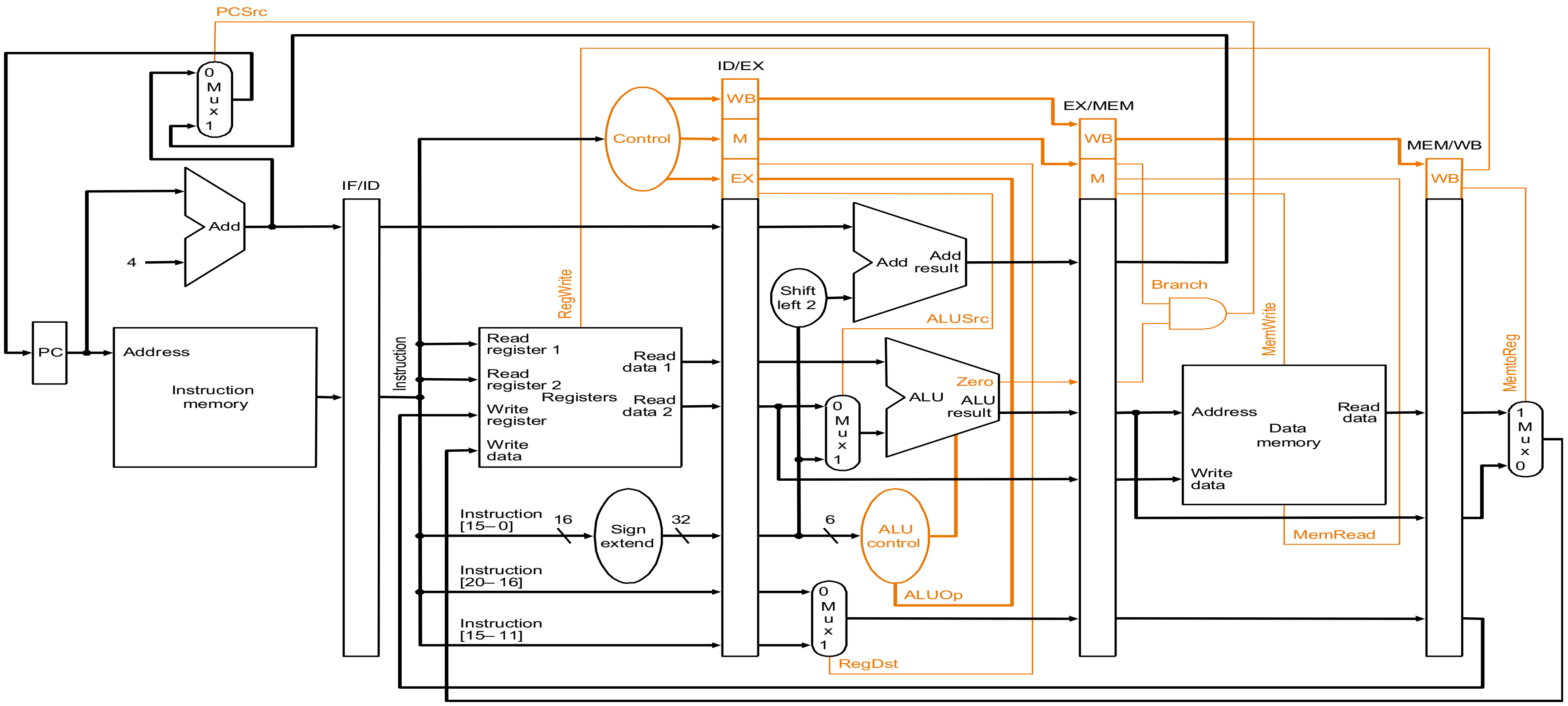
Lecture 7: Pipelining (II)

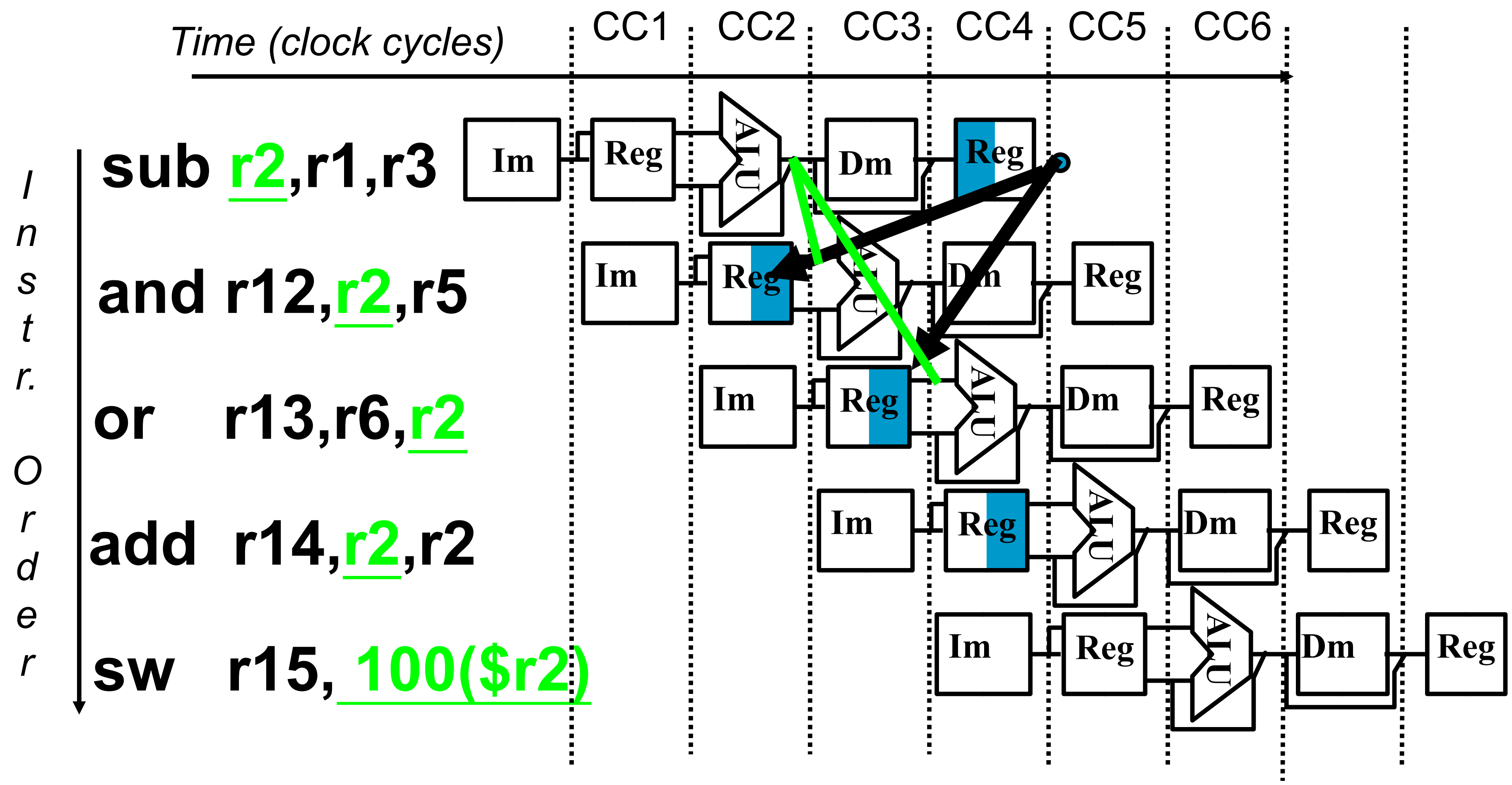
1. Data Hazards and Forwarding
2. Control Hazard Solution
3. How to Handle Exception

Datapath with Control

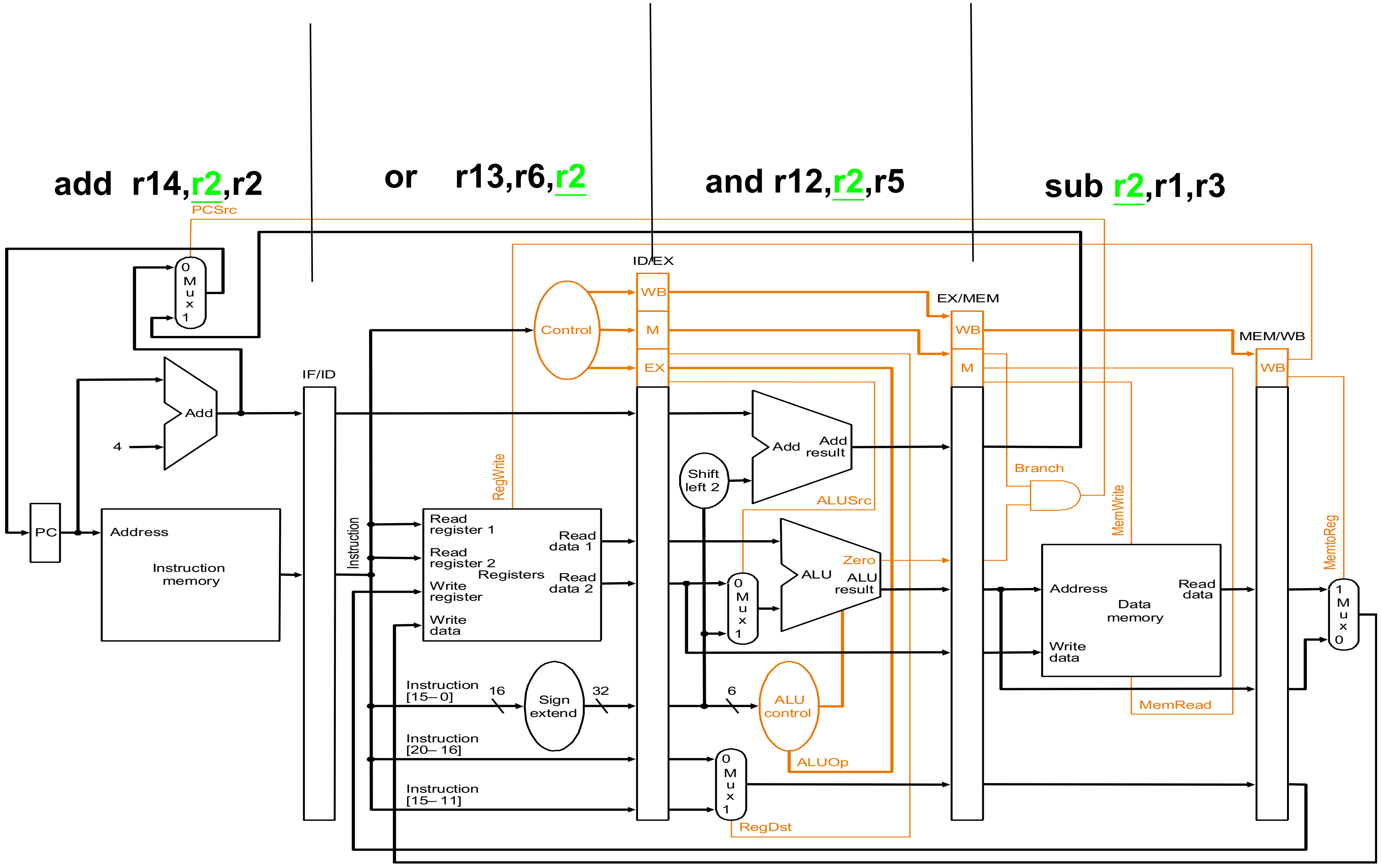


1. How to implement “data forwarding”?
2. How to detect load-use hazard? How to stall pipeline?
3. How to resolve branch in the decode stage?
4. How to flush pipeline?

Data Dependence Detection & Forwarding

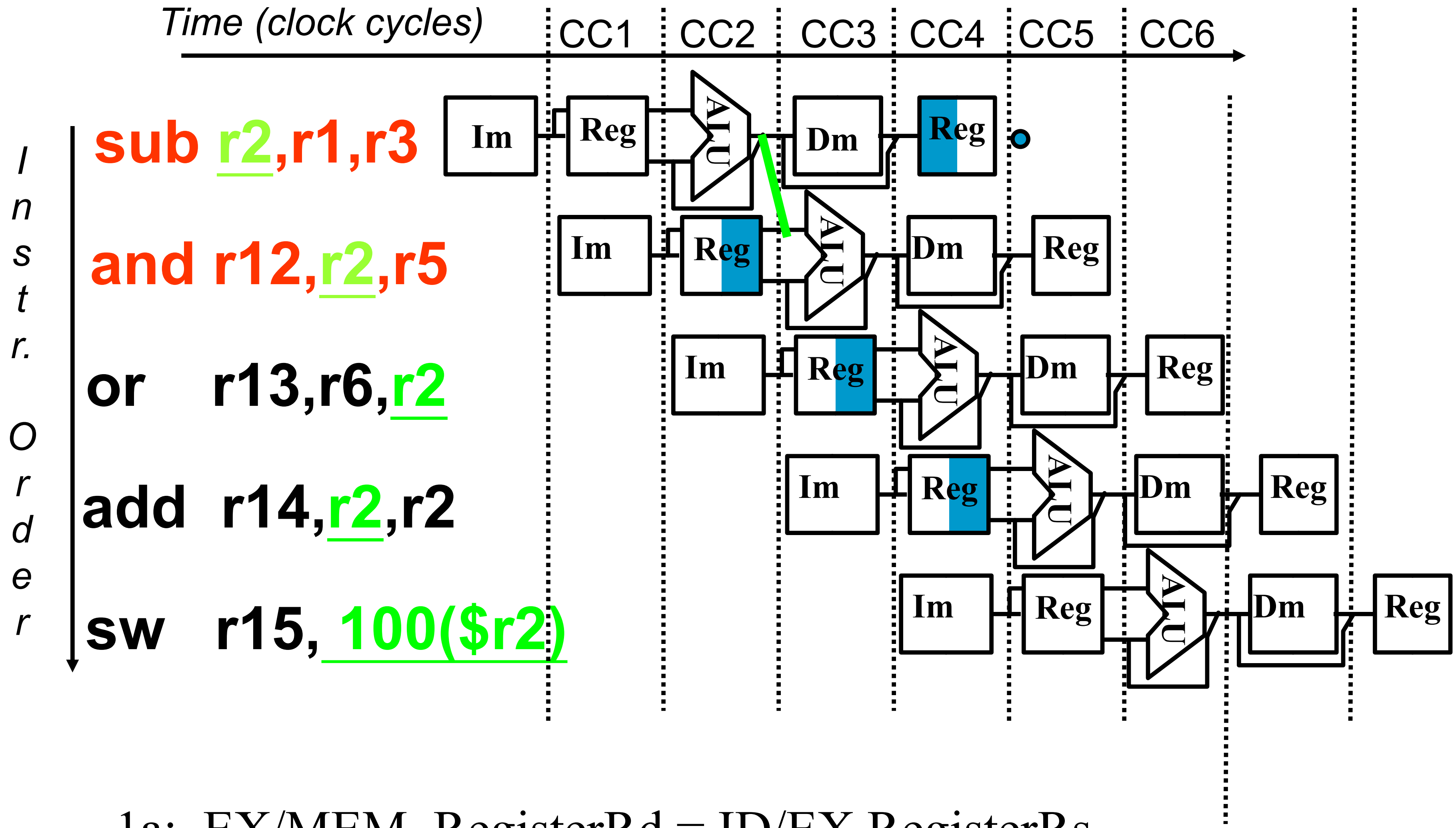


add r14,r2,r2   or   r13,r6,r2   and r12,r2,r5   sub r2,r1,r3



CC3

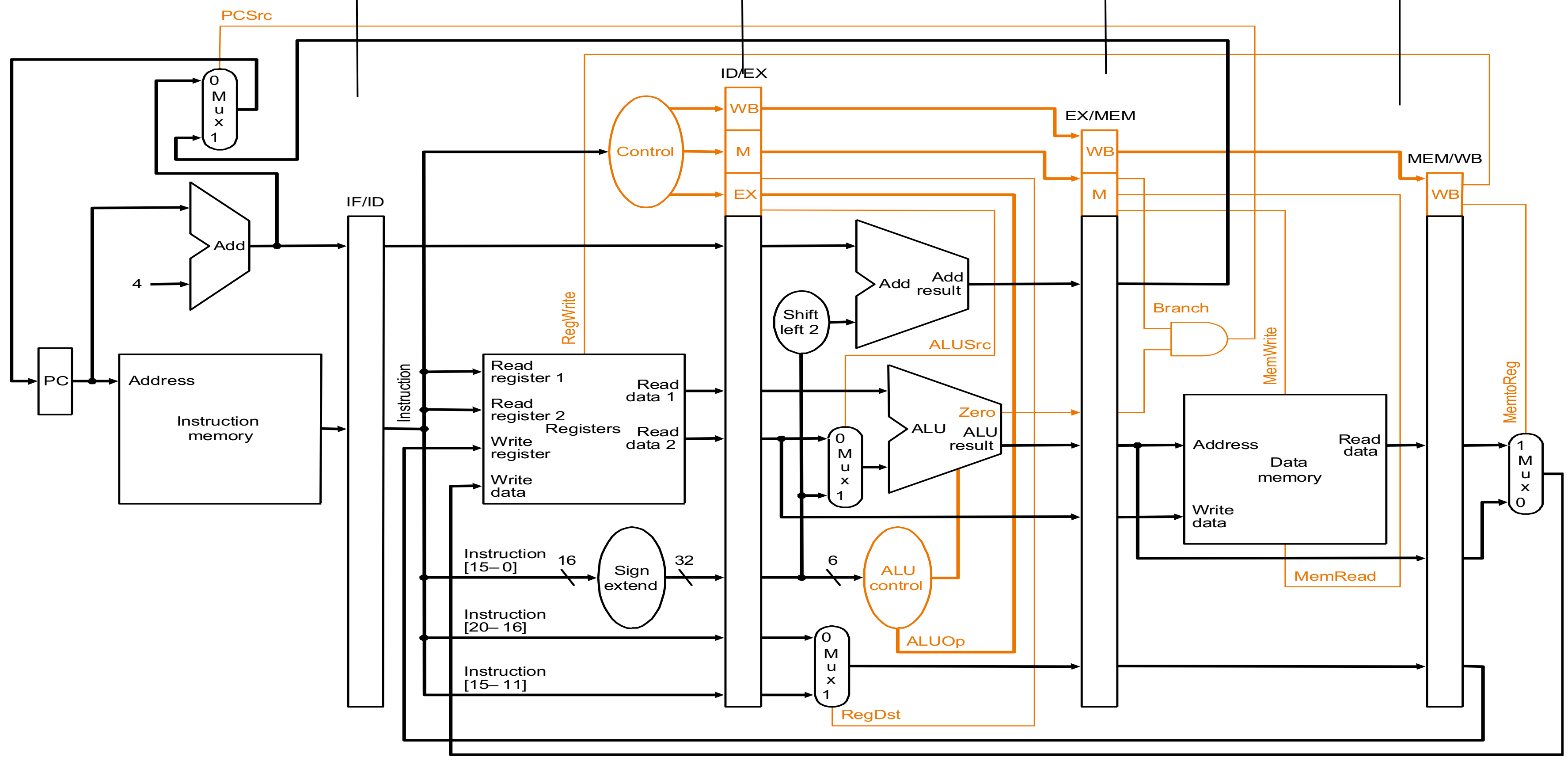
How to detect dependency between (sub, and)?



