UNIT - III
THE PROCESSOR

3.1 INTRODUCTION

The key performance metrics of the computer systems are;

i. Instruction count: This depends on the compiler used and instruction set architecture.

ii. Clock cycle time: This depends on processor implementation.

iii. Clock cycles per instruction (CPI): This depends on processor implementation.

3.1.1 MIPS ARCHITECTURE

MIPS (Million Instructions Per Second) is a simple, streamlined, highly scalable RISC architecture with adopted by the industries.

The features that makes its widely useable are:

- Simple load and store with large number of register
- The number and the character of the instructions
- Better pipelining efficiency with visible pipeline delay slots
- Efficiency with compilers

These features make the MIPS architecture to deliver the highest performance with high levels of power efficiency. It is important to learn the architecture of MIPS to understand the detailed working of the processors.

Implementation of MIPS

IPS has 32 General purpose registers (GPR) or integer registers (64 bit) holding integer data. Floating point registers (FPR**)** are also available in MIPS capable of holding both single precision (32 bit) and double precision data (64 bit). The following are the data types available for MIPS:

Size	Name	Registers
8 bits	Byte	Integer register
bits	Half word	Integer register
bits	Word	Floating point register
bits	Double word	Floating point register

With these resources the MIPS performs the following operations:

- Memory referencing: load word (lw) and store word (sw)
- Arithmetic-logical instructions: add, sub, and, or, and slt
- Branch instructions: equal (beq) and jump (j)

i. Set the program counter (PC) to the address of the code and fetch the instruction from that memory.

ii. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, read only one register and for store word the processor has to operate on two registers.

The ALU operations are done and the result of the operation is stored in the destination register using store operation. When a branching operation is involved, then next address to be fetched must be changes based on the branch target.

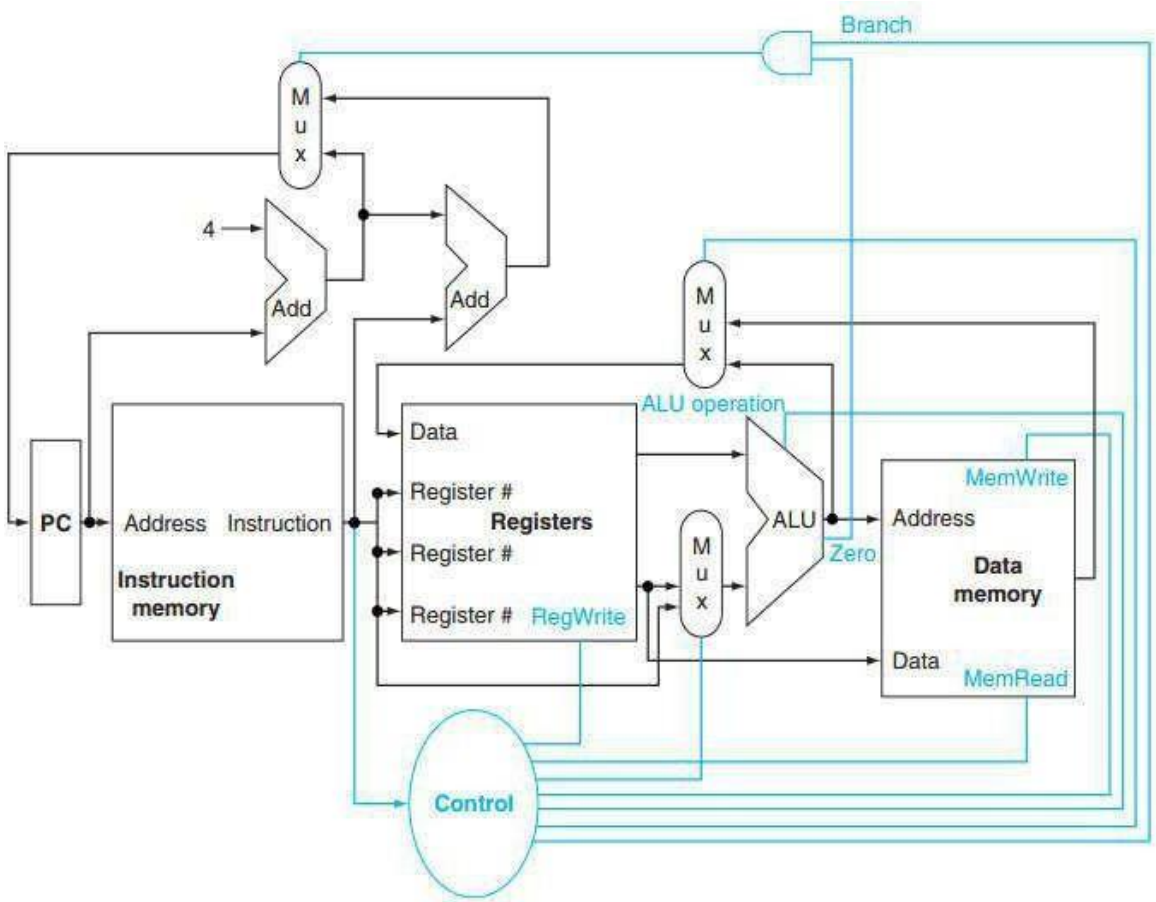


Fig 3.1: Implementation of MIPS architecture with multiplexers and control lines

Sequence of operations

- **Program Counter (PC):** This register contains the address (location) of the instruction currently getting executed. The PC is incremented to read the next instruction to be executed.
- The operands in the instruction are fetched from the registers.
- The ALU or branching operations are done. The results of the ALU operations are stored in registers. If the result is given in load and store forms, then the results are written to the memory address and from there they are transferred to the registers.
- In case of branch instructions, the result of the branch operation is used to determine the next instruction to be executed.

- The multiplexer (MUX1), selects one input control line from multiple inputs. This acts as a **data selector.**
- This helps to control several units depending on the type of instruction.
- The top multiplexor controls the target value of the PC. To execute next instruction the PC is set as PC+4. To execute a branch instruction set the PC to the branch target address.
- The multiplexor is controlled by the AND gate that operates on the zero output of the ALU and a control signal that indicates that the instruction is a branch.
- The multiplexor (MUX2) returns the output to the register file for loading the resultant data of ALU operation into the registers.
- MUX3 determines whether the second ALU input is from the registers or from the offset field of the instruction.
- The control lines determine the operation performed at the ALU. The control lines decide whether to read or write the data.

MIPS instruction format

There are only three instruction formats in MIPS. The instructions belong to any one of the following type:

- Arithmetic/logical/shift/comparison
- Control instructions (branch and jump)
- Load/store
- Other (exception, register movement to/from GP registers, etc.)

All the instructions are encoded in one of the following three formats:

I type: Load and store instructions

Op code	Rs	Rt	Immediate

R-type: Register to register operations

Op code	Rs	Rt	Rd	Shamt	Funct

J-Type: Jump instructions

Op code	Offset

The data and memory are well separated in MIPS implementation because:

- The instruction formats for the operations are not unique; hence the memory access will also be different.
- Maintaining separate memory area is less expensive.
- The operations of the processor are performed in single cycle. A single memory (for both data and memory access) will not allow for two different accesses within one cycle.

3.2 LOGIC DESIGN CONVENTIONS

The information in a computer system is encoded in binary form (0 or 1). The high voltage is encoded as 1 and low voltage as 0. The data is transmitted inside the processors through control wires / lines. These lines are capable of carrying only one bit at a time. So transfer of multiple data can be done through deploying multiple control lines or buses. The data should be synchronized with time by transferring it according to the clock pulses. All the internal operations inside the processor are implemented through logic elements. The logic elements are broadly classified into: **Combinatorial and Sequential elements.**

Differences between Combinatorial and Sequential elements

Combinatorial Elements	Sequential Elements
The output of the combinatorial circuit depends only on the current input.	The output depends on the previous stage outputs.
It has faster operation speed and easy implementation.	It has comparatively low operation speed I and tough implementation.
No feedback connections.	The output is connected with the input through feedback connections.
For a given set of inputs, combinatorial elements give the same output since there is no storage of past data.	The outputs vary based on previous outputs.

The basic building blocks are gates, which are time independent.	The basic building blocks are flip flops, which are time dependent.
It is used for Arithmetic and Logic operations.	It is used for data storage.
No need for trigger.	Triggering is needed to control the clock cycles.
No memory element.	Memory element is needed which is used to store the states.
Eg: Encoder, full adder, Decoder, Multiplier	**Eg:** Counters

Importance of state elements

The state elements characterize the machines. The contain state or status values so that the machine can be restored with the previous values by retaining the values in the state element. A state element has at least two inputs and one output. The required inputs are the data value to be written into the element and the clock, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. The following are the state elements in Fig 3.1: instructions, memories and registers.

3.2.1 Clocking Methodology

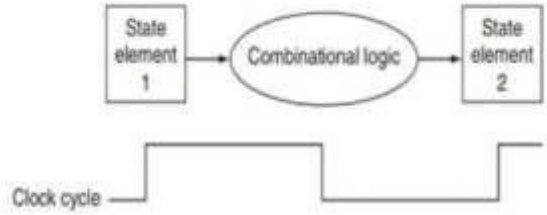


Fig 3.2 a: Combinatorial Logic

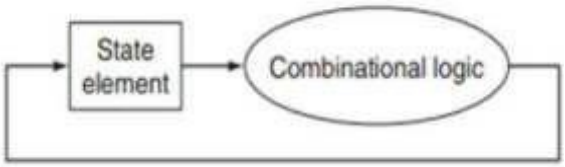


Fig 3.2 b: Edge triggered Logic

A clocking methodology is a set of rules for interconnecting components and clock signals that, when followed, guarantee proper operation of the resulting system.

The primary objective of clocking methodology is timing correlation.

Edge triggered clocking methodology

This allows the processor to read the register contents, send the value through some combinatorial logic and write that register in same clock cycle under the assumption that the state elements are controlled by implicit clock cycles.

- Here, the stored values are updated only on a clock edge.
- In combinatorial logic, the input must be read, processed and the output must be sent to the location, all in one single clock cycle (Fig 3.2 a).
- The driving force of this combinatorial circuit will be an explicit control signal.
- All the changes occur only when the clock signal is triggered.
- In edge-triggered methodology, the contents of a register are read and the value is sent through combinational logic, and written to that register in the same clock cycle.
- This prevents the access of inconsistent intermediate data
- Feedback cannot occur within 1 clock cycle because of the edge-triggered update of the state element.
- The clock cycle still must be long enough so that the input values are stable when the active clock edge occurs.

3.3 BUILDING A DATAPATH

A data path is a representation of the flow of information (data and instructions) through the CPU, implemented using combinatorial and sequential circuitry.

