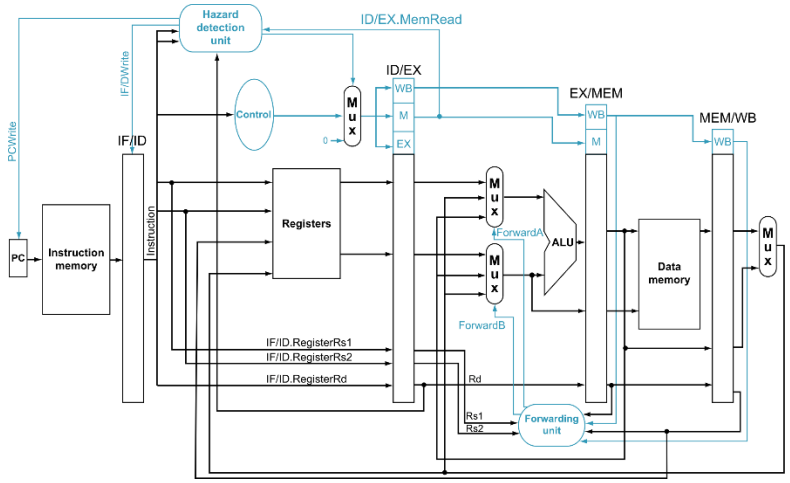
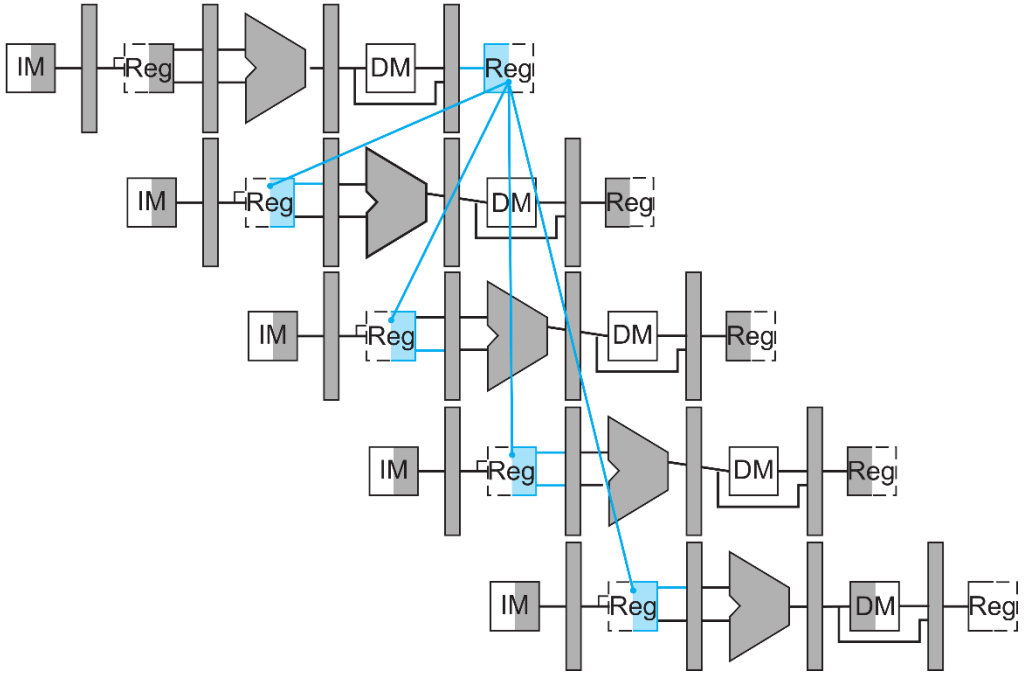
Processor Architecture

Pipelined RISC-V Implementation (Part II)



Module Outline

- Data Hazards
- Control Hazards
- Exceptions
- Module Summary

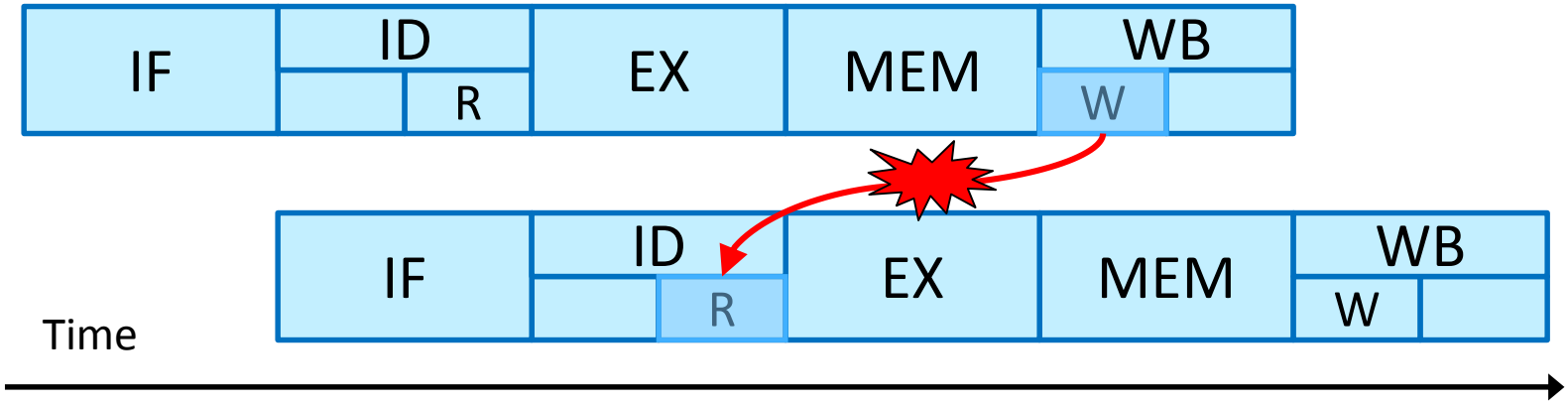


Data Hazards

Recap: Data Hazard

- Caused by data dependencies (arithmetic or load)

```
OP i      add    x5, x11, x12
OP i+1    sub    x7, x5, x7
```



Information cannot flow backwards in time!

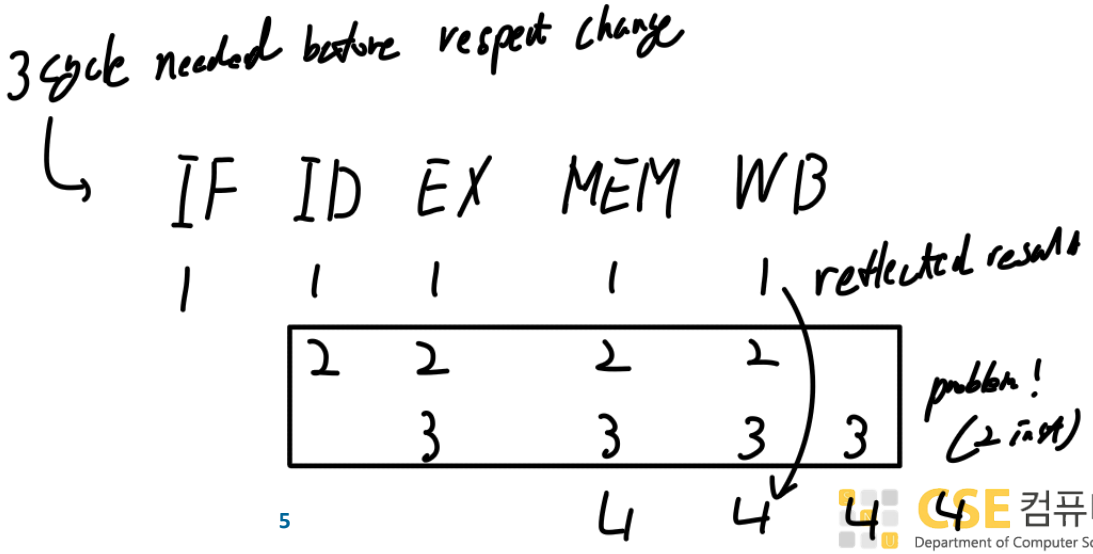
Data Hazards in ALU Instructions

- Consider this code sequence:

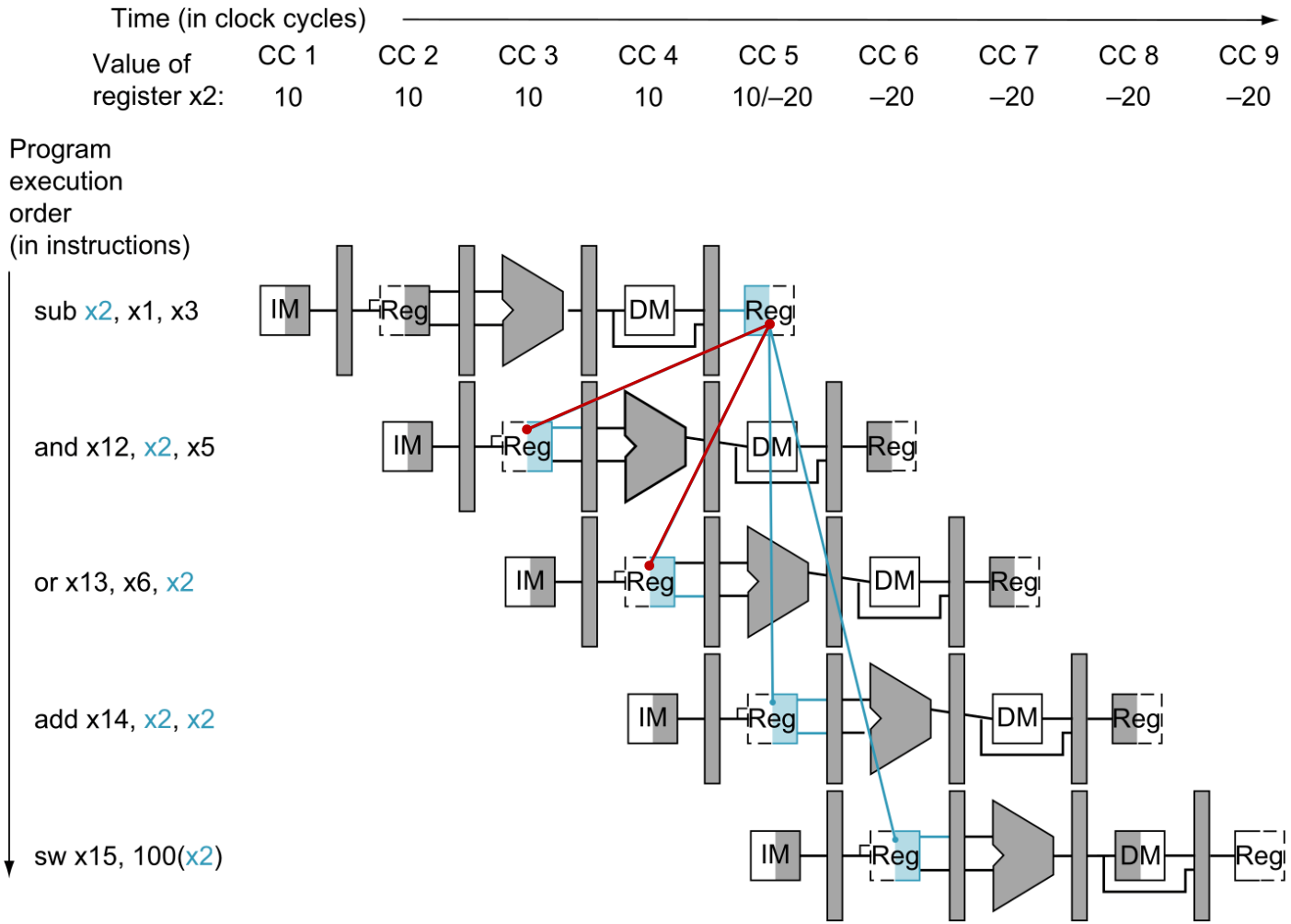
```
sub x2,  x1, x3
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sw  x15, 100(x2)
```

- Several data dependencies (RAW) on register x2

```
sub x2,  x1, x3
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
sw  x15, 100(x2)
```