1a: EX/MEM. RegisterRd = ID/EX.RegisterRs

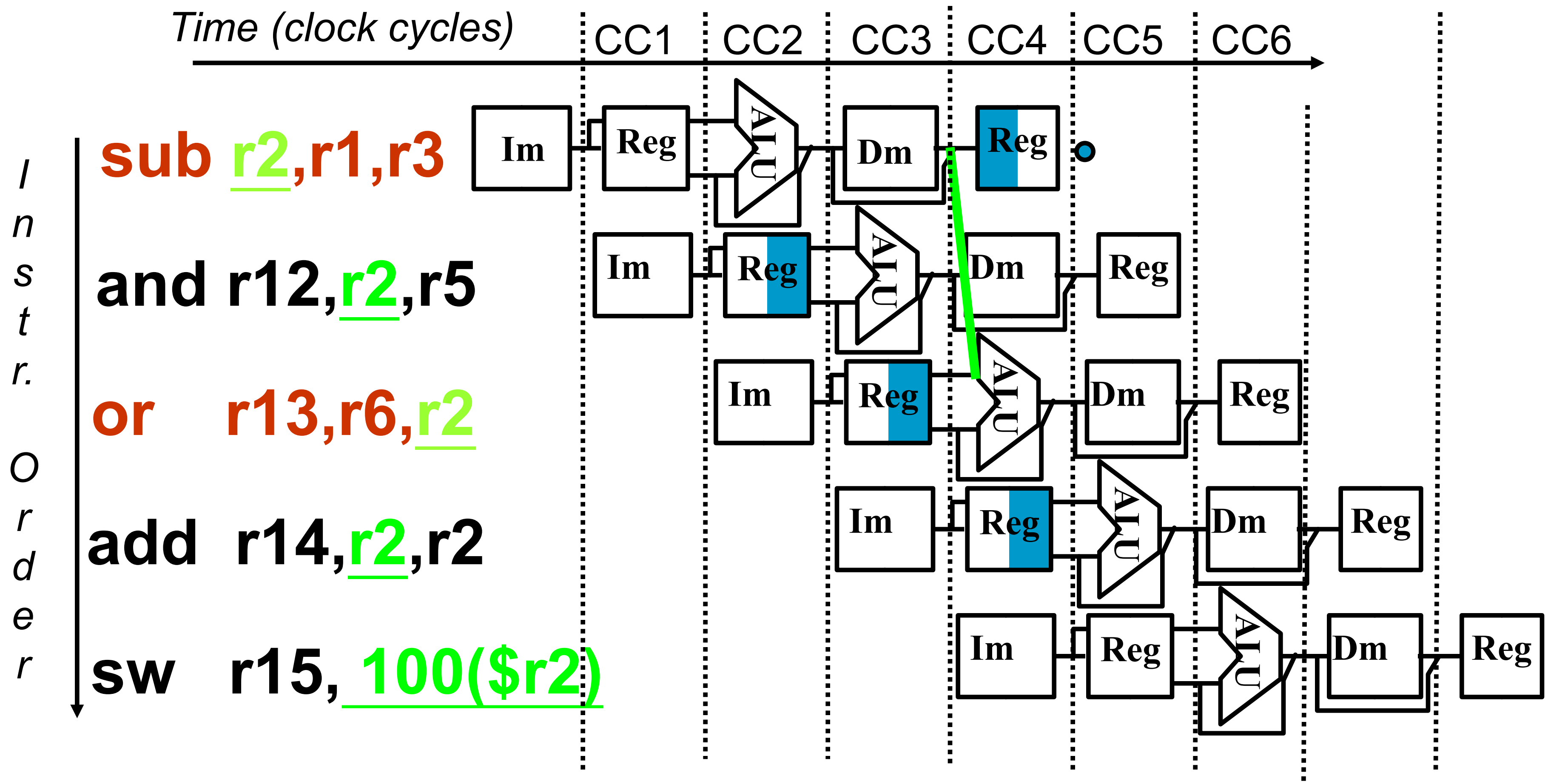
1b: EX/MEM. RegisterRd = ID/EX.RegisterRt

sw r15, 100($r2) | **add r14,r2,r2** | **or r13,r6,r2** | **and r12,r2,r5** | **sub r2,r1,r3**



CC4

How to detect dependency between (sub, or)?



2a: MEM//WB.RegisterRd = ID/EX.RegisterRs (sub & or)

2b: MEM/WB.RegisterRd = ID/EX.RegisterRt

Data Dependence Detection (cont.)

- Hazard conditions:
  - 1a: EX/MEM. RegisterRd = ID/EX.RegisterRs (sub & and)
  - 1b: EX/MEM. RegisterRd = ID/EX.RegisterRt

  EX hazard

  - 2a: MEM/WB.RegisterRd = ID/EX.RegisterRs (sub & or)
  - 2b: MEM/WB.RegisterRd = ID/EX.RegisterRt

  MEM hazard

  - RegWrite signal of WB Control field
    - EX/MEM.RegWrite, MEM/WB.RegWrite
  - EX/MEM.RegisterRd <> $0
  - MEM/WB.RegisterRd <> $0

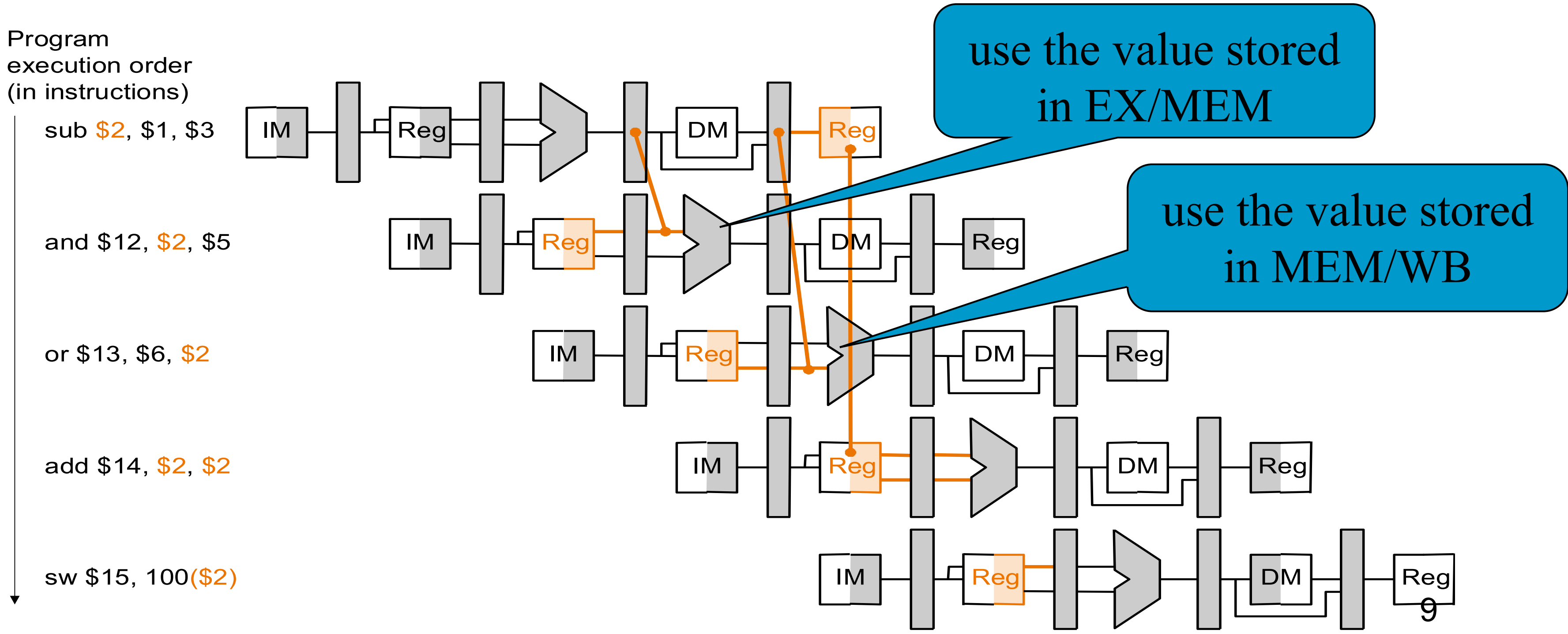
How to forward data?

Resolving Hazards by Forwarding

- Use the value in pipeline registers rather than waiting for the WB stage to write the register file.
  - EX/MEM.Aluout
  - MEM/WB.Aluout

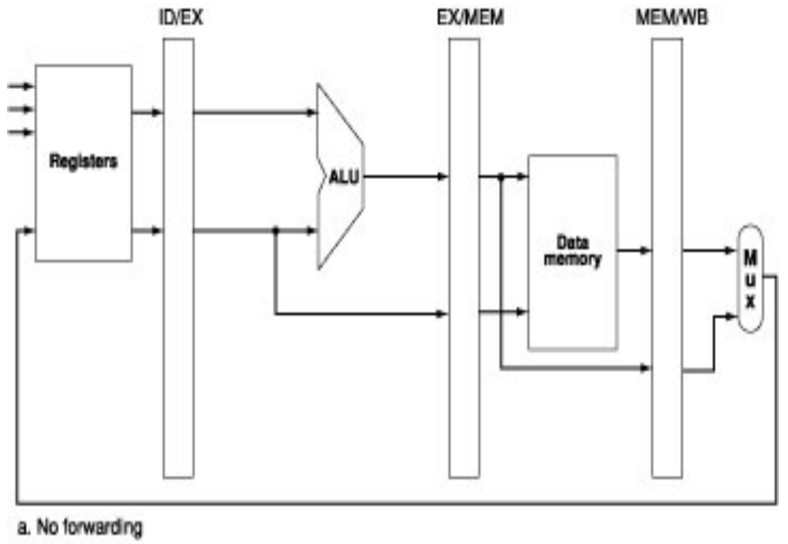
Time (in clock cycles) →

	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

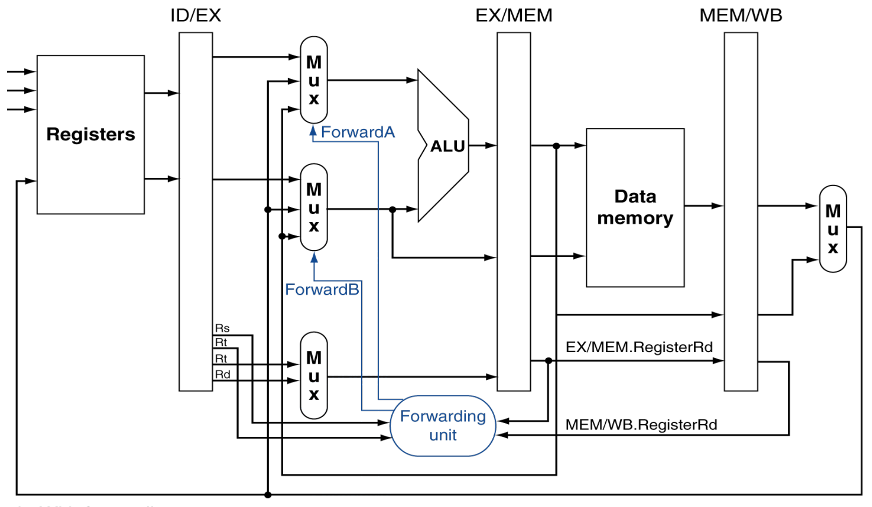

Program execution order (in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
IM
Reg
DM
use the value stored in EX/MEM
use the value stored in MEM/WB

Forwarding Logic

- Forwarding: input to ALU from any pipe reg.
  - Add multiplexors to ALU input
  - Forwarding Control will be in EX



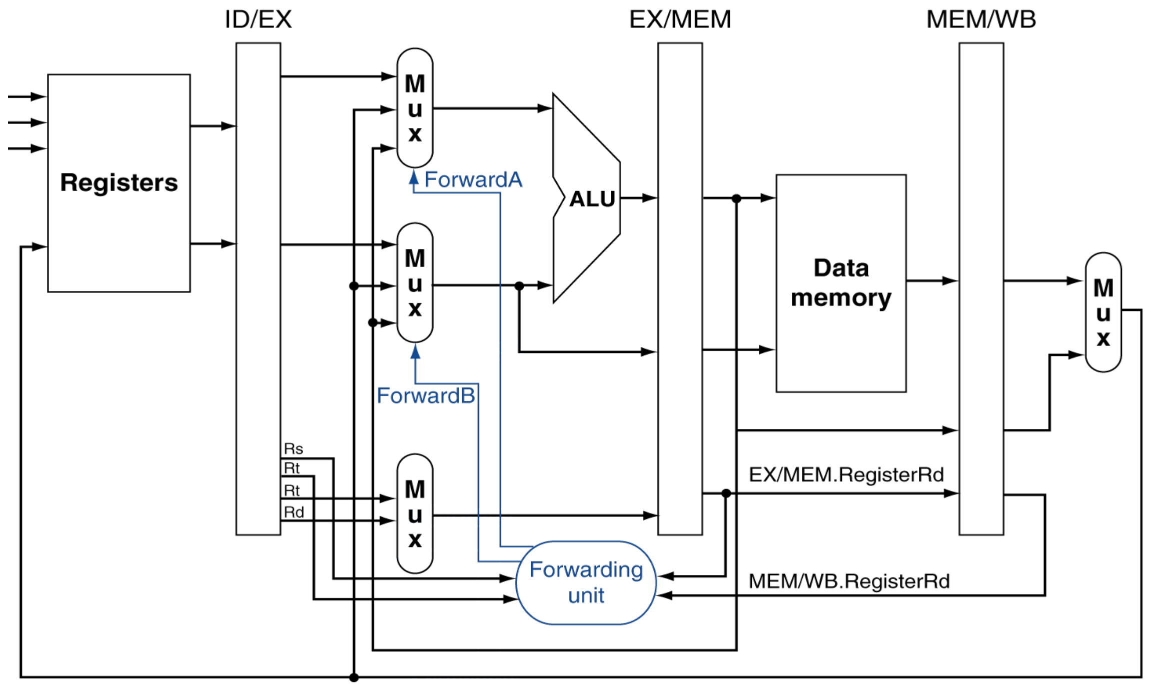
a. No forwarding



b. With forwarding

Forwarding Control

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.



b. With forwarding

Forwarding Control

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.

1. EX hazard

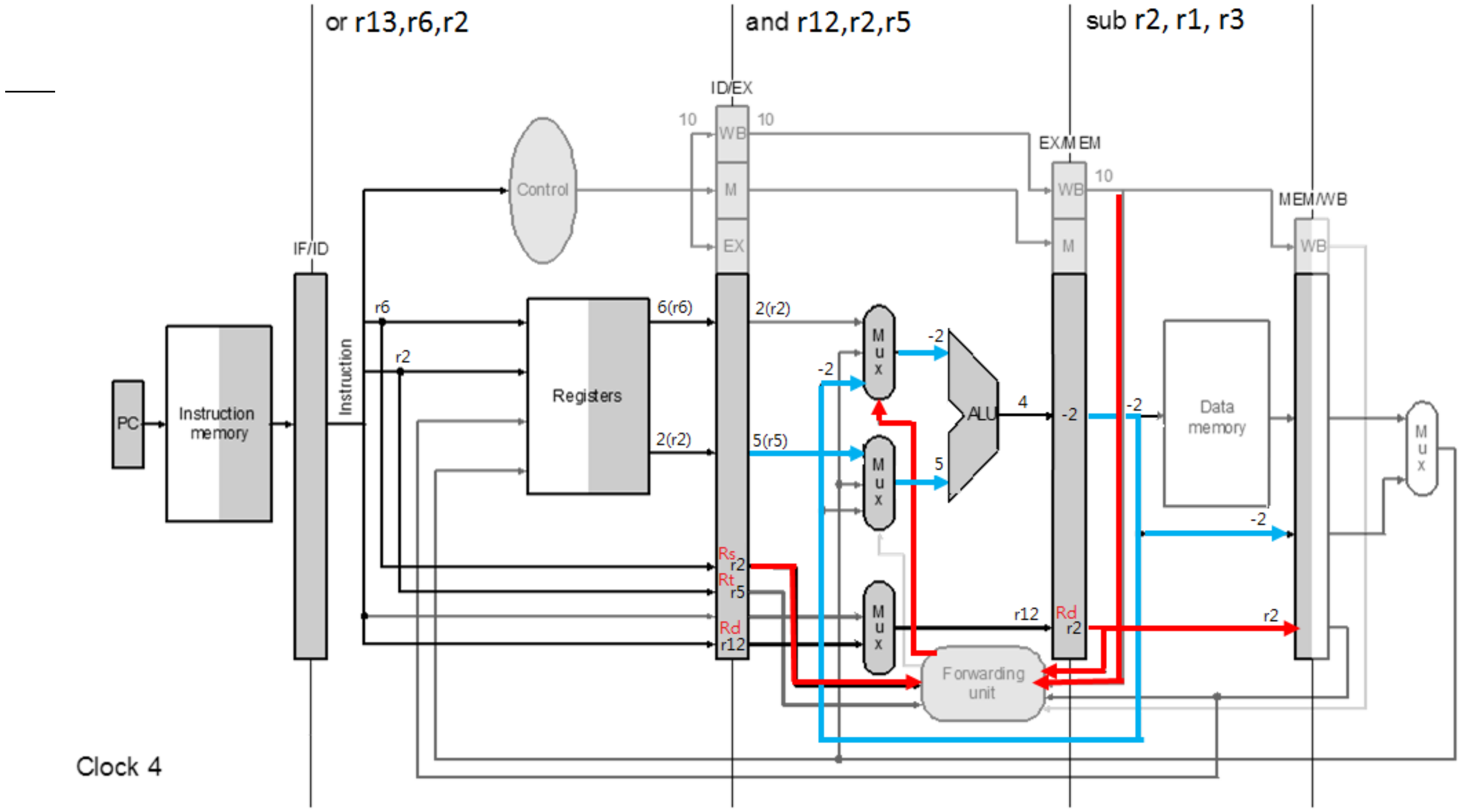
```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd=ID/EX.RegisterRs))
ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd=ID/Ex.RegisterRt))
ForwardB = 10
```

