UNIT III

QNo	Questions
1	Explain the basic MIPS implementation with necessary diagrams. **Basic MIPS Implementation** In this chapter we will see the implementation of a subset of the core MIPS instruction set. These instructions are divided into three classes : • The memory-reference instructions: **load word (lw) and store word (sw)** • The arithmetic-logical instructions: **add, sub, AND, OR, and slt** • The branch instructions: **branch equal (beq) and jump (j)** The subset considered here does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions. The key principles used in creating a datapath and designing the control are discussed here. The implementation of the remaining instructions is somewhat similar. For implementing every instruction, the first two steps are same: **1. Fetch the instruction**: Send the Program Counter (PC) contents (address of instruction) to the memory that contains the opcode and fetch the instruction from that memory. **2. Fetch operand(s) :** Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions we require to read two registers. The remaining actions required to complete the instruction depend on the instruction class. For each of the three instruction classes (memory-reference, arithmetic-logical and branches), the actions are mostly the same, independent of the exact instruction. This shows that the simplicity and regularity of the MIPS instruction set simplifies the implementation by making the execution of many of the instruction classes similar. For example, all instruction classes, except jump, 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 and •Branches use the ALU for comparison. •After using the ALU, the actions required to complete various instruction classes are not same. A memory-reference instruction needs to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. A branch instruction may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction. Fig. 3. 1 shows the block diagram of a MIPS implementation, showing the functional units and interconnection between them.

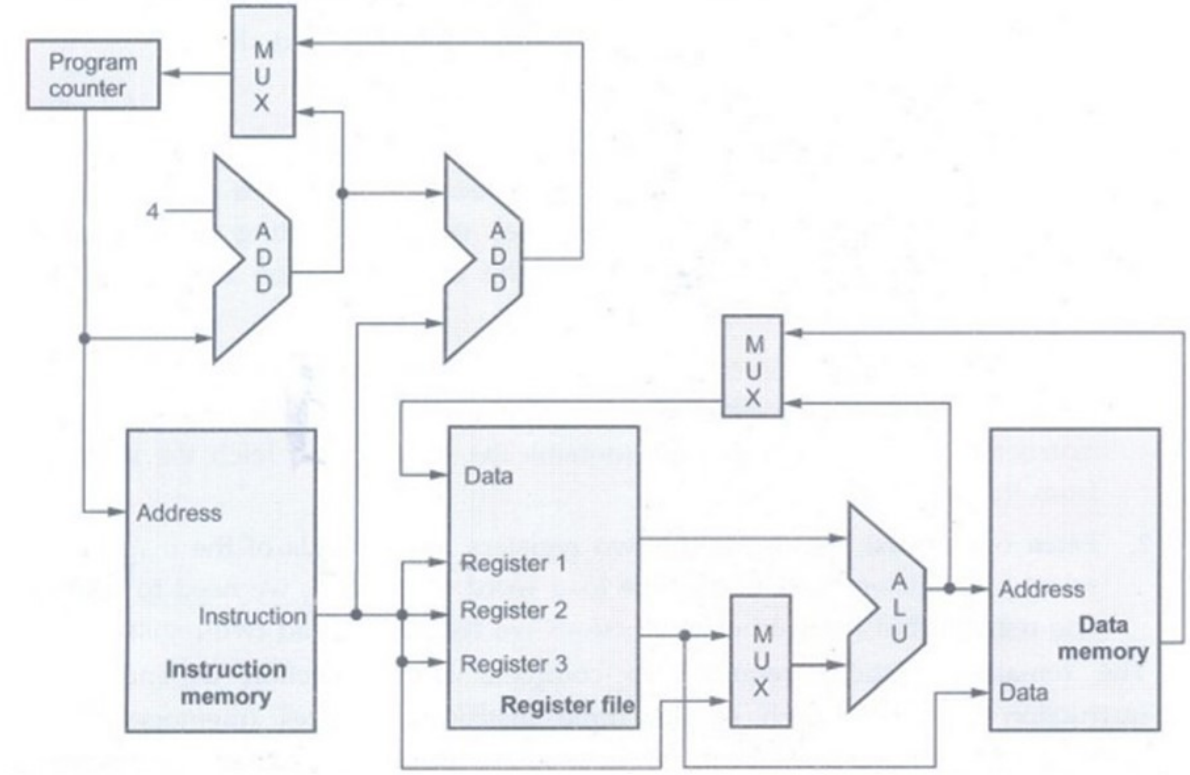


Fig 3.1 Major functional units and interconnections between them for implementation of MIPS subset

3.1.1 Operation

The program counter gives the instruction address to the instruction memory. After the instruction is fetched, the register operands required by an instruction are specified by fields of that instruction.

Once the register operands have been fetched, they can be used to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch).

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 registers. The result from the ALU or memory is written back into the register file.

Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by 4.

2

Explain in detail about building a data path for various instructions.

Building a Data Path

As shown in Fig. 3.4, the MIPS implementation includes, the datapath elements (a unit used to operate on or hold data within a processor) such as the instruction and data memories, the register file, the ALU, and adders.

Fig. 3.3 shows the combination of the three elements (instruction memory, program counter and adder) from Fig. 3.4 to form a datapath that fetches instructions

and increments the PC to obtain the address of the next sequential instruction.