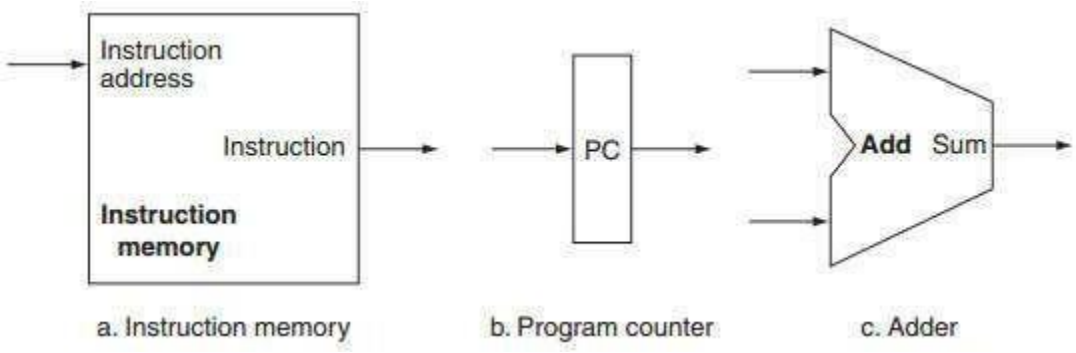


Fig: 3.3 Components of Data path

Data path is a functional unit that operates or hold data. In the MIPS implementation the data path elements includes instruction and data memories, the register file, the arithmetic logic unit (ALU), and adders. The functionalities of basic elements are listed below:

- **Instruction Memory:** It is a state element that provides read access because the data path do not perform write operation. This combinatorial memory always holds contents of location specified by the address.
- **Program Counter (PC):** This is a 32 bit state register containing the address of the current instruction that is being executed. It is updated after every clock cycle and do not require an explicit write signal.
- **Adder:** This is a combinatorial circuit that updates the value of PC after every clock cycle to get that address of the next instruction to be executed.

3.3.1 Instruction Fetch:

The fundamental operation in Instruction Fetch is to send the address in the PC to the instruction memory and obtain the specified instruction, and the increment the PC.

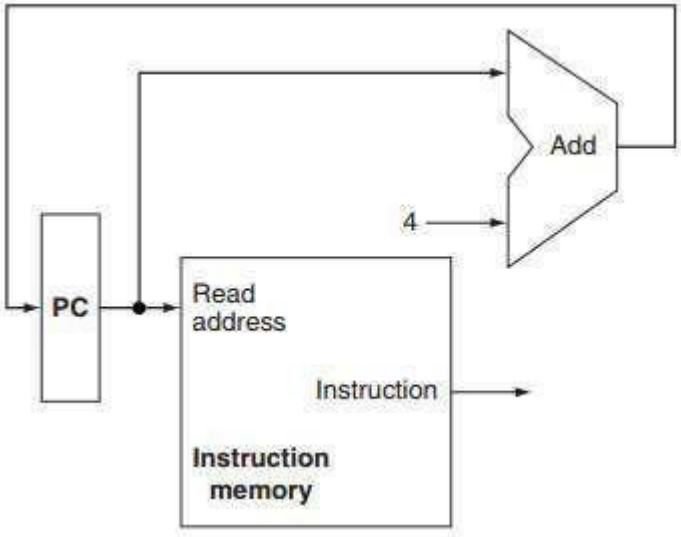


Fig: 3.4: Instruction Fetch

R type instructions:

- They all read two registers, perform an ALU operation on the contents of the registers and write the result.
- This instruction class includes add, sub, and, or, and slt.
- 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 machine.
- The R-format always performs ALU operation that has three register operands (2-read and 1-write).
- The register number must be specified in order to read the data from the register file. Also the output from a register file will contain the data that is read from the register.
- The write operation to a register has two inputs: the register number and the value to be written. This operation is edge triggered.

Load and Store instructions:

- The load and store instructions compute a memory address by adding the base register.
- If the instruction is a load, the value read from memory must be written into the register file in the specified register.
- The memory is computed by adding the address of base register and the16-bit signed offset field (which is a part of the instruction).
- If the instruction is a store, the value to be stored must also be read from the register.

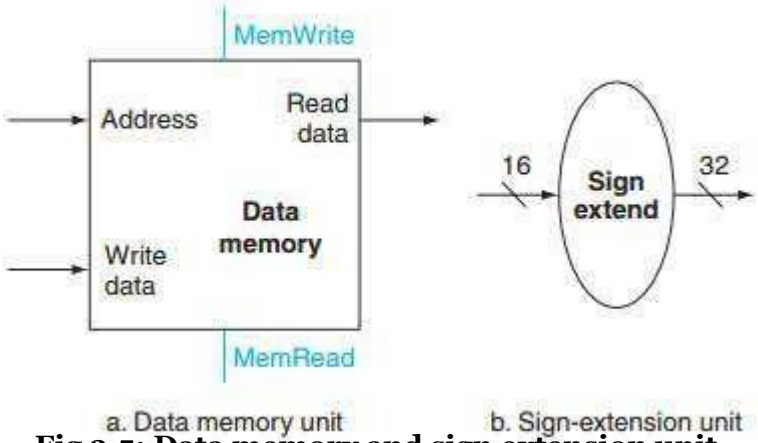


Fig 3.5: Data memory and sign extension unit

- The processor has a sign extension unit to **sign-extend** the 16-bit offset field in the instruction to a 32-bit signed value.
- The data memory unit is necessary to perform write operation of store instruction. So it has both read and write control signals, an address input and data input.

Branch Instructions:

Branch Target is the address specified in a branch, which is used to update the PC if the branch is taken. In the MIPS architecture the branch target is computed as the sum of the offset field of the instruction and the address of the instruction following the branch.

J. The beq instruction (branch instruction) has three operands, two registers that are compared for equality, and a 16-bit offset to compute the branch target address. **beq t1, t2, offset**

K. Thus, the branch data path must do two operations: compute the branch target address and compare the register contents.

L. **Branch Taken** is where the branch condition is satisfied and the program counter (PC) loads the branch target. All unconditional branches are taken branches.

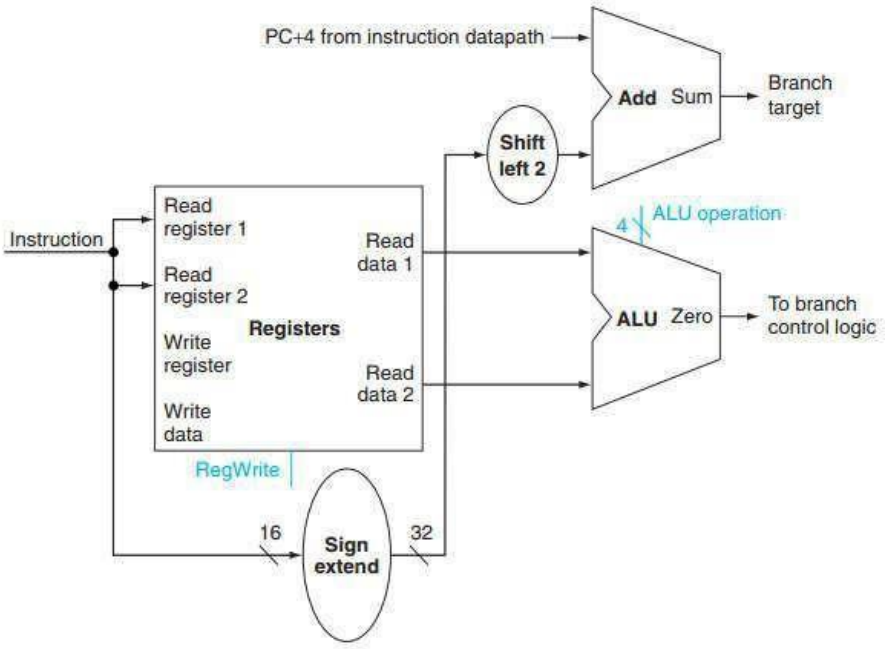
M. **Branch not Taken is** where the branch condition is false and the program counter (PC) loads the address of the instruction that sequentially follows the branch.

N. The branch target is calculated by taking the address of the net instruction after the branch instruction, since the PC value will be updated as PC+4 even before the branch decision is taken

O. The offset field is shifted left 2 bits to increase the effective range of the offset field by a factor of four.

Fig 3. 6: Data path of branch Instructions

JJ. The unit labelled Shift left 2 adds two zeros to the low-order end of the sign-extended offset



field. This operation truncated the sign values.

KK. The control logic decides whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

LL. The jump instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits. This shift is done by concatenating 00 to the jump offset.

MM. **Delayed branch** is where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

NN. MIPS architecture implements delayed branch (i.e.) the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

JJJ. When the condition is false, the execution looks like a normal branch.

KKK. When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address.

KKK. Delayed branches facilitate pipelining.

3.3.2 Creating a single Data path

- A simple implementation of a single data path is to execute all operations within one clock cycle.
- The data path resources can be utilized only for one clock cycle. To facilitate this, some resources must be duplicated for simultaneous access while other resources will be shared.
- One example is having separate memory for instructions and memory.
- When a resource is used in shared mode, then multiple connections must be made. The selection of which control will access the resource will be decided by a multiplexer.

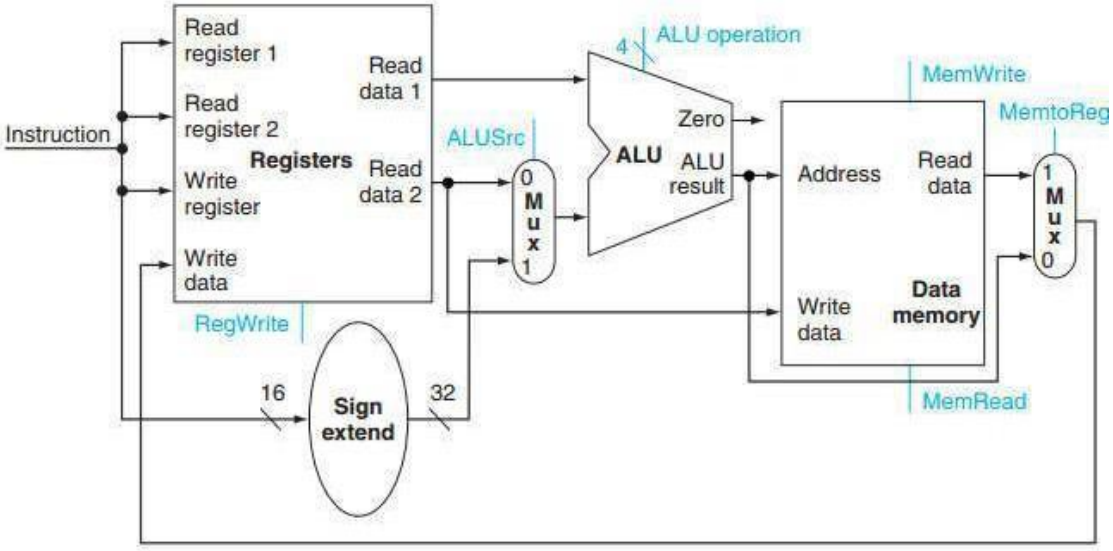


Fig: 3.7: Simple data path

- The data path illustrated in Fig 3.7 shows the assembling of individual elements into a simple data path.

