# Design and Verification of Functional Blocks of 32-Bit Microprocessor

**Asha G[1], Dr. Jamuna S[2]**

*[1]PG Student (M.Tech in VLSI Design & Embedded Systems), Dept. of ECE, DSCE, Bengaluru, Karnataka*
*[2]Professor, Dept. of ECE, DSCE, Bengaluru, Karnataka*

---------------------------------------------------------------------***---------------------------------------------------------------------

**Abstract -** *The core component of an electronic system is a processor. Processors are primarily evaluated solely by their performance, speed, and Instruction Set Architecture (ISA). RISC-V is a selected architecture for the design. RISC-V is an open standard instruction set architecture designed to be scalable for a wide range of applications. Pipelining is a standard feature in RISC-V processors. The technique where multiple instructions are overlapped during execution is known as Pipelining. Therefore, pipelining reduces the latency and improves the overall throughput of the processor.*
*The intended work consists of the design of functional blocks of 32-bit RISC-V processor such as I-Cache, Instruction Issuing Unit, INT ALU, D-Cache and Integer Register File. These functional blocks are designed using Verilog HDL and testbench codes are written for them. For D-cache and Integer reg file, the functional verification is performed with the help of Universal Verification Methodology (UVM). The reusable UVM testbench architecture is built to check the functional correctness of these blocks. Code coverage is performed on the functional blocks designed.*

***Key Words*: RISC-V ISA (Instruction Set Architecture), Pipelining, I-Cache, Instruction Issuing Unit, INT ALU, D-Cache, Integer Register File, Verilog HDL, Universal Verification Methodology (UVM).**

## 1.INTRODUCTION

Processors are primarily evaluated solely by their performance, speed, and Instruction Set Architecture (ISA) [1]. RISC-V is a standard free architecture and is designed to be scalable for a wide variety of applications. RISC-V is suitable for use in some specific application fields such as storage, edge computing, and AI applications. Pipelining is a standard feature in RISC-V processors. It is the process of accumulating instructions from the processor through a pipeline. It allows storing and executing instructions in an orderly process. Pipelining increases the overall instruction throughput.
The intended work consists of the design and verification of functional blocks of 32-bit RISC-V processor such as I-Cache, Instruction Issuing Unit, INT ALU, D-Cache and Integer Register File. These functional blocks are designed using Verilog HDL and are simulated & synthesized using Xilinx Vivado. For D-Cache and Integer Register File UVM testbench has been built to check the functional correctness. Code coverage is performed on the functional blocks designed.
Organization of the paper is as follows. Section 2 provides brief about related work. Section 3 describes the RISC-V processor architecture in brief. Section 4 discusses the design of functional blocks. Section 5 briefs about verification of functional blocks. In section 6 simulation results have been discussed, followed by conclusion in Section 7.

## 2. RELATED WORK

The five- stage pipelined RISC-V processor architecture is presented in [1]. The working of pipelined architecture is explained in this paper. Many sub blocks are included in the processor's design. Design of all the sub blocks is explained in detail along with the block diagrams. The designed processor is implemented on Vertex-7 FPGA board. The maximum frequency attained is 40Mhz.

[2] In this paper authors have discussed about a new processor for a SOC. The processor is based on RISC-V Instruction Set Architecture. Verification is performed at module and integration level. The pipeline is verified for varying configurations of instructions to ensure that all corner situations are covered. The processor is built on a Virtex-7 FPGA, and the results are illustrated in terms of frequency and area.

A 16-bit RISC processor is designed in [3]. The design is done using the Verilog HDL. The processor includes the sub blocks, such as, ALU, data memory unit, controller, and register files. Xilinx ISE tool is used to analyse the design.

Design of ALU which performs addition, subtraction, multiplication, code conversion, and shifting operations is presented in [7]. Different operations are selected using multiplexer based on the select lines or control inputs. The design is developed using the Hardware Description Language (HDL). Structural and behavioural modelling are used in designing the ALU. Results of the ALU are compared with MIPS processor, which has shown reduction in power dissipation. Simulation and synthesis of each block in the design is carried using Xilinx ISE to analyse the results.

A study of UVM's characteristics is offered in [8], [9]. These papers outline the benefits, drawbacks, and prospects. To compare conventional verification with UVM-based verification, a SoC test case is provided. An overview of how to use the UVM verification methodology to create a reusable RTL verification environment is provided and also the usage of UVM in the creation of a testbench using synchronous FIFO as a subsystem undergoing evaluation is examined.

## 3. PROCESSOR ARCHITECTURE

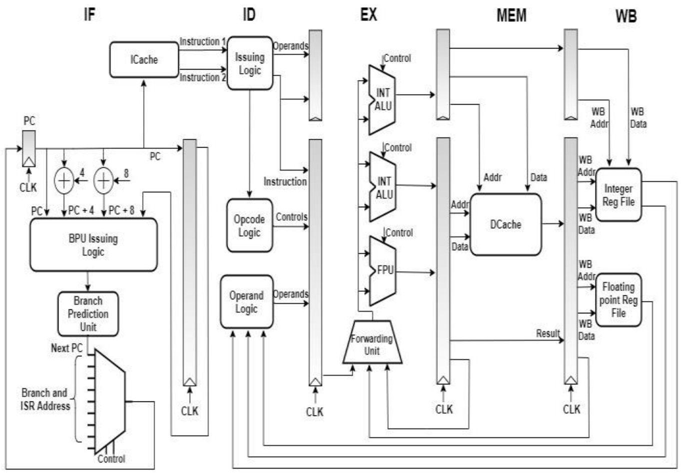


**Fig -1:** RISC-V Pipelined Architecture [1]

A five- stage superscalar pipeline architecture is shown in Fig -1. The five stages are: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write-back (WB).

