FUNCTIONAL VERIFICATION OF MBOX AND I-CLASS RISC-V BRANCH INSTRUCTIONS

A project Report

Submitted by

V.N.V. SAI RAM KUMAR (EE20M024)

in partial fulfilment of requirements

for the award of the degree of

MASTER OF TECHNOLOGY

Department of Electrical Engineering

Indian Institute of Technology Madras

May 2022

Thesis Certificate

This is to undertake that the project report titled **FUNCTIONAL VERIFICATION OF MBOX AND I-CLASS RISC-V BRANCH INSTRUCTIONS**, submitted by me to the Indian Institute of Technology Madras, for the award of M.Tech, is a bonafide record of research work done by me under the supervision of **Prof. Kamakoti Veezinathan.** The contents of this project report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Chennai **V.N.V. SAI RAM KUMAR**

Date: **EE20M024**

Dr. Kamakoti Veezinathan
Guide
Professor
Dept. of Computer Science
Director, IIT Madras.

Dr. Pradeep Sarvepalli
Co-guide
Associate Professor
Dept. of Electrical Engineering
IIT Madras.

Acknowledgements

I express my gratitude to Prof. Kamakoti Veezinathan for giving an opportunity to work in the Shakti Processor projects. I am grateful to him for providing constant suggestions though weekly review meetings.

I also extend my gratitude to Prof. pradeep Sarvepalli, who supported me as co-guide from Electrical department.

My sincere thanks to my mentor Smt. Lavanya Jagan, verification lead at RISE lab, IIT Madras, for mentoring and guiding throughout the project. I express my gratitude to her and prof. Kamakoti Veezinathan for supporting me during tough times.

I would like to thank Ms. Divya, project associate at RISE lab, IIT Madras for explaining the project and tool setup and also guiding me whenever Iam stuck in the project.

Finally, I would like to thank my family for their constant support and encouragement.

Abstract

Dividers and multipliers are basic building blocks for arithmetic and logic units, digital signal processing applications, microprocessors, micro controllers and other data processing units. Implementing these blocks in RISC-V specification, which is open source, provides extra edge over other architectures and it helps in collaborative research. Verification of these blocks and RISC-V instructions has to be done with highest quality since any unnoticed bug leads to the wastage of entire chip as well as time and effort.

CoCoTb (Coroutine Cosimulation based Testbench) , a python based verification framework has been used to verify the design blocks due to its vast libraries and easy to learn nature without compromising on quality. The goal of the project is to verify divider and multiplier blocks using cocotb and hunt for potential bugs. Work has been done also to verify the RISC-V branch instructions by taking possible scenarios.

TABLE OF CONTENTS

Title Page

Acknowledgement………………………………………………………………… 3

Abstract……………………………………………………………………….….. 4

Table of contents………………………………………………………………… 5

List of figures…………………………………………………………….......... 8

List of tables………………………………………………………………….. 9

Abbreviations…………………………………………………………….......... 10

Chapter 1: Introduction……………………………………………………… 11

1.1 Introduction to verification……………………………………………. 11

1.2 Test plan………………………………………………………………… 11

1.3 Coverage……………………………………………………………… 11

1.4 Block level verification……………………………………………… 12

1.5 Core level verification………………………………………………… 13

Chapter 2: Mbox…………………………………………………………… 15

2.1 Non-restoring divider………………………………………………… 15

2.2 Srt radix-2 divider…………………………………………………… 16

2.3 Srt radix-4 divider…………………………………………………… 17

2.4 Four stage pipelined multiplier……………………………………… 18

Chapter 3: RISC-V branch instructions……….. 19

3.1 What is RISC-V……………………………………………………… 19

3.2 I-class processor……………………………………………………… 19

3.2 Branch instructions…………………………………………………… 19

3.2.1 BGE……………………………………………………………… 20

3.2.2 BLT……………………………………………………………… 20

3.2.3 BEQ……………………………………………………………… 21

3.2.4 BNE……………………………………………………………… 21

3.2.5 BGEU…………………………………………………………… 21

3.2.6 BLTU…………………………………………………………… 22

3.2.7 JAL……………………………………………………………… 22

3.2.8 JALR.. 23

3.3 Running a RISC-V test.. 24

Chapter 4: Cocotb.. 28

4.1 Why Cocotb ?.. 28

4.2 Some features of cocotb.. 28

4.2.1 Coroutine.. 28

4.2.2 Yield.. 28

4.2.3 Timer... 29

4.2.4 RisingEdge.. 29

4.2.5 FallingEdge... 29

4.2.6 Clockcycles... 29

4.3 Simple cocotb test... 29

4.4 Test bench architecture.. 31

4.4.1 Transaction... 32

4.4.2 Driver.. 32

4.4.3 Monitor... 32

4.4.4 DUT.. 33

4.4.5 Reference model... 33

4.4.6 Score board... 33

Chapter 5: Verification.. 34

5.1 Verification of srt radix-4 divider....................................... 34

5.1.1 Interface of the design... 34

5.1.2 Test plan.. 34

5.1.3 Verification strategy... 35

5.1.4 Functional Coverage report... 36

5.2 Verification of four stage pipelined multiplier................... 36

5.2.1 Interface of the design... 37

5.2.2 Test plan.. 37

5.2.3 Verification strategy... 38

5.2.4 Functional Coverage report... 39

5.3 Code Coverage report.. 39

5.4 Verification of RISC-V branch instructions........................ 40

5.4.1 Test plan.. 41
5.4.2 Verification strategy... 41
Chapter 6: Verification challenges and bugs........................ 42
Chapter 7: Results.. 43
Chapter 8: Conclusion... 46
8.1 Future work.. 46

References... 47

List of Figures

1.1 Block level verification flow……………………………………………… 12

1.2 Processor core overview………………………………………………… 13

3.1 An assembly test………………………………………………………… 24

3.2 Commands and spike debug mode example……………………………….. 25

3.3 Disassembly file………………………………………………………. 26

3.4 Spike dump……………………………………………………………… 27

3.5 Rtl dump………………………………………………………………… 27

4.1 Verilog code for full adder……………………………………………… 29

4.2 Python test bench for full adder………………………………………… 30

4.3 Makefile………………………………………………………………… 30

4.4 Simulated result………………………………………………………… 31

4.5 Cocotb test bench framework…………………………………………… 31

4.6 Cocotb test bench architecture…………………………………………… 32

5.1 Functional Coverage of srt radix-4 divider……………………………… 36

5.2 Functional Coverage of four stage pipelined multiplier……………………… 39

5.3 Code coverage of non-restoring divider………………………………… 39

5.4 Code coverage of srt radix-2 divider…………………………………….. 39

5.5 Code coverage of srt radix-4 divider…………………………………… 40

5.6 Code coverage of four stage pipelined multiplier………………………….. 40

7.1 Simulated result of non-restoring divider………………………………… 43

7.2 Simulated result of srt radix-2 divider…………………………………… 43

7.3 Simulated result of srt radix-4 divider…………………………………… 44

7.4 Simulated result of four stage pipelined multiplier……………………… 44

7.5 Results of RISC-V branch instructions.. 45

List of Tables

2.1 Quotient digit selection.. 18

5.1 Test plan of srt radix-4 divider... 31

5.2 Test plan of four stage pipelined multiplier...................................... 34

Abbreviations

CoCoTb	Coroutine Cosimulation based Testbench
RISC	Reduced Instruction Set Computer
ALU	Arithmetic and Logic Unit
ISA	Instrucation Set Architecture
MSB	Most Significant Bit
LSB	Least Significant Bit
DUT	Design Under Test
PC	Program Counter
RTL	Register Transfer Level
UVM	Universal Verification Methodology

Chapter 1
INTRODUCTION

1.1 What is design verification?

Design verification means verifying functional or logic correctness of a system. Design verification is most important step of a chip development cycle and it takes as it takes as much as 80% of the chip development cycle. The goal is to verify that our design meets the system specifications. The different approaches in design verification are logic simulation where detailed functionality and timing of design are checked by means of simulation. Functional verification where a functional model is developed that describes the behavioral specification of design without detailed timing simulation and formal verification where the functionality is checked against a mathematical model.

In our work, we have done the functional verification of various divider, multiplier blocks and core verification of RISC-V branch instructions in i-class processor. Some important metrics in the development of testbench are

1.2 Test plan

Test plan contains the set of features of the design to be verified. Before verification, it is important to prepare to test plan for all the features we like to verify. After the development of testbench, test plan can be referred and can check whether a test plan item has been verified or not. So, it keeps track on features to be verified.

1.3 Coverage

Coverage gives information about how much of the functionality of the design has been verified. It has to be defined in functional coverage model in testbench code. We have to specify which values we want to cover for a particular signal. We can also specify cross coverage between two signals i.e. which set of values should both signals cover in a cycle. For example, in fulladder each signal is a binary value, but we need to check cross coverage between the three input signals so that those cover all the eight possible combinations of inputs.

1.4 Block level verification

Block level verification deals with verifying the design blocks as a whole by driving the interface signals of the block. Testbenches should be created at the highest level of abstraction where the design is modelled. For example, the interface signals for a multiplier block could be a multiplier and multiplicand. Driving values to these signals and verifying the functionality is called as block level verification. Most of the time, we work only on the interface signals and won't dive into the internal signals or wires.

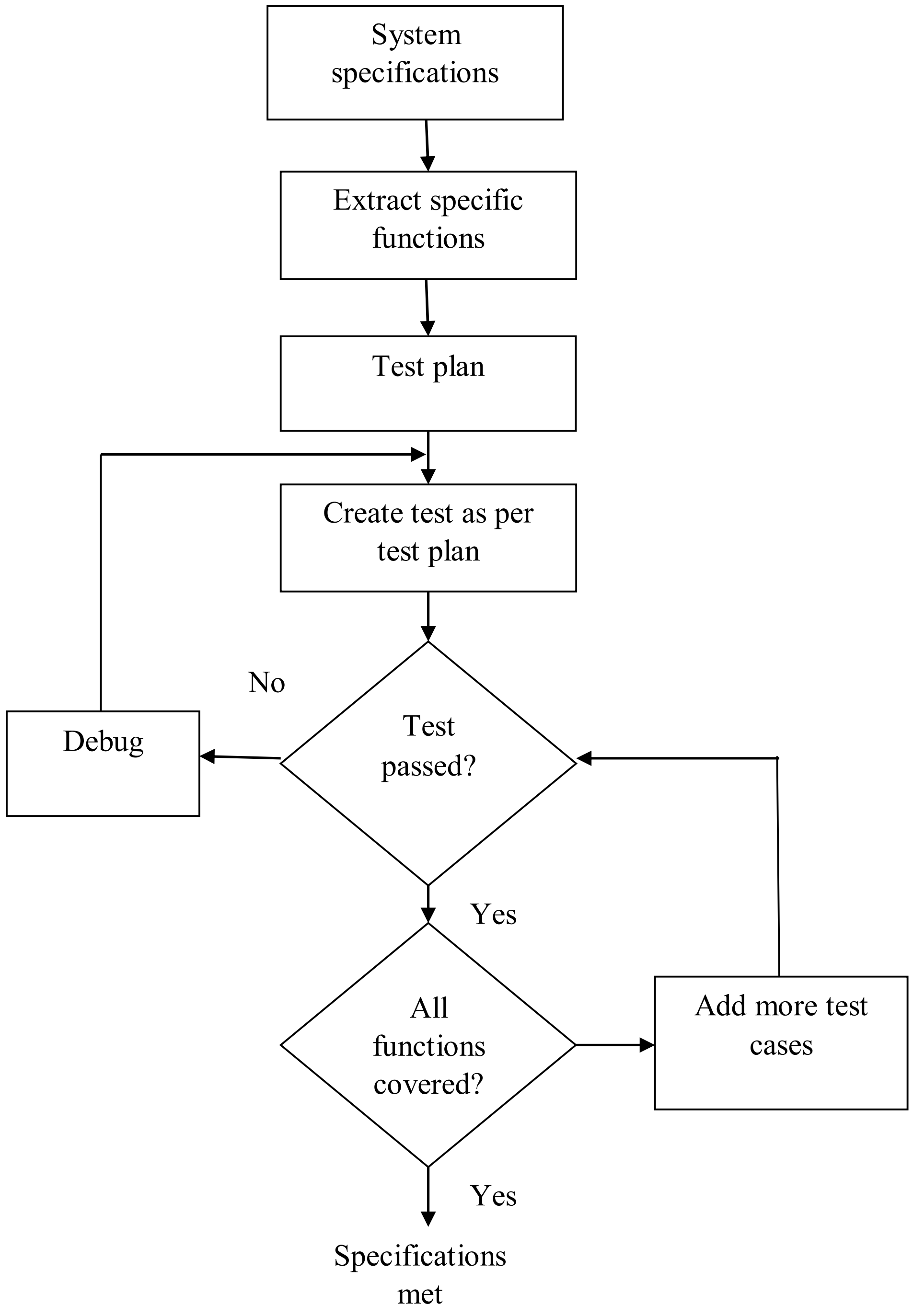


Fig. 1.1: Block level verification flow

The testbenches are used in different verification processes to validate the functionality of the design as it proceeds through increasing levels of refinement and becomes more detailed.

1.5 Core level verification

RISC-V is a load-store architecture and all the operations are handled on registers. Only load and store instructions transfer data to and from memory and data must first be loaded into a register before it can be operated on.

