Computer Architecture Chapter 4

Processor 3

Joonho Kong
School of EE, KNU

Control Hazards – simple solutions

Just stall the pipeline until the branch is resolved

- We should pay 3 cycle delay
- How can we avoid the delay?

Assume (Predict) Branch Not Taken – static branch prediction

- Assume (Predict) that conditional branch will not be taken

→ continue execution of the following instructions

- If the conditional branch is taken,

→ Fetched and decoded instructions must be discarded (change all control signals to 0s)

→ Execution continues at the branch target

- The flush operation is needed in the IF, ID, and EX stages to discard instructions

Control Hazards

Assume (Predict) Branch Not Taken

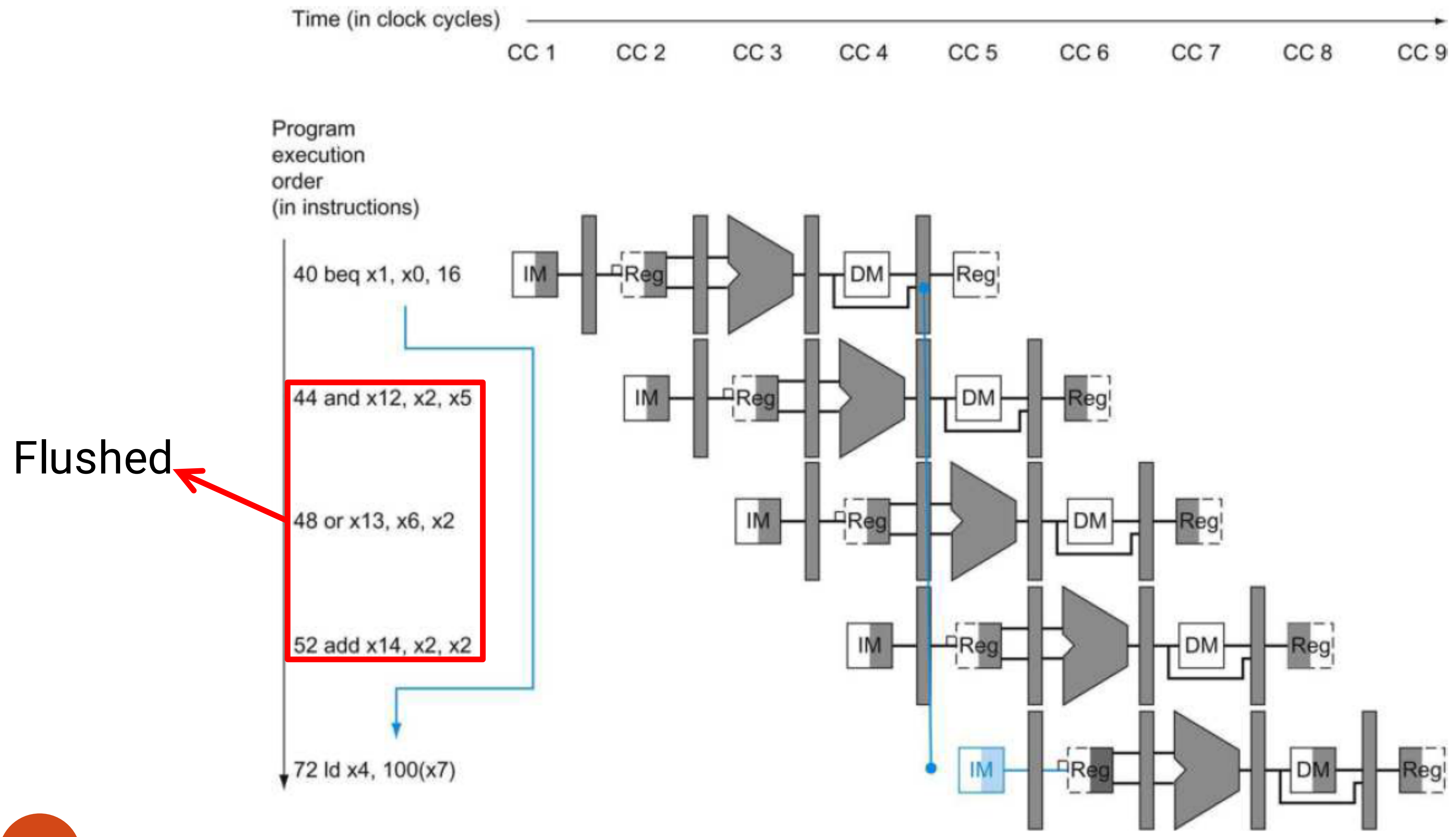


Figure is from "Computer Organization and Design"

Control Hazards

Reducing the Delay of Branches

- Move the conditional branch execution from MEM stage to ID stage
  → Fewer instructions can be flushed

- Computing target address and evaluating branch decision in ID stage
  → For target address, a new adder is required for taken branches
  → For branch decision, we need comparator

- IF.Flush: A new control line to flush instruction in IF stage
- IF.Flush zeros the instruction field of the IF/ID pipeline registers
  → The instruction has no action (No operation: NOP)

Control Hazards

Reducing the Delay of Branches

Difficulties of the branch decision

1. Equality test in the ID stage requires new forwarding logic

 → Bypassed source operands are from EX/MEM or MEM/WB pipeline registers

2. A stall will be needed due to data hazard

 → 1 cycle stall is needed when:

```
add     x5, x30, x31
beq     x5, x6, Loop
```

 → 2 cycle stall is needed when:

```
ld      x5, 0(x30)
beq     x5, x0, Loop
```

Control Hazards

Example

Q) Show what happens when the branch is taken in this instruction sequence, assuming the pipeline is optimized for branches that are not taken, and that we moved the branch execution to the ID stage:

```
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)
```

