Verification of Microprocessor without Interlocked Pipeline (MIPS) Processor using Self-Checking Testbench

TENG WEN JUN

MASTER OF

ENGINEERING (ELECTRONIC SYSTEMS)

FACULTY OF ENGINEERING AND GREEN

TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

MAY 2023

Verification of Microprocessor without Interlocked Pipeline (MIPS) Processor using Self-Checking Testbench

By

TENG WEN JUN

A dissertation submitted to the Faculty of Engineering and Green Technology, Universiti Tunku Abdul Rahman, in partial fulfillment of the requirements for the degree of Master of Engineering (Electronic Systems)

May 2023

ABSTRACT

Verification of Microprocessor without Interlocked Pipeline (MIPS) Processor using Self-Checking Testbench

Teng Wen Jun

MIPS stand for Microprocessor without Interlocked Pipeline Stages. It is a reduced instruction set computer (RISC) instruction set architecture (ISA). RISC is a well-stablished architecture due to its efficiency and simplicity. Thus, it is widely used in the processor industry. However, verifying and validating the correctness of the processor if a complex work as it consists of about 111 total instructions (Stanford.edu, 2020). Various types of hazards might be arise due to the complexity of the pipeline structures. Thus, the verification process will be time consuming as validators need to verify the whole design by checking the waveforms after they make some minor changes. This project is to improve the efficiency of verification process of the current RISC32 5-stage pipeline processor that developed in Universiti Tunku Abdul Rahman which is under Faculty of Information Technology by developing a complete self-checking testbench using SystemVerilog to verify the functional correctness of the MIPS design at system level.

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to Dr. Loh Siu Hong, my esteemed supervisor, for his invaluable guidance, unwavering support, and expertise throughout the duration of this project.

APPROVAL SHEET

This dissertation entitled "Verification of Microprocessor without Interlocked Pipeline (MIPS) Processor using Self-Checking Testbench**"** was prepared by TENG WEN JUN and submitted as partial fulfillment of the requirements for the degree of Master of Master of Engineering (Electronic Systems) at Universiti Tunku Abdul Rahman.

Approved by:

_____ _____

(Dr. Loh Siu Hong)

Date: 14/8/2023

Supervisor

Department of Electronic Engineering

Faculty of Engineering and Green Technology

Universiti Tunku Abdul Rahman

FACULTY OF ENGINEERING AND GREEN TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

Date: 16-07-2023

SUBMISSION OF DISSERTATION

It is hereby certified that **TENG WEN JUN** (ID No: **2106710**) has completed this dissertation entitled **"Verification of Microprocessor without Interlocked Pipeline (MIPS) Processor using Self-Checking Testbench"** under the supervision of **Dr. Loh Siu Hong** (Supervisor) from the Department of Electronic Engineering, Faculty of Engineering and Green Technology.

I understand that University will upload softcopy of my dissertation in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

_____

(**TENG WEN JUN**)

DECLARATION

I hereby declare that the dissertation is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.

(TENG WEN JUN)

DATE: 16-07-2023

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 MIPS

MIPS is the acronym for Microprocessor without Interlocked Pipeline Stages, representing an instruction set architecture (ISA) for reduced instruction set computers (RISC) that was formulated by MIPS Technologies. MIPS is mainly used as an embedded processor for the large market for embedded applications due to its simple design and high performance instead of the Intel 80x86 processor that is primarily “CISC” design with emphasis on backward compatibility which is lot more complex. Nowadays, MIPS architecture supports 64-bit addressing and operation and high-performance floating point. This is the reason why it is popular in the embedded systems implementation such as video game consoles. The MIPS architecture products include the MIPS32 and MIPS64.

1.1.1 MIPS Instruction Format

Instruction format is the layout of the instruction bits in field. There are 3 basic types of instruction formats. These instruction formats include:

- I-format: for arithmetic, logic, data transfer and branch.
- J-format: for j and jal.
- R-format: for all other instructions.

Figure 1.1 shows the instruction layout.

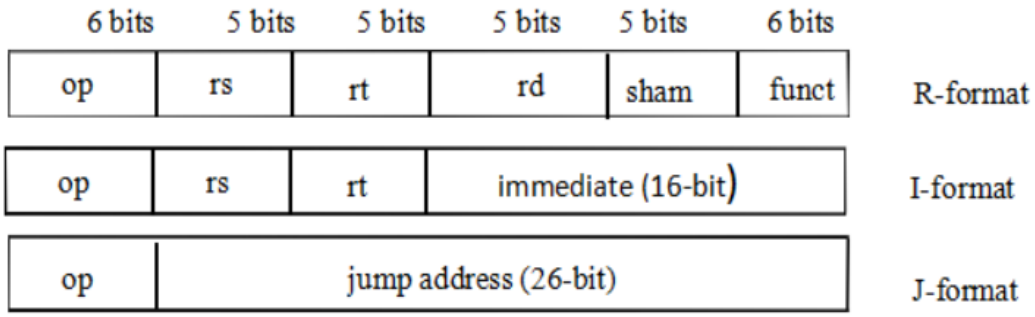

Figure 1. 1: Instruction layout for MIPS

1.1.2 MIPS Execution Cycle

The execution of an instruction can be done in 5 basic stages and the execution of an instruction is partially completed with each stage. These 5 basic stages include:

- IF: Instruction fetch and update PC
- ID: Instruction decode and registers fetch
- EX: Execute
- MEM: For lw and sw instruction. Data will be written and read from data memory
- WB: Write back the result data into the register file

"Stages" implies datapath resources at each stage. Figure 1.2 shows the instruction execution cycles for lw instruction. Besides, the structural view for datapath is shown in Figure 1.2.

