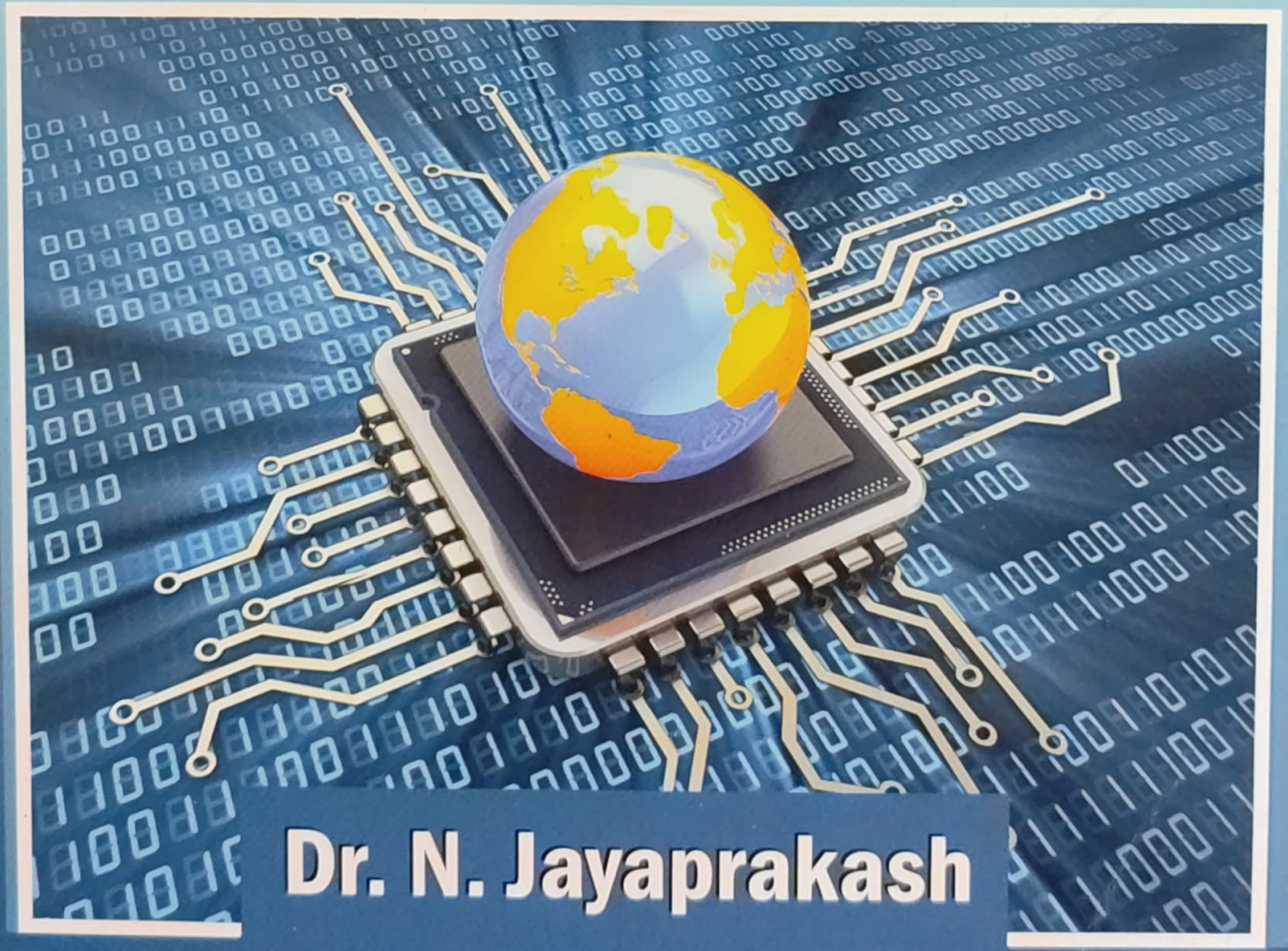
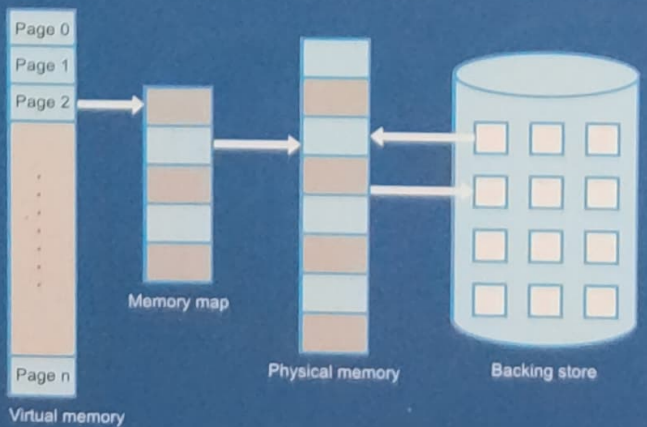
COMPUTER ARCHITECTURE



Dr. N. Jayaprakash



ANURADHA PUBLICATIONS
KARUPPUR
Since 1982

UNIT - 3 : PROCESSOR AND CONTROL UNIT

UNIT - 4 : PARALLELISM

3

PROCESSOR AND CONTROL UNIT

3.1 BASIC MIPS IMPLEMENTATION

Let us examine the implementation that includes a subset of the core MIPS instruction set.

- The memory-reference instructions **load word (lw)** and **store word (sw)**.
- The arithmetic-logical instructions **add, sub, AND, OR** and **Slt**.
- The instructions **branch equal (beq)** and **jump (j)**.

The subset mentioned here does not include all the integer instructions (for example, shift, multiply and divide are missing). This subset does not include any floating-point instructions also.

To implement every instruction, initially two steps must be followed. These two steps are identical irrespective of the exact class of instruction. They are

1. **Fetch the instruction from the memory:** Send the Program Counter (PC) to the memory that contains the code and fetch the instruction from the memory.
2. **Fetch the Operand/Operands:** For the load word instruction we have to read only one register. For most of the other instructions we have to read two registers.

After these two steps, the steps required to complete the instruction depend on the instruction class. For each of the above given three instruction classes (memory-reference, arithmetic-logical and branches), the actions are largely the same, independent of the exact instruction. MIPS instruction set is simple and regular. This instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

For example, except jump, all instruction classes use the Arithmetic-Logical Unit (ALU) after reading the registers.

- Memory-reference instructions use the ALU for an **address calculation.**
- Arithmetic-logical instructions use the ALU for the **operation execution.**
- Branch instructions use ALU for **comparison.**

After using the ALU, the actions required to complete various instruction classes are different from each other.

- A memory-reference instruction has to access the memory either to read data for a load or write data for a store.
- An arithmetic-logical instruction or load instruction has to write the data from the ALU or memory back into a register.
- Finally, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise, we have to increment the PC by 4 to get the address of the next instruction.

Figure 3.1 shows an abstract view of a MIPS implementation.

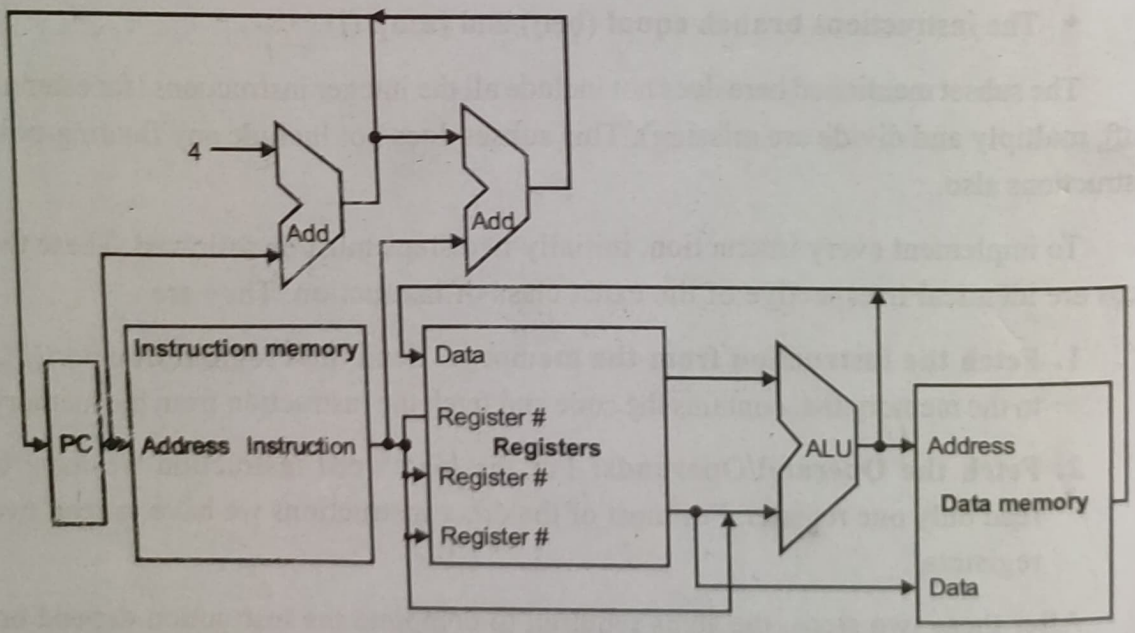


Figure 3.1 An abstract view of the implementation of the MIPS subset, which shows the major functional units and the major connections between them

- By using the program counter all instructions begin to supply the instruction address to the instruction memory.
- After the instruction is fetched the register operands needed by an instruction are specified by fields of that instruction.
- Register operands can be operated to compute a memory address.

- If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register.
- If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the register files.
- Branches are in need of the ALU output to determine the next instruction address. This instruction address comes either from the ALU (At ALU, the PC and branch offset are summed) or from an adder. The adder increments the current PC by 4.

Figure 3.1 shows that a data going to a particular unit is coming from two different sources. For example,

- The value written into the PC can come from one of two adders.
- The data written into the register file can come from a register or the immediate field of instruction.
- The selection of appropriate source is done by using a multiplexer (Data Selector).

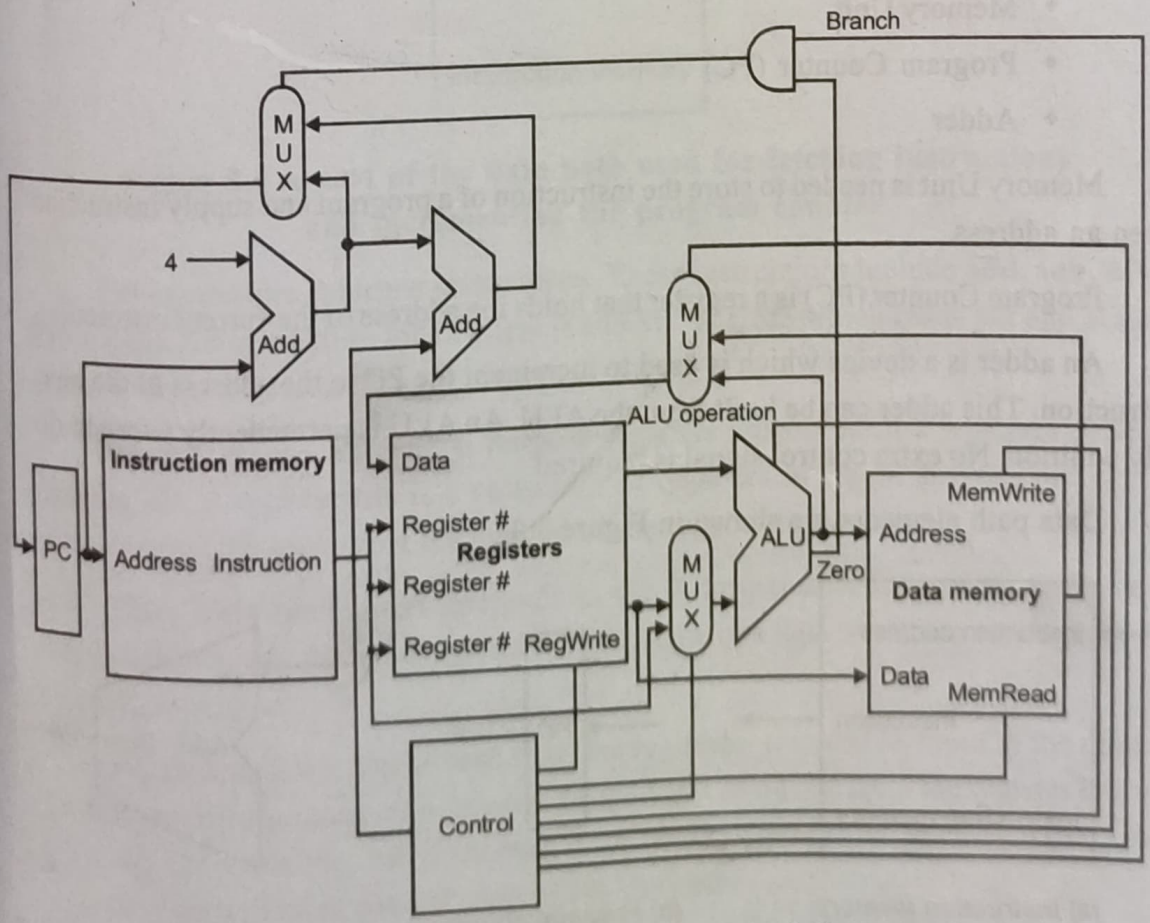


Figure 3.2 The necessry multiplexers and control lines are included in the basic implementation of the MIPS subset

Figure 3.2 shows the datapath of Figure 3.1 with three multiplexers and some control lines for the major functional units. **A control unit** which has the instruction as an inupt is used to determine how to arrange the control lines for the functional units and the two multiplexers.

- The top multiplexer (in the Figure 3.2) controls what value replaces the PC (PC + 4 or the branch destination address).
- The middle multiplexer (in the Figure 3.2) is used to steer the output of the ALU or the output of the data memory for writing into the register file.
- The bottom most multiplexer (in the Figure 3.2) is used to determine whether the second input of ALU is from registers or from the offset field of the instruction.

3.2 BUILDING A DATA PATH

The 'data path' is a representation of the flow of information (data, instructions).

Data path elements are:

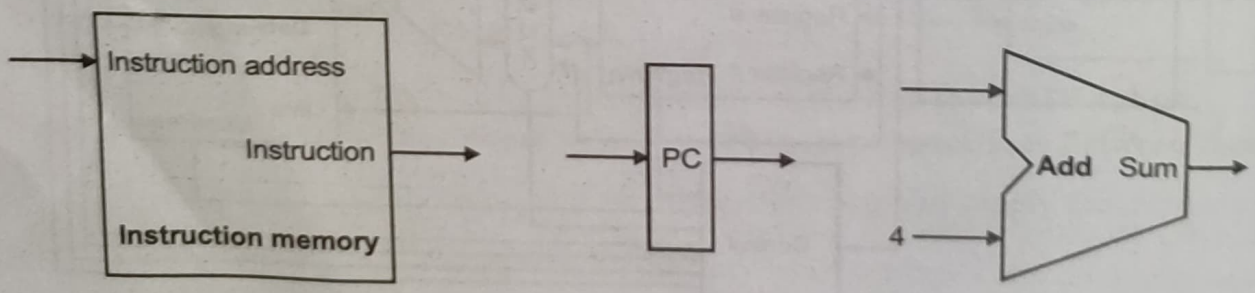
- Memory Unit
- Program Counter (PC)
- Adder

Memory Unit is needed to store the instruction of a program and supply instruction given an address.

Program Counter (PC) is a register that holds the address of the current instruction.

An adder is a device which is used to increment the PC to the address of the next instruction. This adder can be built from the ALU. An ALU is permanently wired to do only addition. No extra control signal is required.

Data path elements are shown in Figure 3.3.



(a) Instruction memory *(b) Program counter* *(c) Adder*

Figure 3.3 Two state elements are required to store and access instructions

To execute any instruction, the instruction must be fetched from memory unit. To prepare for executing the next instruction, we have to increment the program counter so that it points at the next instruction, 4 bytes later.

Figure 3.4 shows how to combine the three elements shown in Figure 3.3 to form a data path. This datapath can fetch instructions and increment the PC to obtain the address of the next sequential instruction.