Control Hazards

Example

Clock 3

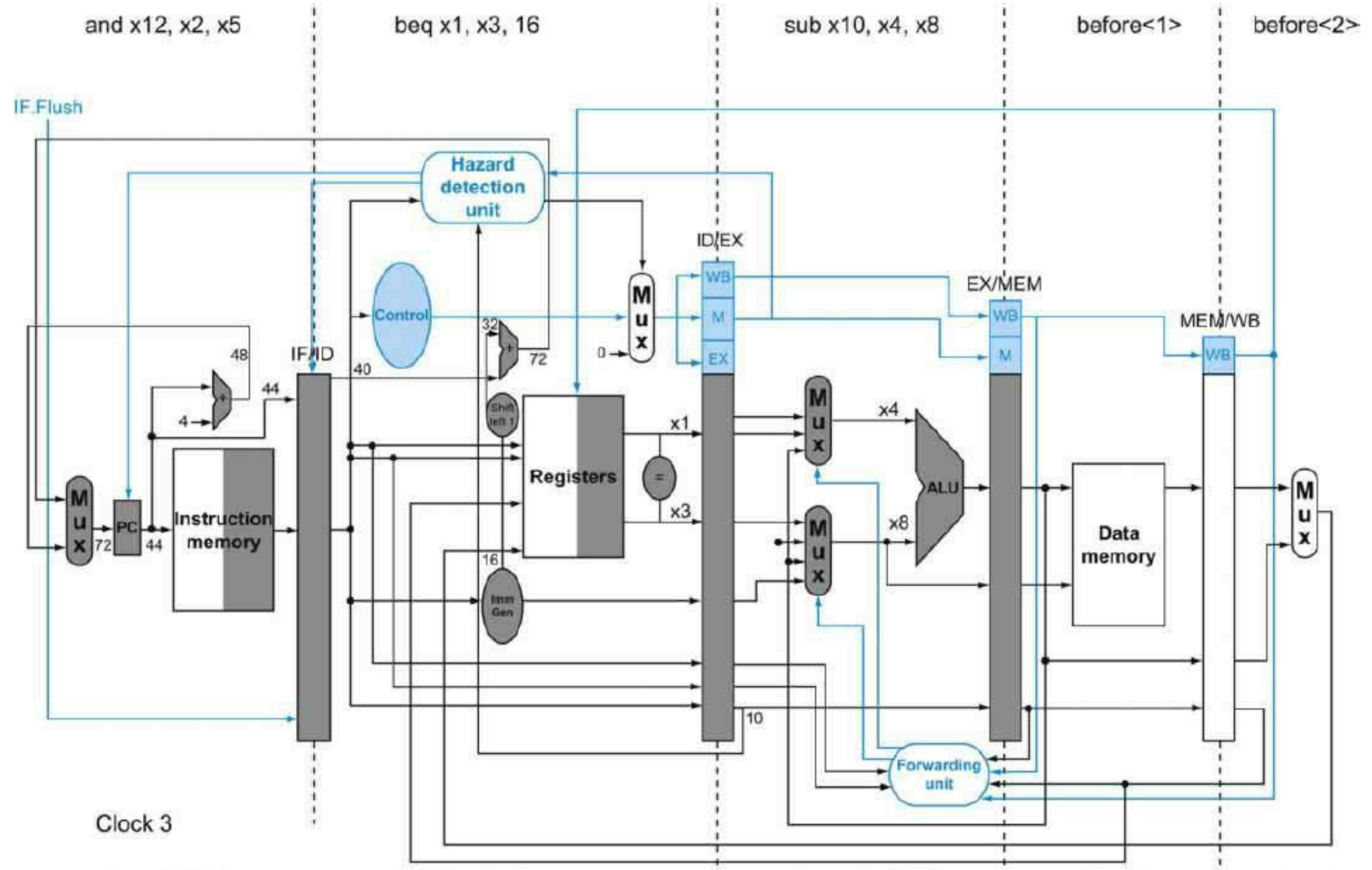


Figure is from "Computer Organization and Design"

Control Hazards

Example

Clock 4

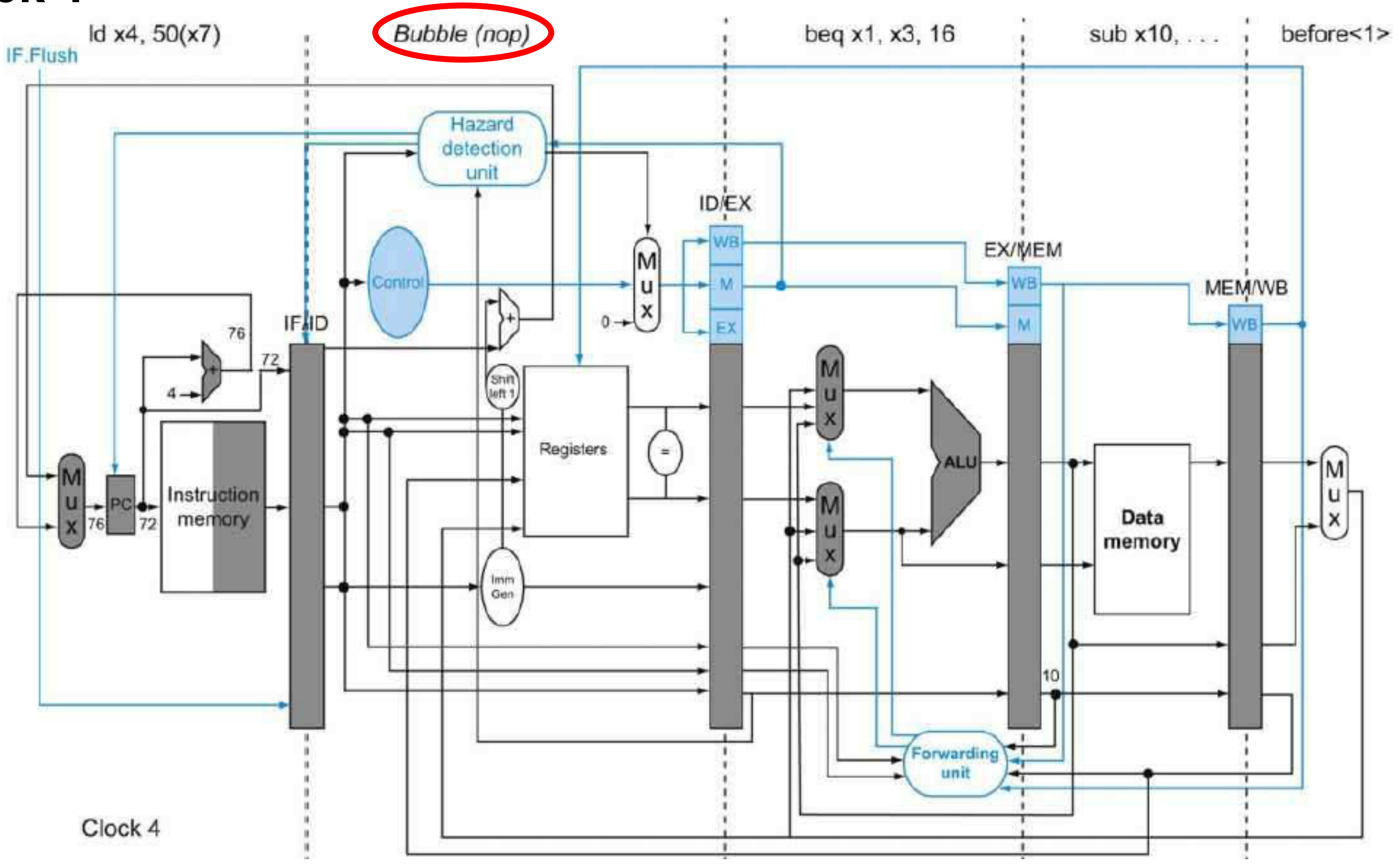


Figure is from “Computer Organization and Design”

Control Hazards

Branch Prediction

Dynamic branch prediction
Branch prediction considering the dynamic program behavior

- This scheme uses a branch prediction buffer or branch history table
- Indicate the recent result of branch using 1 bit
- This scheme doesn't affect correctness

- If we perform right prediction:
  → Continue the sequential instructions

- If we perform wrong prediction:
1) The incorrectly predicted instructions are deleted
2) The prediction bit is inverted and store back
3) The correct instruction is fetched and executed

Control Hazards

2-Bit Dynamic Branch Prediction

- Maintained by 2-bit state machine (4 states)
- Check both a local branch and the global behavior of branches together yields greater prediction accuracy

