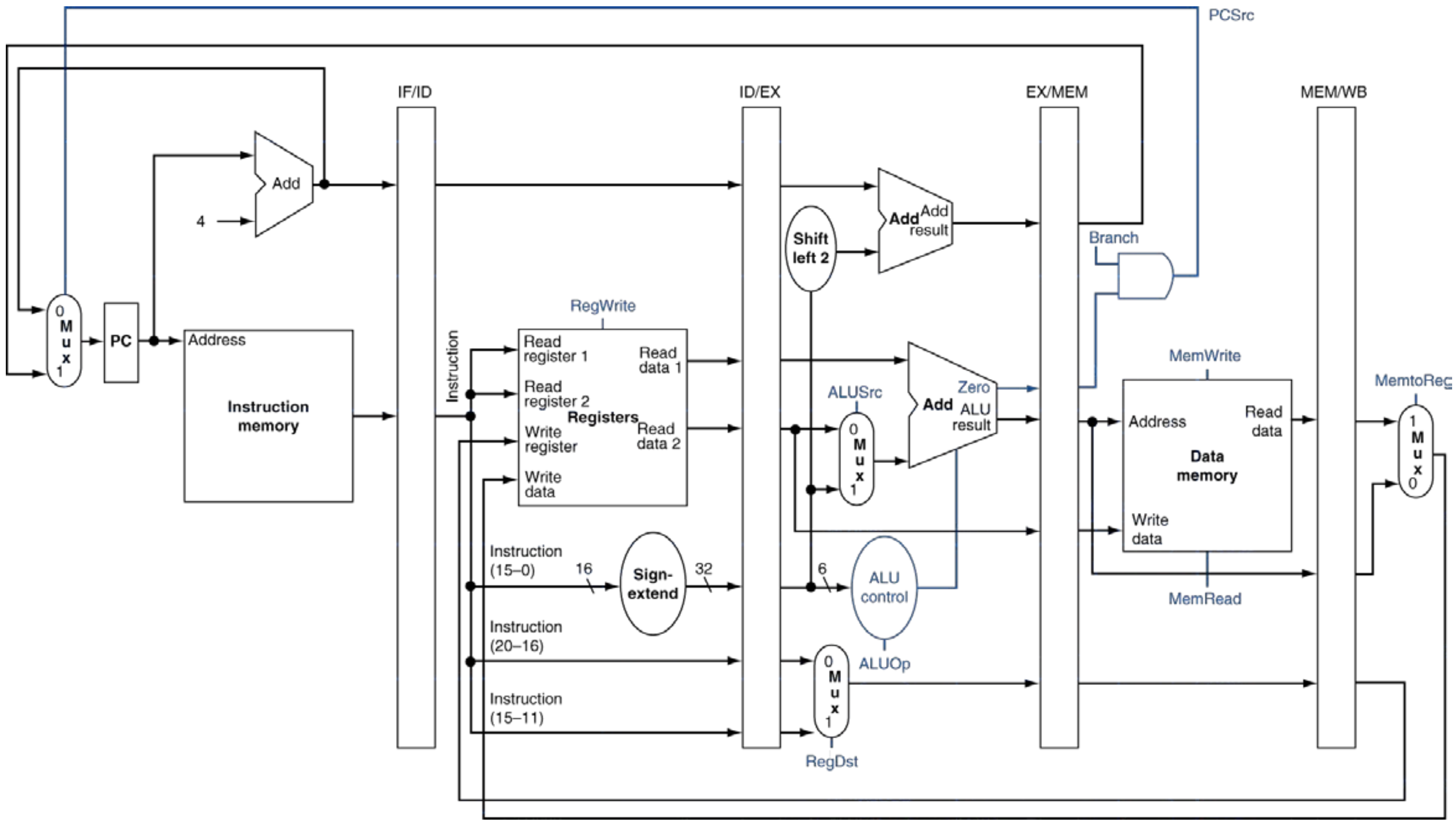
Pipelined Control (Ch. 4.7, p. 312)

Pipelined Control



Pipeline Control (Fig. 4.46)

- We borrow as much as we can from single-cycle control scheme.



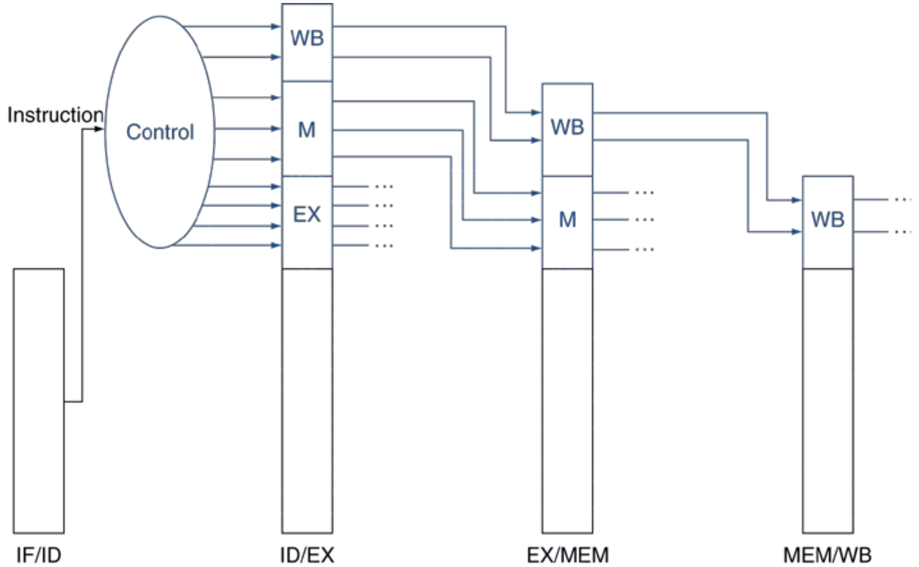
Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
    - Identical for all instructions
  - Instruction Decode / Register Fetch
    - Identical for all instructions
  - Execution/Address Calculation
    - RegDest, ALUOp, ALUSrc
  - Memory Stage
    - Branch, MemRead, MemWrite
  - Write Back
    - MemToReg, RegWrite

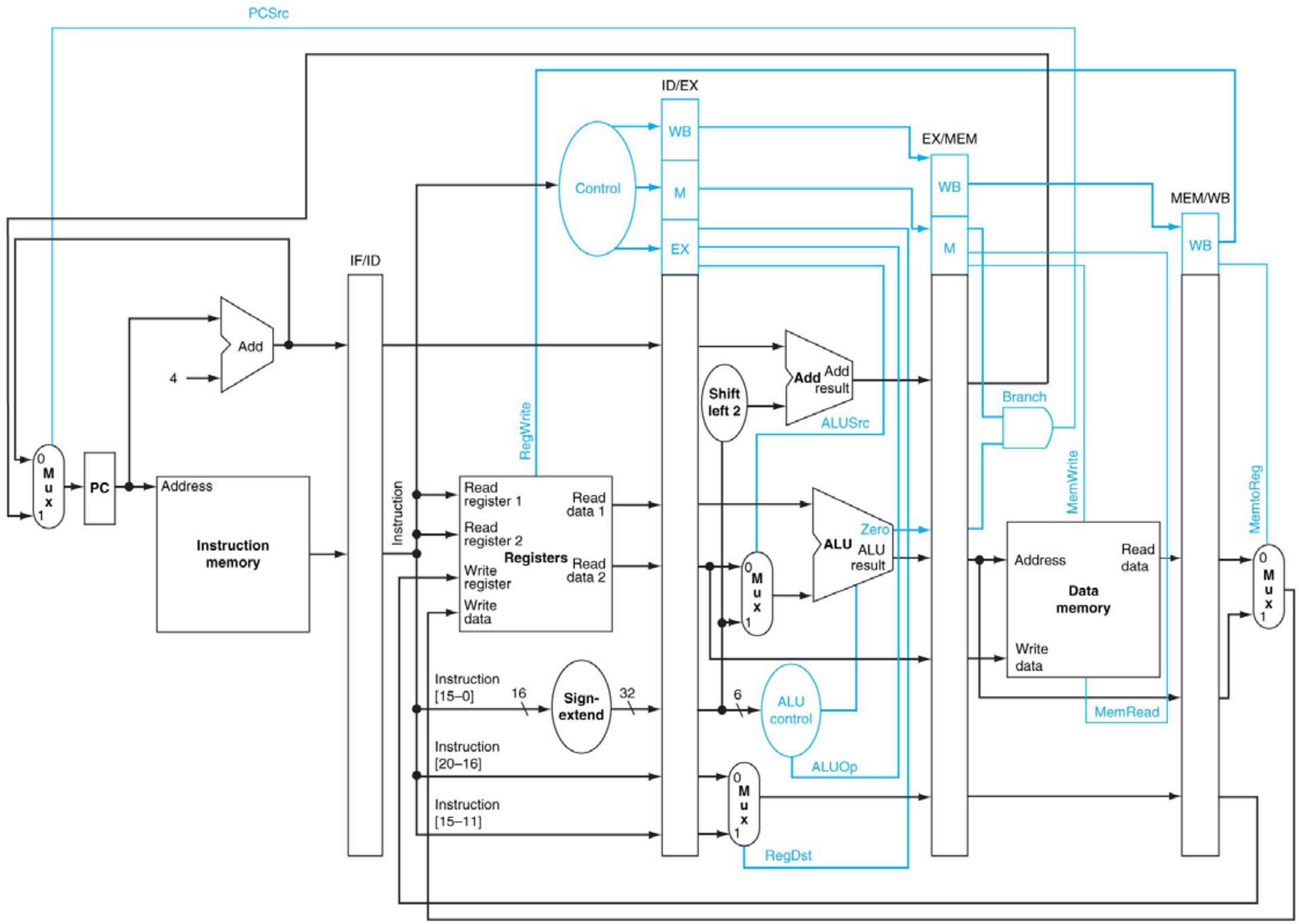
Pipeline Control (Fig. 4.49 & 4.50)

