PROJECT REPORT

on

TESTBENCH AUTOMATION ON A MIPS-BASED MICROPROCESSOR IMPLEMENTED IN VERILOG

Submitted by

Nidhin Chandran

(Reg no:20320054)

in partial fulfillment of requirement for the award of the degree

Of

BACHELOR OF TECHNOLOGY

In

ELECTRONICS AND COMMUNICATION

DIVISION OF ELECTRONICS ENGINEERING

SCHOOL OF ENGINEERING

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

APRIL 2024

KOCHI -682022

DIVISION OF ELECTRONICS ENGINEERING

SCHOOL OF ENGINEERING

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

KOCHI-682022

CERTIFICATE

*Certified that the Project report entitled "**TEST BENCH AUTOMATION ON A MIPS-BASED MICROPROCESSOR IMPLEMENTED IN VERILOG**" is a bonafide work of **Nidhin Chandran(Reg no:20320054)** towards the partial fulfillment for the award of the degree of B.Tech in Electronics and Communication of Cochin University of Science and Technology, Kochi-682022.*

Project Coordinator

Dr. Shahana T K

Head of the Division

Dr. Anju Pradeep

ABSTRACT

Microprocessor verification is a critical aspect of modern microprocessor design, necessitating rigorous testing to ensure correct functionality. This project addresses the challenges associated with the manual creation of testbenches for a microprocessor implementing a 5-stage pipeline architecture. The chosen architecture, based on the MIPS 32 instruction set, is renowned for its simplicity and efficiency in enhancing performance.

The primary objective of this project is to streamline the microprocessor verification process by developing a Python script capable of automating testbench generation from assembly programs. Manual testbench creation for complex microprocessors can be time-consuming and error-prone, hindering development efficiency. By automating this process, the project aims to significantly reduce development time and enhance productivity.

The microprocessor's 5-stage pipeline architecture divides instruction execution into discrete stages: instruction fetch, decode, execute, memory access, and write-back. This approach facilitates concurrent instruction processing, thereby improving throughput. Leveraging the advantages of the MIPS 32 architecture, the project aims to create a comprehensive solution for microprocessor verification.

The Python script developed for this purpose plays a pivotal role in the transformation of human-readable assembly code into its corresponding 32-bit instruction representation. Given that the microprocessor operates on 32-bit instructions, the script ensures seamless translation of assembly code into machine-readable instructions. This translation is crucial for the microprocessor to execute programs accurately.

In summary, this project presents an innovative solution for microprocessor verification by automating the generation of testbenches from assembly programs. The incorporation of a Python script not only reduces the complexity and potential errors associated with manual testbench creation but also contributes to the efficient verification of a MIPS 32 microprocessor implementing a 5-stage pipeline architecture

TABLE OF CONTENT

TABLES OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

In the realm of microprocessor design, our project undertakes the ambitious task of revolutionising the verification process for a Verilog-implemented microprocessor. The crux of our endeavour lies in harnessing the power of automation through a sophisticated Python script, meticulously crafted for the automated generation of testbenches from assembly programs. The driving force behind this initiative is rooted in the recognition that manual testbench creation for intricate microprocessors is a laborious and error-prone endeavour, demanding extensive time and expertise.

The microprocessor under development stands as a testament to contemporary design principles, adhering to a robust 5-stage pipeline architecture. This well-established approach in modern microprocessor design elegantly dissects instruction execution into distinct stages: instruction fetch, decode, execute, memory access, and write-back. Such segmentation not only enhances the overall performance of the microprocessor but also facilitates concurrent instruction processing, ushering in a new era of improved throughput and efficiency.

Central to our project is the strategic selection of the MIPS 32 architecture—a revered paradigm recognized for its simplicity and efficiency. This architectural choice not only aligns with industry best practices but also sets the stage for our primary objective. At the forefront of our endeavours stands a Python script meticulously crafted to orchestrate the transformation of assembly language code into its machine-readable counterpart—32-bit instructions. As the microprocessor inherently operates on 32-bit instructions, this script assumes a pivotal role in seamlessly bridging the gap between the human-readable assembly code, serving as the embodiment of program logic, and the machine-readable instructions essential for execution.

In essence, our project emerges as a comprehensive solution to the challenges posed by manual testbench creation. Through the amalgamation of cutting-edge microprocessor design principles, automation via Python scripting, and the efficiency of MIPS 32 architecture, we envision a paradigm shift in microprocessor verification. The journey is not merely about reducing development time and enhancing productivity but about redefining the landscape of microprocessor design with innovation, precision, and efficiency at its core.

CHAPTER 2

BASIC OBJECTIVES AND EXPLANATION OF PROJECT

The project aims to be executed in two distinct phases, focusing first on the hardware design of a 32-bit microprocessor based on the MIPS 32 instruction set with a 5-stage pipeline architecture. The second phase will involve the development of a Python script for automated testbench generation.

2.1 Hardware Design:

1) **Microprocessor Architecture:** Develop a detailed design for a 32-bit microprocessor, adhering to the MIPS 32 instruction set. Define the architecture, instruction formats, and data paths to ensure compatibility with the chosen instruction set.[1]
2) **5-Stage Pipeline Implementation:** Implement a 5-stage pipeline architecture, comprising instruction fetch, decode, execute, memory access, and write-back stages. Optimise each stage to facilitate concurrent instruction processing and enhance overall microprocessor throughput.[3]
3) **ALU (Arithmetic Logic Unit) Design:** Design and integrate a robust ALU capable of executing arithmetic and logical operations specified by the MIPS 32 instruction set. Ensure the ALU supports a diverse range of operations and adheres to performance requirements.
4) **Instruction Set Implementation:** Realise the complete set of instructions specified by the MIPS 32 architecture in hardware, ensuring accurate execution of assembly programs on the microprocessor.[3]
5) **Data Path and Control Unit Design:** Design and integrate the data path and control unit, ensuring efficient data flow and control signals within the microprocessor. Optimise for performance while meeting the architectural requirements.

6) **Register File and Memory Integration:** Incorporate a register file for temporary data storage and seamless interaction with memory components. Integrate data and instruction memory units to enable program execution.
7) **Testing and Validation:** Conduct thorough testing and validation of the designed microprocessor to ensure correct functionality and adherence to the MIPS 32 architecture. Verify the accuracy of instruction execution, proper functioning of the 5-stage pipeline, and ALU operations.

2.2 Python Script Development:[9]

1) **Automated Testbench Generation Script:** Develop a Python script capable of automatically generating testbenches for assembly programs written for the MIPS 32 microprocessor. Ensure the script efficiently interfaces with the hardware design, translating human-readable assembly code into machine-readable 32-bit instructions.
2) **Integration with Microprocessor:** Seamlessly integrate the Python script with the designed microprocessor, establishing a robust connection between the automated testbench generation tool and the hardware implementation. Ensure compatibility and reliable communication.
3) **Script Optimization, Flexibility, and Operator Assignment:** Optimise the Python script for efficiency and flexibility, allowing it to handle a variety of assembly programs and test cases. Implement user-friendly features, provide clear documentation, and specify operator assignments for ALU operations.
4) **Verification of Automated Testbenches:** Verify the correctness of the automated testbenches generated by the Python script by comparing the results with manually created testbenches. Ensure that the automated approach maintains the integrity of the verification process.