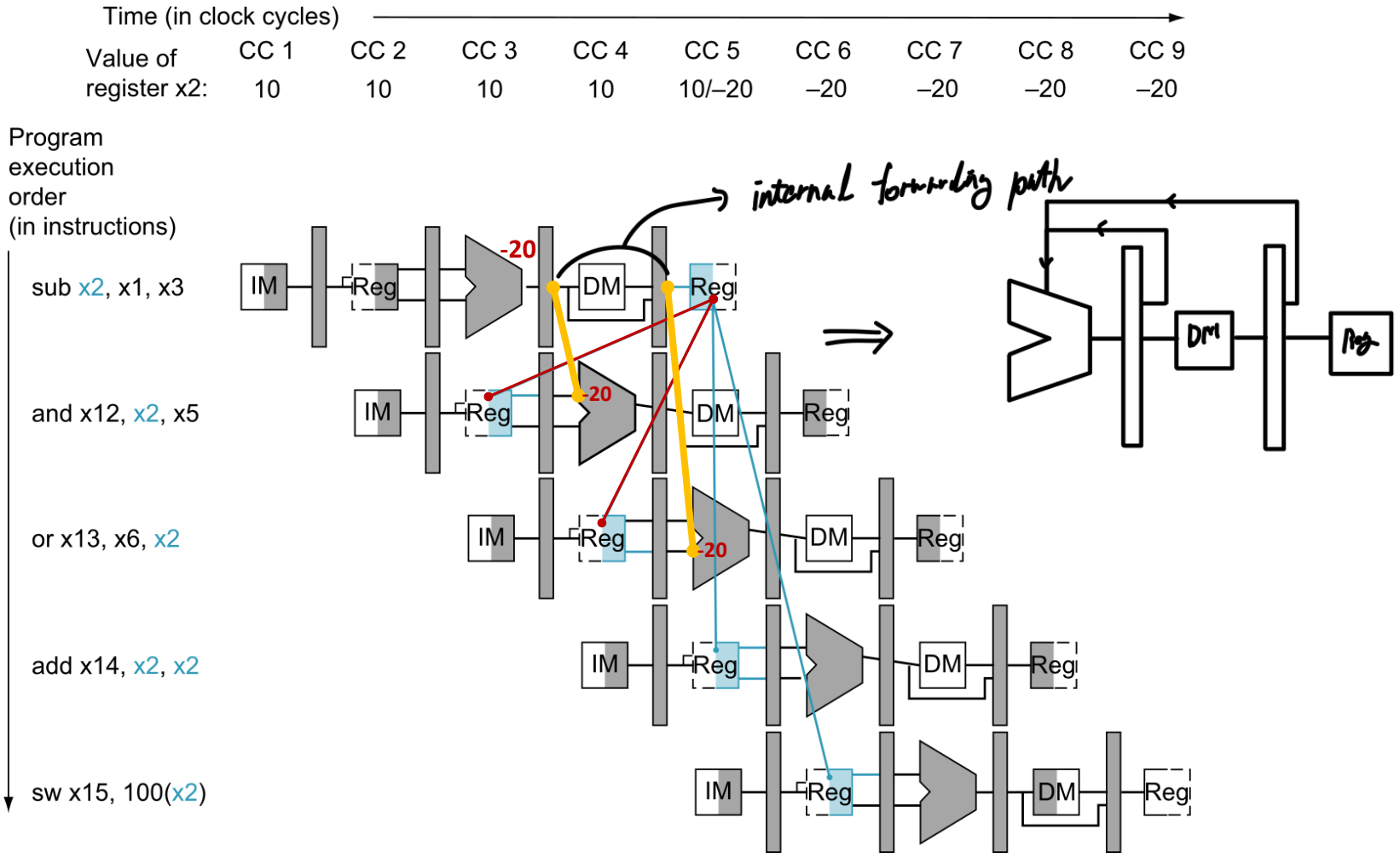
Data Hazards in ALU Instructions



- **How do we detect data hazards in the pipeline and how do we fix them?**

Data Hazards in ALU Instructions

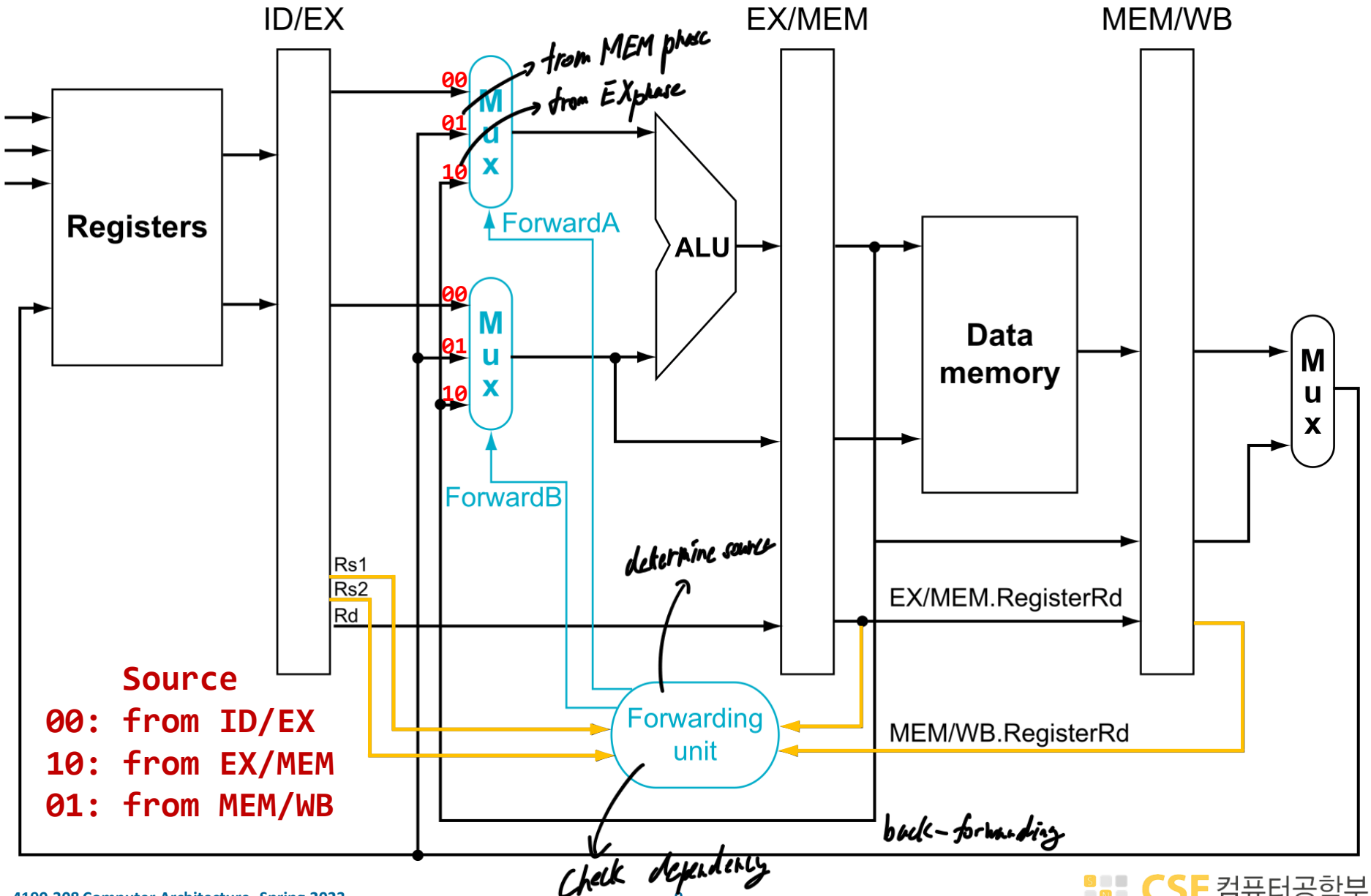
- **The updated value of x2 is available as soon as sub has passed the EX stage**



Forwarding Paths and Control

- **Pass source register indices along in pipeline** (destination register already passed on)
  - Pass register source 1 and register source 2 indices from ID to EX stage
  - Naming
    - `ID/EX.RegisterRs1` = register # for `Rs1` sitting in ID/EX pipeline register
    - `ID/EX.RegisterRs2` = register # for `Rs2` sitting in ID/EX pipeline register

Forwarding Paths and Control



Forwarding Control

MUX control	Source	Example
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.

Detecting the Need to Forward

- **Condition for data hazards**

Arithmetic data hazard

```
EX/MEM.RegisterRd = ID/EX.RegisterRs1
EX/MEM.RegisterRd = ID/EX.RegisterRs2
```

Forward from EX/MEM pipeline register

```
MEM/WB.RegisterRd = ID/EX.RegisterRs1
MEM/WB.RegisterRd = ID/EX.RegisterRs2
```

Forward from MEM/WB pipeline register

Memory data Hazard

Detecting the Need to Forward (cont'd)

- **But only if forwarding instruction will write to a register**

```
EX/MEM.RegWrite = 1,
MEM/WB.RegWrite = 1
```

- **And only if Rd for that instruction is not x0**

```
EX/MEM.RegisterRd ≠ 0
MEM/WB.RegisterRd ≠ 0
```

Forwarding Conditions : Computing inside Forwarding unit

⇒ (rd == rs1/2) & (regwrite) & (! regrd)

- **EX hazard**

```
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
   forwardA = 10
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
   forwardB = 10
```

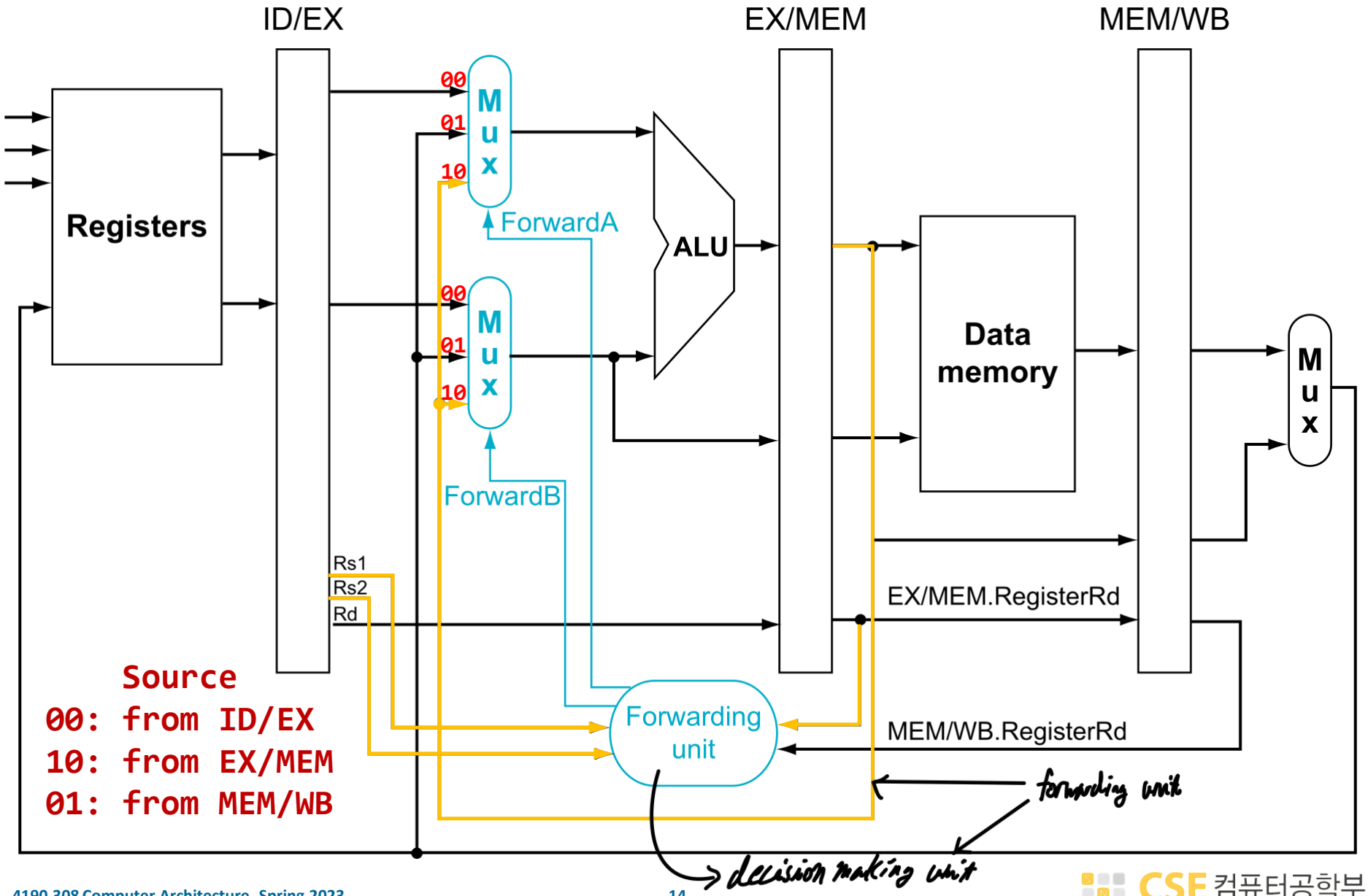
Completed

- **MEM hazard**

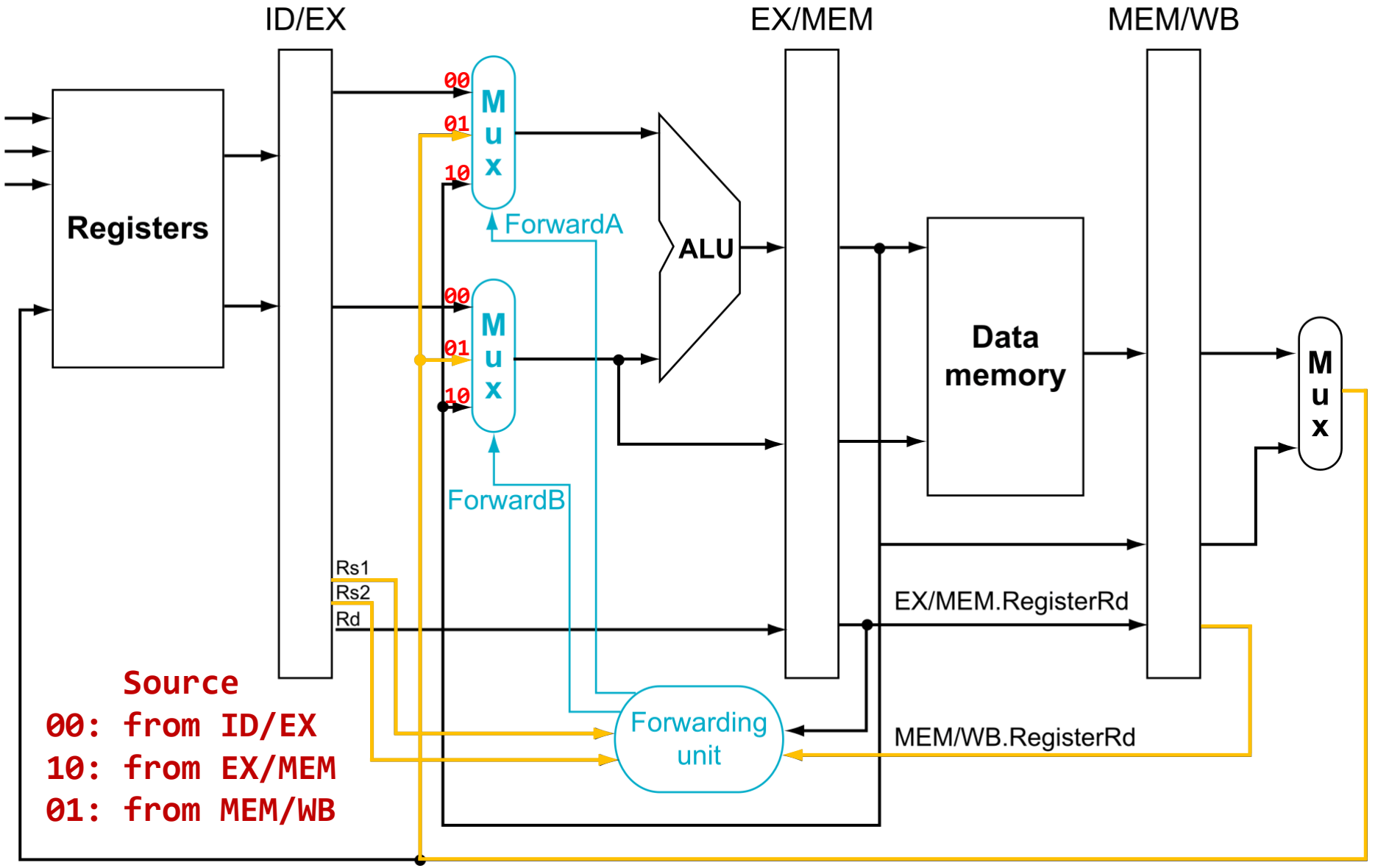
```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
   forwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
   forwardB = 01
```

incomplete

Forwarding from EX/MEM



Forwarding from MEM/WB



Simultaneous Data Hazards

- Consider this sequence:

```
add  x1, x1, x2   ①
add  x1, x1, x3   ②
add  x1, x1, x4   ③
```

→ ③ can refer to
result of ① in MEM/WB Reg
or
result of ② in EX/MEM Reg
⇒ have to use recent!
∴ using ② for ③ operation

