Test Plan

R type Instruction

Test case 1: Basic functionality of ADD

This test case verifies the basic functionality of the ADD instruction in a RISC-V processor by performing an addition operation using immediate and register values. The expected result is that register x7 correctly stores the sum of x5 (0x10) and x6 (0x5), resulting in x7 = 0x15.

ADDI x5, x0, 0x10 # x5 = 0x10

ADDI x6, x0, 0x5 # x6 = 0x5

ADD x7, x6, x5 # x7 = x6 + x5 (0x10 + 0x5)

Test case 2: Basic functionality of SUB

This test case verifies the basic functionality of the SUB instruction by performing a subtraction operation between two registers. The expected result is that register x7 correctly stores the result of x5 (0x10) minus x6 (0x5), resulting in x7 = 0xB (0x10 - 0x5 = 0xB).

ADDI x5, x0, 0x10 # x5 = 0x10

ADDI x6, x0, 0x5 # x6 = 0x5

SUB x7, x6, x5 # x7 = x6 - x5 (0x10 – 0x5)

Test case 3: Basic functionality of SLL

This test case verifies the functionality of the SLL (Shift Left Logical) instruction by shifting the value in x6 left by the lower 5 bits of x5. The expected result is that register x7 stores the value of x6 (0x5) shifted left by 0x10 & 0x1F (16) positions, resulting in x7 = 0x50000 (0x5 << 16).

ADDI x5, x0, 0x10 # x5 = 0x10

ADDI x6, x0, 0x5 # x6 = 0x5

SLL x7, x6, x5 # x7 = x6 << (x5 & 0x1F) (Logical shift left)

Test case 4: Differentiate SLTU and SLT

This test case verifies the difference between signed and unsigned comparisons using the SLT (Set Less Than) and SLTU (Set Less Than Unsigned) instructions. The expected result is that SLT sets x7 to 1 since -1 is less than 1 in signed comparison, while SLTU sets x8 to 0 because 0xFFFFFFFF is greater than 0x00000001 in unsigned comparison.

Initialize registers with values that show the difference between SLT and SLTU

ADDI x5, x0, -1 # x5 = 0xFFFFFFFF (-1 in signed, large positive in unsigned)

ADDI x6, x0, 1 # x6 = 0x00000001 (1 in both signed and unsigned)

SLT x7, x5, x6 # Signed comparison: (-1 < 1) -> x7 = 1

SLTU x8, x5, x6 # Unsigned comparison: (0xFFFFFFFF < 0x00000001) -> x8 = 0

Test case 5: Basic functionality of XOR

This test case verifies the basic functionality of the XOR instruction by performing a bitwise exclusive OR operation between two registers. The expected result is that register x7 stores the result of x6 (0x5) XOR x5 (0x10), resulting in x7 = 0x15 (0x5 ^ 0x10 = 0x15).

ADDI x5, x0, 0x10 # x5 = 0x10

ADDI x6, x0, 0x5 # x6 = 0x5

XOR x7, x6, x5 # x7 = x6 ^ x5 (Bitwise XOR)

Test case 6: Differentiating between SRA and SRL

This test case verifies the difference between SRL (Logical Shift Right) and SRA (Arithmetic Shift Right) by shifting a negative value (0xFFFFFFFF). The expected result is that SRL (x7) performs a logical shift, filling with zeros, resulting in 0x0FFFFFFF, while SRA (x8) maintains the sign bit, resulting in 0xFFFFFFFF, demonstrating how arithmetic shift preserves sign extension.

ADDI x5, x0, 0xFFFFFFFF # x5 = 0xFFFFFFFF (-1 in two's complement)

ADDI x6, x0, 4 # x6 = 4 (shift amount)

Perform Logical Shift Right (SRL)

SRL x7, x5, x6 # x7 = x5 >> x6 (Logical shift, fills with 0s)

Perform Arithmetic Shift Right (SRA)

SRA x8, x5, x6 # x8 = x5 >> x6 (Arithmetic shift, fills with sign bit)

Expected Output

- SRL x7, x5, x6: **Logical Shift Right**
    - 0xFFFFFFFF >> 4 results in 0x0FFFFFFF (fills with 0s).
- SRA x8, x5, x6: **Arithmetic Shift Right**

- 0xFFFFFFFF >> 4 results in 0xFFFFFFFF (fills with 1s, maintaining the sign bit).

Test case 7: Basic functionality of OR

This test case verifies the basic functionality of the OR instruction by performing a bitwise OR operation between two registers. The expected result is that register x7 stores the result of x6 (0x5) OR x5 (0x10), resulting in x7 = 0x15 (0x5 | 0x10 = 0x15).

ADDI x5, x0, 0x10 # x5 = 0x10

ADDI x6, x0, 0x5 # x6 = 0x5

OR x7, x6, x5 # x7 = x6 | x5 (Bitwise OR)

Test case 8: Basic functionality of AND

This test case verifies the basic functionality of the AND instruction by performing a bitwise AND operation between two registers. The expected result is that register x7 stores the result of x6 (0x5) AND x5 (0x10), resulting in x7 = 0x0 (0x5 & 0x10 = 0x0) since there are no common set bits.

ADDI x5, x0, 0x10 # x5 = 0x10

ADDI x6, x0, 0x5 # x6 = 0x5

AND x7, x6, x5 # x7 = x6 & x5 (Bitwise AND)

Test Case 9: Overflow condition for ADD

This test case verifies integer overflow behaviour when adding the maximum positive 32-bit signed integer (0x7FFFFFFF) with 1. The expected result is that x7 stores 0x80000000, which represents -2147483648 in signed two's complement. Although this results in an overflow, RISC-V does not raise an exception for signed integer overflow in standard ADD operations.
ADDI x5, x0, 0x7FFF_FFFF # x5 = 2147483647 (Max positive value)

ADDI x6, x0, 0x1 # x6 = 1

ADD x7, x5, x6 # x7 = x5 + x6 (Expected: 0x80000000, which is -2147483648 in signed 2's complement)

Expected result : This results in an overflow, but no exception is raised.

Test Case 10: Overflow condition for SUB

This test case verifies the behaviour of the SUB instruction when subtracting from the minimum signed 32-bit value. The expected result is that x7 stores 0x7FFFFFFF, which is +2147483647 in signed two's complement, after performing the subtraction of 1 from the minimum signed value (0x80000000). This case checks how the processor handles the wrap-around behaviour when performing operations near the limits of signed integers.

ADDI x5, x0, 0x80000000 # x5 = 0x80000000 (-2147483648, Min signed 32-bit value)

ADDI x6, x0, 0x00000001 # x6 = 0x00000001 (1)

SUB x7, x5, x6 # x7 = x5 - x6 (Expected: 0x7FFFFFFF, which is +2147483647 in signed 2's complement)

Test case 11: attempt to write a value to register 0

This test case verifies the behaviour of attempting to write a value to register x0, which is the zero register in RISC-V and always holds the value 0. The instruction ADD x0, x5, x6 attempts to add the values in x5 (0x10) and x6 (0x5) and store the result in x0. However, since x0 is hardwired to 0, the operation will have no effect, and x0 will remain 0, regardless of the operation. This ensures that writes to register x0 are ignored, as expected.

ADDI x5, x0, 0x10 # x5 = 0x10

ADDI x6, x0, 0x5 # x6 = 0x5

ADD x0, x5, x6

S type Instruction

To test the correct functionality of s type instruction , I categorised my test cases into following categories.

- **Basic Functionality** to ensure correct basic functionality
  - ➔ **SB x5, 0(x10)** store a byte from x5 at address x10

    Expected output : mem[x10] = x5[7:0]

  - ➔ **SH x6, 0(x11)** Store a halfword from x6

    ex output : mem[x11] = x6[15:0]

  - ➔ **SH x7, 0(x12)** Store a word from x7

    ex output : mem[x12] = x7[31:0]
- **Store with Positive and Negative Offsets**
  - ➔ **SW x8, 4(x9)** Store word at x9 + 4

    expected output: mem[x9 + 4] = x8[31:0]

  - ➔ **SH x8, -4(x9)** Store halfword at x9 - 4

    expected output : mem[x9 - 4] = x8[15:0]

- **Memory Alignment & Misalignment**
  - ➔ **SW x11, 0(x12)** Store word at aligned address

expected output = mem[x12] = x11[31:0]

➔ **SW x14, 3(x12)** Store word at misaligned address

This Should trigger exception

- **Memory Boundary Testing**
  - ➔ **SW x16, 0xFFF(x17)** Store at max memory range this Should succeed if memory exists
  - ➔ **SW x16, 0x1004(x17)** Store outside memory this Should trigger exception

- **Overwriting Memory**

Using the same location which was written previously to check this scenario

➔ SW x18, 0(x19) should Store word mem[x19] = x18[31:0]
➔ SW x20, 0(x19) This time it should Overwrite word

expected output : mem[x19] = x20[31:0]

B type Instruction

Testcase 1: Functionality of BEQ

The BEQ (Branch if Equal) instruction is a conditional branch instruction in RISC-V. It compares two registers and, if they hold the same value, the program jumps to the specified branch target (label). If they are not equal, execution continues with the next instruction in sequence.

```
ADDI x5, x0, 0x10          # Load immediate 0x10  into register x5
ADDI x6, x0, 0x10          # Load immediate 0x10  into register x6
BEQ  x5, x6, branch        #  If x5 == x6, jump to label branch
ADDI x7, x0, 1             # If branch is not taken, x7 is set to 1 (this should be skipped)
branch: ADDI x7, x0, 2          # If branch is taken, x7 is set to 2
```

