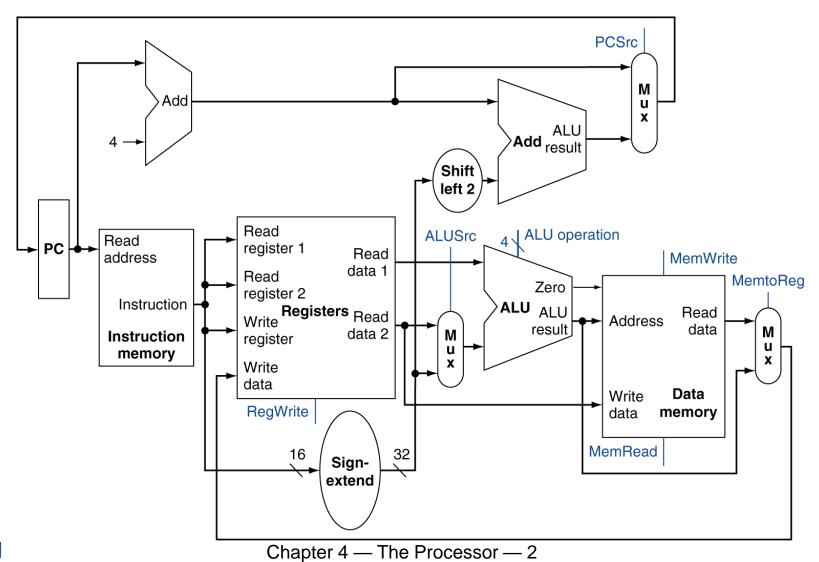
# Computer Architecture Chapter 4: The Processor Part 2



Adapted from Computer Organization the Hardware/Software Interface – 5th



# Single-cycle Datapath



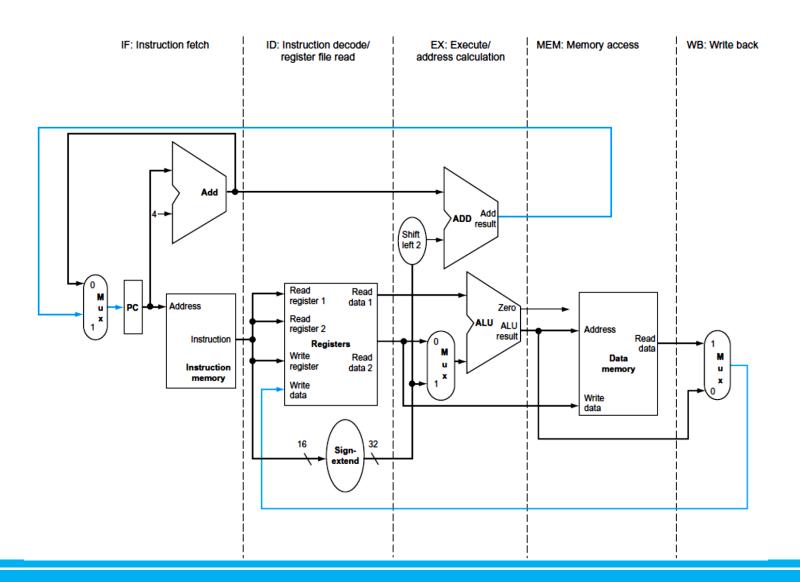


# (Multi-cycle) Pipelined Datapath

- Up to five instruction will be in execution in one clock cycle
- Separate the datapath into five pieces:
  - IF: Instruction fetch
  - Instruction decode and register file read
  - EX: Execution and address calculation
  - MEM: Data memory access
  - WB: Write back