2. MEM hazard

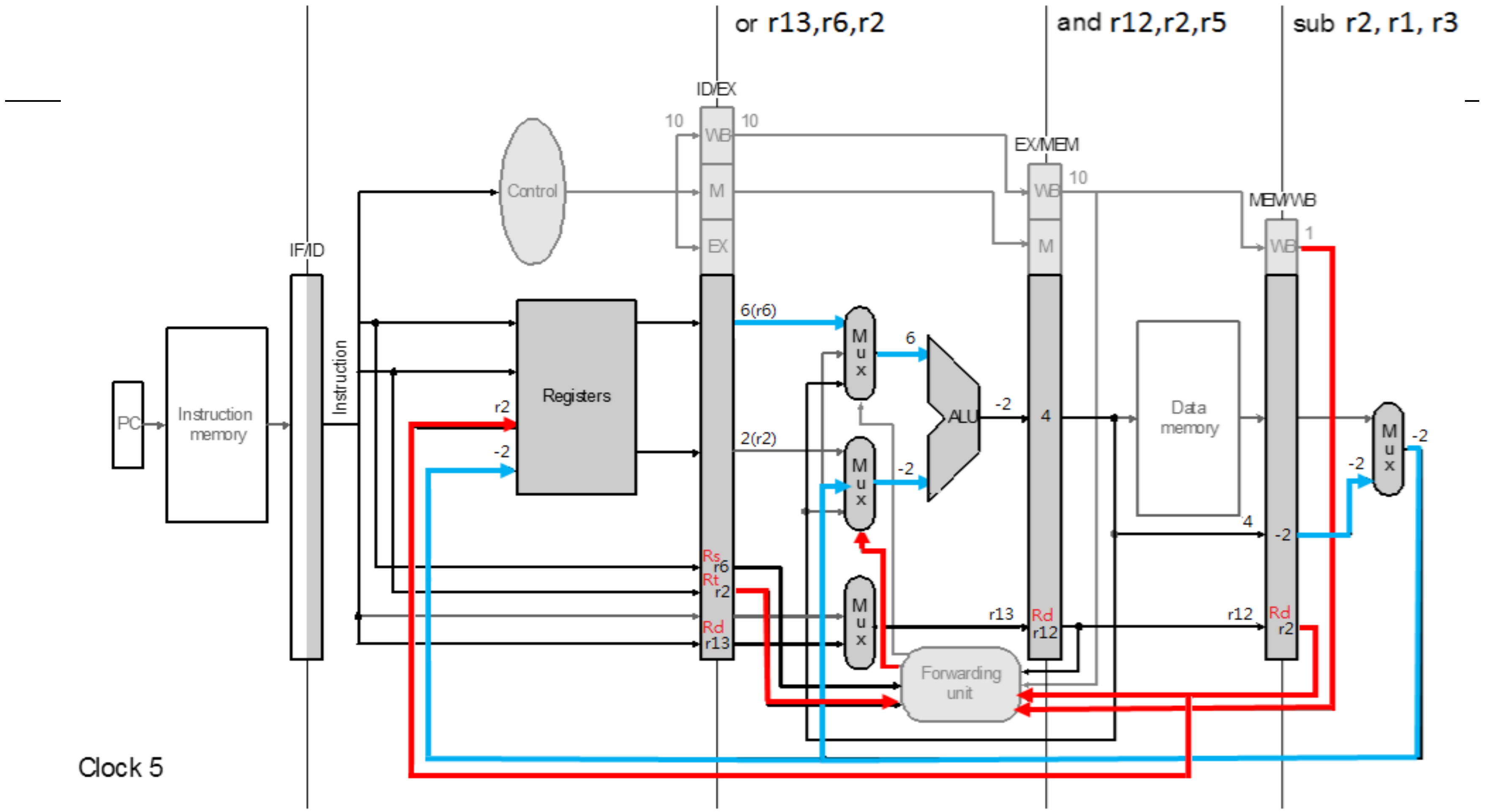
```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd=ID/Ex.RegisterRs))
ForwardA = 01

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



1. EX hazard

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd=ID/EX.RegisterRs))
ForwardA = 10



2. MEM hazard

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

Forwarding Control (cont.)

```
inst1    add $1,$1,$2;        IF ID EX  MEM WB
inst2    add $1,$1,$3;           IF ID  EX  MEM  WB
inst3    add $1,$1,$4;              IF ID  EX  MEM WB
         ......
```

=> Which instruction should forward its results to instruction 3?

if Ex/MEM RegRd = ID/Ex RegRs(Rt)

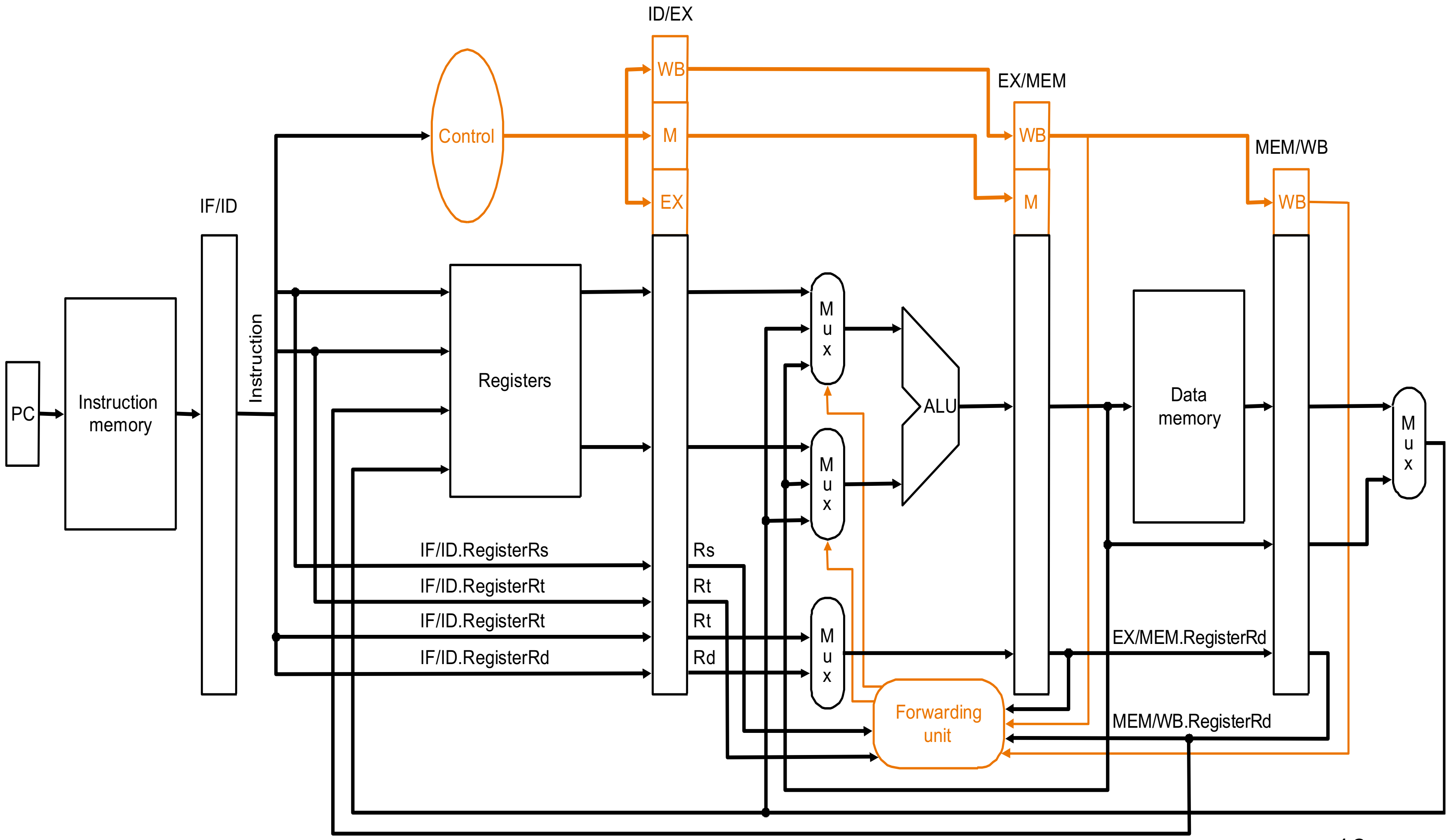
MEM hazard condition becomes

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (Ex/MEM.RegisterRd ≠ ID/Ex.RegisterRs)
and (MEM/WB.RegRd=ID/Ex.RegisterRs))  ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegRd ≠ 0)
and (Ex/MEM.RegisterRd ≠ ID/Ex.RegisterRt)
and (MEM/WB.RegRd=ID/Ex.RegisterRt))  ForwardB = 01
```

會相衝

Datapath with Forwarding

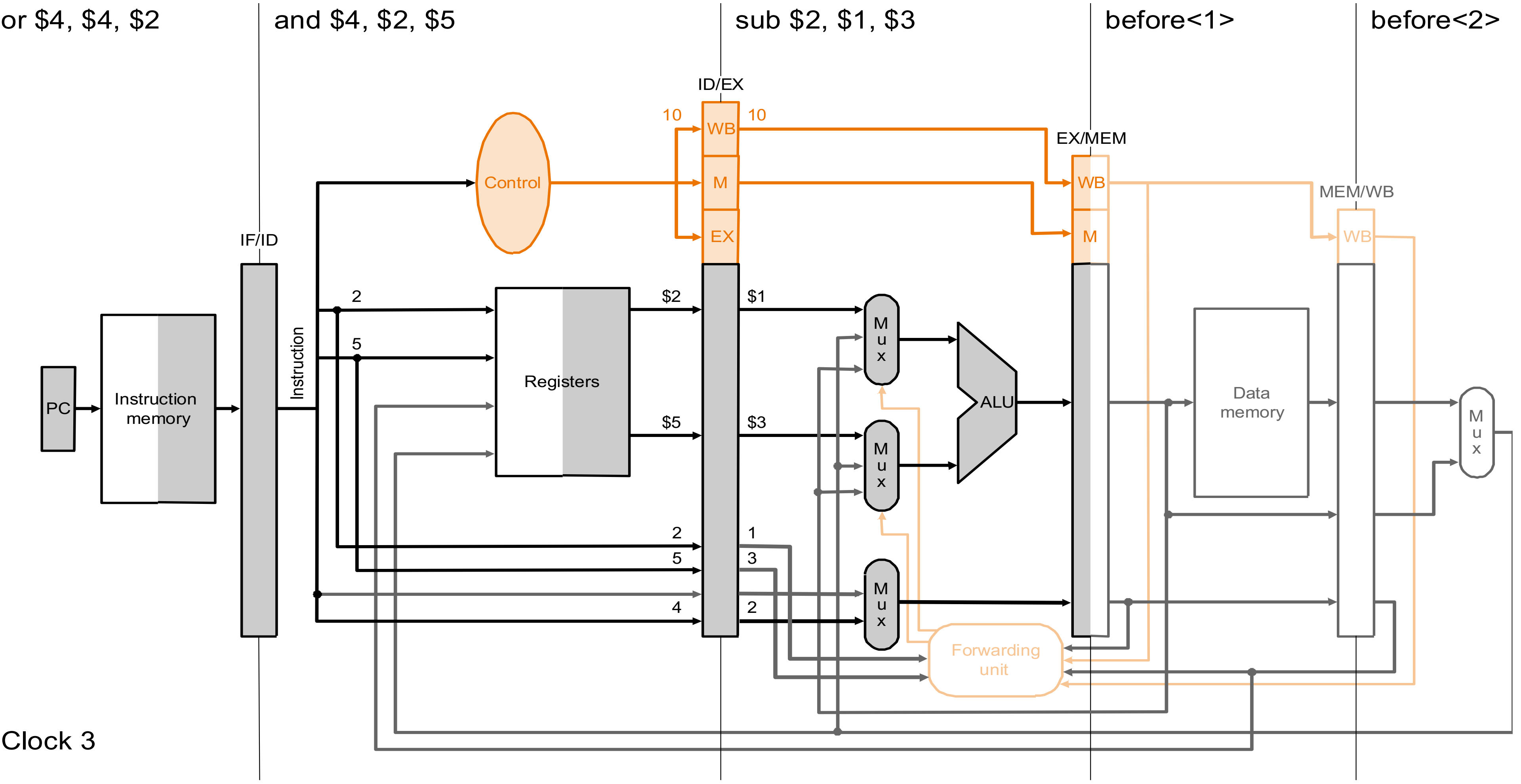


Example

- Show how forwarding works with this instruction sequence (with dependencies highlighted):

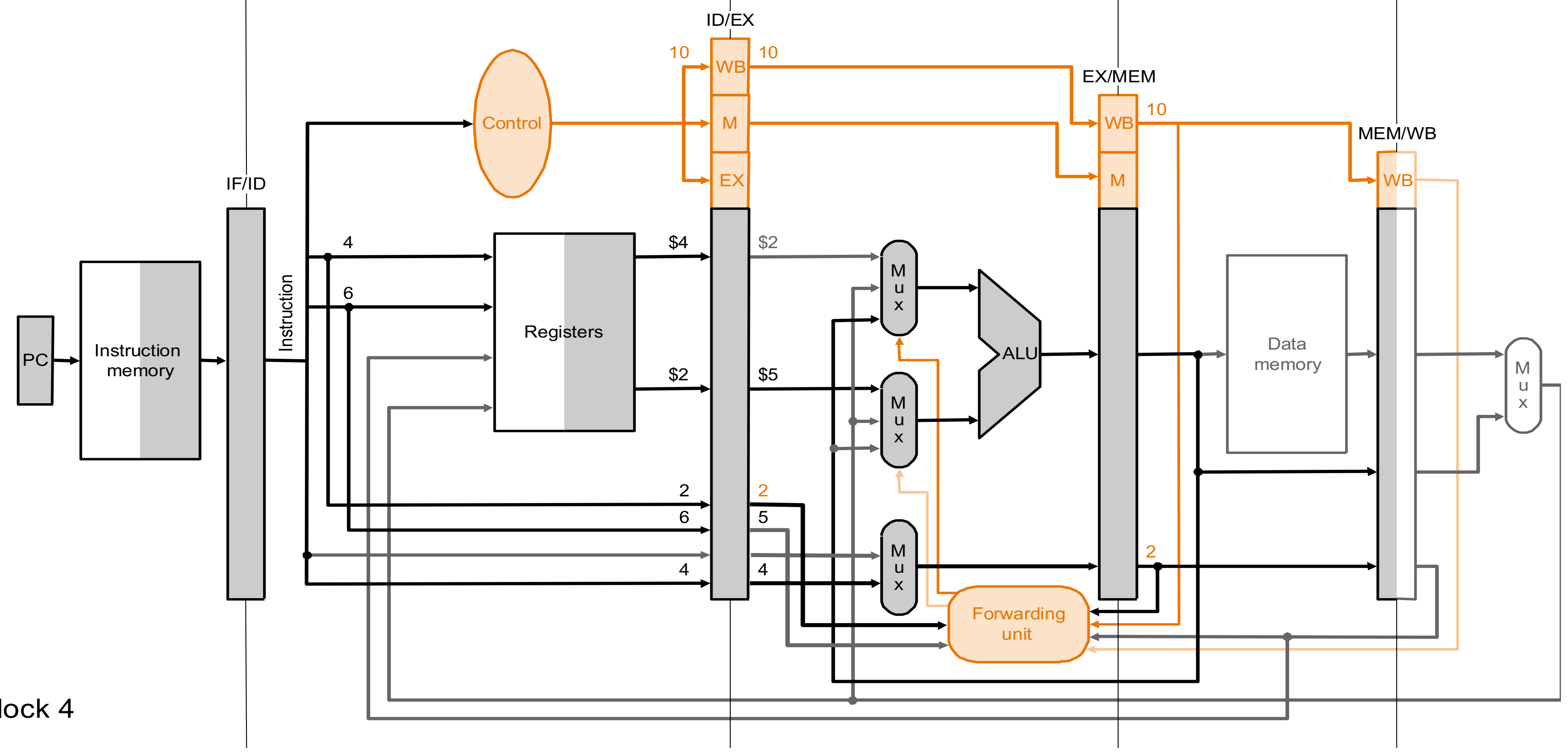
```
sub  $2, $1, $3
and  $4,  $2, $5
or   $4,  $4,  $2
add  $9,  $4, $2
```

EX/MEM.Rd
= ID/EX.Rs

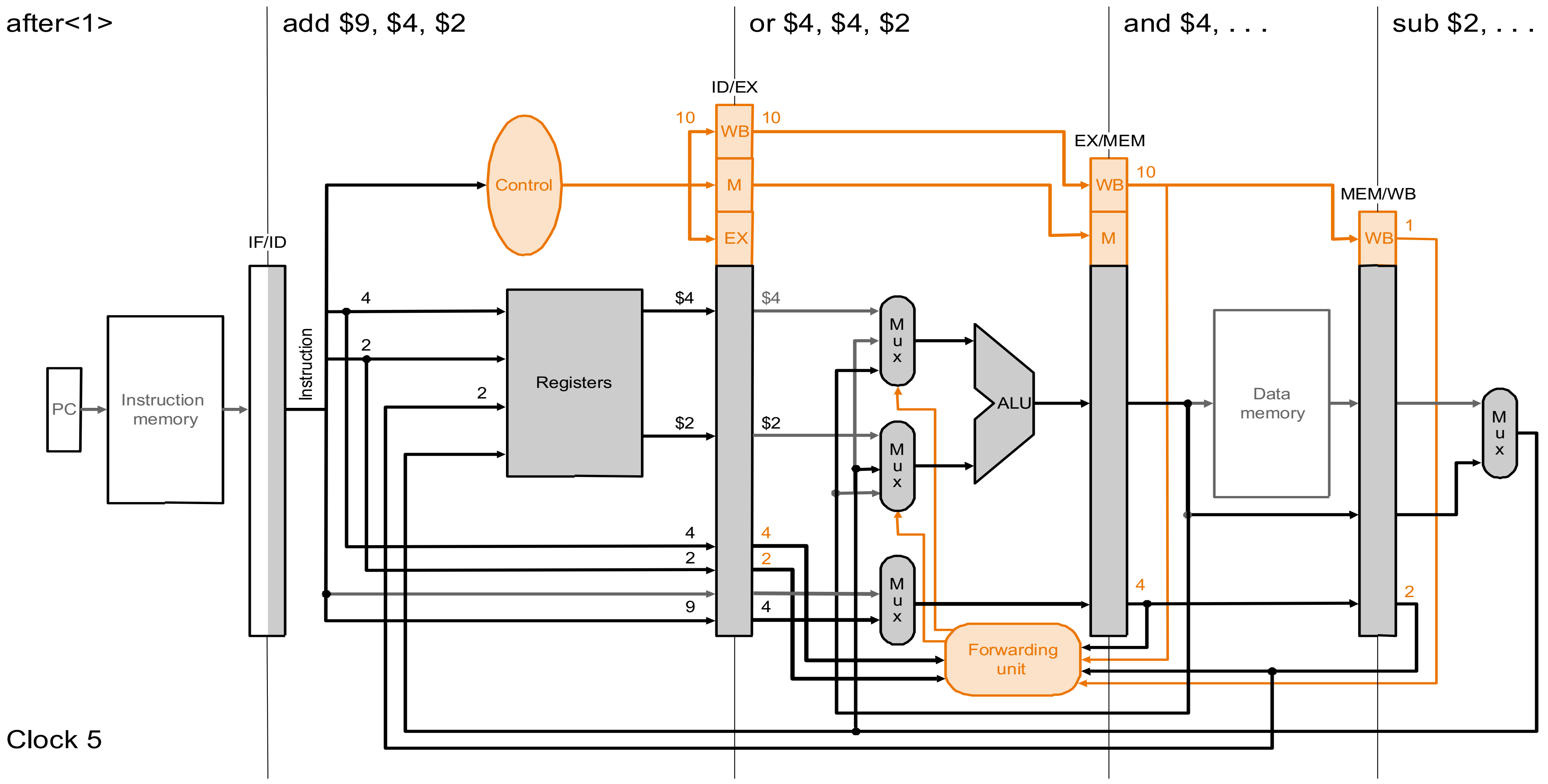


or $4, $4, $2
and $4, $2, $5
sub $2, $1, $3
before<1>
before<2>
ID/EX
10
WB
10
EX/MEM
Control
M
WB
MEM/WB
EX
M
WB
IF/ID
2
$2
$1
M
u
x
5
Instruction
PC
Instruction memory
Registers
ALU
Data memory
M
u
x
$5
$3
M
u
x
2
1
5
3
M
u
x
4
2
Forwarding unit
Clock 3