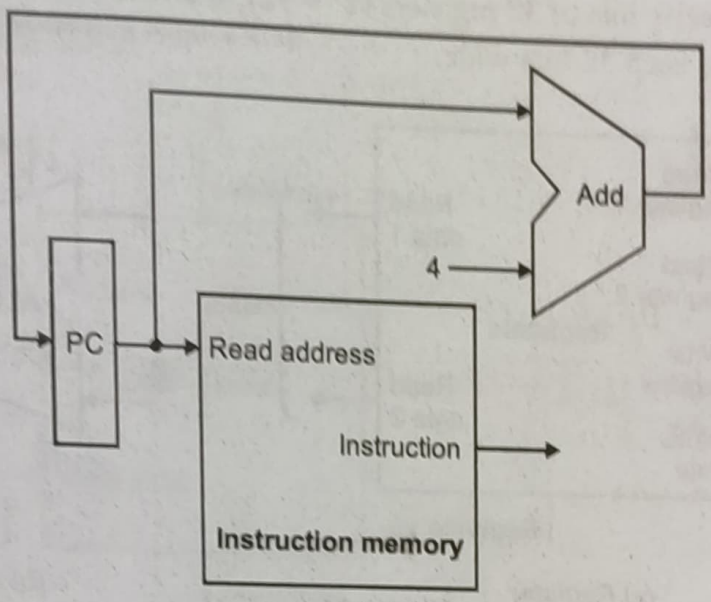


Figure 3.4 A part of the data path used for fetching instructions and incrementing the program counter

Let us consider R format instructions. These instructions include **add, sub, AND, OR** and **Slt**. An instruction for example is add $t1, $t2, $t3 which reads $t2 and $t3 and writes/saves the result in $t1.

The processor's 32-general purpose registers are stored in a structure called **a register file**. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file.

There are three register operands in the R-format instructions. So we have to read two data words from the register file and write one data word into the register file for each instruction.

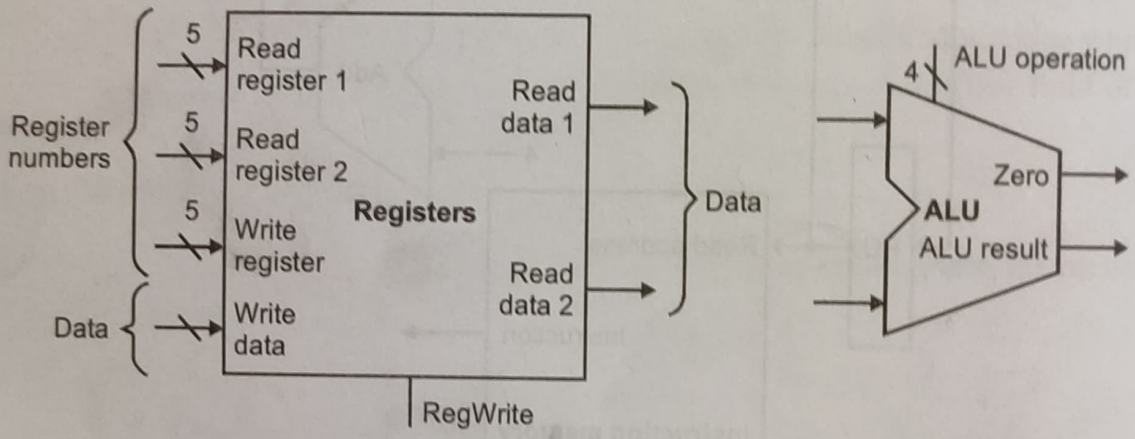
For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers. If we want to write a data word then we will be in need of two inputs.They are:

1. To specify the **register number** to be written

and

2. To supply the **data** to be written into the register.

Figure 3.5 (a) indicates that we need a total of four inputs (three for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers ($2^5 = 32$), whereas the data input and two data output buses are each 32 bits wide.



(a) Register *(b) ALU*

Figure 3.5 Register file and the ALU are the two elements needed to implement R-format ALU operations

Figure 3.5(b) shows the ALU, which takes two inputs (size of each input = 32 bits) and produces one output of 32 bits, as well as an 1-bit signal if the result is 0.

The operation to be performed by the ALU is controlled with the ALU operation signal. The ALU operation signal is 4-bits wide.

Now let us consider the MIPS load word and store word instructions which have the general form.

lw $t1, offset_value ($t2)

or

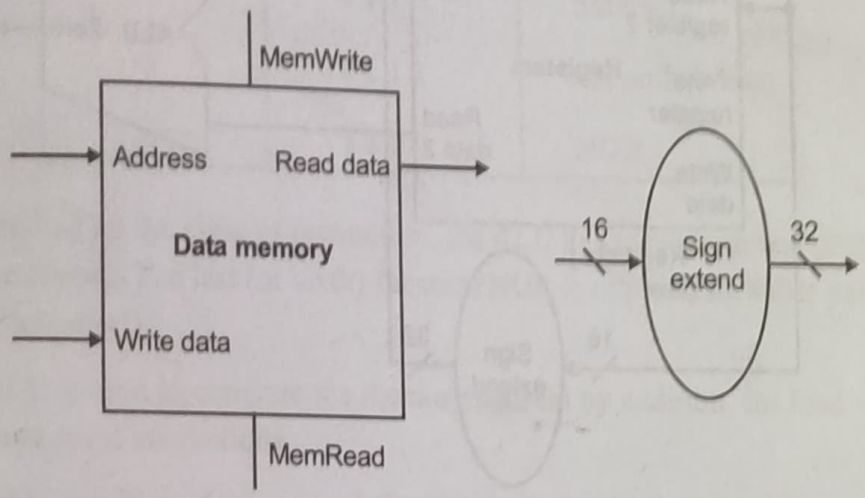
sw $t1, or offset_value ($t2)

The load word and store word instructions compute a memory address by adding the base register, $t2, to the 16-bit signed offset field contained in the instruction.

If the instruction is a store, the value to be stored must also be read from the register file where it resides in $t1.

If the instruction is aload, the value read from memory must be written into the register file in the specified register, $t1.

So we are in need of both the register file (Figure 3.5(a)) and the ALU (Figure 3.5(b)). In addition to these items, we require Data memory unit and sign extention unit (shown in Figure 3.6).



(a) Data memory unit *(b) Sign extension unit*

Figure 3.6 Data memory unit and the sign extension are the two units needed to implement loads and stores, in addition to the register file and the ALU, shown in Figure 3.5

The **beq** instruction is having

1. Three operands.
2. Two registers that are used to compare equality.
3. 16-bit offset used to compute the branch target address. This branch target address is relative to the branch instruction address.

The form of **beq** instruction is

beq $t1 $t2, offset

To implement this instruction, we have to compute the branch target address by adding the sign-extended offset field of the instruction to the PC.

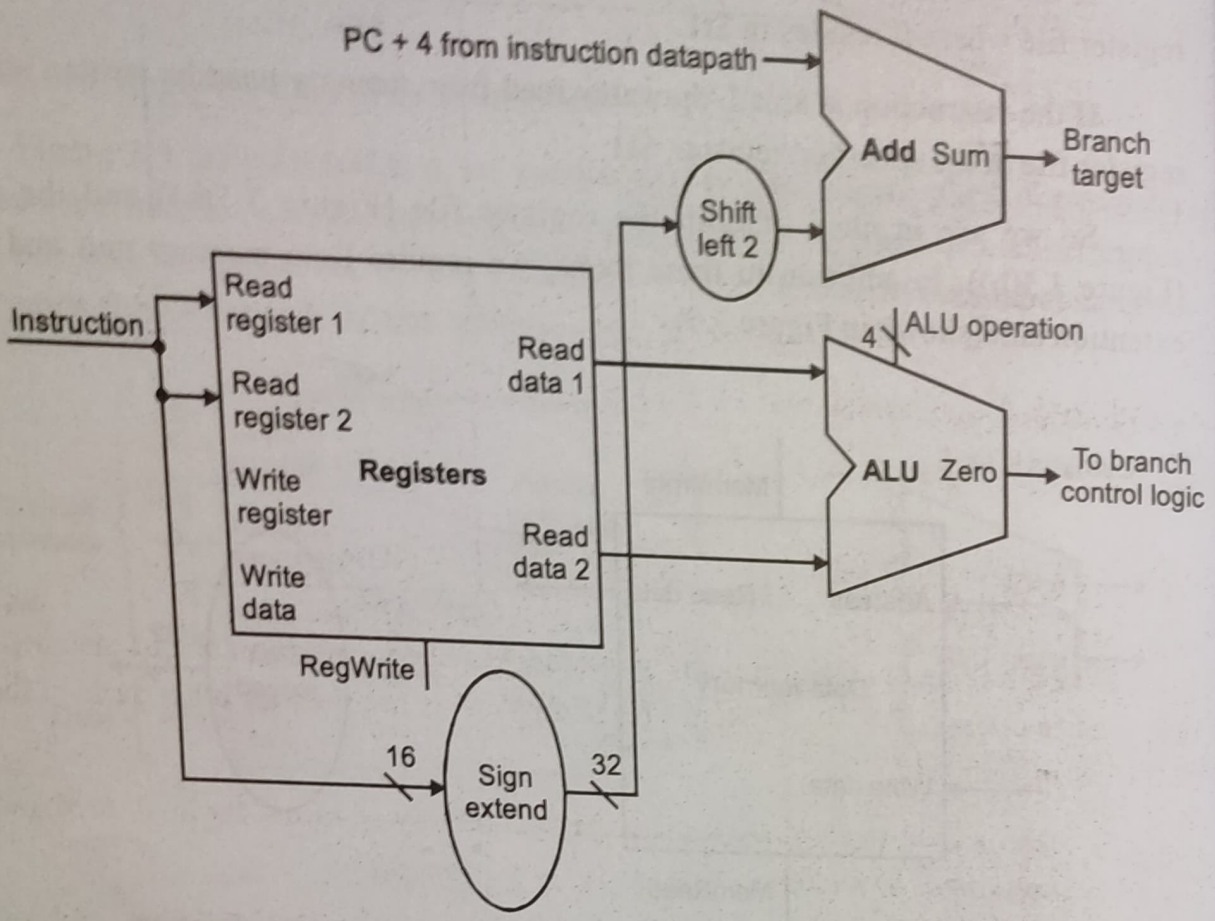


Figure 3.7 The structure of the datapath segment that handles branches

Figure 3.7 shows the structure of data path segment that handles branches. To compute the branch target address, the branch data path includes a sign extension unit from Figure 3.6 and an adder.

For comparing, we have to use the register file shown in Figure 3.5 a) to supply the two register operands.

3.3 CONTROL IMPLEMENTATION SCHEME

Control implementation scheme covers load word (lw), Store word (sw), branch equal (beq) and arithmetic logical instructions add, sub, AND, OR and set on less than. Let us design this implmentation scheme so that this scheme includes a jump instruction (j).

3.3.1 The ALU Control

The following six combinations of four control inputs are defined by the MIPS ALU.

Table 3.1 Combinations of four control inputs

Combination Sl. No.	ALU Control lines	Function
1	0 0 0 0	AND
2	0 0 0 1	OR
3	0 0 1 0	add
4	0 1 1 0	Subtract
5	0 1 1 1	Set on less than
6	1 1 0 0	NOR

Depending on the class of instruction, the ALU has to perform one of the above first five functions. (The last (or sixth) function NOR is required for other parts of the MIPS instruction set).

- ALU is used to compute the memory address by addition, for load word and store word instructions.
- ALU has to perform one of the five actions (AND, OR, Subtract, add, or Set on less than) for the R-type instructions.
- ALU has to perform a subtraction, for branch equal.

The operation of ALU can be controlled by the 4-bit ALU control input and 2-bit control field (2-bit control field is called as ALUOp).

The ALUOp is indicated by the two bits, as shown in the Table.

Table 3.2 ALUOp activity/operation

Sl.No.	Bits	Operation
1.	0 0	Add Loads and Stores
2.	0 1	Subtract for beq
3.	1 0	Determined by the operation encoded in the function field
4.	1 1	Nil

R-Format Instructions

- Used for arithmetic instructions,
- Format

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Fields
  - **op:** Op code (bits 26-31) that selects a specific operation
  - **rs:** First source register operand (bits 21-25)
  - **rt:** Second source register operand (bits 16-20)
  - **rd:** Destination register operand (bits 11-15)
  - **shamt:** Shift amount (bits 6-10)
  - **funct:** Function code - selects variant of operation specified in opcode (bits 0-5) – used to select an arithmetic instruction
- For example: add $4, $3, $2

	000000	00011	00010	00100	00000	100000
	op	rs	rt	rd	shamt	func
Sit positions	31:26	25:21	20:16	15:11	10:6	5:0
Number of bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

In Table we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code.

Table 3.3 shows the setting of ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code.

Table 3.3 Setting of ALU control inputs

Instruction opcode	ALUOp	Instruction operation	Function field	Desired ALU action	ALU Control Input
LW	00	load word	xxxxxx	add	0010
SW	00	store word	xxxxxx	add	0010
Branch equal	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	Set on less than (slt)	0111

In the above table (Table 3.3)

- The op code, listed in the first column determines the setting of the ALUOp.
- All the encodings are in the binary form.
- It is to be observed that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field. Now the value of the function code is not taken care of. Don't care condition is represented by xxxxxx in the function field column.
- When the ALUOp code is 10, then the function code is used to set the ALU control input.

The size of the main control can be reduced by using multiple levels of control.

Use of many smaller control units may increase the speed of the control unit.

The following Table (Table 3.4) shows the truth table for the 4-bit ALU control. The ALU control set is depending on the 2-bit ALUOp field and the 6-bit function field.

Table 3.4 The Truth table for the 4 ALU control (operation) bits

ALU Op		Function field						Operation
ALU Op1	ALU Op0	F5	F4	F3	F2	F1	F0	
0	0	x	x	x	x	x	x	0010
0	1	x	x	x	x	x	x	0110
1	0	x	x	0	0	0	0	0010
1	x	x	x	0	0	1	0	0110
1	0	x	x	0	1	0	0	0000
1	0	x	x	0	1	0	1	0001
1	x	x	x	1	0	1	0	0111

After the truth table has been constructed, it can be optimized and can be implemented using logic gates.

3.3.2 Designing the Main Control Unit

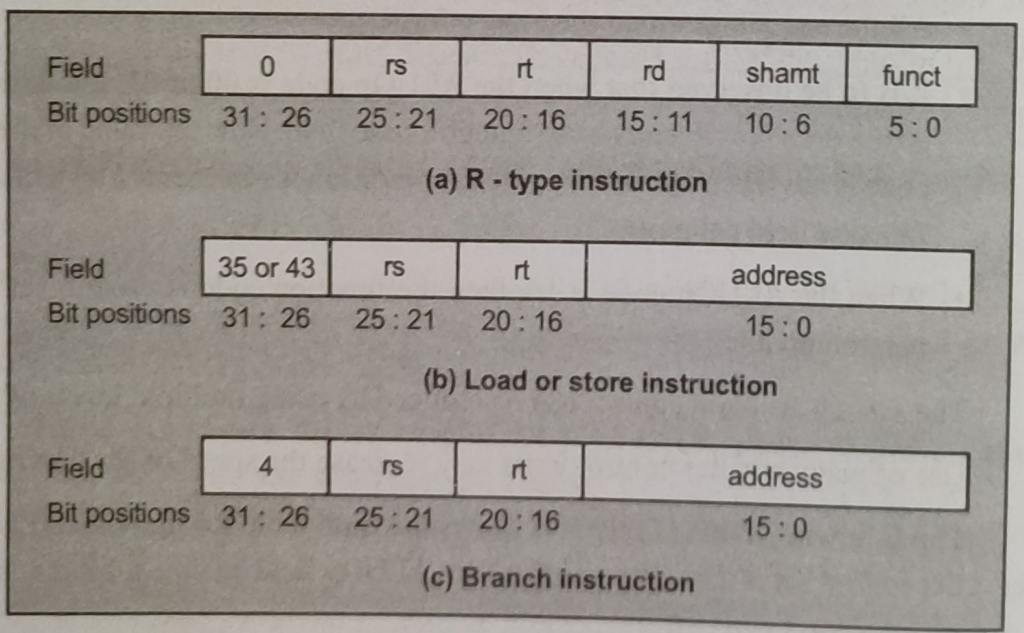


Figure 3.8 Three instruction classes

It is helpful to review the formats of the three instruction classes.

1. R-type instructions
2. Load-store instructions
3. Branch instructions

The three instruction classes are shown in Figure (Figure 3.8).

- R-format instructions have an opcode 0. These instructions have three register operands: rs, rt and rd. Fields rs and rt are sources and rd is the destination.

  R-type instructions, which we implement are add, sub, AND, OR and slt. The shamt field is used only for shifts.

- The shamt field is used only for Load instruction the format is Load (op code = 35_{10}). for store instruction the format is store (op code = 43_{10}).

  For loads, rt is the destination register for the loaded value; for stores, rt is the source register.

- For branch instruction the op code = 4; the registers rs and rt are the source registers that are compared for equality.