- To implement branch instructions the data path must include an adder circuitry to compute branch target (Refer Fig: 3.6).
- The control unit for this data path must take inputs and generate a write signal for each state element. Apart from the inputs a selector control must be included for each multiplexor and the ALU control.
- The operations of arithmetic-logical (or R-type) instructions and the memory instructions data path are almost similar.
- The arithmetic-logical instructions use the ALU with the inputs coming from the two registers. The memory instructions can also use the ALU to do the address calculation, but the second input is the sign-extended 16-bit offset field from the instruction.

3.4 SIMPLE IMPLEMENTATION SCHEME

The basic implementation 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).

For any instruction, the following two steps are same:

- Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
- Read one or two registers, using fields of the instruction to select the registers to read.

Load instruction needs to read only one register, but most other instructions require reading two registers. The remaining actions required to complete the instruction depend on the instruction class. For the three instruction classes namely memory-reference, arithmetic-logical, and branches, the actions are mostly the same. This is due to the simplicity and regularity of the MIPS instruction set.

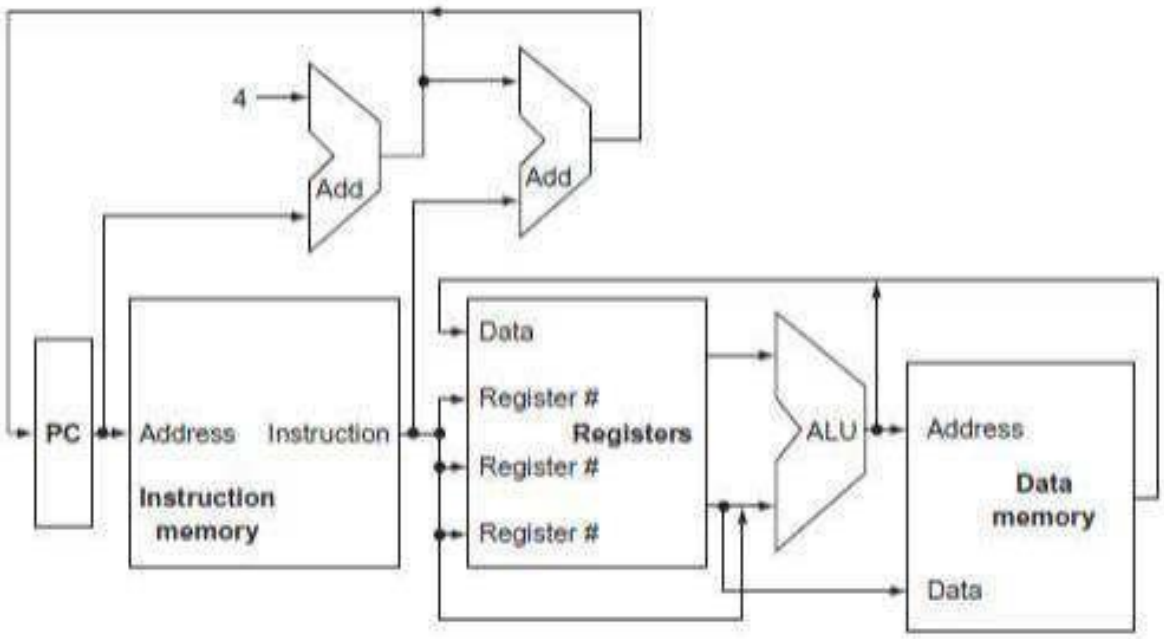


Fig 3.8: An abstract view of MIPS implementation Instruction Formats of MIPS

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

Fig 3.9: R-Format Instruction

Instruction format for R-format instructions have an op code of 0. These instructions have three register operands: sources: rs, rt, and destination: rd. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. This instruction type is used to implement are add, sub, and, or, and slt. The shamt field is for shifting operation.

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

Fig 3.10: Load or store instruction

Instruction format for load specified by op code = 35ten and store is specified by op code 43ten) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory.

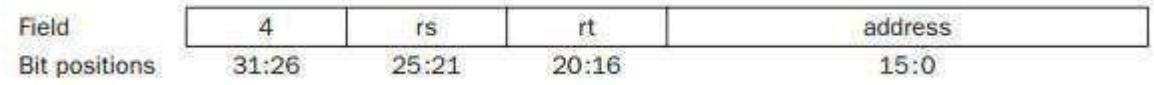


Fig 3.11: Branch Instructions

Instruction format for branch equal (op code = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign extended, shifted, and added to the PC to compute the branch target address.

- All instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.
- The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and branches for comparison.
- After using the ALU, the actions required to complete various instruction classes differ.
- A memory-reference instruction will need 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.
- Branch instruction 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.
- All instructions start by using the program counter to supply the instruction address to the instruction memory.
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction.

- Once the register operands have been fetched, they can be operated on 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 off set are summed) or from an adder that increments the current PC by 4.
- The thick lines interconnecting the functional units represent buses, which consist of multiple signals.
- Fig.3.12shows the data path of Fig 3.8 with the three required multiplexors added, and control lines for the major functional units.
- A control unit, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors.
- The third multiplexor, which determines whether PC + 4 or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a beq instruction.

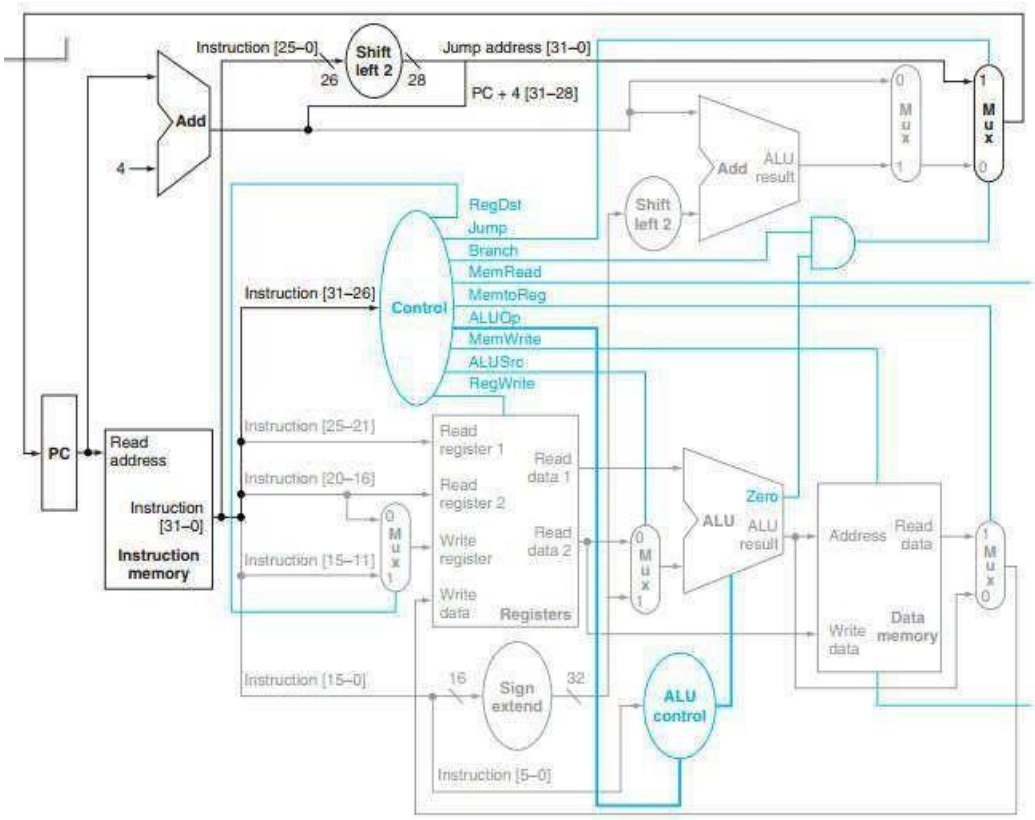


Fig 3.12: Implementation scheme with control lines

Operation of the Data path given in Fig 3.12:

Four steps to execute the instruction; these steps are ordered by the flow of information:

- The instruction is fetched, and the PC is incremented.
- Two registers, $t2 and $t3, are read from the register file. the main control unit computes the setting of the control lines during this step.
- The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.
- The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

Effect of the control signals

Signal	When deasserted	When asserted
RegDst	The register destination number for Write register comens from the rt field (bits 20:16)	The register destination number for Write register comes from the rd field (bits 15:11)
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second operand comes from the second register file output (Read data 2).	The second operand is the sign extended lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder after computing PC+4	The PC is replaced by the output of the adder after computing the branch target.
MemRead	None	Data memory contents designated by the address input are placed on the Read data input.
MemtoReg	The value given to Write data input is got from the ALU.	The value given to Write data input is got from the data memory.

The setting of the control lines is completely determined by the op code fields of the instruction as given below:

Instruc-tion	Reg Dst	ALU Src	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op1	ALU Opo
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

Finalizing the Controls

The logic values for a comprehensive control unit can be expressed as a single large truth table. This table combines all the outputs and uses the op code bits as inputs. It completely specifies the control function.

Input / Output	Signal	R-format Name	Lw	Sw	Beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	X
	RegWrite	1	1	0	0
		0		0	
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

3.5 PIPELINING

Pipelining is an implementation technique in which multiple instructions are executed simultaneously by overlapping them in execution to save time and resource. The previous instruction will be in the execution phase when the current instruction is fetched from the memory.

Need for Pipelining

Without a pipeline, a computer processor fetches the first instruction from memory, performs the operation mentioned in it, and then goes to fetch the next instruction from memory. While fetching the instruction, the arithmetic unit of the processor is idle. It must wait until it is loaded with next instruction.

With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor. The result is an increase in the number of instructions that can be performed during a given time period.

3.5.1 Stages in MIPS pipelining:

The following are the various stages in pipelining:

- **Instruction Fetch (IF):** Fetch instruction from memory.
- **Instruction Decode (RD):** Read registers while decoding the instruction. The format of MIPS instructions allows reading and decoding to occur simultaneously.
- **Execute:** Execute the operation or calculate an address. This involves ALU operations.
- **Memory access (MEM):** Access an operand in data memory.
- **Write Back (WB):** Write the result into a register.