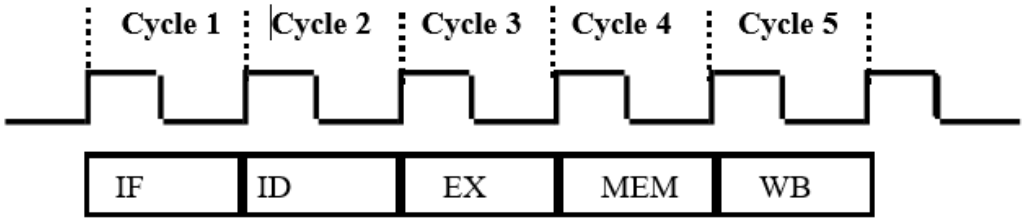

Figure 1. 2: Instruction execution cycle for lw instruction

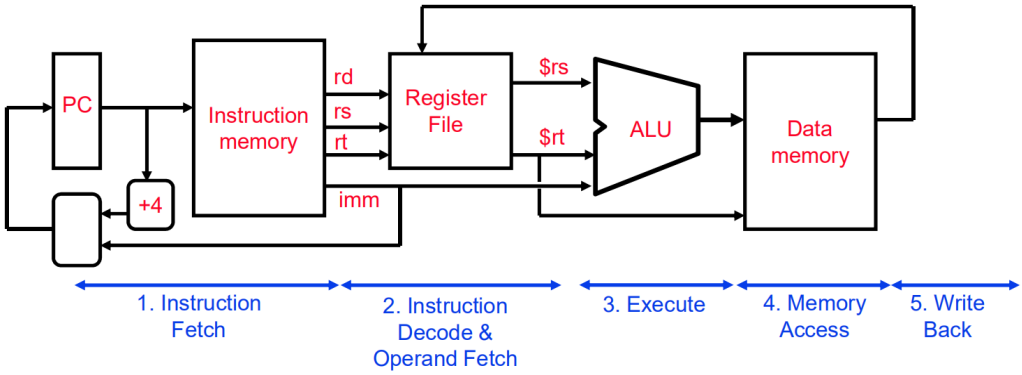

Figure 1. 3: Structural view of datapath

1.2 Pipelining

Pipelining is a usage strategy whereby more than 1 instruction are overlapped during execution, and it exploits parallelism that exists among the actions expected to execute an instruction (Patterson and Hennessy, 2001, C-2). Performance is

improved by increasing throughput which is average instructions completed per clock cycle. The basic pipeline processor for RISC32 has only 5 stages. Figure 1.4 illustrates the hardware components allocate in each pipeline stages of the 5-stage pipeline processor RISC32.

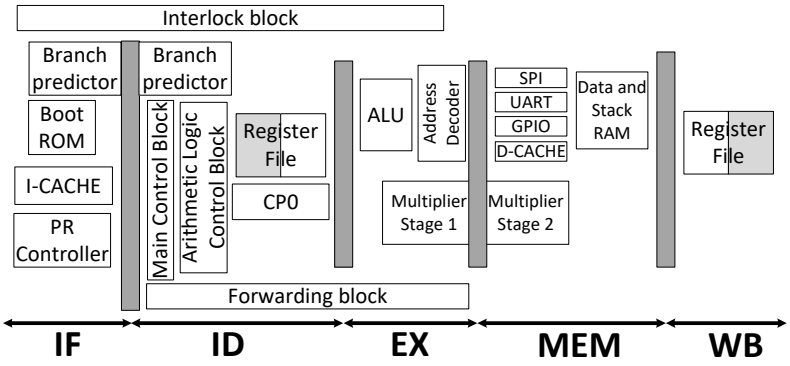

Figure 1. 4: Abstract view of 5-stage pipeline processor. (Kiat, 2018, p.49)

1.3 RISC

RISC, abbreviated as Reduced Instruction Set Computer, represents a microprocessor architecture that utilizes a highly optimized and small set of instructions. RISC has 5 design principles:

- Simple instructions
- Efficient, deep pipelining
- Hard-wired control
- Single-cycle execution
- Load and store

UC-Berkeley, Stanford, and IBM started the first RISC projects in the late 70s and early 80s. Nowadays, there are a lot of computer systems that take advantage of a

RISC processor. For examples, A16 Bionic, which integrated in iPhone 14 Pro models.

1.4 Self-Checking Testbench

It is a VHDL program responsible for independently validating the accuracy of the device under test without necessitating manual output inspection by a validator (Jensen, 2019). This self-checking testbench operates autonomously and produces messages, as defined by the validator at the end of the test. Within the industry, each VHDL module is typically accompanied by a dedicated self-checking testbench for the ease of verifying all the modules have the intended behavior.

1.5 Problem Statement

The verification of MIPS processor is a time-consuming and complex process as MIPS instruction set consists of about 111 total instructions (Stanford.edu, 2020). Traditionally, verification is carried out through simulation and check the waveforms manually to make sure the behavior of the design is correct. This process is prone to human errors. If changes are made to the design, validators need to verify the whole design again due to its complexity of the pipeline structures by

checking the waveforms. A better and suitable approach is to write a self-checking testbench.

1.6 Objectives

The objectives of this project are as follow:

i. To develop a complete self-checking testbench using SystemVerilog to verify the functional correctness of the MIPS design at system level.

ii. To develop a function in SystemVerilog that will output a log file of the instruction execution flow for the ease of debugging.

iii. To reduce the time spent in the validation process by utilizing the automation capabilities of the self-checking testbench.

1.7 Contribution

The main contributions of this project are:

1) Development of a self-checking testbench methodology in system level. This testbench will focus on the pipeline of the processor by comparing the internal signals of the processor with the expected value and clock cycle.
2) Detection of hazards and design flaws. The self-checking testbench developed in this project will play a crucial role in detecting hazards and design flaws in MIPS processor.

1.8 Dissertation Organization

The dissertation is organized as follows:

Chapter 2 discusses the existing methodologies, techniques, and tools used for microprocessor verification, with a specific focus on MIPS processors.

Chapter 3 discusses the methodology of the self-checking testbench for the verification of MIPS processors.

Chapter 4 discusses the results and findings of the projects. It also identifies limitations and potential areas for future improvement.

Chapter 5 discusses the conclusions of the project and provides recommendations for future project direction.