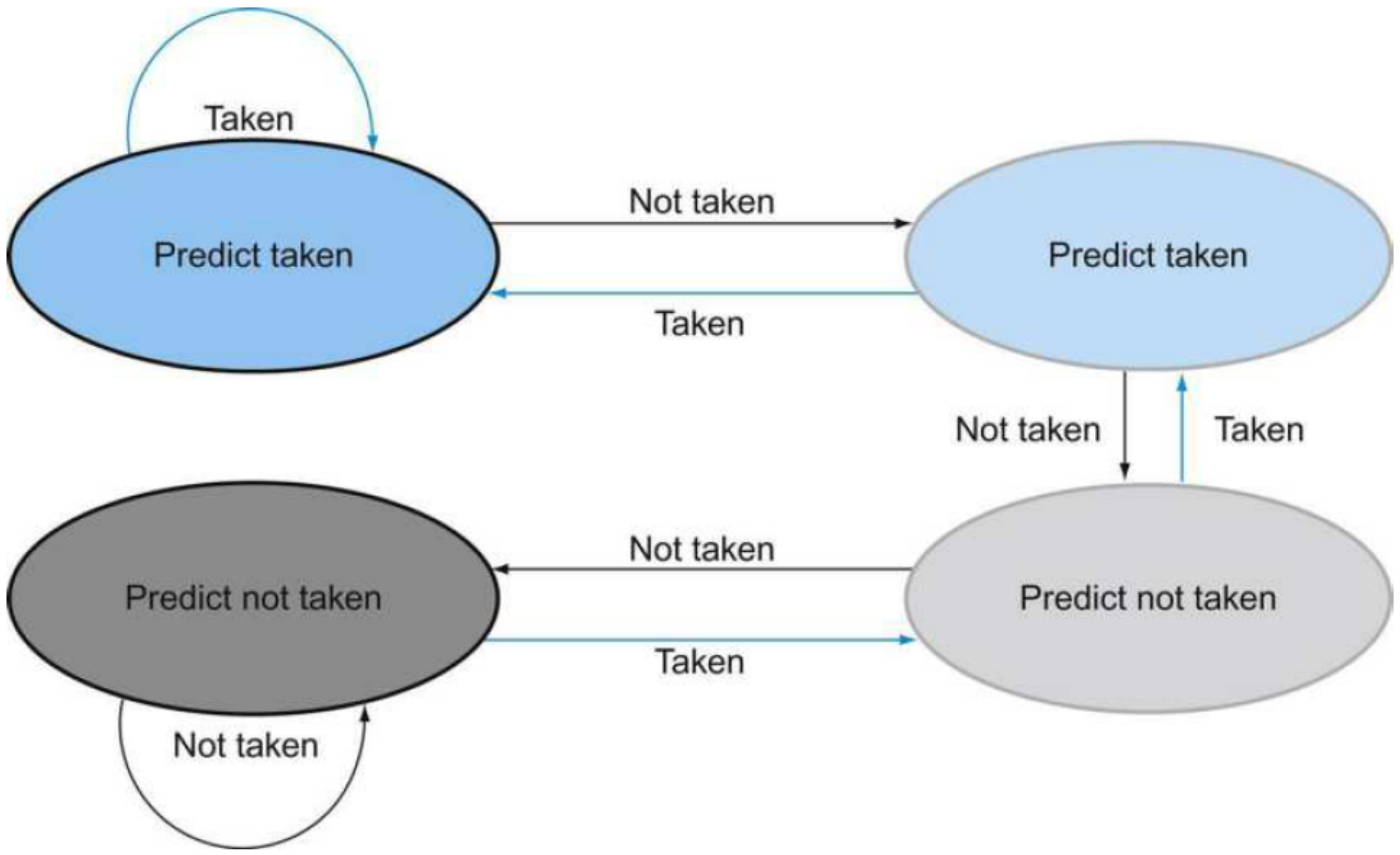


Figure is from “Computer Organization and Design”

Control Hazards

Another Approach to Branch Prediction

Tournament branch predictor:
Use multiple predictors and the prediction result from the best predictor

- This predictor performs multiple predictions for each branch with multiple predictors:
1. Local branch (branch behavior of the same branch instruction)
2. Global branch behavior (branch behaviors among the different branch instruction)

- A selector chooses more accurate predictor among the two predictors

Exceptions

Exception and Interrupt

- Exception or interrupt: unexpected events within the processor or from the outside
- Example of exception: access to undefined memory address or instruction and divide by 0
- Example of interrupt: I/O device request
- In RISC-V, we do not distinguish the two terms

Exceptions

How Exceptions are Handled in the RISC-V Architecture

- Two main reasons for internally caused exception: undefined instruction and hardware malfunction
- The basic action when an exception occurs:
1) Save the address of affected instruction into Supervisor Exception Cause register (SEPC)
2) Transfer control to operating system (OS)
3) The OS takes the appropriate action

Maintaining and reporting the reason of exception to the operating system
- Using Supervisor Exception Cause Register (SCAUSE)
- This register holds a field that indicates the reason for the exception

- To handle the exceptions, OS must know the reason and the instruction that caused exception
- Processor automatically jumps to the exception handling entry point (0x000000001C090000 – instruction memory address for exception handling function)

Exceptions

Exception in the Pipelined Implementation

- A pipelined implementation treats an exception as another form of control hazard
  : We need to jump to the memory address that holds the exception handler instruction

- Two added signal to flush instructions:

1) ID.Flush
   - This is ORed with the stall signal from hazard detection unit
   - It controls the mux to flush instructions in the ID stage
2) EX.Flush
   - This signal controls two multiplexors to zero the control lines in EX stage

Why do we need to flush ID and EX stages?

- Additional input as an address (0x1C090000) for exception to the PC mux
- Save the address of affected instruction into SEPC

Exceptions

Exception in a Pipelined Implementation

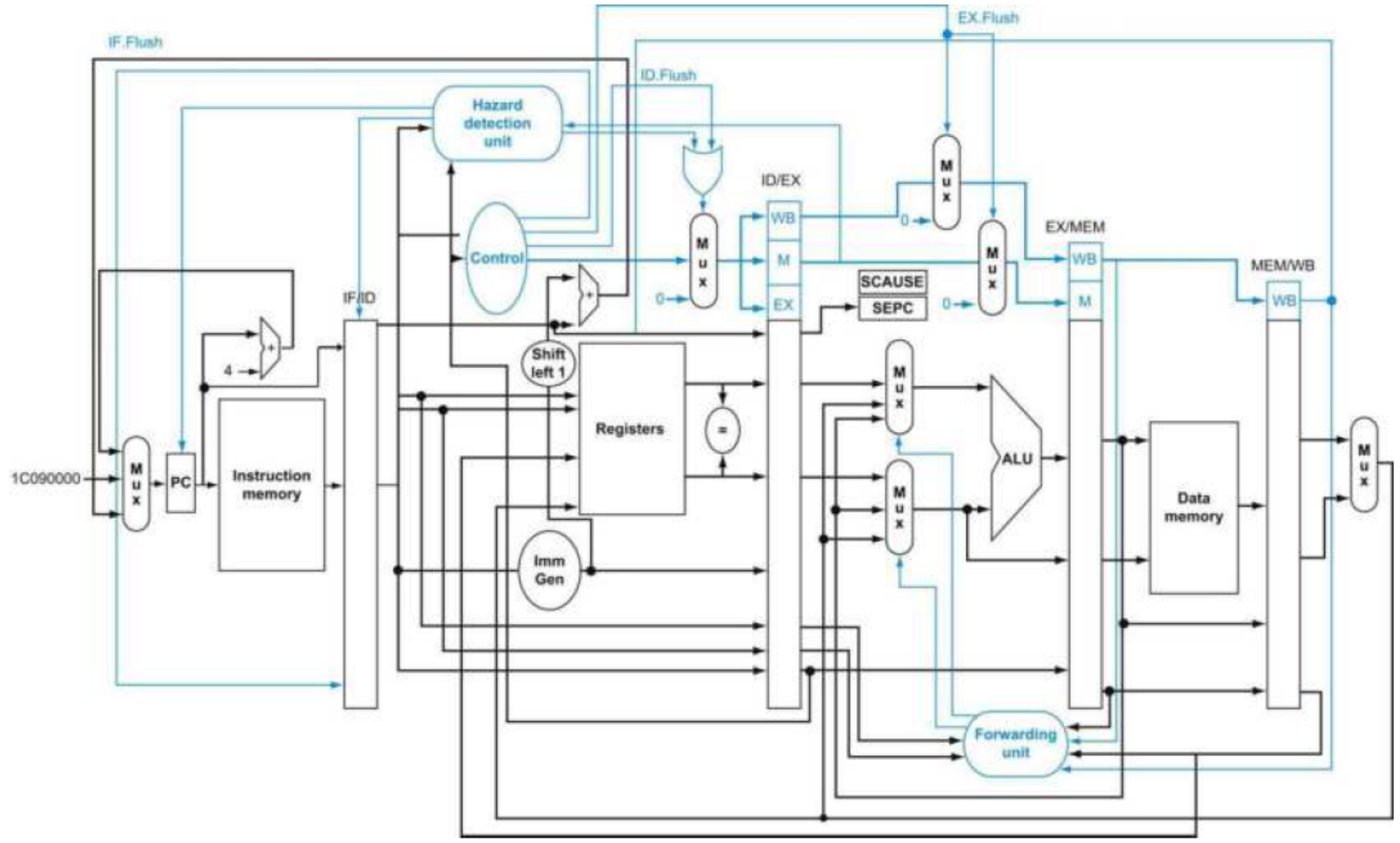


Figure is from “Computer Organization and Design”

Exceptions

Example

Q) Given this instruction sequence,

```
40hex   sub   x11, x2, x4
44hex   and   x12, x2, x5
48hex   or    x13, x2, x6
4Chex   add   x1, x2, x1
50hex   sub   x15, x6, x7
54hex   ld    x16, 100(x7)
. . .
```

assume the exception handler procedure begins like this:

```
1C090000hex   sd   x26, 1000(x10)
1C090004hex   sd   x27, 1008(x10)
. . .
```

Show what happens in the pipeline if a hardware malfunction exception occurs in the add instruction.

Exceptions

Example

In the clock 6, the address of add instruction ($4C_{hex}$) is saved into SEPC and exception handling address ($1C090000_{hex}$) is forced into the PC

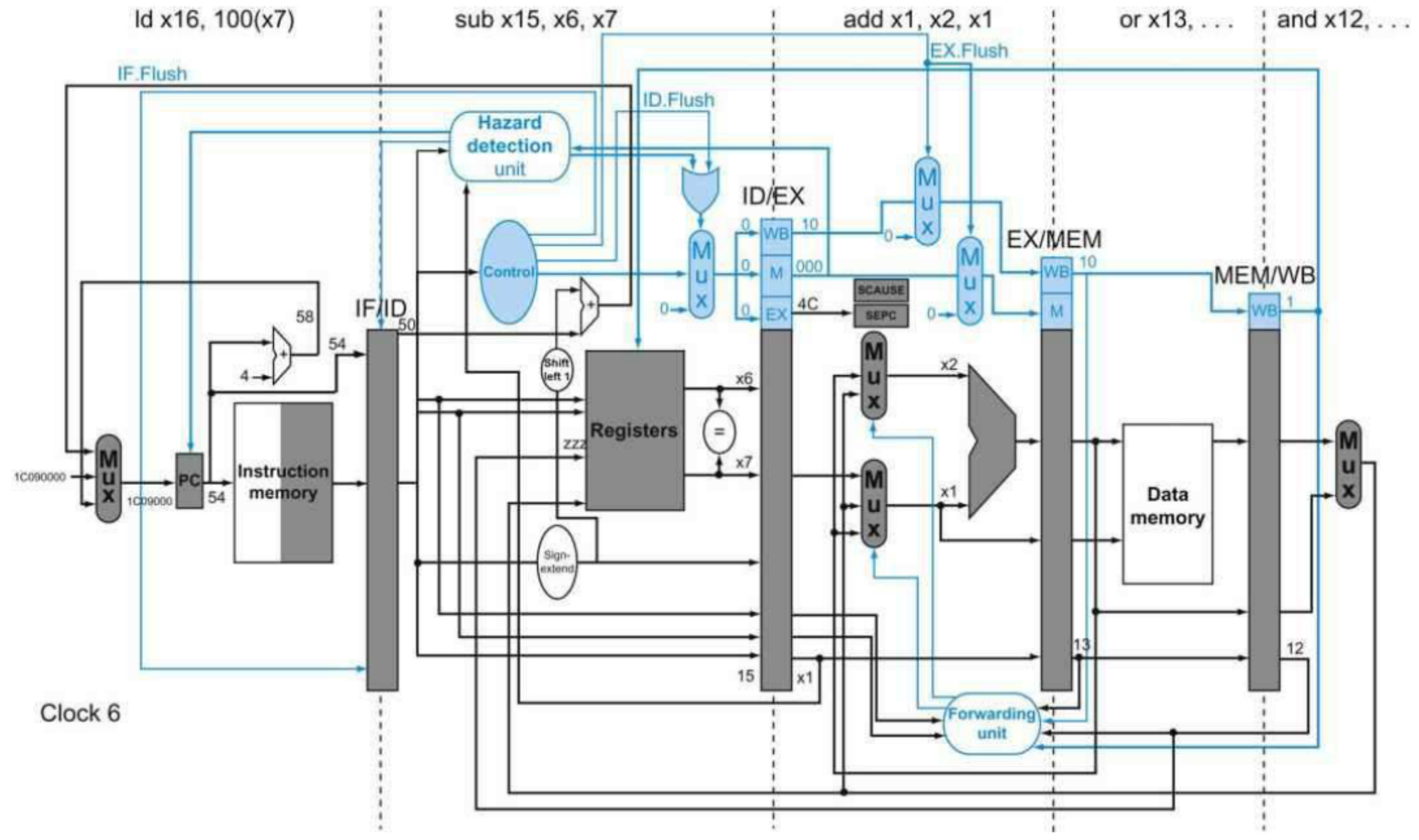


Figure is from "Computer Organization and Design"

Exceptions

Example

In the clock 7, flush the add and following fetched instruction, then the first instruction of the exception handling code is fetched

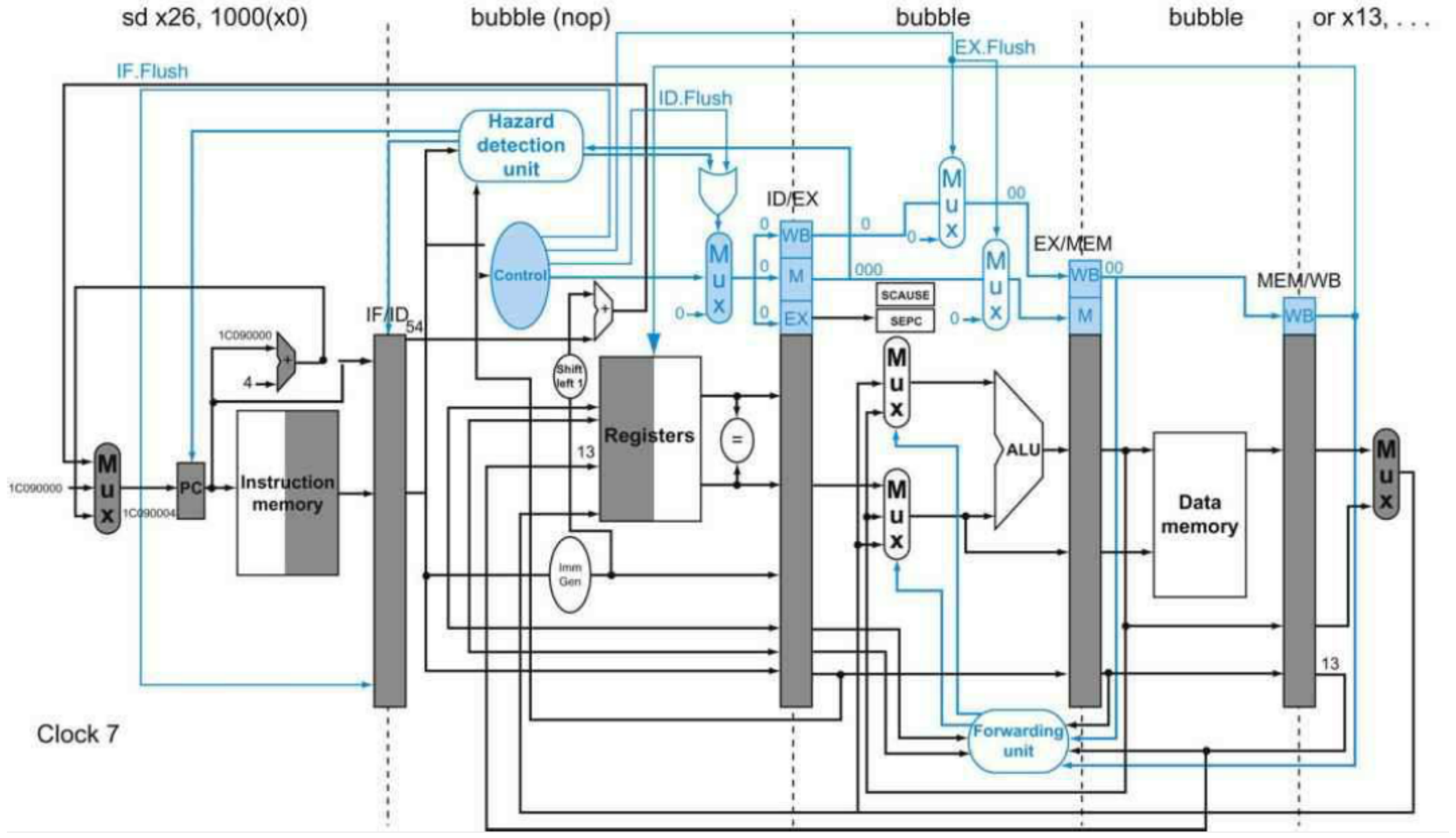
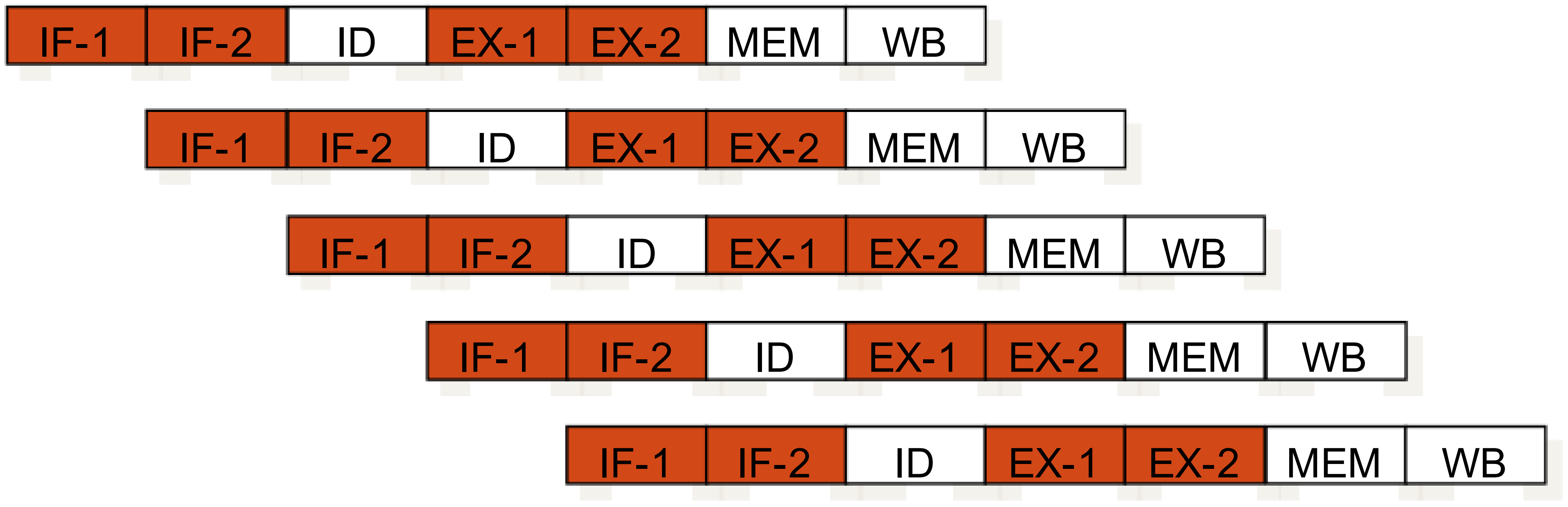


Figure is from "Computer Organization and Design"

Parallelism via Instructions

Instruction-Level-Parallelism (ILP)

- To increase the amount of ILP:
  1. Increasing the depth of the pipeline to overlap more instructions
  → Performance can be improved since the clock cycle time can be shorter
  → More number of the in-flight instructions



Parallelism via Instructions

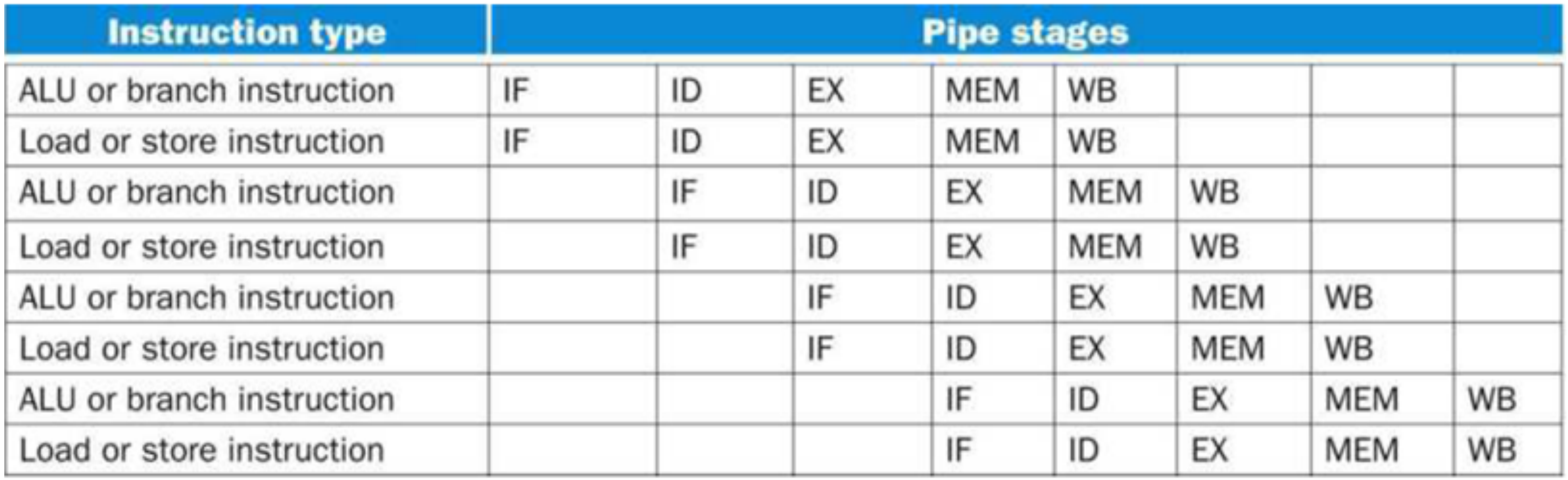
Instruction-Level-Parallelism (ILP)

- To increase the amount of ILP:

  2. Launch multiple instructions in every pipeline stage (Multiple Issue)

  → The CPI can be less than 1

  → But, there are many constraints for multiple issue



Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Figure 4.65 is from a book named “Computer Organization and Design”

Parallelism via Instructions

Multiple Issue Processor

There are two main constraints:
1. What type of instructions are executed simultaneously?
2. What happens when there are dependencies?