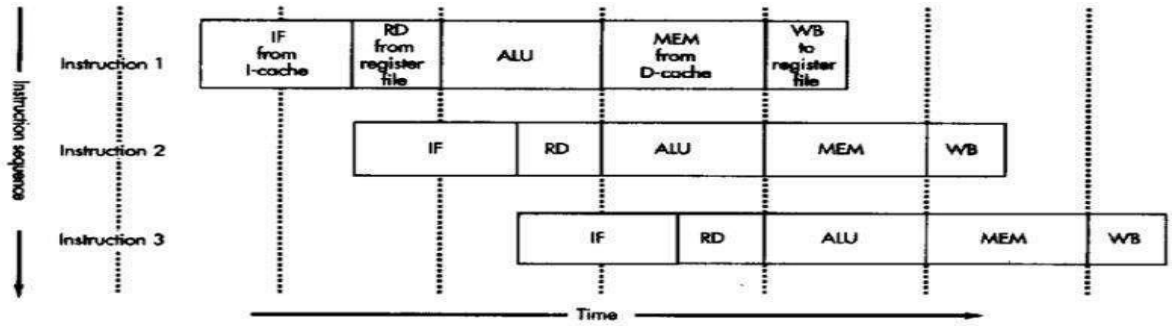


Fig 3.13: 5 stage pipelining of MIPS architecture

The pipelining speed can be manipulated using the expression:

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Pipelining improves performance by increasing instruction throughput. It is not decreasing the execution time of an individual instruction, but increases the number of instructions that complete its execution for a given time period. Thus the overall performance of the processor is improved both in terms of resource utilization and throughput.

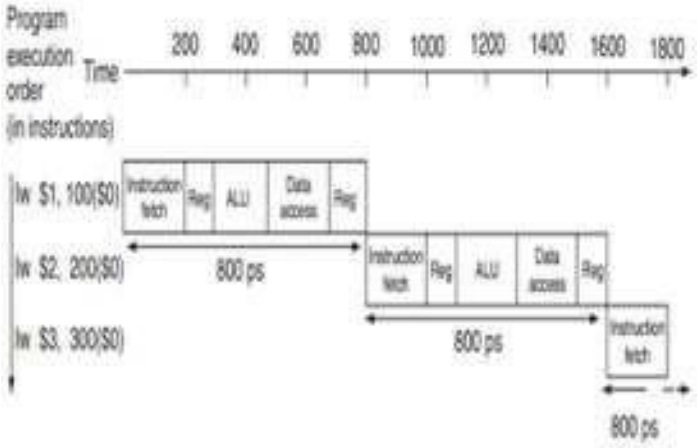


Fig 3.14 a) Non pipelined Execution

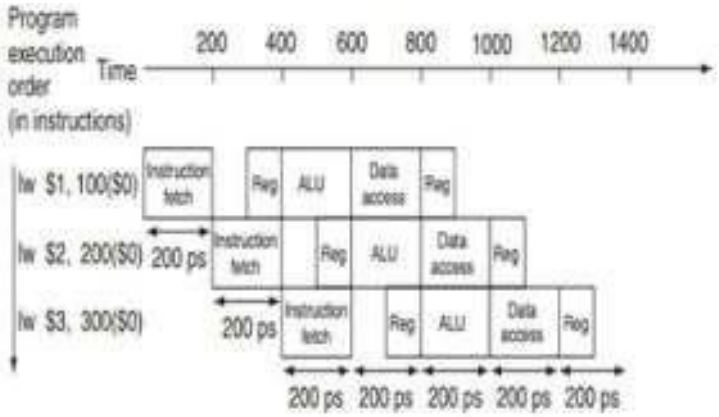


Fig 3.14 b) Pipelined Execution

Fig 3.14 shows the comparison of execution of instructions with and without pipelining on same hardware components. The timeline clearly indicates that there is a difference in execution time and resource utilization. The challenges in implementing pipelining may arise due to slowest resource.

3.5.2 Designing instruction sets for Pipelining

- The simplicity and generality of MIPS instructions are that they are of same length. This facilitates easy instruction fetching in the first stage of pipelining.
- MIPS has only a few instruction formats. In every instruction format, the source operand register is located at the same position in the instruction format.
- This symmetry eases the instruction decode stage by reading the register file simultaneously while the hardware is determining the type of instruction format.
- Also, the memory operands appear in only in load or store instruction type in MIPS. So that the execute stage can calculate the memory address and then access memory in the following stage.
- Operands must be aligned in memory. Hence, a single data transfer instruction requiring two data memory accesses can be done in a single pipeline stage.

3.5.3 Hazards in Pipelining

Hazards are situations that prevent the next instruction in the instruction cycle from being executing during its designated clock cycle. Hazards reduce the performance of the pipelining.

They are attempt to use same resource by two or more instruction at the same time.

Example: In case of single memory is used for instructions and data access and when two instructions are accessing the same register one at instruction fetch stage and other at memory access stage. This leads to inconsistent data access.

Types of hazard:

Structural Hazards: They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

- **Data Hazards:** They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- **Control Hazards**: They arise from the pipelining of branches and other instructions that change the PC. This is also known as **branch hazard.** The flow of instruction addresses is not what the pipeline had expected. This results in control hazard.

3.5.4 Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.

Data hazards occur in register files due to inconsistencies in file. This is an occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. In other words, data hazards occur when the pipeline must be stalled because one step must wait for another to complete. This is due to the data dependence.

Example: Consider the following instructions:

add $s0, $t0, $t1

sub $t2, $s0, $t3

Here the sub instruction uses the result of add instruction ($s0). The add instruction cannot not write its result until the fifth stage. This results in wasting three clock cycles in the pipeline. Since the stall occurs due to the non availability of data, this is termed as data hazards.

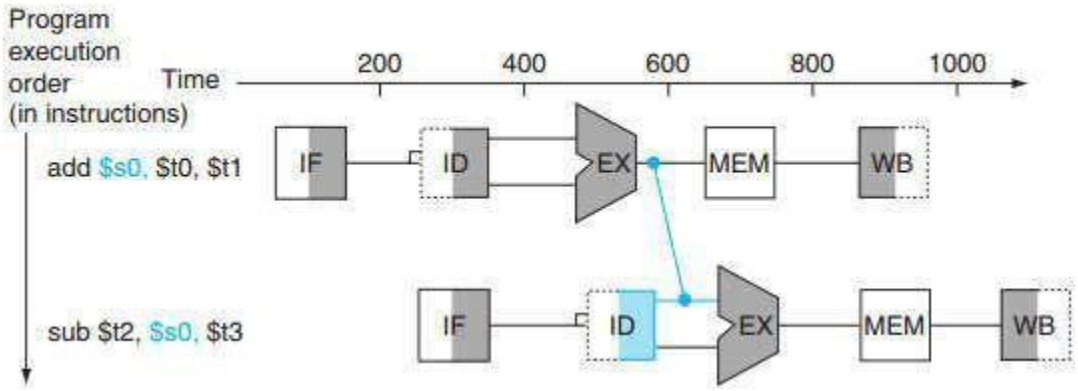


Fig 3.15: Data Hazard

Solution to resolve data hazard:

Forwarding or bypassing is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory. This can be done by adding extra memory element or hardware that acts as an internal buffer.

Forwarding cannot be a universal solution to solve data hazards. Consider the following instructions:

lw $s0, 20($t1)

sub $t2, $s0, $t3

The desired data would be available only after the fourth stage of the first instruction in the dependence, which is too late for the input of the third stage of sub. Hence, even with forwarding, there will be a hazard called as **load-use data hazard**.

A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested. This is Load-use data hazard.

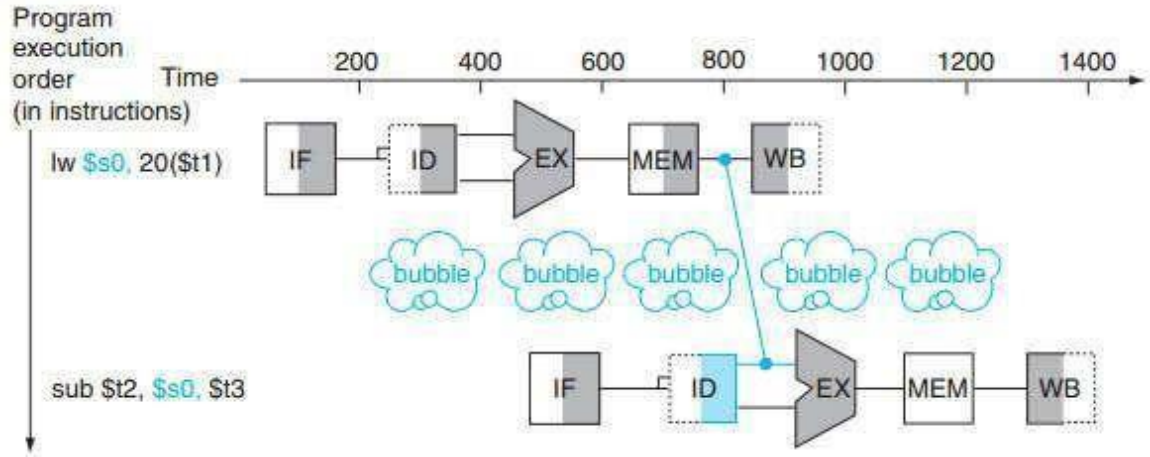


Fig 3.16: Load-Use data hazard

The stall mentioned in Fig 3.16 is called **bubble or pipeline stall.** A pipeline stall is a delay in execution of an instruction in order to resolve a hazard. During the decoding stage, the control unit will determine if the decoded instruction reads from a register that the instruction currently in the execution stage writes to.

Problem 3.1

Find the hazards in the following code segment and reorder the instructions to avoid any pipeline stalls.

```
lw $t1, 0($t0)
lw $t2, 4($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
lw $t4, 8($01)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

Solution:

Both the add instructions have a hazard because of their dependence on the immediately preceding lw instruction. Bypassing eliminates several other potential hazards including the dependence of the first add on the first lw and any hazards for store instructions. Moving up the third lw instruction eliminates both hazards. This is possible since the lw instruction is independent of other operations:

```
lw $t1, 0($t0)
lw $t2, 4($t1)
lw $t4, 8($01)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

3.6 A PIPELINED DATAPATH

The stages of pipelined data path are:

- IF: Instruction fetch
- ID: Instruction decode and register file read

- EX: Execution or address calculation
- MEM: Data memory access
- WB: Write back

The two exceptions to the normal flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the data path.
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

  Data flowing from right to left does not affect the current instruction; only later instructions in the pipeline are influenced by these reverse data movements. Note that the first right-to-left arrow can lead to data hazards and the second leads to control hazards. One way to show what happens in pipelined execution is to pretend that each instruction has its own data path, and then to place these data paths on a time line to show their relationship.