By systematically achieving these objectives, the project aims to deliver a fully functional and verified 32-bit microprocessor with a 5-stage pipeline architecture, including a well-designed ALU, coupled with an efficient Python script for automated testbench generation. This holistic approach contributes to advancing microprocessor design and verification methodologies.[9]

CHAPTER 3

MOTIVATION BEHIND THE PROJECT

The motivation behind undertaking this project is rooted in the critical need for efficient and reliable microprocessor verification methodologies in the field of computer architecture and design. The development of a 32-bit microprocessor based on the MIPS 32 instruction set with a 5-stage pipeline architecture, coupled with an advanced Python script for automated testbench generation, addresses several key challenges and aligns with the following motivational factors:[3]

1) **Complexity of Microprocessor Verification:** The verification of microprocessors is a complex and resource-intensive task, particularly when dealing with sophisticated architectures such as a 5-stage pipeline. Manually creating testbenches for diverse assembly programs is time-consuming, error-prone, and demands significant expertise[4]. The project seeks to alleviate this complexity by automating the testbench generation process, thereby enhancing the overall efficiency of microprocessor verification.[3]
2) **Performance Optimization through Pipeline Architecture:** The adoption of a 5-stage pipeline architecture is motivated by the desire to enhance microprocessor performance. By breaking down instruction execution into discrete stages, concurrent processing becomes possible, leading to improved throughput. The project aims to contribute to the advancement of pipeline architecture implementation, addressing challenges and optimising the design for maximum efficiency.
3) **Significance of the MIPS 32 Instruction Set[3]:** The selection of the MIPS 32 instruction set is motivated by its reputation for simplicity and efficiency. Leveraging this well-established architecture not only ensures the streamlined implementation of instructions but also facilitates a smooth transition from assembly code to machine-readable instructions. The project's focus on MIPS 32 aligns with the industry's preference for architectures that balance complexity and performance.
4) **Automation for Development Time Reduction:** The automation of testbench generation in the second phase is driven by the recognition that manual creation of testbenches for diverse assembly programs can be a bottleneck in the development process. Automation is expected to significantly reduce development time, enabling

quicker iterations, and allowing designers to focus more on refining the microprocessor's architecture and features.

3.1 Problem definition

Microprocessor design forms the backbone of modern computing systems, and ensuring the correct functionality of these processors is imperative. Microprocessor verification involves rigorous testing to identify and rectify potential issues, particularly in complex architectures like the 5-stage pipeline.

1) **Challenges in Manual Testbench Creation:**The manual creation of testbenches for microprocessor verification poses substantial challenges. In the context of a 5-stage pipeline architecture, the intricacies involved in creating comprehensive testbenches can lead to time-consuming processes and introduce the risk of errors. This manual approach hinders development efficiency and may impede progress in microprocessor design.[8]
2) **Motivation for Automation:**The chosen architecture, based on the MIPS 32 instruction set, is selected for its well-established reputation for simplicity and efficiency. However, the manual creation of testbenches remains a bottleneck in the verification process. The primary motivation for this project is to automate the testbench generation from assembly programs, thereby overcoming the challenges associated with manual methods.
3) **Primary Objective:**The core objective of this project is to streamline the microprocessor verification process through the development of a Python script capable of automating testbench generation. The focus is on alleviating the time-consuming and error-prone nature of manual testbench creation, ultimately enhancing development efficiency and productivity.[5]
4) **Significance of 5-Stage Pipeline Architecture:**The microprocessor under consideration employs a 5-stage pipeline architecture, a design choice that breaks down instruction execution into discrete stages. These stages—instruction fetch, decode, execute, memory access, and write-back—facilitate concurrent instruction processing, enhancing overall throughput. Leveraging the advantages of this architecture, the project aims to contribute to the efficiency of microprocessor verification.[4]

5) **Leveraging MIPS 32 Architecture:**The MIPS 32 instruction set is chosen for its simplicity and efficiency, aligning with the project's objective to create a comprehensive solution for microprocessor verification. The script development focuses on translating human-readable assembly code into its corresponding 32-bit instruction representation, crucial for accurate program execution on the microprocessor.[7]
6) **Role of Python Script:**The Python script developed for automating testbench generation plays a pivotal role in bridging the gap between human-readable assembly code and machine-readable 32-bit instructions. Given the microprocessor's reliance on 32-bit instructions, the script ensures a seamless translation process, allowing the microprocessor to accurately execute programs.

In summary, the project addresses the challenges of manual testbench creation in the context of a 5-stage pipeline microprocessor architecture based on the MIPS 32 instruction set. The primary objective is to enhance efficiency and productivity in microprocessor verification by automating the generation of testbenches through a Python script. This innovation not only reduces the complexity and potential errors associated with manual processes but also contributes to the overall efficiency of verifying a MIPS 32 microprocessor implementing a 5-stage pipeline architecture.[4]

3.2 Challenges[5]

1) **Complexity of 5-Stage Pipeline Architecture:**

   Problem: The intricacies of verifying a microprocessor with a 5-stage pipeline architecture add complexity to the automation process.

   Challenge: Developing an automated testbench generation script that comprehensively addresses the nuances of each pipeline stage and their interdependencies.

2) **Diverse Instruction Set of MIPS 32:**

   Problem: The MIPS 32 instruction set encompasses a wide range of instructions, each with specific functionalities.

Challenge: Designing the script to handle the diverse set of MIPS 32 instructions and ensuring accurate translation from assembly to 32-bit machine-readable code.

3) **Data Hazards in Concurrent Instruction Processing:**

Problem: The concurrent processing nature of a 5-stage pipeline introduces data hazards.

Challenge: Implementing mechanisms within the script to detect and mitigate data hazards, ensuring accurate and hazard-free execution of instructions.

4) Control Hazards and Branch Prediction:

Problem: Control hazards, particularly in branches, can impact the flow of instructions and introduce delays.

Challenge: Incorporating branch prediction mechanisms into the script to optimise testbenches for control hazards and improve overall pipeline efficiency.

5) Optimising ALU Operations:

Problem: The Arithmetic Logic Unit (ALU) must support a diverse range of operations specified by MIPS 32.

Challenge: Ensuring the ALU design is optimised for performance without compromising accuracy, accommodating various arithmetic and logical operations.

6) Efficient Handling of Memory Access:

Problem: Memory access, including interactions with the register file and external memory units, requires synchronisation.

Challenge: Designing the script to efficiently manage memory access, minimising conflicts, and ensuring accurate data retrieval and storage.

7) Pipeline Stall Handling:

Problem: Situations like data hazards or branch mispredictions may necessitate pipeline stalls.

Challenge: Implementing stall handling mechanisms in the script to accurately reflect pipeline stalls during testbench generation.

8) **Simulation and Debugging:**

Problem: Simulation and debugging are critical for verifying the correctness of the microprocessor design.

Challenge: Developing robust debugging features within the script and ensuring compatibility with simulation tools for effective testing and validation.

9) **Flexibility for Varied Assembly Programs:**

Problem: Assembly programs can vary widely in complexity and structure.

Challenge: Ensuring the script is flexible enough to handle diverse assembly programs, accommodating different program structures and minimising the need for manual adjustments.

Addressing these challenges is crucial for the successful development of the automated testbench generation script and, consequently, for achieving the overarching goal of streamlining microprocessor verification in the context of a 5-stage pipeline architecture based on the MIPS 32 instruction set.

