Project 1: Pipelined CPU using verilog

TA: 陳炫均

Announcement

- Individual Project
- Project Deadline: 05/25 (Tue.) 23:59

- Demo:
  - Time slot: TBD
  - Show the execution of your program to TA and answer a few questions

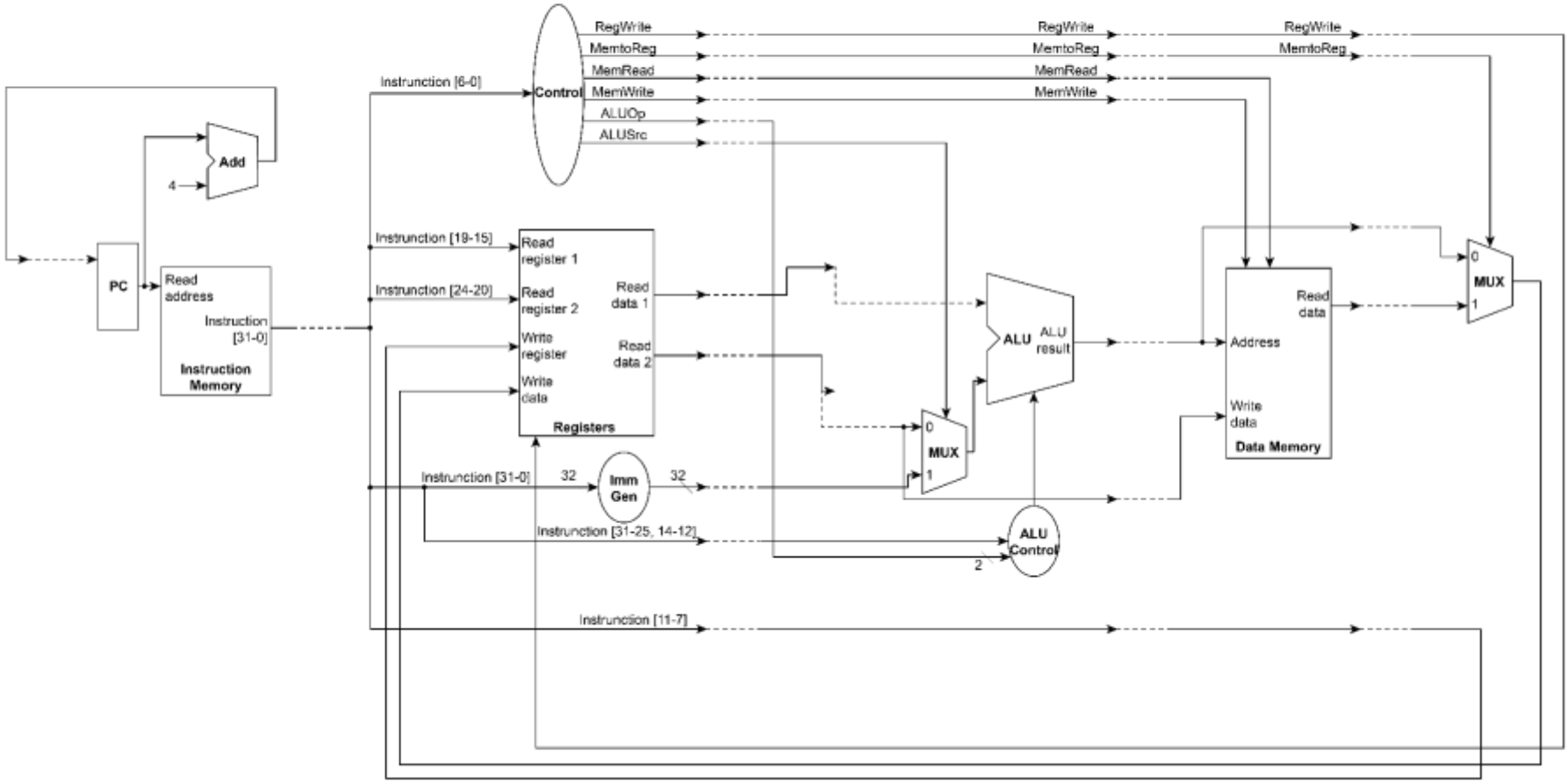
Instructions

- Required Instruction Set
  - and, xor, sll, add, sub, mul, addi, srai
  - lw
  - sw
  - beq

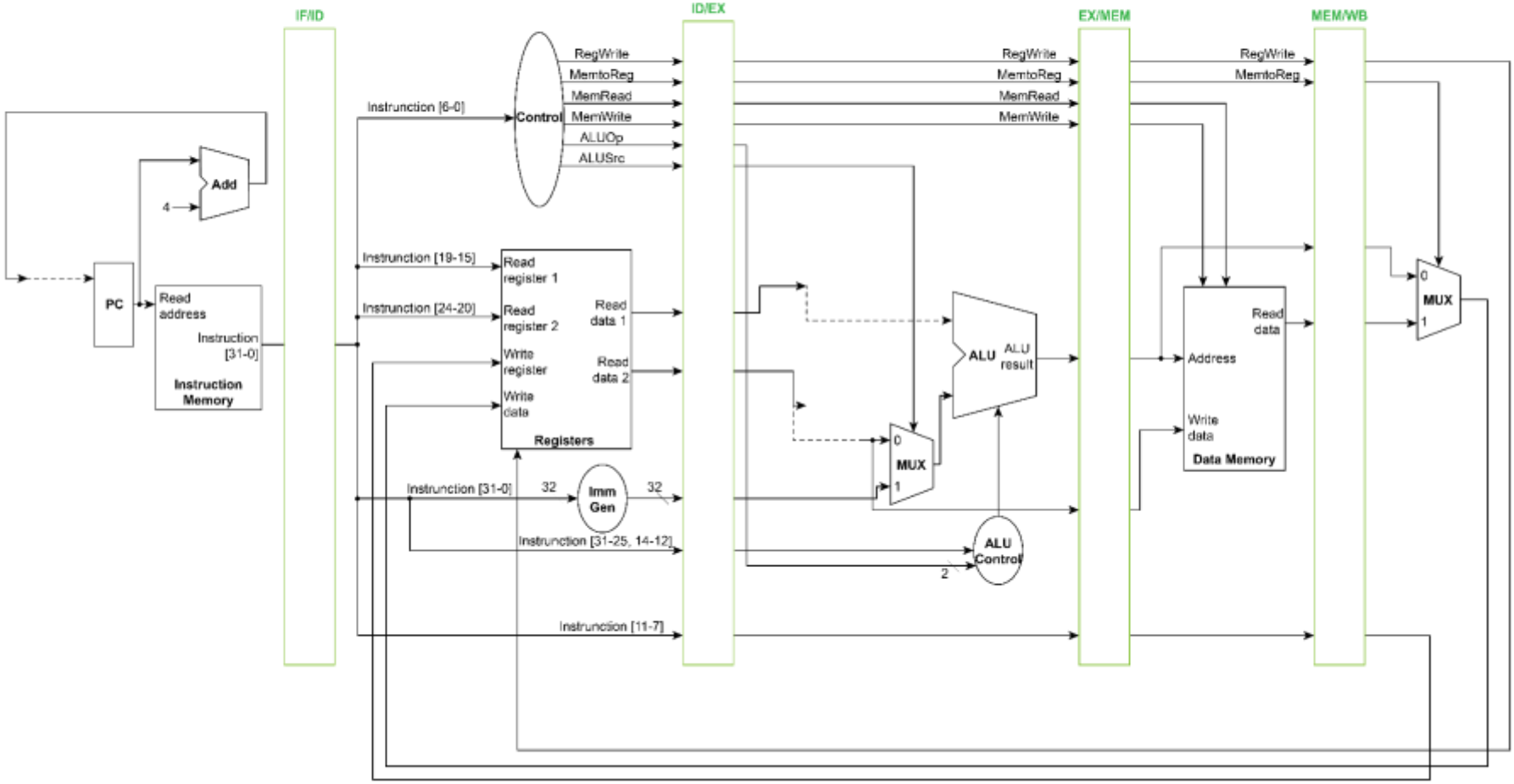
Hardware Specification

- Register File: 32 Registers (Write at the rising edge of the clock), 32-bit
- Instruction Memory: 1KB
- Data Memory: 4 KBytes
- 5-stage pipeline (IF, ID, EX, MEM, WB)
- Hazard handling
  - Data hazard
    - Implement the forwarding unit to reduce or avoid the stall cycles
    - The data dependency instruction following `lw` must stall 1 cycle
    - No need to forward to ID stage
  - Control hazard
    - The instruction following `beq` instruction may need to stall 1 cycle
    - Pipeline Flush