- 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	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-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

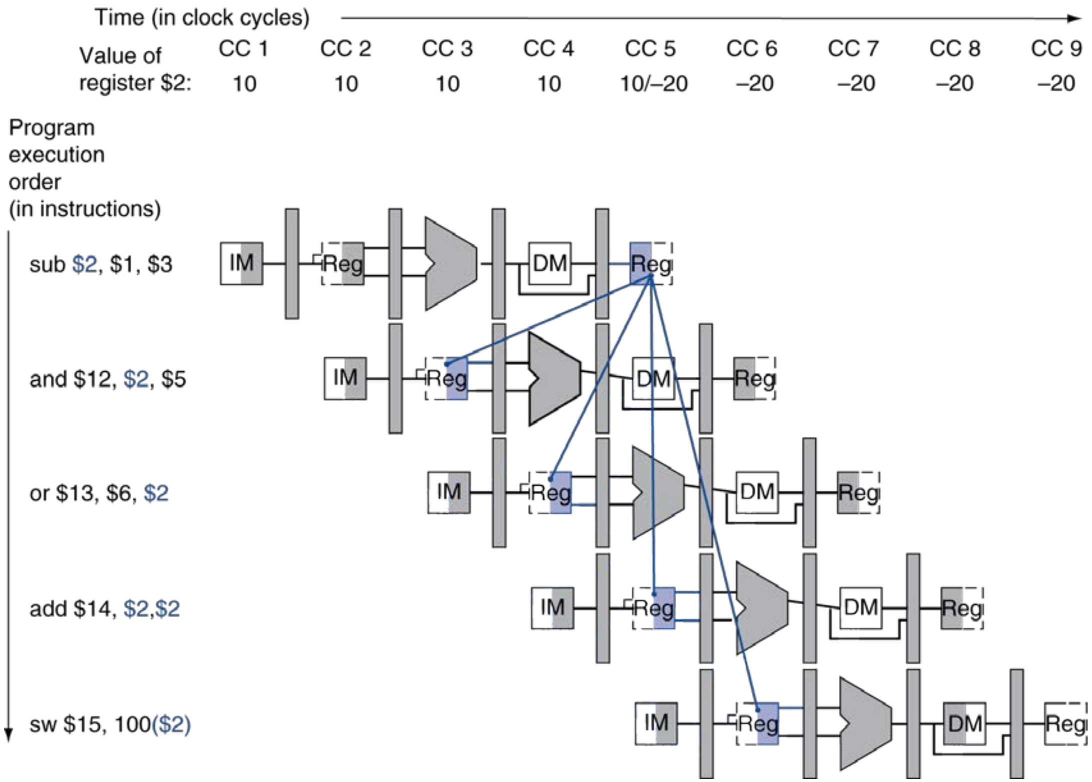


Datapath with Control (Fig. 4.51)



Data Hazards – Dependencies (Ch. 4.8, Fig. 4.52)

- Problem with starting next instruction before first is finished
  - dependencies that "go backward in time" are data hazards



Data Hazard Prevention

- Register file solution
  - Write occurs in the first half of the clock cycle, and read in the second half.
  - The read delivers what is written in the same clock cycle
- Software Solution
  - Insert independent instructions.
  - Insert “nop” instructions, but cycles are wasted.

Software Solution

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

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

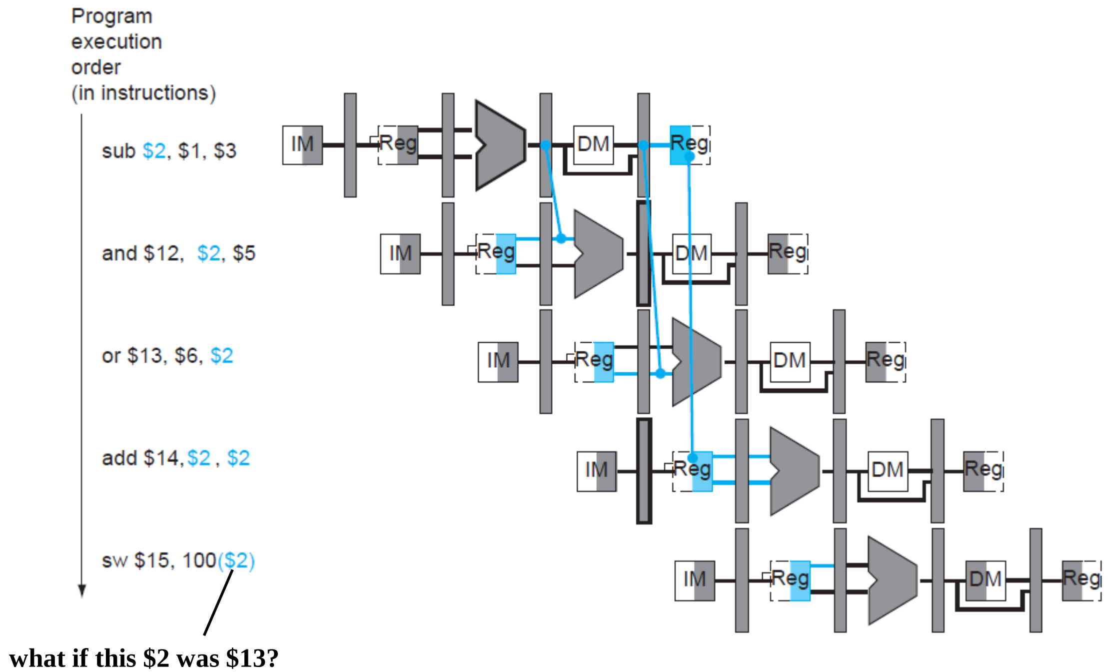
Problem: this really slows us down!

Forwarding (Fig. 4.53)

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

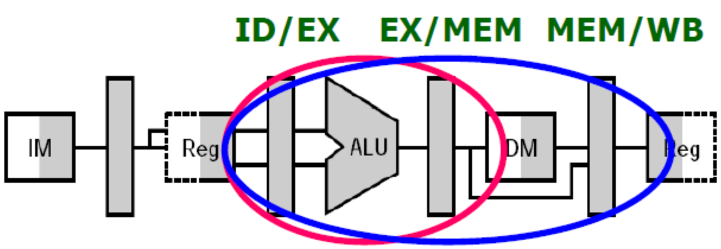
Time (in clock cycles) →

	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



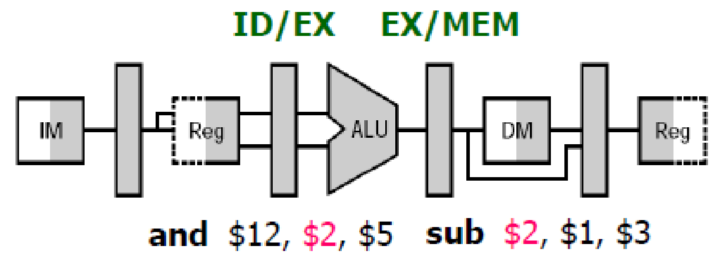
Two Pairs of Hazard Conditions (P. 318)

