Computer Organization

Lecture 18 - Pipelined Processor Design 2

Reading: 4.7-4.9

Pipelined Processor Design

- Datapath
- Control

- Dealing with Hazards & Forwarding
- Branch Prediction
- Exceptions
- Performance

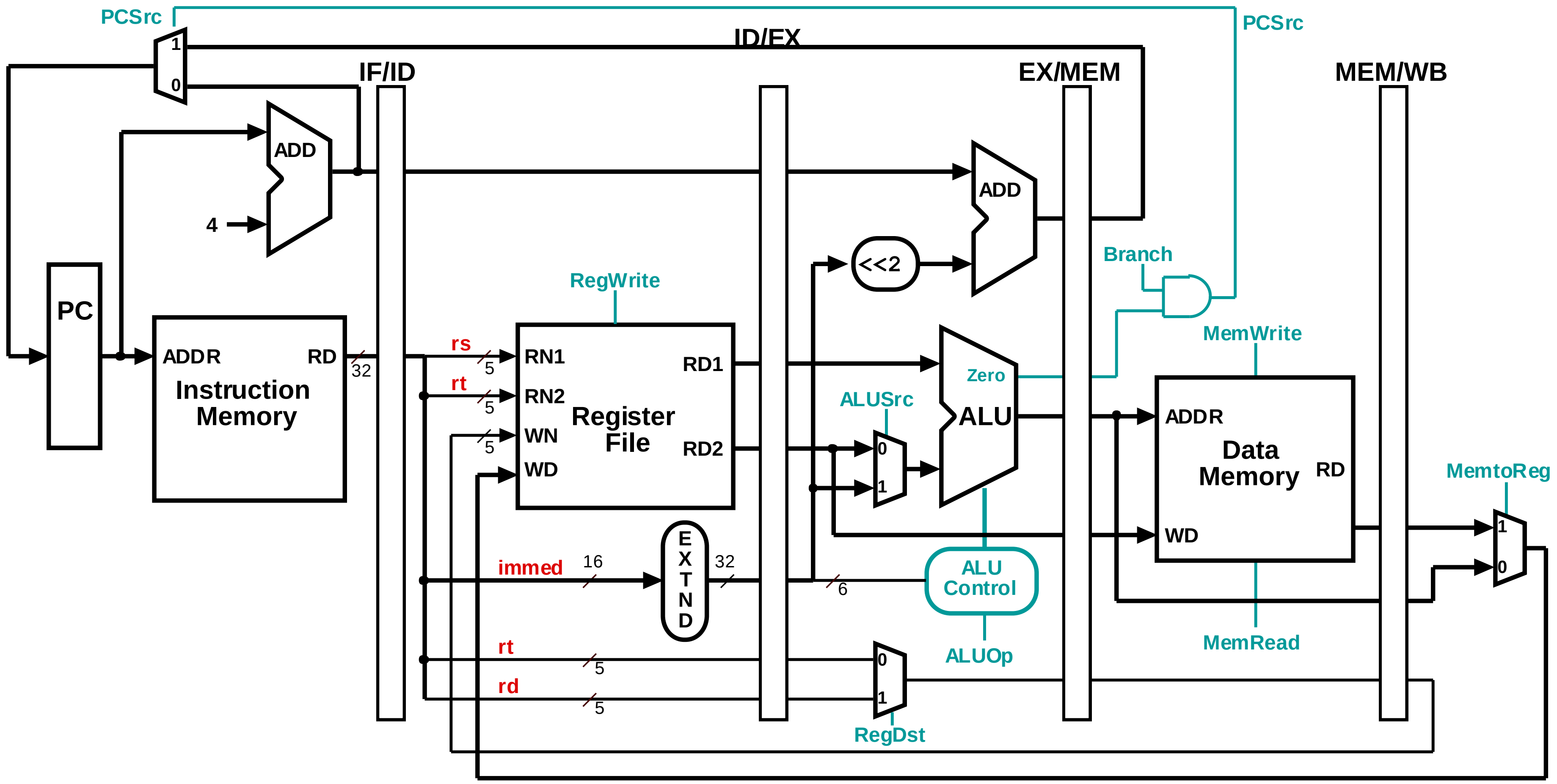
Pipelining in MIPS*

- MIPS architecture was designed to be pipelined
  - Simple instruction format (makes IF, ID easy)
    - Single-word instructions
    - Small number of instruction formats
    - Common fields in same place (e.g., rs, rt) in different formats
  - Memory operations only in lw, sw instructions (simplifies EX)
  - Memory operands aligned in memory (simplifies MEM)
  - Single value for writeback (limits forwarding)

- Pipelining is harder in CISC architectures like x86

*Original Acronym: Microprocessor without Interlocked Pipe Stages

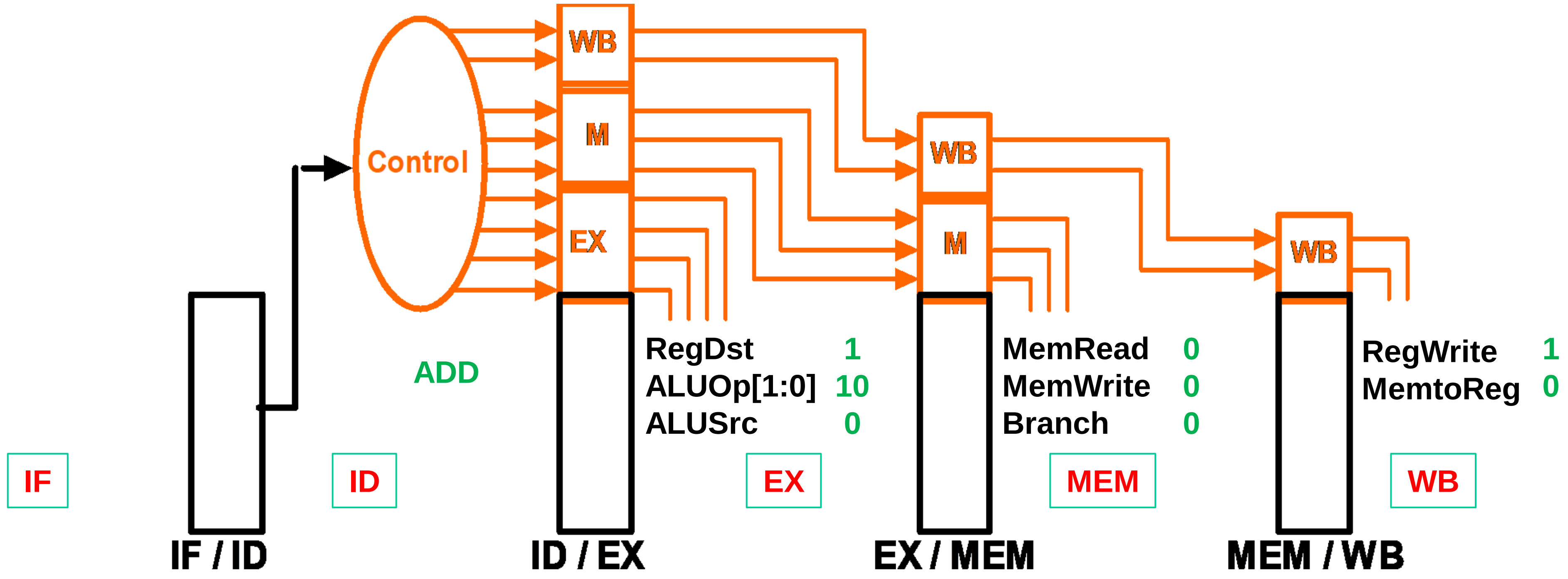
Pipelined Datapath with Control Signals



Next Step: Adding Control

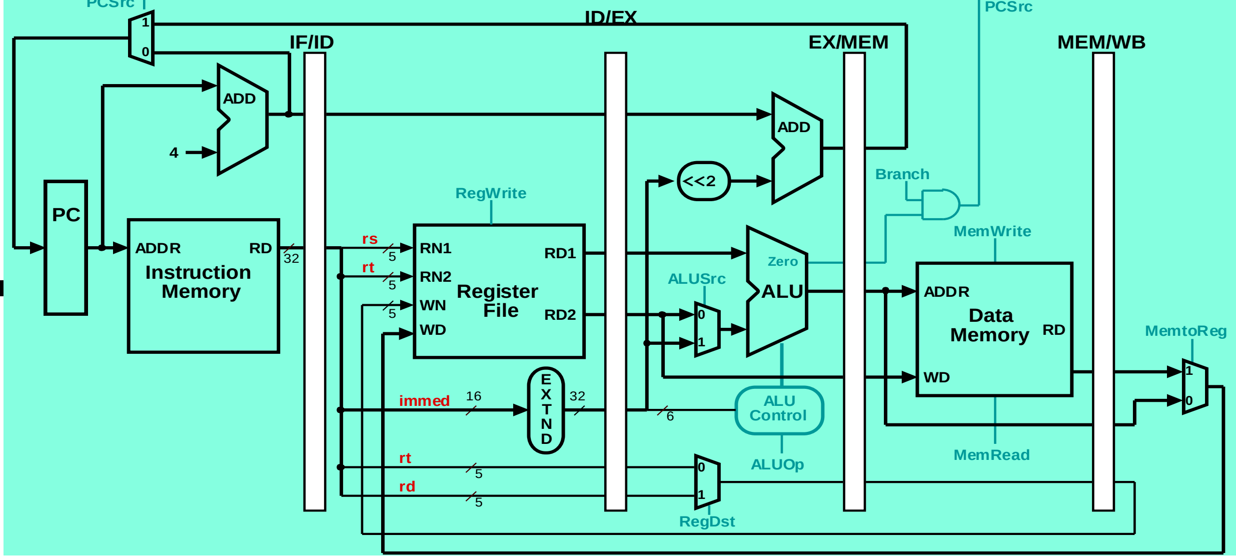
- **Basic approach: build on single-cycle control**
  - **Place control unit in ID stage**
  - **Pass control signals to following stages**
- **Later: extra features to deal with:**
  - **Data forwarding**
  - **Stalls**
  - **Exceptions**