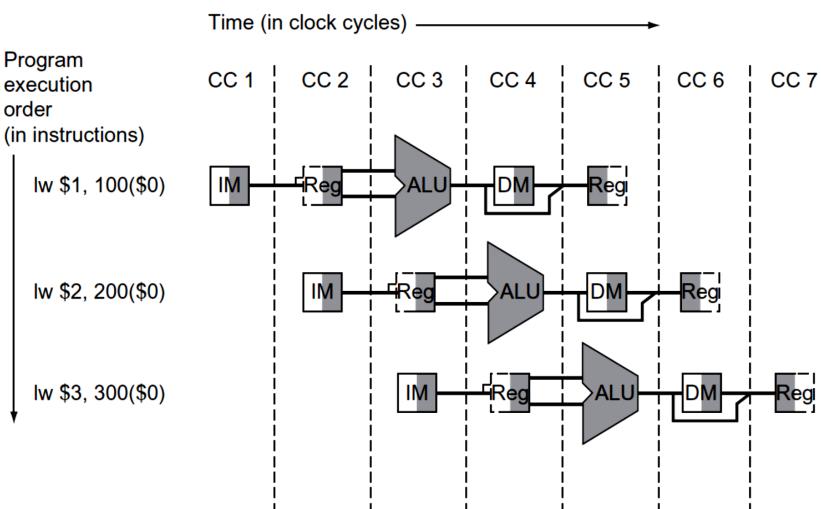


# Five Stage Datapath





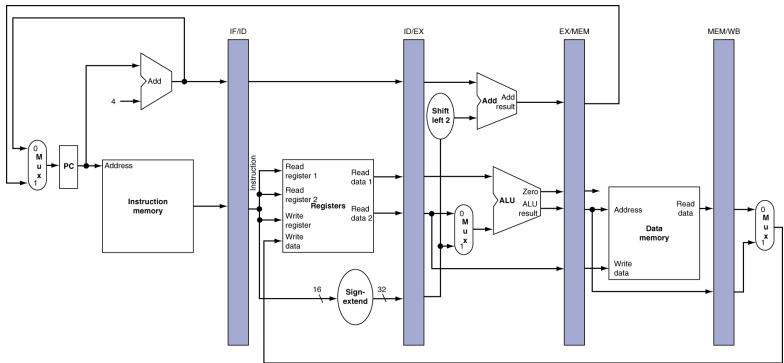
#### Instructions Execution





#### Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle



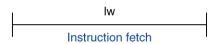


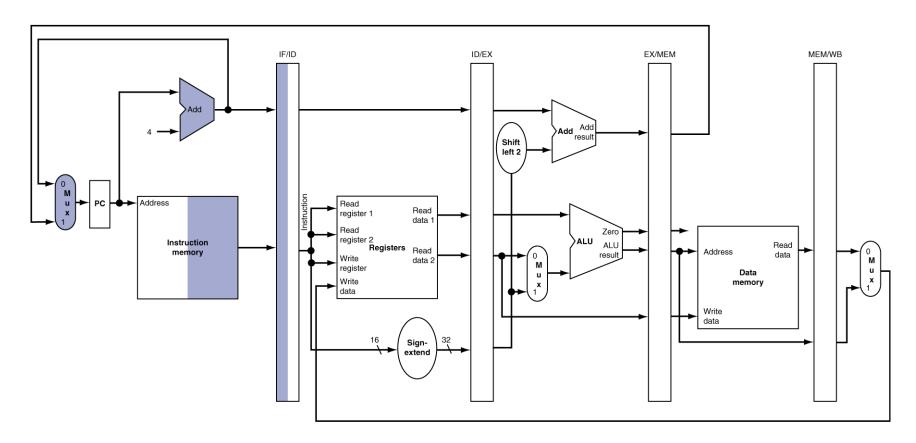
# Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store