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



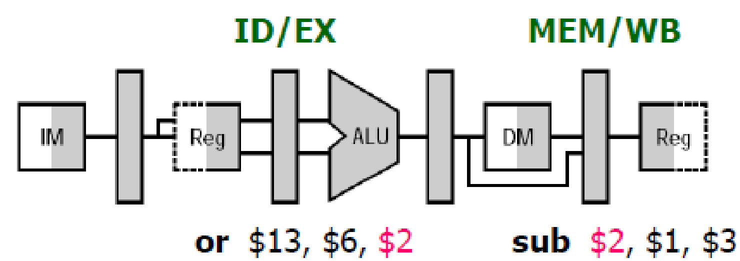
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Two Pairs of Hazard Conditions



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

E.g. Type 1-a hazard

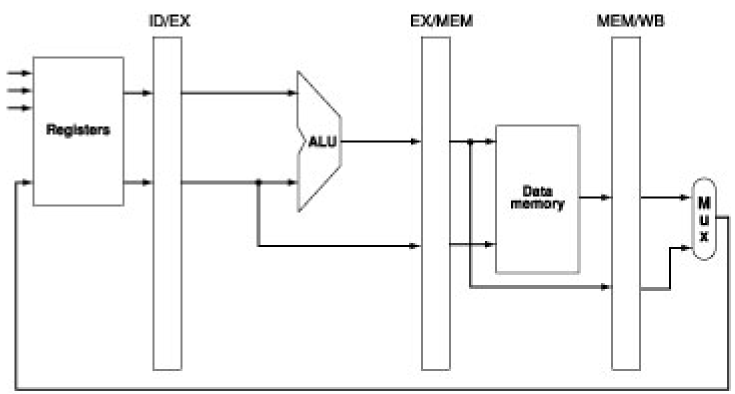


MEM/WB.RegisterRd = ID/EX.RegisterRt = $2

E.g. Type 2-b hazard

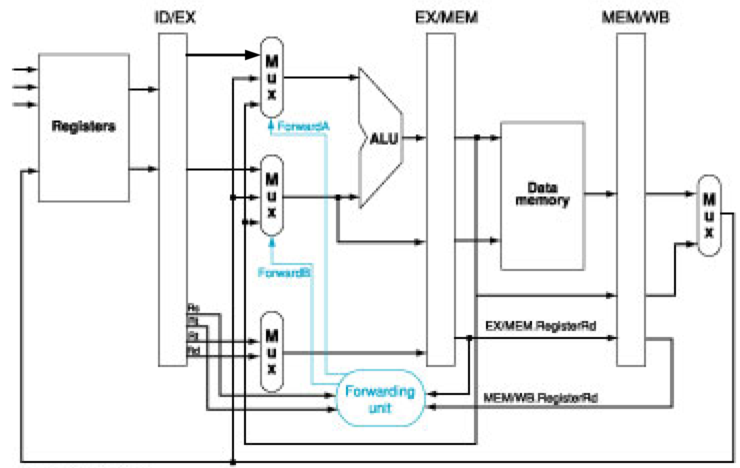
Datapath with Forwarding (Fig. 4.54, 4.55)

- The main idea (some details not shown)



a. No forwarding

Before Forwarding



b. With forwarding

After Forwarding

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

Data Hazard Detection for Forwarding (P. 322)

- Do we have to forward once the hazard condition exist?
  - No!! On what exceptions?
    - If register doesn't need to be written back
    - If the register to be written is $zero
    - When will these happen?

- **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 (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01
  if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd ≠ 0)
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01

Corrected Data Hazard Detection for Forwarding (P. 322)

- What happen if

  `EX/MEM.RegisterRd = MEM/WB.RegisterRd = ID/EX.RegisterRs(or Rt)`

  - When will this happen? (Double Data Hazard)
  - e.g.

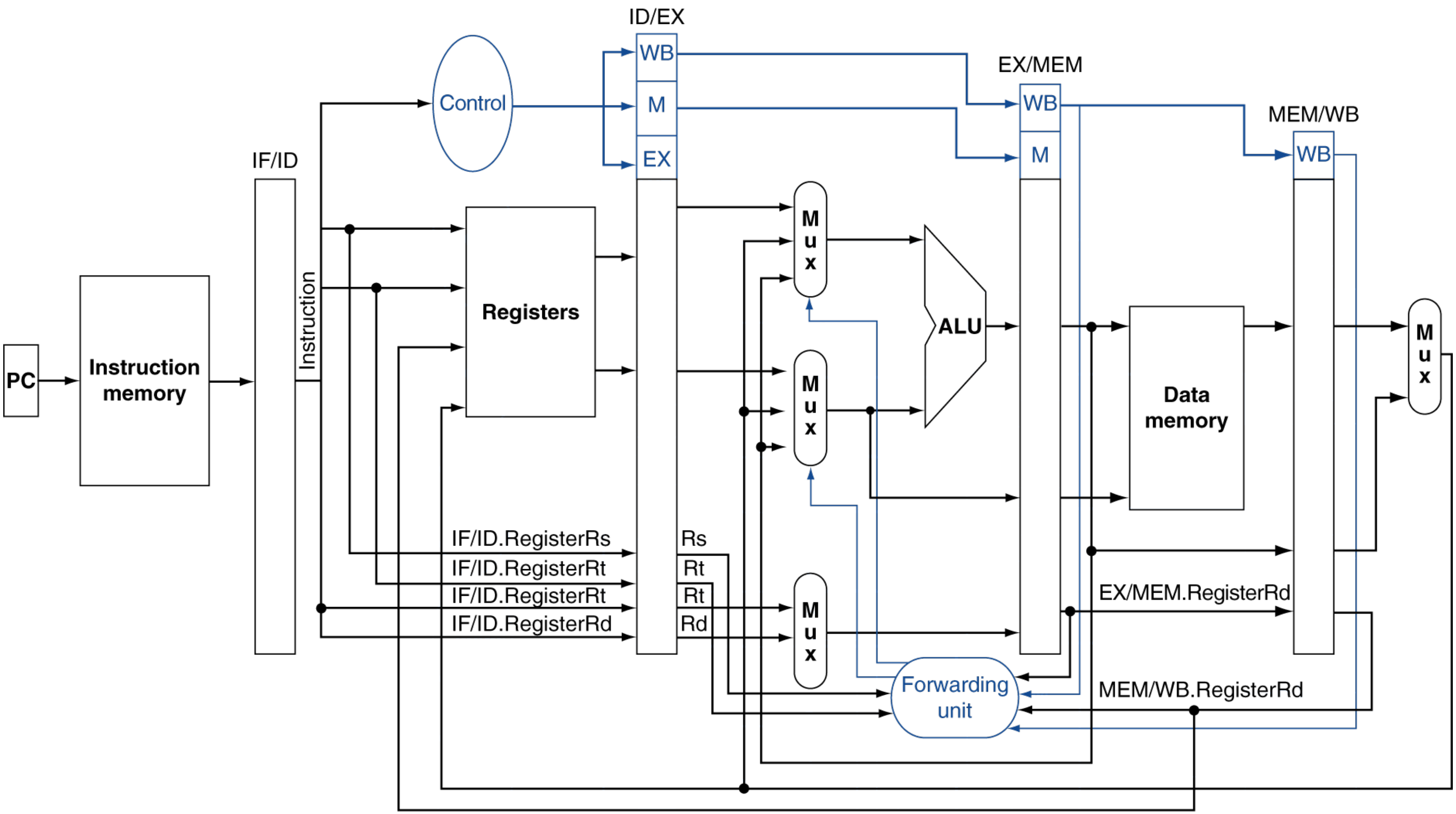
    add $1, $1, $2
    add $1, $1, $3
    add $1, $1, $4
  - Where should be the value of $1 forwarded from for add $1, $1, $4?
    - EX/MEM, because it has the more recent result.