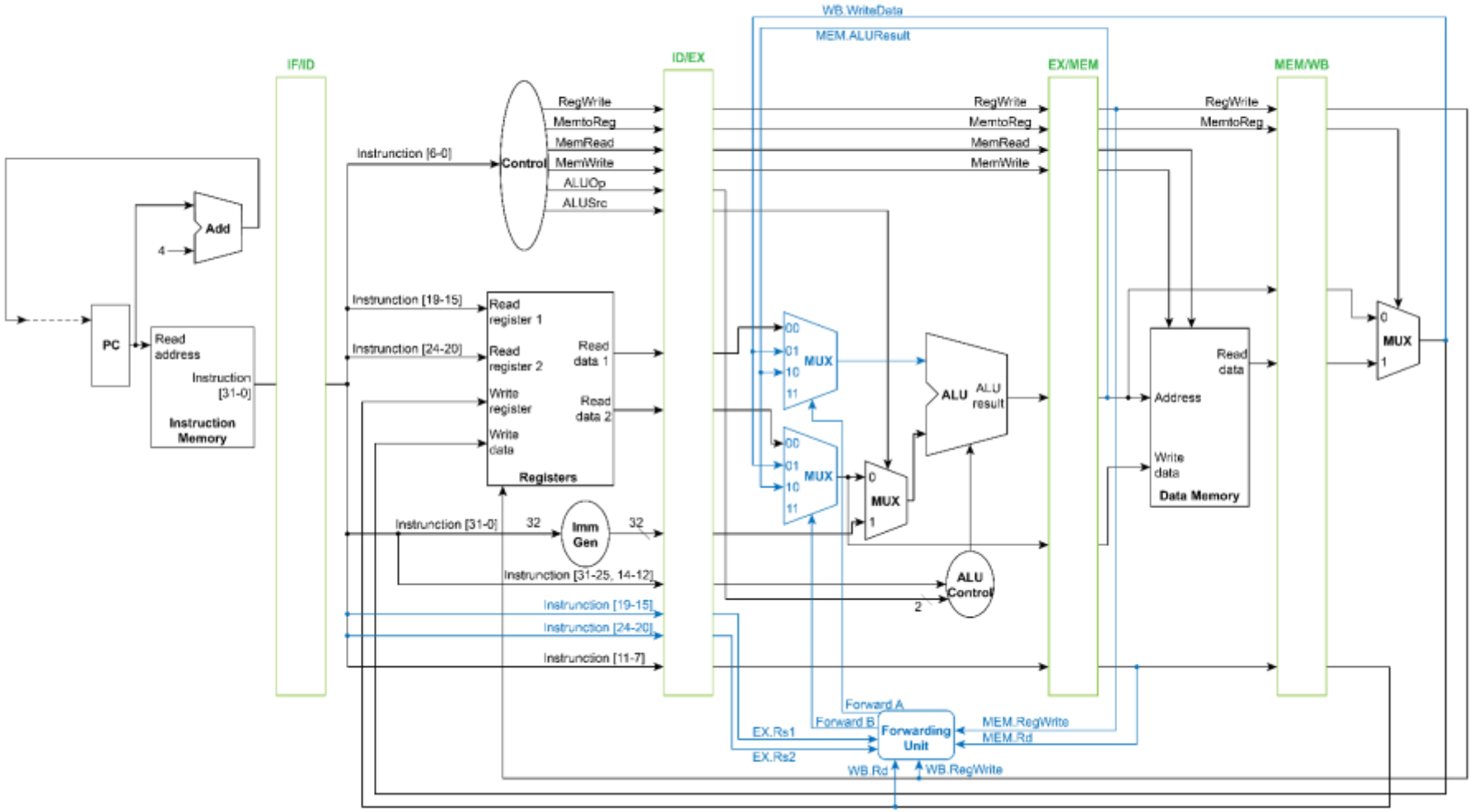
Adding Data Memory



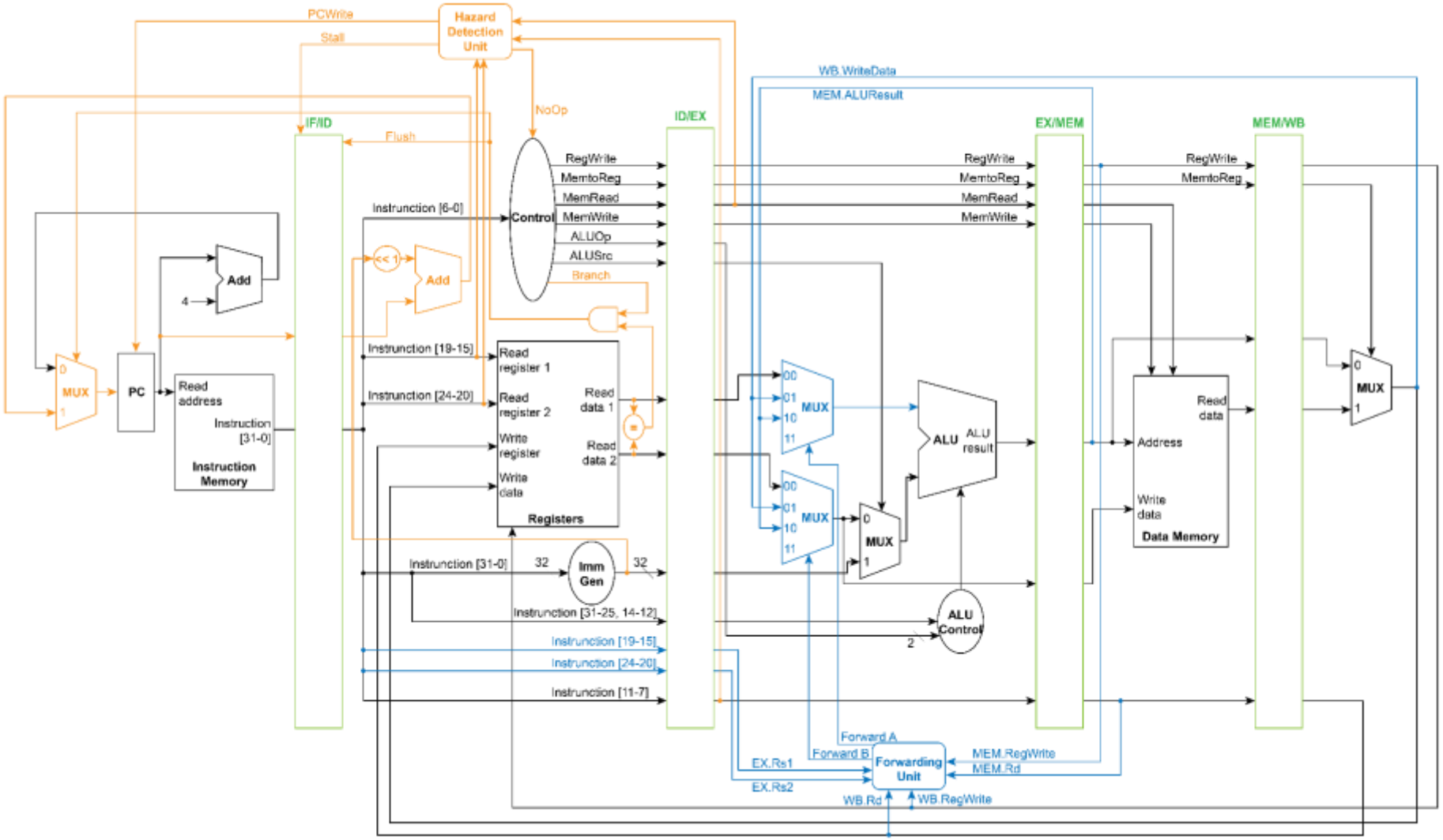
Adding Pipeline Register



Forwarding Control



Handling Branch and Hazard



Control Signals

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

FIGURE 4.45 A copy of Figure 4.12. This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different opcodes for the R-type instruction.

Control Signals

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

FIGURE 4.47 The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.

Machine Code

funct7	rs2	rs1	funct3	rd	opcode	function
0000000	rs2	rs1	111	rd	0110011	and