The first stage is Instruction Fetch (IF), which generates the program memory address using the Program Counter (PC). Using the pc value, two consecutive instructions are obtained from the Instruction Cache and transmitted to the Instruction Decode (ID) stage. The ID stage includes an Instruction Issuing Unit, which sends one or two instructions to the pipeline depending on the dependencies between instructions. The instruction is decrypted in the decode stage, and select signals are created for the multiplexers in the EX-stage for data transmission. The ID stage provides the operands for the Execute-stage. There are three units in the Execute-stage that may execute at most two instructions in the same clock cycle; two of the three units are integer ALUs and one is a floating- point unit. Forwarding lines toggle the multiplexers, allowing either forwarded data or data from register/memory to be used in the Execute stage. The Execute-stage results are passed to the Memory stage, which performs data transactions for load, store, or atomic instructions. In other circumstances, the Execute-stage results are transferred to the Write-Back stage. In the Write-Back stage, distinct register files are implemented for integer and floating-point instructions.

## 4. DESIGN OF FUNCTIONAL BLOCKS

In this section the design of different functional blocks such as, I-Cache & Instruction Issuing Unit, INT ALU, D-Cache and Integer Register File are discussed. The design of all these blocks is done by using Verilog HDL.

### 1) I-Cache & Instruction Issuing Unit

Main memory is too slow to use every time when we want an instruction or a data. Therefore, Instruction Cache and Data Cache are used to speed up memory accesses. Separate D-Cache and I-Cache makes it possible to fetch instructions and data in parallel.

The instruction memory subsystem regulates the flow of instructions. The Instruction Decode stage has an instruction issuing unit (IIU). IIU decodes both instructions from the I-Cache and compares the operand values in each instruction to determine interdependence between the instructions. When an interdependence occurs, the second instruction is placed in a hold register and a "rollback" signal is asserted to alter the next-PC value. The held instruction will be executed in the following clock cycle. The block diagram of IIU is shown in Fig. -2.

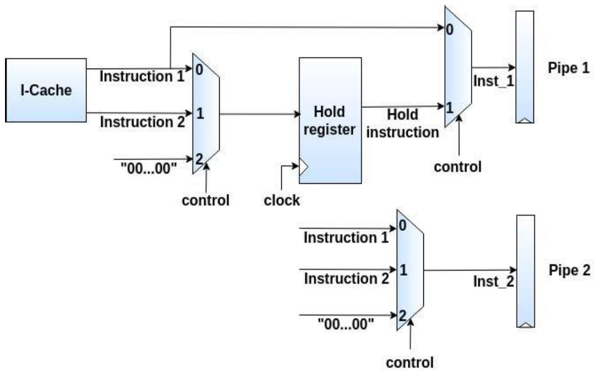


**Fig -2:** Instruction Issuing Unit [1]

### 2) Arithmetic Logic Unit (ALU)

The arithmetic logic unit is a combinational circuit which is capable of executing arithmetic and logic operations. An ALU is a major component of the processor. The 32-bit ALU designed is capable of performing addition, subtraction, increment, decrement, logical, shifting and comparison operations. Fig -3 shows block diagram of ALU.

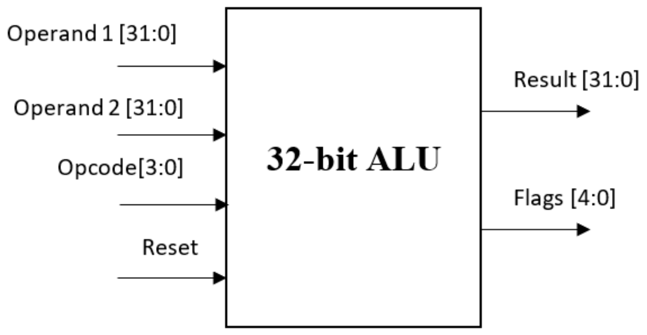


**Fig -3:** Block diagram of ALU

The ALU designed operates on 32-bit operands. The particular function to be performed is controlled by 4-bit opcode, whose value encodes the function according to the Table-1.

**Table-1:** ALU operations

| OPCODE | OPERATION | ALU OUTPUT |
|---|---|---|
| 4'b0000 | Addition | A+B |
| 4'b0001 | Subtraction | A-B |
| 4'b0010 | Increment | A+1 |
| 4'b0011 | Decrement | A-1 |
| 4'b0100 | NOT | ~A |
| 4'b0101 | AND | A AND B |
| 4'b0110 | OR | A OR B |
| 4'b0111 | NAND | ~ (A AND B) |
| 4'b1000 | NOR | ~ (A OR B) |
| 4'b1001 | XOR | A XOR B |
| 4'b1010 | XNOR | ~ (A XOR B) |
| 4'b1011 | Right shift | LSR |
| 4'b1100 | Left shift | LSL |
| 4'b1101 | Greater than | CMPGT |
| 4'b1110 | Less than | CMPLT |
| 4'b1111 | Equal | CMPEQ |

### 3) D-Cache

The data memory subsystem directs the flow of data from the main memory to the Execute and Write-Back stages using the Instruction Decode and Memory stages.
The designed D-Cache stores the 32-bit write data and gives back 32-bit read data from requested 8-bit address. It consists of 256 32-bit memory locations. At every clock cycle, the read operation will take place. The write operation takes place only when write enable signal is high.

### 4) Integer Register File

A register file is an array of processor registers in a central processing unit (CPU). Write Back (WB) stage has integer register file. This unit gets data and address from the Memory stage and stores it in its register file. It stores 32-bit data at 5-bit address. The register file is used as CPU registers for the operations defined by the given instruction.