- Corrected detection for MEM hazard

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
 and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/RX.RegisterRs)) ForwardA = 01

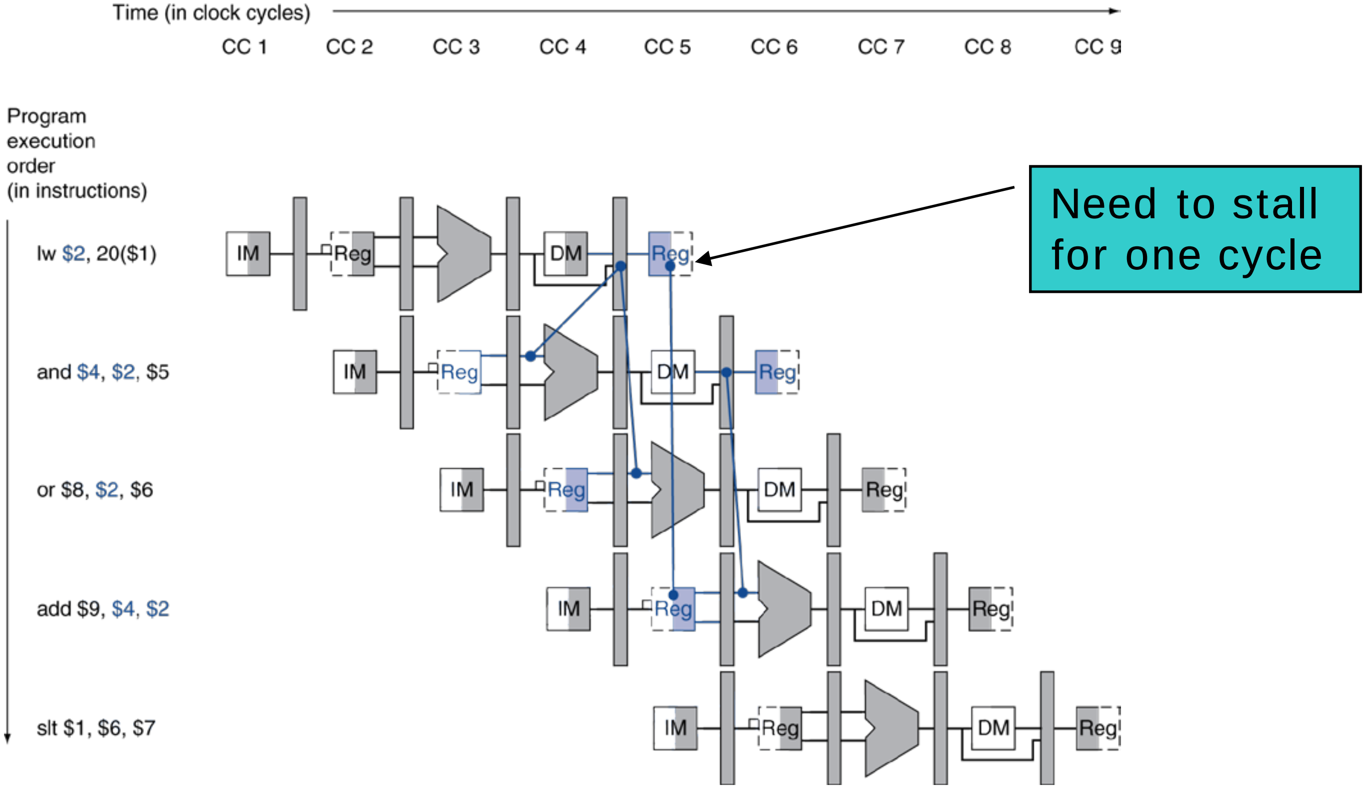
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
 and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/RX.RegisterRt)) ForwardB = 01

Datapath to Resolve Hazards via Forwarding (Fig. 4.56)



Can‘t Always Forward (p. 325, Fig. 4.58)

- 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 (include Load-Use Hazard Detection)

- Hazard detection unit
  - Operates during the ID stage.
  - Insert the stall between the load and its use.

Check if previous instruction requires Memory Read, i.e. only lw instruction

- The control

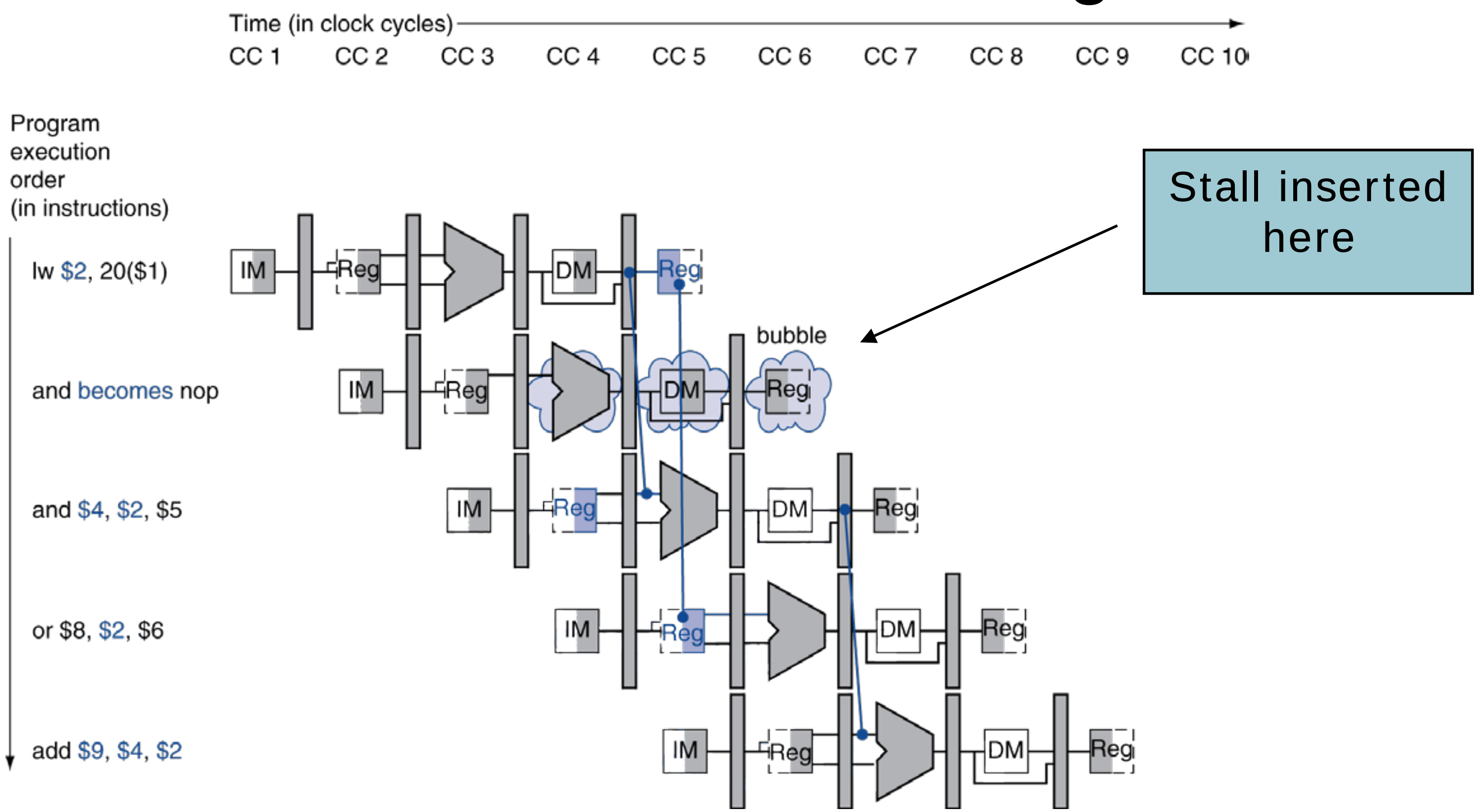
```
If (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline
```