CHAPTER 4

METHODOLOGY OF IMPLEMENTATION

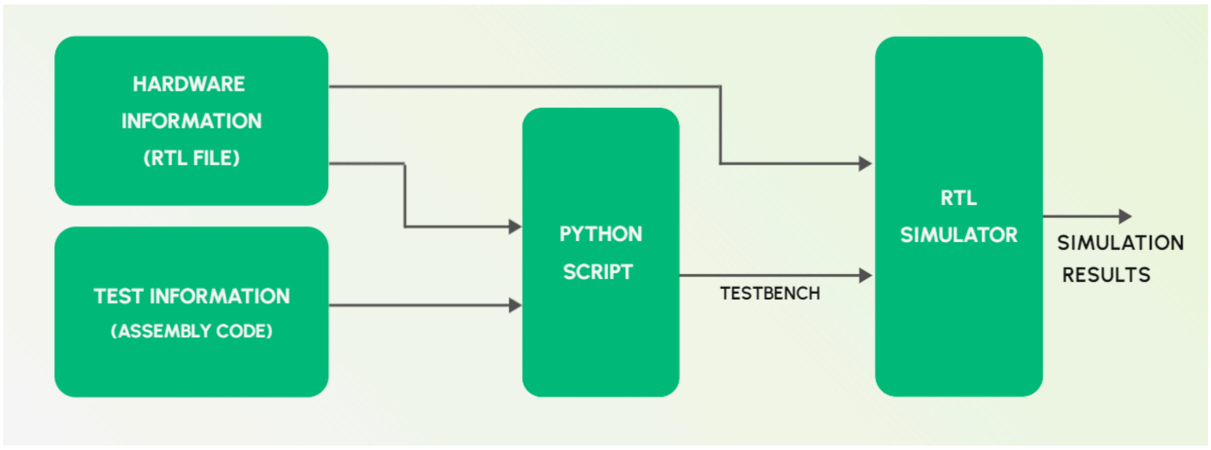

HARDWARE INFORMATION (RTL FILE)
TEST INFORMATION (ASSEMBLY CODE)
PYTHON SCRIPT
TESTBENCH
RTL SIMULATOR
SIMULATION RESULTS

Figure 4.1 *:Method of implementation*

1) **Hardware Information (RTL File):** This box refers to the hardware description of the device under test (DUT). RTL stands for Register Transfer Level, which is a level of abstraction used in describing the operation of a synchronous digital circuit. An RTL file typically contains the hardware description language (HDL) code, such as VHDL or Verilog, which defines the logic and structure of the hardware.[7]
2) **Test Information (Assembly Code):** This likely refers to the specific tests or instructions written in assembly language that will be used to verify the functionality of the hardware. Assembly code is a low-level programming language that is closely related to machine code, and in this context, it is used to write test cases that simulate the operation of the hardware.[7]
3) **Python Script:** This denotes a script written in Python that probably acts as a tool or bridge. The script's role is to automatically generate a testbench based on the hardware information and test information. A testbench is an environment used to apply test vectors to the hardware design and to verify its correctness.[5]
4) **Testbench:** The testbench is an automatically generated outcome from the Python script. It is used to verify the functionality and performance of the RTL model. The

testbench contains the stimuli required to test the DUT and is crucial in verifying that the hardware design behaves as expected under different scenarios.[6]

5) **RTL Simulator:** This is a software tool that simulates the behaviour of the RTL hardware model. It takes the testbench as input and simulates the hardware's response to the test vectors. The RTL simulator helps to catch functional errors and ensure that the design meets its specifications before any actual hardware is fabricated.[6]
6) **Simulation Results:** These are the outcomes from the RTL simulator, providing feedback on whether the hardware model passes or fails the tests defined in the testbench. The results are used to analyse the performance and behaviour of the design under test conditions.

Hardware Information (RTL File) describes the DUT's operation in VHDL or Verilog. Test Information (Assembly Code) contains assembly tests. Python Script generates testbenches. Testbench verifies RTL using an RTL Simulator, yielding Simulation Results.

4.1 Architecture(processor)

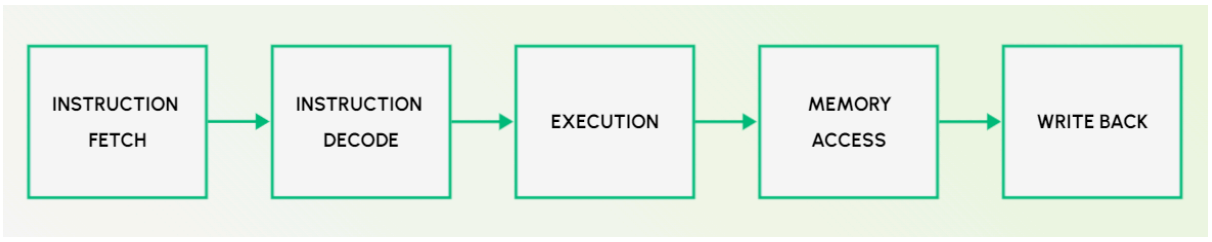


Figure 4.1.2: *Pipeline architecture*

The architecture described in the image pertains to a MIPS processor with the following specifications:[7]

1) **Processor Type:** It is a 32-bit processor, indicating that it can process data and memory addresses that are 32 bits wide.
2) **Internal Memory:** The processor has 4 GB of internal memory, which suggests a large on-chip memory capacity, although typically, this may refer to the addressable external memory range.

3) **Stack Memory:** There is a stack memory configuration of 32 x 32, which could imply a stack depth of 32 levels with each level being 32 bits wide, suitable for storing temporary data and addresses during execution.
4) **General-Purpose Registers:** The processor has 32 general-purpose registers labeled R0 to R31. Registers are small storage locations within the CPU that are used to quickly access and store data that the processor needs to perform operations.
5) **Pipelining:** The processor uses a 5-stage pipeline to improve instruction throughput. The stages include instruction fetch, instruction decode, execution, memory access, and write back. Pipelining allows multiple instructions to be processed simultaneously at different stages.
6) **Clock Speed:** The clock speed of the processor is 100 MHz, determining how many operations per second the processor can perform.

The instruction processing path depicted indicates a classic 5-stage instruction cycle:[8]

1) Instruction Fetch (IF)

The processor retrieves the next instruction from memory by utilizing the program counter (PC). The PC holds the address of the instruction to be fetched. Once obtained, the PC is updated to point to the next sequential instruction, preparing for the subsequent fetch.

2) Instruction Decode (ID)

The fetched instruction is decoded during this stage to determine the specific operation it represents and identify the operands involved. Operand addresses are computed, and control signals are generated to guide subsequent stages. This stage essentially translates the instruction into actionable information for the processor.

3) Execution (EX)

In the execution stage, the actual computation specified by the instruction is carried out. This involves arithmetic and logic operations, as well as the evaluation of branch conditions. The Arithmetic Logic Unit (ALU) plays a central role in executing a diverse range of operations, contributing to the versatility of the processor.

4) Memory Access (MEM)

If the instruction requires interaction with memory, the memory access stage is activated. For load (LW) or store (SW) instructions, data is either read from or written to memory. The interaction typically involves data transfer between the processor and memory, often utilising the data cache for efficiency.

5) Write Back (WB)

The final results of the executed instruction are written back to the register file or memory during the write-back stage. This is the concluding step where the outcome of the computation or memory operation is stored in the appropriate location, completing the instruction cycle.

4.2 Pipeline Advantages and hazards

Pipeline Advantages:[5]

- Concurrent Processing: Different instructions progress simultaneously through different stages, enhancing overall throughput.
- Resource Utilization: While one instruction is being executed, the pipeline concurrently fetches the next one, maximizing resource usage.
- Performance Improvement: Pipelining improves the microprocessor's overall performance by enabling a continuous flow of instruction processing.

Pipeline Hazards:

- Data Hazards: Dependencies between instructions that may lead to stalls or incorrect results due to reading data before it's written.
- Control Hazards: Arise with branch instructions, often resulting in pipeline flushes or stalls until the branch condition is resolved.
- Structural Hazards: Resource conflicts, such as attempting to read and write to the same register simultaneously, may cause stalls or require careful management.