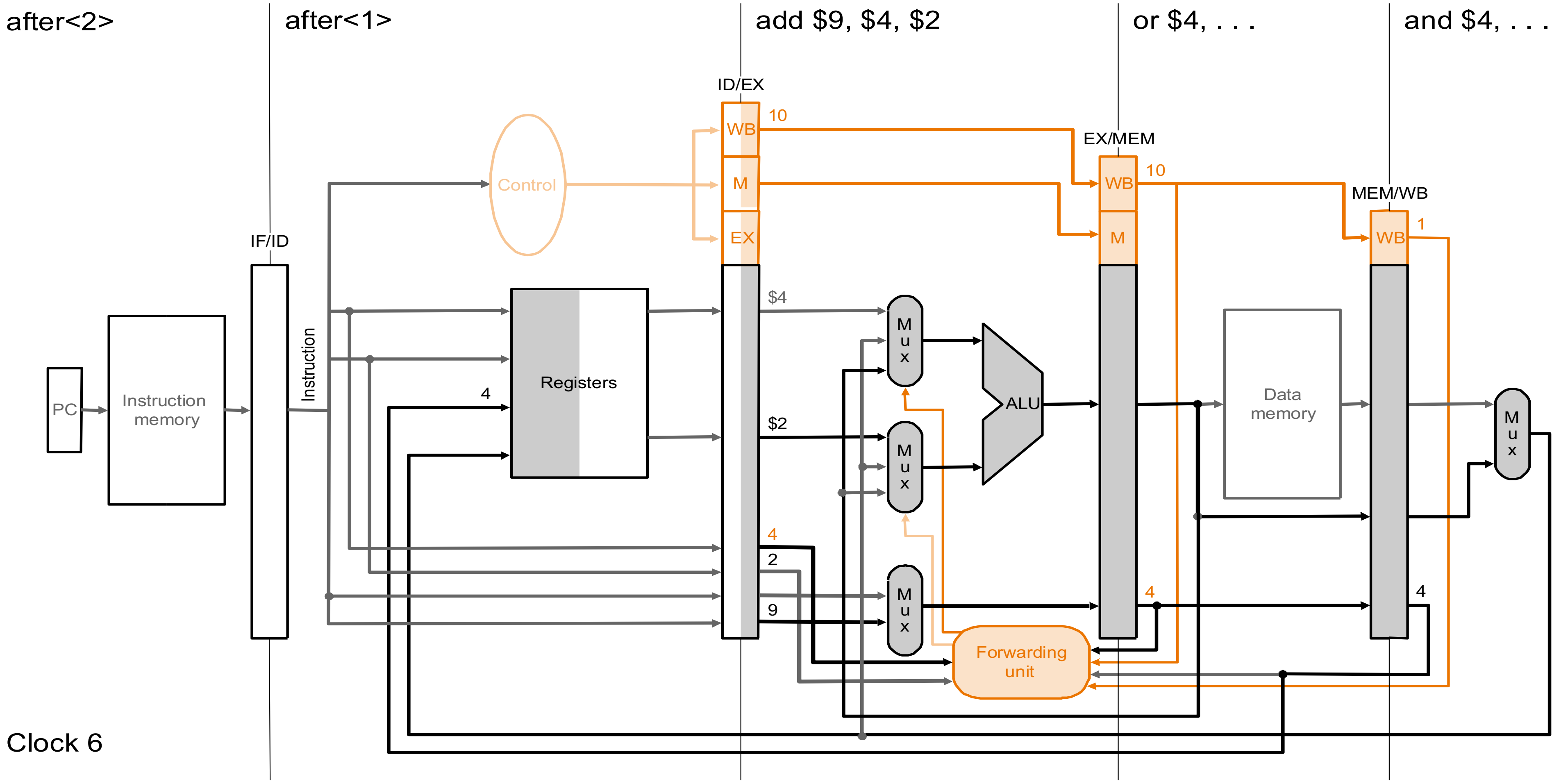
add $9, $4, $2 | or $4, $4, $2 | and $4, $2, $5 | sub $2, . . . | before<1>



Clock 4

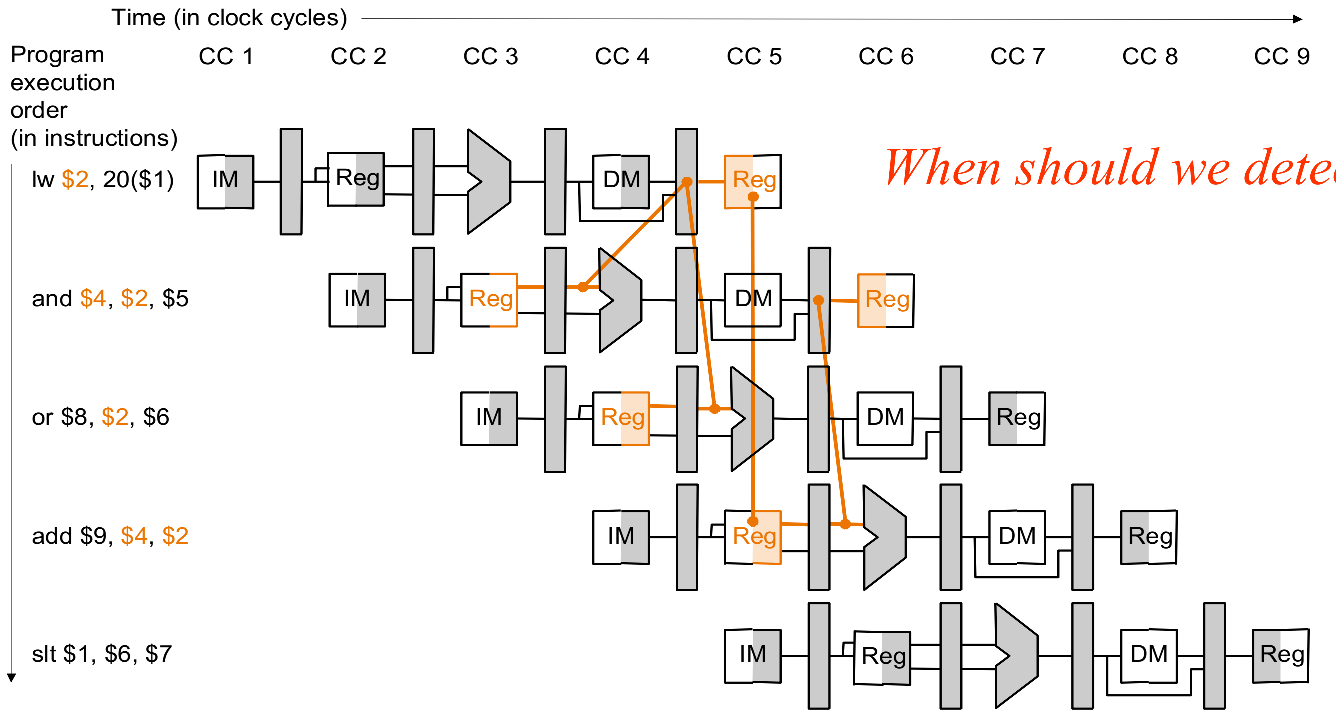


after<1>
add $9, $4, $2
or $4, $4, $2
and $4, . . .
sub $2, . . .
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
10
10
10
1
PC
Instruction memory
Instruction
Registers
4
2
2
$4
$2
$4
$2
4
2
9
4
2
4
M u x
ALU
Data memory
Forwarding unit
4
2
Clock 5


after<2>
after<1>
add $9, $4, $2
or $4, . . .
and $4, . . .
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
10
10
1
PC
Instruction memory
Instruction
Registers
4
$4
$2
4
2
9
Mux
ALU
Data memory
Forwarding unit
4
4
Clock 6

Can't always forward

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.
  -



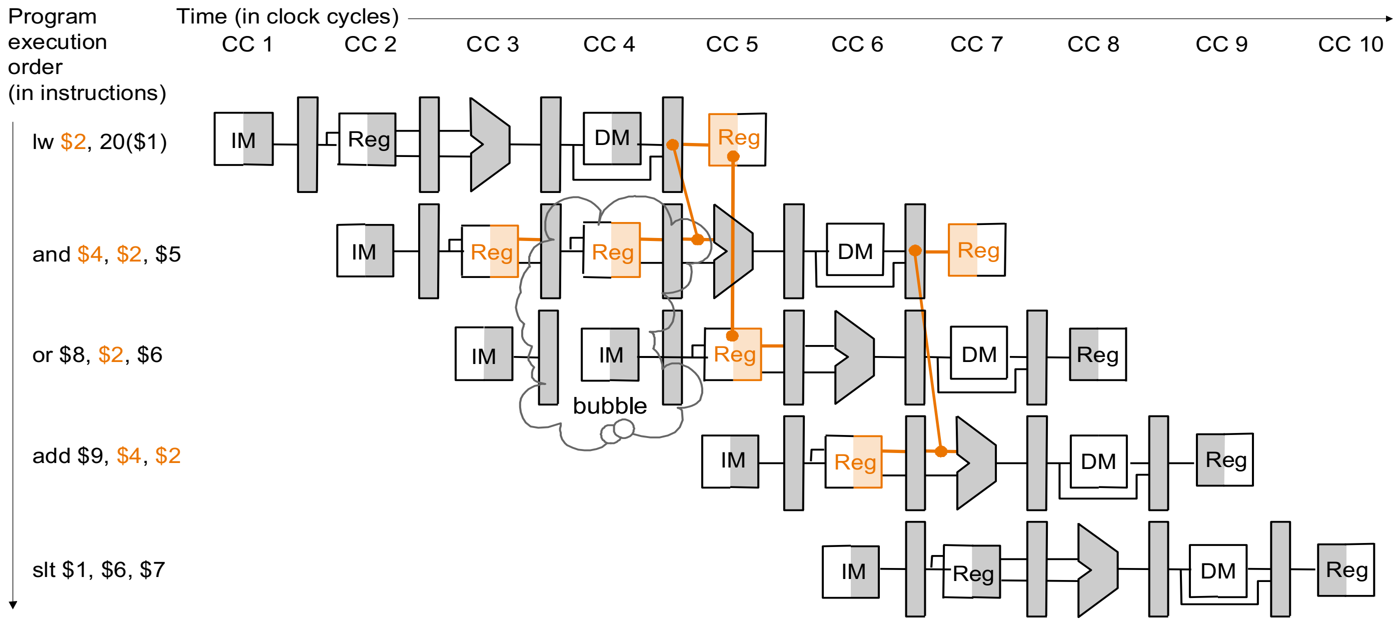
When should we detect this hazard?

If (ID/EX.MemRead and
((ID/EX.RegisterRt = IF/ID.RegisterRs) or
(ID/EX.RegisterRt = IF/ID.RegisterRt)))

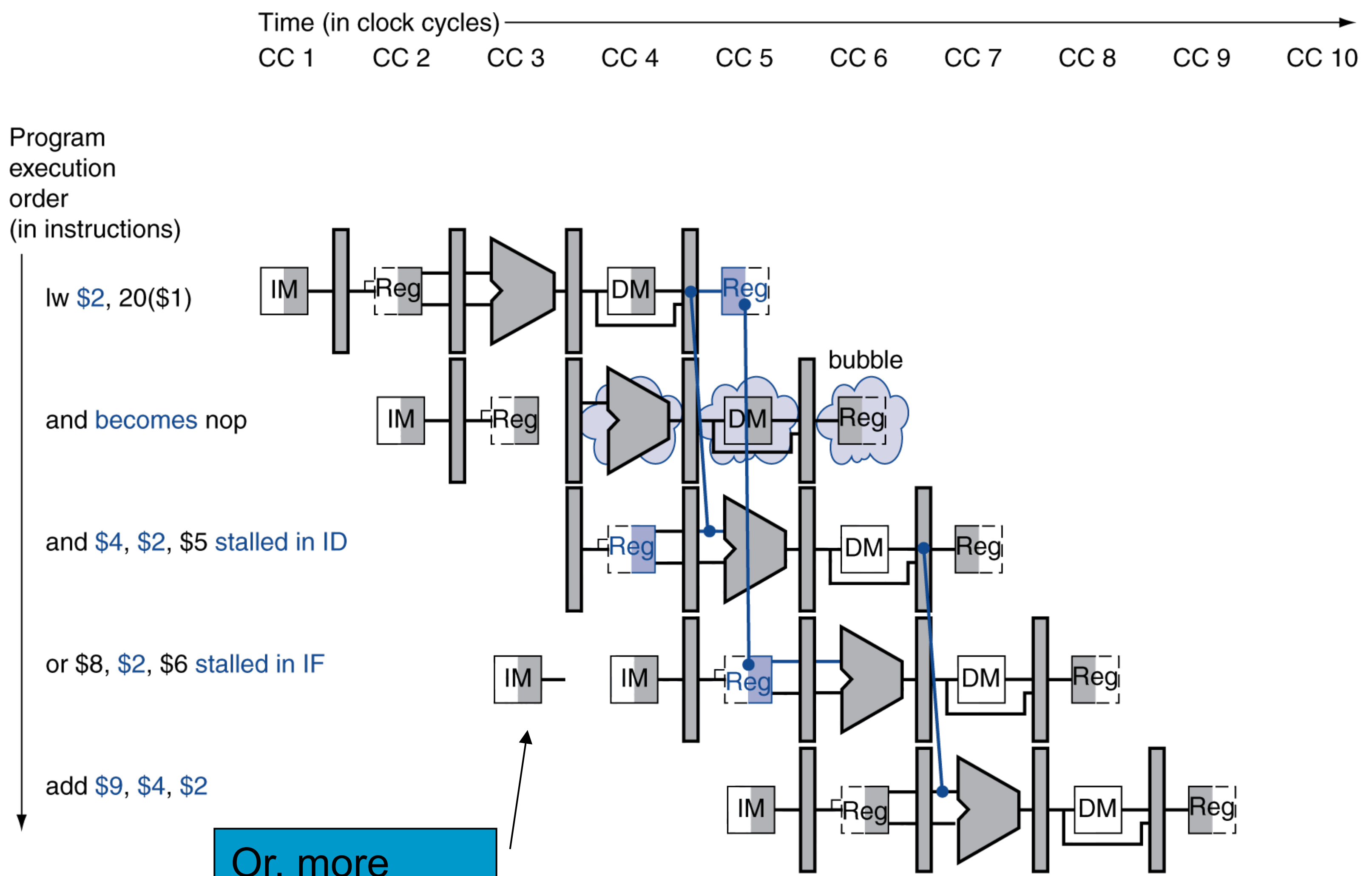
stall the pipeline

Hazard Detection and Stall

- If (ID/EX.MemRead and
  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
  (ID/EX.RegisterRt = IF/ID.REgisterRt)))
  stall the pipeline
- Stall the pipeline
  - Preventing instructions in the IF and ID stages from making progress
    - Preserve the PC and IF/ID pipeline registers
  - We need to do nothing in EX at CC4, MEM at CC5 , WB at CC6
    - Deasserting all nine control signals in the EX, MEM and WB stage