Stall the Pipeline

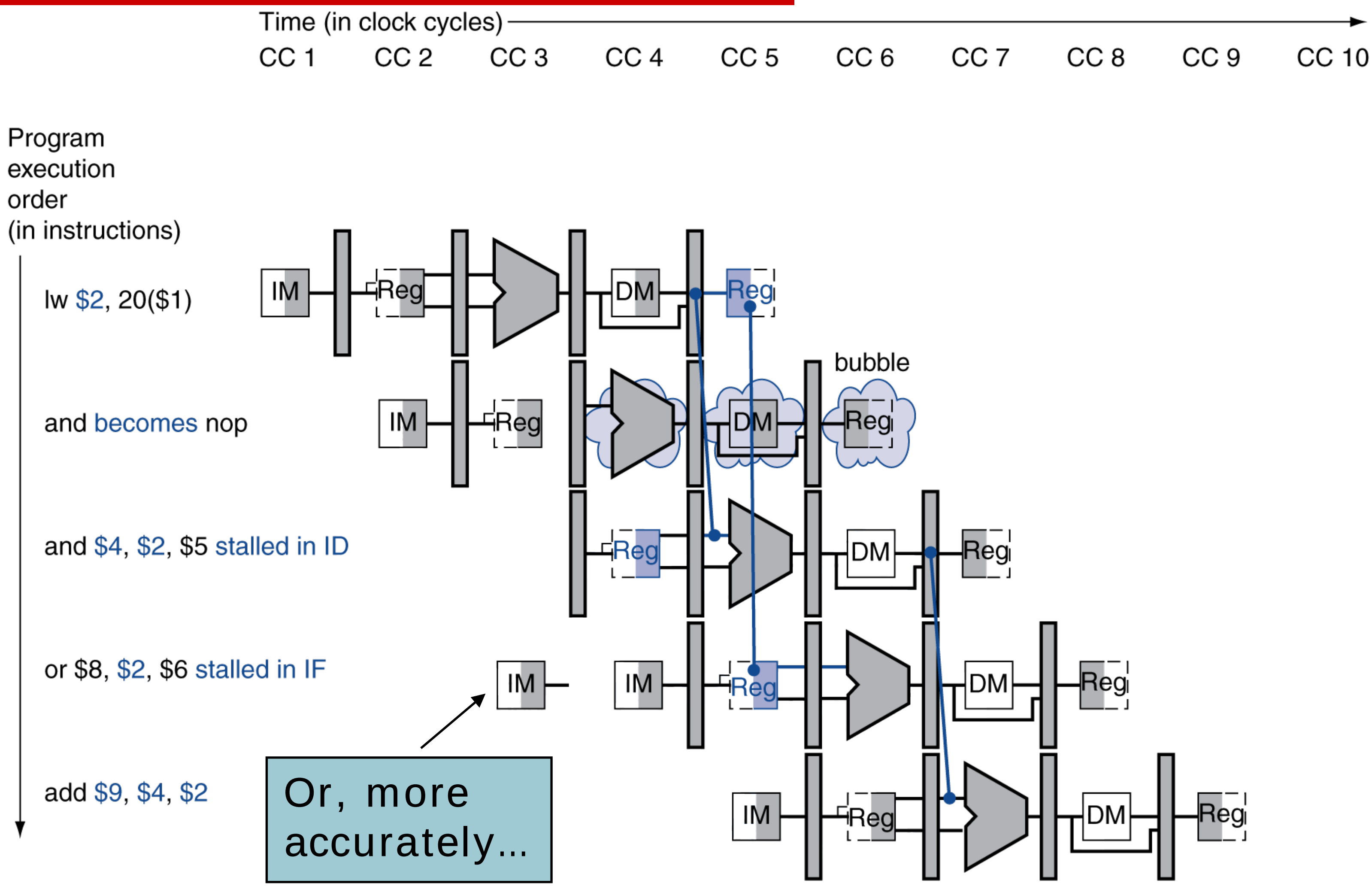
- After the 1-cycle stall, the forwarding logic can handle the dependency and execution proceeds.
- If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled.
- This is accomplished by preventing the PC register and the IF/ID pipeline register from changing.
- Deasserting all nine control signals (setting to 0s) in the Ex, MEM, and WB stages wil create a “do nothing” instruction.
- By identifying the hazards in the ID stage, we can insert a bubble into the pipeline by changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0.

Stalling (Fig. 4.59)

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



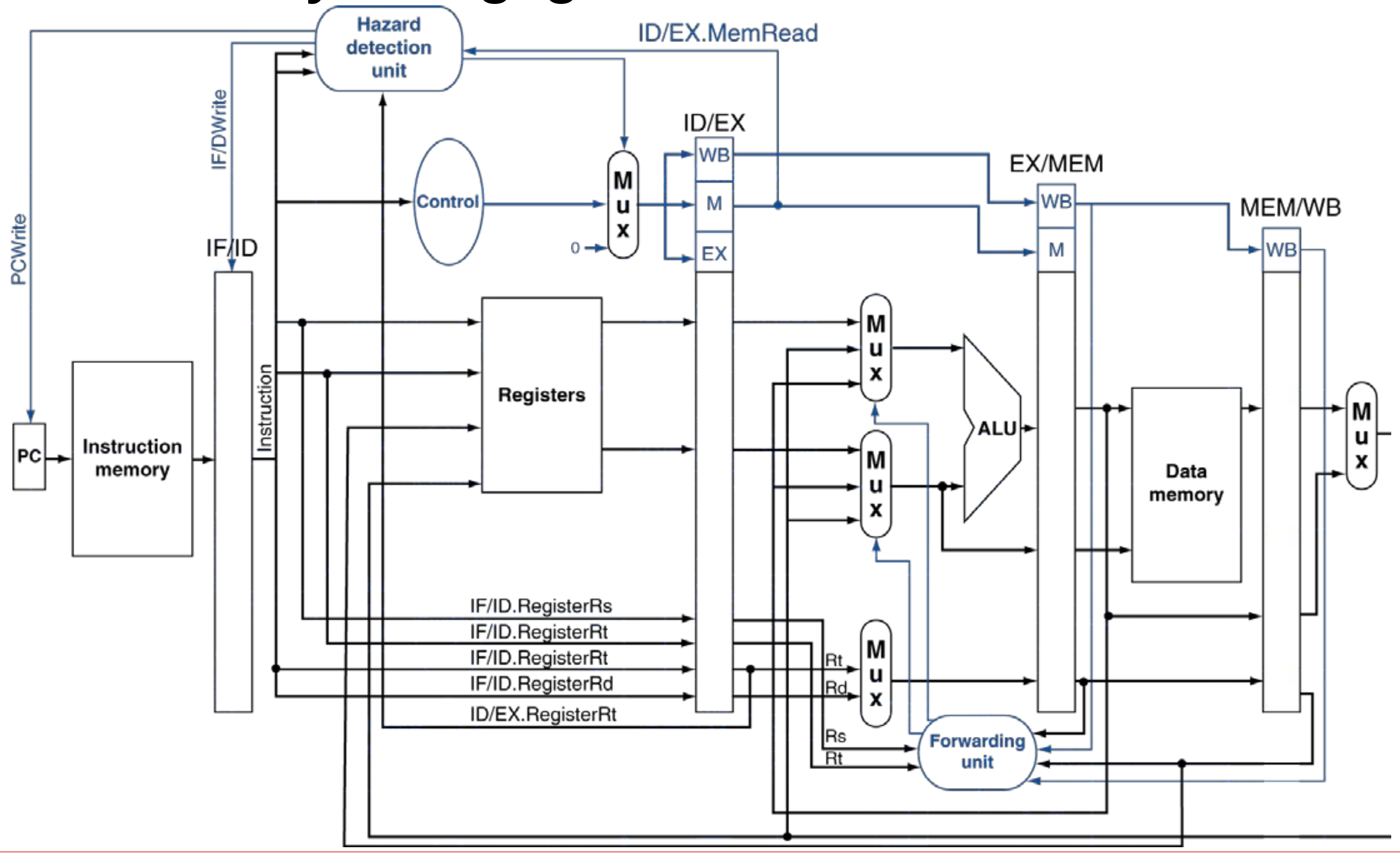
Stalling



Or, more accurately…

Hazard Detection Unit (Fig. 4.60)

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



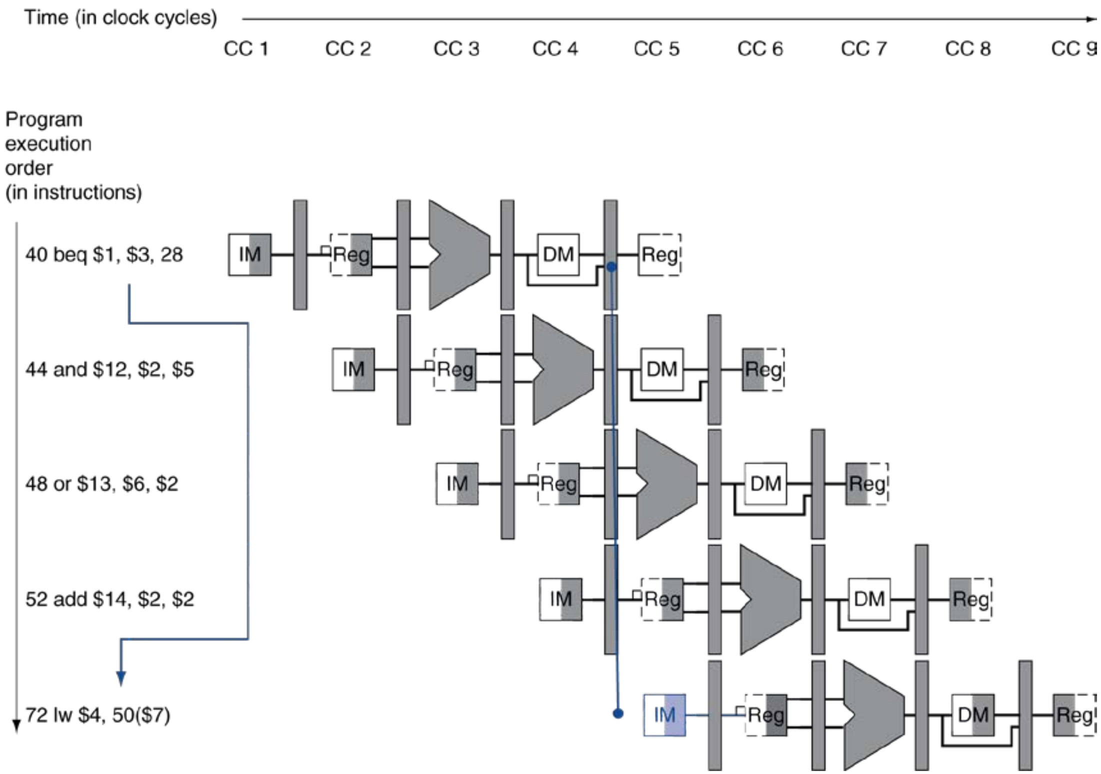
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Control Hazards (Ch. 4.9, Fig. 4.61)