In summary, a microprocessor's 5-stage pipeline architecture optimizes instruction execution by dividing the process into stages: Instruction Fetch (IF), Instruction Decode (ID), Execution

(EX), Memory Access (MEM), and Write Back (WB). Each stage contributes to efficient processing, allowing for concurrent execution of instructions. The pipeline enhances resource utilization, throughput, and overall performance. Despite its advantages, pipeline hazards, such as data hazards and control hazards, need careful management. Understanding and optimizing these stages is crucial for designing microprocessors that meet performance and efficiency goals.

4.3 Python Script

The Python script used is to mimic the functions of an assembler in a simplest way. The script consists of a lookup table which will map the given assembly code to testbench. Also the script acknowledges the syntax of the assembly code and intercepts whenever the code contains some error. This Python script reads an assembly file (specified by the user) containing MIPS instructions and translates them into their corresponding binary representations, following specific encoding rules. It defines functions to identify instruction types, determine instruction binary codes, and convert register values into binary representations. The script iterates through each line of the assembly file, parsing instructions, extracting relevant information, and writing the binary codes into a Verilog file. The Verilog file is structured to simulate a MIPS processor testbench, including clock signals and memory initialization. Additionally, error handling is implemented to catch syntax errors in the assembly file. Upon completion, the script outputs either a success message with the generated Verilog file or error messages indicating the number and type of mistakes encountered during the translation process. Overall, this script automates the conversion of MIPS assembly code into Verilog testbench code for further simulation and testing.[9]

4.4 Tools Used

1) Vivado

Vivado stands as a cornerstone in the realm of FPGA development, meticulously crafted by Xilinx to empower engineers and designers in crafting cutting-edge digital systems. Offering a rich array of tools and functionalities, Vivado serves as a comprehensive environment for the entire FPGA design process. From initial conception to final implementation, Vivado facilitates each step with precision and efficiency. Its RTL synthesis capabilities enable the translation of high-level design descriptions into optimized hardware descriptions. Moreover, its robust simulation features allow engineers to validate and refine their designs before committing to hardware. Vivado's place-and-route algorithms ensure optimal resource utilization and timing closure, crucial for achieving high-performance FPGA designs. With its intuitive interface and powerful capabilities, Vivado continues to be the preferred choice for FPGA development across a multitude of industries, driving innovation and pushing the boundaries of digital design.[[2]

2) PyCharm

PyCharm stands as a pinnacle in the landscape of integrated development environments (IDEs) tailored specifically for Python. Developed by JetBrains, PyCharm offers a wealth of features and tools designed to streamline the Python development workflow. Its intuitive interface and smart code editor empower developers to write clean, efficient code with ease. PyCharm's robust debugging capabilities allow for seamless troubleshooting and error resolution, enhancing productivity throughout the development process. Moreover, its extensive support for popular frameworks such as Django, Flask, and Pyramid accelerates web development projects. With features like intelligent code completion, code inspection, and version control integration, PyCharm equips developers with everything they need to build, test, and deploy Python applications swiftly and efficiently. As a result, PyCharm has become an indispensable tool for Python developers worldwide, fostering innovation and driving the evolution of Python-based software solutions.[2]

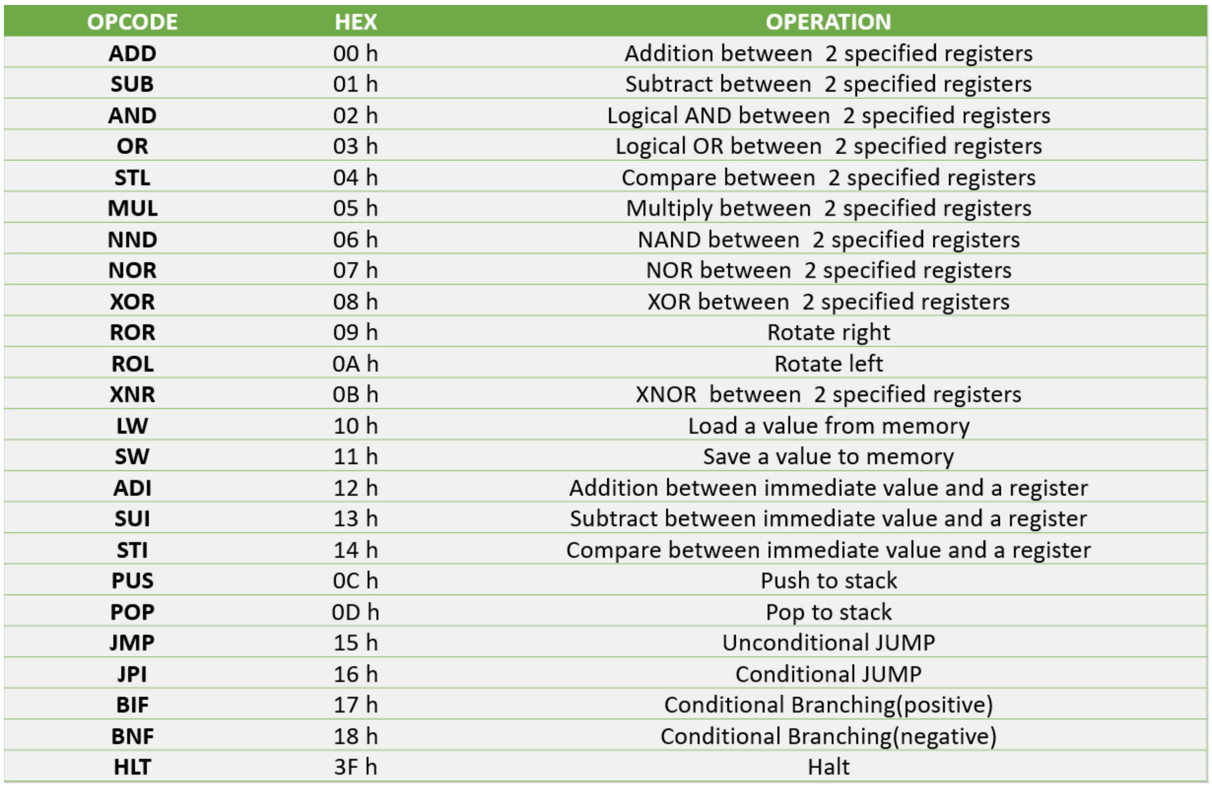
CHAPTER 5

DEVELOPMENTS

The microprocessor we developed undergoes through different stages, We have developed our instruction sets and its classification.