CHAPTER 2

LITERATURE REVIEW

2.1 Functional Verification Methodology of a 32-bit RISC Microprocessor (Zhenyu Gu et al., 2002)

Zhenyu Gu et al. (2002) verified a 32-bit RISC microprocessor by using a simulation-based functional verification methodology. In this project, handwriting, pipeline-focus and pseudo-random are the main method of the testbench generation. Figure 2.1.1 shows the verification environment of the processor.

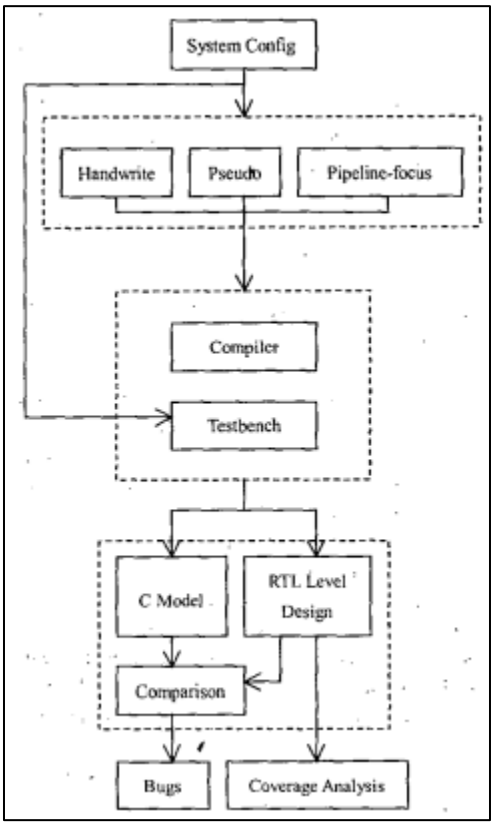

Figure 2. 1: Verification environment. (Zhenyu Gu et al., 2002)

With this verification environment, the efficiency and the automation of the verification process is great. However, there is no constraint that can be controlled by validator in the pseudo-random generator. Thus, a lot of redundant testbenches are generated as mentioned in the paper. Besides, there is no log file of the instruction execution flow is generated. Thus, validator still need to look at waveform from the beginning of the test to debug the failure. To further improve the efficiency, a log file that contains all the instruction execution flow should be generated.

2.2 Verification of a RISC processor IP Core using SystemVerilog (Sethulekshmi et al., 2016)

Sethulekshmi et al. (2016) verified their RISC CPU by using SystemVerilog Verification Methodology (OVM). In the verification process, a testbench that is both extensible and configurable is generated. The DUT and the verification environment are connected through boundary signals of the DUT. The boundary signals are grouped into interfaces. The testbench is split up into components and layers to resolve the complexity of the verification systems and the DUT as well as the reusability of the codes for future projects. Figure 2.2 shows the architecture of the testbench.

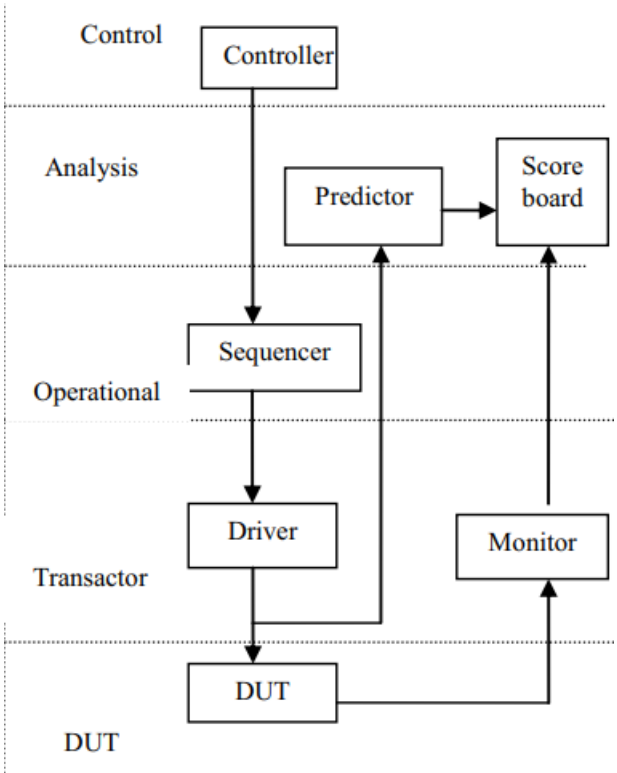

Figure 2. 2: Architecture of the testbench. (Sethulekshmi et al., 2016)

The testbench developed in this project has self-checking functionality by comparing the predicted output with the monitored output. However, there is no log file of the instruction execution flow is generated. Thus, validator still need to look at waveform from the beginning of the test to debug the failure. To improve the efficiency, a log file that contains all the instruction execution flow should be generated.

2.3 Design & Verification of 16 Bit RISC Processor (Jung S.P. et al., 2008)

Jung S.P. et al. (2008) designed and verified a 16-bit RISC processor. In this project, the RISC processor is verified through 3 steps of test. First, a reference model to the processor is constructed by using an instruction set simulator. Figure 2.3 shows the instruction set simulator.

