DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

PUDUCHERRY TECHNOLOGICAL UNIVERSITY

PUDUCHERRY – 605014

EC232 - PROJECT WORK

HAZARD-HANDLING IN A FIVE-STAGE PIPELINE MODEL BASED ON MIPS32

Under the guidance of

DR. A. V. ANANTHALAKSHMI, Associate Professor, Department of ECE, PTU

External Mentor

Mr. S. GOPI KRISHNA, Scientist 'F', DECS, RCI, DRDO

A PROJECT REPORT

Submitted By

NANDI SREE HARSHITHA	**[** 20EC1059 **]**
KRISHNA TEJA MURIKIPUDI	**[** 20EC1042 **]**
NAPPINNAI V	**[** 20EC1060 **]**
HARSHAL KARIYA	**[** 20EC1026**]**

MAY – 2024

BONAFIDE CERTIFICATE

This is to certify that the project report titled "**HAZARD-HANDLING IN A FIVE-STAGE PIPELINE MODEL BASED ON MIPS32"** is a bonafide work carried out by:

- **NANDI SREE HARSHITHA,** [20EC1060]
- **KRISHNA TEJA MURIKIPUDI,** [20EC1042]
- **NAPPINNAI V,** [20EC1060]
- **HARSHAL KARIYA,** [20EC1026]

of eighth semester B.Tech. class of **Electronics and Communication Engineering** at **PUDUCHERRY TECHNOLOGICAL UNIVERSITY** as a part of the eighth semester project work during the academic year 2023-2024**.**

UNIVERSITY GUIDE	HEAD OF THE DEPARTMENT
DR. A. V. ANANTHALAKSHMI	**DR. V. SAMINADAN**

ACKNOWLEDGEMENT

This project report would not be possible without all those who have contributed to its successful completion. We take this opportunity to appreciate and acknowledge the efforts of the people who made this project possible. We are delighted to express our profound gratitude towards our external mentor, **Mr. S. GOPI KRISHNA,** Scientist 'F' at Research Centre Imarat, Defence Research & Development Organisation, Hyderabad. Your unwavering support and guidance proved to be invaluable to us throughout the project duration.

We would like to express our heartfelt appreciation to our university guide **Dr. A. V. ANANTHA LAKSHMI,** Associate Professor, Department of Electronics and Communication Engineering, Puducherry Technological University, for her ceaseless support and valuable guidance and suggestions during the course of the project.

We would also like to thank **Dr. V. SAMINADAN,** Professor and Head of the Department of Electronics and Communication Engineering, Puducherry Technological University for encouraging us to pursue this wonderful opportunity at Defence Research & Development Organisation and providing us with his immense support when asked for.

We feel grateful towards the review by the panel members, **Dr. V. VIJAYALAKSHMI** Professor**, Dr. S. BATMAVADY** Professor**, Dr. G. NAGARAJAN** Professor, of Electronics and Communication Engineering Department, Puducherry Technological University, for their valuable suggestions, support and encouragement.

We extend our gratitude to all teaching and non-teaching staff of the department. We are also thankful to our families and friends for their unwavering support and encouragement throughout this project This project would not have been possible without the contributions of the aforementioned individuals and entities. We are truly grateful for their support.

ABSTRACT

The purpose of this project is to demonstrate the detailed design and simulation of a MIPS (Microprocessor without Interlocked Pipeline Stages) processor model with Verilog Hardware Description Language. The objective of this project is to gain a deeper understanding of the MIPS architecture and how it is implemented.

This project involves designing a single-cycle MIPS32 processor, which is then extended to a five-stage pipelined MIPS32 processor. The project presents an analysis and simulation of the MIPS32 processor model using Verilog. For high performance, the design incorporates a five-stage pipeline consisting of Fetch, Decode, Execute, Memory Access and Write Back stages.

As part of this project, Instruction Memory, Fetch Unit, Control Unit, Decode Unit, Execute Unit, and Data Memory are the primary Verilog program modules. The DataMem256x32 register memory has a capacity of 256x32 and is instantiated in Data Memory. An R, I, and J-type instruction set is given in Instruction Memory, and respective output waveforms are generated with it. After loading the hexadecimal object file into Instruction Memory, a five-stage pipeline is used to execute it.

As a final step, Hazard-Handling capabilities are added to the existing design. The processor is tested under various bottleneck scenarios and the resulting outputs are generated and verified using Xilinx's Vivado simulation tool. A variety of performance metrics are used to gain insights into the capabilities of the MIPS32 processor design by analysing the performance of the processor.

Keywords: MIPS, Verilog, MIPS32, Five-Stage Pipeline, Verification, Xilinx Vivado, Hazard-Handling, Bottlenecks.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

MIPS (Microprocessors without Interlocked Pipeline Stages) is a family of Reduced Instruction Set Computing (RISC) architectures that has played an important role in the development of microprocessors since its invention in the 1980s. RISC is a way of organizing a computer's processor focusing on simplicity and speed. RISC is a type of microprocessor design where each instruction is executed in a short amount of time for faster overall computation.

The concept of pipelining involves the overlapping of multiple instructions to optimize the time and ability of hardware units. The term "Without Interlocked Pipeline Stages" refers to the MIPS pipeline structure with stages that are not fully interlocked, allowing it to operate at higher clock speeds and greater flexibility. Its improved performance also introduces challenges in handling hazards, such as data hazards, control hazards and structural hazards.

This project involves designing and simulating a single-cycle 32-bit MIPS and extending it into a five-stage pipeline in HDL. It features a fixed-length instruction format, simplifying instruction decoding and execution. The architecture includes a 32-bit instruction word length, facilitating uniformity in instruction handling. It is ideal for environments that require high performance and efficiency. Many applications use this technology, such as network routers, digital televisions, embedded systems, video game consoles, and even some supercomputers. The simplicity of the MIPS instruction set makes it ideal for these types of applications.

CHAPTER 2
LITERATURE REVIEW

TITLE	AUTHOR NAME	JOURNAL NAME / MONTH / YEAR OF PUBLICATION	TECHNIQUE PROPOSED
Design of Instruction Set Architecture Based 16 Bit MIPS Architecture with Pipeline Stages	Kanaka Sai Hemanth Dogga, Chand Basha Shaik, Sreemukhi Muddusetty	International Research Journal of Engineering and Technology (IRJET) Volume:08, Issue:07/July 2021	16-bit MIPS processor with 5 stage pipeline. It is a smartly optimized, fast and simple version consisting of the most required instructions.
Review Of Low Power Multicycle MIPS Processor Using HDL	Tripti Mahajan, Prof. Nikhil P. Wyawahare	Journal of Emerging Technologies and Innovative Research Volume 6, Issue 4/ April 2019	5 stage pipelined 32-bit MIPS Processor to perform multiple instructions in a single clock cycle with no interlocked pipeline stages.
A Review on MIPS RISC Processor	Ms. Ashwini Golghate, Vishal Jaiswal	International Journal of Innovative Research in Technology (IJIRT) Volume: 4, Issue: 12, May 2018	32-bit Microprocessor interlocked Pipeline Stage RISC processor based on floating point concept will be done by using VHDL to cope with parallelism.
Design and Implementation of 32 – bit RISC Processor using Xilinx	Galani Tina G., Riya Saini and R.D.Daruwala	International Journal of Emerging Trends in Electrical and Electronics (IJETEE) Vol. 5, Issue.1, July-2013	32 – bit RISC processor using XILINX VIRTEX4 for testing performance issues like area, power dissipation and propagation delay.
A 16-bit MIPS Based Instruction Set Architecture for RISC Processor	Sagar Bhavsar, Akhil Rao, Abhishek Sen, Rohan Joshi	International Journal of Scientific and Research Publications, Volume 3, Issue 4, April 2013	16-bit MIPS for low-cost, compact and low power RISC instruction set architecture.

CHAPTER 3
PROBLEM FORMULATION

3.1. NEED FOR PIPELINING IN VLSI SYSTEMS

VLSI digital system design is fraught with many complex features. Multitasking and parallelism make the system slower and consume more power to meet the needs of their users, and designers are forced to settle for important factors. The best way to overcome this problem is to implement pipeline technology in the VLSI system design.

3.2. PIPELINING

Pipelining is an implementation technique that performs multiple operations of instruction that are simultaneous to optimise the speed, area, and throughput of the task. Implementing an instruction pipeline increases power efficiency, delay, time, and speed, as well as taking full advantage of the hardware. The purpose of this pipeline process is to reduce power, increase speed and get all the benefits of high performance.

Pipelining improves the efficiency and speed of the processor by breaking down tasks into smaller stages that can be executed concurrently.

3.3. RISC vs CISC

RISC (Reduced Instruction Set Computer) architectures have a simplified instruction set, typically with a smaller number of instructions. Each instruction performs a relatively simple operation, and the instructions are designed to execute in a single clock cycle. Whereas, CISC (Complex Instruction Set Computer) architectures have a more complex instruction set, with instructions capable of performing multiple operations or accessing memory directly. Instructions in a CISC architecture can vary in length.

RISC architectures prioritize simplicity, efficiency, and pipelining, while CISC architectures offer more flexibility and complexity in their instruction sets, often at the expense of some efficiency. Both RISC and CISC architectures have their advantages and are suitable for different types of applications and use cases.

For our implementation of the MIPS processor, RISC architecture is efficient in various ways as compared to CISC architecture because it consumes less power, executes faster because the number of instructions is smaller, and has simplified addressing modes with a simpler design.

3.4. MIPS RISC

MIPS RISC processors can perform many advanced desirable tasks qualitatively, without interlocking millions of instructions carried at once. Pipelining is a well-constructed linear structure, where the implementation of instructions is done step by step to enforce ease of operation. Instruction implementation optimizes synchronization time requirements and resources. MIPS is a RISC-based instruction set architecture that not only increases the speed by RISC set but is also efficient in handling threats.

CHAPTER 4
DESIGN LIMITATIONS

4.1 INSTRUCTION SET ARCHITECTURE COMPLEXITY

Implementing the full MIPS instruction set architecture, which includes various instruction types such as arithmetic, logical, load/store, branch, and control transfer instructions, can be complex.

4.2 PIPELINE DESIGN

Implementing the full MIPS instruction set architecture, which includes various instruction types such as arithmetic, logical, load/store, branch, and control transfer instructions, can be complex.

4.3 MEMORY HIERARCHY

Managing the memory hierarchy, including instruction and data memory, memory access, and handling memory misses efficiently, requires careful design to balance performance and complexity.