## 5. VERIFICATION OF FUNCTIONAL BLOCKS

The process of analysing the design to determine whether it meets the specified requirements is known as verification. For the functional blocks such as I-Cache & Instruction Issuing Unit, ALU, D-Cache and Integer Register File testbench codes are written to check whether the designs are functionally correct. For D-cache and Integer Register File UVM testbench has been built using System Verilog.
The Universal Verification Methodology (UVM) is a standardised approach for verifying integrated circuits, ASICs, and SoC architectures. The UVM method was developed by the Accellera Systems Initiative, with the support of several companies including Cadence, Mentor Graphics and Synopsys.
Some benefits of UVM are (1) Modularity and Reusability, (2) Separating test from testbenches, (3) Simulator independent, (4) Sequence methodology, (5) Configuration mechanisms, (6) Factory mechanisms
A typical UVM testbench architecture contains many components. Fig 4 shows a simple UVM testbench diagram.

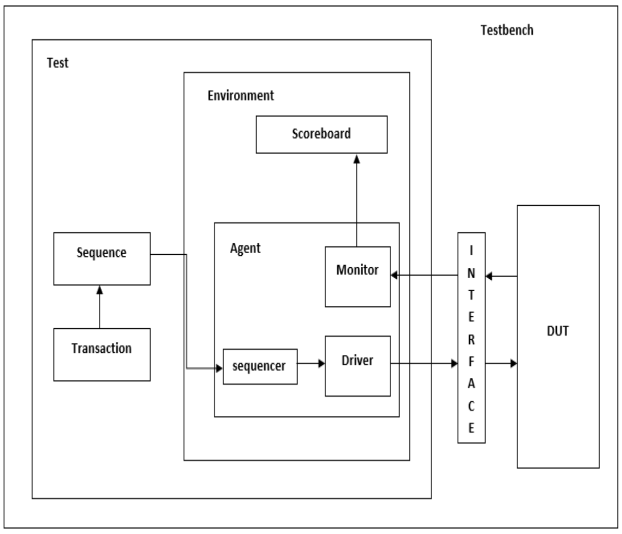


**Fig -4:** UVM testbench architecture

A typical UVM testbench architecture has the following components:

- Testbench- Testbench instantiates unit under test and the UVM test class and set-up the connection between them.
- Test- It is the top-level class responsible for configuring the testbench, start the testbench parts development measure by building a higher level down in the chain of hierarchy, and it starts the stimulus by initiating sequence.
- Environment- groups interrelated verification components such as Agents and Scoreboards.
- Scoreboard- contains checkers and verifies the functionality of the design.
- Agent- groups the verification components which deals with specific interfaces.
- Monitor- Samples the unit under test and reference model interfaces, captures and compares the data included in transactions.
- Driver- Receives data items from the sequencer and sends it to the interfaces for unit under test and reference model.
- Sequencer- The sequencer is in charge of directing transactions (sequence items) generated in sequence to driver or vice-versa.
- Sequence-contains a behavior for generating stimulus.
- Sequence item- consists of data fields required to generate the stimulus.

## 6. RESULTS

The functional blocks that are discussed in section 4 are designed using Verilog HDL and are simulated using Xilinx Vivado. Code coverage analysis is performed on the functional blocks using ModelSim SE. The simulation results and code coverage analysis report are shown below.

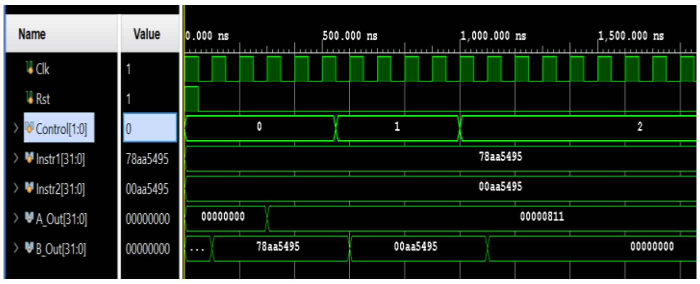


**Fig -5:** Simulation result of Instruction Issuing Unit

Here instructions are manually fed. When Control = 00, B_Out =78aa5495 and when Control = 01, B_Out = 00aa5495. For other Control values B_out = 0. A_out will get values from I-Cache unit.

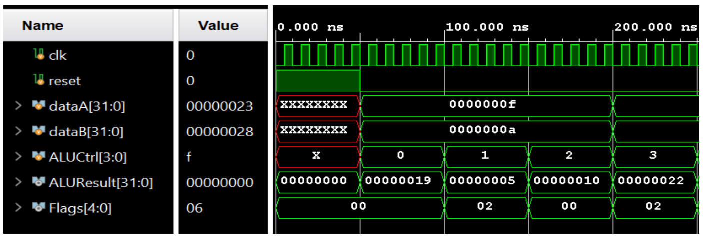


**Fig -6:** Simulation result of Arithmetic operations

In the above simulation, addition, subtraction, increment and decrement operations are performed.

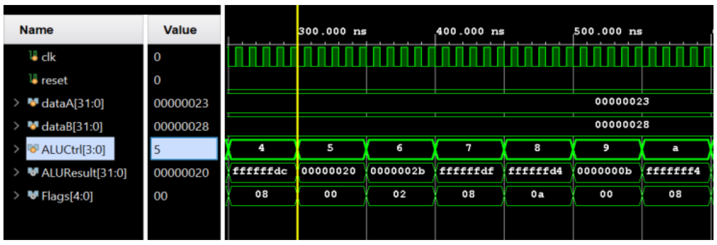


**Fig -7:** Simulation result of Logic operations

In the above simulation, logical operations such as NOT, AND, OR, NAND, NOR, XOR and XNOR are performed.

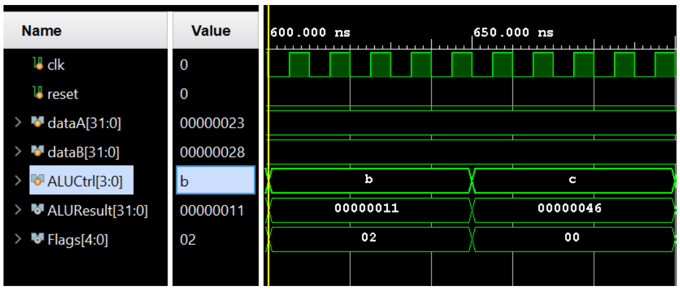


**Fig -8:** Simulation result of Shift operations

In Fig-8 right shift and left shift operations are performed.

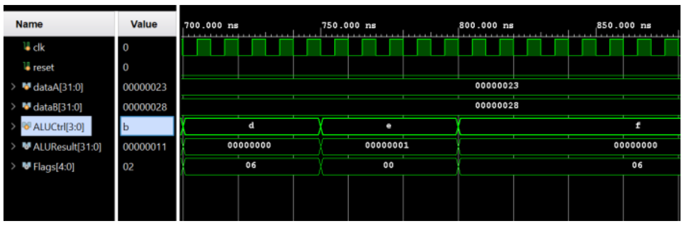


**Fig -9:** Simulation result of Comparison operations

In the above simulation comparison operation are performed

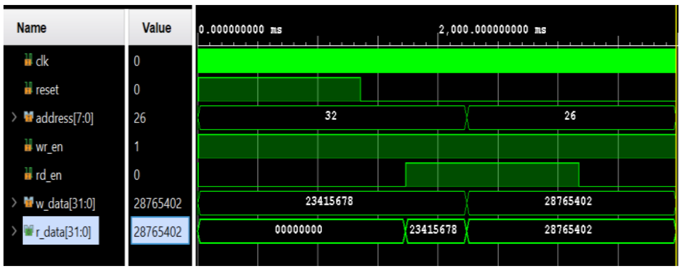


**Fig -10:** Simulation result of D-cache

Above figure shows simulation result of D-cache. The data is stored at particular address and it is read out from that address.

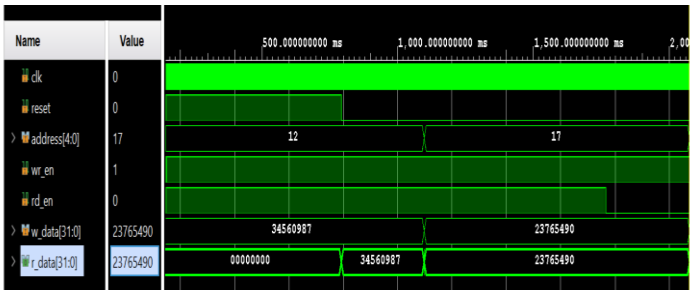


**Fig -11:** Simulation result of Integer Register File

In the above figure simulation results are shown for Integer Register File. Output is based on data and address.

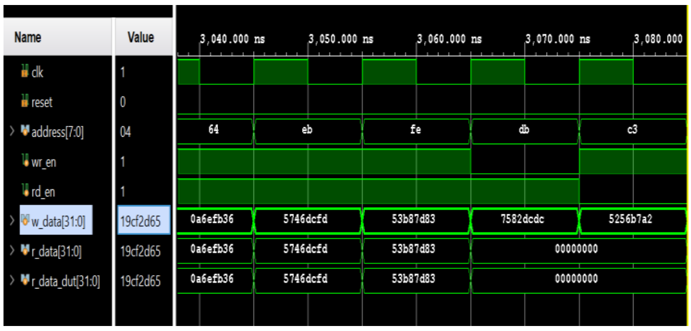
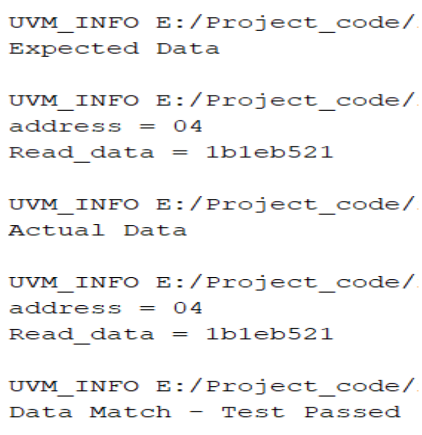


**Fig -12:** UVM simulation result of D-Cache without error

Fig-12 shows the simulation result of D-Cache without error and Fig-13 shows an example of D-Cache output without error.



```
UVM_INFO E:/Project_code/
Expected Data

UVM_INFO E:/Project_code/
address = 04
Read_data = 1b1eb521

UVM_INFO E:/Project_code/
Actual Data

UVM_INFO E:/Project_code/
address = 04
Read_data = 1b1eb521

UVM_INFO E:/Project_code/
Data Match - Test Passed
```

**Fig -13:** An example of D-Cache output without error

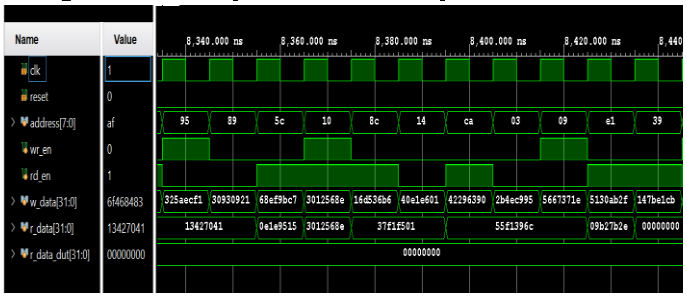
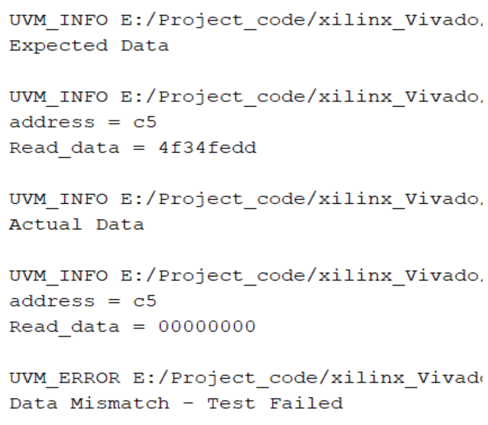


**Fig -14:** UVM simulation result of D-Cache with error



```
UVM_INFO E:/Project_code/xilinx_Vivado.
Expected Data

UVM_INFO E:/Project_code/xilinx_Vivado.
address = c5
Read_data = 4f34fedd

UVM_INFO E:/Project_code/xilinx_Vivado.
Actual Data

UVM_INFO E:/Project_code/xilinx_Vivado.
address = c5
Read_data = 00000000

UVM_ERROR E:/Project_code/xilinx_Vivad
Data Mismatch - Test Failed
```

**Fig -15:** An example of D-Cache output with error

Fig-14 shows the simulation result of D-Cache with error and Fig-15 shows an example of D-Cache output with error.