## IF for Load, Store, ...

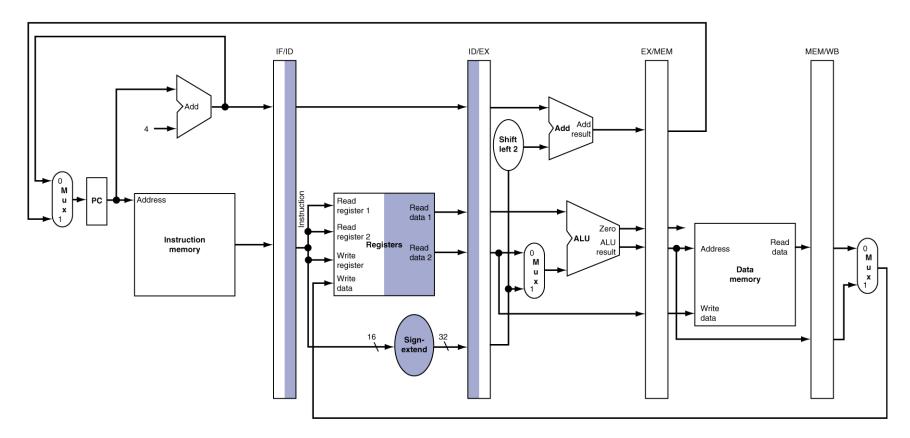






# ID for Load, Store, ...

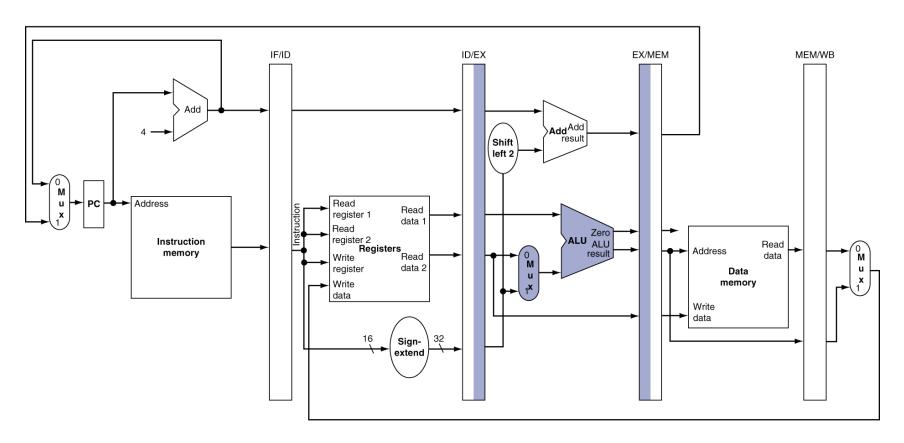






#### **EX for Load**

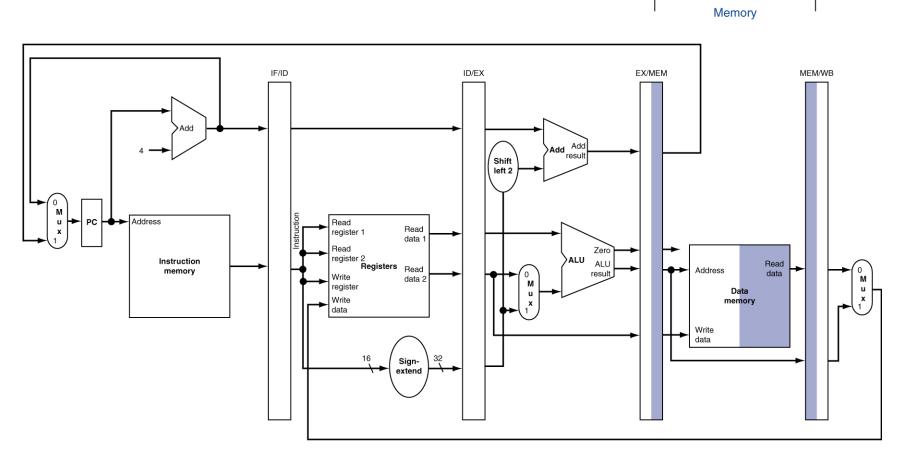






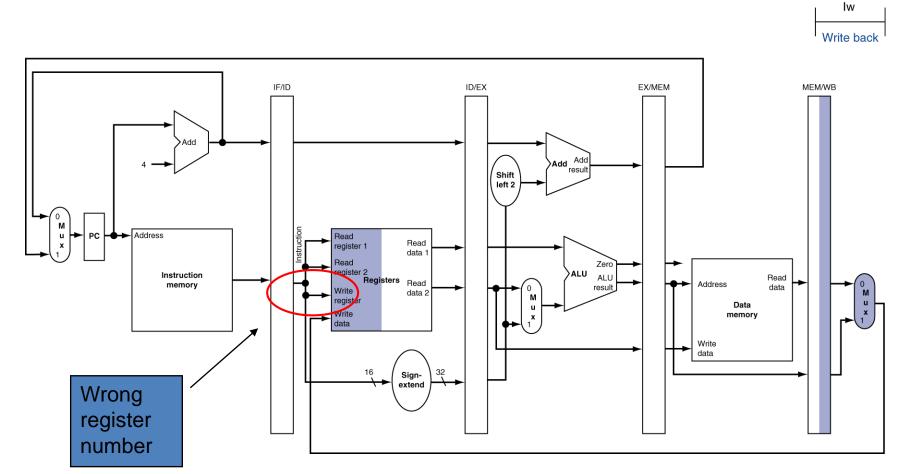
#### MEM for Load

lw



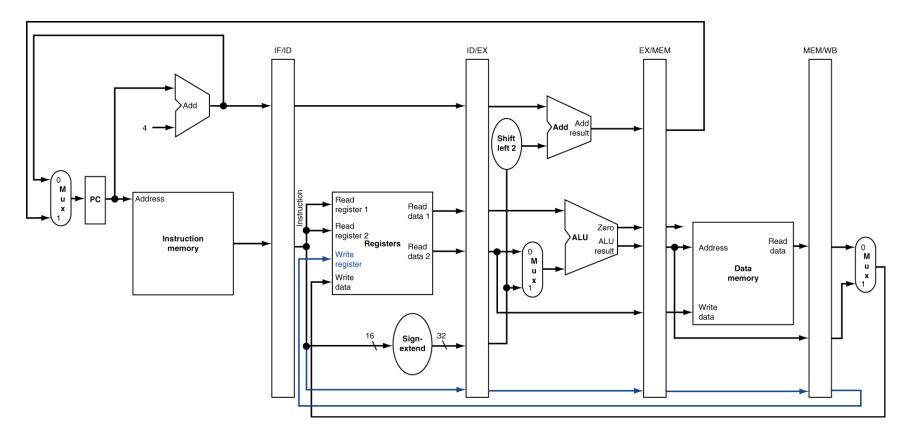


#### WB for Load





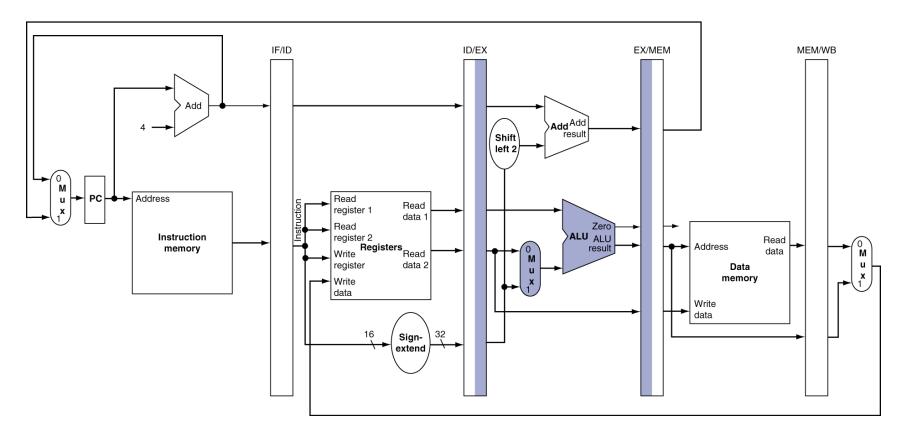
# Corrected Datapath for Load





#### **EX for Store**

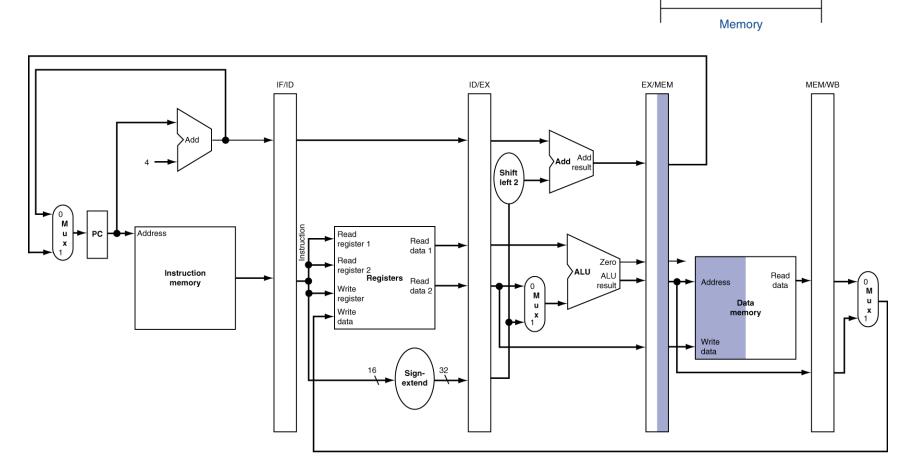