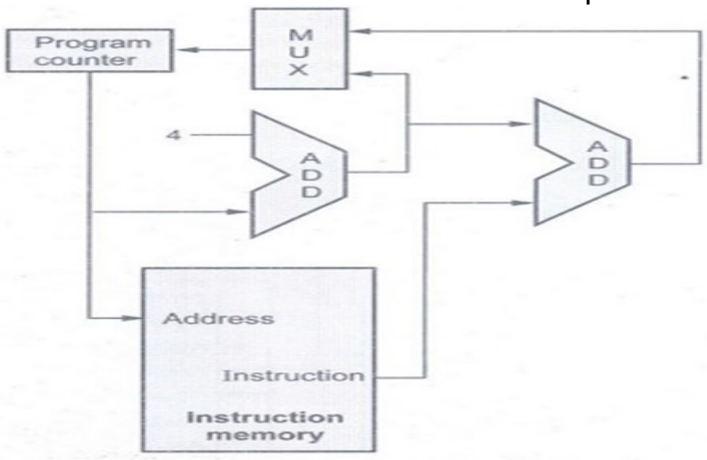


Fig 3.3 Data path to fetch instruction and increment PC

The instruction memory stores the instructions of a program and gives instruction as an output corresponding to the address specified by the program counter. The adder is used to increment the PC by 4 to the address of the next instruction.

Since the instruction memory only reads, the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed.

The program counter is a 32-bits register that is written at the end of every clock cycle and thus does not need a write control signal.

The adder always adds its two 32-bits inputs and place the sum on its output.

3.2.1. Datapath Segment for Arithmetic - Logic Instructions

The arithmetic-logic instructions read operands from two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions as R-type instructions. This instruction class includes add, sub, AND, OR, and slt. For example, OR $t1, $t2, $t3 reads $t2 and $t3, performs logical OR operation and 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. The register file contains the register state of the computer.

Fig. 3.4 shows multiport register file (two read ports and one write port) and the ALU section of Fig. 3.4 We know that, the R-format instructions have three register operands: numbers Two source operands and one destination operand.

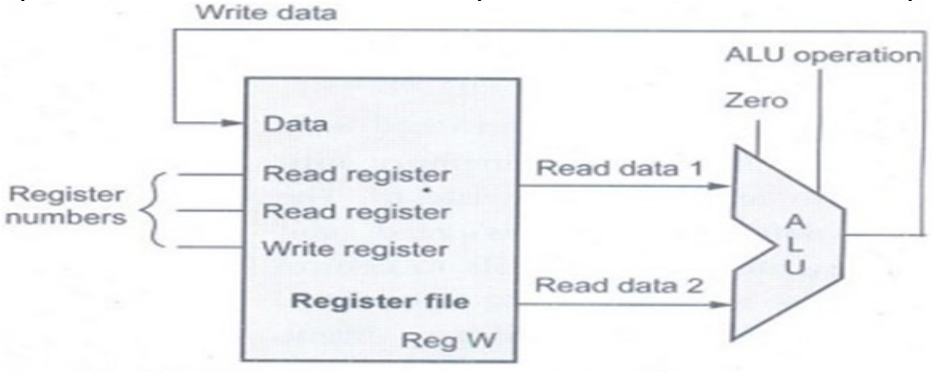


Fig 3.4 Multiport register file and the ALU

For each data word to be read from the register file, we need to specify the

register number to the register file. On the other hand, to write a data word, we need two inputs: One to specify the register number to be written and one to supply the data to be written into the register.

The register file always outputs the contents of whatever register numbers are on the Read register inputs. Write operations, however, are controlled by the write control (Reg W) signal. This signal is asserted for a write operation at the clock edge.

Since writes to the register file are edge-triggered, it is possible to perform read and write operation for the same register within a clock cycle: The read operation gives the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle.

As shown in Fig. 3.4, the register number inputs are 5 bits wide to specify one of 32 registers, whereas the data input and two data output buses are each 32 bits wide.

3.2.2. Datapath Segment for Load Word and Store Word Instructions

Now, consider the MIPS load word and store word instructions, which have the general form lw $t1, offset_value($t2) or sw $t1, offset_value ($t2).

In these instructions $t1 is a data register and $t2 is a base register. The memory address is computed by adding the base register ($t2), to the 16-bits signed offset value specified in the instruction.

In case of store instruction, the value from the data register ($t1) must be read and in case of load instruction, the value read from memory must be written into the data register ($t1). Thus, we will need both the register file and the ALU from Fig. 3.4.

We know that, the offset value is 16-bits and base register contents are 32-bits. Thus, we need a sign-extend unit to convert the 16-bits offset field in the instruction to a 32-bits signed value so that it can be added to base register.

In addition to sign extend unit, we need a data memory unit to read from or write to. The data memory has read and write control signals to control the read and write operations. It also has an address input, and an input for the data to be written into memory. Fig. 3.5 shows these two elements.