- 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 (P. 330)

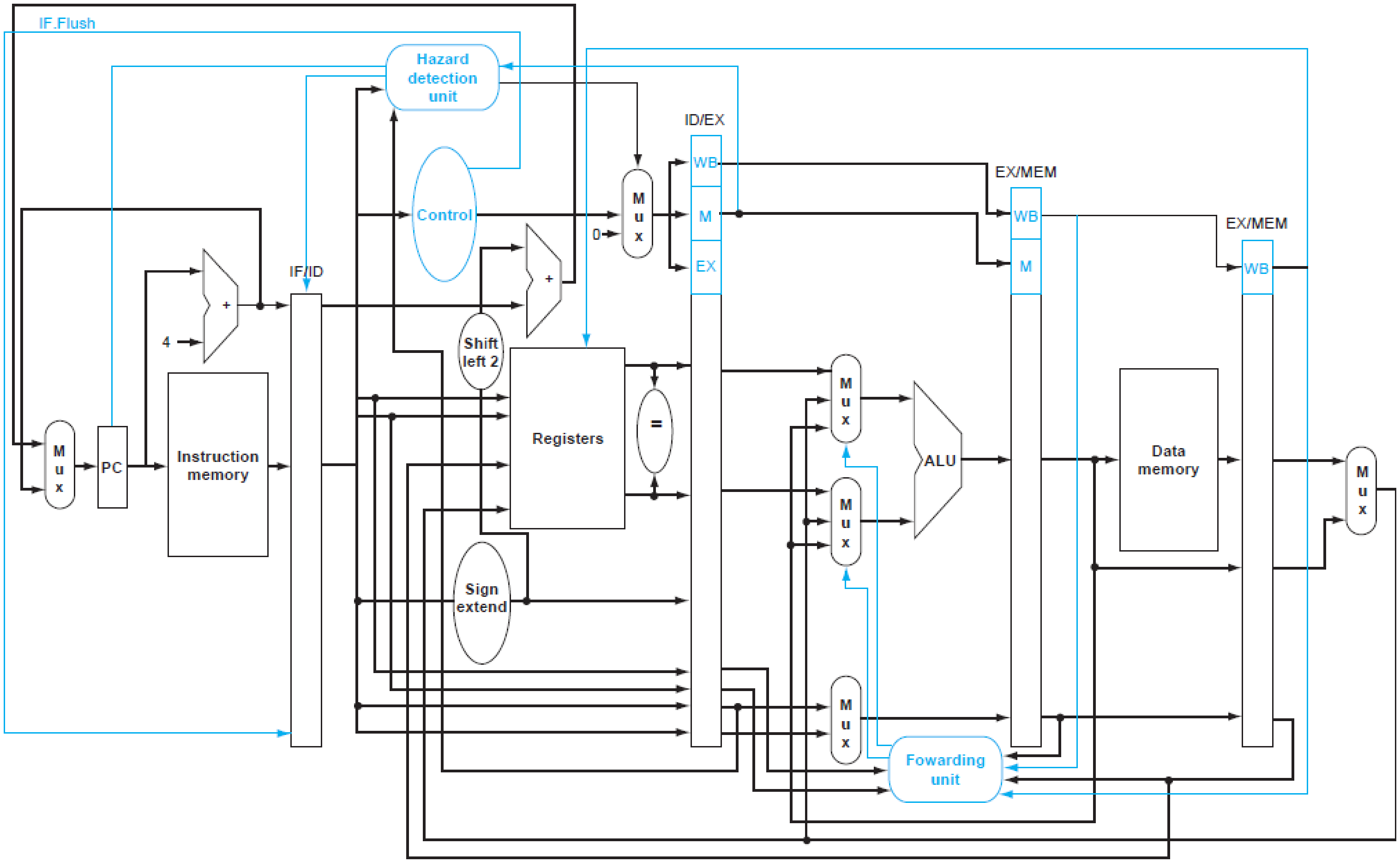
- If the branch is taken, the instructions that are being fetched, decoded and executed must be discarded, i.e. flushing instructions
- To discard (flushing) instructions, we merely change the original control values to 0's.
- We must change the three instructions in the IF, ID, and EX stages when the branch reaches the MEM stage.

⇒ So far, we need to discard following 3 instructions, when the branch is taken.

- Reducing the Delay of Branches
  - Move the branch decision to earlier stage, i.e. from MEM stage to the ID stage, then only one instruction needs to be flushed.
  - Let's borrow the similar idea as multi-cycle datapath to calculate PC-Relative address in the 2nd step. Therefore, the branch target address calculated in the ID stage.
  - Add additional HW in the ID stage to make the decision.

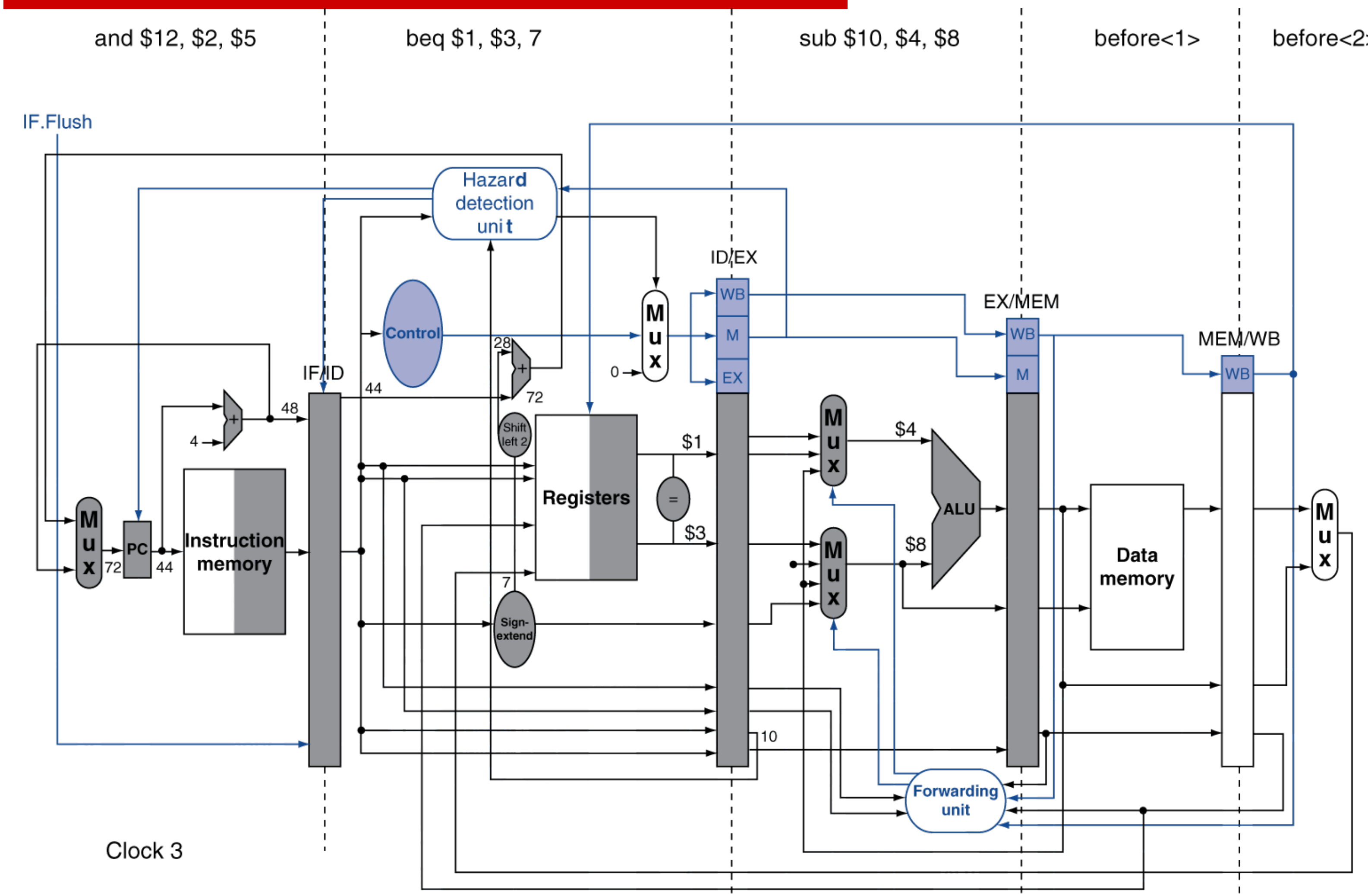
    For example, to test the equality of the two registers, first exclusive-ORing their respective bits and then ORing all the results.
  - Then we only need to flush the instruction in the IF stage by adding a control line, IF.Flush, which zeros the instruction field of the IF/ID pipeline register. i.e. change the featched instruction as "nop".
  - Additional forwarding and hazard detection HWs are required.