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000001	rs2	rs1	000	rd	0110011	mul
imm[11:0]		rs1	000	rd	0010011	addi
0100000	imm[4:0]	rs1	101	rd	0010011	srai
imm[11:0]		rs1	010	rd	0000011	lw
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw
imm[12,10:5]	rs2	rs1	000	imm[4:1,11]	1100011	beq

Be careful on your Imm_Gen !

Branch Address

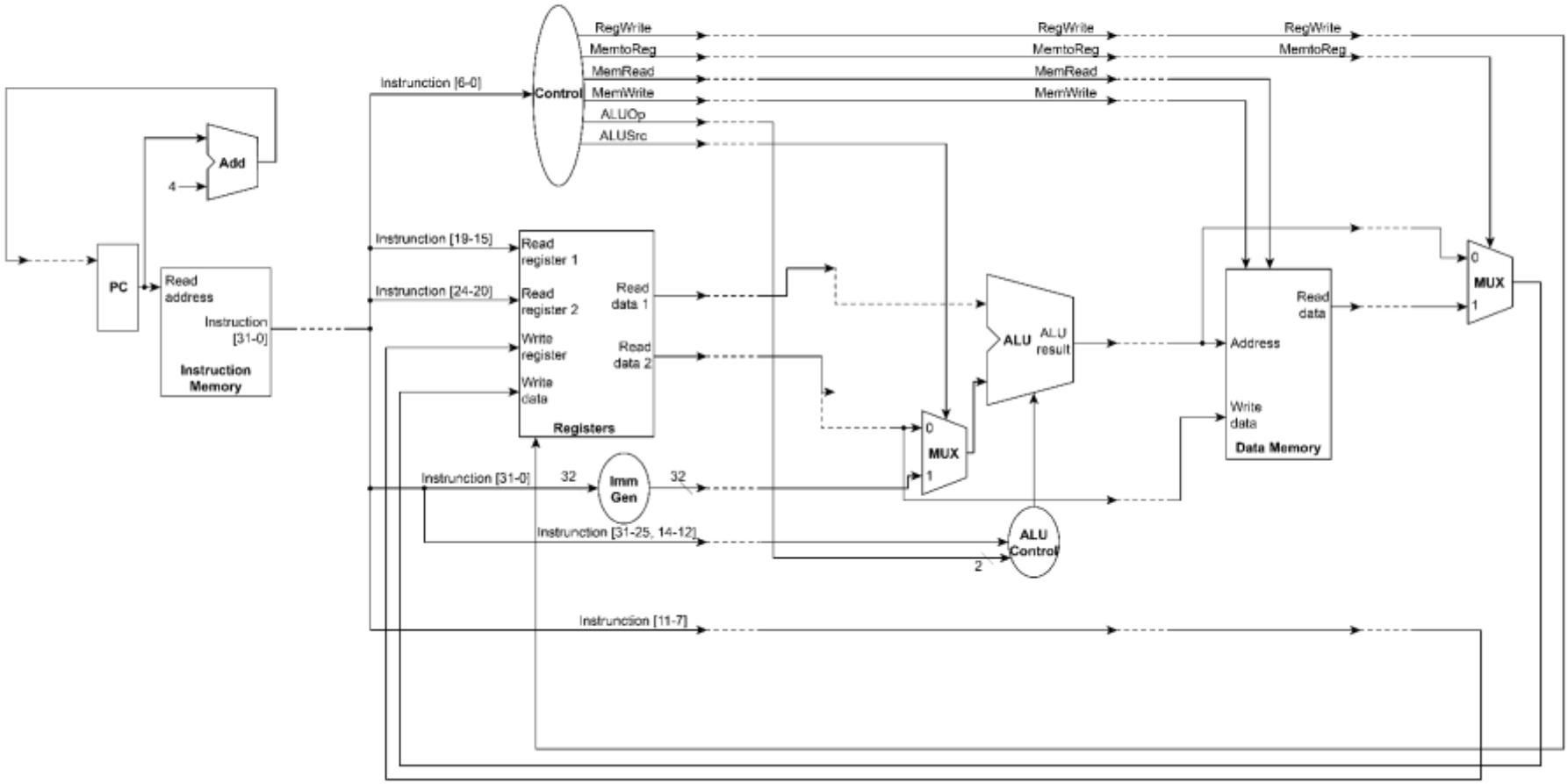
bne x10, x11, 2000 // if x10 != x11, go to location 2000_{ten} = 0111 1101 0000

0	111110	01011	01010	001	1000	0	~~1100111~~ 1100011
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

(The opcode in textbook conflicts with RISC-V spec. Please follow this one)

next PC = current PC + Branch offset

Adding Data Memory



Pipeline Register

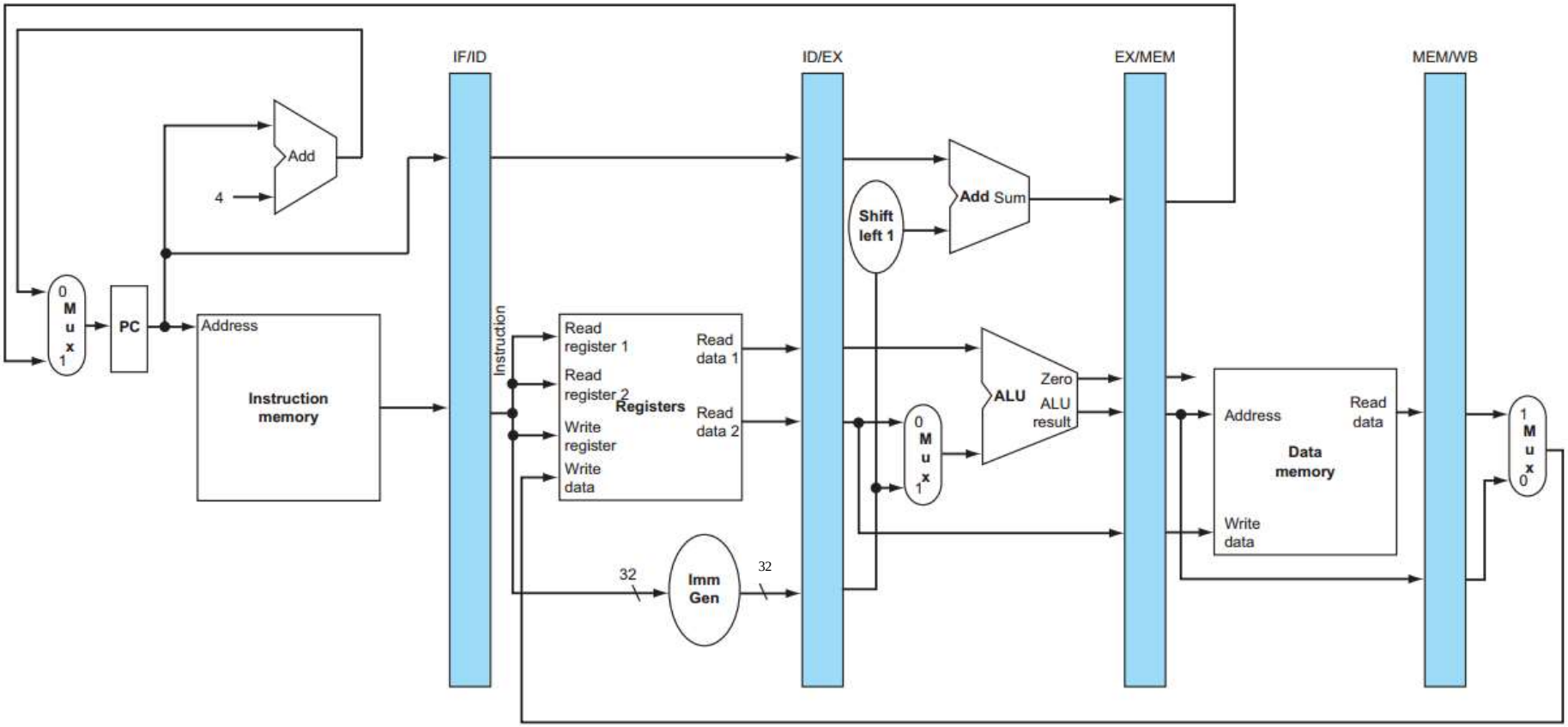
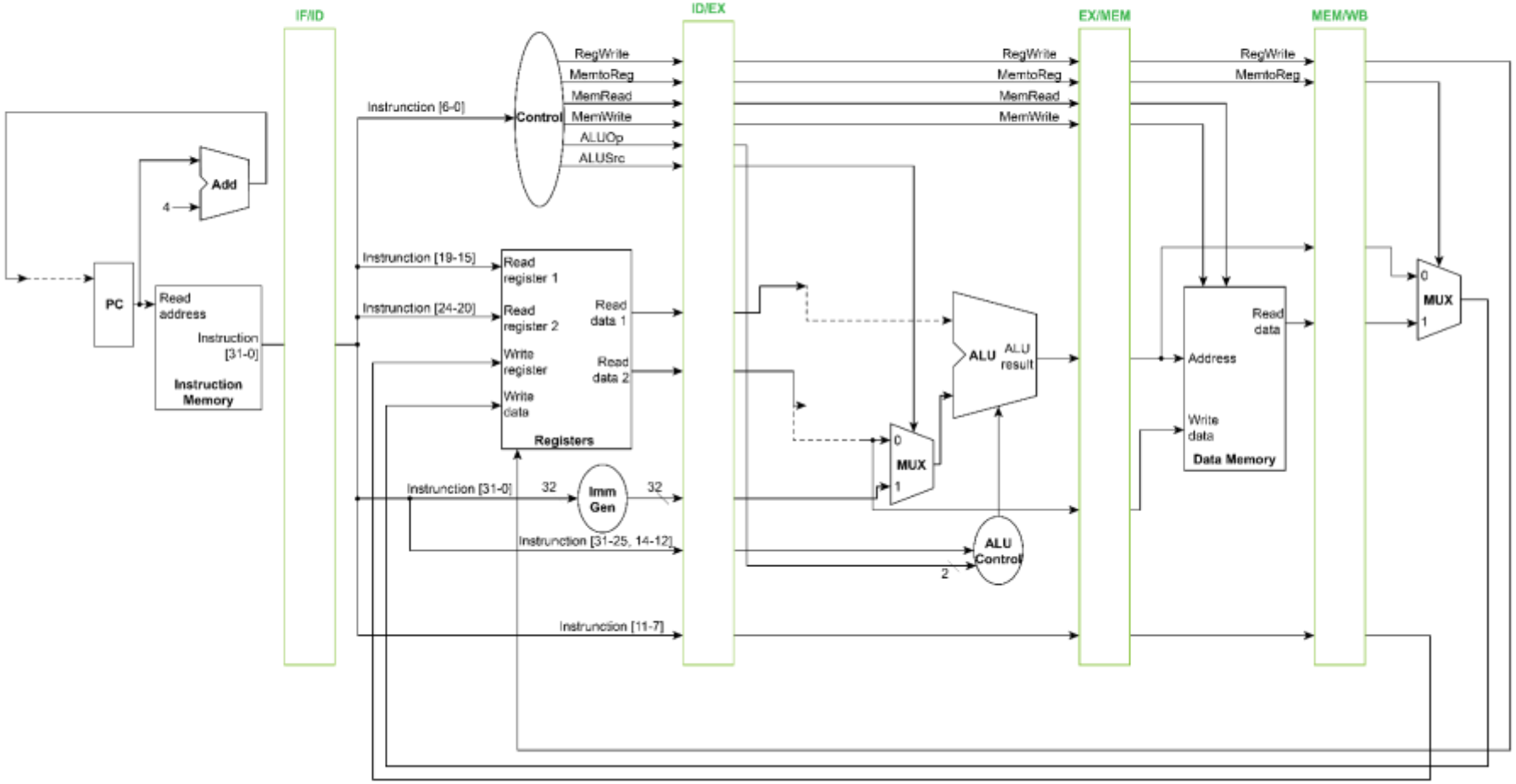


FIGURE 4.33 The pipelined version of the datapath in Figure 4.31. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 bits, respectively.

Adding Pipeline Register



Data Hazard and Forwarding

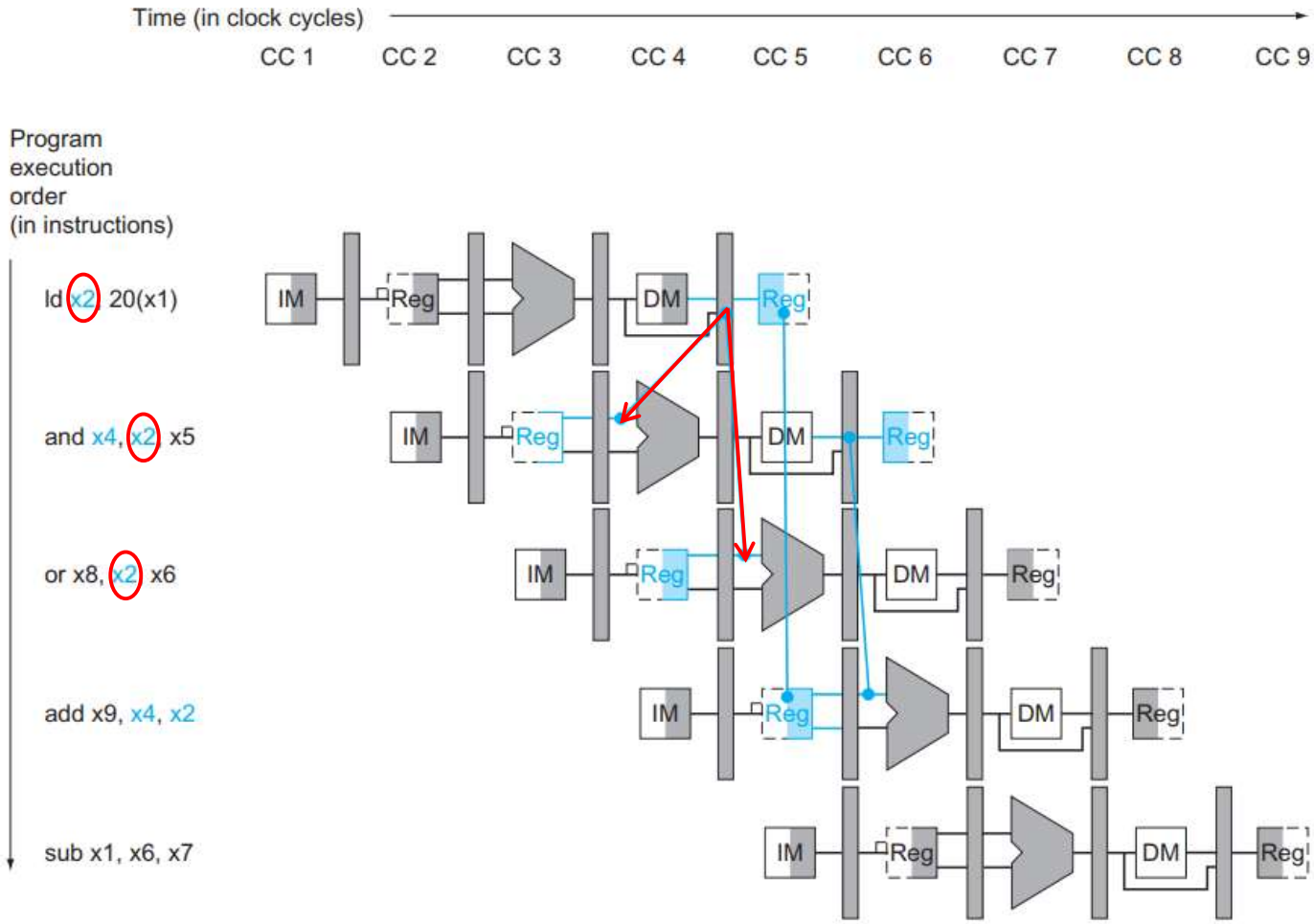


FIGURE 4.56 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

Data Hazard and Forwarding

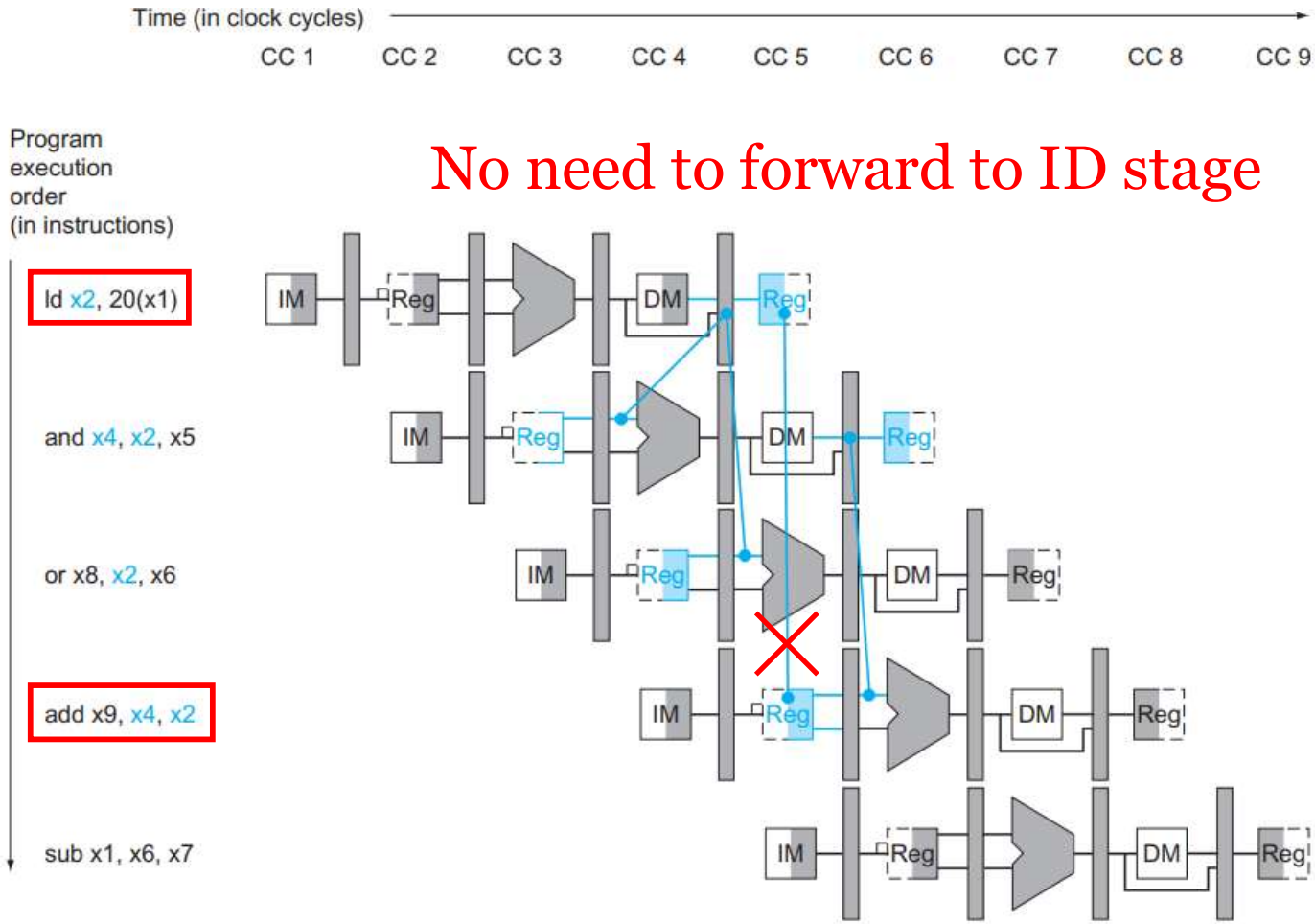


FIGURE 4.56 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

ID Stage Forwarding

```
add x5, x6, x7
beq x5, x4, BRANCH
```

such instruction sequence will not be in our evaluation test cases

Forwarding Unit

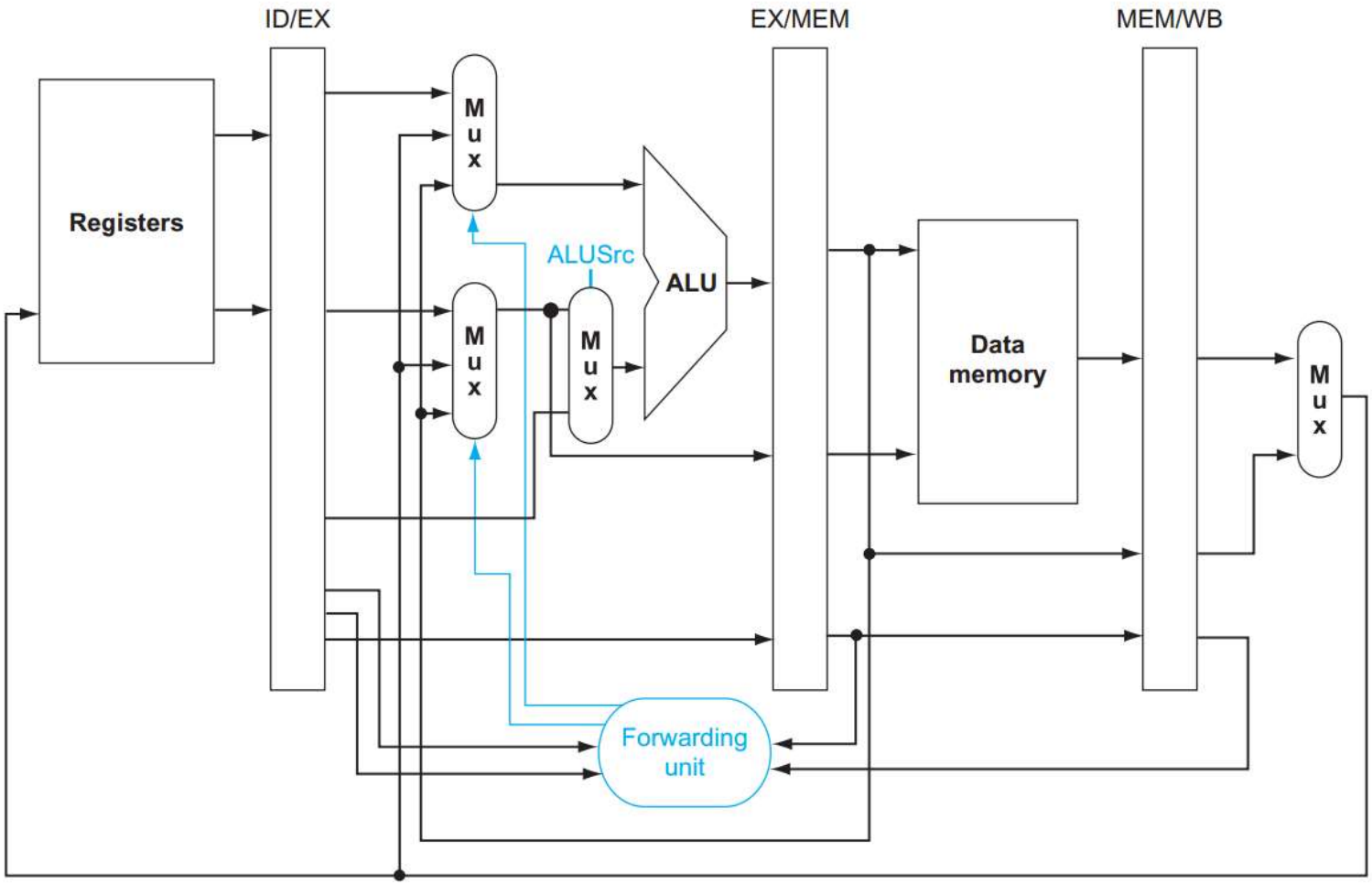