4.4 CONTROL UNIT DESIGN

Implementing the control unit responsible for generating control signals to coordinate the operation of various processor components based on the fetched instruction adds complexity to the design**.**

4.5 HAZARD HANDLING

There is a possibility that hazards can occur in pipelining, which might lead to incorrect outputs from the processor. These types of hazards include:

- STRUCTURAL HAZARD: This happens when more than one instruction wants to access the same memory simultaneously in the same clock cycle.

- DATA HAZARD: This hazard occurs when a new instruction uses the register data whose value has to be modified by the previous instruction but it is retrieved by the new instruction before it gets modified. This hazard may cause the new instruction to use the wrong data.

- CONTROL HAZARD: These occur while using branching instruction. While using branch instruction branching takes place only when the condition is met. And the processor gets to know whether the condition is met or not in the “execute” stage.

4.6 CLOCK SPEED AND TIMING CONSTRAINTS

Achieving high clock speeds while meeting timing constraints and ensuring proper synchronization of pipeline stages is a significant challenge in MIPS processor design.

CHAPTER 5
OBJECTIVE

The objective of this project is to design a 32-bit MIPS processor with five- stage pipeline, namely Fetch (F), Decode (D), Execute (EX), Memory Access (MEM) & Write Back (WB) and to the hazards by optimizing the design further. Outcome specific objectives include:

- Designing a single-cycle MIPS processor and extending it into a five-stage pipeline.
- Gaining comprehensive understanding of the MIPS architecture and its instruction set.
- Understanding pipelining techniques and their impact on the processor's performance.
- Implementing the MIPS-I subset of the MIPS instruction set.
- Exploring fundamental processor design principles.
- Handling Structural, Data & Control Hazards
- Verification of the designed processor.
- Comparing our design with existing MIPS implementations.

And henceforth analyse the strengths and shortcomings of the designed model

CHAPTER 6
PROPOSED TECHNIQUE

To begin the design of a MIPS processor, it is required we understand in depth about the architecture of both the single-cycle model and the pipelined model of the MIPS processor. The design takes care of the data path and control unit responsible for the working of the processor. The control unit and data path are hence updated according to the necessity to implement a subset of MIPS instruction set. The same is repeated for the pipelined model of MIPS.

The correctness of the coded modules is verified by tallying the behaviour of object code generated with the ideal behaviour of the code, which includes checking various registers and memory elements by single-step execution. Deviation in ideal behaviour implies a faulty data path or control unit that needs to be addressed.

Then various performance metrics are tested and compared with existing models to analyse the performance of the designed processor and improve it by varying one or more simulation parameters. By following the above steps, we can verify, test and enhance the performance of the processor.

1. **Task Segmentation**: The task to be performed is divided into smaller sub-tasks, each of which can be executed independently.

2. **Pipeline Stages**: The processor is divided into several stages, with each stage responsible for completing one portion of the overall task. These stages are interconnected, forming a pipeline.

3. **Concurrent Execution**: Each stage in the pipeline operates concurrently, meaning that while one stage is processing a particular sub-task, other stages are simultaneously processing different sub-tasks.

4. **Overlap of Tasks**: As one sub-task completes processing in one stage, its output is passed to the next stage for further processing, while a new sub-task enters the pipeline at the initial stage. This overlap of tasks ensures continuous processing and maximizes throughput.

5. **Pipeline Registers**: To facilitate the flow of data between pipeline stages, pipeline registers are used to temporarily store intermediate results as they move through the pipeline.

6. **Control Logic**: Control signals coordinate the flow of data and ensure that each stage operates in synchronization with the others.

7. **Hazard Handling**: Special mechanisms are implemented to handle hazards such as data hazards, control hazards, and structural hazards, which may arise due to dependencies between pipeline stages or resource conflicts.

8. **Optimization**: Pipelining can be optimized by balancing the workload among pipeline stages, minimizing pipeline stall cycles, and reducing overheads introduced by pipeline registers and control logic.

6.1 REGISTERS AND THEIR TYPES

1.The zero register ($zero or $0) always contains a value of 0. It is built into the hardware and therefore cannot be modified.

2.The $at (Assembler Temporary) register is used for temporary values.

3.The $v Registers are used for returning values from functions.

4.The temporary registers ($8-$15) are used by the assembler or assembly language programmer to store intermediate values.

5.Saved Temporary ($16-$23) registers are used to store longer lasting values. They are preserved across function calls.

6.The k registers ($26-$27) are reserved for use by the OS kernel.

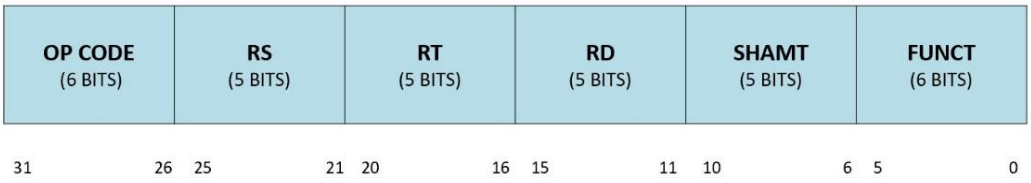
7.Pointer Registers

- Global Pointer ($gp) - Usually stores a pointer to the global data area
- Stack Pointer ($sp) - Used to store the value of the stack pointer.
- Frame Pointer ($fp) - Used to store the value of the frame pointer.
- Return Address ($ra) - Stores the return address

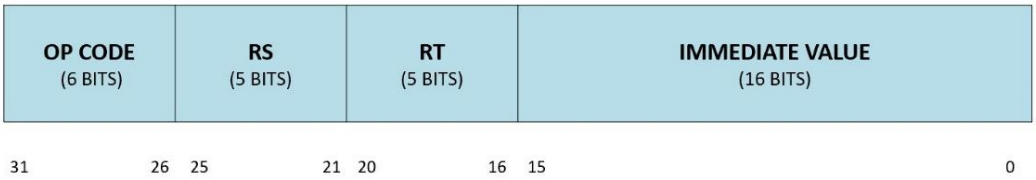
6.2 INSTRUCTION FORMAT AND ADDRESS FORMAT

In MIPS there are 3 ways to format the instruction. They are register format, immediate format and jump format. MIPS is organized by 3 addressing formats. Rs, Rt are two inputs for the ALU and Rd is the destination of the result.

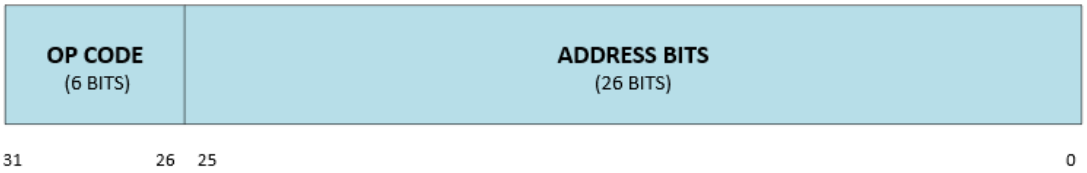
R FORMAT INSTRUCTION



I FORMAT INSTRUCTION



J FORMAT INSTRUCTION



EXAMPLES:

I FORMAT INSTRUCTIONS

ASSEMBLY CODE	MACHINE CODE (HEX)
ADDI $t4 $t3 0x0101	**0x216c0101**
ORI $t3 $t4 0x1000	**0x0358B100**
SRL $t4 $t1 0x0001	**0x000B6042**

R FORMAT INSTRUCTIONS

ASSEMBLY CODE	MACHINE CODE (HEX)	RS	RT	RD
ADD $t2 $t1 $t0	0x01285020	9	8	0a
SUB $t3 $t2 $t1	0x01495822	0a	9	0b
ADD $t4 $t3 $t2	0x016a6020	0b	0a	0c

J FORMAT INSTRUCTIONS

ASSEMBLY CODE	MACHINE CODE (HEX)
JAL 0x2500	0x0C002500
J 0x2950	0x08002950

6.3 CONTROL SIGNALS

Control signals play a crucial role in MIPS. They simplify the data path design of the MIPS Processor. The significance of these control signals is given below:

1.RegDst (Destination Register Select):

RegDst is 0: The result is written to the register specified in the rt field (second operand register) of the instruction. This is mainly used for load instructions (lw, lb) where the loaded data needs to be stored in the designated register.
RegDst is 1: The result of the instruction is written to the register specified in the rd field (destination register) of the instruction. This is typically used for R-type instructions (arithmetic, logical) and certain immediate instructions where the outcome needs to be stored in a register for further use.
In some instructions, like store instructions (sw, sb), RegDst is irrelevant because the result is not written to a register but directly sent to the memory unit for storage. In such cases, the control signal might be set to a "don't care" state.

2.RegWrite (Destination Register Select):

The RegWrite control signal in MIPS plays a crucial role in determining whether the result of an instruction is written back to a register.
RegWrite is 0: The result of the instruction is not written back to any register. This occurs in several scenarios: Load instructions, Store instructions, Branch instructions.
RegWrite is 1: The result of the instruction is written back to the register specified in the rd field (destination register) of the instruction.
R-type instructions (arithmetic, logical): The result of the ALU operation is stored in the specified register.
I-type instructions (certain immediate instructions): The calculated result involving the immediate value is stored in the rd register.

3. ALUSrc:

This control signal plays a crucial role in directing the second operand to the Arithmetic Logic Unit (ALU). It essentially acts as a switch that determines the source of the second operand based on the instruction being executed.
The ALUSrc is the control signal of MUX that modifies the 2nd operand of ALU as below:
ALUSrc is 0: Second operand comes from the second register file output (Read data 2).
ALUSrc is 1: Second operand is the sign-extended lower 16 bits of the instruction.
For R-type instructions ALUSrc is 0 and rt is taken as the second operand.
For I and J-type instructions ALUSrc is 1; directing the immediate value from the instruction itself to the ALU as the second operand.

4. PCSrc (Program Counter Source):

The Program Counter holds the memory address of the next instruction to be executed. It is also called as Instruction Pointer.
After fetching the instruction, the PC is typically incremented to point to the address of the subsequent instruction in the program sequence.
However, the PC can be updated to different addresses based on certain conditions. For example, if the program encounters a jump instruction, the PC might be modified to point to the target address of the jump, altering the execution flow.
The PCSrc is the control signal of MUX that modifies the PC as below:
PCSrc is 0: PC is replaced by the output of adder that computes PC + 4.
PCSrc is 1: PC is replaced by the output of adder that computes branch target.