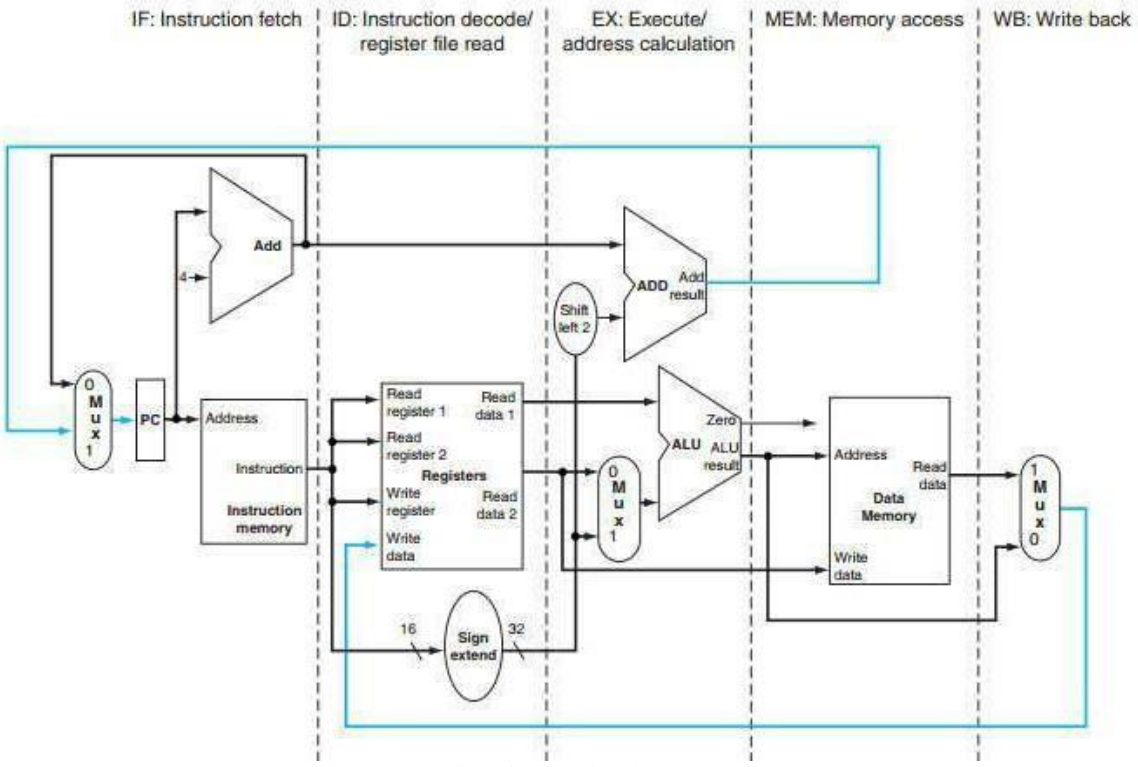


Fig 3.17: Single cycle data path

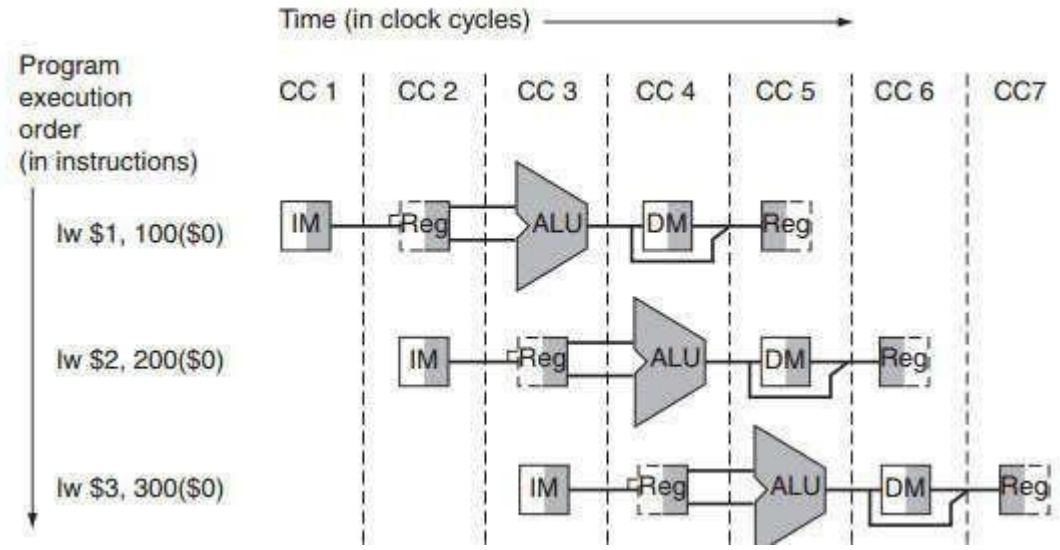


Fig 3.18: Instructions in single cycle data path

- The above fig shows that each instruction has its own data path, and each stage is labeled by the physical resource used in that stage, corresponding to the portions of the data path.
- IM represents the instruction memory and the PC in the instruction fetch stage, Reg stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on.
- To maintain proper time order, the data path breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB).
- This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read.
- As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

Operations in each stage of Pipeline:

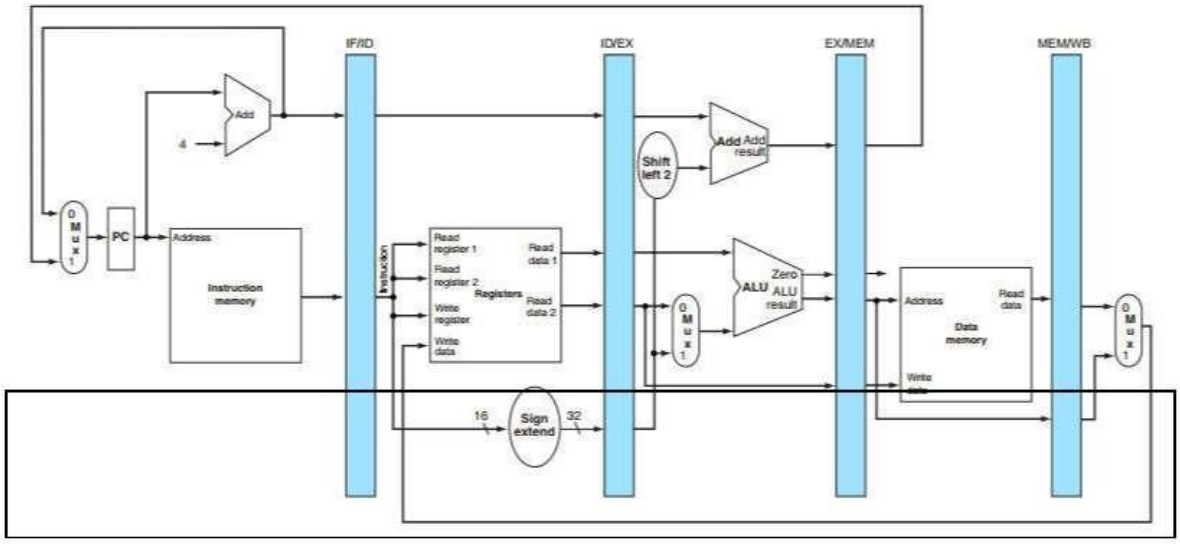


Fig 3.19: Five stages of Pipeline

Instruction fetch:

The instruction is read from memory using the address in the PC and then placed in the IF/ID pipeline register.

The IF/ID pipeline register is similar to the Instruction register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle.

This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq.

The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

Instruction decode and register file read:

The instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers.

All three values are stored in the ID/EX pipeline register, along with the incremented PC address.

Transfer everything that might be needed by any instruction during a later clock cycle.

These first two stages are executed by all instructions, since it is too early to know the type of the instruction.

- **Execute or address calculation:**

  The load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU.

  That sum is placed in the EX/MEM pipeline register.

- **Memory access:**

  The load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

  The register containing the data to be stored was read in an earlier stage and stored in ID/EX.

  The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

- **Write back:**

  This involves reading the data from the MEM/WB pipeline register and writing it into the register file.

3.7 PIPELINED CONTROL

This section describes the necessary control lines for implementing a pipelined data path. The control logic is needed for PC source, register destination number, and ALU control. A 6-bit funct field (function code) is needed for the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. These 6 bits are 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 these bits unchanged.

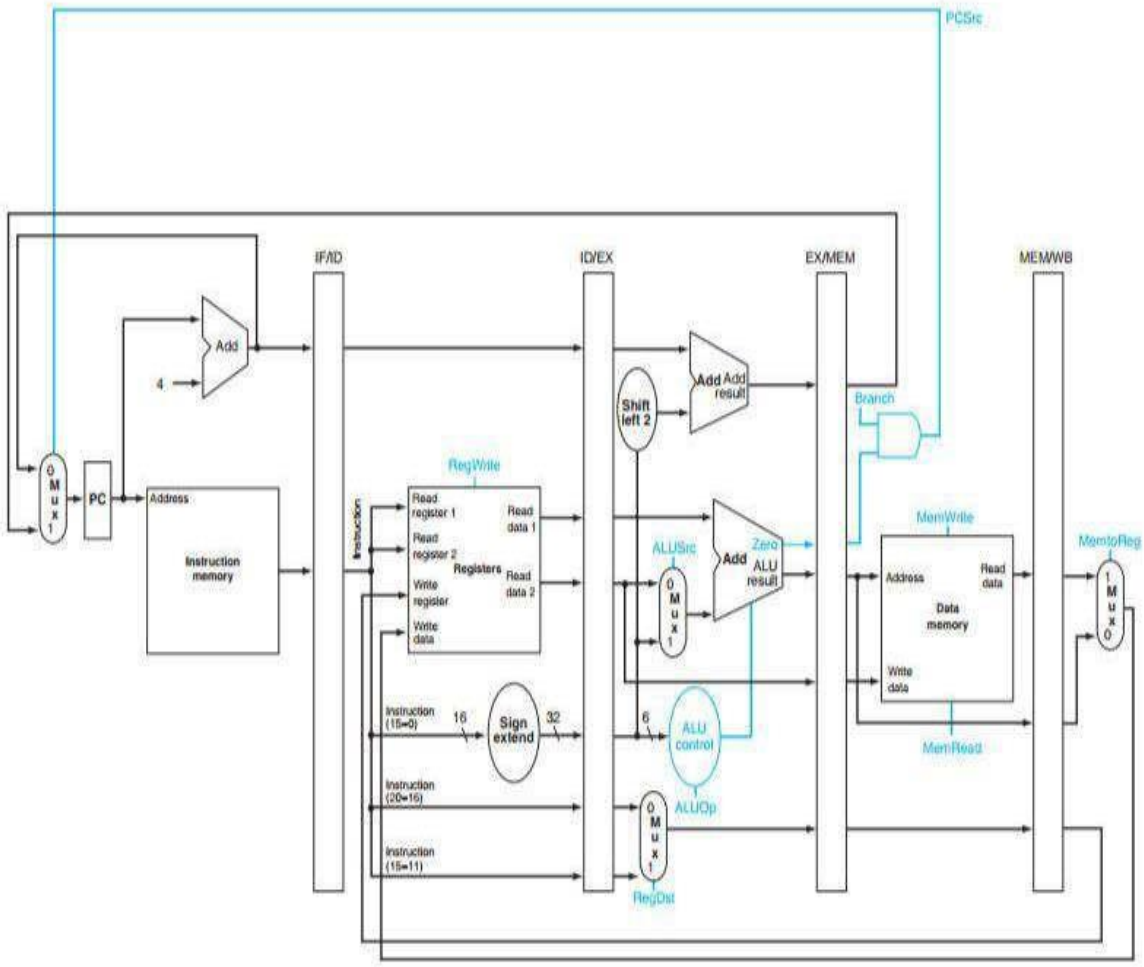


Fig 3.20: Control signals in single cycled data path

Sequence of operations:

- he PC is written on each clock cycle, so there is no separate write signal for the PC.
- There are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/ MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.
- To specify control for the pipeline, set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage.
- The control lines are also divided into five groups according to the pipeline stage:

- **Instruction fetch:** The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
- **Instruction decode/register file read:** As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
- **Execution/address calculation**: The signals to be set are Reg Dst, ALU Op, and ALU Src. The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.
- **Memory access:** The control lines set in this stage are Branch, Mem Read, and Mem Write. These signals are set by the branch equal, load, and store instructions, respectively.
- **Write back:** The two control lines are Mem to Reg, which decides between sending the ALU result or the memory value to the register file, and Reg Write, which writes the chosen value.

