EECS4030: Computer Architecture

The Processor (III)

Prof. Chung-Ta King
Department of Computer Science
National Tsing Hua University, Taiwan

Outline

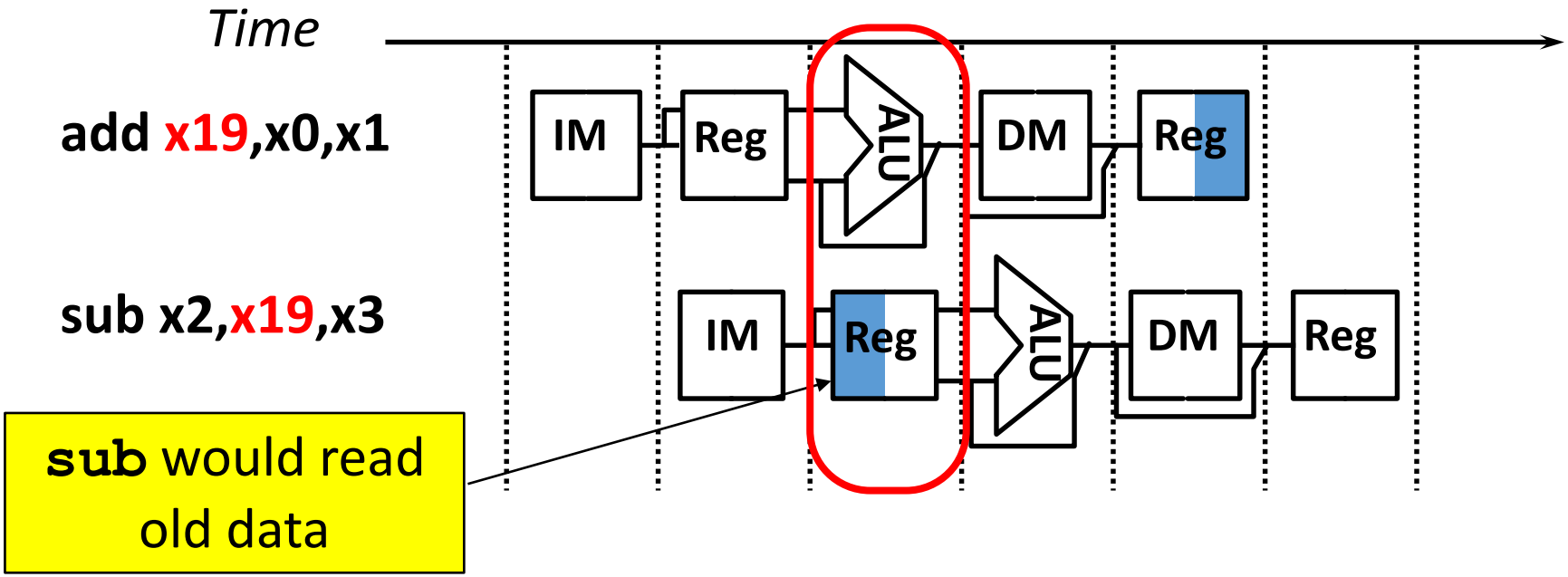
- Problem statement and logic review (Sec. 4.1, 4.2)
- Building a simple RISC-V processor with datapath and control (Sec. 4.3, 4.4)
- Building a pipelined RISC-V processor with datapath and control (Sec. 4.5, 4.6)
- Dealing hazards in pipelined processor: data and control hazards (Sec. 4.7, 4.8)
- Handling exceptions (Sec. 4.9)
- Instruction-level parallelism (Sec. 4.10)
- ARM Cortex-A53 and Intel Core i7 Pipelines (Sec. 4.11, 4.12)

Pipeline Hazards

- Situations that prevent starting the next instruction in the next cycle, e.g.,

```
add  x19,x0,x1
sub  x2,x19,x3
```

Note: *Data flow between instructions through registers*

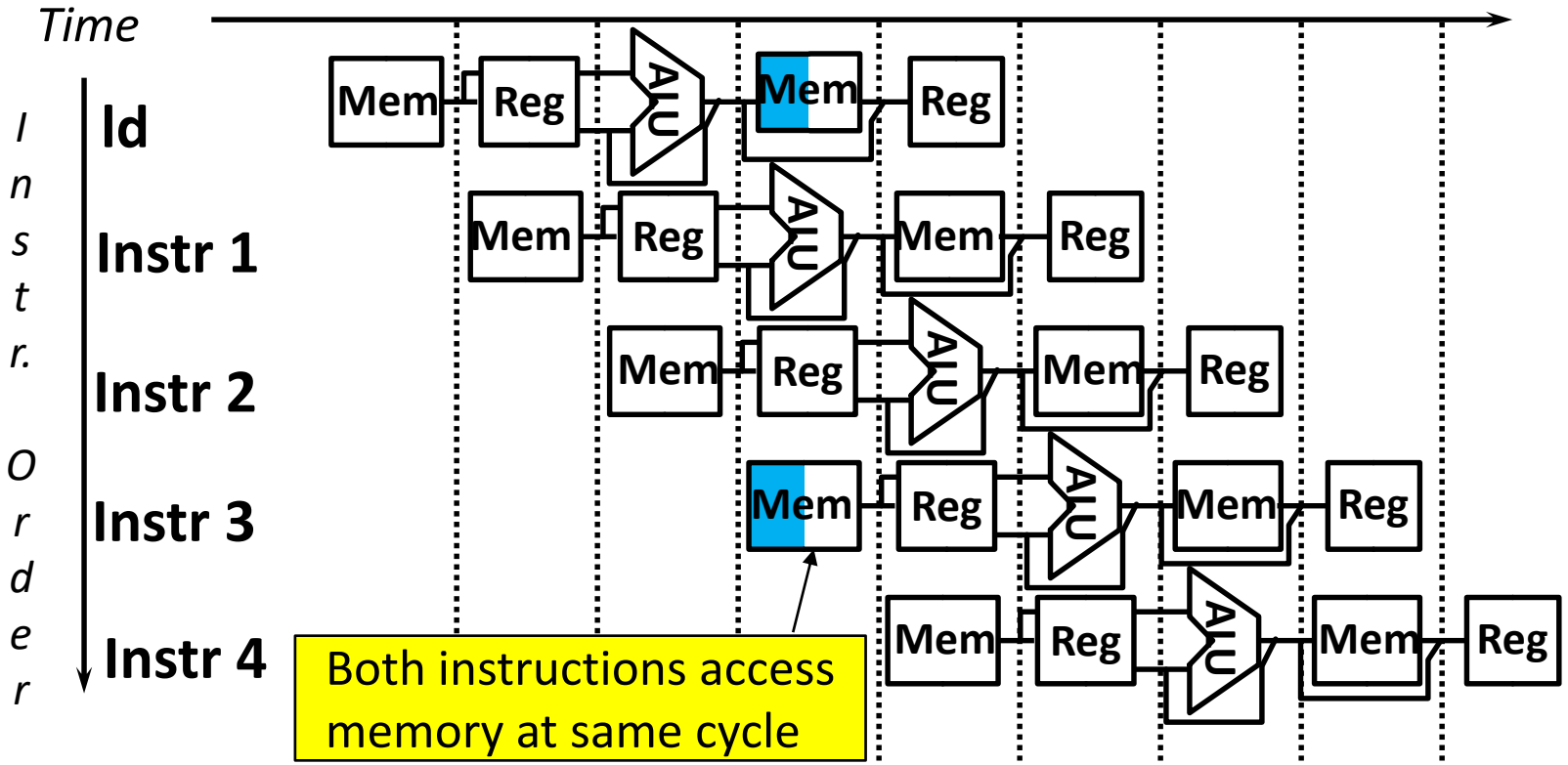


Three Types of Pipeline Hazards

- Structure hazards
  - Conflict for use of a resource, e.g. ALU, memory
    → need to wait for previous instruction to complete so to use the resource
- Data hazard
  - Need to wait for previous instruction to complete its data read/write so to use the storage (registers/memory)
- Control hazard
  - Need to wait for previous control-flow instruction to complete to decide whether to execution
- Can always resolve hazards by waiting
  - Pipeline control must detect the hazard
  - Take action (or delay action) to resolve hazards

Structure Hazards

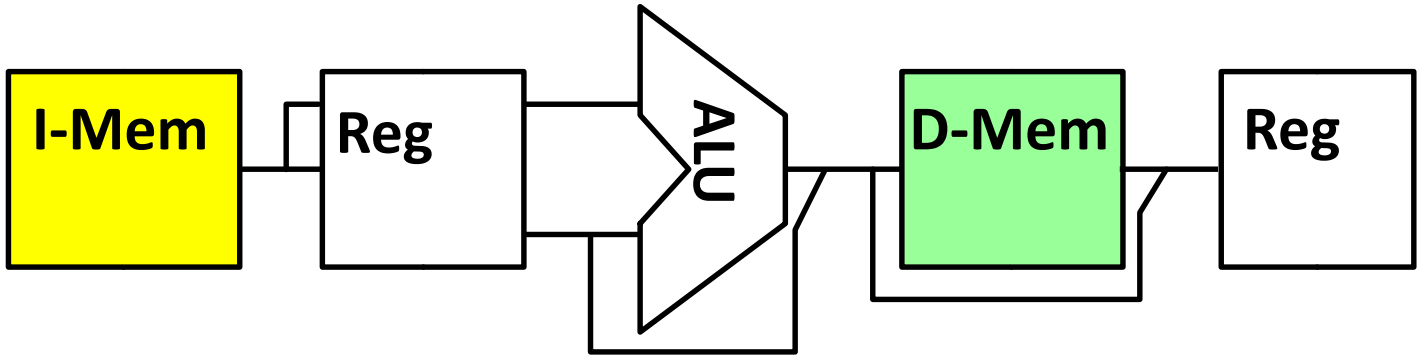
- Suppose RISC-V pipeline has only a single memory
  - Load/store requires data access at its MEM stage
  - The third following instruction in IF also requires memory



Structure Hazards

- Suppose RISC-V pipeline has only a single memory
  - The third following instruction would have to stall for that cycle → cause a pipeline "bubble"

- Solution: use two memories (data, instruction)
  - Or separate instruction/data caches

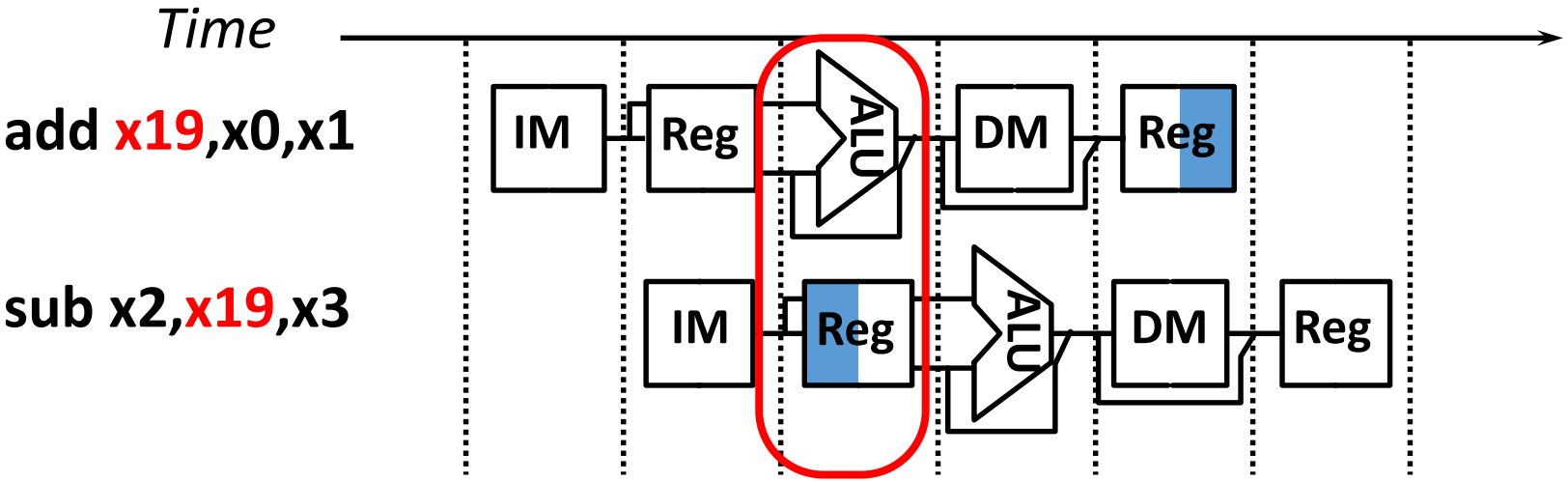


Data Hazards

- An instruction depends on completion of data access to the same register by a previous instruction

```
add x19, x0, x1
sub x2, x19, x3
```

  - Their overlapped execution in pipeline would cause the dependent instruction to fetch wrong data in registers



Dependences and Data Hazards

- Data dependences between instructions may cause data hazards, depending on relative positions in pipe
- RAW (*read after write*):

```
sub x2,x1,x3
and x6,x2,x5
```

Flow dependence:
i2 tries to read operand in the pipeline before i1 writes it
→ get old data instead of new

- WAR (*write after read*):

```
sub x6,x2,x5
and x2,x1,x3
```

Anti-dependence:
i2 tries to write operand in the pipeline before i1 reads it
→ get new data instead of old

- WAW (*write after write*):

```
sub x2,x1,x3
and x2,x6,x5
```

Output dependence:
i2 tries to write operand in the pipeline before i1 writes it
→ leave wrong data

Data Hazards in RISC-V 5-Stage Pipeline

- WAR (write after read) and WAW (write after write) can't happen in RISC-V 5-stage pipeline because:
  - All instructions take 5 stages
  - Reads are always in stage 2
  - Writes are always in stage 5

- Thanks to the regularity in RISC-V ISA design

- But, we still need to resolve RAW hazards
  - RAW hazards may occur between different stages

RAW Hazards among Different Stages

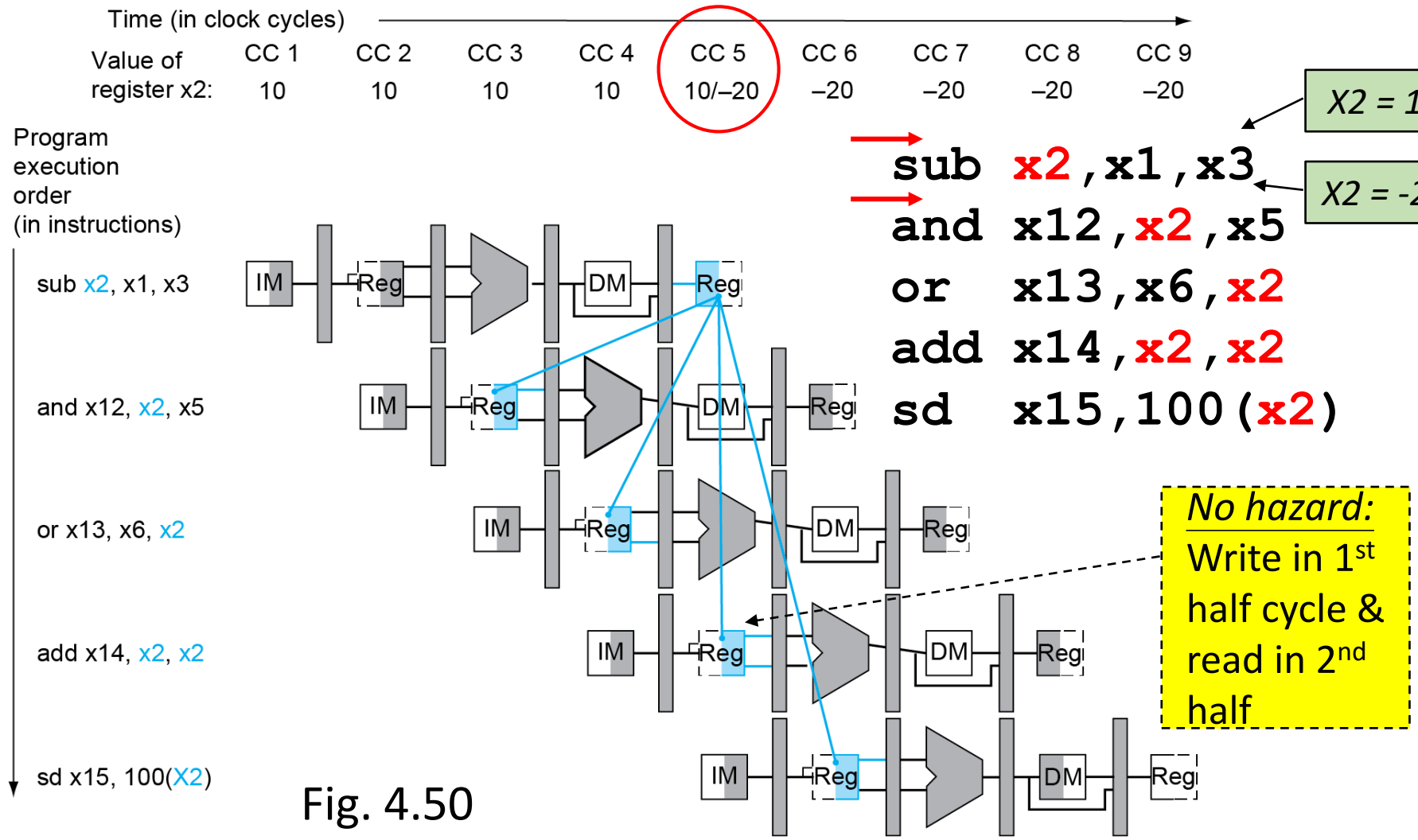
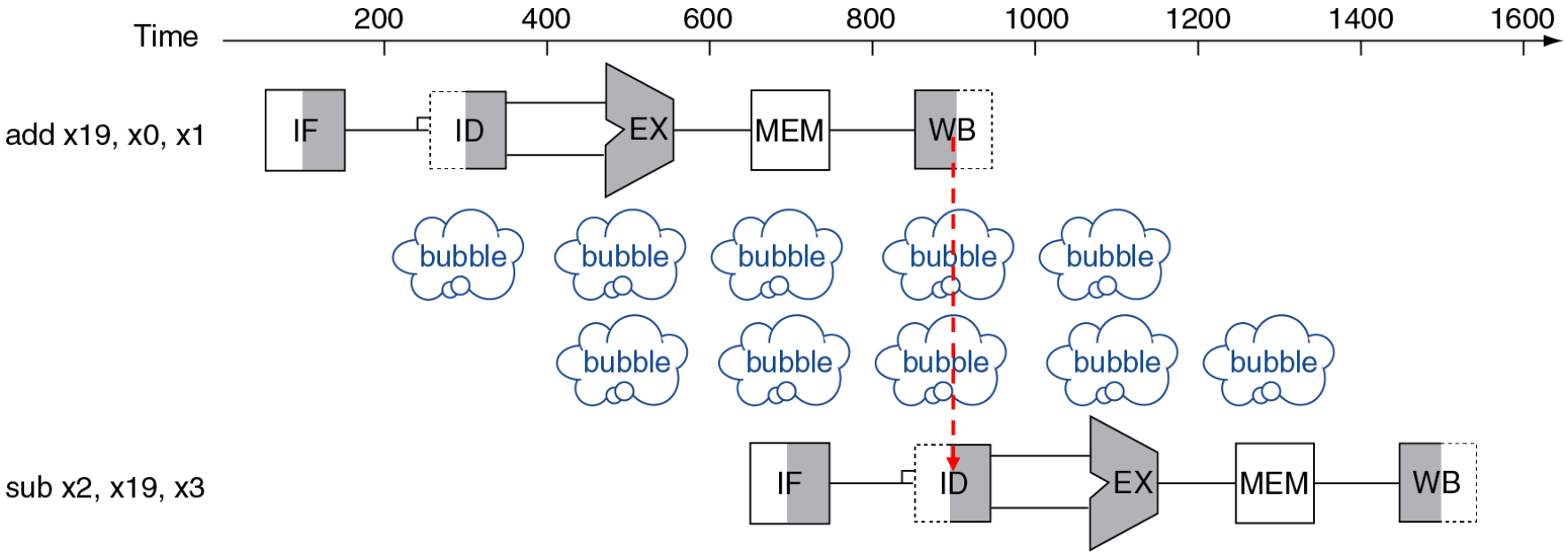


Fig. 4.50

Resolving Data Hazards

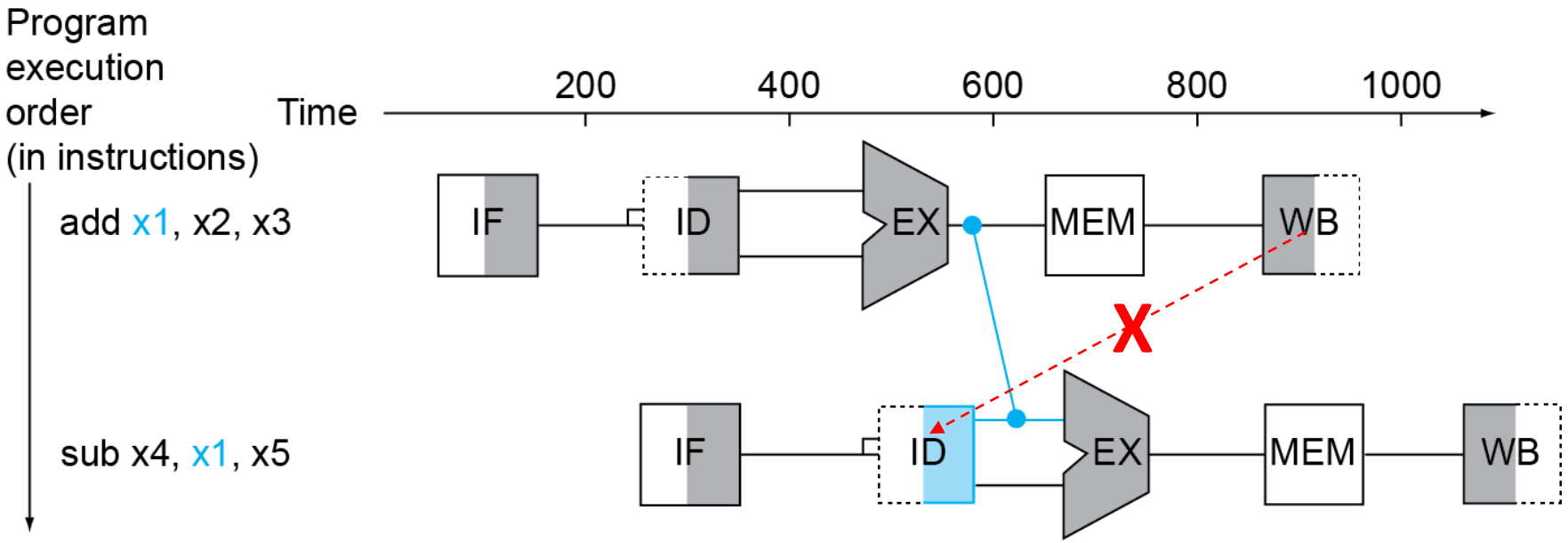
- Let dependent instruction wait until hazard is resolved, i.e., stall i2 until i1 writes back to register at the WB stage → insert pipeline bubbles



- Better: forward data as soon as it is computed

Forwarding (Bypassing)

- Use result when it is computed, e.g., by ALU at the EX stage
  - Don't wait for it to be stored back in register at WB stage
  - Requires extra connections in the datapath



Forwarding between Stages

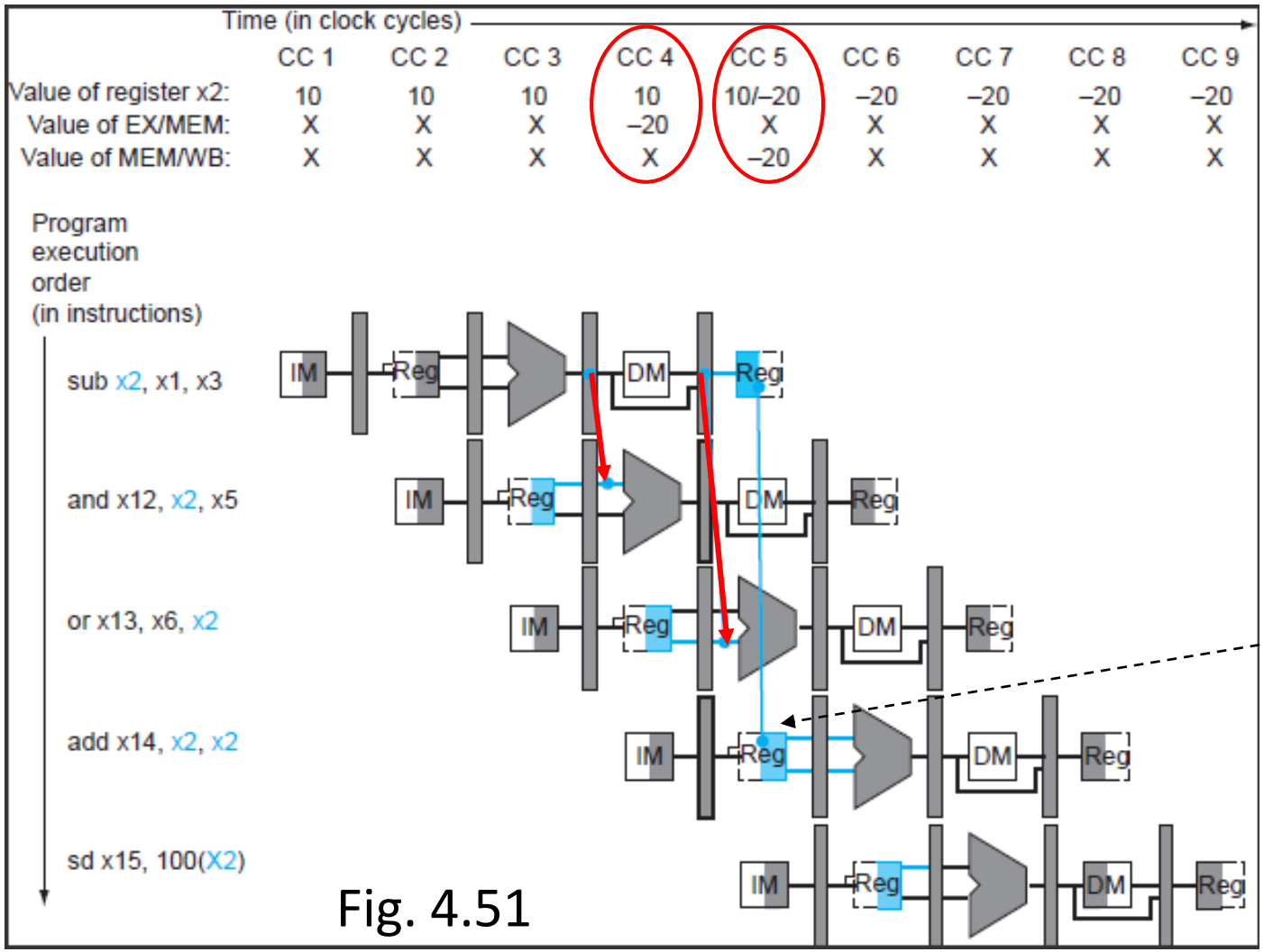


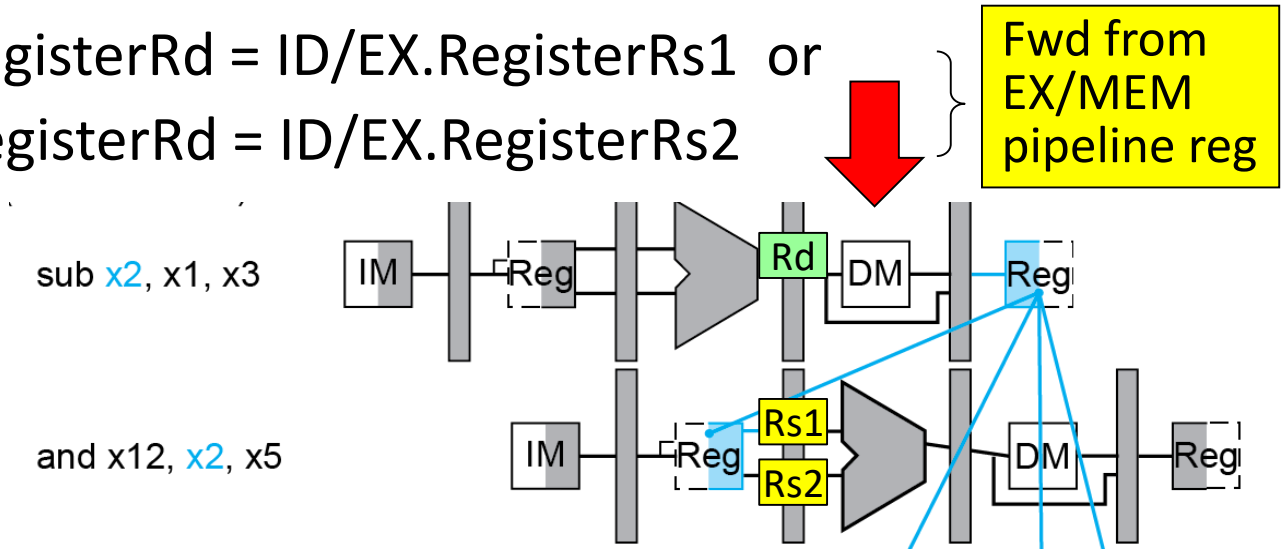
Fig. 4.51

Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage given by
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when

  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1 or

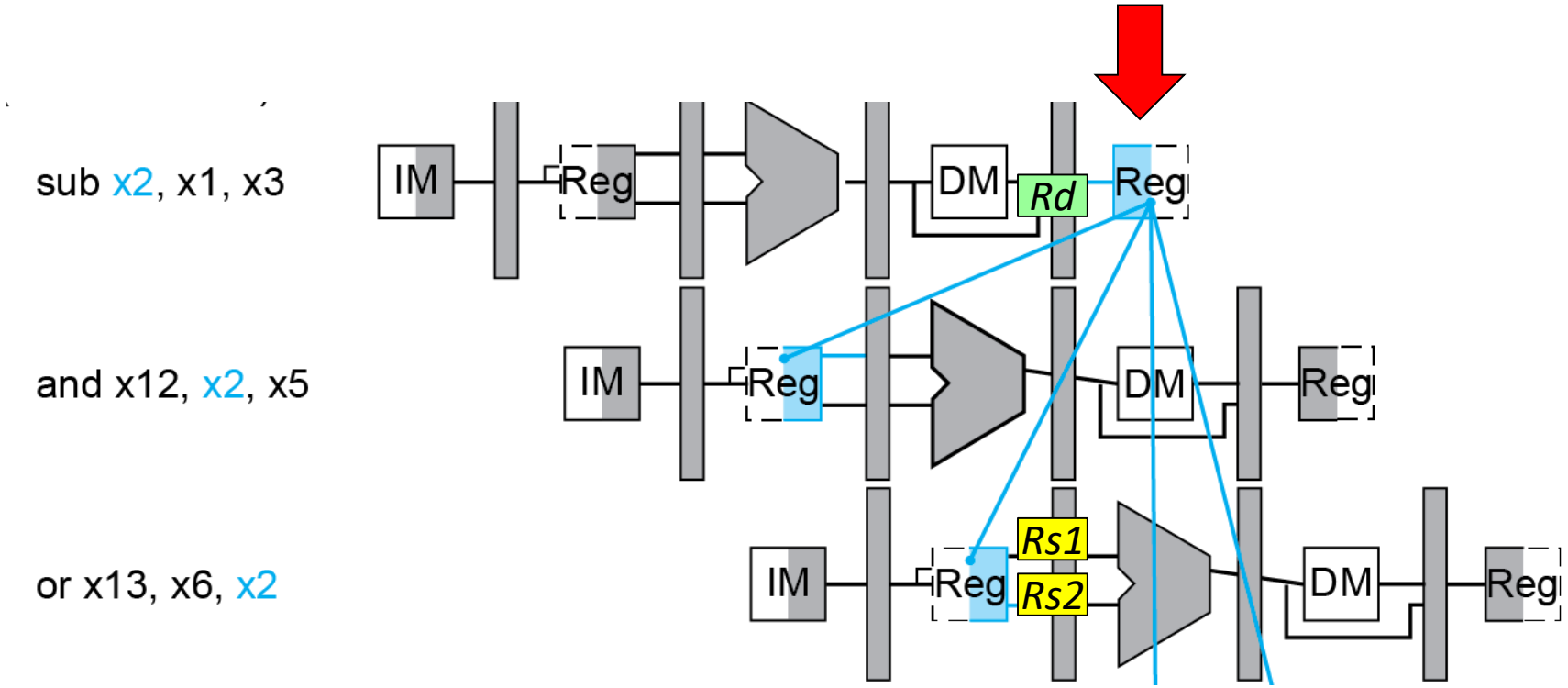
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2



Detecting the Need to Forward

- Data hazards when (cont.)
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

Fwd from MEM/WB pipeline reg



Detecting the Need to Forward

- But data hazard will occur only if the forwarding instruction will write to a register!
  - EX/MEM.RegWrite (for EX/MEM to ID/EX)
  - MEM/WB.RegWrite (for MEM/WB to ID/EX)

- And only if Rd for that instruction is not x0
  - EX/MEM.RegisterRd ≠ 0 (for EX/MEM to ID/EX)
  - MEM/WB.RegisterRd ≠ 0 (for MEM/WB to ID/EX)

Forwarding Paths

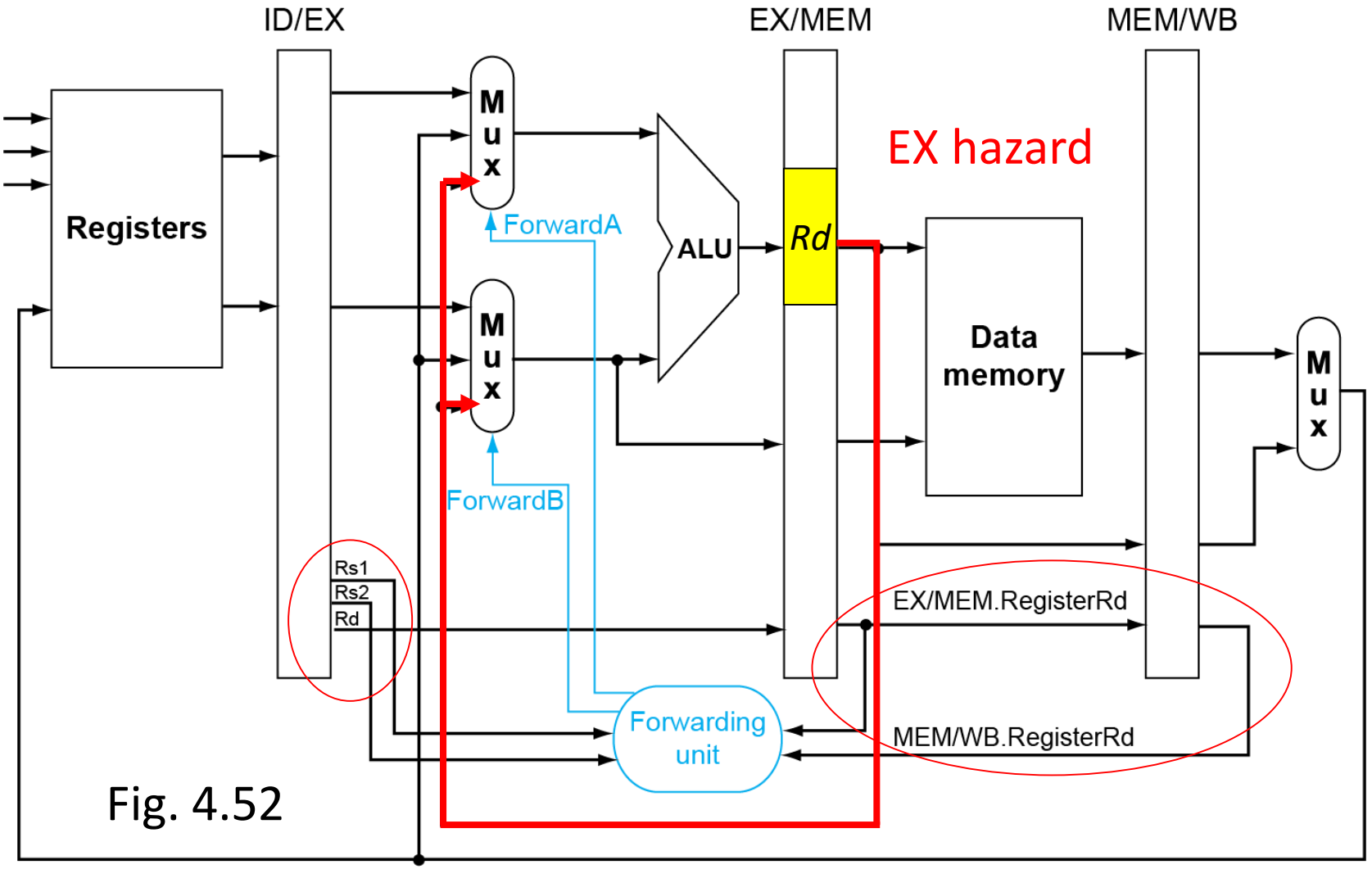


Fig. 4.52

Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
    then ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
    then ForwardB = 10

Forwarding Paths

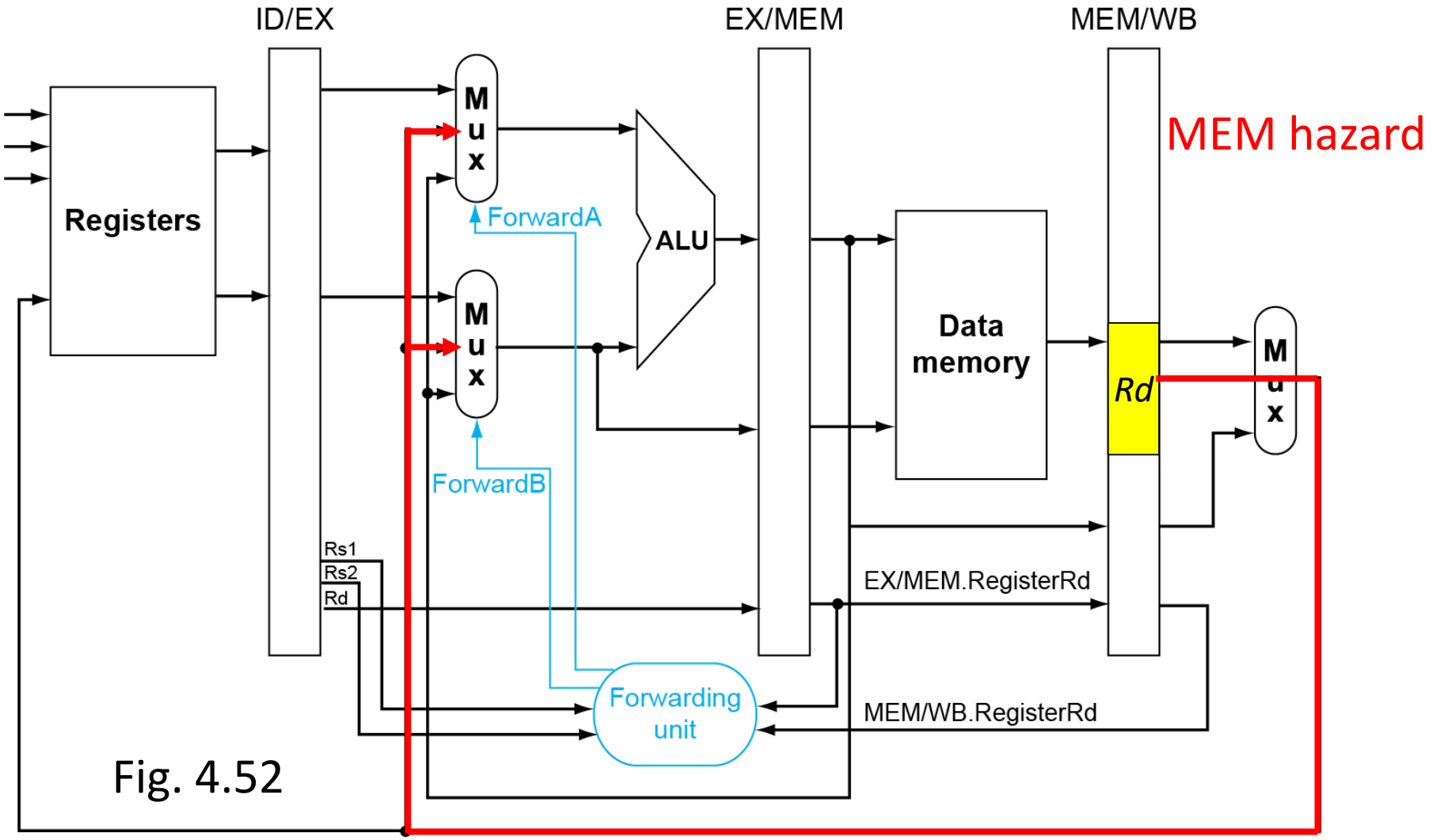


Fig. 4.52

Forwarding Conditions

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
    then ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
    then ForwardB = 01

- But there is a problem …

Double Data Hazards

- Consider the sequence:

```
add x1,x1,x2
sub x1,x1,x3
and x1,x1,x4
```

- Both hazards occur and both **add** and **sub** want to forward to **and**

  ➔ use the most recent one, i.e., **sub**

- Thus, need to revise MEM hazard condition
  - Forward only 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.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
    then 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.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
    then ForwardB = 01

Datapath with Forwarding

Why between MEM-EX & WB-EX?

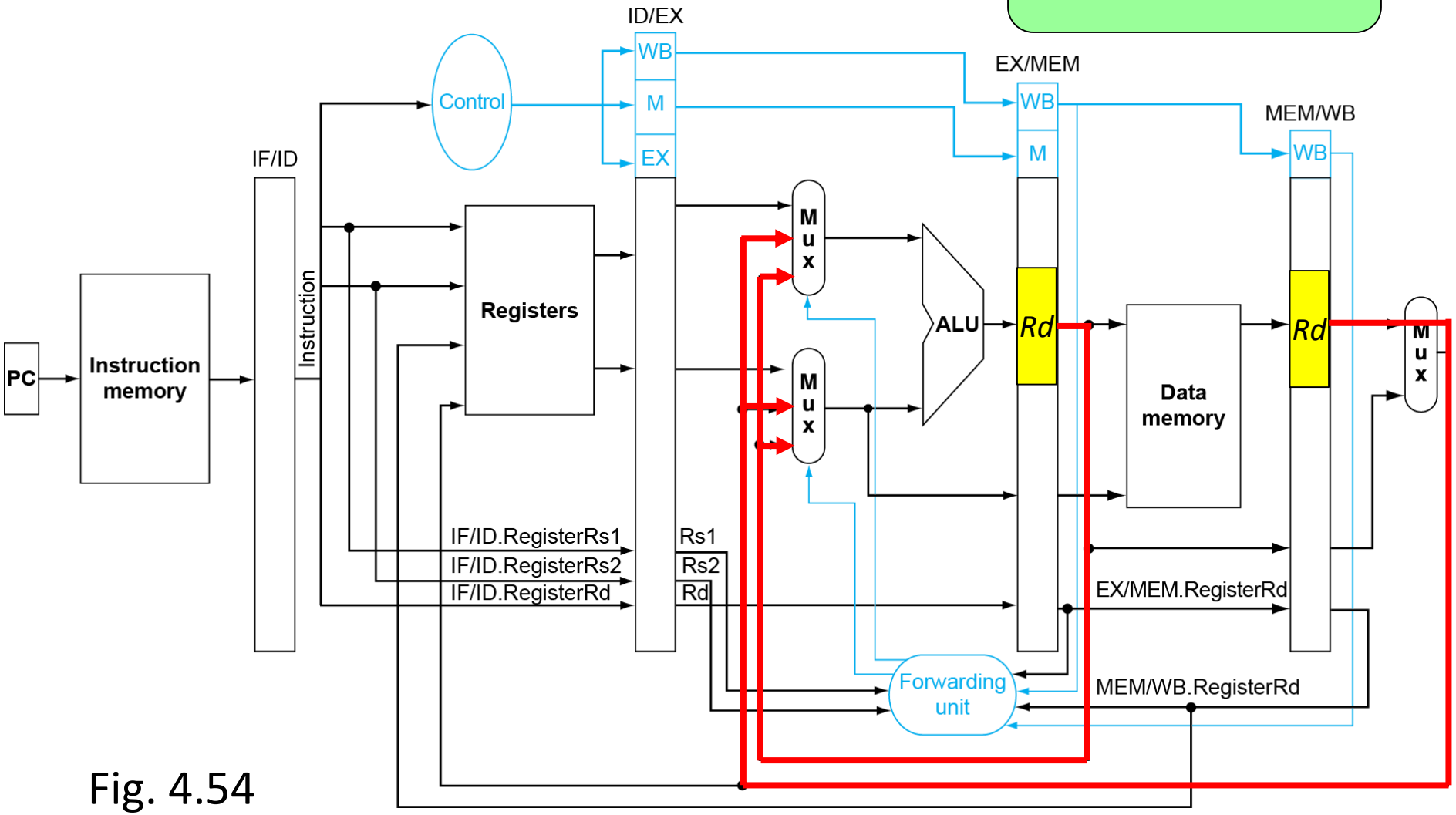


Fig. 4.54

Datapath with Forwarding

Why not between EX-ID & MEM-ID?

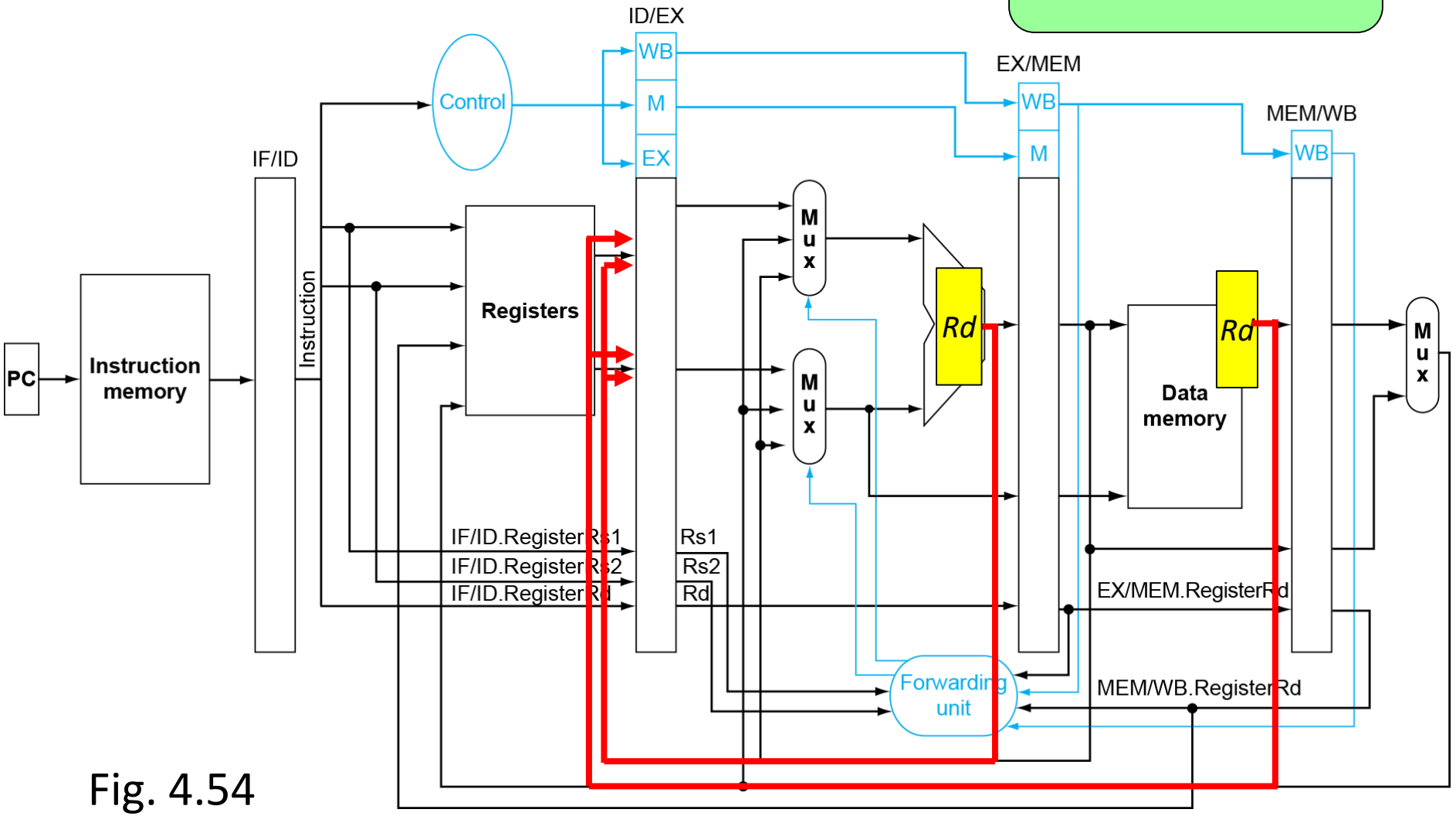


Fig. 4.54

Can't Always Forward

- `ld` can still cause a hazard: (*load-use* data hazard)
  - If it is followed by an instruction to read the loaded reg.

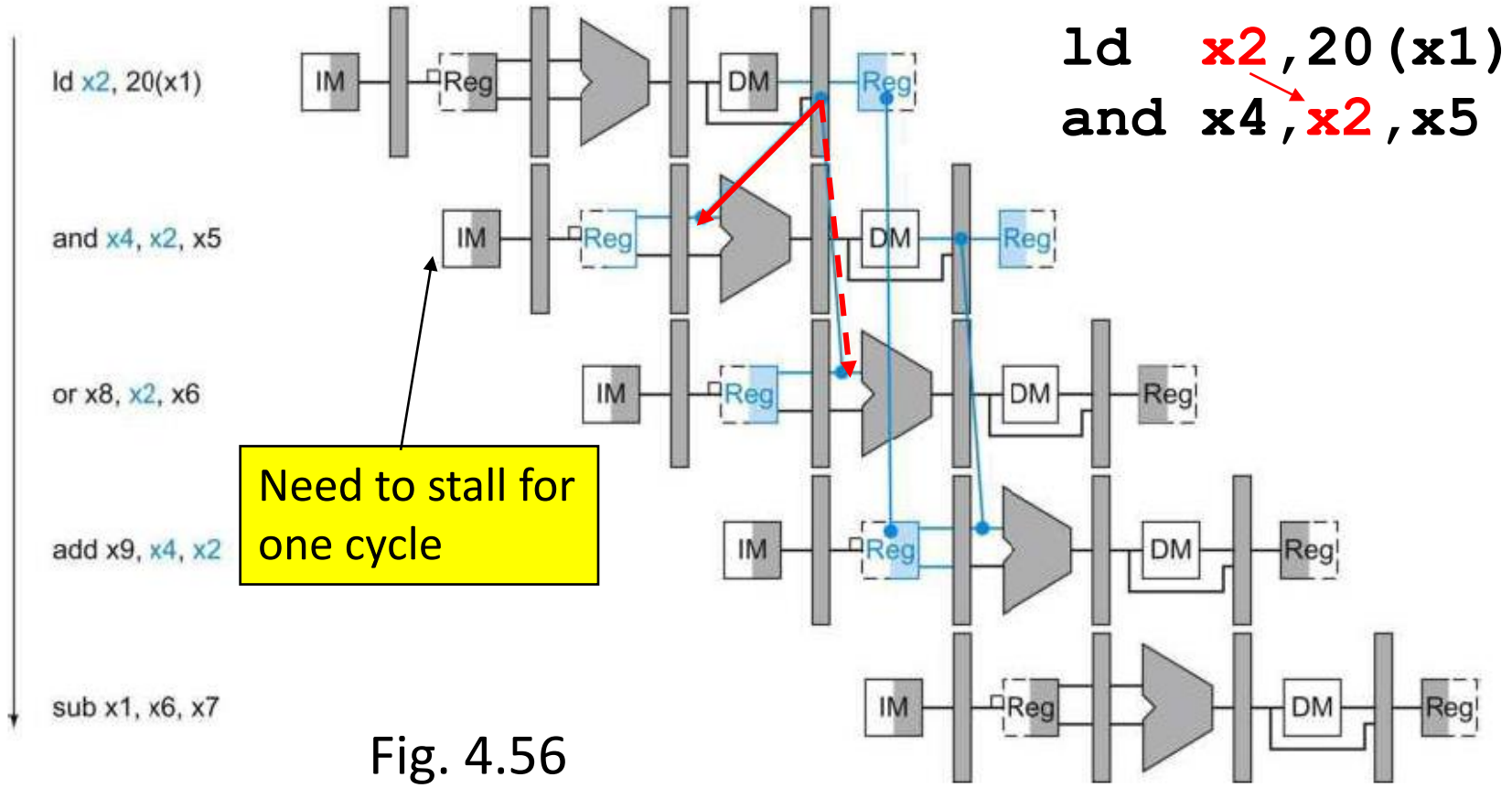


Fig. 4.56

Load-Use Hazard Detection

- Check when the using instruction is decoded in the ID stage (`ld` is in the EX 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))
  - Note that ID/EX.MemRead=1 indicates a load instruction
- If detected, stall and insert a bubble

How to Stall an Instruction in the Pipeline?

- Stall pipeline by keeping instructions in same stage and inserting an NOP instead

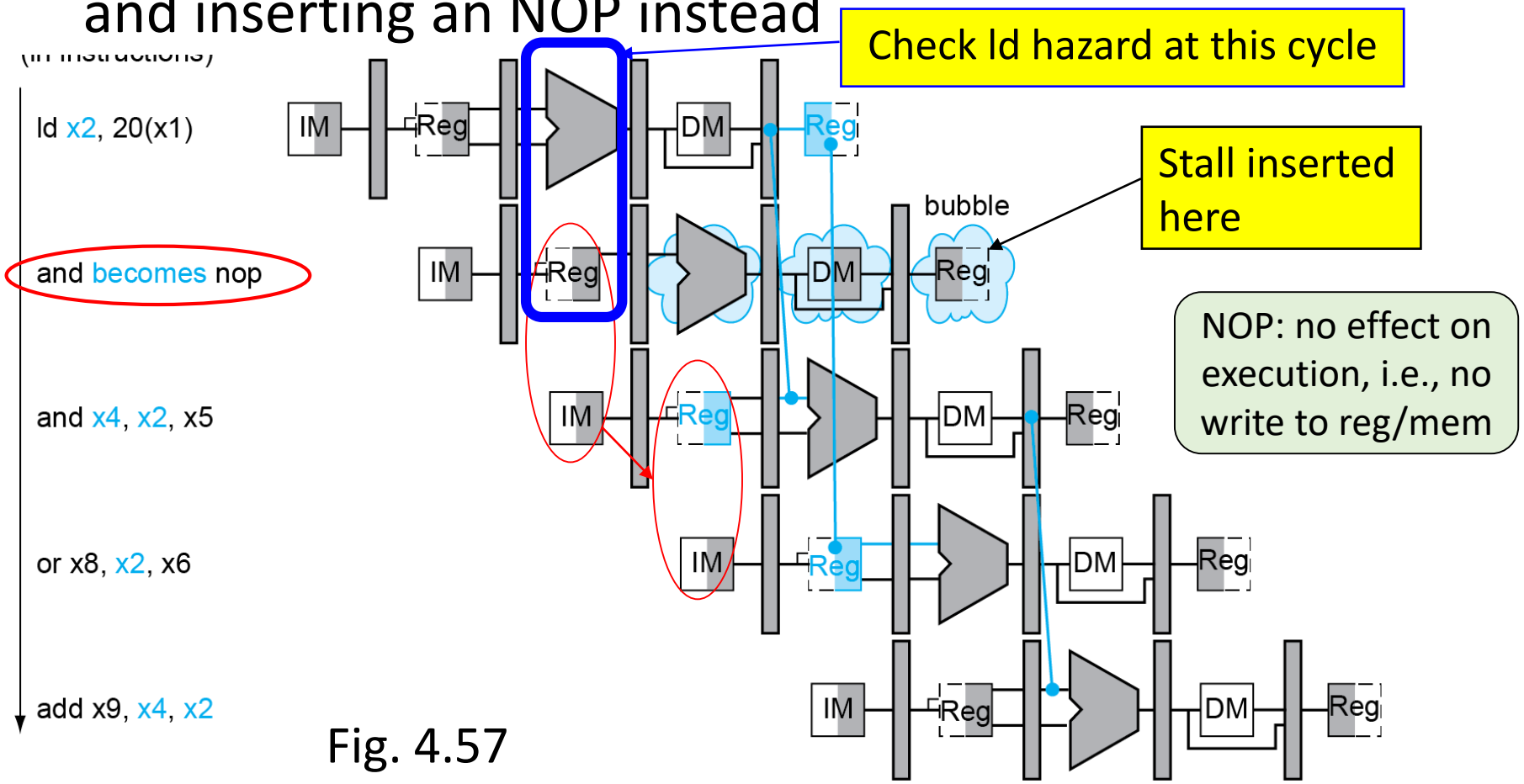


Fig. 4.57

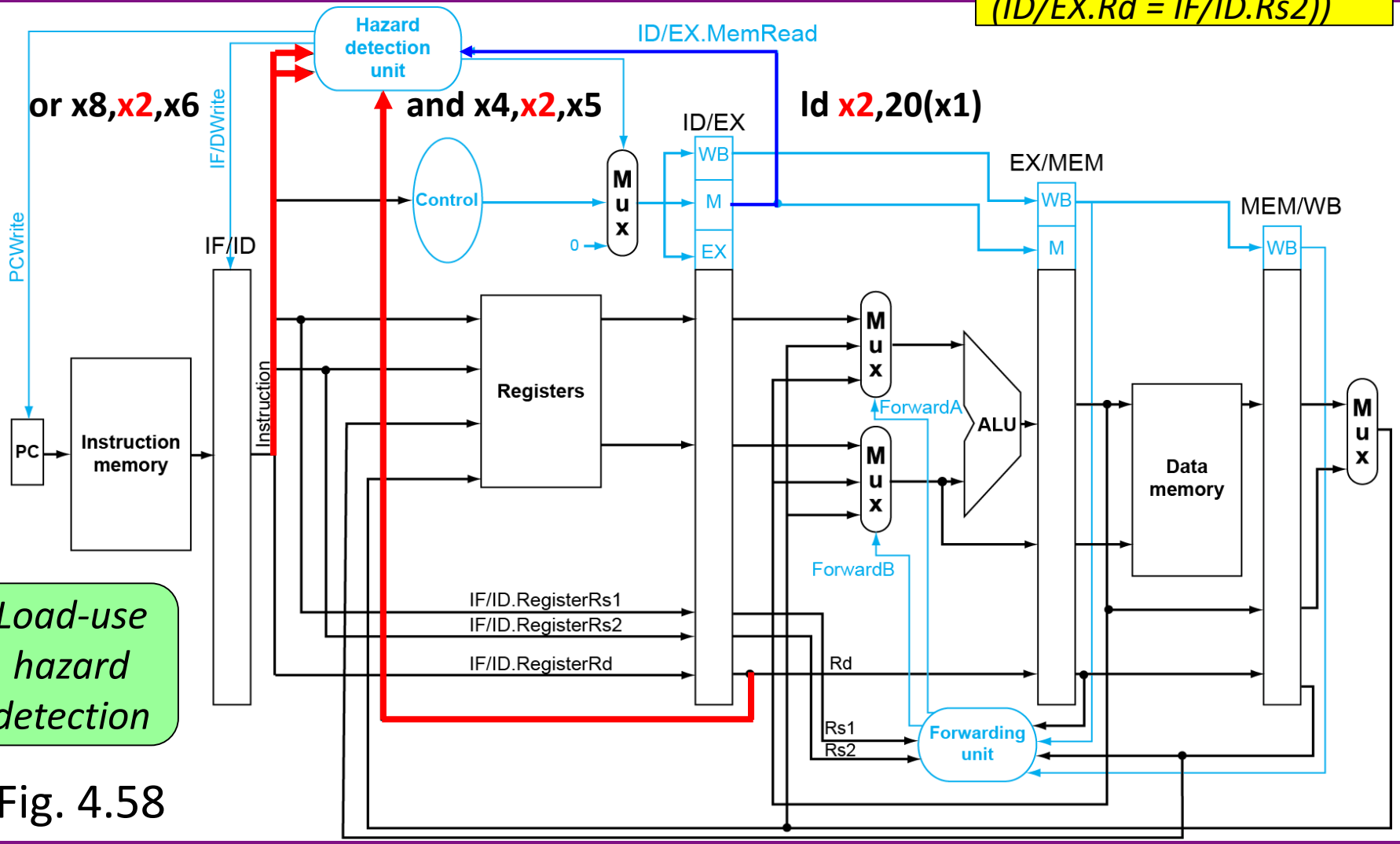
How to Stall the Pipeline?

- Stall instructions in IF and ID for one cycle:
  Prevent update of PC and IF/ID register
  - The instruction in ID stage is decoded again
  - The following instruction is fetched again
- Instructions in EX, MEM and WB are not affected
  - Let **`ld`** enter MEM in next cycle to read data and forward it to the EX stage
- What to move into EX for the next cycle?
  Force control signals in ID/EX register to 0
  - Turn EX into NOP (no-operation) → in following cycles, control signals of this NOP in MEM, WB are deasserted in sequence, no registers or memories are written

Hardware inserts an NOP instruction dynamically

Datapath with Stalling Unit

ID/EX.MemRead and ((ID/EX.Rd = IF/ID.Rs1) or (ID/EX.Rd = IF/ID.Rs2))



Load-use hazard detection

Fig. 4.58

Datapath with Stalling Unit

Handling hazard

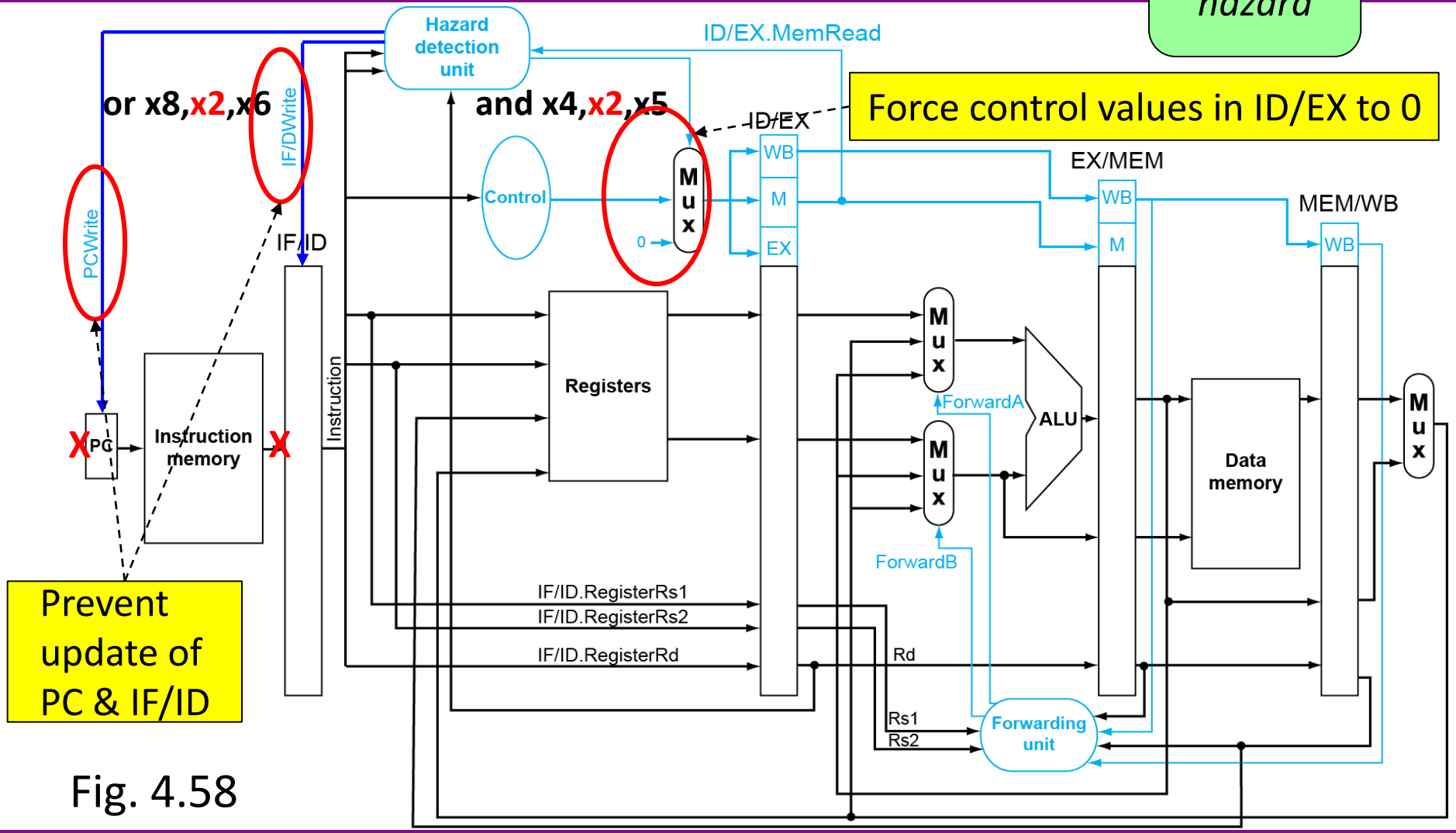


Fig. 4.58

Datapath with Stalling Unit

Handling hazard

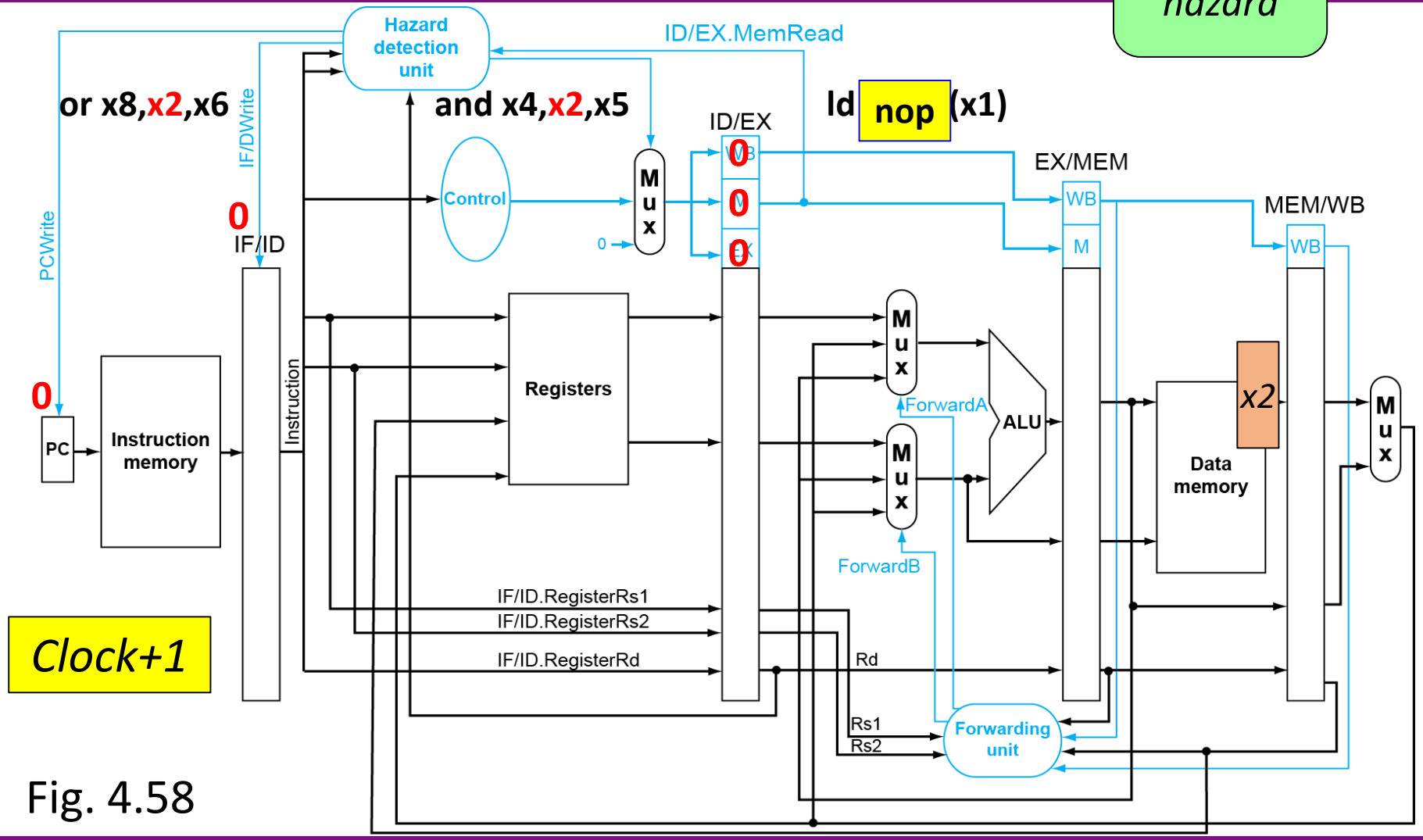


Fig. 4.58

Datapath with Stalling Unit

Handling hazard

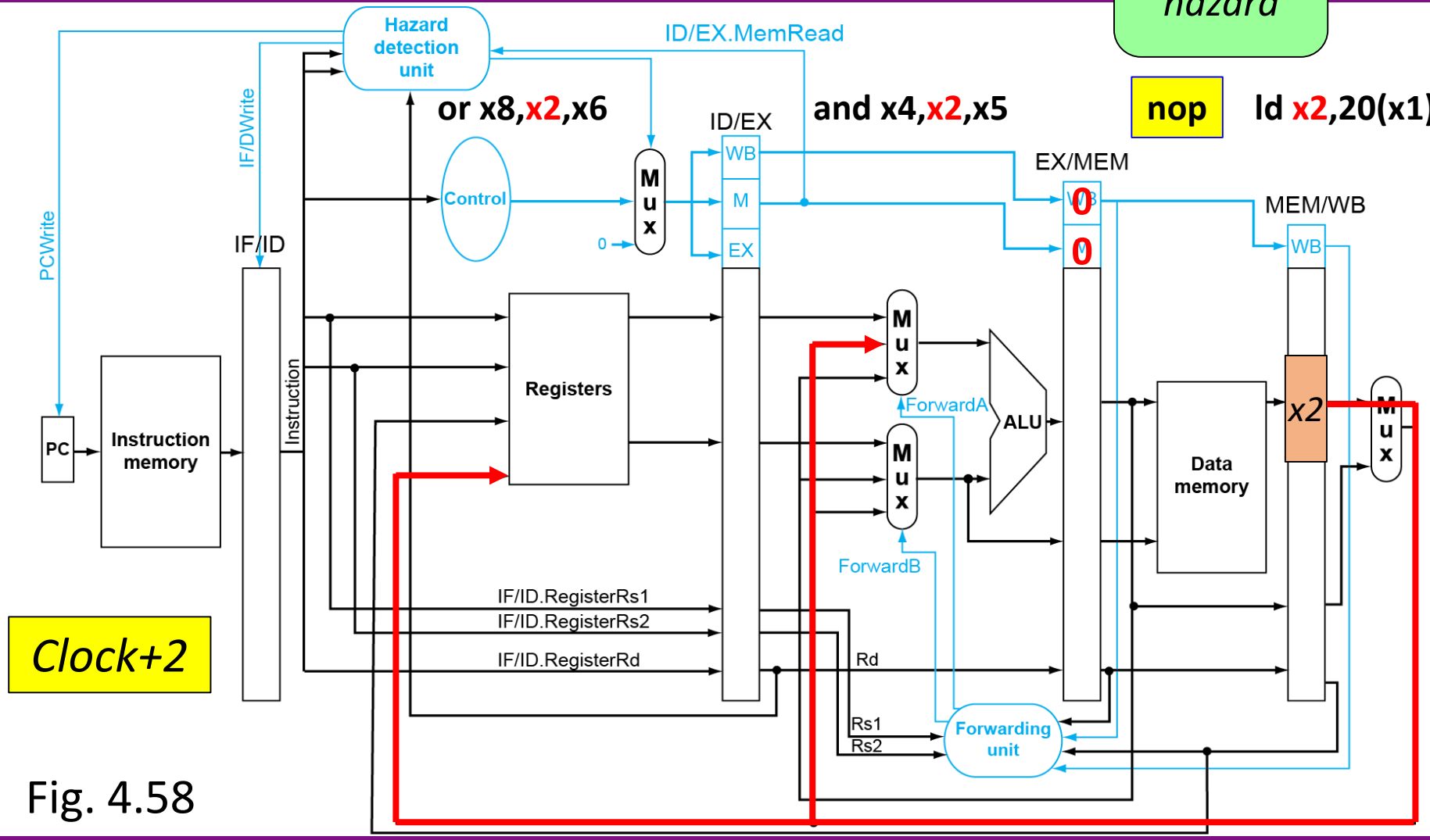
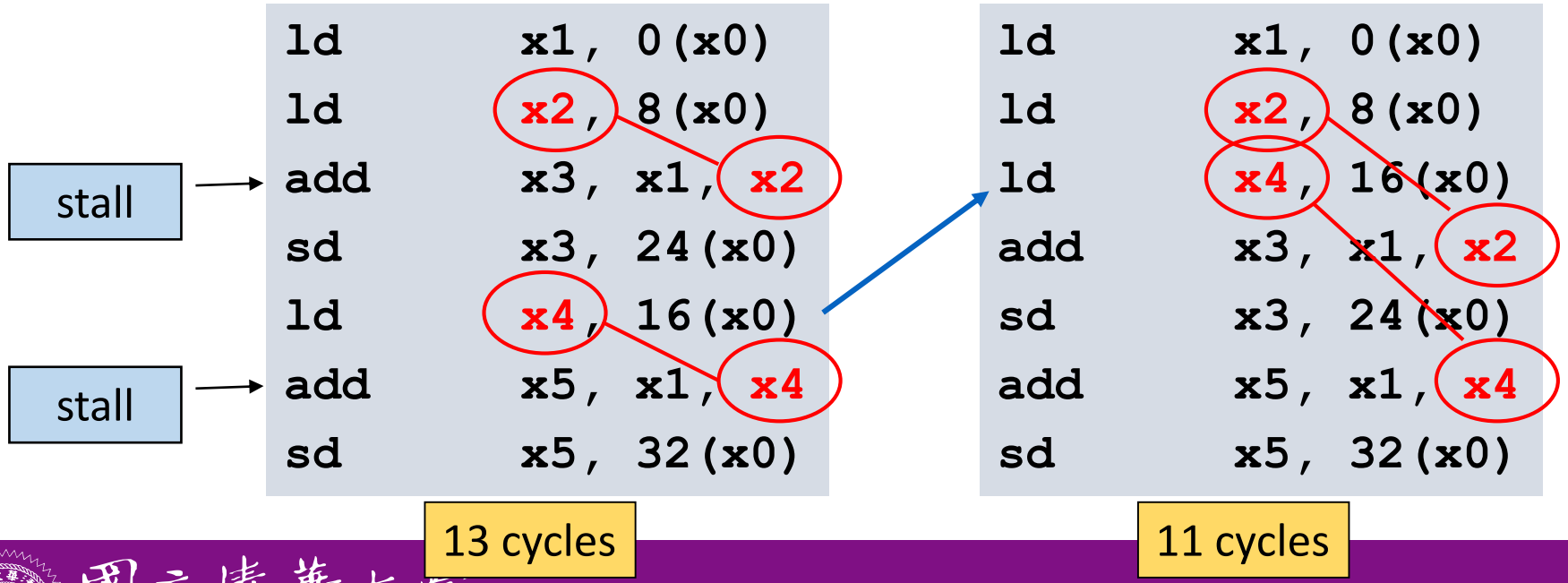


Fig. 4.58

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
- C code for `A = B + E; C = B + F;`



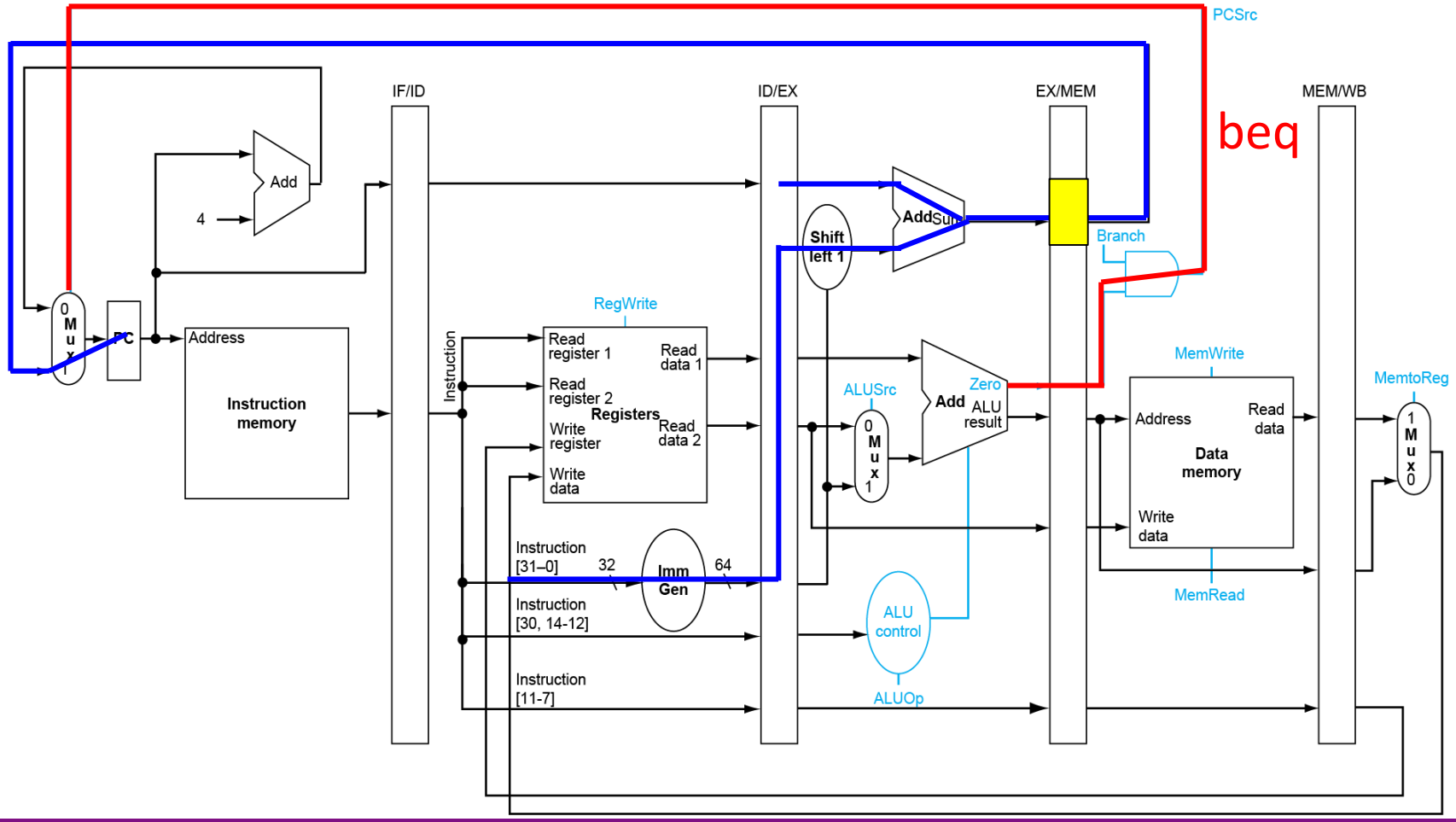
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
  - Need to wait until branch outcome determined before fetching next instruction

- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - How early can we do it in the pipeline?