5.1 Instruction set

Table 4.1:Table of instruction



OPCODE	HEX	OPERATION
ADD	00 h	Addition between 2 specified registers
SUB	01 h	Subtract between 2 specified registers
AND	02 h	Logical AND between 2 specified registers
OR	03 h	Logical OR between 2 specified registers
STL	04 h	Compare between 2 specified registers
MUL	05 h	Multiply between 2 specified registers
NND	06 h	NAND between 2 specified registers
NOR	07 h	NOR between 2 specified registers
XOR	08 h	XOR between 2 specified registers
ROR	09 h	Rotate right
ROL	0A h	Rotate left
XNR	0B h	XNOR between 2 specified registers
LW	10 h	Load a value from memory
SW	11 h	Save a value to memory
ADI	12 h	Addition between immediate value and a register
SUI	13 h	Subtract between immediate value and a register
STI	14 h	Compare between immediate value and a register
PUS	0C h	Push to stack
POP	0D h	Pop to stack
JMP	15 h	Unconditional JUMP
JPI	16 h	Conditional JUMP
BIF	17 h	Conditional Branching(positive)
BNF	18 h	Conditional Branching(negative)
HLT	3F h	Halt

These operators enable the processor to perform arithmetic, logical, memory access, control flow, and system control operations. Each operator is crucial for implementing the functionality of the MIPS assembly language, allowing programmers to write code that can manipulate data, control the execution flow, and interact with memory. The hexadecimal codes are used to uniquely identify each operator when writing machine code or when the processor fetches and decodes instructions during the execution cycle.[7]

5.2 Operation Types

The table describes the types of operations that a MIPS processor can perform, all of which are encoded in a 32-bit instruction format. Each operation type is associated with a unique binary code that identifies the operation for the processor's control unit during the instruction decode stage. Here's a detailed overview of each operation type:[8]

1) **REGISTER REGISTER ALU OPERATION (RR_ALU):** This operation, coded as "000," involves performing arithmetic or logical operations using the processor's Arithmetic Logic Unit (ALU) on operands that are both in registers.
2) **REGISTER MEMORY ALU OPERATION (RM_ALU):** Represented by "001," this operation involves performing ALU operations where one operand is in a register and the other is in memory.
3) **LOAD FROM MEMORY OPERATION:** Coded as "010," this operation allows the processor to load data from memory into a register.
4) **STORE TO MEMORY OPERATION:** With the code "011," this operation enables the processor to store data from a register into memory.
5) **BRANCHING OPERATION:** Identified by "101," branching operations are critical for controlling the flow of execution in a program based on conditional statements.
6) **HALT:** The code "111" represents the halt operation, which is used to stop the processor's execution.

These operators enable the processor to perform arithmetic, logical, memory access, control flow, and system control operations. Each operator is crucial for implementing the functionality of the MIPS assembly language, allowing programmers to write code that can manipulate data, control the execution flow, and interact with memory. The hexadecimal codes are used to uniquely identify each operator when writing machine code or when the processor fetches and decodes instructions during the execution cycle.

5.3 Error Types

The python script not only translate the assembly code to testbench, it can detect the syntax and semantic errors in assembly code[7]

1. **Register Syntax Error** : refers to a mistake or inconsistency in the syntax used to specify registers. The register name or identifier is misspelt or incorrectly formatted. Also when the register number is out of range (e.g., attempting to reference a register beyond the valid range). In a register to register operation, 3 registers should be assigned after the opcode. Any change to this call can be referred to as Register Syntax Error.
2. **Immediate Value Error : r**efers to an issue encountered when attempting to parse or process an immediate value.Immediate values in MIPS assembly instructions represent constants or literals that are directly encoded into the instruction itself. Incorrect formatting of the immediate value (e.g., using an invalid hexadecimal format). Attempting to use an immediate value that is out of range or not supported by the instruction.
3. **Instruction Error :** refers to a problem encountered when attempting to parse or process an instruction. Each line of Assembly code should start with an Opcode. This error occur whenever the first field of the line is not a predefined opcode or comment statement starting with dollar symbol ($)

CHAPTER 6

RESULTS AND OBSERVATION

The successful implementation of a microprocessor in register-transfer level (RTL) and the accompanying Python script for testbench generation. Python testbench script facilitates efficient verification and validation of the microprocessor's functionality, enabling comprehensive testing under various scenarios and edge cases. Through meticulous observation and analysis of simulation results, crucial insights into the microprocessor's behavior and performance characteristics are gleaned, affirming its correctness and robustness. This integrated approach not only validates the microprocessor's design but also enhances understanding of its operation, paving the way for further optimization and refinement in future iterations.[3]

6.1 Final observations from project

From Phase 1, several key observations and findings emerged from the implementation of the MIPS processor in Verilog:[4]

Pipeline Efficiency Impact: The adoption of a 5-stage pipeline architecture significantly impacted the microprocessor's efficiency, allowing for concurrent instruction processing. This observation was evident in reduced clock cycles per instruction (CPI) and improved overall throughput, validating the advantages of pipelining.

Versatility in Instruction Handling: The implemented microprocessor demonstrated a high degree of versatility in handling various instruction types and operand combinations. This adaptability was crucial for accommodating the diverse MIPS 32 instruction set and contributed to the overall flexibility of the processor.

Testing and Result Validation: The rigorous testing procedures conducted during Phase 1 ensured the functional correctness of the microprocessor. The successful execution of diverse instruction sequences verified the accuracy of the implemented design and laid the groundwork for a reliable and robust processor.

Collaborative Project Management: Effective collaboration within the team was essential for meeting project timelines and milestones. The observation of seamless teamwork and communication underscored the importance of collaborative project management in the successful implementation of complex designs.

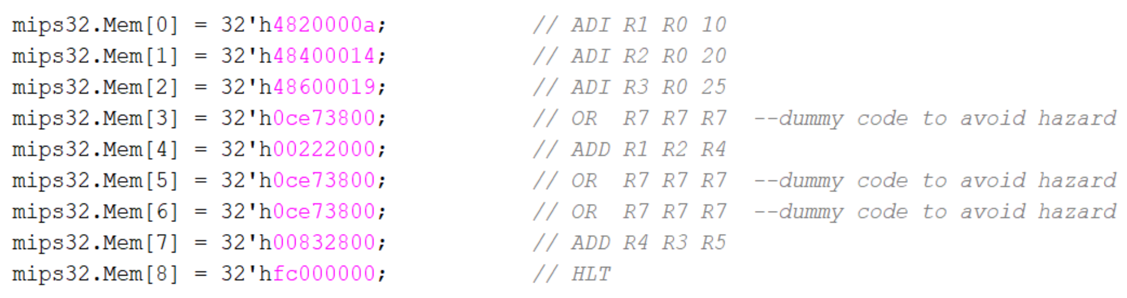
Trade-offs and Performance Metrics: Exploration of trade-offs between clock cycles per instruction (CPI) and throughput was a notable finding. Benchmarking analyses provided valuable insights into the performance characteristics of the microprocessor, guiding future optimization efforts.[6]

Practical Understanding of Design Complexity: The implementation phase deepened the team's practical understanding of the complexities involved in microprocessor design. Observations highlighted the intricate interplay of components, shedding light on real-world challenges and considerations in achieving a balance between performance, reliability, and resource utilization.

These findings from Phase 1 not only validate the theoretical concepts but also provide a comprehensive understanding of the microprocessor's behavior, guiding future development and optimization efforts in subsequent project phases

6.2 Simulation Results

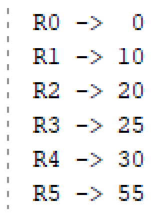
Currently, we have tested our microprocessor using a custom manual-written test bench and checked manually whether the expected result was obtained or not.



```
mips32.Mem[0] = 32'h4820000a;          // ADI R1 R0 10
mips32.Mem[1] = 32'h48400014;          // ADI R2 R0 20
mips32.Mem[2] = 32'h48600019;          // ADI R3 R0 25
mips32.Mem[3] = 32'h0ce73800;          // OR  R7 R7 R7  --dummy code to avoid hazard
mips32.Mem[4] = 32'h00222000;          // ADD R1 R2 R4
mips32.Mem[5] = 32'h0ce73800;          // OR  R7 R7 R7  --dummy code to avoid hazard
mips32.Mem[6] = 32'h0ce73800;          // OR  R7 R7 R7  --dummy code to avoid hazard
mips32.Mem[7] = 32'h00832800;          // ADD R4 R3 R5
mips32.Mem[8] = 32'hfc000000;          // HLT
```

figure 6.2.1 Results Obtained

The above image is an example testbench code for a simple ALU operations, and the register changes will print in the TCL CONSOLE.



```
R0 ->  0
R1 -> 10
R2 -> 20
R3 -> 25
R4 -> 30
R5 -> 55
```

Figure 6.2.2:*Simulation Result*

At the end of the simulation, the TCL console will show all register values if it undergoes any changes at each time stamp. After this, the values of all general purpose registers.