Some important observations about the instruction format

- Op code (op field) is always contained in bits 31:26. We will refer to this field as Op[5:0].
- rs and rt are specified by the bits 25:21 and 20:16 respectively.
- The base register for load and store instructions is always in bit positions 25:21 (rs).
- The 16-bits offset for branch equal, load and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction, it is in bit positions 15:11 (rd). So we need of a multiplexer to select which field of instruction is used.

Figure 3.9 shows seven single-bit control lines plus the 2-bit ALUOp control signal.

Table 3.5 shows the effect/function of each of the seven control signals.

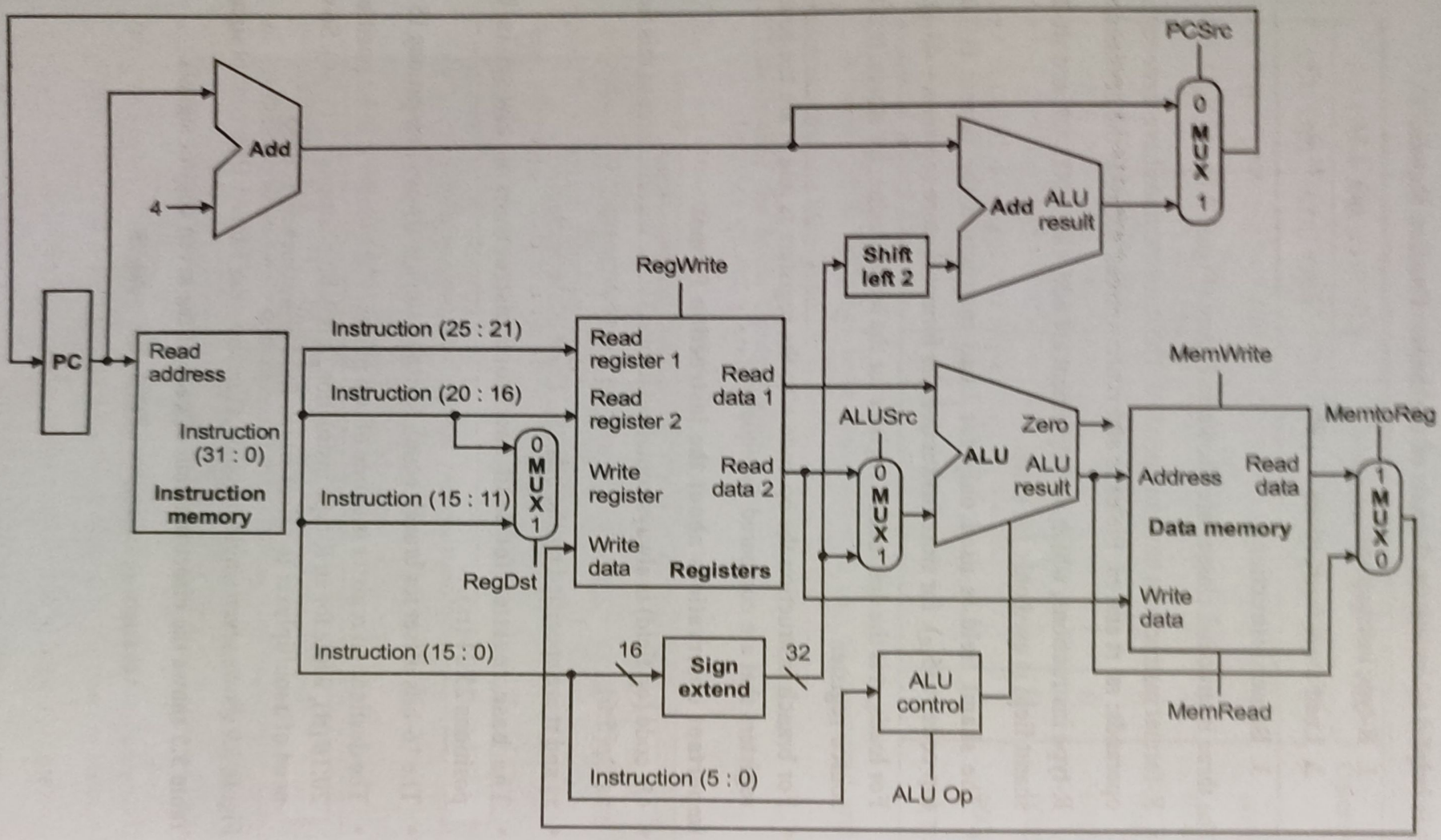


Figure 3.9 Seven Significant Control lines plus 2-bit ALUOp Control Signal (Data path with all necessary multiplexers and all control lines)

Table 3.5 The effect of each of the Seven Control Signals

Signal Name	Effect (when in-activated) *(Multiplexer selects the 0 input)*	Effect (When activated) *(Multiplexer selects the 1 input).*
RegDst	Destination register number for the write register comes from the rt field (bits 20:16).	Destination register number for the write register comes from rt field (bits 15:11).
RegWrite	None	The value on the write data input is written on the write register.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operands is the sign extended, lower 16 bits of the instruction.
PCSrc	PC is replaced by the output of the adder that computes the value of PC + 4.	PC is replaced by the output of the adder that computes the branch target.
MemRead	None	The address input designates the data memory contents. These data memory contents are put on the Read data output.
MemWrite	None	The address input designates the data memory contents. These data memory contents are replaced by the write data input.
MemtoReg	ALU gives a result/value. This result/value is fed to the register write data.	Data memory gives a result/value. This result/value is fed to the register write data.

- These 9 control signals (Seven from Table (Table 3.5) and 2 for ALUOp) can now be set on the basis of 6 input signals to the control unit which are the op code bits 31 to 26.
- Figure 3.10 shows the data path with the control unit and the control signals.
- The outputs of the control unit consist of three 1-bit signals. These three signals are used to control multiplexers. (Reg Write, MemRead and Mem Write)
- 1-bit signal is used in determining whether to possibly branch
- 2-bit signal is used for the ALU (ALUOp).

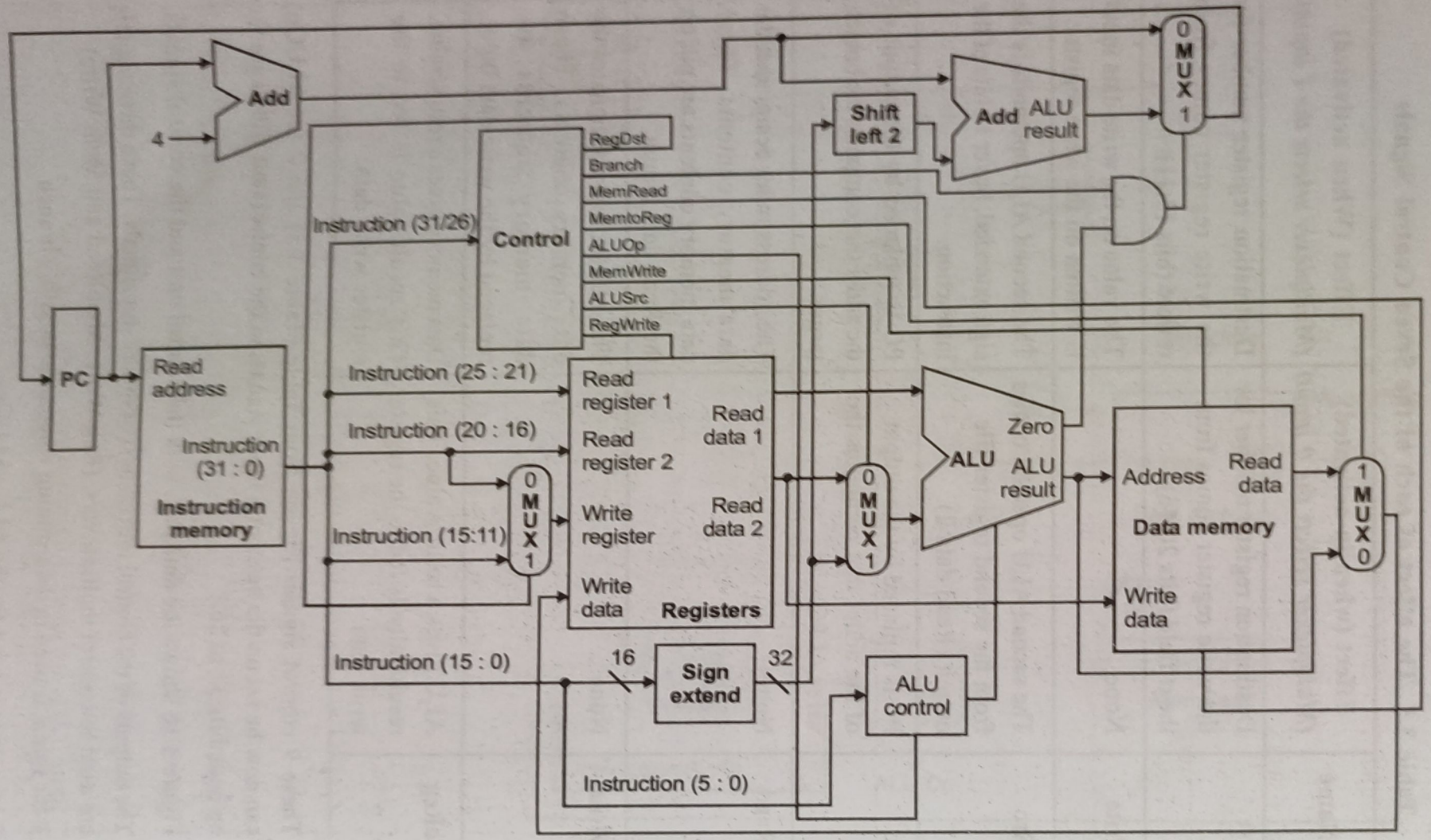


Figure 3.10 Simple datapath with the control unit and the control signals

- One AND gate is used to combine the branch control signal and the zero output from the ALU; The AND gate output controls the selection of the next PC.

Table 3.6 indicates whether each control signal should be 0, 1 or X (don't care) for each of the op code values.

Table 3.6 Setting the control lines using the opcode fields of the instruction

Instruction	Reg-Dst	ALU-Src	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALU Op1	ALU Op0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

- For all the R-format instructions (add, sub, AND, OR and Slt), the source register fields are rs and rt and the destination register field rd.
- R-type instruction writes a register (RegWrite/Reg W = 1), but neither reads nor writes data memory.
- When the Branch Control Signal is 0, the PC is replaced with PC + 4 unconditionally. If the zero output of the ALU is high, the PC is replaced by the branch target.
- For R-type, the ALUOp field instruction is set to 10. Setting to 10 indicates that the ALU control should be generated from the funct field.
- The second row of the above table (Table 3.6) gives the control signal for lw.
- The third row of the above table (Table 3.6) gives the control signal for sw.
- For calculating the address, the ALUSrc and ALUOp fields are used.
- MemRead and MemWrite are used to perform the memory accesss.
- RegDST and RegWrite are used to set for a load to cause the result to stored into the rt register.
- The branch instruction sends the rs and rt registers to the ALU.

  The ALUOp field for branch is set for a Subtract (ALU Control = 01), which is used to test for equality.
- It is to be noticed that the MemReg field is irrelevant when the RegWrite Signal is 0. Since the register is not being written, the value of data on the data Write part is not used. Thus the entries Mem to Reg in the case of sw instruction and beq instruction are replaced by X (Don't care). When RegWrite is 0, Don't cares can also be added to Reg DST.

3.3.3 Operation of the Data path

3.3.3.1 Data path for an R-type instruction

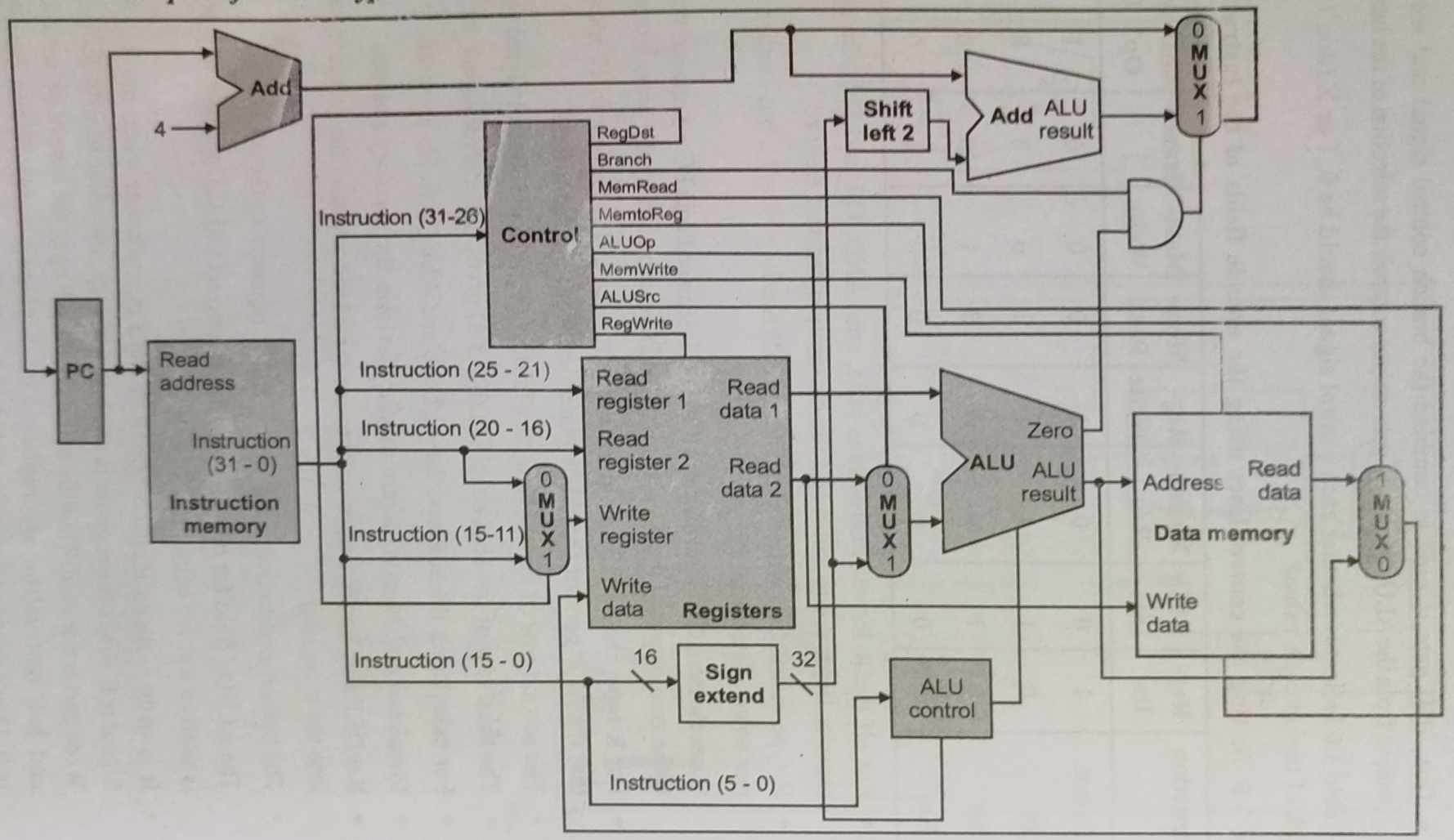


Figure 3.11 Datapath in operation for an R-type instruction

For Note: The control lines, data path units and connections that are active–are highlighted in Figure 3.11

Figure 3.11 shows the operation in the datapath for the R-type instructions (add, $t1, $t2, $t3). The execution of each instruction can be divided into four sequential steps. The steps are:

1. Fetch the instruction and then increment the PC.
2. Read the registers $t2 and $t3 from the register file; At the time of this step the setting of the control lines is computed by the main control unit.
3. Data is read from the register file. Using the function code (bits 5:0, the funct field of the instruction). The ALU operates on this data to generate the ALU function.
4. Using bits 15:11 of the instruction the result from ALU is written into the register file. The 15:11 bits of the instruction selects the destination register ($t1).

3.3.3 Data path for Load Instruction

For Note: The control lines, data path units and connections that are active–are highlighted in Figure 3.12

Figure 3.12 shows the Data path for a load instruction. The execution of the load instruction can be divided into five sequential steps. The steps are:

1. Fetch the instruction and then increment the PC.
2. A register ($t2) value is read from the register file.
3. The sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset) is computed by the ALU.
4. The sum, calculated by the ALU, is used as the address for the data memory.
5. The data, obtained/received from the memory unit, is written into the register-Bits 20:16 of the instruction ($t1) give the register destination.

Here the operations is much like R-format instruction. But the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address.