- Both hazards occur simultaneously
  - Want to mimic serial execution
  - → **Use most recent result**
- Revise MEM hazard condition
  - Only forward if EX hazard condition isn't true

Revised Forwarding Conditions

- MEM hazard

Check forwarding

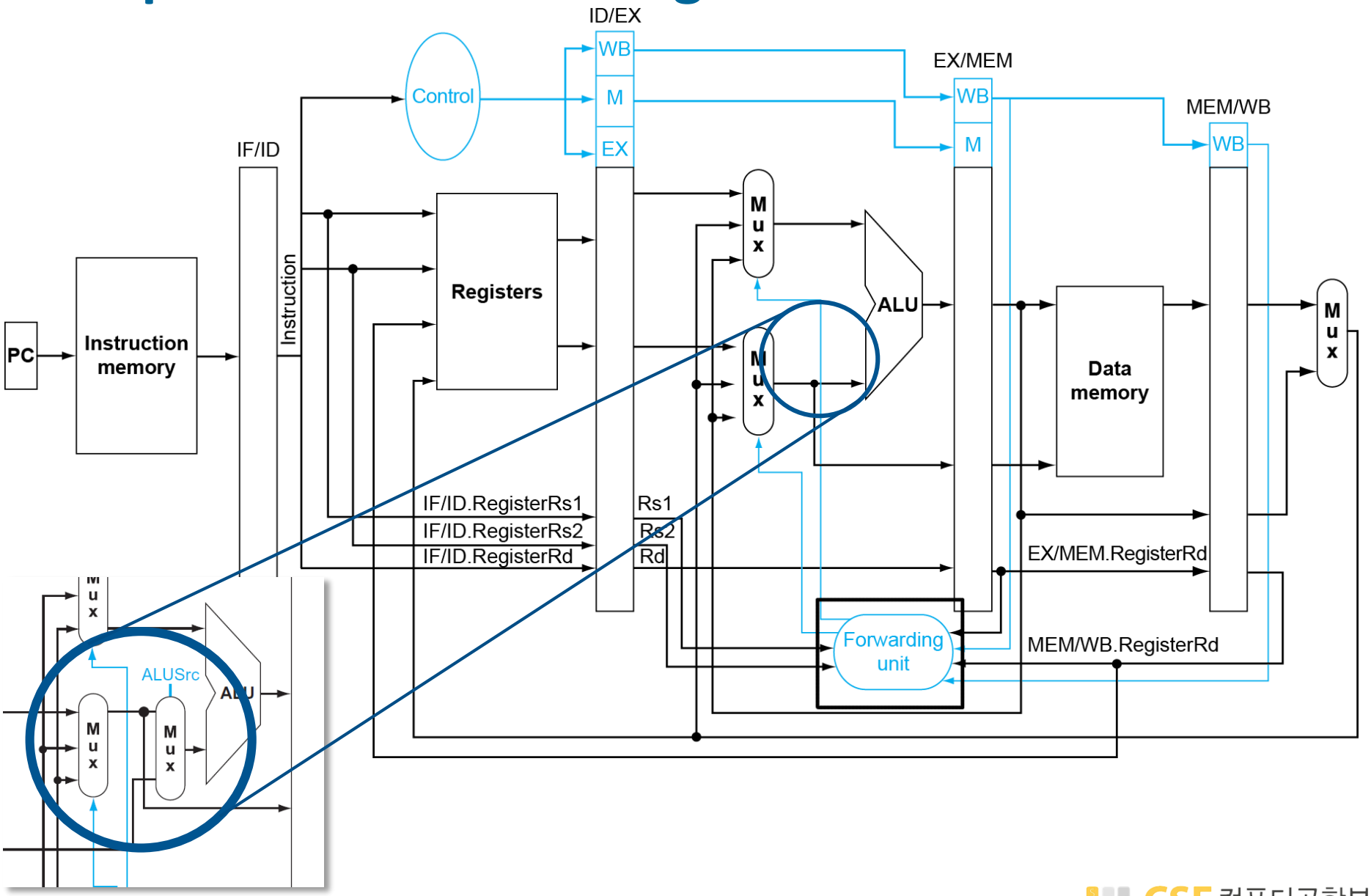
```
if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
    forwardA = 01
```

and not EX data hazard occur

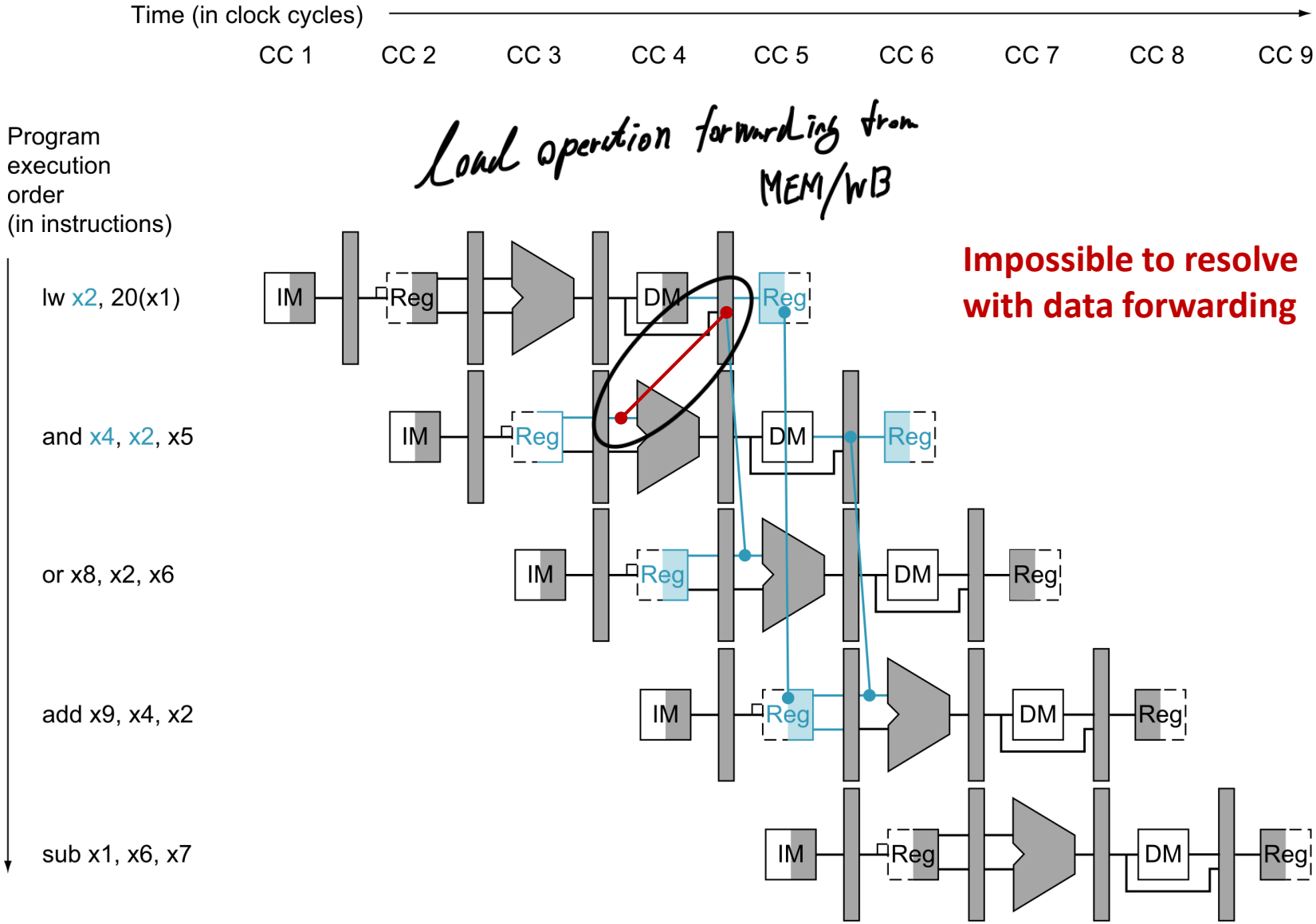
Check MEM data hazard occur

```
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.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
    forwardB = 01
```

Datapath with Forwarding



Load-Use Hazard (Load Interlock)



Load-Use Hazard Resolution

- **Action**
  - Impossible to solve with forwarding
  - Stall pipeline, insert bubble

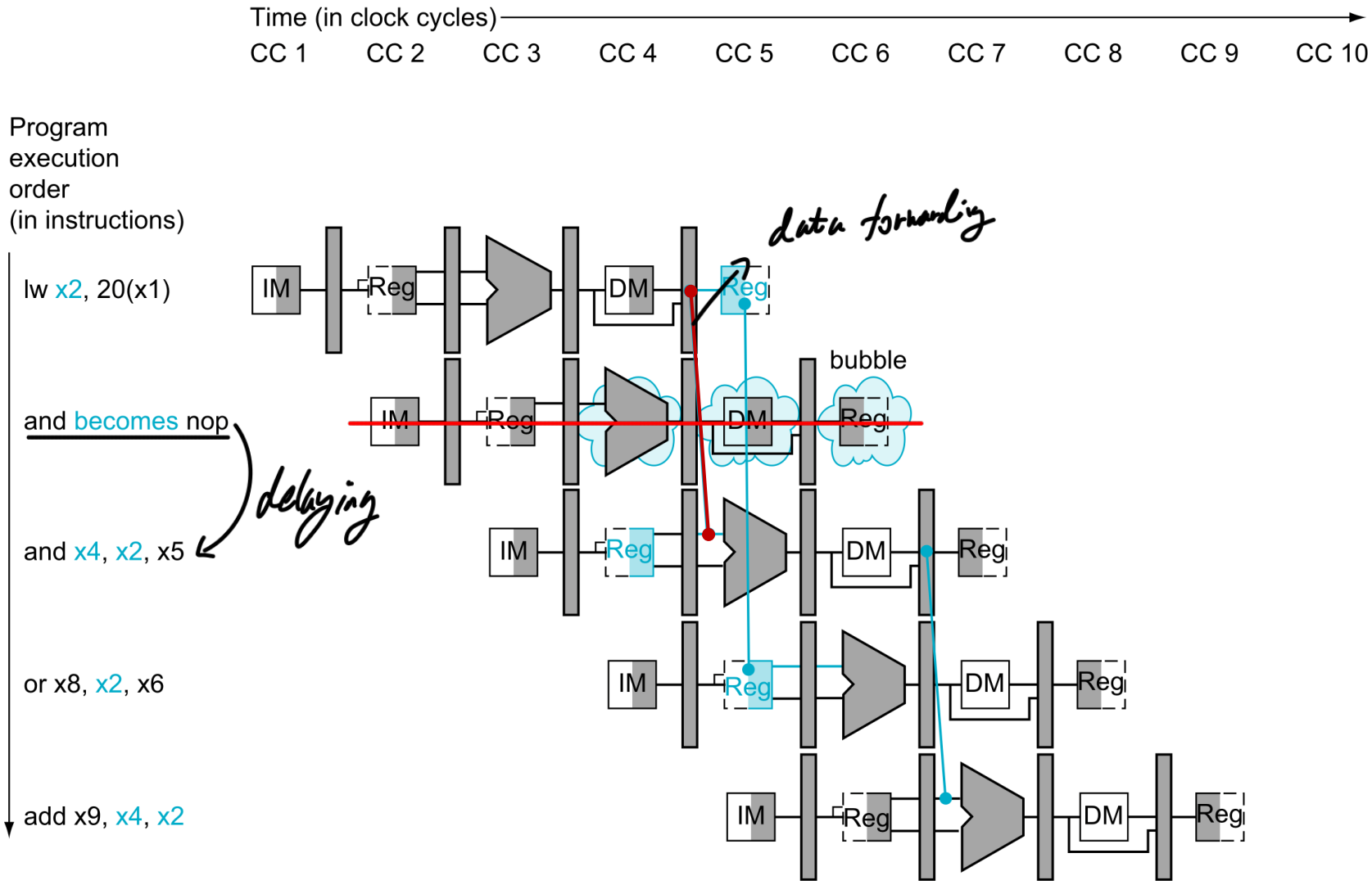
- **Detection**
  - Check for dependency when instruction is decoded in ID stage
  - ALU operand register numbers in ID stage are given by
    - `IF/ID.RegisterRs1`, `IF/ID.RegisterRs2`
  - Load-use hazard when

```
ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
     (ID/EX.RegisterRd = IF/ID.RegisterRs2))
```

Query type

Hazard condition

Load-Use Hazard Resolution



How to Stall the Pipeline

- **Force control values in ID/EX register to 0**
  - Inserts a nop (no operation) into the ID/EX register
    - By setting the control signals to 0
  - The nop bubble will propagate through EX, MEM and WB like a regular instruction

- **Prevent update of PC and IF/ID register**
  - Current instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `ld`
    that can subsequently be forwarded to EX stage