5.MemRead (Memory Read):

This is a control signal that instructs the data memory unit of the MIPS processor to read data from a specific memory address.
It essentially triggers the memory unit to retrieve the value stored at the address specified by the base address and offset during a load instruction.
MemRead is 0: Instructions where memory read shouldn't be involved (arithmetic etc...)
MemRead is 1: Execution of load instructions.

6.MemWrite (Memory Write):

MemWrite acts as a control signal that instructs the data memory unit of the MIPS processor to write data to a specific memory address.
It triggers the memory unit to store the value present on the data write port of the processor into the memory location specified by the base address and offset during a store instruction.
MemWrite is 0: Instructions where memory unit shouldn't be involved in writing data.
MemRead is 1: Execution of load instructions.

7.MemtoReg (Memory to Register):

The MemtoReg control signal plays a crucial role in determining the source of data for the write-back stage of certain instructions. It acts as a multiplexer (MUX), choosing between two potential sources
MemtoReg is 0: The result from the ALU becomes the source for writing back to a register.
MemtoReg is 1: The data read from memory during a load instruction becomes the source for writing back to a register. This is typically used in load instructions like lw (load word) and lb (load byte) where the loaded data needs to be stored in the intended register.

6.4 HAZARDS

- Hazards prevent the next instruction in the instruction stream from executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.
- Hazards in pipelines can make it necessary to stall the pipeline.
- Avoiding a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed

i. STRUCTURAL HAZARDS

Structural hazards occur when there are conflicts in resource allocation, preventing certain combinations of instructions from being executed simultaneously. Functional unit bottleneck: If a functional unit is not fully pipelined, a sequence of instructions using that unit cannot proceed at the rate of one per clock cycle.

Resource Duplication Issue: Insufficient duplication of resources can lead to conflicts. For instance, if there's only one register-file write port but the pipeline requires two writes in a clock cycle, a structural hazard occurs. Structural Hazards cause Pipeline stalls and Pipeline bubbles.

ii. DATA HAZARDS

Data hazards occur when the pipeline changes the order of read/write accesses to operands, causing discrepancies from sequentially executing instructions on an unpipelined processor.

Impact: Incorrect data may be used by dependent instructions. Unpredictable behavior can occur, especially if interrupts interrupt the pipeline between instructions.

Solution: Techniques such as forwarding or data forwarding can be employed to eliminate stalls caused by data hazards. Forwarding involves forwarding data directly from the execution stage to the stage that needs it, bypassing the writeback stage.

iii. BRANCH/CONTROL HAZARDS

Control hazards occur when the execution of instructions is affected by the outcome of control flow instructions, such as branches. Branch instructions can cause pipeline stalls and performance loss due to their unpredictable nature. Branch instructions may change the program counter (PC) to a different value based on their outcome (taken or untaken). The pipeline must handle branches by predicting their outcome and managing instruction fetch accordingly.

iv. PIPELINE STALL SOLUTIONS

- Redo Instruction Fetch: If a branch is detected during instruction decode (ID), the pipeline stalls by repeating the instruction fetch stage (IF).
  This solution incurs a fixed penalty for every branch, leading to performance loss.
- Predicted-Not-Taken Scheme: Treat every branch as not taken initially, allowing the pipeline to continue fetching instructions.
  If a branch is taken, turn the fetched instruction into a no-op and restart fetching from the branch target address.
- Predicted-Taken Scheme: Assume every branch is taken immediately upon decode, fetching and executing at the target address.
  Suitable for processors where branch targets are known before the branch outcome.
- Delayed Branch Scheme: In a delayed branch, the execution cycle with a branch delay of one executes the instruction immediately following the branch.
  Helps mitigate the impact of branch penalties by overlapping execution with the branch instruction.

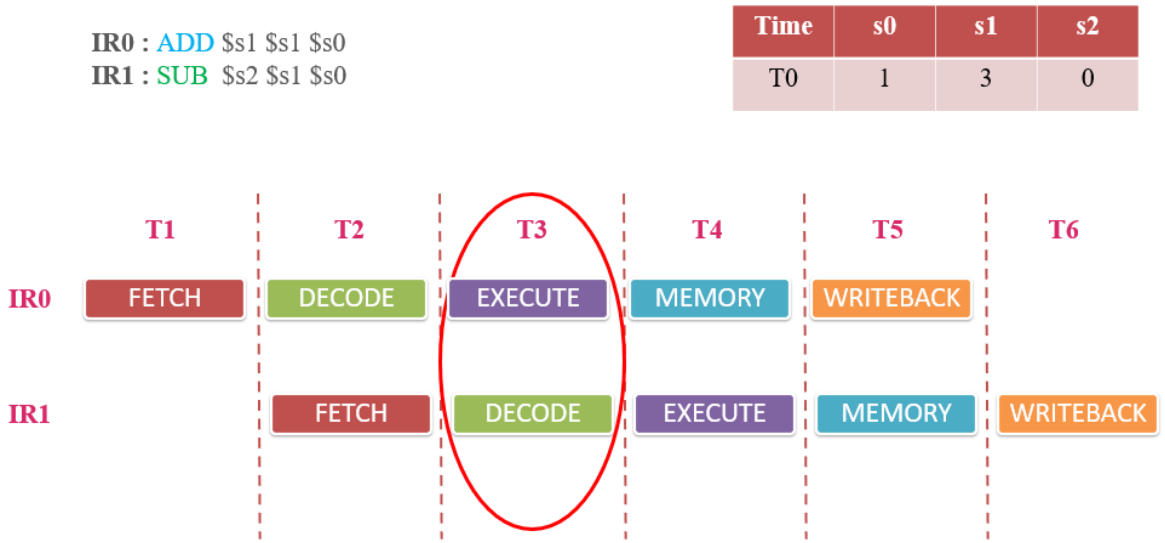


Fig 1. DATA HAZARD IN A FIVE STAGE PIPELINE

v. DEPENDENCY RESOLVER

Data Hazards occur when the value of a register in use is not up-to date, producing undesired outputs. The hazards can be broadly categorized as:

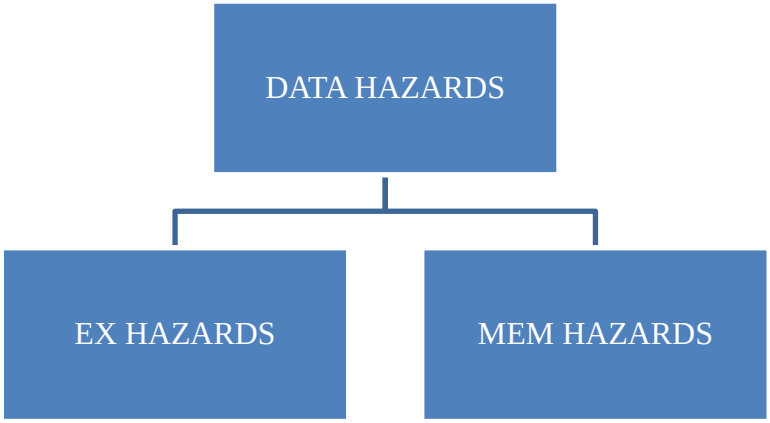


Fig 2. TYPES OF DATA HAZARDS

EX HAZARD:

The value of the register is used in the *Execute Stage* of the instruction either as an operand input to the ALU or for the calculation of effective address. So, those instructions whose *Execute Stage* occurs before the *Write Back Stage* of the prior Instruction with dependencies are prone to EX hazard. For example:

```
add  $1, $2, $3
sub  $3, $1, $2
or   $4, $2, $1
and  $13, $1, $1
sw   $12, 100($1)
```

In the above example there is a dependency between add-sub, add-or and there is no dependency between add-and as the *Write Back Stage* of the *add* instruction occurs in the same clock cycle as the *Execute Stage* of the *and* instruction. And evidently, no dependency between add-sw as *sw's Execute* occurs after *Write Back* of *add.*

The control signals for the multiplexers used to handle EX Hazards (*Generated in Decode Module*):

DepRes_Rt_Sel_pre: *Compares Rt1(IF/ID) with Destination Register0 (ID/EX).*
DepRes_Rs_Sel_pre: *Compares Rs1(IF/ID) with Destination Register0 (ID/EX).*
DepRes_Rt_Sel_EX_pre: *Compares Rt2(IF/ID) with Destination Register0 (EX/MEM).*
DepRes_Rs_Sel_EX_pre: *Compares Rs2(IF/ID) with Destination Register0 (EX/MEM).*