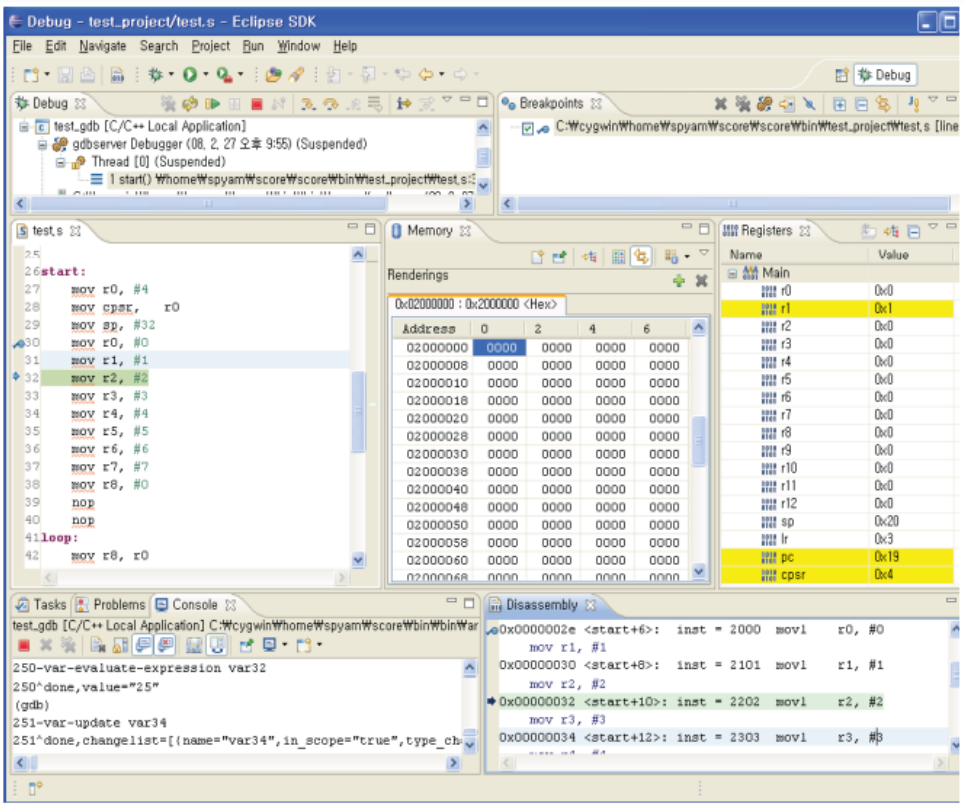

Figure 2. 3: Instruction set simulator. (Jung S.P. et al., 2008)

Secondly, a high complexity of algorithm test is accomplished to verify the processor by using the HDL simulator and the instruction set simulator. Lastly, manual inspection of the waveform is conducted.

The drawback of the verification method proposed in the system level is time consuming due to the testbench developed do not have self-checking functionality. Validator needs to verify the whole design at system level by checking the waveforms even though the design engineer makes a minor change at the module level. To reduce the time spent in the validation process, develop a self-checking testbench is a better approach.

2.4 HW/SW Co-Verification of a RISC CPU using Bounded Model Checking (Große et al., 2005)

Große et al. (2005) verified a RISC CPU through BMC, also known as Bounded Model Checking method, an inclusive method for formally verifying hardware and software components. BMC can simplify the challenge into a Boolean satisfiability problem by checking whether the design adheres to a temporal property. Figure 2.4 shows the implementation of BMC. The code snippet below shows the property will be evaluated when x=1, then y must be 2 in two clock cycles later. Figure 2.5 shows the implementation of BMC in ADD instruction.

```
property test
always
 // assume part
 ( x = 1 )
 ->
 // prove part
 next[2] ( y = 2 );
```

Figure 2. 4: Property test. (Große et al., 2005)

```
property ADD
always
 // assume part
 ( reset = 0 && OPCODE = "00111" &&
   Ri_A > 1 && Rj_A > 1 && Rk_A > 1 )
 ->
 // prove part
  next(
   (reg.reg[prev(Ri_A)] + (65536 * stat.C) = prev(Rj) + prev(Rk))
    && ((reg.reg[prev(Ri_A)] = 0) <-> (stat.Z = 1))
```

Figure 2. 5: Property ADD. (Große et al., 2005)

All the hardware is verified formally by describing their behavior with temporal properties. However, there is no log file of the instruction execution flow is generated. Thus, validator still need to look at waveform from the beginning of the test to debug the failure. Besides, the design of the RISC processor is a single-cycle design CPU. To improve the efficiency, a log file that contains all the instruction execution flow should be generated.

2.5 Verification of a 32-bit RISC Processor Core (Kasanko, T. and Nurmi, J., 2004)

Kasanko, T. and Nurmi, J. (2004) verified COFFEE™ RISC Core which is developed in the Institute of Digital and Computer Systems at Tampere University of Technology. The RISC processor consists of a six-stage pipeline. Difference methods were used to make sure the design operates without any bug or error. The methods include FPGA prototyping, formal verification, and pseudo-random input generation. At system level verification, a precisely designed model emulates the system-level behavior and is created for the COFFEE™ core reference design. This model exclusively contains the instruction functionality without pipeline stages and make sure the proper functioning of the entire design.

The drawback of the verification method proposed in the system level is time consuming due to the testbench developed do not have self-checking functionality. Validator needs to verify the whole design at system level by checking the waveforms even though the design engineer makes a minor change at the module level. To reduce the time spent in the validation process, develop a self-checking testbench is a better approach.

CHAPTER 3

METHODOLOGY

3.1 Verification Methodology

This project will focus on the use of formal verification techniques, including assertion and property in SystemVerilog to ensure the reliability and the correctness of the MIPS processor implementation.

SystemVerilog will be used to develop the self-checking testbench. It is a Hardware Description Languages (HDL) that supports Bounded Model Checking (BMC). SystemVerilog is significantly superior to Verilog because it provides constructs such as constrained random testing, coverage, and assertions that can be used in BMC.

Assertion is an expression or statement that define the behavior of a system that should be always true during simulation. Therefore, assertions are used to validate the behavior of a system defined as properties and can be used in functional coverage (ChipVerify, n.d.). If an assertion finds that a property of the design being examined does not behave as anticipated, it results in the failure of the assertion.

Property is similar to assertion, but it is used to specify requirements for specific scenarios within a bounded context. It checks for property within a finite number of clock cycles, making it particularly useful for bounded verification.

To efficiently validate the MIPS processor design, Bounded Model Checking (BMC) in combination with assertion and property will be used to develop the testbench. BMC allows validator to explore the design space within a finite bound, and identity potential bugs in the design. Appropriate bounds based on the complexity of the MIPS processor design will be set. Through BMC, the MIPS processor design is systematically unrolled for a specific number of clock cycles and check the validity of the defined properties and assertions.

Figure 3.1.1 shows the abstract view of MIPS processor. The development of self-checking testbench will base on the design and will focus on the pipeline of the processor.