Testcase 2: Functionality of BNE

The BNE (Branch not Equal) instruction is a conditional branch instruction in RISC-V. It compares two registers and, if they do not hold the same value, the program jumps to the specified branch target (label). If they are equal, execution continues with the next instruction in sequence.

```
ADDI x5, x0, 10              # Load 10 into x5
ADDI x6, x0, 20              # Load 20 into x6
BNE  x5, x6, branch          # If x5 != x6, branch to branch
ADDI x7, x0, 1               # If branch is NOT taken, x7 = 1
branch: ADDI x7, x0, 2       # If branch is taken, x7 = 2
```

Testcase 3: Functionality of BLT

The BLT (Branch if Less Than) instruction compares two registers and, if the first register holds a value less than the second register (signed values), the program jumps to the specified branch target (label). If the condition is false, execution continues with the next instruction in sequence.

```
ADDI x5, x0, 5          # Load 5 into x5 (x5 = 5)
ADDI x6, x0, 10         # Load 10 into x6 (x6 = 10)
BLT  x5, x6, branch     # If x5 < x6, branch to branch
ADDI x7, x0, 1          # If branch is NOT taken, x7 = 1 (this should be skipped)
branch: ADDI x7, x0, 2  # If branch is taken, x7 = 2
```

Testcase 4: Functionality of BGE

The BGE (Branch if Greater Than or Equal) instruction compares two registers and, if the first register holds a value greater than or equal to the second register (signed values), the program jumps to the specified branch target (label). If the condition is false, execution continues with the next instruction in sequence.

```
ADDI x5, x0, 10          # x5 = 10
ADDI x6, x0, 10          # x6 = 10 (equal)
BGE x5, x6, branch       # Should branch (x5 >= x6)
ADDI x7, x0, 1           # Should be skipped if branch works
branch: ADDI x7, x0, 2   # Should execute
```

Testcase 5: Functionality of BLTU

The BLTU (Branch if Less Than Unsigned) instruction compares two registers as unsigned values, and if the first register holds a value less than the second register, the program jumps to the specified branch target (label). If the condition is false, execution continues with the next instruction in sequence.

```
ADDI x5, x0, 1          # x5 = 0x00000001
ADDI x6, x0, -1         # x6 = 0xFFFFFFFF
BLTU x5, x6, branch     # Should branch (unsigned)
ADDI x7, x0, 1          # Should be skipped
branch: ADDI x7, x0, 2  # Should execute
```

Testcase 6: Functionality of BGEU

The BGEU (Branch if Greater Than or Equal Unsigned) instruction compares two registers as unsigned values, and if the first register holds a value greater than or equal to the second register, the program

jumps to the specified branch target (label). If the condition is false, execution continues with the next instruction in sequence.

```
ADDI x5, x0, -1          # x5 = 0xFFFFFFFF (x0 + -1 = 0xFFFFFFFF)

ADDI x6, x0, 1           # x6 = 0x00000001 (x0 + 1 = 0x00000001)

BGEU x5, x6, branch      # Should branch (unsigned)

ADDI x7, x0, 1           # x7 = 1 (x0 + 1 = 1) - should be skipped

branch:  ADDI x7, x0, 2  # x7 = 2 (x0 + 2 = 2)
```

J type Instruction

For J-type instructions in RISC-V 32I, we will develop and execute test cases to validate their correct functionality. This includes verifying the proper calculation of jump target addresses, ensuring that the PC (Program Counter) is updated correctly, and confirming that relative offsets are sign-extended properly. We will test both forward and backward jumps to check branch accuracy and program flow. Additionally, we will verify the behavior of JAL (Jump and Link) by ensuring the return address is correctly stored in the destination register. These test cases will help ensure the correct implementation and execution of J-type instructions in our design.

I type Instruction

For I-type instructions in RISC-V32I, we will develop and execute test cases to verify their functionality and correctness. This includes testing arithmetic operations such as ADDI, SLTI, SLTIU, and logical operations like XORI, ORI, ANDI, ensuring correct immediate value handling and sign extension. Additionally, we will test shift instructions (SLLI, SRLI, SRAI) to validate proper bitwise shifting behavior. Load instructions (LB, LH, LW, LBU, LHU) will be tested to confirm correct memory access, address alignment, and sign extension where applicable. Special cases, such as boundary values, zero immediates, and negative immediates, will also be included to ensure robustness. These test cases will help verify the correct execution and handling of I-type instructions within our design.

U Type Instruction

For U-type instructions in RISC-V32I, we will design and execute test cases to validate their correctness and functionality. The primary focus will be on testing LUI (Load Upper Immediate) and AUIPC (Add Upper Immediate to PC) instructions. We will verify whether the immediate values are correctly loaded into the upper 20 bits of the destination register and ensure proper zero-extension of the lower bits. For AUIPC, we will validate if the instruction correctly computes the PC-relative address by adding the immediate value to the current program counter. Additionally, we will test various edge cases, including different immediate values (minimum, maximum, and random values), register variations, and

interactions with subsequent instructions to ensure seamless execution. These test cases will help in ensuring full compliance with the RISC-V32I specifications.

Future Works :

We are currently working on modifying our sign extension function. We will develop more test cases for R , I and B type instructions to test the corner cases. We will be using the roadmap mentioned above to generate test cases for J, I and U type instructions.