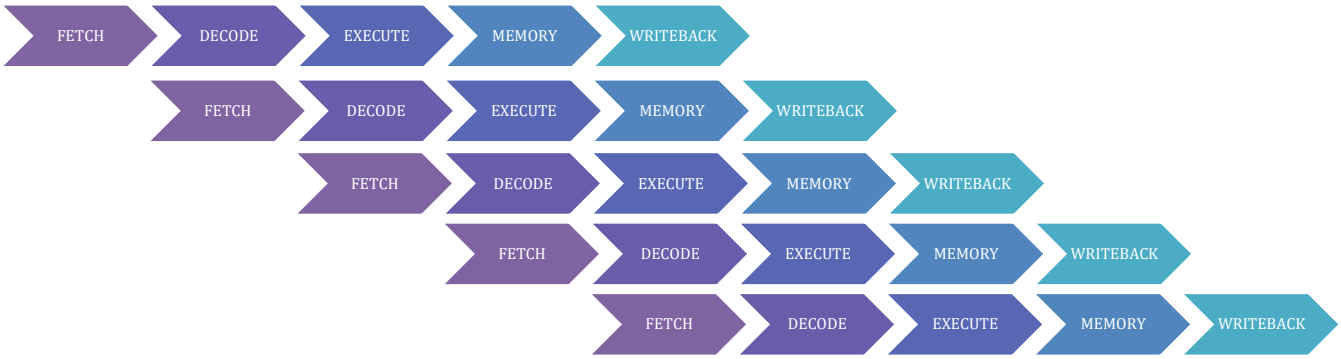


Fig 3. PIPELINE STAGES OF VARIOUS INSTRUCTIONS

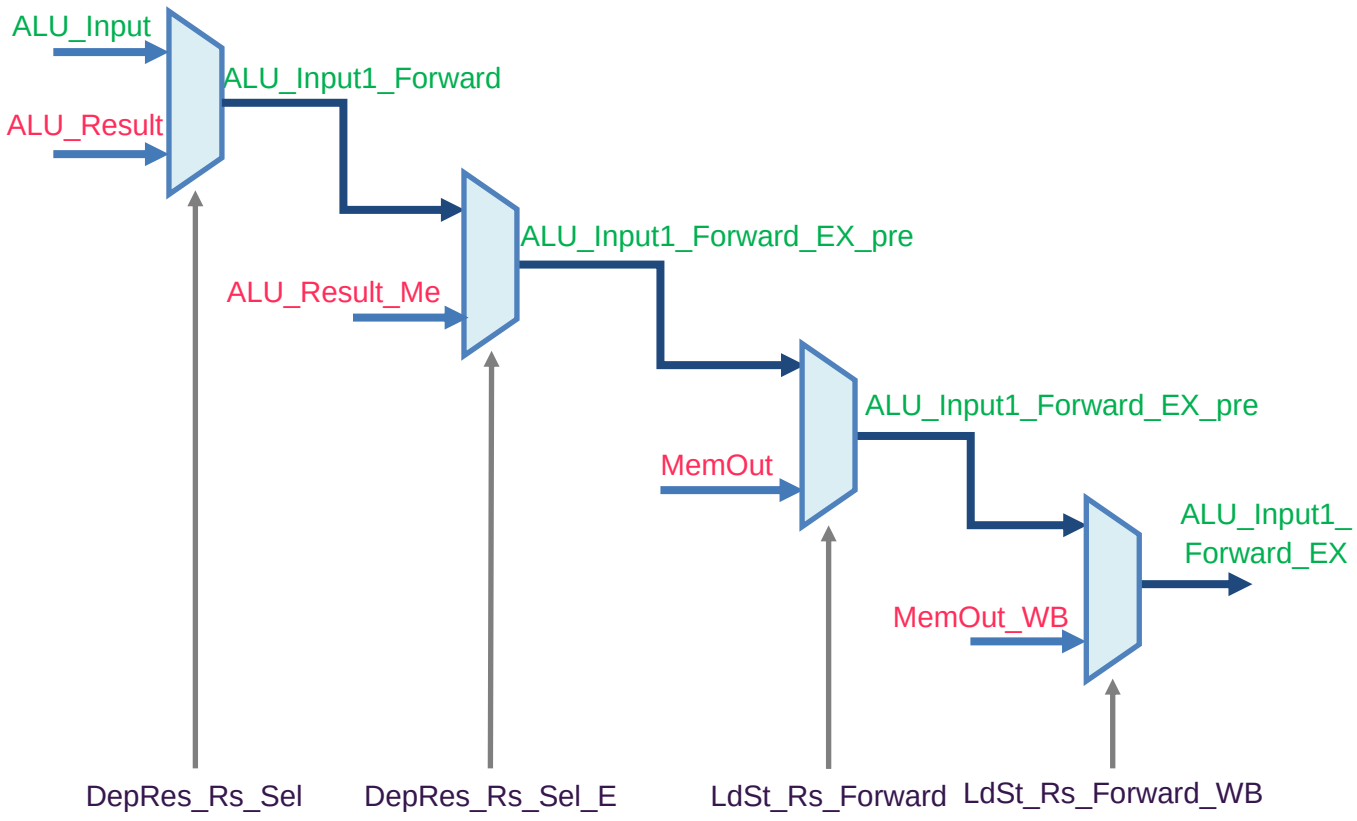


Fig 4. DEPENDENCY RESOLVER DIAGRAM FOR Rs

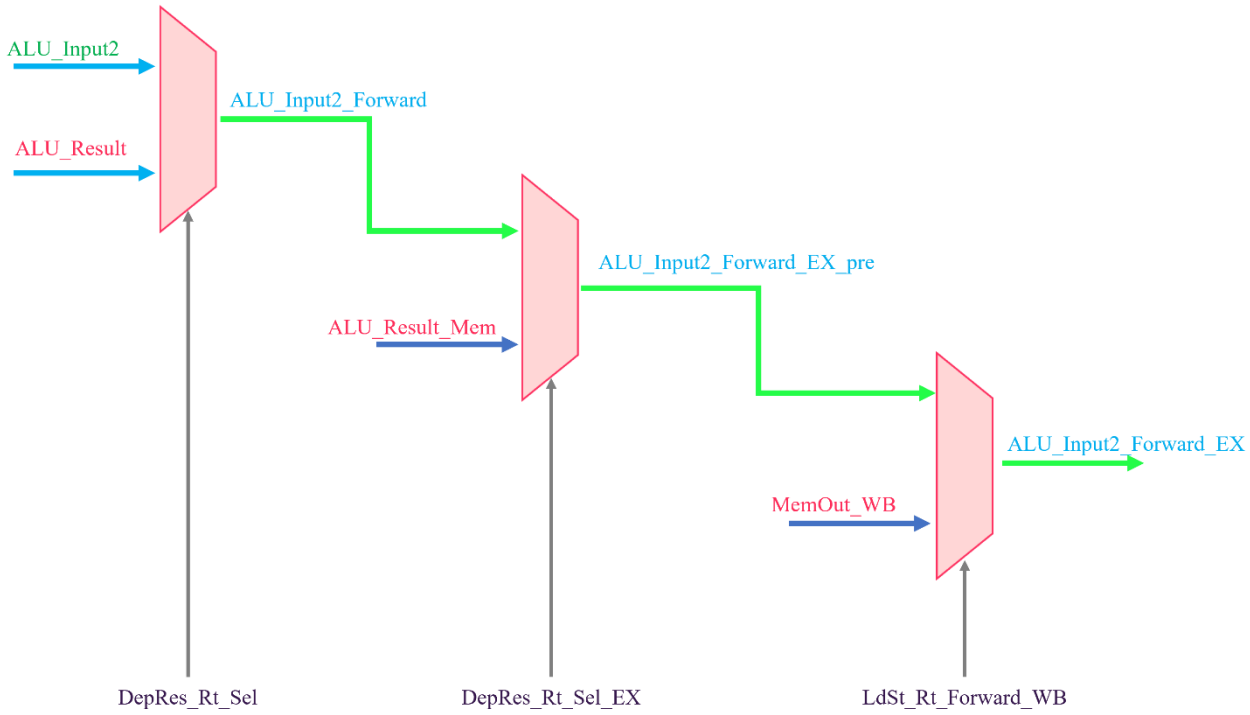


Fig 5. DEPENDENCY RESOLVER DIAGRAM FOR Rt

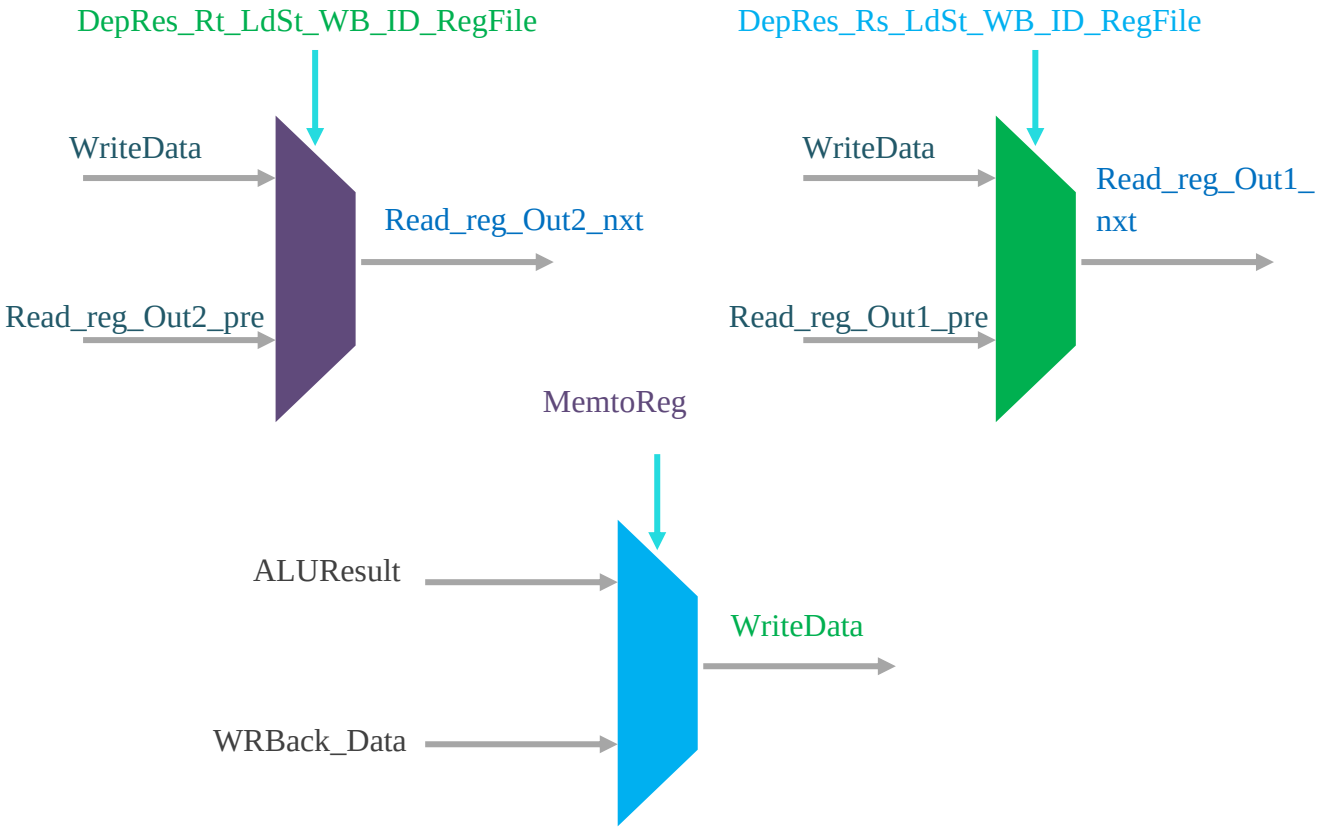


Fig 6. MEM DEPENDENCY RESOLVER MUXES

MEM HAZARD:

There is no hazard in the WB stage, because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file**.**
It is assumed that within the same clock cycle, Register Write occurs in first half cycle and the Register Read occurs in the second half cycle.