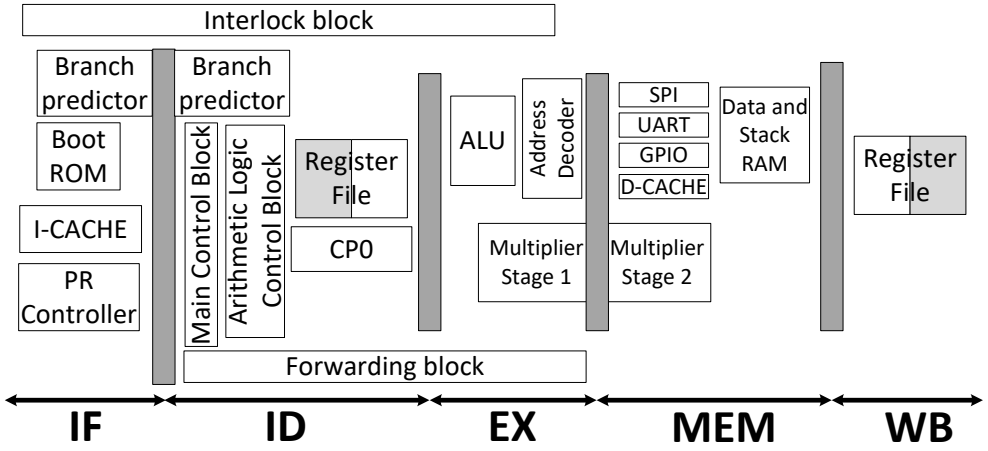

Figure 3.1. 1: Abstract view of 5-stage pipeline processor. (Kiat, 2018, p.49)

```
property ADD;
  int rs,rt,rd;
  @(`PC_CLK)
//assuming part
  (`OPCODE == `OP_RTYPE && `FUNCT == `FN_ADD ,
   rs=`RS,
   rt=`RT,
   rd=`RD
   )
//check no overflow is not happened in next cc
   |-> `DELAY_ID_EX (~`OVERFLOW)
//checking part
   |-> `DELAY_EX_WB (`REG_RAM[rd] == (rs+rt));
endproperty
```

Code snippet above shows the implementation of Bounded Model Checking for an arithmetic instruction – ADD. Same method will be implemented to the rest of the arithmetic instruction as well. Following are the explanations of the implementation of BMC:

1. Three integer variables `rs`, `rt`, and `rd` are declared. `rs` and `rt` will be used to store the value of source register while `rd` will be used to store the destination register number.
2. The `@(`PC_CLK)` is a clocking event, which means the property is evaluated on each rising edge of the clock signal. The property is checked and evaluated at specific points in the pipeline based on this clocking event.
3. The "assuming" part defines the conditions under which this property is assumed to hold. It checks if the opcode is of the R-type and the function code corresponds to the addition operation (ADD). If these conditions are met, the property assumes that the instruction is an add instruction and assigns the values of rs, rt and rd to the corresponding variables.

4. The first checking part uses the implication operator |-> to check that the "assuming" part implies the "check" part. The check part verifies that no overflow occurs in the next clock cycle which is ID stage during the instruction execution.
5. The second checking part also uses the implication operator |-> to check that the "check" part implies this second "check" part. The second check part verifies that the result of the addition operation (rs + rt) matches the value stored in the destination register rd after the delay in the `DELAY_EX_WB clock cycle.

Lastly, to facilitate the tracing and analysis of the processor's instruction flow during simulation, a log file is generated using SystemVerilog. The log file captures the program counter (PC) value, the corresponding instruction in hexadecimal format, and a decoded string representation of the instruction. This log file provides valuable insights into the execution of different MIPS instructions in the processor pipeline.

3.2 Design Tools

ModelSim from Intel is the industry-leading simulation and debugging environment for HDL-based design in which its license can be obtained freely. Furthermore, ModelSim supports the SystemVerilog and other VHDL languages. This stimulator is also able to provide syntax error checking and waveform

simulation which play an important part in developing the project. The timing diagrams and the waveforms are very useful in verifying the model functionalities after writing the testbench.

PCSpim is a Windows-based software stimulator that loads and executes assembly language program for the MIPS RISC architecture. It provides a simple assembler, debugger, and a set of operating services Thus, it is used for developing the MIPS test program for functional verification in this project.

CHAPTER 4

RESULT & DISCUSSION

Bounded Model Checking (BMC) has been successfully implemented to the design by using assertion in SystemVerilog. Code snippet below shown the implementation of "add" instruction using assertion.

```
property ADD;
  int rs,rt,rd;
  @(`PC_CLK)
  (`OPCODE == `OP_RTYPE && `FUNCT == `FN_ADD ,
   rs=`RS,
   rt=`RT,
   rd=`RD
   , $display("value of rs is %0h, value of rt is %0h, expected data at REG[%0h] is %0h",rs,rt,rd, (rs+rt))
   )
   |-> `DELAY_ID_EX (~`OVERFLOW) //check no overflow is not happened in next cc
   |-> `DELAY_EX_WB (`REG_RAM[rd] == (rs+rt));
endproperty
```

Table 4.1 below shows the instruction "add" that will be verified during simulation.

Machine code	Address	Instruction
01294820	0x004000F0	add *$t1*, *$t1*, *$t1*

Table 4. 1: Details of "add" instruction.

Figure 4.1 shows the simulation result of “add” instruction and the output of the transcript the incorrect behavior is found.