Flushing Instructions

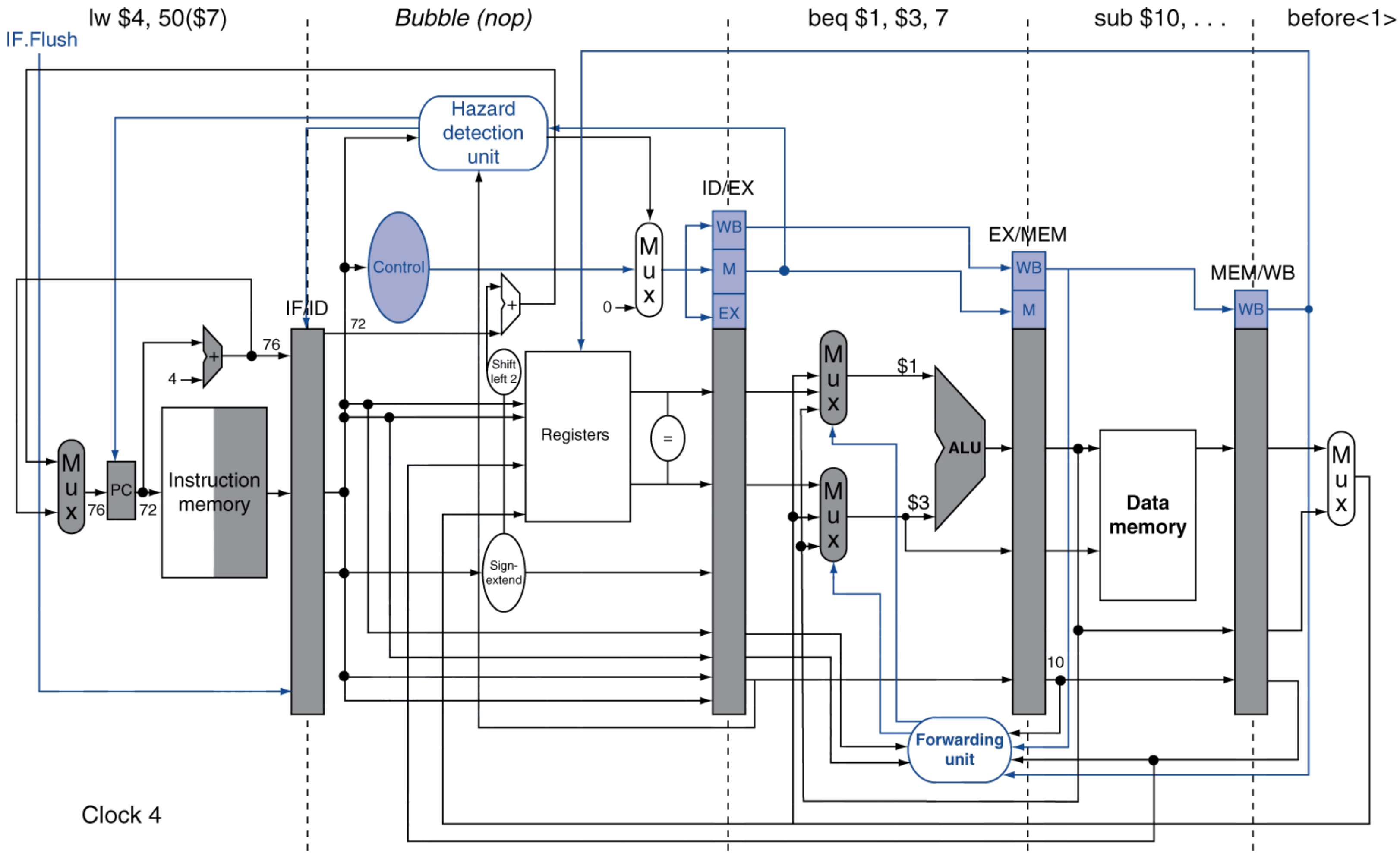


Note: we've also moved branch decision to ID stage

Example: Branch Taken (Fig. 4.62)



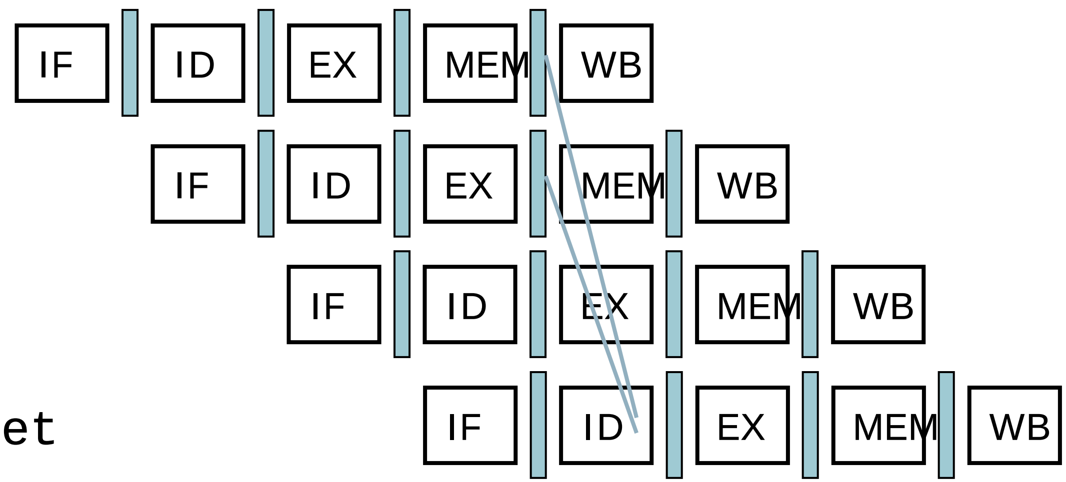
Example: Branch Taken (Fig. 4.62)



Data Hazards for Branches

- If a comparison register is a destination of 2^{nd} or 3^{rd} preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

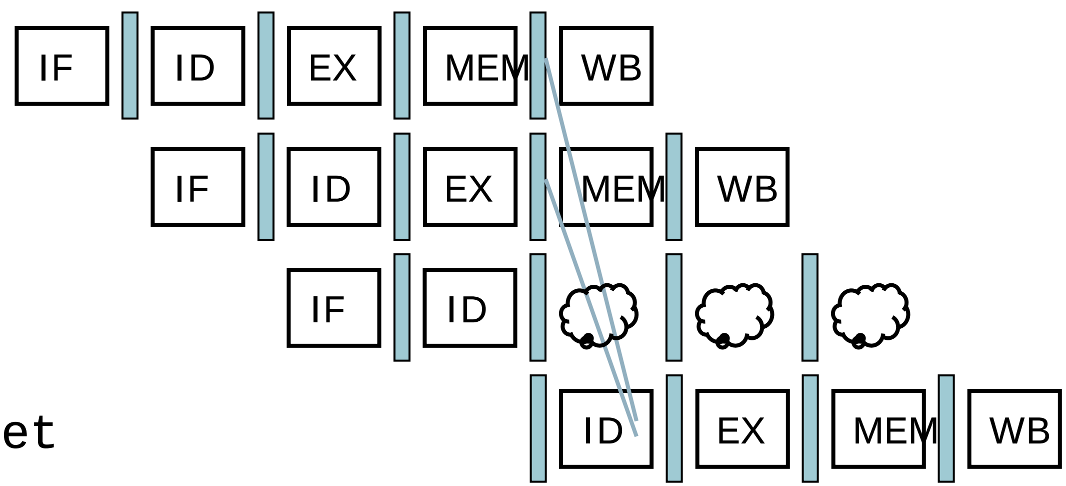


- Can resolve using forwarding

Data Hazards for Branches

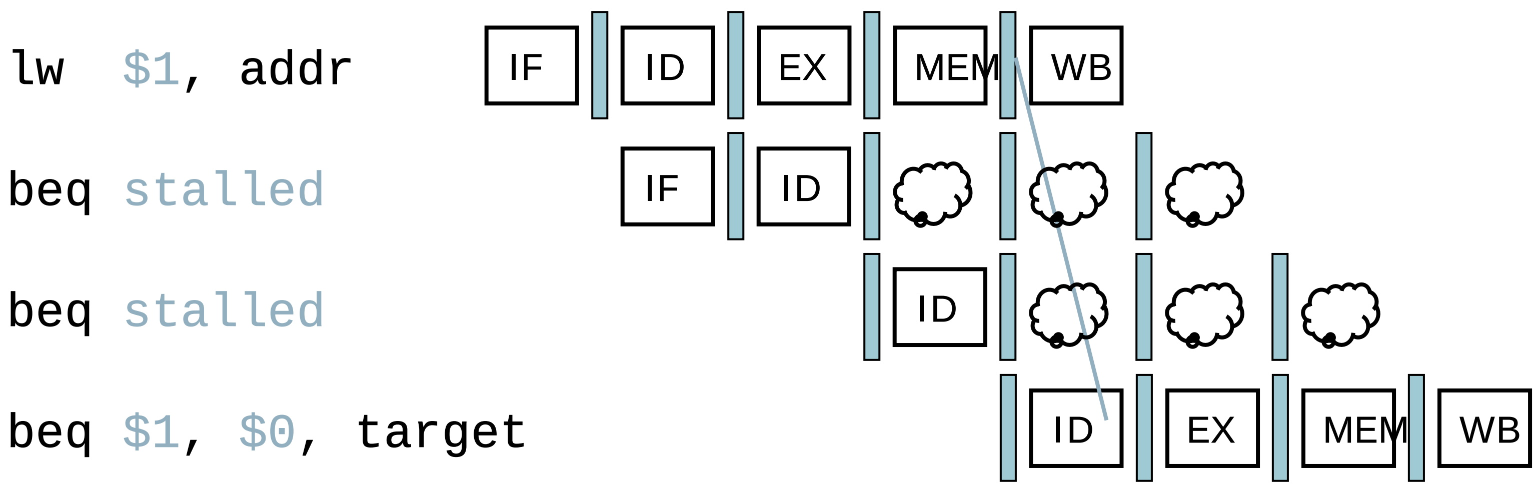
- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - ⇨ Need 1 stall cycle

```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - ⇨ Need 2 stall cycles



Dynamic Branch Prediction (P. 333)

- If the branch is taken, we have a penalty of one cycle
- For our simple design, this is reasonable
- With deeper pipelines, penalty increases and static branch prediction drastically hurts performance
- Solution: dynamic branch prediction
  - Base on the historical results on this branch instruction

Branch Prediction

- Sophisticated Techniques:
  - A “branch target buffer” to help us look up the destination
  - Correlating predictors that base prediction on global behavior and recently executed branches (e.g., prediction for a specific branch instruction based on what happened in previous branches)
  - Tournament predictors that use different types of prediction strategies and keep track of which one is performing best.
  - A “branch delay slot” which the compiler tries to fill with a useful instruction (make the one cycle delay part of the ISA)
- Branch prediction is especially important because it enables other more advanced pipelining techniques to be effective!
- Modern processors predict correctly 95% of the time!

1-Bit Predictor: Shortcoming

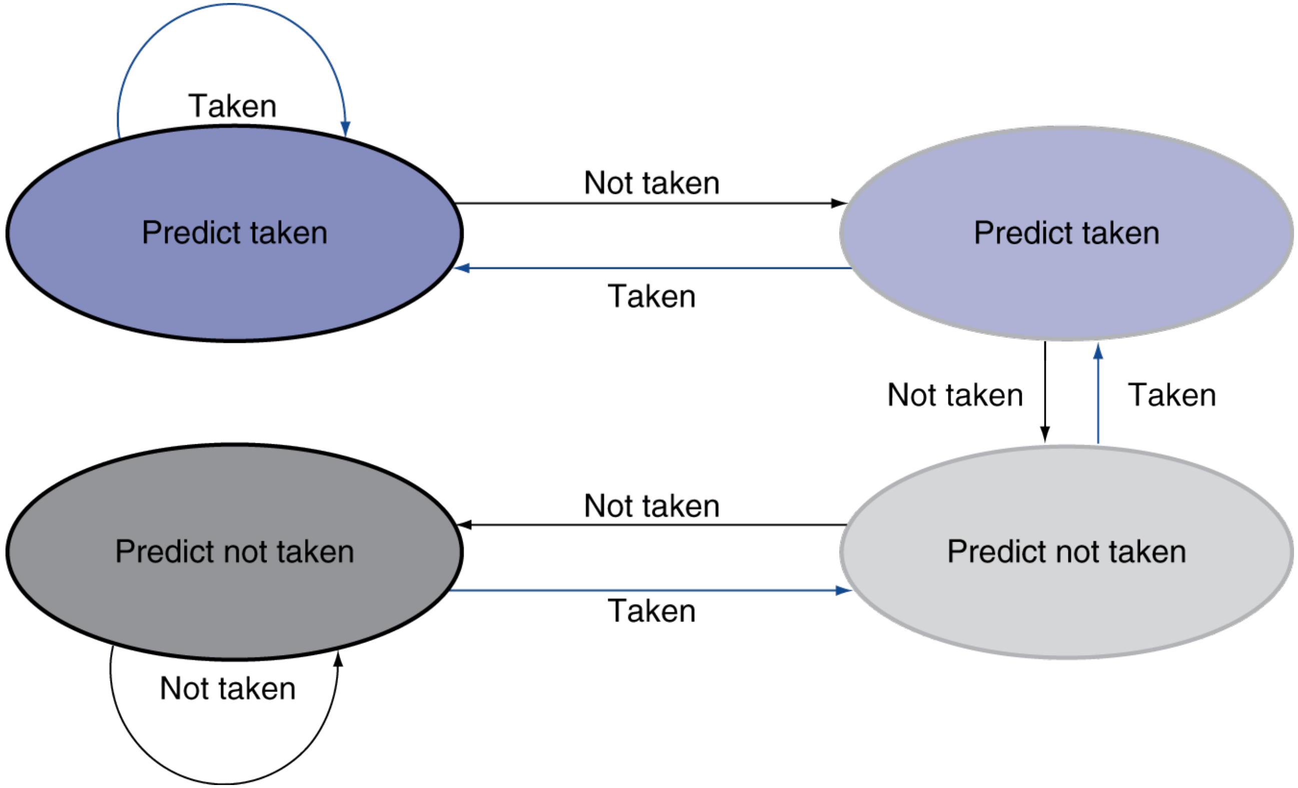
- Inner loop branches mispredicted twice!

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor (Fig. 4.63)

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

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)
```

- Dynamic Pipeline Scheduling
  - Hardware chooses which instructions to execute next
  - Will execute instructions out of order (e.g., doesn't wait for a dependency to be resolved, but rather keeps going!)
  - Speculates on branches and keeps the pipeline full (may need to rollback if prediction incorrect)

- Trying to exploit instruction-level parallelism

Performance Comparison

- Facts:
  - Functional unit times
    - Memory Access: 200ps
    - ALU operation: 100ps
    - Register file read/write: 50ps
  - A program which consists of:
    - 25% loads
    - 10% stores
    - 11% branches
    - 2% jumps
    - 52% ALU instructions
  - For pipelined design
    - 50% of loads are immediately followed by an instruction that uses the result, which cause additional one cycle of stall
    - 25% of branches are mis-predictions, which cause 1 cycle of branch delay.
    - All jumps cause 1 full cycle of delay.
- Single-cycle Performance
  - Cycle time = 200 + 50 + 100 + 200 + 50 = 600ps (CPI = 1)
- Pipelined Performance
  - Cycle time = 200ps
  - CPI = 0.25x1.5 + 0.1x1 + 0.11x1.25 + 0.02x2 + 0.52x1 = 1.17

Advanced Pipelining

- Increase the depth of the pipeline
- Start more than one instruction each cycle (multiple issue)
- Loop unrolling to expose more ILP (better scheduling)
- "Superscalar" processors
  - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
- All modern processors are superscalar and issue multiple instructions usually with some limitations (e.g., different "pipes")
- VLIW: very long instruction word, static multiple issue (relies more on compiler technology)
- This class has given you the background you need to learn more!

Fallacies (Ch. 4.15)

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven’t we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks (Ch. 4.16)

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism but latency not reduced
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall