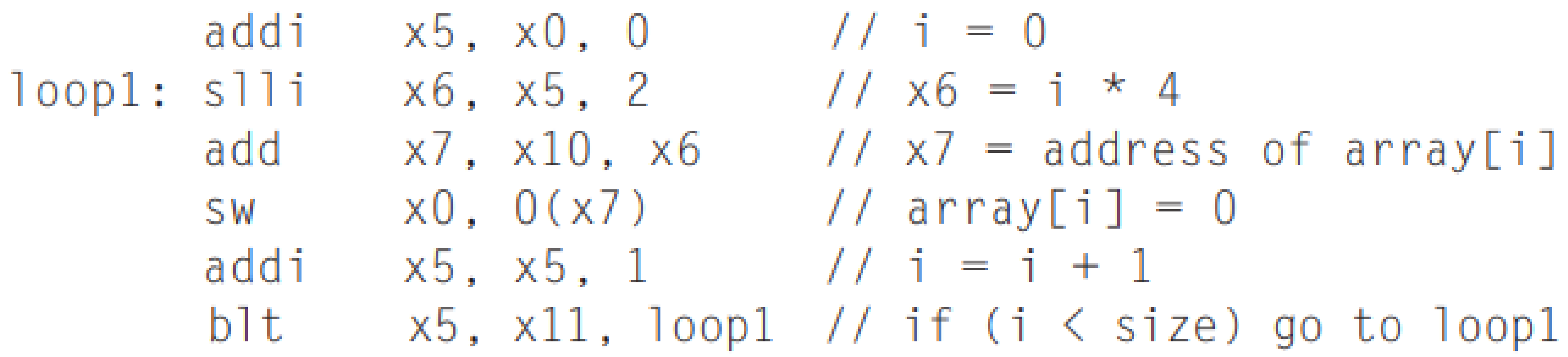
Data Flow Model- RISC-V

Arrays versus Pointers

- Consider two procedures to clear a sequence of words in memory: one using array indices and one with pointers

```
clear1(int array[], size_t int size){

    size_t i;
    for (i = 0; i < size; i += 1)
          array[i] = 0;
}
```

- We assume that the two parameters array and size are found in the registers x10 and x11, and that i is allocated to register x5



```
        addi   x5, x0, 0       // i = 0
loop1: slli   x6, x5, 2       // x6 = i * 4
        add    x7, x10, x6     // x7 = address of array[i]
        sw     x0, 0(x7)       // array[i] = 0
        addi   x5, x5, 1       // i = i + 1
        blt    x5, x11, loop1  // if (i < size) go to loop1
```

```
clear2(int *array, size_t int size){

    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
         *p = 0;
}
```

- The second procedure that uses pointers allocates the two parameters array and size to the registers x10 and x11 and allocates p to register x5

```
        addi x5, x10, 0      // p = address of array[0]
loop2:  sw   x0, 0(x5)       // Memory[p] = 0
        addi x5, x5, 4       // p = p + 4
        slli x6, x11, 2      // x6 = size * 4
        add  x7, x10, x6     // x7 = address of array[size]
        bltu x5, x7, loop2   // if (p<&array[size]) go to loop2
```

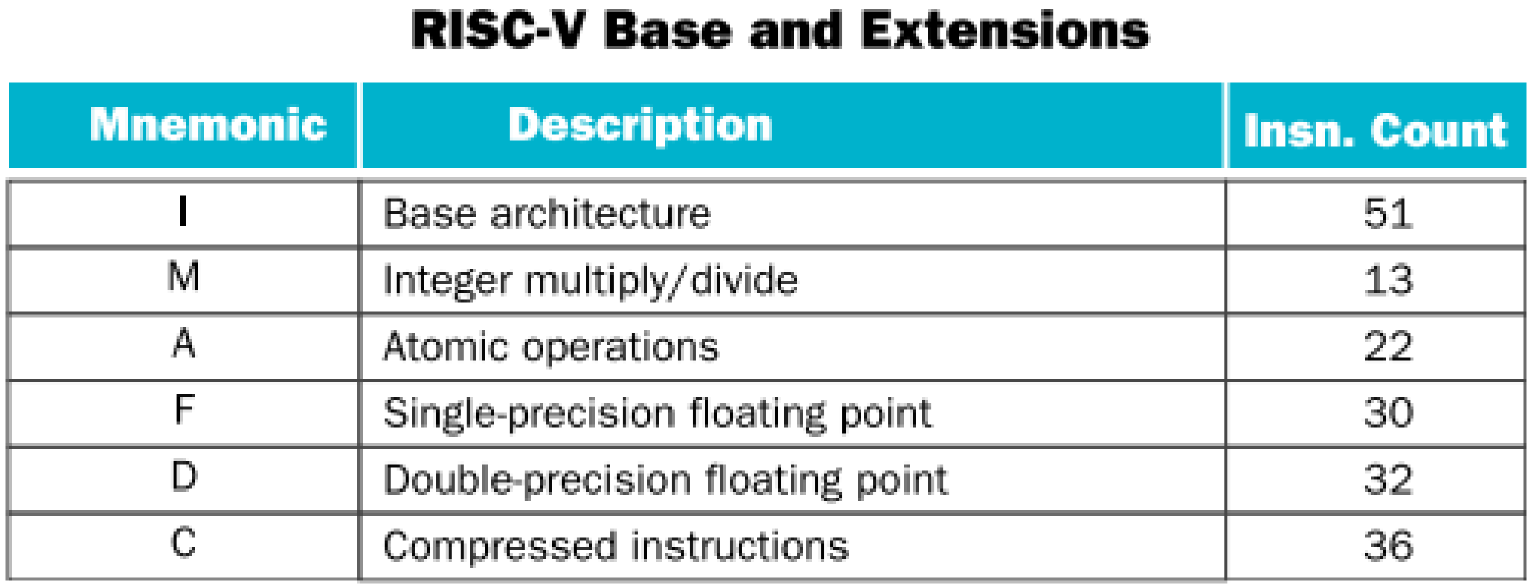
- A faster version of the code moves this calculation outside the loop

```
        addi  x5, x10, 0      // p = address of array[0]
        slli  x6, x11, 2      // x6 = size * 4
        add   x7, x10, x6     // x7 = address of array[size]
loop2:  sw    x0, 0(x5)       // Memory[p] = 0
        addi  x5, x5, 4       // p = p + 4
        bltu  x5, x7, loop2   // if (p < &array[size]) go to loop2
```

- The index version must have the “multiply” and add inside the loop because i is incremented and each address must be recalculated from the new index.
- The memory pointer version increments the pointer p directly. The pointer version moves the scaling shift and the array bound addition outside the loop, thereby reducing the instructions executed per iteration from five to three.
- This manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops)

The Rest of RISC-V instructions

- RISC-V architects partitioned the instruction set into a base architecture and several extensions

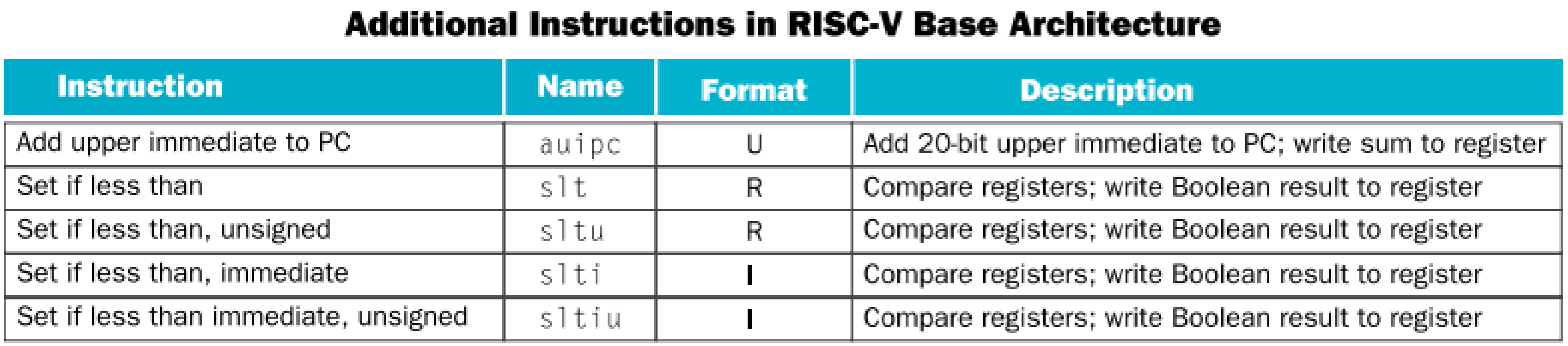


RISC-V Base and Extensions

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

Extensions

- five standard extensions of base architecture



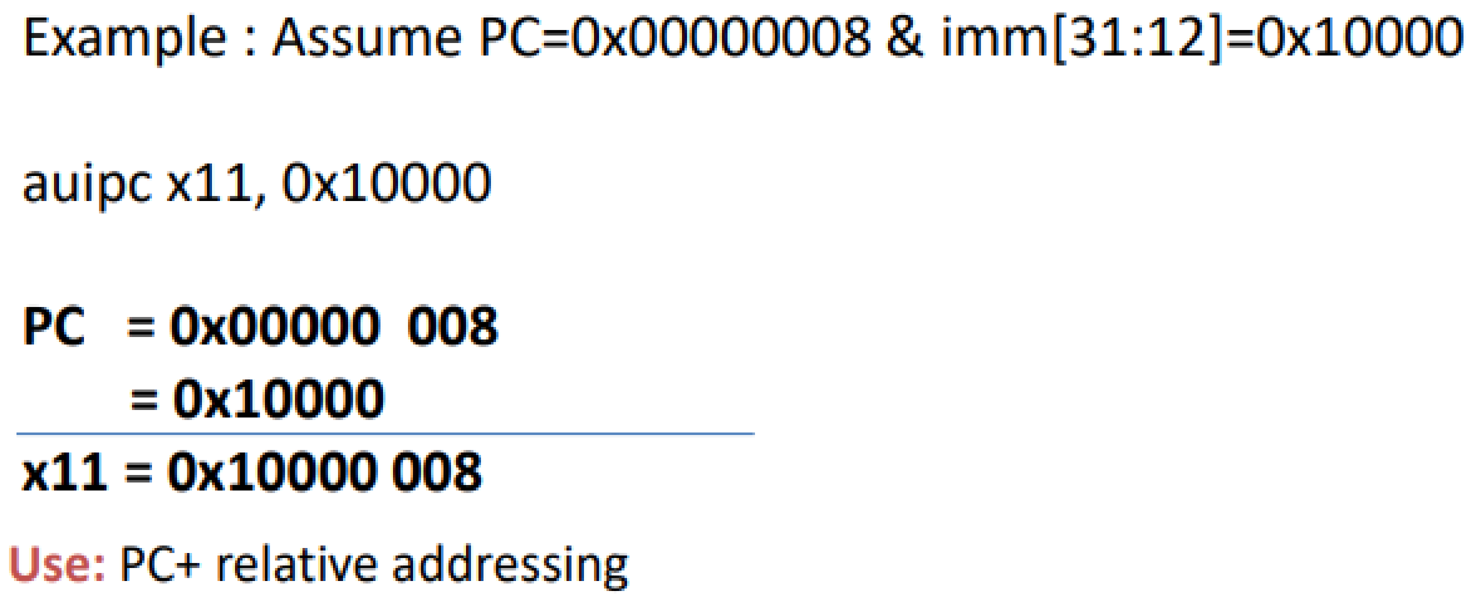
Additional Instructions in RISC-V Base Architecture

Instruction	Name	Format	Description
Add upper immediate to PC	`auipc`	U	Add 20-bit upper immediate to PC; write sum to register
Set if less than	`slt`	R	Compare registers; write Boolean result to register
Set if less than, unsigned	`sltu`	R	Compare registers; write Boolean result to register
Set if less than, immediate	`slti`	I	Compare registers; write Boolean result to register
Set if less than immediate, unsigned	`sltiu`	I	Compare registers; write Boolean result to register

auipc

- auipc, is used for PC-relative memory addressing
- it holds a 20-bit constant that corresponds to bits 12 through 31 of an integer
- auipc’s effect is to add this number to the PC and write the sum to a register
- Combined with an instruction like addi, it is possible to address any byte of memory within 4 GiB of the PC
- This feature is useful for position-independent code, which can execute correctly no matter where in memory it is loaded.

• Syntax: auipc rd, imm[31:12]



Example : Assume PC=0x00000008 & imm[31:12]=0x10000

auipc x11, 0x10000

```
PC   = 0x00000  008
     = 0x10000
-------------------------
x11 = 0x10000 008
```

Use: PC+ relative addressing

- Use 1: auipc used for PC+relative addressing to calculate base address of symbol defined in data segment
- Assume base address of num3 in data memory to be accessed is 0x1000 0020 and PCpresent = 0x0000 0008. Write program to initialize base address in x11 using PC+relative addressing using auipc and addi

Step1: Calculate Relative address

Relative address = Base address – PCpresent

Relative address = 0x1000 0020 – 0x0000 0008 =0x1000 0018

Step2: Divide Relative address in to two parts as Upper 20 bits and Lower 12 bits

0x1000 0   018

Step3: Use following set of Instructions

```
auipc x10, 0x10000
addi x10,0x018
```

num1	**0x1000 0000**	num1[0]
	0x1000 0004	num1[1]
	0x1000 0008	num1[2]
	0x1000 000C	num1[3]
	0x1000 0010	num1[4]
num2	**0x1000 0014**	num2[0]
	0x1000 0018	num2[1]
	0x1000 001C	num2[2]
num3	**0x1000 0020**	num3[0]
	0x1000 0020	Num3[1]

x10 =PCpresent[31:12] + 0x10000 | PCpresent[11:0] =0x00000 +0x10000 | 0x008 = 0x10000 008

x10= x10+imm[11:0] = 0x10000008 +0x018 = 0x10000020

- Use 2: auipc used for to calculate long target address of symbol beyond the range of jal
- Assume address of symbol is 0x07FFF F018 and PCpresent = 0x0000 0008. Write program to change the control of execution to address of symbol.

Step1: Calculate Relative address

Relative address = Base address – PCpresent

Relative address = 0x07FFF F018 – 0x0000 0008 =0x7FFF F010

Step2: Divide Relative address in to two parts as Upper 20 bits and Lower 12 bits

0x7FFF F | 0x010

Step3: Use following set of Instructions

```
auipc x10, 0x7FFFF
jalr x1,x10,0x010
```

PCnew=PCpresent[31:12] + 0x7FFFF | PCpresent[11:0] = 0x00000 +0x7FFFF | 0x008 = 0x7FFFF 008
PCnew= x10+imm[11:0] = 0x7FFFF 008 + 0x010 = 0x07FFF F018

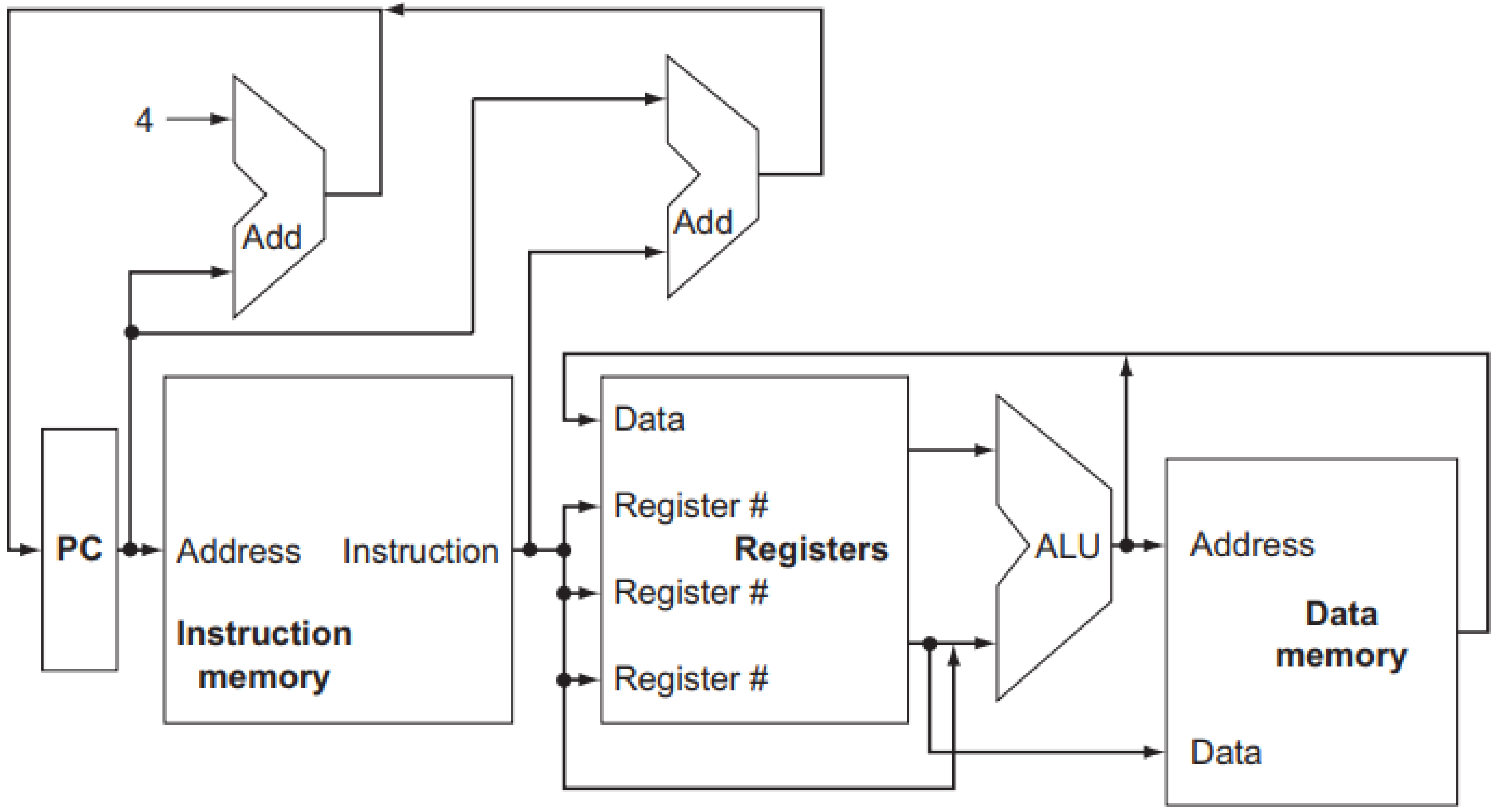
Extension A

- extension, A, supports atomic memory operations for multiprocessor synchronization
- The load-reserved (lr.w) and store-conditional (sc.w) instructions are members of the A extension
- The store-conditional is defined to both store the value of a (presumably different) register in memory and to change the value of another register to a 0 if it succeeds and to a nonzero value if it fails
- Thus, sc.w specifies three registers: one to hold the address, one to indicate whether the atomic operation failed or succeeded, and one to hold the value to be stored in memory if it succeeded

```
again:
lr.w x10, (x20)        // load-reserved
sc.w x11, x23, (x20)  // store-conditional
bne x11, x0, again    // branch if store fails (0)
addi x23, x10, 0      // put loaded value in x23
```

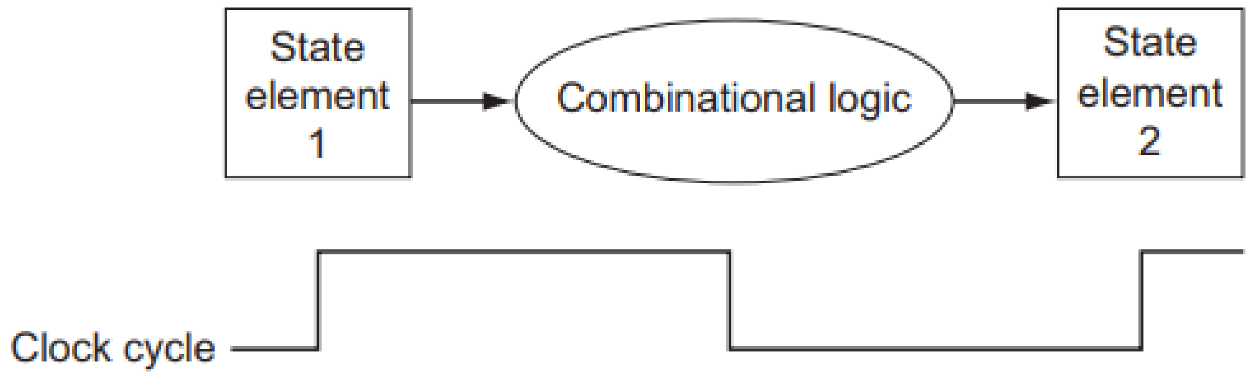
Logic Design Conventions

- The datapath elements in the RISC-V implementation consist of two different types of logic elements: elements that operate on data values and elements that contain state.
- The elements that operate on data values are all combinational, which means that their outputs depend only on the current inputs.
- state element: A memory element, such as a register or a memory
- A state element has at least two inputs(data and clk) and one output

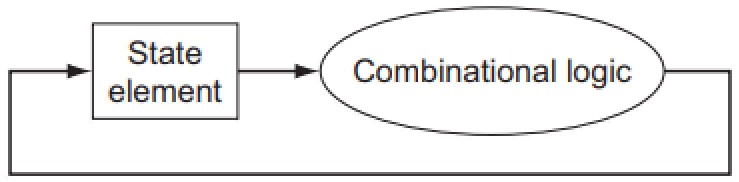

4
Add
Add
Data
Register #
Registers
Register #
Register #
ALU
Address
Data memory
Data
PC
Address
Instruction
Instruction memory

Clocking Methodology

- clocking methodology The approach used to determine when data are valid and stable relative to the clock.
- Ex: edge-triggered clocking

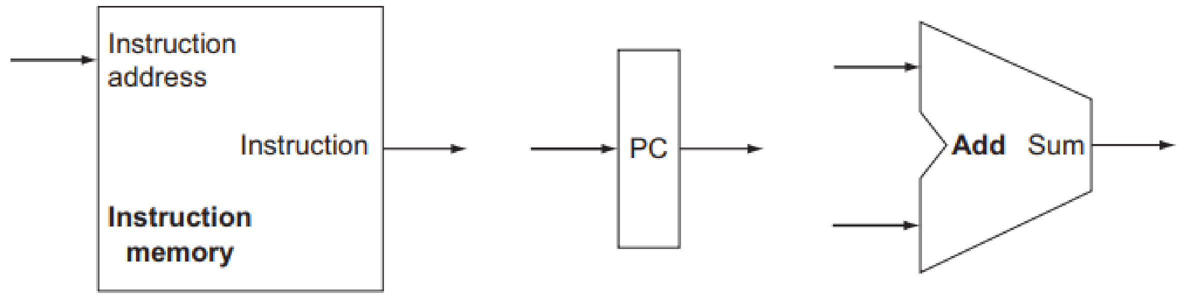


- control signal A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a data signal, which contains information that is operated on by a functional unit.
- asserted The signal is logically high or true
- deasserted The signal is logically low or false
- Edge triggered circuit that reads and writes in the same clk cycle



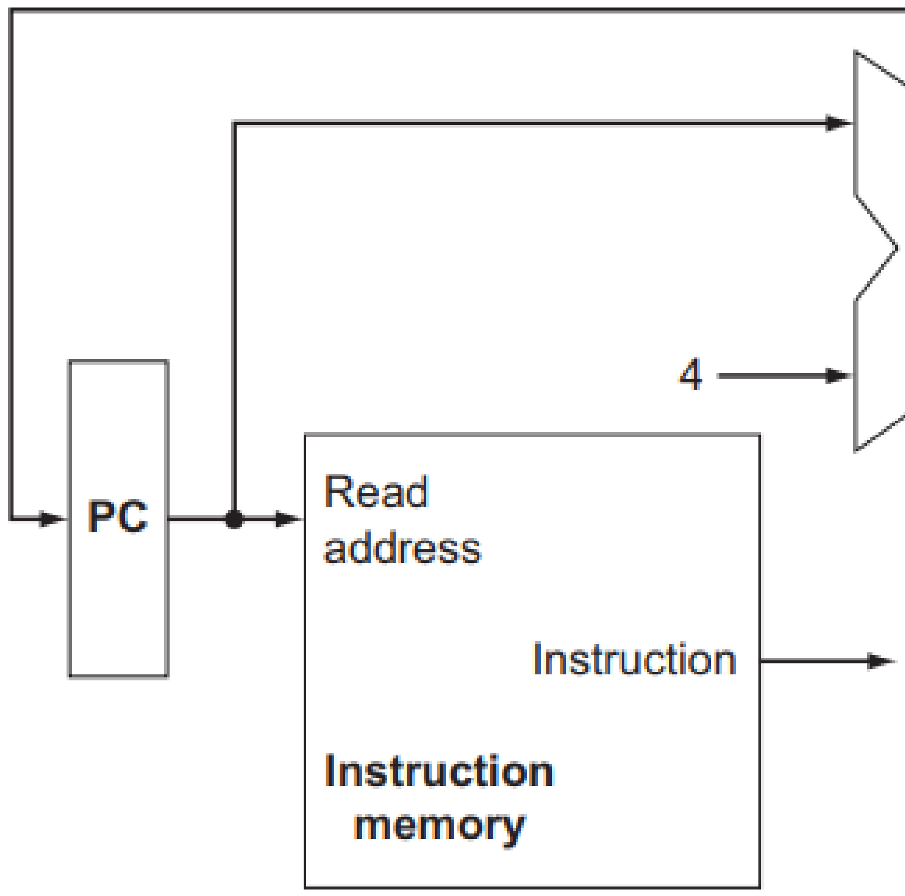
Building a Datapath

- datapath design is to examine the major components required to execute each class of RISC-V instructions
- Fig a is the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure shows the program counter (PC),
- Lastly, we will need an adder to increment the PC to the address of the next instruction



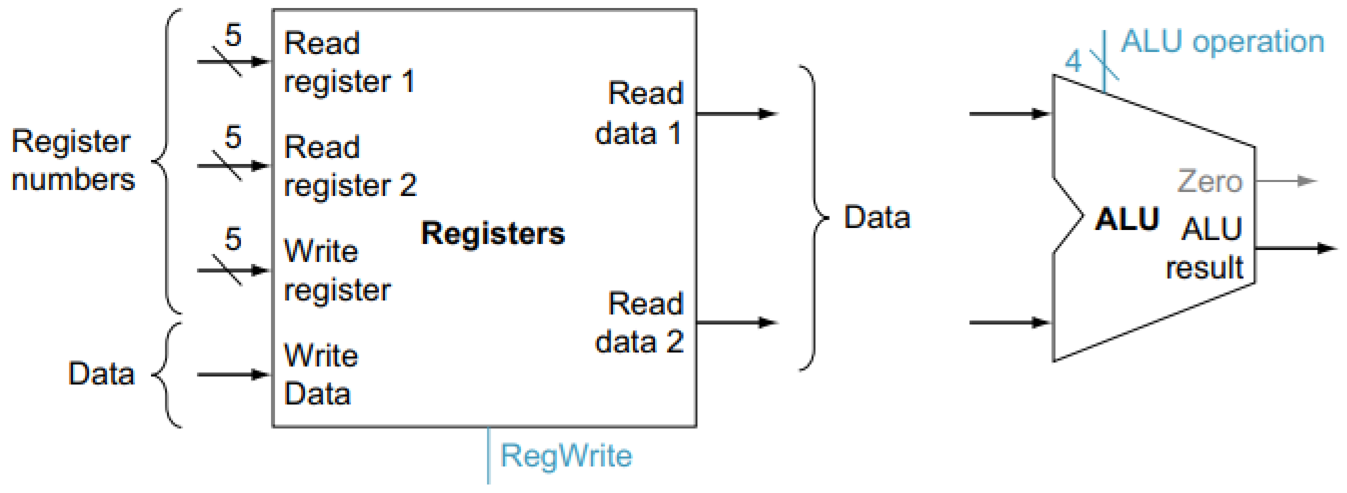
a. Instruction memory b. Program counter c. Adder

A portion of the datapath used for fetching instructions and incrementing the program counter



R-format instructions

- register file A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed
- R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction
- writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge



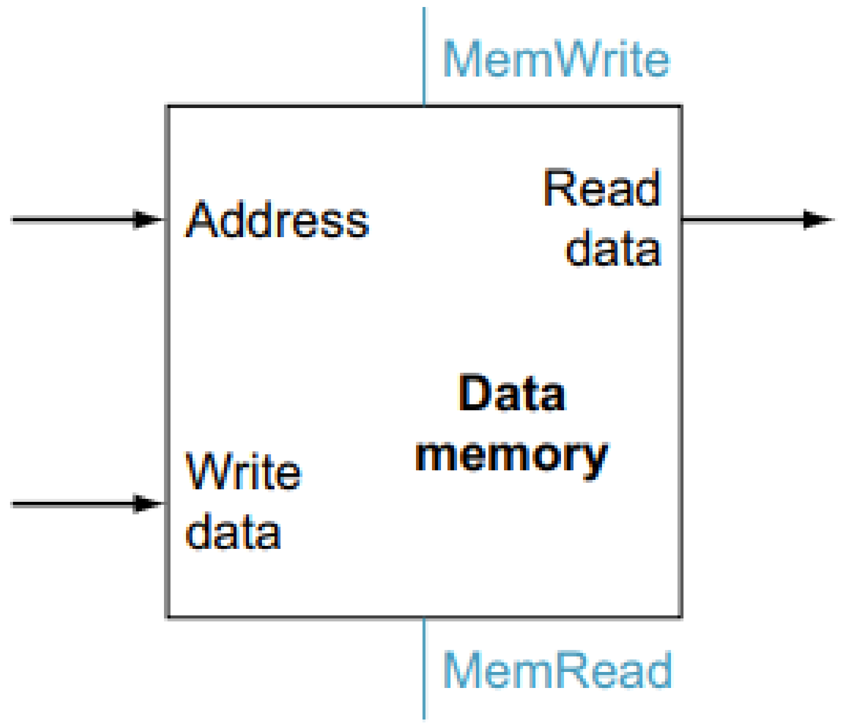
a. Registers

b. ALU

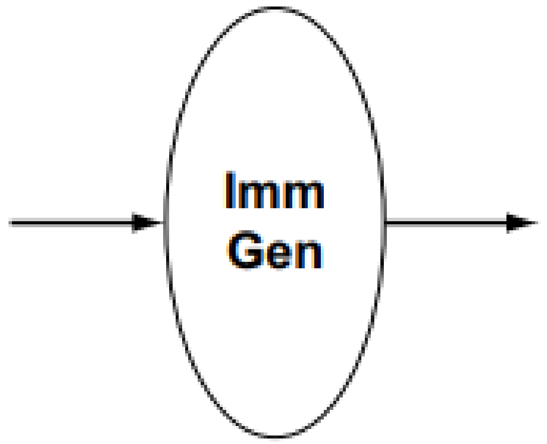
RISC-V load register and store register instructions

- the RISC-V load register and store register instructions, which have the general form lw x1, offset(x2) or sw x1, offset(x2)
- If the instruction is a store, the value to be stored must also be read from the register file where it resides in x1.
- If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is x1. Thus, we will need both the register file and the ALU

The two units needed to implement loads and stores, in addition to the register file and ALU



a. Data memory unit



b. Immediate generation unit

branch target address

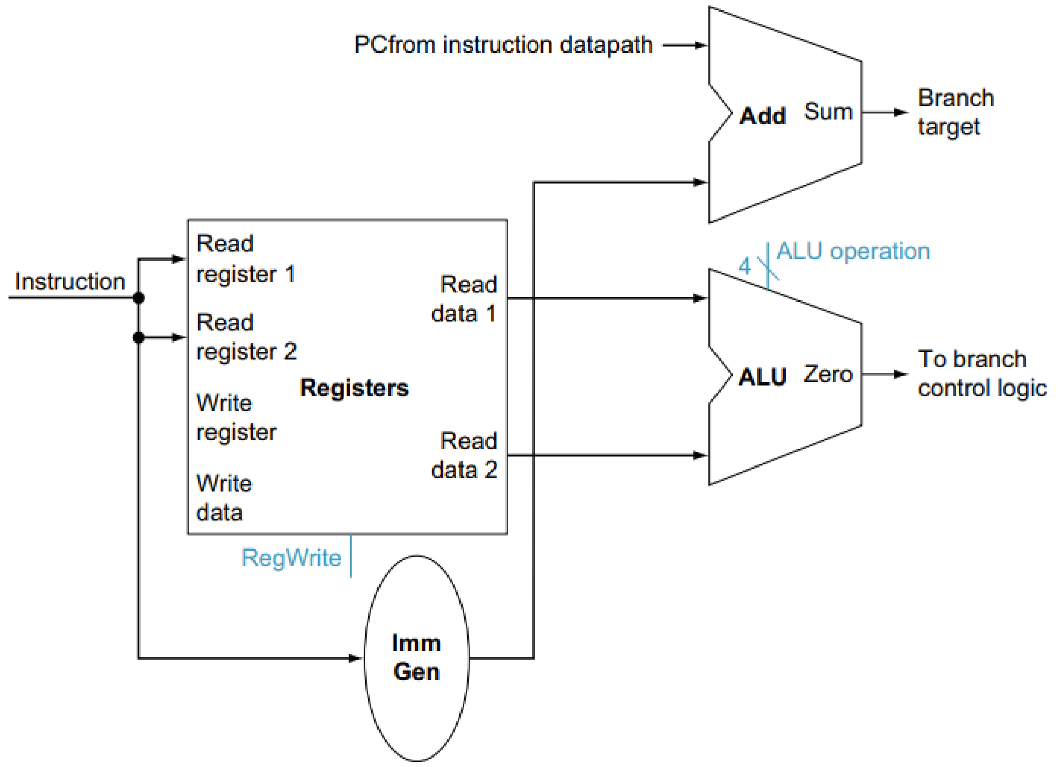
- branch target address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the RISC-V architecture, the branch target is given by the sum of the offset field of the instruction and the address of the branch.
- we will need a unit to sign-extend the 12-bit offset field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to
- The data memory must be written on store instructions; hence, data memory has read and write control signals, an address input, and an input for the data to be written into memory

- The beq instruction has three operands, two registers that are compared for equality, and a 12-bit offset used to compute the branch target address relative to the branch instruction address
- Its form is beq x1, x2, offset

-The instruction set architecture specifies that the base for the branch address calculation is the address of the branch instruction.
-The architecture also states that the offset field is shifted left 1 bit so that it is a half word offset; this shift increases the effective range of the offset field by a factor of 2

- the branch datapath must do two operations: compute the branch target address and test the register contents. (Branches also affect the instruction fetch portion of the datapath.)

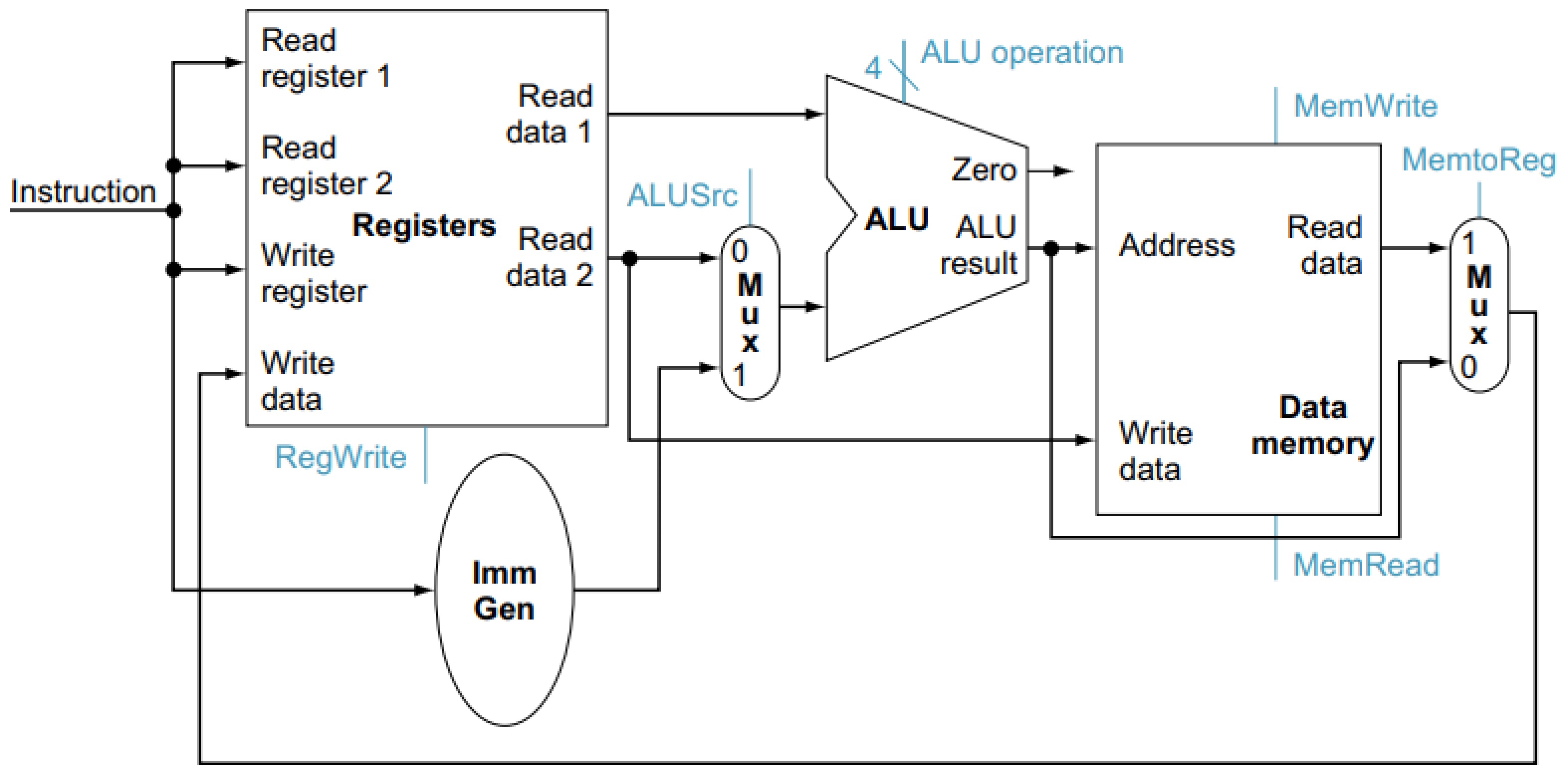
The portion of a datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and immediate (the branch displacement)



Creating a Single Datapath

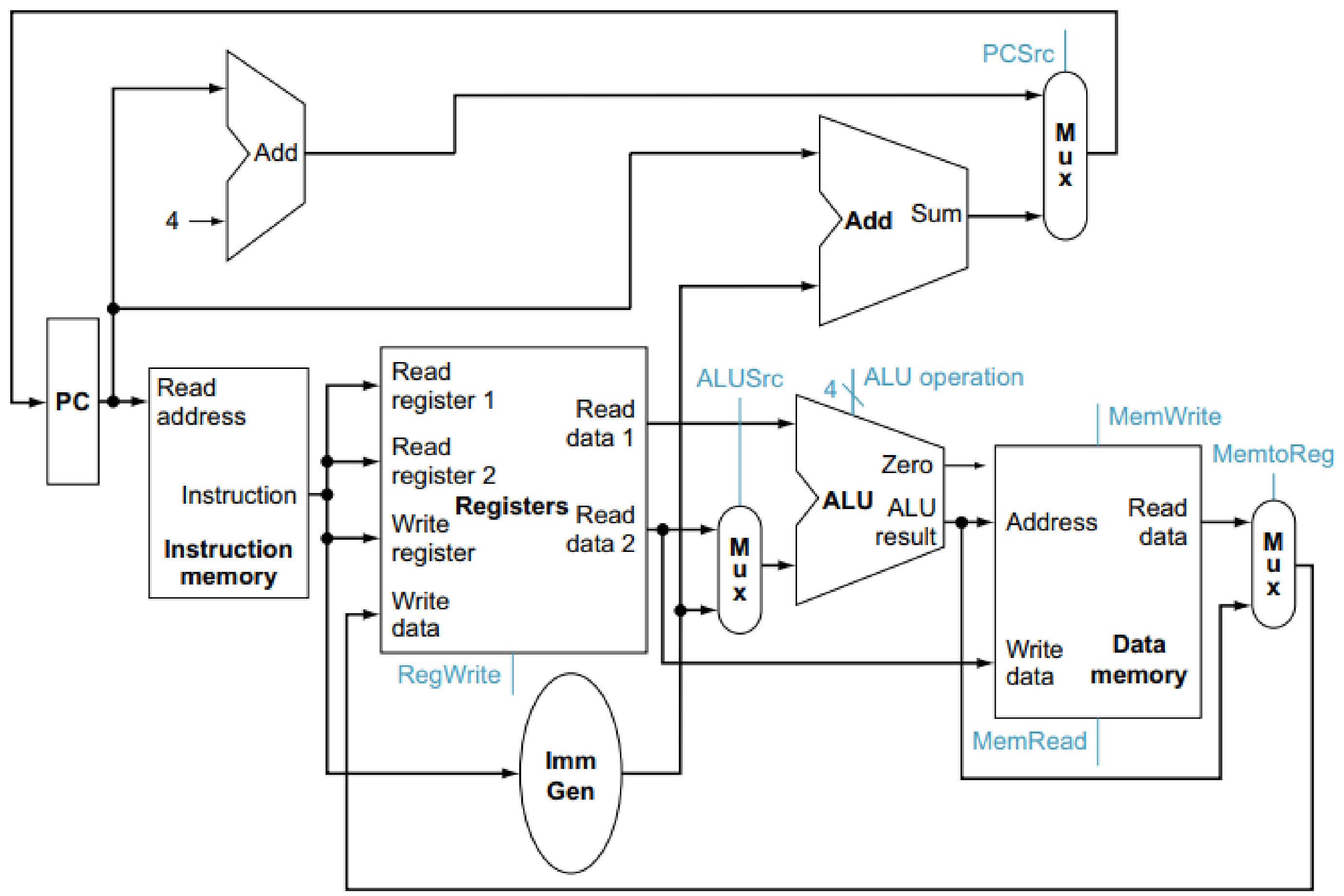
- The simplest datapath will attempt to execute all instructions in one clock cycle
- Thus, that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated

The datapath for the memory instructions and the R-type instructions



- The operations of arithmetic-logical (or R-type) instructions and the memory instructions datapath are quite similar. The key differences are the following:
- The arithmetic-logical instructions use the ALU, with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, although the second input is the signextended 12-bit offset field from the instruction.
- The value stored into a destination register comes from the ALU (for an R-type instruction) or the memory (for a load).

The simple datapath for the core RISC-V architecture combines the elements required by different instruction classes

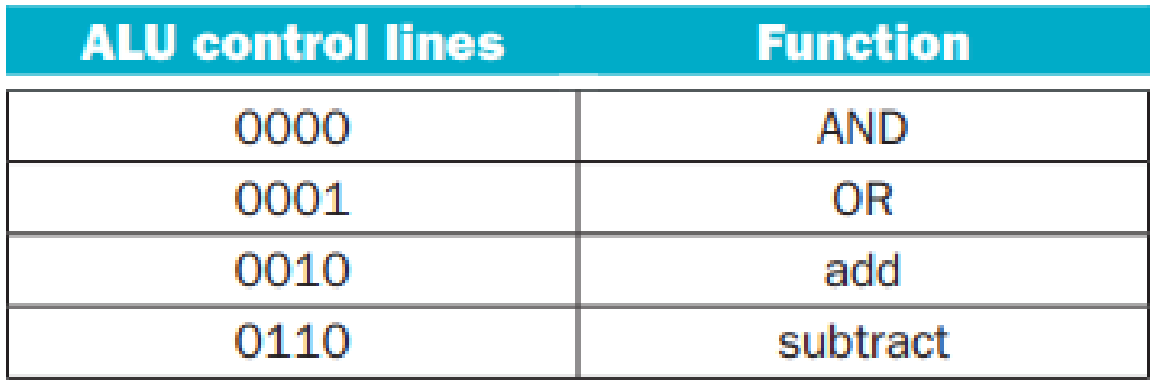


A Simple Implementation Scheme

- The simple implementation covers load word (lw), store word (sw), branch if equal (beq), and the arithmetic logical instructions add, sub, and, and or

The ALU Control

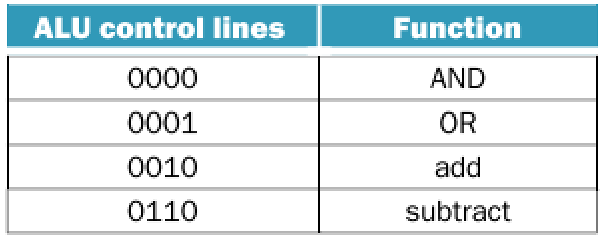
- The RISC-V ALU defines the four following combinations of four control inputs:



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

- Depending on the instruction class, the ALU will need to perform one of these four functions.
- For load and store instructions, we use the ALU to compute the memory address by addition
- For the R-type instructions, the ALU needs to perform one of the four actions (AND, OR, add, or subtract), depending on the value of the 7-bit funct7 field (bits 31:25) and 3-bit funct3 field (bits 14:12) in the instruction
- For the conditional branch if equal instruction, the ALU subtracts two operands and tests to see if the result is 0

- We can generate the 4-bit ALU control input using a small control unit that has as inputs the funct7 and funct3 fields of the instruction and a 2-bit control field, which we call ALUOp
- Add(00)for loads and stores
- Subtract and test if zero (01) for beq
- Operation encoded in the func7 and func3 fields(10)

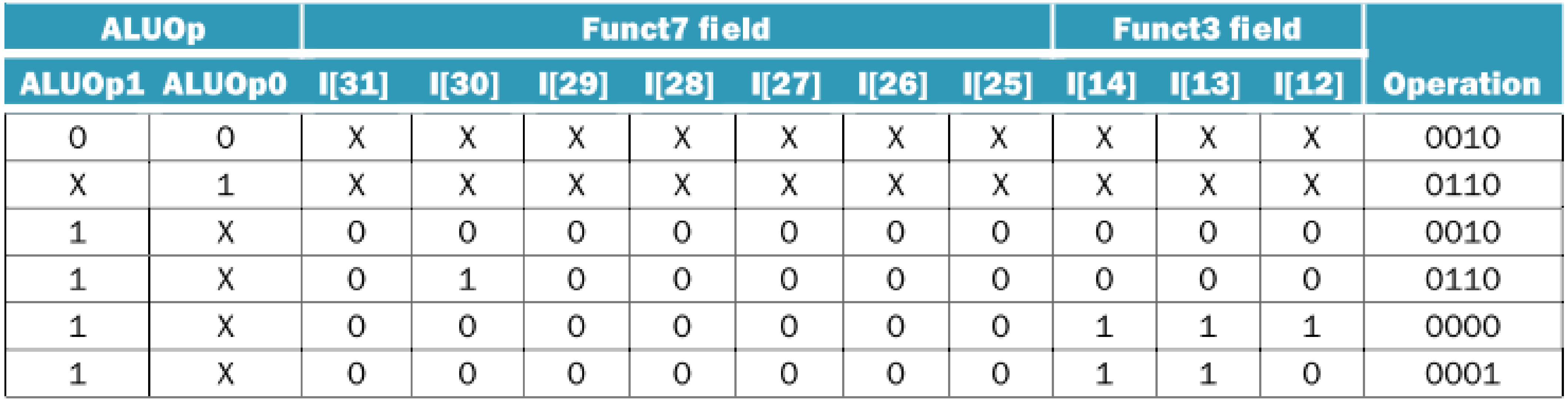


ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	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

- multiple levels of decoding is used—that is, the main control unit generates the ALUOp bits, which then are used as input to the ALU control that generates the actual signals to control the ALU unit—is a common implementation technique
- Multiple level control- reduces size of main control unit
- Reduces latency- which is critical in defining clock cycle time

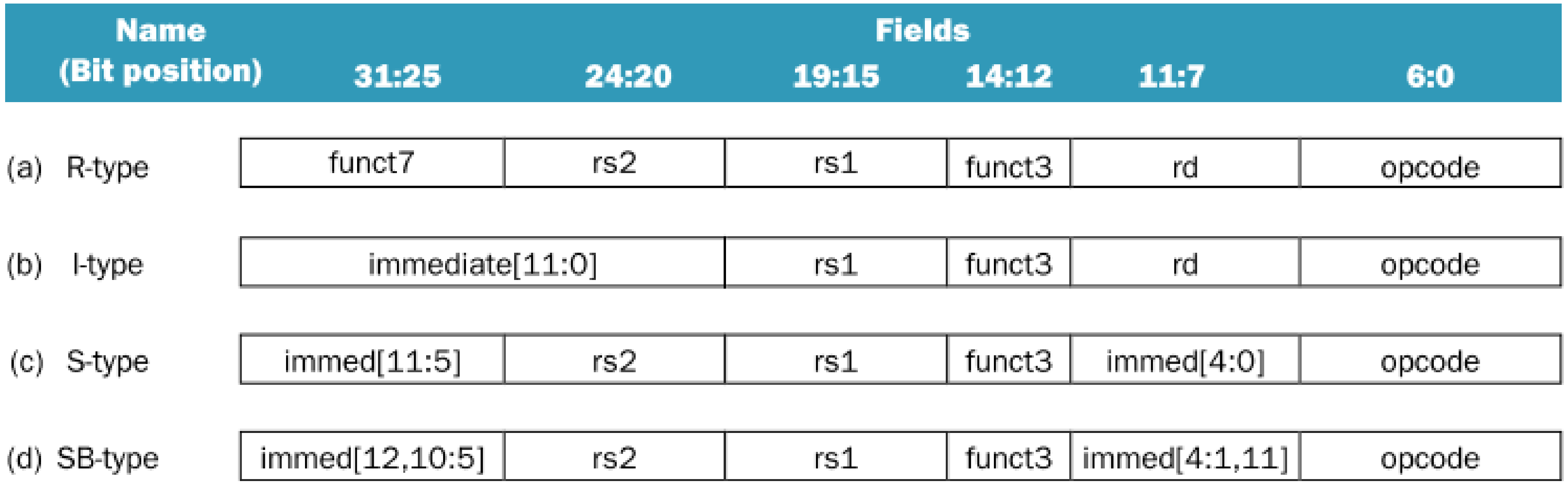
The truth table for the 4 ALU control bits (called Operation)



ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

Designing the Main Control Unit

- The four instruction classes

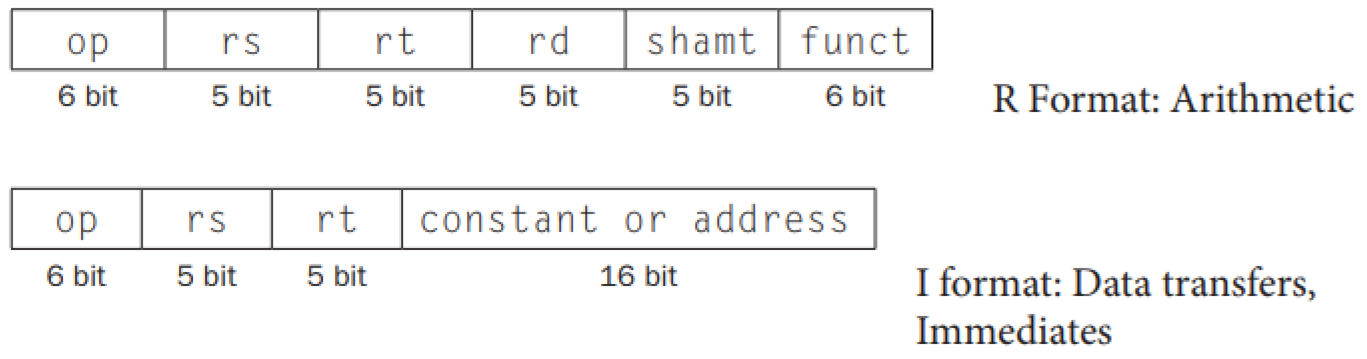


Name	Fields					
(Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

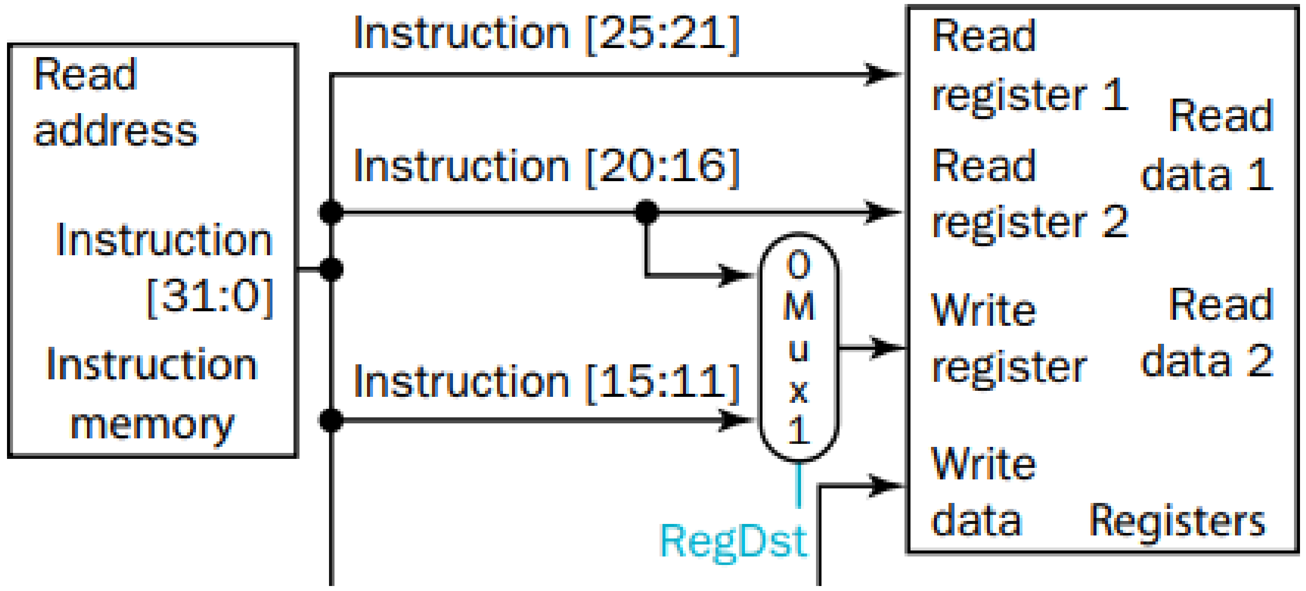
several major observations about the instruction format

- The opcode field, is always in bits 6:0. Depending on the opcode, the funct3 field (bits 14:12) and funct7 field (bits 31:25) serve as an extended opcode field.
- The first register operand is always in bit positions 19:15 (rs1) for R-type instructions and branch instructions. This field also specifies the base register for load and store instructions
- The second register operand is always in bit positions 24:20 (rs2) for R-type instructions and branch instructions. This field also specifies the register operand that gets copied to memory for store instructions

- Another operand can also be a 12-bit offset for branch or load-store instructions
- The destination register is always in bit positions 11:7 (rd) for R-type instructions and load instruction
- MIPS instruction format



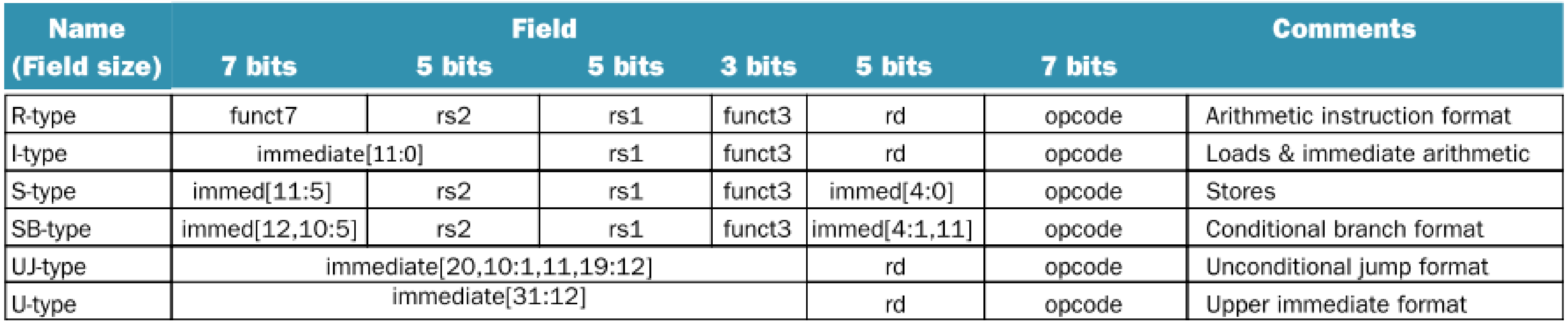
The MIPS, arithmetic instruction format, data transfer instruction format, and their impact on the MIPS datapath



- To keep the destination register always in bits 11 to 7 of all instructions, the RISC-V S format has to split the immediate field into two pieces: bits 31 to 25 have immediate[11:5] and bits 11 to 7 have immediate[4:0]. It looks odd compared with MIPS, which keeps the immediate field contiguous, but the RISC-V assembler hides this complexity, and the hardware benefits

The actual RISC-V formats

- RISC-V has two formats where all the fields are the same size and are immediates as in two other formats—SB versus S and UJ versus U—but the bits are swirled around

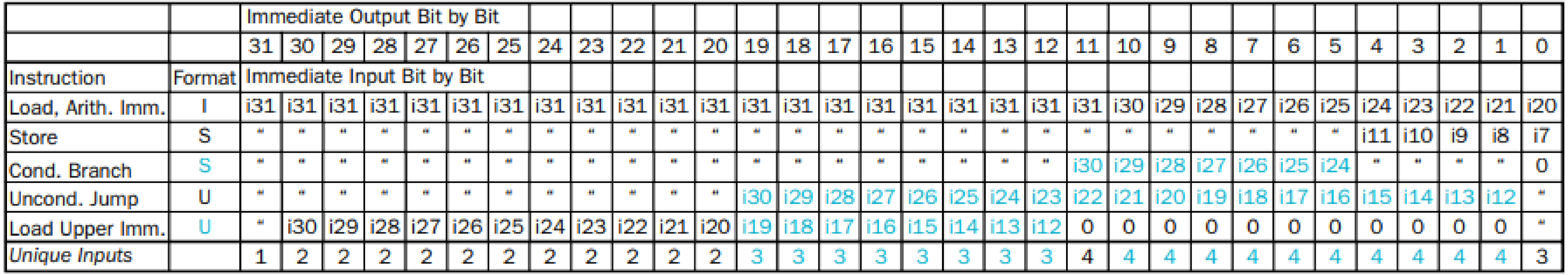


Name	Field						Comments
(Field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

- The SB and UJ formats once again simplify hardware by giving the assembler more work to do

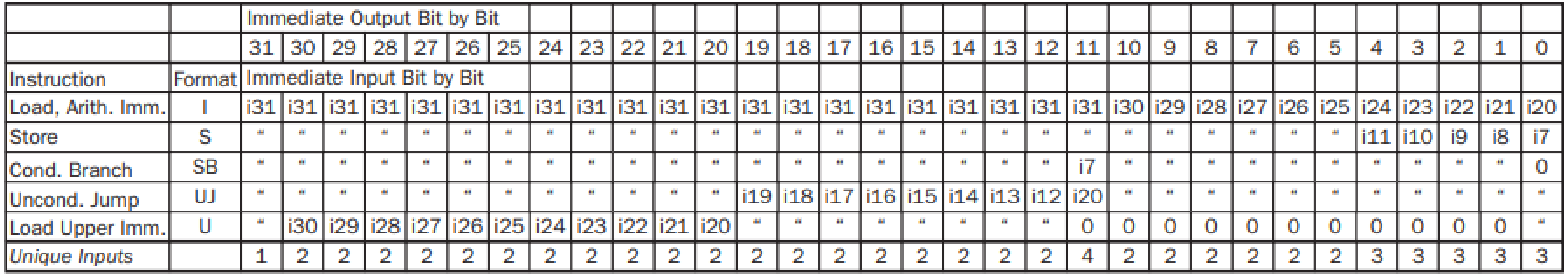
Inputs to immediate if hypotheticaly conditional branches use the S format, and if jumps, use the U format

- The last row shows the number of unique inputs per output bit, which determines the number of ports to a multiplexor in the immediate generator



		Immediate Output Bit by Bit																															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Format	Immediate Input Bit by Bit																															
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	"	"	"	"	0
Uncond. Jump	U	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	"
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	0	0	0	0	0	0	0	0	0	0	0	"
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	3

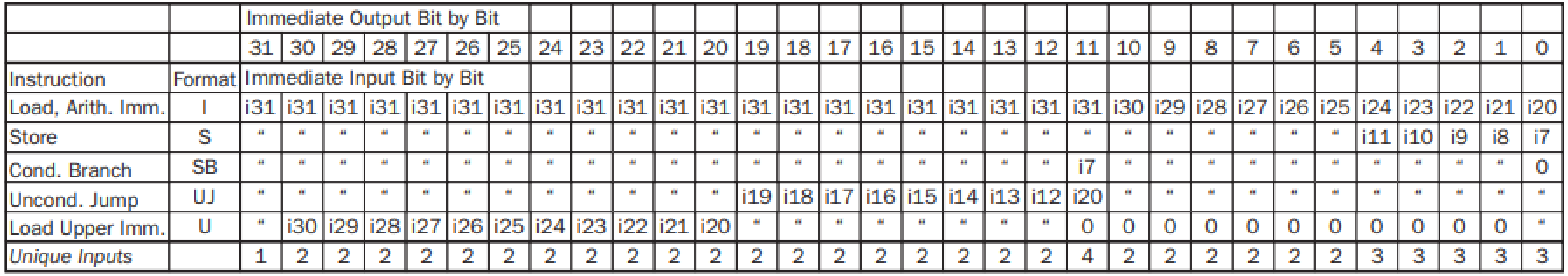
Inputs to immediate given that branches use the SB format and jumps use the UJ format



		Immediate Output Bit by Bit																													
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Format	Immediate Input Bit by Bit																													
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	SB	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i7	"	"	"	"	"	"	"	"	"	"	0
Uncond. Jump	UJ	"	"	"	"	"	"	"	"	"	"	"	"	i19	i18	i17	i16	i15	i14	i13	i12	i20	"	"	"	"	"	"	"	"	"	"	"
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	"	"	"	"	"	"	"	"	0	0	0	0	0	0	0	0	0	0	0	"
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	3	3	3	3	3

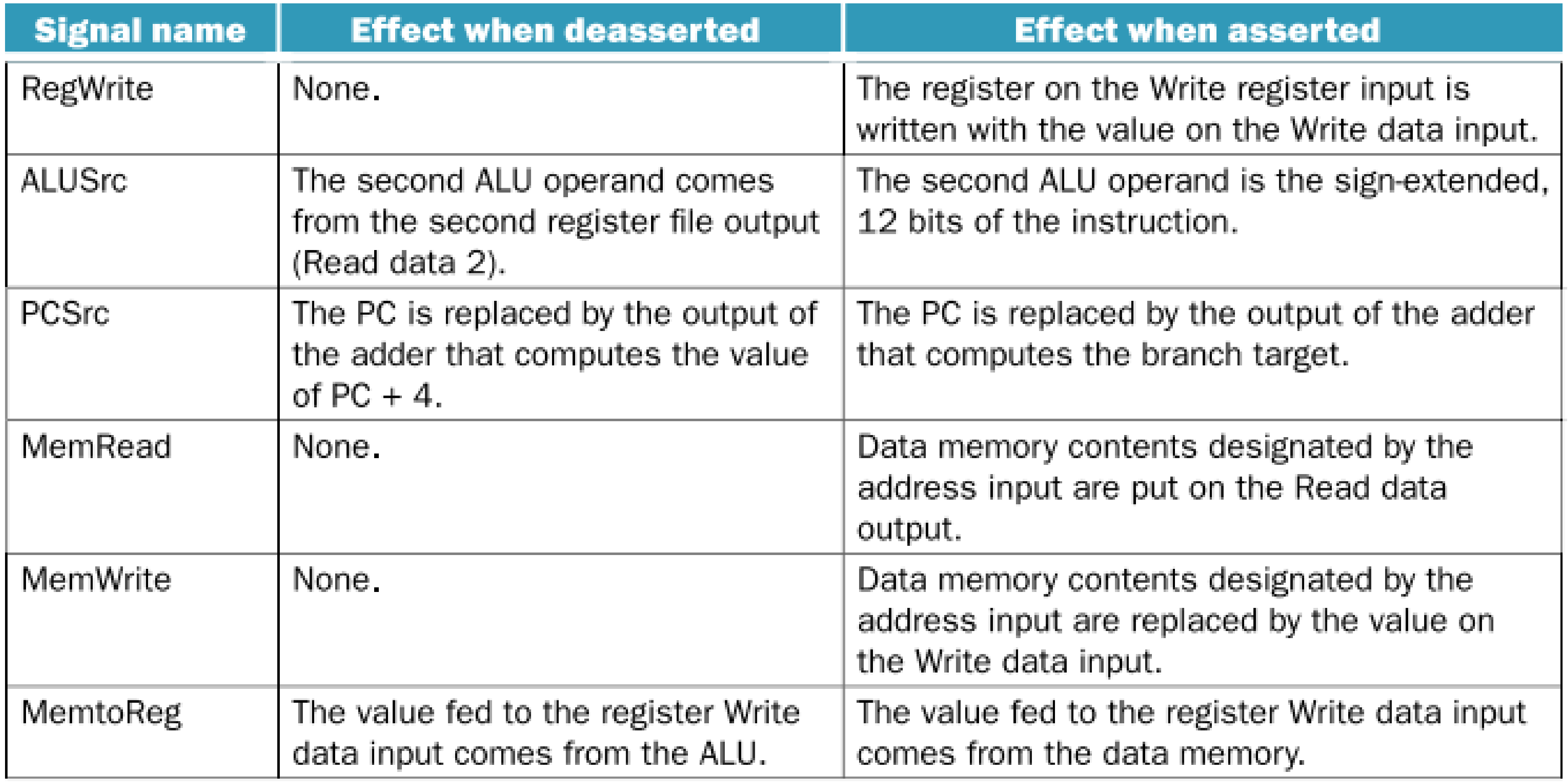
Inputs to immediate given that branches use the SB format and jumps use the UJ format, which is what RISC-V uses

- The SB and UJ formats reduce the multiplexors for immediate bits 19 to 12 from 3:1 to 2:1 multiplexors and immediate bits 10 to 1 from 4:1 to 2:1



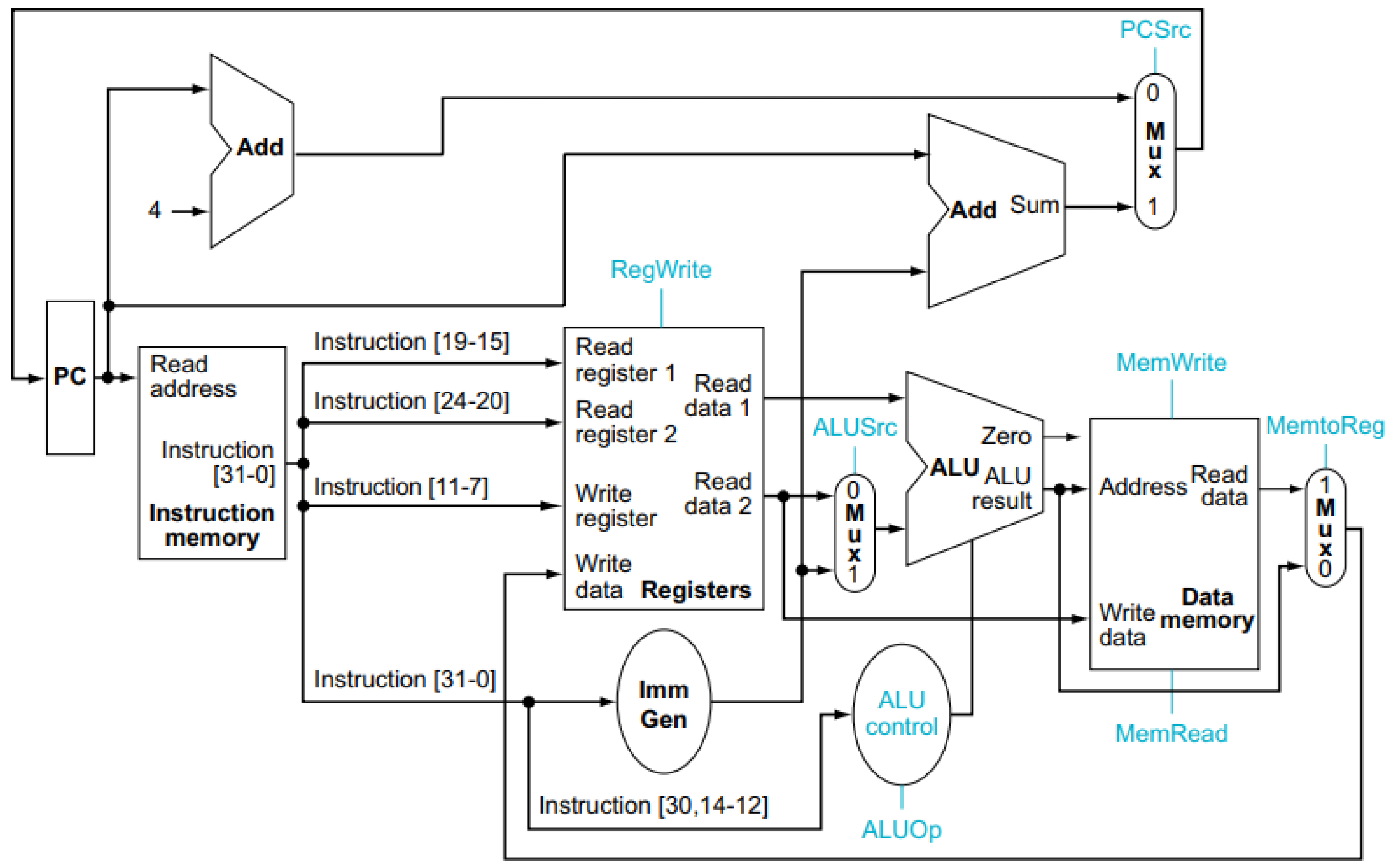
		Immediate Output Bit by Bit																															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Format	Immediate Input Bit by Bit																															
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	SB	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i7	"	"	"	"	"	"	"	"	"	"	0
Uncond. Jump	UJ	"	"	"	"	"	"	"	"	"	"	"	"	i19	i18	i17	i16	i15	i14	i13	i12	i20	"	"	"	"	"	"	"	"	"	"	"
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	"	"	"	"	"	"	"	"	0	0	0	0	0	0	0	0	0	0	0	"
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	3	3	3	3	3

The effect of six control signals

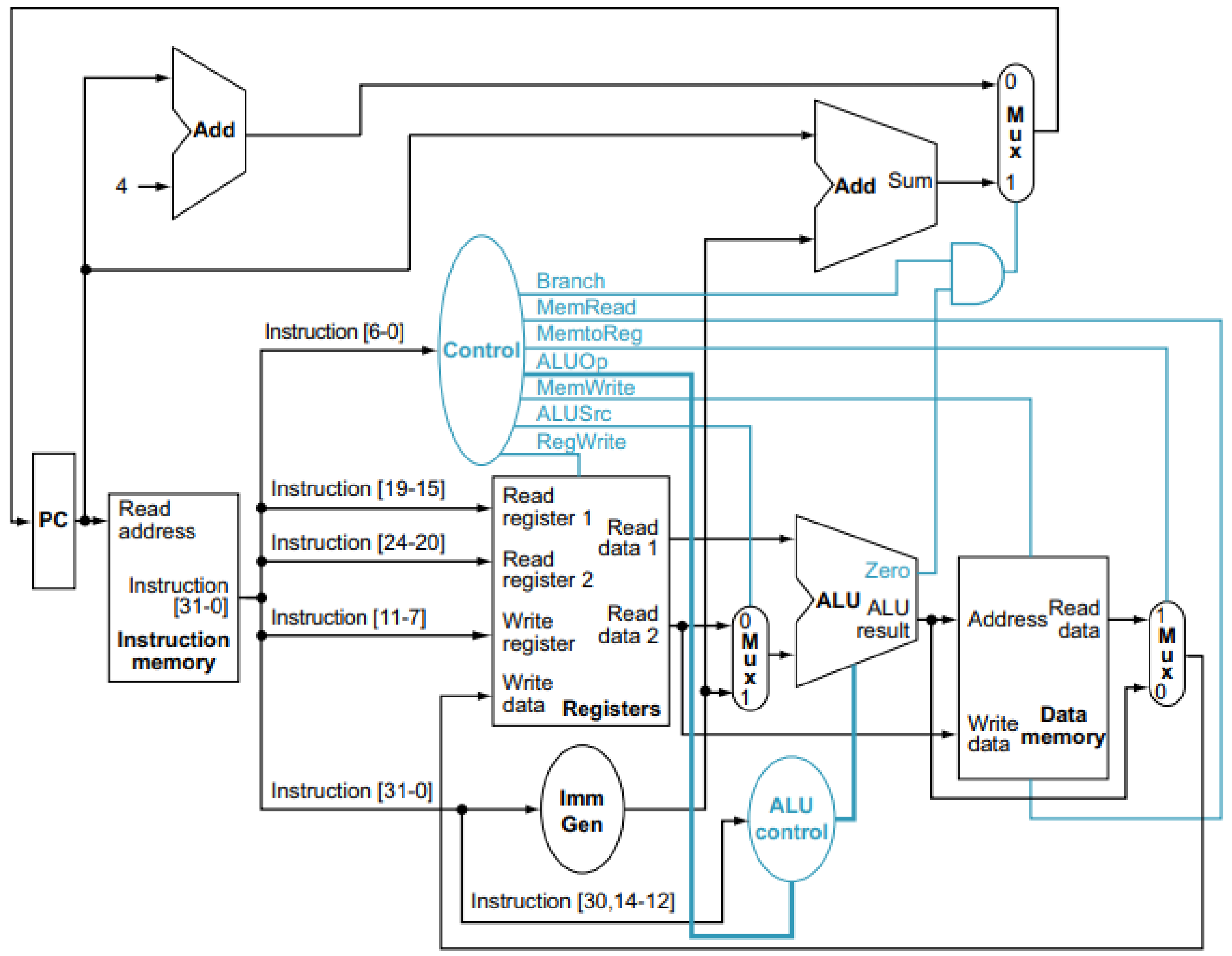


Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

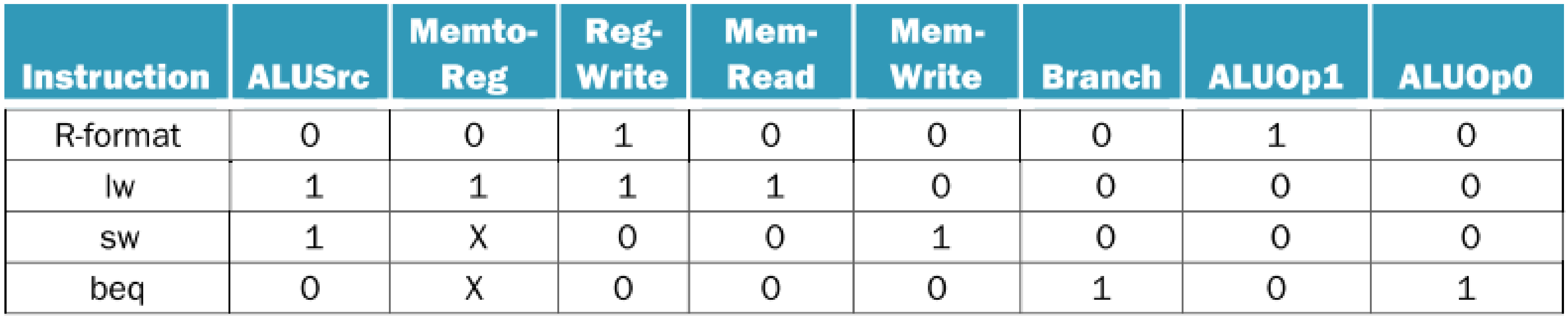
The datapath with all necessary multiplexors and all control lines identified



The simple datapath with the control unit

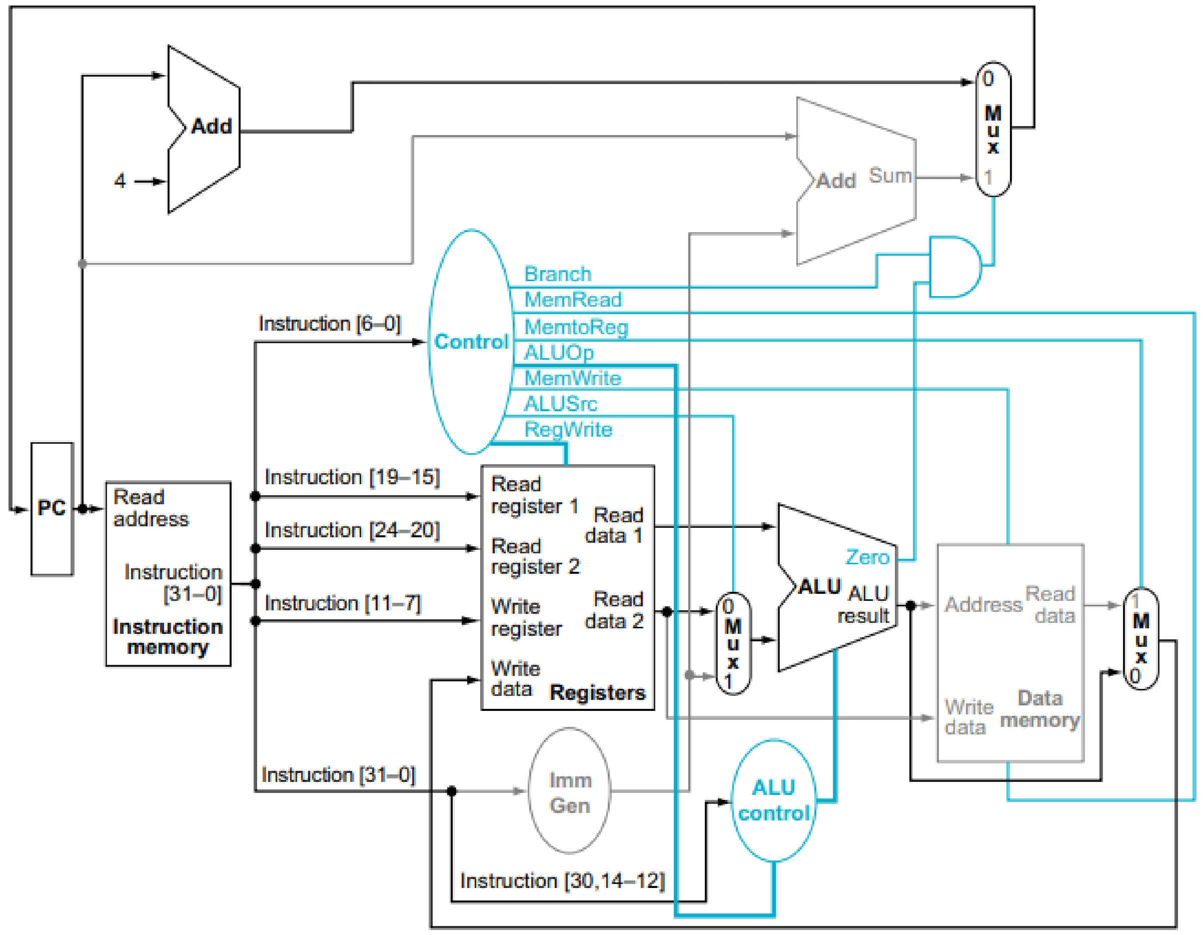


The setting of the control lines is completely determined by the opcode fields of the instruction.



Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
lw	1	1	1	1	0	0	0	0
sw	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

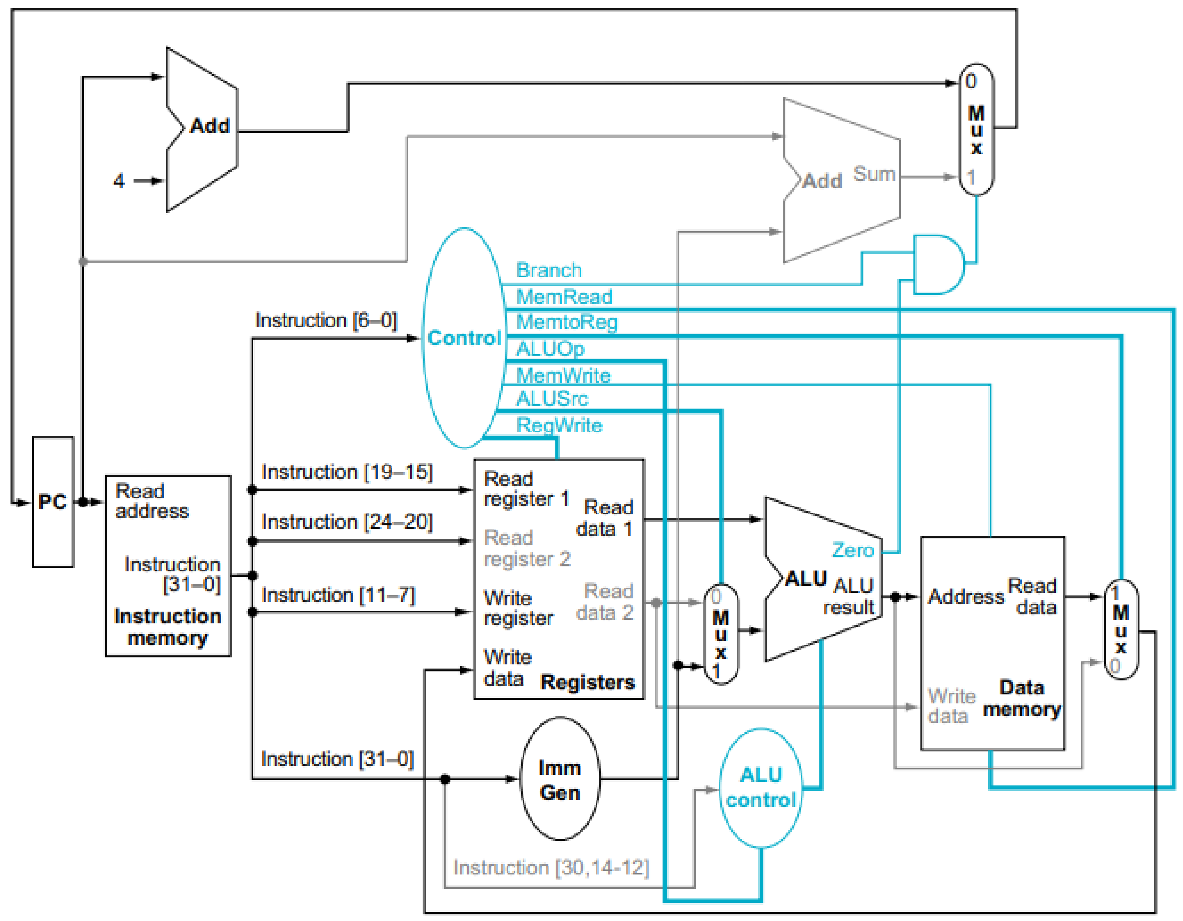
Operation of the Datapath for an R-type instruction



Steps for register type (add x1, x2, x3)

1. The instruction is fetched, and the PC is incremented.
2. Two registers, x2 and x3, are read from the register file; also, the main control unit computes the setting of the control lines during this step
3. The ALU operates on the data read from the register file, using portions of the opcode to generate the ALU function.
4. The result from the ALU is written into the destination register (x1) in the register file

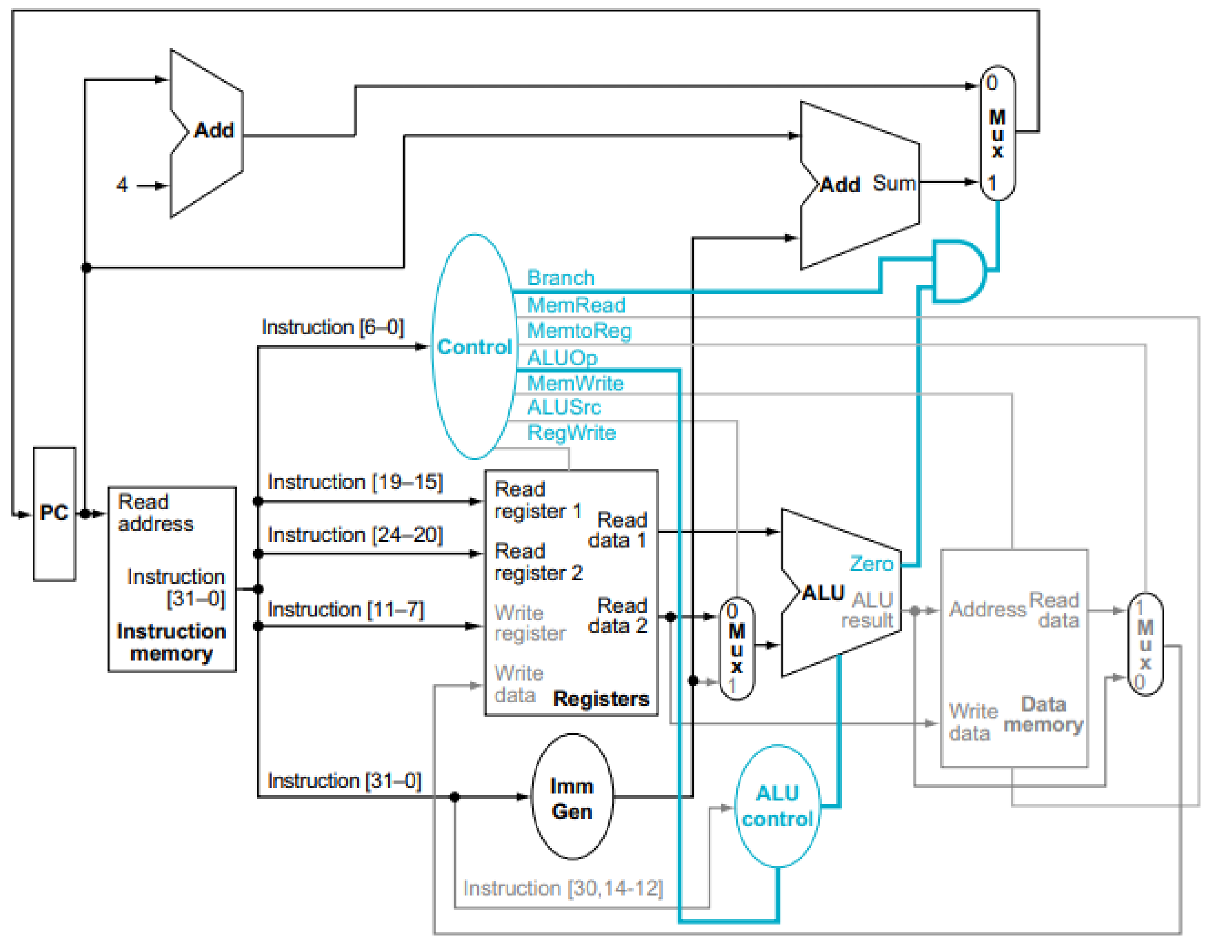
The datapath in operation for a load instruction.



Steps for load type (lw x1, offset(x2))

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register (x2) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended 12 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file (x1).

The datapath in operation for branch if equal.

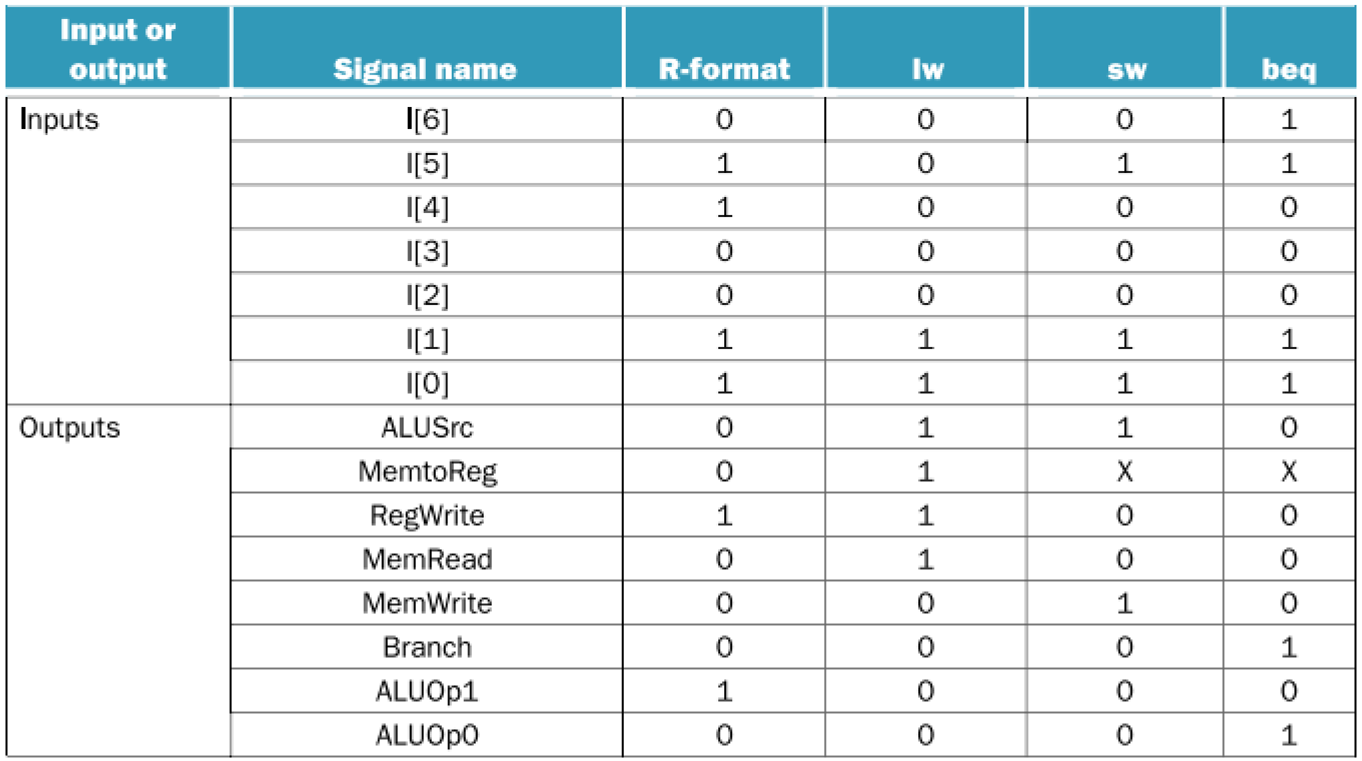


Steps for branch type(beq x1, x2, offset)

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. Two registers, x1 and x2, are read from the register file
3. The ALU subtracts one data value from the other data value, both read from the register file. The value of PC is added to the sign-extended, 12 bits of the instruction (offset) left shifted by one; the result is the branch target address.
4. The Zero status information from the ALU is used to decide which adder result to store in the PC

The control function for the simple single-cycle implementation specified by truth table

- The outputs are the control lines, and the inputs are the opcode bits



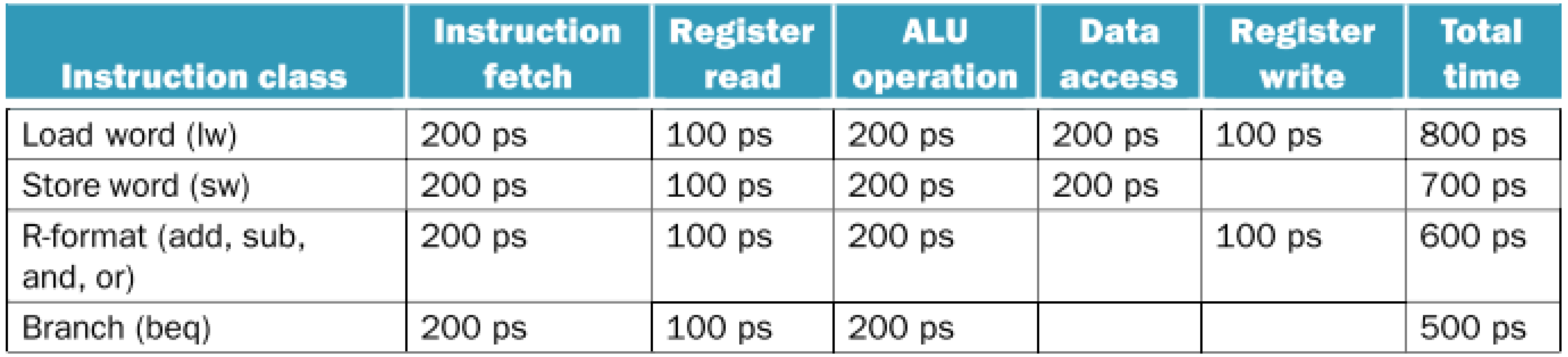
Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Pipelining

- An implementation technique in which multiple instructions are overlapped in execution.

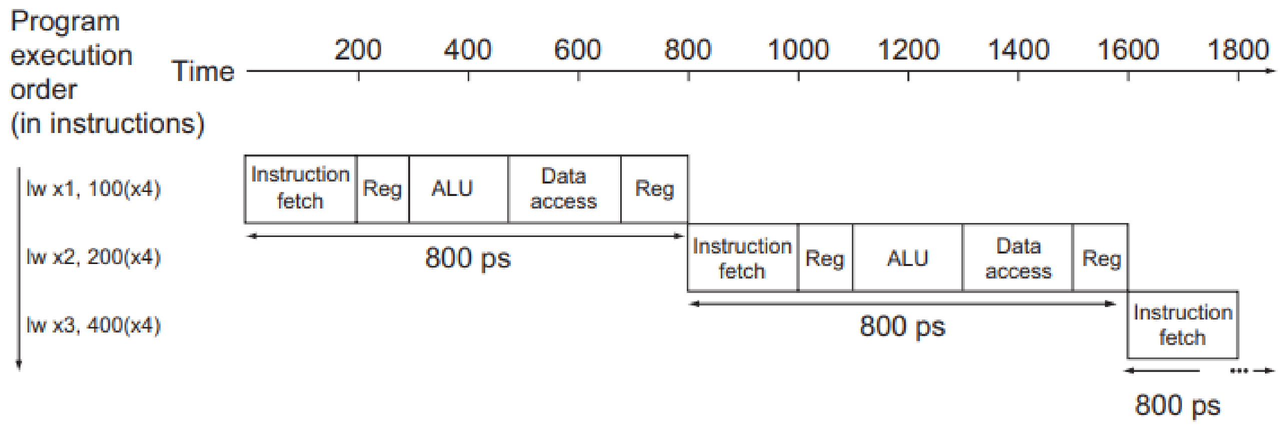
1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

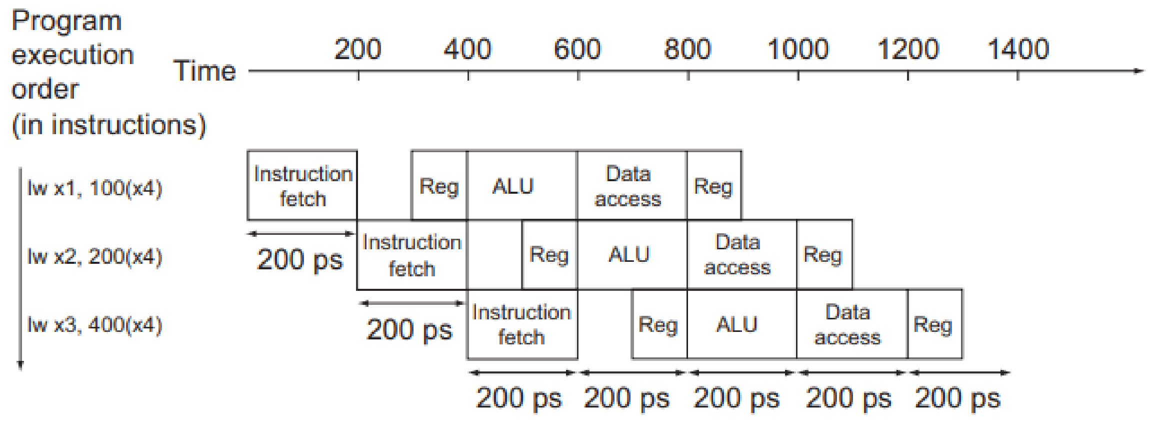
Single-Cycle versus Pipelined Performance



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Single-Cycle versus Pipelined Performance





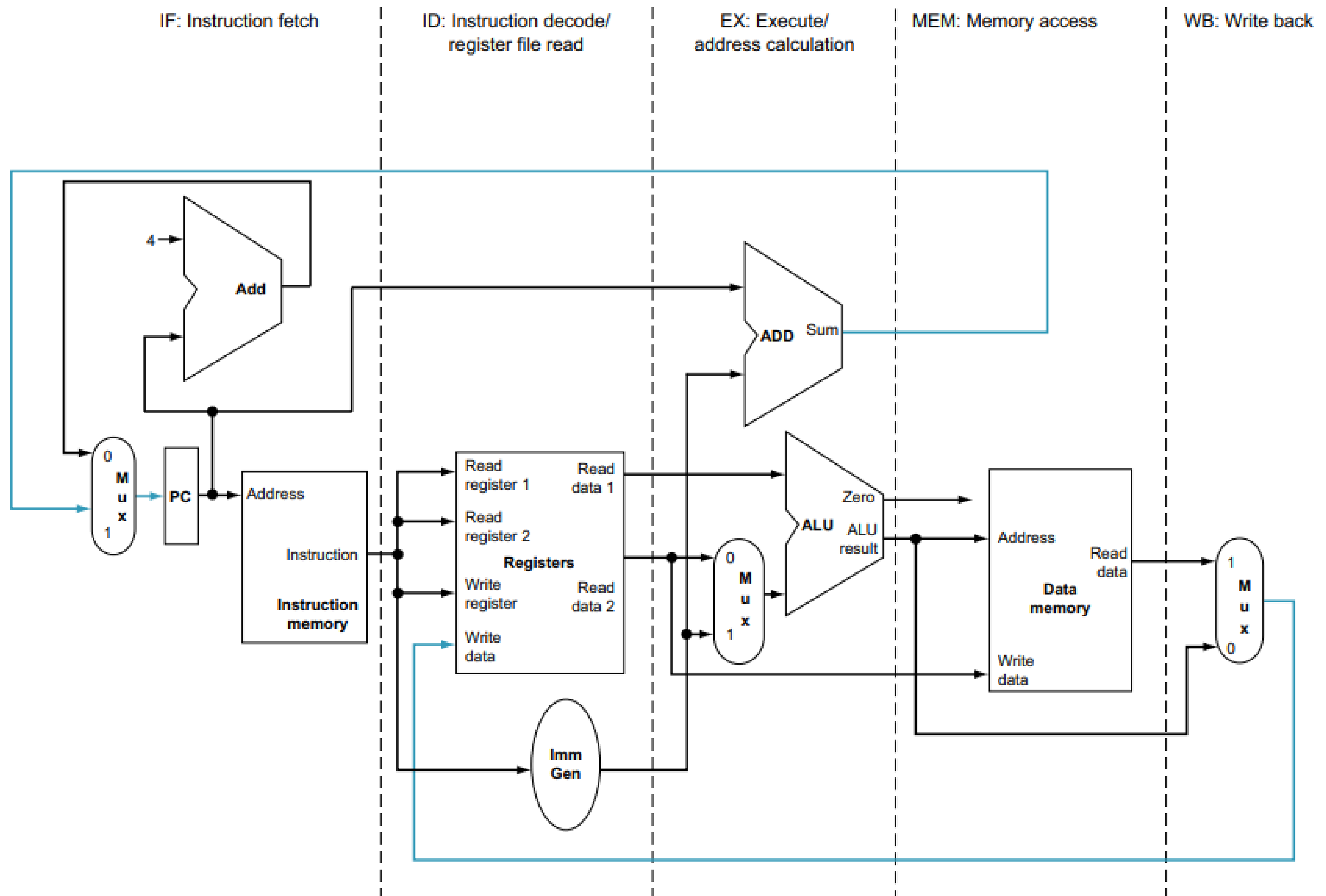
Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types
- structural hazard When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
- data hazard When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available.

- control hazard Also called branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
- Stall: Just operate sequentially until the first batch is dry and then repeat until you have the right formula

Pipelined Datapath and Control

- The single-cycle datapath

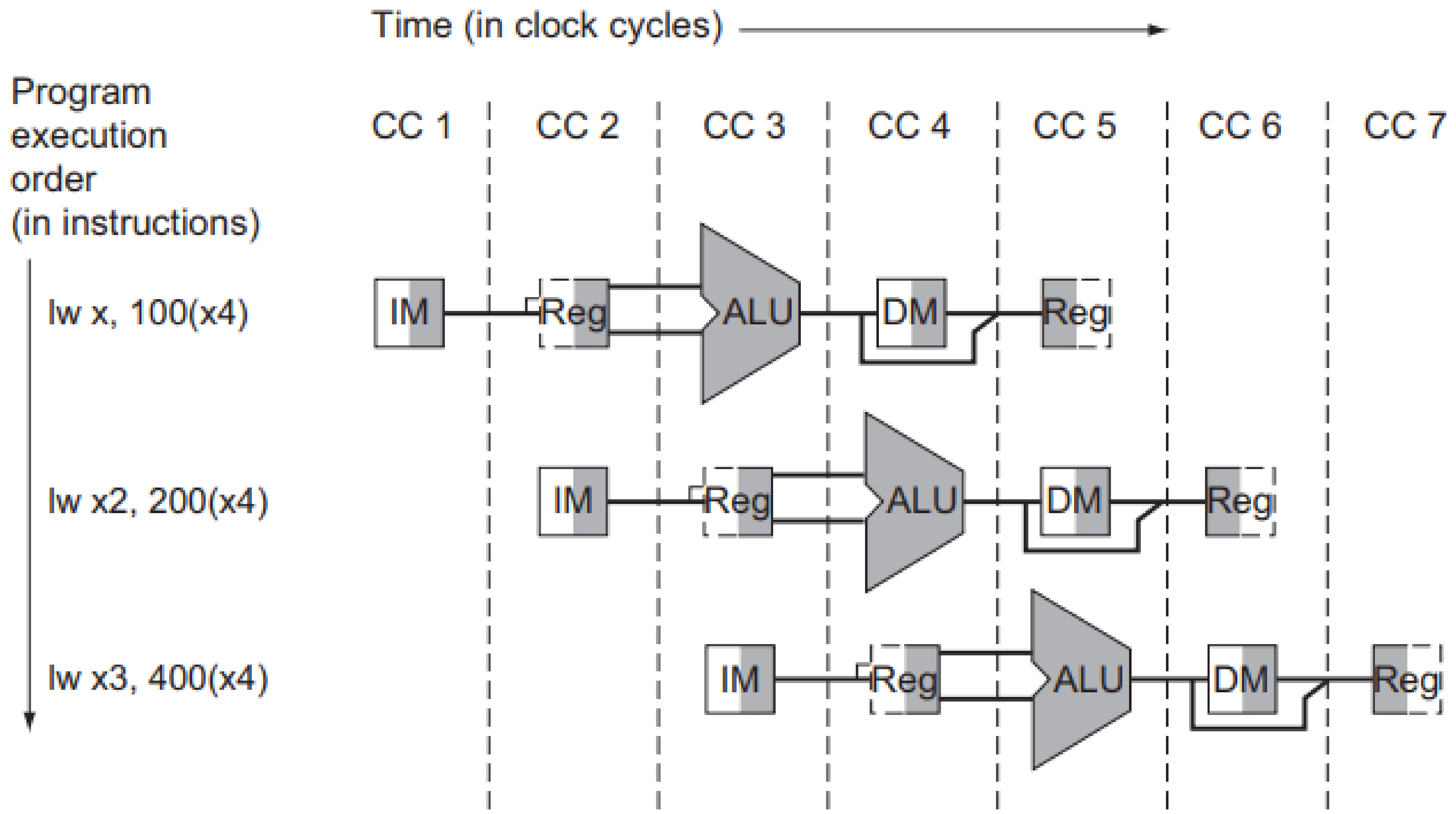


- The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle.