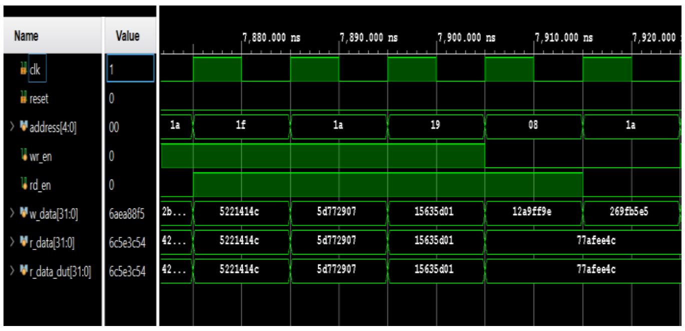
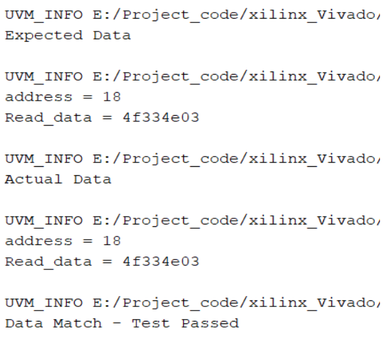


**Fig -16:** UVM simulation result of Integer Register File without error



```
UVM_INFO E:/Project_code/xilinx_Vivado/
Expected Data

UVM_INFO E:/Project_code/xilinx_Vivado/
address = 18
Read_data = 4f334e03

UVM_INFO E:/Project_code/xilinx_Vivado/
Actual Data

UVM_INFO E:/Project_code/xilinx_Vivado/
address = 18
Read_data = 4f334e03

UVM_INFO E:/Project_code/xilinx_Vivado/
Data Match - Test Passed
```

**Fig -17:** An example of Integer Register File output without error

Fig-16 shows the simulation result of Integer Register File without error and Fig-17 shows an example of Integer Register File output without error.

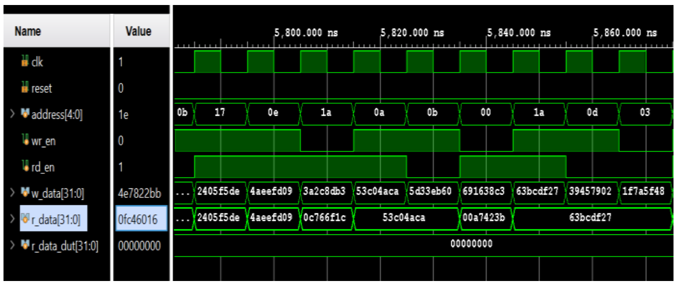
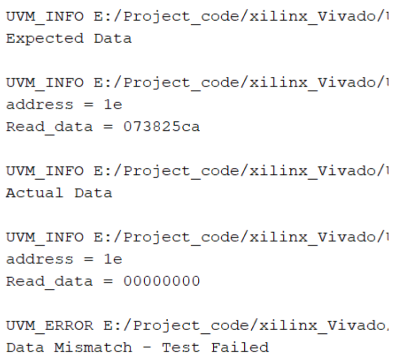

Name
Value
5,800.000 ns
5,820.000 ns
5,840.000 ns
5,860.000 ns
clk 1
reset 0
address[4:0] 1e
wr_en 0
rd_en 1
w_data[31:0] 4e7822bb
r_data[31:0] 0fc46016
r_data_dut[31:0] 00000000
0b 17 0e 1a 0a 0b 00 1a 0d 03
... 2405f5de 4aeefd09 3a2c8db3 53c04aca 5d33eb60 691638c3 63bcdf27 39457902 1f7a5f48
... 2405f5de 4aeefd09 0c766f1c 53c04aca 00a7423b 63bcdf27
00000000

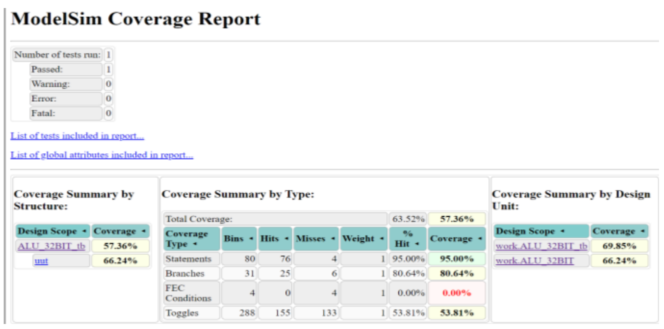
**Fig -18:** UVM simulation result of Integer Register File with error



```
UVM_INFO E:/Project_code/xilinx_Vivado/
Expected Data

UVM_INFO E:/Project_code/xilinx_Vivado/
address = 1e
Read_data = 073825ca

UVM_INFO E:/Project_code/xilinx_Vivado/
Actual Data

UVM_INFO E:/Project_code/xilinx_Vivado/
address = 1e
Read_data = 00000000

UVM_ERROR E:/Project_code/xilinx_Vivado.
Data Mismatch - Test Failed
```

**Fig -19:** An example of Integer Register File output with error

Fig-18 shows the simulation result of Integer Register File with error and Fig-19 shows an example of Integer Register File output with error.