#### **MEM for Store**

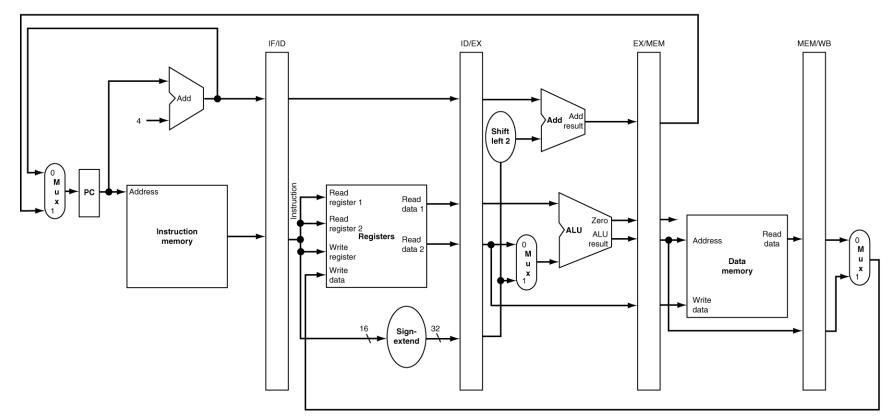
SW





#### WB for Store



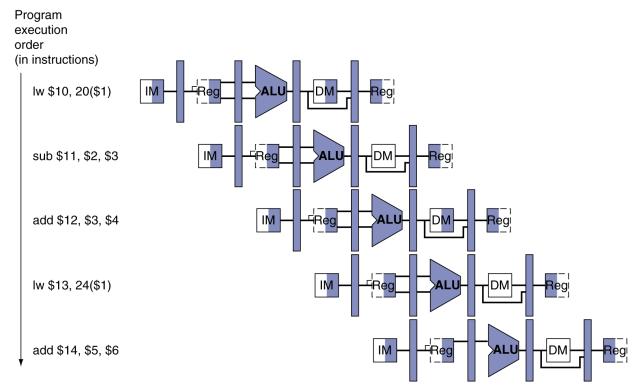




# Multi-Cycle Pipeline Diagram

Form showing resource usage

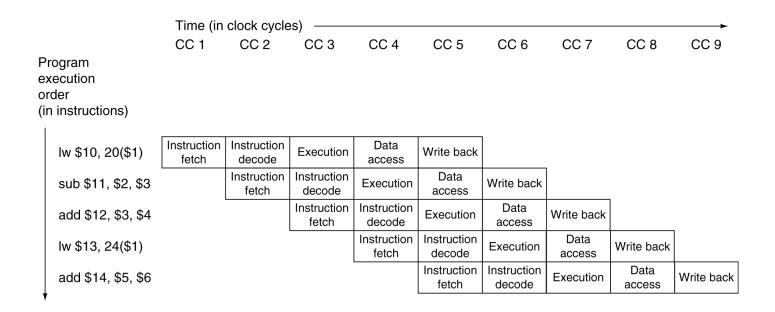






# Multi-Cycle Pipeline Diagram

#### Traditional form

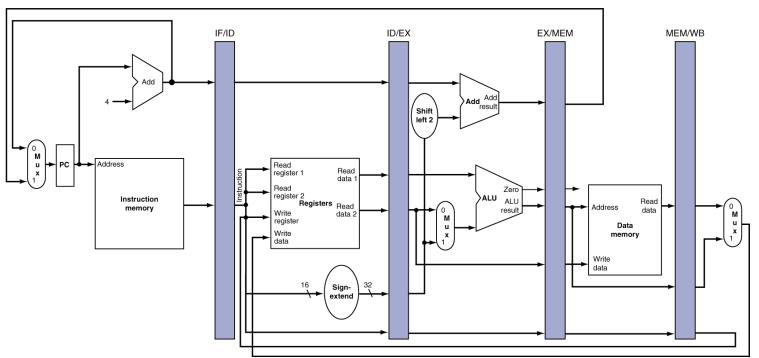




# Single-Cycle Pipeline Diagram

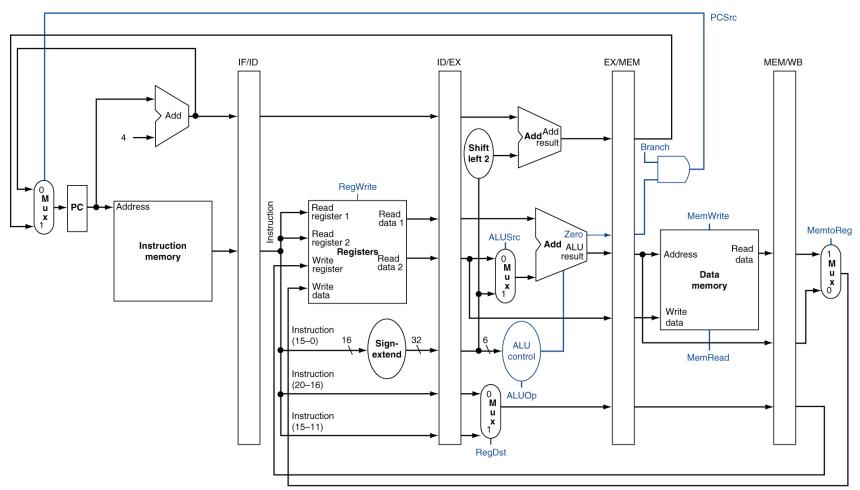
State of pipeline in a given cycle







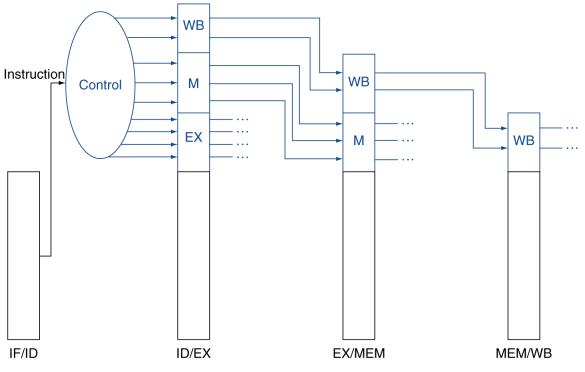
# Pipelined Control (Simplified)