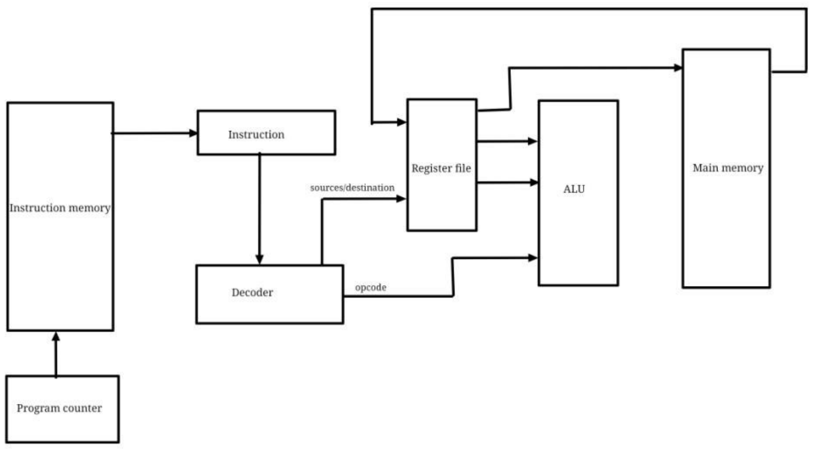


Fig. 1.2: Processor core overview

The instruction to be executed is fetched from instruction memory. Each instruction has predefined hexadecimal value and it has been stored in instruction memory. The next instruction to be executed is stored in program counter as shown in figure. After fetching the instruction, it is decoded by the decoder to its opcode (Ex: add) from its hexadecimal value. Also, the decoder also decodes the source and destination registers, immediate values if any are present. Decoder sends source and destination registers information into a file called register file and opcode information to ALU (if instruction to be executed is an arithmetic operation). Before performing operation, source operands has to be loaded into registers and operate on registers, but not directly load the operands from memory. This is because reading and writing into memory is costly and time consuming. And in RISC-V, we cannot directly write data to memory from ALU, instead the data is fed back to a register in register file and stores into memory by means of store instructions. This is one of the feature of RISC architectute. Not only ALU operations, the instructions can be of operation on memory like load and store, branch instructions etc.

Unlike block verification, core verification involves verification of all these operations whether the instruction is correctly fetched and decoded, whether the registers are operating correctly, whether load from memory and store to memory are functioning correctly. For example, ‘add’ instruction size is 32 bits where it stores the information about source, destination registers and the opcode add itself in those 32 bits. If any of those 32 bits are toggled or shorted to ground then it would not execute ‘add’ instruction, may be it executes any other instruction. There are 32 registers x0 to x31 and x0 is always hardwired to zero, there could be a situation that any other register apart from x0 hardwired to zero where it leads to incorrect processor operation. Therefore, verification of the core involves verification of all these things like using all possible registers as source and destination registers and different scenarios as per the test plan.

Chapter 2

MBOX

Mbox is a name given to the integer multiplier and divider which includes non-restoring divider, srt radix-2 divider, srt radix-4 divider and pipelined multiplier. These blocks are implemented as per RISC-V ISA which are termed as M-extension operations in specification document of RISC-V.

2.1 Non-restoring divider

All the slow division methods are based on standard partial remainder recurrence equation. It involves choosing the quotient digits from a particular set on each cycle and number cycles are equal to number of bits in dividend for radix-2 divider and half of them for radix-4 divider. Generally, in radix-2 dividers, one quotient bit is selected in each cycle and in radix-4 dividers, two quotient bits are selected. The (j+1)th partial remainder is given by the equation

$R_{j+1} = rR_j – Dq_j$ ------------------------ (1)

Where

r is the radix

R_j is the partial remainder of cycle j

D is the divisor

The radix for the non-restoring divisor is 2 and the quotient digit set is 1, -1. The initialization starts by taking the first partial remainder as dividend and multiplication by radix is done by shifting left. So, in non-restoring divisor q_j is always 1 or –1. The steps involved in non-restoring divisor are shown below

Step-1: Initialize the registers. ‘Q’ is dividend, ‘M’ is divisor, ‘n’ is number of cycles, and accumulator ‘A’ is initialized to 0.

Step-2: Check the MSB (sign bit) of A. It is obviously 0 initially.

Step-3: If sign bit of A is 0, then shift left the contents of AQ (performing multiplication with radix) and perform A=A+M (from equation). Otherwise perform A=A-M

Step-4: Check the MSB (sign bit) of A.

Step-5: If it is 1, then make LSB of ‘Q’ as 1, otherwise 0

Step-6: Decrement ‘n’ by 1. If ‘n’ is not equal to zero, then jump to step-2. Otherwise perform the next step

Step-7: If sign bit of A is 1, perform A=A+M. Otherwise leave as it is. Now, the register ‘Q’ contains quotient and ‘A’ contains remainder.

Our non-restoring divider is of 64 bits. Generally, it should take 64 cycles to produce the result. But it needs extra 3 cycles for processing the sign of the result and sign of the operands. The type of division whether signed or unsigned and type of operation whether quotient or remainder is figured out in first clock cycle. If division is of 32 bits, then sign bits are padded. Two’s complement is taken if any operand is negative and sign of result is decided, if signed division, in the second clock cycle. Non-restoring algorithm is carried out for the next 64 clock cycles, deciding each quotient digit in each cycle and returns result in the last clock cycle and two’s complement is taken if the result is negative.

2.2 Srt radix-2 divider

Srt radix-2 divider generates one quotient bit per cycle like non-restoring divider, but this divider has 3 quotient bits to select in each clock cycle, therefore this makes easy in hardware because the choice zero requires only shifting. Selection of partial remainders is based on equation (1) above with initial partial remainder is taken as dividend.

In fractional division, the selection of quotient digits depends on whether the partial remainder range. If $2S_j$, where S_j is the partial remainder in j^{th} cycle, is greater than or equal to 0.5, then q_j, the quotient of j^{th} cycle, is selected as 1 and if it is less than -0.5, the quotient is selected as -1. If $2S_j$ is between -0.5 and 0.5, the quotient is selected as zero. The same can be applied to integer division, if two MSBs' of partial remainder are 01, then quotient bit in that cycle is selected as 1 and if they are 10, then the quotient bit is selected as -1. And, if two MSBs' are either 11 or 00, then quotient bit is selected as 0.

Since, our design is a radix-2 divider of 64 bits, at least 64 clock cycles are required to generate the result producing a quotient bit in each cycle. The type of division whether signed or unsigned and type of operation whether quotient or remainder is figured out in first clock cycle. If division is of 32 bits, then sign bits are padded. Special cases like divide by zero and sign overflow are also checked. Two's complement is taken if any operand is negative and sign of result is decided, dividend is taken as the first partial remainder in the second clock cycle. Each quotient bit is selected in each cycle for the next 63 clock cycles from the set {-1,0,1}. The result is returned in the last clock cycle and two's complement is taken if the result is negative, the sign which is decided in the second clock cycle.

2.3 Srt radix-4 divider

Srt radix-4 divider is similar to srt radix-2, but two quotient digits are selected here in each clock cycle. So, atleast 32 clock cycles are required to generate the 64 bits result. But some extra clock cycles are required to get the sign of the result, to take two's complement etc. So, this is a fast divider but hardware complexity is more. Two quotient bits selected in each cycle depends on four most MSBs' of divisor, and six most MSBs' of partial remainder. Let 'd' be the integer value of four MSBs' of divisor, 'r' be the range of six MSBs' of partial remainder and 'q' be the quotient bit selected. Then the selection of quotient bit is shown in the table below.

d	Range of p	q	d	Range of p	q
8	[-12,-7]	-2	12	[-18,-10]	-2
8	[-6,-3]	-1	12	[-10,-4]	-1
8	[-2,-1]	0	12	[-4,3]	0
8	[2,5]	1	12	[3,9]	1

8	[6,11]	2	12	[9,17]	2
9	[-14,-8]	-2	13	[-19,-11]	-2
9	[-7,-3]	-1	13	[-10,-4]	-1
9	[-3,2]	0	13	[-4,3]	0
9	[2,6]	1	13	[3,9]	1
9	[7,13]	2	13	[10,18]	2
10	[-15,-9]	-2	14	[-20,-11]	-2
10	[-8,-3]	-1	14	[-11,-4]	-1
10	[-3,2]	0	14	[-4,3]	0
10	[2,7]	1	14	[3,10]	1
10	[8,14]	2	14	[10,19]	2
11	[-16,-9]	-2	15	[-22,-12]	-2
11	[-9,-3]	-1	15	[-12,-4]	-1
11	[-3,2]	0	15	[-5,4]	0
11	[2,8]	1	15	[3,11]	1
11	[8,15]	2	15	[11,21]	2

Table 2.1: Quotient digit selection

It takes upto 38 clock cycles to generate the result. Extra 6 clock cycles for choosing the type of operation (quotient or remainder) and the type of division(32 bit or 64 bit) and calculating two's complement if the result is negative.

2.4 Four stage pipelined multiplier

Pipelined multiplier helps in computing the result faster than normal multiplier. A 'k' stage pipeline has 'k' registers in every path from input to output. Computation of result involves calculating partial products which are simply calculated by using AND gates. The speed of a multiplier lies on how efficiently we are adding partial products and how efficiently we design the adder that sums the final stage partial products. Therefore, efficient adder also plays an important role in design of a multiplier.

In this design, the multiplier is pipelined with four stages. It takes multiplier and divides it into parts of 6 bits and obtain partial products with multiplicand and finally adds them to get the product.

Chapter 3

RISC-V BRANCH INSTRUCTIONS

3.1 What is RISC-V

RISC-V is an Instruction Set Architecture (ISA) based on Reduced Instruction Set Computer (RISC) principles. Unlike Intel's x86 and Arm's arm architecture, RISC-V is open source and free to use.

3.2 I-class processor

I-class is a 64-bit out-of-order processor that targets computing, mobile, storage and networking platforms. Its key features include aggressive branch prediction, multithreading, pipelined functional units except divider and square root, non-blocking cache. Operating frequency of the processor is as high as 1.5-2.5 GHz. It supports RV64IMAFDC instructions(multiplier/divider, atomic, single and double precision floating point, compressed).

3.3 Branch instructions

Branch instructions are also called control transfer instructions. These are used to control transfer to other instructions in the program which are far away,maybe either backward or forward. These are RV32 instructions i.e instruction length is 32 bits. The branch instructions bge, blt, beq, bne, bgeu, bltu are named as B-type instructions and their instruction encoding is as follows

31 25	24 20	19 15	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode

Since there are 32 general purpose registers, 5 bits are required to encode each register. So, 10 bits are required for two registers of whom values will be compared for branching. One question is, how long can the execution jump in a program? It depends on the imm value that we give. Since RISC-V is byte addressable and each instruction is 32 bits wide, aimm value of 8 corresponds to a jump of two instructions. But, what if the imm value is odd? It never happens because we are always setting the LSB of imm to 0 as we can see above. imm value is in signed representation and it is of 12 bits. So, the execution can jump a memory of 4MB in both forward and backward direction. There are 6 branch instructions B-type. To differentiate them, we require 3 bits, so funct3 does that differentiation between branch instructions. Opcode is same for all branch instructions. All these instructions use PC relative addressing i.e., jump is relative to program counter.

3.3.1 BGE

Branch greater than or equal to (bge) compares contents in two registers and takes the target address if value in first register is greater than or equal to value in second register.

Syntax: bge rs1, rs2, label.

Jump to 'label' if value in rs1 is greater than or equal to value in rs2. The syntax is the same for the coming branch instructions, namely blt, beq, bne, bgeu, bltu.

Usage:

```
label1: li x1, 2

label2: li x2, 1

        bge x1, x2, label1      #branch1

        bge x2, x1, label2      #branch2
```

As shown, above x1 is loaded with 1 and x2 with 2. Since [x1]>[x2] the branch1 will be taken and execution jumps to 'label1'. Otherwise, it jumps to label2.

3.3.2 BLT

Branch less than (blt) compares contents in two registers and takes the target address if the value in first register is less than the value in second register.

Usage:

```
label1: li x1, 2

label2: li x2, 1

        blt x1, x2, label1      #branch1

        blt x2, x1, label2      #branch2
```

As shown above, x1 is loaded with 1 and x2 with 2. Since [x1]>[x2] the branch1 will not be taken. So, the branch2 will be taken here and execution jumps to label2.

3.3.3 BEQ

Branch equal to (beq) compares contents in two registers and takes the target address if value in first register is equal to value in second register.

Usage:

```
label1: li x1, 2

label2: li x2, -1

        beq x1, x2, label1      #branch1

        beq x1, x1, label2      #branch2
```

As shown above, x1 is loaded with 1 and x2 with 2. Since [x1] is not equal to [x2] the branch1 will not be be taken. Obviously, branch2 will be taken and execution jumps to label2.

3.3.4 BNE

Branch not equal to (bne) compares contents in two registers and takes the target address if value in first register is not equal to value in second register.

Usage:

```
label1: li x1, 2

label2: li x2, 1

        bne x1, x2, label1      #branch1

        bne x1, x1, label2      #branch2
```

As shown above, x1 is loaded with 1 and x2 with 2. Since [x1] is not equal to [x2] the branch1 will be taken and execution jumps to label1.

3.3.5 BGEU

Branch greater than or equal to unsigned (bge) compares contents in two registers and takes the target address if value in first register is greater than or equal to value in second register. The difference between bge and bgeu is that, bgeu makes unsigned comparison between two registers i.e the contents in the registers should unsigned. While, imm values can still be both signed and unsigned.

Usage:

```
label1: li x1, 2

label2: li x2, 1

        bgeu x1, x2, label1     #branch1

        bgeu x2, x1, label2     #branch2
```

As shown above, x1 is loaded with 1 and x2 with 2. Since [x1]>[x2] the branch1 will be taken and execution jumps to 'label1'. Otherwise, it jumps to label2.

3.3.6 BLTU

Branch less than unsigned (bltu) compares contents in two registers and takes the target address if the value in first register is less than the value in second register.

Usage:

```
label1: li x1, 2

label2: li x2, 1

        bltu x1, x2, label1     #branch1

        bltu x2, x1, label2     #branch2
```

As shown above, x1 is loaded with 1 and x2 with 2. Since [x1]>[x2] the branch1 will not be taken. So, the branch2 will be taken here and execution jumps to label2. The difference between blt and bltu is that, bltu makes unsigned comparison between two registers i.e the contents in the registers should unsigned. While, imm values can still be both signed and unsigned.

3.3.7 JAL

Jump and link(JAL) is an unconditional branch instruction. No comparison of registers to make jumps, so the execution jumps straight to the target address when using this instruction. So, this instruction can be used to call a function.

