Chapter 4

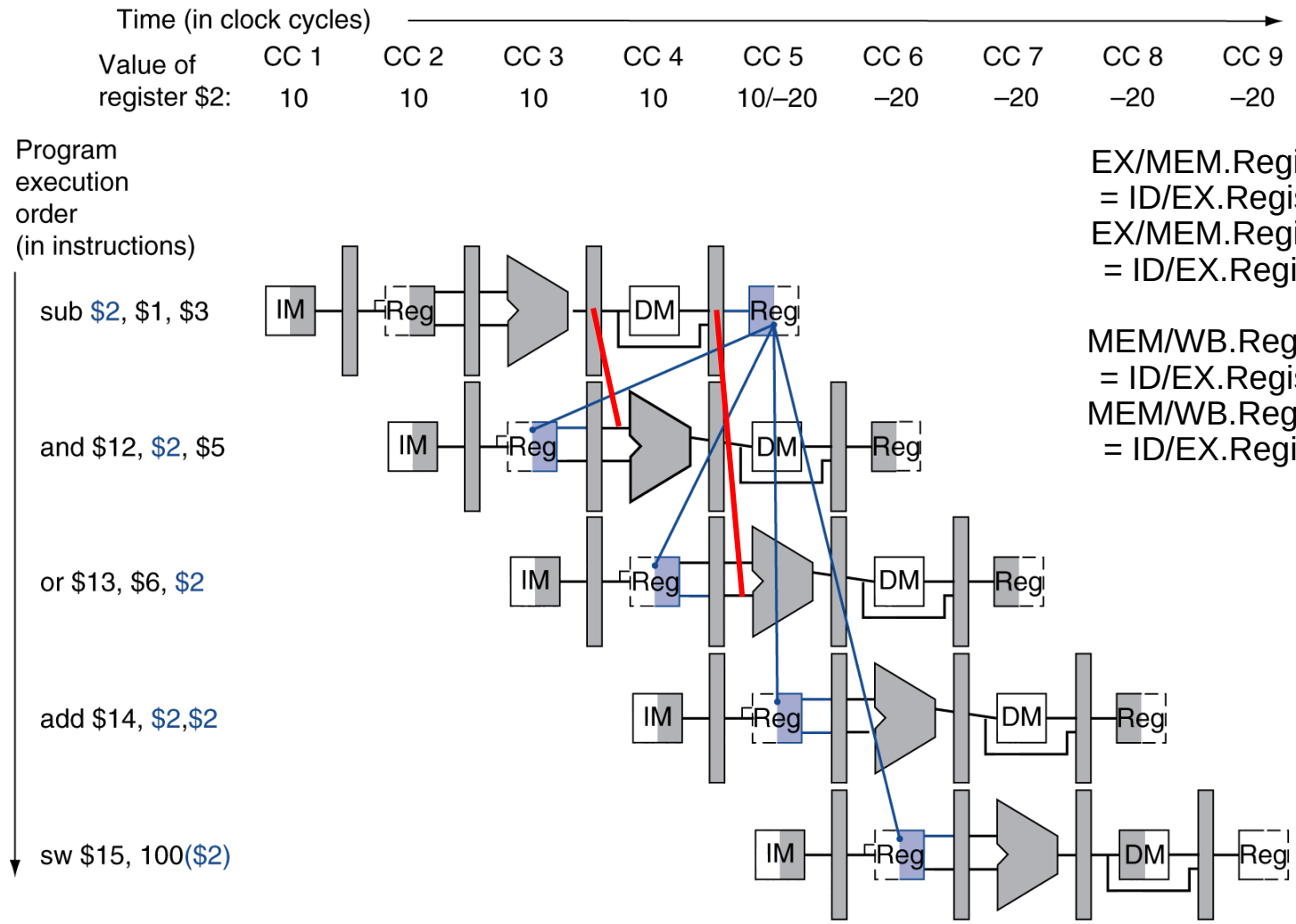
Part D :