Implementing control means setting the nine control lines to these values in each stage for each instruction (explained in simple implementation scheme). The simplest way to do this is to extend the pipeline registers to include control information.

3.8 DATA HAZARDS

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.

Data hazards occur in register files due to inconsistencies in file. This is an occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available. In other words, data hazards occur when the pipeline must be stalled because one step must wait for another to complete. This is due to the data dependence.

3.8.1 Forwarding or Bypassing

Forwarding or bypassing is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer visible registers or memory. This can be done by adding extra memory element or hardware that acts as an internal buffer.

Forwarding cannot be a universal solution to solve data hazards. Consider the following instructions:

lw $s0, 20($t1)

sub $t2, $s0, $t3

The desired data would be available only after the fourth stage of the first instruction in the dependence, which is too late for the input of the third stage of sub. Hence, even with forwarding, there will be a hazard called as **load-use data hazard**.

A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested. This is Load-use data hazard.

Consider the following code:

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)

There are several dependences in this code fragment:

- The first instruction, SUB, stores a value into $2.
- That register is used as a source in the rest of the instructions this is no problem for 1-cycle and multi cycle data path.
- Each instruction executes completely before the next begins.
- This ensures that instructions 2 through 5 above use the new value of $2.

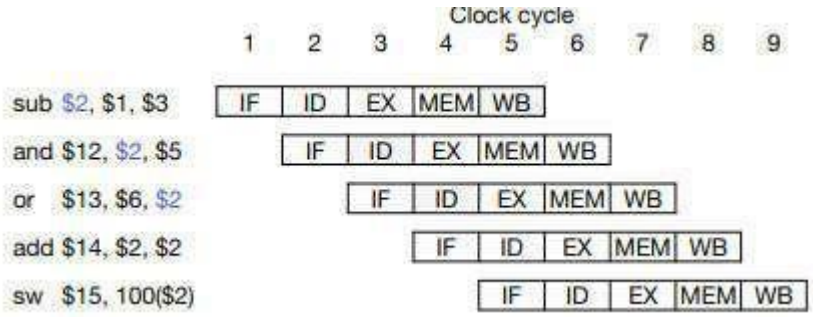


Fig 3.21: Pipelined diagram

- The SUB does not write to register $2 until clock cycle 5 causing 2 data hazards in our pipelined data path.
- The AND reads register $2 in cycle 4. Since SUB hasn't modified the register yet, this is the old value of $2
- The OR instruction uses register $2 in cycle 5, again before it's actually updated by SUB.

To avoid data hazard, rewrite the instructions (sll means stall):

sub $2, $1, $3

sll $0, $0, $0

sll $0, $0, $0

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)

Since it takes two instruction cycles to get the value stored, one solution is for the assembler to insert no-ops or for compilers to reorder instructions to do useful work while the pipeline proceeds. Since the pipeline registers already contain the ALU result, we could just forward the value to later instructions, to prevent data hazards

- In clock cycle 4, the AND instruction can get the value of $1 - $3 from the EX/MEM pipeline register used by SUB.
- Then in cycle 5, the OR can get that same result from the MEM/WB pipeline register being used by SUB.

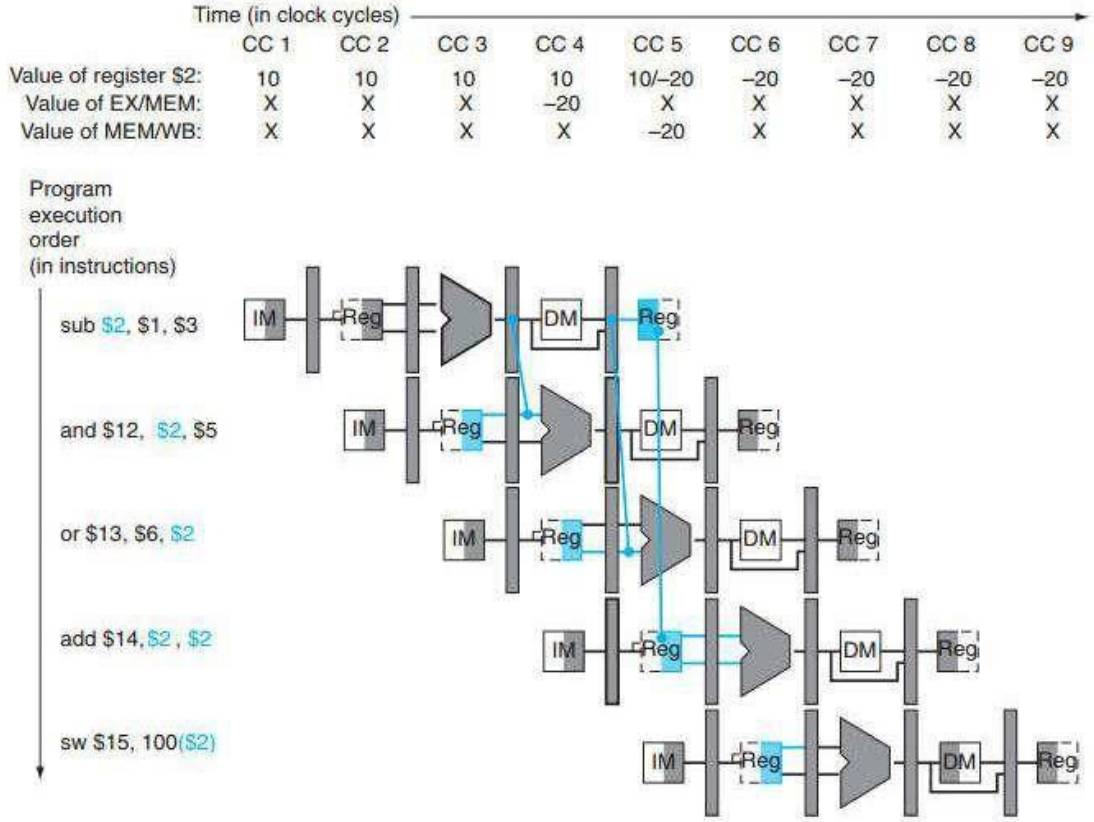


Fig 3.22: Pipelined dependencies

Forward the data as soon as it is available to any units that need it before it is available to read from the register file. This is forwarding in data hazards.

When an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU. Te general format for specifying dependencies is given by:

Pipeline register. Field in the register

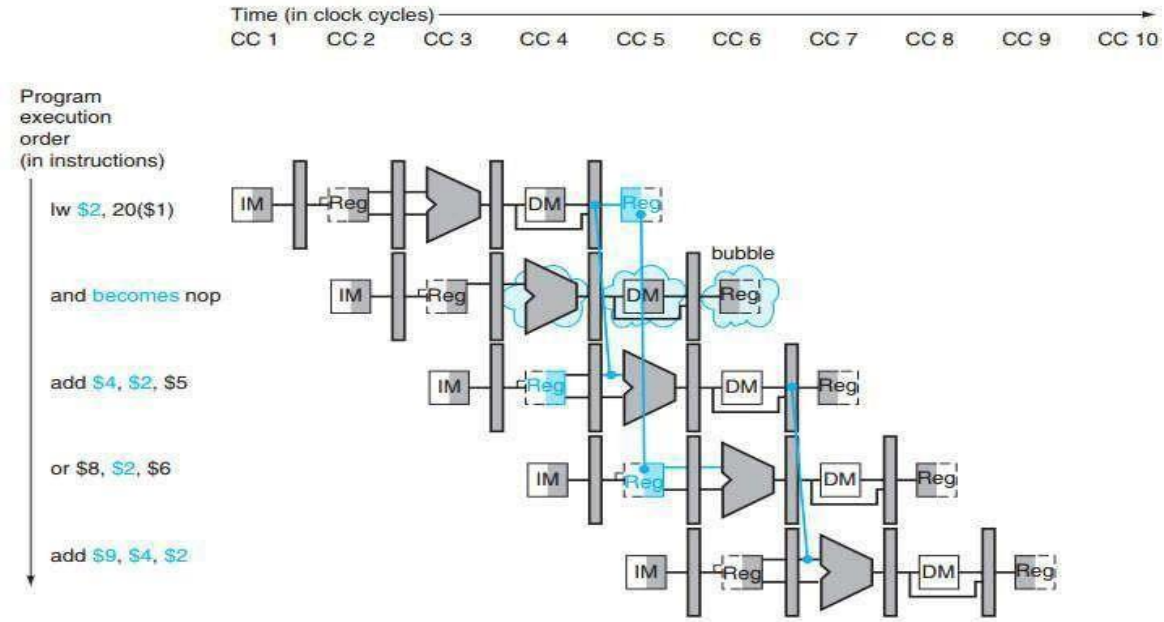
Example: ID/EX .Register Rs- refers that the value is found in the pipeline register ID/EX in the field Register Rs. The dependencies in the given example are:

- EX/MEM .Register Rd = ID/EX .Register Rs
- EX/MEM. Register Rd = ID/EX .Register Rt
- MEM/WB. Register Rd = ID/EX .Register Rs
- MEM/WB .Register Rd = ID/EX .Register Rt

The first hazard in the sequence is on register $2, between the result of sub $2,$1,$3 and the first read operand of and $12,$2,$5. This hazard can be detected when the AND instruction is in the EX stage and the prior instruction is in the MEM stage.

EX/MEM .Register Rd = ID/EX .Register Rs = $2

Forwarding the inputs to the ALU from any pipeline registers done by adding multiplexors to the input of the ALU and with the proper controls. By this the pipeline can be executed at full speed in the presence of these data dependences.