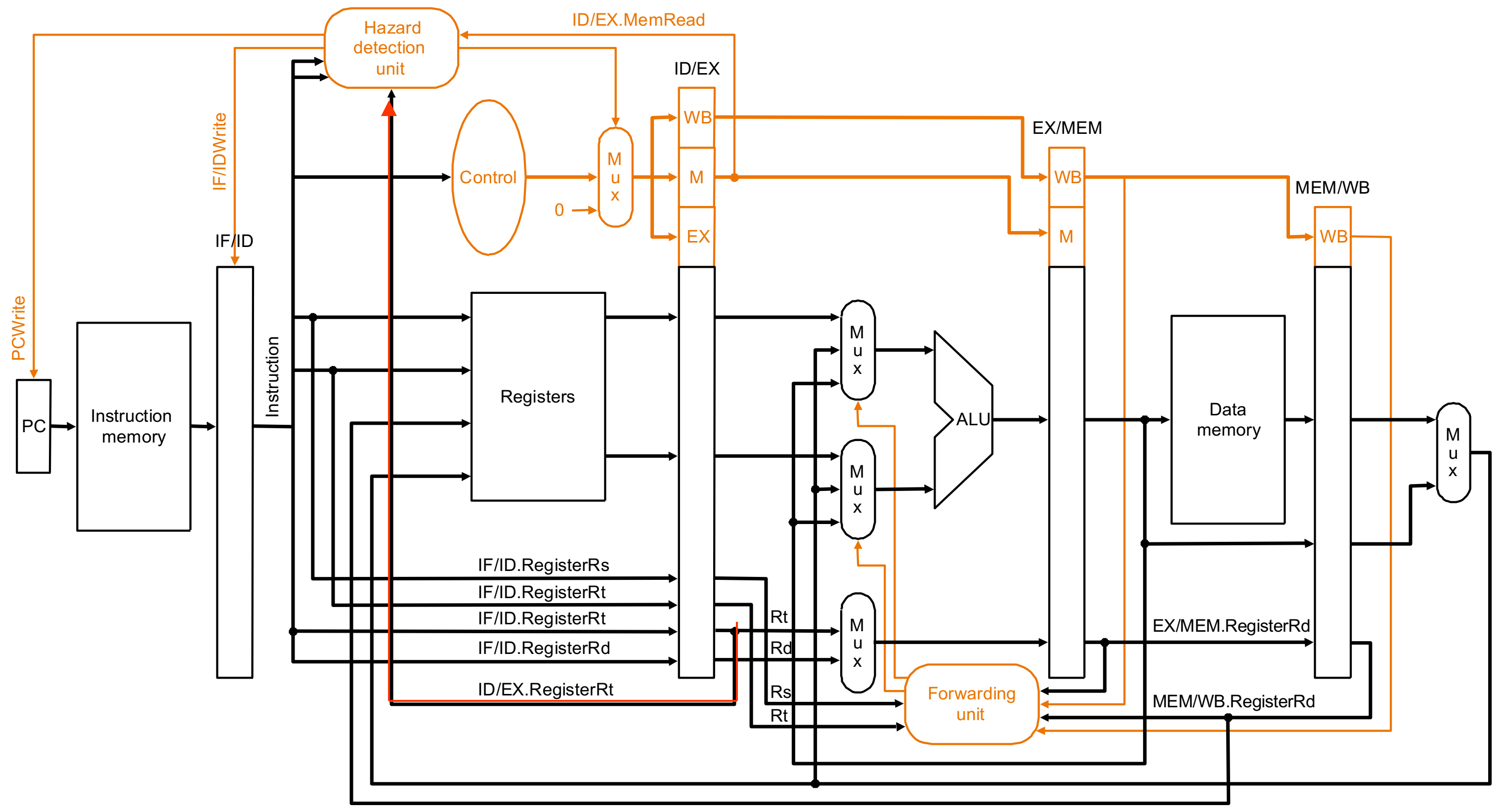
Stall/Bubble in the Pipeline



Or, more accurately…

Hazard Detection Unit

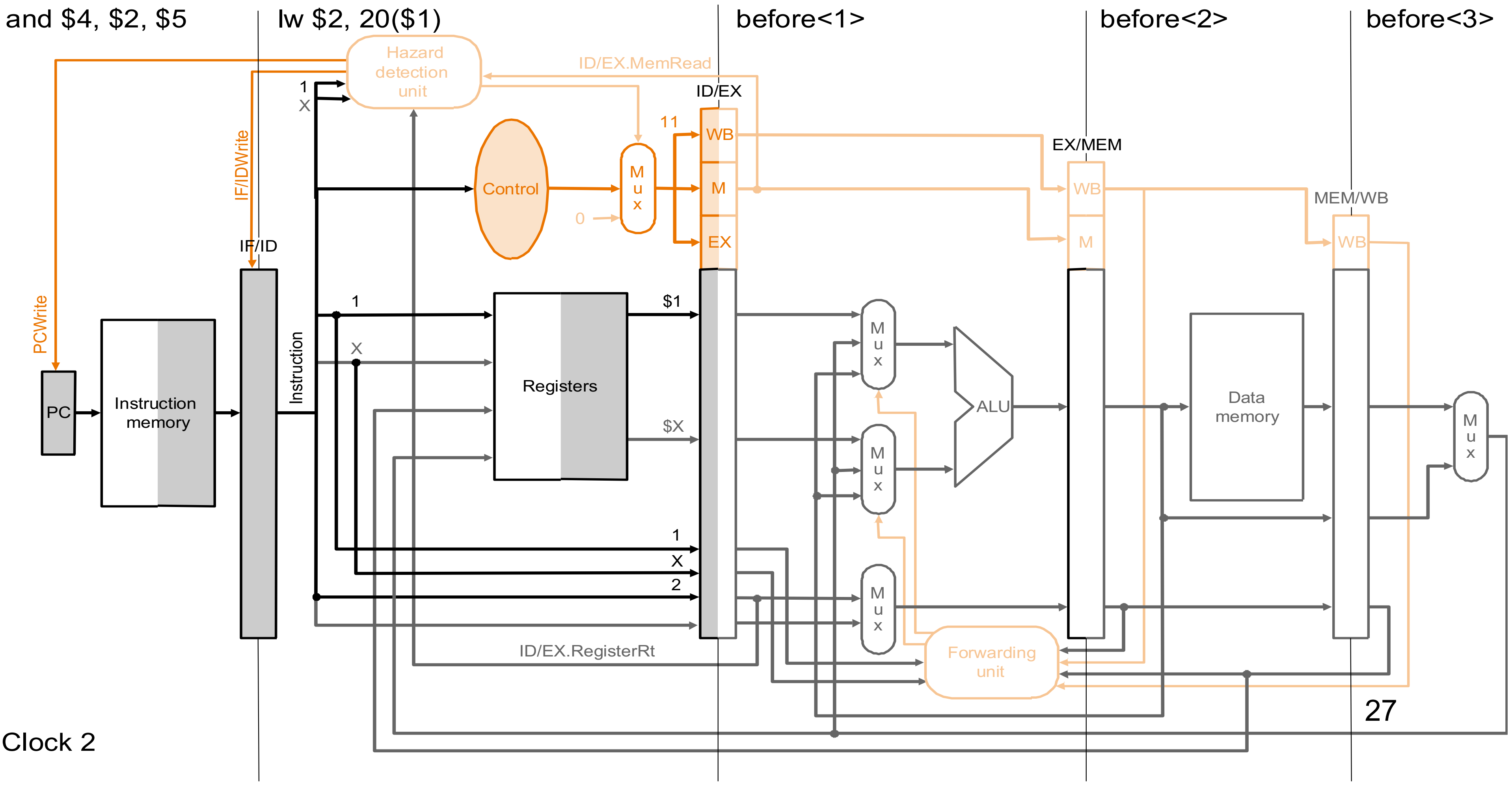
- Stall by letting an instruction that won't write anything go forward



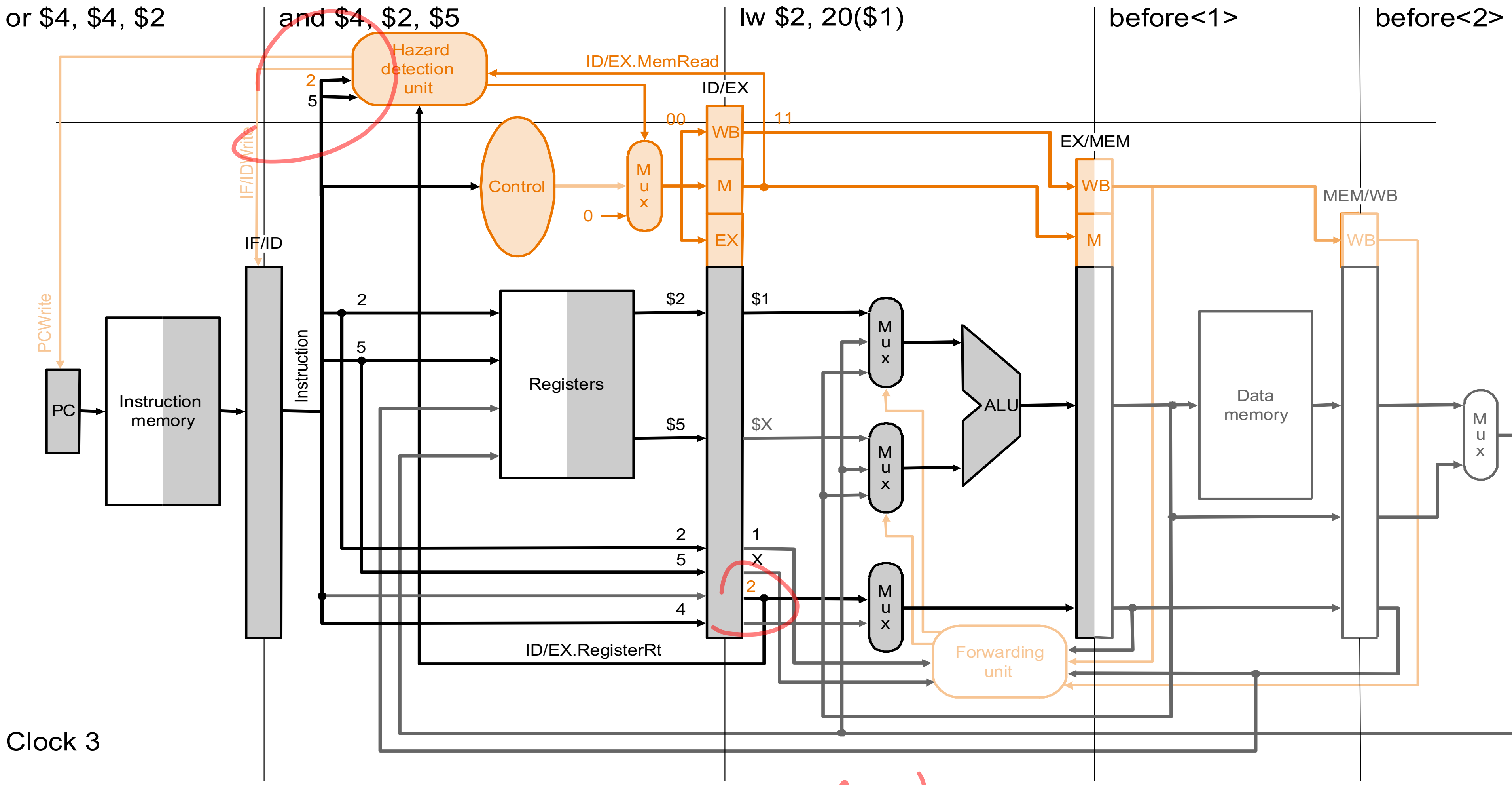
Example

- Show how hazard detection unit works with this instruction sequence (with dependencies highlighted):

```
lw   $2, 20($1)    } load-use data hazard
and  $4, $2, $5
or   $4, $4, $2    } Forwarding
add  $9, $4, $2
```

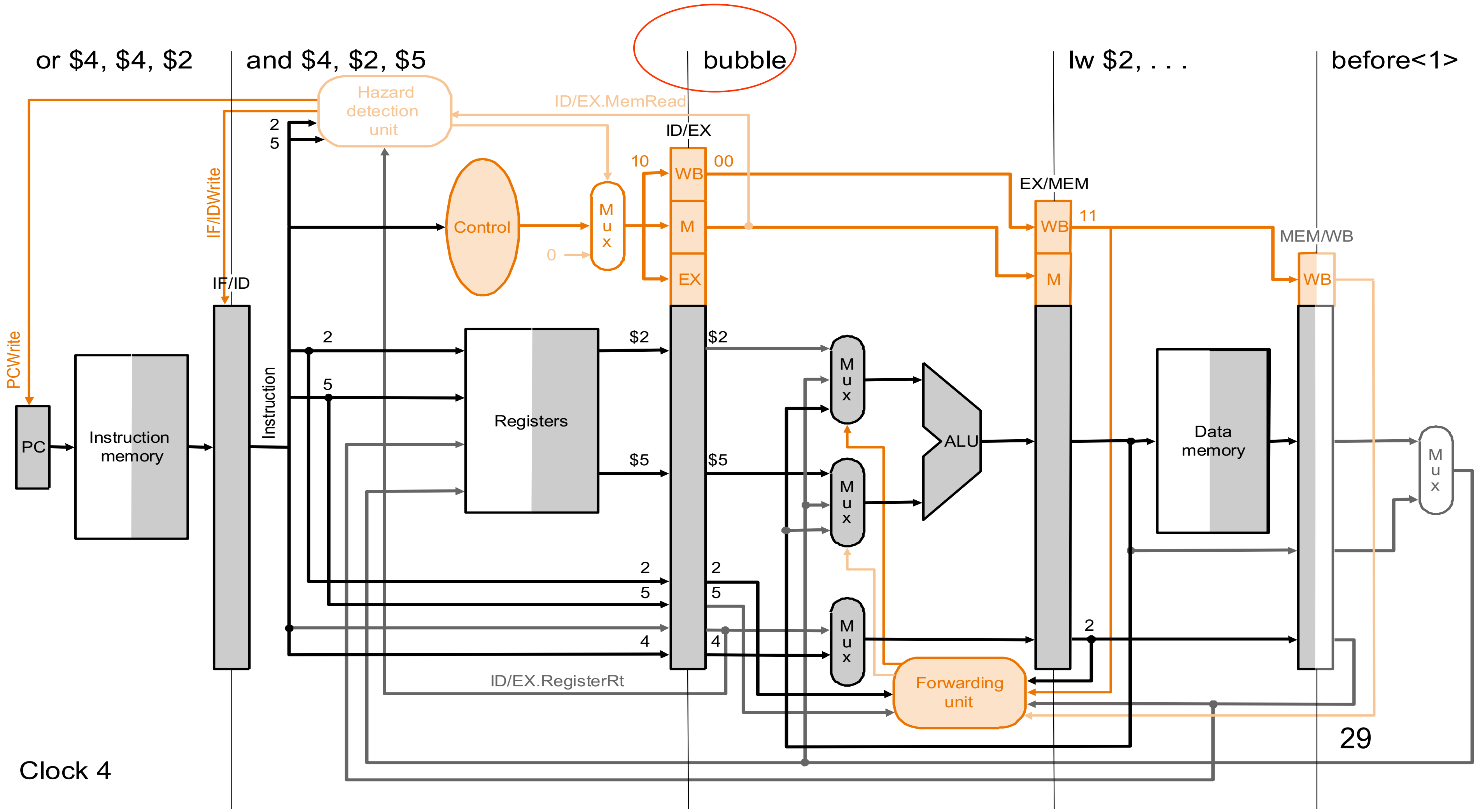


and $4, $2, $5
lw $2, 20($1)
before<1>
before<2>
before<3>
Hazard detection unit
ID/EX.MemRead
ID/EX
1
X
11
WB
M
EX
Control
M u x
0
IF/IDWrite
PCWrite
IF/ID
EX/MEM
WB
M
MEM/WB
WB
PC
Instruction memory
Instruction
Registers
1
X
$1
$X
1
X
2
M u x
M u x
M u x
ALU
Data memory
M u x
ID/EX.RegisterRt
Forwarding unit
Clock 2



IF/ID. Rs(Rd) = ID/EX

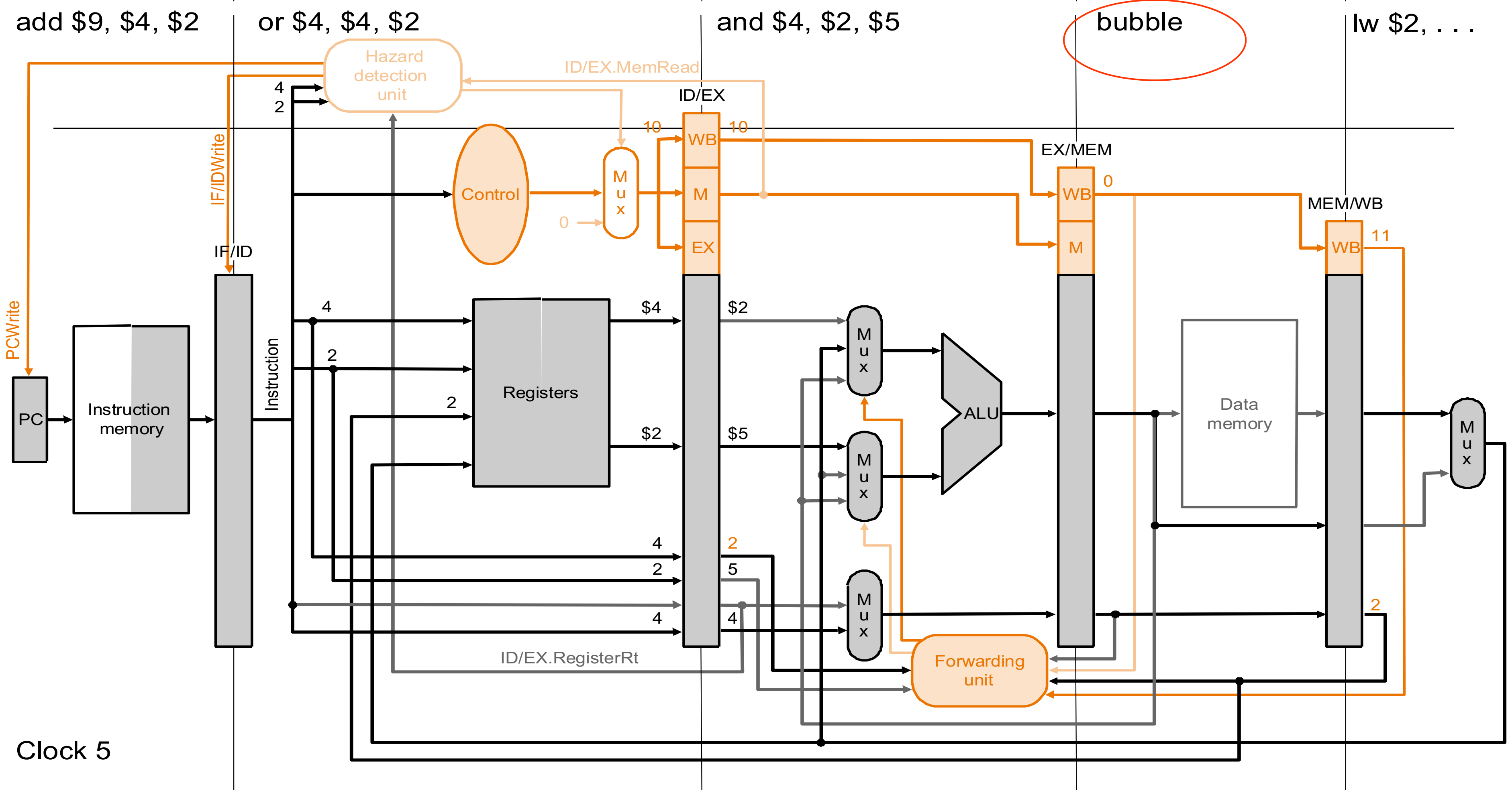
$2 $5 Rt($2)



or $4, $4, $2
and $4, $2, $5
bubble
lw $2, . . .
before<1>
Hazard detection unit
ID/EX.MemRead
PCWrite
IF/IDWrite
2
5
ID/EX
10
00
WB
M
EX
Control
M u x
0
EX/MEM
11
MEM/WB
IF/ID
PC
Instruction memory
Instruction
Registers
$2
$5
ALU
Data memory
4
ID/EX.RegisterRt
Forwarding unit

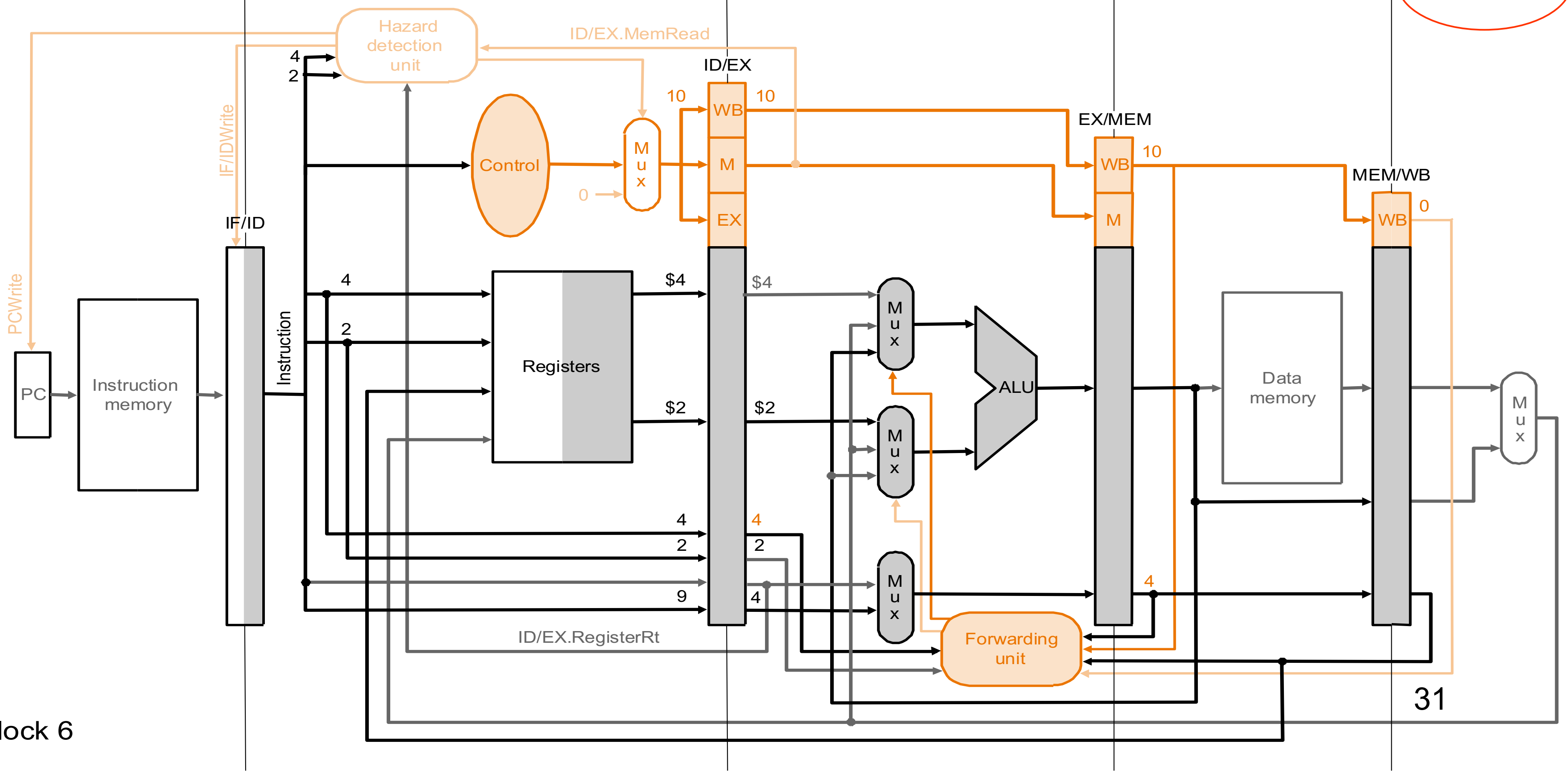
Clock 4

add $9, $4, $2 | or $4, $4, $2 | and $4, $2, $5 | bubble | lw $2, . . .