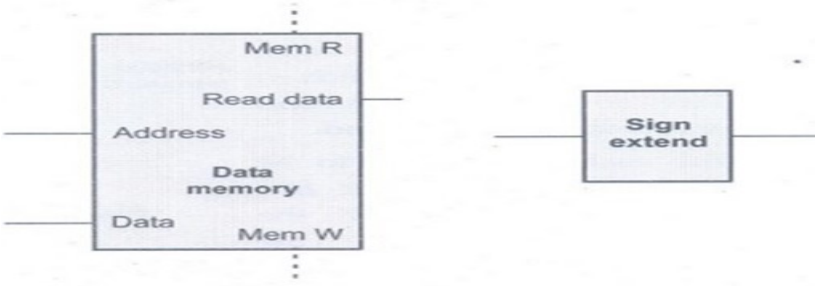


Fig 3.5 Data memory unit and the sign extension unit

Sign extension is implemented by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

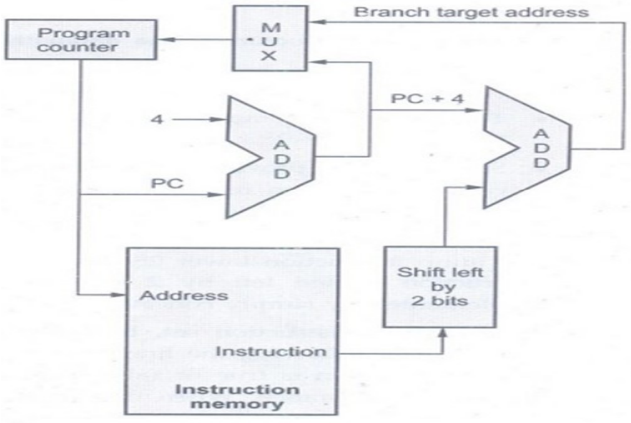
Therefore, two units needed to implement loads and stores, in addition to the register file and ALU of Fig. 3.4, are the data memory unit and the sign extension unit.

3.2.3. Datapath Segment for Branch Instruction

The beq instruction has three operands, two registers that are compared for equality, and a 16-bits offset which is used to compute the branch target address relative to the branch instruction address. It has a general form beq $t1, $t2, offset.

To implement this instruction, it is necessary to compute the branch target address by adding the sign-extended offset field of the instruction to the PC. The two important things in the definition of branch instructions which need careful attention are:

The instruction set architecture specifies that the base for the branch address

	calculation is the address of the instruction following the branch (i.e., PC + 4 the address of the next instruction. The architecture also states that the offset field is shifted left 2 bits so that it is a word offset; this shift increases the effective range of the offset field by a factor of 4. Therefore, the branch target address is given by Branch target address = PC+4 + offset (shifted left 2 bits) In addition to computing the branch target address, we must also see whether the two operands are equal or not. If two operands are not equal the next instruction is the instruction that follows sequentially (PC= PC+4); in this case, we say that the branch is not taken. On the other hand, if two operands are equal (i.e., condition is true), the branch target address becomes the new PC, and we say that the branch is taken. Thus, the branch datapath must perform two operations : Compute the branch target address and compare the registercontents.  **Fig 3.6 Computation of branch target address** Fig. 3.7 shows the structure of the datapath segment that handles branches.
3	Explain in detail about designing a control unit for MIPS. **Designing a Control Unit** Here, we restrict ourselves to implement load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than. We will also the design to include a jump instruction (j). **3.3.1. The ALU Control** The MIPS ALU defines the six following combinations of four control inputs shown in the Table 3.1: Table 3.1 Combination of Four control inputs ALU control lines / Function 0000 AND 0001 OR 0010 Add 0110 Subtract 0111 Set on less than 1100 NOR Depending on the instruction class, the ALU will need to perform one of these first five functions. (NOR function is needed for other parts of the MIPS instruction set. It is not included in the subset we are implementing.)

In case of load word and store word instructions, we use the ALU to compute the memory address by addition.

In case of the R-type instructions, the ALU needs to perform one of the five actions - AND, OR, subtract, add, or set on less than.

In case of branch equal, the ALU must perform a subtraction.

We can control the operation of ALU by the 4-bits ALU control input and 2-bits ALUOP. The 2-bits ALUOP is interpreted as shown in Table 3.2.

Table 3.2 ALUOp field

ALUOp	Action
00	Loads and stores
01	Subtract for beq
10	The operation encoded in the function field
11	-

Table 3.3 shows how to set the ALU control inputs based on the 2-bits ALUOP control and the 6-bits function code.

Instruction opcode	ALUOp	Instruction operation	Funct 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	0111

Here, multiple levels of decoding technique is used.

3.3.2. Advantages of using multiple levels of decoding

It reduces the size of the main control unit.

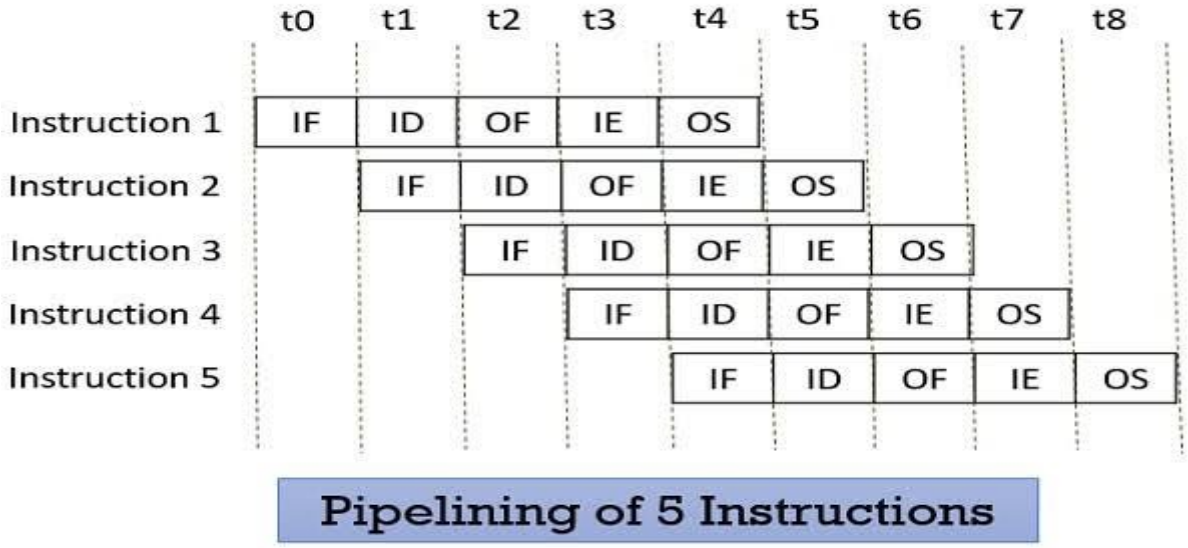
Use of several smaller control units may also potentially increase the speed of the control unit.