Datapath with Hazard Detection

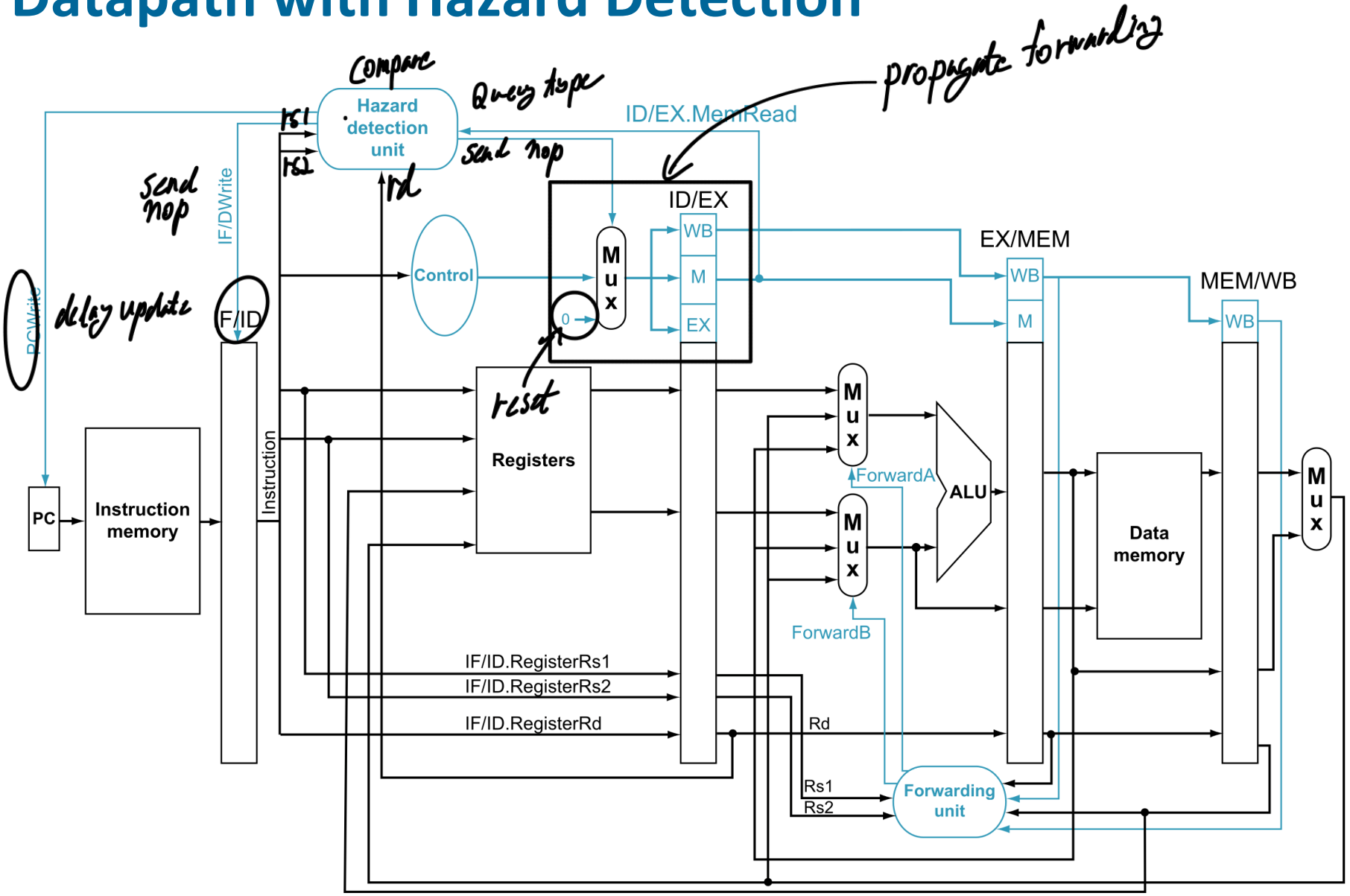


Illustration of Load Interlock Resolution

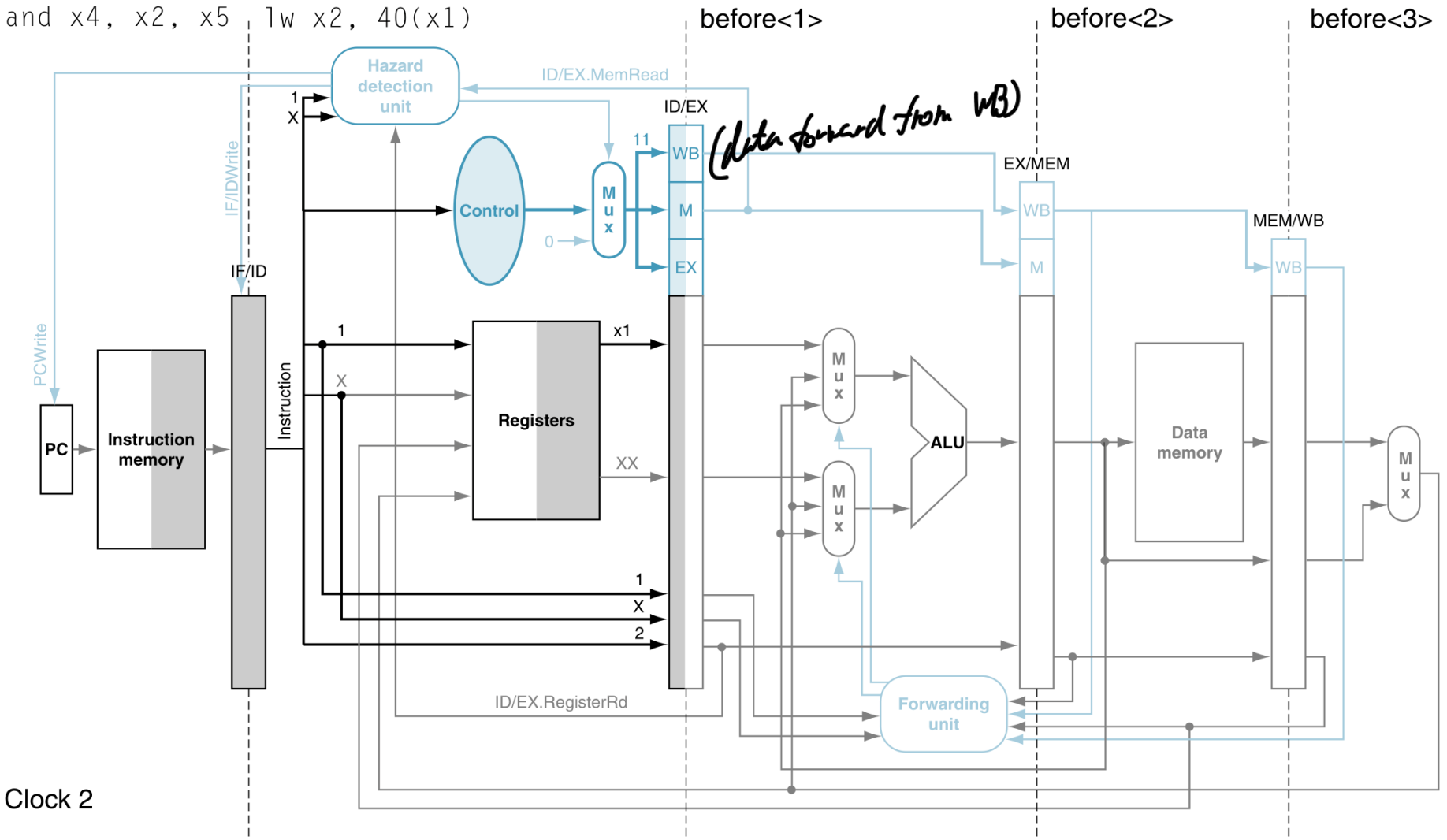


Illustration of Load Interlock Resolution

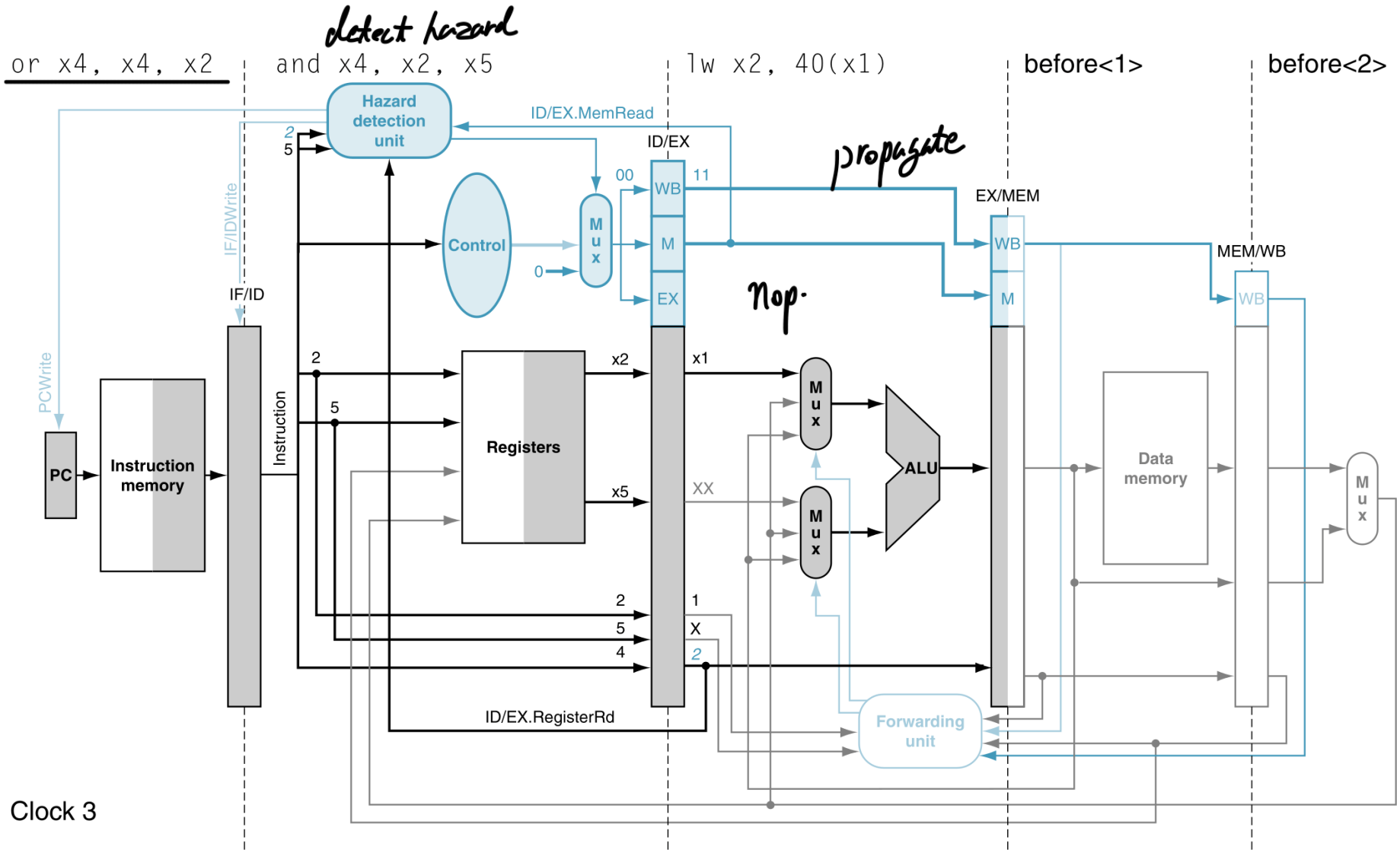


Illustration of Load Interlock Resolution

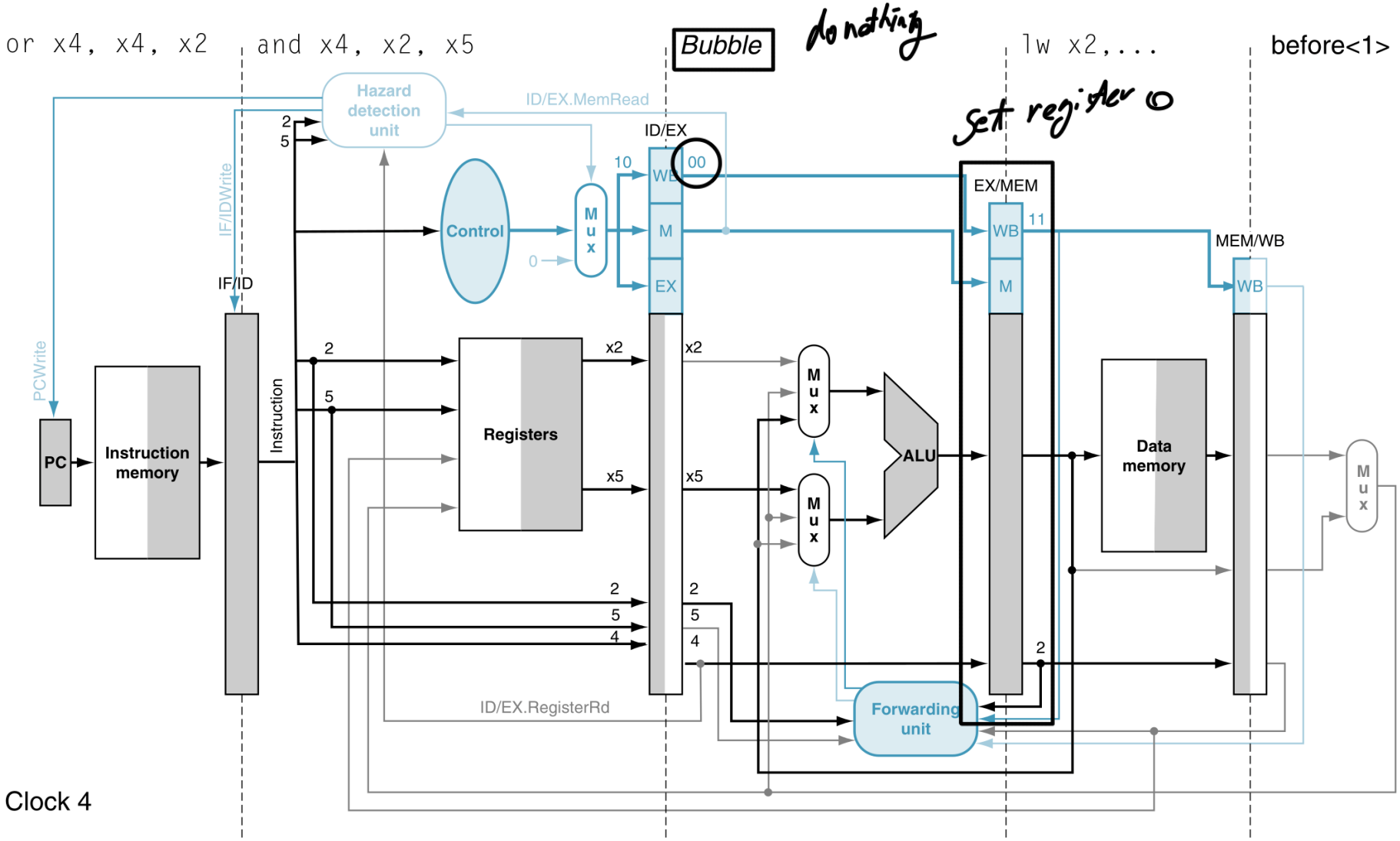


Illustration of Load Interlock Resolution

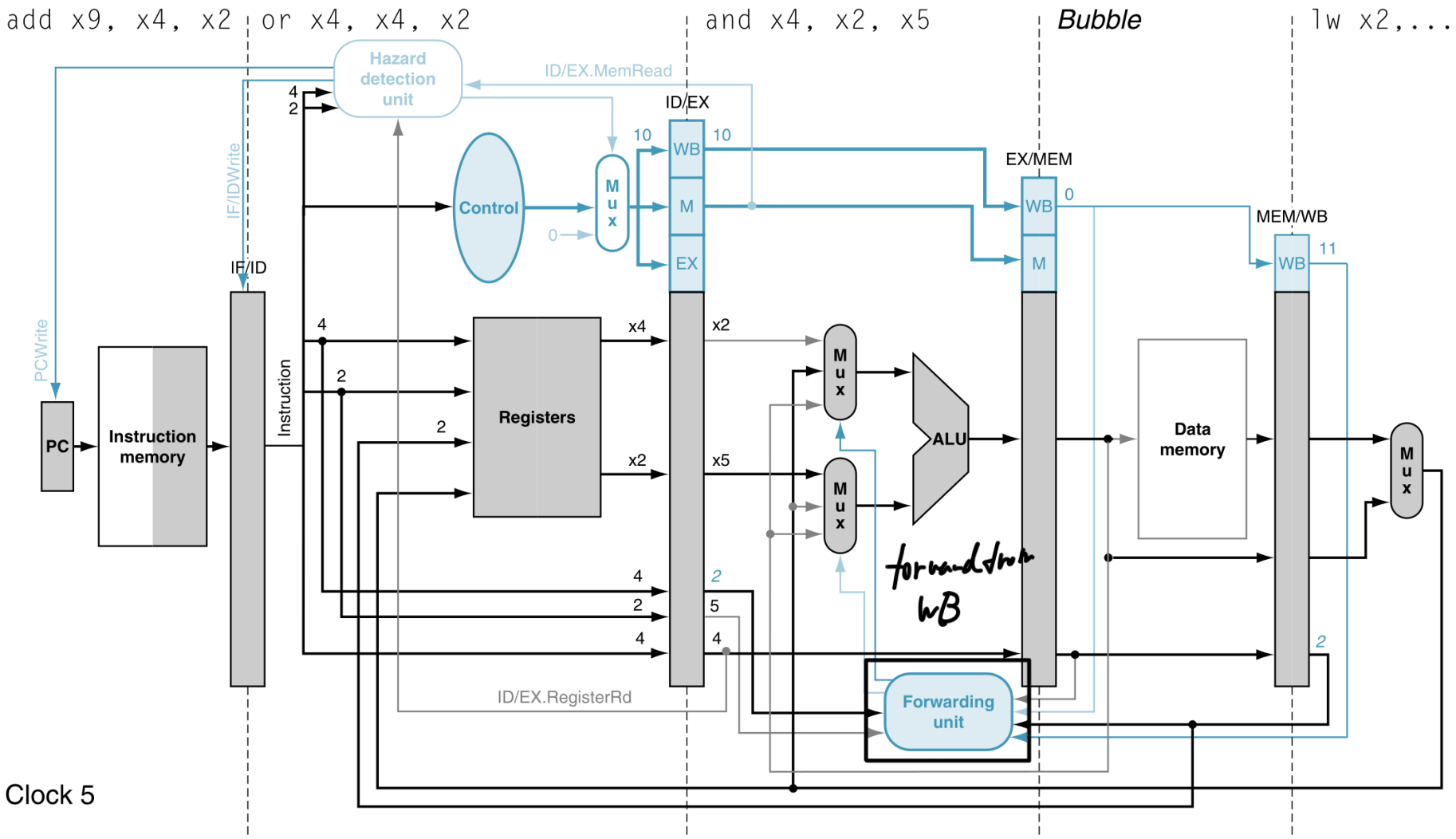
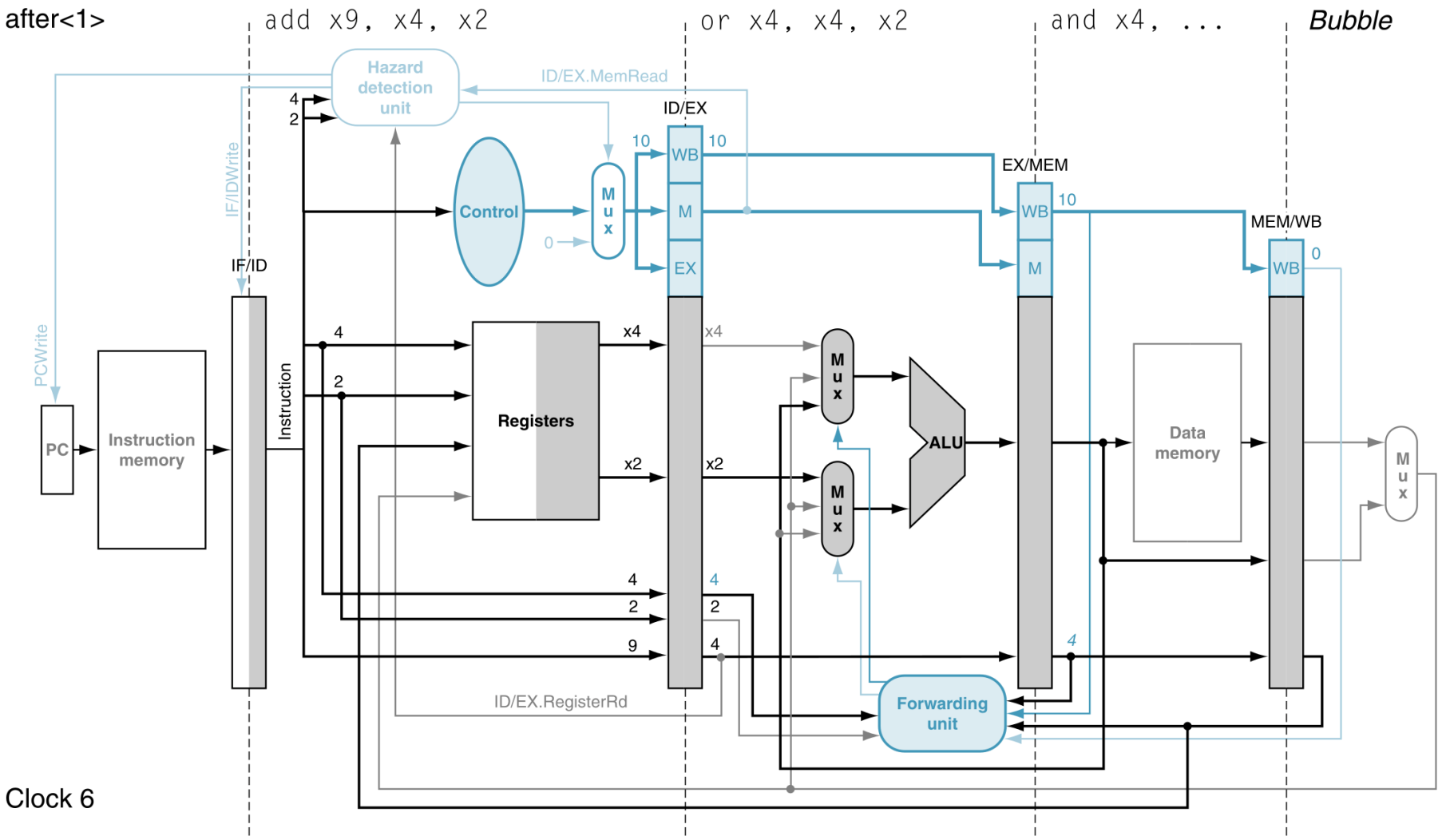


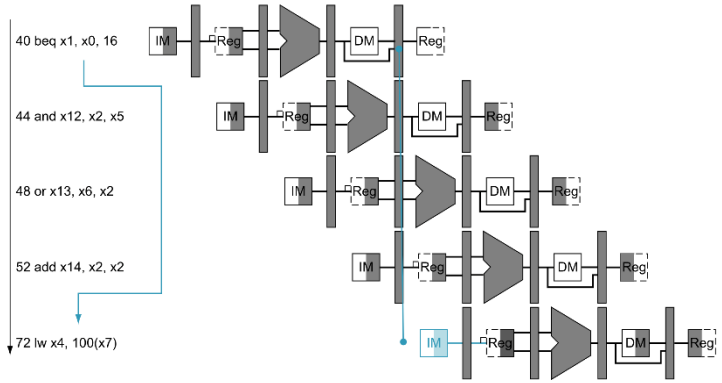
Illustration of Load Interlock Resolution



Stalls and Performance

- 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

Recap: Control Hazards

- Caused by PC-changing instructions
  - conditional branches, function call/return
  - may fetch wrong instructions