Control for Pipelined Datapath



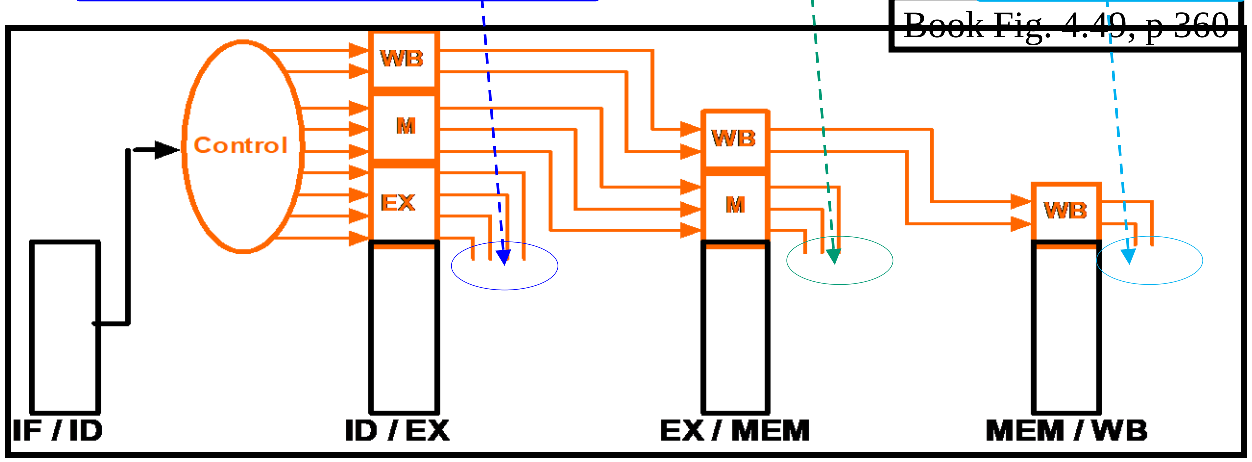
Source: Book Fig. 4.50, p 361

Control for Pipelined Datapath



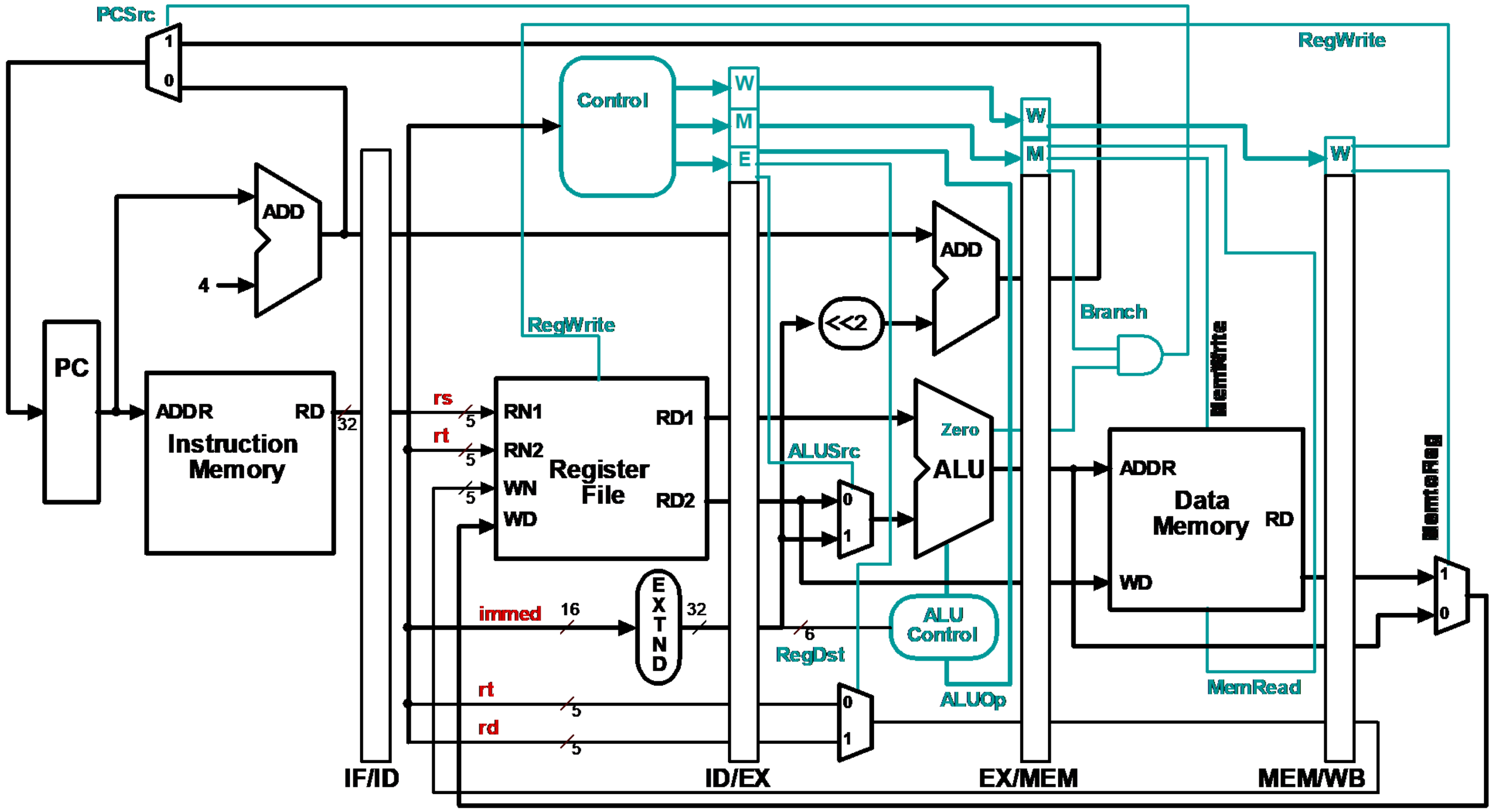
Instruction	Execution/Address Calculation stage control lines: Reg Dst	ALU Op1	ALU Op0	ALU Src	Memory access stage control lines: Branch	Mem Read	Mem Write	Write-back stage control lines: Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Book Fig. 4.49, p 360