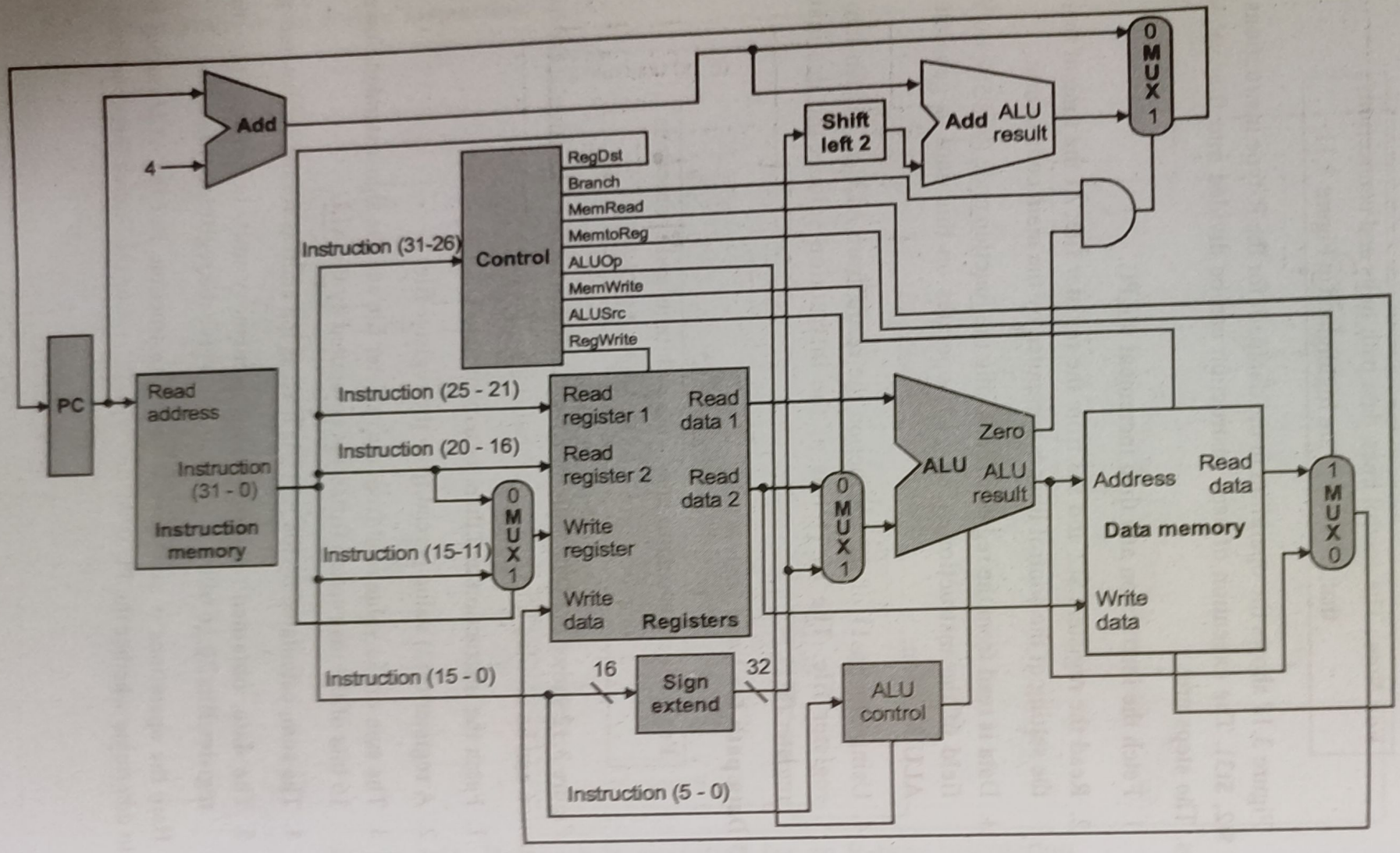


Figure 3.12 The data path in operation for a load instruction

3.3.4 Data path for Branch-on-Equal Instruction

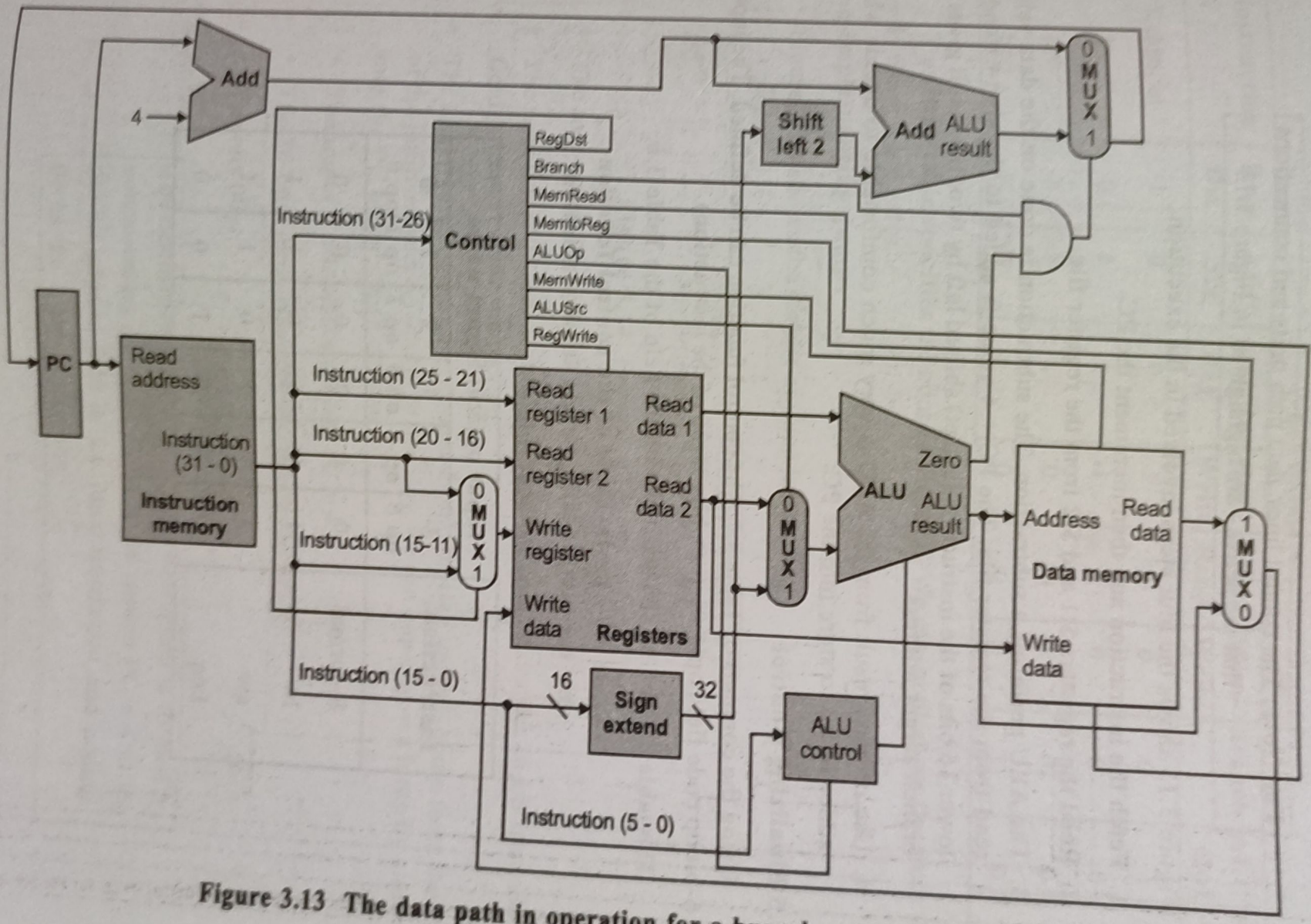


Figure 3.13 The data path in operation for a branch-on-equal instruction

For Note : The control lines, data path units and connections that are active – are highlighted in Figure 3.13

Figure 3.13 shows the four steps involved in the execution.

1. Fetch the instruction and then increment the PC.
2. Read the registers $t1 and $t2 from the register file.
3. The ALU performs a subtraction. The subtraction is done on the data value read from the register file. The PC + 4 value is added to the sign-extended lower 16 bits of the instruction (offset) shifted left by two; the result gives the branch target address.
4. Use of zero result from the ALU is very much comfortable to decide which adder result to store into the PC.

3.3.5 Finalizing Control

Using the contents of Table 3.7. The control function can be defined. The input is the 6-bit op code field op[5:0], (i.e., bits 31:26 of the instruction).

The table 3.7 can be constructed with the help of the Table 3.6.

Table 3.7 Truth Table of the control function for the simple sinlge-cycle implementation

a) Input

Instruction	Input op[5:0]/Signal name					
	5 op 5	4 op 4	3 op 3	2 op 2	1 op 1	0 op 0
R-format	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

b) Output

Instruction	Reg-Dst	ALU-Src	Mem to-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALU Op1	ALU Op0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

This truth table specifies the control function. We can implement the truth table directly using logic gates.

3.3.6 Implementing jumps

Figure 3.14 shows the format of jump instruction.

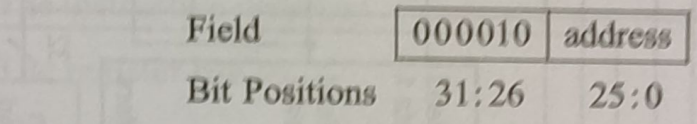


Figure 3.14 Format of jump instruction

- The low-order 2 bits of a jump are always 00_2.
- The next lower 26 bits of this 32 bit address come from the 26-bit immediate field (address) in the instruction.
- The upper 4 bits of the address, that should replace the PC, come from the PC of the jump instruction plus 4. In this way, we can implement a jump by storing into the PC of the concatenation of
  - The upper 4 bits of the current PC + 4.
  - The 26-bits (jump instruction's immediate field).
  - The bits 00_2.

> The destination address for a jump instruction is formed by concatenating the upper 4 bits of the current PC + 4 to the 26-bit address field in the jump instruction and adding 00 as the 2 low-order bits.

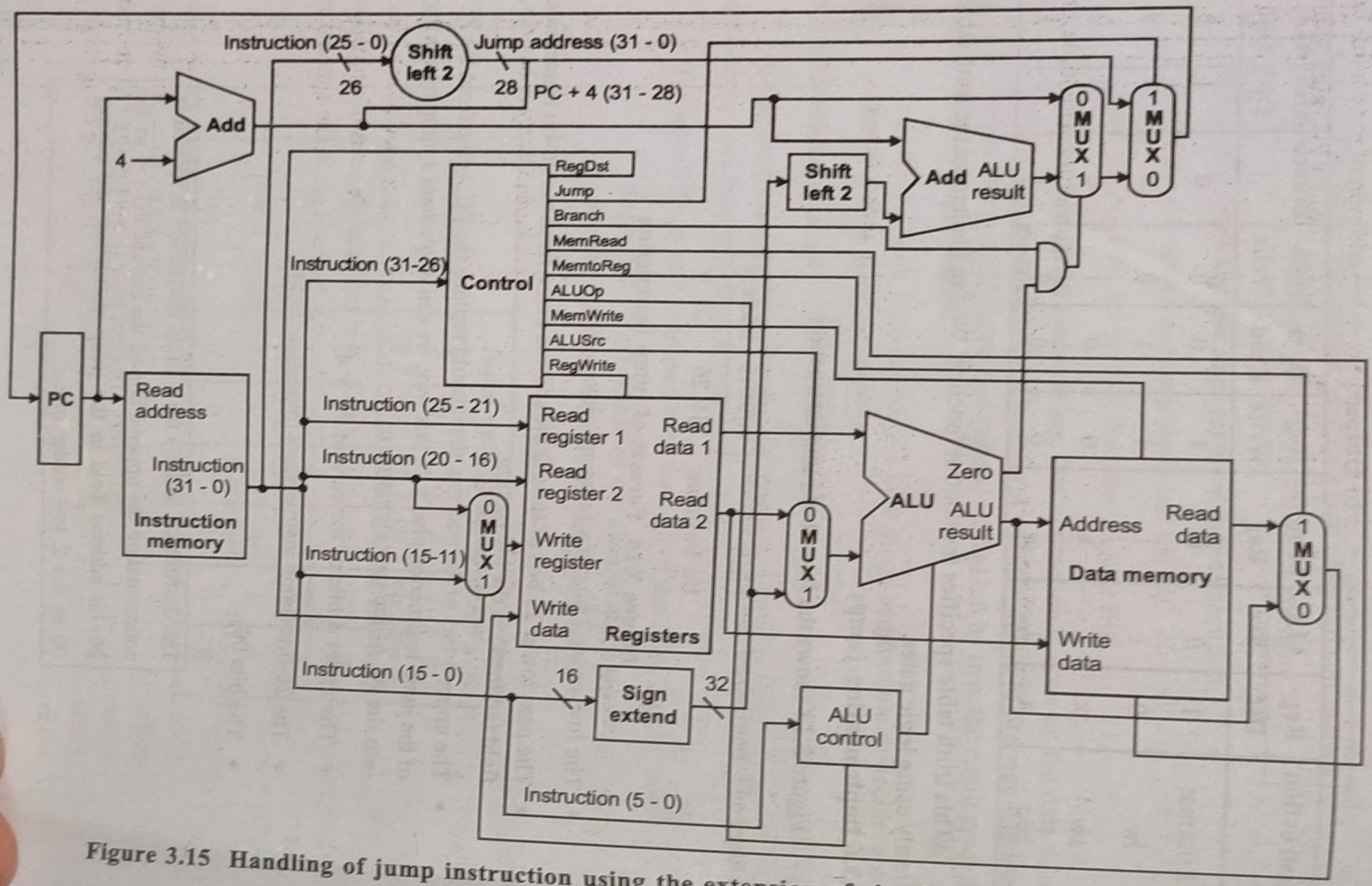


Figure 3.15 Handling of jump instruction using the extension of simple control and data path

The Figure 3.15 is the same as Figure 3.10 excep the addition of the control for jump. An additional multiplexer is used to select the source for the new PC value which is

Either the incremented PC (PC + 4), the branch target PC

Or

The jump target PC

One additional control signal, called jump is needed for the additional multiplexer. If the op code is 2, then only the control signal, jumps is asserted.

3.3.7 Reasons for not using a Single-cycle implementation

- Single-cycle design is inefficient.
- The clock cycle is determined by the longest path in the processor.

  This path uses five functional units in series.

  1. The instruction memory
  2. The register file
  3. The ALU
  4. The Data memory
  5. The Register file

- The overall performance of a Single-cycle implementation is likely to be poor, since the clock cycle is too long.

3.4 PIPELINING

In computing, a **pipeline** is a set of data processing elements connected in series, where the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements.

Computer-related pipelines include

- Instruction pipelines, such as the classic RISC pipeline, which are used in central processing units (CPUs) to allow overlapping execution of multiple instructions with the same circuitry. The circuitry is usually divided up into stages, including instruction decoding, arithmetic, and register fetching stages, wherein each stage processes one instruction at a time.

- Graphics pipelines, found in most graphics processing units (GPUs), which consist of multiple arithmetic units, or complete CPUs, that implement the various stages of common rendering operations (perspective projection, window clipping, color and light calculation, rendering, etc.).
- Software pipelines, where commands can be written where the output of one operation is automatically fed to the next following operation. The Unix system call pipe is a classic example of this concept, although other operating systems do support pipes as well.

Classification of Pipelining

- Arithemetic Pipelining
- Instruction Pipelining
- Vector Pipelining
- Unifunction and Multifunction Pipelining
- Scalar and Vector Pipelining

Here let us see about the instruction pipelining.

Instruction Pipelining

Instruction pipelining is a technique that implements a form of parallelism called instruction-level parallelism within a single processor. It therefore allows faster CPU throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate.

The basic instruction cycle is broken up into a series called a pipeline. Rather than processing each instruction sequentially (finishing one instruction before starting the next), each instruction is split up into a sequence of steps so different steps can be executed in parallel and instructions can be processed concurrently (starting one instruction before finishing the previous one).

Pipelining increases instruction throughput by performing muliple operations at the same time, but does not reduce instruction latency, which is the time to complete a single instruction from start to finish, as it still must go through all steps. Indeed, it may increase latency due to additional ovehead from breaking the computation into separate steps and worse, the pipeline may stall (or even need to be flushed), further increasing the latency.

Thus, pipelining increases throughput at the cost of latency, and is frequently used in CPUs but avoided in real-time systems, in which latency is a hard constraint.

Each instruction is split into a sequence of dependent steps. The first step is always to fetch the instruction from memory; the final step is usually writing the results of the instruction to processor registers or to memory.

Pipelining seeks to let the processor work on as many instructions as there are dependent steps, just as an assembly line builds many vehicles at once, rather than waiting until one vehicle has passed through the line before admitting the next one.