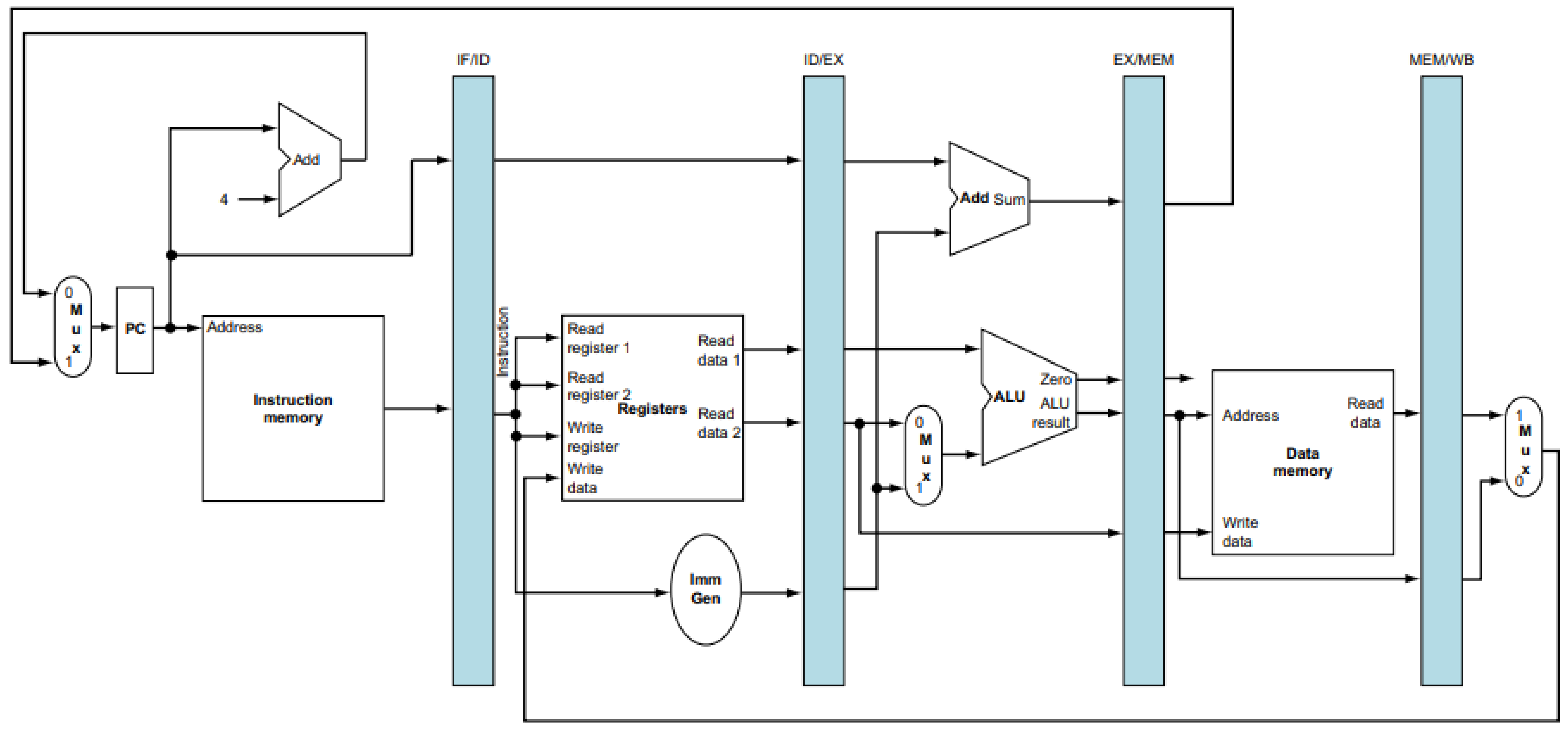
1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

- Data flows from left to right in the data path except when exceptions
- The write-back stage, which places the result back into the register file in the middle of the datapath (can lead to data hazard)
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage(can lead to control hazard)

Instructions being executed using the single-cycle datapath

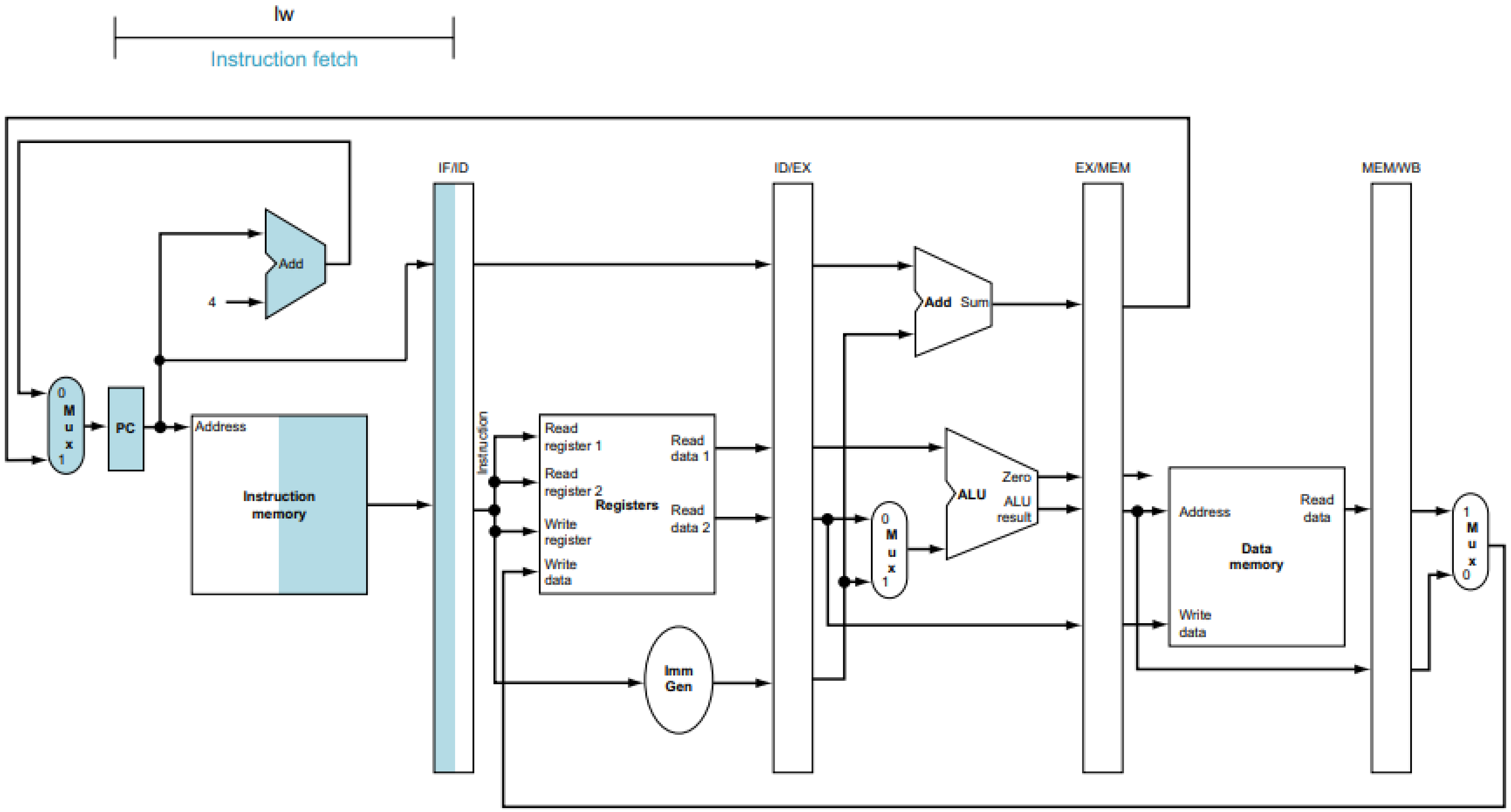


The pipelined version of the datapath



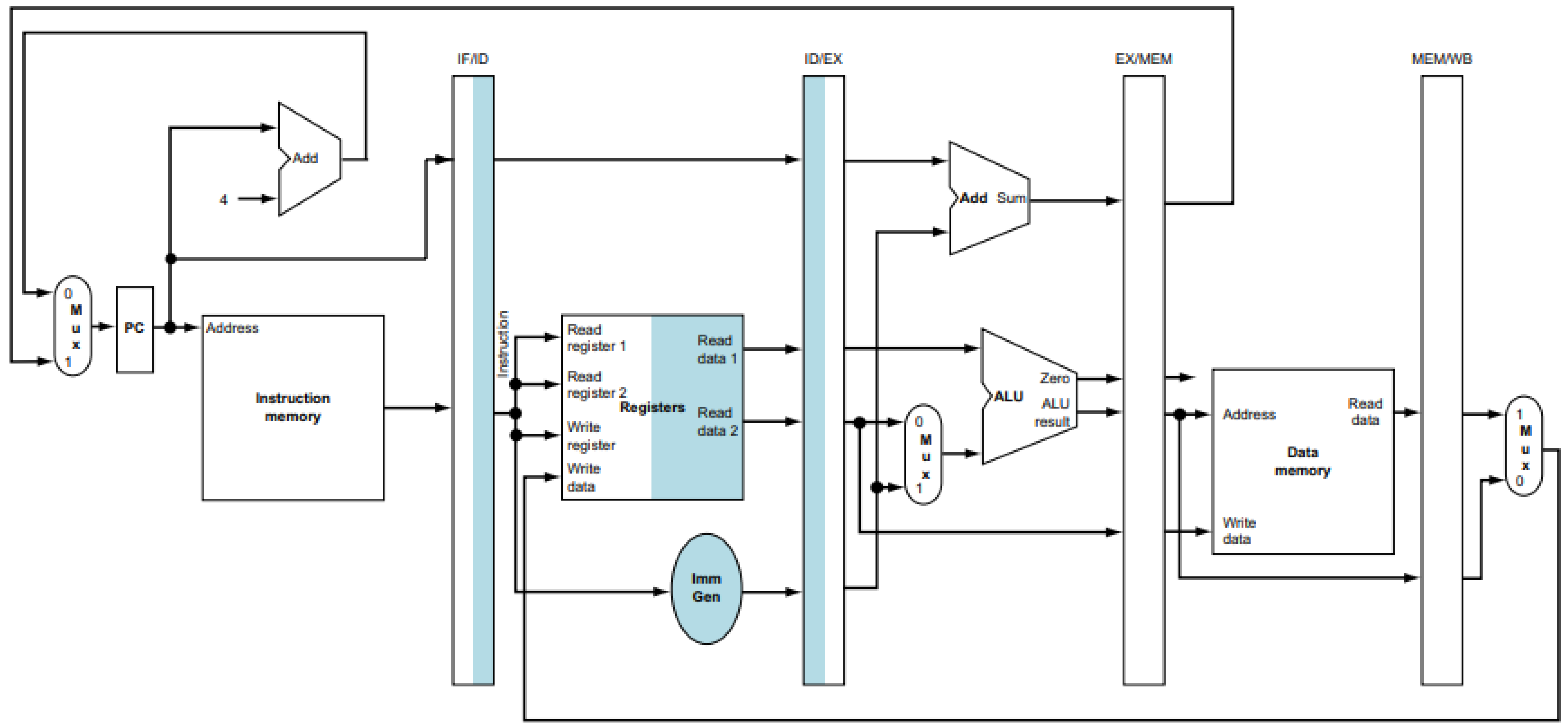
For Load Instruction

- The instruction lw with the name of the pipe stage that is active in each figure. The five stages are the following:
- Instruction fetch: The top portion of Figure shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This PC is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline

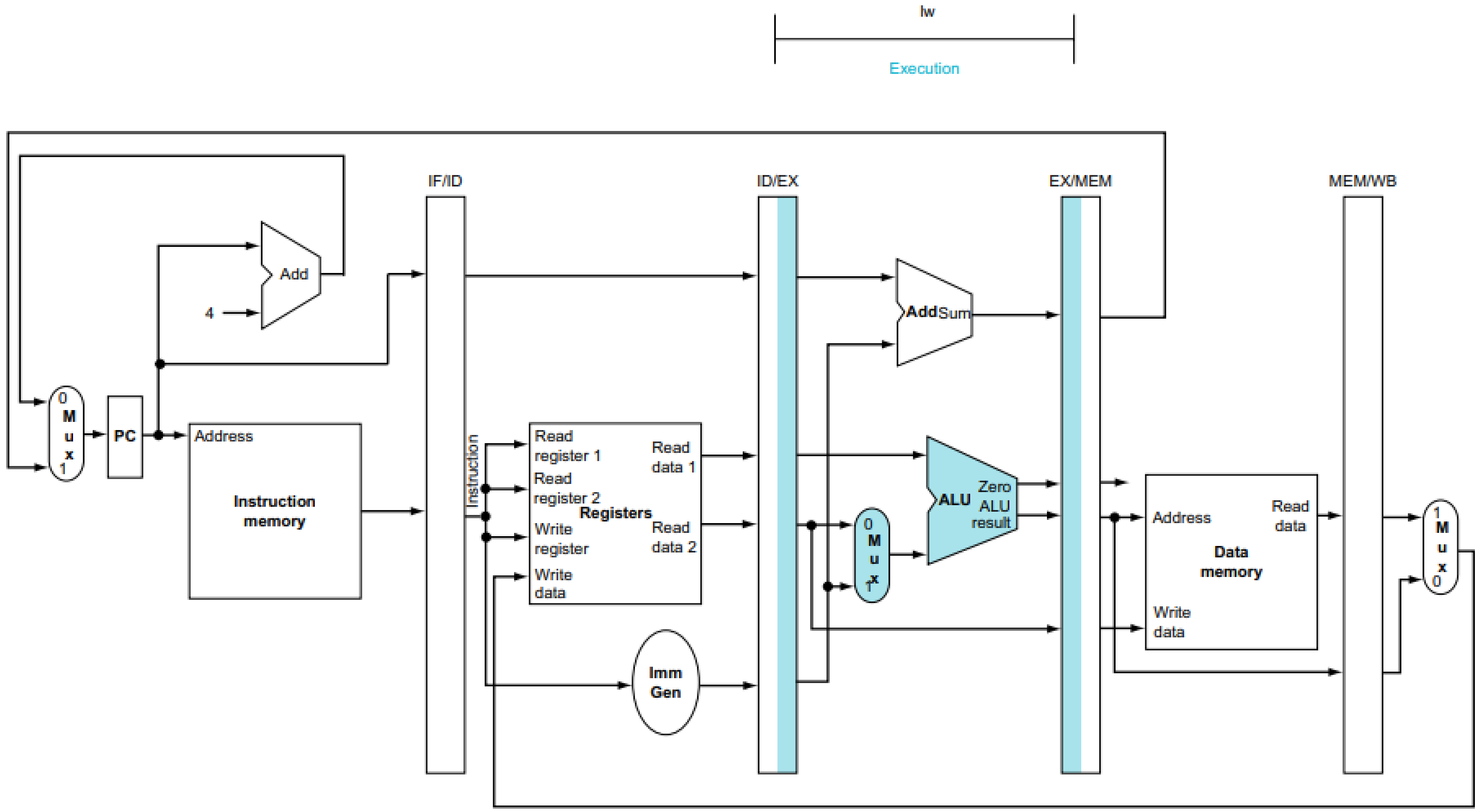

lw
Instruction fetch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction
memory
Instruction
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
Imm
Gen
Add Sum
ALU
Zero
ALU
result
Address
Data
memory
Read
data
Write
data

- Instruction decode and register file read: The bottom portion of Figure shows the instruction portion of the IF/ID pipeline register supplying the immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the PC address. We again transfer everything that might be needed by any instruction during a later clock cycle lw

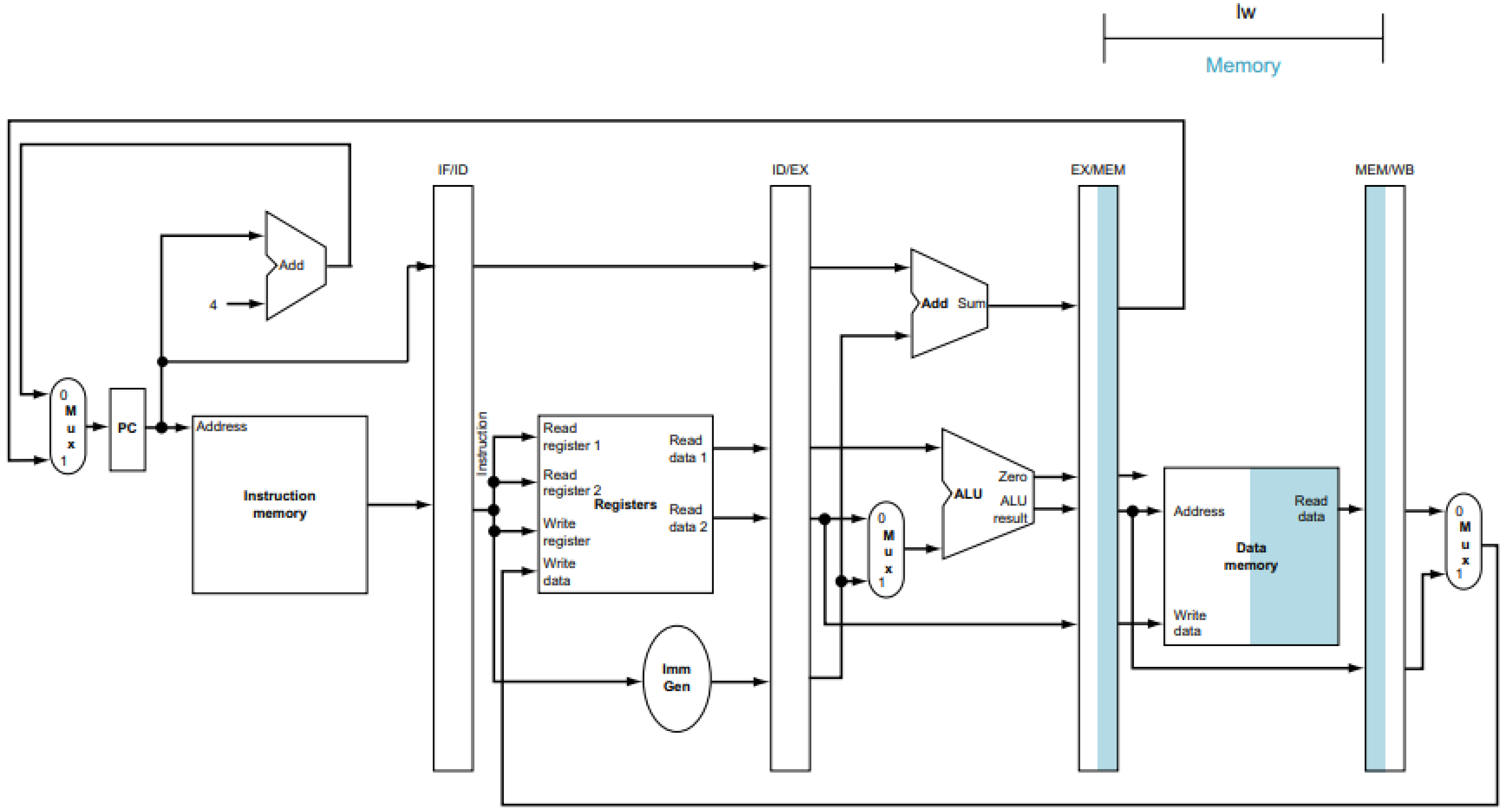
Instruction decode



- Execute or address calculation: Figure shows that the load instruction reads the contents of a register and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Imm Gen
Add Sum
0
M
u
x
1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
1
M
u
x
0

- Memory access: The top portion of Figure shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Imm Gen
Add Sum
ALU
Zero
ALU result
Address
Data memory
Write data
Read data

- Write-back: The bottom portion of Figure shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure

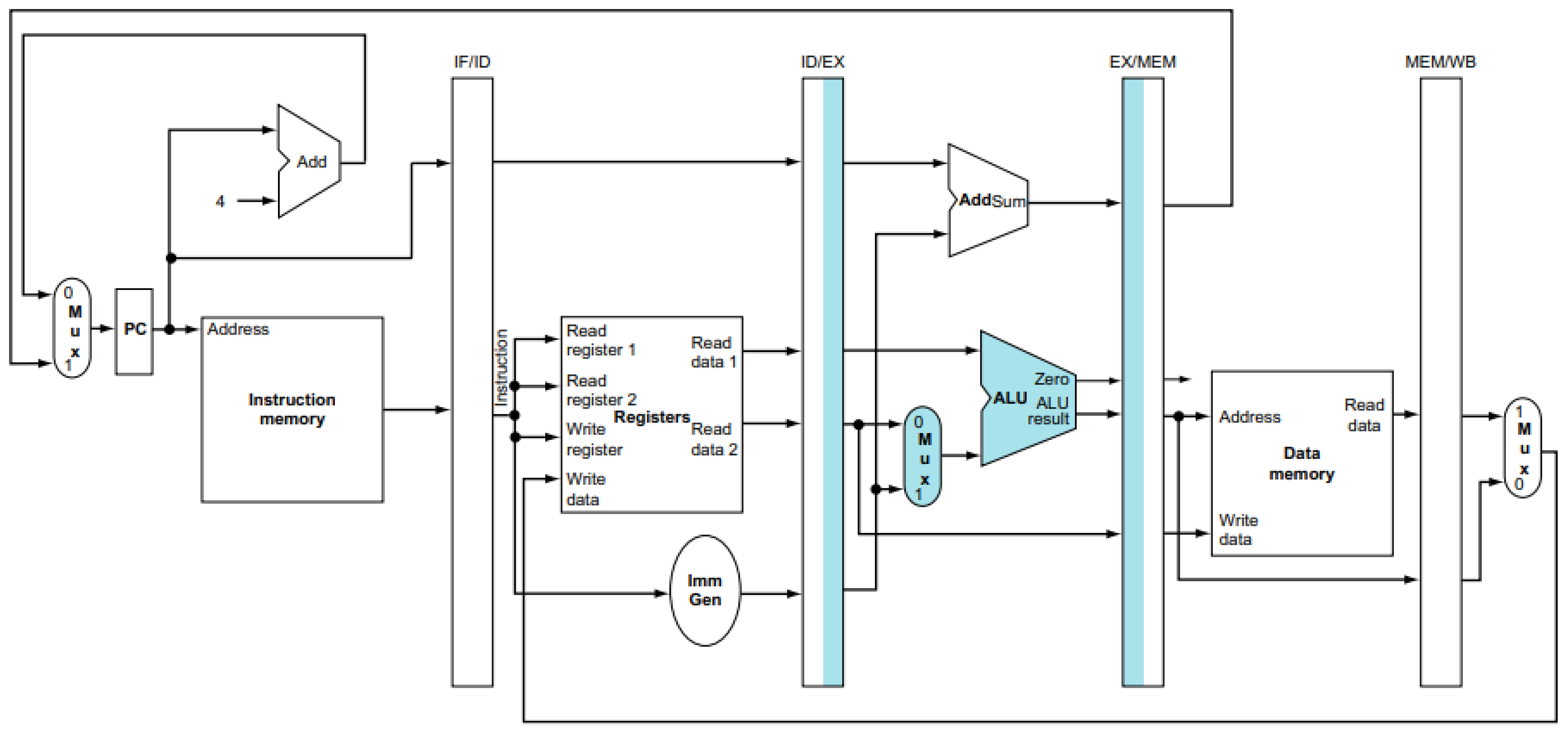
The five pipe stages of the store instruction:

- Instruction fetch: The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the Figure works for store as well as load
- Instruction decode and register file read: The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the immediate operand. These three 32-bit values are all stored in the ID/EX pipeline register. The Figure for load instructions also shows the operations of the second stage for stores. (While the store instruction uses the rs2 field to read the second register in this pipe stage, that detail is not shown in this pipeline diagram, so we can use the same figure for both.)