Table 3.4 shows how the 4-bits ALU control is set depending on these two input fields: 6-bits function fields and 2-bits ALUOp field.

Table 3.4

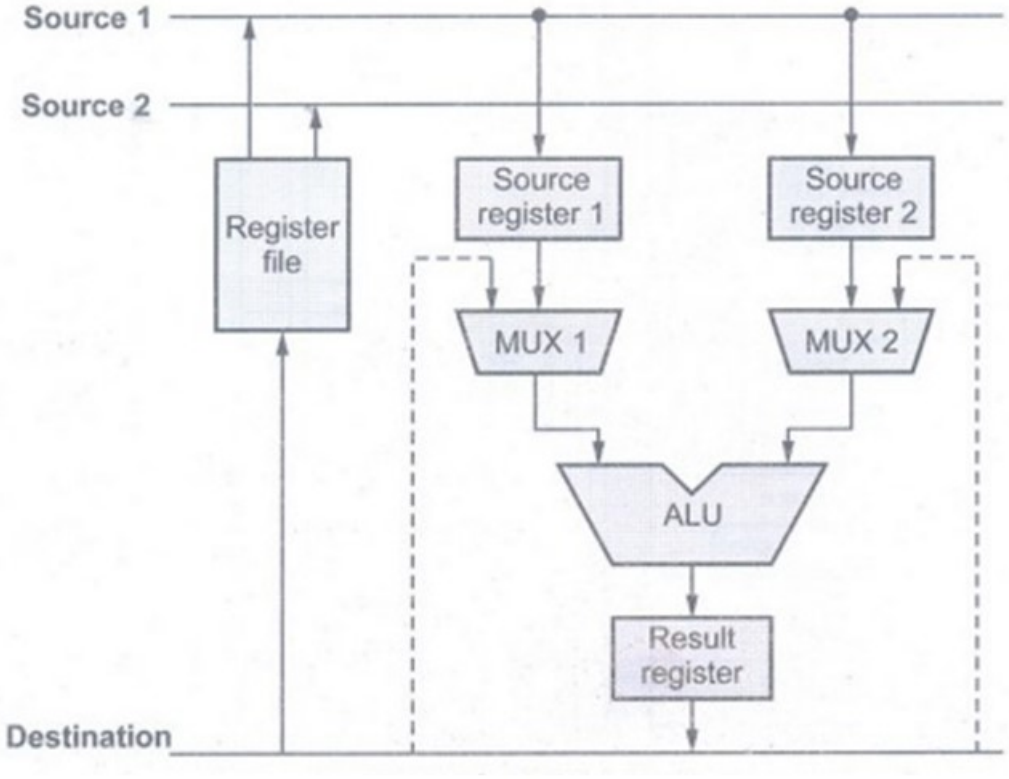
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	×	×	×	×	×	×	0010
×	1	×	×	×	×	×	×	0110
1	×	×	×	0	0	0	0	0010
1	×	×	×	0	0	1	0	0110
1	×	×	×	0	1	0	0	0000
1	×	×	×	0	1	0	1	0001
1	×	×	×	1	0	1	0	0111

Once the truth table has been constructed, it can be optimized and can be

	implemented using logic gates.
4	Explain in detail about five stage pipelining. Also write its advantage and disadvantage.  Pipelining of 5 Instructions 1. Instruction Fetch (IF) The Instruction Fetch stage is the first step in the pipeline process. Here, the CPU retrieves an instruction from the program memory. The primary tasks in this stage include: -Program Counter (PC) Management: The Program Counter holds the address of the next instruction to be fetched. It increments after each fetch, pointing to the subsequent instruction in the sequence. - Instruction Memory Access: The instruction is fetched from the instruction memory (usually a cache or RAM). This access is typically very fast due to the high-speed memory technologies used. - Buffering: The fetched instruction is placed into an Instruction Register (IR) or buffer to be used in the subsequent stages. Efficiency at this stage is paramount as delays here can stall the entire pipeline, leading to performance bottlenecks. Advanced CPUs often use techniques like prefetching to mitigate potential delays.