Just as the goal of the assembly line is to keep each assembler productive at all times, pipelining seeks to keep every portion of the processor busy with some instruction. Pipelining lets the computer's cycle time be the time of the slowest step, and ideally lets one instruction complete in every cycle.

The term pipeline is an analogy to the fact that there is fluid in each link of a pipeline, as each part of the processor is occupied with work.

Number of steps

The number of dependent steps varies with the machine architure. For example:

- The IBM Stretch project proposed the terms Fetch, Decode, and Execute that have become common.
- The classic RISC - (Reduced Instruction Set Computer) pipeline comprises:
  1. Instruction Fetch (IF)
  2. Instruction decode and register fetch (ID)
  3. Execute (EX)
  4. Memory access (MEM)
  5. Register write back (WB)

Table 3.8 Pipeline stages

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

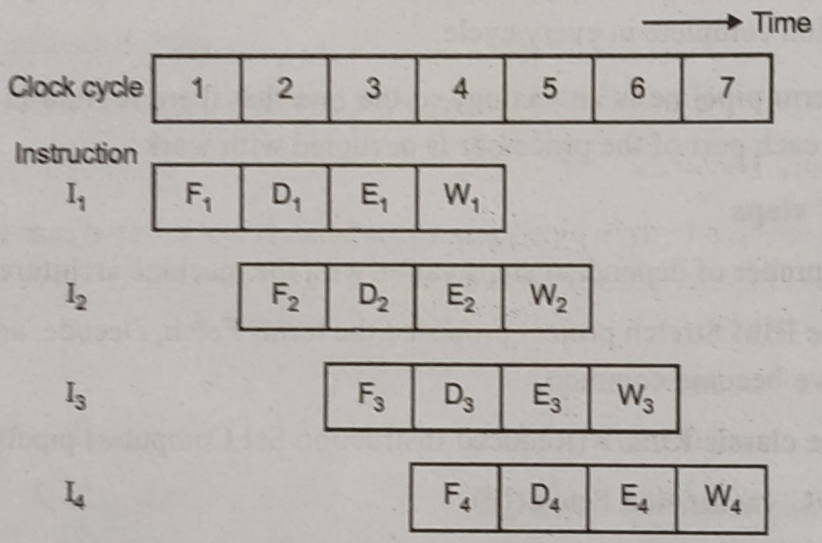
The processing of an instruction need not be divided into only two steps. A pipelined processor may process each instruction in four steps or more than that. For example let us consider a pipelined process which processes each instruction in four steps only. They are

Step 1: F- Fetch : Read the instruction from the memory

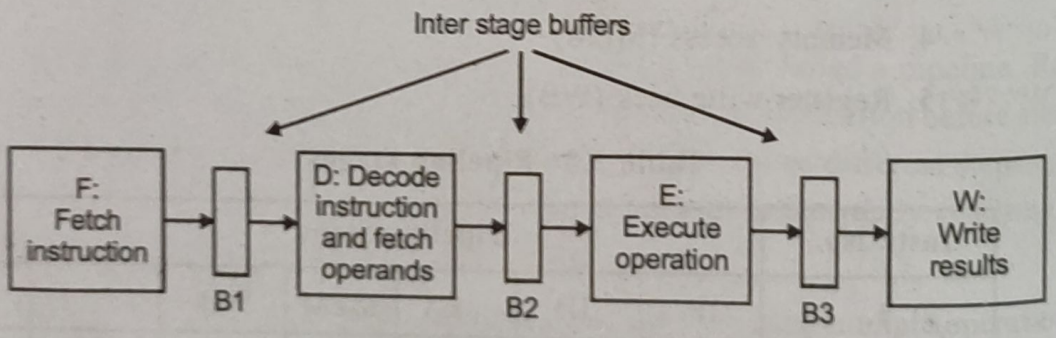
Step 2: D- Decode : Decode the instruction and fetch the operand(s) if needed.

Step 3: E- Execute : Perform the operation specified by the instruction.

Step 4: W-Write : Store the result in the destination place



(a) Instruction execution divided into four steps



(b) Hardware organization

Figure 3.16 A 4-stage pipeline

Figure 3.16(a) shows the sequence of events. Figure 3.16(b) shows the requirements of the four distinct hardware units.

These units must have the ability to perform their tasks simultaneously.

These units should not interfere with each other while performing their tasks.

3.4.1 Designing instruction sets for Pipelining

1. All MIPS instructions are of same length.

   Due to this restriction, it is more convenient to fetch instructions in the first pipeline stage, and decode them in the second pipeline stage.

2. There are a few instruction formats in MIPS. In this format, the source register fields are located in the same place in each instruction.

3. Memory operands appear only in loads/stores in MIPS. Because of their restriction, for calculating the memory address, we can use the execute stage. Then memory can be accessed.

4. Operands must be aligned in memory. In a single pipeline stage, the requested data can be transferred between processor and memory.

3.4.2 Pipeline Hazards

There are situations in pipelining when the next instructuion cannot execute in the following clock cycle. These events are called **hazards**. There are three different types of hazards. They are:

1. Data Hazards
2. Structural Hazards
3. Control Hazards

1. Data Hazards

In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline.

2. Structural Hazards

A structural hazards occurs when a part of the processor's hardware is needed by two or more instructions at the same time.

Structural hazards occur when different instructions collide while trying to access the same piece of hardware in the same segment of pipeline.

3. Control Hazards (Branch Hazards)

Control hazards (Branch hazards) occur with branches. On many instruction pipeline micro architectures, the processor will not know the outcome of the branch.

When it needs to insert a new instruction into the pipeline (Normally the fetch stage), control hazard arises from the need to make a decision based on the results of one instruction while others are executing.

3.4.3 Pipeline performance

A pipeline's performance can be measured with the help of its throughput in terms of Millions of Instructions Per Second (MIPS).

To calculate the performance we need the number of Clockcycle Per Instruction (CPI) also.

MIPS and CIP are related by the following equation.

$$\boxed{CPI = \frac{f}{MIPS}}$$

Where f is the pipeline's clock frequency in MHz.

The speedup is used to determine the pipeline performance.

$$\text{Speedup, } S_k = \frac{T_{1,n}}{T_{k,n}}$$

here $T_{k,n}$ is the total time required for a pipeline with k stage to execute n instructions. $T_{1,n}$ is the total time required for the same program on a similar, nonpipelined processor.

The cycle time τ of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline. Each column in Figure 3.17 represents one cycle time.

The cycle time τ (τ is the processing time in each stage and is also known as stage delay) can be determined using the formula:

$$\tau = \max_i [\tau_i] + d \qquad 1 \le i \le k$$

$$= \tau_m + d \qquad \left(Here\ \tau_m = \max_i [\tau_i]\right)$$

Where,

τ_i = Time delay of the circuitry in the i^{th} stage of the pipeline

τ_m = Maximum stage delay

k = Number of stages in the instruction pipeline

d = Time delay of a latch, needed to advance signals and data from one stage to the next

In general, the time delay d is equivalent to a clock pulse and $\tau_m >> d$. Therefore, pipeline's clock frequency is the inverse of clock period/cycle time.

$$f = \frac{1}{\tau}$$

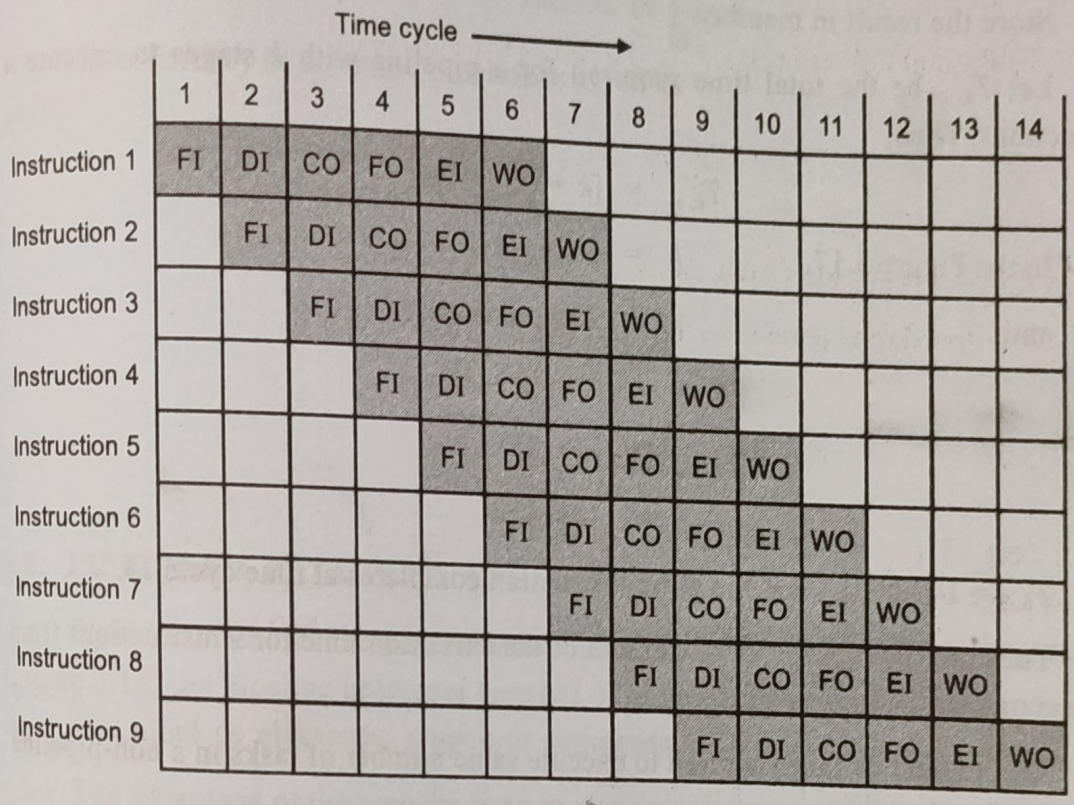


Figure 3.17 Timing Diagram for Instruction Pipeline Operation

The Figure 3.17 indicates the Timing Diagram for Instruction Pipeline Operation. In this operation there are 6 stages. They are

Stage 1 – Fetch Instruction (FI)

Read the next expected instruction into a buffer.

Stage 2 – Decode Instruction (DI)

Determine the opcode and the operand specifiers.

Stage 3 – Calculate Operands (CO)

Calculate the effective address of each source operand.

Stage 4 – Fetch Operands (FO)

Fetch each operand from memory.

Stage 5 – Execute Instruction (EI)

Perform the indicated operation and store the result, if any, in the specified destination operand location.

Stage 6 – Write Operand (WO)

Store the result in memory.

Let $T_{k,n}$ be the total time required for a pipeline with k stages to execute n instructions. Then,

$$T_{k,n} = [k + (n-1)]\tau$$

In the Figure 3.17, $k = 6$ stages

and $n = 9$ instructions.

$$\therefore \quad T_{6,9} = [6 + (9-1)]\tau$$

$$= [6 + 8]\tau$$

$$= 14\tau$$

$T_{6,9} = 14\tau$ means that the ninth instruction completes at time cycle 14.

Thus the six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

The amount of time required to execute same number of tasks in a non-pipeline processor can be given as

$$T_{1,n} = nk\tau$$

Speedup factor

The speedup factor of a k-stage pipeline over an equivalent non-pipelined processor is given as

$$S_k = \frac{T_{1,n}}{T_{k,n}}$$

$$= \frac{nk\tau}{[k + (n-1)]\tau}$$

$$\boxed{S_k = \frac{nk}{k + (n-1)}}$$

Efficiency

The efficiency of a linear pipeline is defined as a ratio of speedup factor and the number of stages in the pipeline.

$$\text{i.e., Efficiency} = \frac{\text{Speedup factor}}{\text{Number of stages}}$$

The efficiency of a k-stage pipeline is given as

$$E_k = \frac{S_k}{k}$$

$$= \frac{nk}{\left[k + (n-1)\right]} \div k$$

$$\therefore \quad \boxed{E_k = \frac{n}{k + (n-1)}}$$

3.5 PIPELINED DATA PATH AND CONTROL

Figure 3.18 shows the general structure of multi stage pipeline. In this Figure (Figure 3.18) the pipeline processor consists of m processing units. These processing units are called as **elements, stages** or **segments**.

The advantage of the pipeline is an m-stage pipeline can simultaneously process up to m independent sets of data operands.

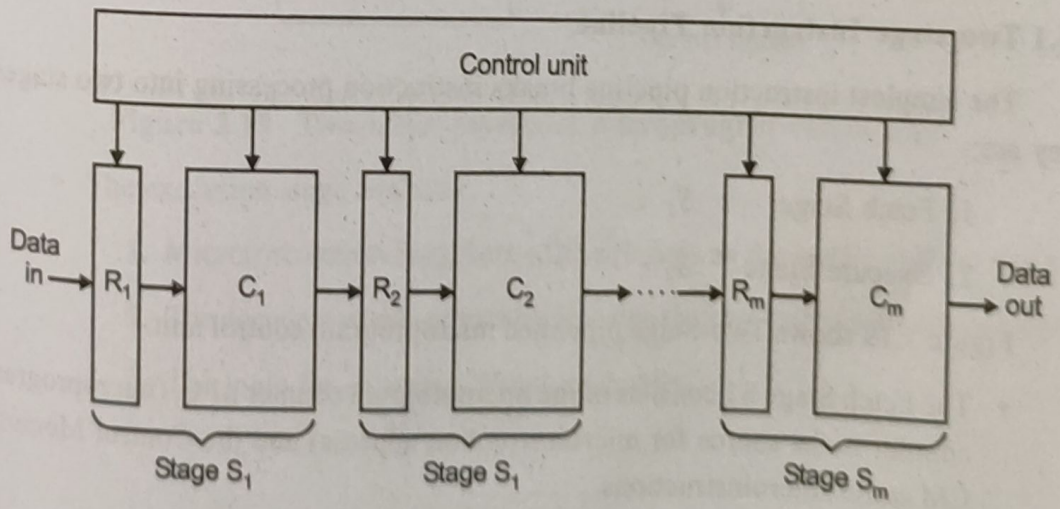


Figure 3.18 Structure of a pipeline processor

Each stage is having two main blocks. They are:

1. Multiword Input Register
2. Data path circuit

The given data sets move through the pipeline stage by stage. When the pipeline is full, m individual/separate operations are being executed simultaneously, each in a different stage.

When the partially processed results move they will be held by multiword input registers R_i. These registers serve as buffers also. These registers prevent neighbouring stages from interfering with each other.

In every individual clock period the individual stage processor processes its data and transfers the corresponding result to the next stage. So we will get a new final result from the pipeline for every clock cycle.

If T is the time in seconds required to perform a single suboperation and m stages are there, then the time required to complete a single operation is mT seconds. This is called the **delay** or **latency of the pipeline**.

The throughput of the pipeline is the maximum number of operations completed per second. This is called the **throughput of the pipeline**.

It the pipeline's clock period is T seconds and m is the number of stages then the delay or latency of the pipeline is mT seconds.

Throughput of the pipeline is defined as the maximum number of operations completed per second and it is given by $1/T$.

3.5.1 Two stage Instruction Pipeline

The simplest instruction pipeline breaks instruction processing into two stages. They are:

1. Fetch Stage S_1
2. Execute Stage S_2

Figure 3.19 shows Two-Stage pipelined microprogram control unit.

- The Fetch Stage S1 consists of the microprogram counter μPC, (microprogram counter is ths source for microinstruction address) and the Control Memory CM stores microinstructions.