Datapath and Control for Data and Control Hazards

Exceptions

Parallelism via Instructions

Data Hazard Example



Detecting the Need to Forward

- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

  } Fwd from EX/MEM pipeline reg

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

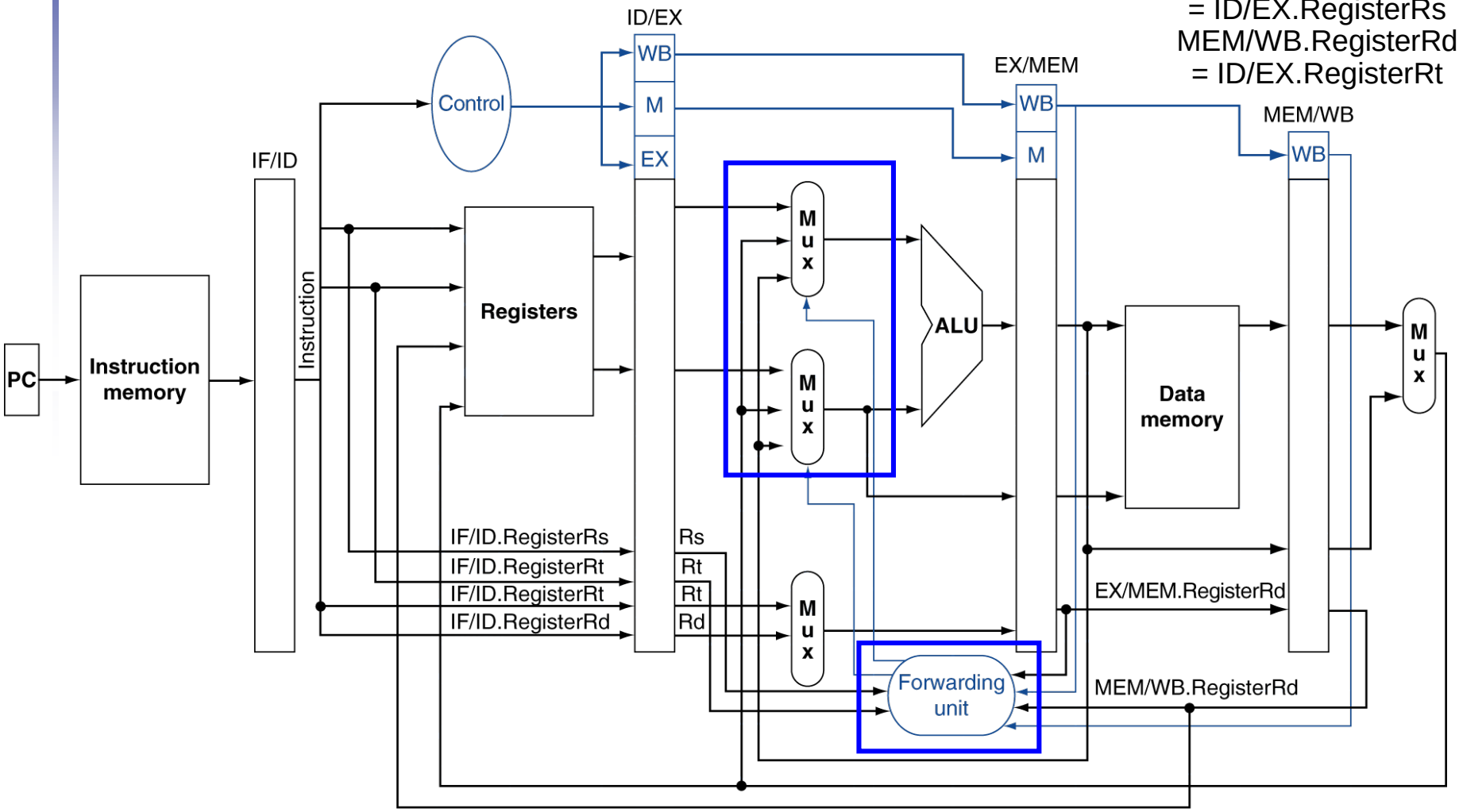
  } Fwd from MEM/WB pipeline reg

- *Example Dependence Detection*

Forwarding Logic : Fig. 4.56

EX/MEM.RegisterRd
= ID/EX.RegisterRs
EX/MEM.RegisterRd
= ID/EX.RegisterRt

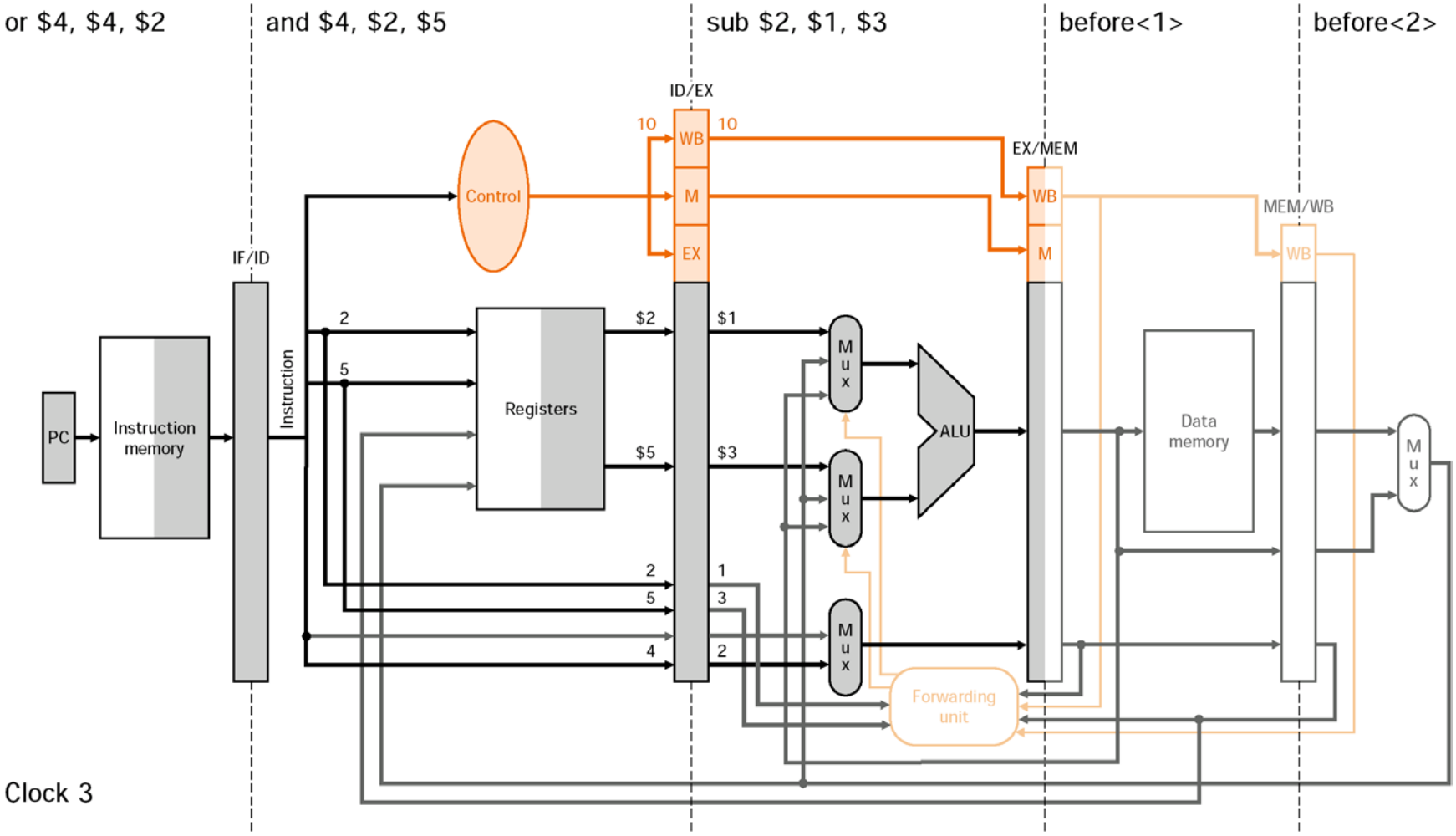
MEM/WB.RegisterRd
= ID/EX.RegisterRs
MEM/WB.RegisterRd
= ID/EX.RegisterRt