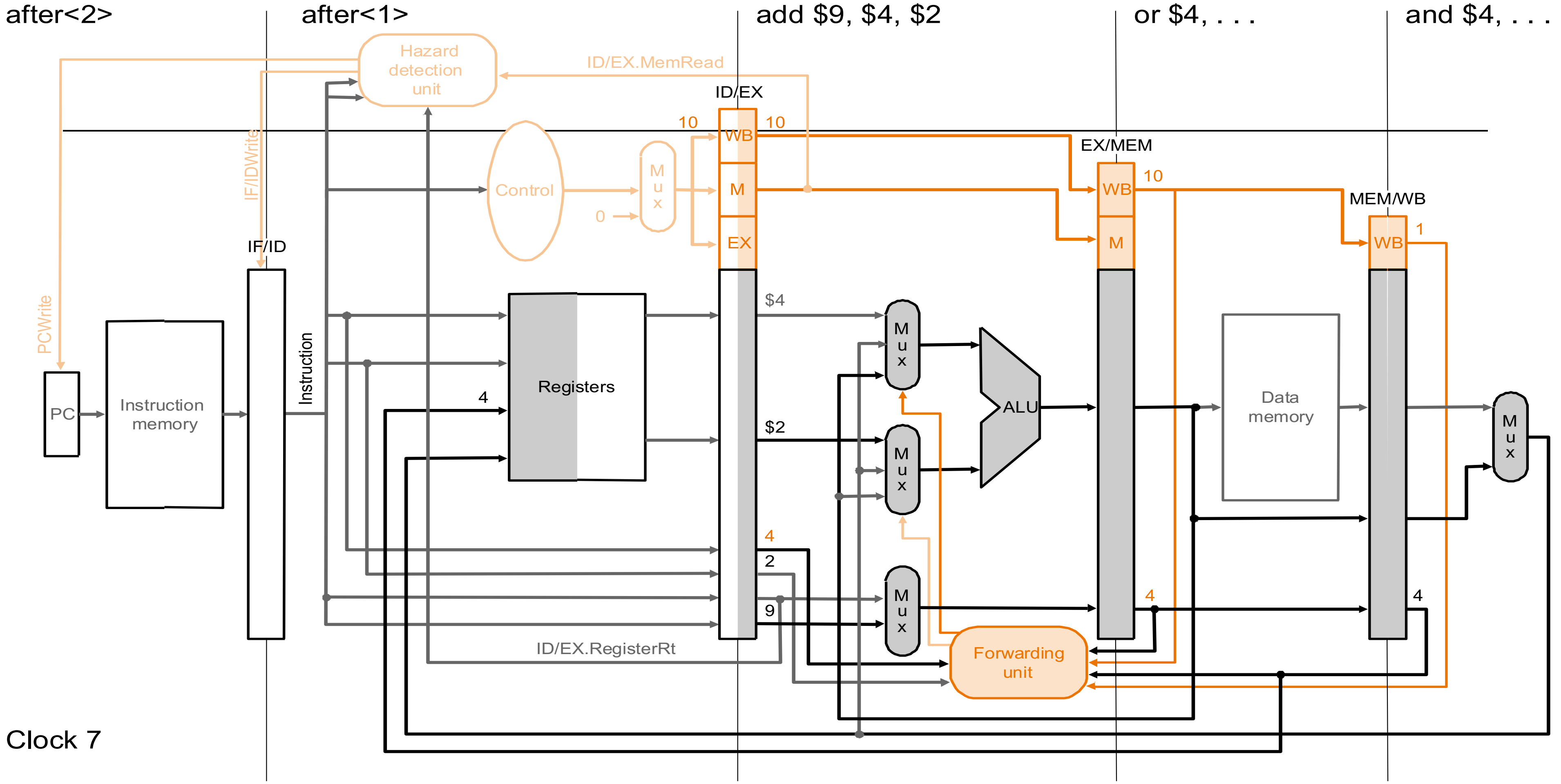


Clock 5

after<1>   add $9, $4, $2   or $4, $4, $2   and $4, . . .   bubble



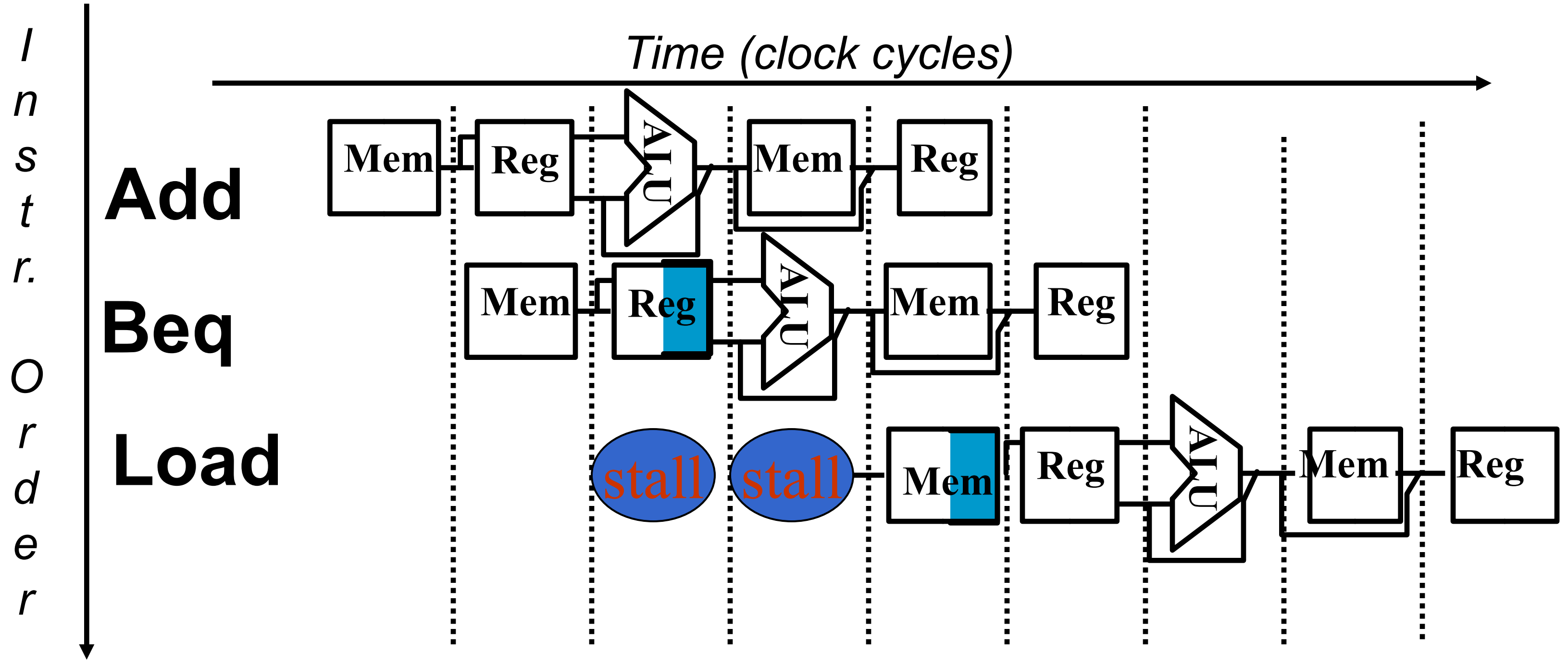
Clock 6



after<2>
after<1>
add $9, $4, $2
or $4, . . .
and $4, . . .
Hazard detection unit
ID/EX.MemRead
ID/EX
10
10
WB
M
EX
EX/MEM
10
WB
M
MEM/WB
1
WB
Control
0
M u x
IF/IDWrite
PCWrite
IF/ID
PC
Instruction memory
Instruction
Registers
4
$4
$2
4
2
9
M u x
M u x
M u x
ALU
Data memory
M u x
4
4
Forwarding unit
ID/EX.RegisterRt
Clock 7

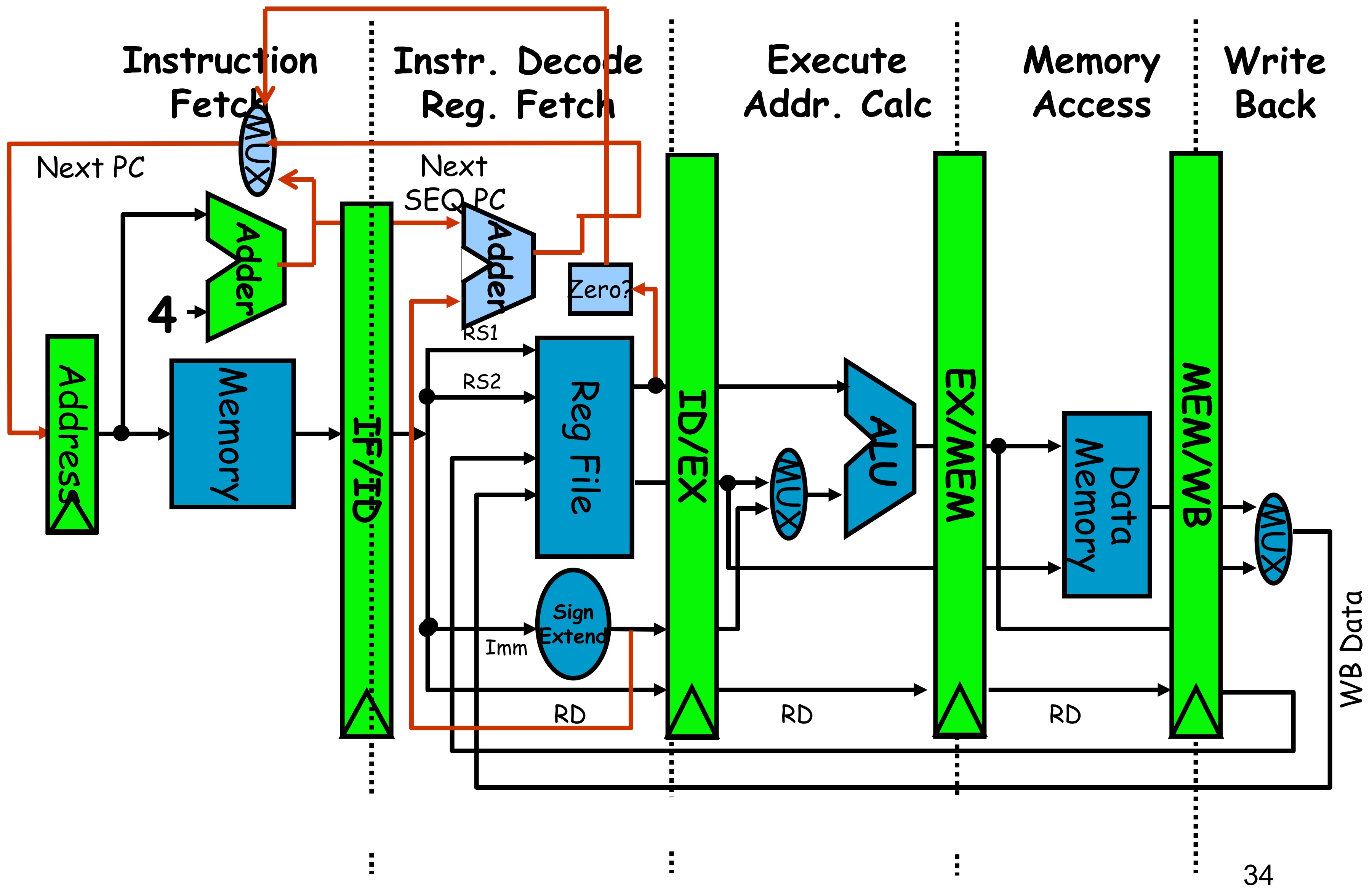
Control Hazard Solutions

- Stall: wait until decision is clear



- Impact: 3 clock cycles per branch instruction => slow

Control Hazard Solution (1): Reducing the Delay of Branches (Example: BEQZ, BNEZ)



Data Hazards for Branches

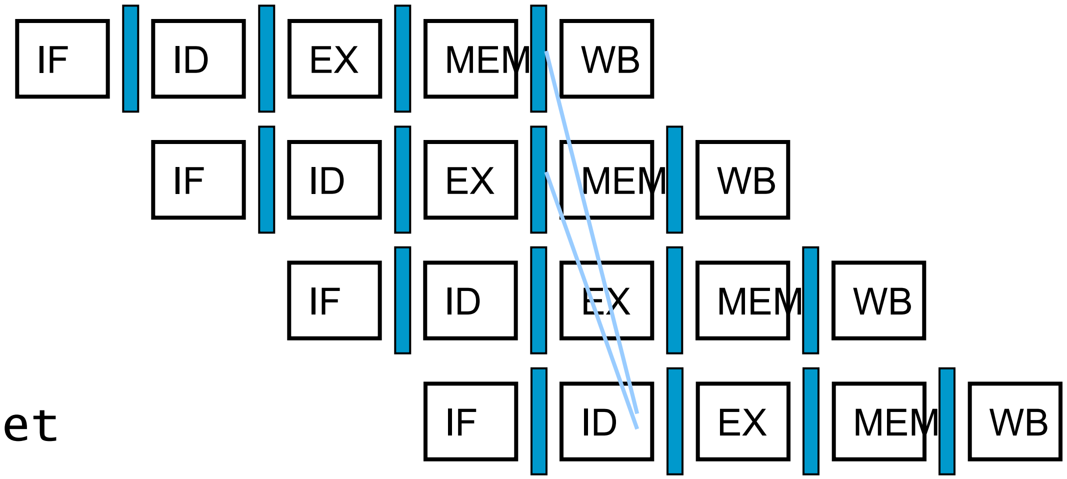
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