FIGURE 4.55 A close-up of the datapath in Figure 4.52 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

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.

FIGURE 4.53 The control values for the forwarding multiplexors in Figure 4.52. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

Forwarding Control

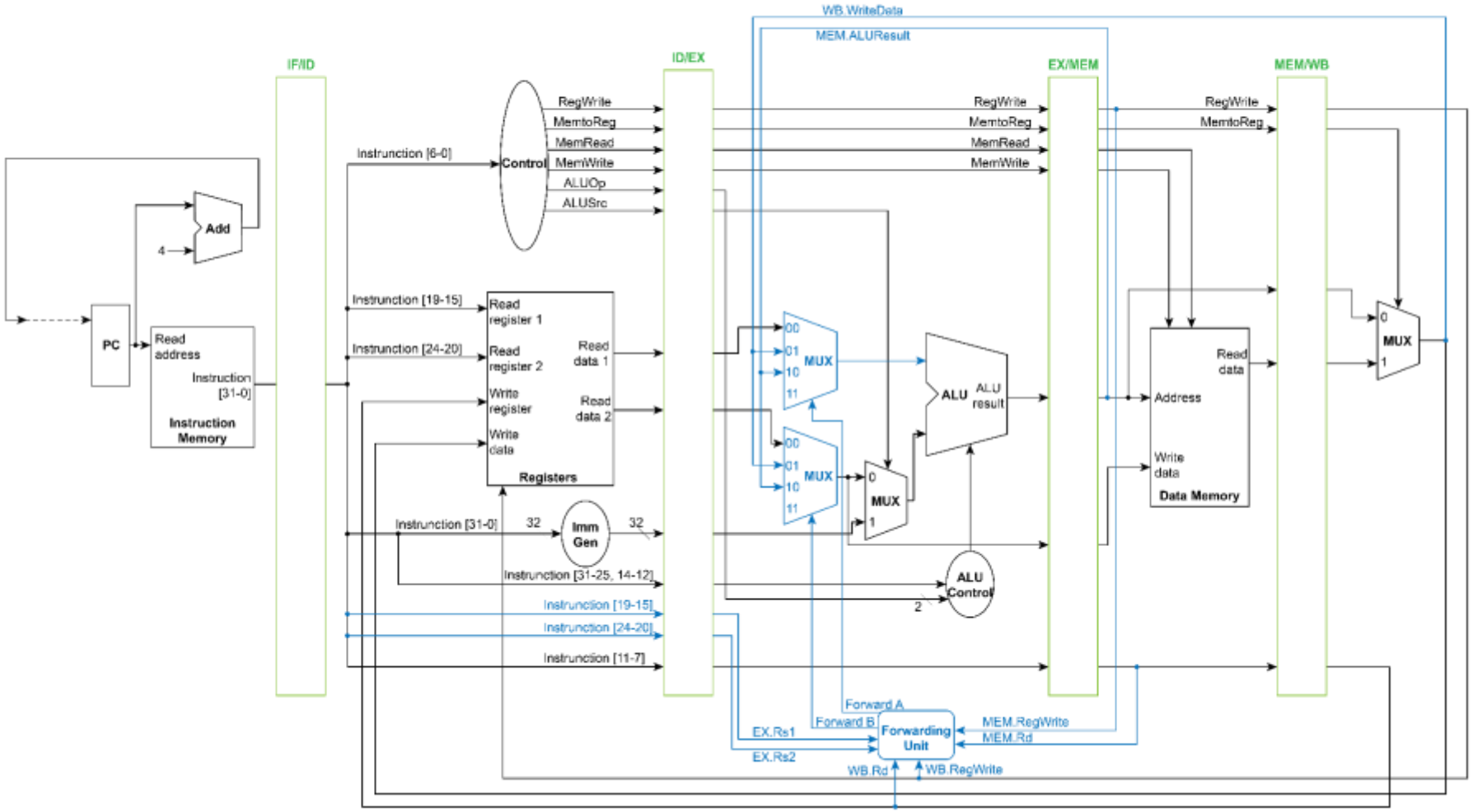
1. *EX hazard:*

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs1)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