3.8.2 Stalling

Fig 3.23: Introducing stalls in pipelining

A bubble is inserted beginning in clock cycle 4, by changing the AND instruction to a nop (no operation). Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5. The or instruction is fetched in clock cycle 3, but its IF stage is delayed until clock cycle 5. After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

In short forwarding requires:

- Recognizing when a potential data hazard exists, and
- Revising the pipeline to introduce forwarding paths.

3.9 CONTROL HAZARDS

This occurs when there is a need for an instruction to take a decision based on the results of another instruction's result that has no yet completed its execution.

Control or branching hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

Instructions that disrupt the sequential flow of control present problems for pipelines are potential candidates for control hazards. The effects of these instructions cannot be exactly determined until late in the pipeline, so instruction fetch cannot continue unless it is explicitly managed. The following types of instructions can introduce control hazards:

- Unconditional branches
- Conditional branches
- Indirect branches
- Procedure calls
- Procedure returns

Example:

```
        ld      r2, 0(r4)       // r2 := memory at r4
        ld      r3, 4(r4)       // r3 := memory at r4+4
        sub     r1, r2, r3      // r1 := r2 - r3
        beqz    r1, L1          // if r1 is not 0, goto L1
        ldi     r1, 1           // r1 := 1
L1:     not     r1, r1          // r1 := not r1
        st      r1, 0(r5)       // store r1 to memory at r5
```

This code compares two memory locations and stores the result of that comparison (1 for equal, 0 for not equal) to another location. If the beqz branch is taken, then a 1 is stored; otherwise, a 0 is stored. The beqz instruction sources two hazards:

- When the beqz instruction is in the decode stage, the sub instruction is in the execute stage. The branch cannot read the output of the sub until it has been written to the register file; if it reads it early, it will read the wrong value.
- The instruction that is to be fetched after beqz is not known in advance. At this point, the status of the branch instruction is totally unknown whether it depends on the previous instruction or not. This is because it hasn't been de coded yet, so bypassing also can't help in resolving the hazard. Even if the decision is known, the location from where to fetch the instruction if the branch is taken is unknown because the effective address computation for branches do not happen until the EX stage.

Solutions for control hazards:

The following are solutions that can reduce control hazards:

- **Pipeline stall cycles:** Freeze the pipeline until the branch outcome and target are known, then proceed with fetch. Thus, every branch instruction incurs a penalty equal to the number of stall cycles. This solution is unsatisfactory if the instruction mix contains many branch instructions, and/or the pipeline is very deep.
- **Branch delay slots:** The instruction set architecture is constructed such that one or more instructions sequentially following a conditional branch instruction are executed whether or not the branch is taken. The compiler or assembly language writer must fill these branch delay slots with useful instructions or NOPs (no-operation op codes).
- **Branch prediction:** The outcome and target of conditional branches are predicted using some heuristic. Instructions are speculatively fetched and executed down the predicted path, but results are not written back to the register file until the branch is executed and the prediction is verified. When a branch is predicted, the processor enters a speculative mode in which results are written to another register file that mirrors the architected register file. Another pipeline stage called the commit stage is introduced to handle writing verified speculatively obtained results back into the real register file. Branch predictors can't be など% accurate, so there is still a penalty for branches that is based on the branch mis prediction rate.

- **Indirect branch prediction:** Branches such as virtual method calls, computed goto and jumps through tables of pointers can be predicted using various techniques.
- **Return address stack (RAS):** Procedure returns are a form of indirect jump that can be perfectly predicted with a stack as long as the call depth doesn't exceed the stack depth. Return addresses are pushed onto the stack at a call and popped off at a return.

3.9.1 Static Branch Prediction

Branch prediction is a method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting for the actual outcome.

- In general, the bottoms of loops are branches that jump back to the top of the loop. These types of loops can easily be predicted as branch taken.
- The decision about a branch whether taken or not taken is arrived from the heuristics.
- **Dynamic hardware predictors,** guess the behavior of each branch and may change predictions for a branch over the life of a program.
- Dynamic prediction is performed by maintaining a history for each branch as taken or untaken, and then using the recent past behavior to predict the future.
- When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address.

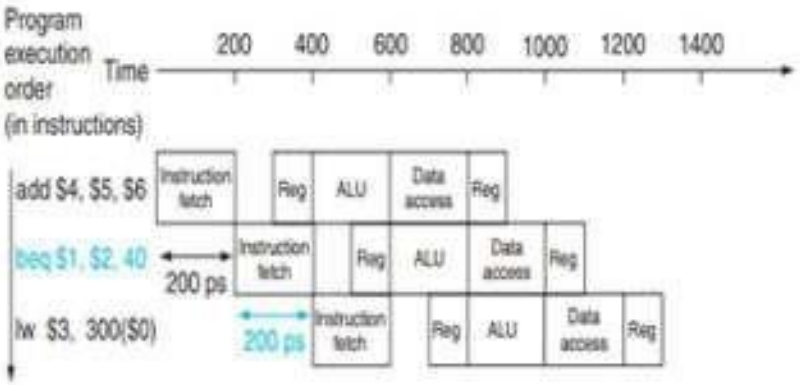


Fig 3.24 a) Branch not taken

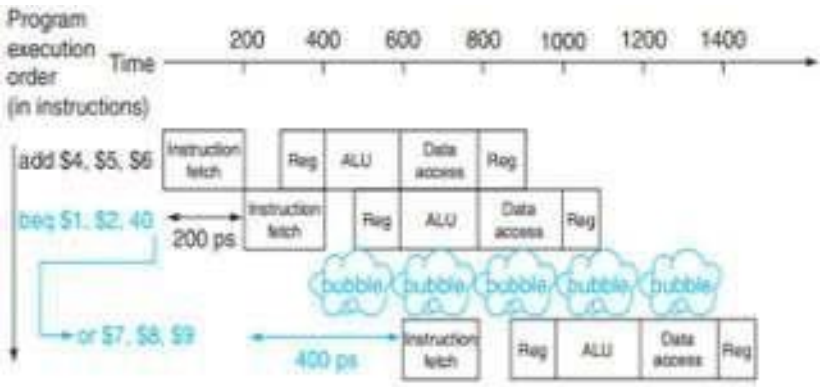


Fig 3.24 b) Branch taken

Branch Stalling

- This is stalling the instructions until the branch is complete is too slow.
- One improvement over branch stalling is to predict that the branch will not be taken and thus continue execution down the sequential instruction stream.
- If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.
- If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.
- To discard instructions, change the original control values to 0s.

Delayed Branches:

The delayed branch always executes the next sequential instruction, with the branch taking place after that one instruction delay. It is hidden from the MIPS assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer.

- One way to improve branch performance is to reduce the cost of the taken branch.
- The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty.

- Moving the branch decision up requires two actions to occur earlier:
  - Computing the branch target address
  - Evaluating the branch decision.
- The easy part of this change is to move up the branch address calculation.
- Despite these difficulties, moving the branch execution to the ID stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched.

3.9.2 Dynamic Branch Prediction

Prediction of branches at runtime using runtime information is called dynamic branch prediction.

- One implementation of that approach is a branch prediction buffer or branch history table.
- 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.

1 bit Prediction scheme

This scheme will be incorrect twice when not taken:

- Assume predict bit=0 to start (indicates branch not taken) and loop control is at the bottom of the code.
- First iteration in the loop, the predictor mispredict the branch since the branch is taken back to the top of the loop. Now invert the prediction bit (predict bit=1).
- Till the branch is taken, the prediction is correct.
- Exiting the loop, the predictor again mispredict the branch since this time the branch is not taken falling out of the loop. Now invert the prediction bit (**predict bit=0**).

Loop: first loop instruction

Second loopinstruction

--

--

--

Last loop instruction

Bne $1,$2, loop

Fall out instruction

2bit prediction scheme:

By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The two-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the midpoint of its range as the division between taken and not taken.

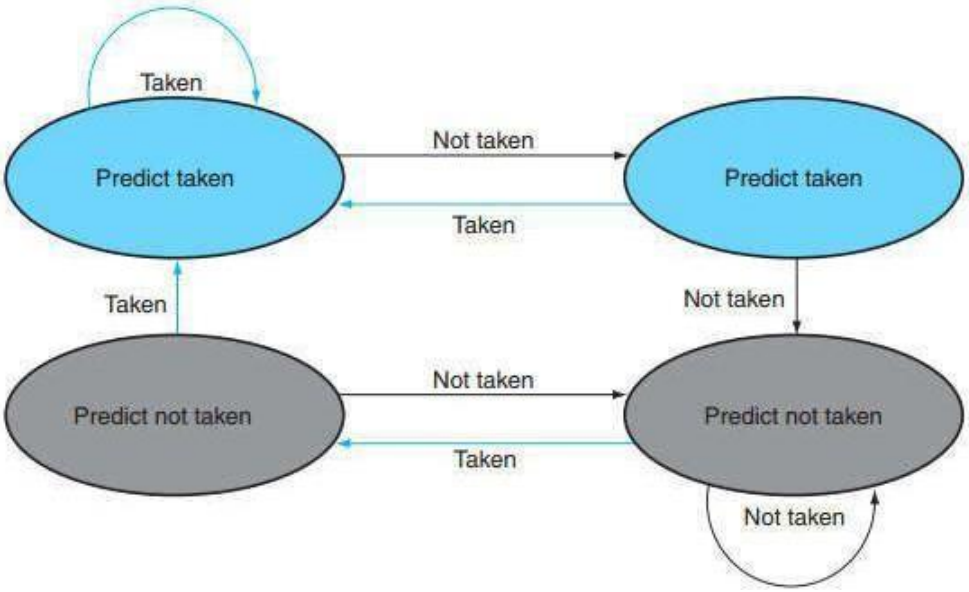


Fig 3.24: 2 bit prediction scheme

Branch delay slot:

- The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

- The limitations on delayed branch scheduling arise from the restrictions on the instructions that are scheduled into the delay slots the ability to predict at compile time whether a branch is likely to be taken or not.
- Delayed branching was a simple and effective solution for a five-stage pipeline issuing one instruction each clock cycle.
- As processors go to both longer pipelines and issuing multiple instructions per clock cycle, the branch delay becomes longer, and a single delay slot is insufficient.
- Hence, delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches.

3.10 EXCEPTIONS

Control is the most challenging aspect of processor design: One of the hardest parts of control is implementing exceptions and interrupts events other than branches or jumps that change the normal flow of instruction execution. They were initially created to handle unexpected events from within the processor, like arithmetic overflow. The term exception refer to any unexpected change in control flow without distinguishing whether the cause is internal or external. Interrupt is when the event is externally caused. The following are the causes of exceptions:

- R-type arithmetic overflow
- Executing undefined instruction
- I/O device request
- OS service request
- Hardware malfunction

Event	Location	MIPS term
I/O device request	External	Interrupt
OS service request	Internal	Exception
R-type arithmetic overflow	Internal	Exception
Executing undefined instruction	Internal	Exception
Hardware malfunction	Either	Exception / Interrupt

Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a processor, which determines the clock cycle time and performance.