Branch Instruction	IF	ID	EX	MEM	WB				
Branch successor		~~IF~~	~~*stall*~~	~~*stall*~~	IF	ID	EX	MEM	WB
Branch successor + 1						IF	ID	EX	MEM
Branch successor + 2							IF	ID	EX
Branch successor + 3								IF	ID
Branch successor + 4									IF
Branch successor + 5									

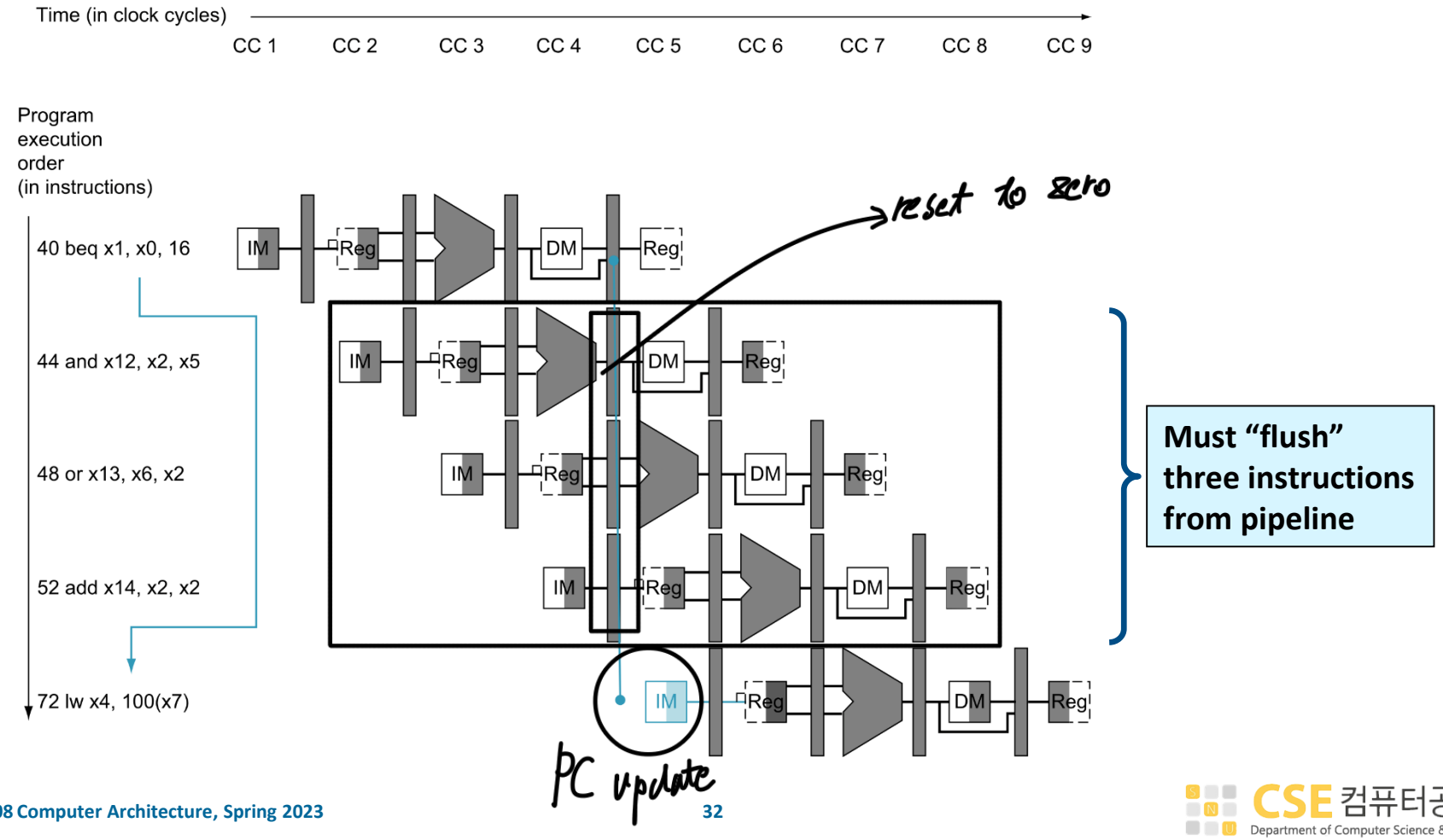
run after PC updated

Cancel process

Control Hazards

- Branch outcome determined in MEM
- Always assume branch not taken

static prediction



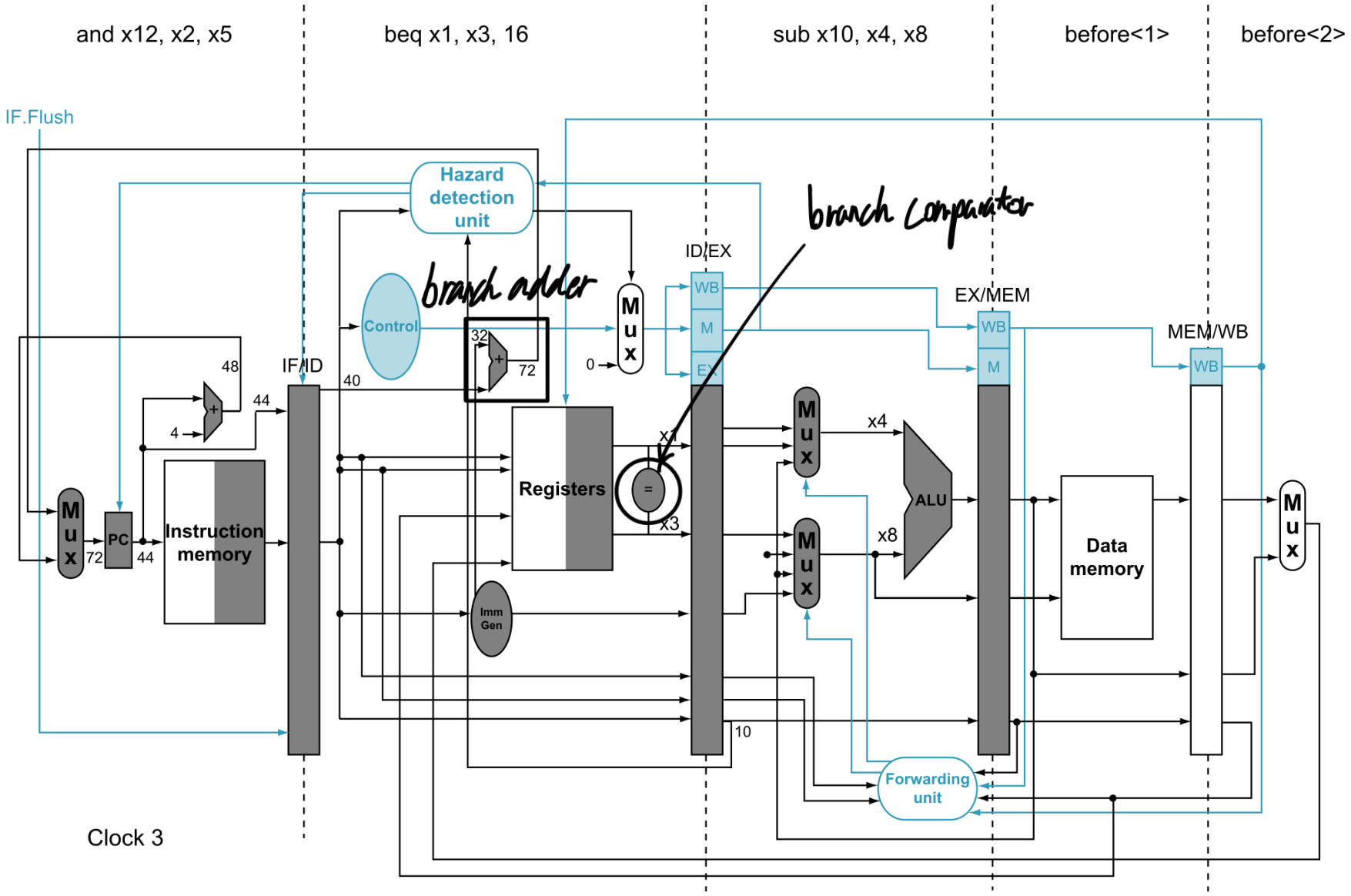
Reducing the Branch Delay

- **Move hardware to determine outcome to ID stage**
  - Easy
    - Target address adder
    - Register comparator
  - Difficult
    - Branch decision: comparing two registers in ID stage requires additional forwarding and hazard detection logic
- Example: branch taken