31 12	11 7	6 0
imm[20|10:1|11|19:12]	rd	opcode

Syntax: jal rd, imm

Imm is a 21 bit signed integer where LSB is always set to zero. In RV32 the jump should always be multiple of 4, since each instruction is 4 bits wide. So, the jump range is ±1MB. If imm[1:0] is not zero, then it leads to misaligned exception. rd is destination register and it saves the return address (the address of the next instruction after jal). Instruction encoding of JAL done based on opcode value and its value is 1101111. 5 bits needed for ‘rd’, since it can be any of the available 32 registers.

Usage:

```
loop: li x3, 3
addi x4, x3, 1
        add x5, x4, x3
        jal x1, loop      #branch
        and x6, x5, x4    #return address
```

The execution jumps to ‘loop’ after jal instruction and PC is loaded with the address of the ‘loop’ and x1 stores the address of ‘and’ instruction. If the return address is not needed then we can simply use ‘j loop’. It’s a pseudo instruction which executes ‘jal x0, imm’. So, the return address is not stored as x0 is always hardwired to zero.

3.3.8 JALR

Jump and link register (JALR) is an unconditional branch instruction like JAL. It is an I-format instruction where it has a source register, destination register and an immediate operand. Unlike all other branch instructions the LSB is not forced to zero.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode

Imm value is a 12 bit signed integer and LSB is not forced to zero. Any misalignment with memory leads to misalignment exception. rd stores the return address (the address of the next instruction after jalr). The value of funct3 is 000 and opcode is 1100111.

Syntax: jalr rd, rs1, imm

The target address is calculated not only by using imm but also with the contents in rs1 register. The effective imm value is [rs1]+imm.

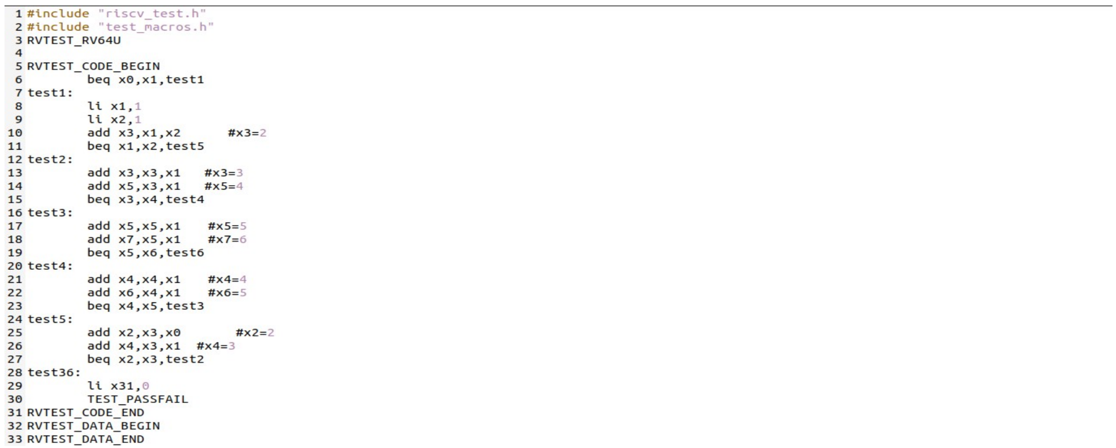
Usage:

```
loop: li x3, 3

addi x4, x3, 1

        add x5, x4, x3

        jalr x1, x5, loop  #branch

        and x6, x5, x4     #return address
```

If return address and imm values are not required, we can write ‘jr x1’ where x1 contains the target address. It is a psedo instruction for jalr x0, x1, imm, where imm is 0. Along with AUIPC instruction, where it stores the upper 20 bits of PC into a register, jalr can be used to make long unconditional jumps to anywhere in the program.

3.3 Running a RISC-V test

First of all, an assembly test code has to be build. Let us consider a simple assembly test as shown below.

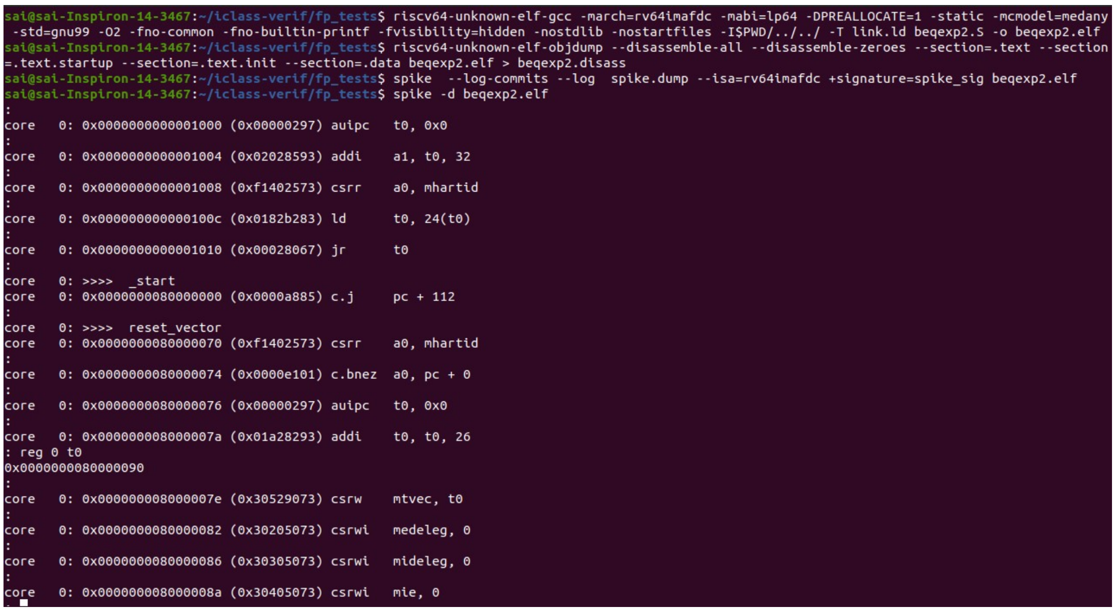


```
#include "riscv_test.h"
#include "test_macros.h"
RVTEST_RV64U

RVTEST_CODE_BEGIN
        beq x0,x1,test1
test1:
        li x1,1
        li x2,1
        add x3,x1,x2      #x3=2
        beq x1,x2,test5
test2:
        add x3,x3,x1   #x3=3
        add x5,x3,x1   #x5=4
        beq x3,x4,test4
test3:
        add x5,x5,x1    #x5=5
        add x7,x5,x1    #x7=6
        beq x5,x6,test6
test4:
        add x4,x4,x1    #x4=4
        add x6,x4,x1    #x6=5
        beq x4,x5,test3
test5:
        add x2,x3,x0       #x2=2
        add x4,x3,x1  #x4=3
        beq x2,x3,test2
test36:
        li x31,0
        TEST_PASSFAIL
RVTEST_CODE_END
RVTEST_DATA_BEGIN
RVTEST_DATA_END
```

Fig. 3.1: An assembly test

Above shows the simple RISC-V test for beq instruction. RVTEST_CODE_BEGIN indicates the start of assembly code section and RVTEST_CODE_END indicates the end of it. TEST_PASSFAIL determines whether the test passed or failed. For some of the tests, we need to store the data into memory of load data from memory into registers, we need to add the data for these operations in RVTEST_DATA_BEGIN section and it allocates the addresses for the data. All these sections and the codes for their operations are defined in the riscv_test.h and test_macros.h files.

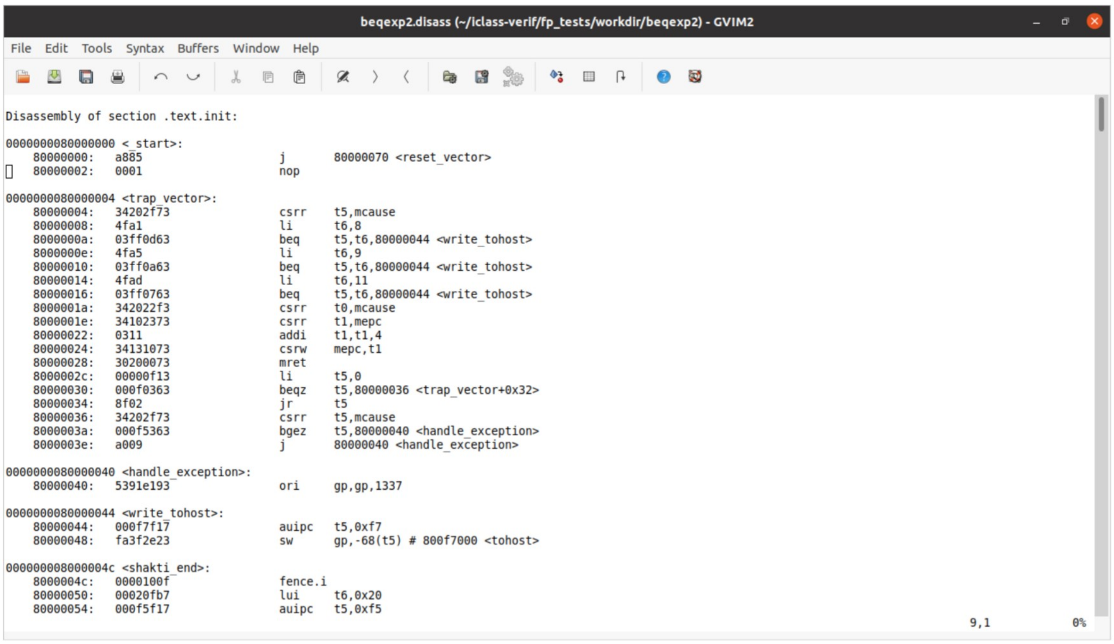
After building the test, we need to run it on a simulator to check whether the test is passed in the core and we use spike simulator for the same. We need to generate an elf file to run and test on the spike. Also, we generate disassembly file for the test which contains the program counter value of each instruction, the hexadecimal value of the instruction. We can run the elf file on spike and can check whether it is passing or not. But to check whether it is passing on i-class core or not, we need to generate a spike dump file which is generated through spike and a rtl dump file which is generated through i-class core. Obviously, for the test to pass both the dump files should be identical. To generate rtl dump, we use elf2hex which converts elf file to hex files which are suitable for verilog's readmemh. The commands to generate elf, disassembly, spike dump are shown below



```
sai@sai-Inspiron-14-3467:~/iclass-verif/fp_tests$ riscv64-unknown-elf-gcc -march=rv64imafdc -mabi=lp64 -DPREALLOCATE=1 -static -mcmodel=medany
 -std=gnu99 -O2 -fno-common -fno-builtin-printf -fvisibility=hidden -nostdlib -nostartfiles -I$PWD/../../ -T link.ld beqexp2.S -o beqexp2.elf
sai@sai-Inspiron-14-3467:~/iclass-verif/fp_tests$ riscv64-unknown-elf-objdump --disassemble-all --disassemble-zeroes --section=.text --section
=.text.startup --section=.text.init --section=.data beqexp2.elf > beqexp2.disass
sai@sai-Inspiron-14-3467:~/iclass-verif/fp_tests$ spike  --log-commits --log  spike.dump --isa=rv64imafdc +signature=spike_sig beqexp2.elf
sai@sai-Inspiron-14-3467:~/iclass-verif/fp_tests$ spike -d beqexp2.elf
:
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
:
core   0: 0x0000000000001004 (0x02028593) addi    a1, t0, 32
:
core   0: 0x0000000000001008 (0xf1402573) csrr    a0, mhartid
:
core   0: 0x000000000000100c (0x0182b283) ld      t0, 24(t0)
:
core   0: 0x0000000000001010 (0x00028067) jr      t0
:
core   0: >>>>  _start
core   0: 0x0000000080000000 (0x0000a885) c.j     pc + 112
:
core   0: >>>>  reset_vector
core   0: 0x0000000080000070 (0xf1402573) csrr    a0, mhartid
:
core   0: 0x0000000080000074 (0x0000e101) c.bnez  a0, pc + 0
:
core   0: 0x0000000080000076 (0x00000297) auipc   t0, 0x0
:
core   0: 0x000000008000007a (0x01a28293) addi    t0, t0, 26
: reg 0 t0
0x0000000080000090
:
core   0: 0x000000008000007e (0x30529073) csrw    mtvec, t0
:
core   0: 0x0000000080000082 (0x30205073) csrwi   medeleg, 0
:
core   0: 0x0000000080000086 (0x30305073) csrwi   mideleg, 0
:
core   0: 0x000000008000008a (0x30405073) csrwi   mie, 0
```

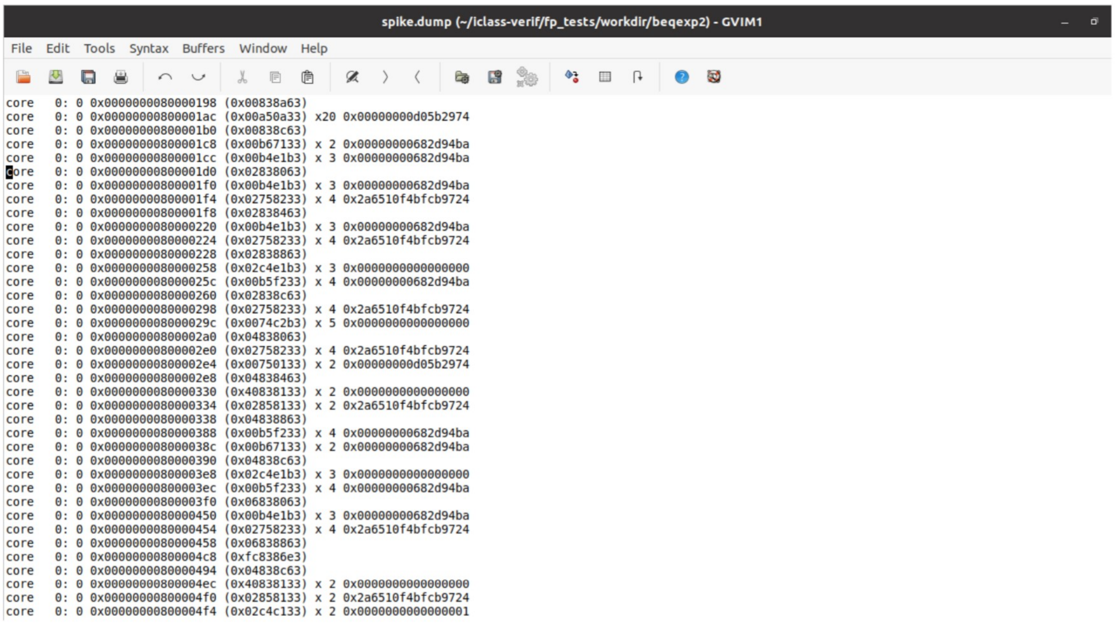
Fig. 3.2: Commands and spike debug mode example