As here in the example, the register R0 holds the value 0, R1 holds the 10, R2 20, R3 25, R4 30 and R5 holds 55 as expected.

6.3 Hardware Implementation

We have performed the FPGA synthesis and implementation of processor using Xilinx Vivado in part Xilinx XC7K70TFBV676-1 FPGA Library. This report focus on the implementation report of our processor focusing on this FPGA.

6.3.1 Xilinx XC7K70TFBV676-1

The Xilinx XC7K70TFBV676-1 is a member of the Kintex-7 family of FPGAs. It features a generous amount of programmable logic cells, high-speed transceivers, block RAM, and DSP slices, making it suitable for a wide range of applications including signal processing, networking, and embedded systems.

The Xilinx XC7K70TFBV676-1 FPGA boasts an impressive array of features tailored to meet the demands of advanced digital design and prototyping. Equipped with a fixed oscillator generating a differential 200MHz output, it ensures stable and precise timing for high-speed applications. Its onboard JTAG configuration circuitry facilitates seamless configuration via USB, offering convenient accessibility for programming and reprogramming tasks. Additionally, the UART to USB bridge enhances connectivity and communication capabilities, while the XADC header enables analog-to-digital conversion for sensing and monitoring functions. With 5 push buttons and 4 DIP switches, user interaction and input are intuitive and flexible. The inclusion of a differential pair I/O, featuring 1 SMA pair, expands interfacing possibilities for differential signalling requirements. Boasting a robust memory subsystem, it incorporates a 4GB DDR3 SODIMM operating at 800MHz / 1600Mbps, ensuring ample storage and high-speed data access. Furthermore, its HDMI video output and 2x16 LCD display offer versatile visual output options, catering to diverse display needs

Table 6.3.1.1 :*Table of resource*

Resource	**Number**
Logic cell	65,600
CLB Slices	10250
CLB (max distributed)	838
DSP Slices	240
Block RAM 18kB	270
Block RAM 36kB	135
Block RAM max	4860
CMTs	6
PCle	1
XADC	1
GTXs	8
I/O Bank	6
Max user input	300

FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributedRAM or SRLs. Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.

6.3.2 Utilisation Report

A utilization report for an FPGA provides detailed information about how the resources of the FPGA are being utilized by a specific design or project. This report typically includes metrics such as the percentage of available logic cells (LUTs), flip-flops, block RAM (BRAM), and DSP slices that are utilized by the design. It may also include information about other resources such as I/O pins, clock resources, and specialized blocks like multipliers or memory controllers.

Table 6.3.2.1:*Table of utilisation*

Resources	**Utilisation**	**percent**
Look up table	45039	12.05%
Block ram	5396	87%
Flipflops	274	5.94%
DSP	9	3.75%
I/O block	1	16.67%

.

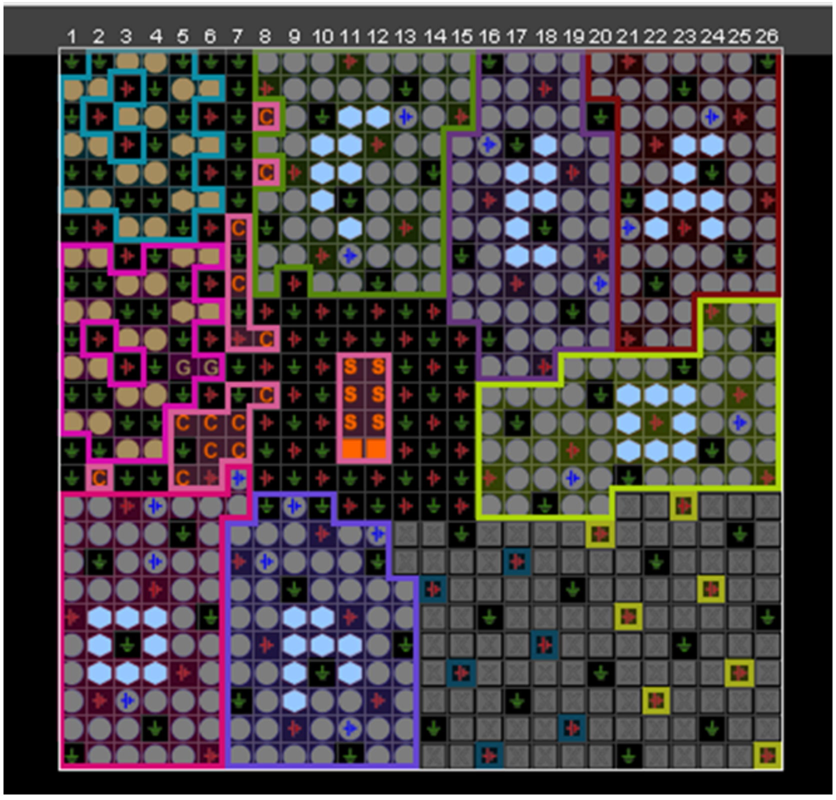
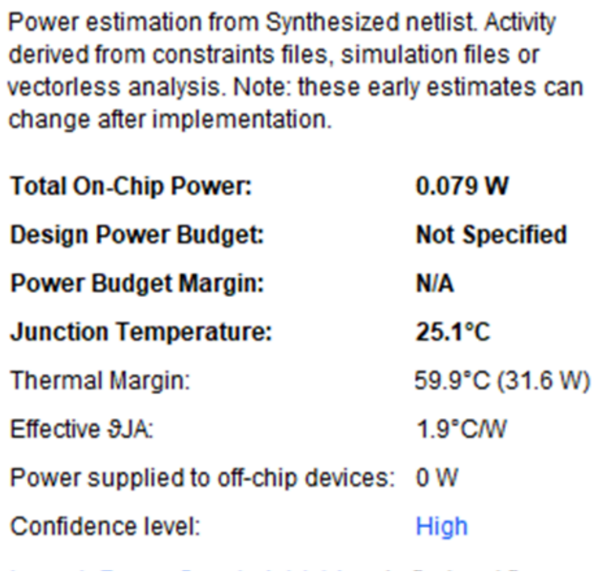


Figure 6.3.2.2: Ultilization Floor map

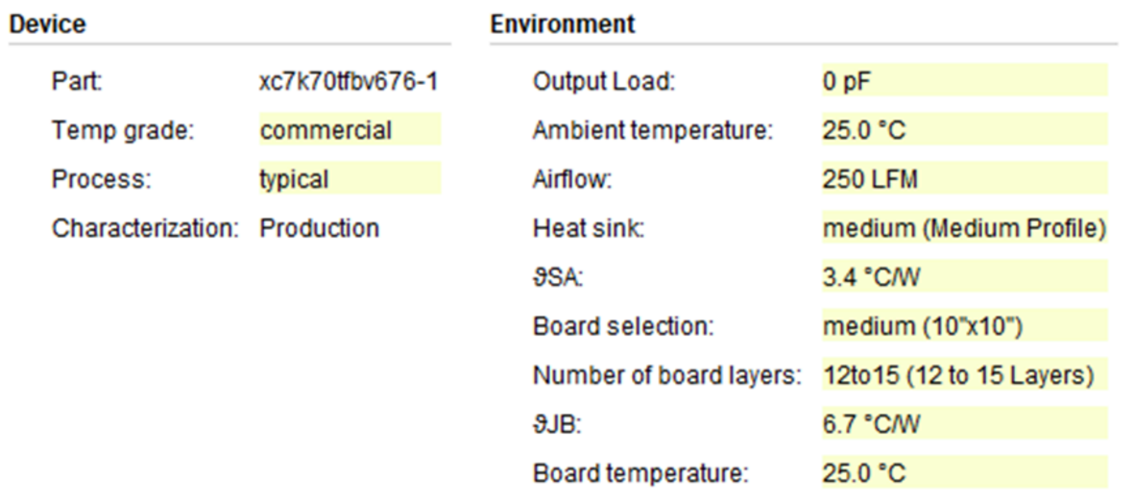
6.3.3 Power Report



Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.079 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.1°C**
Thermal Margin:	59.9°C (31.6 W)
Effective ϑJA:	1.9°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

