Five Stage Pipelined MIPS Processor Verification using UVM

Anish Gupta, Shubham Lunawat

Abstract

The MIPS processor design is one of the very first foundations of the modern day processor designs. It brings various different fundamental concepts of modern day computing like Pipelining, Data Dependency handling and Forwarding into one place to enhance the capabilities and speed of processing. This project proposes to verify the functionality of a five-stage pipelined MIPS processor. The design under test comes with a total of 16 instructions with a total of 49 variants, 5 pipeline stages and a hazard unit. This project presents the verification of this design. The functionality of each instructions is tested using Constrained Random Verification and the data dependencies being handled by the hazard unit are verified by Assertion Based Verification. To implement these verification techniques in System Verilog Unified Verification Methodology(UVM) is used.

Introduction

The MIPS processor this project intends to verify is a 16 bit, 5 staged pipelined processor. It provides 16 instructions with 49 variants. This verification process of this design involves first verifying individual blocks of each stage and then verifying the overall working of the design. Although this remains a two-step process this is done by

automating the verification process and making use of only one testing system which has interfaces to all the internal blocks as well as the main input-output blocks. By this process, the verification can be highly automated, high-speed and accurate.

Design Under Test The design consists of a 16 bit, 5 stage pipelined processor. The stages involved are as follows:-

- **Instruction Fetch**: This stage holds all instructions to be executed. It also holds a program counter to point the current instruction being executed.
- **Decode Stage**: This stage is used to simplify an instruction and coordinate the working of the processor via various control signals. The general purpose registers are also present in this stage for storing data at runtime.
- **Execution stage**: This stage performs the arithmetic and logical instructions.
- **Memory Access**: This stage has the memory unit and acts as the main data memory storage of the processor.
- **Write back**: The final stage is the write back stage where the data is written back to the appropriate register in the register file after the operations have been performed.

In addition to the above, it also consists of a Hazard Control Unit that takes the necessary steps depending on various data dependencies that occur in the processor working.

Design Specifications The design makes use of these instructions given in table:1.
$A = sc1[15:0]$, $B = scr2[15:0]$, $C = dest[15:0]$, $a = scr1[7:0]$,
$b = scr2[7:0]$, $c = dest[7:0]$, $x = scr[15:8]$, $y = scr2[15:8]$, $z = dest[15:8]$

Instruction	Function	Opcode	Ctrl	Reconfig	Dest	Scr1	Scr2/Mem
Type			1 bit	2 bits			
Addition	C=A+B	0	X	11	[dest]	[scr1]	[scr2]
	z=x+y	0	X	10	[dest]	[scr1]	[scr2]
	c=a+b	0	X	1	[dest]	[scr1]	[scr2]
Subtraction	C=A-B	1	X	11	[dest]	[scr1]	[scr2]
	z=x-y	1	X	10	[dest]	[scr1]	[scr2]
	c=a-b	1	X	1	[dest]	[scr1]	[scr2]
Increment	C=A+1	11	X	11	[dest]	[scr1]	[scr2]
	z=x+1	11	X	10	[dest]	[scr1]	[scr2]
	c=a+1	11	X	1	[dest]	[scr1]	[scr2]
Decrement	C=A-1	10	X	11	[dest]	[scr1]	[scr2]
	z=x-1	10	X	10	[dest]	[scr1]	[scr2]
	c=a-1	10	X	1	[dest]	[scr1]	[scr2]
And/Nand	$C = A\&B$	100	0	11	[dest]	[scr1]	[scr2]
	$z = x\&y$	100	0	10	[dest]	[scr1]	[scr2]
	$c = a\&b$	100	0	1	[dest]	[scr1]	[scr2]
	$!C = A\&B$	100	1	11	[dest]	[scr1]	[scr2]
	$!z = x\&y$	100	1	10	[dest]	[scr1]	[scr2]
	$!c = a\&b$	100	1	1	[dest]	[scr1]	[scr2]
Or/Nor	C=A|B	101	0	11	[dest]	[scr1]	[scr2]
	$z = x\|y$	101	0	10	[dest]	[scr1]	[scr2]
	$c = a\|b$	101	0	1	[dest]	[scr1]	[scr2]