ModelSim Coverage Report

| | |
|---|---|
| Number of tests run: | 1 |
| Passed: | 1 |
| Warning: | 0 |
| Error: | 0 |
| Fatal: | 0 |

List of tests included in report...

List of global attributes included in report...

Coverage Summary by Structure:

| Design Scope | Coverage |
|---|---|
| ALU_32BIT_tb | 57.36% |
| uut | 66.24% |

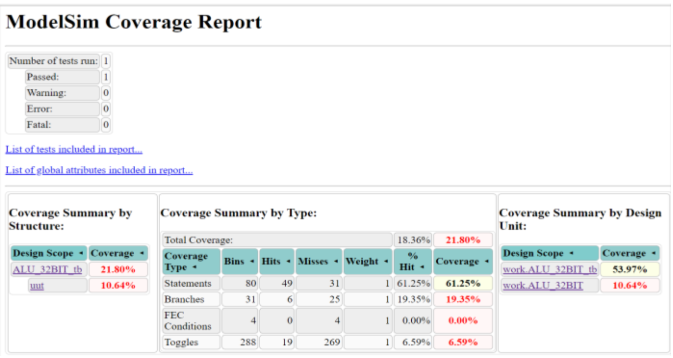
Coverage Summary by Type:

| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
|---|---|---|---|---|---|---|
| Total Coverage: | | | | | 63.52% | 57.36% |
| Statements | 80 | 76 | 4 | 1 | 95.00% | 95.00% |
| Branches | 31 | 25 | 6 | 1 | 80.64% | 80.64% |
| FEC Conditions | 4 | 0 | 4 | 1 | 0.00% | 0.00% |
| Toggles | 288 | 155 | 133 | 1 | 53.81% | 53.81% |

Coverage Summary by Design Unit:

| Design Scope | Coverage |
|---|---|
| work.ALU_32BIT_tb | 69.85% |
| work.ALU_32BIT | 66.24% |

**Fig -20:** Code coverage report of ALU without fault



ModelSim Coverage Report

| | |
|---|---|
| Number of tests run: | 1 |
| Passed: | 1 |
| Warning: | 0 |
| Error: | 0 |
| Fatal: | 0 |

List of tests included in report...

List of global attributes included in report...

Coverage Summary by Structure:

| Design Scope | Coverage |
|---|---|
| ALU_32BIT_tb | 21.80% |
| uut | 10.64% |

Coverage Summary by Type:

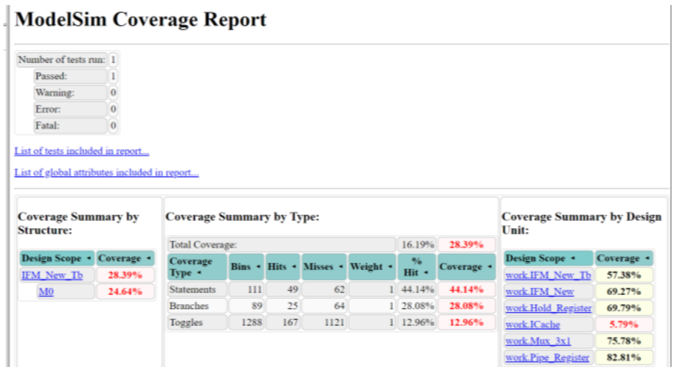
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
|---|---|---|---|---|---|---|
| Total Coverage: | | | | | 18.36% | 21.80% |
| Statements | 80 | 49 | 31 | 1 | 61.25% | 61.25% |
| Branches | 31 | 6 | 25 | 1 | 19.35% | 19.35% |
| FEC Conditions | 4 | 0 | 4 | 1 | 0.00% | 0.00% |
| Toggles | 288 | 19 | 269 | 1 | 6.59% | 6.59% |

Coverage Summary by Design Unit:

| Design Scope | Coverage |
|---|---|
| work.ALU_32BIT_tb | 53.97% |
| work.ALU_32BIT | 10.64% |

**Fig -21:** Code coverage report of ALU with fault

Fig-20 and Fig-21 shows code coverage report of ALU without fault and with fault respectively.



**ModelSim Coverage Report**

| | |
|---|---|
| Number of tests run: | 1 |
| Passed: | 1 |
| Warning: | 0 |
| Error: | 0 |
| Fatal: | 0 |

List of tests included in report...

List of global attributes included in report...

**Coverage Summary by Structure:**

| Design Scope | Coverage |
|---|---|
| IFM_New_Tb | 28.39% |
| M0 | 24.64% |

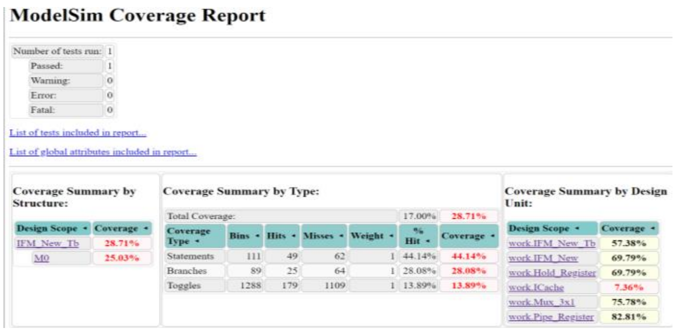
**Coverage Summary by Type:**

| Total Coverage: | | | | | 16.19% | 28.39% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 111 | 49 | 62 | 1 | 44.14% | 44.14% |
| Branches | 89 | 25 | 64 | 1 | 28.08% | 28.08% |
| Toggles | 1288 | 167 | 1121 | 1 | 12.96% | 12.96% |

**Coverage Summary by Design Unit:**

| Design Scope | Coverage |
|---|---|
| work.IFM_New_Tb | 57.38% |
| work.IFM_New | 69.27% |
| work.Hold_Register | 69.79% |
| work.ICache | 5.79% |
| work.Mux_3x1 | 75.78% |
| work.Pipe_Register | 82.81% |

**Fig -22:** Code coverage report of IIU without fault



**ModelSim Coverage Report**

| | |
|---|---|
| Number of tests run: | 1 |
| Passed: | 1 |
| Warning: | 0 |
| Error: | 0 |
| Fatal: | 0 |

List of tests included in report...

List of global attributes included in report...

**Coverage Summary by Structure:**

| Design Scope | Coverage |
|---|---|
| IFM_New_Tb | 28.71% |
| M0 | 25.03% |

**Coverage Summary by Type:**

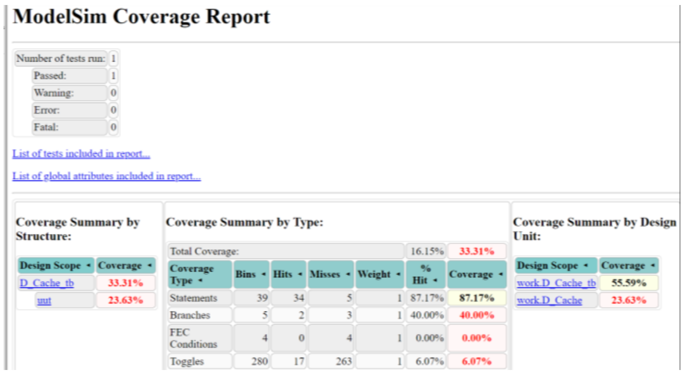
| Total Coverage: | | | | | 17.00% | 28.71% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 111 | 49 | 62 | 1 | 44.14% | 44.14% |
| Branches | 89 | 25 | 64 | 1 | 28.08% | 28.08% |
| Toggles | 1288 | 179 | 1109 | 1 | 13.89% | 13.89% |

**Coverage Summary by Design Unit:**

| Design Scope | Coverage |
|---|---|
| work.IFM_New_Tb | 57.38% |
| work.IFM_New | 69.79% |
| work.Hold_Register | 69.79% |
| work.ICache | 7.36% |
| work.Mux_3x1 | 75.78% |
| work.Pipe_Register | 82.81% |

**Fig -23:** Code coverage report of IIU with fault

Fig-22 and Fig-23 shows code coverage report of IIU without fault and with fault respectively.



**ModelSim Coverage Report**

| | |
|---|---|
| Number of tests run: | 1 |
| Passed: | 1 |
| Warning: | 0 |
| Error: | 0 |
| Fatal: | 0 |

List of tests included in report...

List of global attributes included in report...

**Coverage Summary by Structure:**

| Design Scope | Coverage |
|---|---|
| D_Cache_tb | 33.31% |
| uut | 23.63% |

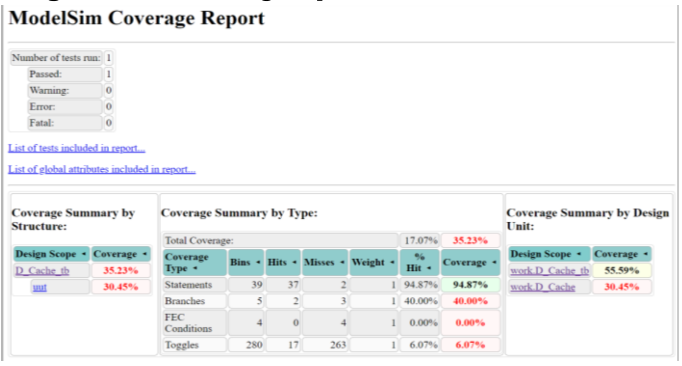
**Coverage Summary by Type:**

| Total Coverage: | | | | | 16.15% | 33.31% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 39 | 34 | 5 | 1 | 87.17% | 87.17% |
| Branches | 5 | 2 | 3 | 1 | 40.00% | 40.00% |
| FEC Conditions | 4 | 0 | 4 | 1 | 0.00% | 0.00% |
| Toggles | 280 | 17 | 263 | 1 | 6.07% | 6.07% |

**Coverage Summary by Design Unit:**

| Design Scope | Coverage |
|---|---|
| work.D_Cache_tb | 55.59% |
| work.D_Cache | 23.63% |

**Fig -24:** Code coverage report of D-Cache without fault



**ModelSim Coverage Report**

| | |
|---|---|
| Number of tests run: | 1 |
| Passed: | 1 |
| Warning: | 0 |
| Error: | 0 |
| Fatal: | 0 |

List of tests included in report...

List of global attributes included in report...

**Coverage Summary by Structure:**

| Design Scope | Coverage |
|---|---|
| D_Cache_tb | 35.23% |
| uut | 30.45% |

**Coverage Summary by Type:**

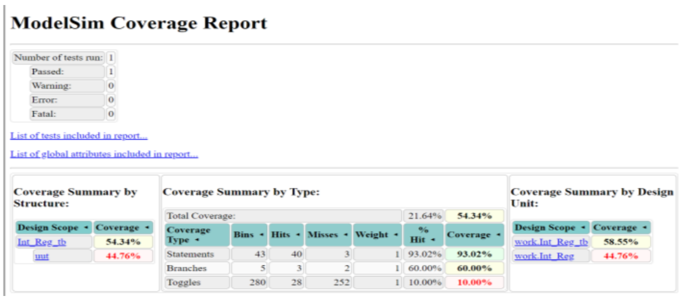
| Total Coverage: | | | | | 17.07% | 35.23% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 39 | 37 | 2 | 1 | 94.87% | 94.87% |
| Branches | 5 | 2 | 3 | 1 | 40.00% | 40.00% |
| FEC Conditions | 4 | 0 | 4 | 1 | 0.00% | 0.00% |
| Toggles | 280 | 17 | 263 | 1 | 6.07% | 6.07% |

**Coverage Summary by Design Unit:**

| Design Scope | Coverage |
|---|---|
| work.D_Cache_tb | 55.59% |
| work.D_Cache | 30.45% |

**Fig -25:** Code coverage report of D-Cache with fault

Fig-24and Fig-25 shows code coverage report of D-Cache without fault and with fault respectively.



**ModelSim Coverage Report**

| | |
|---|---|
| Number of tests run: | 1 |
| Passed: | 1 |
| Warning: | 0 |
| Error: | 0 |
| Fatal: | 0 |

List of tests included in report...

List of global attributes included in report...

**Coverage Summary by Structure:**

| Design Scope | Coverage |
|---|---|
| Int_Reg_tb | 54.34% |
| uut | 44.76% |

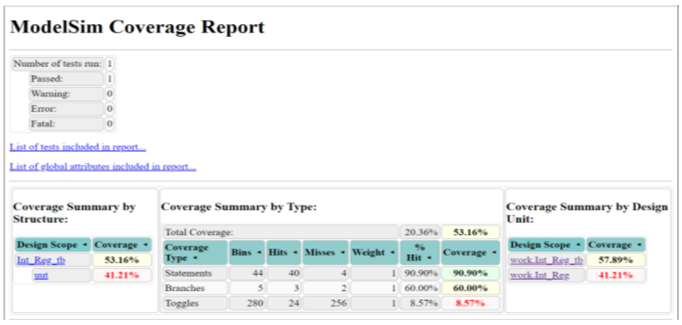
**Coverage Summary by Type:**

| Total Coverage: | | | | | 21.64% | 54.34% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 43 | 40 | 3 | 1 | 93.02% | 93.02% |
| Branches | 5 | 3 | 2 | 1 | 60.00% | 60.00% |
| Toggles | 280 | 28 | 252 | 1 | 10.00% | 10.00% |

**Coverage Summary by Design Unit:**

| Design Scope | Coverage |
|---|---|
| work.Int_Reg_tb | 58.55% |
| work.Int_Reg | 44.76% |

**Fig -26:** Code coverage report of Integer Register File without fault



**ModelSim Coverage Report**

| | |
|---|---|
| Number of tests run: | 1 |
| Passed: | 1 |
| Warning: | 0 |
| Error: | 0 |
| Fatal: | 0 |

List of tests included in report...

List of global attributes included in report...

**Coverage Summary by Structure:**

| Design Scope | Coverage |
|---|---|
| Int_Reg_tb | 53.16% |
| uut | 41.21% |

**Coverage Summary by Type:**

| Total Coverage: | | | | | 20.36% | 53.16% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 44 | 40 | 4 | 1 | 90.90% | 90.90% |
| Branches | 5 | 3 | 2 | 1 | 60.00% | 60.00% |
| Toggles | 280 | 24 | 256 | 1 | 8.57% | 8.57% |

**Coverage Summary by Design Unit:**

| Design Scope | Coverage |
|---|---|
| work.Int_Reg_tb | 57.89% |
| work.Int_Reg | 41.21% |

**Fig -27:** Code coverage report of Integer Register File with fault

Fig-26and Fig-27 shows code coverage report of Integer Register File without fault and with fault respectively.

## 7. CONCLUSION

This paper contains the brief discussion of RISC-V superscalar processor architecture. The work includes design of functional blocks of the architecture such as I-Cache & Instruction Issuing Unit, INT ALU, D-Cache and Integer Register File. They are designed using Verilog HDL. UVM testbench architecture has been discussed briefly and UVM testbench has been built D-Cache and Integer Register File. Functional blocks are simulated using Xilinx Vivado. Code coverage analysis is performed on the functional blocks using ModelSim SE.

## REFERENCES

[1] T. Gokulan, A. Muraleedharan and K. Varghese, "Design of a 32-bit, dual pipeline superscalar RISC-V processor on FPGA," 2020 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, 2020.

[2] Suseela Budi, Pradeep Gupta, Kuruvilla Varghese, and Amrutur Bharadwaj, "A RISC-V ISA Compatible

Processor IP for SoC", International Symposium on Devices, Circuits and Systems (ISDCS) 2018.

[3] Shraddha M. Bhagat, Sheetal U. Bhandari, "Design and Analysis of 16-bit RISC Processor", Fourth International Conference on Computing Communication Control and Automation (ICCUBEA), 2018.

[4] Don Kurian Dennis, Ayushi Priyam, Sukhpreet Singh Virk, Sajal Agrawal, Tanuj Sharma, Arijit Mondal, and Kailash Chandra Ray, "Single Cycle RISC-V Micro Architecture Processor and its FPGA Prototype", 7th International Symposium on Embedded Computing and System Design (ISED), 2017.

[5] Andrew Waterman, "Design of the RISC-V Instruction Set Architecture", PhD thesis, EECS Department, University of California, Berkeley, Jan 2016.

[6] A. Raveendran, V. B. Patil, D. Selvakumar and V. Desalphine, "A RISC-V instruction set processor-micro-architecture design and analysis," 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), Bangalore, 2016.

[7] R. Samanth, A. Amin and S. G. Nayak, "Design and Implementation of 32-bit Functional Unit for RISC architecture applications," 2020 5th International Conference on Devices, Circuits and Systems (ICDCS), Coimbatore, India, 2020.

[8] K. Salah, "A UVM-based smart functional verification platform: Concepts, pros, cons, and opportunities," 2014 9th International Design and Test Symposium (IDT), Algiers, 2014.

[9] T. M. Pavithran and R. Bhakthavatchalu, "UVM based testbench architecture for logic sub-system verification," 2017 International Conference on Technological Advancements in Power and Energy (TAP Energy), Kollam, 2017.