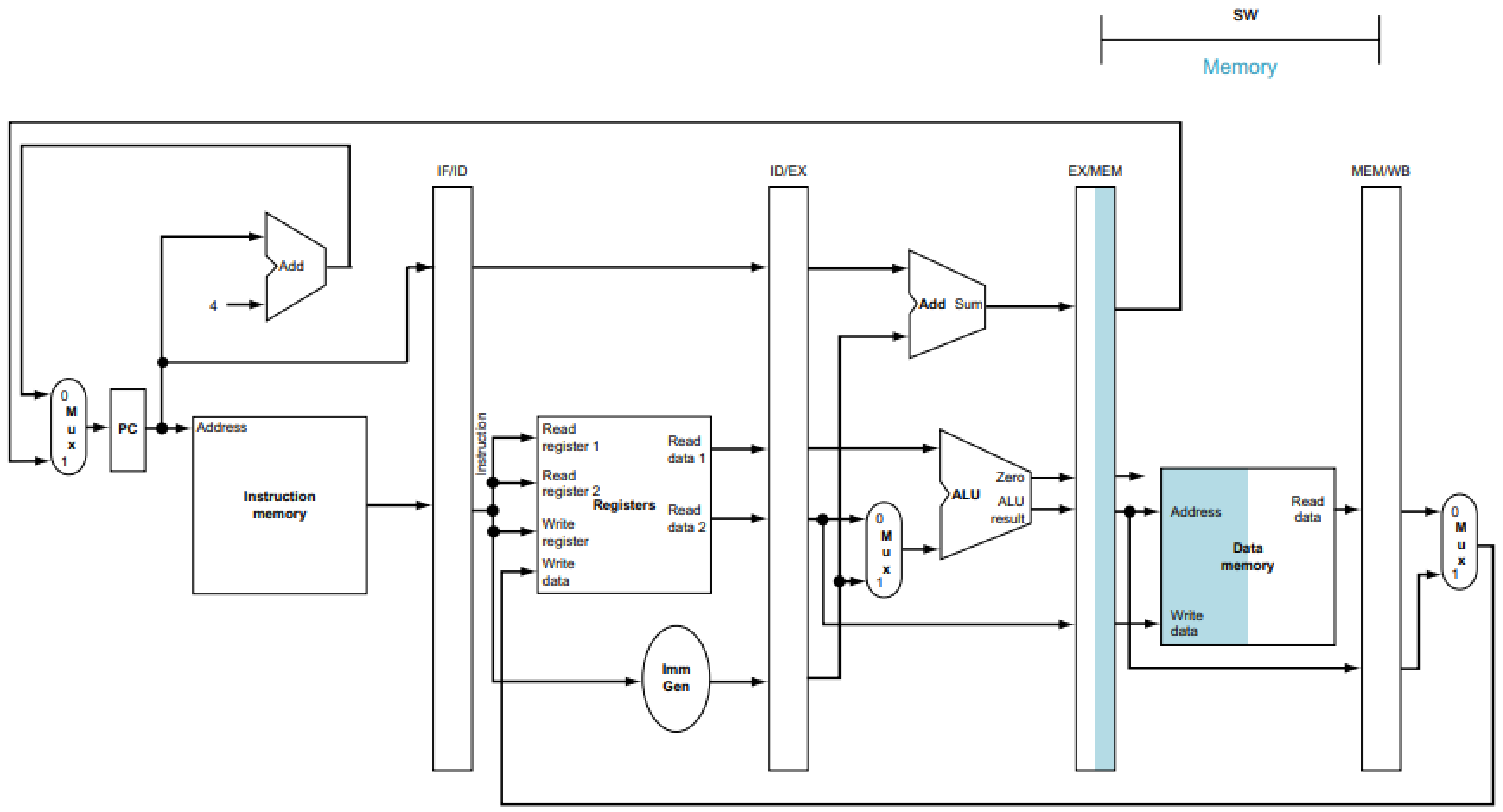
- Execute and address calculation: Figure shows the third step; the effective address is placed in the EX/MEM pipeline register.

sw

Execution



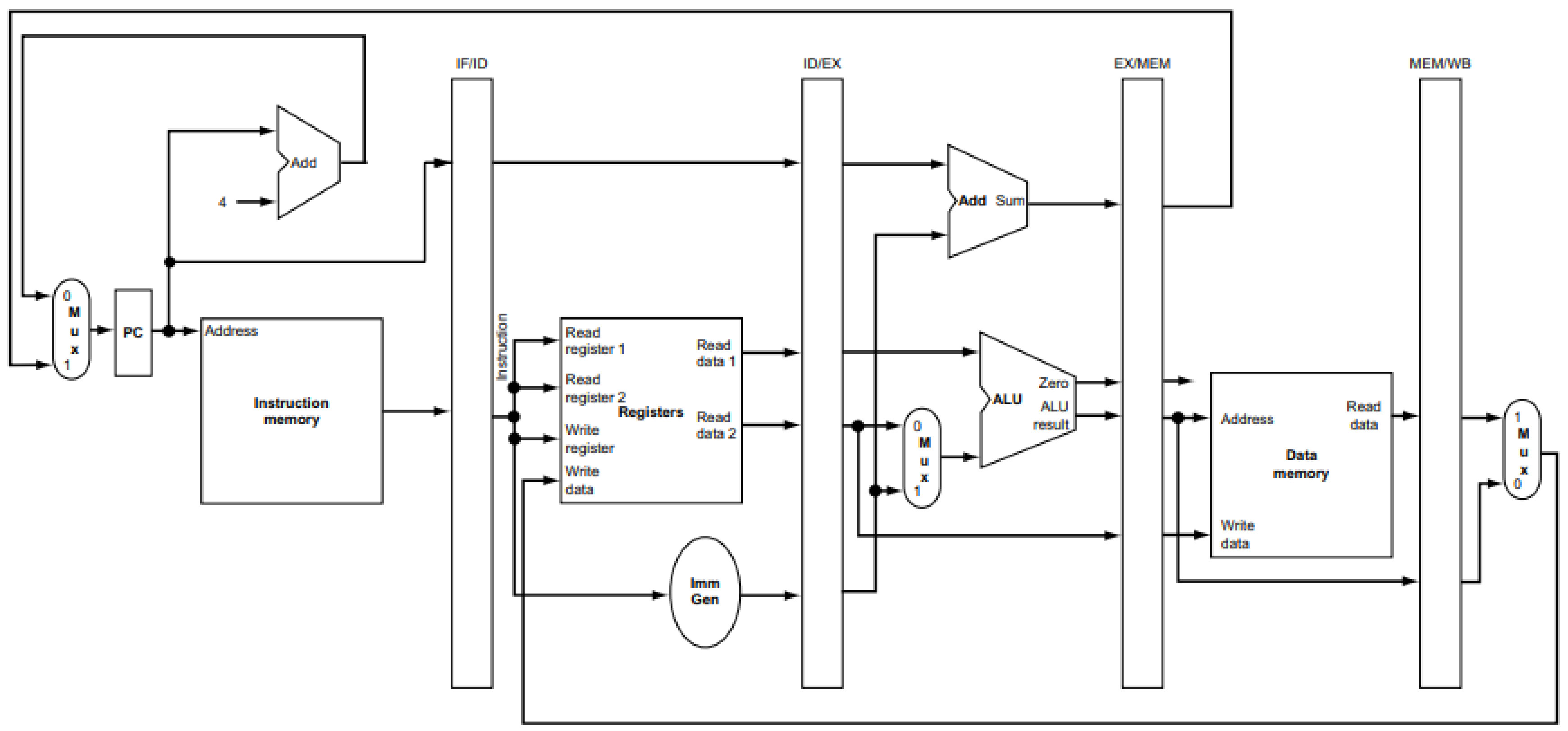
- Memory access: The top portion of Figure shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.


SW
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction
memory
Instruction
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
Imm
Gen
Add Sum
ALU
Zero
ALU
result
Address
Data
memory
Write
data
Read
data

- Write-back: The bottom portion of previous Figure shows the final step of the store. For this instruction, nothing happens in the write-back stage. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions. Hence, an instruction passes through a stage even if there is nothing to do, because later instructions are already progressing at the maximum rate.

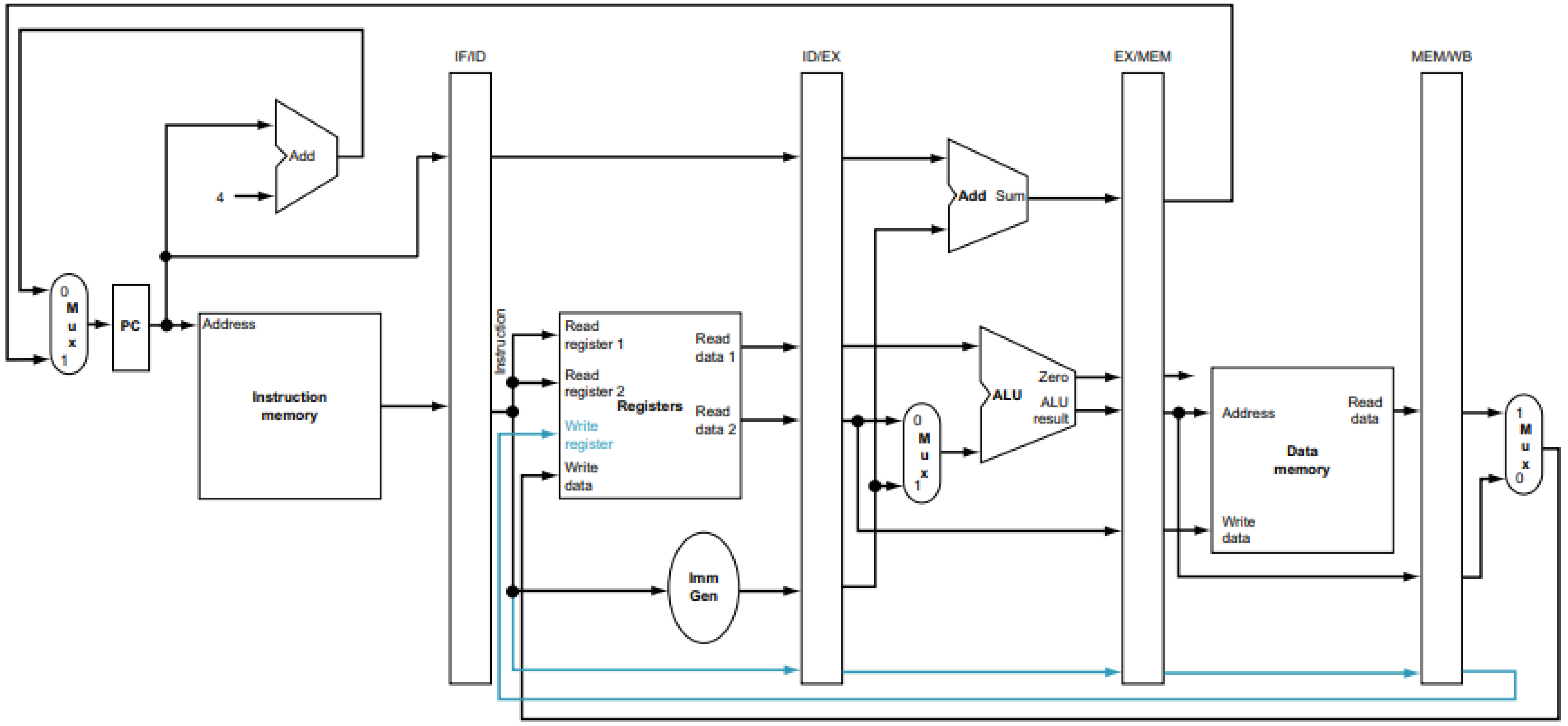
SW

Write-back

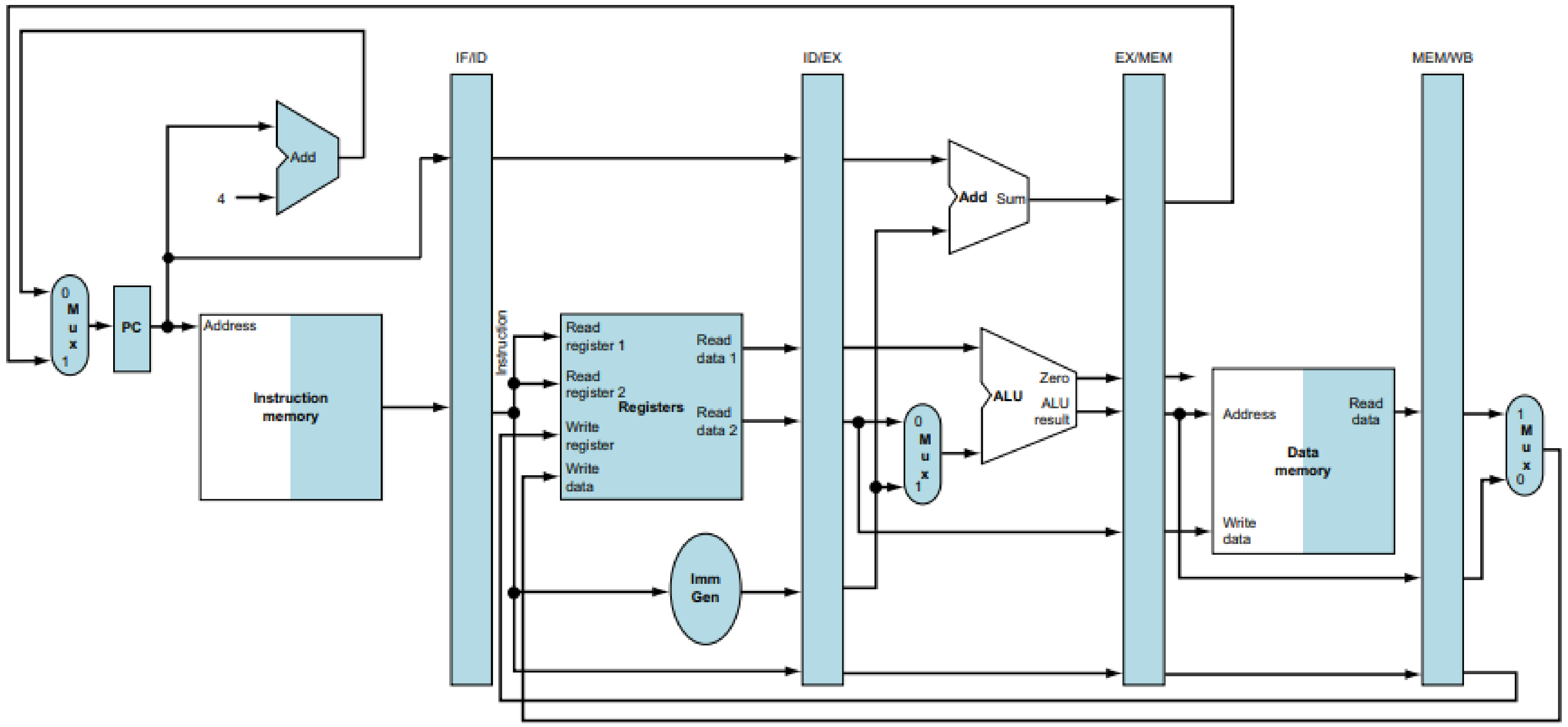


- Load and store illustrate a second key point: each logical component of the datapath—such as instruction memory, register read ports, ALU, data memory, and register write port—can be used only within a single pipeline stage. Otherwise, we would have a structural hazard

The corrected pipelined datapath to handle the load instruction properly.(multiple clock cycle datapath)

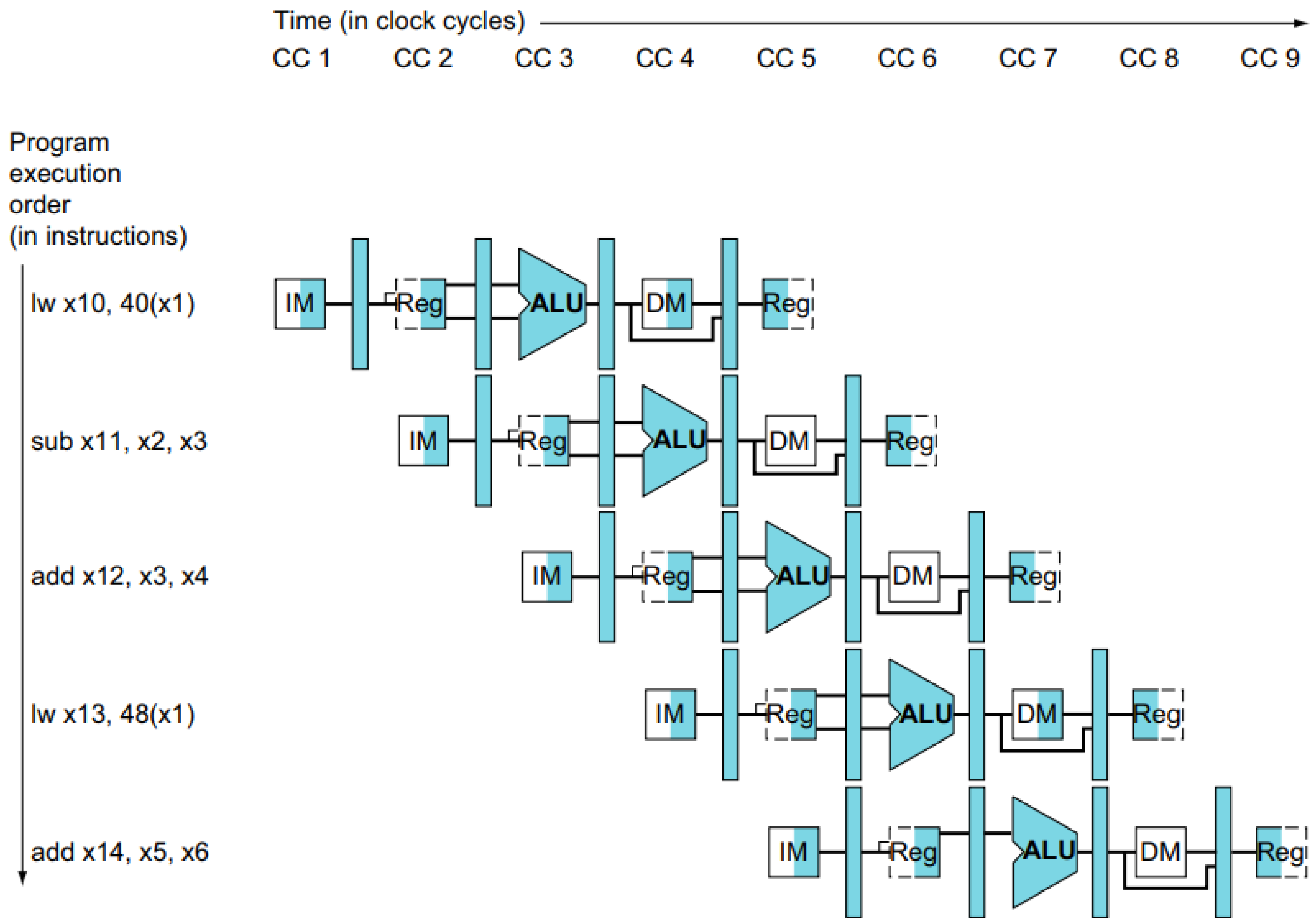


The portion of the datapath that is used in all five stages of a load instruction

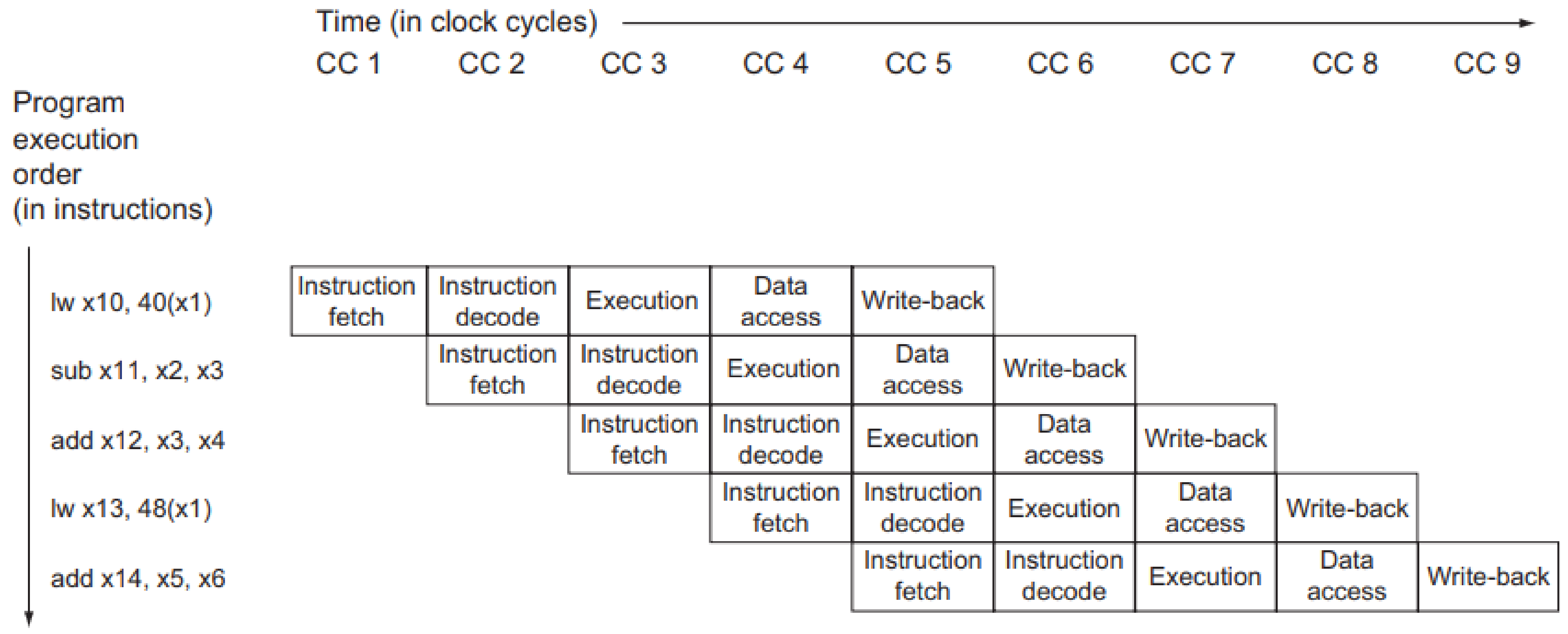


```
lw x10, 40(x1)
sub x11, x2, x3
add x12, x3, x4
lw x13, 48(x1)
add x14, x5, x6
```

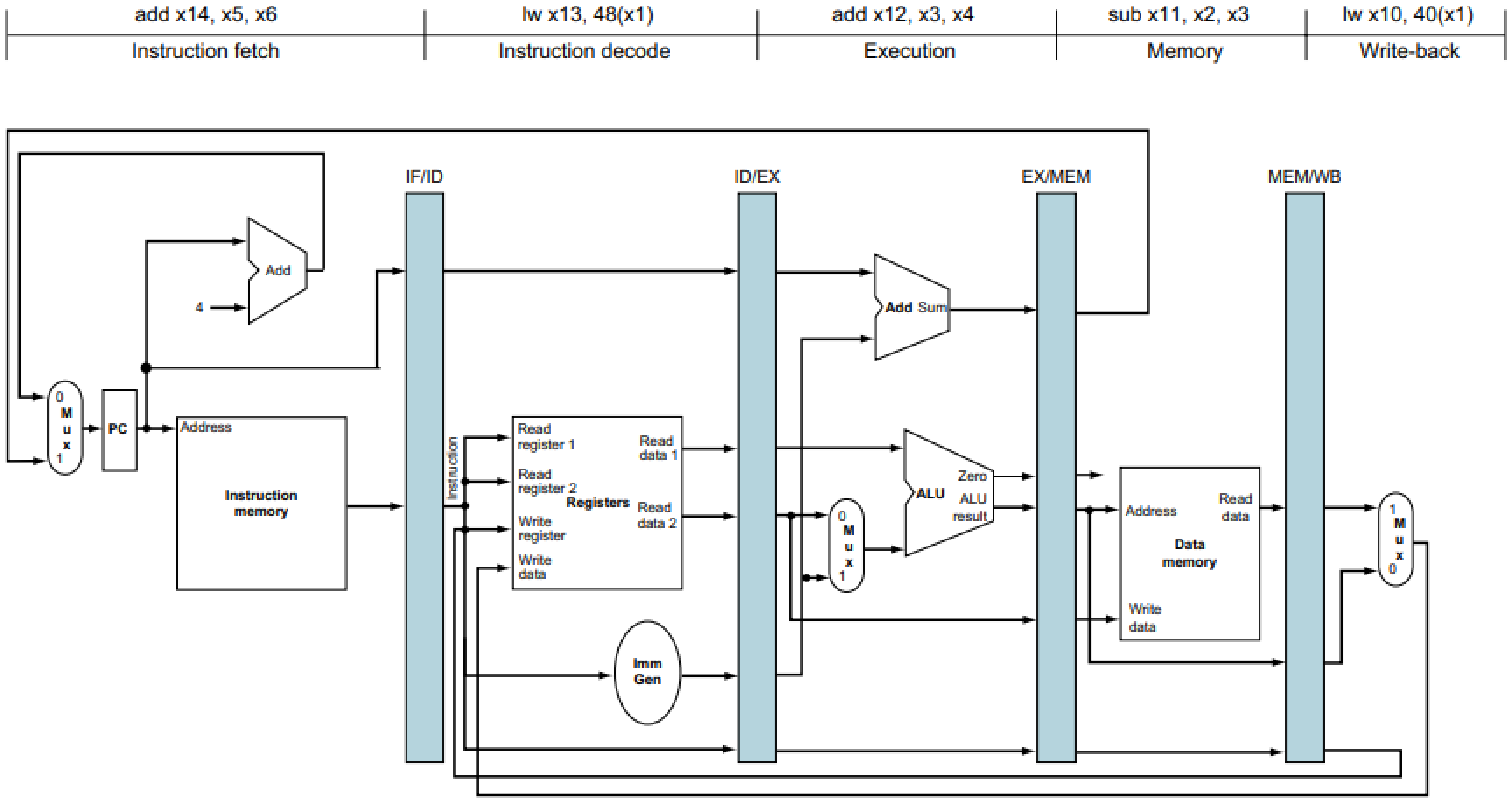
Multiple-clock-cycle pipeline diagram of five instructions



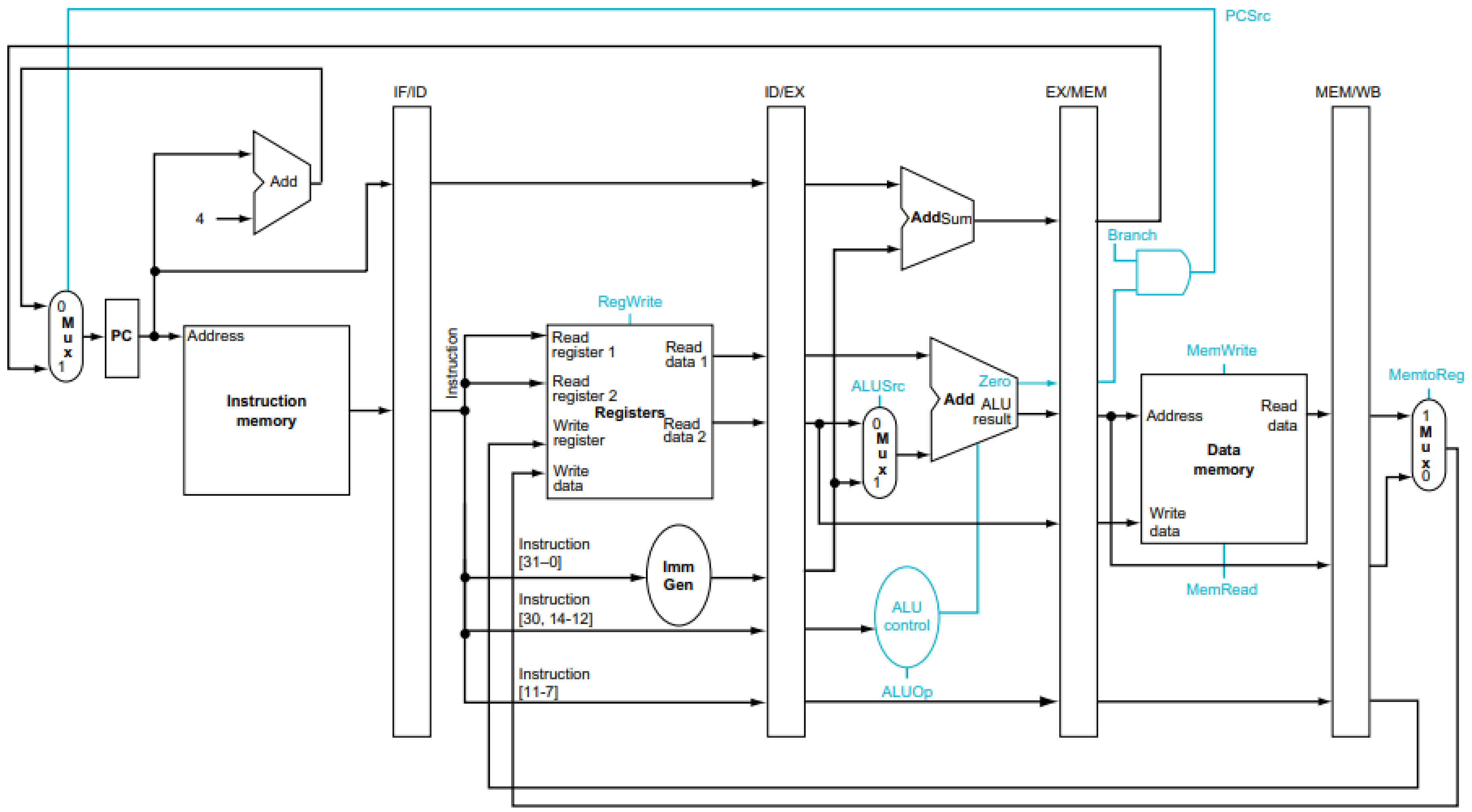
Traditional multiple-clock-cycle pipeline diagram of five instructions