	$C = A \mid B$	101	1	11	[dest]	[scr1]	[scr2]
	$z = x \mid y$	101	1	10	[dest]	[scr1]	[scr2]
	$c = a \mid b$	101	1	1	[dest]	[scr1]	[scr2]
Exor/Exnor	C=A$\hat{\text{B}}$	110	0	11	[dest]	[scr1]	[scr2]
	$z = x^y$	110	0	10	[dest]	[scr1]	[scr2]
	$c = a^b$	110	0	1	[dest]	[scr1]	[scr2]
	$C = A^{\ B}$	110	1	11	[dest]	[scr1]	[scr2]
	$z = x^{\ y}$	110	1	10	[dest]	[scr1]	[scr2]
	$c = a^{\ b}$	110	1	1	[dest]	[scr1]	[scr2]
Buffer	C=A	111	0	11	[dest]	[scr1]	XXX
	z=x	111	0	10	[dest]	[scr1]	XXX
	c=a	111	0	1	[dest]	[scr1]	XXX
Inversion	$C = !A$	111	1	11	[dest]	[scr1]	XXX
	$z = !x$	111	1	10	[dest]	[scr1]	XXX
	$c = !a$	111	1	1	[dest]	[scr1]	XXX
Multiplication	C=a*b	1000	X	1	[dest]	[scr1]	[scr2]
	$C = x * y$	1000	X	10	[dest]	[scr1]	[scr2]
Shift	$C = left_shift(A)$	1100	0	11	[dest]	[scr1]	[shift val]
	$C = right_shift(A)$	1100	1	11	[dest]	[scr1]	[shift val]
Load	C=mem	1010	X	11	[dest]	XXX	[mem]
Store	Mem=A	1011	X	11	XXX	[scr1]	[mem]
Move	C=A	1001	0	11	[dest]	[scr1]	XXX
and	z=x	1001	0	10	[dest]	[scr1]	XXX

Move	c=a	1001	0	1	[dest]	[scr1]	XXX
Immediate	z=8bit data	1001	1	10	"{X ,{8-bit data}}"		
	c=8bit data	1001	1	1	"{X , {8-bit data}}"		
JMP	Jmp unconditional	1101	X	XX	"{XXXXXX , {3-bit disp}}"		
NOP	End(waste a cycle)	1110	X	XX	XXX	XXX	XXX
EDP	EOP(End of Operation	1111	X	XX	XXX	XXX	XXX

Table 1: Instruction Set

The design follows the following specification:

1. **Board**: Kintex7
2. **Time Period**: 50ns
3. **Instructions**: 16

System Level Description Apart from the instruction memory and the register memory the processor contains different components that work in synchronization and perform various operations. These components are as follows:-

Data Memory The Data Memory is a memory unit that the processor uses to store a temporary or final output of different instructions it is working on. This block is almost same as the register memory. It has 8 different memory locations each of 16 bit to store values. It has following parameters:-

1. **Input signal**:

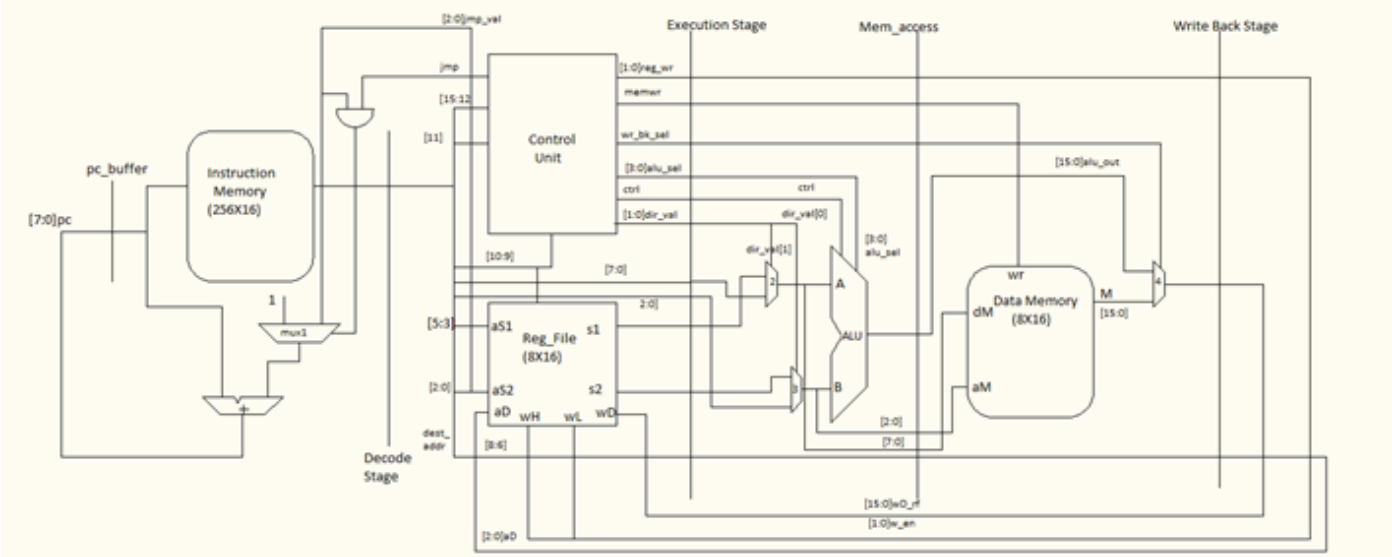

Figure 1: System Level Block Diagram

- [15 : 0dM- Data to be written.
- [2 : 0]aM- Address of the data in the memory.
- clk- For synchronization
- wr- Data is written when this pin is high.

2. **Output signal**:

- [15 : 0m- Data that is read from the memory.
- [2 : 0]aM- Address of the data in the memory.
- clk- For synchronization
- wr- Data is written when this pin is high.

Control Unit This block simplifies an instruction received from the instruction memory and hence controls the working of various blocks of the processor to perform in a

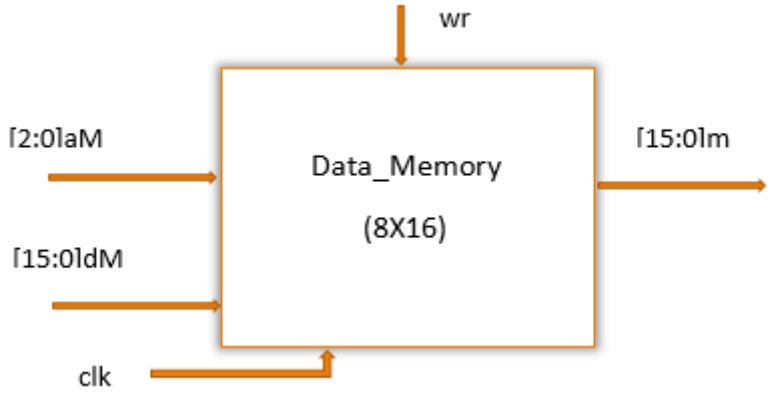

Figure 2: Data Memory Unit

specific way.

Arithmetic and Logic Unit This block is the calculating head that performs various instructions depending on the input signals it has been provided with. The ALU in the design performs the following type of operations:

1. Add/Subtract Operations
2. Logical Operations
3. Multiplication
4. Shift Operations

Hazard Control Unit To handle data dependency based instructions the design makes use of a special unit called Hazard Control Unit which is responsible for preventing incorrect outputs by implementing forwarding. The Hazard Unit covers the following dependent instructions:

1. Add Add

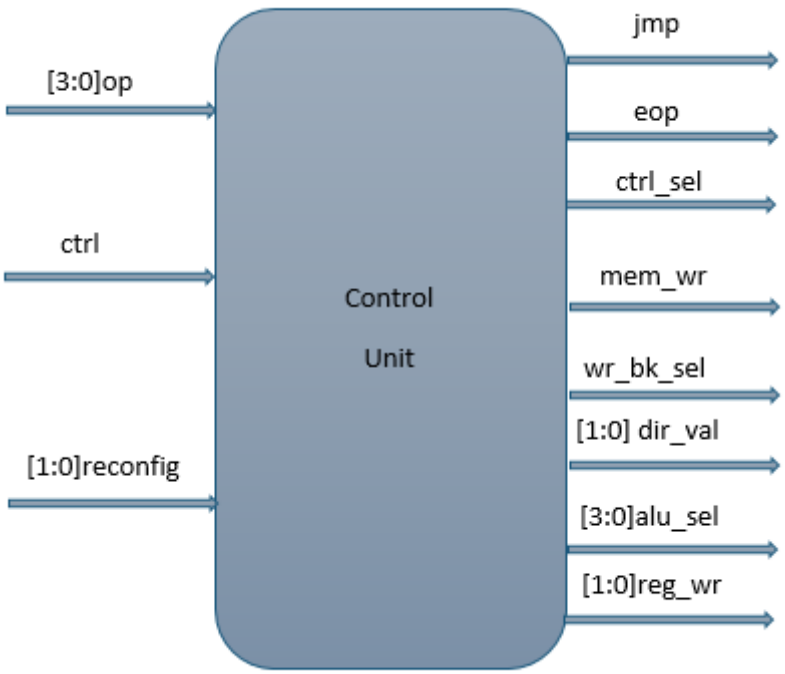

Figure 3: Control Unit

2. Add Add Add
3. Load Add
4. Load xxx Add
5. Add Store
6. Add xxx Store
7. Load Move
8. Mov Store
9. Jmp 0
10. Load Store
11. Store Load
12. Mvih Mvil Mov

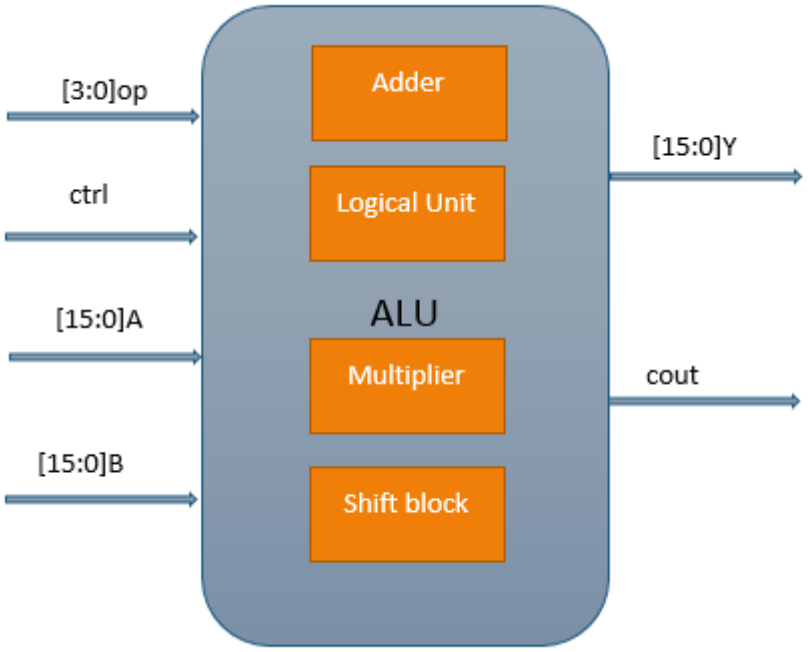

Figure 4: Arithmetic and Logic Unit

Documentation Available The design comes with the following list of documentation:

- **Project Report**: This document give a brief overview of the design of the processor. It describes various different instructions that the processor can operate. It also describes the various units within the processor.
- **Verilog Files**: These are the files which define the entire processor using HDL.
- **README.md**: This file described how to setup the project and get started with using the processor.

Interface The table:2 shows the input-output signals of the design. It must be noted that the processor is a closed system, ie. there are only memory units holding the data both in input and output. Since it is a closed system there is no concrete input or output signals. In the given table we show only the internal signals which carry the data most relevant to our testing.

Name	Direction	Width	Description
inst_in	input	15 : 0	The instruction fed to the pipeline to decode and operate on.
reg_write	output	15 : 0	This is the write back value that can be written to the register for particular conditions.
wr_reg_address	output	2 : 0	It specifies the register address where the value is to be written.
wr_reg_enable	output	1 : 0	This signal is asserted true when writing to the register file.
jump	output	2 : 0	The value of increment in pc.
jump_enable	output	1	States the condition when pc+jump needs to be taken.
mem_write	output	15 : 0	This signal has the write value that can be written to the Data Memory for particular conditions.
wr_mem_addr	output	2 : 0	The address where the value can be written in the memory.

mem_enable	output	1	The value is written only when this signal is asserted true.
reg_read	output	15 : 0	This signal specifies the sources that are read from the register file and moved to the execute stage.
rd_reg_enable	output	1 : 0	This signal specifies the MSB or LSB that needs to be read from a register address.
rd_reg_address	output	2 : 0	Specifies the address from where data is read from.
rd_mem_addr	output	2 : 0	Specifies the memory address from where data is read from.

Table 2: Input/Output Signals

Verification Plan

This project divides the process of verification in the following parts:

1. **Data Inependent Instruction Testing**: This involves testing the instructions and their functionality in the processor. From the design specifications it is known that for any 16 bit number as the input to the DUT, there is always a valid output that should be seen. This permits the use of *Constrained Random Verification* for testing all the instructions. Here for find the coverage of the CRV the bins are designed based on the instructions. Each instruction has a cover point of it's own with individual set of bins based on the source, destination, etc which

are the inputs to that function. The only problem with this approach is that it requires only one instruction to be tested at a time. Since we cannot handle data dependencies while performing CRV, as then there would be need of keeping track of the the instructions being called and also the state of the memories locally in the verification environment. Thus, to keep the verification efficient we can separate the testing of individual instructions and data dependent instructions into two separate tests.

2. **Data Dependent Instruction Testing**: This involves testing only data dependent instructions. The designs specifies the Data Dependent Instructions that it is capable of processing eg: MOV, MOV, MOV. Such instructions are tested using Assertion Based Verification. The test cases are designed to test all the specified data dependent instructions.

For performing verification on the design the project makes use of the *Universal Verification Methodology*. The figure:5 shows the intended setup of the system. The table:3 shows a detailed list of operations to be tested.

Functionality to be tested	Type of Coverage	Test Description
Addition	Constrained Random Verification	The working of this instruction is tested by calling this instruction and testing if the output has the sum of the input operands
Subtraction	Constrained Random Verification	The working of this instruction is tested by calling this instruction and testing if the output has the difference of the input operands

Increment	Constrained Random Verification	The working of this instruction is tested by calling this instruction and testing if the output has the value one more than the input operands value
Decrement	Constrained Random Verification	The working of this instruction is tested by calling this instruction and testing if the output has the value one less than the input operands value
AND & NAND Operation	Constrained Random Verification	The working of this instruction is tested by calling this instruction and testing if the output is AND or NAND of the input operands
OR & NOR Operation	Constrained Random Verification	The working of this instruction is tested by calling this instruction and testing if the output is OR or NOR of the input operands
XOR & XNOR Operation	Constrained Random Verification	The working of this instruction is tested by calling this instruction and testing if the output is XOR or XNOR of the input operands
Multiplication	Constrained Random Verification	This instruction is randomly called on two 8 bit numbers and output is checked to be the product of the inputs
SHIFT Left and Right Operation	Constrained Random Verification	The instruction is called on a random input data and checked if the data gets shifted by one bit

LOAD Data	Constrained Random Verification	A random location of the memory is chosen and data of that memory are loaded into the register file. Then a test is run to check if the register file contains that data.
STORE Data	Constrained Random Verification	A random source register is chosen and data of that register are stored into the memory. Then a test is run to check if the memory contains that data.
MOV Instruction	Constrained Random Verification	Random source and destination registers are chosen for moving the data from the source to destination. Then a test is run to check if the data has actually moved.
JUMP Instruction	Constrained Random Verification	A random number is chosen for incrementing the instruction counter. From the output it is tested of the instruction counter has actually been incremented or not.
MOVI Instruction	Assertion Based Verification	All MOVI instructions used to initialize the processor are tested by asserting each operation once
NOP Instruction	Assertion Based Verification	All NOP instructions used to initialize the processor are tested by asserting each operation once

EOP Instruction	Assertion Based Verification	The EOP instructions working is tested by calling the instruction in a loop for >10 times and asserting the EOP to be performed each time.

Table 3: Verification Functionality and Testcases

Coverage Bins To validate the complete functional behavior of the processor all the instructions have their individual coverage bin. The table:4 shows a detailed list of these bins and their distribution.

Instruction	Signal	Coverage Bins
Addition	Instruction	$[16'h0000 : 16'h0FFF]$
Subraction	Instruction	$[16'h1000 : 16'h1FFF]$
Increment	Instruction	$[16'h3000 : 16'h3FFF]$
Decrement	Instruction	$[16'h2000 : 16'h2FFF]$
AND NAND	Instruction	$[16'h4000 : 16'h4FFF]$
OR NOR	Instruction	$[16'h5000 : 16'h5FFF]$
EXOR EXNOR	Instruction	$[16'h6000 : 16'h6FFF]$
Buff Inv	Instruction	$[16'h7000 : 16'h7FFF]$
Multiplication	Instruction	$[16'h8000 : 16'h8FFF]$
ShiftL ShiftR	Instruction	$[16'hC000 : 16'hCFFF]$
Load	Instruction	$[16'hA000 : 16'hAFFF]$
Store	Instruction	$[16'hB000 : 16'hBFFF]$
MOV MOVI	Instruction	$[16'h9000 : 16'h9FFF]$
Jump	Instruction	$[16'hD000 : 16'hDFFF]$

NOP	Instruction	$[16'hE000 : 16'hEFFF]$

Table 4: Coverage Bins

Layered Testbench

The project makes use of the Unified Verification Methodology framework with System Verilog. It makes use of Object Oriented Programming Constructs like polymorphism and encapsulation. The figure:5 show the structure of the System Verilog program written. In the following paragraphs each of the files in the testbench are described in detail.

Processor Testbench Package The Testbench Package file consists of only include statements to include all the other test modules. This is helpful is packaging the entire testbench into one unit for easy compilation and running.

Processor Test The test module extends *uvm_test* from the UVM structure. It is the main file which initiates and runs tests on the DUT. It is responsible for creation of the environment and the transaction modules. It initiates the testing by initiating the sequencer.

Processor Top The top module is responsible for instantiating the DUT and Interface. It is also responsible for initiating the various tests in the testbench.

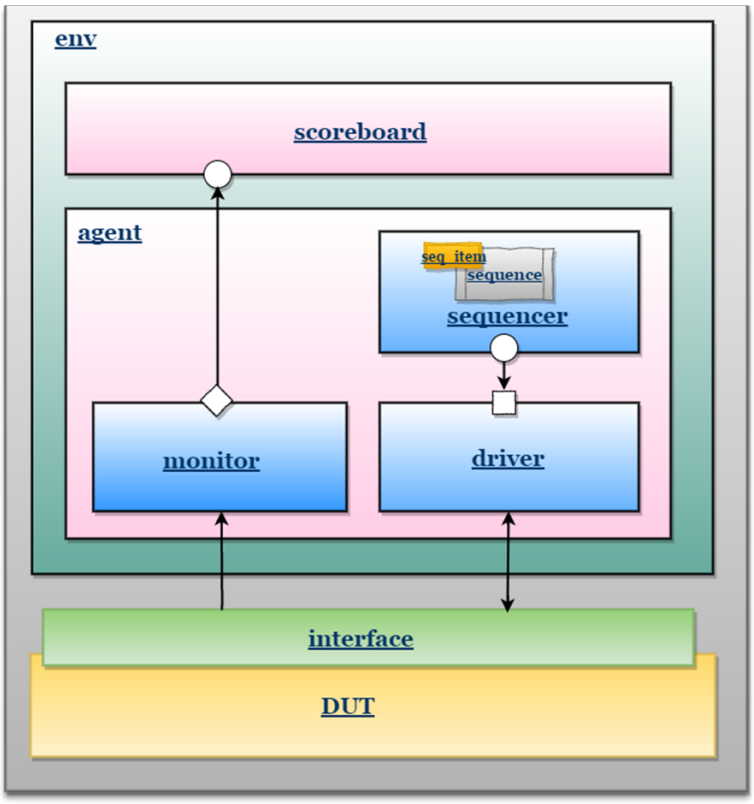

Figure 5: Details of UVM Structure Used

Processor Environment The environment module is needed to setup agent, monitor, subscriber and the scoreboard. In the build phase of the environment these units are instantiated. While in the connect phase the connections between the driver and agent, the monitor and scoreboard, the monitor and subscriber is created.

Processor Agent The agent module contains the driver and the sequencer. It is responsible for their instantiation and the connection of the ports from the sequencer to the driver.

Processor Interface This module is used for holding the connection between the DUT and the Testbench, specifically the Monitor and Driver hold this connection with the

DUT. Table shows the signals being connected and the type of signal being connected.