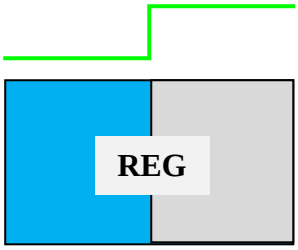


Fig 7. REGISTER STRUCTURE FOR READ AND WRITE

The control signals for the multiplexers used to handle MEM Hazards (*Generated and used in Decode Module*):
DepRes_Rt_LdSt_WB_ID_RegFile: *It is set when Rt of the instruction in Decode Stage (IR3) has a dependency on Destination Register of the instruction in WriteBack Stage (IR0).*
DepRes_Rs_LdSt_WB_ID_RegFile: *It is set when Rs of the instruction in Decode Stage (IR3) has a dependency on Destination Register of the instruction in WriteBack Stage (IR0).*
MemtoReg: *It is set when the value loaded from memory is to be written into the register instead of ALU result.*

In the present code, data forwarding within the register files is taken care of in the *Decode Module*, whereas EX Hazard handling signals were generated in *Decode Module* but Data Forwarding takes place in the *Execute Module*.

CHAPTER 7
BLOCK DIAGRAM

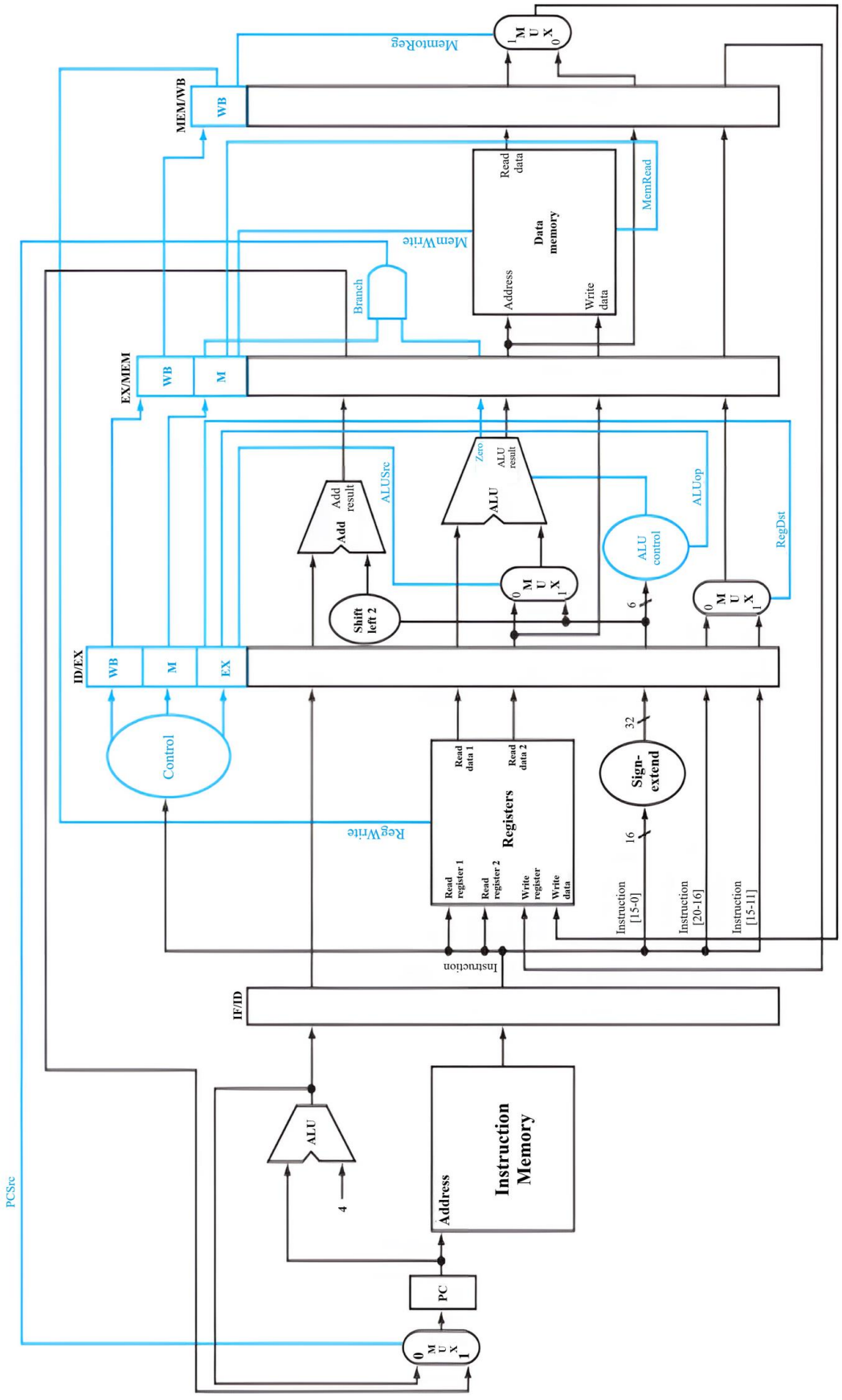


Fig 8. BLOCK DIAGRAM OF THE MIPS ARCHITECTURE

CHAPTER 8
ADVANTAGES OF THE PROPOSED TECHNIQUE

8.1. Processing Speed:

- MIPS technology allows processors to execute many instructions per second, resulting in faster overall processing speed.
- This speed is crucial for tasks that require quick calculations, such as scientific simulations, financial modelling, and multimedia processing.

8.2. Computational Performance:

- Higher MIPS values generally indicate improved computational performance, making the technology suitable for applications that demand intensive calculations or data manipulations.

8.3. Efficiency in Algorithm Execution:

- MIPS technology is beneficial for executing complex algorithms efficiently, enabling smoother operation of software applications that rely on intricate mathematical or logical operations.

8.4. Real-Time Processing:

- In scenarios where real-time data processing is essential, such as in embedded systems, telecommunications, or control systems, the high MIPS count contributes to timely and responsive execution of tasks.

8.5. Energy Efficiency:

- Many modern processors with MIPS architecture focus on optimising power consumption. This leads to energy-efficient computing solutions, reducing the environmental impact and operational costs in various applications.

8.6. Parallel Processing:

- MIPS architecture often supports parallel processing, allowing the execution of multiple instructions simultaneously. This parallelism enhances overall system throughput and contributes to efficient multitasking.

8.7. Improved System Responsiveness:

- Higher MIPS values contribute to improved system responsiveness, ensuring that users experience minimal delays when interacting with applications and services.

8.8. Scalability:

- MIPS technology is often scalable, accommodating advancements in semiconductor technology. This scalability allows for the development of processors with increasing MIPS counts, keeping pace with evolving computing demands.

8.9. Wide Applicability:

- The advantages of MIPS technology make it suitable for a broad range of applications, including servers, personal computers, mobile devices, and embedded systems, providing versatility in meeting diverse computing needs.

CHAPTER 9
MODULES OF THE PROCESSOR

9.1. FETCH

The fetch module in MIPS architecture is responsible for retrieving instructions from memory for execution. It operates by utilizing the Program Counter (PC) to supply the address of the current instruction to be fetched. The PC is typically initialized to the starting address of the program and increments by 4 after each fetch to point to the next instruction unless altered by branch or jump instructions.

Once an instruction is fetched, it is stored in the Instruction Register (IR) for subsequent decoding and execution in the pipeline. This module plays a critical role in the pipeline's overall efficiency by ensuring a steady flow of instructions into the processor, thereby enabling simultaneous processing in the subsequent stages of the pipeline.

9.2. INSTRUCTION MEMORY

The instruction memory serves as the storage unit for the program's instructions. It holds the binary representations of the instructions to be executed by the processor. The instruction memory is typically implemented as a separate component from the data memory, ensuring that program instructions are distinct from program data. During the fetch stage of the instruction processing cycle, the processor retrieves instructions from the instruction memory based on the value of the program counter (PC). Each instruction is fetched sequentially, and its binary representation is loaded into the instruction register (IR) for decoding and execution. The instruction memory thus provides the necessary program instructions to the processor, enabling the execution of the program's logic and operations as specified by the programmer.

9.3. CONTROL UNIT

In MIPS architecture, the control unit plays a pivotal role in coordinating the execution of instructions within the processor. It interprets each instruction fetched from memory, decoding its opcode and determining the sequence of operations needed to execute it. The control unit generates control signals that coordinate various components of the processor, such as the arithmetic logic unit (ALU), memory unit, and registers, to perform the necessary operations specified by the instruction. This includes managing data movement between registers, ALU operations, memory accesses, and control flow decisions. By orchestrating these operations, the control unit ensures the correct execution of instructions according to the program's logic, facilitating efficient processing and maintaining the integrity of the computation.

9.4. DECODE UNIT

The decode module is responsible for interpreting the fetched instruction, extracting relevant fields such as opcode and operands, and preparing the necessary control signals to execute the instruction. This stage decodes the binary representation of the instruction into signals that control the operation of subsequent stages in the pipeline. It identifies the type of instruction (e.g., arithmetic, load/store, branch) and extracts any immediate values or register addresses needed for execution. Additionally, the decode module may detect hazards such as data dependencies or control hazards and take appropriate actions, such as stalling the pipeline or forwarding data to resolve hazards efficiently. This module plays a crucial role in orchestrating the efficient execution of instructions within the MIPS pipeline, ensuring correct operation and maintaining performance.

9.5. EXECUTE UNIT

In this stage, the actual computation or data manipulation specified by the instruction takes place. Once an instruction has been fetched from memory and decoded, it enters the execute stage where arithmetic or logical operations are performed on data stored in registers or memory. This stage includes operations such as arithmetic computations (addition, subtraction, multiplication, division), logical operations (AND, OR, XOR), shifting, comparison, and address calculation. Additionally, the execute stage handles branch instructions by evaluating conditions and determining whether the program counter should be updated to jump to a different location in memory. Overall, the execute stage is responsible for carrying out the core computational tasks necessary to execute the instruction, preparing the result for storage or further processing in subsequent stages of the pipeline.

9.6. DATA MEMORY

In MIPS architecture, the Data Memory unit serves as the primary storage for data accessed by the processor during program execution. It stores both program data and variables, providing a location for read and write operations. The Data Memory unit interfaces with the processor to fetch and store data according to memory addresses provided by instructions. It handles various memory access operations, including loading data from memory into registers, storing data from registers into memory, and performing arithmetic or logical operations directly on

memory operands. Additionally, the Data Memory unit ensures data integrity by managing data alignment and handling memory protection mechanisms. Overall, it serves as a crucial component for storing and accessing data, facilitating efficient computation and program execution in the MIPS architecture.

CHAPTER 10
PERFORMANCE METRICS

10.1. Clock Speed: The rate at which a processor can execute instructions, measured in gigahertz (GHz).

10.2. Instructions Per Cycle (IPC): The average number of instructions a processor can execute per clock cycle. Higher IPC often indicates better efficiency.

10.3. Clock cycles Per Instruction (CPI): It represents the average number of clock cycles required to execute a single instruction. A lower CPI indicates better performance, as it means that instructions are executed more quickly on average.

- **CPU Execution Time = CPU clock cycles x clock cycle time**
- **CPU clock cycle = No. of Instructions x CPI** (Considering 7 instructions)
- **PERFORMANCE = 1 / Execution Time**

Execution Time	MACHINE CODE (HEX)
Clock cycle time	2ns
Clock cycle rate	0.5GHz
Execution Time	6ns
CPU Execution Time	14ns
CPU clock cycles	7
Clock cycles per Instruction	1
PERFORMANCE	**1.6667 x 10^8 Instructions per second**

Fig 9. PERFORMANCE METRICS

CHAPTER 11
TOOLS AND TECHNOLOGIES

Xilinx Vivado is an integrated development environment for FPGA and SoC design. It supports entry through schematics, HDL coding (VHDL, Verilog), and High-Level Synthesis (HLS). With IP Integrator, it eases IP block integration. Vivado handles synthesis, place and route, enabling the conversion of high-level designs into FPGA/SoC configurations. It supports system-level integration, debugging tools, and real-time profiling. Vivado accommodates programming methods like JTAG and Flash. The IDE integrates with Xilinx's ecosystem and third-party tools. Supporting both VHDL and Verilog, it also includes High-Level Synthesis for algorithm description in C or C++. Vivado is widely used for FPGA and SoC development due to its versatility and comprehensive feature set.

MODELSIM: In the realm of electronic design automation (EDA), ModelSim stands as a leading hardware design verification and debug platform. This robust simulator empowers engineers to construct a virtual environment, enabling the testing of hardware designs written in Hardware Description Languages (HDLs) such as Verilog, VHDL, and SystemC. Bypassing the need for physical prototyping, ModelSim facilitates efficient design verification and debugging iterations. This translates to significant cost and time savings throughout the development process. ModelSim's intuitive debugging environment equips engineers with advanced visualization and analysis tools, expediting the identification and rectification of errors within the HDL code. As an industry-standard solution, ModelSim streamlines the entire design verification and debug workflow, making it an invaluable asset for hardware development teams.

CHAPTER 12
SIMULATION PARAMETERS

There are many parameters that can be observed in the waveform window of the simulation. The parameters which are sufficient to verify the functionality of the processor are:

IR0 – IR3: These are the Interface Registers between each stage as shown in the block diagram, these indicate the stages through which instructions are passing.

ALU_Result: This is the 32-bit output generated by the Arithmetic & Logic Unit of the processor in response to the instructions given.

CHAPTER 13
RESULTS AND DISCUSSIONS

The transcript of Modelsim Altera for simulation of the MIPS processor model under test is as follows:

```
source MIPS_FILELIST
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:21 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/Instmem.v "+no_notifier" "+nospecify" "+notimingchecks"
# -- Compiling module InstMem
#
# Top level modules:
#   InstMem
# End time: 11:37:21 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:21 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IFetch.v "+no_notifier" "+nospecify" "+notimingchecks"
# -- Compiling module IFETCH
#
# Top level modules:
#   IFETCH
# End time: 11:37:22 on Mar 11,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:22 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IDecode.v "+no_notifier" "+nospecify" "+notimingchecks"
# -- Compiling module IDecode
#
# Top level modules:
#   IDecode
# End time: 11:37:22 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:22 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IControlUnit.v "+no_notifier" "+nospecify" "+notimingchecks"
# -- Compiling module IControlUnit
#
# Top level modules:
```

```
#   IControlUnit
# End time: 11:37:22 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:22 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IExecute.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module IExecute
#
# Top level modules:
#   IExecute
# End time: 11:37:22 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:22 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/DataMem.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module DataMem
#
# Top level modules:
#   DataMem
# End time: 11:37:22 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:22 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/DataMem256x32.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module DataMem256x32
#
# Top level modules:
#   DataMem256x32
# End time: 11:37:22 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:22 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IMux2x1.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module IMux2x1
#
# Top level modules:
#   IMux2x1
# End time: 11:37:22 on Mar 11,2024, Elapsed time: 0:00:00
```

```
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:22 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/WriteReg1.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module WriteReg1
#
# Top level modules:
#   WriteReg1
# End time: 11:37:23 on Mar 11,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:23 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/WriteReg.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module WriteReg
#
# Top level modules:
#   WriteReg
# End time: 11:37:23 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:23 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/Top_Risc.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module Top_Risc
#
# Top level modules:
#   Top_Risc
# End time: 11:37:23 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:37:23 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/T_Top_Risc.v
# -- Compiling module T_Top_Risc
#
# Top level modules:
#   T_Top_Risc
# End time: 11:37:23 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
vsim \+acc T_Top_Risc
# ** Warning: (vsim-8690) Ignoring plusarg '+acc' with -novopt in effect.
```

```
# vsim "+acc" T_Top_Risc
# Start time: 11:37:23 on Mar 11,2024
# Loading work.T_Top_Risc
# Loading work.Top_Risc
# Loading work.InstMem
# Loading work.IFETCH
# Loading work.IMux2x1
# Loading work.IControlUnit
# Loading work.IDecode
# Loading work.WriteReg
# Loading work.IExecute
# Loading work.DataMem
# Loading work.DataMem256x32
add wave  \
sim:/T_Top_Risc/RISC/FETCHUNIT/Clock
add wave  \
sim:/T_Top_Risc/RISC/DECODE/Rt \
sim:/T_Top_Risc/RISC/DECODE/Rs \
sim:/T_Top_Risc/RISC/DECODE/Rd \
sim:/T_Top_Risc/RISC/DECODE/IR3 \
sim:/T_Top_Risc/RISC/DECODE/IR2 \
sim:/T_Top_Risc/RISC/DECODE/IR1 \
sim:/T_Top_Risc/RISC/DECODE/IR0
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#           File in use by: User  Hostname: SLYCON  ProcessID: 12864
#           Attempting to use alternate WLF file "./wlftvyygvh".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#           Using alternate file: ./wlftvyygvh
source MIPS_FILELIST
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:04 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/Instmem.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module InstMem
#
# Top level modules:
#   InstMem
# End time: 11:38:04 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:04 on Mar 11,2024
```

```
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IFetch.v "+no_notifier" "+nospecify"
"+notimingchecks"
# -- Compiling module IFETCH
#
# Top level modules:
#   IFETCH
# End time: 11:38:05 on Mar 11,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:05 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IDecode.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module IDecode
#
# Top level modules:
#   IDecode
# End time: 11:38:05 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:05 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IControlUnit.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module IControlUnit
#
# Top level modules:
#   IControlUnit
# End time: 11:38:05 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:05 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IExecute.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module IExecute
#
# Top level modules:
#   IExecute
# End time: 11:38:05 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:05 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/DataMem.v "+no_notifier"
"+nospecify" "+notimingchecks"
```

```
# -- Compiling module DataMem
#
# Top level modules:
#   DataMem
# End time: 11:38:05 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:05 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/DataMem256x32.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module DataMem256x32
#
# Top level modules:
#   DataMem256x32
# End time: 11:38:05 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:05 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/IMux2x1.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module IMux2x1
#
# Top level modules:
#   IMux2x1
# End time: 11:38:06 on Mar 11,2024, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:06 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/WriteReg1.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module WriteReg1
#
# Top level modules:
#   WriteReg1
# End time: 11:38:06 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:06 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/WriteReg.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module WriteReg
#
```

```
# Top level modules:
#   WriteReg
# End time: 11:38:06 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:06 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/Top_Risc.v "+no_notifier"
"+nospecify" "+notimingchecks"
# -- Compiling module Top_Risc
#
# Top level modules:
#   Top_Risc
# End time: 11:38:06 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb 28 2020
# Start time: 11:38:06 on Mar 11,2024
# vlog -reportprogress 300 D:/Projects/MIPS_FILES/CODE_10JULY/T_Top_Risc.v
# -- Compiling module T_Top_Risc
#
# Top level modules:
#   T_Top_Risc
# End time: 11:38:06 on Mar 11,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
log -r /*
restart -f
run -all
# ** Note: (vsim-12125) Error and warning message counts have been reset to '0' because of 'restart'.
# Loading work.T_Top_Risc
# Loading work.Top_Risc
# Loading work.InstMem
# Loading work.IFETCH
# Loading work.IMux2x1
# Loading work.IControlUnit
# Loading work.IDecode
# Loading work.WriteReg
# Loading work.IExecute
# Loading work.DataMem
# Loading work.DataMem256x32
```

The object codes used for the simulation of the processor along with the output waveforms are as given below :

ASSEMBLY CODE	MACHINE CODE (HEX)	ASSEMBLY CODE	MACHINE CODE (HEX)
LUI $t0 0x0100	0x3c080100	ADDI $t2 $t1 0x0101	0x212A0101
LUI $t1 0x0101	0x3c090101	ORI $t5 $t0 0x1000	0x350D1000
ADD $t2 $t1 $t0	0x01285020	SRL $t3 $t0 0x0001	0x00085842
SUB $t3 $t1 $t0	0x01285022		