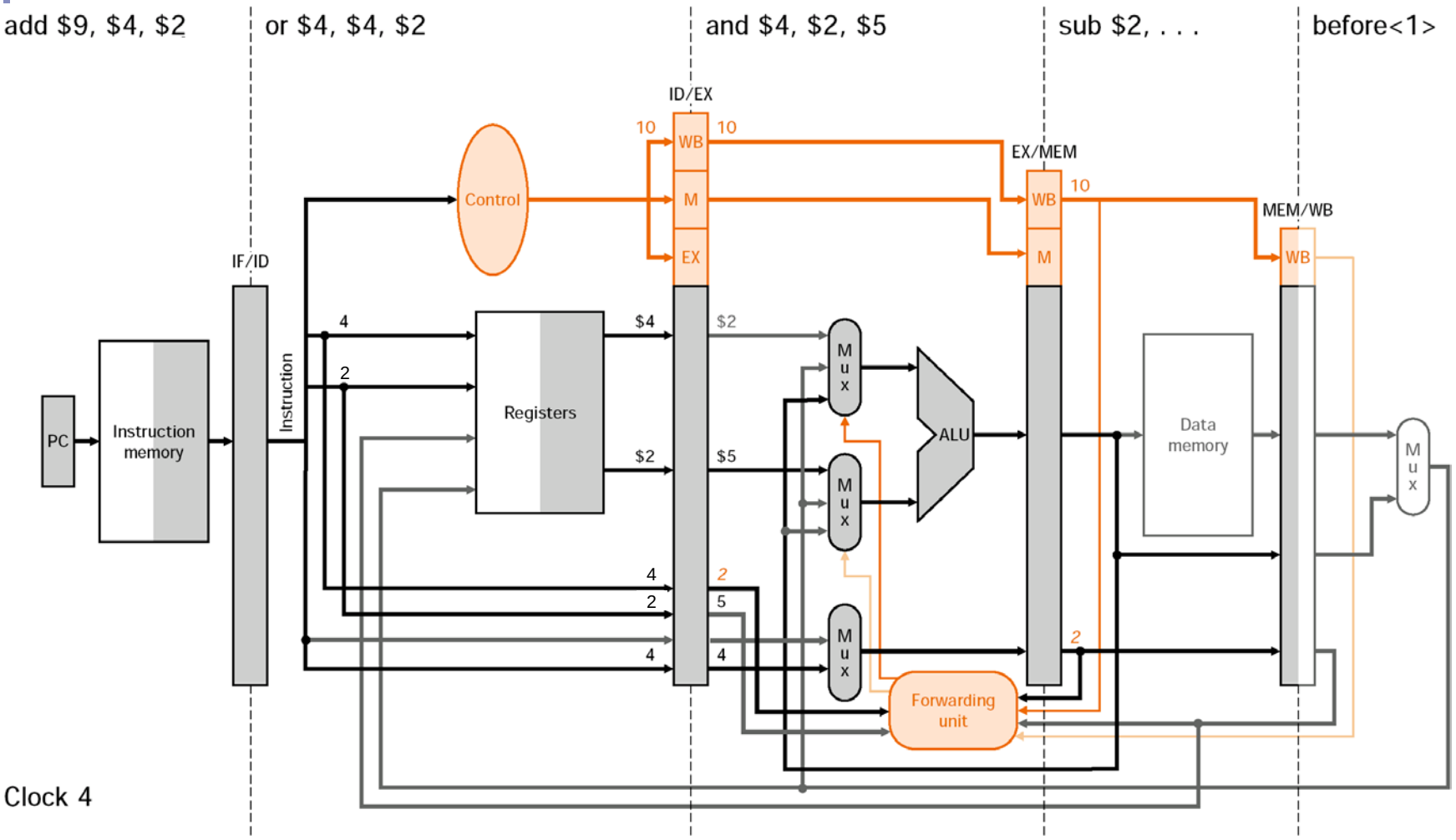
Forwarding Example

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

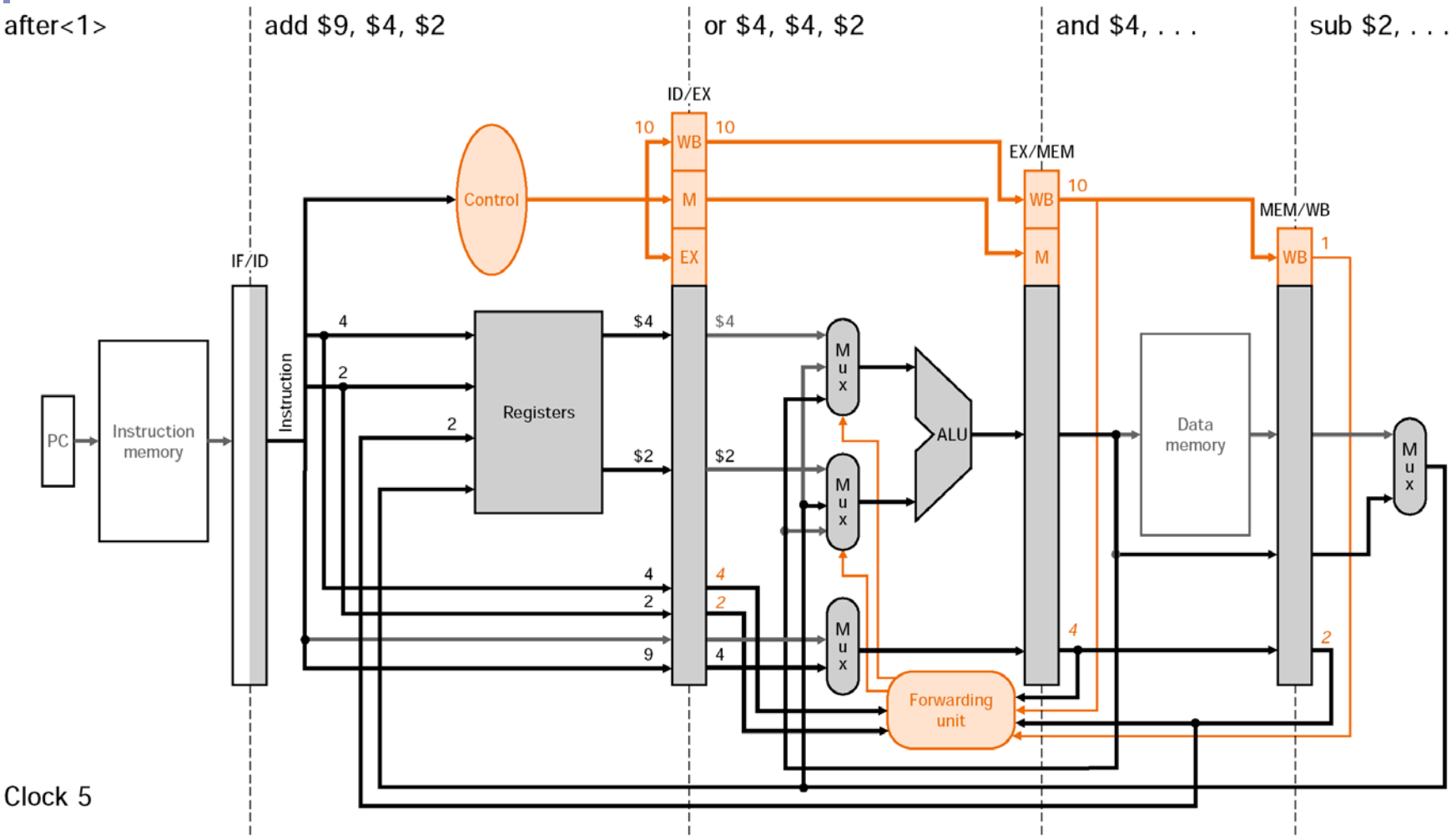
Forwarding Example



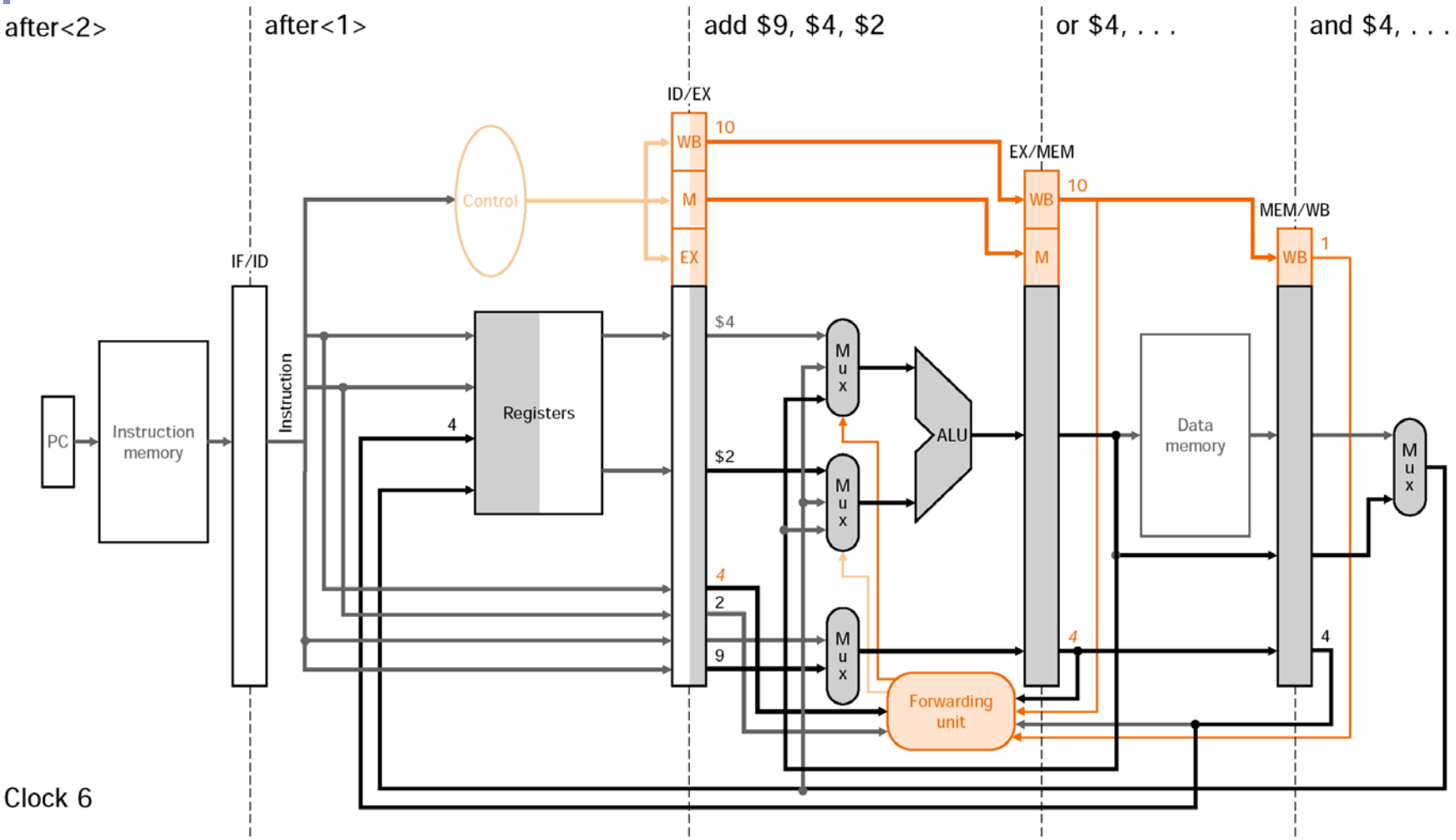