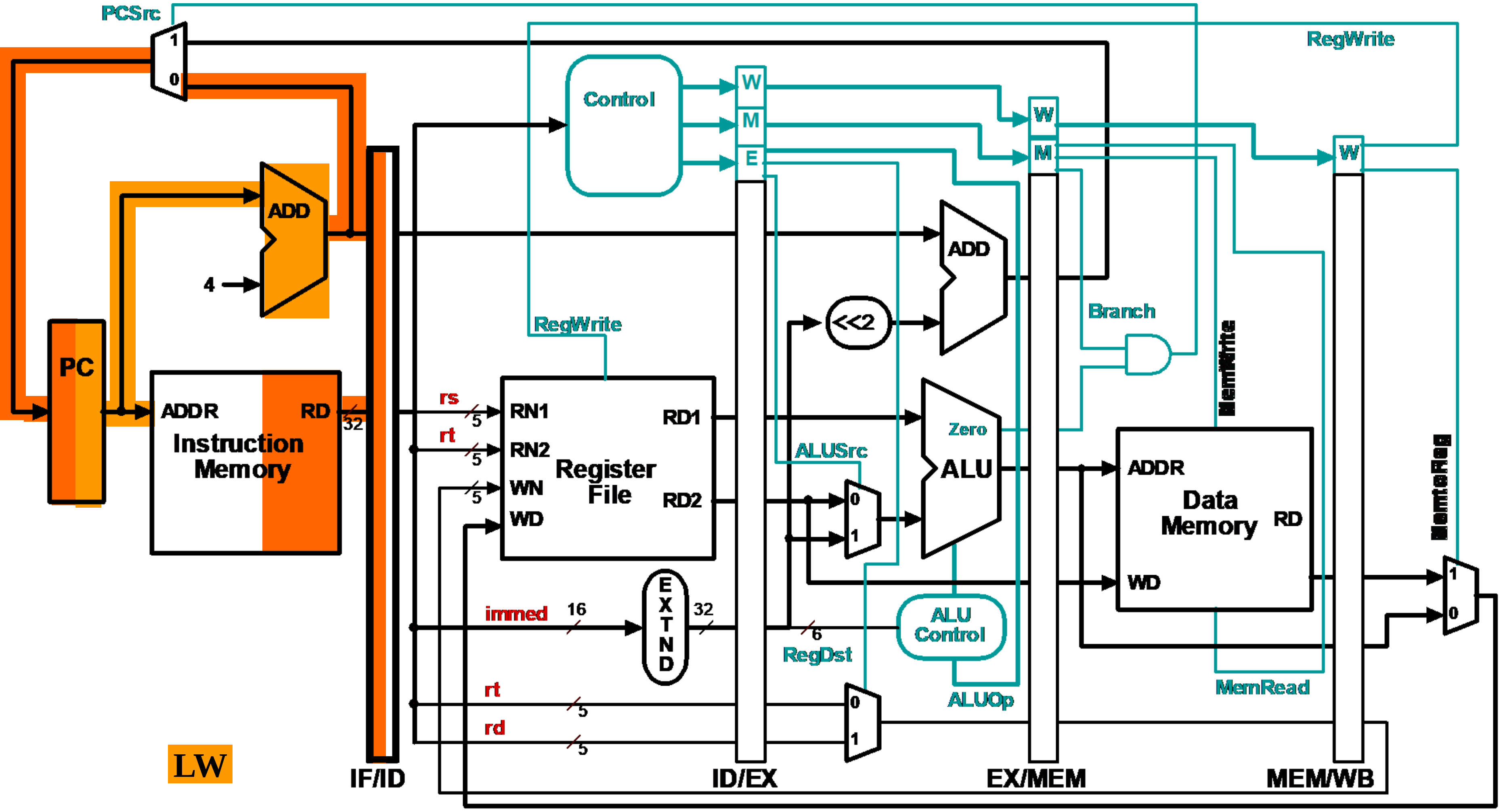


Book Fig. 4.50, p 361

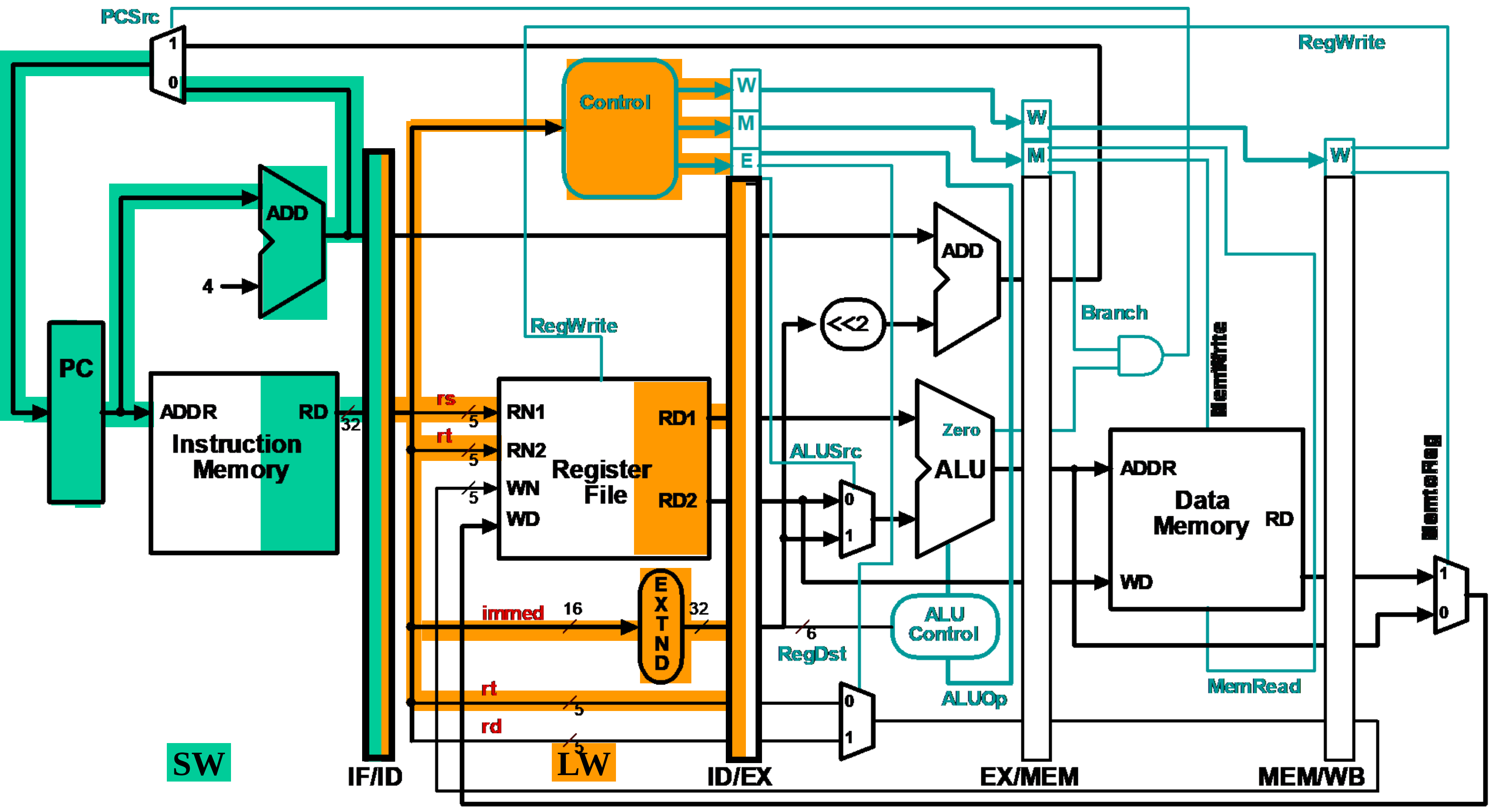
Datapath and Control Unit



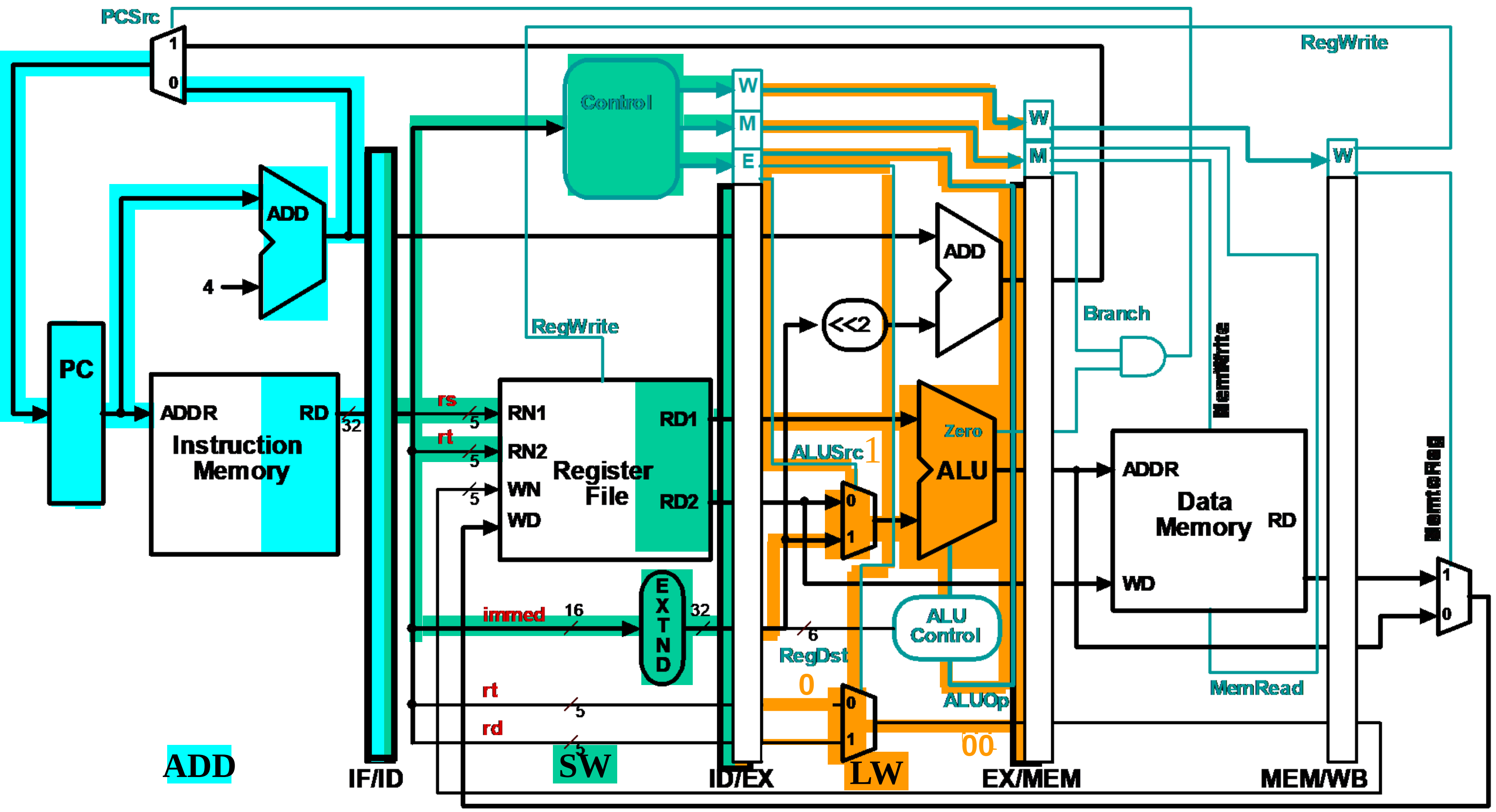
Tracking Control Signals - Cycle 1



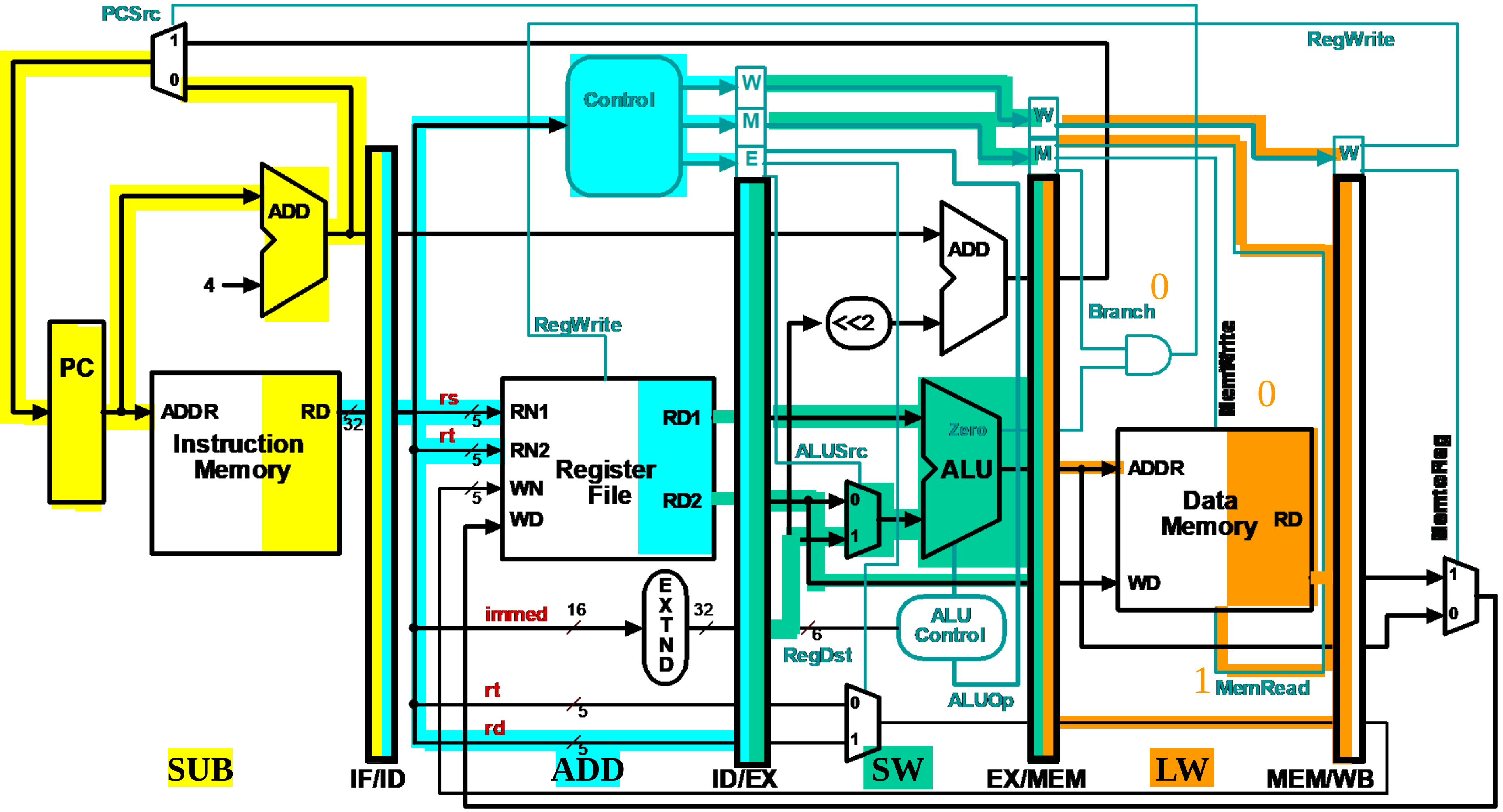
Tracking Control Signals - Cycle 2



Tracking Control Signals - Cycle 3

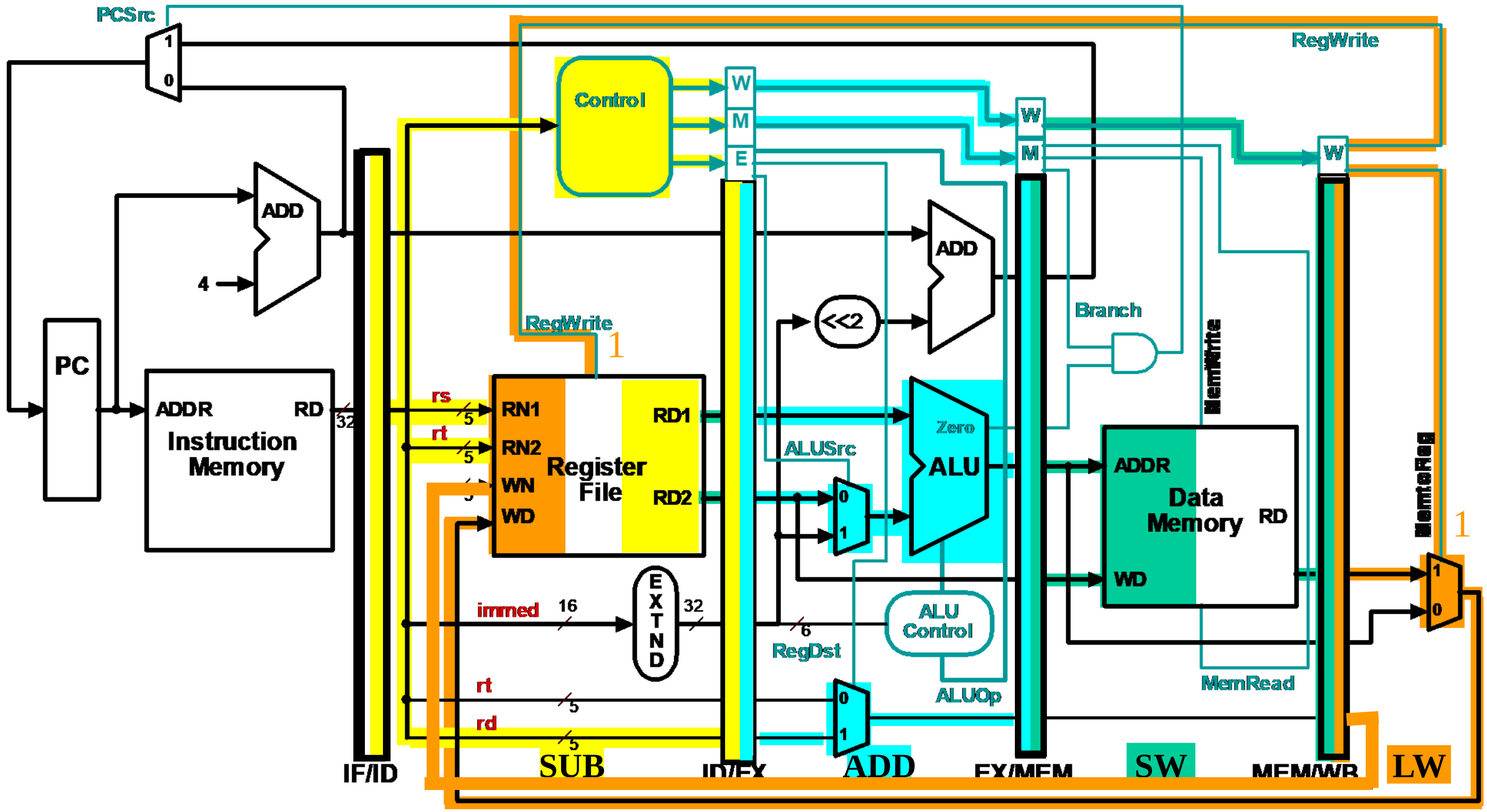


Tracking Control Signals - Cycle 4



Tracking Control Signals - Cycle 5

Back to 30



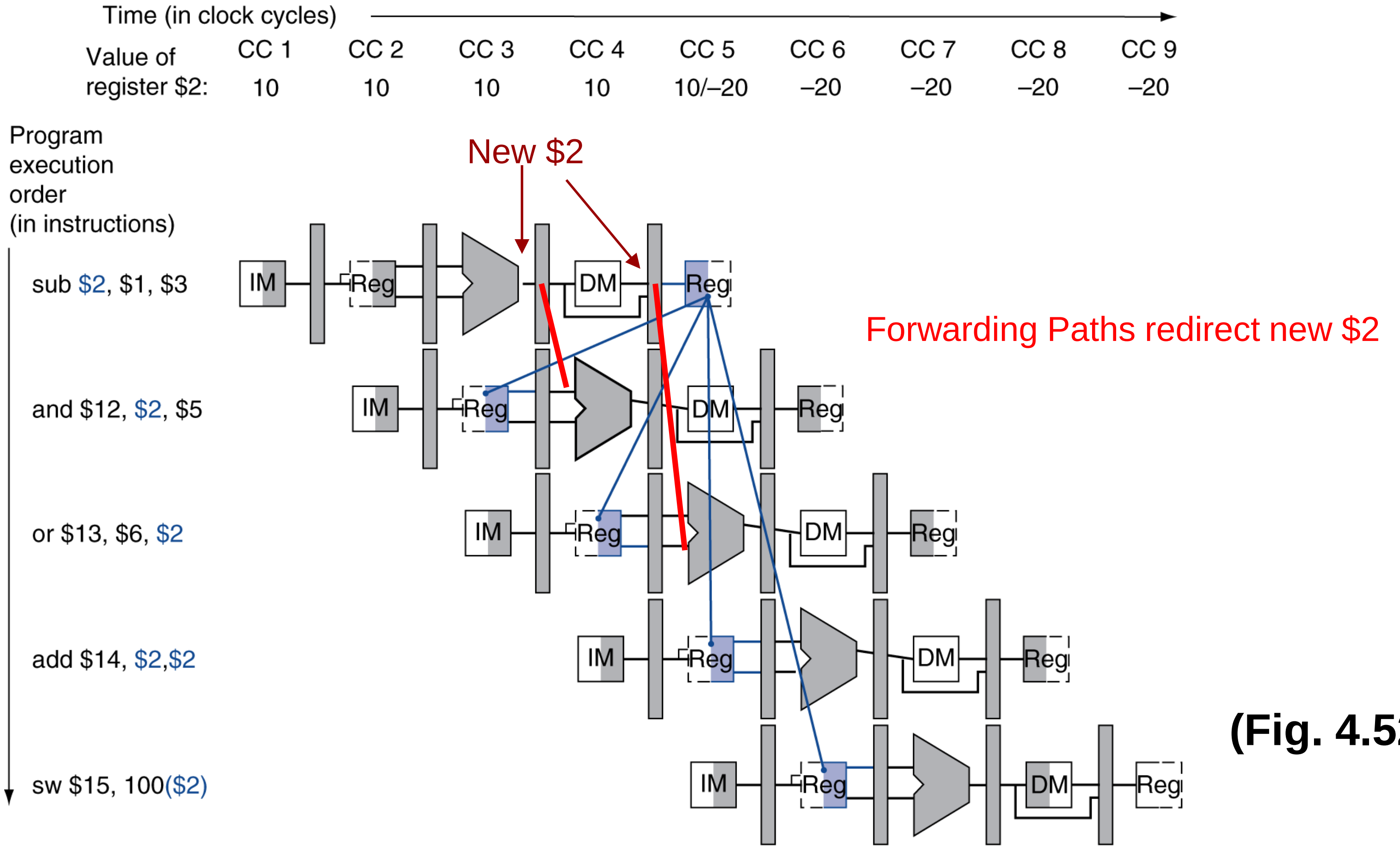
Pipelined Processor Design

- Datapath
- Control

- Dealing with Hazards & Forwarding
- Branch Prediction
- Exceptions
- Performance