The command to run the test on spike is also shown in the figure above. After the execution of the command it goes to debug mode and it executes one instruction after other when we press ‘enter’. ‘ reg 0 “register name”’ can be used to see the contents of a register. ‘Mem “memory location”’ can be used to see the contents in a particular memory location. ‘until pc 0 “pc number”’ can be used to jump the execution to any program counter. Spike and rtl dumps contains the PC value, the hexadecimal value of the instruction, destination register of the instruction and its value, also the memory location and its contents if any memory instructions are executed. The disassembly, spike and rtl dump files are shown below.


beqexp2.disass (~/iclass-verif/fp_tests/workdir/beqexp2) - GVIM2
File Edit Tools Syntax Buffers Window Help
9,1 0%

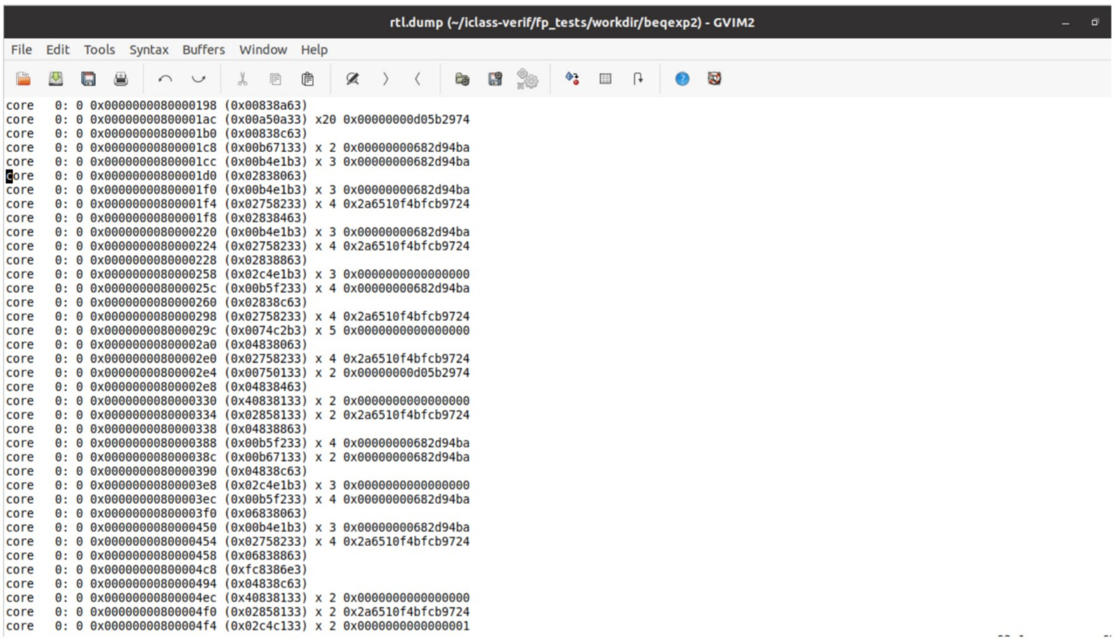
```
Disassembly of section .text.init:

0000000080000000 <_start>:
    80000000:   a885                    j       80000070 <reset_vector>
    80000002:   0001                    nop

0000000080000004 <trap_vector>:
    80000004:   34202f73                csrr    t5,mcause
    80000008:   4fa1                    li      t6,8
    8000000a:   03ff0d63                beq     t5,t6,80000044 <write_tohost>
    8000000e:   4fa5                    li      t6,9
    80000010:   03ff0a63                beq     t5,t6,80000044 <write_tohost>
    80000014:   4fad                    li      t6,11
    80000016:   03ff0763                beq     t5,t6,80000044 <write_tohost>
    8000001a:   342022f3                csrr    t0,mcause
    8000001e:   34102373                csrr    t1,mepc
    80000022:   0311                    addi    t1,t1,4
    80000024:   34131073                csrw    mepc,t1
    80000028:   30200073                mret
    8000002c:   00000f13                li      t5,0
    80000030:   000f0363                beqz    t5,80000036 <trap_vector+0x32>
    80000034:   8f02                    jr      t5
    80000036:   34202f73                csrr    t5,mcause
    8000003a:   000f5363                bgez    t5,80000040 <handle_exception>
    8000003e:   a009                    j       80000040 <handle_exception>

0000000080000040 <handle_exception>:
    80000040:   5391e193                ori     gp,gp,1337

0000000080000044 <write_tohost>:
    80000044:   000f7f17                auipc   t5,0xf7
    80000048:   fa3f2e23                sw      gp,-68(t5) # 800f7000 <tohost>

000000008000004c <shakti_end>:
    8000004c:   0000100f                fence.i
    80000050:   00020fb7                lui     t6,0x20
    80000054:   000f5f17                auipc   t5,0xf5
```

Fig. 3.3: Disassembly file


spike.dump (~/iclass-verif/fp_tests/workdir/beqexp2) - GVIM1
File Edit Tools Syntax Buffers Window Help

```
core   0: 0 0x0000000080000198 (0x00838a63)
core   0: 0 0x00000000800001ac (0x00a50a33) x20 0x00000000d05b2974
core   0: 0 0x00000000800001b0 (0x00838c63)
core   0: 0 0x00000000800001c8 (0x00b67133) x 2 0x00000000682d94ba
core   0: 0 0x00000000800001cc (0x00b4e1b3) x 3 0x00000000682d94ba
core   0: 0 0x00000000800001d0 (0x02838063)
core   0: 0 0x00000000800001f0 (0x00b4e1b3) x 3 0x00000000682d94ba
core   0: 0 0x00000000800001f4 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x00000000800001f8 (0x02838463)
core   0: 0 0x0000000080000220 (0x00b4e1b3) x 3 0x00000000682d94ba
core   0: 0 0x0000000080000224 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x0000000080000228 (0x02838863)
core   0: 0 0x0000000080000258 (0x02c4e1b3) x 3 0x0000000000000000
core   0: 0 0x000000008000025c (0x00b5f233) x 4 0x00000000682d94ba
core   0: 0 0x0000000080000260 (0x02838c63)
core   0: 0 0x0000000080000298 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x000000008000029c (0x0074c2b3) x 5 0x0000000000000000
core   0: 0 0x00000000800002a0 (0x04838063)
core   0: 0 0x00000000800002e0 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x00000000800002e4 (0x00750133) x 2 0x00000000d05b2974
core   0: 0 0x00000000800002e8 (0x04838463)
core   0: 0 0x0000000080000330 (0x40838133) x 2 0x0000000000000000
core   0: 0 0x0000000080000334 (0x02858133) x 2 0x2a6510f4bfcb9724
core   0: 0 0x0000000080000338 (0x04838863)
core   0: 0 0x0000000080000388 (0x00b5f233) x 4 0x00000000682d94ba
core   0: 0 0x000000008000038c (0x00b67133) x 2 0x00000000682d94ba
core   0: 0 0x0000000080000390 (0x04838c63)
core   0: 0 0x00000000800003e8 (0x02c4e1b3) x 3 0x0000000000000000
core   0: 0 0x00000000800003ec (0x00b5f233) x 4 0x00000000682d94ba
core   0: 0 0x00000000800003f0 (0x06838063)
core   0: 0 0x0000000080000450 (0x00b4e1b3) x 3 0x00000000682d94ba
core   0: 0 0x0000000080000454 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x0000000080000458 (0x06838863)
core   0: 0 0x00000000800004c8 (0xfc8386e3)
core   0: 0 0x0000000080000494 (0x04838c63)
core   0: 0 0x00000000800004ec (0x40838133) x 2 0x0000000000000000
core   0: 0 0x00000000800004f0 (0x02858133) x 2 0x2a6510f4bfcb9724
core   0: 0 0x00000000800004f4 (0x02c4c133) x 2 0x0000000000000001
```

Fig. 3.4: Spike dump


rtl.dump (~/iclass-verif/fp_tests/workdir/beqexp2) - GVIM2
File Edit Tools Syntax Buffers Window Help

```
core   0: 0 0x0000000080000198 (0x00838a63)
core   0: 0 0x00000000800001ac (0x00a50a33) x20 0x00000000d05b2974
core   0: 0 0x00000000800001b0 (0x00838c63)
core   0: 0 0x00000000800001c8 (0x00b67133) x 2 0x00000000682d94ba
core   0: 0 0x00000000800001cc (0x00b4e1b3) x 3 0x00000000682d94ba
core   0: 0 0x00000000800001d0 (0x02838063)
core   0: 0 0x00000000800001f0 (0x00b4e1b3) x 3 0x00000000682d94ba
core   0: 0 0x00000000800001f4 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x00000000800001f8 (0x02838463)
core   0: 0 0x0000000080000220 (0x00b4e1b3) x 3 0x00000000682d94ba
core   0: 0 0x0000000080000224 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x0000000080000228 (0x02838863)
core   0: 0 0x0000000080000258 (0x02c4e1b3) x 3 0x0000000000000000
core   0: 0 0x000000008000025c (0x00b5f233) x 4 0x00000000682d94ba
core   0: 0 0x0000000080000260 (0x02838c63)
core   0: 0 0x0000000080000298 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x000000008000029c (0x0074c2b3) x 5 0x0000000000000000
core   0: 0 0x00000000800002a0 (0x04838063)
core   0: 0 0x00000000800002e0 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x00000000800002e4 (0x00750133) x 2 0x00000000d05b2974
core   0: 0 0x00000000800002e8 (0x04838463)
core   0: 0 0x0000000080000330 (0x40838133) x 2 0x0000000000000000
core   0: 0 0x0000000080000334 (0x02858133) x 2 0x2a6510f4bfcb9724
core   0: 0 0x0000000080000338 (0x04838863)
core   0: 0 0x0000000080000388 (0x00b5f233) x 4 0x00000000682d94ba
core   0: 0 0x000000008000038c (0x00b67133) x 2 0x00000000682d94ba
core   0: 0 0x0000000080000390 (0x04838c63)
core   0: 0 0x00000000800003e8 (0x02c4e1b3) x 3 0x0000000000000000
core   0: 0 0x00000000800003ec (0x00b5f233) x 4 0x00000000682d94ba
core   0: 0 0x00000000800003f0 (0x06838063)
core   0: 0 0x0000000080000450 (0x00b4e1b3) x 3 0x00000000682d94ba
core   0: 0 0x0000000080000454 (0x02758233) x 4 0x2a6510f4bfcb9724
core   0: 0 0x0000000080000458 (0x06838863)
core   0: 0 0x00000000800004c8 (0xfc8386e3)
core   0: 0 0x0000000080000494 (0x04838c63)
core   0: 0 0x00000000800004ec (0x40838133) x 2 0x0000000000000000
core   0: 0 0x00000000800004f0 (0x02858133) x 2 0x2a6510f4bfcb9724
core   0: 0 0x00000000800004f4 (0x02c4c133) x 2 0x0000000000000001
```

Fig. 3.5: Rtl dump

Chapter 4

COCOTB

CoCoTb is a coroutine cosimulation based test bench. It is a library in python and it is implemented using the same. Coroutine is a python feature to implement asynchronous interfaces and cosimulation is running python from an RTL simulator. For running the test bench code on the rtl, we need a simulator. CoCoTb supports a wide variety of simulators and for our designs we use verilator as the simulator.

4.1 Why CoCoTb ?

- We know that system verilog with UVM is widely used in industry as standard verification environment which is licensed while cocotb is open source and completely free.

- UVM has a huge library and is difficult to get started with, while python is easy to learn.
- Python has many inbuilt libraries whose features can be used to build a test more comprehensively.
- Python is more popular

4.2 Some features of CoCoTb

As mentioned earlier there are many features in cocotb which are written in python which we can call directly while building a test.

4.2.1 Coroutine

Coroutine is a python feature that runs parallel while running a test. It runs asynchronous with test. It is similar to the always block in verilog which executes as long as the design runs. So, for example we have to design clock using the coroutine decorator.

4.2.2 Yield

Yield keyword is used to make the simulation time to pass. I will explain this feature with example in the next feature.

4.2.3 Timer

Timer is used to pass the simulation time for some amount of time. For example “yield Timer(1, units=’ns’) ” waits for simulation time to pass for one nano second.

4.2.4 RisingEdge

RisingEdge is used to make simulator wait for rising edge of a signal. For example

“yield RisingEdge(dut.CLK)” waits for the next rising edge occurance of ‘CLK’.

4.2.5 FallingEdge

FallingEdge is used to make simulator wait for falling edge of a signal. For example

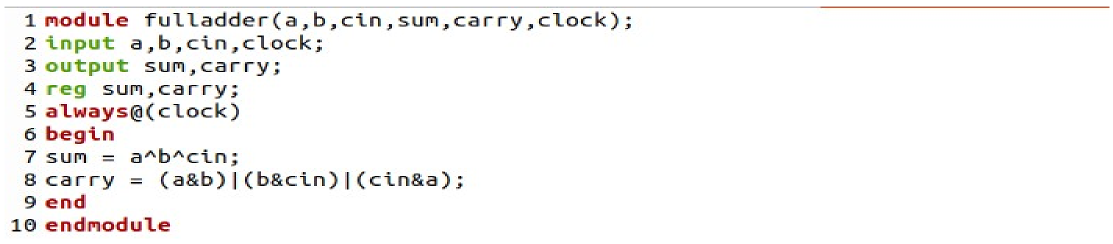
“yield FallingEdge(dut.CLK)” waits for the next falling edge occurance of ‘CLK’.

4.2.6 ClockCycles

ClockCycles is used to make simulator wait for several postive edge or negative edge clock cycles. For example “yield ClockCycles(dut.CLK,10,True)” waits for the signal ‘CLK’ to wait for 10 cycles. If ‘True’ it waits for positive clock cycles, otherwise negative clock cycles.

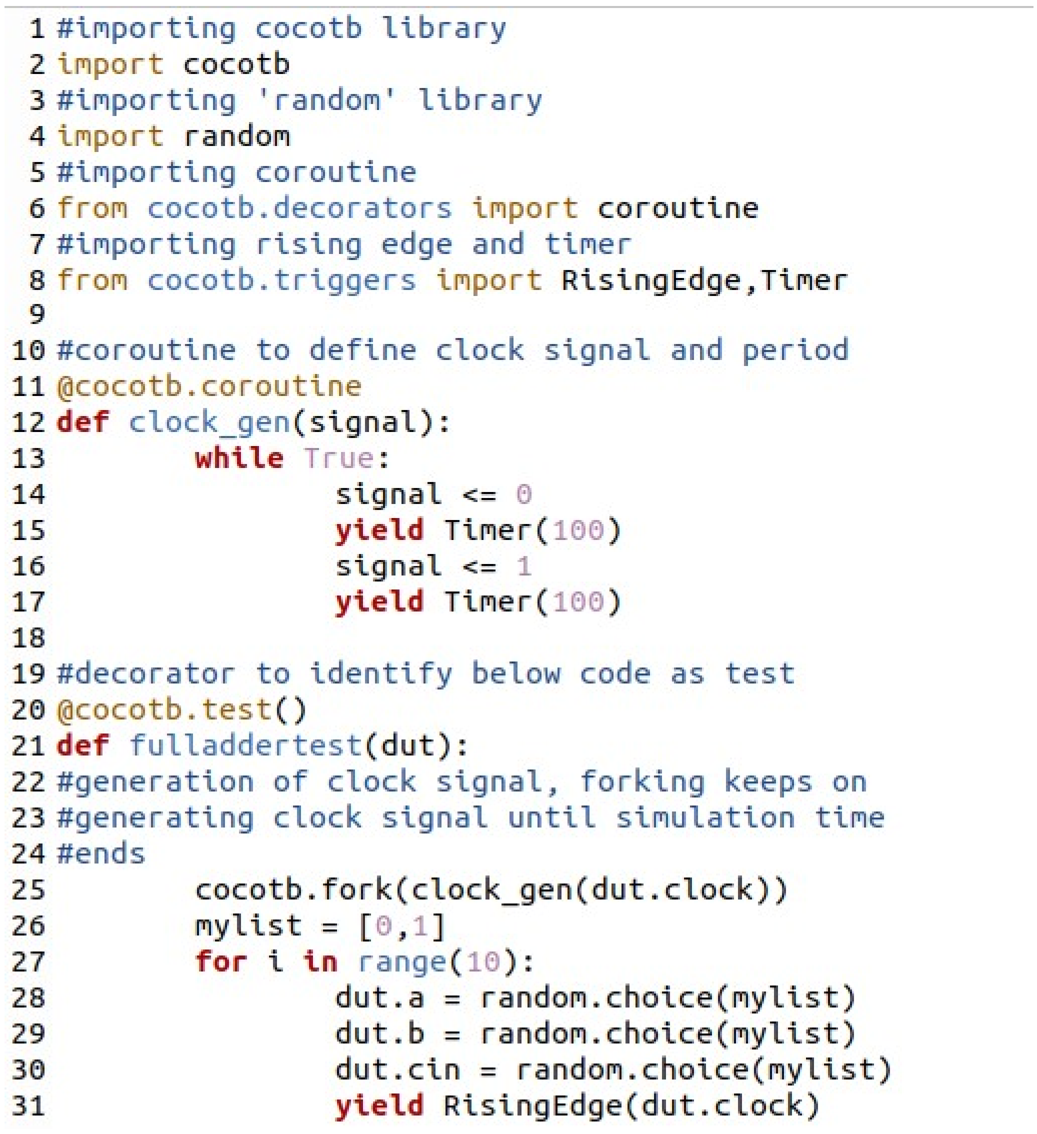
4.3 Simple cocotb test

I will explain how to run a python test using simple full adder example. We need to write a rtl code for full adder and a python test for it. To run this test we need to have a file called ‘Makefile’. We need to include rtl code,top level module in rtl code,python test name etc. To run the test we need to simply execute ‘make’ command in terminal.



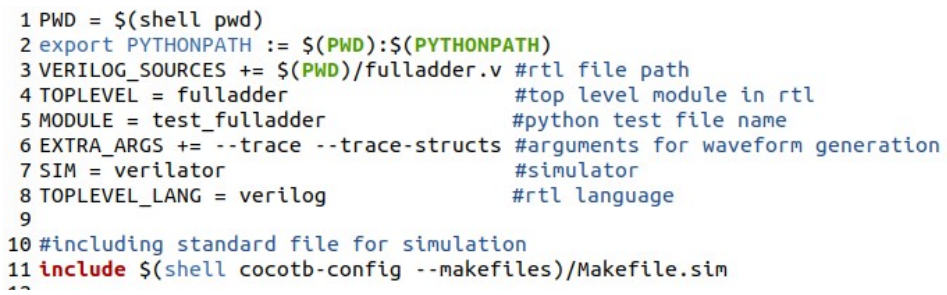
```
module fulladder(a,b,cin,sum,carry,clock);
input a,b,cin,clock;
output sum,carry;
reg sum,carry;
always@(clock)
begin
sum = a^b^cin;
carry = (a&b)|(b&cin)|(cin&a);
end
endmodule
```

Fig. 4.1: verilog code for full adder



```python
#importing cocotb library
import cocotb
#importing 'random' library
import random
#importing coroutine
from cocotb.decorators import coroutine
#importing rising edge and timer
from cocotb.triggers import RisingEdge,Timer

#coroutine to define clock signal and period
@cocotb.coroutine
def clock_gen(signal):
        while True:
                signal <= 0
                yield Timer(100)
                signal <= 1
                yield Timer(100)

#decorator to identify below code as test
@cocotb.test()
def fulladdertest(dut):
#generation of clock signal, forking keeps on
#generating clock signal until simulation time
#ends
        cocotb.fork(clock_gen(dut.clock))
        mylist = [0,1]
        for i in range(10):
                dut.a = random.choice(mylist)
                dut.b = random.choice(mylist)
                dut.cin = random.choice(mylist)
                yield RisingEdge(dut.clock)
```

Fig. 4.2: Python test bench for full adder



```makefile
PWD = $(shell pwd)
export PYTHONPATH := $(PWD):$(PYTHONPATH)
VERILOG_SOURCES += $(PWD)/fulladder.v #rtl file path
TOPLEVEL = fulladder                  #top level module in rtl
MODULE = test_fulladder               #python test file name
EXTRA_ARGS += --trace --trace-structs #arguments for waveform generation
SIM = verilator                       #simulator
TOPLEVEL_LANG = verilog               #rtl language

#including standard file for simulation
include $(shell cocotb-config --makefiles)/Makefile.sim
```

Fig. 4.3: Makefile

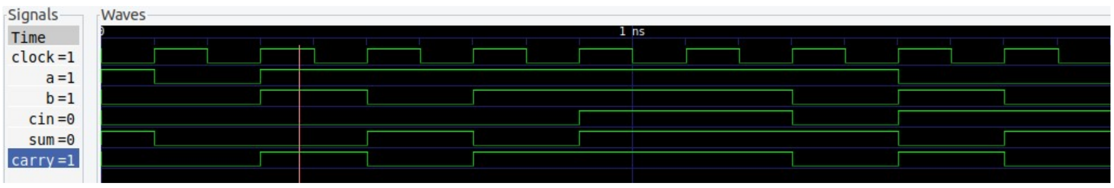


Fig. 4.4: Simulated result

The above test is a simple test to get an idea of how to run a cocotb test and have not used any of high level cocotb components like drivers or monitors. It gives an idea of how to drive values to rtl signals,how to generate clock and how to use triggers like Timer and RisingEdge.

4.4 Test bench architecture

To build a comprehensive test for large blocks of design, we need to use the features of cocotb which makes it easy to build and maintain test. The figure below shows how cocotb test works

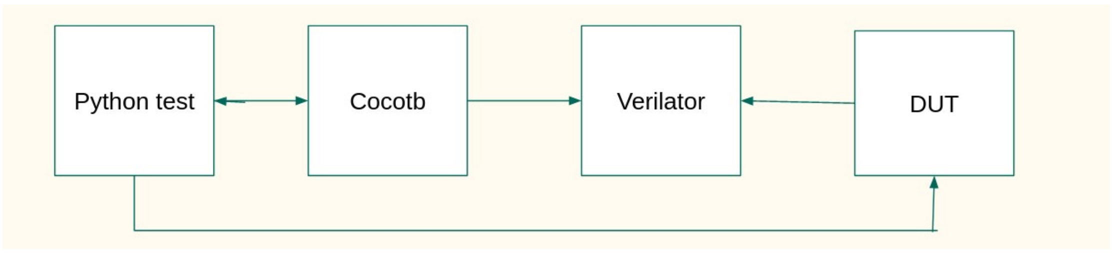


Fig. 4.5: Cocotb test bench framework

DUT is design under test which is the rtl. Verilator is the simulator. We drive inputs to the dut by means of cocotb test and it can the values in dut hierarchy and dut runs on a standard simulator as shown above. Cocotb provides Verilog Procedural Interface(VPI) between rtl simulator and python test

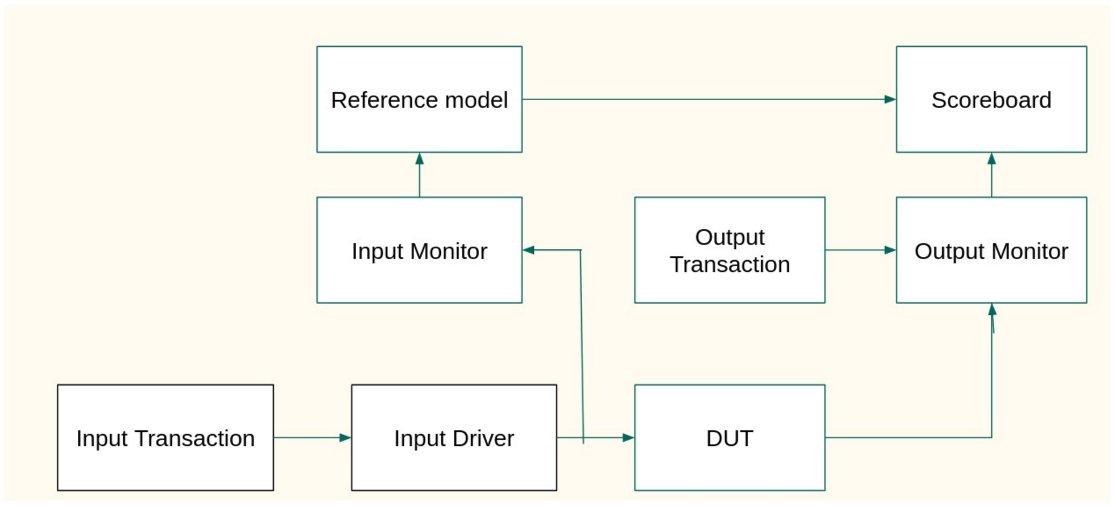


Fig. 4.6: Cocotb test bench architecture

4.4.1 Transaction

Transaction sequences the input values to the driver. We can call it as a sequencer which sequentially sends inputs to driver. So when we want to drive inputs, we have to 'yield' the transaction by means of a driver.

4.4.2 Driver

Driver is used to drive values to the inputs of the DUT. As explained above, sequence of values to be driven will be written in transaction. The parameters of the driver class contains all the interface signals of the design.

4.4.3 Monitor

Monitor is used to keep track of dut signals. Generally dut signals will have corresponding enable signals, we have to provide a sampling condition inside the monitor class such that a particular is to be monitored only when its enable signal goes high. A simple 'if' condition is what we need to write. Monitor has internal method called '_recv' and we need to call this method to receive the monitored signals. And to monitor the signals itself, we need to write a coroutine inside the monitor class which is responsible for, at what points in the simulation to call _recv function and what transaction values to pass to be stored in the monitors receiving queue.

4.4.4 DUT

As mentioned earlier, DUT is rtl specification of the design which may be verilog or vhdl.

4.4.5 Reference model

It is a model which mimics the actual design outputs. In other words, the output signal values of reference model and that of design should be exactly same. This is a good place to explain how cocotb test flow works, first of all we drive input values to the input signals of dut as specified in input transaction. Then the dut generates output signals which is monitored by output monitor. But how do we know that the generated output signals are correct ? This is where the reference model comes into picture. So while driving inputs to the dut, these signals are monitored by input monitor and we drive these inputs parallelly to the reference model. For example, if we consider full adder we have to write code for sum and carry in reference model and make sure that output through reference model is always correct. We get outputs through design as well as reference model and they should match for the design to be bug free.

4.4.6 Scoreboard

Scoreboard compares the outputs from design and reference model and fails the test if both are not same.

Chapter 5
VERIFICATION

5.1 Verification of srt radix-4 divider

Srt radix4 divider performs signed and unsigned division operations for 32 bit and 64-bit operands based on RISCV specification. It performs DIV, DIVU which are signed and unsigned RV32 quotient operations and REM, REMU which are signed and unsigned RV32 remainder operations. Similarly, these operations are named DIVW, DIVUW, REMW, REMUW in RV64. The outputs are received after every 38 clock cycles.

5.1.1 Interface of the design

- Mainly, the interface consists of dividend, divisor, opcode, funct3, enable for input signals, clock, reset as inputs. Result and enable signal for result as output signals. And a flush input signal to flush the division operation.
- Opcode decides whether the operation is 32-bit (RV32) or 64-bit (RV64). Funct3 is used to choose the type of operation whether it is remainder or quotient.

5.1.2 Test plan

Test plan id	Feature	Sub feature	Test plan	Design parameter
div4_1	Inputs	64 bits	Randomize these signals and check the results.Do cross coverage for corner cases	ma_start_divisor,ma_start_dividend
div4_2	Input	4 bits	Check for 32bit and 64bit operation	ma_start_opcode
div4_3	Input	3 bits	Check the correct operation is being executed	ma_start_funct3
div4_4	Input	1 bit	Should be driven high before driving inputs and monitor	EN_ma_start

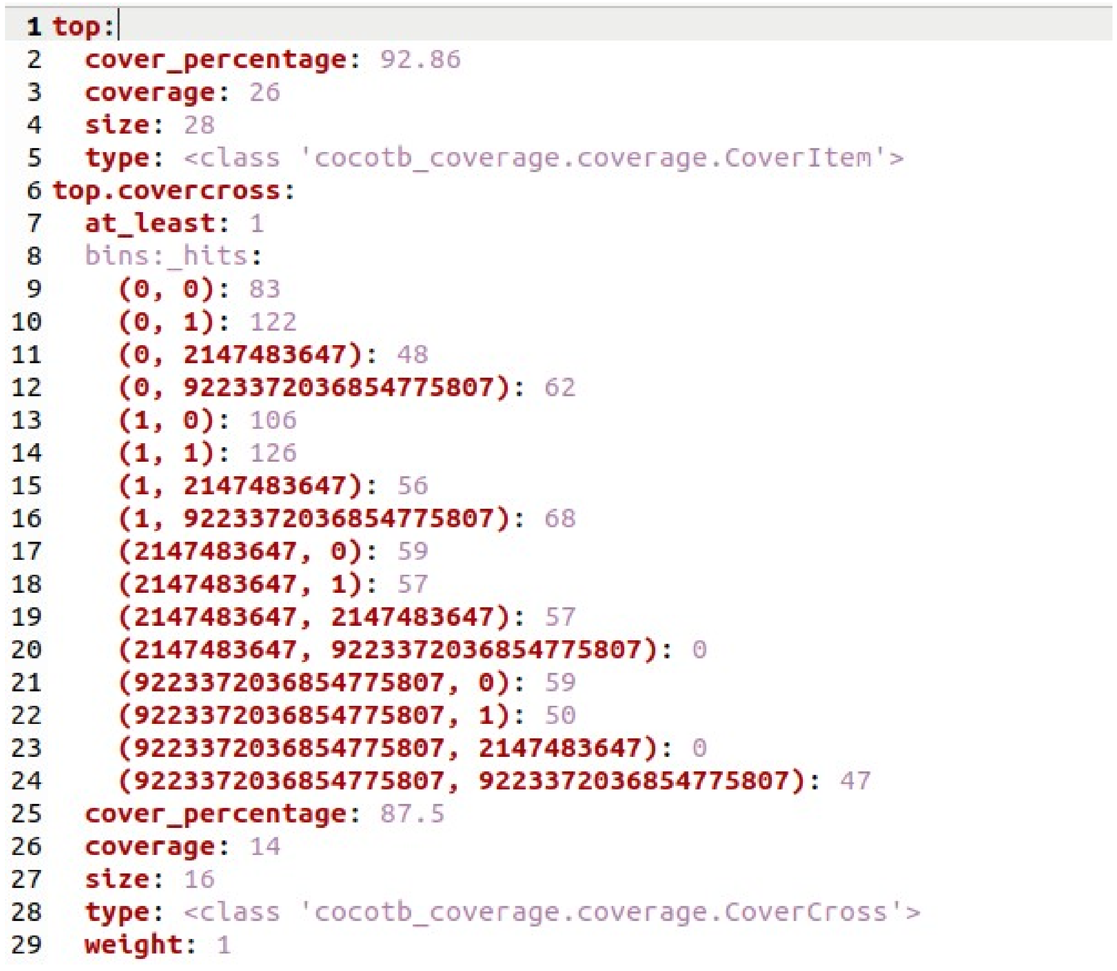
			inputs when it goes high	
div5_5	Model output	65 bits	Use model to generate expected output	expected_output
div5_5	DUT output	65 bits	Verify the generated DUT output and compare with expected_output. Monitor when the valid bit, mav_result[65] goes high.	mav_result

Table 5.1: Test plan of srt radix-4 divider

5.1.3 Verification strategy

- Randomizations have been done for dividend and divisor and ran long running tests and checked for bugs and collected coverage for divisor and dividend.
- Also, exclusive tests are written to verify the corner cases and collected the cross coverage between dividend and divisor.
- The opcode and funct3 are covered in individual tests.
- Dividend and divisor are of 64 bits each and it's impossible to cover all the values of dividend and divisor, instead I have run long running tests of upto 500000 transactions and created bins for the entire range of inputs and if one input value falls in that range of bin, then it said to be covered. So, there is a need to check for corner cases and cover all of them. So, exclusive tests are written for corner cases and collected cross coverage between dividend and divisor.

5.1.4 Functional Coverage report



```
top:
  cover_percentage: 92.86
  coverage: 26
  size: 28
  type: <class 'cocotb_coverage.coverage.CoverItem'>
top.covercross:
  at_least: 1
  bins:_hits:
    (0, 0): 83
    (0, 1): 122
    (0, 2147483647): 48
    (0, 9223372036854775807): 62
    (1, 0): 106
    (1, 1): 126
    (1, 2147483647): 56
    (1, 9223372036854775807): 68
    (2147483647, 0): 59
    (2147483647, 1): 57
    (2147483647, 2147483647): 57
    (2147483647, 9223372036854775807): 0
    (9223372036854775807, 0): 59
    (9223372036854775807, 1): 50
    (9223372036854775807, 2147483647): 0
    (9223372036854775807, 9223372036854775807): 47
  cover_percentage: 87.5
  coverage: 14
  size: 16
  type: <class 'cocotb_coverage.coverage.CoverCross'>
  weight: 1
```

Fig. 5.1: Functional Coverage of srt radix-4 divider

The other two dividers are non restoring divider and srt radix-2 divider. The interface of these two designs is pretty much the same as that of srt radix-4 divider. The output is received after 67 clock cycles for non restoring divider and it is 66 clock cycles for srt radix-2 divider. Same test plan and verification strategy as of srt radix-4 divider is followed for verifying these two dividers.

5.2 Verification of 4 stage pipelined multiplier

- Four stage pipelined multiplier implements 64bit multiplication based on RISCV specifications. It has 4 RV32 operations, MULH (signed*signed), MULHU (unsigned*unsigned), MULHSU (signed*unsigned). For these operations the upper 64 bits are placed in result. MUL operation, where operands can be signed or unsigned places lower 64 bits in the result.

- MULW is an RV64 operation, operands can be signed or unsigned. It places lower 32bits in the result sign extended to 64bits. The first output is received after 4 clock cycles and the next outputs are received continuously at each clock cycle.

5.2.1 Interface of the design

It has a multiplier, multiplicand,word32 that differentiates between RV32 and RV64, funct3 that decides the operation to be performed, enable input signal. It consists of output valid signal and output multiplication result signal.

5.2.2 Test plan

Test plan id	Feature	Subfeature	Testplan	Design parameter
mul_1	Inputs	64 bits	Randomize these signals and check the results. Do cross coverage for corner cases	send_in1,send_in2
mul_2	Input	1 bit	Check for 32 bit or 64 bit operation	send_word32
mul_3	Input	3 bits	Check the correct operation is being executed	send_funct3
mul_4	Enable Input	1 bit	Sample inputs when it goes high.	EN_send
mul_5	Model output	64bits	Use developed reference model to generate expected output	expected_output

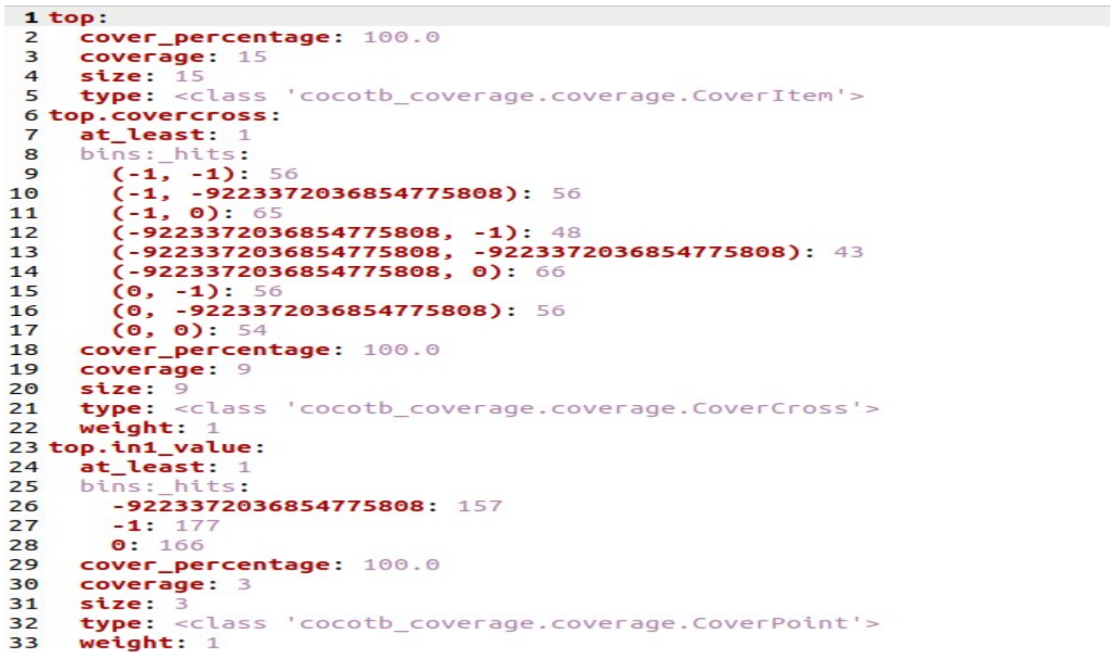
mul_6	Output valid bit	1 bit	Sample DUT output when it goes high	receive_fst
mul_7	DUT output	64 bits	Verify the generated DUT output and compare with expected output using scoreboard	receive_snd

Table 5.2: Test plan of four stage pipelined multiplier

5.2.3 Verification strategy

- Randomizations have been done for multiplier and multiplicand and ran long running tests for each of the specified 5 operations. Coverage was collected for multiplier and multiplicand.
- Exclusive tests were written to verify corner cases in each of these operations and collected cross coverage between multiplier and multiplicand. .
- Funct3 is covered in each individual test for the operations.
- Multiplier and multiplicand are of 64 bits each and it's impossible to cover all the values of multiplier and multiplicand, instead I have run long running tests of upto 500000 transactions and created bins for the entire range of inputs and if one input value falls in that range of bin, then it said to be covered. So, there is a need to check for corner cases and cover all of them. So, exclusive tests are written for corner cases and collected cross coverage between multiplier and multiplicand.

5.2.4 Functional Coverage report

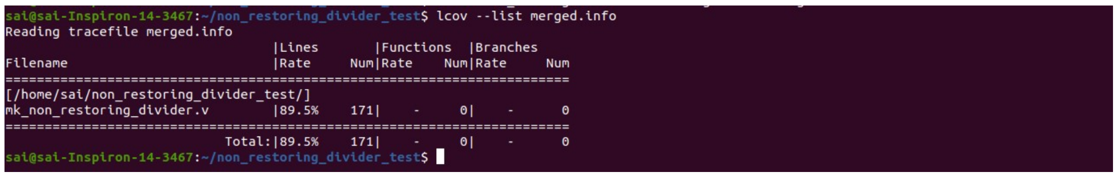


```
top:
  cover_percentage: 100.0
  coverage: 15
  size: 15
  type: <class 'cocotb_coverage.coverage.CoverItem'>
top.covercross:
  at_least: 1
  bins:_hits:
    (-1, -1): 56
    (-1, -9223372036854775808): 56
    (-1, 0): 65
    (-9223372036854775808, -1): 48
    (-9223372036854775808, -9223372036854775808): 43
    (-9223372036854775808, 0): 66
    (0, -1): 56
    (0, -9223372036854775808): 56
    (0, 0): 54
  cover_percentage: 100.0
  coverage: 9
  size: 9
  type: <class 'cocotb_coverage.coverage.CoverCross'>
  weight: 1
top.in1_value:
  at_least: 1
  bins:_hits:
    -9223372036854775808: 157
    -1: 177
    0: 166
  cover_percentage: 100.0
  coverage: 3
  size: 3
  type: <class 'cocotb_coverage.coverage.CoverPoint'>
  weight: 1
```

Fig. 5.2: Functional Coverage of four stage pipelined multiplier

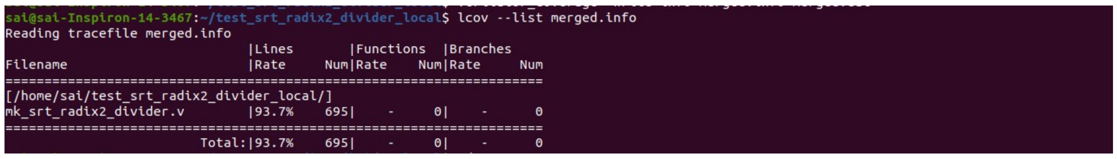
5.3 Code Coverage report

Code coverage is a software testing measure that determines the number of lines of code that are successfully validated throughout a test procedure, which aids in determining how thoroughly a software is tested. Different tests are written to cover Below are the HDL code coverage numbers for the multiplier and divider blocks.



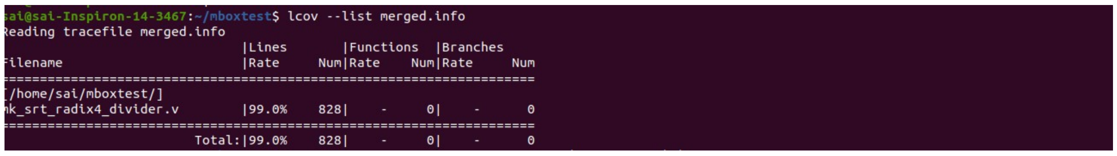
```
sai@sai-Inspiron-14-3467:~/non_restoring_divider_test$ lcov --list merged.info
Reading tracefile merged.info
                                   |Lines        |Functions   |Branches
Filename                           |Rate      Num|Rate     Num|Rate      Num
===========================================================================
[/home/sai/non_restoring_divider_test/]
mk_non_restoring_divider.v         |89.5%    171|    -     0|    -       0
===========================================================================
                             Total:|89.5%    171|    -     0|    -       0
sai@sai-Inspiron-14-3467:~/non_restoring_divider_test$ 
```

Fig. 5.3: Code coverage for non-restoring divider



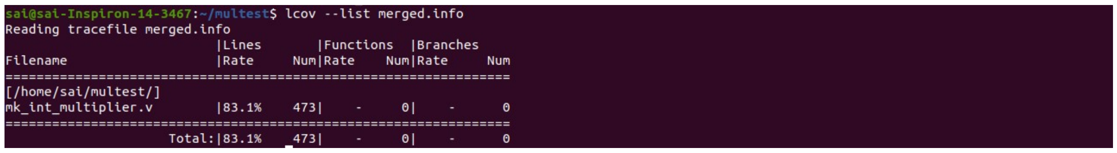
```
sai@sai-Inspiron-14-3467:~/test_srt_radix2_divider_local$ lcov --list merged.info
Reading tracefile merged.info
                                |Lines        |Functions   |Branches
Filename                        |Rate      Num|Rate     Num|Rate      Num
========================================================================
[/home/sai/test_srt_radix2_divider_local/]
mk_srt_radix2_divider.v         |93.7%    695|    -     0|    -       0
========================================================================
                          Total:|93.7%    695|    -     0|    -       0
```

Fig. 5.4: Code coverage for srt radix-2 divider



```
ai@sai-Inspiron-14-3467:~/mboxtest$ lcov --list merged.info
eading tracefile merged.info
                               |Lines       |Functions  |Branches
ilename                        |Rate     Num|Rate    Num|Rate     Num
====================================================================
[/home/sai/mboxtest/]
k_srt_radix4_divider.v         |99.0%    828|    -     0|    -      0
====================================================================
                         Total:|99.0%    828|    -     0|    -      0
```

Fig. 5.5: Code coverage for srt radix-4 divider



```
sai@sai-Inspiron-14-3467:~/multest$ lcov --list merged.info
Reading tracefile merged.info
                           |Lines       |Functions  |Branches
Filename                   |Rate     Num|Rate    Num|Rate     Num
================================================================
[/home/sai/multest/]
mk_int_multiplier.v        |83.1%    473|    -     0|    -      0
================================================================
                     Total:|83.1%    473|    -     0|    -      0
```

Fig. 5.6: Code coverage for four stage pipelined multiplier

5.4 Verification of RISC-V branch instructions

This is a branch instructions verification of i-class core and all the operations are done by loading and storing in registers. In RISC-V a register is a 32 bit or 64 bit memory. For example, we cannot directly multiply two numbers, instead we should load the operands into two registers and store in one of those two registers or any other register. Unlike in mbox verification which is a block verification, there are rtl signals for the operands, this is a core verification and all verify here is whether the registers are taking the correct values, introducing hazards like read after write(RAW), write after read(WAR) etc,. Branch instructions have been verified by taking some scenarios and introducing them in the tests.

5.4.1 Test plan

All the branch instructions are verified by means of adding scenarios to the tests as below

- All possible rs1 registers are used.
- All possible rs2 registers are used.
- Imm values are positive and negative.
- Add back to back branches. One forward branch and one backward branch.
- Add consecutive branches. One forward branch and one backward branch.
- Add exceptions.

5.4.2 Verification strategy

First of all, we need to use all possible 32 registers as rs1 and rs2. It is not possible to use them all in one test since we need some registers to load the values of rs1 and rs2. The input values for rs1 and rs2 which takes part in branch instruction are loaded in the data section of test. But, we need to run the test for not just one case, but many test cases. So, we need a register to store the test case count and the execution of the test stops when its value become zero. Therefore, if we add 10 branches to a test and add 100 test cases, then the test will be running for 1000 branches with different values for rs1 and rs2 in each cycle.

For each branch instruction, six tests have been developed. Each test has six registers as rs1 and rs2, so we need six tests in total to cover all the registers for rs1 and rs2. Further, six registers are needed to store the values of rs1 and rs2 and two registers to work on exceptions. So, nine registers are needed to build the test. As said earlier, six registers take part in branching as rs1 and rs2. Therefore another 15 registers are available to work on adding random instructions which are used as source registers in random instructions between branch instructions. And, no rs1 and rs2 registers have been used as destination registers, since it makes easy to keep track on tests and also reduces the risk of infinite branching. It is optimal way to build the tests.

In each test of a branch instruction, for example if we take one test of bge, it has got 1299 bge instructions. Of which some are back to branches and some are consecutive branches, it includes both forward and backward branches. Consecutive branches has rs1 and rs2 reversed i.e. if one instruction has x1 as rs1 and x2 as rs2, the immediate consecutive branch has x2 as rs1 and x1 as rs2, out of which one is forward branch and the other is backward branch. 150 test cases have been added to the tests.

Chapter 6

VERIFICATION CHALLENGES AND BUGS

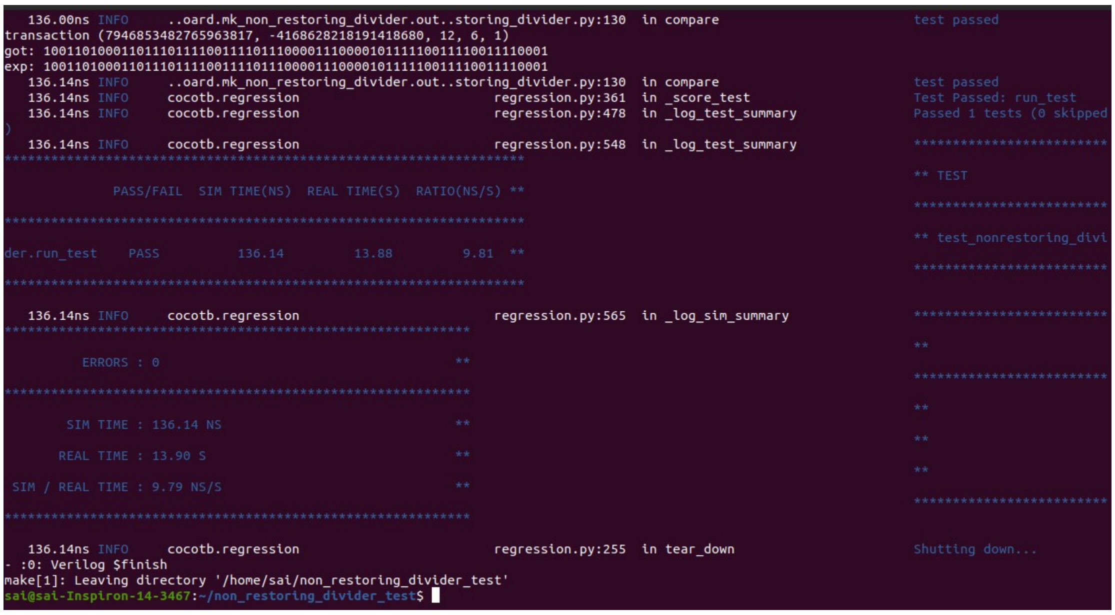
One of the most successful ways to deliver high-quality designs that work is to avoid bugs at the design capture stage. Of course, complexity is your adversary, and design faults are unavoidable, but the best way to avoid costly high-level design/architecture problems is to validate the design architecture before RTL coding.For finding the bugs in the design we may employ several kinds of techniques or practices like building exclusive tests for verifying corner cases, constrained randomization of inputs, exploring design discontinuities etc.

While building the tests for RISC-V branch instructions, since more than 1000 branch instructions have been added, the challenge lies in keeping track of back to back branch instructions. Many back to back and consecutive branches are added and we have to be extra careful while working with registers in back to back branches. Because, if we add different rs1 and rs2, it forms a infinite branch and test keeps on executing. It is such that the size of dump files increases rapidly and it is good to keep a timeout for a test before proceeding to run the entire test. So, in a back to back branches with one forward and one backward branch, both branch conditions should be either satisfied or dissatisfied. It seems simple but one should be careful while developing thousands of branches. In other words, one should be careful while operating with registers in branch instructions. The tests have been run for all the branch instructions and no bugs were found currently.

While building tests for multiplier and divider blocks or any other block level verification, one has to make sure that the reference model should be accurate and should not be biased towards the design. Of all the tests that were run on multiplier and divider blocks, all of them were passed except one case in srt radix-2 divider. When dividend is -2^{63} and dividend is 1, the output from design is coming not as expected while this is passing while running directed test and showing correct value in waveforms and this is a case to look at.

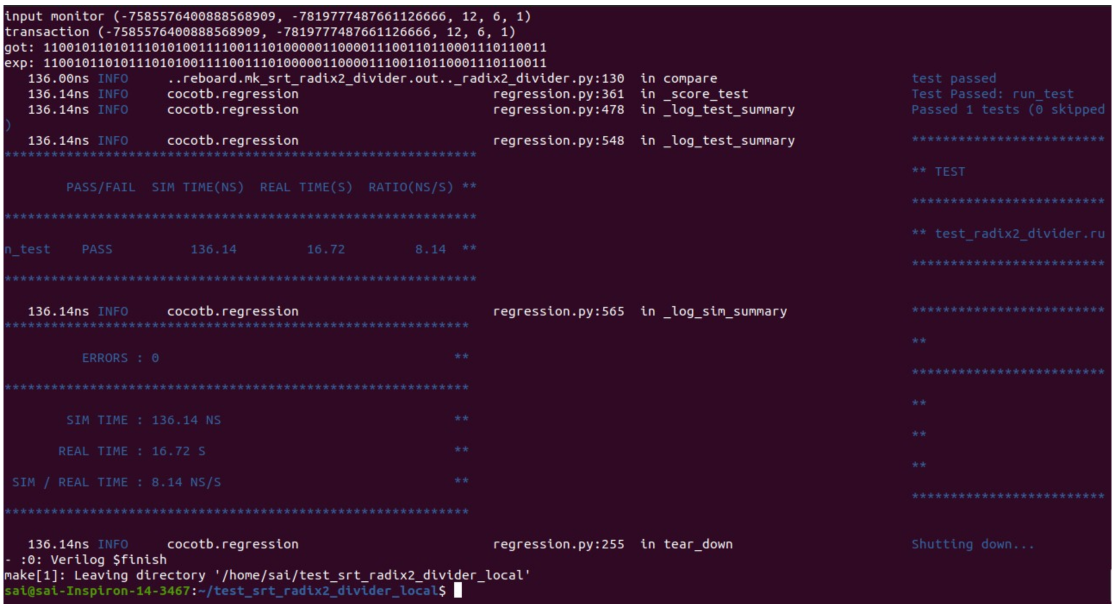
Chapter 7

SIMULATED RESULTS



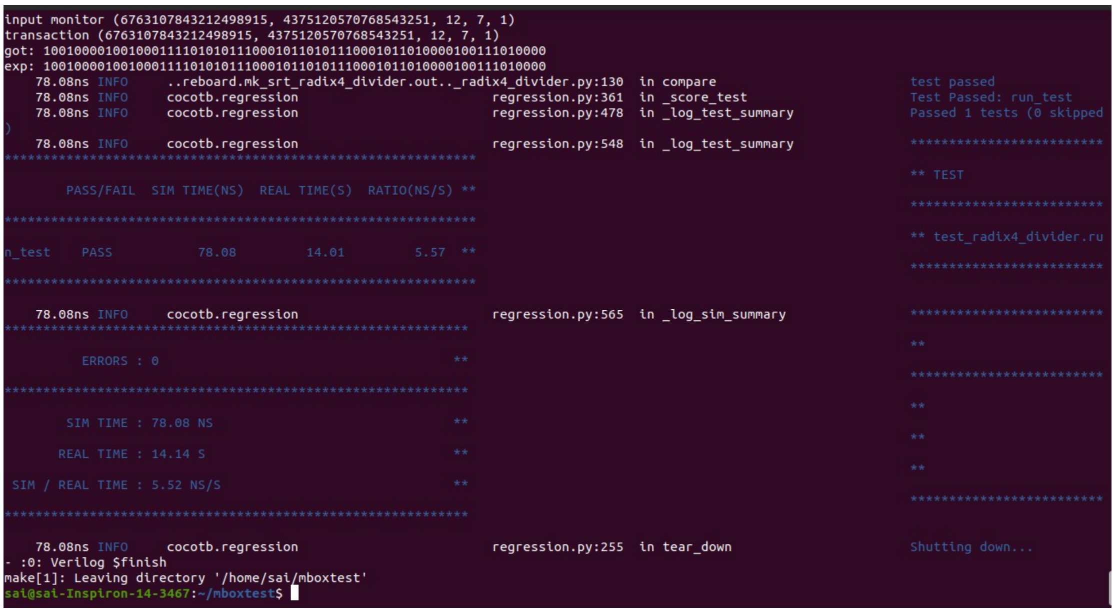
```
   136.00ns INFO     ..oard.mk_non_restoring_divider.out..storing_divider.py:130  in compare                          test passed
transaction (7946853482765963817, -4168628218191418680, 12, 6, 1)
got: 1001101000110111011110011110111000011100001011111001110011110001
exp: 1001101000110111011110011110111000011100001011111001110011110001
   136.14ns INFO     ..oard.mk_non_restoring_divider.out..storing_divider.py:130  in compare                          test passed
   136.14ns INFO     cocotb.regression                             regression.py:361  in _score_test                      Test Passed: run_test
   136.14ns INFO     cocotb.regression                             regression.py:478  in _log_test_summary                Passed 1 tests (0 skipped
)
   136.14ns INFO     cocotb.regression                             regression.py:548  in _log_test_summary                *************************
*******************************************************************
                                                                                                                      ** TEST
              PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
                                                                                                                      *************************
*******************************************************************
                                                                                                                      ** test_nonrestoring_divi
der.run_test    PASS          136.14         13.88         9.81  **
                                                                                                                      *************************
*******************************************************************

   136.14ns INFO     cocotb.regression                             regression.py:565  in _log_sim_summary                 *************************
************************************************************
                                                                                                                      **
         ERRORS : 0                                          **
                                                                                                                      *************************
************************************************************
                                                                                                                      **
        SIM TIME : 136.14 NS                                 **
                                                                                                                      **
       REAL TIME : 13.90 S                                   **
                                                                                                                      **
 SIM / REAL TIME : 9.79 NS/S                                 **
                                                                                                                      *************************
************************************************************

   136.14ns INFO     cocotb.regression                             regression.py:255  in tear_down                        Shutting down...
- :0: Verilog $finish
make[1]: Leaving directory '/home/sai/non_restoring_divider_test'
sai@sai-Inspiron-14-3467:~/non_restoring_divider_test$ 
```

Fig. 7.1: Simulated result of non-restoring divider



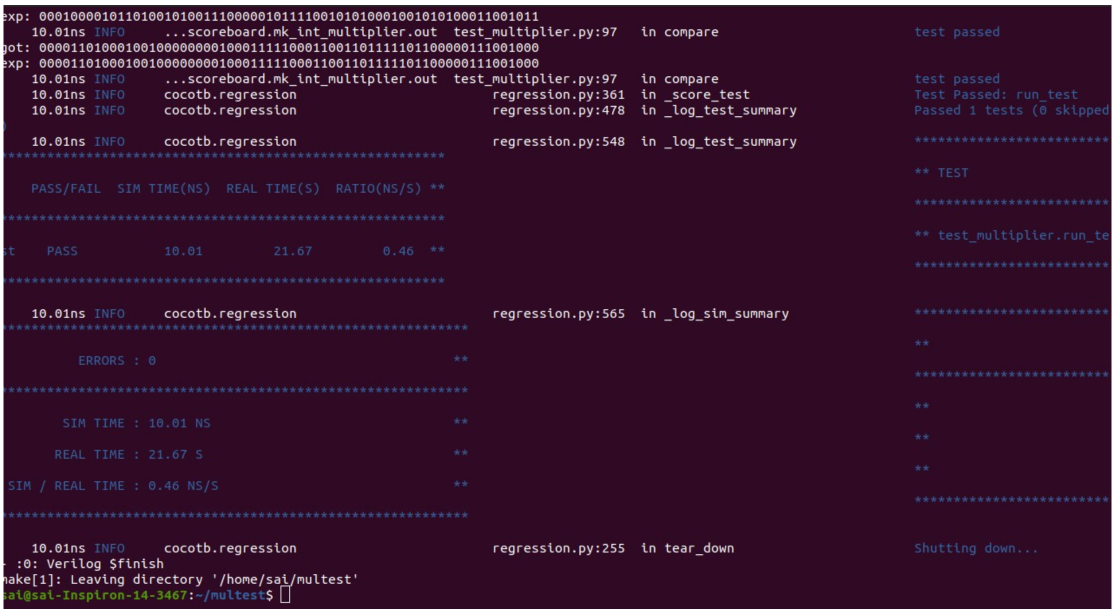
```
input monitor (-7585576400888568909, -7819777487661126666, 12, 6, 1)
transaction (-7585576400888568909, -7819777487661126666, 12, 6, 1)
got: 11001011010111010100111100111010000011000011100110110001110110011
exp: 11001011010111010100111100111010000011000011100110110001110110011
   136.00ns INFO     ..reboard.mk_srt_radix2_divider.out.._radix2_divider.py:130  in compare                          test passed
   136.14ns INFO     cocotb.regression                             regression.py:361  in _score_test                      Test Passed: run_test
   136.14ns INFO     cocotb.regression                             regression.py:478  in _log_test_summary                Passed 1 tests (0 skipped
)
   136.14ns INFO     cocotb.regression                             regression.py:548  in _log_test_summary                *************************
**************************************************************
                                                                                                                      ** TEST
        PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
                                                                                                                      *************************
**************************************************************
                                                                                                                      ** test_radix2_divider.ru
n_test    PASS          136.14         16.72         8.14  **
                                                                                                                      *************************
**************************************************************

   136.14ns INFO     cocotb.regression                             regression.py:565  in _log_sim_summary                 *************************
************************************************************
                                                                                                                      **
         ERRORS : 0                                          **
                                                                                                                      *************************
************************************************************
                                                                                                                      **
        SIM TIME : 136.14 NS                                 **
                                                                                                                      **
       REAL TIME : 16.72 S                                   **
                                                                                                                      **
 SIM / REAL TIME : 8.14 NS/S                                 **
                                                                                                                      *************************
************************************************************

   136.14ns INFO     cocotb.regression                             regression.py:255  in tear_down                        Shutting down...
- :0: Verilog $finish
make[1]: Leaving directory '/home/sai/test_srt_radix2_divider_local'
sai@sai-Inspiron-14-3467:~/test_srt_radix2_divider_local$ 
```

Fig. 7.2: Simulated result of srt radix-2 divider



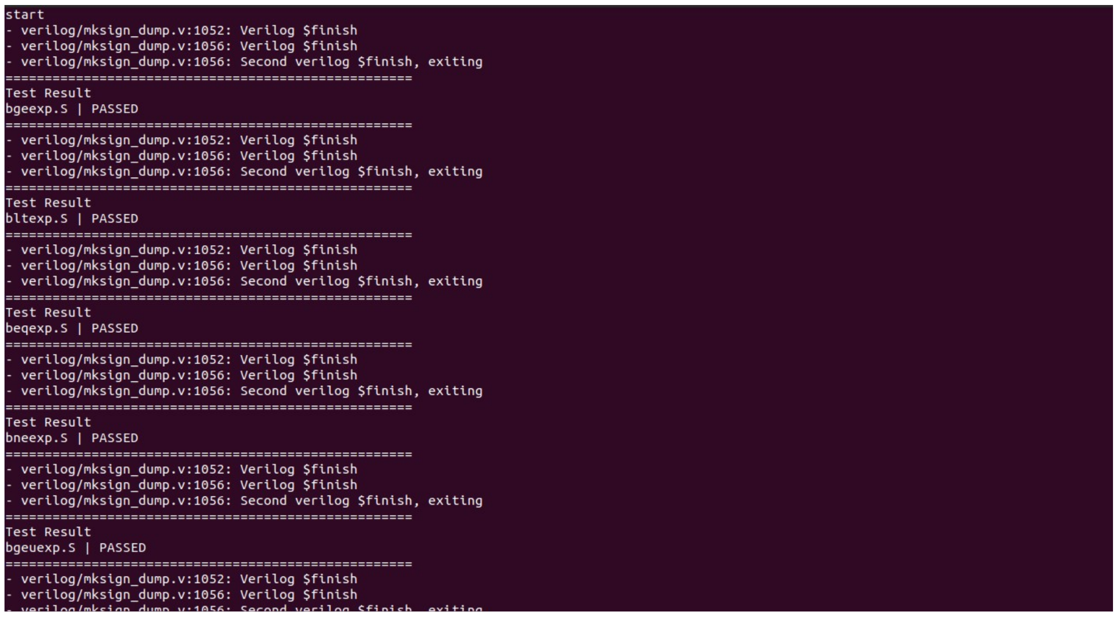
```
input monitor (6763107843212498915, 4375120570768543251, 12, 7, 1)
transaction (6763107843212498915, 4375120570768543251, 12, 7, 1)
got: 10010000100100011110101011100010110101110001011010000100111010000
exp: 10010000100100011110101011100010110101110001011010000100111010000
    78.08ns INFO     ..reboard.mk_srt_radix4_divider.out.._radix4_divider.py:130  in compare                 test passed
    78.08ns INFO     cocotb.regression                           regression.py:361  in _score_test             Test Passed: run_test
    78.08ns INFO     cocotb.regression                           regression.py:478  in _log_test_summary       Passed 1 tests (0 skipped
)
    78.08ns INFO     cocotb.regression                           regression.py:548  in _log_test_summary       *************************
**************************************************************
                                                                                                              ** TEST
        PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
                                                                                                              *************************
**************************************************************
                                                                                                              ** test_radix4_divider.ru
n_test    PASS           78.08         14.01         5.57  **
                                                                                                              *************************
**************************************************************

    78.08ns INFO     cocotb.regression                           regression.py:565  in _log_sim_summary        *************************
*************************************************************
                                                                                                              **
          ERRORS : 0                                        **
                                                                                                              *************************
*************************************************************
                                                                                                              **
        SIM TIME : 78.08 NS                                 **
                                                                                                              **
       REAL TIME : 14.14 S                                  **
                                                                                                              **
 SIM / REAL TIME : 5.52 NS/S                                **
                                                                                                              *************************
*************************************************************

    78.08ns INFO     cocotb.regression                           regression.py:255  in tear_down               Shutting down...
- :0: Verilog $finish
make[1]: Leaving directory '/home/sai/mboxtest'
sai@sai-Inspiron-14-3467:~/mboxtest$
```

Fig. 7.3: Simulated result of srt radix-4 divider



```
exp: 0001000010110100101001110000010111100101010001001010100011001011
    10.01ns INFO     ...scoreboard.mk_int_multiplier.out  test_multiplier.py:97   in compare                test passed
got: 0000110100010010000000001000111110001100110111110110000011100100
exp: 0000110100010010000000001000111110001100110111110110000011100100
    10.01ns INFO     ...scoreboard.mk_int_multiplier.out  test_multiplier.py:97   in compare                test passed
    10.01ns INFO     cocotb.regression                         regression.py:361  in _score_test            Test Passed: run_test
    10.01ns INFO     cocotb.regression                         regression.py:478  in _log_test_summary      Passed 1 tests (0 skipped
)
    10.01ns INFO     cocotb.regression                         regression.py:548  in _log_test_summary      *************************
**********************************************************
                                                                                                            ** TEST
    PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
                                                                                                            *************************
**********************************************************
                                                                                                            ** test_multiplier.run_te
st    PASS           10.01         21.67         0.46  **
                                                                                                            *************************
**********************************************************

    10.01ns INFO     cocotb.regression                         regression.py:565  in _log_sim_summary       *************************
*************************************************************
                                                                                                            **
          ERRORS : 0                                        **
                                                                                                            *************************
*************************************************************
                                                                                                            **
        SIM TIME : 10.01 NS                                 **
                                                                                                            **
       REAL TIME : 21.67 S                                  **
                                                                                                            **
 SIM / REAL TIME : 0.46 NS/S                                **
                                                                                                            *************************
*************************************************************

    10.01ns INFO     cocotb.regression                         regression.py:255  in tear_down              Shutting down...
- :0: Verilog $finish
make[1]: Leaving directory '/home/sai/multest'
sai@sai-Inspiron-14-3467:~/multest$
```

Fig. 7.4: Simulated result of four stage pipelined multiplier



```
start
- verilog/mksign_dump.v:1052: Verilog $finish
- verilog/mksign_dump.v:1056: Verilog $finish
- verilog/mksign_dump.v:1056: Second verilog $finish, exiting
====================================================
Test Result
bgeexp.S | PASSED
====================================================
- verilog/mksign_dump.v:1052: Verilog $finish
- verilog/mksign_dump.v:1056: Verilog $finish
- verilog/mksign_dump.v:1056: Second verilog $finish, exiting
====================================================
Test Result
bltexp.S | PASSED
====================================================
- verilog/mksign_dump.v:1052: Verilog $finish
- verilog/mksign_dump.v:1056: Verilog $finish
- verilog/mksign_dump.v:1056: Second verilog $finish, exiting
====================================================
Test Result
beqexp.S | PASSED
====================================================
- verilog/mksign_dump.v:1052: Verilog $finish
- verilog/mksign_dump.v:1056: Verilog $finish
- verilog/mksign_dump.v:1056: Second verilog $finish, exiting
====================================================
Test Result
bneexp.S | PASSED
====================================================
- verilog/mksign_dump.v:1052: Verilog $finish
- verilog/mksign_dump.v:1056: Verilog $finish
- verilog/mksign_dump.v:1056: Second verilog $finish, exiting
====================================================
Test Result
bgeuexp.S | PASSED
====================================================
- verilog/mksign_dump.v:1052: Verilog $finish
- verilog/mksign_dump.v:1056: Verilog $finish
- verilog/mksign_dump.v:1056: Second verilog $finish, exiting
```

Fig. 7.5: Results of RISC-V branch instructions

Chapter 8

CONCLUSION

Functional verification has been done for multiplier, divider blocks and branch instructions of i-class core. Cocotb and python has been used for functional verification of multiplier and divider blocks which is not an industry standard, as of now. But it is a powerful alternative for UVM and system verilog without compromising on quality of verification. Reference models have been created for these blocks to compare the output from models and DUT output. Many randomizations has been added to input signals to ensure good coverage. Spike, a RISC-V ISA simulator has been used to simulate the assembly tests of RISC-V. The spike dump from spike and rtl dump from design have been compared to know whether the test was passed or not.

8.1 Future Work

For RISC-V branch tests, many number of branch instructions has been added, many of the instruction has repetitive rs1 and rs2 registers. In each test, 6 registers were added as rs1 and rs2 i.e. 5 tests to cover all the registers. But in each test, we can make rs1 and rs2 more random and decrease the repetitions by taking care of back to back and consecutive branches. More and more test cases can de added to make the tests very long running. Also a test for jalr has to be added. While in multiplier and divider blocks verification we can add more randomizations to the input signals.

REFERENCES

1. Cocotb documentation https://docs.cocotb.org/en/stable/
2. Cocotb coverage https://cocotb-coverage.readthedocs.io/en/stable/
3. RISC-V tests https://github.com/riscv-software-src/riscv-tests
4. Shakti RISC-V asm manual. https://shakti.org.in/docs/RISC-V-asm-manual.pdf