Figure 6.3.3.1:*Power analysis result*



Device

Part:	xc7k70tfbv676-1
Temp grade:	commercial
Process:	typical
Characterization:	Production

Environment

Output Load:	0 pF
Ambient temperature:	25.0 °C
Airflow:	250 LFM
Heat sink:	medium (Medium Profile)
ϑSA:	3.4 °C/W
Board selection:	medium (10"x10")
Number of board layers:	12to15 (12 to 15 Layers)
ϑJB:	6.7 °C/W
Board temperature:	25.0 °C

Figure 6.3.3.2:*Power analysis condition*

6.3.4 Timing report

Timing analysis is essential for FPGA designers to guarantee that their designs operate reliably and meet performance goals. Failure to adequately analyze and address timing issues can result in functionality errors, timing violations, and overall design failure.[6]

Table 6.3.4.1 :*Table of timing report*

Worst Negative stack	9.100ns
Worst hold slack	0.219ns
Worst pulse width slack	4.5ns

Worst Negative Slack (WNS) is a critical timing metric used in timing analysis to assess the timing performance of a design. It represents the amount of time by which the arrival time of the data at the destination flip-flop (or register) is later than the required time, taking into account the clock delay and data path delay.

Hold Time Slack (HTS) or **Worst Hold Slack** (WHS) is the amount of time by which the data must be stable before the next clock edge, minus the time available for the data to stabilize. A negative hold slack indicates a hold time violation, where the data changes too close to the clock edge.

Pulse Width Slack (PWS) or **Worst Pulse Width Slack** (WPWS) is the difference between the actual pulse width of the signal and the required minimum or maximum pulse width specified by the timing constraints. A negative pulse width slack indicates a violation, where the actual pulse width deviates from the required pulse width.

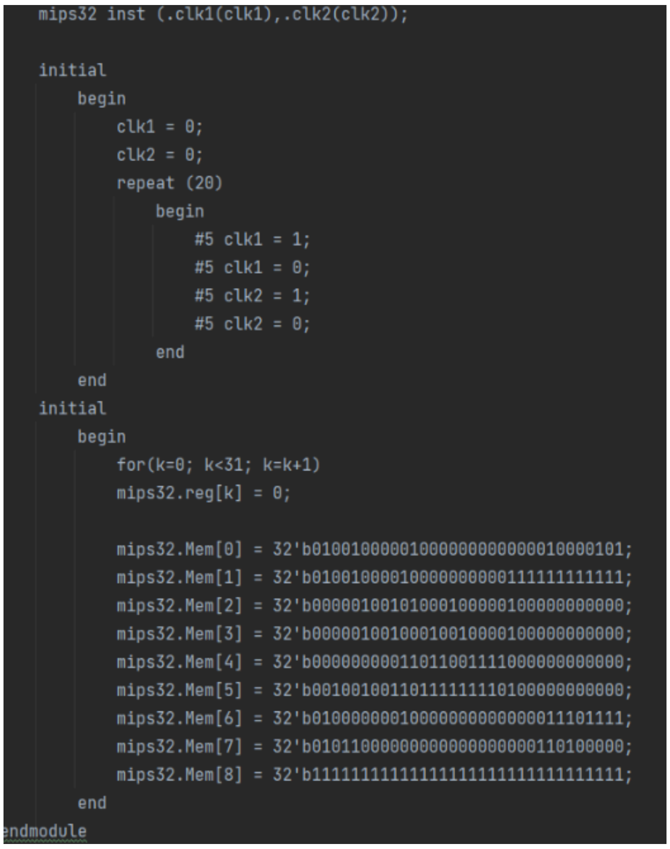
So the maximum usable frequency will be

$$T_{CLK} \geq T_{WNS} + T_{WHS}$$

$$\geq 9.319\text{ns}$$

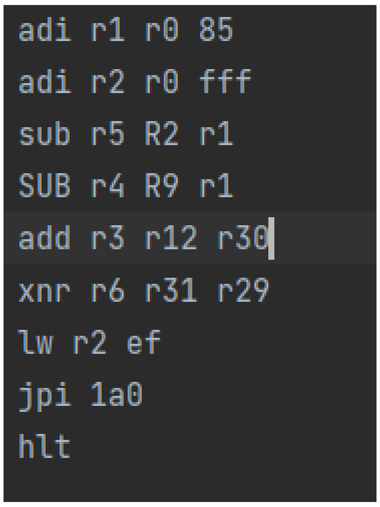
6.4 Python Script

The Python script developed for automating testbench generation has proven to be a valuable asset in our project, streamlining the verification process of the MIPS processor model. By converting human-readable assembly code into machine-readable instructions, the script significantly reduces the time and effort required for testbench creation. Here is an example of automated testbench generation[10]



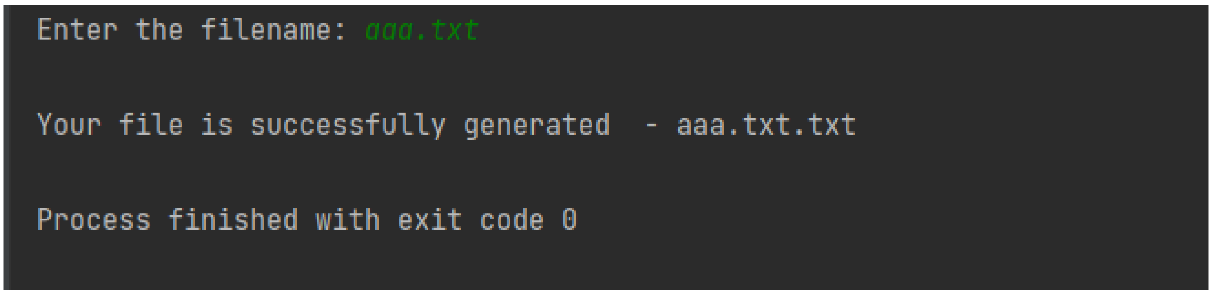
```
    mips32 inst (.clk1(clk1),.clk2(clk2));

    initial
        begin
            clk1 = 0;
            clk2 = 0;
            repeat (20)
                begin
                    #5 clk1 = 1;
                    #5 clk1 = 0;
                    #5 clk2 = 1;
                    #5 clk2 = 0;
                end
        end
    initial
        begin
            for(k=0; k<31; k=k+1)
            mips32.reg[k] = 0;

            mips32.Mem[0] = 32'b01001000001000000000000010000101;
            mips32.Mem[1] = 32'b01001000010000000000001111111111;
            mips32.Mem[2] = 32'b00000100101000100000100000000000;
            mips32.Mem[3] = 32'b00000100100010010000100000000000;
            mips32.Mem[4] = 32'b00000000011011001111000000000000;
            mips32.Mem[5] = 32'b00100100110111111110100000000000;
            mips32.Mem[6] = 32'b01000000010000000000000011101111;
            mips32.Mem[7] = 32'b01011000000000000000000110100000;
            mips32.Mem[8] = 32'b11111111111111111111111111111111;
        end
endmodule
```