Forwarding Example



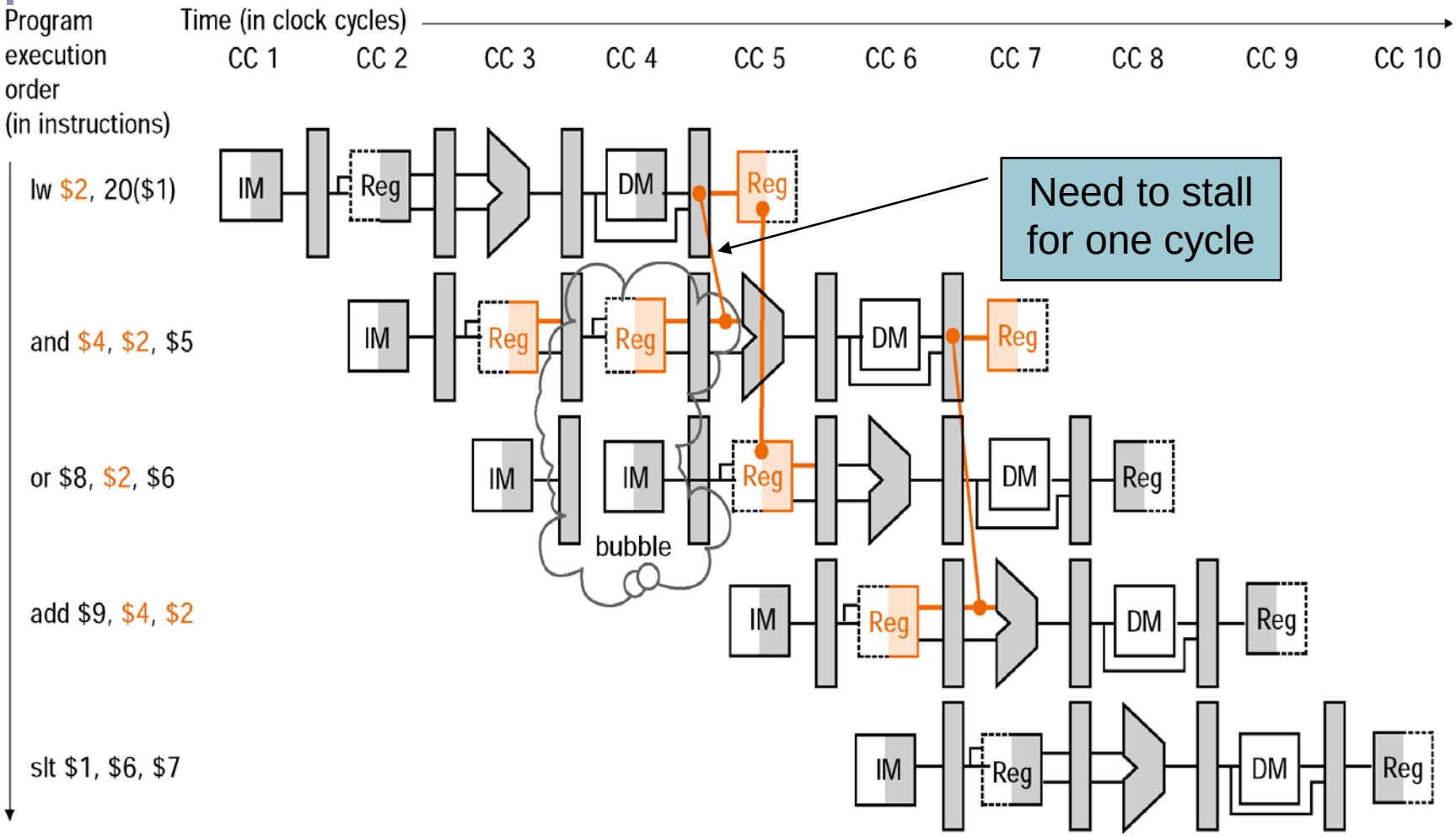
Forwarding Example



Forwarding Example



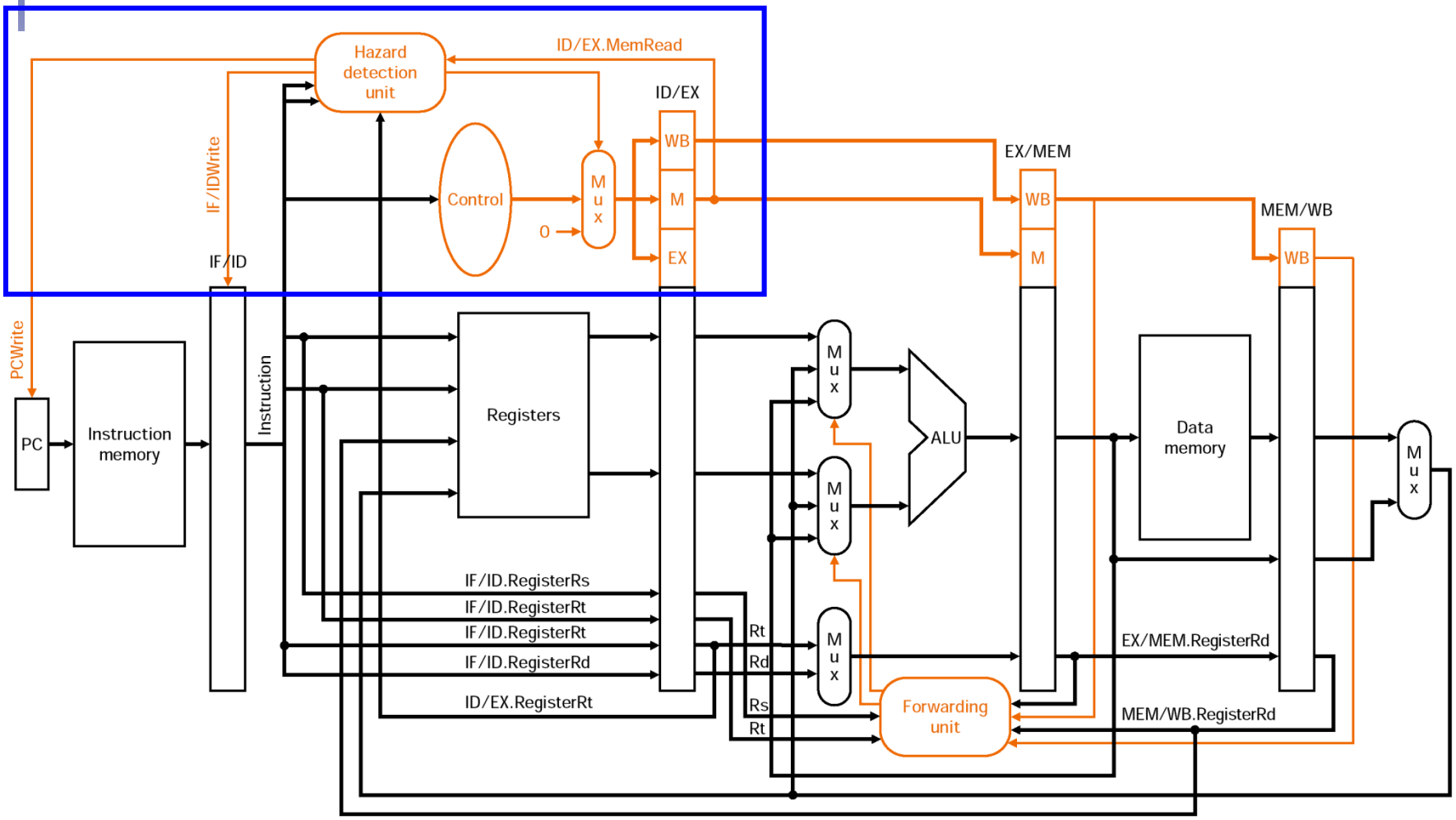
Data Hazard Requiring a Stall Fig. 4.58



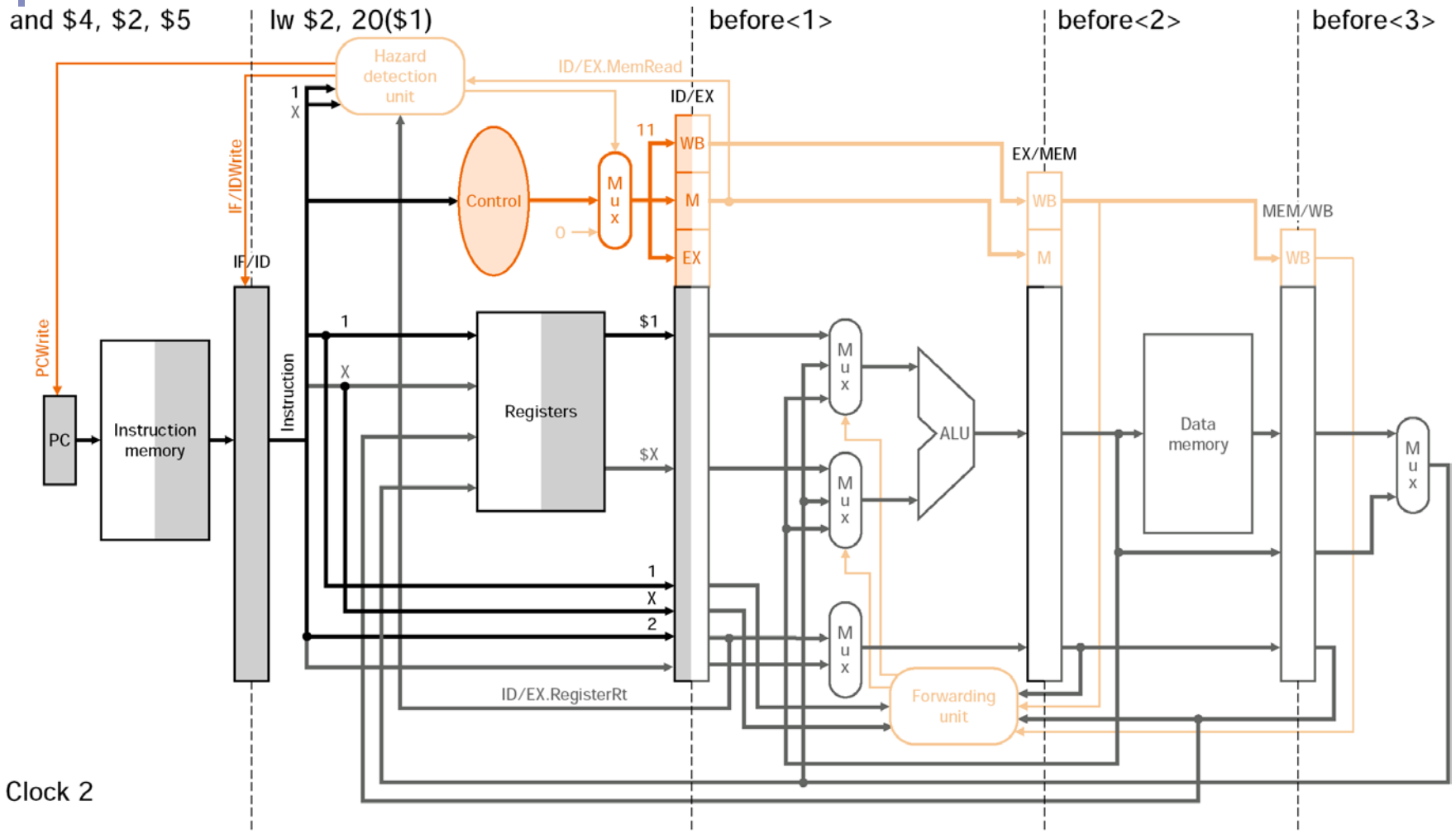
Load-Use Hazard Detection

- 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
- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Hazard Detection Unit