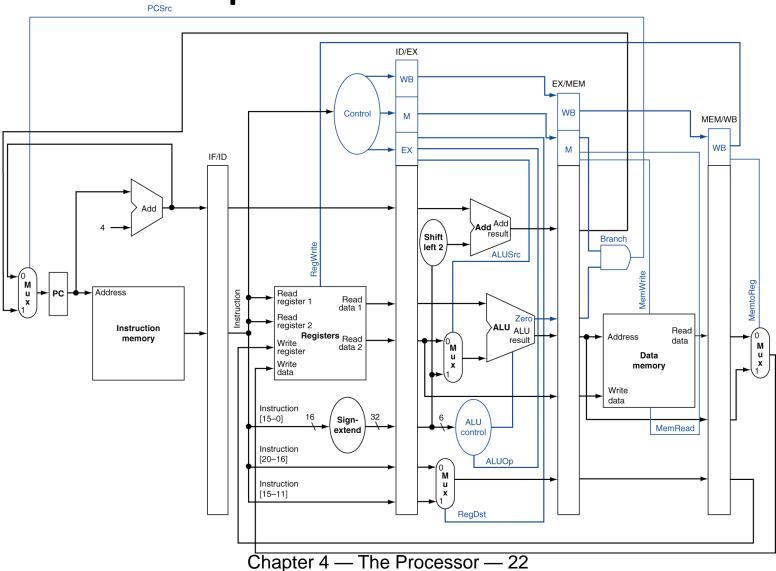
# **Pipelined Control**

- Control signals derived from instruction
  - As in single-cycle implementation





# **Pipelined Control**





#### Data Hazards in ALU Instructions

Consider this sequence:

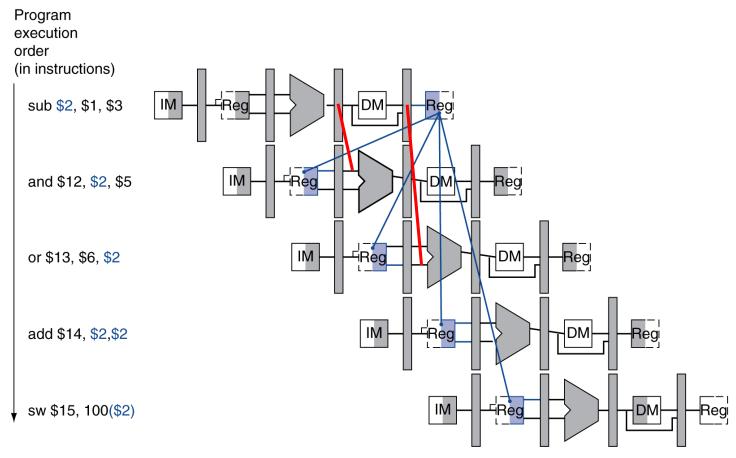
```
sub $2, $1,$3
and $12,$2,$5
or $13,$6,$2
add $14,$2,$2
sw $15,100($2)
```

- We can resolve hazards with forwarding
  - How do we detect when to forward?



#### Dependencies & Forwarding

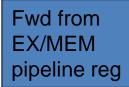
Time (in clock cycles) CC9 CC<sub>1</sub> CC<sub>2</sub> CC3 CC 4 CC 5 CC<sub>6</sub> CC 7 CC8 Value of register \$2: -20 10 10 10 10 10/-20 -20 -20 -20





#### Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  - 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



Fwd from MEM/WB pipeline reg

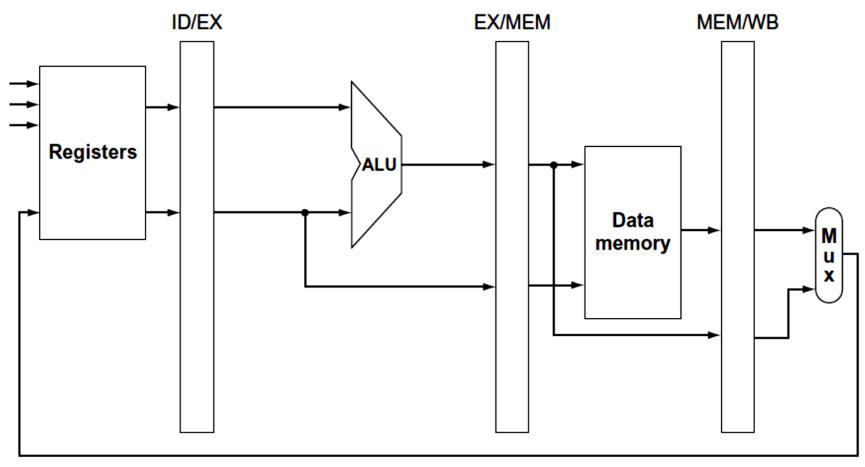


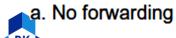
#### Detecting the Need to Forward

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

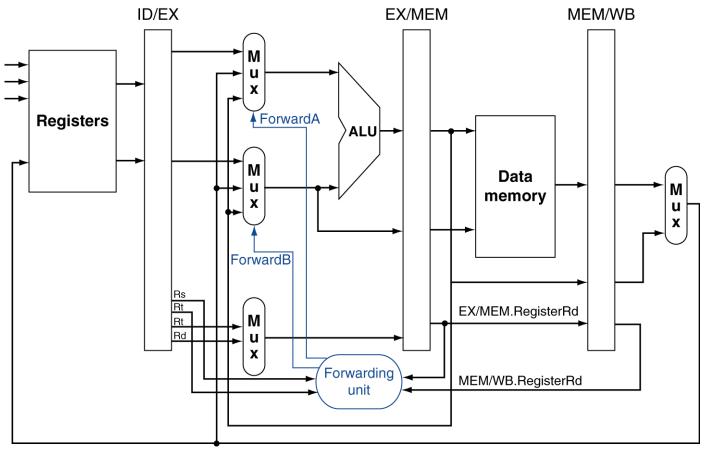


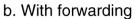
# No Forwarding Paths





# Forwarding Paths







# **Forwarding Control Values**

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.