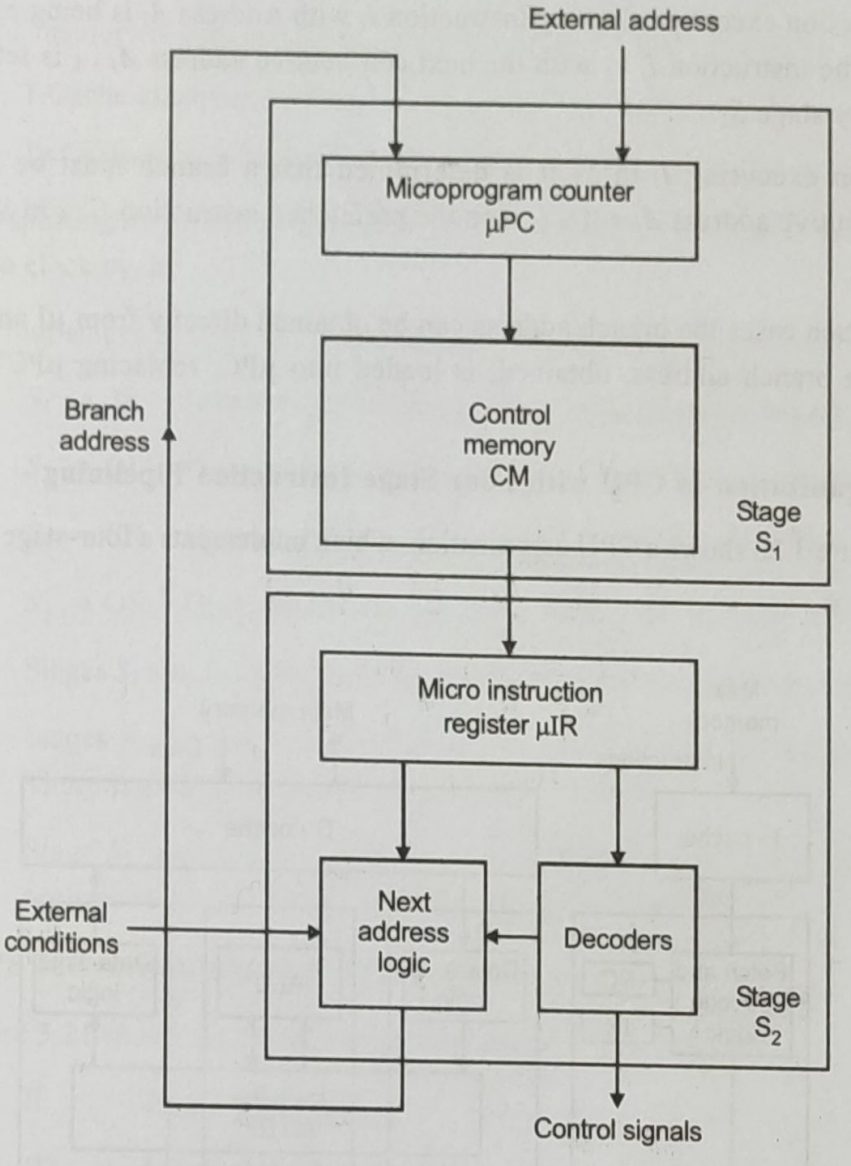


Figure 3.19 Two-stage pipelined microprogrm cotrol unit

- The execution stage contains:
  1. Microinstruction Registers μIR. μIR acts as the buffer register for S_2.
  2. The decoders which extract control signals from microinstructions in μIR.
  3. The logic for choosing branch addresses.

The two-stage pipeline increases throughput by overlapping instruction fetchig and instruction execution. During Instruction I_i with Address A_i is being executed by stage S_2, the instruction I_{i+1} with the next consecutive address A_{i+1} is fetched from memory by stage S_1.

If on executing I_i in S_2 it is determined that a branch must be made to a nonconsecutive address $A_j \neq A_{i+1}$, then the prefetched instruction I_{i+1} in S_1 has to be discarded.

In such cases the branch address can be obtained directly from μI and fed back to S_1. The branch address, obtained, is loaded into μPC, replacing μPC's previous contents.

3.5.2 Organization of CPU with Four Stage Instruction Pipelining

Figure 3.20 shows a CPU organization, which implements a four-stage instruction pipeline.

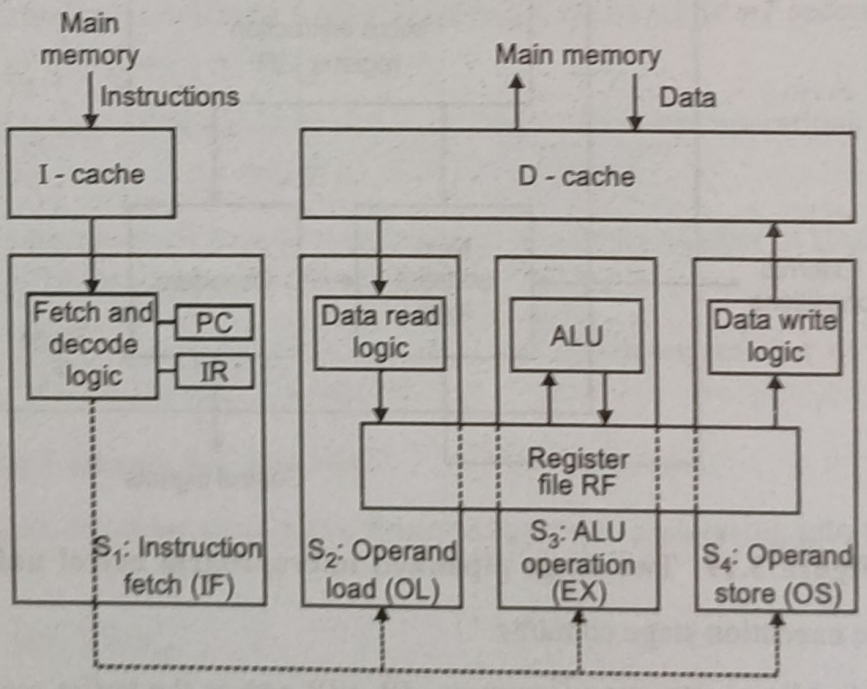


Figure 3.20 Organization of a CPU incorporating a four-stage instruction pipeline

CPU is directly connected to a cache memory.

Cache memory is split into two parts. They are:

1. I-Cache (Instruction Part)
2. D-Cache (Data Part)

This splitting helps both an instruction word and memory data word to be accessed in the same clock cycle.

The four stages S_1 to S_4 of Figure 3.20 perform the following functions.

- $S_1 \rightarrow$ IF: Instruction Fetching and decoding using the I-code.
- $S_2 \rightarrow$ OL: Operand Loading from the D-Cache to Register File (RF).
- $S_3 \rightarrow$ EX: Data processing using the ALU and Register File (RF).
- $S_4 \rightarrow$ OS: Operand storing to the D-Cache from Register File (RF).
- Stages S_2 and S_4 implement memory load and store operations, respectively.
- Stages S_2, S_3 and S_4 share the CPU's local registers in Register File RF. These registers act as interstage buffer registers.
- Stage S_3 implements data-transfer and data-processing operations of the register-to-register type, with the help of ALU of CPU.

3.5.3 MIPS Instruction Pipeline Implementation

Figure 3.21 shows the pipeline stages of single cycle data path.

1. IF : Instruction Fetch
2. ID : Instruction Decode and register file read
3. EX : Execute or address calculation
4. MEM: MEMory access
5. WB : Write Back

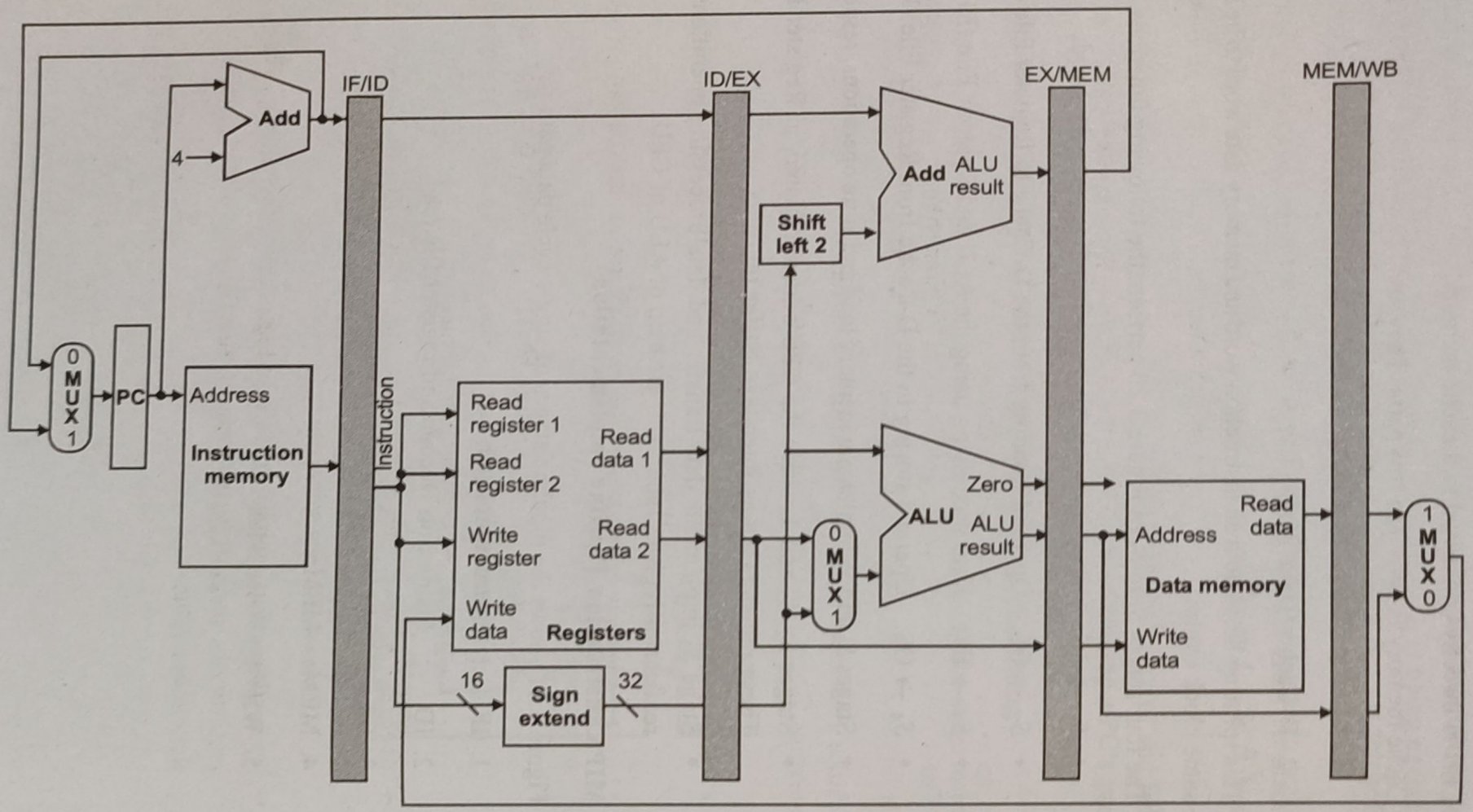


Figure 3.21 Pipeline stages of a Single-cycle Data path

3.5.4 Pipelined Control

As a first step, it is important to label the control lines on the existing data path.

Figure 3.22 shows the pipelined data path with the control signals.

The data path uses the control logic for PC source, register destination number and ALU control.

Now we require the 6-bit function code (function field) of the instruction in the EX stage as input to the ALU control.

So these bits must be included in the ID/EX pipeline register. Remember that the above mentioned 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves the bit unchanged.

There is no separate write signal for PC. Similarly there are no separate write signals for the pipeline registers IF/ID, ID/EX, EX/MEM and MEM/WB.

Since each control line is associated with a component active in only a single pipeline stage, the control lines can be divided into five groups according to the pipeline stage.

- **Instruction Fetch:** The control signals, used to read instruction memory and to write the PC, are always asserted.
- **Instruction decode/Register file read:** There are no optional control lines to set.
- **Execution/Address Calculation:** RegDst, ALUOp and ALUSrc are the signals to be set. The signals select the Result register, the ALU operation and either Read data 2 or sign-extended immediate for the ALU.
- **Memory Access:** The control lines/signals set in this stage are Branch, MemRead and MemWrite. These signals are set by the branch equal, load and store instructions respectively.
- **Write-Back:** The control line MemtoReg decides between sending the ALU result or the memory value to the register file. RegWrite control line writes the chosen value.

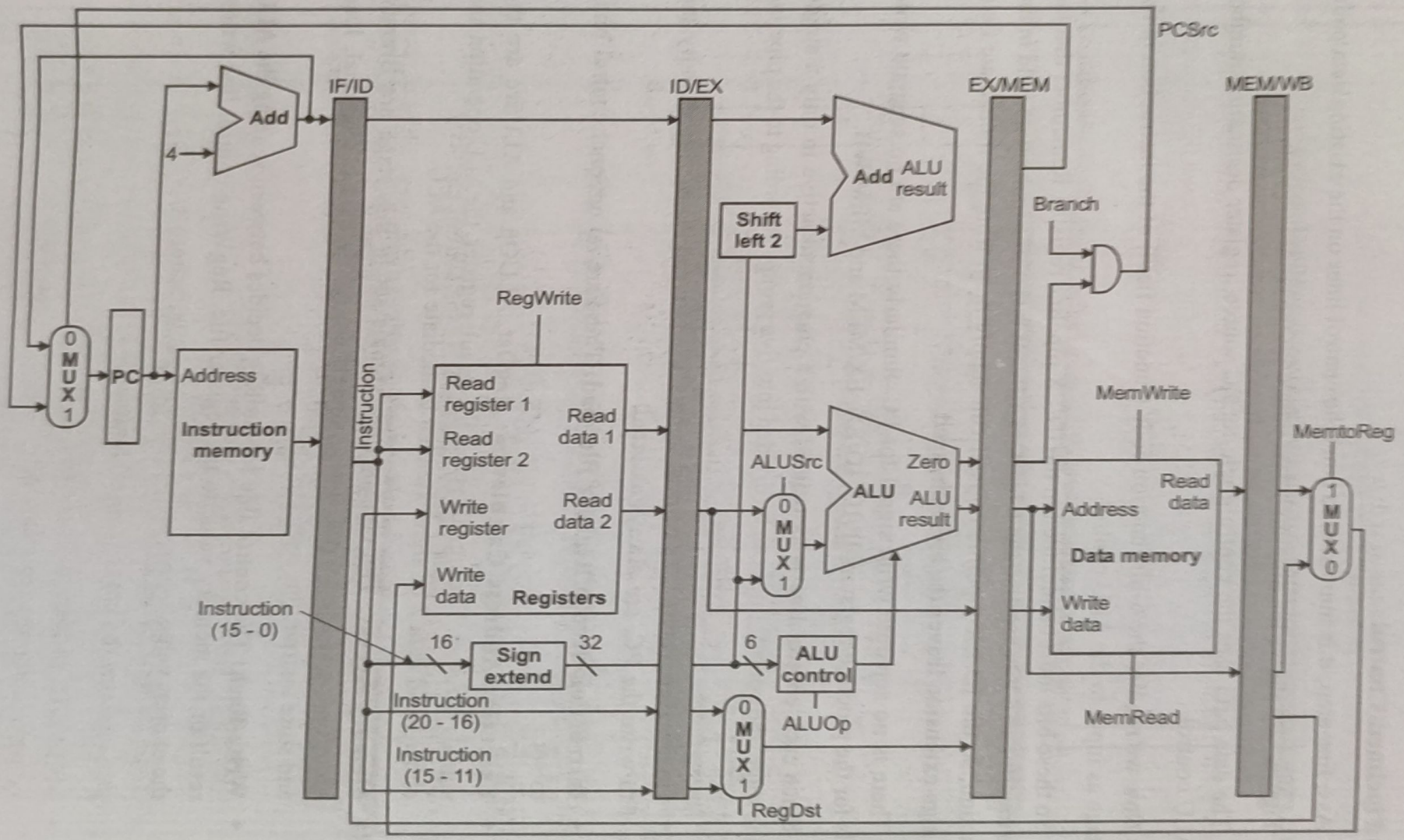


Figure 3.22 Pipelined Data path with the control signals

Pipeline Control Issues and Hardware

Observe that there is nothing to control during instruction fetch and decode (IF and ID). Thus, we can begin our control activities (initialization of control signals) during ID, since control will only be exerted during EX, MEM, and WB stages of the pipeline. Recalling that the various stages of control and buffer circuitry between the pipeline stages are labelled IF/ID, ID/EX, EX/MEM, and MEM/WB, we have the propagation of control, shown in Figure 3.23.

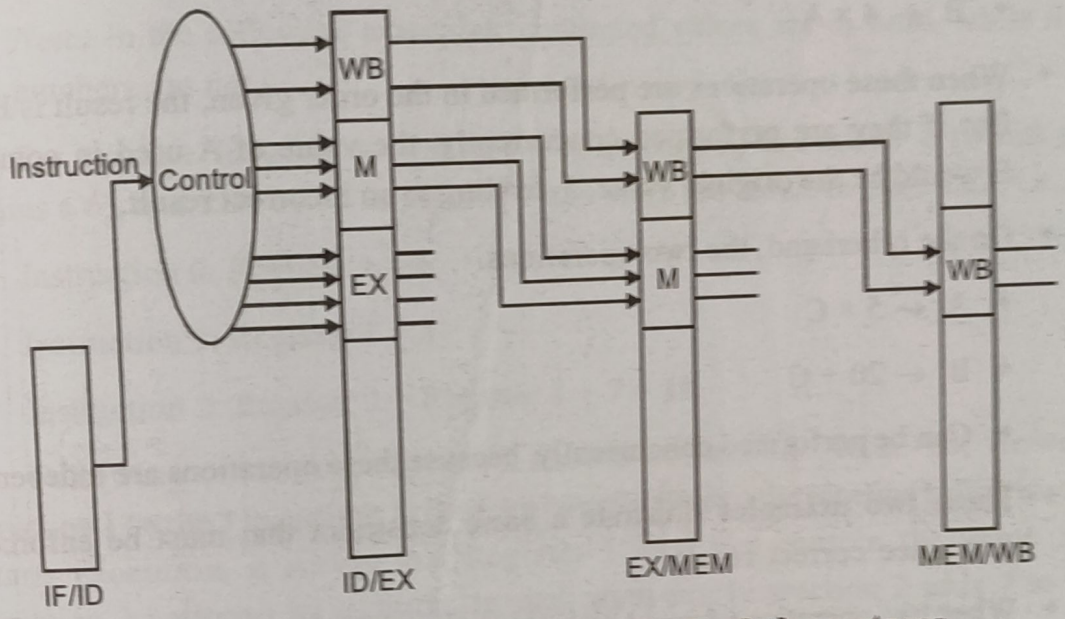


Figure 3.23 Control lines for the final three stages

Here, the following stages perform work as specified.

- IF/ID: Initializes control by passing the *rs*, *rd*, and *rt* fields of the instruction, together with the op code and funct fields, to the control circuitry.
- ID/EX: Buffers control for the EX, MEM, and WB stages, while executing control for the EX stage. Control decides what operands will be input to the ALU, what ALU operation will be performed, and whether or not a branch is to be taken based on the ALU Zero output.
- EX/MEM: Buffers control for the MEM and WB stages, while executing control for the MEM stage. The control lines are set for memory read or write, as well as for data selection for memory write. This stage of control also contains the branch control logic.
- MEM/WB: Buffers and executes for the WB stage, and selects the value to be written into the register file.

3.6 HANDLING DATA HAZARDS

- Consider a program that contains two instructions, I_1 followed by I_2. When this program is executed in a pipeline, the execution of I_2 can begin before the execution of I_1 is completed. This means that the results generated by I_1 may not be available for use by I_2.
- Assume that A = 5, and consider the following two operations.
  - $A \leftarrow 3 + A$
  - $B \leftarrow 4 \times A$
- When these operations are performed in the order given, the result is B = 32. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result.
- On the otherhand, the two operations.
  - $A \leftarrow 5 \times C$
  - $B \leftarrow 20 + C$
  - Can be performed concurrently, because these operations are independent.
- These two examples illustrate a basic constraint that must be enforced to guarantee correct results.
- When two operations depend on each other, they must be performed sequentially in the correct order.

There are several main solutions and algorithms used to resolve data hazards.

- Insert a pipeline bubble whenever a read after write (RAW) dependency is encountered, guaranteed to increase latency, or

> Pipeline bubble is a delay in the execution of an instruction in an instruction pipeline in order to resolve a hazard

- Utilize out-of-order execution to potentially prevent the need for pipeline bubbles.
- Utilize operand forwarding to use data from later stages in the pipeline.

In the case of out-of-order execution, the algorithm used can be

- Scoreboarding, in which case a pipeline bubble will only be needed when there is no functional unit available.

- The Tomasulo algorithm, which utilizes register renaming allowing the continual issuing of instructions.

We can delegate the task of removing data dependencies to the compiler, which can fill in an appropriate number of NOP instructions between dependent instructions to ensure correct operation, or re-order instructions where possible.

Operand forwarding

Examples

Note: In the following examples, computed values are in **bold**, while Register numbers are not.

For instance, let's say we want to write the value 3 to register 1, (which already contains a 6), and then add 7 to register 1 and store the result in register 2, i.e.,

Instruction 0: Register 1 = **6**

Instruction 1: Register 1 = **3**

Instruction 2: Register 2 = Register 1 + **7** = **10**

Following execution, register 2 should contain the value **10**. However, if Instruction 1 (write **3** to register 1) does not completely exit the pipeline before instruction 2 starts execution, it means that Register 1 does not contain the value **3** when Instruction 2 performs its addition. In such an event, Instruction 2 adds *7* to the old value of register 1 (**6**), and so register 2 would contain **13** instead, i.e.,

Instruction 0: Register 1 = **6**

Instruction 2: Register 2 = Register 1 + **7** = **13**

Instruction 1: Register 1 = **3**

The error occurs because Instruction 2 reads Register 1 before Instruction 1 has committed/stored the result of its write operation to Register 1. So when Instruction 2 is reading the contents of Register 1, Register 1 still contains **6**, not **3**.

Forwarding (described below) helps correct such errors by depending on the fact that the output of Instruction 1 (which is **3**) can be used by subsequent instructions before the value **3** is committed to/stored in Register 1.

Forwarding applied to our example means that we do not wait to commit/store the output of Instruction 1 in Register 1 (in this example, the output is **3**) before making thta output available to the subsequent instruction (in this case, Instruction 2).

The effect is that Instruction 2 uses the correct (the more recent) value of Register 1: the commit/store was made immediately and not pipelined.

With forwarding enabled, the ID/EX or Instruction Decode/Execution stage of the pipeline now has two inputs: the value read from the register specified (in this example, the value **6** from Register 1), and the new value of Register 1 (in this example, this value is **3**) which is sent from the next stage (EX/MEM) or Instruction Execute/ Memory Access. Additional control logic is used to determine which input to use.

Operand Forwarding in Data path

- The data hazard just described arises because one instruction, instruction I_2, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E_1.
- Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I_1 to be forwarded directly for use in step E_2.

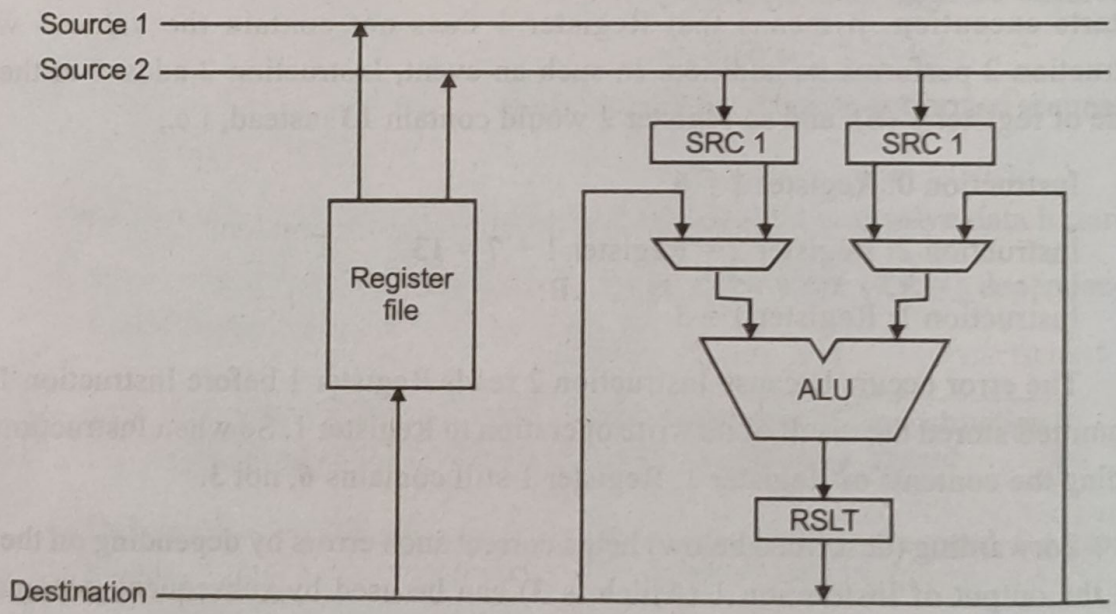


Figure 3.24 Operand Forwarding in Data path

Operand Forwarding in a Pipelined processor

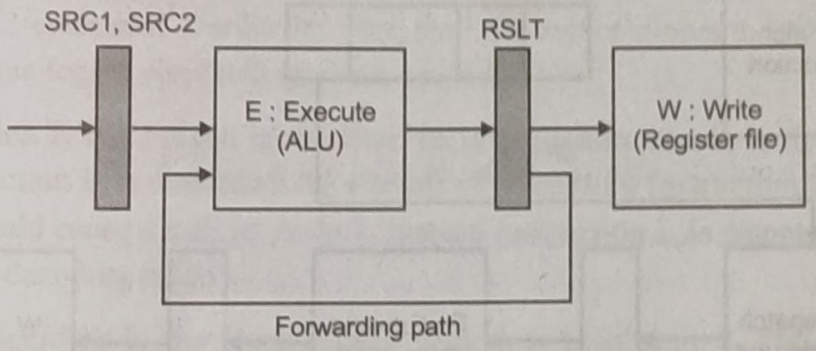


Figure 3.25 Operand Forwarding in a Pipelined processor

Handling Data Hazards in Software

- An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software.
- In this case, the compiler can introduce two-cycle delay needed between instruction I_1, and I_2 by inserting NOP (No-operation) instructions, as follows.
  - I_1: Mul R_2, R_3, R_4
  - NOP
  - NOP
  - I_2: Add R_5, R_4, R_6

Disadvantages of adding NOP instructions

- Lead to larger code size.
- Lead to reduce performace on a different implementation.

3.7 HANDLING CONTROL HAZARDS

3.7.1 Instruction Queue and Prefetching

To reduce the effect of cache miss or branch penalty, many processors use fetch units that can fetch instructions before they are needed and put them in a queue. This is shown in Figure 3.26.

When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However the fetch unit continues to fetch instructions and add them to the queue-conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

Figure 3.26 shows the method of queue length changing. This Figure also shows how the changes in queue length affects the relationship between different pipeline stages.

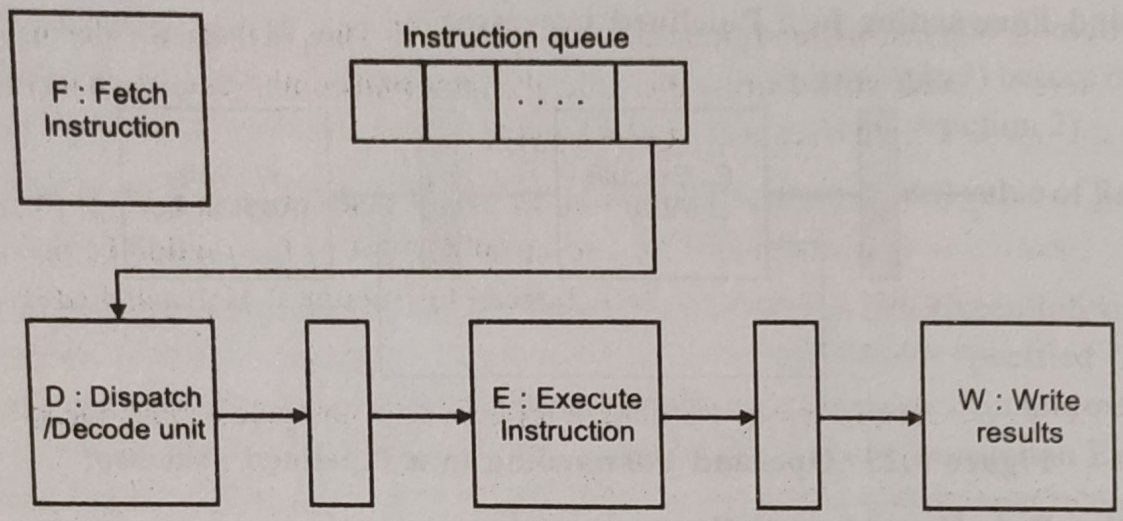


Figure 3.26

Each/Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Because of this act, the queue length remains the same for the first four clock cycles.

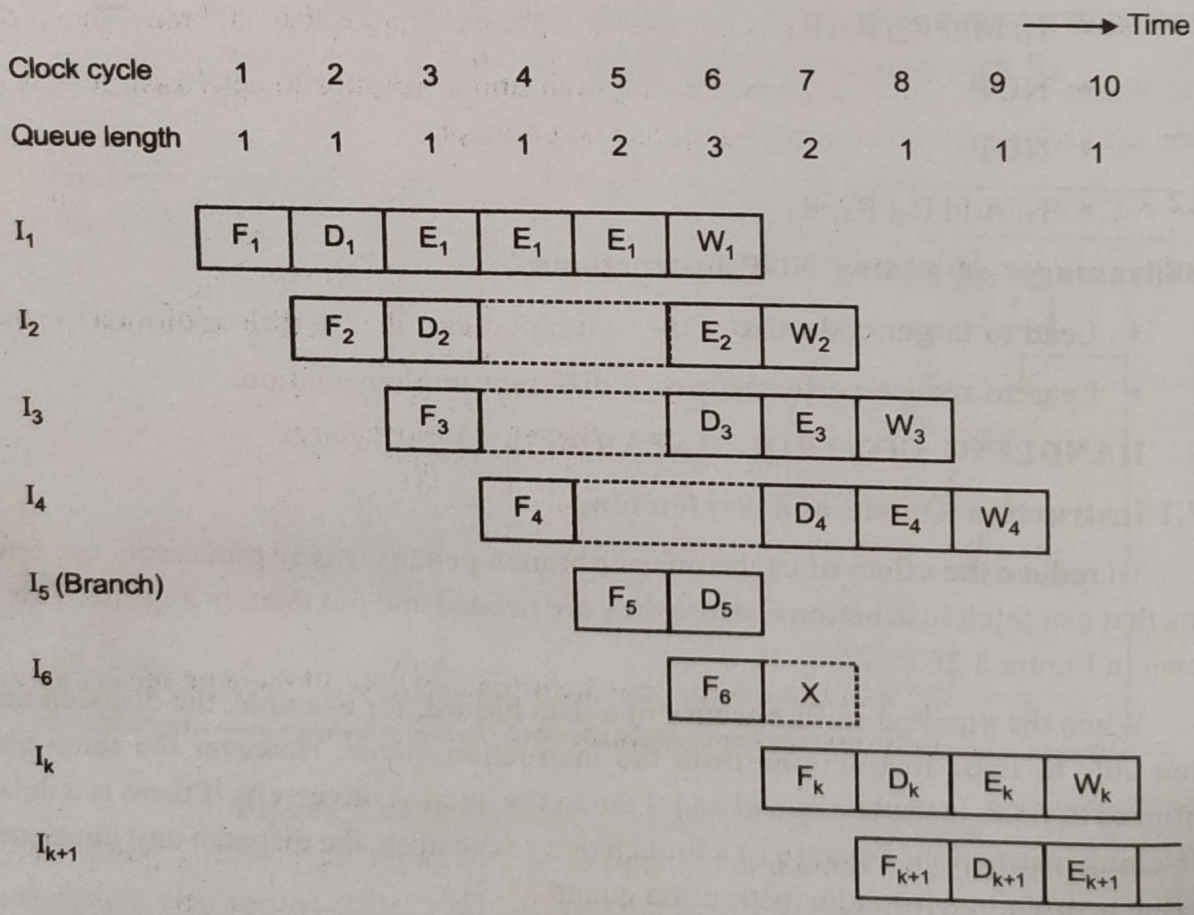


Figure 3.27 Branch timing in the presence of an instruction queue

Assume that instruction I_1 introduces 2-cycle stall. Then in that case, the instruction I_1 introduces a 2-cycle stall. During this time, the fetch unit continues to fetch instructions and so the queue length rises to 3 in clock cycle 6.

Instruction I_5 is a branch instruction. Its target instruction is I_k. I_k is fetched in cycle 7. Instruction I_6 is discarded. As a result of discarding Instruction I_6, the branch instruction would cause a stall in cycle 7. Instead Instruction I_4 is dispatched from the queue and the decoding stage.

After discarding I_6, the queue length reduces to 1, in cycle 8. The queue length will the same (i.e., 1) until another stall is encountered.

I_1, I_2, I_3, I_4 and I_k instructions complete execution in successive clock cycles. Hence the branch instruction does not increase the overall execution time.

The technique in which instruction fetch unit executes the branch instruction simultaneously with the execution of other instructions is called **branch folding**.

It is to be noted that branch folding occurs only if at the time a branch instruction is encountered, atleast one instruction is available in the queue other than the branch instruction.

When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as the instruction queue is not empty.

3.7.2 Approaches to Deal

Conditional Branch Instructions

The ways in which an instruction pipeline can deal with conditional branch instructions. They are:

- Multiple streams
- Loop buffer
- Delayed Branch
- Prefetch branch target
- Branch prediction

Multiple Streams

A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams.

Prefetch Branch Target

When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

Loop Buffer

A loop buffer is a small, very-hig-speed memory maintained by the instruction fetch stage of the pipeline and containing the *n* most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer.

Delayed Branch

It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.

3.7.3 Branch Prediction

A prediction is made whether a conditional branch will be taken when executed, and subsequent instructions are fetched accordingly.

Various prediction techniques canbe used to predict whether a branch will be taken. Various prediction techniques are

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static.

The first two approaches are the simplest. If prediction is wrong in first two approaches, then a page fault or protection violation occurs. Afterwards the processor halts its prefetching and fetches the instruction from the desired address.

The third/final static approach, the prediction decision is based on the opcode of the branch instruction.

The fourth and fifth approaches are dynamic.

Branch Prediction Types

There are two types of branch prediction. They are

1. Static Branch Prediction
2. Dynamic Branch Prediction

Static Branch Prediction

The static branch prediction is just predicting whether all branches are taken or not taken. Static predictor has the worst performance.

Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing.

Dynamic Branch Prediction

In Dynamic branch prediction, the prediction is updated dynamically. In this case, prediction buffers are used to constantly update the prediction decision.

Figure 3.28 shows the k-stage pipeline.

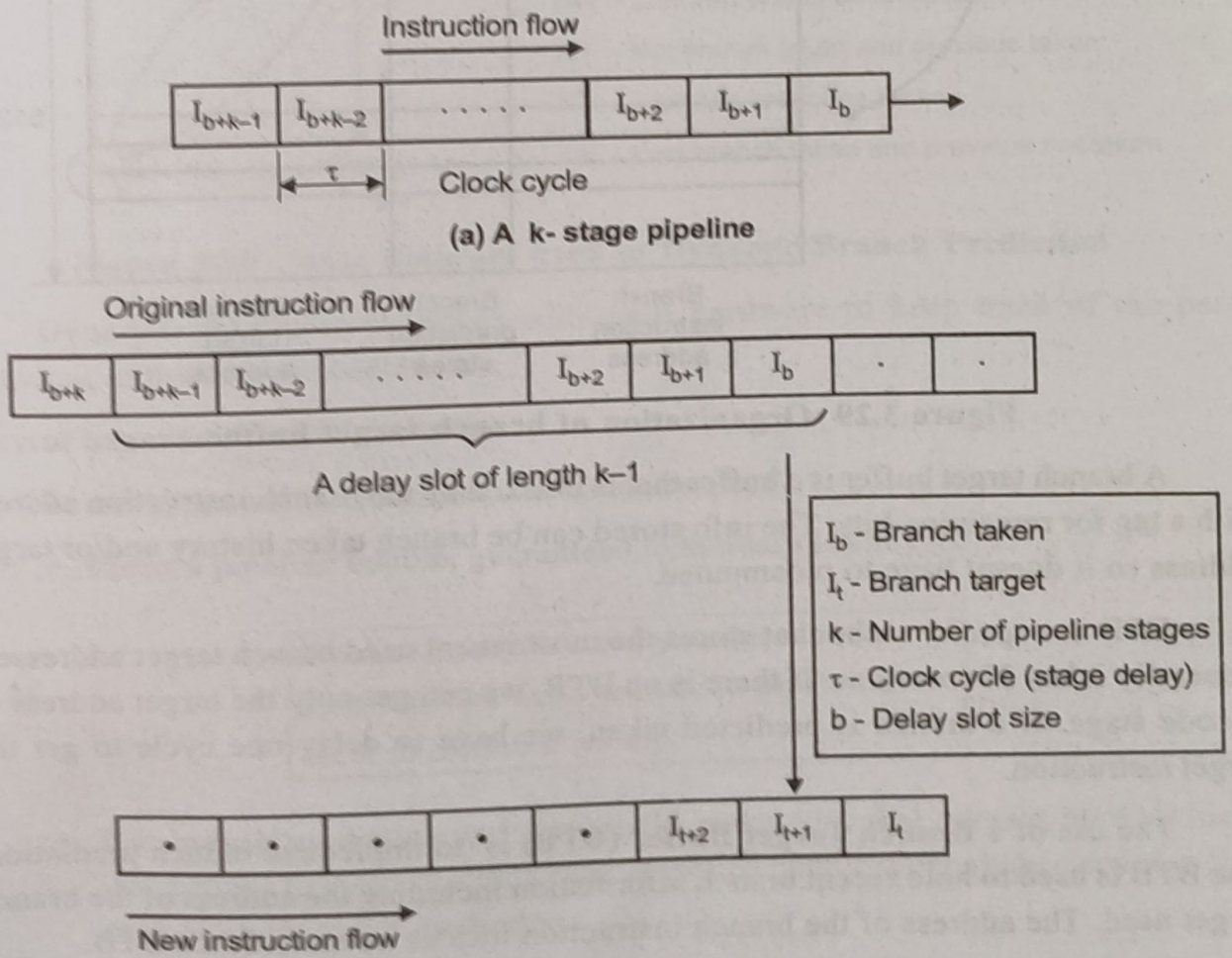


Figure 3.28 k-stage pipeline

The recent branch information is stored in the buffer called Branch Target Buffer (BTB).

Along with the above information Branch Target Buffer also stores the address of the Branch Target.

Branch Target Buffer

To further increase the prediction accuracy, many systems now incorporate branch target buffers. Figure 3.29 shows the organization of a Target Buffer.

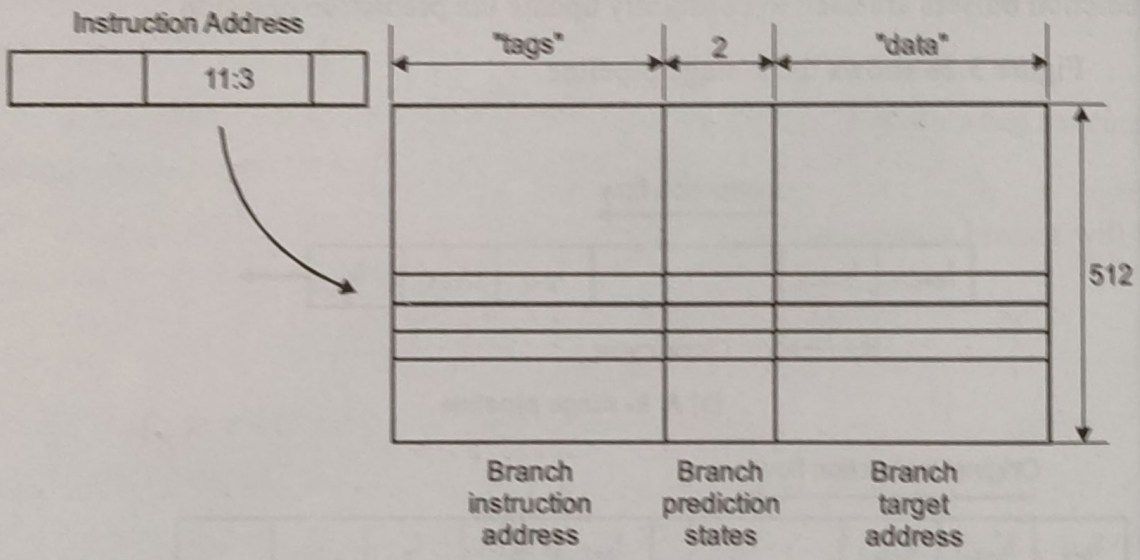


Figure 3.29 Organization of branch target buffer

A branch target buffer is a buffer that is indexed by the branch instruction address with a tag for remaining bits. The info stored can be branch taken history and/or target address so it doesnt have to recomputed.

BTB is a special cache that stores the most recent used branch target addresses. Generally it has 32 locations. If there is no BTB, we can get only the target address in decode stage. If a branch is predicted taken, we have to delay one cycle to get the target instruction.

The use of a **Branch Target Buffer (BTB)** is to implement branch prediction. The BTB is used to hold recent branch information including the address of the branch target used. The address of the branch instruction locates its entry in the BTB.

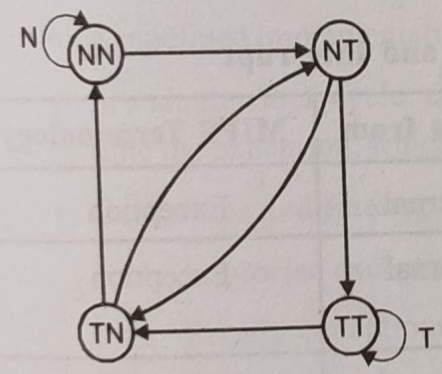
Figure 5.30 shows the state diagram used in Dynamic Branch Prediction.

The state diagram allows the backtracking of last two instructions in a given program.

Dynamic Branch strategies have been classified into three classes. They are

- One class predicts the branch direction based upon information found at the decode stage.
- The second class uses a cache to store target addresses at the stage the effective addresses of the branch target is computed.
- The third class uses a cache to store target instructions at the fetch stage.

All dynamic predictions are adjusted dynamically as a program is executed.



T - Branch taken
N - Branch not taken
NN - Last two branches not taken
NT - Not branch taken and previous taken
TT - Last two branches taken
TN - Last branch taken and previous not taken

Figure 3.30 State diagram used in Dynamic Branch Prediction

Dynamic prediction demands additional hardware to keep track of the past behaviour of the branch prediction.

Control hazards (branch hazards)

To avoid control hazards microarchitectures, can

- Insert a pipeline bubble, guaranteed to increase latency, or

> Pipeline bubble is a delay in the execution of an instruction in an instruction pipeline in order to resolve a hazard

- Use branch prediction and essentially make educated guesses about which instructions to insert in which case a pipeline bubble will only be needed in the case of an incorrect prediction.

In the event that a branch causes a pipeline bubble after incorrect instructions have entered the pipeline, care must be taken to prevent any of the wrongly loaded instructions from having any effect on the processor state excluding energy wasted processing them before they were discovered to be loaded incorrectly.

3.8 EXCEPTIONS

- Implementing **exceptions** and **Interrupts** is one of the difficult parts of the control.
- Exception is an internally unscheduled event that disrupts program execution, used to detect overflow.
- Interrupt comes from outside of the processor. This event is from external.

The following table shows some of the examples to know well about exception and interrupt.

Table 3.9 Examples of Except and Interrupt

Sl.No	Event	Comes from	MIPS Terminology
1.	Arithmetic overflow	Internal	Exception
2.	Invoke the operating system from user program	Internal	Exception
3.	Input Output device request	External	Interrupt
4.	Using an undefined instruction	Internal	Exception
5.	Hardware malfunctions	External or Internal	Interrupt or Exception

Types of Exceptions

1. Execution of an undefined instruction.
2. Arithmetic overflow in the instruction add $1, $2, $1.

Response to an exception

- The exception status vector is carried along as the instruction goes down the pipeline.
- When an exception occurs, the processor has to save the address of the offending instruction in the Exception Program Counter (EPC). Then transfer the control to the operating system at a particular/specified address.
- Then the operating system will take a suitable action.
- After performing the suitable action, the operating system can terminate the program or may continue its execution. With the help of EPC, it can be determined where to restart/continue the execution of the program.

Methods used to communicate the reason for an exception.

There are two methods to communicate the reason for an exception.

1. **Status register method:** In this method, a **Status register (Cause register)** is used by MIPS architecture. This register indicates the reason for the exception.
2. **Vectored interrupts method:** In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception.

Exception in a Pipelined Implementation

- A pipelined implementation treats exception as another form of control hazard.
- In a single clock cycle, multiple exceptions can occur simultaneously. At that time the exception, which is having highest priority, is serviced first.
- Exceptions and Interrupts are further classified into two types. They are
  1. Imprecise exceptions or imprecise interrupts.
  2. Precise exceptions or interrupts.
- **Precise exception or Precise interrupt**

  An exception or interrupt, that is associated with the correct instruction in pipelined computers is called **precise exception or precise interrupt**.
- **Impercise exception or Imprecise interrupt**

  An exception or interrupt, that is not associated with the correct instruction in pipelined computers is called **Imprecise exception** or **Imprecise interrupt.**

More about Data Hazards

In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline.

There are three situations in which a data hazard can occur:

1. Read After Write (RAW), a true dependency
2. Write After Read (WAR), an anti-dependency
3. Write After Write (WAW), an output dependency

Consider two instructions $i1$ and $i2$ with $i1$ occuring before $i2$ in program order.

Read After Write (RAW)

($i2$ tries to read a source before $i1$ writes to it). A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved.

This can occur because even though an instruction is excecuted after a previous instruction, the previous instruction has not been completely processed through the pipeline.

Example

For example,

$$i1 \cdot R2 \leftarrow R1 + R3$$

$$i2 \cdot R4 \leftarrow R2 + R3$$

The first instruction is calculating a value to be saved in register $R2$ and the second is going to use this value to compute a result for regiser $R4$. However, in a pipeline, when we fetch the operands for the 2^{nd} operation, the results from the first will not yet have been saved and hence we have a data dependency.

We say that there is a data dependency with instruction $i2$, as it is dependent on the completion of instruction $i1$.

Write After Read (WAR)

($i2$ tries to write a destination before it is read by $i1$) A write after read (WAR) data hazard represents a problem with concurrent execution.

Example

For example

$$i1 \cdot R4 \leftarrow R1 + R5$$

$$i2 \cdot R5 \leftarrow R1 + R2$$

If we are in a situation that there is a chance that $i2$ may be completed before $i1$ (i.e., with concurrent execution) we must ensure that we do not store the result of register $R5$ before $i1$ has had a chance to fetch the operands.

Write After Write (WAW)

($i2$ tries to write an operand before it is written by $i1$) A write after write (WAW) data hazard may occur in a concurrent execution environment.

Example

For example

$$i1 \cdot R2 \leftarrow R4 + R7$$

$$i2 \cdot R2 \leftarrow R1 + R3$$

We must delay the WB (Write Back) of $i2$ until the execution of $i1$ finishes.

SHORT QUESTIONS AND ANSWERS

1. What are R-type instructions? (Q5, April/May 2015)

R- type instructions have an opcode 0. These instructions have three register operands: **rs, rt** and **rd**. Fields **rs** and **rt** are sources and **rd** is the destination.

2. What is meant by Branch prediction? (Q6, Nov/Dec 2015)

A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

3. What is a Branch prediction buffer? (Q6, April/May 2015)

A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

4. What is the need for Speculation? (Q5, Nov/Dec 2014)

One of the most important methods for finding and exploiting more ILP (Instruction Level Parallelism) is ***Speculation***.

Speculation is an approach that allows the compiler or the processor to "guess" about the properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction.

5. What is Instruction pipelining?

Instruction pipelining is a technique that implements a form of parallelism called instruction level-parallelism within a single processor.

6. What is a Hazard? What are its types? (Q5, Nov/Dec 2015)

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called ***Hazards***.

Any reason that causes the pipeline to stall is called a ***Hazard***. Different types of hazards are

1. Data Hazards
2. Structural Hazards
3. Control Hazards

7. Define Data Hazard.

When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. This type of hazard is called ***Data Hazard***.

8. What are the classification of Data Hazards?

1. RAW (Read After Write) Hazard
2. WAR (Write After Read) Hazard
3. WAR (Write After Write) Hazard

9. Define Structural Hazard.

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time.

Structural hazards occur when different instructions collide while trying to access the same piece of hardware in the same segment of pipeline.

10. Define Control Hazard.

Control Hazard occurs with branches. On many instruction pipeline micro architectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the fetch stage).

Control Hazard arises from the need to make a decision based on the results of one instruction.

11. What is Exception? **(Q6, Nov/Dec 2014)**

Exception is an internally unscheduled event that disrupts program execution, used to detect an overflow.

REVIEW QUESTIONS

1. Explain the basic MIPS implementation with necessary multiplexers and control lines. **(Q13.(a), Nov/Dec 2015)**
   [*Refer Section 3.1*]
2. Explain Data path and its control in detail. **(Q13.(a), Nov/Dec 2014)**
   [*Refer Section 3.5*]
3. Explain how the instruction pipeline works? What are the various situations where an instruction pipeline can stall? Illustrate with an example. **(Q13.(b), Nov/Dec 2015)**
   [*Refer Section 3.5*]
4. What is Hazard? Explain its types with suitable examples. **(Q13.(b),Nov/Dec 2014)**
   [*Refer Sections 3.4.2, 3.6 and 3.7*]
   [*See also more about Data Hazards at the end of 3rd unit*]
5. Explain the different types of pipeline hazards with suitable examples. **(Q13.(a), April/May 2015)**
   [*Refer Sections 3.4.2, 3.6 and 3.7*]
   [*See also more about Data Hazards at the end of 3rd unit*]
6. Explain in detail how exceptions are handled in MIPS architecture. **(Q13.(b), April/May 2015)**
   [*Refer Section 3.8*]