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

add $4, $5, $6

…

beq $1, $4, target
```

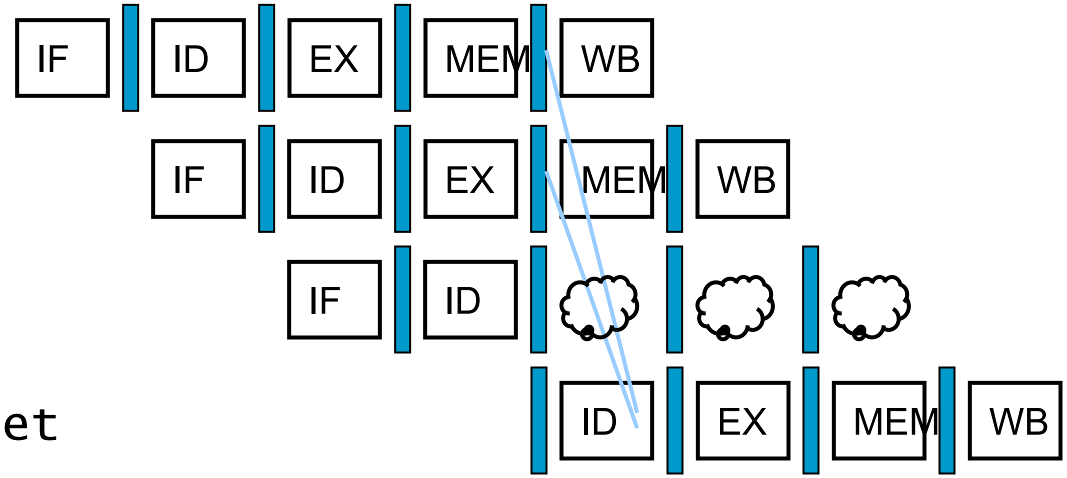


- Can resolve using forwarding

Data Hazards for Branches

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

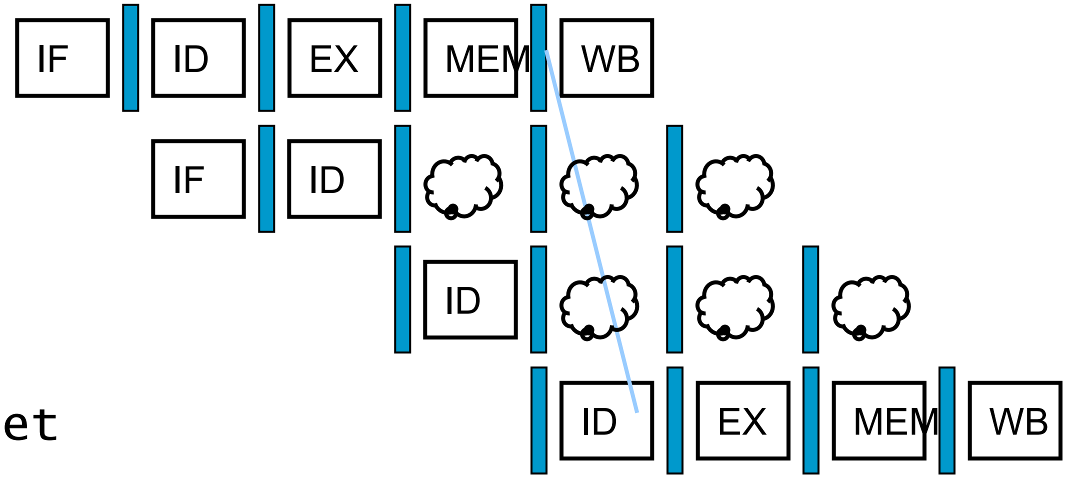
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



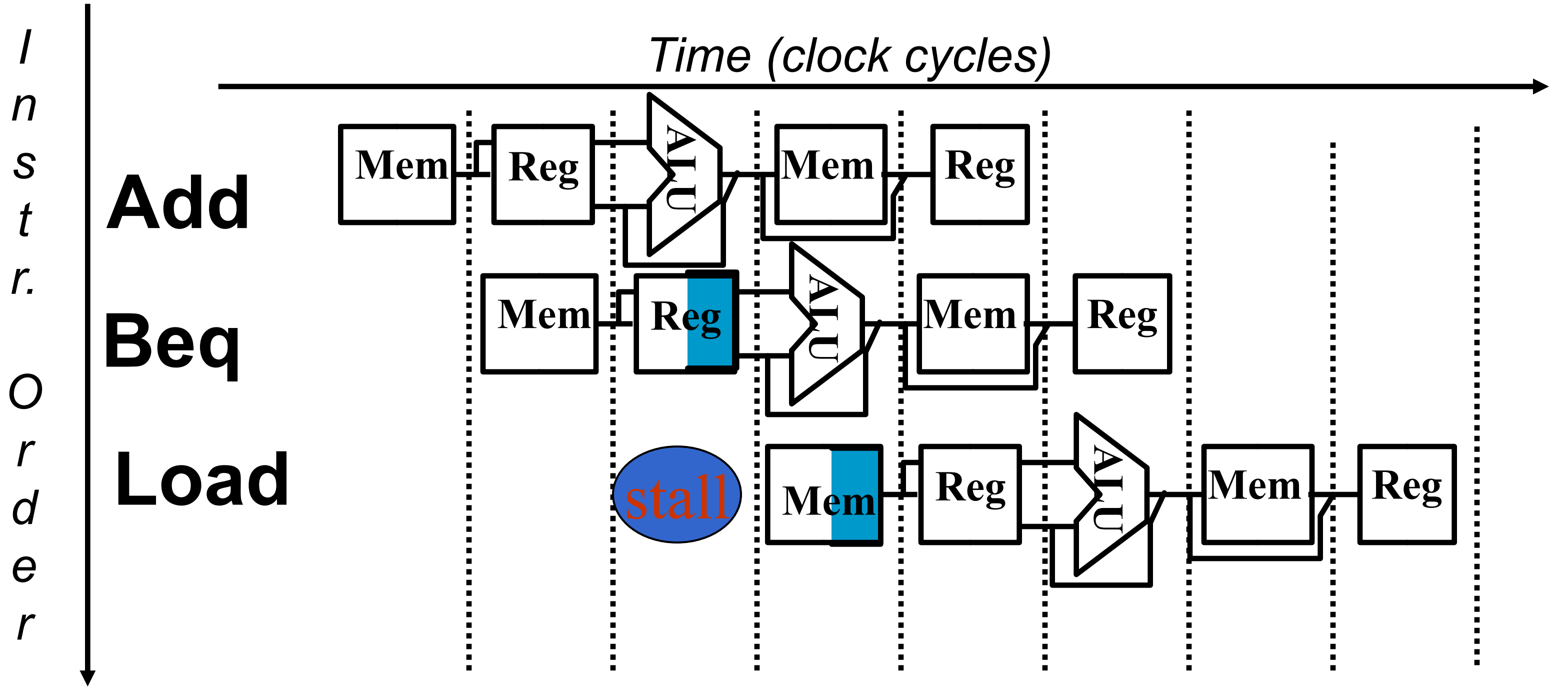
Data Hazards for Branches

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

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```



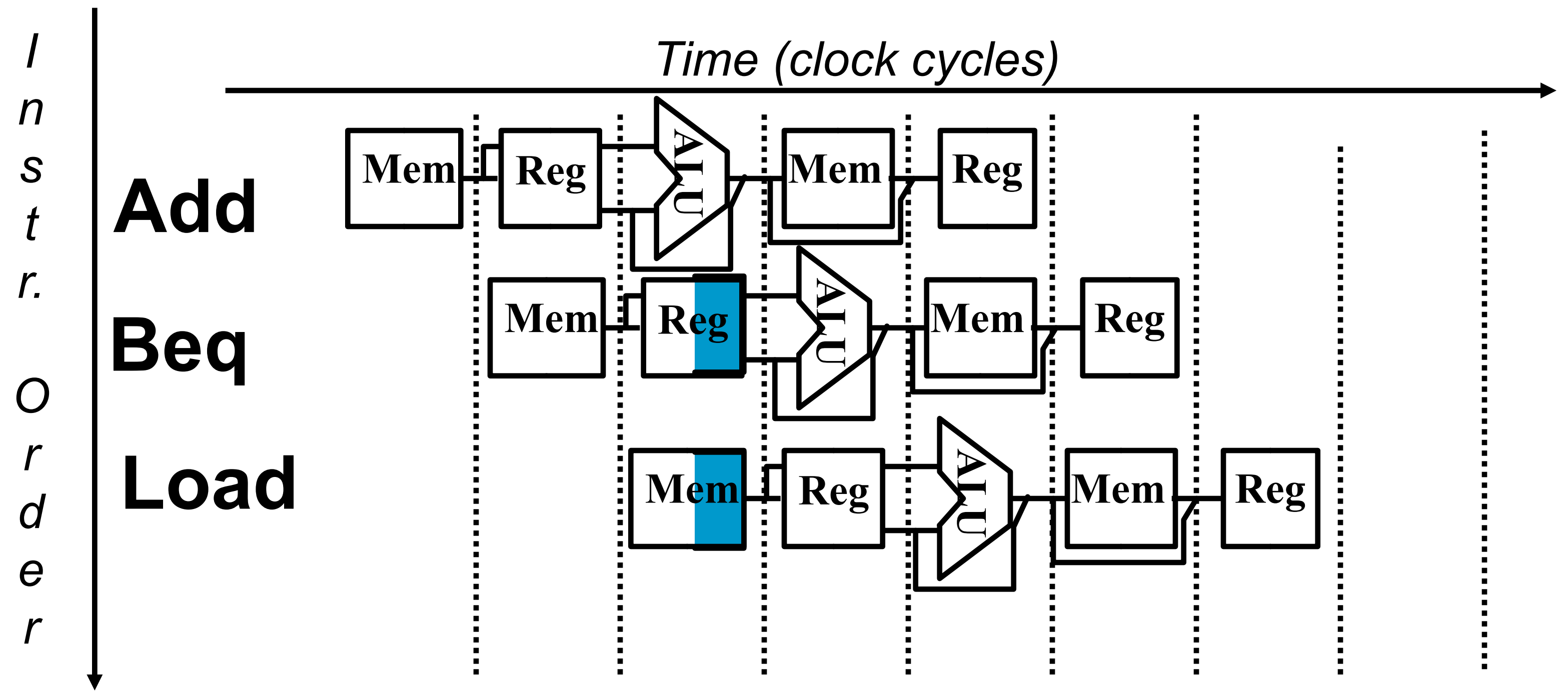
Control Hazard Solutions (1)



- Impact: 2 clock cycles per branch instruction => slow

Control Hazard Solutions (2)

- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right 50% of time)
- More dynamic scheme: history of 1 branch (90%)

Predict branch not taken

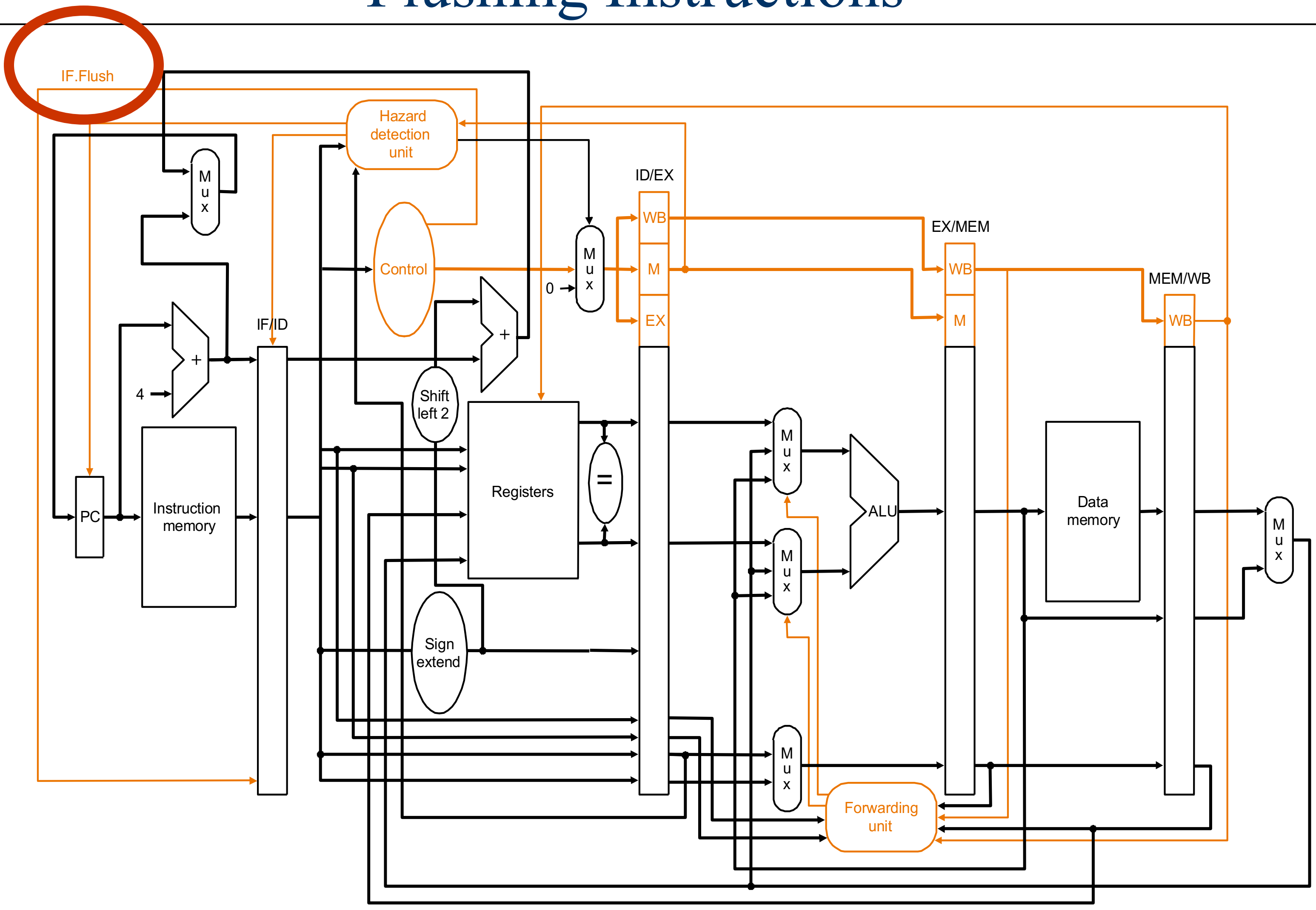
Branch Inst (i)	**IF**	**ID**	**EX**	**MEM**	**WB**			
Inst i+1		**IF**	**ID**	**EX**	**MEM**	**WB**		
Inst i+2			**IF**	**ID**	**EX**	**MEM**	**WB**	
Inst i+3				**IF**	**ID**	**EX**	**MEM**	**WB**
Inst i+4					**IF**	**ID**	**EX**	**MEM**

Correct Prediction : Zero Cycle Branch Penalty!

Branch Inst (i)	**IF**	**ID**	**EX**	**MEM**	**WB**		
Inst i+1		**IF**	**nop**	**nop**	**nop**	**nop**	
Branch target			**IF**	**ID**	**EX**	**MEM**	**WB**

Incorrect Prediction - waste one cycle
How to flush pipeline?

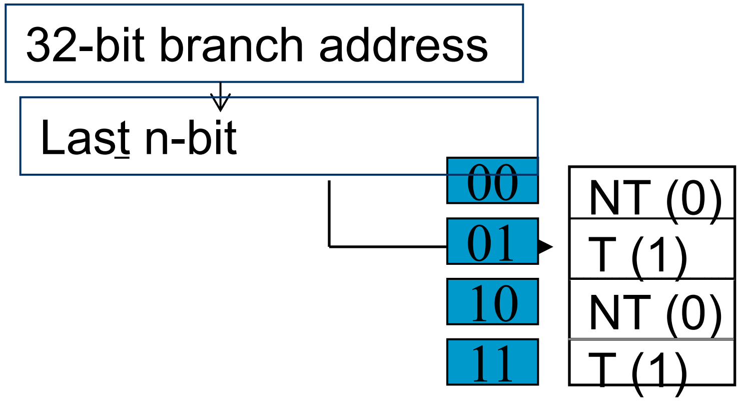
Flushing Instructions



Zero the instruction field of the IF/ID pipeline register
  sll $0, $0, 0

Dynamic Branch Prediction

- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check (saves HW, but may not be right branch)



branch history table = 2^n

Example: --00 T, --01 NT, --10 T, --11 T, --00 NT, --01 NT, --10 NT, --11 T,

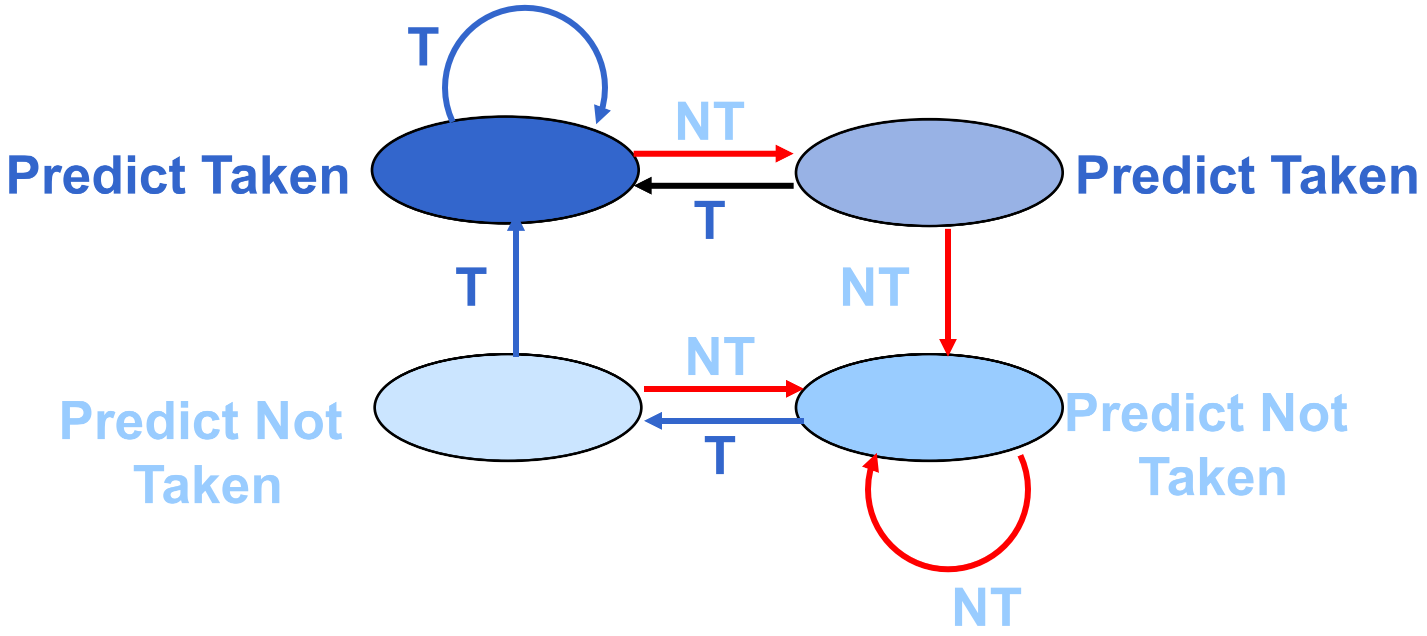
Dynamic Branch Prediction

- Problem: in a loop, 1-bit BHT will cause 2 mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts *exit* instead of looping
  - Only 80% accuracy even if loop 90% of the time