Data Hazards Revisited…

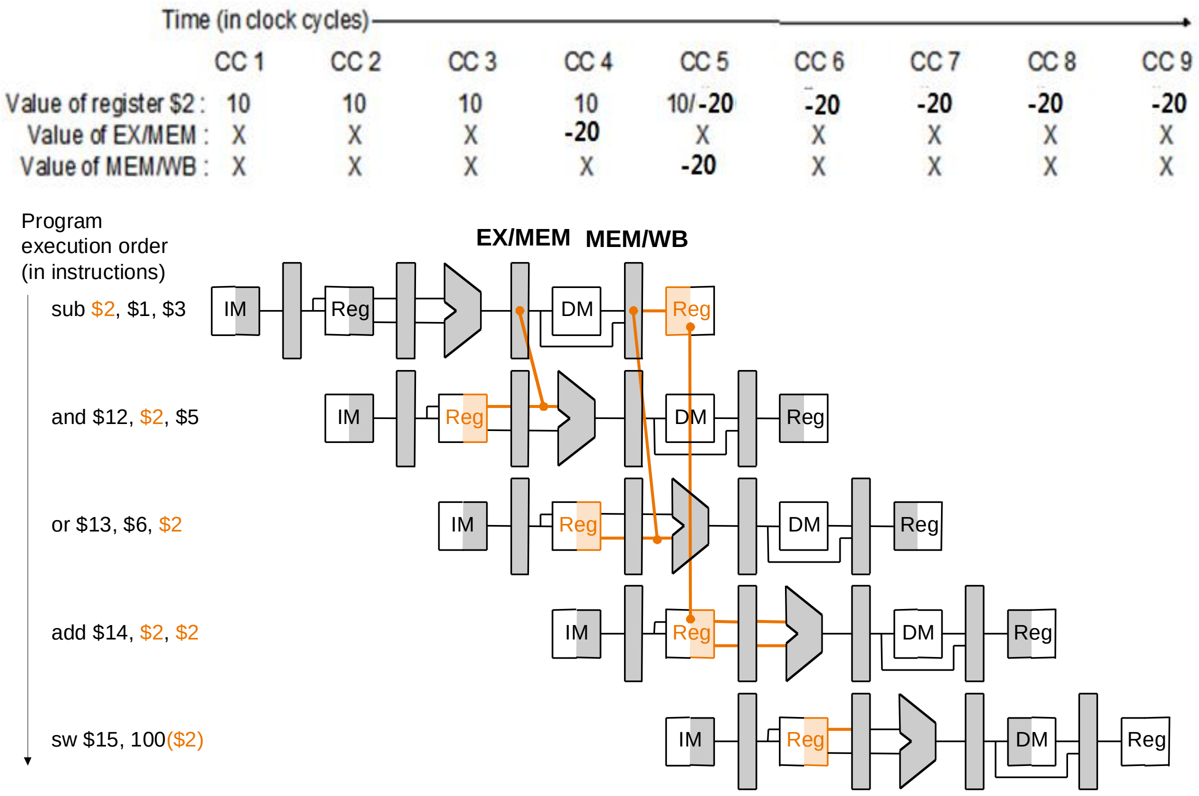
- **Data hazards - when data is used before it is stored**



(Fig. 4.52)

Data Hazard Solution: Forwarding

- Key idea: connect data internally before it is stored

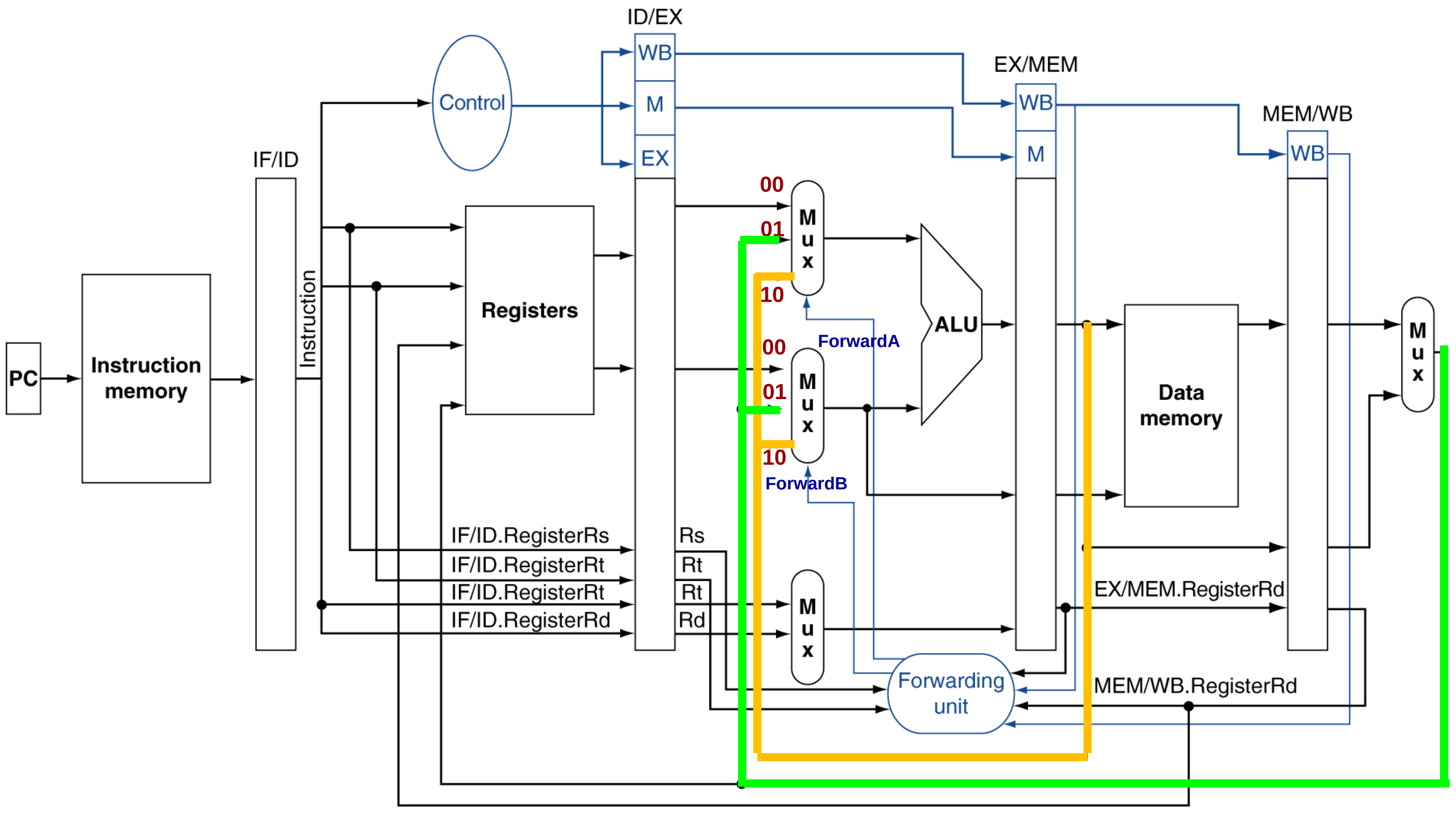


Time (in clock cycles)

	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

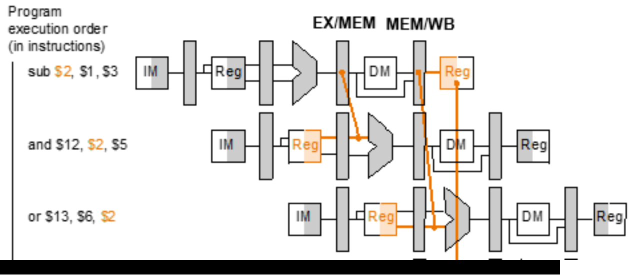
(Fig. 4.53)

Data Hazard Solution: Forwarding



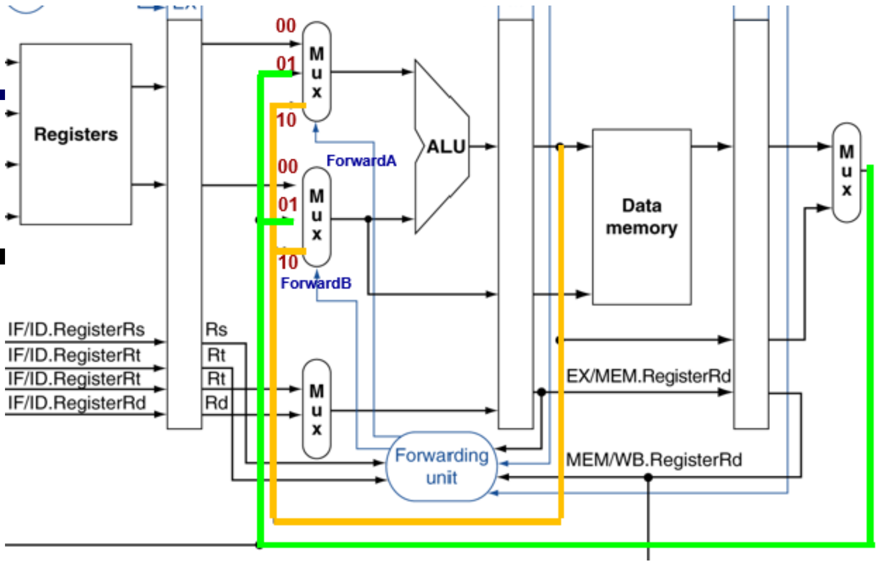
(Fig. 4.54)

Controlling Forwarding



- Need to test when register numbers match in `rs`, `rt`, and `rd` fields stored in pipeline registers
- "EX" hazard: EX 해저드는 연산하려는 레지스터 값이 EX/MEM에 있는 경우
  - EX/MEM - test whether instruction writes register file (WB) and examine `rd` register
  - ID/EX - test whether instruction reads `rs` or `rt` register and matches `rd` register in EX/MEM
- "MEM" hazard: MEM 해저드는 연산하려는 레지스터 값이 MEM/WB에 있는 경우
  - MEM/WB - test whether instruction writes register file (WB) and examine `rd` (`rt`) register
  - ID/EX - test whether instruction reads `rs` or `rt` register and matches `rd` (`rt`) register in MEM/WB

Forwarding Unit Detail - 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

Forwarding Unit Detail - 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
```

EX Hazard Complication

- **What if a register is changed more than once?**

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

- **Answer: forward most recent result (in MEM stage)**

Forwarding Unit Detail - MEM Hazard Revised