2. *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)) 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)) ForwardB = 01
```

Forwarding Control



Hazard Detection and Stall

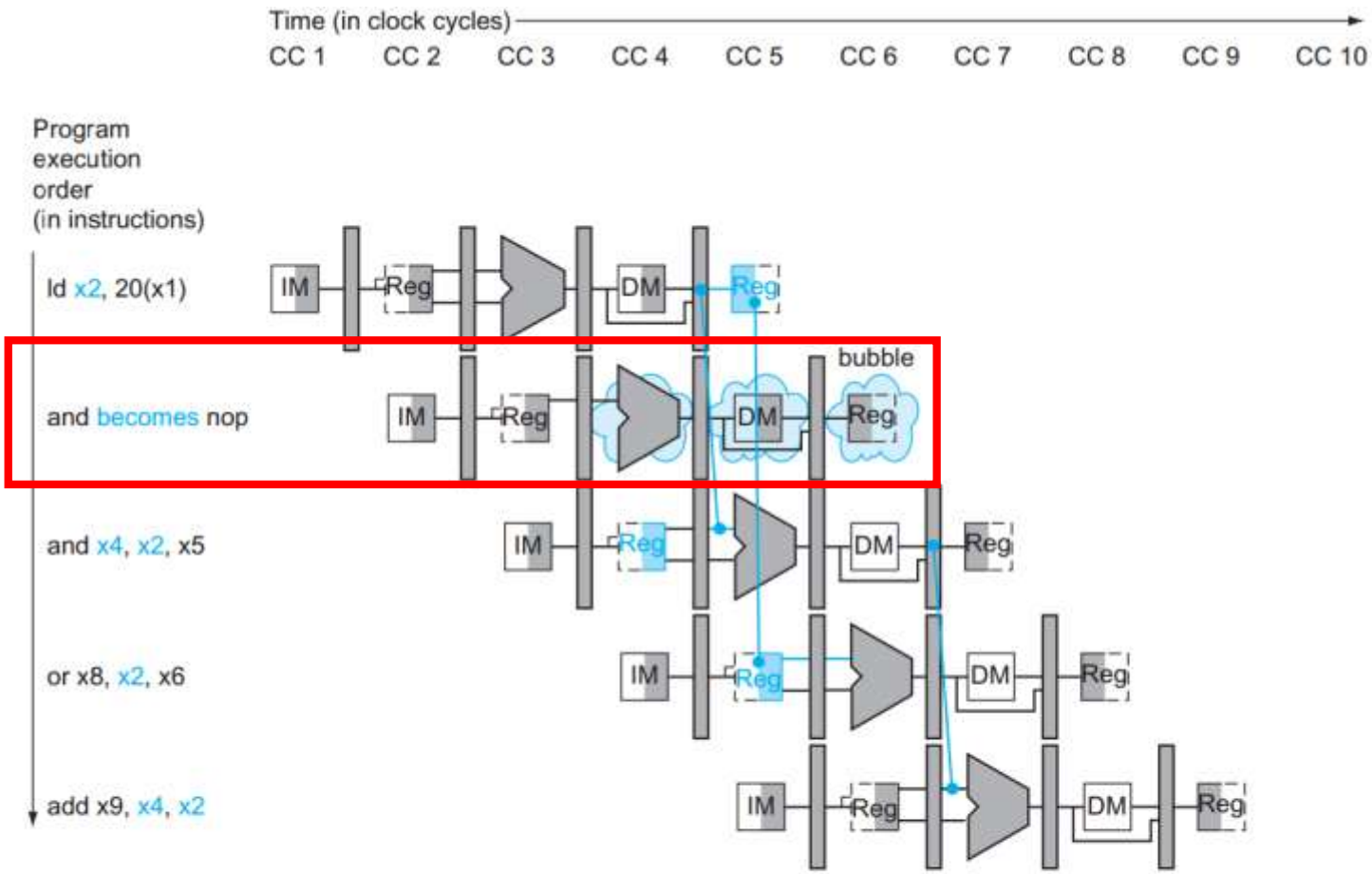


FIGURE 4.57 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise, the `or` instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

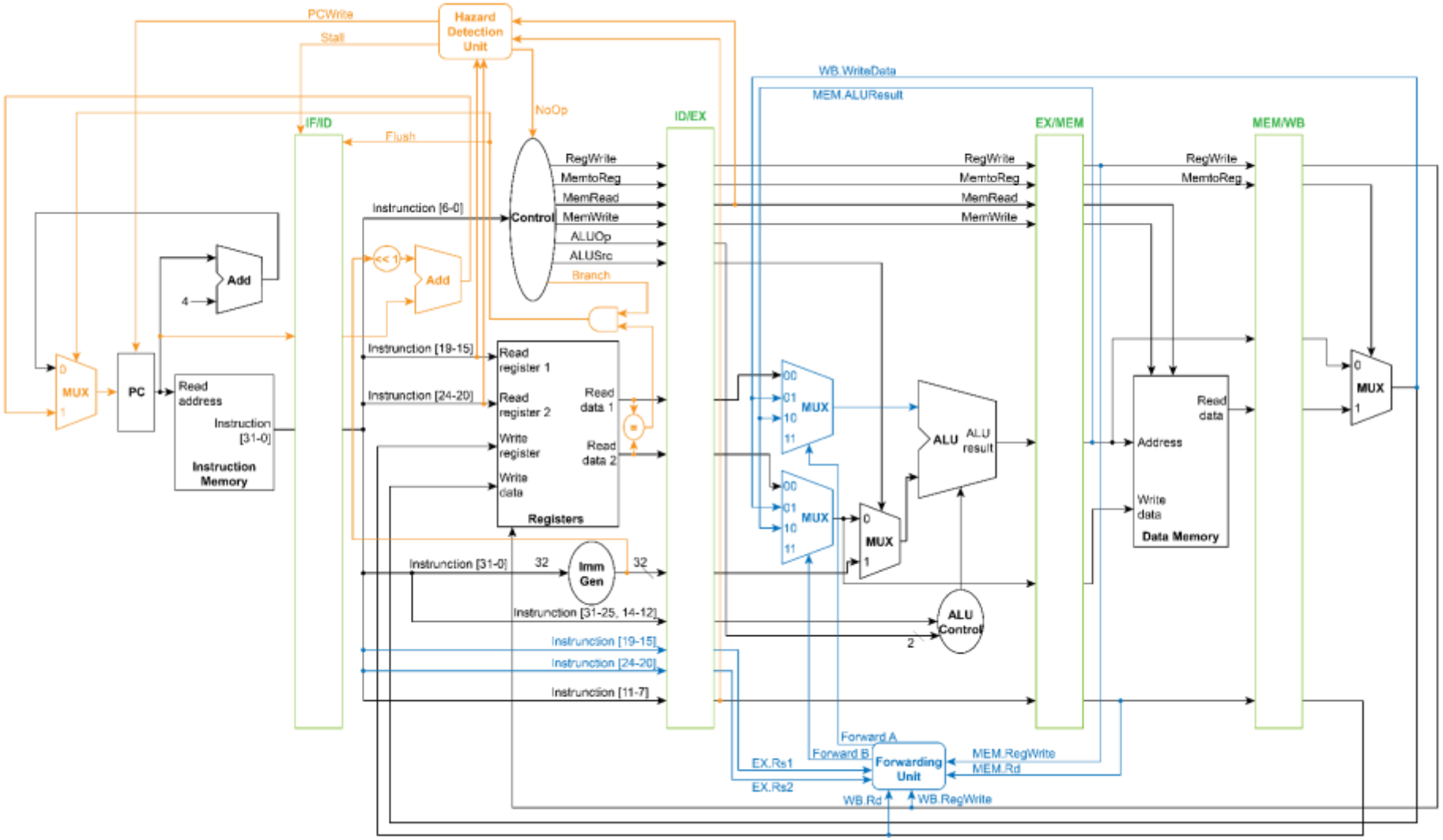
Stall & Flush

- Counted in `testbench.v`
- Can be changed depend on your own design

Example:

```
// put in your own signal to count stall and flush

  if (CPU.HazardDetection.Stall_o == 1 && CPU.Control.Branch_o == 0)
        stall = stall + 1;

  if (CPU.Flush == 1)
        flush = flush + 1;
```

Handling Branch and Hazard



testbench.v

- Initialize registers in all modules
- Load `instruction.txt` into instruction memory
- Create clock signal
- Dump Register files & Data memories in each cycle
- Count number of `stall` and `flush`
- Print result to `output.txt`

testbench.v (cont.)

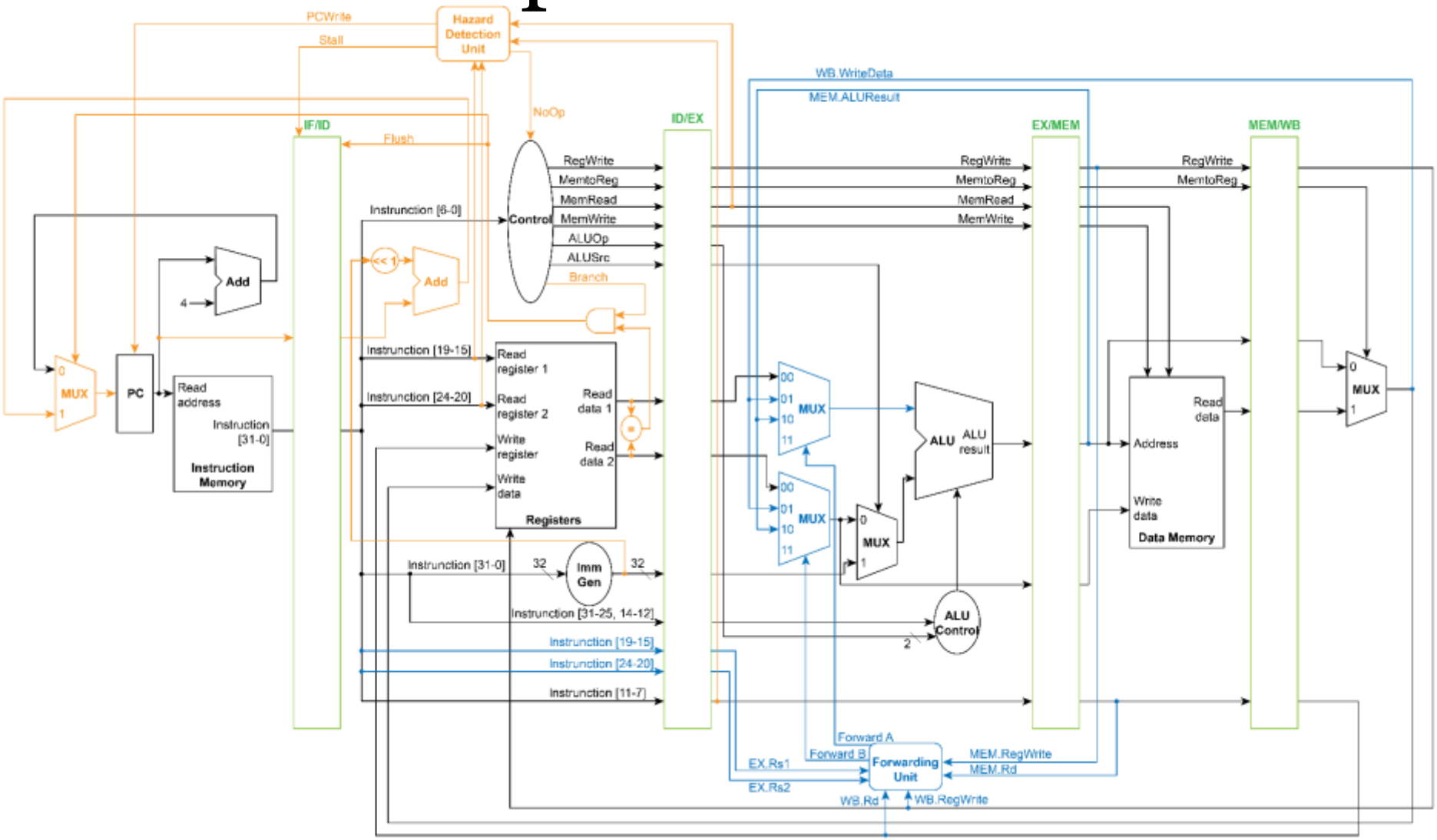
```
    // initialize instruction memory
    for(i=0; i<256; i=i+1) begin
        CPU.Instruction_Memory.memory[i] = 32'b0;
    end

    // initialize data memory
    for(i=0; i<32; i=i+1) begin
        CPU.Data_Memory.memory[i] = 8'b0;
    end
    CPU.Data_Memory.memory[0] = 8'h5;        // n = 5 for example
    // [D-MemoryInitialization] DO NOT REMOVE THIS FLAG !!!

    // initialize Register File
    for(i=0; i<32; i=i+1) begin
        CPU.Registers.register[i] = 32'b0;
    end
    // [RegisterInitialization] DO NOT REMOVE THIS FLAG !!!

    // TODO: initialize your pipeline registers


    // Load instructions into instruction memory
    // Make sure you change back to "instruction.txt" before submission
    $readmemb("instruction.txt", CPU.Instruction_Memory.memory);

    // Open output file
    // Make sure you change back to "output.txt" before submission
    outfile = $fopen("output.txt") | 1;

```

Do not change them !

Execution results

```
cycle =          1, Start = 1, Stall = 0, Flush = 0
PC =          4
Registers
x0 =          0, x8  =          0, x16 =          0, x24 =          0
x1 =          0, x9  =          0, x17 =          0, x25 =          0
x2 =          0, x10 =          0, x18 =          0, x26 =          0
x3 =          0, x11 =          0, x19 =          0, x27 =          0
x4 =          0, x12 =          0, x20 =          0, x28 =          0
x5 =          0, x13 =          0, x21 =          0, x29 =          0
x6 =          0, x14 =          0, x22 =          0, x30 =          0
x7 =          0, x15 =          0, x23 =          0, x31 =          0
Data Memory: 0x00 =          5
Data Memory: 0x04 =          0
Data Memory: 0x08 =          0
Data Memory: 0x0C =          0
Data Memory: 0x10 =          0
Data Memory: 0x14 =          0
Data Memory: 0x18 =          0
Data Memory: 0x1C =          0
```

Final Datapath



Grading Policy

- (80%) Programming
  - Basic pipeline implementation without hazard and forwarding (30%)
  - Data forwarding (20%)
  - Data hazard (stall) (20%)
  - Control hazard (flush) (10%)
  - You will get 0 point if your code cannot be compiled
  - Grading at demo. You have to answer several questions about how you implement at demo. You may get 0 point on this part if you cannot clearly answer the questions (regarded as plagiarism)
- (20%) Report
  - Implementation of each modules
  - Difficulties encountered and solutions in this projects
  - Development environment
- Late policy: 10 points per day

Deadline

- 05/25 (Tue.) 23:59
- Late policy: 10 points per day

Submission Rules

- studentID_project1 (dir)
  - studentID_project1/codes/*.v
  - studentID_project1/studentID_project1_report.pdf
- studentID should be ASCII-printable characters
- MUST REMOVE
  - Data_Memory.v
  - Instruction_Memory.v
  - Registers.v
  - PC.v
  - testdata/*