```
L1:      :::
         :::
         :::
      SUBI      R1,  #8
      BNEZ      R1,  L1
```

Dynamic Branch Prediction

- Solution: 2-bit scheme which changes prediction only if get misprediction *twice:*



Exercise

?

- Branch outcome of a single branch
  - T T T N N N T T T
- How many instances of this branch instruction are mis-predicted with a 1-bit predictor?
- How many instances of this branch instruction are mis-predicted with a 2-bit predictor?

T	T	T	N	N	N	T	T	T
X	T	T	T	T	N	N	N	T
			X	X		X	X	

More on branch prediction

- Problems with predicted taken?
  - Need to calculate target address
  - Solution: branch target buffer
- Correlating predictor
  - A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches
- Tournament branch predictor
  - A branch predictor with multiple prediction for each branch and a selection mechanism that chooses which predictor to enable for a given branch

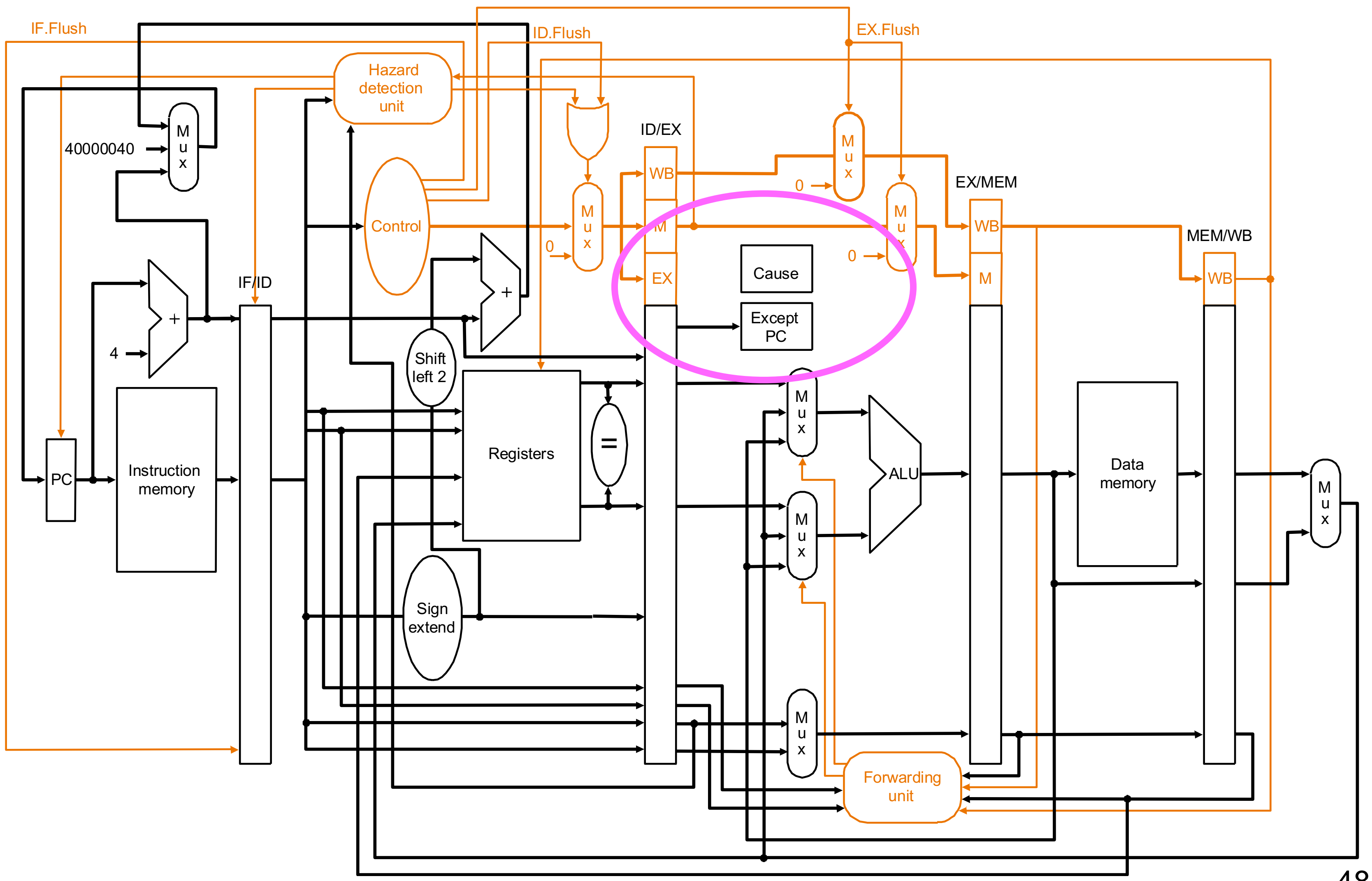
Exceptions

inst i	IF	ID	EX	MEM	WB	
inst i+1		IF	ID	EX	MEM	
inst i+2			IF	ID	EX	← faulting instruction - overflow
inst i+3				IF	ID	
inst i+4					IF	

Steps to handle exceptions:

- Flush the instruction in the IF, ID and EX stages.
- Let all preceding instructions complete if they can
- EPC = address of (offending instruction) + 4
- Call the OS to handle the exception
  - PC = 0x40000040
- Return from the exception handler
  - PC = EPC -4

Datapath with Controls to Handle Exceptions

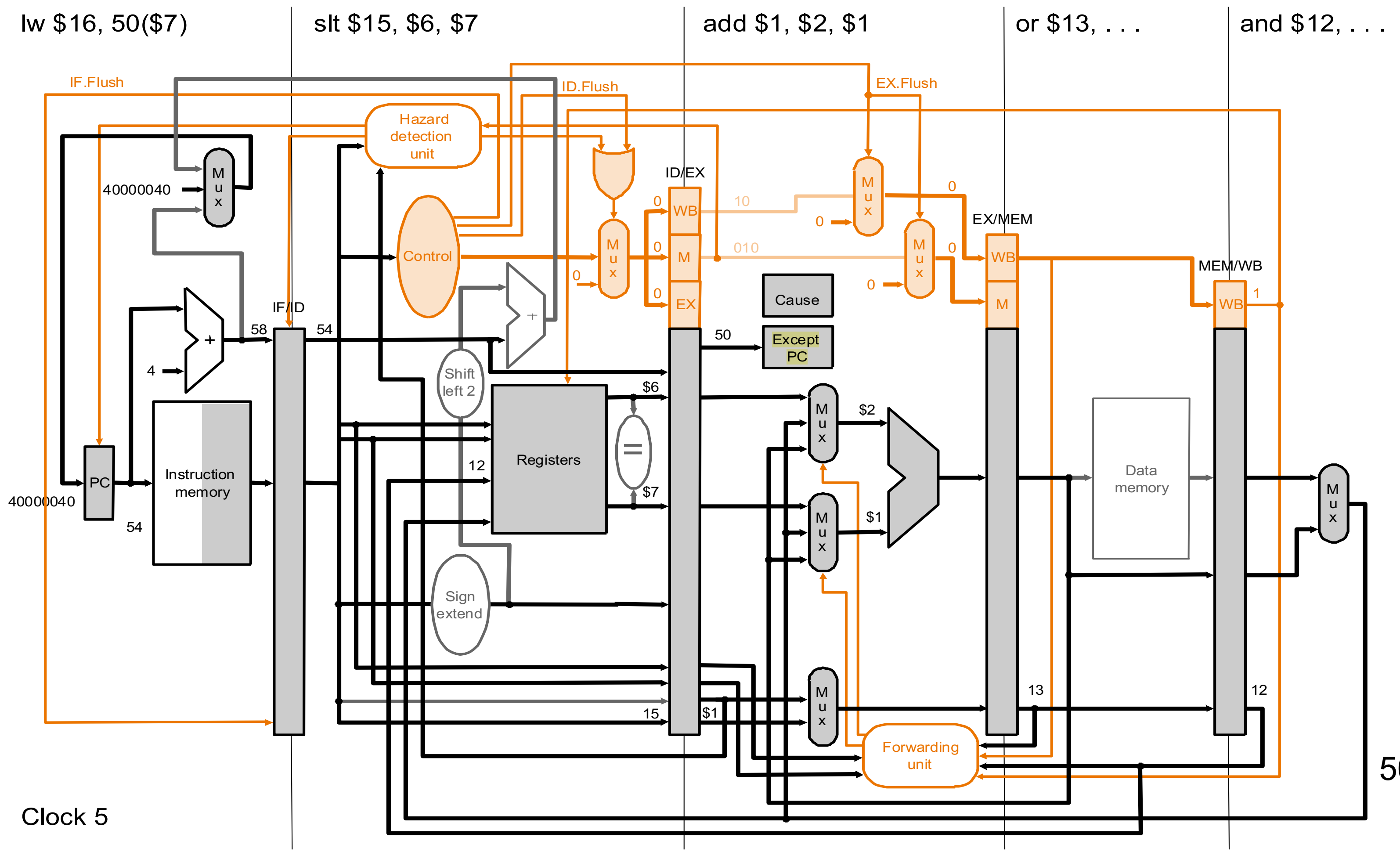


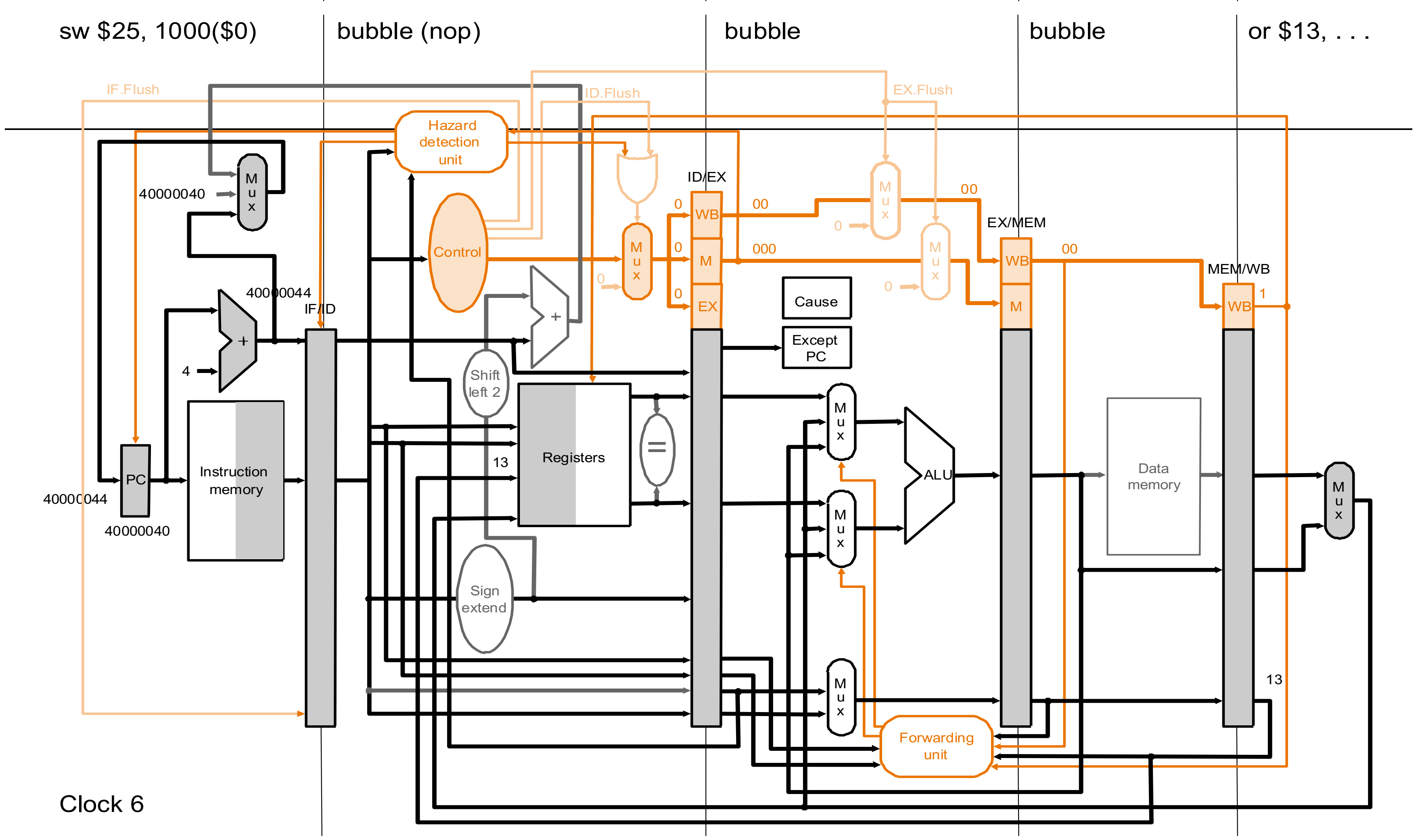
Example: Handling Exception

- Given the following instruction sequence:
  - 40hex sub $11, $2, $4
  - 44hex and $12, $2, $5
  - 48hex or $13, $2, $6
  - 4Chex add $1, $2, $1
  - 50hex slt $15, $6, $7
  - 54hex lw $16, 50($7)
- Assume the instruction to be invoked on an exception begin like this
  - 40000040hex sw $25, 1000($0)
  - 40000044hex sw $26, 1004($0)
- Show what happens in the pipeline if an overflow exception occurs in the add instruction.

Example: Handling Exception

1. The overflow is detected when "add" is in the EXE stage.
2. Save PC+4 (50) in EPC
3. Assert IF.Flush, ID. Flush, and EX.Flush





1. Fetch stage: the first instruction of the exception routine
2. Instruction prior to the add instruction complete

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

IF	ID	EX	MEM	WB		
IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB
		IF	ID	EX	MEM	WB

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Static Multiple Issue

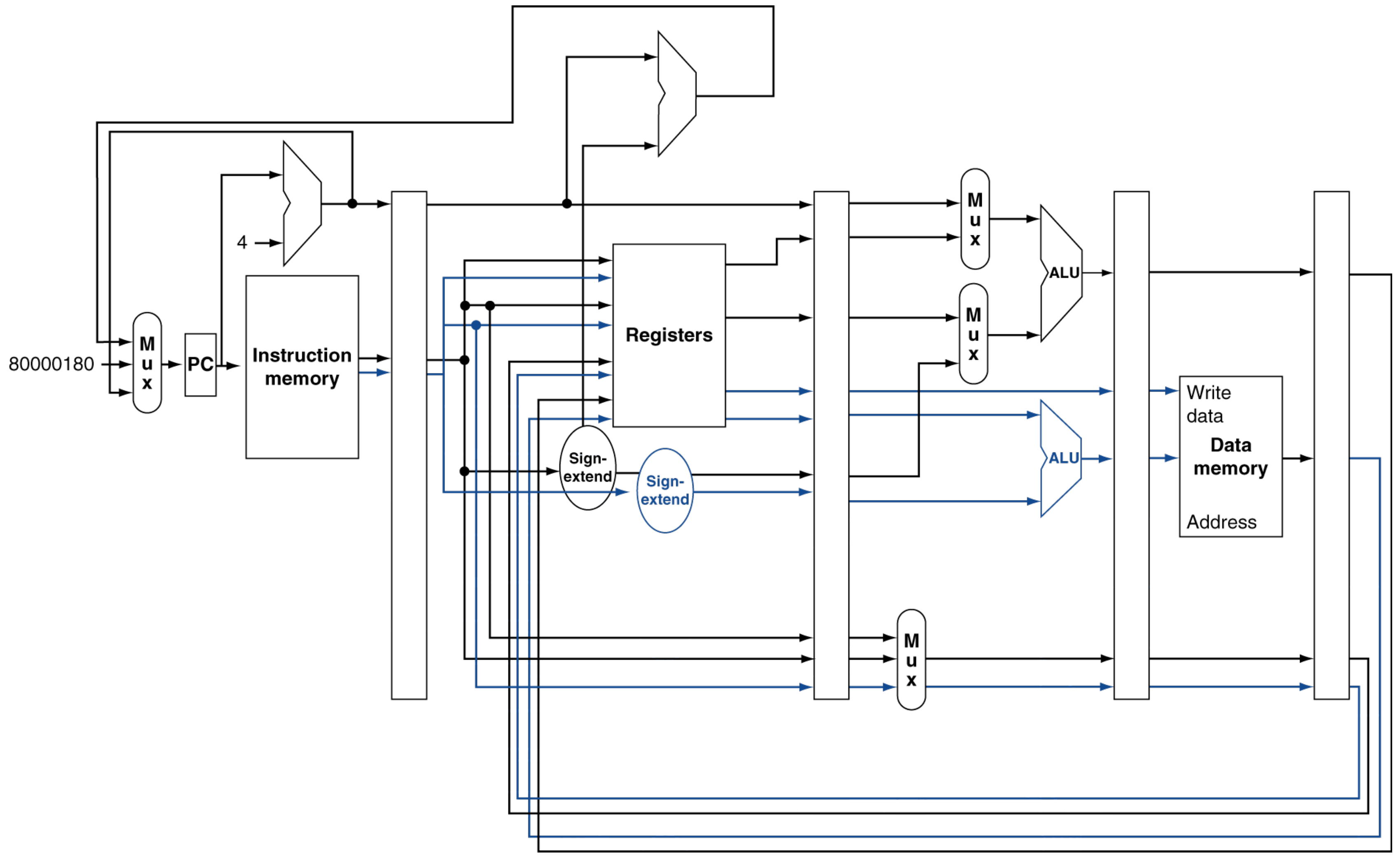
- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue



Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet

    -
      ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,–4       # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
`Loop:`	nop	`lw   $t0, 0($s1)`	1
	`addi $s1, $s1,–4`	nop	2
	`addu $t0, $t0, $s2`	nop	3
	`bne  $s1, $zero, Loop`	`sw   $t0, 4($s1)`	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      bne  $s1, $zero, Loop
```

Unrolled Loop That Minimizes Stalls

Renaming

```
Loop:     lw    $t0, 0($s1)
          addu  $t0, $t0, $s2
          sw    $t0, 0($s1)

          lw    $t0, -4($s1)
          addu  $t0, $t0, $s2
          sw    $t0, -4($s1)

          lw    $t0, -8($s1)
          addu  $t0, $t0, $s2
          sw    $t0, -8($s1)

          lw    $t0, -12($s1)
          addu  $t0, $t0, $s2
          sw    $t0, -12($s1)

          addi  $s1, $s1,-16
          bne   $s1, $zero, Loop
```

$\longrightarrow$

```
Loop:     lw    $t0, 0($s1)
          addu  $t0, $t0, $s2
          sw    $t0, 0($s1)

          lw    $t1, -4($s1)
          addu  $t1, $t0, $s2
          sw    $t1, -4($s1)

          lw    $t2, -8($s1)
          addu  $t2, $t0, $s2
          sw    $t2, -8($s1)

          lw    $t3, -12($s1)
          addu  $t3, $t0, $s2
          sw    $t3, -12($s1)

          addi  $s1, $s1,-16
          bne   $s1, $zero, Loop
```

Loop Unrolling Example

	ALU/branch	Load/store	cycle
`Loop:`	`addi $s1, $s1,-16`	`lw $t0, 0($s1)`	1
	`nop`	`lw $t1, 12($s1)`	2
	`addu $t0, $t0, $s2`	`lw $t2, 8($s1)`	3
	`addu $t1, $t1, $s2`	`lw $t3, 4($s1)`	4
	`addu $t2, $t2, $s2`	`sw $t0, 16($s1)`	5
	`addu $t3, $t4, $s2`	`sw $t1, 12($s1)`	6
	`nop`	`sw $t2, 8($s1)`	7
	`bne $s1, $zero, Loop`	`sw $t3, 4($s1)`	8

- IPC = 14/8 = 1.75
  - – Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

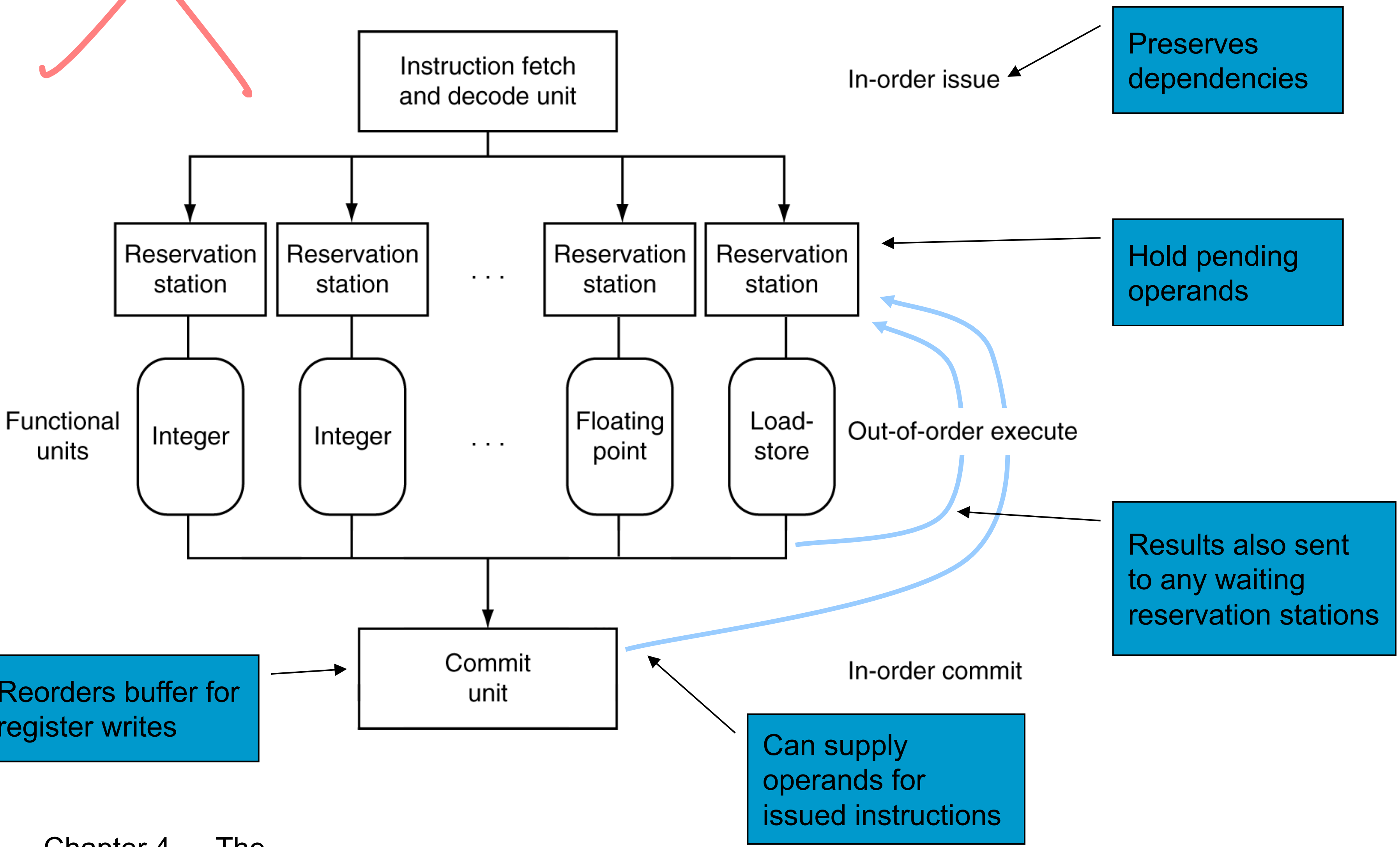
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start `sub` while `addu` is waiting for lw

Dynamically Scheduled CPU



Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - 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