Figure:6.4.1:*Generated Text*



```
adi r1 r0 85
adi r2 r0 fff
sub r5 R2 r1
SUB r4 R9 r1
add r3 r12 r30
xnr r6 r31 r29
lw r2 ef
jpi 1a0
hlt
```

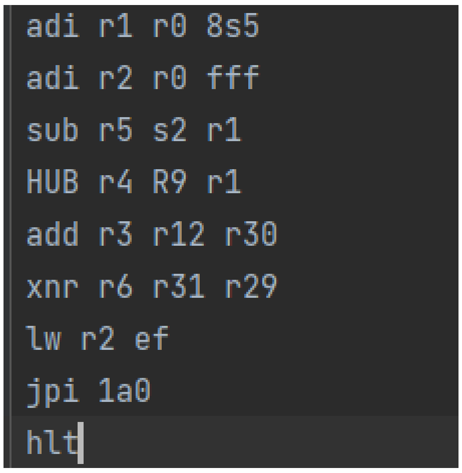
Figure 6.4.2:*Input assembly file*



```
Enter the filename: aaa.txt

Your file is successfully generated  - aaa.txt.txt

Process finished with exit code 0
```

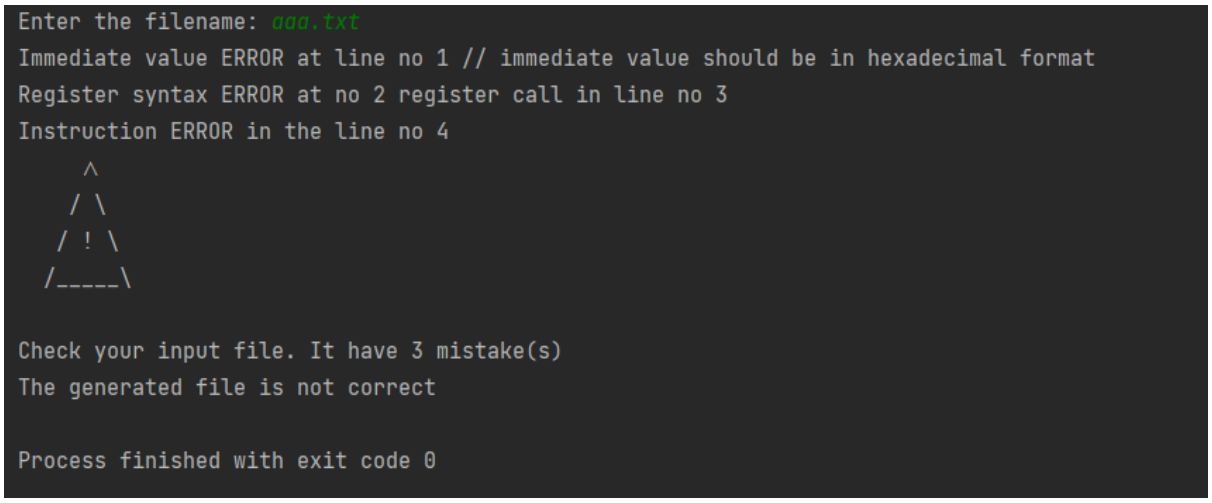
Figure 6.4.3:*Terminal Result*

The script is capable to check the error in the assembly input file. Next is an example shows the error checking ability of script



```
adi r1 r0 8s5
adi r2 r0 fff
sub r5 s2 r1
HUB r4 R9 r1
add r3 r12 r30
xnr r6 r31 r29
lw r2 ef
jpi 1a0
hlt
```

Figure 6.4.4:*Input assembly file 2*



```
Enter the filename: aaa.txt
Immediate value ERROR at line no 1 // immediate value should be in hexadecimal format
Register syntax ERROR at no 2 register call in line no 3
Instruction ERROR in the line no 4
     ^
    / \
   / ! \
  /_____\

Check your input file. It have 3 mistake(s)
The generated file is not correct

Process finished with exit code 0
```

Figure 6.4.5:*Terminal Result 2*

CHAPTER 7

CONCLUSION AND FUTURE PROSPECTS

7.1 Conclusion

With the completion of both phases, our project has achieved significant milestones in the development and verification of a MIPS processor model. The implementation of a 5-stage pipeline architecture in Verilog demonstrates our team's commitment to producing a robust and efficient microprocessor design. Rigorous testing and verification processes have ensured the correctness and reliability of our model, reinforcing our understanding of microprocessor principles and digital design intricacies. Transitioning into Phase 2, the focus shifted towards automating testbench generation, a strategic move to streamline verification processes. Through the development of a Python script, we aimed to mitigate the time-consuming and error-prone nature of manual testbench creation, thus enhancing the efficiency and reliability of our microprocessor verification workflow.

The successful integration of the MIPS processor model and pipeline architecture underscores our proficiency in digital design and hardware description languages. Our project's strategic approach not only addresses the challenges of microprocessor verification but also aligns with industry demands for efficient and reliable design methodologies. By automating testbench generation, we have significantly reduced development time while maintaining a rigorous verification process. Looking ahead, our project stands poised to contribute meaningfully to the advancement of microprocessor design, offering a comprehensive and effective solution to testing and verification needs. The achievements of both phases form a solid foundation for continued innovation and success in our project, as we continue to explore new frontiers in microprocessor design and development.

7.2 Future prospects

Looking ahead, the successful completion of this project sets the stage for promising future prospects in microprocessor design and verification. The automated testbench generation system, can serve as a foundational tool for future projects involving MIPS 32 microprocessors or similar architectures. It can be upgraded to perform any architecture and design, since the current model is only for this specific design. Its versatility allows for adaptability to diverse microprocessor designs, fostering broader applications beyond the current scope.

Also we can implement more instructions like vector function, floating point operations, signal processing operations and SIMD operations to the processor. Another change which can be implemented is that the integration of parallel processing and out of order processing which will greatly improve the speed and performance of the processor

7.3 Challenges

During the project phase, we faced many challenges, some of them are solved and others still can't. The main challenge was to synchronise the pipeline stages. The next is to implement an interrupt function but can't completely implement it as interrupt, but only as a normal function call using jump and return statements. Another one is Out of Order execution where when we execute a line of instructions, if there are some co-dependent lines of code, some independent code will execute between them for faster execution time. We are not able to implement out-of-order execution

,

CHAPTER 8

REFERENCE

[1] Hardware Modelling Using Verilog NPTEL Lecture Prof Indranil Sen Gupta https://archive.nptel.ac.in/courses/106/105/106105165/

[2] Javatpoint Verilog tutorials https://www.javatpoint.com/verilog (20/sept/23)

[3] Computer Architecture with (MIPS) Assembly by Peter Stalliga

[4] Computer Organization and Design MIPS Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design) by David A. Patterson, John L. Hennessy

[5] Geek for geeks - MIPS (4/oct/23) https://www.geeksforgeeks.org/what-is-mipsmillion-of-instructions-per-second/

[6] Digital System Design by Charles H Roth Jr, Lizy Kurian John, Byeong Kil Lee

[7] Implementation of a 32-bit MIPS based RISC processor using Cadence https://ieeexplore.ieee.org/document/7019240

[8] Design and Implementation of 32 bit MIPS based RISC Processor https://ieeexplore.ieee.org/document/9566007

[9] Python for everybody by Charles Severance

[10] Geek for geeks - text manipulation using Python (15/jan/24)