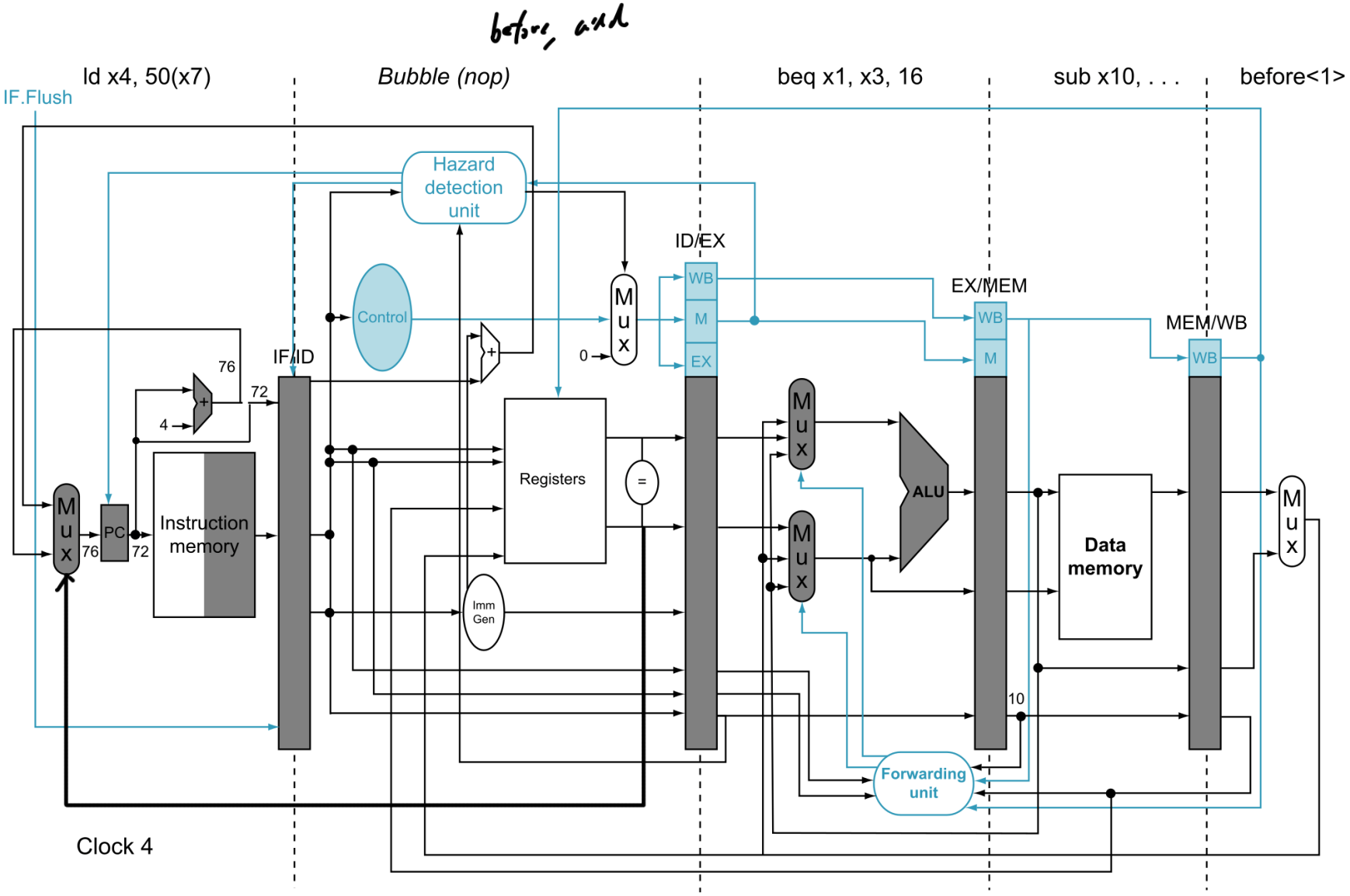
EX → ID

```
36:    sub    x10, x4, x8
40:    beq    x1, x3, 16     // PC-relative branch to 40+16*2 = 72
44:    and    x12, x2, x5
48:    or     x13, x2, x6
52:    add    x14, x4, x2
56:    sub    x15, x6, x7
       ...
72:    ld     x4, 50(x7)
```

Example: Branch Taken



Example: Branch Taken



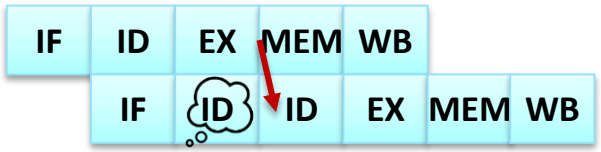
Cost of Moving Branch Test to ID

⇒ must forward to 'ID' phase

- Register operands may require forwarding
  - New forwarding logic from EX/MEM or MEM/WB pipeline registers to ID needed

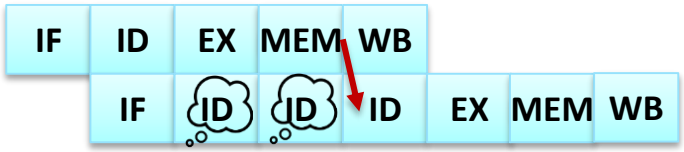
- Stalls due to data hazard (if data from forward, can't determine)
  ⇒ need stall
  - 1-cycle stall if the preceding instruction is an ALU instruction

```
add   x3,x4,x5
beq   x8,x3,10
```



  - 2-cycle stall if the preceding instruction is a load instruction

```
ld    x3,0(x4)
beq   x8,x3,10
```



Dynamic Branch Prediction

Static Branch Prediction
- always taken
- always not taken

50%

- In deeper and superscalar pipelines branch penalty is more significant

+2 : Based on previous decision

- Use dynamic prediction (caching address to compare before exe phase)
  - Branch prediction buffer (or branch history table) : history based predict (super expensive)
  - Indexed by recent branch instruction addresses (lower part of address)
  - Stores outcome (taken / not taken)

0x104
0x108
0x112
0x116
0x120

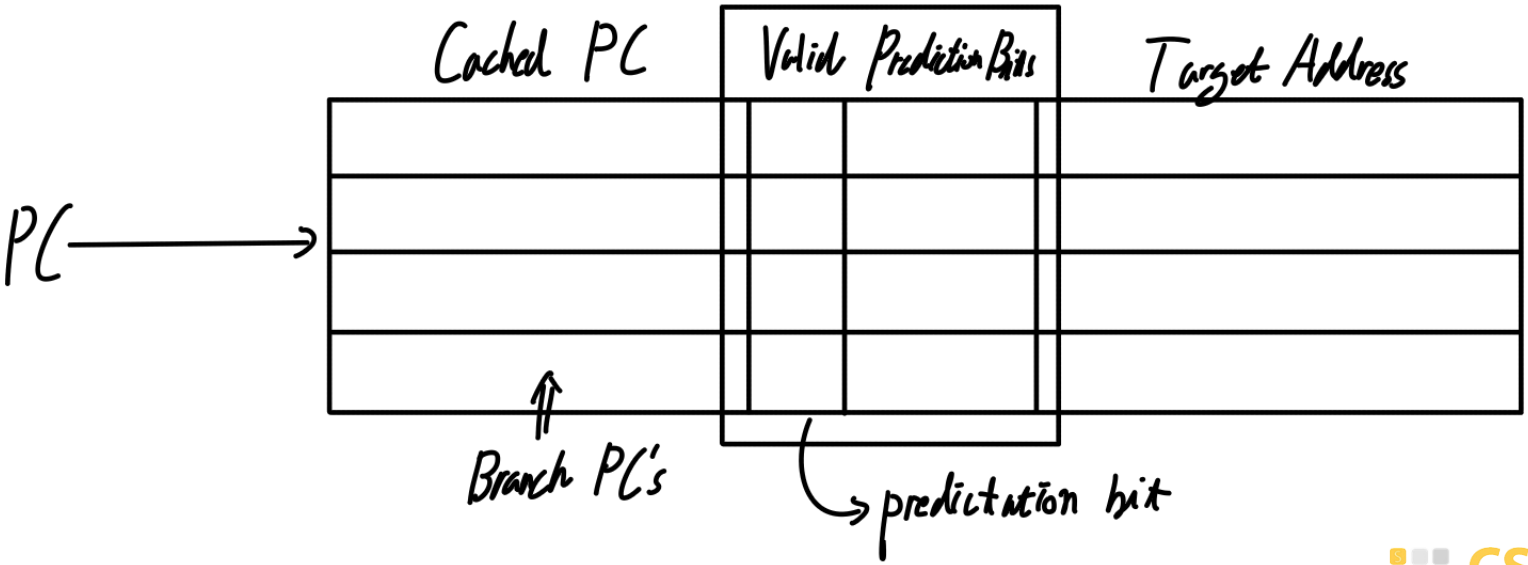
4 (=> ignore low 2 bits)

- To execute a branch
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction

Calculating Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch

- Branch target buffer (BTB)
  - Cache of target addresses
  - Indexed by PC when instruction fetched
  - If hit and instruction is branch predicted taken, can fetch target immediately



1-Bit Predictor Example

BTB

Address tag	Valid	Branch Target Buffer
…0	0	
…4	0	
…8	0	
…c	0	

- Code

```
                …
0x14380:  L1: add  x3,x3,x5
0x14384:      beq  x8,x3,L1
                …
```

- Execution

previous branch history

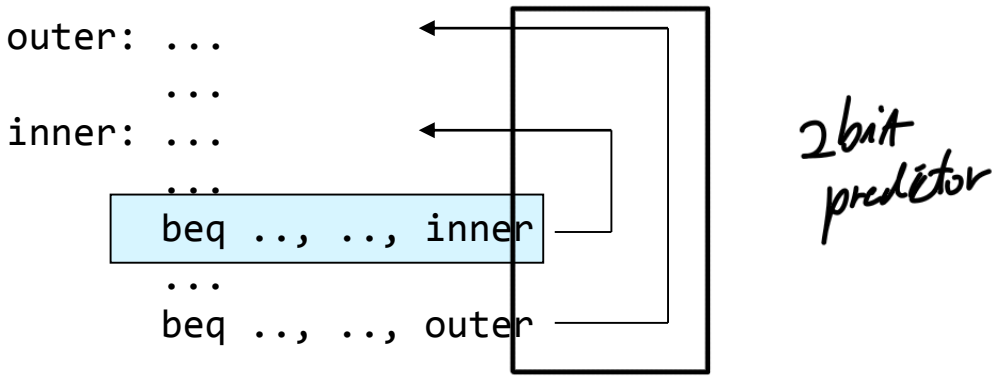
CC	Address	Instruction	x8	x3	Adr tag	Valid	BTB
10	14384	beq x8,x3,L1	10	7	4	0	
11	14380	add x3,x3,x5		8	4	1	14380
12	14384	beq x8,x3,L1	10	8	4	1	14380
13	14380	add x3,x3,x5		9			
14	14384	beq x8,x3,L1	10	9	4	1	14380
15	14380	add x3,x3,x5		10			
16	14384	beq x8,x3,L1	10	10	4	1	14380
17	14388	…					

first need

invalid now!

1-Bit Predictor: Shortcoming

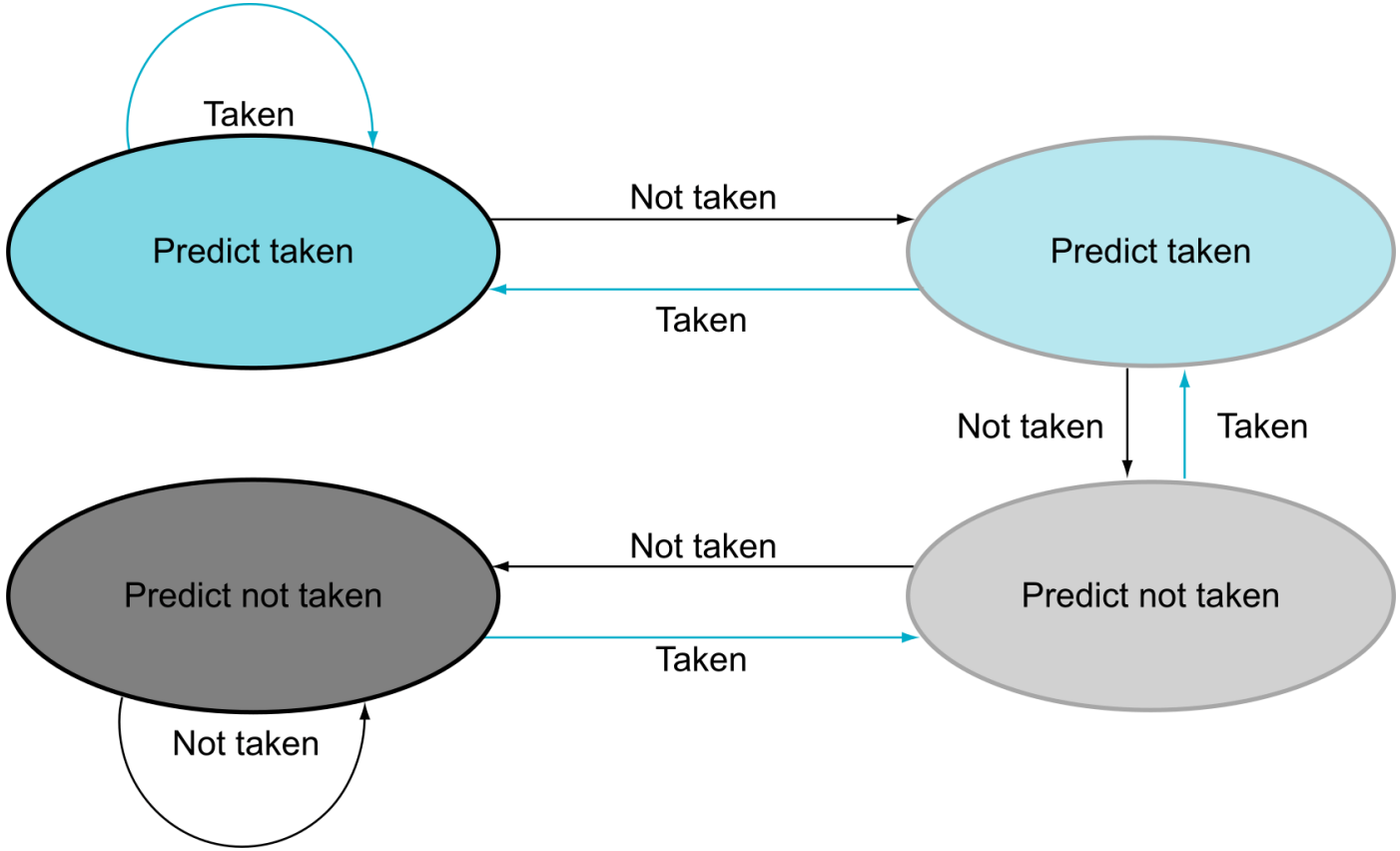
- Inner loop branches always mispredicted twice!