Signal Name	Monitor	Driver
[7:0]pc	Input	N/A
[15:0]inst_out	Input	N/A
[15:0]inst_in	N/A	Output
[15:0]reg_data	Input	N/A
[1:0]reg_en	Input	N/A
[2:0]reg_add	Input	N/A
[15:0]mem_data	Input	N/A
mem_en	Input	N/A
[2:0]mem_add	Input	N/A

Processor Monitor The monitor module is responsible for handling the inputs to the testbench from the DUT. For every 20 instruction change there is a condition in the monitor for recording the signal. Once these signals are recorded they are sent to the scoreboard of the testbench for further processing.

Processor Driver This testbench makes use of two drivers. The first driver is used to run all Constrained Random Verification tests. While the other driver is used to run all the Assertion Based Test. It must be noted that these drivers are used separately by the test module. The driver on driving the signals to the DUT also sends the same signals to the scoreboard.

Processor Scoreboard The scoreboard is the main module performing all the checking of the signals and reporting of the verification. It gets signals from the monitor

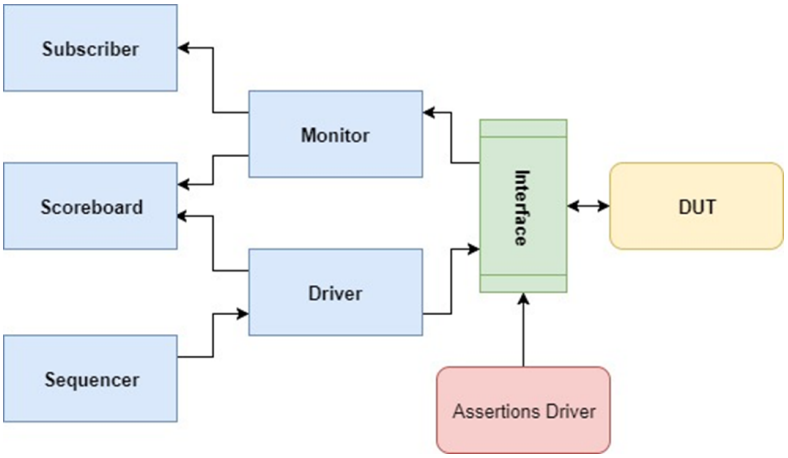

Figure 6: Program Flow

and the driver. From the driver it only gets the data about what instruction was sent to the DUT and from monitor it gets the output signals from the DUT as well. The input instruction is decoded and processed in the DUT and then the output signals from the monitor are compared to the expected output signals.

Processor Sequence This is the base transaction object being send around from the Sequencer to the Driver, from the Driver to the Scoreboard and from the Monitor to the Scoreboard. This object holds all the instruction and output related data.

Processor Subscriber The subscriber is responsible for keeping track of line coverage and group coverage. It takes input from the monitor and keeps track of the transactions that have been hit. Finally this unit generates the report giving statistics of code coverage and group coverage.

Conclusion

The project milestone covers basic testing and understanding of the design. It provides a roadmap to be followed for all future task. At this stage, the 5 stage MIPS pipelined processor has been studied. The design makes use of 5 pipelined stages to perform its operations. The basic blocks include ALU, Control Unit, Instruction Memory, Register Memory and Data Memory. The process of verification has been divided into three separate steps, basic testing of all the instructions, testing of individual units, detailed testing of all instructions and detailed testing of instruction combinations. Of these steps, the first set results have been displayed. The results show the working of all the instructions of the design. The report finalizes the use of Constraint Random Verification for all subunits of the design and the use of Direct Tests for overall verification. The preliminary test results show the proper working of the design. It also gives insight into the working of the design, specifically timing importance in a pipelined design. For the future tasks, the use of UVM has been finalized to streamline the testing.

Refrences

- C. Spear and Greg Tumbush, SystemVerilog for Verification, Third Edition, Springer, 2012.
- SystemVerilog topics at: http://testbench.in
- SystemVerilog Tutorial at http://electrosofts.com/systemverilog
- A Practical Guide to Adopting the Universal Verification Methodology (UVM) by Sharon Rosenberg, Kathleen Meade Cadence Design Systems (2010).
- David Money Harris and Sarah L. Harris, Digital Design and Computer Architecture, Second Edition, Elsevier, 2013.