To implement a multiple-issue-processor:
1. Packaging instructions into issue slots:
  - Determine how many instructions and which instructions can be issued
  - In static issue processor, the compiler handles it
  - In dynamic issue processor, the processor deals with it at runtime
2. Dealing with data and control hazards:
  - In static issue processors, the compiler handles these hazards statically
  - In dynamic issue processors, the hardware attempts to alleviate the hazards at execution time

Statically scheduled multiple issue processor is also called as **VLIW (Very Long Instruction Word) processor**

Parallelism via Instructions

Static Two-Issue Datapath

- More ports in register file and memory
- More ALUs
- More wires
- Much more complicated
- For dynamic multiple issue processor?

Even much more complicated

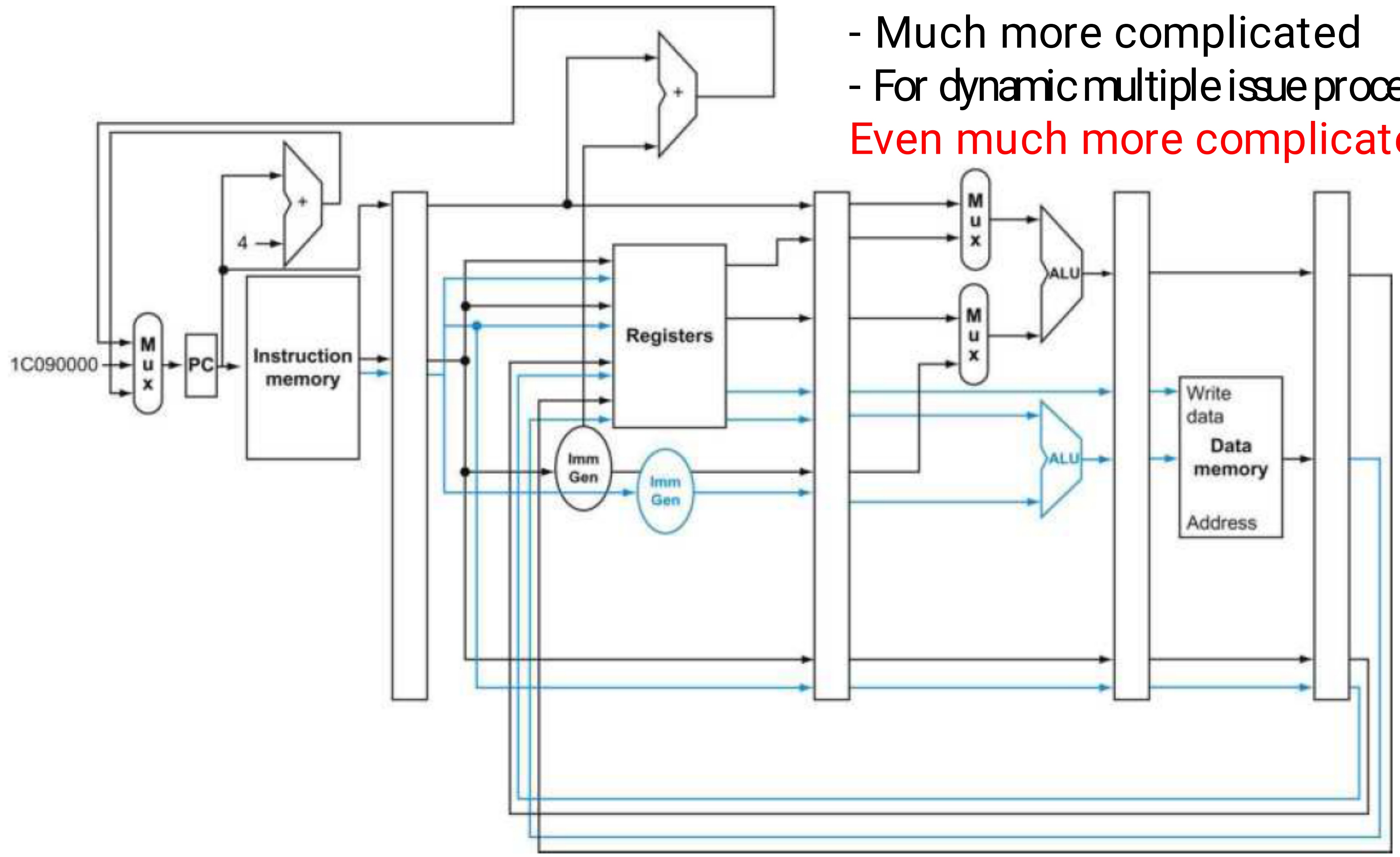


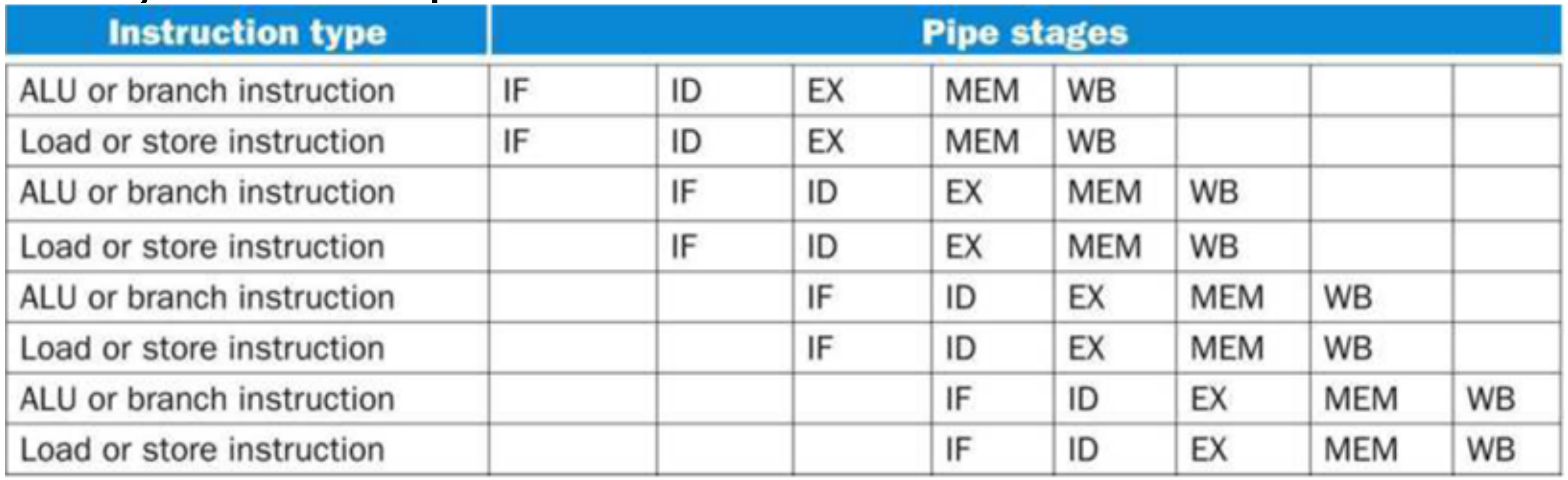
Figure is from "Computer Organization and Design"

Parallelism via Instructions

Multiple-Issue Processor (Static Multiple Issue)

- Static multiple-issue processors rely on the compiler
- The compiler packages instructions and handles hazards

Very Long Instruction Word (VLIW)
- Packet issues as a single instruction
- The two instructions should be paired (like below table) and aligned on a 64-bit boundary
  → possibly with a "nop" for stall in one slot



Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Figure is from "Computer Organization and Design"

Parallelism via Instructions

Example of Static Multiple Issue

Q) How would this loop be scheduled on a static two-issue pipeline for RISC-V?

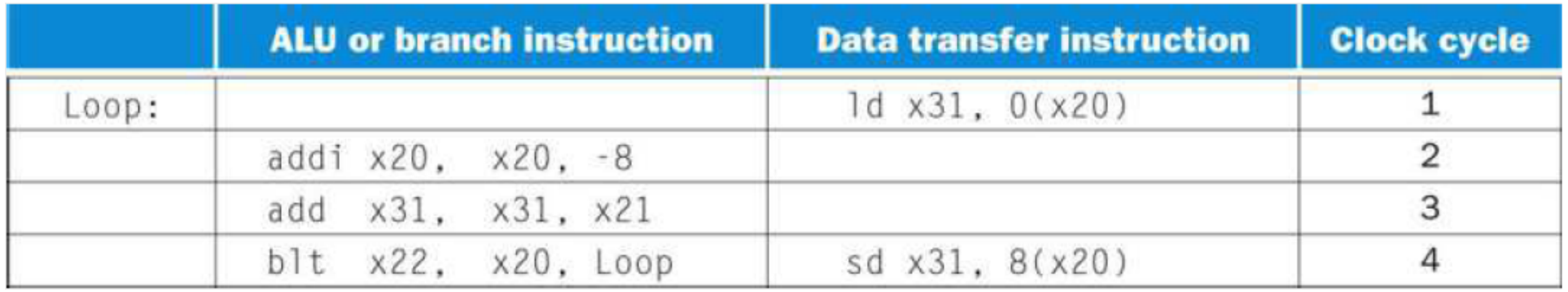
```
Loop:   ld      x31, 0(x20)      // x31=array element
        add     x31, x31, x21    // add scalar in x21
        sd      x31, 0(x20)      // store result
        addi    x20, x20, -8     // decrement pointer
        blt     x22, x20, Loop   // compare to loop limit, branch if x20 > x22
```

Reorder the instructions to avoid as many pipeline stalls as possible. Assume branches are predicted, so that control hazards are handled by the hardware.

Parallelism via Instructions

Example of Static Multiple Issue

A)
- The best schedule of the instructions is represented in below table
- The ideal CPI is 0.5 for two-issue processor. In this time, however, the CPI is 0.8
  → In calculating CPI or IPC, nops are not counted



	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		ld x31, 0(x20)	1
	addi x20, x20, -8		2
	add x31, x31, x21		3
	blt x22, x20, Loop	sd x31, 8(x20)	4

Figure is from “Computer Organization and Design”

Parallelism via Instructions

The Concept of Speculation

Speculation: A method for finding and exploiting more ILP

- Based on the prediction, speculation guesses the properties of an instruction
- Assuming the guesses are correct, processor executes instructions anyway
- Typically used in branch prediction

- This mechanism must include:
  1) Check if the guess was right or not
  2) Unroll or back out the effects of the speculated instructions (complex)

Parallelism via Instructions

The Concept of Speculation

Recovery mechanism: This method is for incorrect speculation

In software speculation:
- Compiler adds some instructions to check the accuracy of the speculation
- Also, compiler provides a fix-up routine for incorrect speculation

In hardware speculation:
- The processor buffers the speculative results

The behavior for mis-speculation:
- The hardware flushes the buffers and re-executes the correct instruction

→ Speculation can improve the performance when it is done properly
→ When the prediction is wrong, the execution through wrong paths causes energy wastes

Parallelism via Instructions

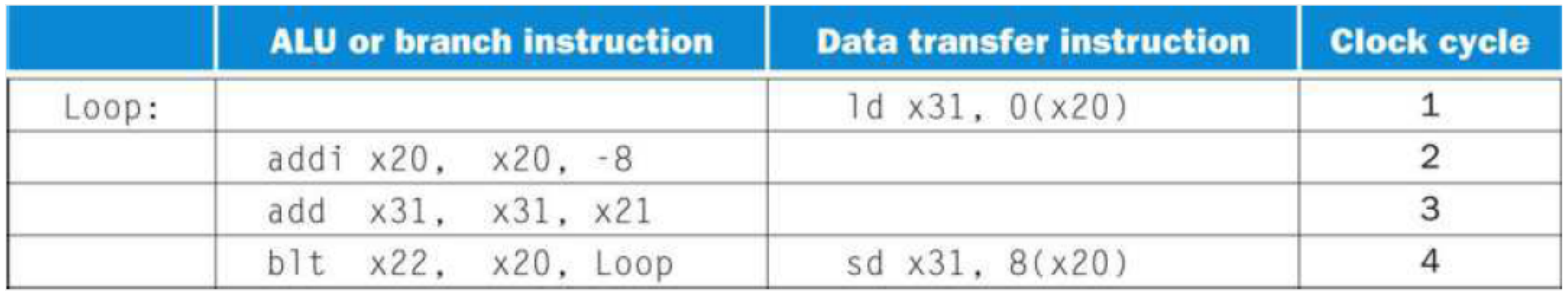
Loop Unrolling Technique

Loop unrolling:
Important technique for compiler to get more performance from loops

After unrolling, there is more ILP available since we remove branch instructions and data dependencies

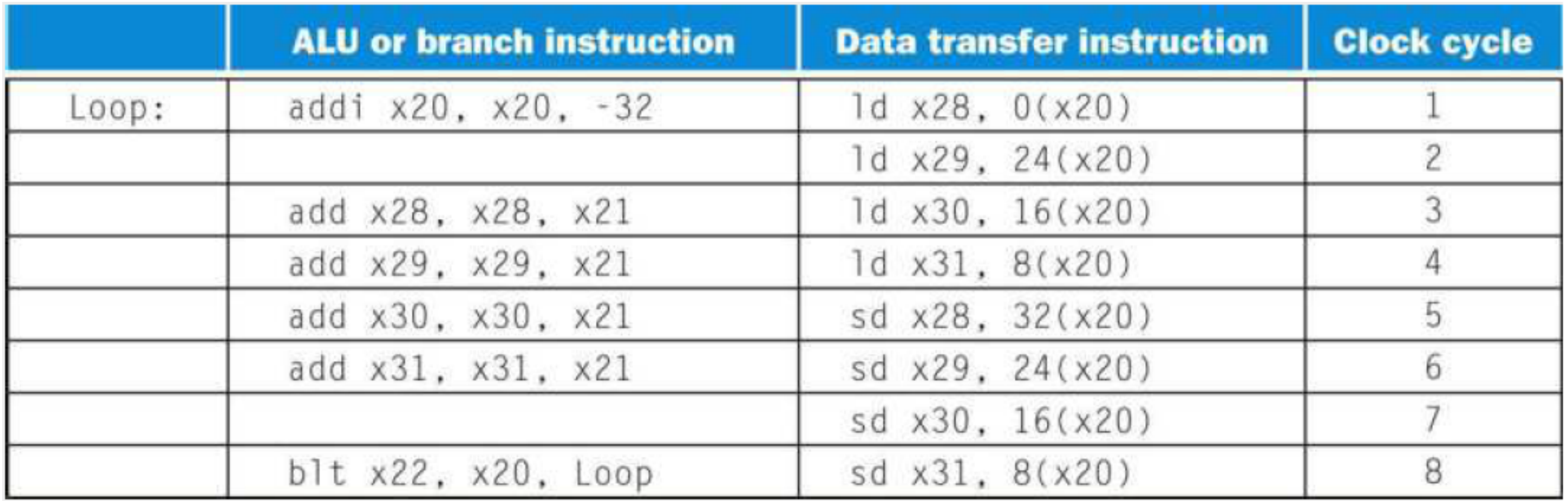
Example of Loop Unrolling

Q) See how well loop unrolling and scheduling work in the previously example. For simplicity, assume that the loop index is a multiple of four.