#### Forwarding 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 (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



#### **Double Data Hazard**

Consider the sequence:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true



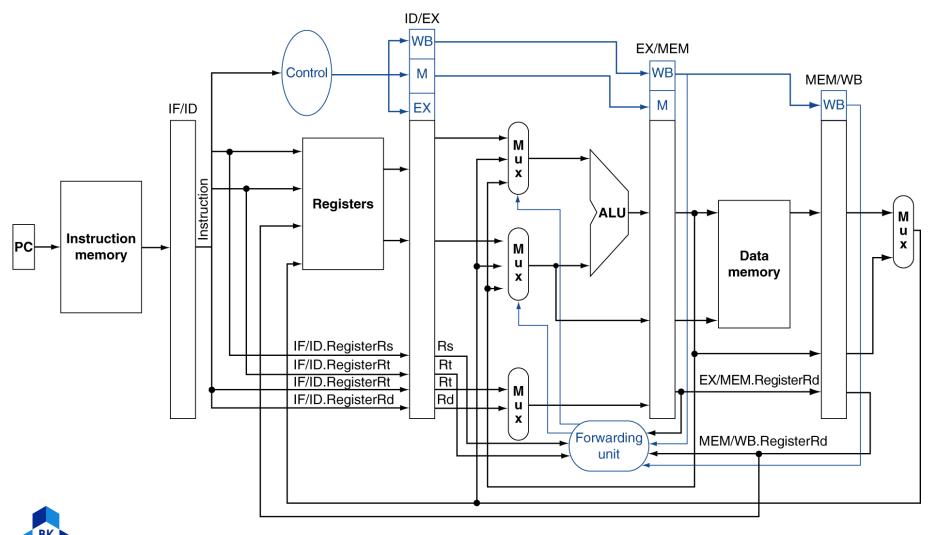
#### Revised Forwarding Condition

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/EX.RegisterRs))
   ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
           and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
   ForwardB = 01
```



# Datapath with Forwarding



#### Load-Use Data Hazard



Program execution order (in instructions) Need to stall lw \$2, 20(\$1) DM for one cycle IM and \$4, \$2, \$5 P.Reg Reg or \$8, \$2, \$6 add \$9, \$4, \$2 slt \$1, \$6, \$7



#### Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble



# How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for \( \frac{1}{V} \)
    - Can subsequently forward to EX stage



# Stall/Bubble in the Pipeline



execution order (in instructions)

Iw \$2, 20(\$1)

IM Heg

IM Reg

And becomes nop

and \$4, \$2, \$5

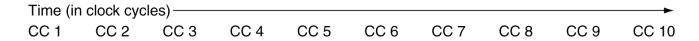
or \$8, \$2, \$6

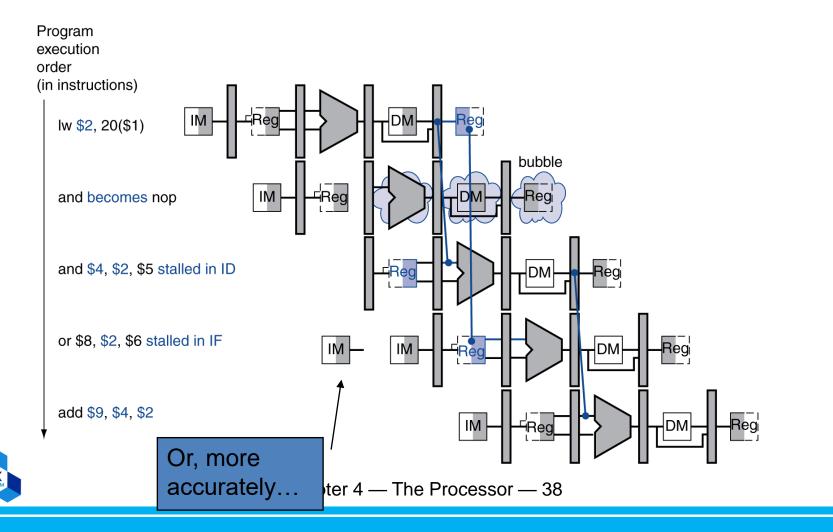


Program

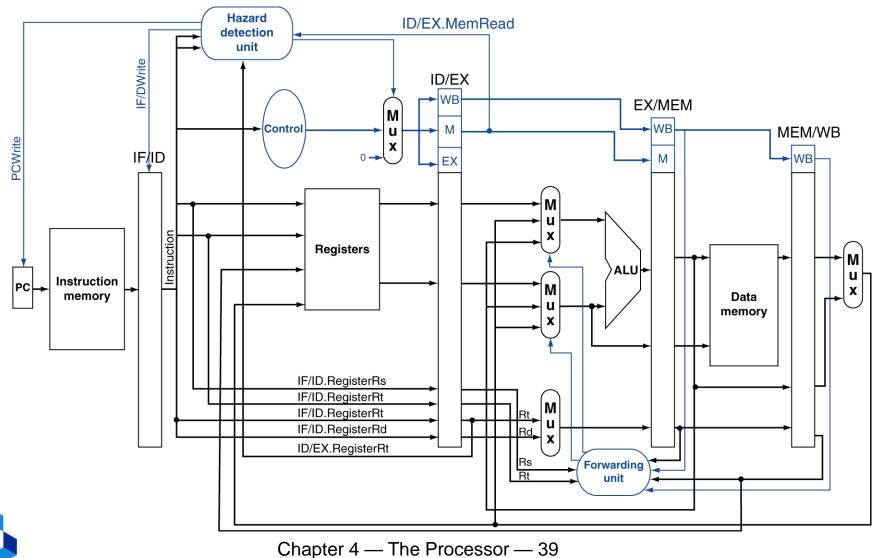
add \$9, \$4, \$2

# Stall/Bubble in the Pipeline





#### Datapath with Hazard Detection





#### Stalls and Performance

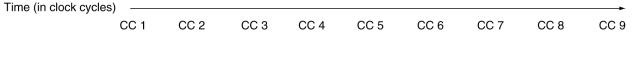
#### **The BIG Picture**

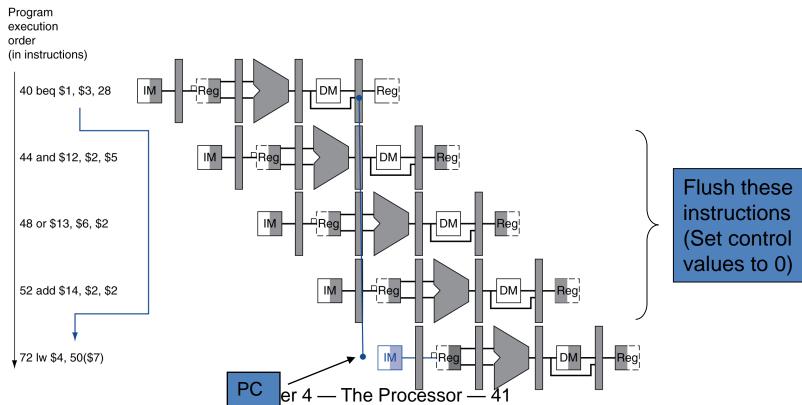
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure



#### **Branch Hazards**

If branch outcome determined in MEM







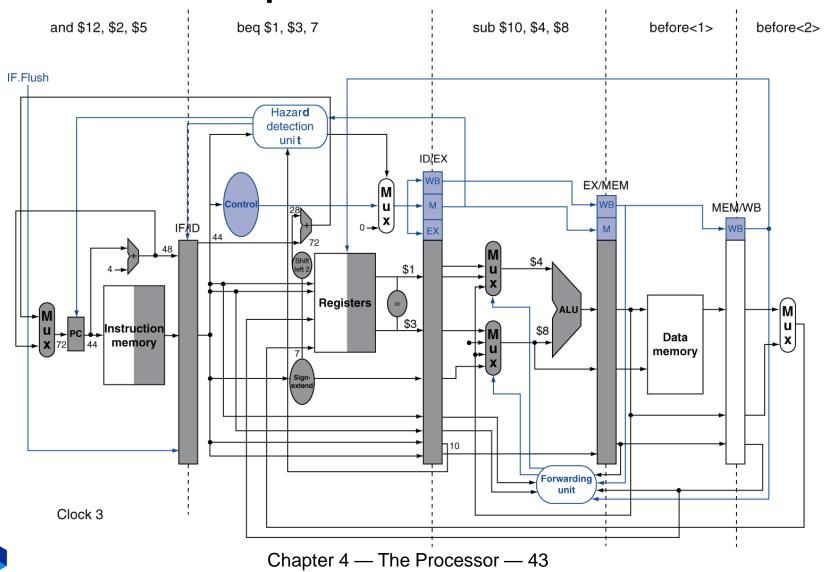
### Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

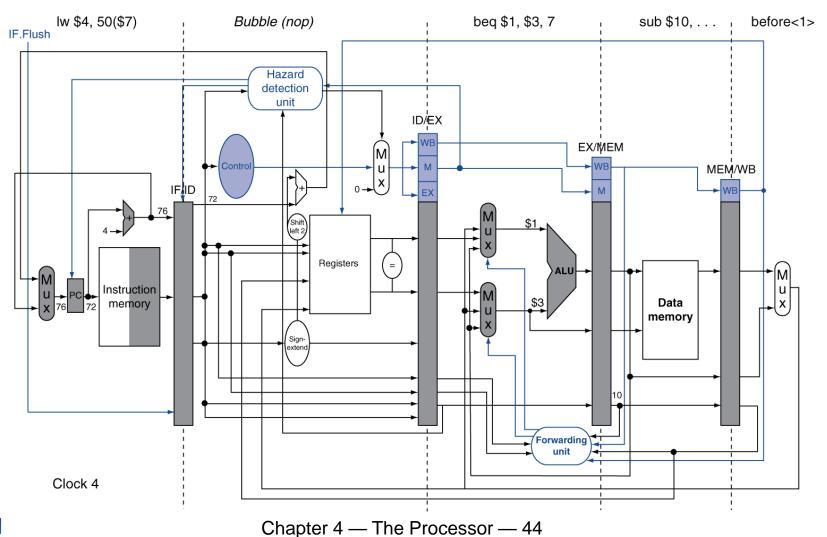
```
36: sub $10, $4, $8