Exception Handling in the MIPS Architecture:

- The two types of exceptions that MIPS implementation can generate are execution of an undefined instruction and an arithmetic overflow.

A pipelined implementation treats exceptions as another form of control hazard. For example, suppose there is an arithmetic overflow in an add instruction. Flush the instructions that follow the add instruction from the pipeline and begin fetching instructions

from the new address. This is done by turning the IF stage into a nop. Because of careful planning, the overflow exception is detected during the EX stage; hence, we can use the EX.Flush signal to prevent the instruction in the EX stage from writing its result in the WB stage. The final step is to save the address of the off ending instruction in the exception program counter (EPC). In reality, we save the address +4, so the exception handling the software routine must first subtract 4 from the saved value.

3.11 PARALLELISM VIA INSTRUCTIONS

The simultaneous execution of multiple instructions from a program is called Instruction Level Parallelism (ILP). It is a measure of how many of the instructions in a computer program can be executed simultaneously.

The ILP increases the depth of the pipeline to overlap more instructions. This is facilitated by adding extra hardware resources to replicate the internal component of the computer, so that it can launch multiple instructions in every pipeline stages. This is called **multiple issue.**

In Multiple Issue technique, multiple instructions are launched in one clock cycle.

This will improve the performance of the processor. The pipelined performance is estimated from the given formula (CPI-Cycles Per Instruction):

$$Pipeline\ CPI = Ideal\ CPI + Structural\ stalls + RAW\ stalls + WAR\ stalls + WAW\ stalls + Control\ stall$$

Launching multiple instructions per stage allows the instruction execution rate (CPI) to be less than 1. To obtain substantial increase in performance, we need to exploit parallelism across multiple basic blocks.

Implementing multiple issue processor

- **Static multiple issue processor:** Here the decisions are made by the compiler before execution.
- **Dynamic multiple issue processor:** Here the decisions are made during the execution by the processor.

The challenges in implementing a multiple issue pipeline are:

- **Packaging instructions into issue slots**: Issue slots are the positions from which instructions could be issued in a given clock cycle. To find the exact location of the current issue slot is the greatest challenge. So the process partially handled by the compiler. ; In dynamic issue designs, it is normally dealt with at runtime by the processor.
- **Dealing with data and control hazards:** In static issue processors, the consequences of data and control hazards are handled statically by the compiler. In dynamic issue processors, use hardware techniques to mitigate the control and data hazard.

3.11.1 Speculation

Speculation is an approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

- This allows the execution of complete instructions or parts of instructions before being certain whether this execution should take place.
- A commonly used form of speculative execution is control flow speculation where instructions past a control flow instruction are executed before the target of the control flow instruction is determined.
- Speculation may be done in the compiler or by the hardware.
- The uses speculation to reorder instructions, moving an instruction across a branch or a load across a store. The compiler usually inserts additional instructions that check the accuracy of the speculation and provide a fix-up routine to use when the speculation was incorrect.
- The processor hardware can perform the same transformation at runtime using techniques. The processor usually buffers the speculative results until it knows they are no longer speculative. If the speculation was correct, the instructions are completed by allowing the contents of the buffers to be written to the registers or memory. If the speculation was incorrect, the hardware flushes the buffers and re executes the correct instruction sequence.

Issue in Speculation:

Speculating on certain instructions may **introduce exceptions** that were formerly not present. The result would be that an exception that should not have occurred will occur. In

Compiler-based speculation, such problems are avoided by adding special speculation support that allows such exceptions to be ignored until it is clear that they really should occur. In hardware-based speculation, exceptions are simply buffered until it is clear that the instruction causing them is no longer speculative and is ready to complete; at that point the exception is raised, and normal exception handling proceeds.

3.11.2 Static Multiple Issue

The set of instructions that issues together in 1 clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

Static multiple-issue processors use compiler to assist with packaging instructions and handling hazards. The issue packet is treated as one large instruction with multiple operations. This is otherwise termed as **Very Long Instruction Word (VLIW).**Since the Intel IA-64 architecture supports this approach, it is known as **Explicitly Parallel Instruction Computer (EPIC**).

Loop unrolling is a technique used by compiler to solve static multiple issue.

Loop Unrolling is a technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different it rations are scheduled together.

Loop unrolling is a compiler optimization applied to certain kinds of loops to reduce the frequency of branches and loop maintenance instructions. It is easily applied to sequential array processing loops where the number of iterations is known prior to execution of the loop. After unrolling, there is more ILP available by overlapping instructions from different iterations.

- During the unrolling process, the compiler introduced additional registers, since multiple copies of the loop body are made.
- Augmenting new registers in loop unrolling is called **register renaming**. This is done to eliminate dependences that are not true data dependences, but may lead to potential hazards or may prevent the compiler from scheduling the code.
- To identify the independent instructions, it is necessary to trace the data dependencies.
- If there is no data values flow between the instructions, it is termed as **anti-dependence or name dependence.** This is an ordering forced purely by the reuse of a name.

- Renaming the registers during the unrolling process allows the compiler to the independent instructions for better code schedule.
- An **instruction group** is a sequence of consecutive instructions with no register data dependences among them.
- All the instructions in a group could be executed in parallel if sufficient hardware resources existed and if any dependences through memory were preserved.
- The compiler must explicitly indicate the boundary between one instruction group and another. This boundary is indicated by placing a stop between two instructions that belong to different groups.
- An explicit indicator of a break between independent and dependent instructions is termed as **stop**.
- **Predication** is a technique that can be used to eliminate branches by making the execution of an instruction dependent on a predicate, rather than dependent on a branch.
- Speculation and Predication improves ILP. Branches reduce the opportunity to exploit ILP by restricting the movement of code.
- Branches within a loop cannot be eliminated by loop unrolling. Predication eliminates this branch, by allowing more flexible exploitation of parallelism.
- **Speculation** consists of separate support for control speculation, which deals with deferring exceptions for speculated instructions, and memory reference speculation, which supports speculation of load instructions.
- Deferred exception handling is supported by adding speculative load instructions, which, when an exception occurs, tag the result as poison.
- **Poison** is the result generated when a speculative load yields an exception, or an instruction uses a poisoned operand. When a poisoned result is used by an instruction, the result is also poison, the software can then check for a poisoned result when it knows that the execution is no longer speculative.
- The speculation on memory references can be made by moving loads earlier than stores on which they may depend. This is done with an advanced load instruction.

- Advanced load is speculative load instruction with support to check for aliases that could invalidate the load. This demands the use of a special table to track the address that the processor loaded from.
- A subsequent instruction must be used to check the status of the entry after the load is no longer speculative.

3.11.2 Dynamic Multiple-Issue Processors

- Dynamic multiple issue processors are implemented using **superscalar processors** that are capable of executing more than one instruction per clock cycle.
- The compiler must schedule the instructions to the processors without any dependencies.
- To facilitate this, **dynamic pipeline scheduling** is performed by providing hardware support for reordering the order of instruction execution so as to avoid stalls.

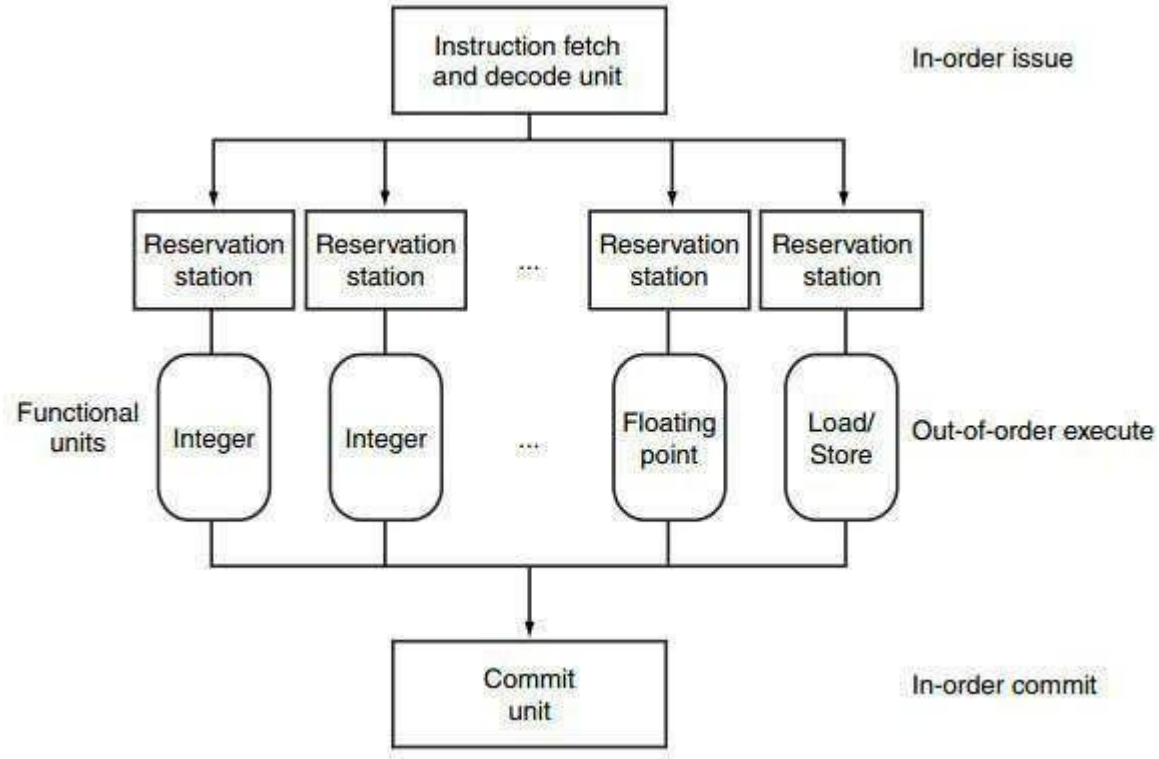


Fig 3.25: Units of dynamic scheduling pipeline

The following are the important components of dynamic scheduling pipelines:

- **Instruction Fetch Unit: This** unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution.
- **Functional unit:** They have buffers, called **reservation stations** that hold the operands and the operation. As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit.
- **Commit Unit:** This buffers the result until it is safe to put the result into the register file or, for a store, into memory. The buffer in the commit unit, called the **reorder buffer**, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.

Operation of dynamic scheduling pipeline:

- When an instruction issues, if either of its operands is in the register file or the reorder buffer, it is copied to the reservation station immediately, where it is buffered until all the operands and an execution unit are available. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.
- If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers.

Dynamic scheduling is often extended by including hardware-based speculation, especially for branch outcomes. By predicting the direction of a branch, a dynamically scheduled processor can continue to fetch and execute instructions along the predicted path.