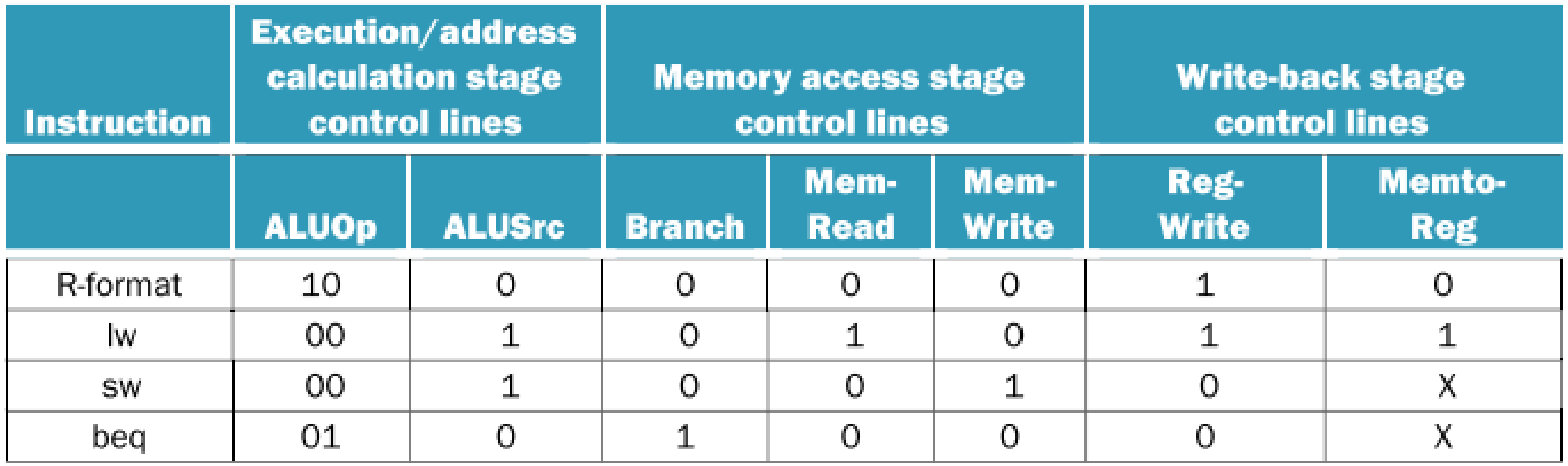
The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline



Pipelined Control-The pipelined datapath of with the control signals identified

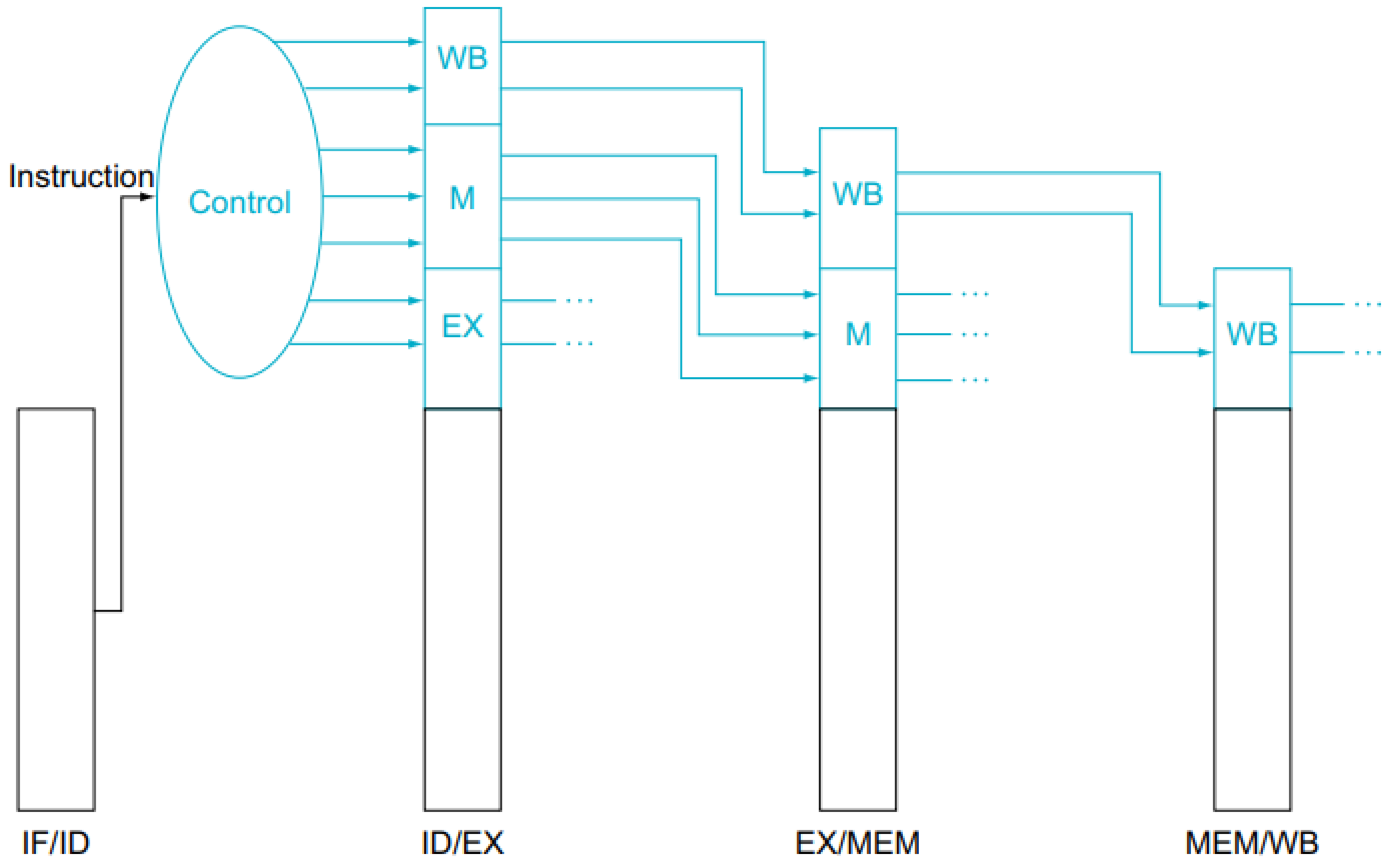


The values of the control lines

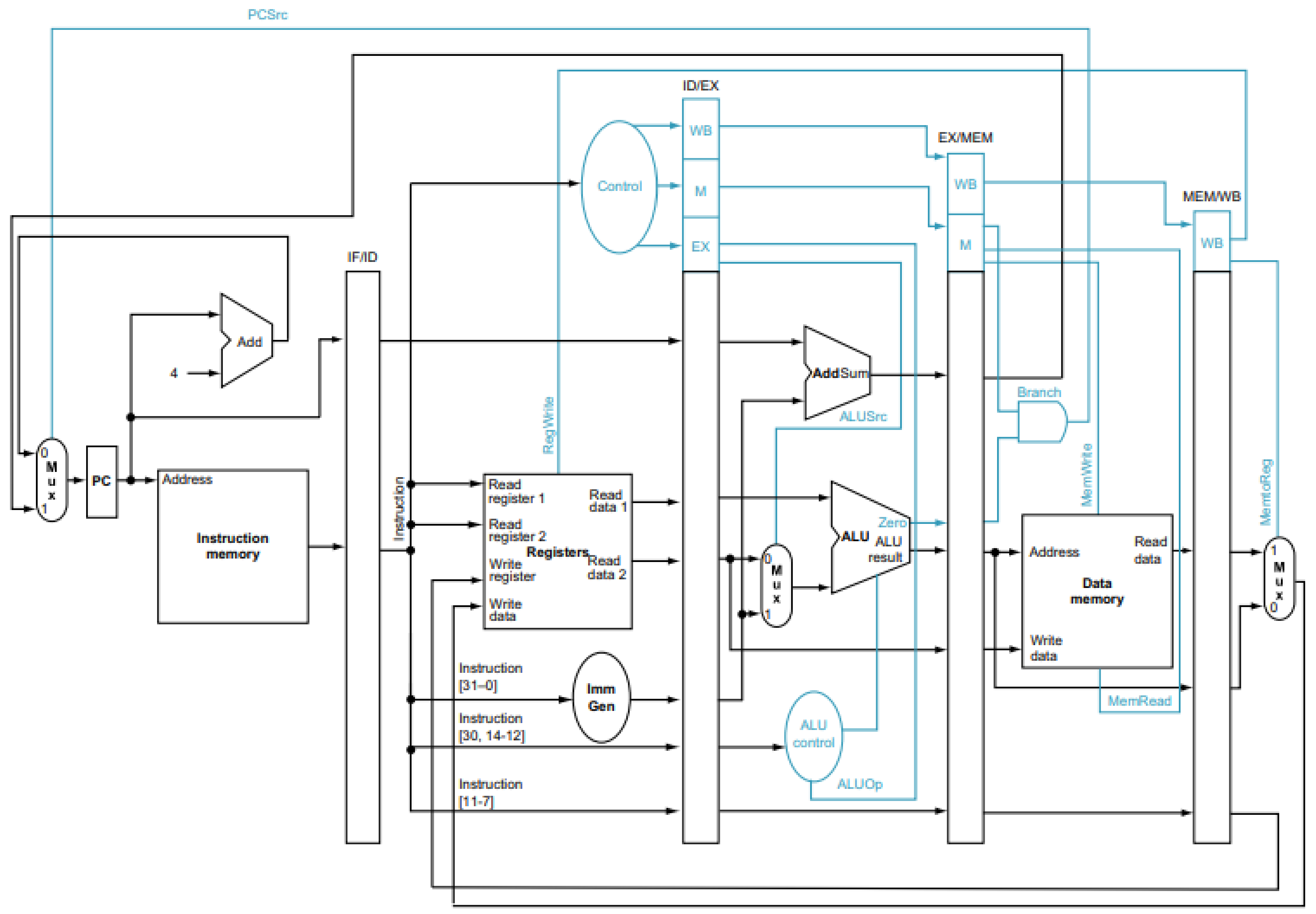


Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

The seven control lines for the final three stages



The pipelined datapath, with the control signals connected to the control portions of the pipeline registers



Data Hazards: Forwarding versus Stalling

```
sub   x2, x1, x3     // Register z2 written by sub
and   x12, x2, x5    // 1st operand(x2) depends on sub
or    x13, x6, x2    // 2nd operand(x2) depends on sub
add   x14, x2, x2    // 1st(x2) & 2nd(x2) depend on sub
sw    x15, 100(x2)   // Base (x2) depends on sub
```

- the two pairs of hazard conditions are

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

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

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1

2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

Classify the dependences in this sequence

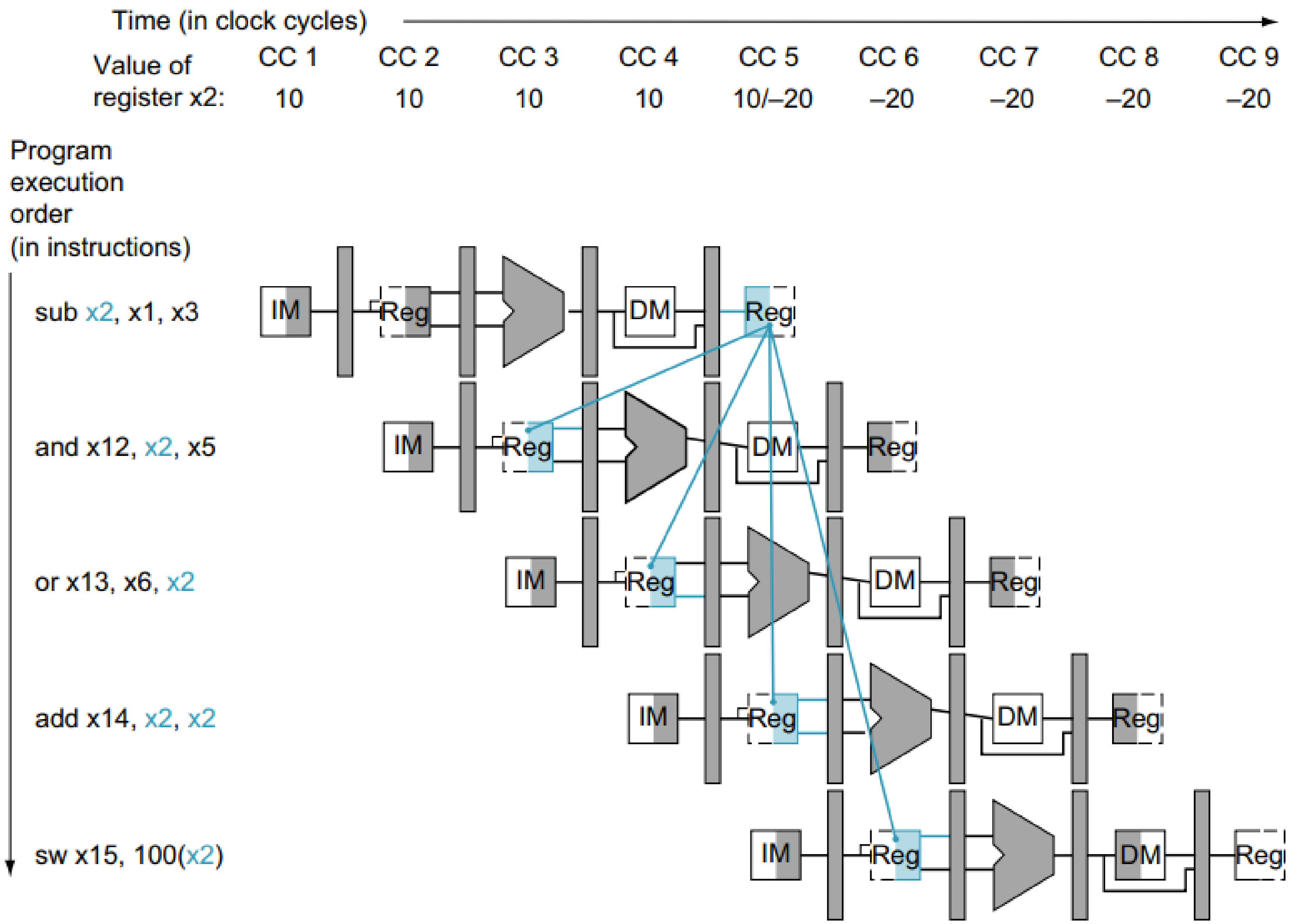
```
sub  x2, x1, x3       // Register x2 set by sub
and  x12,  x2, x5     // 1st operand(z2) set by sub
or   x13,  x6, x2     // 2nd operand(x2) set by sub
add  x14,  x2, x2     // 1st(x2) & 2nd(x2) set by sub
sw   x15,  100(x2)    // Index(x2) set by sub
```

- The `sub-or` is a type 2b hazard:

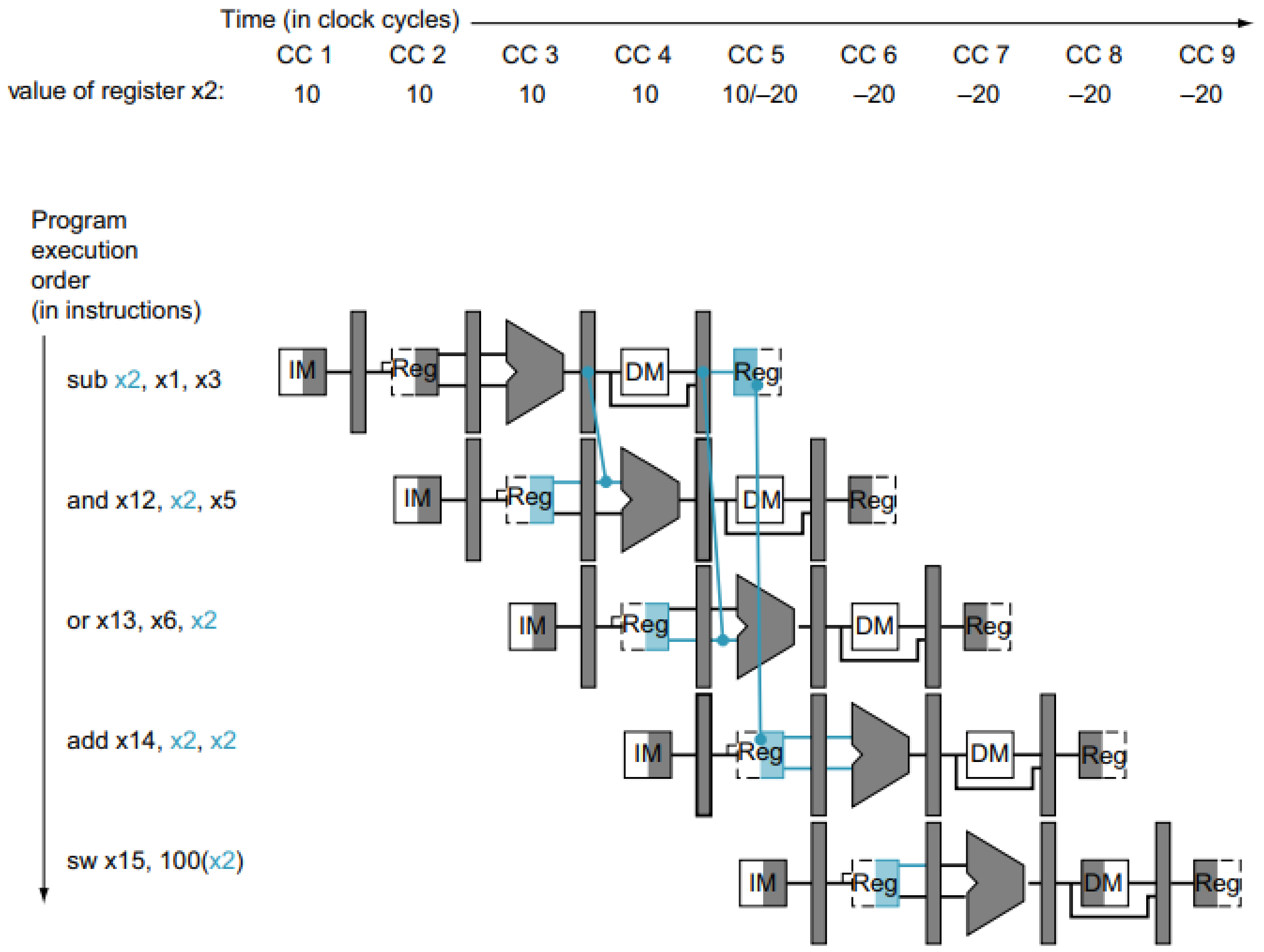
  MEM/WB.RegisterRd = ID/EX.RegisterRs2 = `x2`

- The two dependences on `sub-add` are not hazards because the register file supplies the proper data during the ID stage of `add`.
- There is no data hazard between `sub` and `sw` because `sw` reads `x2` the clock cycle *after* `sub` writes `x2`.

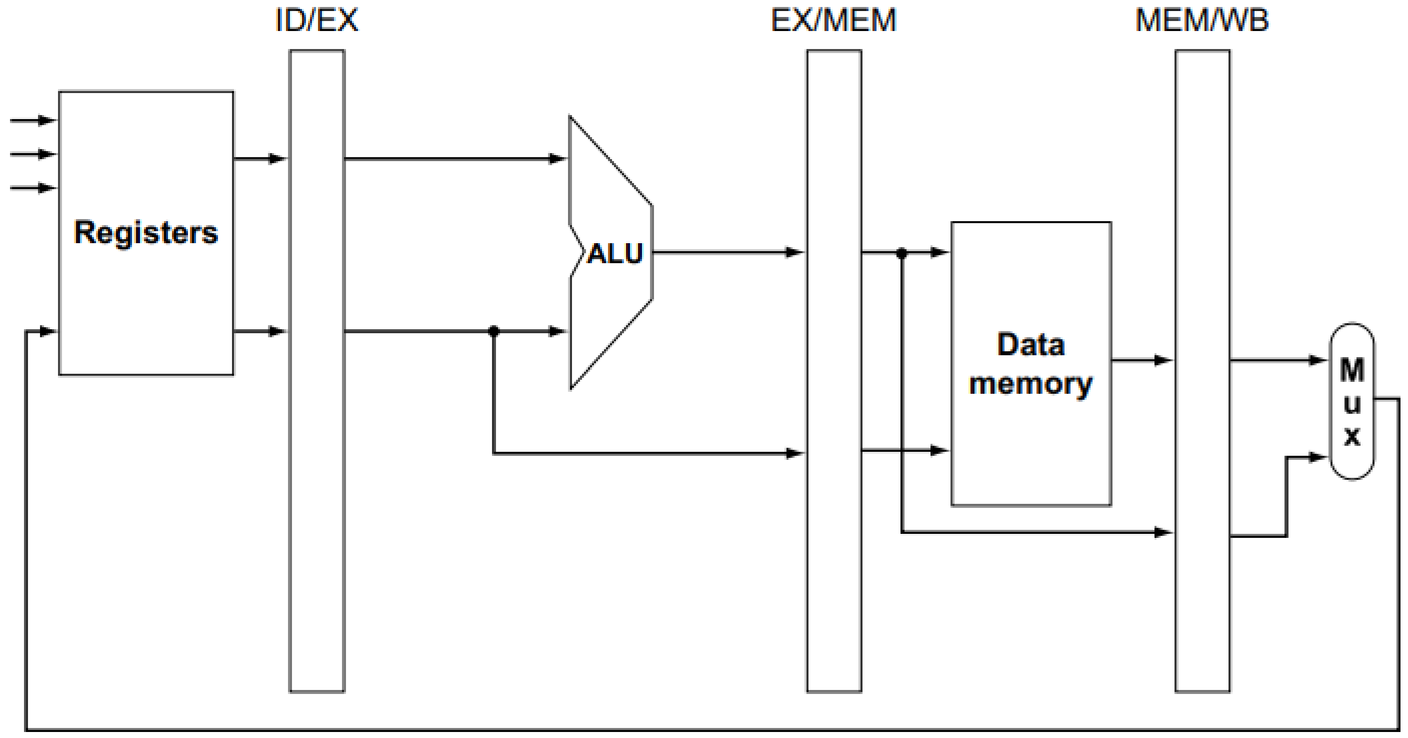
Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences



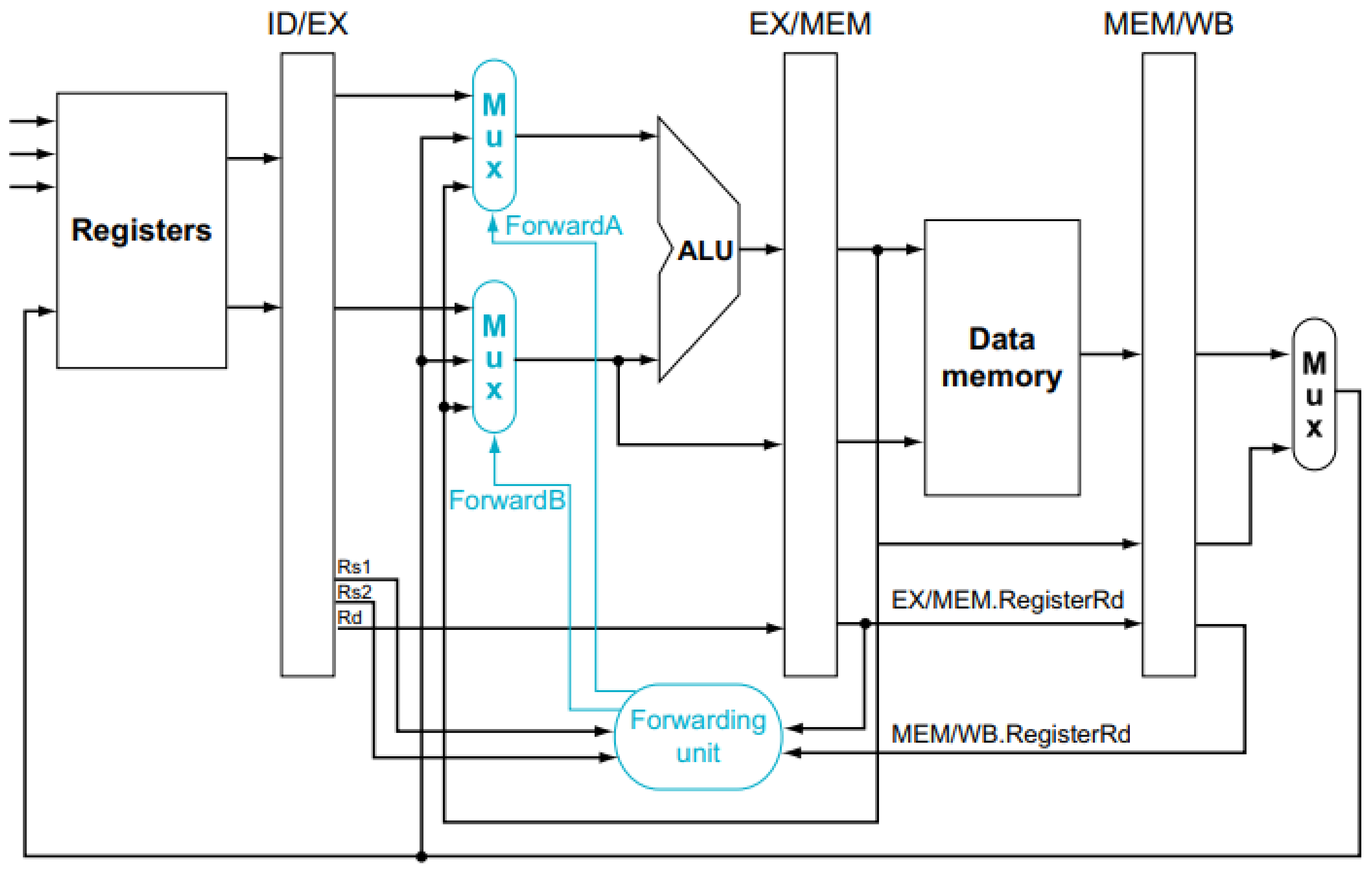
The dependences between the pipeline registers