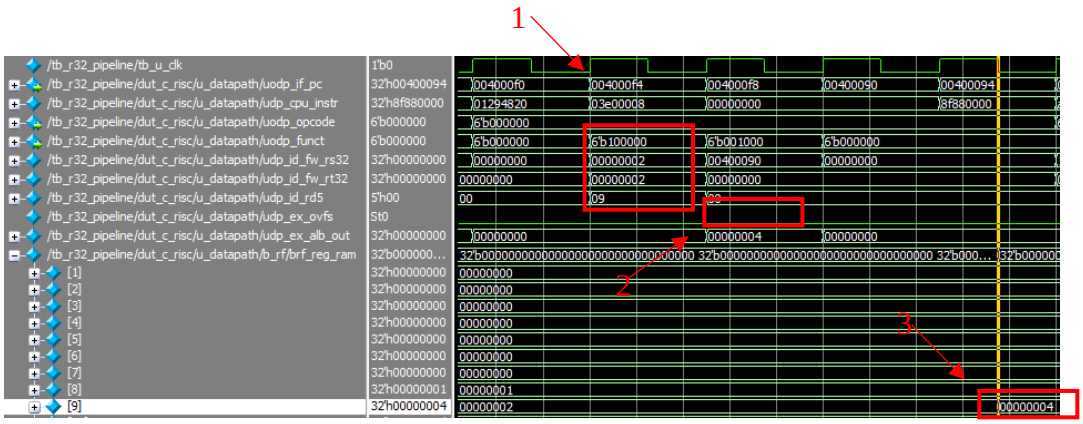

Figure 4. 1: Simulation result of “add” instruction.

1. In ID stage, the property will be evaluated if the test expression is evaluated to true. The value from the register file will be stored in the correspond variable that will be used in self-check. In this stage, the expected data can be evaluated.
2. In the next clock tick, which is EX stage, overflow will be checked.
3. In the next 2 clock ticks, which is MEM stage, expected data will be compared with the actual data.

Figure 4.2 shows the counterexample's output generated by BMC, which illustrate the scenarios where properties were violated. It violates `REG_RAM[rd] == (rs+rt) property where the expected data of reg_ram[9] is 4 but the actual data in reg_ram[9] is 3.

```
VSIM 39> run -all
# ** Error: ADD failed
#    Time: 12600 ns Started: 12450 ns
```

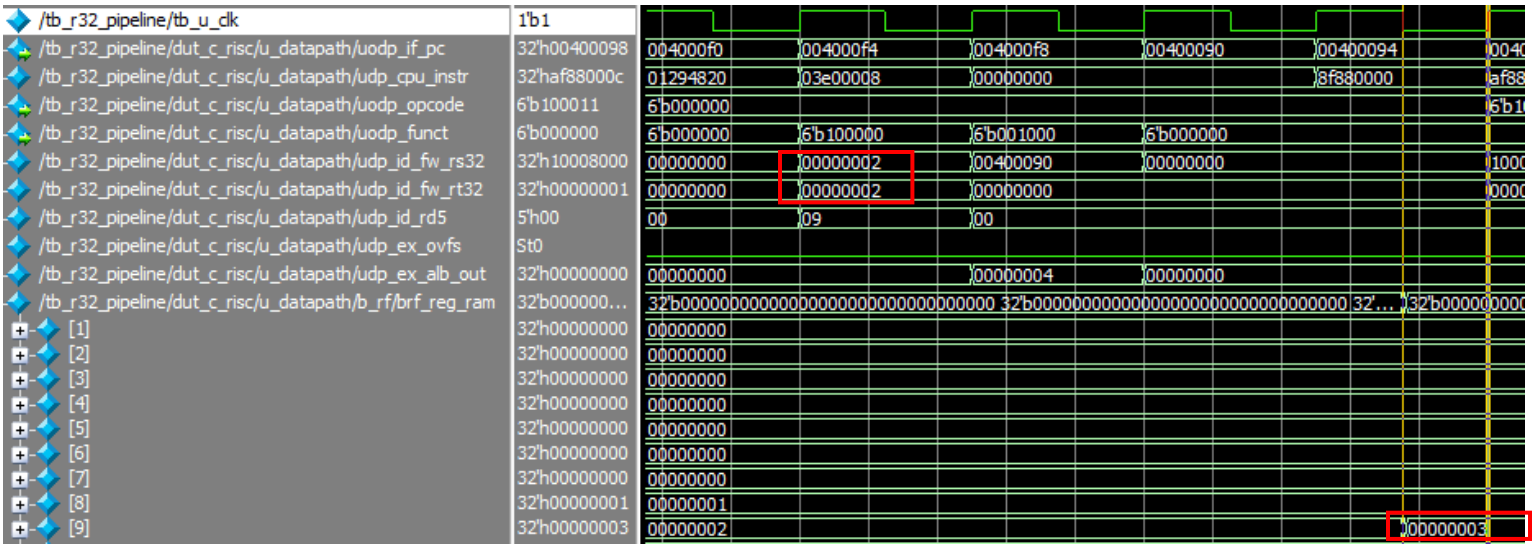

Figure 4. 2: Counterexample's output generated by BMC.

Bounded Model Checking (BMC) in combination with assertion and property has been implemented to most of the MIPS instruction to check the correct behavior of the instruction. Refer Appendix A for the full source code.

However, there are some limitations to implement it in branch instructions. For this MIPS processor design, branch predictor is implemented in this design. The branch predictor operates using sophisticated algorithms and history information to make educated guesses about the direction of conditional branches. The challenge arises because the branch predictor introduces non-determinism into the microprocessor's behavior. The prediction made by the branch predictor determines the path taken by the processor during conditional branches, and this prediction is not explicitly determined by the processor's instruction set architecture. The non-deterministic behavior introduced by the branch predictor makes it difficult to explore all possible paths within a bounded context during BMC.

To facilitate the tracing and analysis of the processor's instruction flow during simulation, a log file has been generated using SystemVerilog. The log file captures the program counter (PC) value, the corresponding instruction in hexadecimal format, and a decoded string representation of the instruction. Refer Appendix B for the output of the log file and Appendix C for the function to generate the log file.

CHAPTER 5

CONCLUSION

In a conclusion, the objectives of this project, which is the development of a self-checking testbench, development of the log file generator, and reduce the time spent in the validation process has been achieved. All objectives are achieved by implementing Bounded Model Checking (BMC) in combination with assertion and property. Through a comprehensive and rigorous verification process, these objectives are successfully accomplished, contributing to the field of microprocessor verification and reliability.

One of the key areas of future work for enhancing the verification process is the development of a random instruction generator. The random instructions generator would serve as a valuable addition to the self-checking testbench methodology, further diversifying the test scenarios and improving the verification coverage for the MIPS processor.

REFERENCES

ChipVerify. (n.d.). SystemVerilog Tutorial. [online] Available at: https://www.chipverify.com/systemverilog/systemverilog-tutorial.

ChipVerify. (n.d.). UVM Tutorial for Beginners. [online] Available at: https://www.chipverify.com/uvm/uvm-tutorial.

Große, D., Kuhne, U. and Drechsler, R., 2005, November. HW/SW co-verification of a RISC CPU using bounded model checking. In 2005 Sixth International Workshop on Microprocessor Test and Verification (pp. 133-137). IEEE.

Gu, Z., Yu, Z., Shen, B. and Zhang, Q., 2002, June. Functional verification methodology of a 32-bit risc microprocessor. In IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions (Vol. 2, pp. 1454-1457). IEEE.

Jensen, J.J. (2019). How to create a self-checking testbench. [online] VHDLwhiz. Available at: https://vhdlwhiz.com/how-to-create-a-self-checking-testbench/ [Accessed 16 Oct. 2022].

Jung, S.P., Song, S.W., Lee, D.H., Kim, K.J., Cho, K.S. and Park, J.S., 2008. Design & verification of 16 bit RISC processor. In Proceedings of the IEEK Conference (pp. 423-424). The Institute of Electronics and Information Engineers.

Kasanko, T. and Nurmi, J., 2004, November. Verification of a 32-bit RISC processor core. In 2004 International Symposium on System-on-Chip, 2004. Proceedings. (pp. 107-110). IEEE.

Kiat, W.P., 2018. The design of an FPGA-based processor with reconfigurable processor execution structure for internet of things (IoT) applications (Doctoral dissertation, UTAR).

Sethulekshmi, R., Jazir, S., Rahiman, R.A., Karthik, R. and Abdulla, M.S., 2016, March. Verification of a RISC processor IP core using SystemVerilog. In 2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET) (pp. 1490-1493). IEEE.

Stanford.edu. (2020). MIPS. [online] Available at: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html.

APPENDICES

Appendix A: BMC Source Code

```
property ADD;
  int rs,rt,rd;
  @(`PC_CLK)
  (`OPCODE == `OP_RTYPE && `FUNCT == `FN_ADD ,
   rs=`RS,
   rt=`RT,
   rd=`RD
   //, $display("value of rs is %0h, value of rt is %0h, expected data at REG[%0h]
is %0h",rs,rt,rd, (rs+rt))
   )
   |-> `DELAY_ID_EX (~`OVERFLOW) //check no overflow is not happenned
in next cc
   |-> `DELAY_EX_WB (`REG_RAM[rd] == (rs+rt));
endproperty
.
.
.
sequence LB_WB (int rt);
  int data;
  @(`PC_CLK)
  (~`RESET, data = `DCACHE_DATA
  )
  `DELAY_MEM_WB (`REG_RAM[rt] == {{24{data[7]}},data[7:0]});
endsequence

property LB;
  int rt, data;
  @(`PC_CLK or `ITL_PC_EN)
  (`OPCODE == `OP_LB,
   rt=`RT_REG
   )
   //|-> `DELAY_ID_EX (`ITL_PC_EN)
   //|-> (`ITL_PC_EN)
   |-> `DELAY_ID_MEM LB_WB(rt)
endproperty
```

Appendix B: Instruction Log

```
 4500.0 ns      | 00400024   | jal 0x0100015       => PC:0x00400054        | User
program code
  4550.0 ns     | 00400028   | sll $r0, $r0, 0 / NOP                       | User program
code
  5350.0 ns     | 00400054   | addi $r16, $r0, 0x0014                      | User program
code
  5400.0 ns     | 00400058   | addi $r17, $r0, 0xfff8                      | User program
code
  5450.0 ns     | 0040005c   | addi $r8, $r17, 0x006c                      | User program
code
  6250.0 ns     | 00400060   | addiu $r18, $r17, 0x0002                    | User program
code
  6300.0 ns     | 00400064   | sub $r19, $r18, $r8                         | User program
code
  6350.0 ns     | 00400068   | subu $r20, $r18, $r8                        | User program
code
  6400.0 ns     | 0040006c   | addu $r21, $r19, $r19                       | User program
code
  6450.0 ns     | 00400070   | jr $r31                                | User program code
  6500.0 ns     | 00400074   | sll $r0, $r0, 0 / NOP                       | User program
code
  6550.0 ns     | 00400028   | sll $r0, $r0, 0 / NOP                       | User program
code
  6600.0 ns     | 0040002c   | and $r8, $r18, $r19                         | User program
code
  6650.0 ns     | 00400030   | andi $r9, $r8, 0x000f                       | User program
code
  6700.0 ns     | 00400034   | nor $r10, $r8, $r9                          | User program
code
```

Appendix C: Log File Generation Function

```
function automatic string decodeMIPSInstruction(input logic [31:0] pc, input logic [31:0] instruction);
  logic [5:0] opcode;
  logic [4:0] rs, rt, rd, shamt;
  logic [15:0] imm;

  opcode = instruction[31:26];
  rs = instruction[25:21];
  rt = instruction[20:16];
  rd = instruction[15:11];
  shamt = instruction[10:6];
  imm = instruction[15:0];

  case (opcode)
    // R-type instructions
    6'b000000: begin
      case (instruction[5:0])
        6'b100000: return $sformatf("add $r%0d, $r%0d, $r%0d", rd, rs, rt);
        6'b100001: return $sformatf("addu $r%0d, $r%0d, $r%0d", rd, rs, rt);
        6'b100010: return $sformatf("sub $r%0d, $r%0d, $r%0d", rd, rs, rt);
        6'b100011: return $sformatf("subu $r%0d, $r%0d, $r%0d", rd, rs, rt);
```

```
  6'b100100: return $sformatf("and $r%0d, $r%0d, $r%0d", rd, rs, rt);
  6'b100101: return $sformatf("or $r%0d, $r%0d, $r%0d", rd, rs, rt);
  6'b100110: return $sformatf("xor $r%0d, $r%0d, $r%0d", rd, rs, rt);
  6'b000000: if (rd == 0)
               return $sformatf("sll $r%0d, $r%0d, %0d / NOP", rd, rt, shamt);
             else
               return $sformatf("sll $r%0d, $r%0d, %0d", rd, rt, shamt);
  6'b000010: return $sformatf("srl $r%0d, $r%0d, %0d", rd, rt, shamt);
  6'b000011: return $sformatf("sra $r%0d, $r%0d, %0d", rd, rt, shamt);
  6'b001000: return $sformatf("jr $r%0d", rs);
  6'b001001: return $sformatf("jalr $r%0d, $r%0d", rd, rs);
  6'b001100: return "syscall";
  6'b010000: return $sformatf("mfhi $r%0d", rd);
  6'b010001: return $sformatf("mthi $r%0d", rs);
  6'b010010: return $sformatf("mflo $r%0d", rd);
  6'b010011: return $sformatf("mtlo $r%0d", rs);
  6'b011000: return $sformatf("mult $r%0d, $r%0d", rs, rt);
  6'b011001: return $sformatf("multu $r%0d, $r%0d", rs, rt);
  6'b100111: return $sformatf("nor $r%0d, $r%0d, $r%0d", rd, rs, rt);
  6'b101010: return $sformatf("slt $r%0d, $r%0d, $r%0d", rd, rs, rt);
  6'b101011: return $sformatf("sltu $r%0d, $r%0d, $r%0d", rd, rs, rt);
  default: return "Unknown";
endcase
```

```
    end
    6'b010000: return $sformatf("mfc0 $r%0d, $r%0d", rt, rd);
    6'b010001: return $sformatf("mtc0 $r%0d, $r%0d", rt, rd);

    // I-type instructions
    6'b001000: return $sformatf("addi $r%0d, $r%0d, 0x%h", rt, rs, imm);
    6'b001001: return $sformatf("addiu $r%0d, $r%0d, 0x%h", rt, rs, imm);
    6'b001100: return $sformatf("andi $r%0d, $r%0d, 0x%h", rt, rs, imm);
    6'b001101: return $sformatf("ori $r%0d, $r%0d, 0x%h", rt, rs, imm);
    6'b001110: return $sformatf("xori $r%0d, $r%0d, 0x%h", rt, rs, imm);
    6'b001111: return $sformatf("lui $r%0d, 0x%h", rt, imm);
    6'b000100: return $sformatf("beq $r%0d, $r%0d, 0x%h      => PC:0x%h", rs,
rt, imm,convertBranchAddressToPC(imm, pc));
    6'b000101: return $sformatf("bne $r%0d, $r%0d, 0x%h      => PC:0x%h", rs,
rt, imm,convertBranchAddressToPC(imm, pc));
    6'b000110: return $sformatf("blez $r%0d, 0x%h          => PC:0x%h", rs,
imm,convertBranchAddressToPC(imm, pc));
    6'b000111: return $sformatf("bgtz $r%0d, 0x%h          => PC:0x%h", rs,
imm,convertBranchAddressToPC(imm, pc));
    6'b001010: return $sformatf("slti $r%0d, $r%0d, 0x%h", rt, rs, imm);
    6'b001011: return $sformatf("sltiu $r%0d, $r%0d, 0x%h", rt, rs, imm);
    6'b001111: return $sformatf("lui $r%0d, 0x%h", rt, imm);
    6'b100011: return $sformatf("lw $r%0d, %0d($r%0d)", rt, imm, rs);
```

```
    6'b101011: return $sformatf("sw $r%0d, %0d($r%0d)", rt, imm, rs);
    6'b100001: return $sformatf("lh $r%0d, %0d($r%0d)", rt, imm, rs);
    6'b100101: return $sformatf("lhu $r%0d, %0d($r%0d)", rt, imm, rs);
    6'b101001: return $sformatf("sh $r%0d, %0d($r%0d)", rt, imm, rs);
    6'b100000: return $sformatf("lb $r%0d, %0d($r%0d)", rt, imm, rs);
    6'b100100: return $sformatf("lbu $r%0d, %0d($r%0d)", rt, imm, rs);
    6'b101000: return $sformatf("sb $r%0d, %0d($r%0d)", rt, imm, rs);

    // J-type instructions
    6'b000010: return $sformatf("j  0x%h                    => PC:0x%h",
instruction[25:0],convertJumpAddressToPC(instruction, pc));
    6'b000011: return $sformatf("jal  0x%h                  => PC:0x%h",
instruction[25:0],convertJumpAddressToPC(instruction, pc));



    default: return "Unknown";
  endcase
endfunction

function automatic logic [31:0] convertJumpAddressToPC(input logic [31:0]
jumpInstruction, input logic [31:0] currentPC);
  logic [31:28] upperBits;
  logic [25:0] lowerBits;
```

```
  logic [31:0] newPC ;
  // Extracting the relevant bits from the jump instruction
  upperBits = jumpInstruction[31:28];
  lowerBits = jumpInstruction[25:0];

  // Concatenating the upper bits with the current PC's upper bits

  newPC= {currentPC[31:28], upperBits, lowerBits, 2'b00};

  return newPC;
endfunction

function automatic logic [31:0] convertBranchAddressToPC(input logic [15:0]
imm, input logic [31:0] currentPC);

  logic [31:0] newPC;

  // Calculating the new PC value by adding the branch offset to the current PC
  newPC = currentPC + 4 + {{16{imm[15]}},imm}*4 ;

  return newPC;
endfunction
```

```
function automatic string decodePC(input logic [31:0] pc);
  if (pc >= 32'hBFC00000 && pc <= 32'hBFC01000)
    return "Boot code";
  else if (pc >= 32'h00400000 && pc <= 32'h0041B400)
    return "User program code";
  else if (pc >= 32'h00800180 && pc <= 32'h00804180)
    return "Exception handler code";
  else
    return "Unknown";
endfunction
```