40: beq $1, $3, 7
44: and $12, $2, $5
48: or $13, $2, $6
52: add $14, $4, $2
56: slt $15, $6, $7
...
72: lw $4, 50($7)
```



# Example: Branch Taken



### Example: Branch Taken





#### Data Hazards for Branches

 If a comparison register is a destination of 2<sup>nd</sup> or 3<sup>rd</sup> preceding ALU instruction

Can resolve using forwarding



#### Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2<sup>nd</sup> preceding load instruction
  - Need 1 stall cycle



#### Data Hazards for Branches

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



### Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction



# 1-Bit Predictor: Shortcoming

• Inner loop branches mispredicted twice!

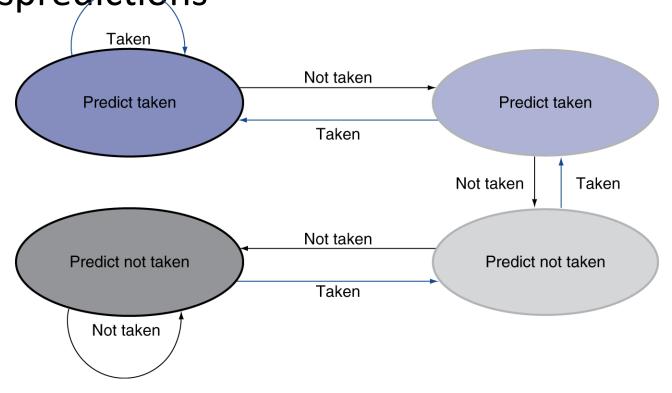
```
outer: ...
inner: ...
beq ..., ..., inner
...
beq ..., outer
```

- 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

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



#### **Exceptions and Interrupts**

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, ...
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard



### Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CPO)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180



#### An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - ...: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler



#### **Handler Actions**

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, ...

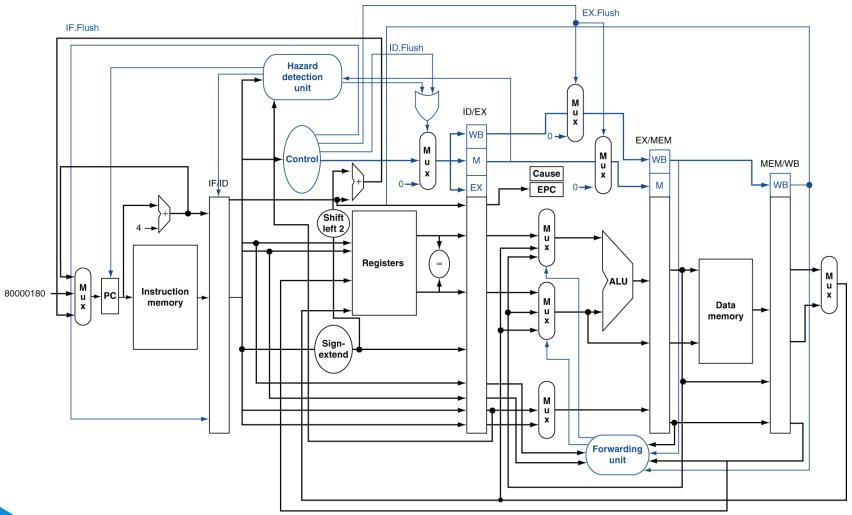


### Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage add \$1, \$2, \$1
  - Prevent \$1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware



## Pipeline with Exceptions





#### **Exception Properties**

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust



### **Exception Example**

Exception on add in