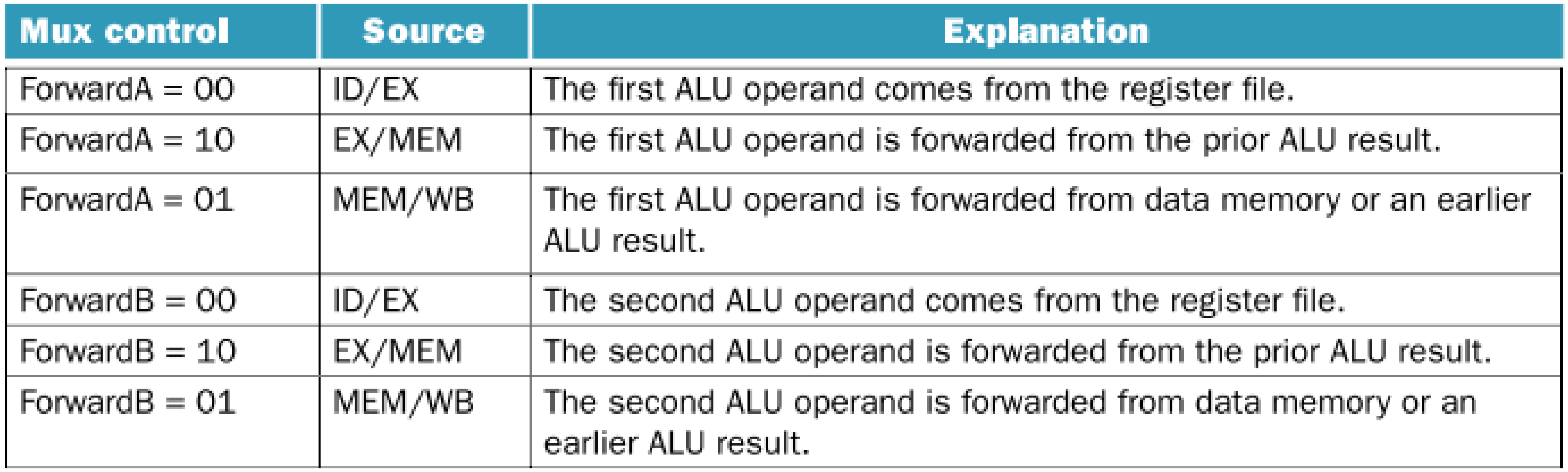
No forwarding



With forwarding

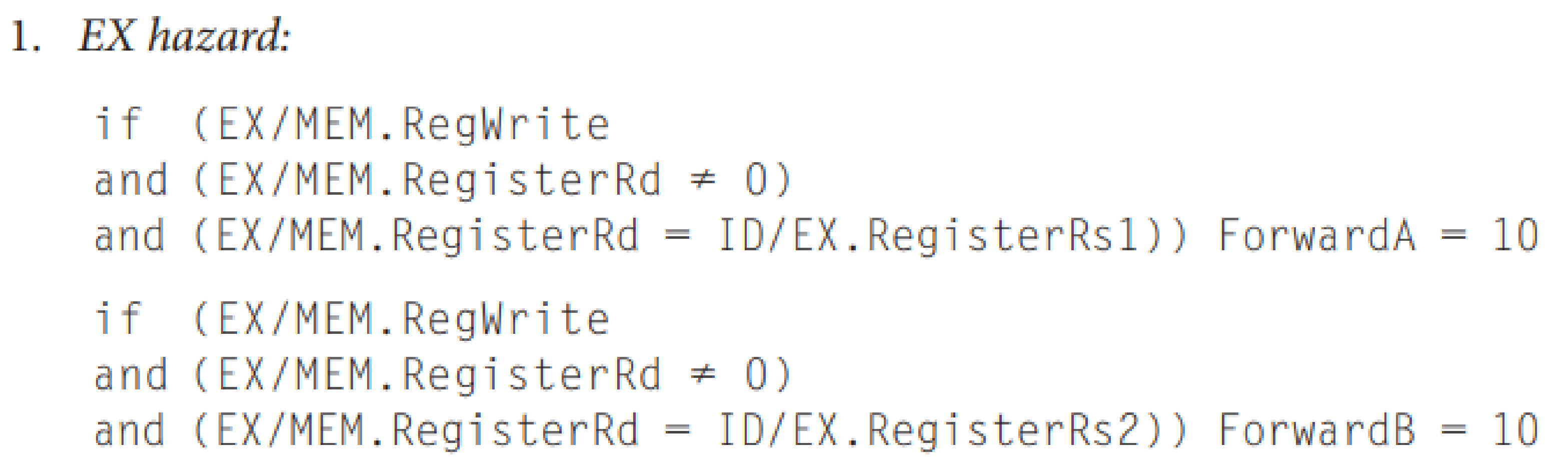


The control values for the forwarding multiplexors



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.

conditions for detecting hazards, and the control signals to resolve them:



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
```

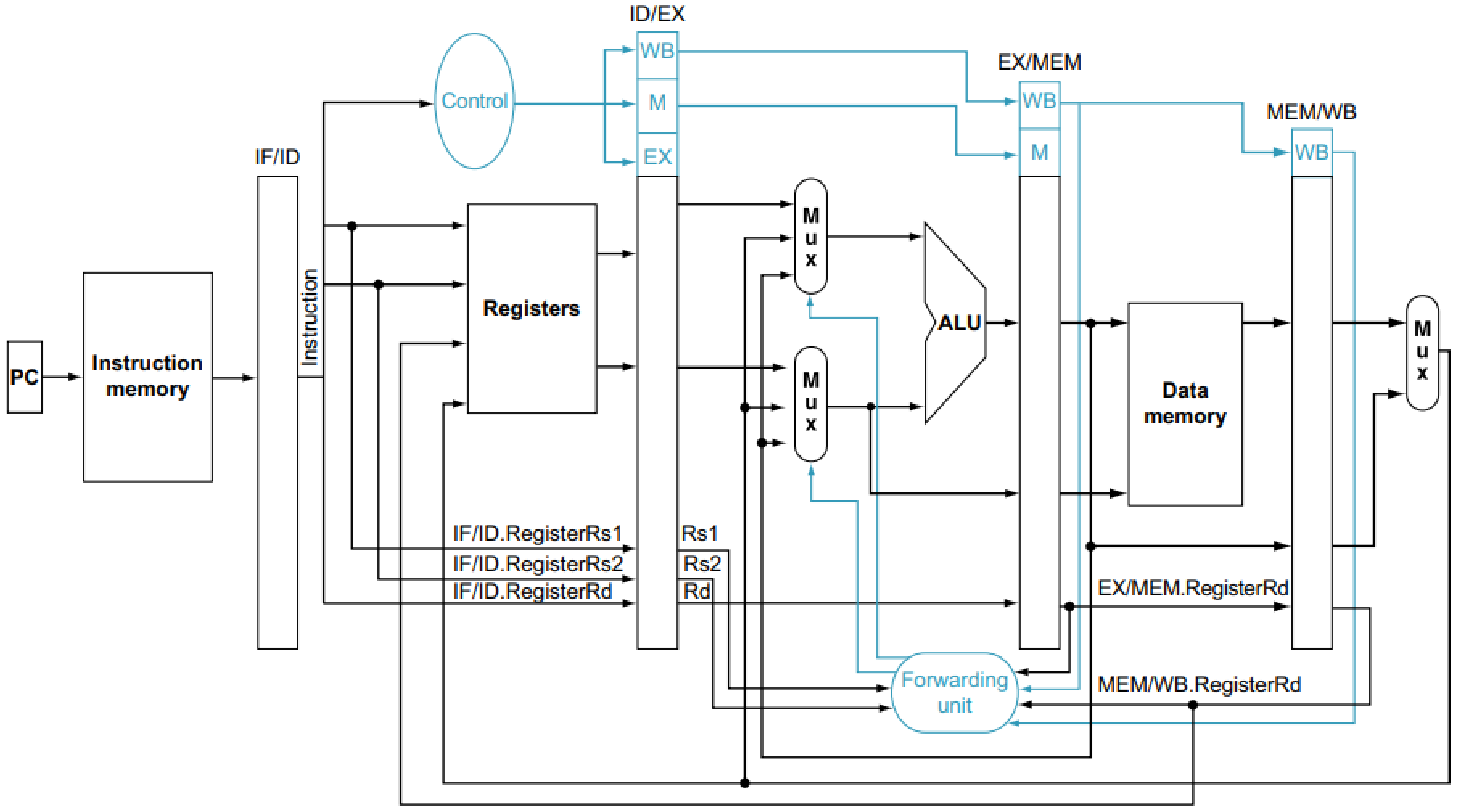
This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file, and the write register number matches the read register number of ALU inputs A or B, provided it is not register 0, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.

- MEM hazard:

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

```
add x1, x1, x2
add x1, x1, x3
add x1, x1, x4
. . .

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
```

The datapath modified to resolve hazards via forwarding

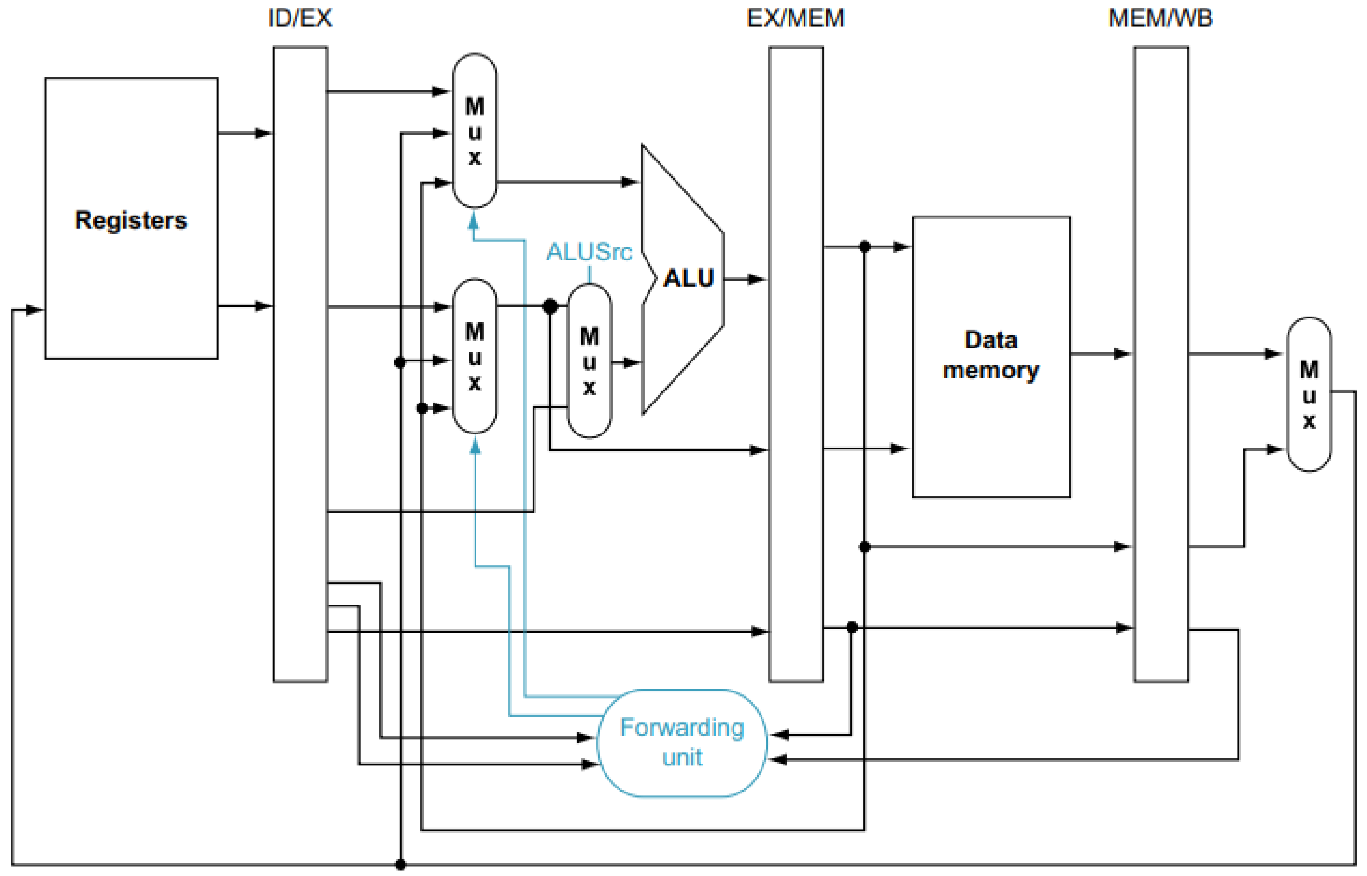


Data Hazards and Stalls

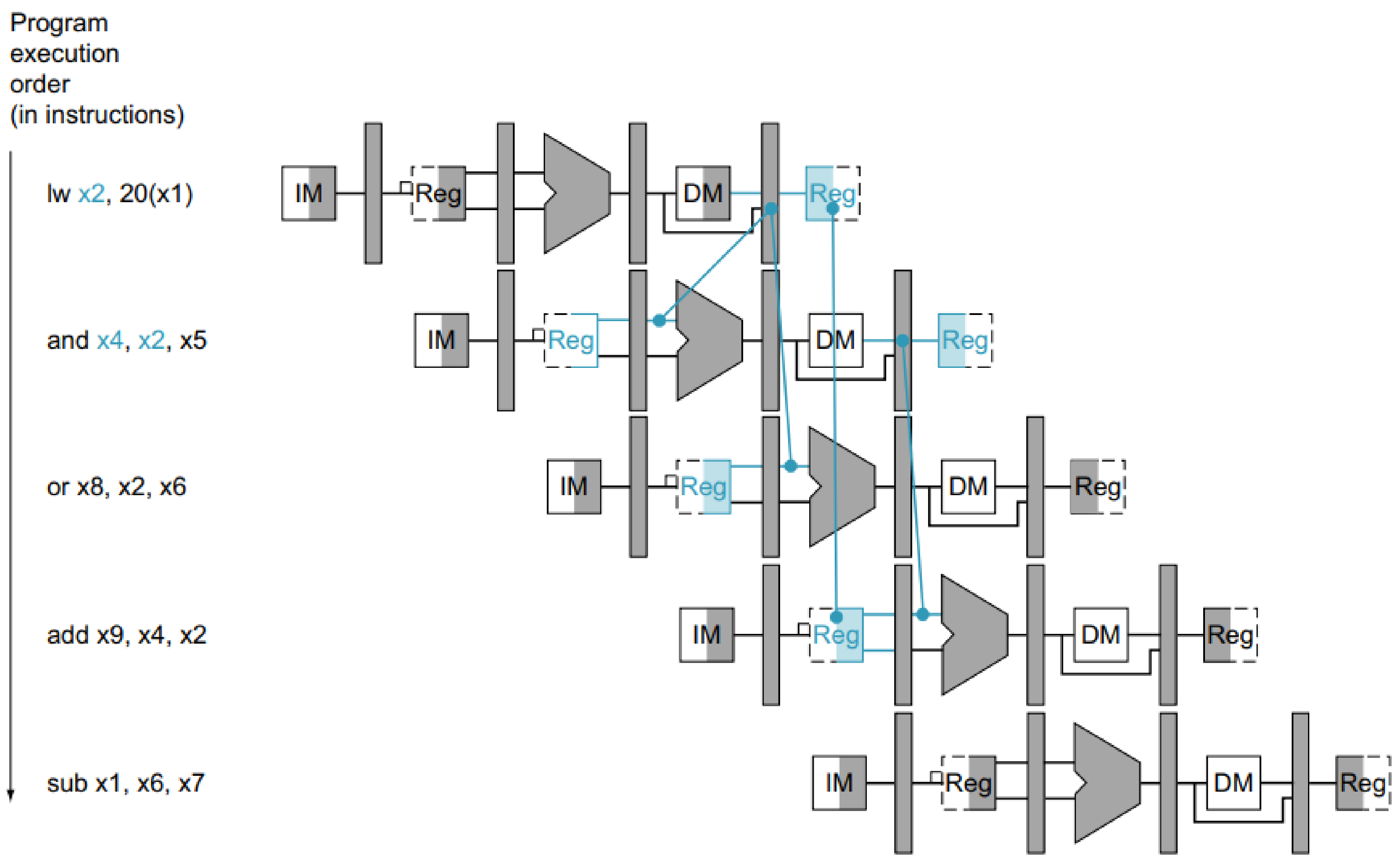
- in addition to a forwarding unit, we need a hazard detection unit
- It operates during the ID stage so that it can insert the stall between the load and the instruction dependent on it
- Checking for load instructions, the control for the hazard detection unit is this single condition:

```
if  (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
      stall the pipeline
```

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

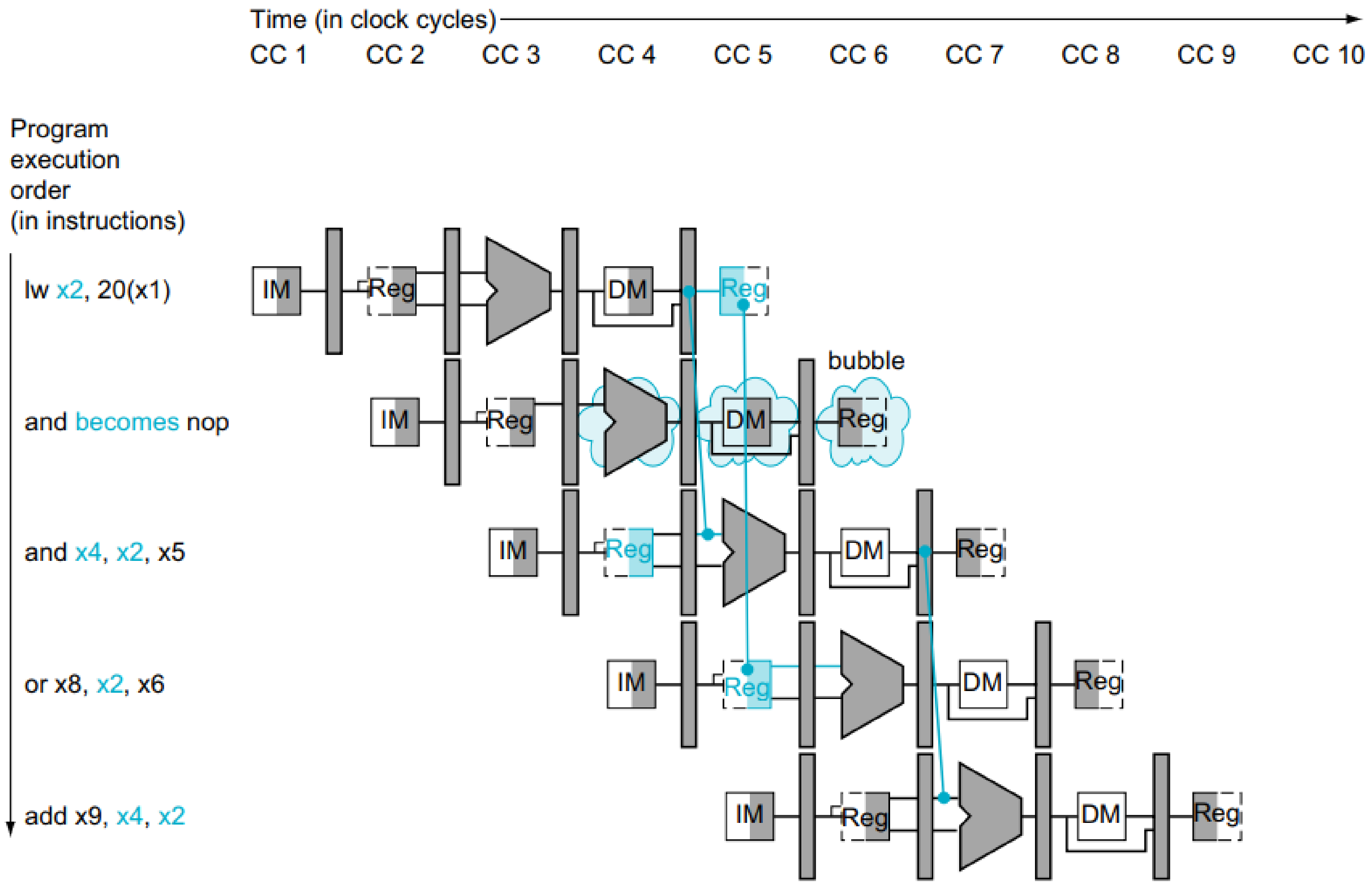


A pipelined sequence of instructions

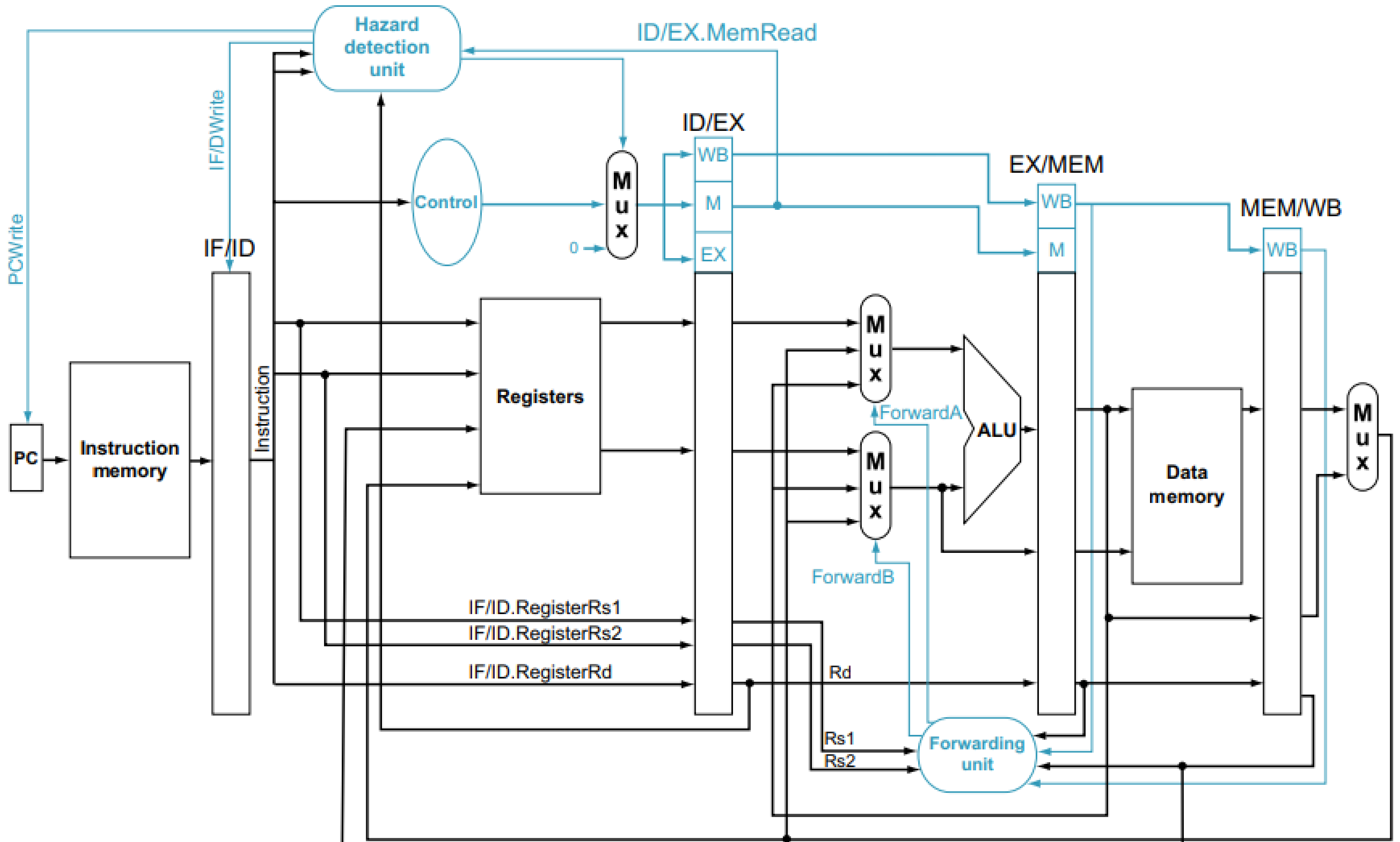


- nops An instruction that does no operation to change state.

The way stalls are really inserted into the pipeline



Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit


Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
0
M u x
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
Instruction
PC
Instruction memory
Registers
M u x
ForwardA
ALU
M u x
ForwardB
Data memory
M u x
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd
Rd
Rs1
Rs2
Forwarding unit