Control Hazards

- beq determines whether to branch at the MEM stage



Control Hazards

- When beq decides to branch, the following 3 instructions are already in pipeline!

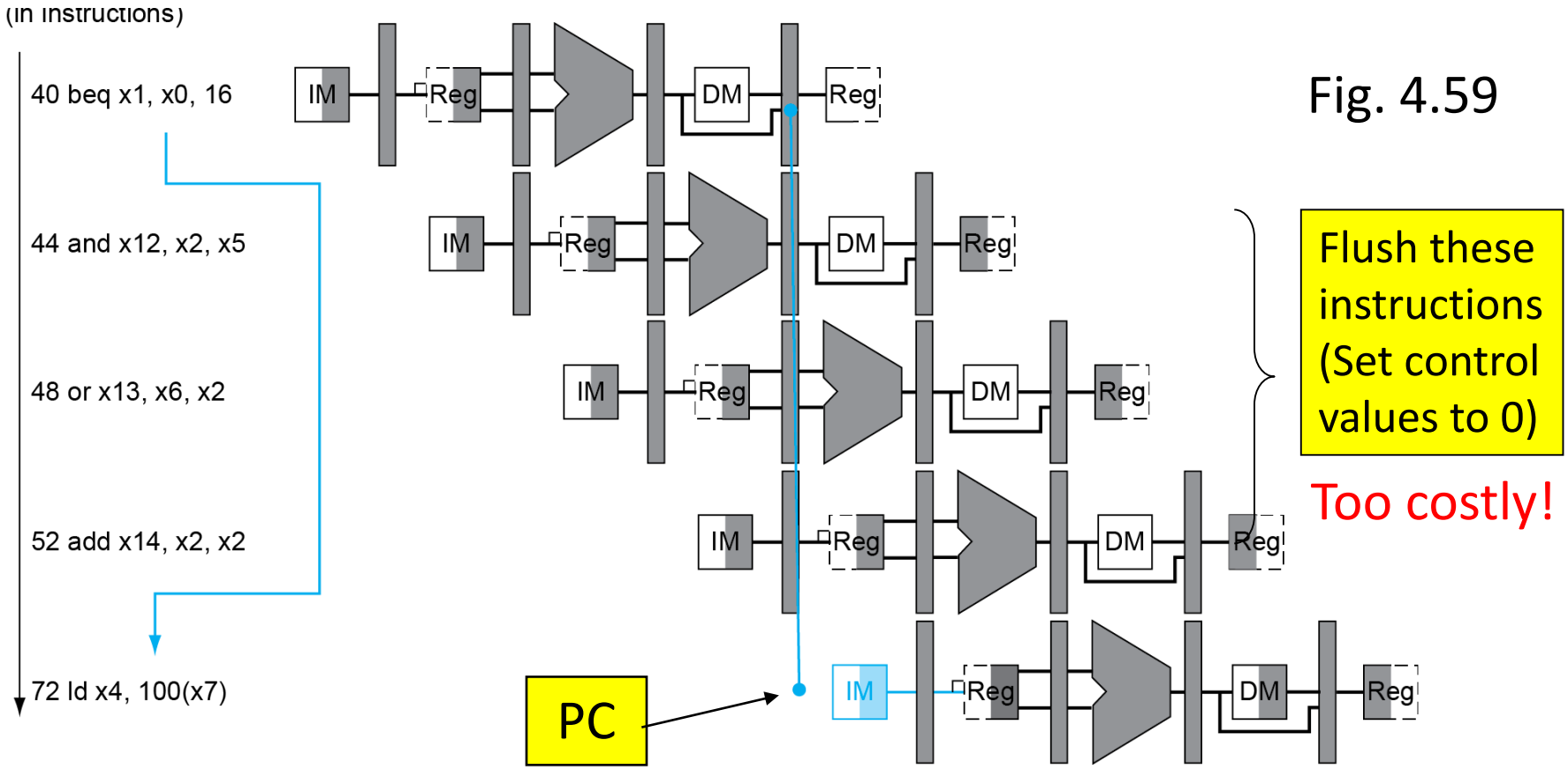


Fig. 4.59

Reducing Branch Delay

- Add hardware to determine outcome at ID stage
  - Check branch equality at ID (using XOR)
  - Target address adder at ID stage
  - Will still fetch the next sequential instruction

    ➔ add a control signal, IF.Flush, to zero instruction field of IF/ID (making the following instruction an NOP)

Not setting control signals to 0!

- Example: branch taken

```
36:   sub   x10, x4, x8
40:   beq   x1,  x3, 16   // 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

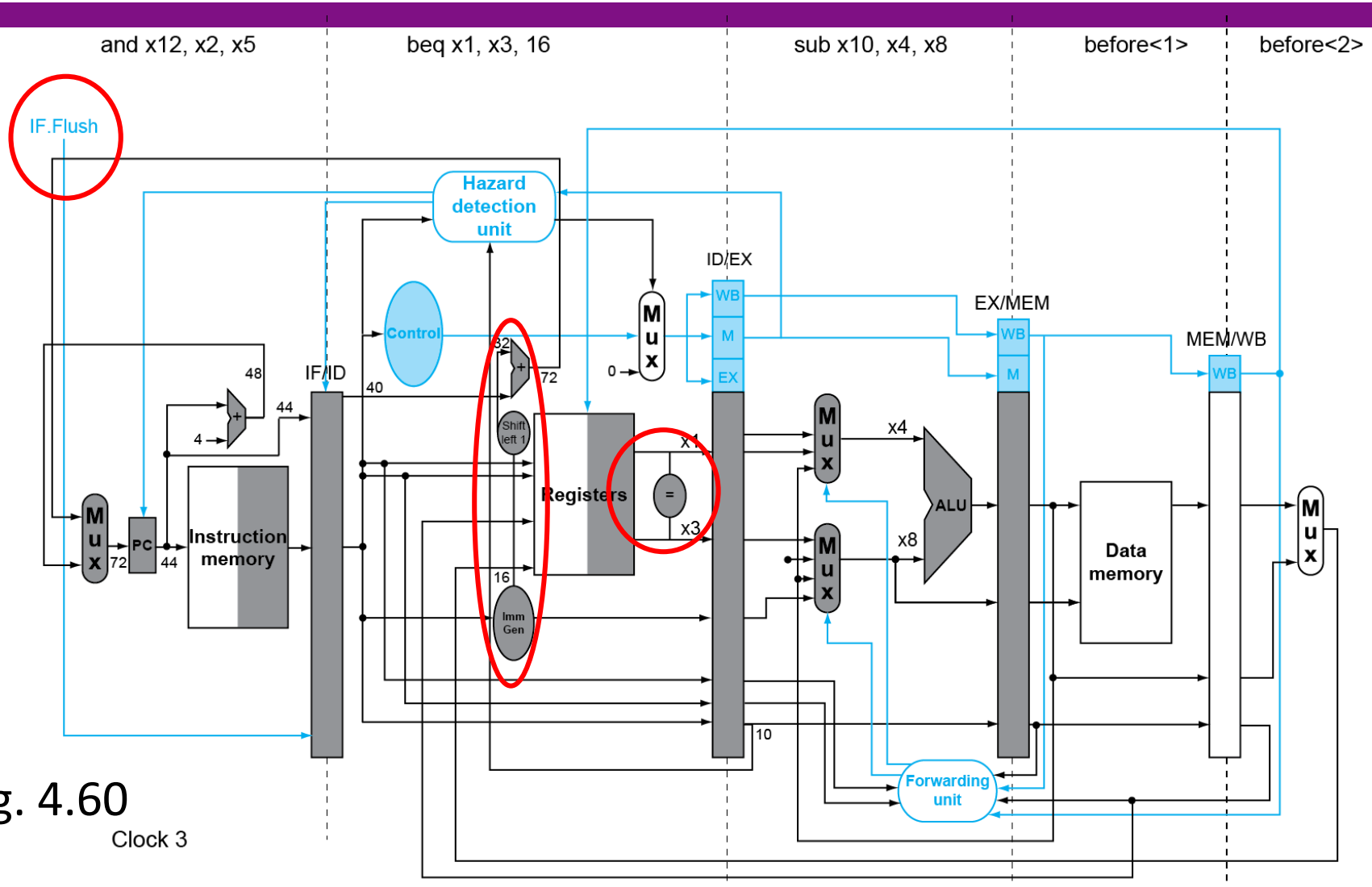


Fig. 4.60

Clock 3

Example: Branch Taken

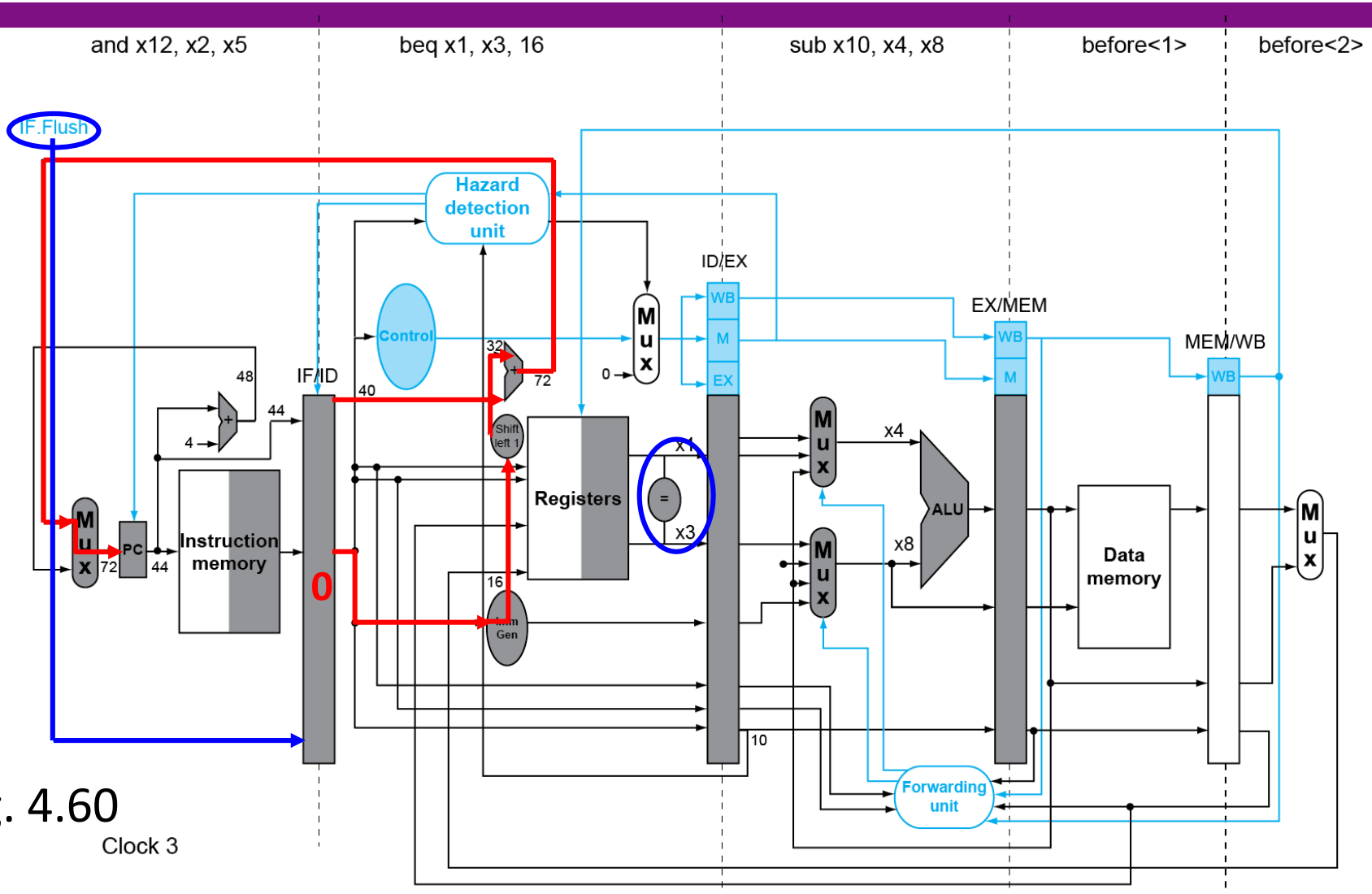


Fig. 4.60

Clock 3

Example: Branch Taken

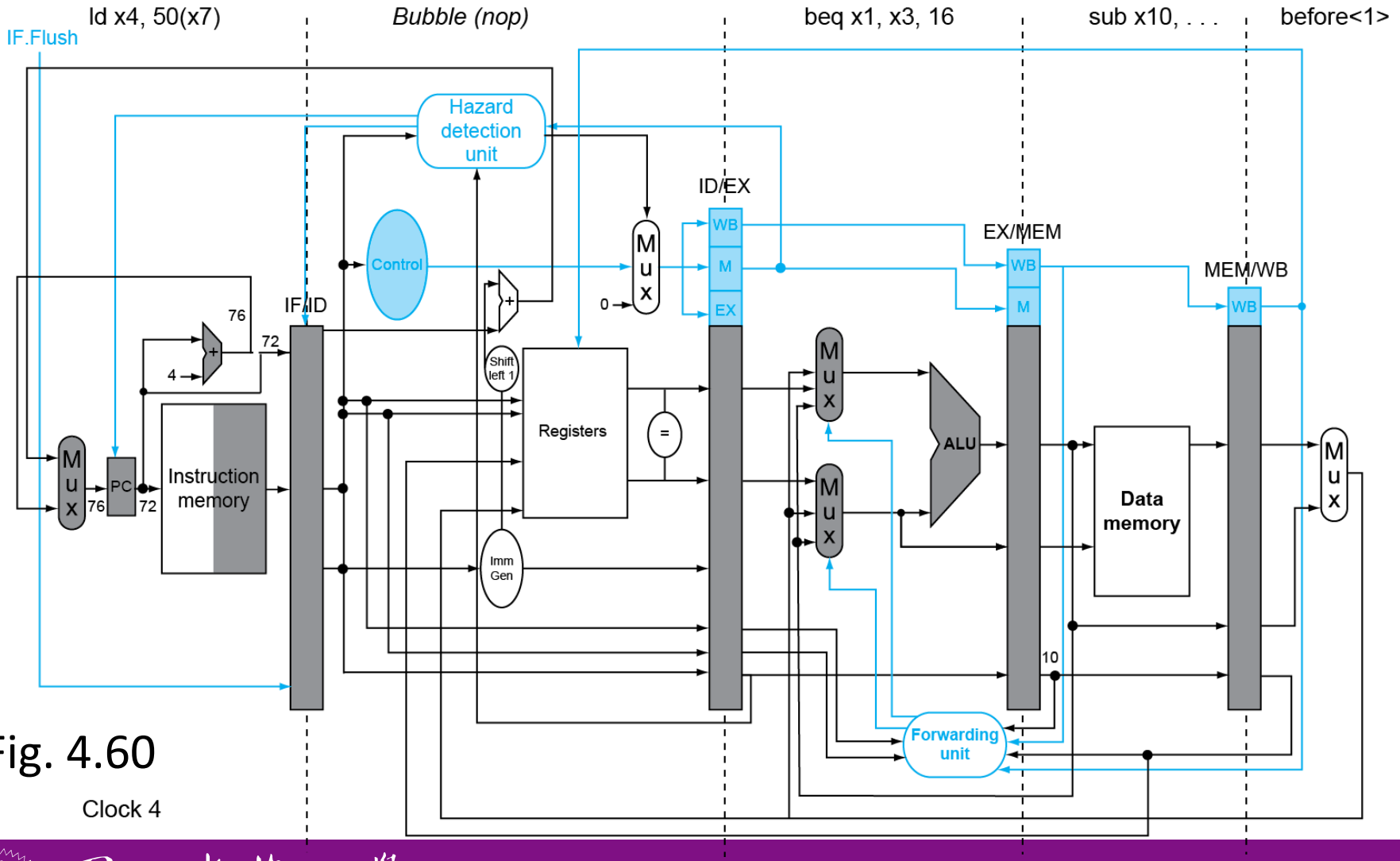
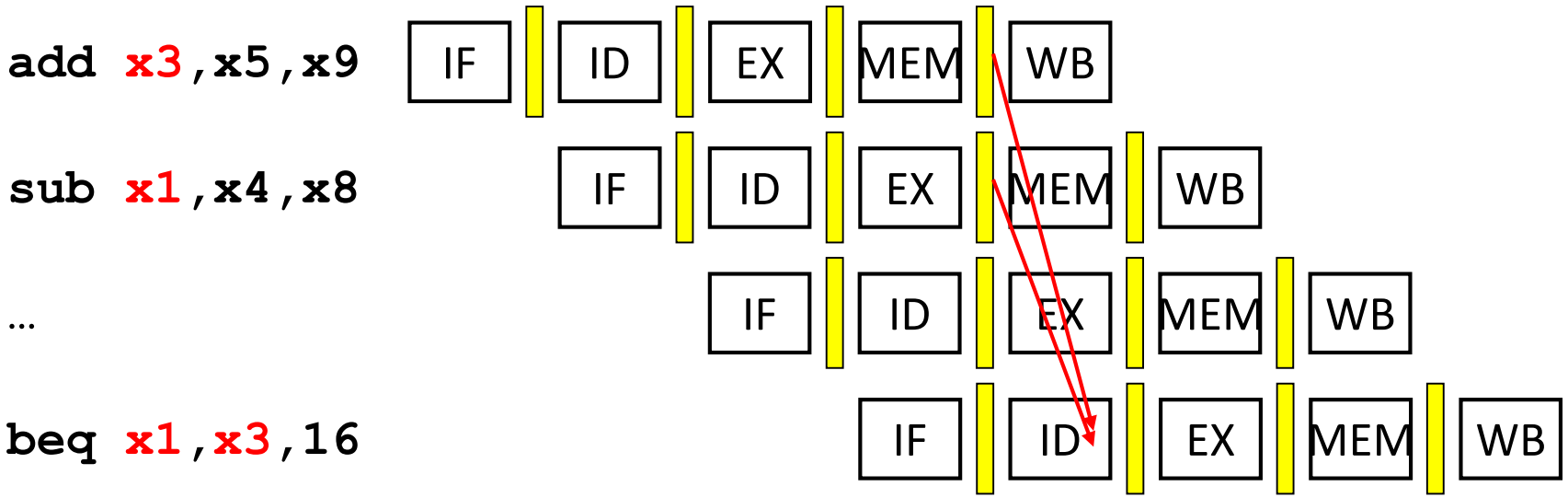


Fig. 4.60

Clock 4

Data Hazards for Branches

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



- Can resolve using forwarding (how?)
  - Need to forward to ID stage in addition to EX stage!

Data Hazards for Branches

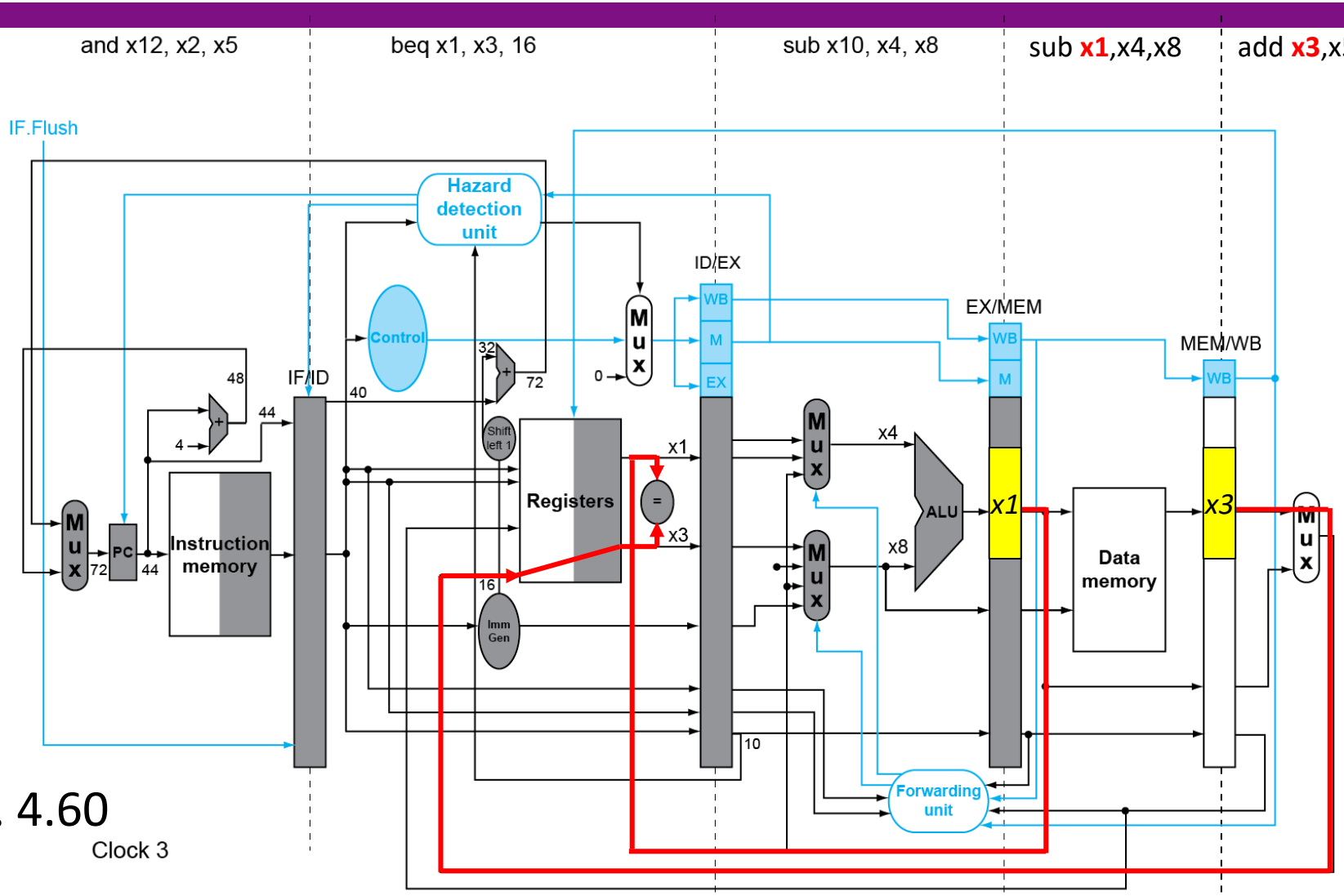
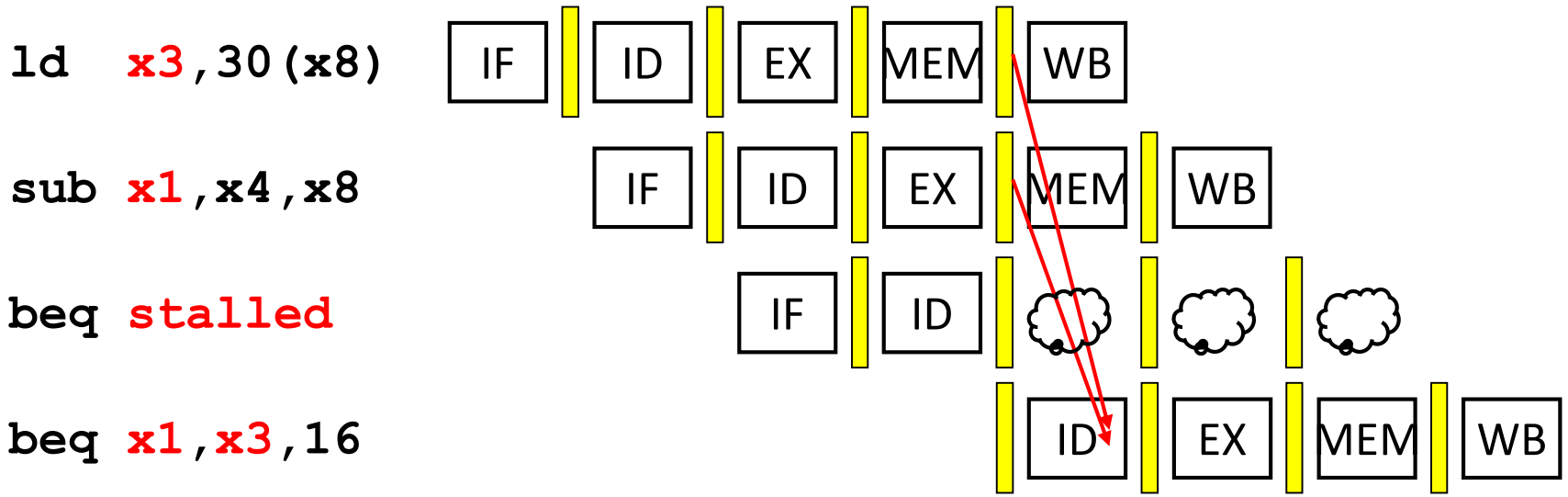


Fig. 4.60

Clock 3

Data Hazards for Branches

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



Data Hazards for Branches

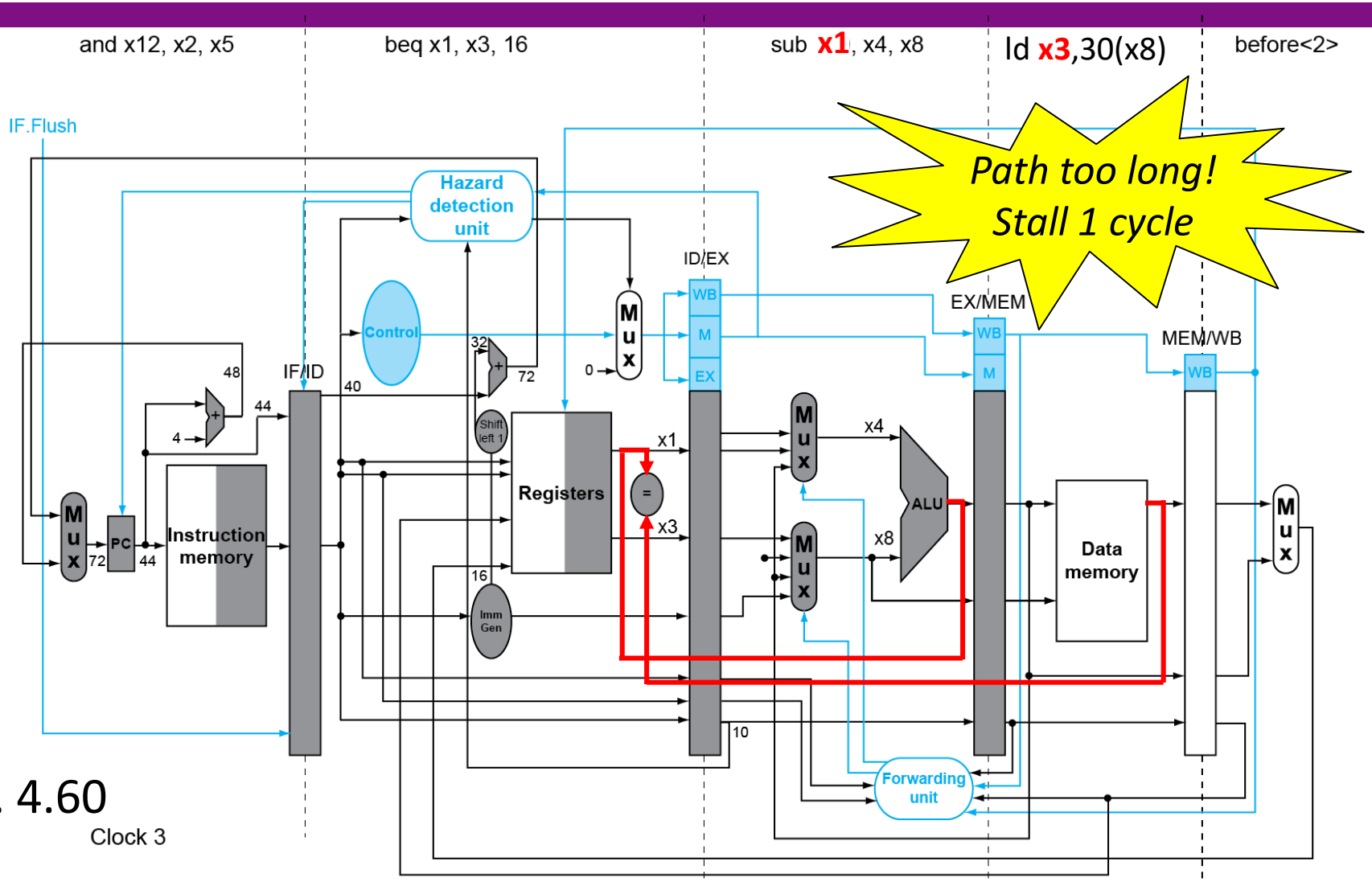


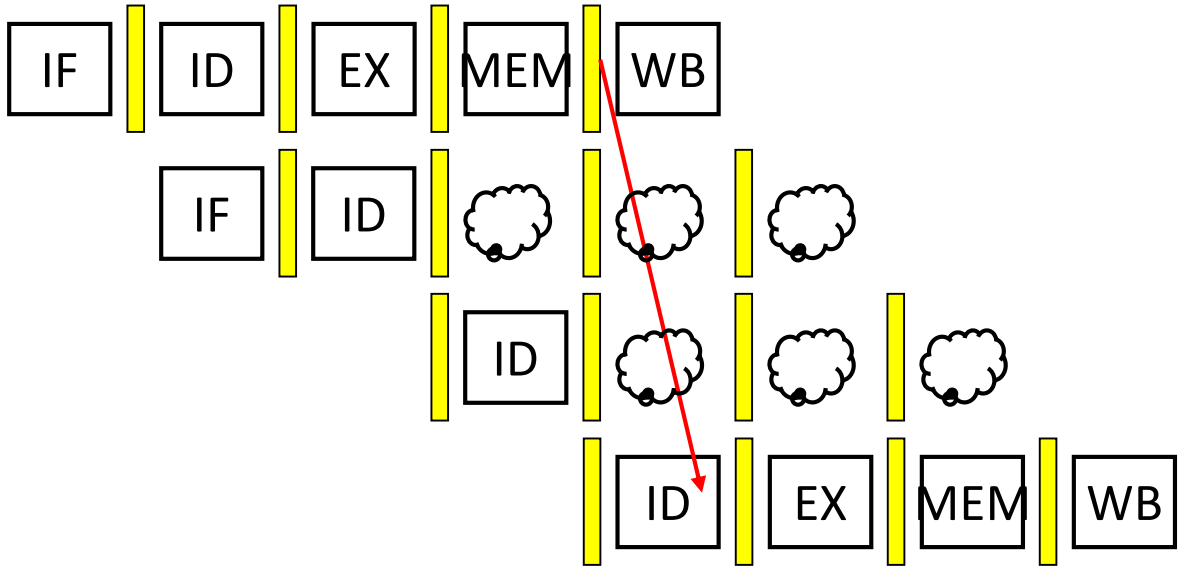
Fig. 4.60

Clock 3

Data Hazards for Branches

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