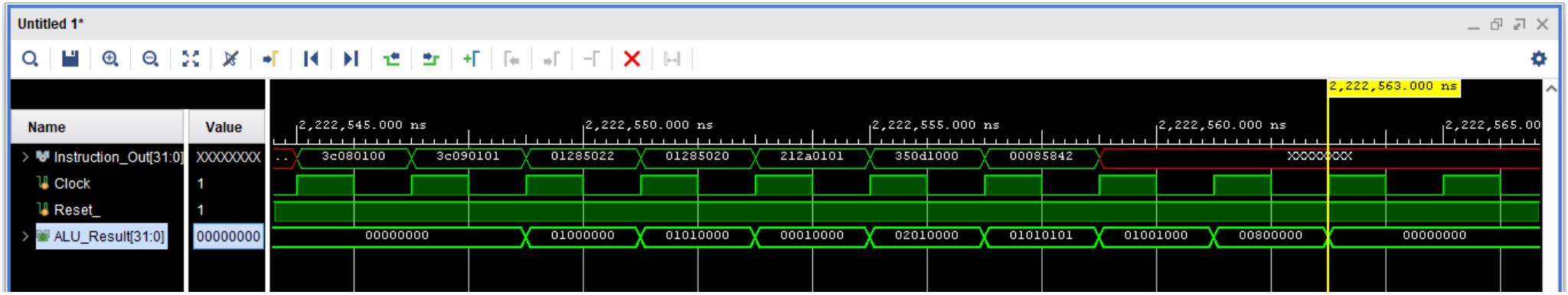


Fig 10. INSTRUCTIONS WITHOUT DEPENDECNY

ASSEMBLY CODE	MACHINE CODE (HEX)	RS	RT	RD
LUI $t0 0x0100	0x3c080100	0	8	0
LUI $t1 0x0101	0x3c090101	0	9	0
ADD $t2 $t1 $t0	0x01285020	9	8	0a
SUB $t3 $t2 $t1	0x01495822	0a	9	0b
ADD $t4 $t3 $t2	0x016a6020	0b	0a	0c

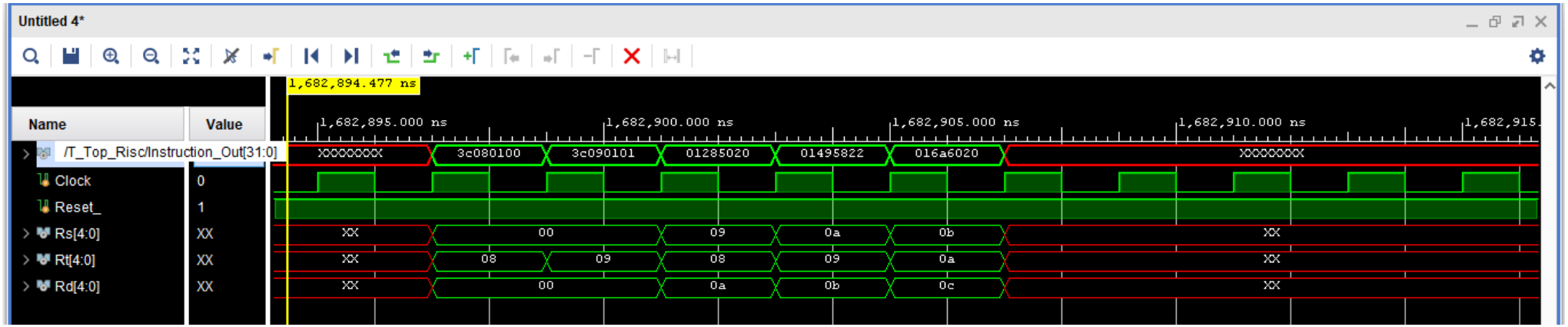


Fig 11. LOADING OF THE DATA INTO THE REGISTERS

ASSEMBLY CODE	MACHINE CODE (HEX)
LUI $t0 0x0100	0x3c080100
LUI $t1 0x0101	0x3c090101
ADD $t2 $t1 $t0	0x01285020
SUB $t3 $t0 $t2	0x010a5822

ASSEMBLY CODE	MACHINE CODE (HEX)
ADDI $t4 $t3 0x0101	0x216c0101
ORI $t3 $t4 0x1000	0x0358B100
SRL $t4 $t3 0x0001	0x000B6042

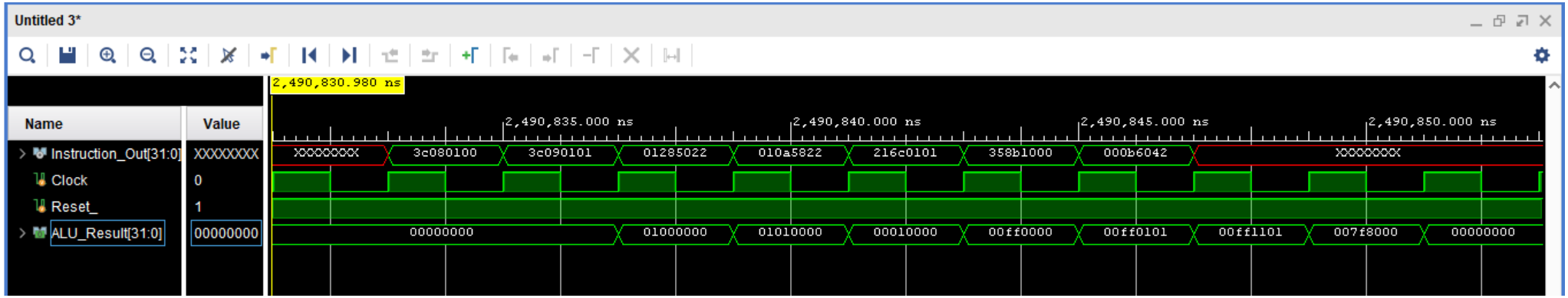
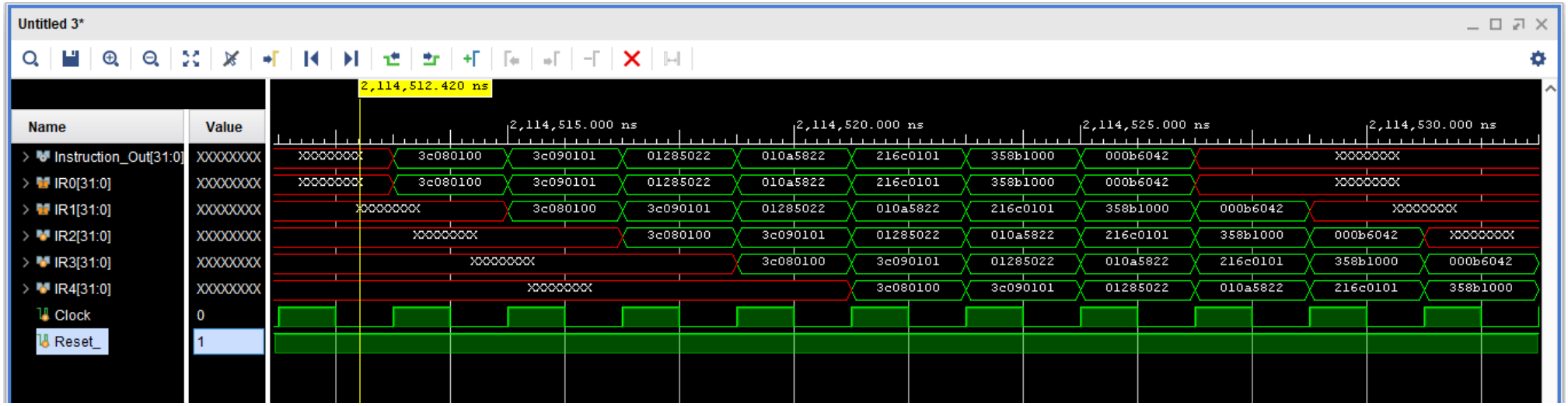
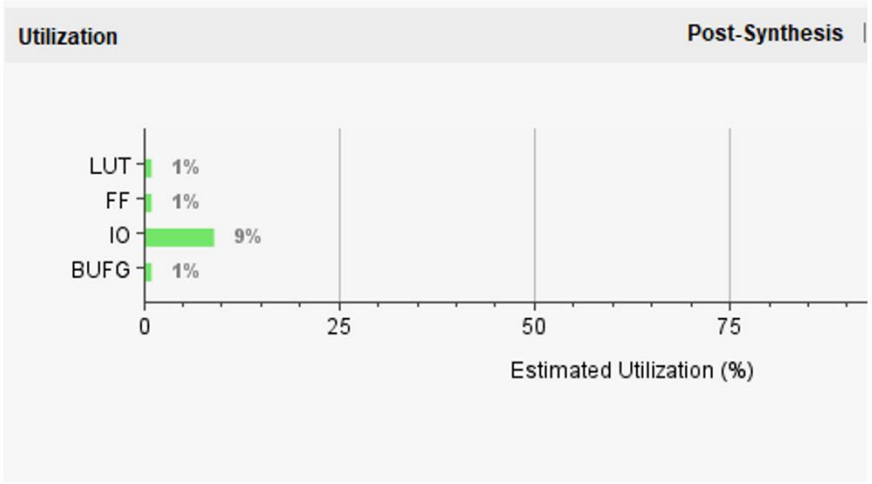


Fig 12. INSTRUCTIONS WITH DEPENDECNY


Untitled 3*
2,114,512.420 ns
Name
Value
Instruction_Out[31:0]
IR0[31:0]
IR1[31:0]
IR2[31:0]
IR3[31:0]
IR4[31:0]
Clock
Reset_
XXXXXXXX
0
1
2,114,515.000 ns
2,114,520.000 ns
2,114,525.000 ns
2,114,530.000 ns
3c080100
3c090101
01285022
010a5822
216c0101
358b1000
000b6042

C



Resource	Estimation	Available	Utilization %
LUT	679	230400	0.29
FF	666	460800	0.14
IO	34	360	9.44
BUFG	2	544	0.37

Fig 14. POST SYNTHESIS UTILIZATION REPORT

CHAPTER 14
PROGRAM CODE

Top Sheet of MIPS CORE:

Top sheet of MIPS core in this MIPS design instantiates all the major modules of the design such as;

- Instruction memory
- Fetch Unit
- Control Unit
- Decode Unit
- Execute Unit
- Data Memory