```
40 sub $11, $2, $4
44 and $12, $2, $5
48 or $13, $2, $6
4C add $1, $2, $1
50 slt $15, $6, $7
54 lw $16, 50($7)
```

...

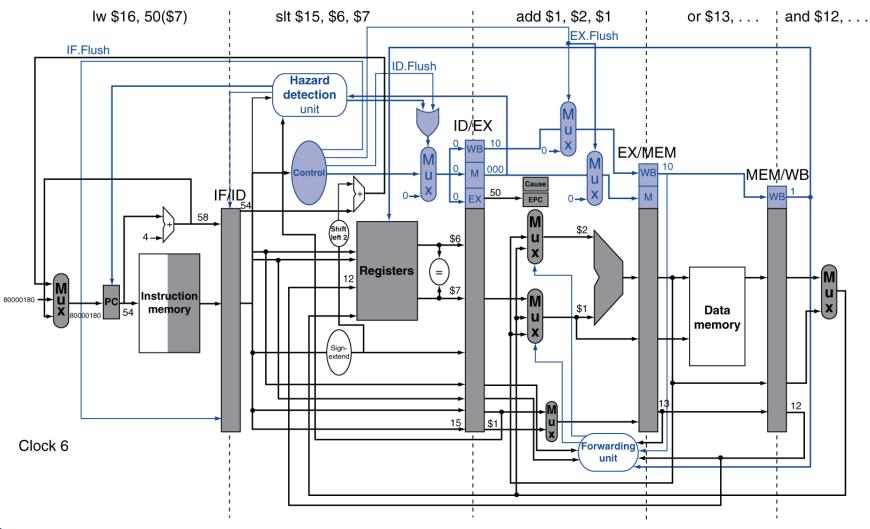
Handler

```
80000180 sw $25, 1000($0)
80000184 sw $26, 1004($0)
```

...

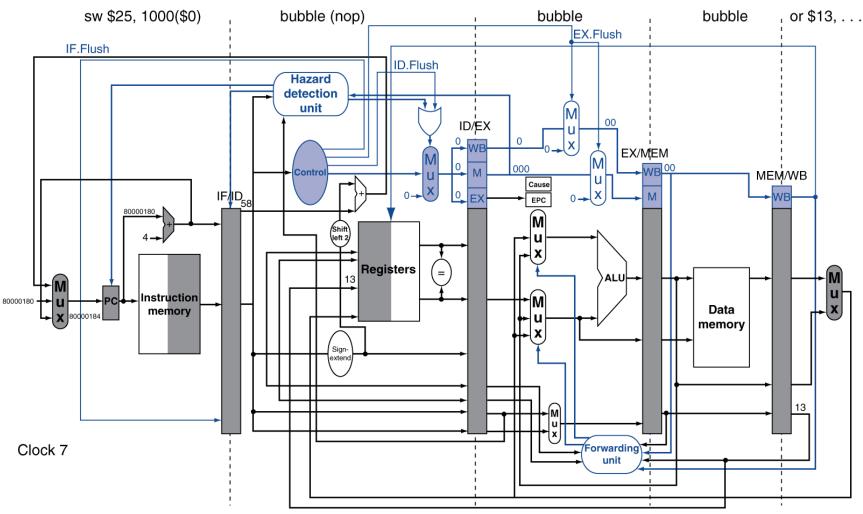


### **Exception Example**





### **Exception Example**





### Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - "Precise" exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!



#### Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require "manual" completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