```
ld   x1,30(x8)
beq  stalled
beq  stalled
beq  x1,x0,16
```



- How to check the conditions and stall pipeline stage?

Handling Control Hazards

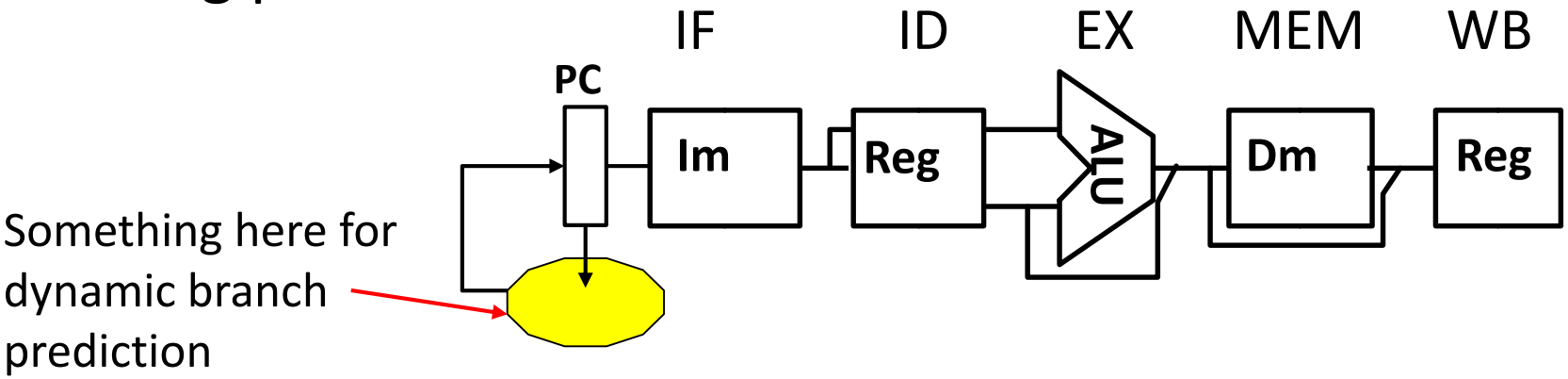
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Option 1: dynamic branch prediction
  - Predict outcome of branch
  - Only stall if prediction is wrong
- Option 2: compiler rescheduling
  - e.g., move independent instructions before the branch to after the branch, if effects of the branch will be delayed

Branch Prediction

- Static branch prediction
  - Based on typical branch behavior, e.g., loop, if-statement
    - Predict backward branches taken
    - Predict forward branches not taken
  - In RISC-V pipeline, always predict branches not taken
    - Fetch instruction after branch, with no delay
  - Need to add hardware for flushing instructions if wrong
- Dynamic branch prediction
  - Hardware tracks actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
  - Ideal: branches handled in IF and fetch right way next cycle

Dynamic Branch Prediction

- In IF stage, need to know/predict:
  - The instruction that is fetching in this cycle is a branch
  - If so, its branch direction
  - If so, its branch target
- If everything works smoothly, we can fetch in the next cycle the instructions from predicted branch direction
- The big picture

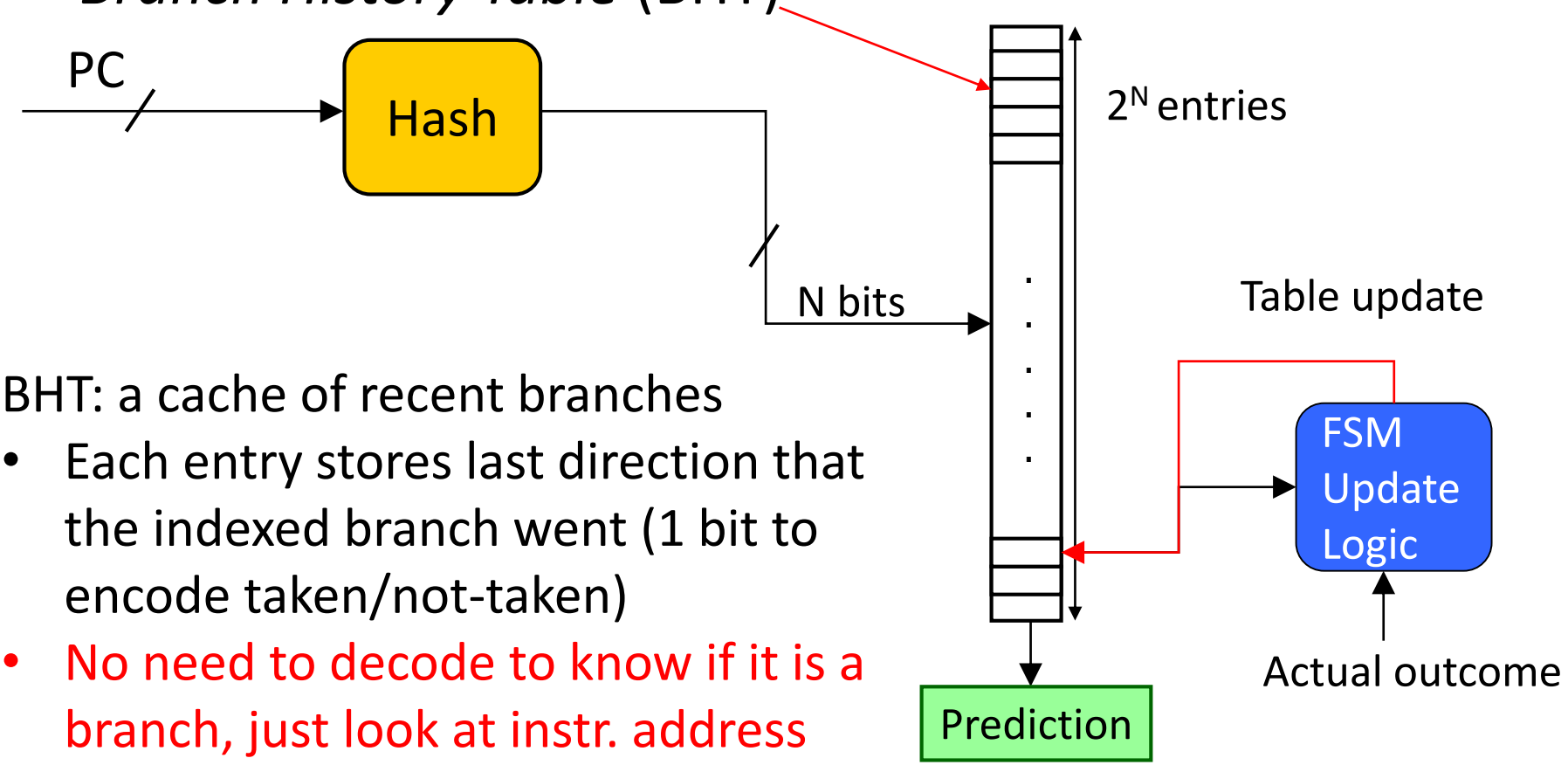


Dynamic Branch Prediction

- How do I know the instruction that is fetching in this cycle is a branch?
  - Easy if have seen it before → track its instruction address
- Branch direction:
  - *Branch prediction buffer* (*branch history table*), indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken) in the table
  - To execute a branch
    - Check table and expect the same outcome
    - Start fetching from the predicted fall-through or target
    - If wrong, flush pipeline and flip prediction

Predicting Branch Direction

- Predict branch based on past history of branch
- *Branch History Table* (BHT)



BHT: a cache of recent branches

- Each entry stores last direction that the indexed branch went (1 bit to encode taken/not-taken)
- No need to decode to know if it is a branch, just look at instr. address

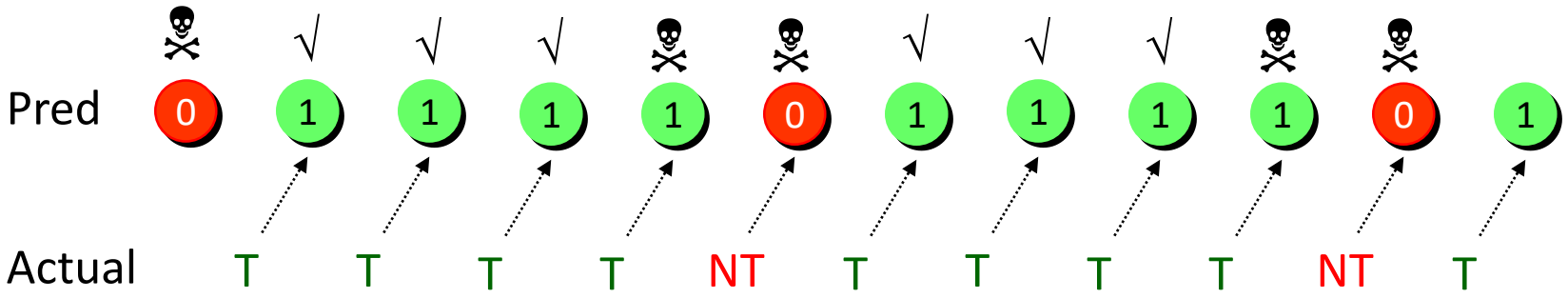
Problem with 1-Bit Predictor

- Inner loop branches always mispredicted twice!

```
outer:  …
        …
inner:  …
        …
        beq …, …, inner
        …
        beq …, …, outer
```

4 iterations

Mispredict as taken on the last iteration of the inner loop and then mispredict as not taken on the first iteration of the inner loop next time around



Solution: 2-Bit Predictor

- Change prediction on two successive mispredictions

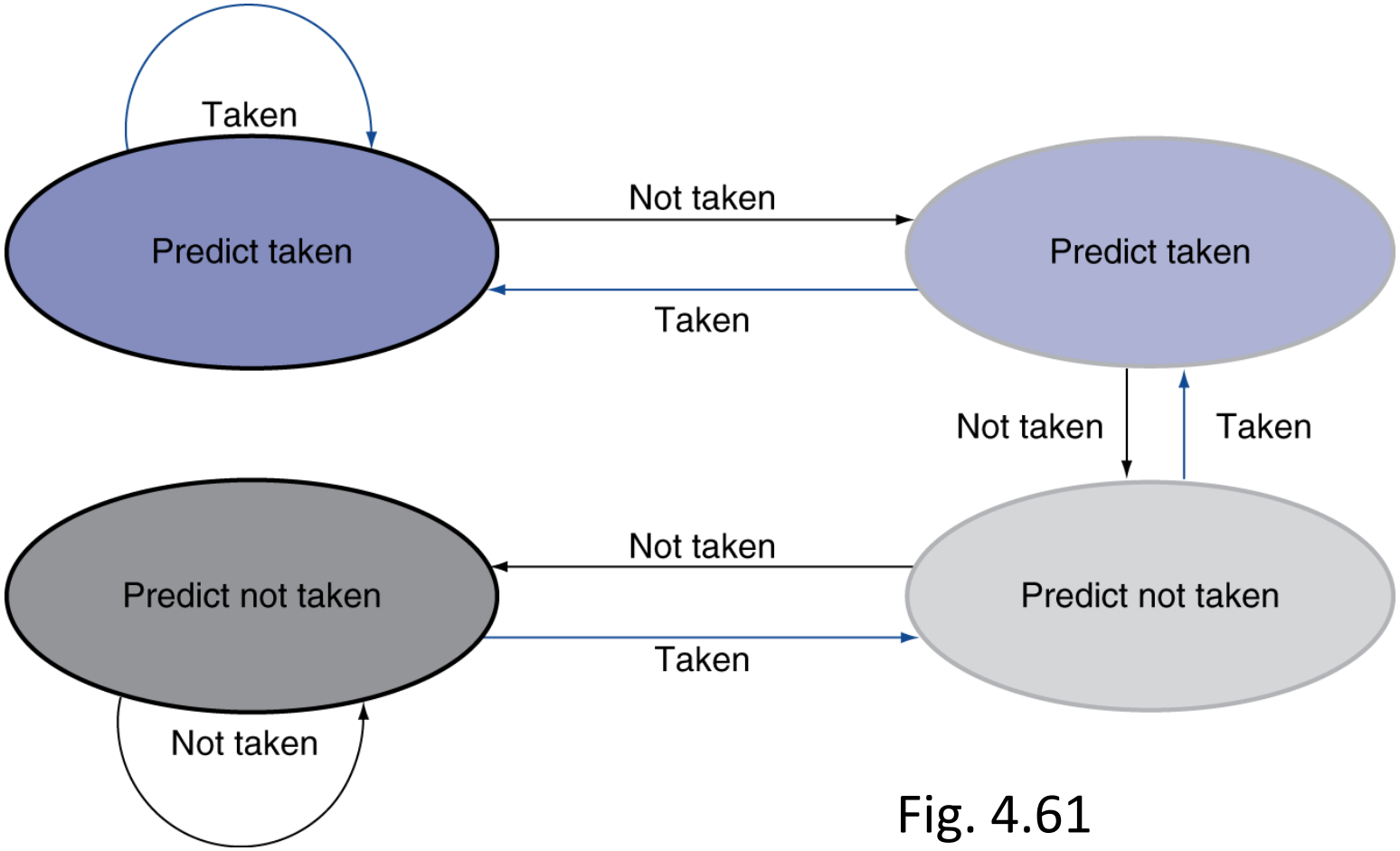
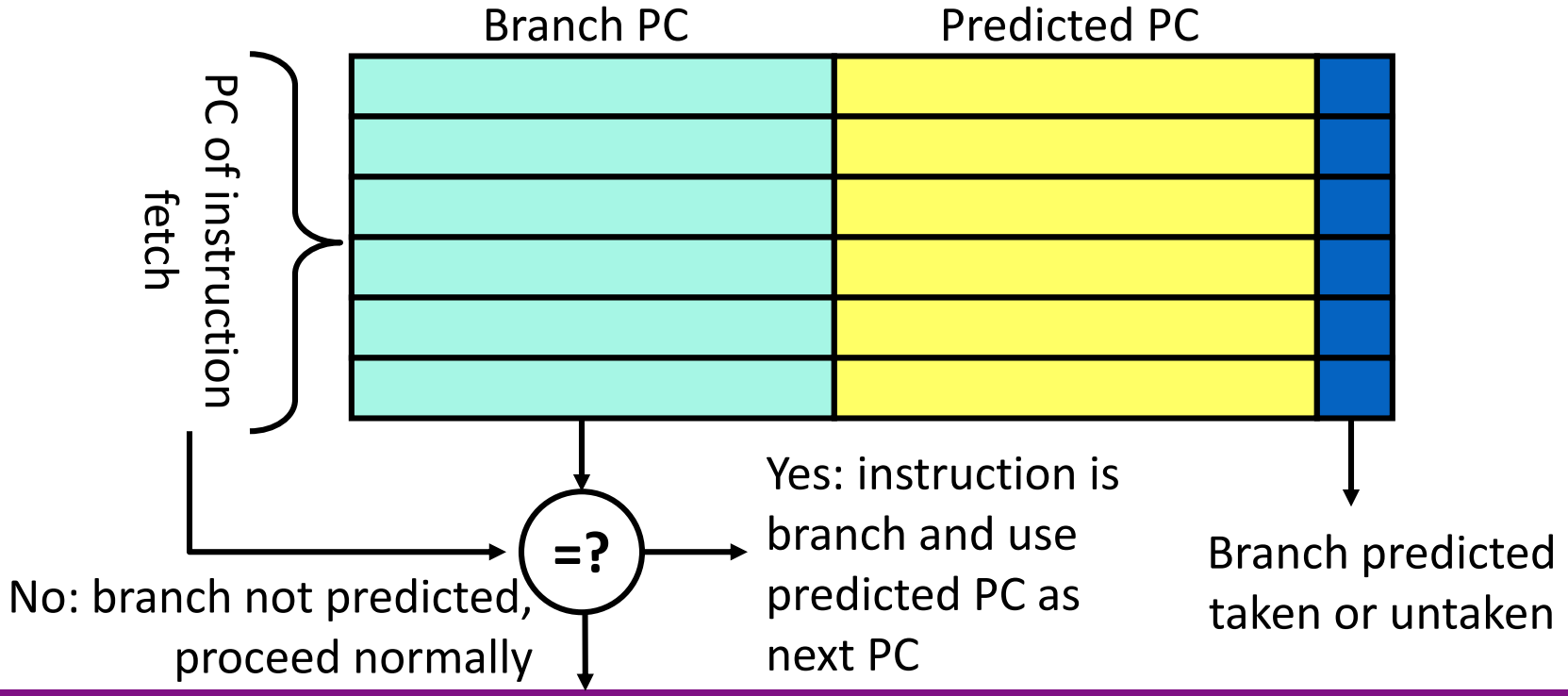


Fig. 4.61

Calculating the Branch Target

- Need target address at same time as prediction
  - *Branch Target Buffer* (BTB): use PC to fetch instruction and simultaneously look up BTB to get prediction AND branch address (if taken)



BTB in the Pipeline

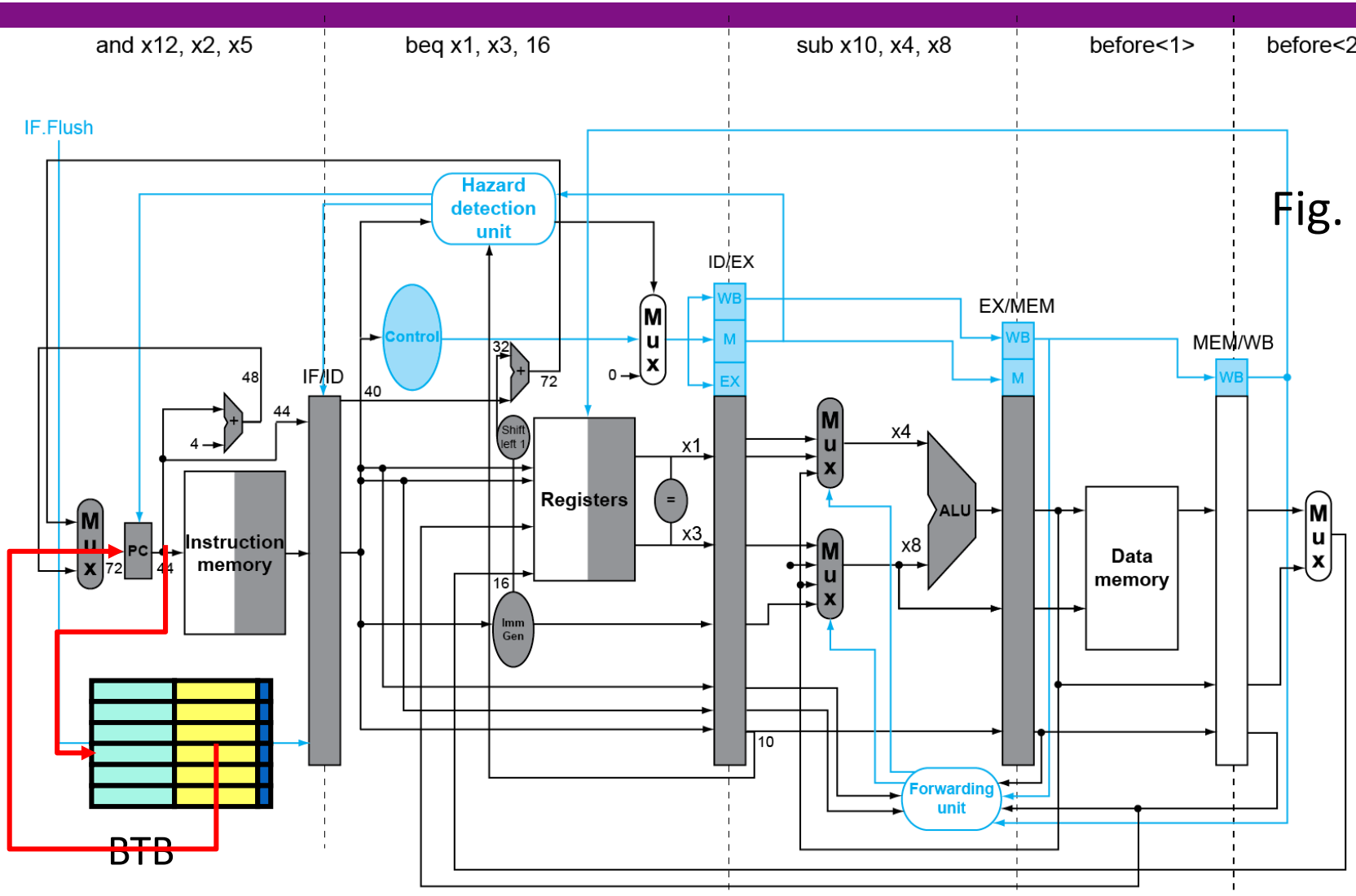


Fig. 4.60

Outline

- Problem statement and logic review (Sec. 4.1, 4.2)
- Building a simple RISC-V processor with datapath and control (Sec. 4.3, 4.4)
- Building a pipelined RISC-V processor with datapath and control (Sec. 4.5, 4.6)
- Dealing hazards in pipelined processor: data and control hazards (Sec. 4.7, 4.8)
- Handling exceptions (Sec. 4.9)
- Instruction-level parallelism (Sec. 4.10)
- ARM Cortex-A53 and Intel Core i7 Pipelines (Sec. 4.11, 4.12)

Events and Interrupts

- What happen if CPU fetches an instruction and cannot understand its opcode?

Event from internal

- What happen if you type a key on the keyboard? How does the computer know and start processing your input?

Event from external

- When these special “events” occur, the CPU needs to “interrupt” the currently executing program and switch to execute some special subroutines to handle the events

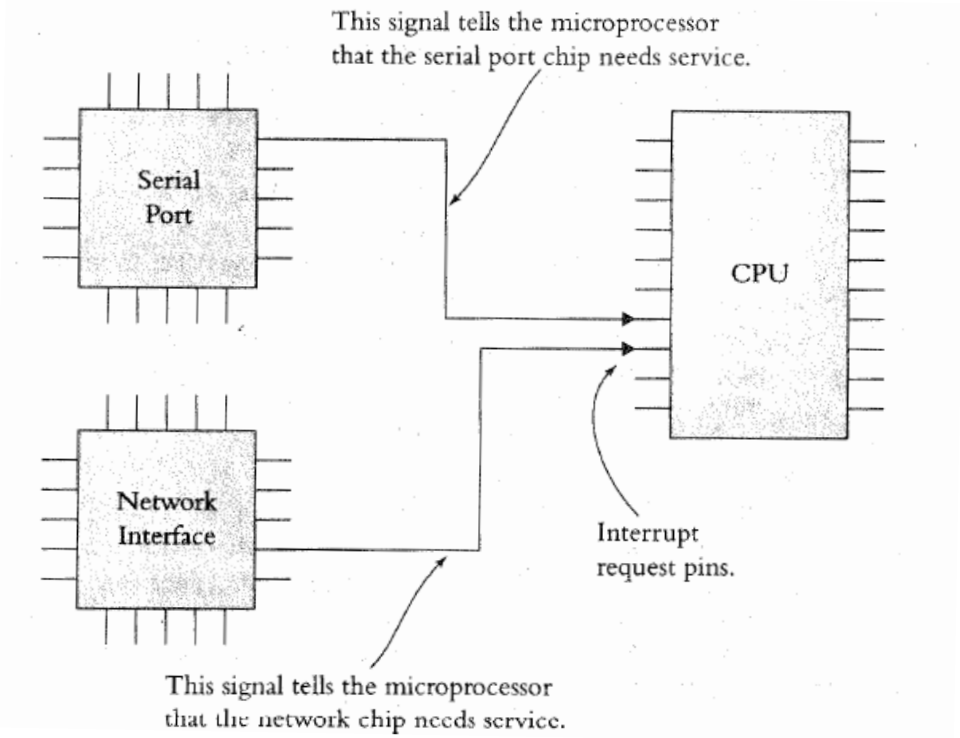
Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control in CPU to service the events
  - Similar to a procedure call, but unplanned
- *Exception*
  - Arises within CPU, e.g., undefined opcode, overflow, syscall
- *Interrupt*
  - From an external I/O controller

Type of event	From where?	RISC-V terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Either

Detection of Exceptions and Interrupts

- How does the CPU know when there is an interrupt?
  - Usually when it receives a signal from one of the IRQ (interrupt request) pins, which are connected to one or more external I/O controllers
- How about exceptions?
  - CPU controller may raise a flag, e.g., undefined opcode, overflow, …

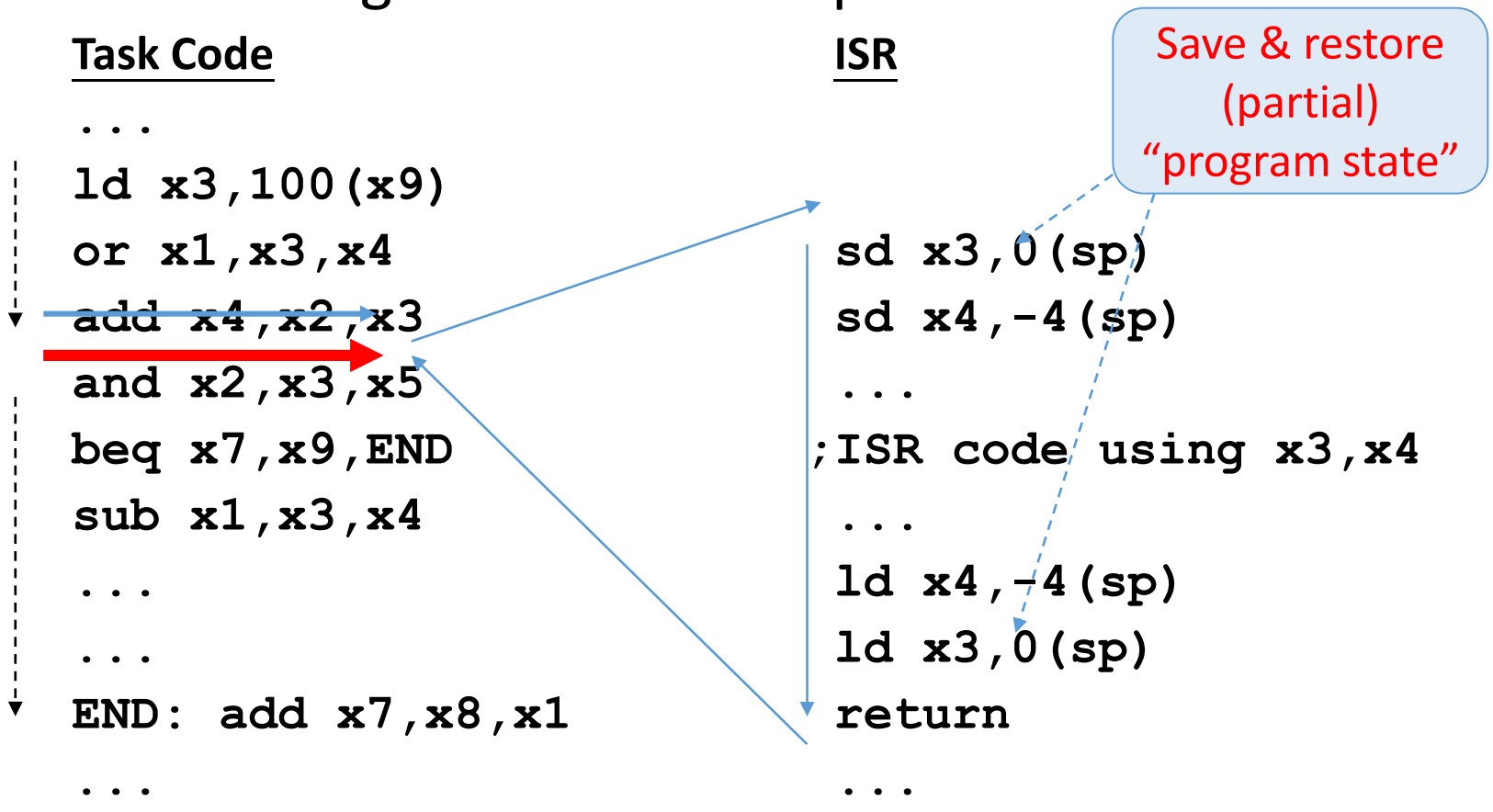


Handling "External" Interrupts

- What does the CPU do in handling an interrupt?
  - When receiving an interrupt signal from IRQ, the CPU
    (1) stops at the next instruction
    (2) saves the address of the next instruction somewhere
    (3) jumps to a specific *interrupt service routine* (ISR) or *interrupt handler*
      - ISR is basically a subroutine (stored at some specific location in memory and not part of user programs) to perform operations to handle the interrupt with a RETURN at the end
    (4) returns back to the code where left
- How to be transparent to the running program?
  - Similar to handling a procedure call: call/return, save registers ("program state"), etc.

Interrupt Service Routine

- The following shows an example of an ISR



Handling "Internal" Exceptions in RISC-V

- Save PC of offending instruction
  - In RISC-V: *Supervisor Exception Program Counter* (SEPC)
- Save indication of the problem
  - In RISC-V: *Supervisor Exception Cause Register* (SCAUSE)
  - 64 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
  - CPU controller will supply this based on exception
- Jump to *interrupt handler* in OS to handle the exception based on the cause
  - Assume at 0000 0000 1C09 0000_{16} (single entry point for all exceptions)

OS Handling Exceptions in RISC-V

The interrupt handler in the OS:

- Reads cause from SCAUSE and transfer to relevant *interrupt handler*
- Determines action required
- If restartable
  - Takes corrective actions
  - Uses SEPC to return to the program
- Otherwise
  - Terminates the program
  - Reports error using SEPC, SCAUSE, …

Alternate Way to Indicate Cause

- Different handlers handle different causes/interrupts
  - Handler's address in memory determined by the cause
    → *vectored Interrupts*
- An *exception vector address*, supplied by the CPU controller based on interrupt source, to be added to a *vector table base register* to give handler address
  - Undefined opcode: $00\ 0100\ 0000_2$
  - Hardware malfunction: $01\ 1000\ 0000_2$
  - …: …
- Instructions inside the handler either
  - Deal with the interrupt, or
  - Jump to the real handler

Exception vector address to be added to a Vector Table Base Register

"Internal" Exceptions in a RISC-V Pipeline

- Treated as another form of control hazard
- Consider malfunction on **`add`** in EX stage

  `add x1, x2, x3`

  – Must prevent **`x1`** from being clobbered

  (1) Complete previous instructions

  (2) Flush **`add`** and subsequent instructions, IF.Flush, ID.Flush, EX.Flush

  (3) Set SEPC and SCAUSE register values

  (4) Transfer control to handler at $0000\ 0000\ 1C09\ 0000_{16}$
- Similar to mispredicted branch
  – Use much of the same hardware

Pipeline with Exceptions

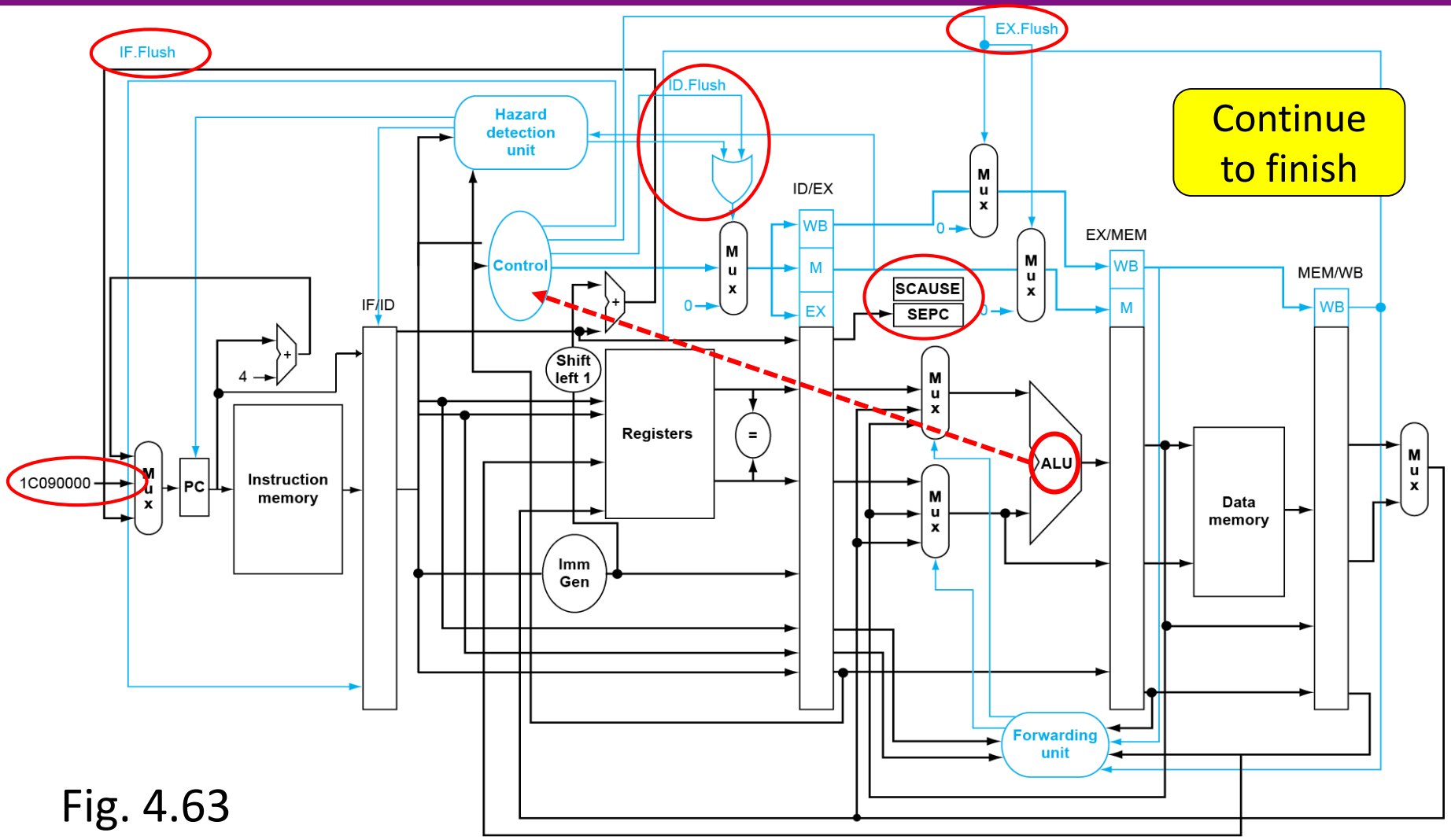


Fig. 4.63

Exception Handling

- Restartable exceptions
  - Pipeline flushes the offending instruction and following instructions
  - Handler executes and then returns to the instruction
    - The instruction is refetched and executed from scratch
- PC saved in SEPC register
  - Identifies the instruction causing the exception

Exception Example

- Exception on **add** in

```
40      sub   x11, x2, x4
44      and   x12, x2, x5
48      or    x13, x2, x6
4C      add   x1,  x2, x1
50      sub   x15, x6, x7
54      ld    x16, 50(x7)
…
```

- Handler

```
1C090000    sd    x26, 1000(x0)
1C090004    sd    x27, 1008(x0)
…
```

Exception Example

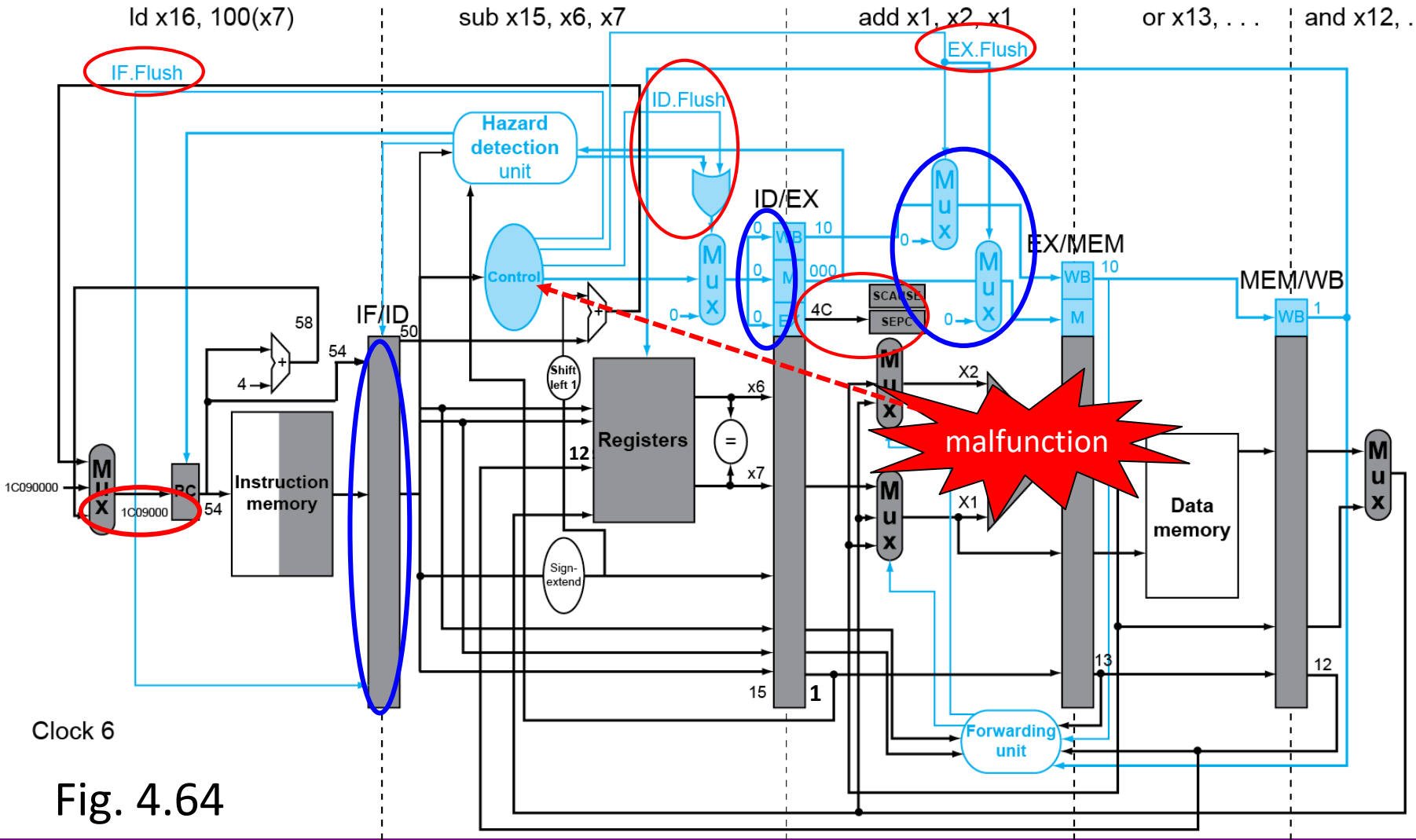


Fig. 4.64

Exception Example

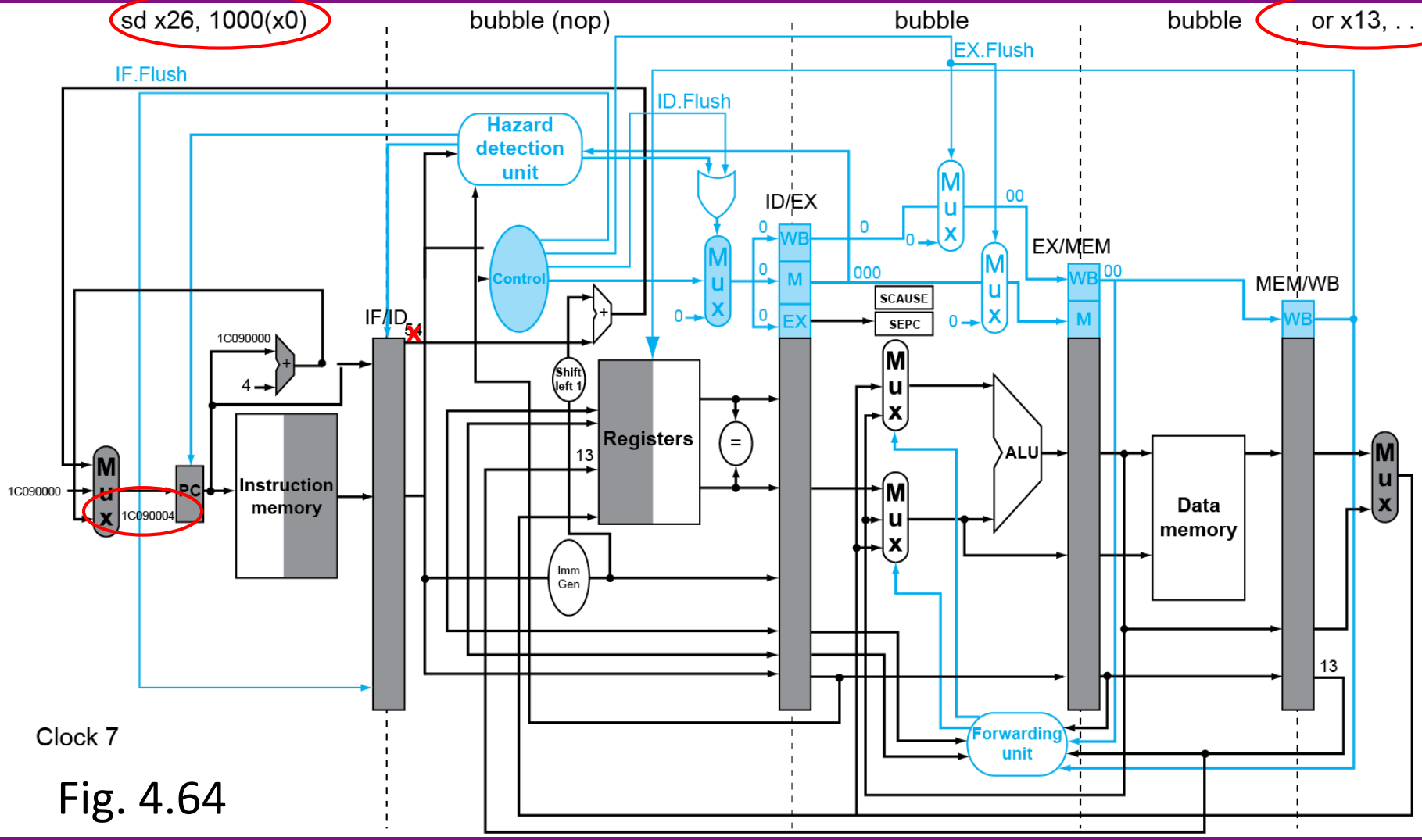


Fig. 4.64

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
  - Good to know in which pipeline stage a type of exception can occur
- In complex pipelines
  - Multiple instructions per cycle, out-of-order completion
  - Maintaining precise exceptions is difficult!
- Some computers provide imprecise exceptions
  - Simplifies hardware, but more complex handler software

Outline

- Problem statement and logic review (Sec. 4.1, 4.2)
- Building a simple RISC-V processor with datapath and control (Sec. 4.3, 4.4)
- Building a pipelined RISC-V processor with datapath and control (Sec. 4.5, 4.6)
- Dealing hazards in pipelined processor: data and control hazards (Sec. 4.7, 4.8)
- Handling exceptions (Sec. 4.9)
- Instruction-level parallelism (Sec. 4.10)
- ARM Cortex-A53 and Intel Core i7 Pipelines (Sec. 4.11, 4.12)

Instruction-Level Parallelism (ILP)

Exploiting the parallelism among instructions

- Pipelining: execute multiple instructions in parallel
- To increase ILP:
  - Deeper pipeline ⇒ less work per stage, shorter clock cycle
    - Still one cycle per instruction
  - Multiple-issue
    - Fetch and start multiple instructions per clock cycle by replicating pipeline stages ⇒ multiple pipelines
    - Need to
      → Packaging instructions into *issue slots*
      → Dealing with data and control hazards
    - CPI < 1, e.g., 4GHz 4-way multiple-issue
      → 16 BIPS, peak CPI = 0.25, peak IPC = 4
      → dependencies among instructions reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler has global knowledge; detects and avoids hazards, and groups parallel instructions to issue them together
  - Pack into issue slots → *very long instruction word* (VLIW)
  - CPU does not need to check hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue in each cycle (limited by instruction fetch window)
  - Compiler can help by reordering instructions so that CPU may find more parallel instructions
  - CPU resolves hazards using advanced techniques at runtime

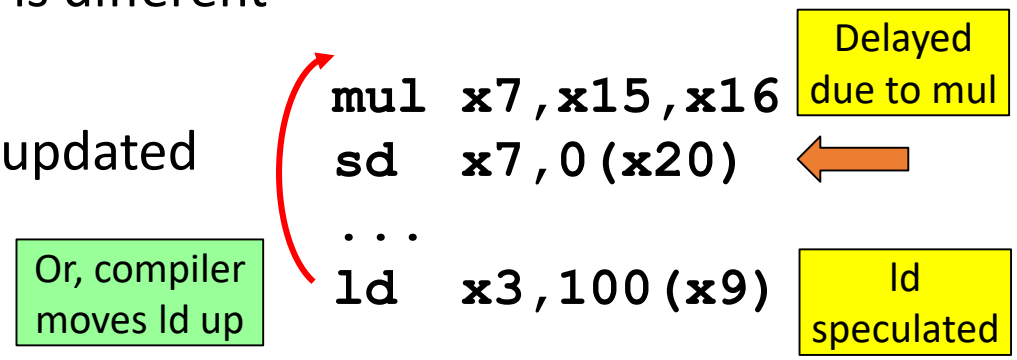
Challenge of Multiple Issue

- Typical integer programs have close to 20% branch instructions → 1 branch in every 5 instructions
  - Difficult to find sufficient number of independent, parallel instructions within the 5 instructions
  - Cannot find instructions across branches either, because we don't know whether they will actually be executed

- How to find sufficient ILP in instruction stream?
  → *speculation*

Speculation for Exploiting More ILP

- "Guess" what to do with an instruction
  - Start operations as soon as possible
  - Check whether guess was right
    - If so, complete the operations
    - If not, roll-back and do the right things
- Examples:
  - Speculate on branch outcome, e.g., branch prediction
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated



Speculation for Exploiting More ILP

- Speculation is common to static and dynamic multiple issue
- Static multiple issue: compiler reorders instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Dynamic multiple issue: hardware can look ahead for instructions to execute, effectively reordering execution of the instructions at run-time
  - 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) architecture
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

RISC-V with Static Dual Issue

- Compiler prepares the two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - The two are paired and aligned on a 64-bit boundary
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Fig. 4.65

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

RISC-V with Static Dual Issue

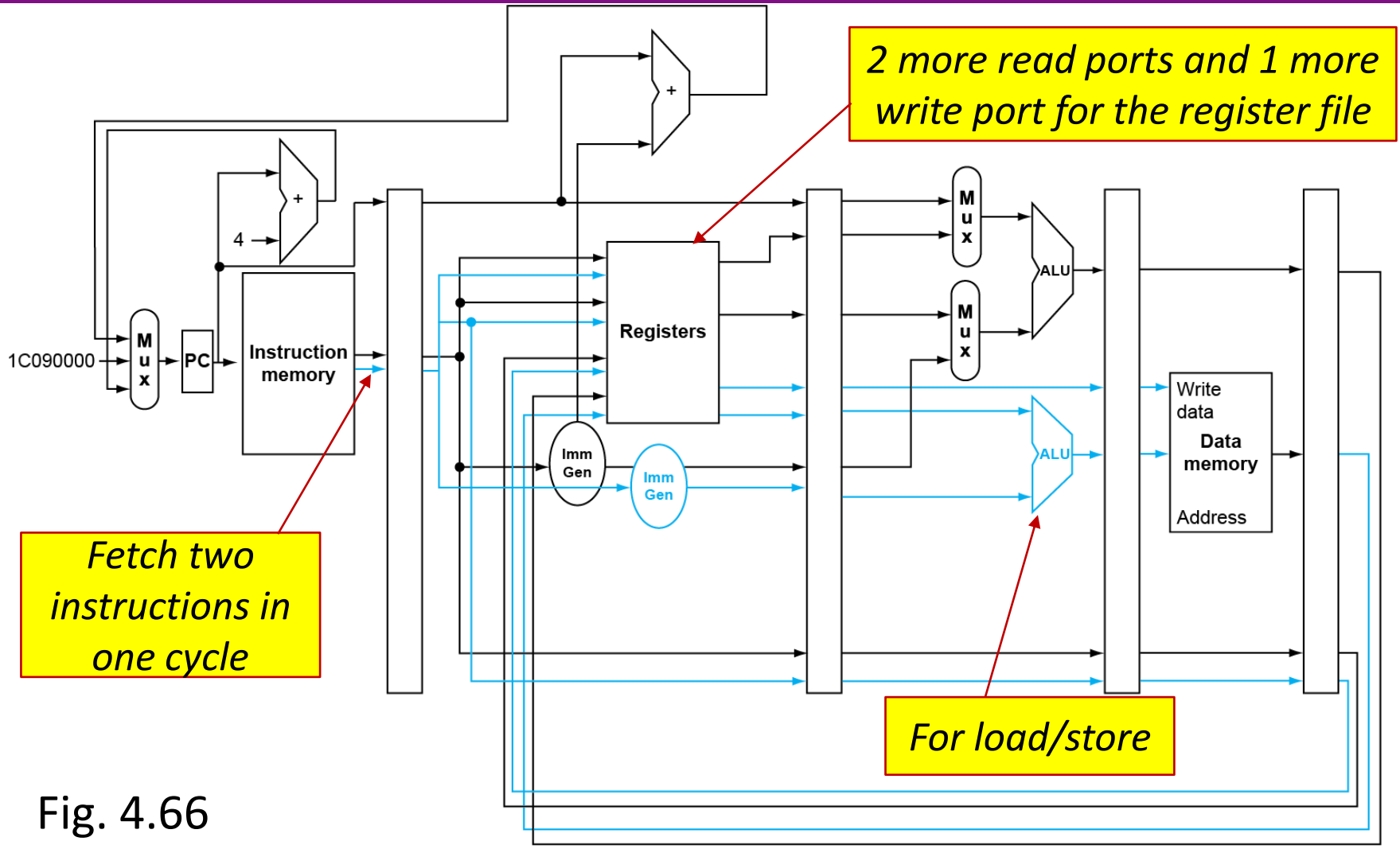


Fig. 4.66

Hazards in the Dual-Issue RISC-V

- With more instructions executing in parallel → more hazards to handle and effects of hazards more severe
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet

```
add   x10,x0,x1
ld    x2,0(x10)
```

    - Split into two packets, effectively a stall

```
add x10,x0,x1
nop
nop
ld  x2,0(x10)
```

- Load-use hazard
  - Still one cycle use latency, but now two instructions

```
ld  x2,0(x10)
nop
nop
add x10,x0,x2
```

- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld   x31,0(x20)     // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)     // store result
      addi x20,x20,-8     // decrement pointer
      blt  x22,x20,Loop   // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:	nop	ld x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	4

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

Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld   x31,0(x20)
      add  x31,x31,x21
      sd   x31,0(x20)
      addi x20,x20,-8
      blt  x22,x20,Loop
```

```
nop
ld   x31,0(x20)
addi x20,x20,-8
nop
add  x31,x31,x21
nop
blt  x22,x20,Loop
sd   x31,8(20)
```

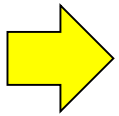
	ALU/branch		
Loop:	nop	ld x31,0(x20)	1
	addi x20,x20,-8	nop	2
	add x31,x31,x21	nop	3
	blt x22,x20,Loop	sd x31,8(x20)	4

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

Loop Unrolling for More ILP

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- May induce new loop-carries dependences
  - E.g., loop-carried "anti-dependencies"
  - Output and anti-dependencies are "*name dependence*" (due to reuse same register)
  - Sol.: use different registers per replication → *register renaming* (use different registers)

```
Loop: ld   x31,0(x20)
      add  x31,x31,x21
      sd   x31,0(x20)
      addi x20,x20,-8
      blt  x22,x20,Loop
```



```
Loop: ld   x31,0(x20)
      add  x31,x31,x21
      sd   x31,0(x20)
      addi x20,x20,-8
      blt  x22,x20,Loop
      ld   x31,0(x20)
      add  x31,x31,x21
      sd   x31,0(x20)
      addi x20,x20,-8
      blt  x22,x20,Loop
```

Loop Unrolling Example

```
Loop: ld   x31,0(x20)
      add  x31,x31,x21
      sd   x31,0(x20)
      addi x20,x20,-8
      blt  x22,x20,Loop
```

- Unroll 4 times:

	ALU/branch	Load/store	cycle
Loop:	addi x20,x20,-32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28,x28,x21	ld x30, 16(x20)	3
	add x29,x29,x21	ld x31, 8(x20)	4
	add x30,x30,x21	sd x28, 32(x20)	5
	add x31,x31,x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	blt x22,x20,Loop	sd x31, 8(x20)	8

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

Static Multiple Issue: Summary

- Minimal hardware: replicate datapath with minimal control → easy to implement, make good use IC
- Software does all the control, i.e., scheduling
  - Programs have to be recompile or rewritten, otherwise get wrong results → not backward compatible and architectural invisible
- Dynamic multiple issue: opposite (see next)

Dynamic Multiple Issue

- "*Superscalar*" processors
- CPU decides whether to issue 0, 1, 2, … instructions each cycle
  - While avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - *Sequential semantics* of the program is ensured by the CPU
- Allows backward compatibility of code
  - No need to change program

Dynamic Multiple Issue

- Requirements for dynamic multiple issue:
  1. Avoid structural and data hazards: detect and foward
  2. Allow instructions executed *out-of-order* (OOO) to avoid stalls by long-latency instructions
     - But commit result to registers *in-order* (specified by the code)
  3. Support speculative execution: use branch prediction and schedule instructions across branches
     - Nullify operations on wrong path
  4. Use multiple functional units (e.g., adder, multiplier, FP) that may with varying latencies
     - Reduce the chance of structural hazards

Instruction scheduling by hardware

Loop unrolling by hardware

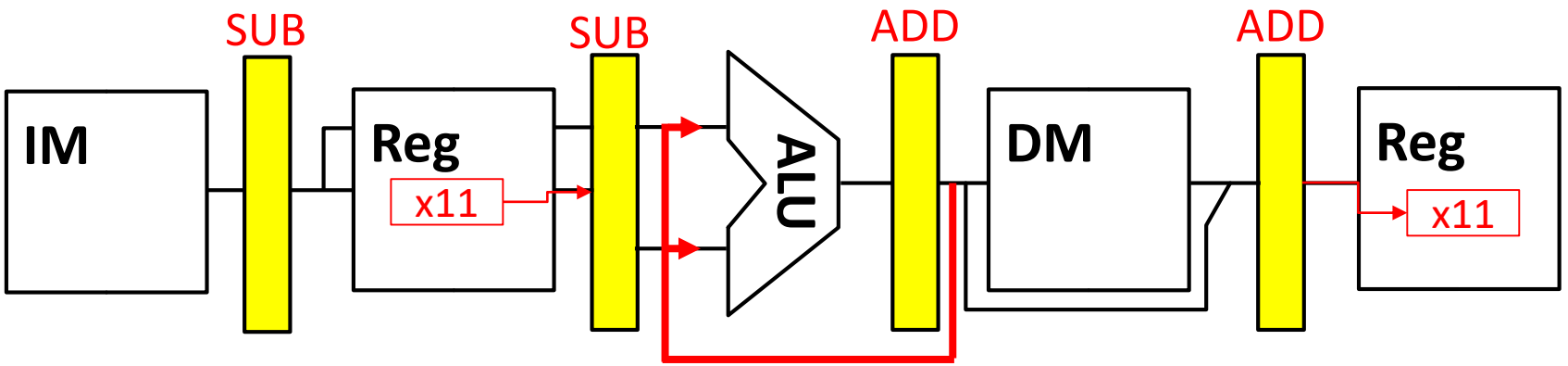
```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

Can start **sub** while **add** is waiting for **ld**

Recall the RISC-V 5-Stage Pipeline

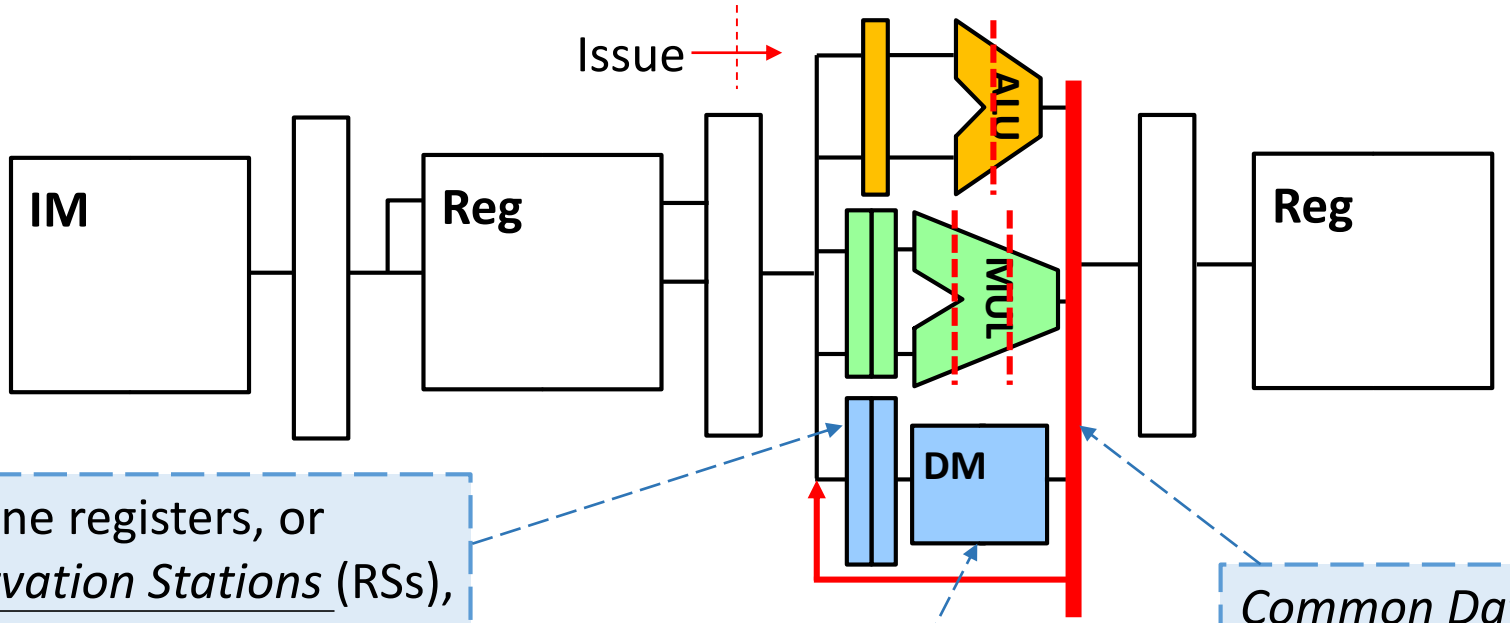
- Sequential, in-order, single issue
- Pipeline registers carry all information needed for execution of the corresponding instruction
- Forwarding or stalling to resolve pipeline hazards
  - Normally data flows between instructions through registers
  - Forwarding provides a shortcut that bypasses register file

```
add   x11,x12,x13
…
sub   x14,x11,x15
```



4. Supporting Multiple Functional Units

- Multiple functional units with varying latencies



Pipeline registers, or *Reservation Stations* (RSs), carry all information for the execution of an instruction; multiple RSs per functional unit

Data memory can be treated as a functional unit, supporting load/store instructions

Common Data Bus (CDB) provides a forwarding path between functional units and RSs

Dynamically Scheduled CPU

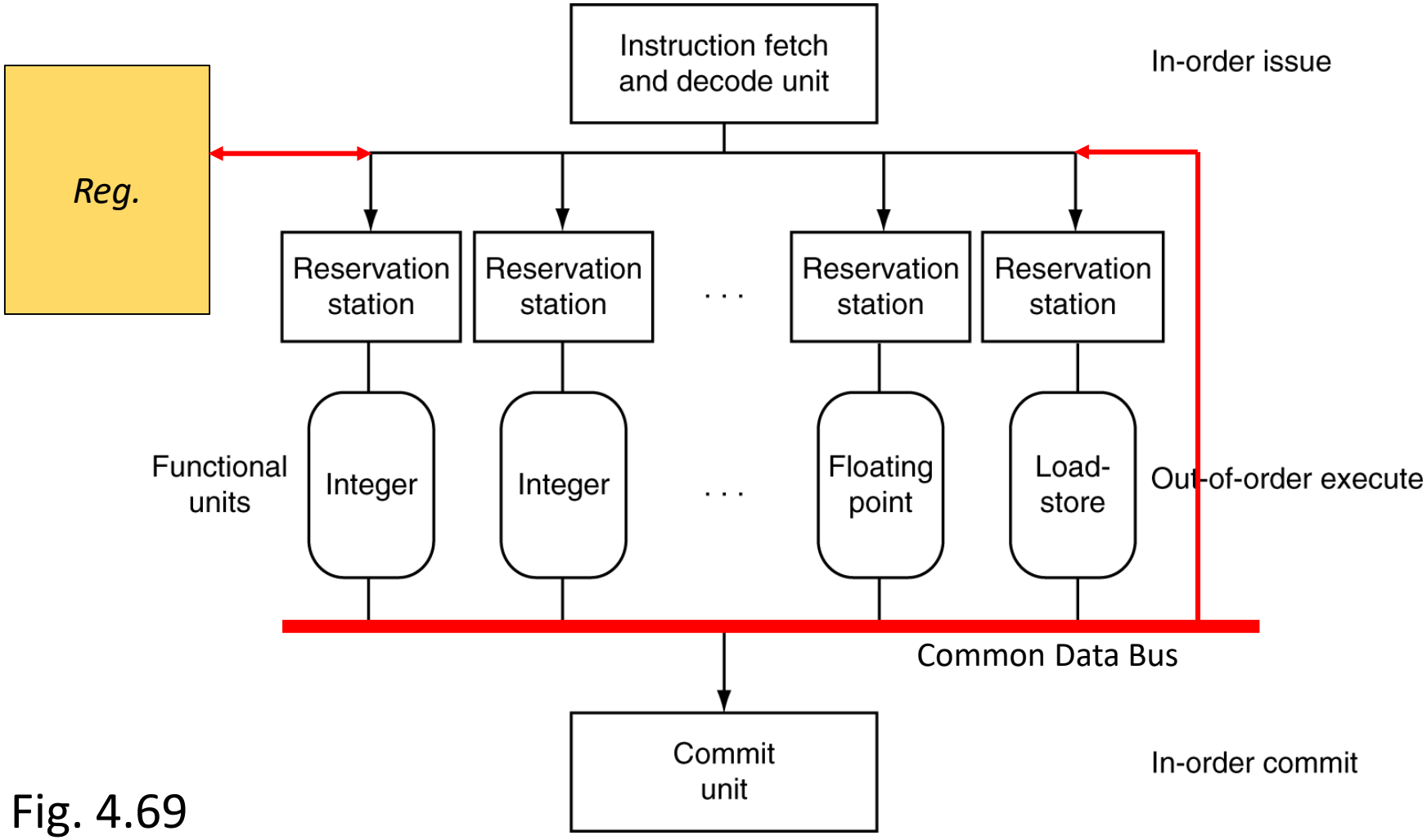
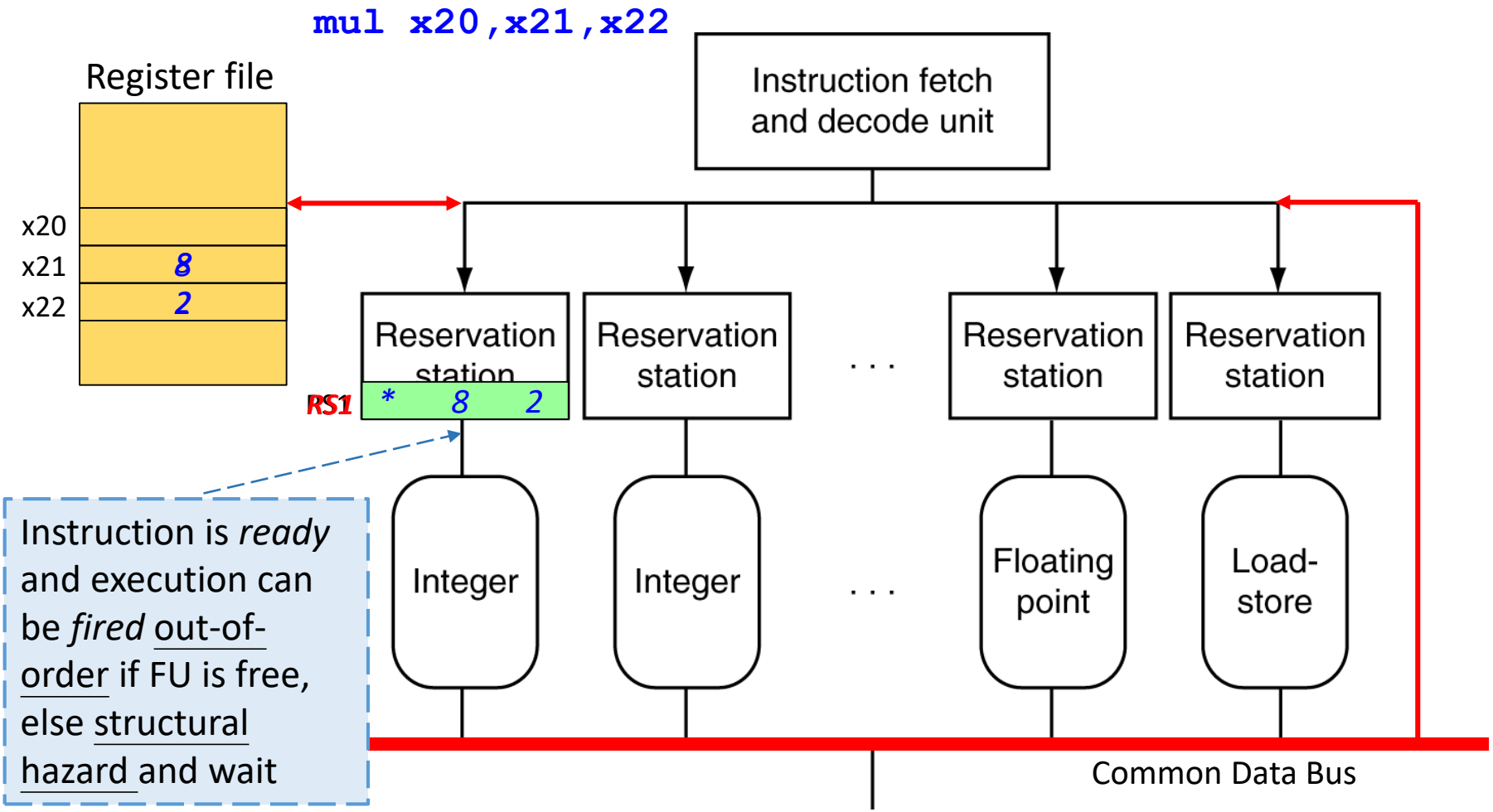
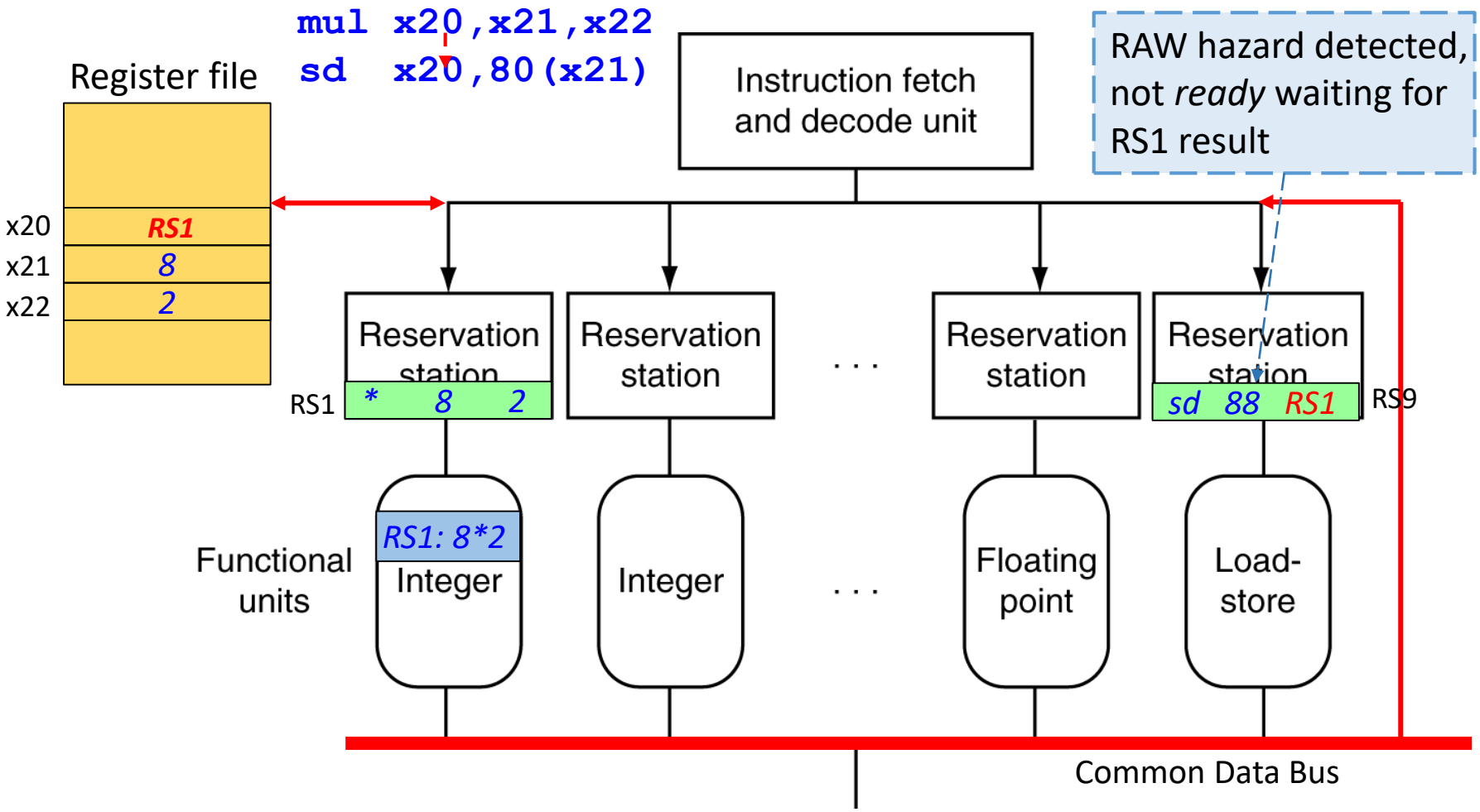


Fig. 4.69

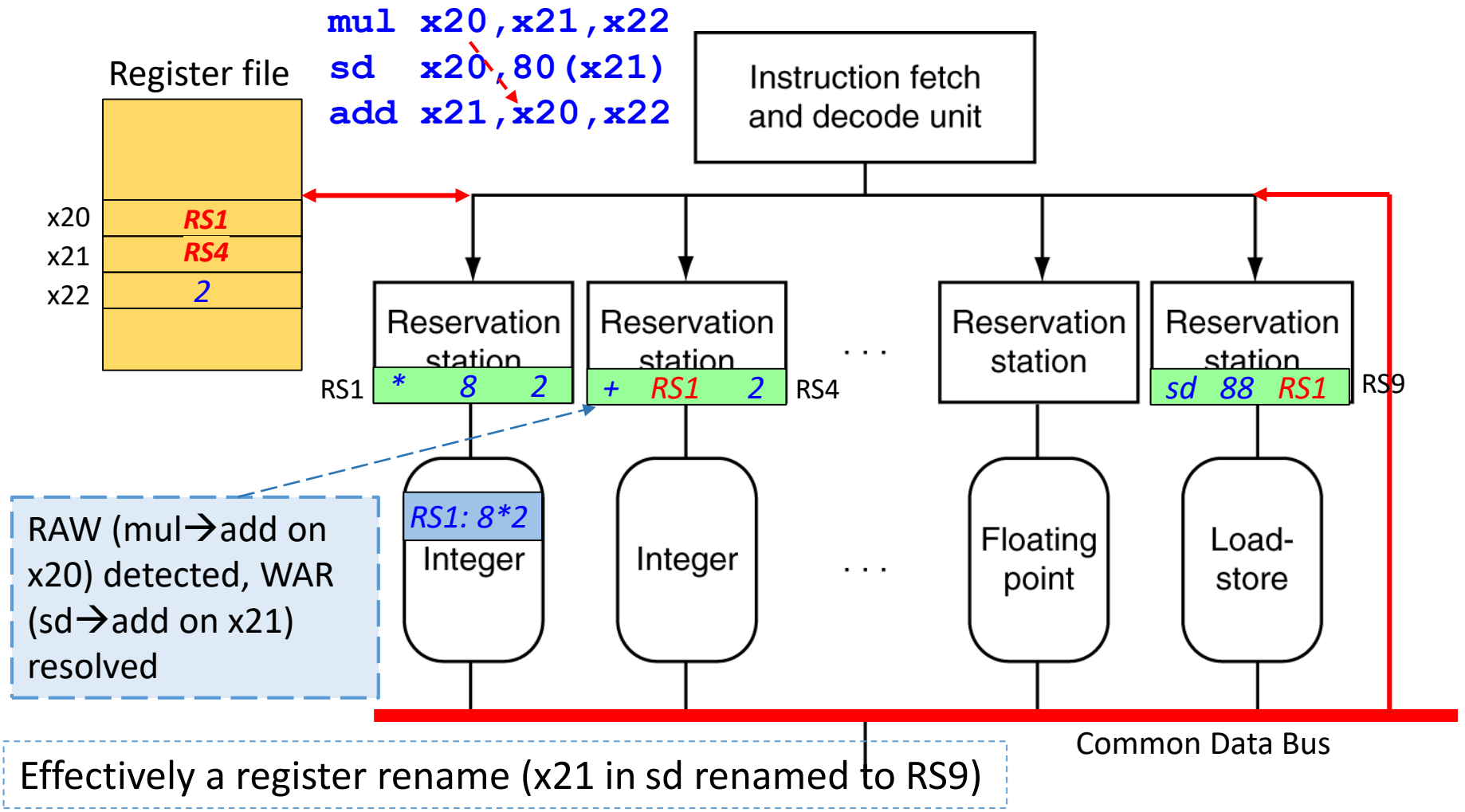
1. Avoid Hazards 2. Allow OOO



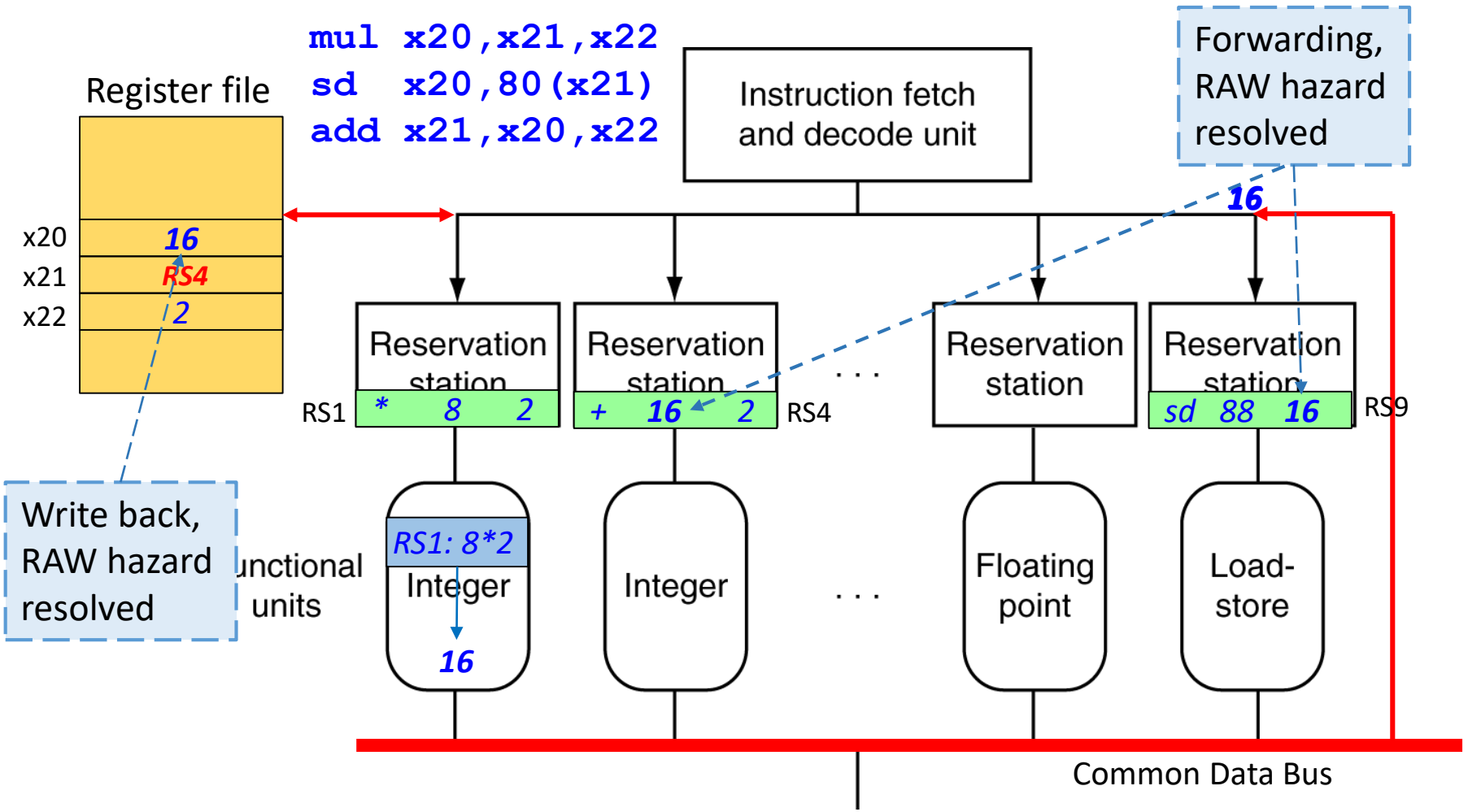
1. Avoid Hazards 2. Allow OOO



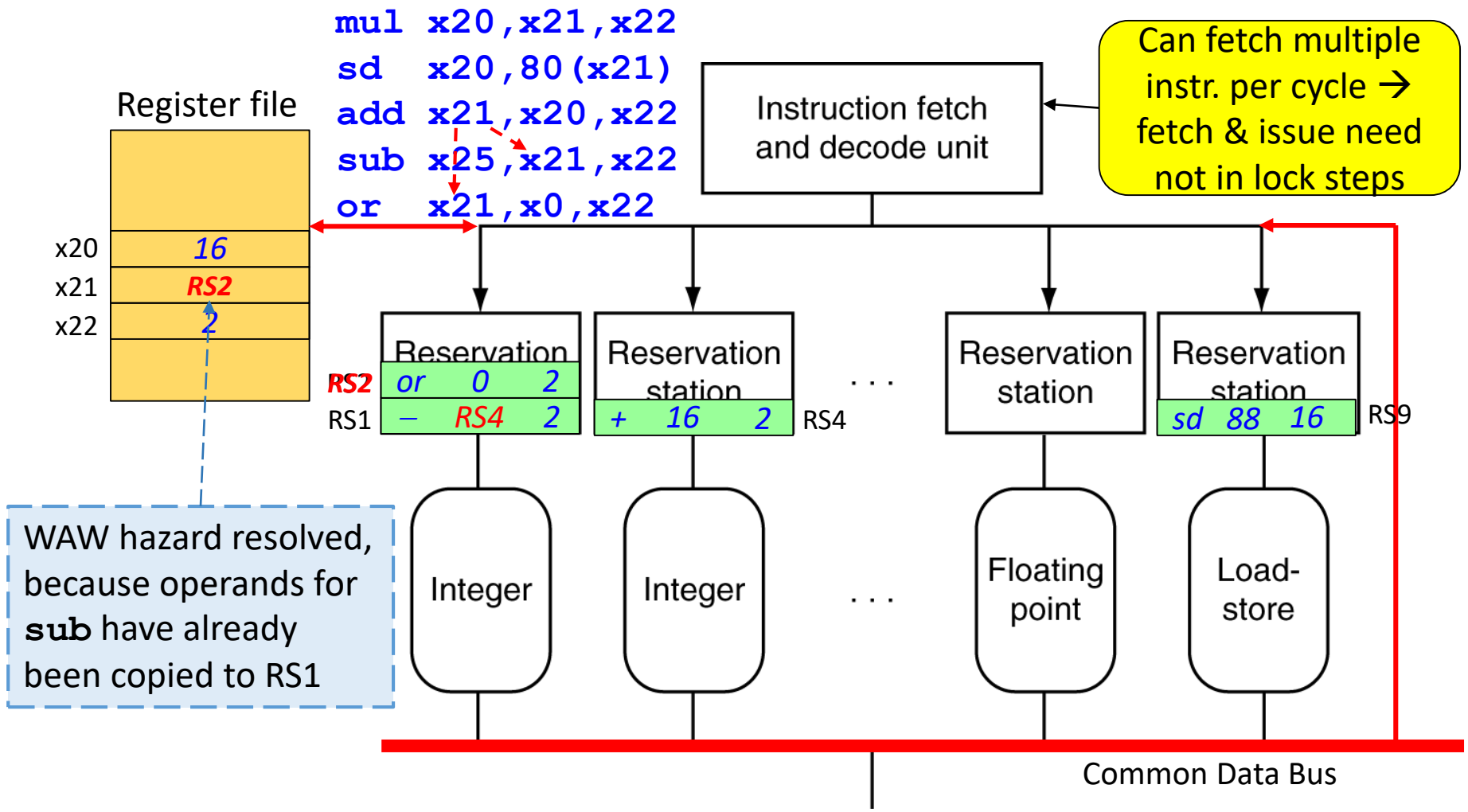
1. Avoid Hazards 2. Allow OOO



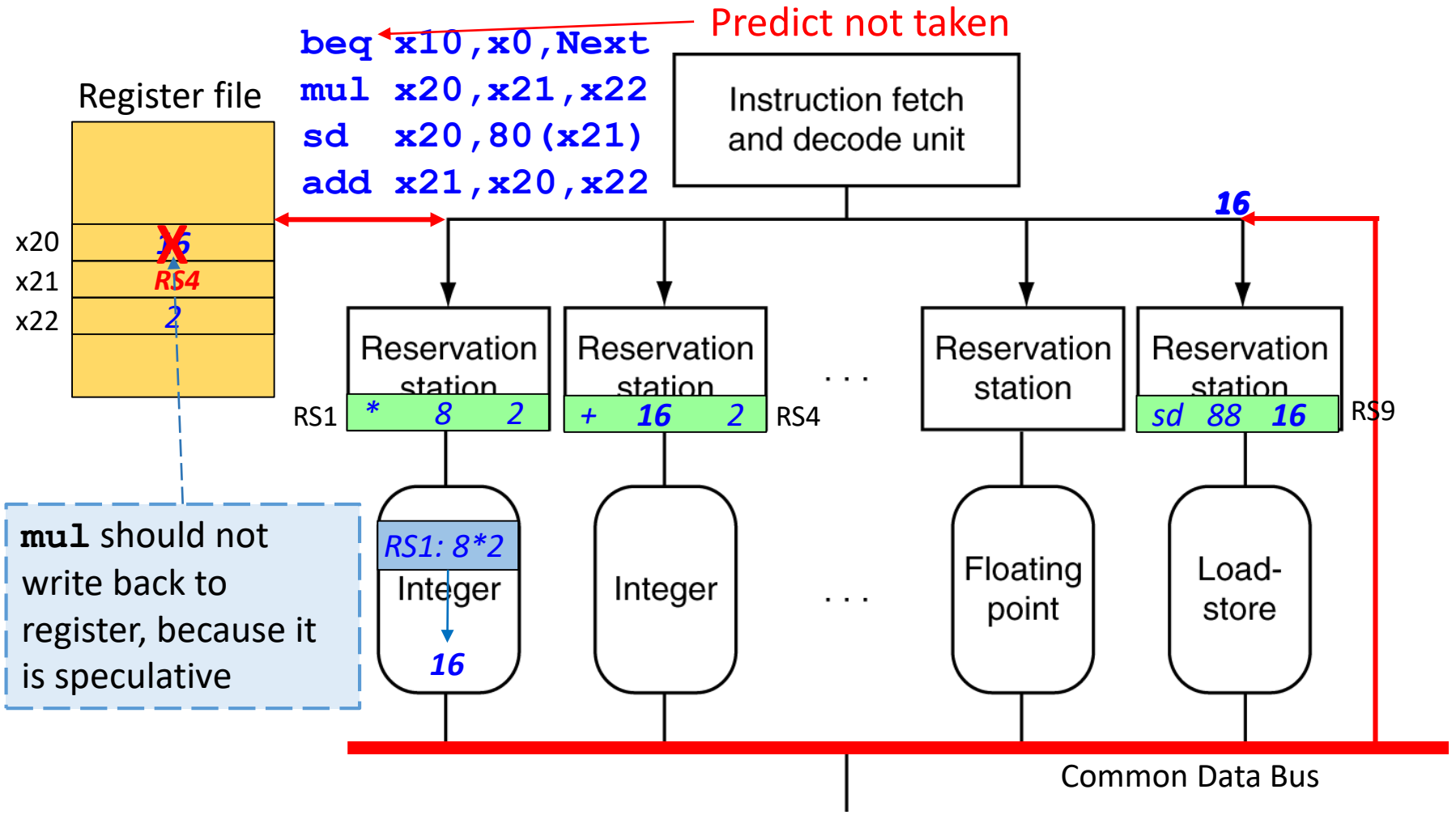
1. Avoid Hazards 2. Allow OOO



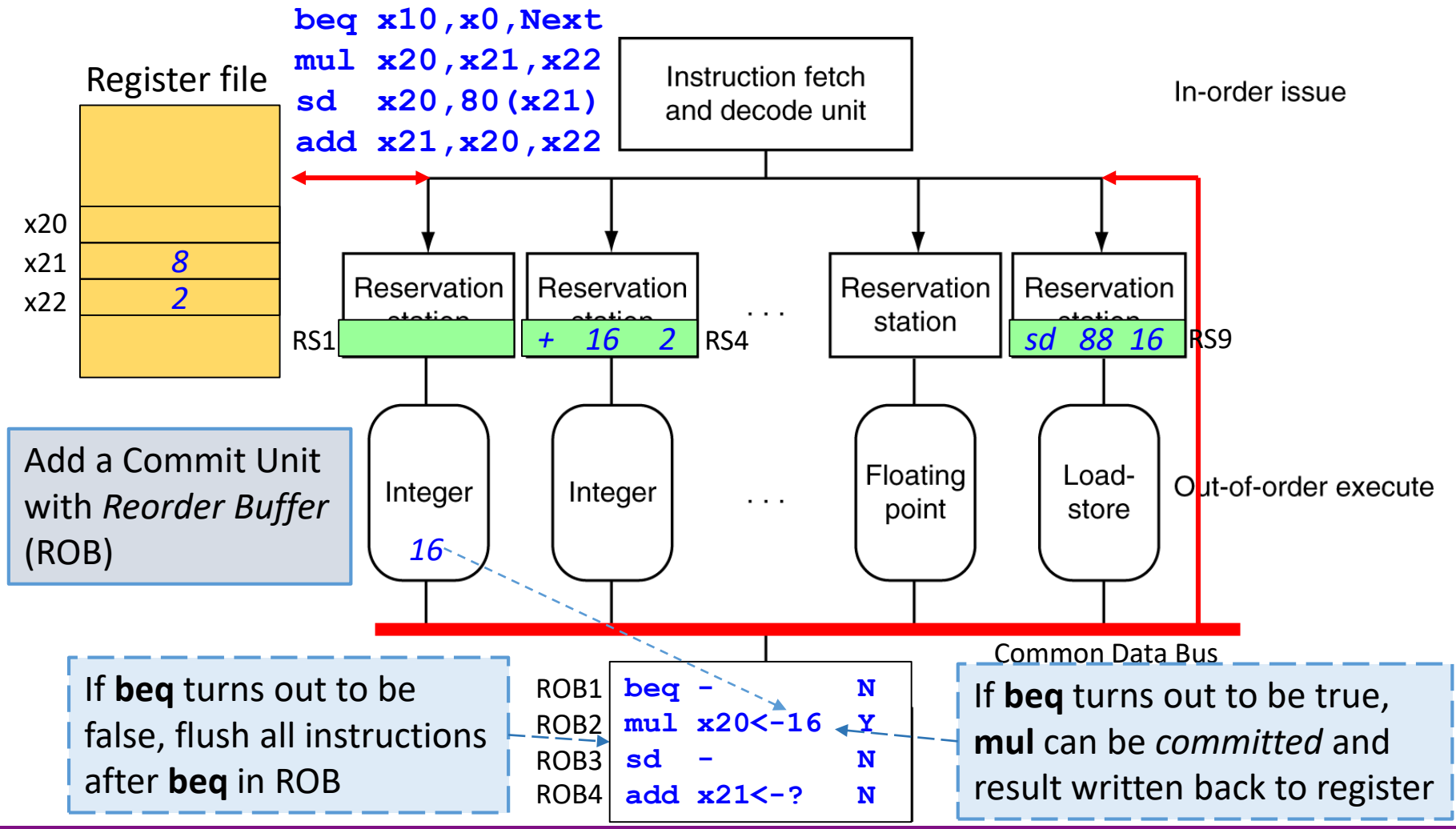
1. Avoid Hazards 2. Allow OOO



3. Support Speculative Execution



3. Support Speculative Execution



Dynamically Scheduled CPU

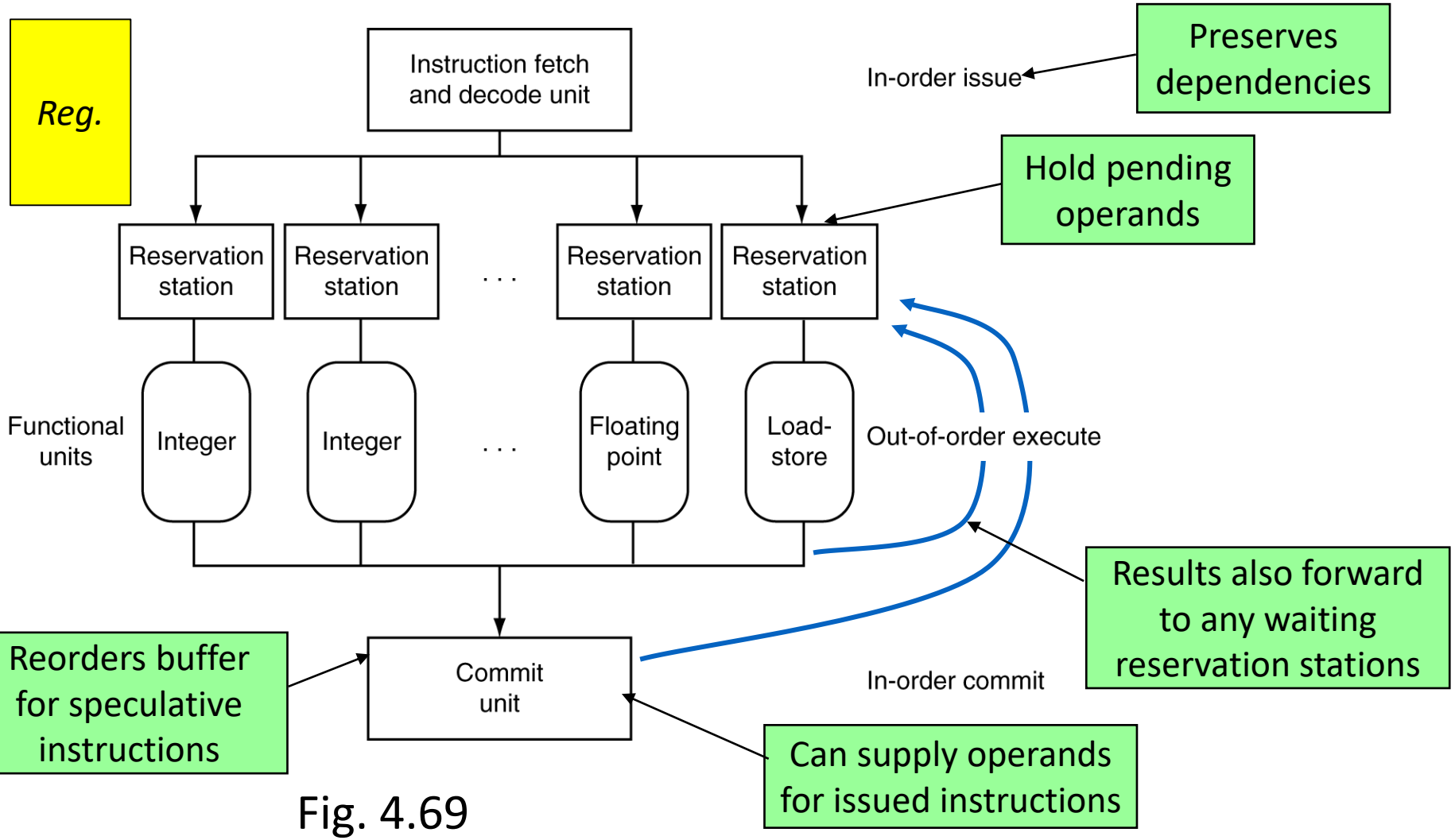


Fig. 4.69

Can Handle Register Renaming

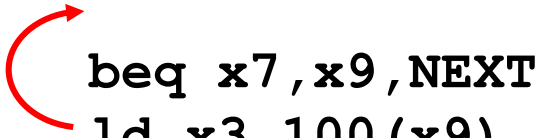
- Reservation stations and reorder buffer (ROB) effectively provide extra "named registers" for *register renaming*
- On instruction issue to a reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station (just like another register)
    - Since operand is no longer required in the register, register can be overwritten → resolve WAR and WAW hazards
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit via Common Data Bus → forwarding for RAW

Can Handle Speculation

- Branch speculation: predict branch & continue issuing
  - Speculated results stored in ROB
  - Don't commit (write into register file) until branch outcome determined
- Load speculation:
  - To avoid load and cache miss delay, can do the followings:
    - Predict the effective address, or
    - Predict loaded value, or
    - Load before completing outstanding stores, or
      → That is why the load is speculative
    - Forward stored values to load unit before written to memory
  - Don't commit load until speculation cleared

Speculation Complication: Exceptions

- What if exception occurs on a speculatively executed instruction?
  - Speculating on certain instructions may introduce exceptions that were formerly not present, e.g., speculative load before null-pointer check



- Static speculation
  - Can add ISA support for deferring exceptions until it is clear that they really should occur
- Dynamic speculation
  - Can buffer exceptions in ROB until completion of the speculated instruction; take exception if speculation is taken

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?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- 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

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards, handling exceptions
- Pipelining is independent of technology
  - More transistors make more advanced techniques feasible, e.g., longer pipeline, multiple functional units, scheduling
  - So why haven't we always done pipelining? → power, cost
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach to address this problem for CISC
  - e.g., complex addressing modes
    - Register update side effects (e.g., auto-increment of register), memory indirection that requires multiple memory accesses
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
  - Handling hazards by forwarding, stalling, compiler scheduling
- Must handle exceptions properly
- Instruction set design affects complexity of pipeline implementation