```
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 ≠ 0)
          and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
          ForwardB = 01
```

Forwarding Elaboration

- **Extra 2-1 mux needed for immediate instructions**

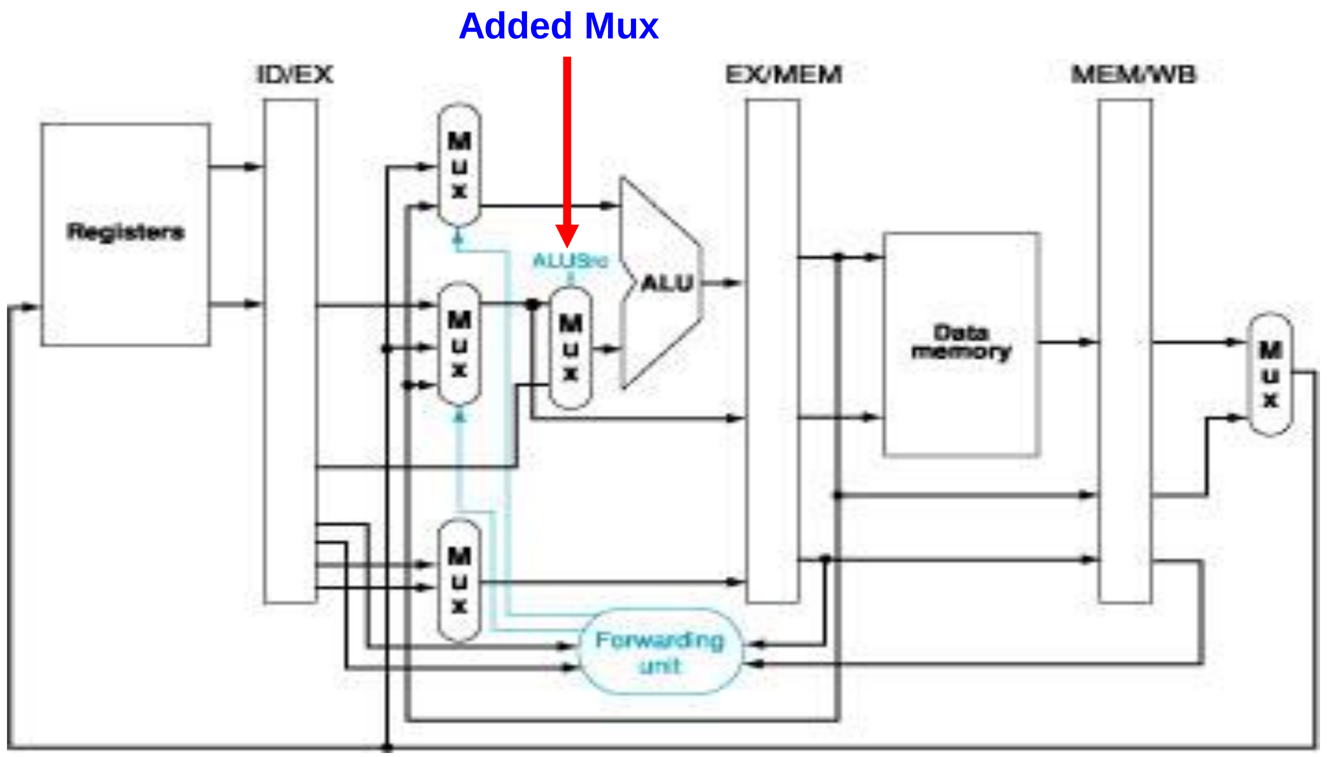
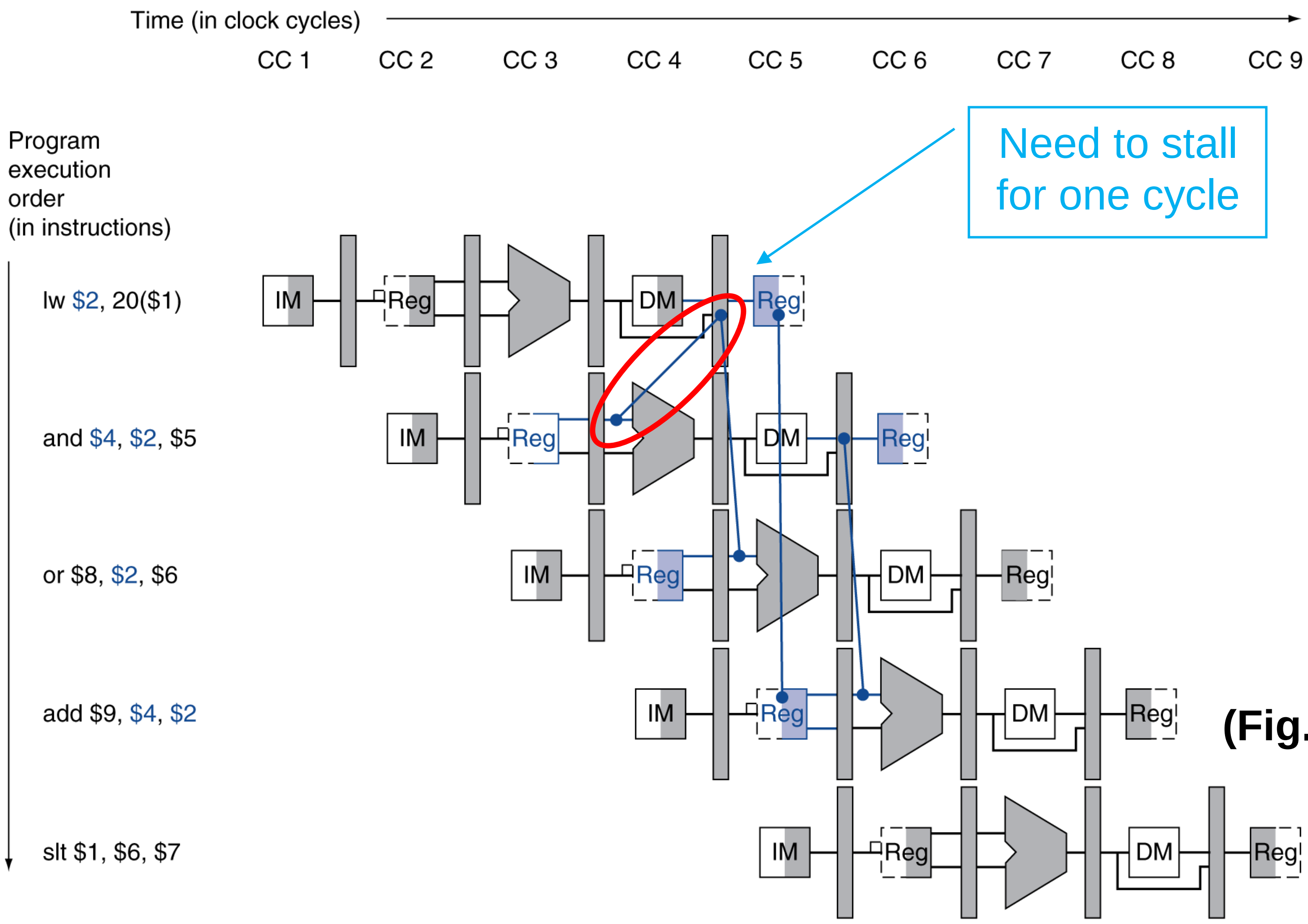


Fig (6.33)

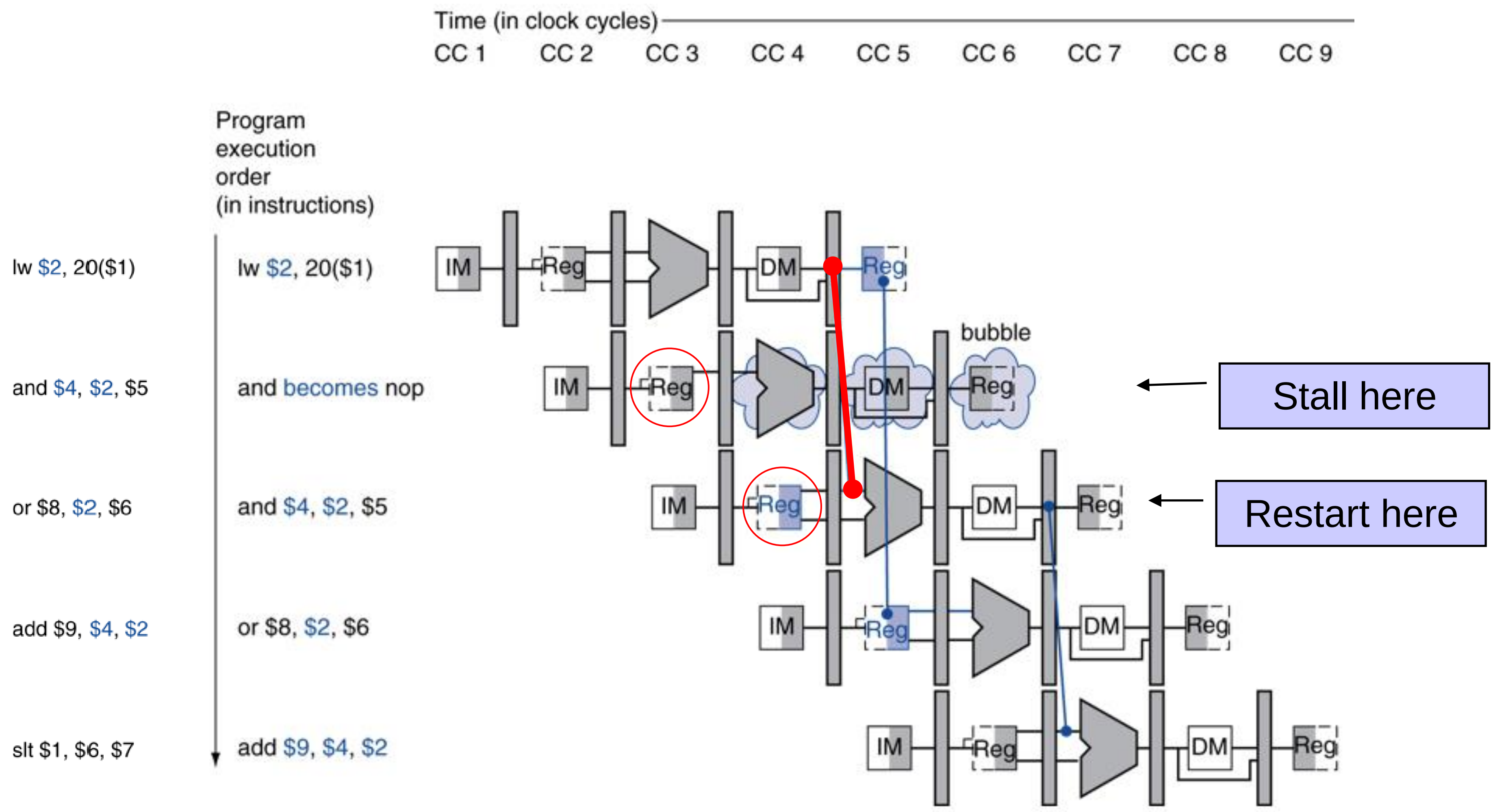
Load-Use Data Hazard



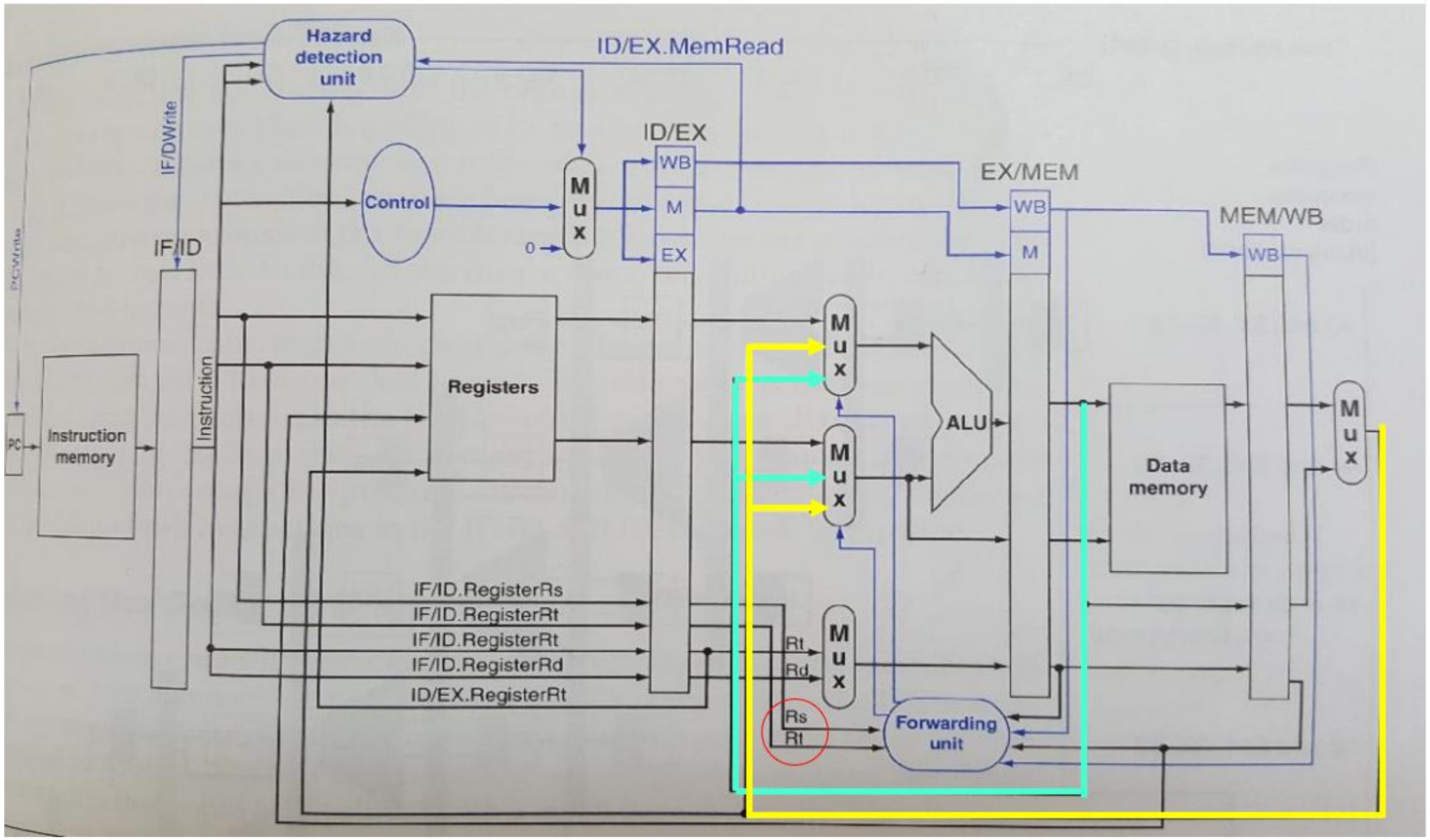
Need to stall for one cycle

(Fig. 4.58)

Load-Use Stall



Processor w/ Load-Use Hazard Detection



(Fig. 4.60)

Load-Use Hazard Detection

- Check “using” instruction in ID stage
- ALU operand register numbers in ID stage:?

  IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard test:

```
if ( ID/EX.MemRead and
     ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
      (ID/EX.RegisterRt = IF/ID.RegisterRt)) )
              … stall and insert bubble
```

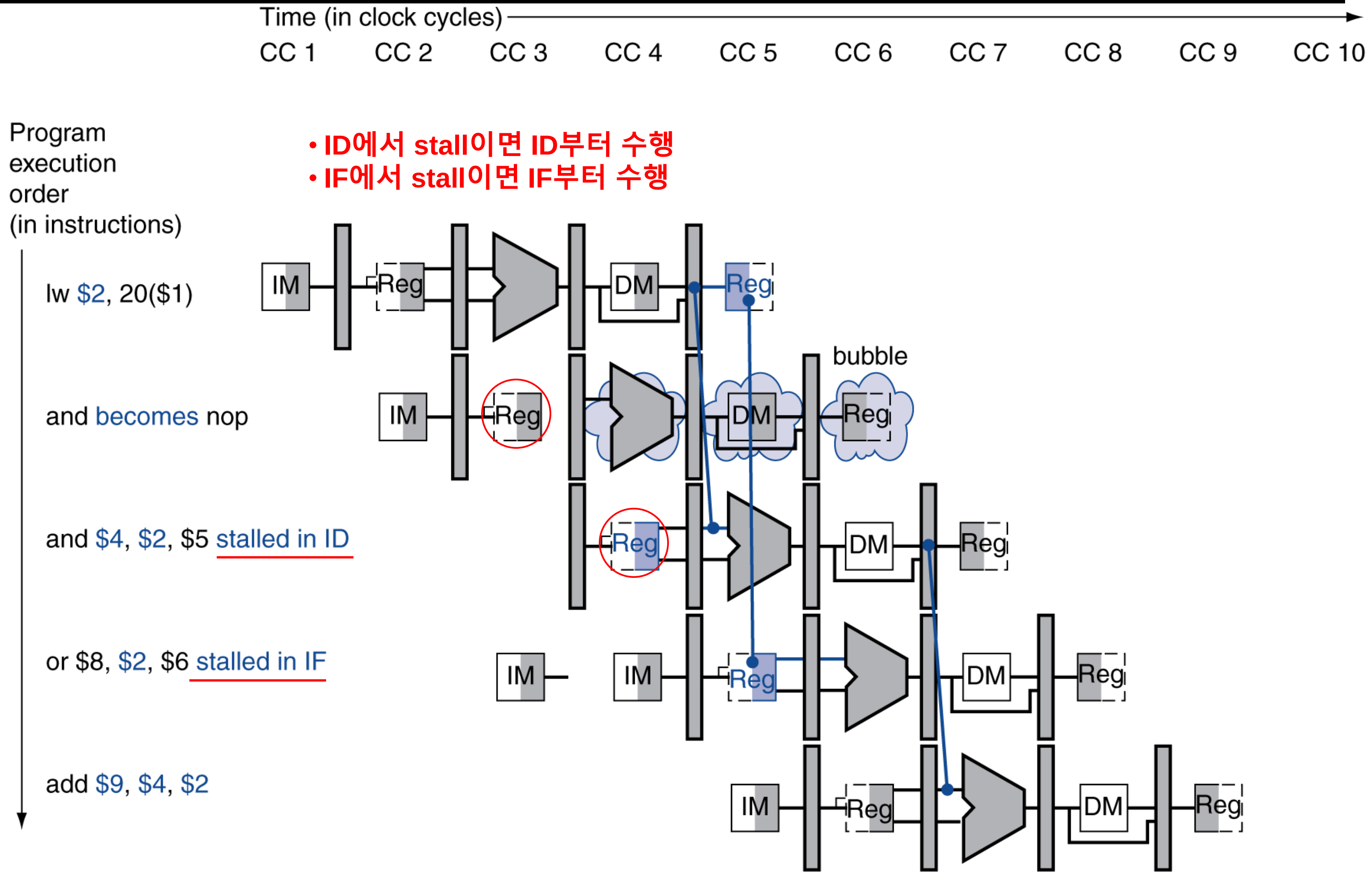
The rest of operation follows MEM hazard control!

Stalling the Pipeline for Load-Use Hazard

- MUX zeros out control signals for instruction in ID
  - "squashes” the instruction
  - “no-op” propagates through following stages

- IF/ID holds stalled instruction until next clock cycle
- PC holds current value until next clock cycle (re-loads first instruction)

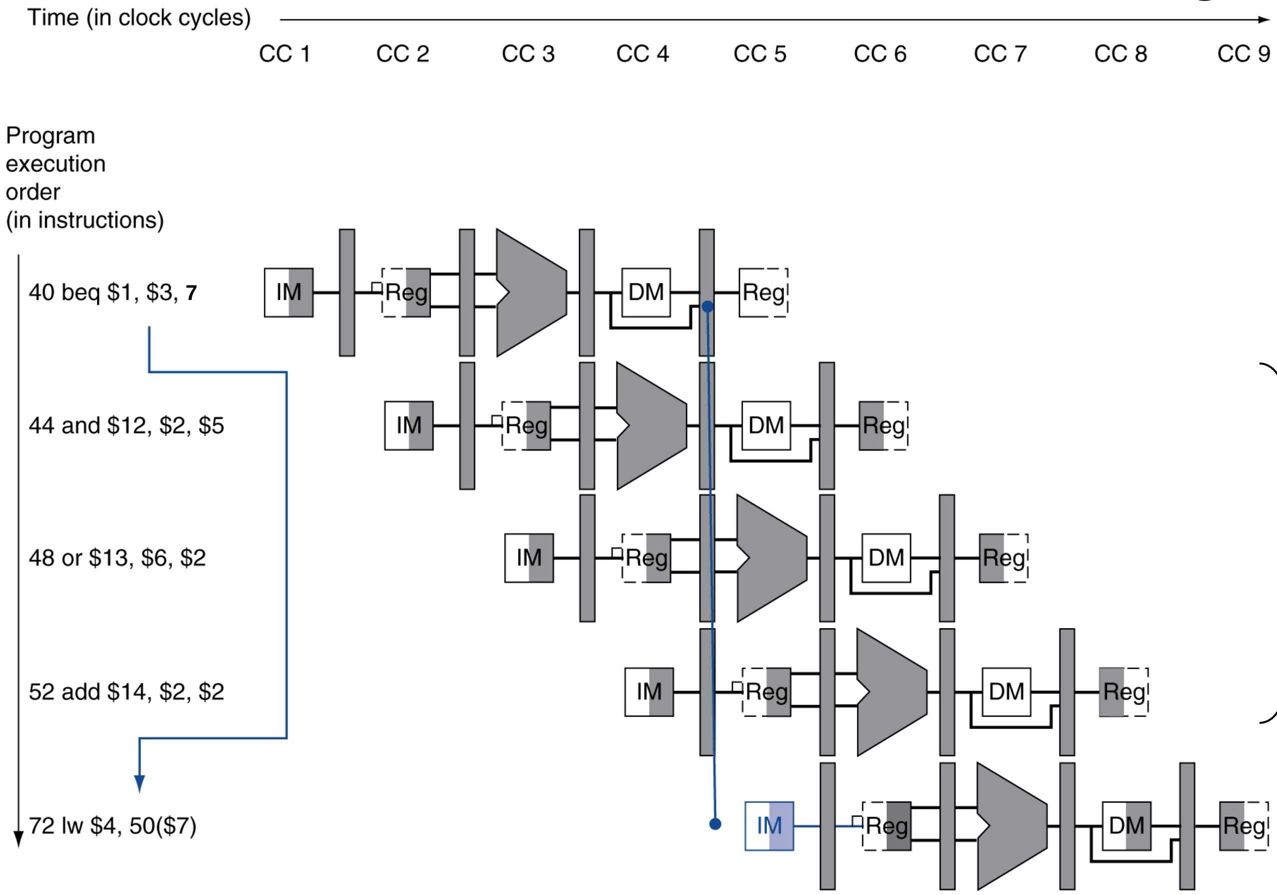
Load-Use Stall (showing stalled instr.)



Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
CC 10
Program execution order (in instructions)
• ID에서 stall이면 ID부터 수행
• IF에서 stall이면 IF부터 수행
lw $2, 20($1)
and becomes nop
and $4, $2, $5 stalled in ID
or $8, $2, $6 stalled in IF
add $9, $4, $2
bubble
IM
Reg
DM

Branch Hazards

- **When outcome is determined in MEM stage (Refer to pp 13):**



Flush these instructions (Set control values to 0)

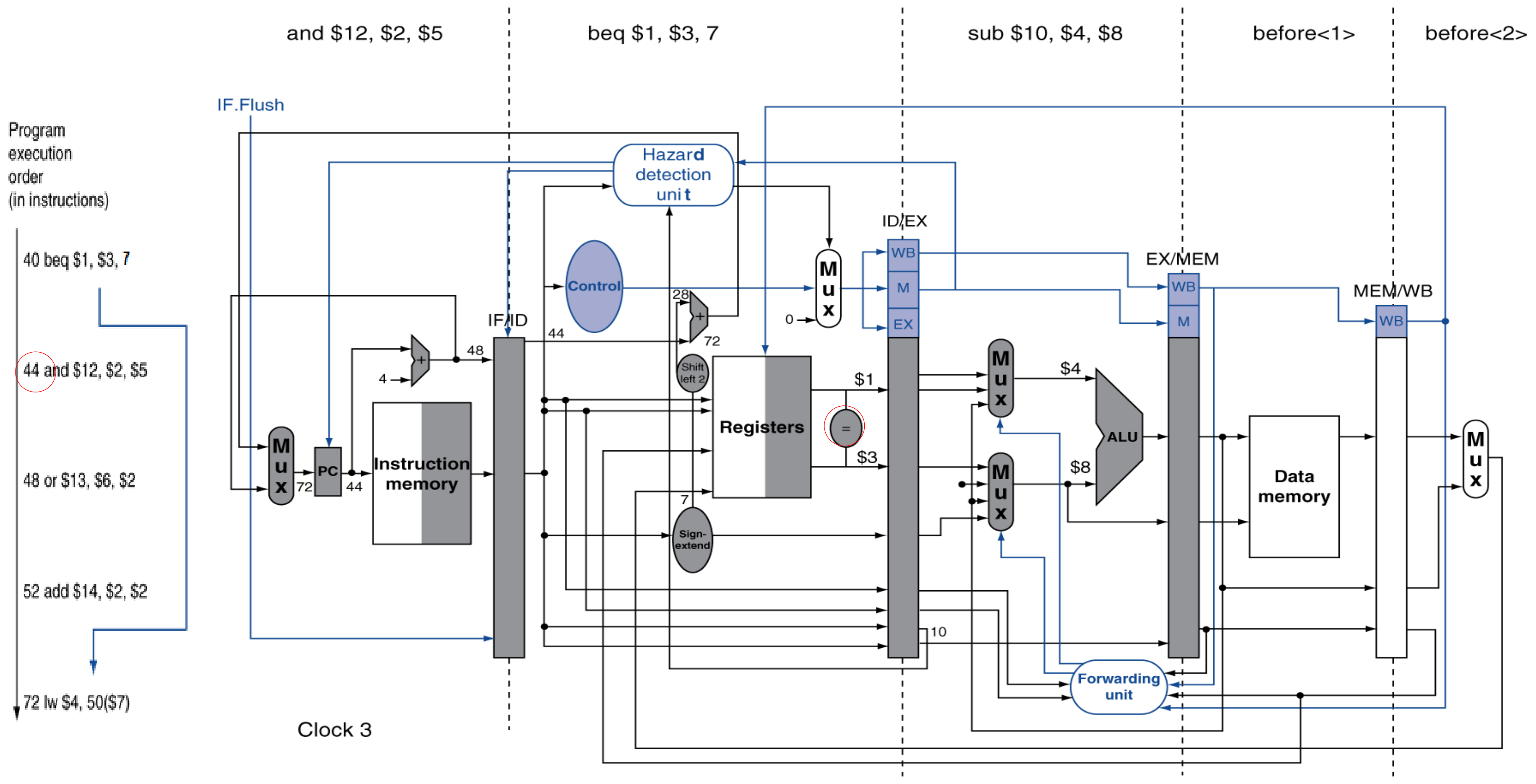
(Fig. 4.61)

Reducing Branch Delay

- Key idea: move branch logic to ID stage of pipeline
  - New adder calculates branch target

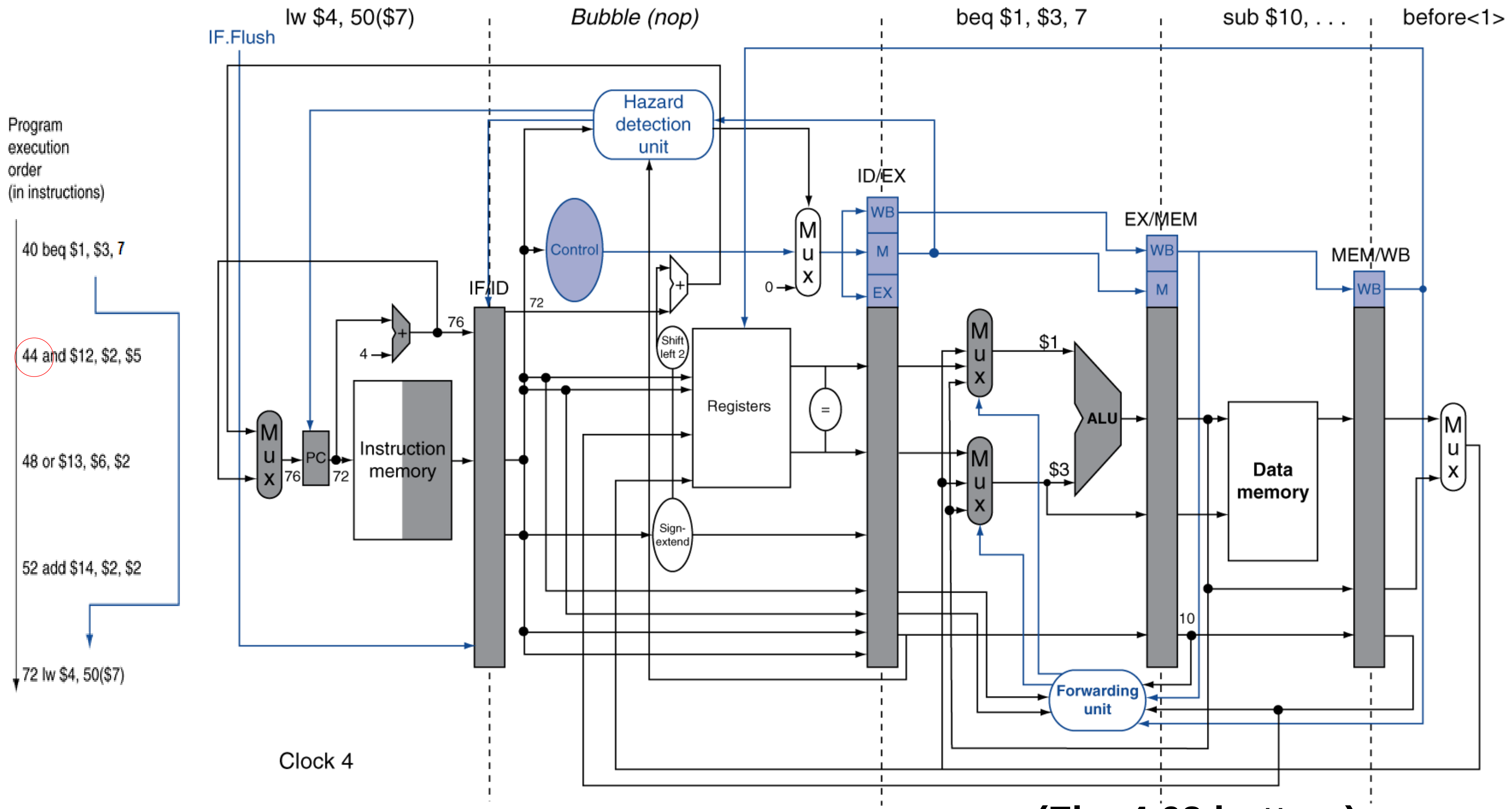
    `(PC + 4 + extend(IMM) << 2)`
  - New hardware tests `rs == rt` after register read
  - Add flush signal to squash instruction in IF/ID register
- Reduced penalty (1 cycle) when branch taken
- Example: Figure 4.62, p. 379

Example: Branch Hardware in ID



(Fig. 4.62 top)

Example: Branch Hardware in ID



(Fig. 4.62 bottom)

Pipelined Processor Design

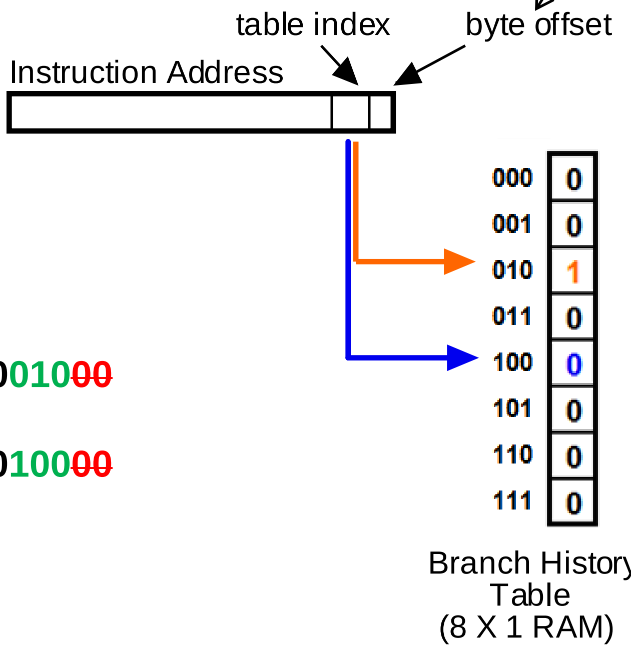
- Datapath
- Control

- Dealing with Hazards & Forwarding
- Branch Prediction
- Exceptions
- Performance

Branch Prediction

- Key idea: instead of always assuming branch not taken, use a prediction based on previous history
  - Branch history table: small memory
    - index using lower bits instruction address
    - save “what happened” on last execution
      - branch taken OR
      - branch not taken
  - Use history to make prediction

중복 확률 높음