CODE:

```
module   Top_Risc
            (
             ///OUTPUTS
             Instruction_Out,
             ///INPUTS
             Clock,
             Reset_
            );

parameter PCWIDTH =8;
parameter DATAWIDTH =32;
input Reset_,Clock;
output[31:0] Instruction_Out;
wire [31:0]Instruction_Out;
wire MemWen,MemCen;
reg [31:0] ALU_Result_reg;
reg [31:0] ALUResult;
reg [31:0] Write_Back_Reg;
reg [31:0] Write_Back;
wire [31:0]IR1;
wire [31:0]IR0;
wire [31:0]MemOut
wire [PCWIDTH - 1: 0] PC_Out_Mem;
wire [PCWIDTH - 1: 0] PC_plus_4_ID;
wire [PCWIDTH - 1: 0] PC_plus_4_Out;
wire [7:0]  Branch_Addr,PC_Plus_4;
wire [DATAWIDTH-1:0]  IR2;
wire [4:0]  WriteAddr_EX;
wire [PCWIDTH - 4 :0] Rs_ID,Rt_ID,Rd_ID,Shift_ID,WriteAddr_WB;
wire [DATAWIDTH-1:0]  ALU_Result_Mem;
wire                                                    [DATAWIDTH-1:0]
WRBACK_Reg,Sign_Ext_Out_Reg,Read_reg_Out1,Read_reg_Out2;
```

```
wire [DATAWIDTH-17:0] Immediate_ID;
wire [DATAWIDTH-1:0] ALU_Result,Mem_Store_EX,result;
wire DepRes_Rs_Sel,DepRes_Rt_Sel;
wire DepRes_Rs_Sel_WB,DepRes_Rt_Sel_WB;

/////////TopSheet Output Signals//////////
assign Instruction_Out = IR0;

///////----------WRITE BACK STAGE OF THE PROCESSOR----------///////
always@(posedge Clock)
if(Reset_ == 1'b0)
        begin
        ALU_Result_reg <= 32'b0;
        end
else
       begin
        ALU_Result_reg <=  ALUResult;
        end
always@(posedge Clock)
if(Reset_)
        Write_Back_Reg  <= 32'b0;
else
        Write_Back_Reg <= Write_Back ;


InstMem      INSTMEM(

              /////OUTPUTS
              .IR0              (IR0),
              ////INPUTS
              .IFLUSH       (Branch_ID),
              .PC_Out_Mem   (PC_Out_Mem),
              .Clock            (Clock),
              .Reset_           (Reset_)
              );


IFETCH       FETCHUNIT (

             ///Outputs
             .PC_plus_4_Out  (PC_plus_4_Out),
             .PC_Out_Mem           (PC_Out_Mem),

             //Inputs
             .PC_Halt          (PC_Halt),
             .Branch_Addr          (Branch_Addr),   //EFFECTIVE BRANCH ADDRESS
             .Branch_ID            (Branch_ID),
             .Clock                (Clock),
             .Reset_               (Reset_)

             );
```

```
IControlUnit    CONTROL(

                ///INPUT////
                .Opcode                 (IR0),
                .IFLUSH             (Branch_ID),

               ///OUTPUT/////
                .IR1                    (IR1),
                .Byte_reg               (Byte_reg),
                .ByteU_reg              (ByteU_reg),
                .HalfWord_reg           (HalfWord_reg),
                .Word_reg               (Word_reg),
                .StoreByte_reg          (StoreByte_reg),
                .StoreWord_reg          (StoreWord_reg),
                .StoreHalfWord_reg      (StoreHalfWord_reg),
                .MemWrite_reg           (MemWrite_reg),
                .MemRead_reg            (MemRead_reg),
                .RegDst_reg             (RegDst_reg),
                .MemtoReg_reg           (MemtoReg_reg),
                .RegWrite_reg           (RegWrite_reg),
                .ALUSrc_reg             (ALUSrc),
                .Clock                  (Clock),
                .Reset_                 (Reset_)

                );


IDecode         DECODE  (

                ////INPUTS   THIS MODULE ALSO CONTAINS THE REGFILE   ////////

                .IR0                    (IR0),
                .IR1            (IR1),
                .ALUResult              (ALU_Result_Mem),
                .WRBack_Data            (WRBACK_Reg),
                .MemtoReg                       (MemtoReg_ID),
                .RegFileWr              (RegWrite_Mem),
                .Clock                  (Clock),
                .Reset_                 (Reset_),
                .PC_plus_4              (PC_Out_Mem),
                .WriteAddr_WB           (WriteAddr_WB),

                ////OUTPUTS

                .DepRes_Rt_Sel_EX       (DepRes_Rt_Sel_EX),
                .DepRes_Rs_Sel_EX       (DepRes_Rs_Sel_EX),
                .DepRes_Rt_Sel          (DepRes_Rt_Sel),
                .DepRes_Rs_Sel          (DepRes_Rs_Sel),
                .DepRes_Rt_LdSt_EX      (DepRes_Rt_LdSt_EX),
                .DepRes_Rs_LdSt_EX      (DepRes_Rs_LdSt_EX),
                .DepRes_Rt_LdSt_WB_EX   (DepRes_Rt_LdSt_WB_EX),
                .DepRes_Rs_LdSt_WB_EX   (DepRes_Rs_LdSt_WB_EX),
                .PC_Halt            (PC_Halt),
                .Branch_Addr            (Branch_Addr),
                .Branch_ID              (Branch_ID),
```

```
    .Rs_ID                  (Rs_ID),
    .Rt_ID                  (Rt_ID),
    .Rd_ID                  (Rd_ID),
    .Immediate_ID           (Immediate_ID),
    .Shift_ID               (Shift_ID),
    .Sign_Ext_Out_Reg               (Sign_Ext_Out_Reg),
    .Read_reg_Out1          (Read_reg_Out1),
    .Read_reg_Out2          (Read_reg_Out2)

);

IExecute    EXECUTE (

    ///INPUTS

    .DepRes_Rt_Sel_EX       (DepRes_Rt_Sel_EX),
    .DepRes_Rs_Sel_EX       (DepRes_Rs_Sel_EX),
    .DepRes_Rt_Sel          (DepRes_Rt_Sel),
    .DepRes_Rs_Sel          (DepRes_Rs_Sel),
    .DepRes_Rt_LdSt_EX      (DepRes_Rt_LdSt_EX),
    .DepRes_Rs_LdSt_EX      (DepRes_Rs_LdSt_EX),
    .DepRes_Rt_LdSt_WB_EX   (DepRes_Rt_LdSt_WB_EX),
    .DepRes_Rs_LdSt_WB_EX   (DepRes_Rs_LdSt_WB_EX),
    .ALU_Result_Mem         (ALU_Result_Mem),
    .ALU_Input1             (Read_reg_Out1),
    .Read_data_2            (Read_reg_Out2),
    .Sign_Extend            (Sign_Ext_Out_Reg),
    .MemOut             (MemOut),
    .opcode                 (IR1[31:26]),   //MSB 6 Bits of Inst for comparison
    .Rs_EX                  (Rs_ID), //Address rt of contents2
    .Rt_EX                  (Rt_ID), //Address rt of contents2
    .Rd_EX                  (Rd_ID), //Address rd of destination register
    .Immediate              (Immediate_ID),  //Immediate field  Imm
    .Shift                  (Shift_ID),         //Shift value
    .Function_opcode        (IR1[5:0]),
    .Byte_reg               (Byte_reg),
    .ByteU_reg              (ByteU_reg),
    .HalfWord_reg           (HalfWord_reg),
    .Word_reg               (Word_reg),
    .StoreByte_reg          (StoreByte_reg),
    .StoreWord_reg          (StoreWord_reg),
    .StoreHalfWord_reg      (StoreHalfWord_reg),
    .MemWrite_reg           (MemWrite_reg),
    .MemRead_reg            (MemRead_reg),
    .MemtoReg_reg           (MemtoReg_reg),
    .RegWrite_reg           (RegWrite_reg),
    .RegDst_EX              (RegDst_reg),
    .ALUSrc                 (ALUSrc),
    .Clock                  (Clock),
    .Reset_                 (Reset_),

    ///OUTPUTS

    .Byte_EX                (Byte_EX),
    .ByteU_EX                       (ByteU_EX) ,
```

```
                .HalfWord_EX            (HalfWord_EX) ,
                .Word_EX                        (Word_EX) ,
                .MemWrite_EX            (MemWrite_EX) ,
                .MemRead_EX             (MemRead_EX),
                .MemtoReg_EX            (MemtoReg_EX),
                .RegWrite_EX            (RegWrite_EX),
                .result                 (result),
                .ALU_Result             (ALU_Result),
                .WriteAddr_EX           (WriteAddr_EX),
                .Mem_Store_EX           (Mem_Store_EX) );
 );

DataMem         DATAMEM (

                 /////INPUTS
                .Byte_EX                (Byte_EX),
                .ByteU_EX                       (ByteU_EX) ,
                .HalfWord_EX            (HalfWord_EX) ,
                .Word_EX                        (Word_EX) ,
                .MemWrite_EX            (MemWrite_EX) ,
                .MemRead_EX             (MemRead_EX),
                .MemtoReg_EX            (MemtoReg_EX),
                .ALU_Result             (ALU_Result),
                .WriteAddr_EX           (WriteAddr_EX),
                .RegWrite_EX            (RegWrite_EX),
                .Mem_Store              (Mem_Store_EX),
                .Clock          (Clock),
                .Reset_         (Reset_),

               //////OUTPUTS

                .WriteAddr_WB       (WriteAddr_WB),
                .MemtoReg_ID        (MemtoReg_ID),
                .MemOut         (MemOut),
                .WRBACK_Reg         (WRBACK_Reg),  ////Output of data Mem
                .RegWrite_Mem       (RegWrite_Mem),
                .ALU_Result_Mem     (ALU_Result_Mem)

                );
endmodule;
```

REFERENCES

[1] Patterson, David A., and John L. Hennessy. *Computer Organization and Design*: The Hardware/Software Interface. Fifth Edition. Morgan Kaufmann, 2013. p. 287.

[2] Kanaka Sai Hemanth Dogga, Chand Basha Shaik, Sreemukhi Muddusetty "Design of Instruction Set Architecture Based 16 Bit MIPS Architecture with Pipeline Stages", *International Research Journal of Engineering and Technology* (IRJET), Volume: 0, Issue: 07, page no.2581-2583, July 2021.

[3] Tripti Mahajan, Prof. Nikhil P. Wyawahare "Review of Low Power Multicycle MIPS Processor Using HDL" *Journal of Emerging Technologies and Innovative Research* (JETIR), Volume 6, Issue 4, page no.418-421, April 2019.

[4] Ms. Ashwini Golghate, Vishal Jaiswal " A Review on MIPS RISC Processor ", *International Journal of Innovative Research in Technology* (IJIRT), Volume 4, Issue 12, page no.469-472, May-2018.

[5] Galani Tina G., Riya Saini and R.D.Daruwala "Design and Implementation of 32 – bit RISC Processor using Xilinx", *International Journal of Emerging Trends in Electrical and Electronics* (IJETEE) Vol. 5, Issue. 1, July-2013.

[6] Sagar Bhavsar, Akhil Rao, Abhishek Sen, Rohan Joshi "A 16-bit MIPS Based Instruction Set Architecture for RISC Processor", I*nternational Journal of Scientific and Research Publications*, Volume 3, Issue 4, April 2013.