	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		`ld x31, 0(x20)`	1
	`addi x20, x20, -8`		2
	`add x31, x31, x21`		3
	`blt x22, x20, Loop`	`sd x31, 8(x20)`	4

Parallelism via Instructions



	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20, x20, -32	ld x28, 0(x20)	1
		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
		sd x30, 16(x20)	7
	blt x22, x20, Loop	sd x31, 8(x20)	8

A)

- The compiler uses additional registers (x28, x29, x30)
  → To remove data dependences
  → To better schedule the code
- 12 of the 14 instructions are executed as pairs
- The IPC of four loop iterations is 14/8 = 1.75 (CPI is 0.57)

→ Loop unrolling results more than doubled performance (8 vs 20 clock cycles)
→ The cost of performance improvement is using only four temporary registers than one

Parallelism via Instructions

Dynamic Multiple-Issue Processors

Superscalar processor:

An advanced pipelining technique that can execute **more than one instruction per clock cycle (dynamically scheduling the codes in runtime)**

Dynamic Pipeline Scheduling:

Chooses the instructions to execute in a given clock cycle while trying to avoid hazards and stalls

- How this instructions can be scheduled to avoid stall?

```
ld      x31, 0(x21)
add     x1, x31, x2
sub     x23, x23, x3
addi    x5, x23, 20
```

How about replacing add and sub? – dynamic pipeline scheduling can do this in runtime

Parallelism via Instructions

Dynamic Multiple-Issue Processors

- Superscalar processor consists of three major units
  1. Instruction fetch and issue unit
  2. multiple functional units with reservation station
  3. commit unit

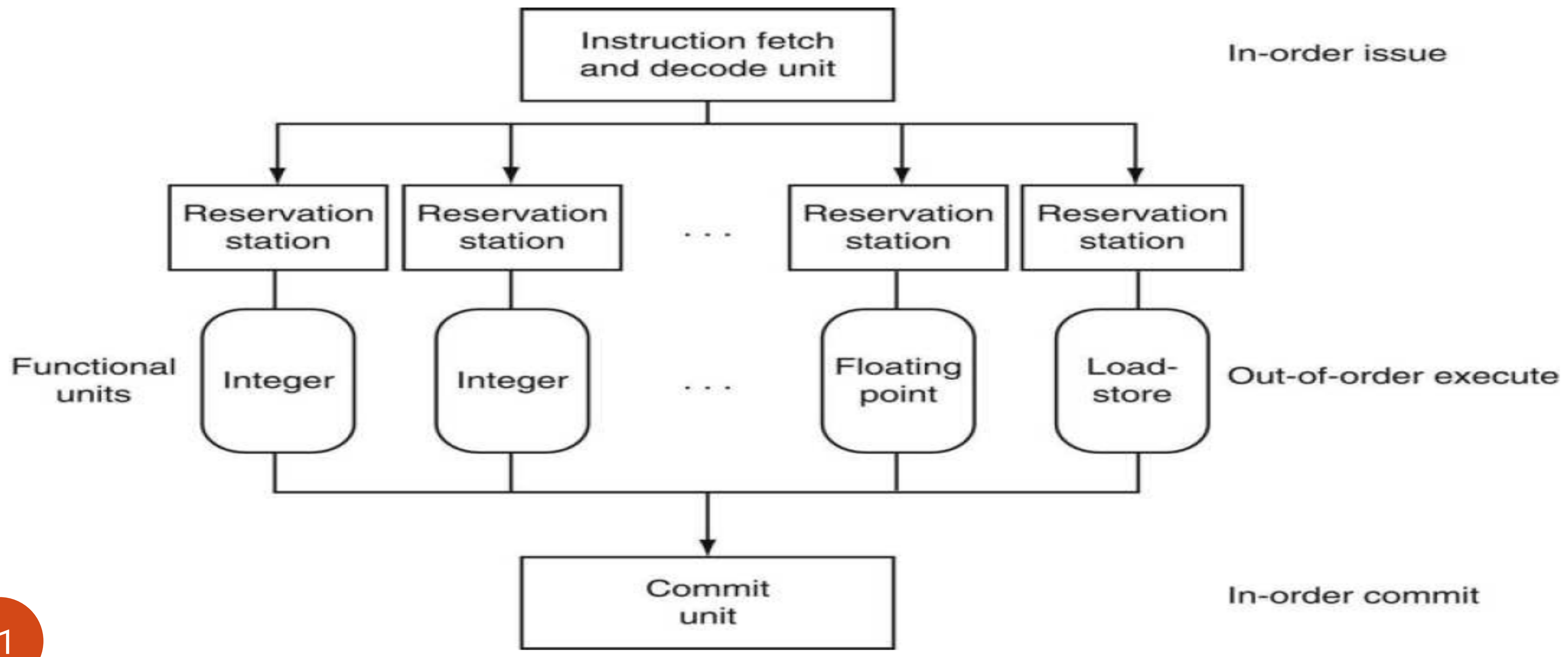


Figure is from "Computer Organization and Design"

Parallelism via Instructions

Dynamic Multiple-Issue Processors

1. Instruction fetch and issue unit

- It fetches instructions, decodes them, and sends each instruction to a corresponding functional unit's **reservation station** (it is called as 'issue')

2. Multiple functional units

- Each functional unit has buffers (**reservation stations**) to hold the operands and the operation
- If an instruction is ready to execute, the instruction is executed
  Ready to execute means: available functional units, all dependencies are resolved
- When the execution is completed, it is sent to the commit unit

3. Commit unit

- Commit unit buffers the result until it is safely put into register file or memory
- The buffer is also called as Reorder Buffer (ROB)
- The buffer in the commit unit is also used to supply operands like forwarding logic

Parallelism via Instructions

The Execution steps of Dynamic Multiple-Issue Processors

When an instruction is issued,

1. The instruction is copied to a reservation stations
2. Any operands for the instruction are also copied from register file or reorder buffer to the reservation station
2-1. But if an operand is not produced yet from the previous instructions (data hazard) or all functional units are busy (structural hazard)
    It must stall until a functional unit produces a result for a stalling reservation station
2-2. If all operands are ready for a certain instruction, then the processor executes the instructions
    Processor does not care the instruction order defined in the program
    This means the program execution can be out-of-order

Parallelism via Instructions

Dynamically Scheduled Pipeline

Out-of-order execution
A processor executes instructions in an order governed by the availability of input data and execution units, rather than by their original order in a program **AS LONG AS THE RESULT OF THE PROGRAM IS NOT AFFECTED**

In-order commit
The commit unit writes results to registers and memory in the program order
- Committing instructions means that the results are updated in register file or memory (visible to outside)
- Out-of-order commit is very hard to recover the register and memory; hence it is not used

Parallelism via Instructions

Energy Efficiency and Advanced Pipelining

- Increasing exploitation of ILP (instruction level parallelism) via dynamic multiple issue and speculation may be inefficient for power consumption
- It requires more transistors
- This leads to the power wall and thermal problems

- That’s why the main processor vendors turned into multi-core design
- Single-core processor performance improvement has limitations
- But, performance improvement via multi-core is not easy. Why?