	2. Instruction Decode (ID) Once the instruction is fetched, it enters the Instruction Decode stage. This stage involves interpreting the fetched instruction and preparing the necessary operands for execution. Key activities include: - Opcode Decoding: The instruction is broken down into its constituent parts, such as the operation code (opcode), source operands, and destination operands. - Register Read: The required operands are read from the register file. Modern CPUs often have multiple read ports to facilitate simultaneous access to several registers. - Instruction Classification: Instructions are classified into various types (e.g., arithmetic, logical, load/store, control) to determine the subsequent steps in the pipeline. This stage is also responsible for hazard detection and forwarding to resolve data dependencies and prevent pipeline stalls. 3. Execute (EX) The Execute stage is where the actual computation or operation specified by the instruction takes place. This stage includes: - ALU Operations: Arithmetic and logical operations are performed using the Arithmetic Logic Unit (ALU). This includes operations like addition, subtraction, AND, OR, and shifts. - Address Calculation: For memory access instructions, the effective address is calculated by the ALU. - Branch Evaluation: If the instruction involves a branch, the branch condition is evaluated, and the Program Counter may be updated accordingly. The execution stage is critical for the overall performance, as complex instructions can take multiple cycles, potentially causing pipeline stalls.

	4. Memory Access (MEM) After execution, some instructions require access to memory to read or write data. The Memory Access stage handles these operations. Key functions include: - Load Operations: Data is read from memory and placed into a register. - Store Operations: Data from a register is written to memory. - Address Translation: In systems with virtual memory, the effective address calculated during the Execute stage is translated to a physical address. Efficiency in this stage is achieved through techniques like caching, which reduces latency by storing frequently accessed data closer to the CPU. 5. Write Back (WB) The final stage in the pipeline is Write Back. Here, the results of the executed instructions are written back to the CPU's register file. This stage involves: - Register Write: The computed data or the data fetched from memory is written into the destination register. - Completion Logging: The completion of the instruction is logged, and any necessary updates to status registers or flags are made. Write Back is crucial for ensuring that subsequent instructions have access to the correct and updated data, maintaining the integrity and consistency of the processor’s state. Pipelining is a fundamental technique in modern CPU design that significantly improves instruction throughput. Understanding the five stages of a pipeline—Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back—provides insights into how processors manage and execute multiple instructions efficiently. Each stage is intricately linked and optimized to minimize delays and maximize performance. Advanced techniques like parallelism, hazard detection, and caching are employed to

	further enhance the pipeline's efficiency, ensuring that modern processors can handle complex and demanding computational tasks with speed and precision.
5	Explain in detail about the methods for dealing with the data hazards. **Handling Data Hazards** **3.7.1. Operand Forwarding** A simple hardware technique which can handle data hazard is called operand forwarding or register by passing. In this technique, ALU results are fed back as ALU inputs. When the forwarding logic detects the previous ALU operation has the operand for current instruction, it forwards ALU output instead of register file. This is illustrated in Fig. 3.30 shows a portion of the processor datapath involving the ALU and the register file.  **Fig 3.30 Operand forwarding in a pipelined processor- Data path** The source and result register constitute the interstage buffers needed for pipelined operation, as shown in Fig. 3.31

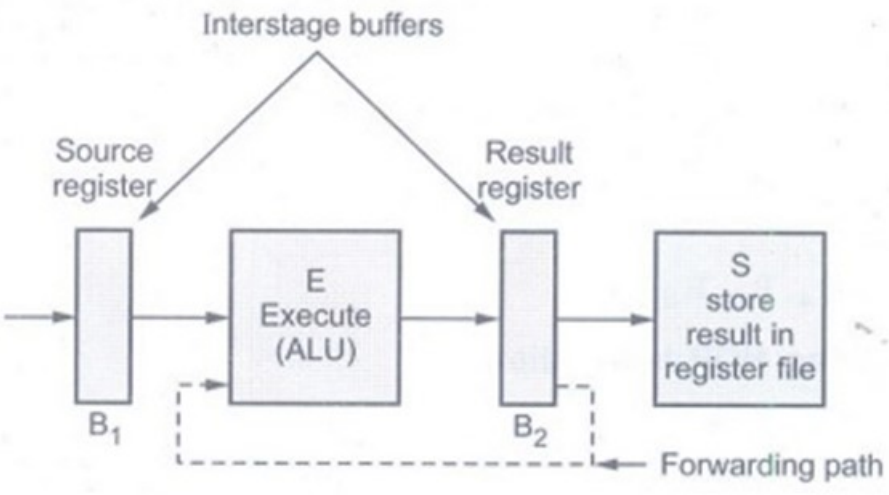


Fig 3.31 Operand forwarding in a pipelined processor – Forwarding path in the processor pipeline

The data forwarding mechanism is indicated by dashed lines.

The two multiplexers select the data for ALU either from destination bus or from source 1 and source 2 registers.

When the forwarding logic detects data dependency, it forwards ALU output available in the result register using data forwarding path to the ALU for the next operation. Hence the execution of next (dependent) instruction proceeds without interruption.

3.7.2. Handling Data Hazards in Software

In this approach, software (compiler) detects the data dependencies. If data dependencies are found it introduces necessary delay between two instructions by inserting NOP (no-operation) instructions as follows :

I_1 :MUL R_2, R_3, R_4
NOP
NOP
ADD R_4, R_5, R_6

Disadvantages of adding NOP instructions

It leads to larger code size.

A given processor may have several hardware implementations. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and hence would lead to reduce performance on a different implementation.

To achieve better performance, the compilers are designed such that they can reorder instructions to perform useful task in the NOP slots.

6

Explain the techniques for handling control hazards in pipelining.

. Handling Control Hazards

3.8.1 Instruction Queue and Prefetching

To reduce the effect of cache miss or branch penalty, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. This is illustrated in Fig. 3.33

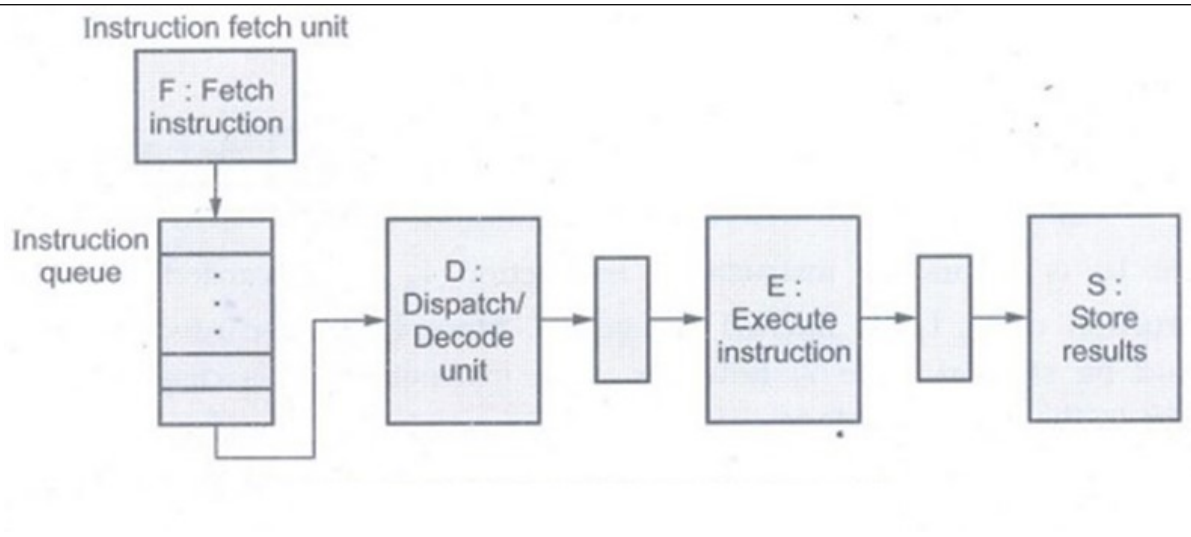


Fig 3.33 Use of instruction queue

A separate unit called dispatch unit takes instructions from the front of the queue and sends them to execution unit. It also performs the decoding function.

The fetch unit attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions during cache miss.

In case of data hazard, 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.

3.8.2. Use of instruction queue during branch instruction

Fig. 3.34 shows instruction time line. It also shows how the queue length changes over the clock cycles. Every fetch operation adds one instruction to the queue and every dispatch unit operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles.

The instruction I has 2-cycle stall. In these two cycles fetch unit adds two instructions but dispatch unit does not issue any instruction. Due to this, the queue length rises to 3 in clock cycle 6.

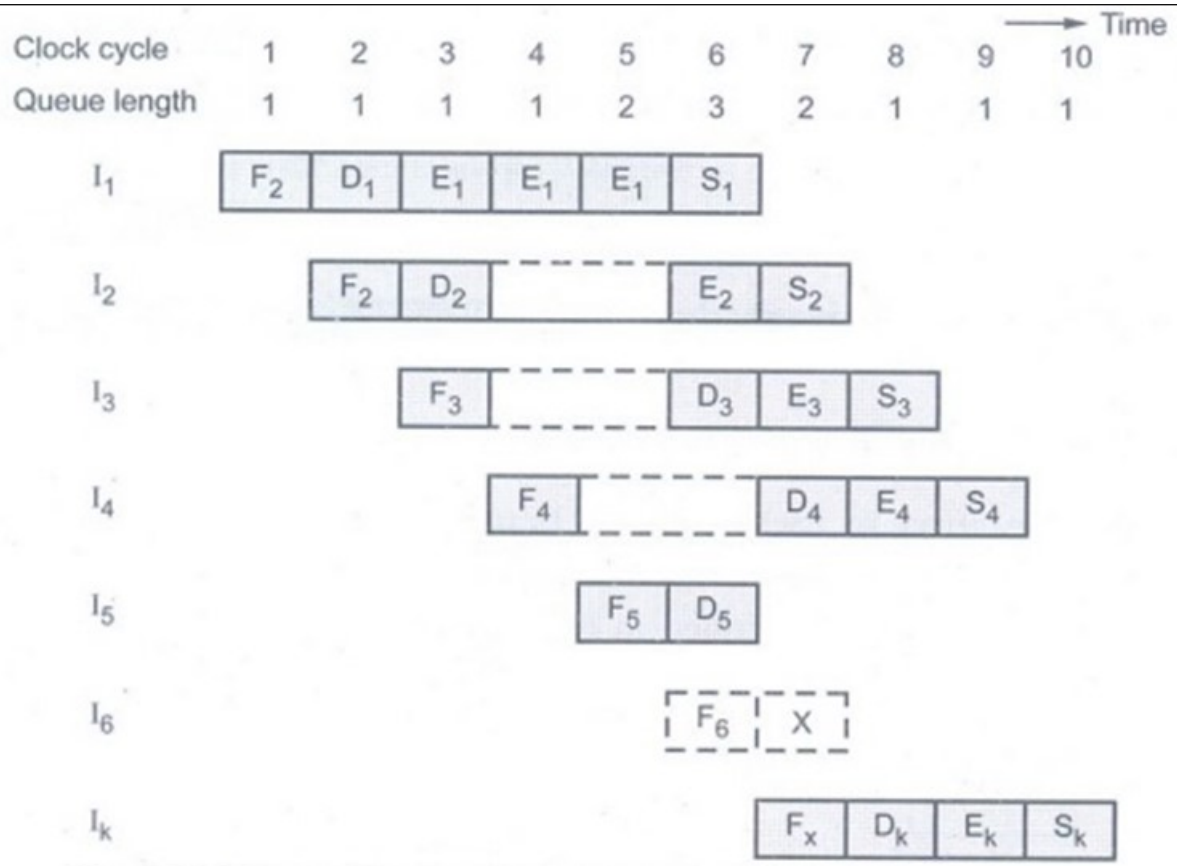


Fig 3.34 Use of instruction queue during branch instruction

Since I_5 is a branch instruction, instruction I_6 is discarded and the target instruction of I_5, I_K is fetched in cycle 7. Since I_6 is discarded, normally there would be stall in cycle 7; however, here instruction I_4 is dispatched from the queue to the decoding stage.

After discarding I_6, the queue length drops to 1 in cycle 8. The queue length remains one until another stall is encountered. In this example, instructions I_1, I_2, I_3, I_4 and I_K complete execution in successive clock cycles. Hence, the branch instruction does not increase the overall execution time.

3.8.3 Branch Folding

The technique in which instruction fetch unit executes the branch instruction (by computing the branch address) concurrently with the execution of other instructions is called branch folding.

Branch folding occurs only if there exists at least one instruction in the queue other than the branch instruction, at the time a branch instruction is encountered.

In case of cache miss, the dispatch unit can send instructions for execution as long as the instruction queue is not empty. Thus, instruction queue also prevents the delay that may occur due to cache miss.

3.8.6. Prefetch branch target

In this approach, when a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This already fetched target is used when branch is valid.

3.8.6.2. Delayed branch

The location following a branch instruction is called a branch delay slot. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction.

There are three ways to fill the delay slot :

1. The delay slot is filled with an independent instruction before branch. In this case performance always improves.
2. The delay slot is filled from the target branch instructions. Performance improves if only branch is taken.
3. The delay slot is filled with an instruction which is one of the fall through instruction. In this case performance improves if branch is not taken.

These above techniques are called delayed branching.

3.8.8. Branch Prediction Strategies

There are two types of branch prediction strategies :

- Static branch strategy
- Dynamic branch strategy.

3.8.8.1. Static Branch Strategy: In this strategy branch can be predicted based on branch code types statically. This means that the probability of branch with respect to a particular branch instruction type is used to predict the branch. This branch strategy may not produce accurate results every time.

3.8.8.2. Dynamic Branch Strategy: This strategy uses recent branch history during program execution to predict whether or not the branch will be taken next time when it occurs. It uses recent branch information to predict the next branch. The recent branch information includes branch prediction statistics such as:

T: Branch taken
N: Not taken
NN: Last two branches not taken
NT: Not branch taken and previous takes
TT: Both last two branch taken
TN: Last branch taken and previous not taken

• The recent branch information is stored in the buffer called Branch Target Buffer (BTB).
• Along with above information branch target buffer also stores the address of branch target.

Fig. 3.36 shows the organization of branch target buffer.

Fig 3.36 Branch target buffer organization

Fig. 3.37 shows a typical state diagram used in dynamic branch prediction.

This state diagram allows backtracking of last two instructions in a given program. The branch target buffer entry contains the backtracking information which guides the prediction.

The prediction information is updated upon completion of the current branch.

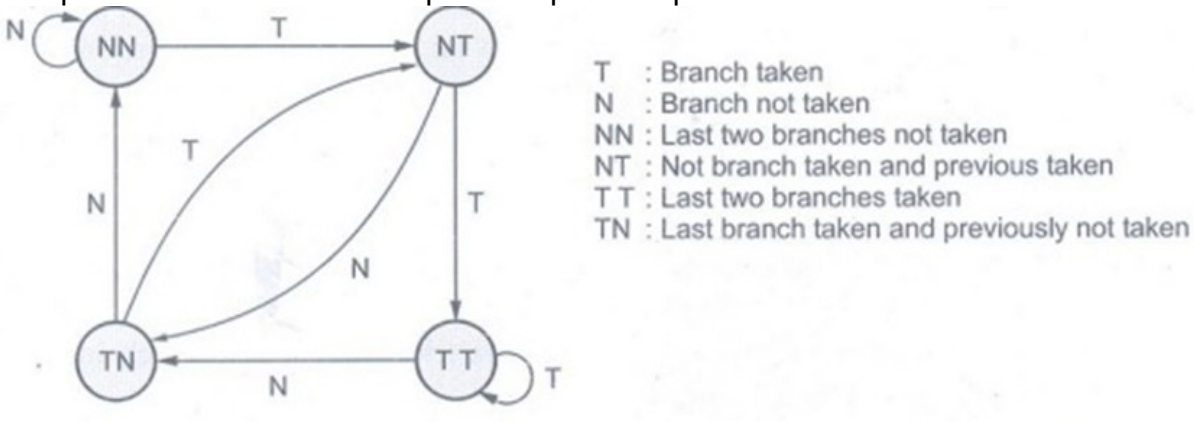


	Fig 3.37 A typical state diagram used in dynamic branch prediction •To make branch overhead zero, the branch target buffer is extended to store the target instruction itself and a few of its successor instructions. This allows processing of conditional branches with zero delay.
7	A pipelined processor uses the delayed branch technique. You are asked to recommend one of two possibilities for the design of the processor. In the first possibility, the processor has a 4-stage pipeline and one delay slot, and in the second possibility it has a 6-stage pipeline with two delay slots. Assume that 20% of the instructions are branch instructions and that an optimizing compiler succeeds in filling 80% of the single delay slot. For the second alternative, the compiler is able to fill the second slot 25% of the time. **1st Alternative: 4-stage Pipeline (1 Delay Slot)** ▪ **Branch penalty without delay slot optimization**: 1 cycle per branch. ▪ **Effective branch penalty**: ▪ 80% of the time, the slot is filled → no penalty. ▪ 20% of the time, the slot is wasted → 1 cycle penalty. $\text{Penalty per branch} = (0.8 \times 0) + (0.2 \times 1) = 0.2 \text{ cycles per branch}$ $\text{Overall penalty per instruction} = 0.2 \times 0.2 = 0.04 \text{ cycles per instruction}$ --- **2nd Alternative: 6-stage Pipeline (2 Delay Slots)** ▪ **Branch penalty without delay slot optimization**: 2 cycles per branch. ▪ **Effective branch penalty**: ▪ 80% of the time, the first slot is filled. ▪ 25% of the time, the second slot is also filled. ▪ Only in 20% of cases, the first slot is wasted. ▪ In $(1 - 0.25) = 75\%$ of cases, the second slot is wasted. $\text{Penalty per branch} = (0.8 \times 0) + (0.2 \times 1) + (0.75 \times 1) = 0.95 \text{ cycles per branch}$ $\text{Overall penalty per instruction} = 0.95 \times 0.2 = 0.19 \text{ cycles per instruction}$ --- **Comparison:** ▪ **4-stage pipeline penalty per instruction = 0.04 cycles.** ▪ **6-stage pipeline penalty per instruction = 0.19 cycles.** The **4-stage pipeline** is significantly better in terms of branch handling efficiency. The 6-stage pipeline suffers a higher penalty due to the additional delay slot, which is harder to fill. **Recommendation:** The **4-stage pipeline with one delay slot** is the better design choice because it incurs a lower branch penalty, leading to higher overall performance.
8	Find out the hazards in the following instructions and eliminate them by using stalls:

LW R_1 , 0(R_2)

SUB R_4 , R_1 , R_5

AND R_6 , R_1 , R_7

OR R_8 , R_1 , R_9

Identifying Hazards

Let's analyze the instruction sequence and identify potential hazards:

1. **LW R1, 0(R2)**
   - Loads a value from memory into register **R1**.
2. **SUB R4, R1, R5**
   - Uses **R1**, which is still being loaded from memory (Data Hazard).
3. **AND R6, R1, R7**
   - Again uses **R1**, which may not be available yet (Data Hazard).
4. **OR R8, R1, R9**
   - Also depends on **R1**, creating another Data Hazard.

Since the **LW** instruction has a latency of one or more cycles (depending on the pipeline design), the subsequent instructions using **R1** will face **Load-Use Data Hazards**. Without intervention, the **SUB, AND, and OR** instructions will attempt to read an unready value of **R1**.

To resolve these hazards, we insert **NOP (No Operation) instructions** (or stalls) to allow time for the **LW** instruction to complete.

Rewriting with Stalls

```
scss
CopyEdit
LW  R1, 0(R2)     // Load R1 from memory
NOP               // Stall to allow the load to complete
SUB R4, R1, R5    // Now R1 is ready
AND R6, R1, R7    // No additional stall needed (forwarding possible)
OR  R8, R1, R9    // No additional stall needed (forwarding possible)
```

Alternatively, if the pipeline requires two cycles for **LW** to complete before the next instruction can use the loaded value, we add an extra **NOP**:

```
LW  R1, 0(R2)
NOP
NOP
SUB R4, R1, R5
AND R6, R1, R7
OR  R8, R1, R9
```

This ensures **R1** is correctly available before being used.

Alternative: Forwarding with Minimal Stalls

If the processor supports **data forwarding**, then:

- The **SUB** instruction may still require a **single stall** if forwarding doesn't resolve the hazard.
- **AND** and **OR** may execute without stalls, as **R1** will be available after the **SUB**.

Thus, a **1-cycle stall** (one NOP) should be enough in some architectures:

```
LW  R1, 0(R2)
NOP
SUB R4, R1, R5
AND R6, R1, R7
OR  R8, R1, R9
```

7	Consider a system, which transfers 2MB file from memory to pen drive. i) If memory is using handshaking protocol to send the file, depict clearly how the data transfer takes place in case of source initiated and destination-initiated data transfer. (7) ii) When the file is being transferred there should be minimal intervention of the processor. Suggest a suitable technique for the above operation and explain it with proper justification and diagrams. (8)	CO 4	K3
8	Consider a cache of 256 blocks in size, each block has 2^4 words. The main memory size is 2^{12} blocks, each block has 2^4 words. How many bits are required for each of the TAG, SET/BLOCK and WORD FIELDS for different mapping techniques? Wherever needed assume that there are 8 ways in each set.	CO 4	K3