```
0x40000   L1: addi $t0, $t0, 1
0x40004       slti $t1, $t0, 5
0x40008       bne  $t1, $zero, L1  00001000
0x4000C       add  $t4, $t5, $60
0x40010       beq  $t4, $zero, L2  00010000
```

More about Branch Prediction

- Consider nested loops:

```
for (i=1; i<M; i++)         oloop: . . .
   for (j=1;j<N; j++) {     iloop: . . .
         . . .                     . . .
   }  // bne $1,$2, iloop
}     // bne $3,$4, oloop
```

- Prediction fails on first and last branch
- More history can improve performance

# of prediction failures

= 2(M-1) + 2 = 2M if one bit is used

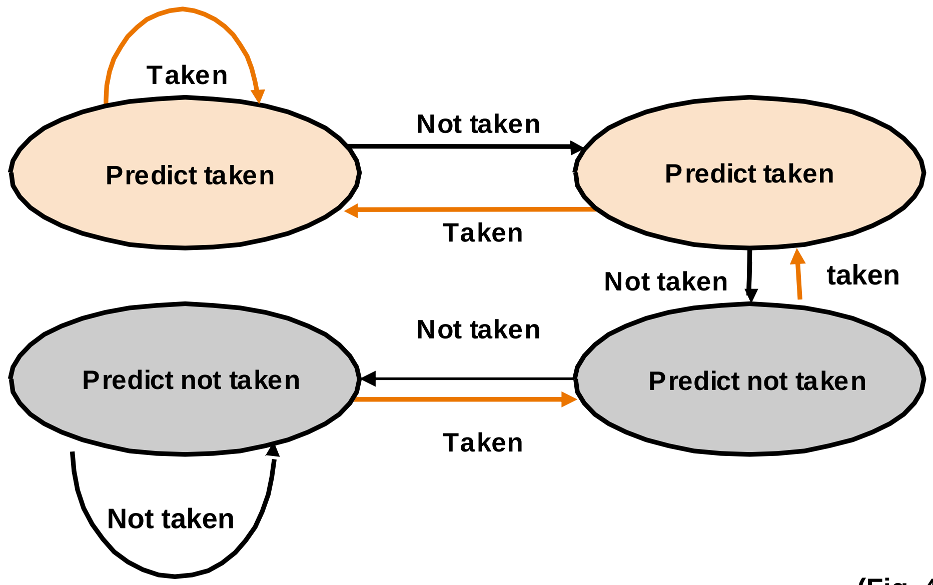
- iloop를 M-1번 수행하고 iloop돌때마다 2번 실패
- oloop는 2번 실패

= 2 + (M-1) + 2 + 1 = M+4 if two bits are used

2: iloop 처음 2번 실패 (처음 두 번 실패 후에 branch taken으로 수정)
(M-1): iloop 마지막 매번 1번 실패
2: oloop 처음 두 번 실패 (두 번 실패 후에 Branch Taken으로 수정)
1: oloop 마지막 한번 실패

Branch Prediction w/2-Bit History

- Key idea: must be wrong twice before changing prediction



(Fig. 4.63)

- # of prediction failures = 2(M-1) + 2 = 2M if one bit history is used
- # of prediction failures = 2 + (M-1) + 2 + 1 = M+4 if two bits are used

Pipelined Processor Design

- Datapath
- Control

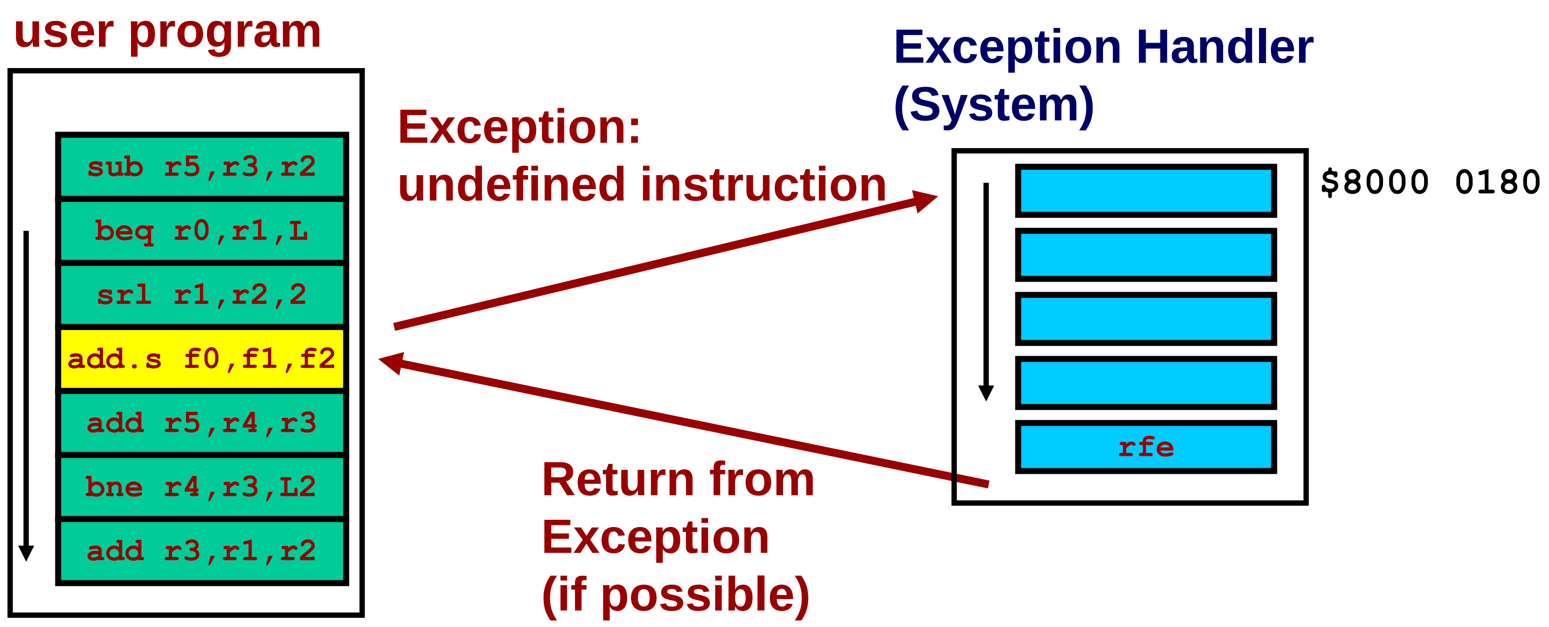
- Dealing with Hazards & Forwarding
- Branch Prediction
- Exceptions
- Performance

Exceptions - “Stuff Happens”

- Definition: "unexpected change in control flow"
- Used to handle runtime errors
  - Overflow
  - Undefined Instruction
  - Hardware malfunction
- Used to handle external events, "service" functions
  - Interrupts - external I/O Device request
  - Page fault - virtual memory
  - System call - user request for OS action

What happens during an exception

- Save user state - register values, etc.
- Take action to handle exception
- Restore user state and continue execution if possible; otherwise terminate



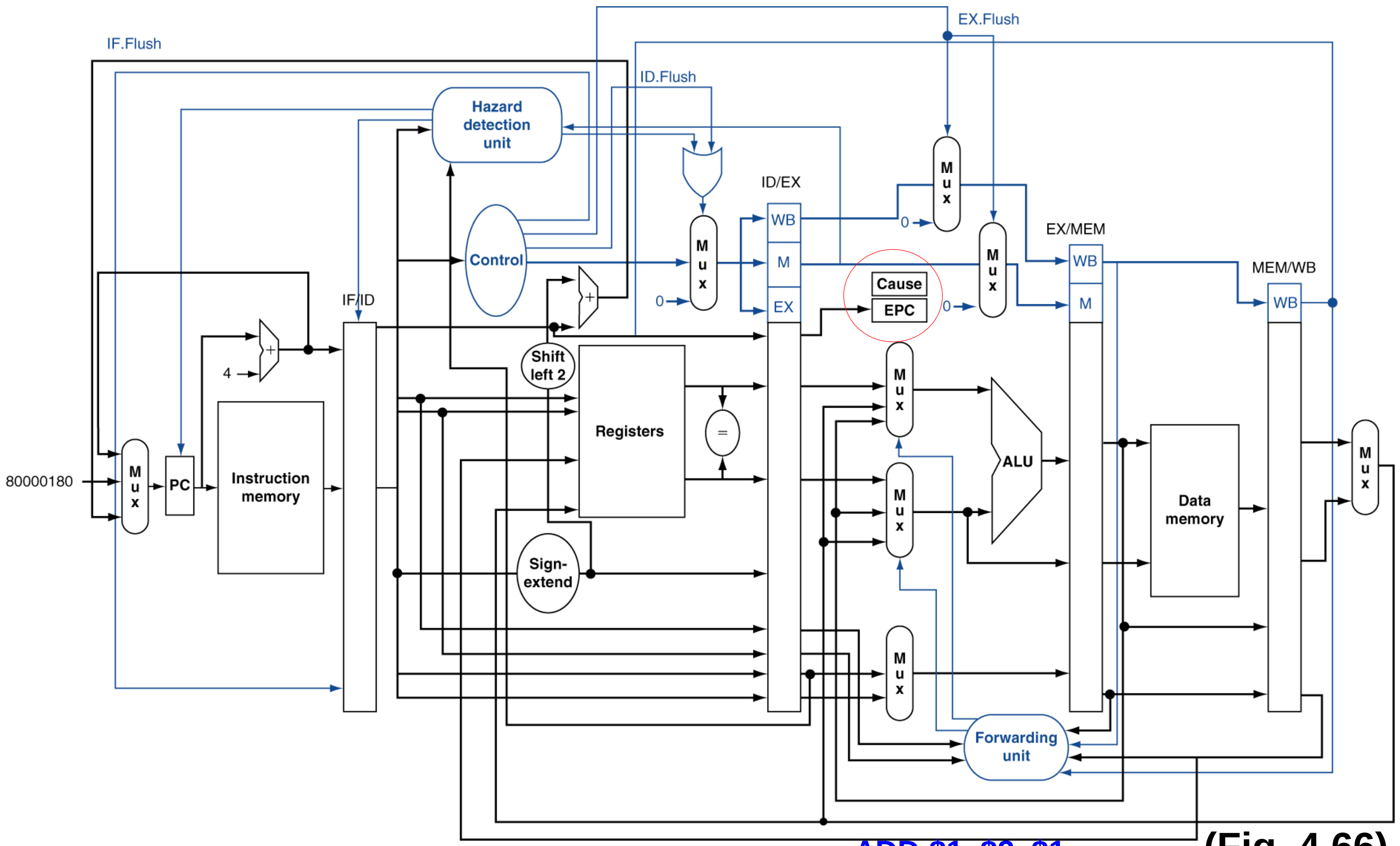
Exceptions in MIPS

- **Two exceptions (for now):**
  - **Undefined instruction**
  - **Arithmetic overflow**
- **Add registers to architecture to save state as part of a System Control Coprocessor (CP0)**
  - **EPC - Exception Program Counter (32 bits)**
  - **Cause - records cause of exception (32 bits)**
    - **Undefined instruction: Cause <- 0**
    - **Arithmetic overflow: Cause <- 1**
- **Alternatives used by other architectures**
    - **Save PC on stack**
    - **Communicate exception type using Exception Vector**

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

Exception Hardware



ADD \$1, \$2, \$1

(Fig. 4.66)

Exception Example

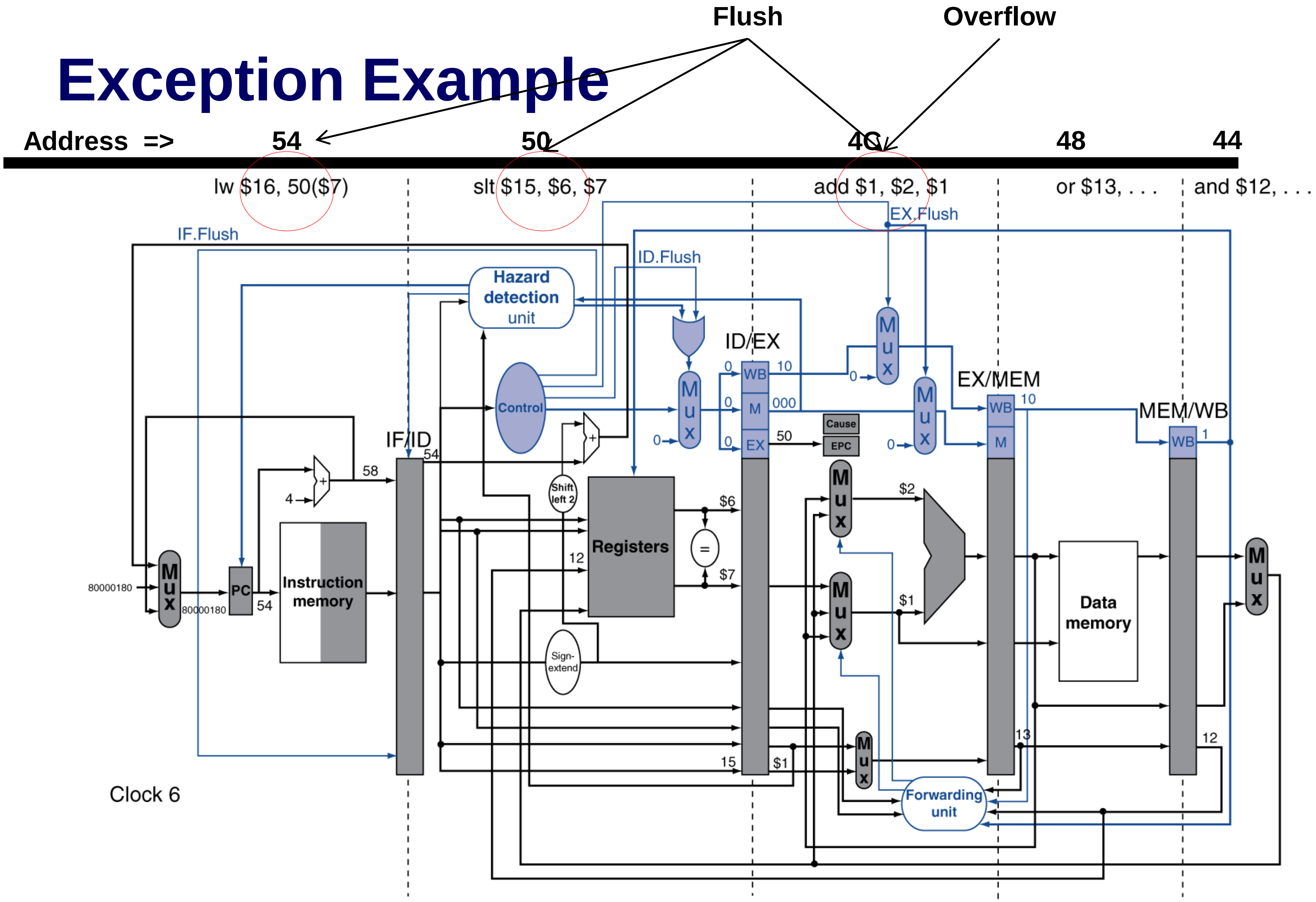
- Exception on `add` due to arithmetic overflow:

```
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

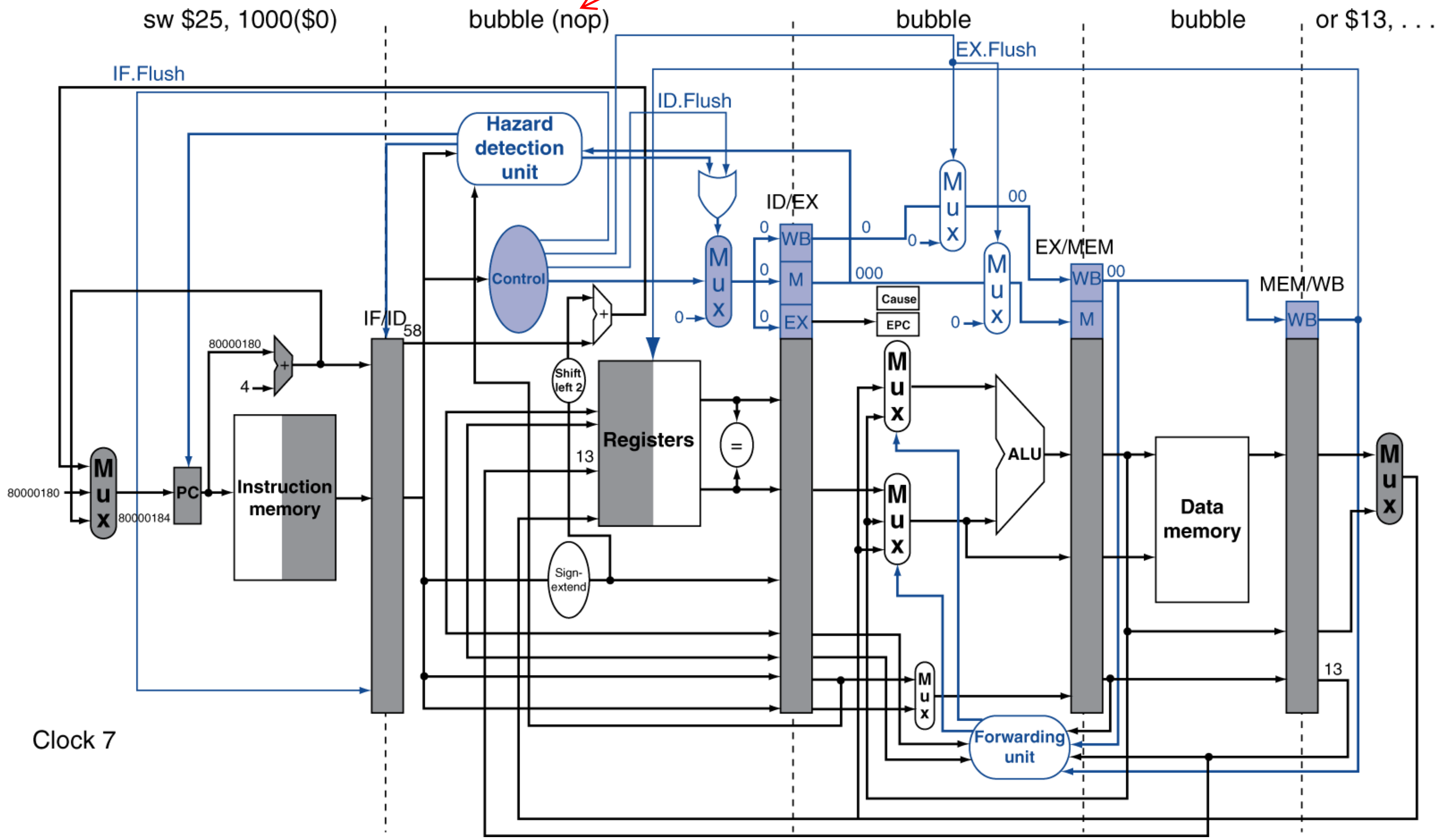
```
80000180    sw   $25, 1000($0)
80000184    sw   $26, 1004($0)
…
```

Exception Example



Exception Example

nop => sll $0, $0, 0



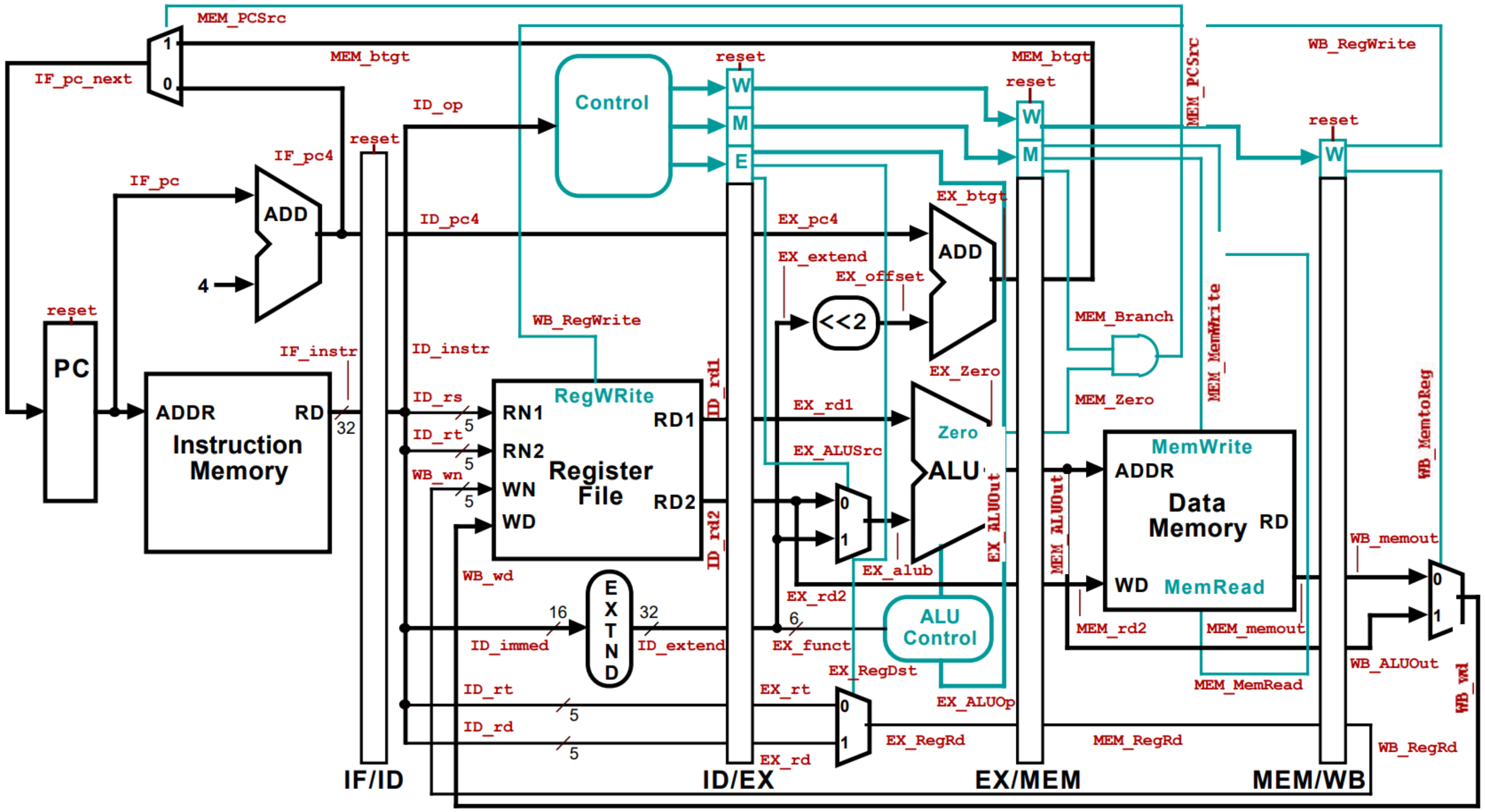
Multiple Exceptions

- 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**

Pipelined MIPS Verilog Model



Pipelined Processor Design

- Datapath
- Control

- Dealing with Hazards & Forwarding
- Branch Prediction
- Exceptions
- Performance

Performance of the Pipelined Implementation

- Use "gcc" instr. mix to calculate CPI

lw	25%	1 cycle (2 cycles when load-use hazard)
sw	10%	1 cycle
R-type	52%	1 cycle
branch	11%	1 cycle (2 cycles when prediction wrong)
jump	2%	2 cycles

- Assmptions:
  - 50% of load instructions are followed by immed. use
  - 25% of branch predictions are wrong
- Calculating CPI
  - CPI = (1.5 cycles * 0.25) + (1 cycle * 0.10) + (1 cycle * 0.52) + (1.25 cycles * 0.11) + (2 cycles * 0.02)
  - CPI = 1.17 cycles per instruction

Performance of the Pipelined Implementation (cont'd)

- Calculate the average execution time:

Pipelined	1.17 CPI * 200ps/clock	=	234ps
Single-Cycle	1 CPI * 600ps/clock	=	600ps
Multicycle	4.12 CPI * 200ps / clock	=	824ps

- Speedup of pipelined implementation
  - 2.56X faster than single cycle
  - 3.4X faster than multicycle
- "Your mileage may differ" as instruction mix changes

Summary

- Pipelined processor
- Hazards
  - Structural hazards - attempt to use same resource twice
    - ⇒ Delay
    - ⇒ Seprate memories
  - Control hazards - attempt to make decision before condition is evaluated
    - ⇒ Stall
    - ⇒ Predict
    - ⇒ Delayed branch
  - Data hazards - attempt to use data before it is ready
    - ⇒ Stall
    - ⇒ Forwarding
    - ⇒ Reordering instructions