EE446 RISC V Project

Introduction:

RISC-V is an innovative instruction-set architecture (ISA) that was initially developed to support research and education in computer architecture at UC Berkeley. Its design aspirations have since expanded, aiming to become a standard, free, and open architecture for industry implementations. The architecture includes a modular structure with a base integer ISA and optional extensions supporting 32-bit and 64-bit address spaces. RISC-V is designed to facilitate efficient implementations and is fully virtualizable, simplifying hypervisor development.

Objective: In this project, a part of RISC-V ISA will be implemented. In this project, the Unprivileged RV32I Base Integer Instruction Set will be implemented. Additionally, one new instruction will be added as an extension. FENCE, ECALL, EBREAK, and HINT instructions will not be implemented, as they are irrelevant. The list of instructions to be implemented is given in Table 1.

Arithmetic instructions:	ADD[I], SUB
Logic instructions:	AND[I], OR[I], XOR[I]
Shift instructions:	SLL[I], SRL[I], SRA[I]
Set if less than:	SLT[I][U]
Conditional branch:	BEQ, BNE, BLT[U], BGE[U]
Unconditional jump:	JAL, JALR (Return-address stack push/pop functionality will not be implemented)
Load:	LW, LH[U], LB[U]
Store:	SW, SH, SB
Others:	LUI, AUIPC
Extra instruction:	XORID

Table 1: List of Instructions

An extra instruction is also to be implemented; the extra instruction is XORID. It will take the xor of rs1 with an embedded constant and write the result to rd. It has the following format:

$$XORID\ rd,rs1\ rd \leftarrow rs1 \oplus (studentId1 \oplus studentId2)$$

Instruction encoding details are given below:
• opcode [6:0] = 0001011
• I type instruction, but immediate value will not be used
• funct3[2:0] =100
In order to achieve this task the solution approach has been divided into three part namely the Datapath, Controller and Testbench. We'll be approaching each part in detail while also explaining the relevant actions that were taken to achieve the required set of instruction executions while modifying the pre-existing data path given in the project file.

Syed Muhammad Farrukh Aijaz-2417152 Mubeen Ahmad Fayyaz-2415842

Datapath Design

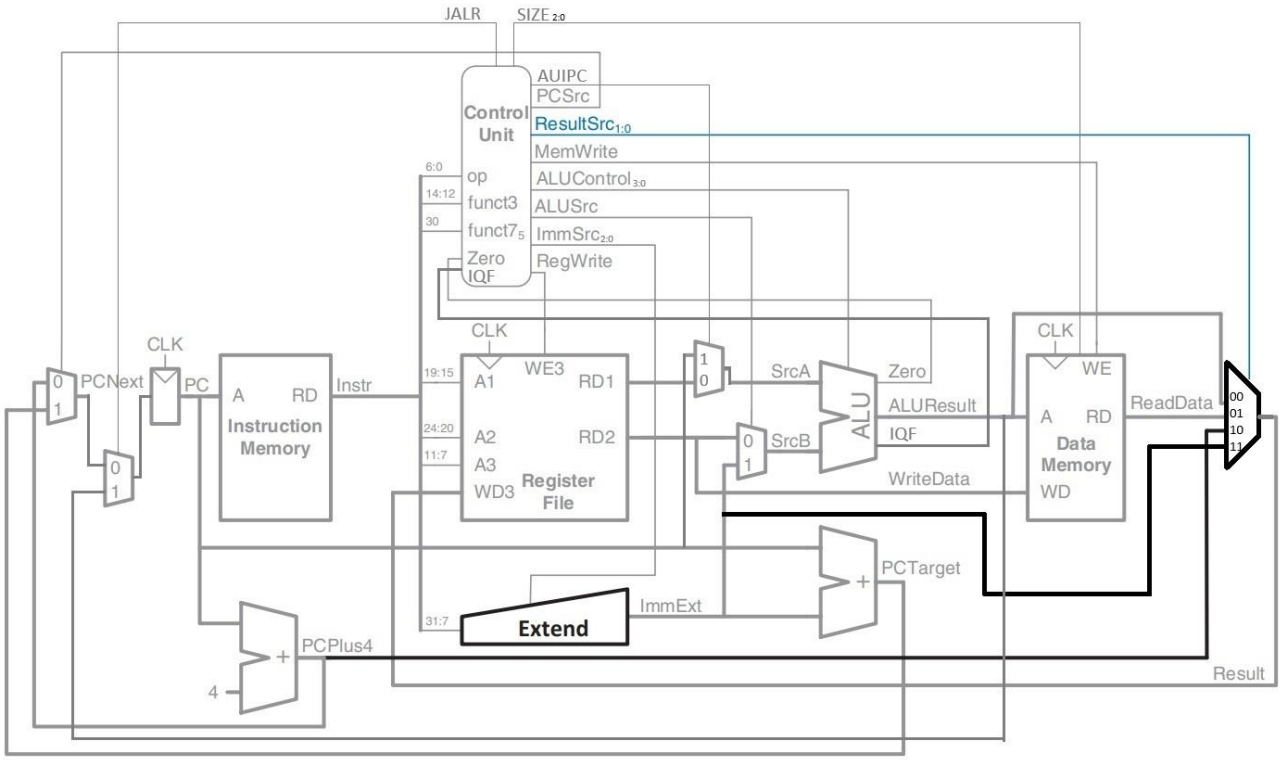

Fig 1.1 Modified Datapath to occupy ISRBUJ-Type Instructions

Now we'll dive into the modifications that were required for each type of Instruction.

I/S-Type Instructions: The previous data path was easily able to handle all I type instructions except for the JALR instruction and all S-type instructions, however the loading and storing of halfwords or bytes being unsigned or signed was an issue, in order to tackle these issues especially for the halfword and byte storing/loading we had changed the Data Memory code block which now includes a 3-bit size control signal, this control signal governs the type of load/store command so that it modifies the storage accordingly. The relevant control signals for the commands are given below:

Command	Description	Size [2:0]
lw	Load word	000
lh	Load half word (least significant 2 bytes)	001
lhu	Load unsigned half word (least significant 2 bytes)	010
lb	Load byte (least significant)	011
lhu	Load unsigned byte (least significant)	100
sb	Store byte (least significant)	000
sh	Store half word (least significant 2 bytes)	001
sw	Store word	011

Table 1.1 New Control Signals for Size

The controller asserts these size control signals, and the commands are carried out like a normal load/store command.

JALR:

EE446 RISC V Project

The JALR instruction needed modifications in the data path, the relevant command for the jalr is given as

$$\text{jalr rd, rs1, imm} \rightarrow \text{PC} = \text{rs1} + \text{SignExt(imm)}, \text{rd} = \text{PC} + 4$$

The Datapath already was able to add the contents of rs1 and ImmExt, by setting the ALUSrc as 1 and rs1 being directed through SrcA into the ALU, therefore the ALUResult already contained the addition of these two values after asserting the control signal for the add operation in the ALU. Then the result was directly fetched and passed on the Multiplexer just before the program counter register with a select named JALR which is generated by the controller. If the JALR control signal is 1 the ALUResult value is loaded into the PC at the next clock cycle. However, we also have to consider the current PCPlus4 to be saved into the destination register, thus to achieve that we also asserted the control for ResultSrc to be 10 so that the destination register (WD3 line) has the PC+4 value and with write enable 1 we would save the incremented PC value in to the destination register. Other control signals are not modified and set to 0.

Instruction	Description	JALR	Size	AUIPC	PCSrc	ResultSrc	MemWrite	ALUControl	ALUSrc	ImmSrc	RegWrite
jalr (I)	Jump and link register	1	000	0	0	10	0	0000	1	000	1

R/B-Type: All R-Type instructions are carried out correctly with the ALU controls being asserted depending on the type of the instruction, all logical shifts, arithmetic shifts and comparison has been move inside the ALU with relevant control signals that are given below, the Z flag is the zero flag which is asserted when the output is a 0, a new flag was introduced called the Inequality flag named IQF for B-type instructions, which checks whether the inequality is true or not and then alters the PCSrc for the branching instructions.

```
ADD=4'b0000, // OUT = DATA_A + DATA_B
SUB=4'b0001, // OUT = DATA_A - DATA_B , Z is set to 1 if answer is 0
ORR=4'b0010, // OUT = DATA_A | DATA_B
AND=4'b0011, // OUT = DATA_A & DATA_B
XOR=4'b0100, // OUT = DATA_A ^ DATA_B
LSL=4'b0101, // OUT = DATA_A << DATA_B [24:20]
LSR=4'b0110, // OUT = DATA_A >> DATA_B [24:20]
ASR=4'b0111, // OUT = DATA_A >>> DATA_B [24:20]
SLT=4'b1000, // OUT = (DATA_A < DATA_B) ? 1 : 0 , IQF is set to 1 if inequality is true else 0
SLTU=4'b1001, // OUT = (DATA_A unsigned < DATA_B unsigned) ? 1 : 0
//IQF is set to 1 if inequality is true else 0
GEQ=4'b1010, // OUT = (DATA_A >= DATA_B) ? 1 : 0
//IQF is set to 1 if inequality is true else 0
GEQU=4'b1011, // OUT = (DATA_A unsigned >= DATA_B unsigned) ? 1 : 0
IQF is set to 1 if inequality is true else 0
XORID=4'b1100; // OUT = DATA_B ^ student id1 ^ student id2
```

J-Type: The modified Datapath is able to handle the jal instruction as the PCTarget already computes the addition of the ImmExt and PC values and loads it at the multiplexer which then can be activated by setting the PCSrc to 1 to change the PC in the next clock cycle, just in order to save the PC+4 value in the destinated register we just set the ResultSrc to 10 so that the destination register (WD3 line) has the PC+4 value and with write enable 1 we would save the incremented PC value in to the destination register. Other control signals are not modified and set to 0.

Instruction	Description	JALR	Size	AUIPC	PCSrc	ResultSrc	MemWrite	ALUControl	ALUSrc	ImmSrc	RegWrite
jal (J)	Jump to address and save link	0	000	0	1	10	0	0000	1	011	1

Syed Muhammad Farrukh Aijaz-2417152 Mubeen Ahmad Fayyaz-2415842

U-Type: For these type of instructions the ImmSrc control signal was increased by 1 bit making it a 3 bit signal in order to accommodate the extensions of bits for a specific type of format for the instructions lui and auipc, both of these instructions required upper 20 bits for the immediate value and the lower 12 bits to be set to 0 there fore we have a new control signal whereas the control signals for the old instructions were bit extended by a leading 0. The extender was also reconfigured to adhere to the changes.

ImmSrc [2:0]	Type of Instruction
000	I
001	S
010	B
011	J
111	U
XXX	R

Table 1.2 New Control signals for ImmSrc

The new extender now can perform a new operation for the U type instructions with control signal 111 as follows:

$$3'b111: Extended\_data = \{DATA[24:5], 12'b0\};$$

Upper 20 bits of the data of the incoming 24-bit data are used and then 12 bits of 0's are added to the end by concatenation. After implementing the new extended data for the control signal 111, ImmExt wire contained the output but lui and auipc required different operations which were conducted as follows:

- **lui :** The ImmExt output was directly fed to the 4 to 1 Multiplexer which now had ImmExt output being linked to the select value 11, this would ensure that when ResultSrc is 11 for lui instruction the ImmExt is loaded into the result line which is connected to WD3 thus the result can be directly stored into Rd in the next clock cycle.
- **auipc:** For this instruction we had to create a new control signal named AUIPC with a 2 to 1 multiplexer as shown in the figure for the datapath, this multiplexer had the current pc as the input when AUIPC is set to 1 otherwise it takes the data input from the rs1 source register, by setting the ALUSrc to 1 we can now add the values of the PC and ImmExt in the ALU using control signal 0000, and then setting the ResultSrc to be 00 we can forward this result to the WD3 line in order to be stored in the destination register.

The synthesized RTL view of the Datapath is given below:

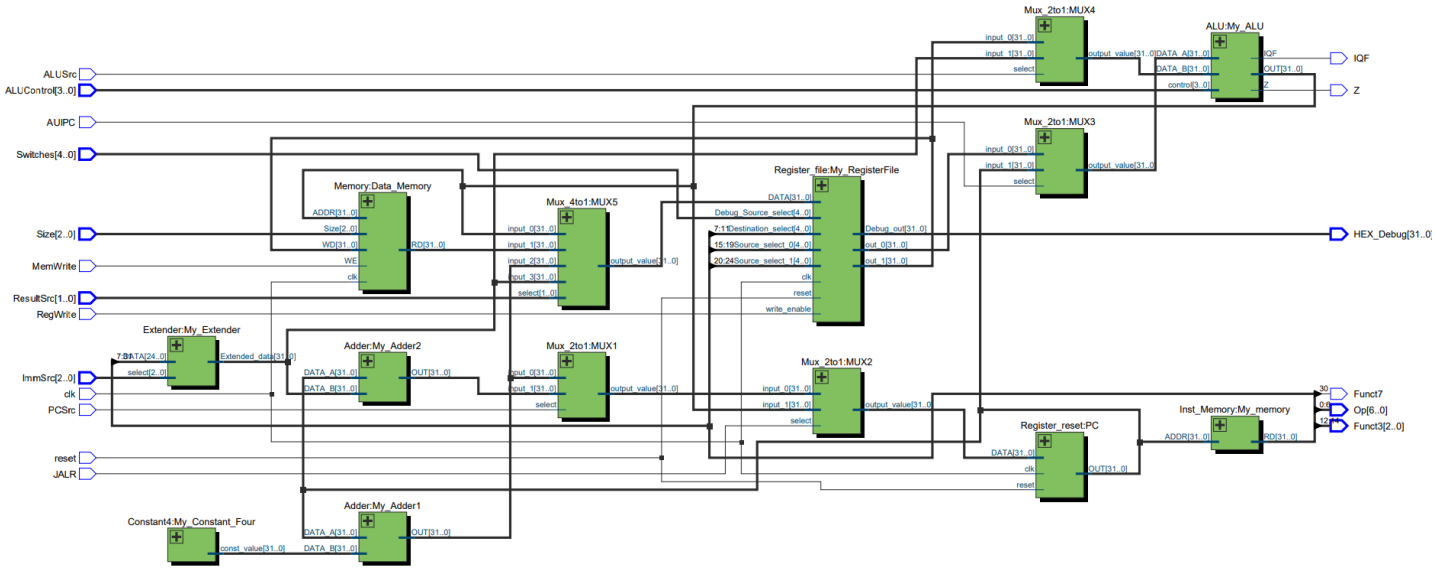

Fig 1.2 RTL View of Datapath

Controller: For the controller we tried to write all the possible combinations for the commands in order to help us in debugging, we asserted all the command-and-control signals as explained in the additional instructions above, Control signals were like JALR, AUIPC, Size [2:0] and an Inequality flag was also added to the controller which can be observed below. The explanations for these control signals have been given in detail in the Datapath section.

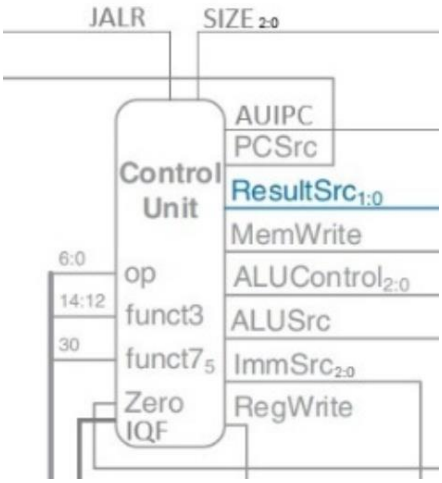

Fig 1.3 Controller Diagram

The RTL view changes upon the display of selection of the control signals mainly due to not being completely displayed in one pdf file.

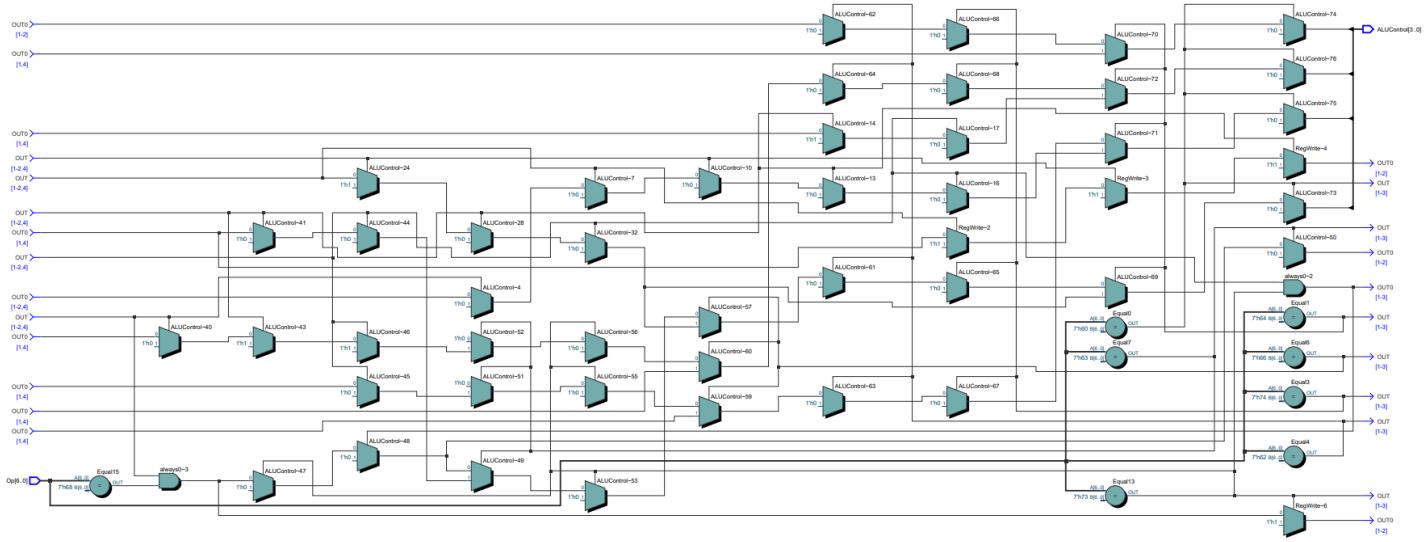

Fig 1.3 Controller RTL View

Each instruction is checked with it's relevant op code by comparing the op codes and funct3 codes we are able to distinguish the command and especially the funct7 bit is used to distinguish between the cases in which both the op code and the funct3 are the same like in addition and subtraction and as well as shift right logical and shift right arithmetic, these cases have been separated with the help of just if-else-if statements in the controller.

Test Bench:

We have modified the single cycle test bench with some of our own classes in helper library as well these include a memory capable of writing and reading a byte, halfword, and word. Instruction class has been created to extract relevant parameters like rs1,rs2 etc based on RISCV ISA. Logging functions were also changed to log appropriate values and register values were extended to 32 where 0 register is always asserted against 0. The major change was in performance_model which is summarized below.

1. **Instruction Parsing**:

- The performance model first reads instructions from the provided instruction list.

- It parses each instruction, extracting relevant fields such as opcode, destination register (rd), source registers (rs1, rs2), and immediate values.
- This parsing allows the model to understand the type and parameters of each instruction, necessary for subsequent execution.

2. **Instruction Execution**:

- After parsing an instruction, the performance model simulates its execution.
- It identifies the type of instruction (R-type, I-type, S-type, B-type, U-type, or J-type) based on the opcode.
- Depending on the instruction type, the model performs the appropriate operation, which may include arithmetic/logical operations, memory accesses, branching, etc.
- For arithmetic/logical operations, it performs the specified operation (e.g., addition, bitwise AND) using the provided source registers and immediate values.
- For memory access instructions, it reads from or writes to memory based on the specified memory address and type. Sine memory is modified, we can easily store and read bytes, half words and words just by specifying these as a string.
- For branching instructions, it updates the program counter (PC) based on the result of the comparison between source register values and immediate values.
- After execution, the model updates the register file and program counter accordingly.

3. **Comparing Results**:

- After executing each instruction, the performance model compares its results with the expected behavior.
- It logs the state of the processor, including the PC, register file contents, and memory contents.
- It then compares these logged values with the expected values from the DUT (Design Under Test), which represents the hardware implementation of the processor.
- Any discrepancies between the performance model and the DUT indicate potential errors in the hardware implementation.

4. **Logging and Debugging**:

- Throughout the simulation, the performance model logs various details such as instruction fields, PC, register values, and memory contents.
- These logs help in debugging and understanding the behaviour of both the performance model and the DUT.
- If an error occurs, these logs can provide valuable insights into what went wrong and where.

5. **Iterative Simulation**:

- The performance model simulates the execution of instructions iteratively, advancing through each clock cycle.
- It continues executing instructions until it encounters a termination condition, such as reaching the end of the instruction list or encountering a specific termination instruction.

These RISCV instructions were used in a instruction.hex file to test the workings of testbench and computer.

```
93 02 A1 00
93 02 F1 00
93 02 41 01
93 B3 92 01
93 22 23 00
93 D2 32 00
93 D2 52 40
93 E2 E3 01
93 E2 33 02
```

B3 82 A2 00
B3 82 A2 40
B3 B2 A2 00
B3 A2 A2 00
B3 C2 A2 00
B3 92 A2 00
B3 D2 A2 00
B3 D2 A2 40
B3 E2 A2 00
B3 F2 A2 00
23 20 21 00
23 12 85 00
23 02 C7 00
93 02 A1 00
A3 A9 59 00
83 A3 39 01
00 00 00 00

And a second set was also used:

93 02 A1 00
E7 87 62 00
EF 00 C0 00
63 08 E4 01
A3 A9 59 00
83 A3 39 01
93 02 A1 00
A3 A9 59 00
83 A3 39 01
B7 FA DE 8C
B7 1E 00 00
0B C5 F2 FF
00 00 00 00

Among these instructions we have these instructions which adheres to the special instructions for the U type and J type which are given above and explained below:

At the end of both of these tests, we obtained pass status for the test.

- JALR Instruction: jalr x15, x5, 6 Hex Code: 00 62 87 E7
- XORID Instruction: xorid r10, r5 Hex Code: FF F2 C5 0B
- BEQ Instruction: bew x8, x30, 16 Hex Code: 63 08 E4 01
- LUI Instruction: lui x21, 0x8CDEF Hex Code: 8C DE FA B7
- AUIPC Instruction: auipd x29,0x1 Hex Code: 00 00 1E B7
- JAL Instruction: jal x1, 0x000C Hex Code: 00 C0 00 EF

Other instructions given in the instructions hex file are arithmetic instruction with and without immediate and branch instructions which are effectively carried out.

```
350000.00ns INFO     cocotb.RISCV_Computer            Size:0x0
350000.00ns INFO     cocotb.RISCV_Computer            ALUControl:0x0
350000.00ns INFO     cocotb.RISCV_Computer            ImmSrc:0x0
350000.00ns INFO     cocotb.RISCV_Computer            ResultSrc:0x1
360000.00ns DEBUG    riscv                            ************ Performance Model / DUT Data  **************
360000.00ns DEBUG    riscv                            PC:148   PC:148
360000.00ns DEBUG    riscv                            mem: 0
360000.00ns DEBUG    riscv                            Register0: 0     0
360000.00ns DEBUG    riscv                            Register1: 0     0
360000.00ns DEBUG    riscv                            Register2: 0     0
360000.00ns DEBUG    riscv                            Register3: 0     0
360000.00ns DEBUG    riscv                            Register4: 0     0
360000.00ns DEBUG    riscv                            Register5: 10    10
360000.00ns DEBUG    riscv                            Register6: 0     0
360000.00ns DEBUG    riscv                            Register7: 10    10
360000.00ns DEBUG    riscv                            Register8: 0     0
360000.00ns DEBUG    riscv                            Register9: 0     0
360000.00ns DEBUG    riscv                            Register10: 16104        16104
360000.00ns DEBUG    riscv                            Register11: 0    0
360000.00ns DEBUG    riscv                            Register12: 0    0
360000.00ns DEBUG    riscv                            Register13: 0    0
360000.00ns DEBUG    riscv                            Register14: 0    0
360000.00ns DEBUG    riscv                            Register15: 8    8
360000.00ns DEBUG    riscv                            Register16: 0    0
360000.00ns DEBUG    riscv                            Register17: 0    0
360000.00ns DEBUG    riscv                            Register18: 0    0
360000.00ns DEBUG    riscv                            Register19: 0    0
360000.00ns DEBUG    riscv                            Register20: 0    0
360000.00ns DEBUG    riscv                            Register21: 2363420672   2363420672
360000.00ns DEBUG    riscv                            Register22: 0    0
360000.00ns DEBUG    riscv                            Register23: 0    0
360000.00ns DEBUG    riscv                            Register24: 0    0
360000.00ns DEBUG    riscv                            Register25: 0    0
360000.00ns DEBUG    riscv                            Register26: 0    0
360000.00ns DEBUG    riscv                            Register27: 0    0
360000.00ns DEBUG    riscv                            Register28: 0    0
360000.00ns DEBUG    riscv                            Register29: 4096         4096
360000.00ns DEBUG    riscv                            Register30: 0    0
360000.00ns DEBUG    riscv                            Register31: 0    0
361000.00ns INFO     cocotb.regression                Single_cycle_test ←[32mpassed←[49m←[39m
361000.00ns INFO     cocotb.regression                **************************************************************************************
                                                      ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                      **************************************************************************************
                                                      ** riscv.Single_cycle_test       ←[32m PASS ←[49m←[39m     361000.00           0.37     966332.66  **
                                                      **************************************************************************************
                                                      ** TESTS=1 PASS=1 FAIL=0 SKIP=0             361000.00           0.46     788400.54  **
                                                      **************************************************************************************

make[1]: Leaving directory '/c/Users/mubee/Downloads/Single_Cycle_Test/Test'
```

Fig 1.4 Test Bench results for all the instructions