- 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 after two successive mispredictions

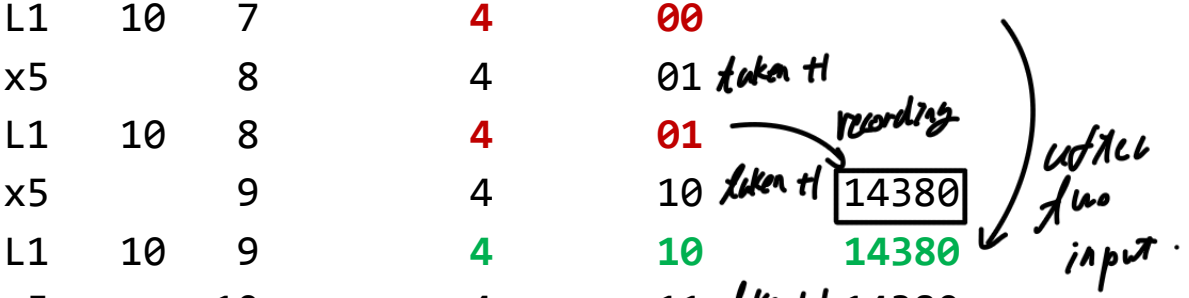


2-Bit Predictor Example

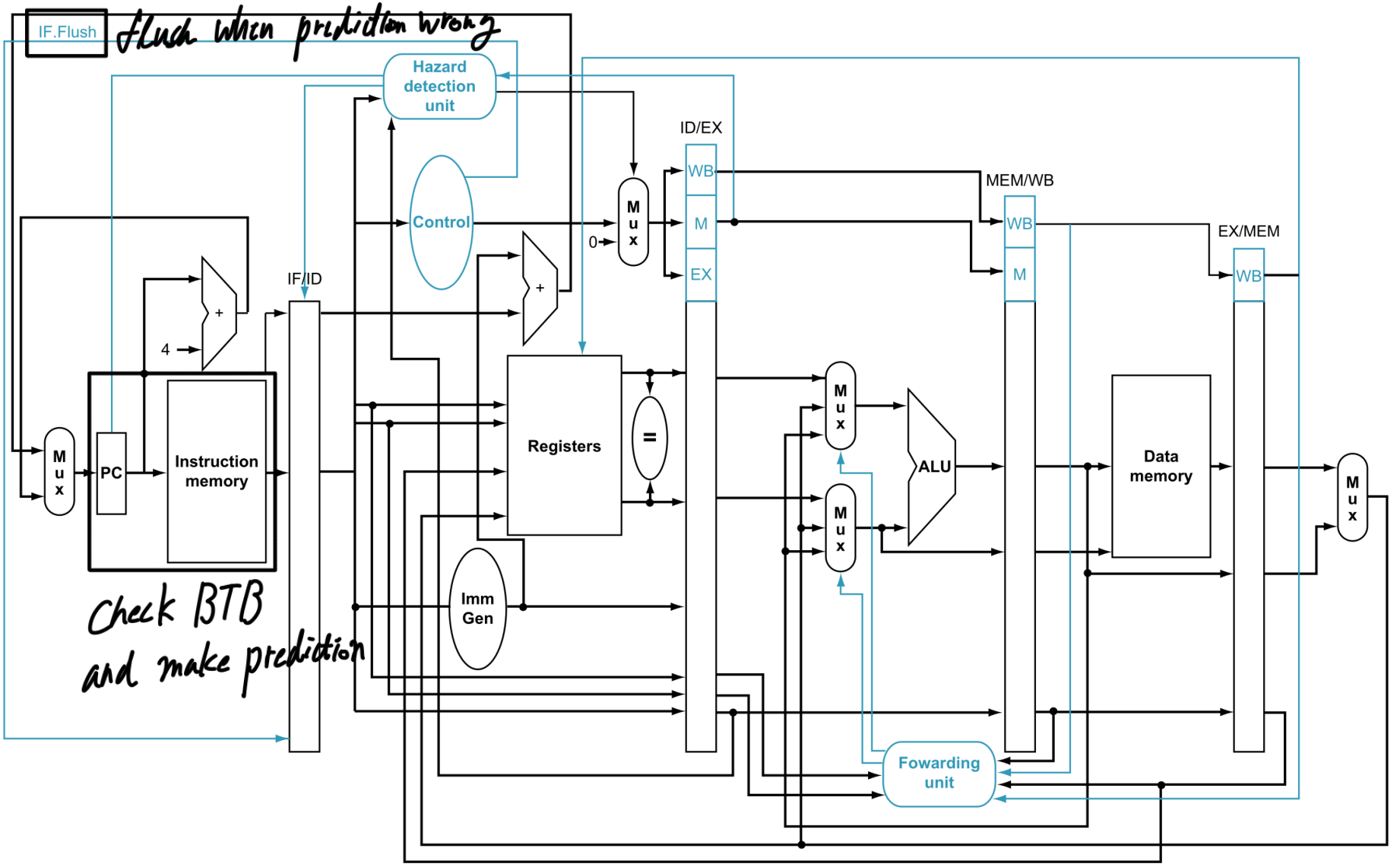
```
11: Predict taken
10: Predict taken
01: Predict not taken
00: Predict not taken
```

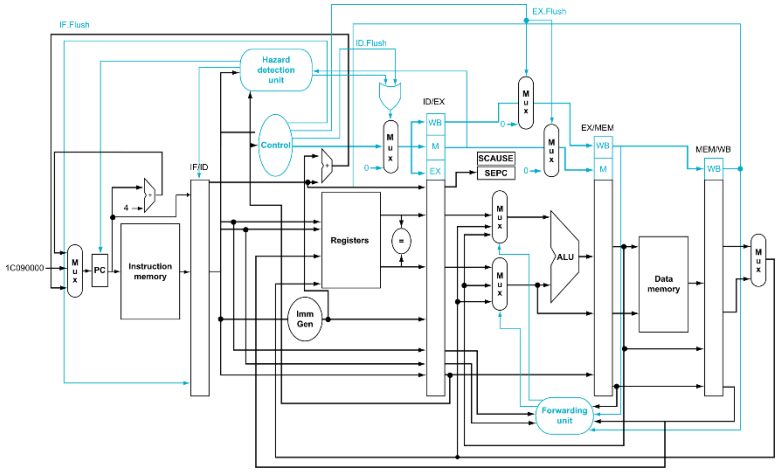
- Execution

	CC	Address	Instruction	x8	x3	Adr tag	Pred	BTB
	10	14384	beq x8,x3,L1	10	7	4	00	
	11	14380	add x3,x3,x5		8	4	01	
inner Loop1	12	14384	beq x8,x3,L1	10	8	4	01	
	13	14380	add x3,x3,x5		9	4	10	14380
	14	14384	beq x8,x3,L1	10	9	4	10	14380
	15	14380	add x3,x3,x5		10	4	11	14380
	16	14384	beq x8,x3,L1	10	10	4	11	14380
	17	14388	…			4	10	14380
	…							
	20	14384	beq x8,x3,L1	10	7	4	10	14380
	21	14380	add x3,x3,x5		8	4	11	14380
inner Loop2	22	14384	beq x8,x3,L1	10	8	4	11	14380
	23	14380	add x3,x3,x5		9	4	11	14380
	24	14384	beq x8,x3,L1	10	9	4	11	14380
	25	14380	add x3,x3,x5		10	4	11	14380
	26	14384	beq x8,x3,L1	10	10	4	11	14380
	27	14388	…			4	10	14380



Datapath and Control





Exceptions

Accident type	Source	RISC-V
System reset	External	Exception
I/O device request	External	Interrupt
OS system call	Internal	Exception
Undefined Operation	Internal	Exception
Hardware Error	In/External	Exception / Interrupt

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently

ex) divide by zero

- Exceptions
  - RISC-V: any unexpected change in control flow (either internal or external)
  - Internal exceptions arise within the CPU (e.g., undefined opcode, syscall, …)

명확한 구분이 있는건 아님

- Interrupt
  - RISC-V: event from an external I/O controller
  - e.g., hard disks, network adapters, keyboard, …

handle in OS

- Dealing with them without sacrificing performance is hard
  - But also not necessary if exceptions do not happen frequently
  - Make the *common* case fast

(Non-error phase)

Handling Exceptions in RISC-V (Undefined Operation & Hardware error)

- Save PC of offending (or interrupted) instruction
  - Supervisor Exception Program Counter (SEPC)

save address for decision or return

- Save indication of the problem

Send to O.S. (specific address)

  - Supervisor Exception Cause Register (SCAUSE) : reason
  - 64-bits, but most bits unused
  - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …

detect reason of error by looking for SEPC + state register

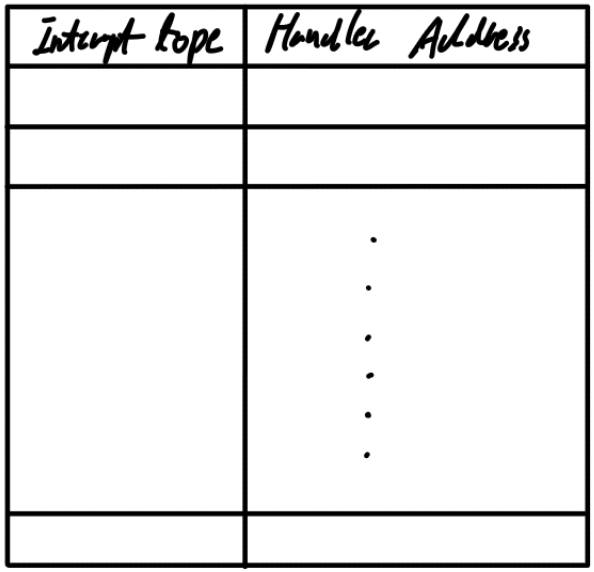
- Jump to handler
  - Assume at `0x0000 0000 1C09 0000` (if error's handler address)

An Alternate Mechanism

- Vectored interrupts
  - Handler address determined by the cause

- Exception vector address to be added to a vector table base register: (Basic address + Vectoize address)
  - Undefined opcode: 00 0100 0000
  - Hardware malfunction: 01 1000 0000

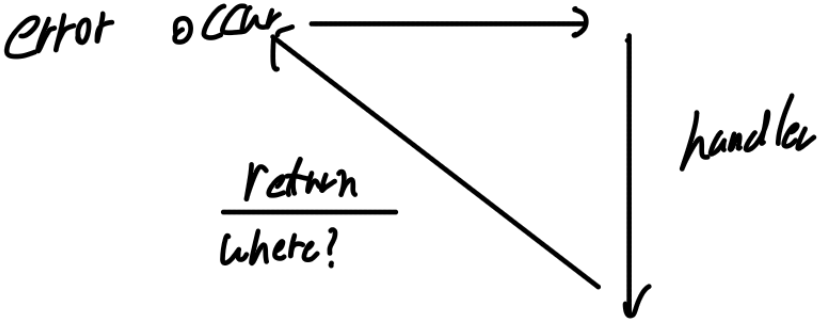
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler



Handler Actions

from information and compare it with SCAUSE

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - Use SEPC to return to program
- Otherwise
  - Terminate program
  - Report error using SEPC, SCAUSE, …



① after 'sys call' : next operation

② after 'load' : same address

③ general : stop program

Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction:
    Refetched and executed from scratch

- PC saved in SEPC register
  - Identifies causing instruction

Exceptions in a Pipeline

- Another form of control hazard

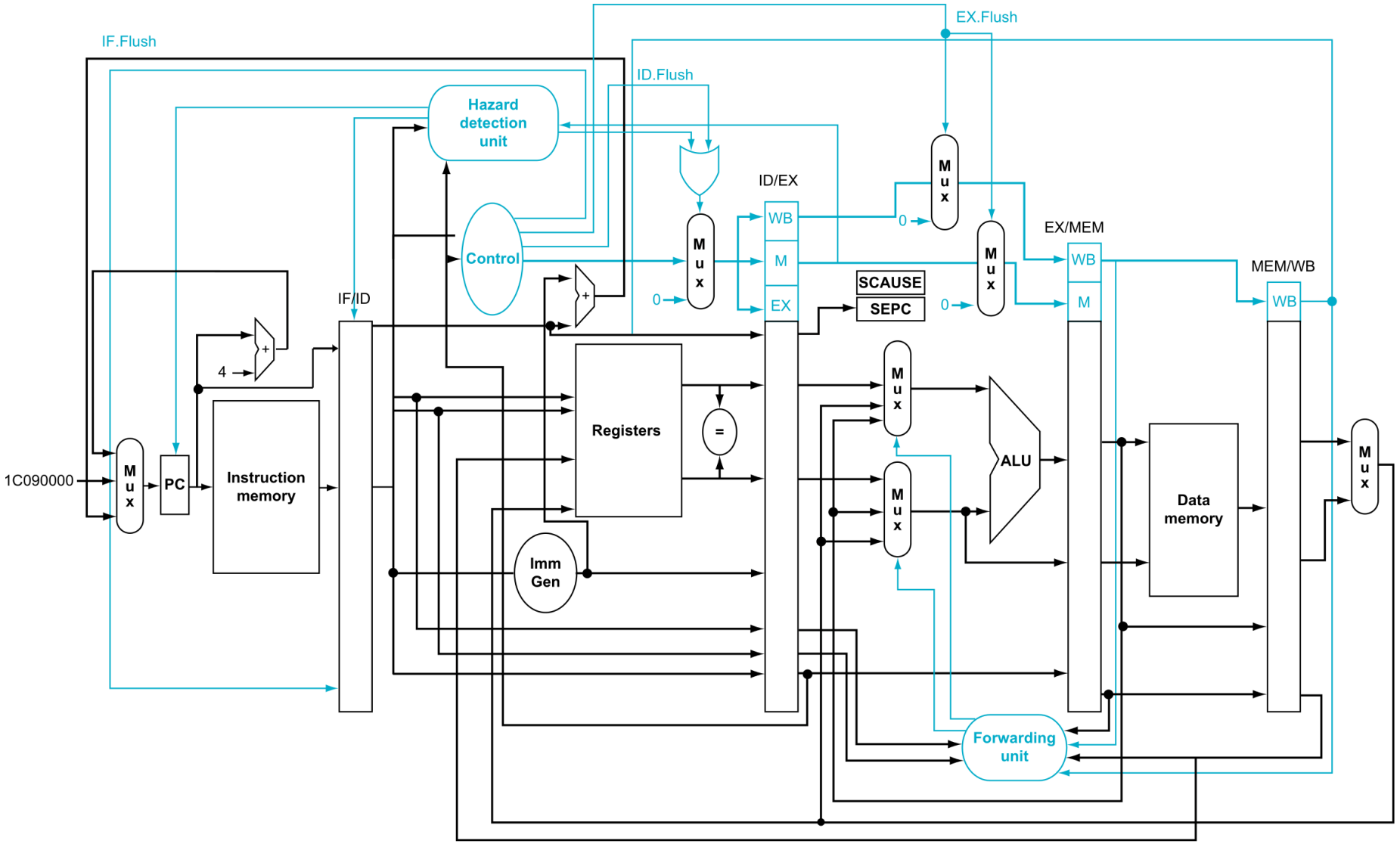
- Consider malfunction on add in EX stage

```
add   x1, x2, x1
```

  - Prevent x1 from being clobbered : operation fail, so must not overwrite
  - Complete previous instructions
  - Flush add and subsequent instructions (flush register)
  - Set SEPC and SCAUSE register values (address and reason)
  - Transfer control to handler

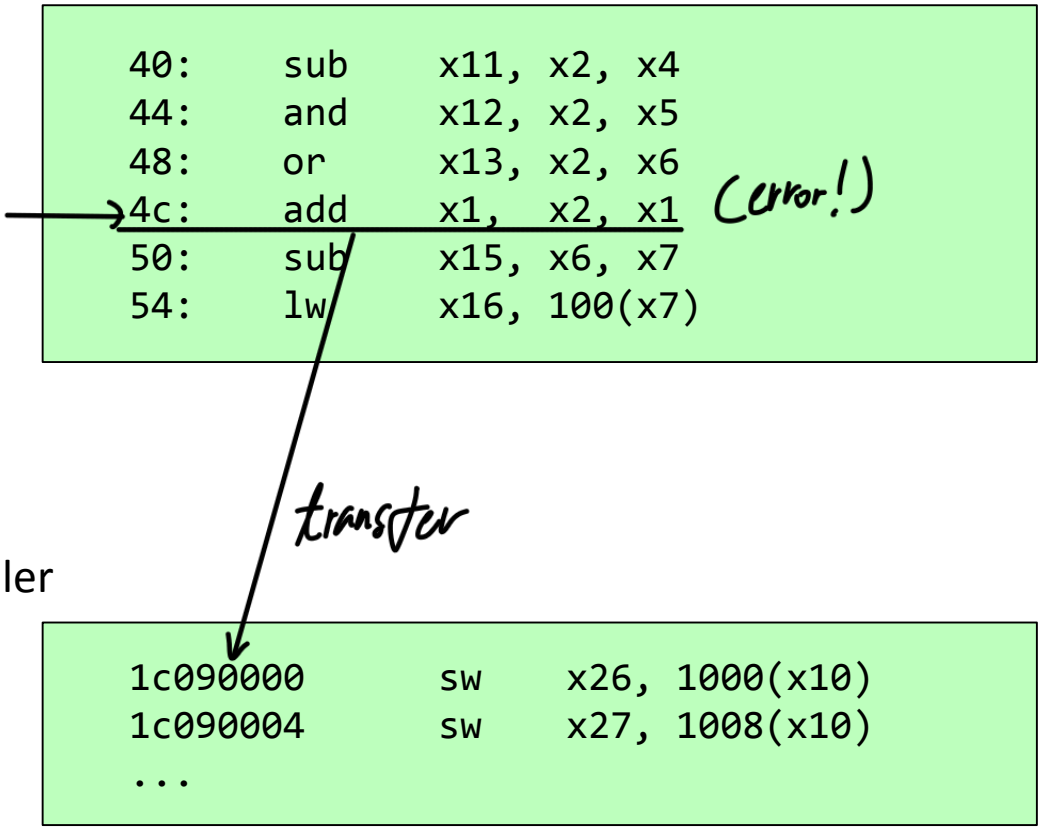
- Similar to mispredicted branch: use much of the same hardware

Pipeline with Exception Control



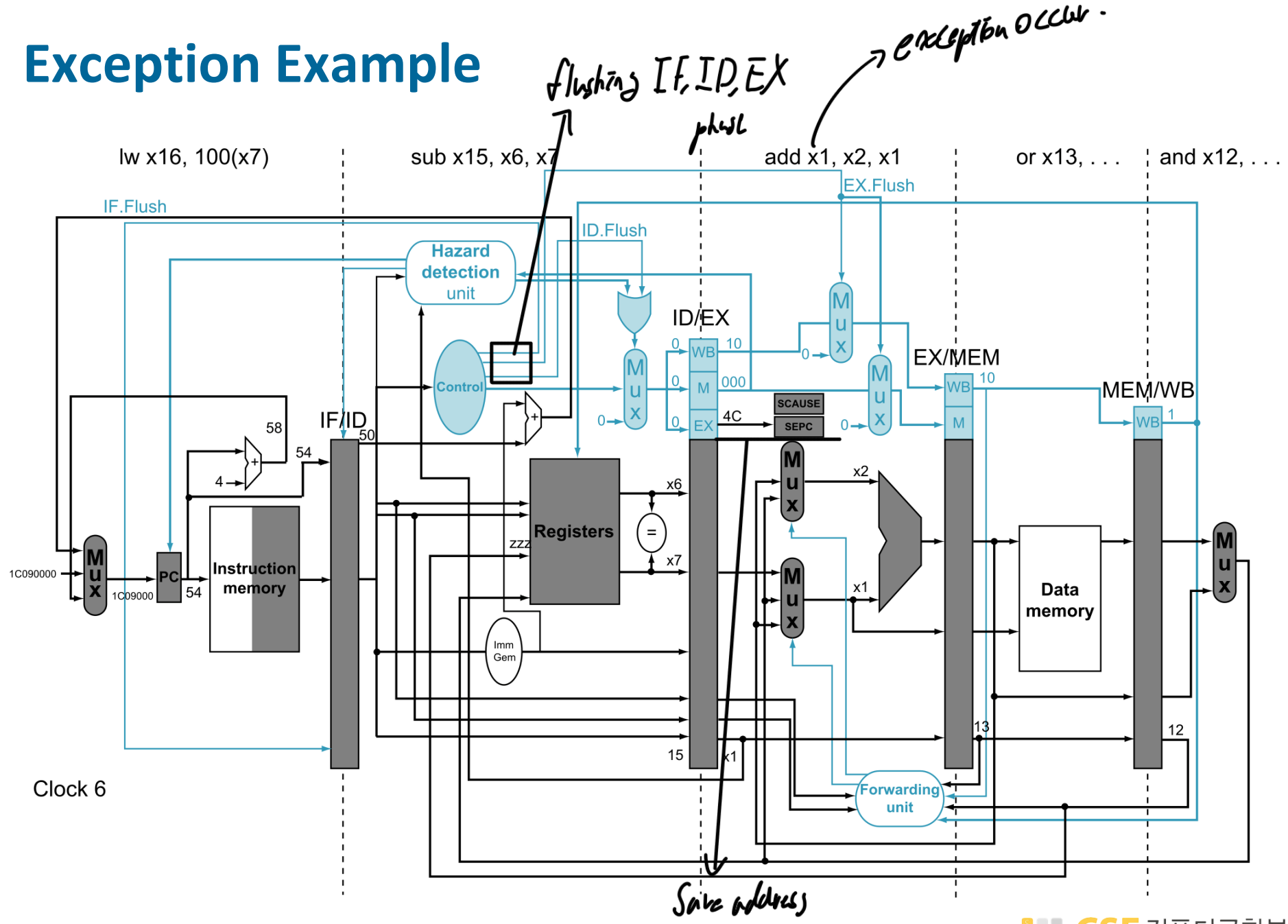
Exception Example

- Exception (hardware fault) on add in EX stage

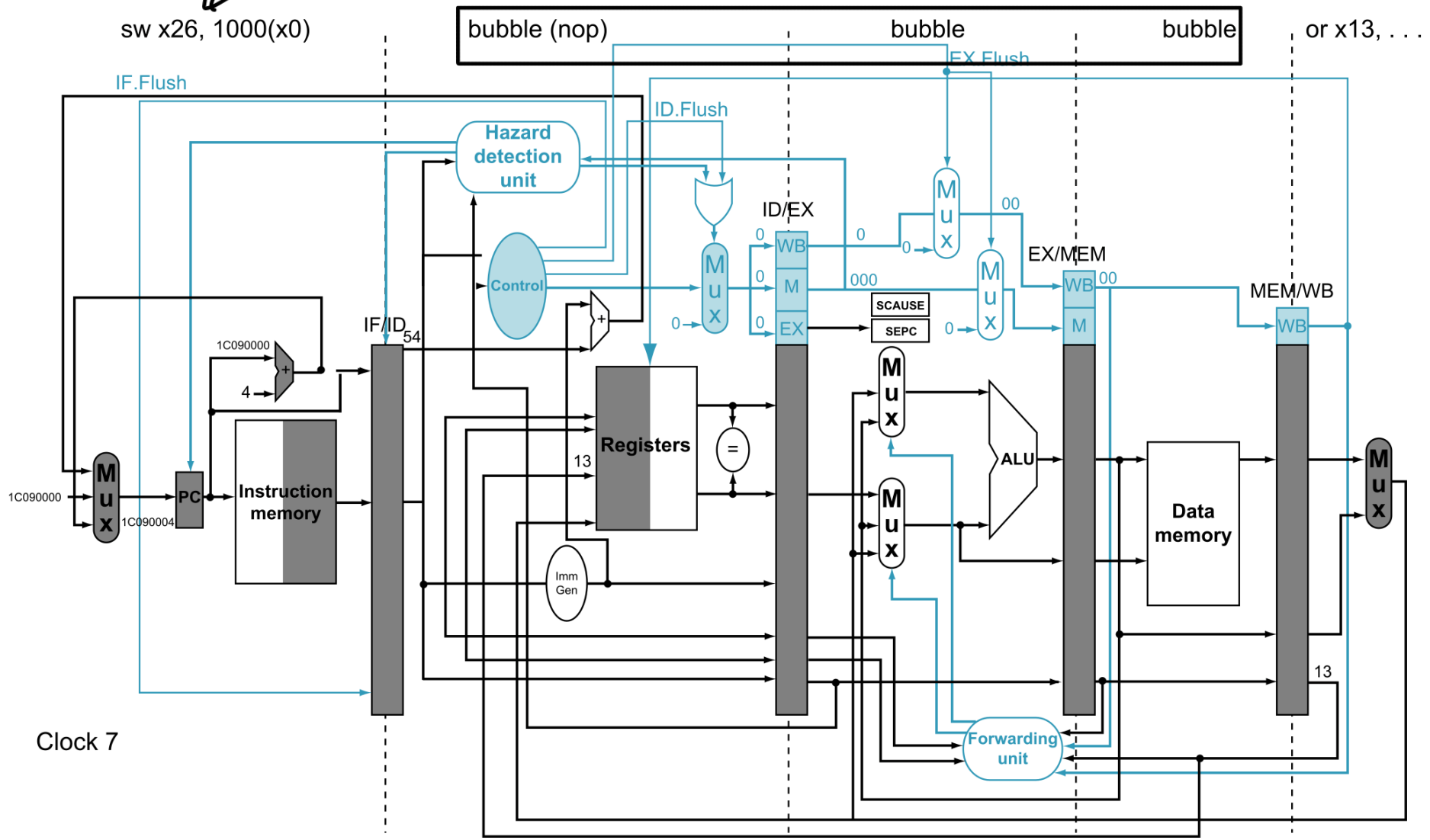


- Handler

Exception Example

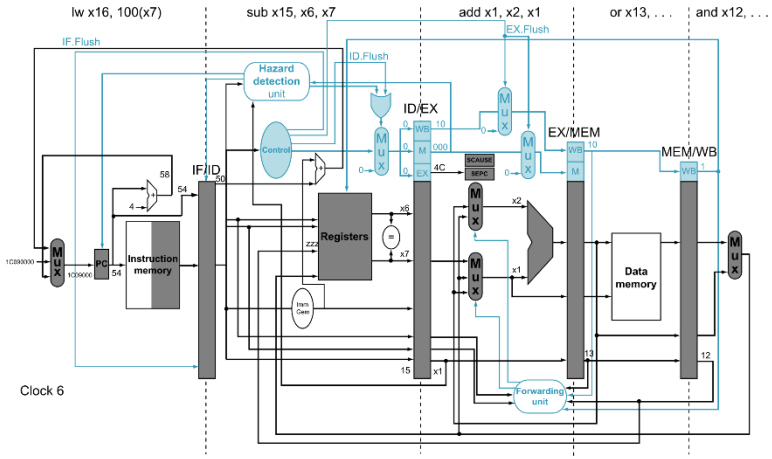


Exception Example



More on Exceptions

- **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
    - Exception not associated with exact instruction that caused the interrupt
  - Majority of modern processors, incl. RISC-V, support precise interrupts



Module Summary

Module Summary

- **Data hazards**
  - Solved with data forwarding and hazard detection
  - Can eliminate stalls completely except for load interlock
  - Relatively simple

- **Control hazards**
  - Reduce branch penalty by moving branch decision to the ID stage
    - At the expense of additional forwarding paths and hazard detection logic
  - Speculative execution
    - Speculate, flush pipeline if incorrect
    - Branch prediction with 1 and 2 bit buffers

- **Exceptions**
  - Unforeseen internal or external events trigger a jump to an interrupt handler
  - Address of affected instruction and cause of exception recorded in registers
  - Exceptions in pipelines are passed on, similar to other data, and triggered when the instruction retires

Storage Technology and Trend

SATA, M2, NVMe, PCIe

Multi-Level Cells