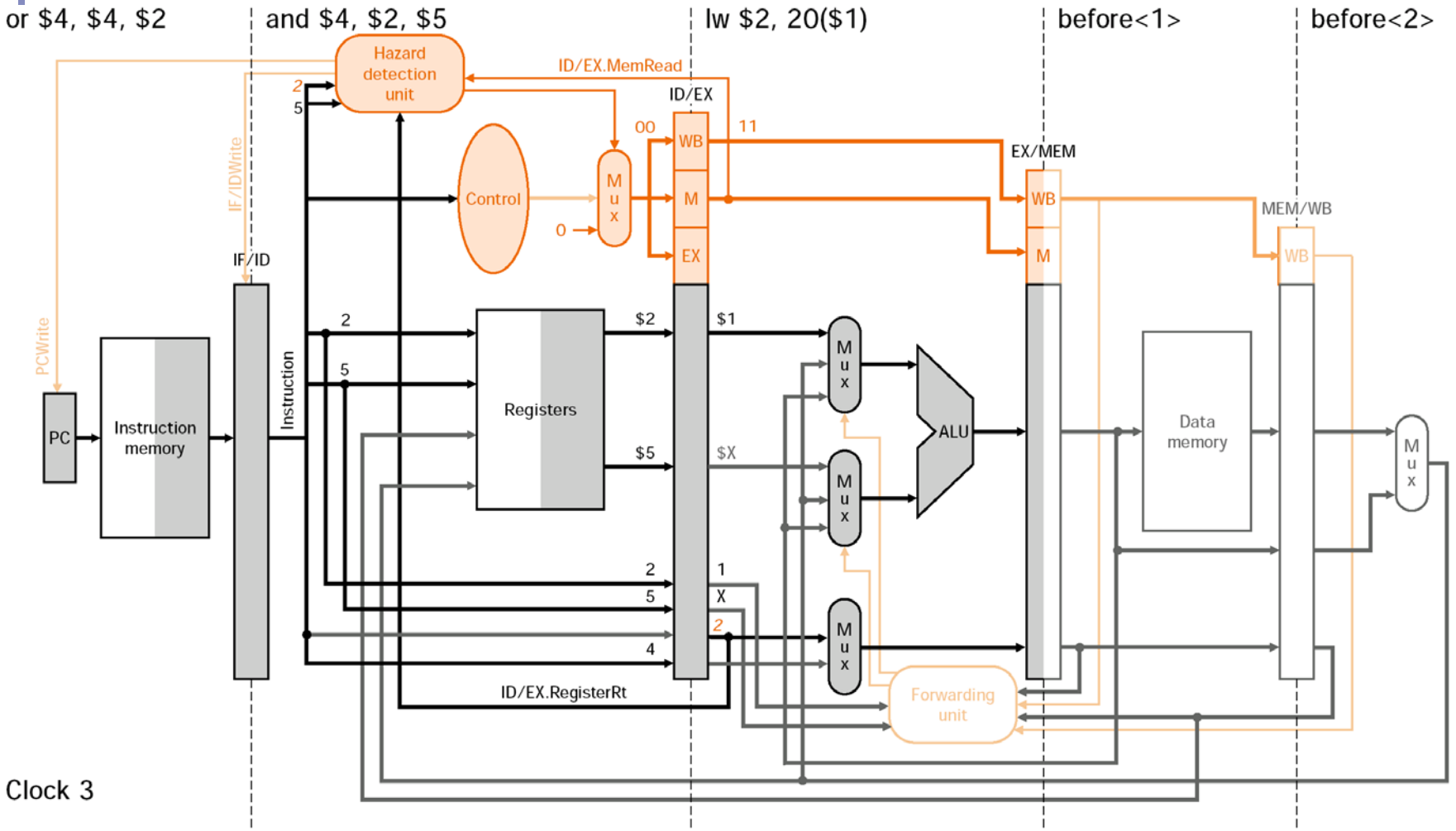
Stall Logic Fig. 4.60



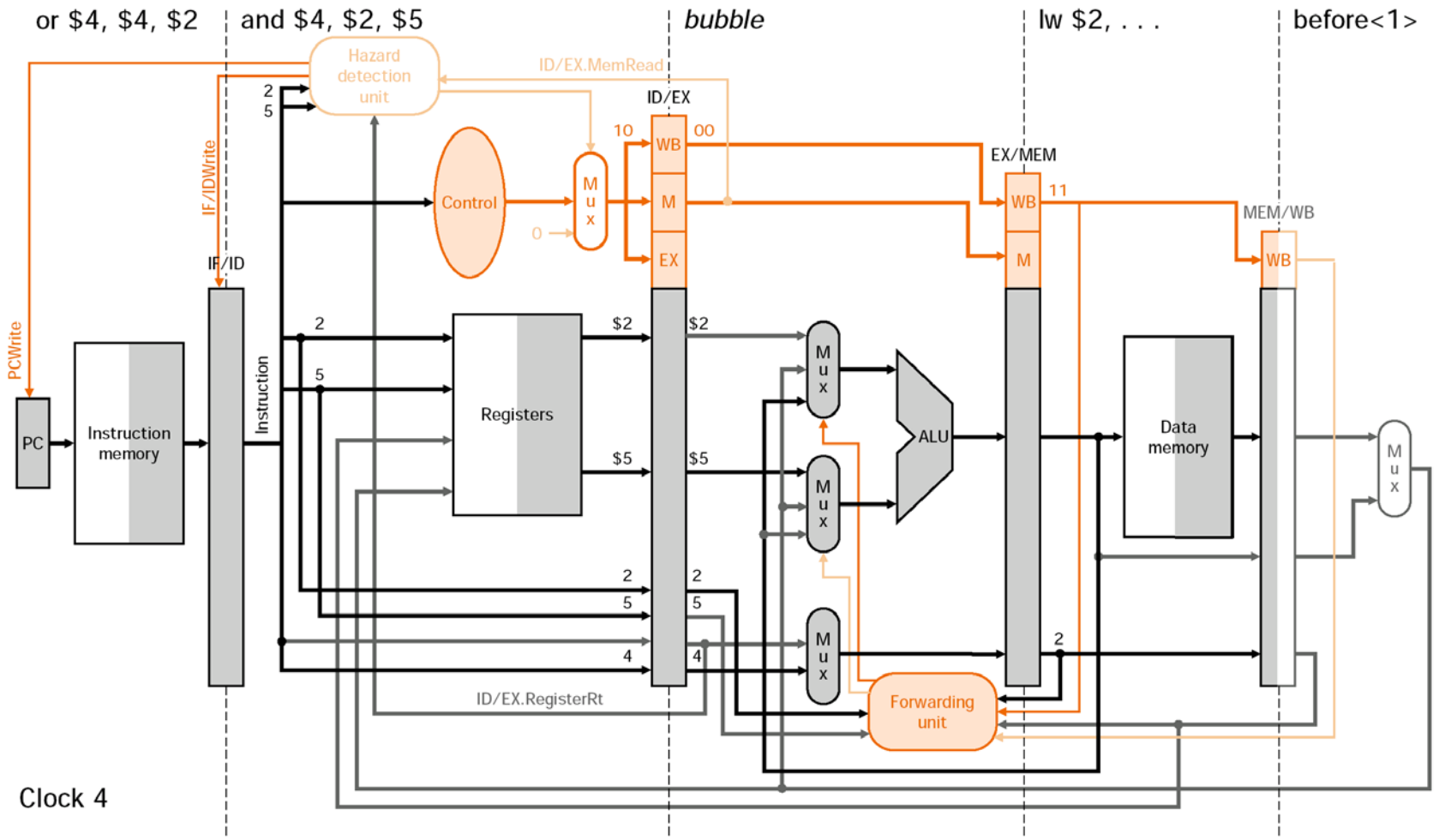
Stall Example



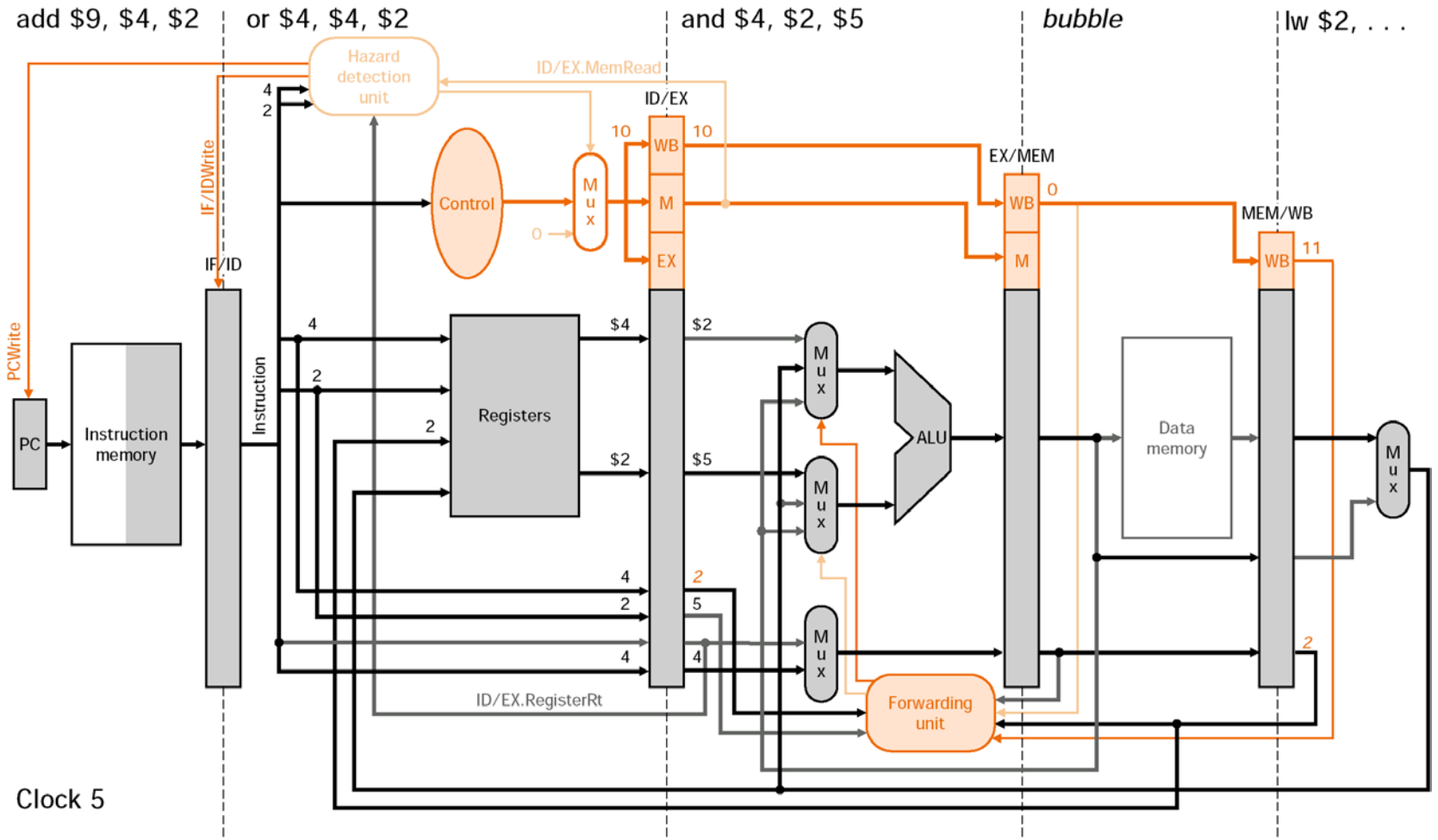
Stall Example



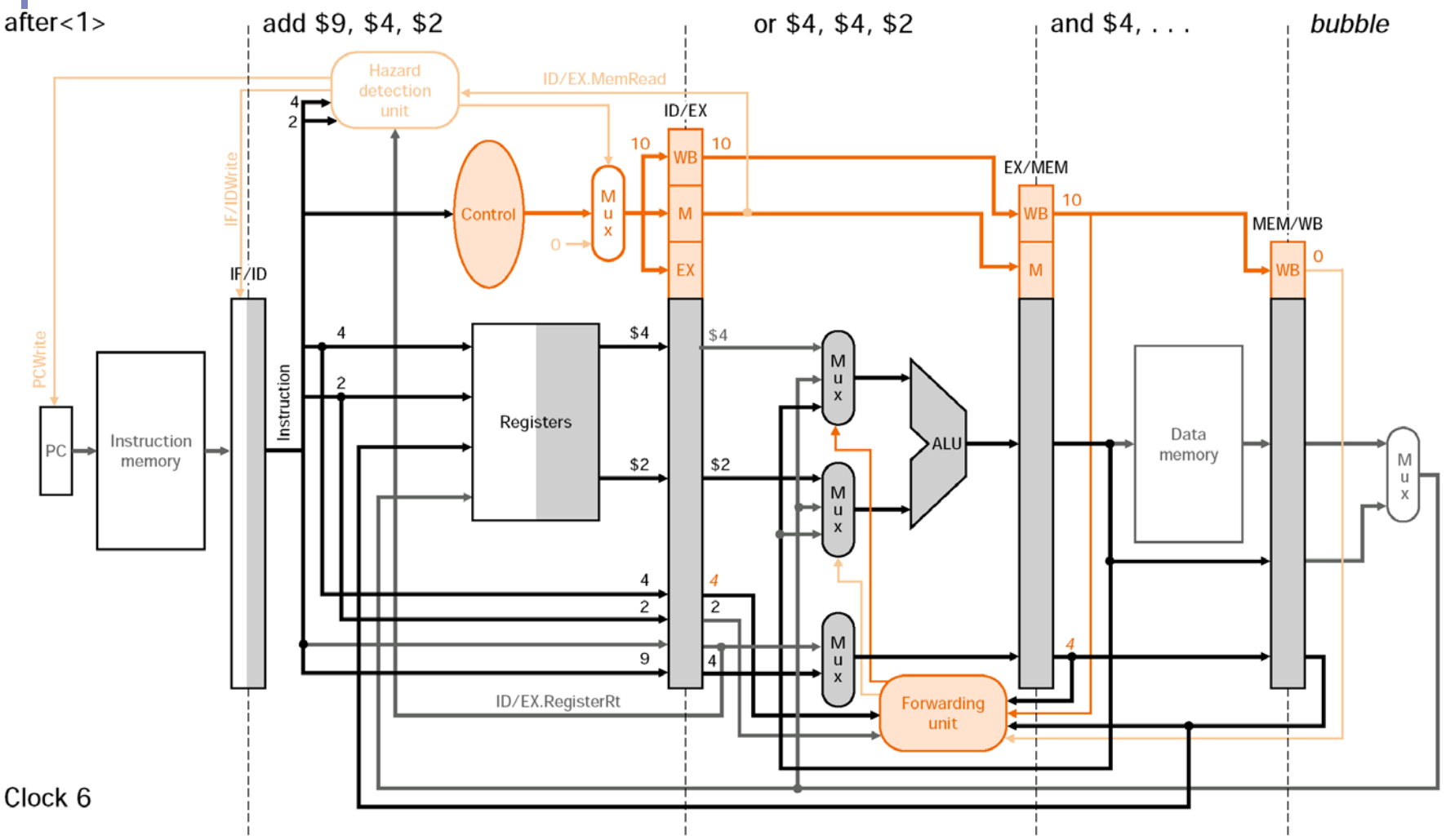
Stall Example



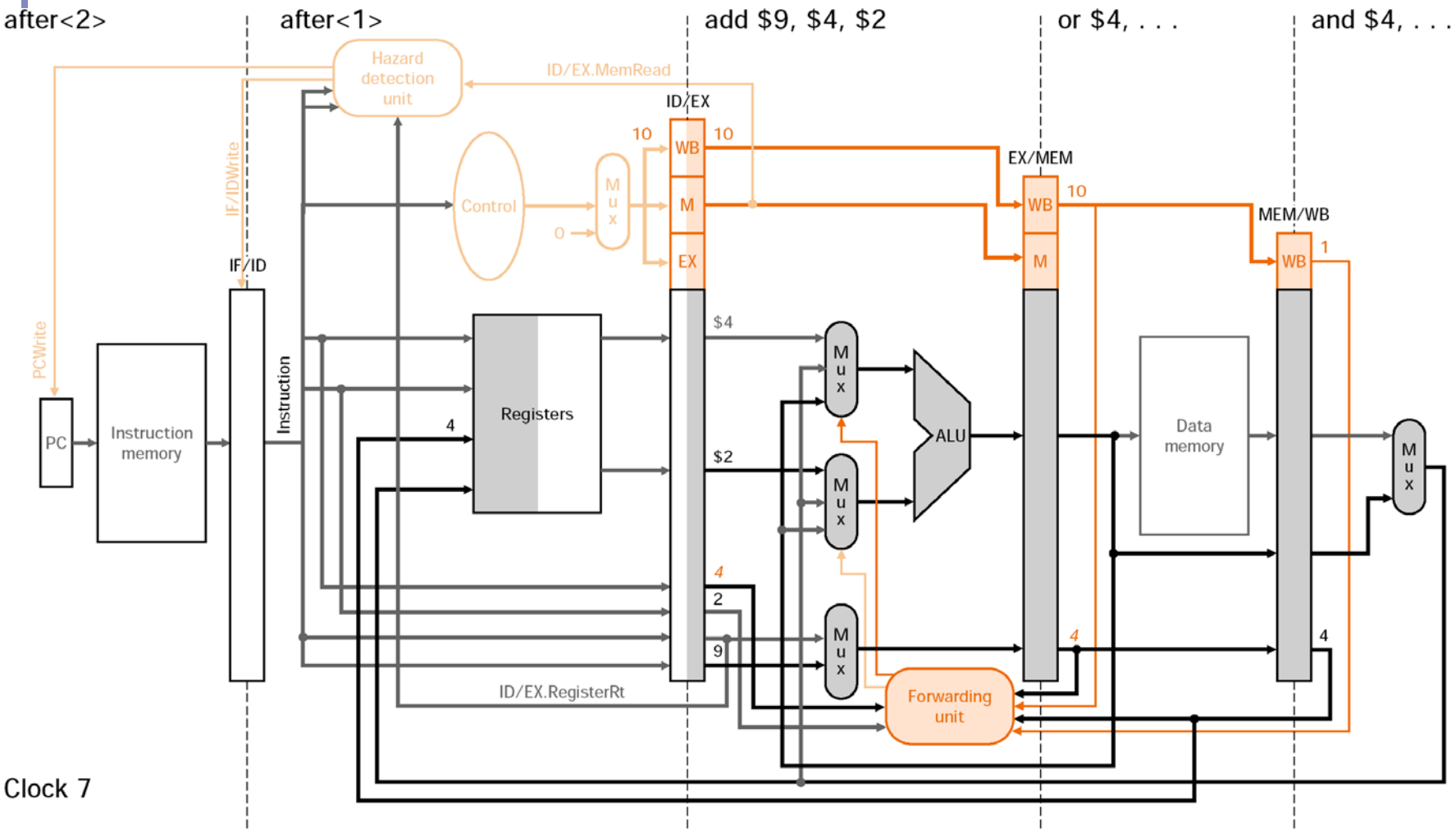
Stall Example



Stall Example



Stall Example



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

Exercise #1: Without / With Forwarding

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

	1	2	3	4	5	6	7	8	9	10	11
sub	IF	ID	EX	MEM	WB						
and		IF									
or											
add											
sw											

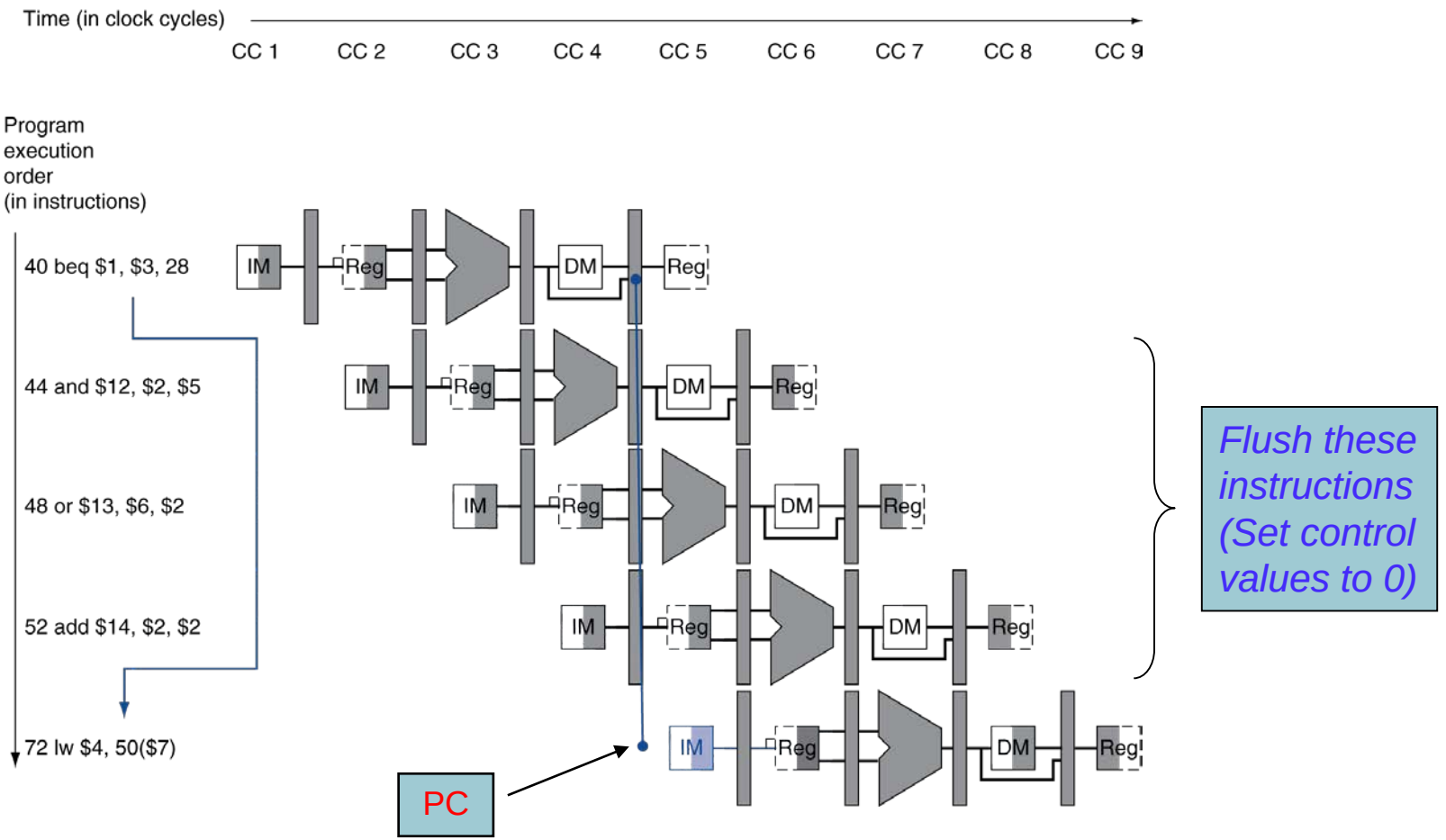
Exercise #2: Without / With Forwarding

```
lw  $2,  20($3)
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

	1	2	3	4	5	6	7	8	9	10	11
lw	IF	ID	EX	MEM	WB						
and		IF									
or											
add											
sw											

Branch / Control Hazards

- If branch outcome determined in MEM Fig. 4.61



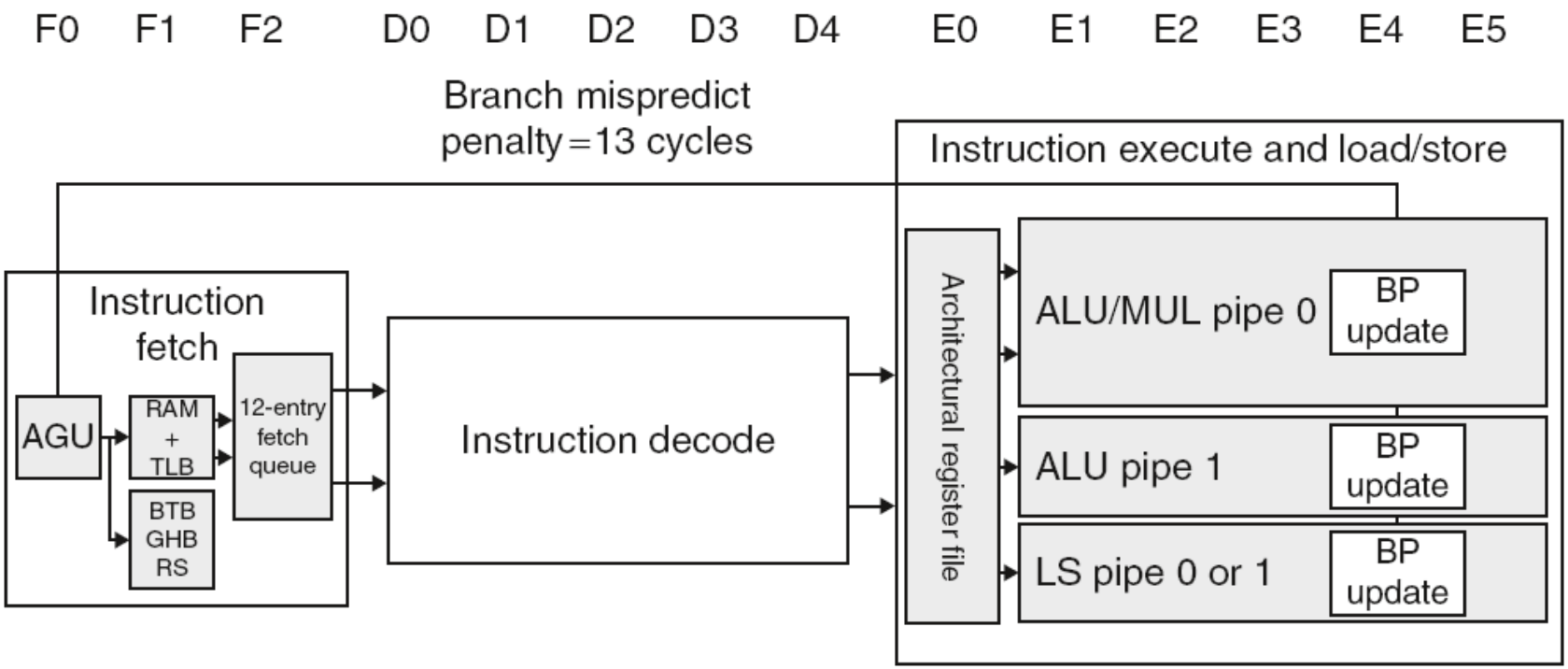
Control Hazards: Reducing the Penalty

- Branching is very common in code:
  - A 3-cycle stall penalty is too heavy!
- Many techniques invented to reduce the control hazard penalty:
  - Move branch decision calculation to earlier pipeline stage
    - **Early Branch Resolution**
  - Guess the outcome before it is produced
    - **Branch Prediction**
  - Do something useful while waiting for the outcome
    - **Delayed Branching**

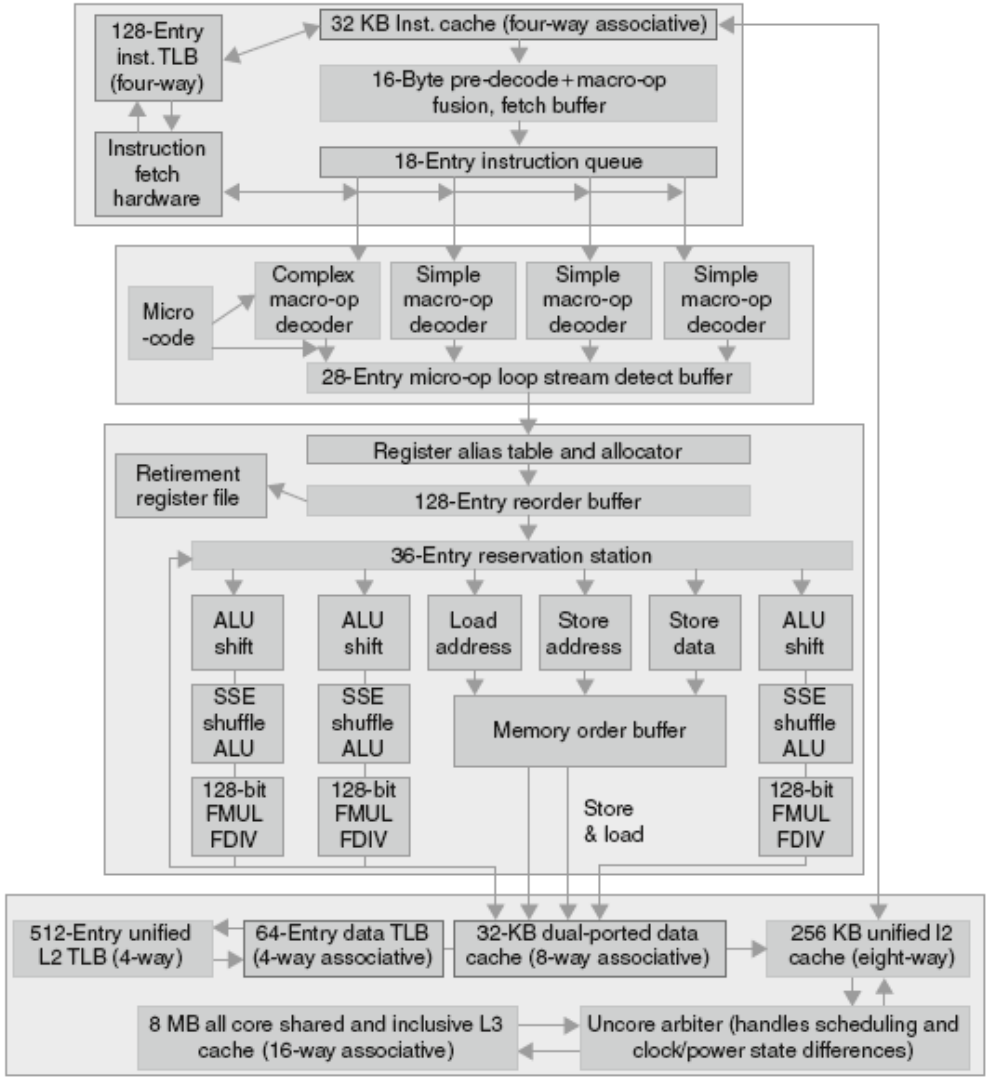
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
  - *Reading pp. 318 – 319*
  - *Example pp. 319*
- 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

ARM Cortex-A8 Pipeline



Core i7 Pipeline



Pipeline Summary

- ISA influences design of datapath and control
- Pipelining improves instruction throughput using parallelism
- Hazards : structural, data, control
- Implementation

Chapter 4

Part D :

Exceptions

Parallelism via Instructions

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 (CP0)
- Save (E)PC of offending (or interrupted) instruction

- Cause register vs. Vectored Interrupts
  - Read cause, and transfer to relevant handler
    - Undefined opcode: C000 0000
    - Overflow: C000 0020
  - Determine action required
  - If restartable
    - Take corrective action
    - Use EPC to return to program
  - Otherwise
    - Terminate program
    - Report error using EPC, cause, ...

Instruction-Level Parallelism (ILP)

- Pipelining : executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - But dependencies reduce this in practice

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

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - No dependencies with a packet

- Think of an issue packet as a very long instruction
  Very Long Instruction Word (VLIW)
  - Specifies multiple concurrent operations
  - Two-issue packets Fig. 4.68, Example pp. 325
    - One ALU/branch instruction
    - One load/store instruction

Dynamic Multiple Issue

- "Superscalar" processors

- CPU decides whether to issue 0, 1, 2, each cycle
  - Avoiding structural and data hazards

- Allow the CPU to execute instructions out of order to avoid stalls
  - 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

Reading

- *pp. 342 ~ 344*

Cortex A8 and Intel i7

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-1024 KiB	256 KiB
3rd level caches (shared)	-	2- 8 MB

Concluding Remarks

- ISA influences design of datapath and control
- Pipelining improves instruction throughput using parallelism
- Hazards : structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall