ECE 586
RISC-V Test Plan, Methodology, and Overview

Team 2

Portland State University
Maseeh College of Engineering & Computer Science
Department of Electrical & Computer Engineering

Authors:
Kane-Pardy, Chris
Monti, Violet

March 18, 2025

ECE 586
Computer Architecture

Contents

1 Methodology

Our testing methodology for this project is to use Case Analysis for every single instruction to ensure no bugs are within the program. Our addressing formats are each tested with debug statements when our debug mode is active, which ensures the proper decoding of the instruction based on the opcode. Memory structures and read/write functions are to be tested alongside our load and store instructions, which are the only way to read and write in our program. The PC is another one that we can see updated with the use of debug statements as our code is run.

As far as actually generating test cases goes, we're both using the Linux RISC-V toolchain provided by Mark, as well as using a function we made in C. The function takes in the arguments:

```
func7, rs2, rs1, func3, rd, opcode, imm_21, imm_13, imm_12, instruction;
```

Depending on the desired format the user enters, and creates them by concatenating the fields in the correct order to create an instruction of the desired format.

We're splitting the tasks of creating test code and test cases between the two of us, so that expected results vs actual results are unbiased. We're also verifying some of them using online sources such as hex calculators, 2's complement calculators, etc if we're ever doubtful of anything we've seen. All written (English) test cases can be found in this document. The ".mem" files can be found in our GitHub in the "`Mem_files`" folder.

2 Memory

This section covers the tests we did to ensure proper functionality of the memory in our simulation (including the stack, registers, and main memory).

Given the constraint of 64KB, we chose to statically declare an array (with `uint32_t` members), of size 16ki.

$(\frac{\text{Memory Size}}{\text{Word Size}} = \frac{2^{19}}{2^5} = 2^{14} = 16384 \text{ words})$

Both the stack and data portion of memory will be contained in this array, with the memory portion being initialized to address 0x0000 (array[0]), and the starting stack address being initialized to 65536, from which it will grow down in address (starting from array[2048]).

To maintain the possibility of being extensible, we set runtime arguments `-sp` to change the starting stack address, and `-s` to change the starting memory address.

For our registers, we also used a static array of size 32(`x0 - x31`), with `x0` (`zero`), `x1` (`ra`), and `x2` (sp) being set to 0. The data type of each member is an `uint32_t` to ensure anything we store maintains the proper length for proper use in ALU operations. This format should allow us to easily index our array `x` to find our desired array in code without introducing any further complications.

2.1 Reading and Storing Memory Image

Our function to open a file, parse through, and sort the contents into the proper format utilizes fscanf, and splits the contents into two portions. The first is the address, which is used to index our main memory array as `address / 4`, and the second is the instruction to be stored at the calculated location.

To test the functionality of this loop, we used the given prog.mem example:

```
0: fd010113
4: 02812623
8: 03010413
c: fca42e23
```

After compiling and running in our debug mode (determined at compile time by the argument `-DDEBUG`), we first used the following print statements:

```
#ifdef DEBUG
    fprintf(stderr, "Extracted memory addresss:      0x%08X\n", address);
    fprintf(stderr, "Extracted instruction contents: 0x%08X\n", instruction);
#endif
```

To confirm the file contents were being properly split up, then we called our memory dumping function:

```
void printAllMem(uint32_t array[], int size){
#ifdef DEBUG
for (int i = 0; i < size; i++){
    printf( "Array Member: %4d     Memory Address: 0x%08X     Contents: 0x%08X\n",
    i, 4*i, array[i]);
}
#endif
    return;
}
```

the output of which was sent to a log file and checked to verify proper functionality.

2.2 Reading and Writing Values

To read and write values from the input file to memory, we implemented 6 separate functions (readByte, readHalf-Word, readWord, writeByte, writeHalfWord, and writeWord).
Reads: Our read functions all take in the arguments (`uint32_t array[], int size, int address`), and use them to compute the target address and load the desired byte from the address into the desired location. These are the only places in the code that directly read from our memory array after its declaration. Our functions are:
readByte
readHalfWord
readWord
All of which are extensively tested in the `Load` section of this document.
Writes: The write instructions take in the same arguments as the reads, however, they have an added `uint32_t value`, which is used to compute the value to be written. These are the only places in code that can "write" to our memory array. Our functions are:
writeByte
writeHalfWord
writeWord
All of these are tested extensively in the `Store` section of this document.

2.3 Registers

The register array can be checked for accuracy after the run of a test file using the `printAllReg` function, which essentially functions the same as the `printAllMem` function, and prints the label of each register, along with its contents after exit from the **.mem** file.

2.4 Stack

The contents of the stack will also print with the `printAllMem` function and will start from the highest address in memory, and decrement by 4 as the stack is filled.

3 Program Counter

The program counter is the reference point for where the processor currently references the instructions within memory. Upon program start, a register called **PC** is initialized to 0, unless the argument **-s** <**address**> is added at runtime in which the program counter is initialized to the value set in <**address**>. After the initialization and the program code is loaded into memory, the contents of the PC register are used to index the array in which all of the program data, including the stack, is stored.

3.1 Incrementing

After each instruction is executed, the program counter *must* be updated. If the program fails to update the program counter, the program will execute the same instruction, unintentionally, until it is terminated. However, we cannot simply increment the program after every instruction. Some instructions, such as the Branch and Jump instruction types (and JALR), can set the program counter to an offset from the current PC, or set the PC to a value directly. If we were to increment after these jumps, we would end up overshooting the desired instruction by one. Therefore, each instruction that does **not** directly influence the PC after its execution has a statement to increment the program counter properly. We were able to implement this dedicated incrementing after we had a moderate skeleton of the data path for the simulation, including basic functions to parse the instructions and run them depending on which opcode was detected.

3.2 Fetching and Decoding

For the main simulation to work, it needs to grab instructions, decode them, and execute functions depending on the opcode and the functions. To start, the instruction is fetched using the current Program Counter register contents, jumping to the location in our data array containing the required four bytes. With the new instruction fetched, the operation code is extracted. Within RISC-V, the opcode is always the seven least significant bits of the instruction. Using the opcode, one of six functions can be called: **r-type**, **i-type**, **s-type**, **b-type**, **u-type**, and **j-type**, one for each main instruction format. This is needed, as each instruction format has a different interpretation of the bits within the instruction, so multiple interpretation formats are required.

Once within the -type function, the instruction is broken down into the components given within the instruction, such as source registers, destination, immediate fields, and function fields. If an instruction format has multiple opcodes in its category, such as the i-type format having LOAD, IMM, and JALR, sub-functions are created to help complete their desired actions and execution.

4 Instruction Formats and Instructions

With the sheer amount of individual instructions that will need to be tested, we created a program that can be called to piece together an RV32I instruction by giving it the individual pieces of the function, such as the humanreadable Immediate format, Funct3, and Funct7, and the opcode, all depending on which addressing mode is desired. This program can be found in the GitHub repository with the name **make_instruction.c**, and must be compiled by your machine before use.
Using this program is simple: call the compiled code with one of six arguments depending on the addressing mode you would like:

-r for Register

-i for Immediate (including Load)

-s for Store

-b for Branch

-u for Upper

-j for Jump

Once the function has been called with your desired mode, you will be prompted to enter the required fields for that mode. For the modes with immediate fields, the immediate should be entered *as is*, not in the jumbled mess that it will become in the instruction. Upon entering all the required arguments, the program will stitch together the instructions and output them into a hexadecimal format, which can be easily put into a .mem file to be inputted into the main ISA simulation.

4.1 Register Format

Register Mode instructions are all sorted by their opcode (`0b0110011`) into the function `r_type`. This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rd`, `func3`, `rs1`, `rs2`, `func7`.

We ensured that this mode was correctly chosen through the use of our debugging mode and the print statement-

```
#ifdef DEBUG
fprintf(stderr, "R-Type instruction breakdown:\n  Opcode: 0x%02X\n
R_Des: 0x%02X\n  Func3: 0x%02X\n  R_S1: 0x%02X\n  R_S2: 0x%02X\n
Func7: 0x%02X\n", opcode, rd, func3, rs1, rs2, func7);
#endif
```

-Which we've used to ensure every **R format** instruction is accurately chosen. In our `r_type` function exists a switch case based upon the `func3` bits.

4.1.1 add & sub

When the `func3` bits are `0x0`, there is a nested switch case that checks the `func7` bits of the instruction. For `0x00` the `add` instruction is selected, and for `0x20` the `sub` instruction is selected. These use the existing `+` and `-` functions in C.

add-

Case 1: Positive to Positive

```
li  a0, 4
li  a1, 6
add a0, a0, a1
```

Expected Result: `a0 = 10`

Case 2: Positive to Negative (Negative Result)

```
li  a0, 4
li  a1, -6
add a0, a0, a1
```

Expected Result: `a0 = -2`

Case 3: Positive to Negative (Positive Result)

```
li  a0, -4
li  a1, 6
add a0, a0, a1
```

Expected Result: `a0 = 2`

Case 4: Negative to Negative (Negative Result)

```
li  a0, -4
li  a1, -6
add a0, a0, a1
```

Expected Result: `a0 = -10`

sub-

Case 1: Positive from Positive (Positive Result)

```
li  a0, 6
li  a1, 4
sub a0, a0, a1
```

Expected Result: `a0 = 2`

Case 2: Positive from Positive (Negative Result)

```
li  a0, 4
li  a1, 6
sub a0, a0, a1
```

Expected Result: `a0 = -2`

Case 3: Positive from Negative (Negative Result)

```
li  a0, -6
li  a1, 4
sub a0, a0, a1
```

Expected Result: `a0 = -10`

Case 4: Negative from Positive (Positive Result)

```
li  a0, 4
li  a1, -6
sub a0, a0, a1
```

Expected Result: `a0 = 10`

Case 5: Negative from Negative (Positive Result)

```
li  a0, -4
li  a1, -6
sub a0, a0, a1
```

Expected Result: `a0 = 2`

Case 6: Negative from Negative (Negative Result)

```
li  a0, -6
li  a1, -4
sub a0, a0, a1
```

Expected Result: `a0 = -2`

4.1.2 `xor`

When the `func3` bits are `0x4`, the `xor` function is selected, which utilizes the already existing `^` function in C.

xor-

Case 1: 0's with 0's

```
li  a0, 0x00000000
li  a1, 0x00000000
li  a2, 0xFFFFFFFF
xor a2, a0, a1
```

Expected Result: `a2 = 0`

Case 2: 1's with 1's

```
li  a0, 0xFFFFFFFF
li  a1, 0xFFFFFFFF
li  a2, 0xFFFFFFFF
xor a2, a0, a1
```

Expected Result: `a2 = 0`

Case 3: 1's with 0's

```
li  a0, 0xFFFFFFFF
li  a1, 0x00000000
li  a2, 0x00000000
xor a2, a0, a1
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 4: 1's with 1's and 0's

```
li  a0, 0xFFFFFFFF
li  a1, 0x0F0F0F0F
li  a2, 0x00000000
xor a2, a0, a1
```

Expected Result: `a2 = 0xF0F0F0F0`

Case 5: 1's with 1's and 0's with sign extension (MSB 0)

```
li  a0, 255
li  a1, 0x00
li  a2, 0x00
xor a2, a0, a1
```

Expected Result: `a2 = 255`

Case 6: 1's with 1's and 0's with sign extension (MSB 1)

```
li  a0, -255
li  a1, 0x00
li  a2, 0x00
xor a2, a0, a1
```

Expected Result: `a2 = -255`

4.1.3 `or`

When the `func3` bits are `0x6`, the `or` function is selected, which utilizes the already existing `|` function in C.

or-

Case 1: 0's with 0's

```
li  a0, 0x00000000
li  a1, 0x00000000
li  a2, 0xFFFFFFFF
or a2, a0, a1
```

Expected Result: `a2 = 0x00000000`

Case 2: 1's with 1's

```
li  a0, 0xFFFFFFFF
li  a1, 0xFFFFFFFF
li  a2, 0x00000000
or a2, a0, a1
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 3: 1's with 0's

```
li  a0, 0xFFFFFFFF
li  a1, 0x00000000
li  a2, 0x00000000
or a2, a0, a1
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 4: 1's with 1's and 0's

```
li  a0, 0xFFFFFFFF
li  a1, 0x0F0F0F0F
li  a2, 0x00000000
or a2, a0, a1
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 5: 1's with 1's and 0's with sign extension (MSB 0)

```
li  a0, 255
li  a1, 0x00
li  a2, 0x00
or a2, a0, a1
```

Expected Result: `a2 = 255`

Case 6: 1's with 1's and 0's with sign extension (MSB 1)

```
li  a0, -255
li  a1, 0x00
li  a2, 0x00
or a2, a0, a1
```

Expected Result: `a2 = -255`

4.1.4 and

When the `func3` bits are `0x7`, the **`and`** function is selected, which utilizes the already existing `&` function in C.

and-

Case 1: 0's with 0's

```
li  a0, 0x00000000
li  a1, 0x00000000
li  a2, 0xFFFFFFFF
and a2, a0, a1
```

Expected Result: `a2 = 0x00000000`

Case 2: 1's with 1's

```
li  a0, 0xFFFFFFFF
li  a1, 0xFFFFFFFF
li  a2, 0x00000000
and a2, a0, a1
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 3: 1's with 0's

```
li  a0, 0xFFFFFFFF
li  a1, 0x00000000
li  a2, 0xFFFFFFFF
and a2, a0, a1
```

Expected Result: `a2 = 0x00000000`

Case 4: 1's with 1's and 0's

```
li  a0, 0xFFFFFFFF
li  a1, 0x0F0F0F0F
li  a2, 0x00000000
and a2, a0, a1
```

Expected Result: `a2 = 0x0F0F0F0F`

Case 5: 1's with 1's and 0's with sign extension (MSB 0)

```
li  a0, 255
li  a1, 128
li  a2, 0x00
and a2, a0, a1
```

Expected Result: `a2 = 128`

Case 6: 1's with 1's and 0's with sign extension (MSB 1)

```
li  a0, -255
li  a1, -512
li  a2, 0x00
and a2, a0, a1
```

Expected Result: `a2 = -512`

4.1.5 `sll`

When the `func3` bits are `0x1`, the `sll` function is selected, which utilizes the already existing `<<` function in C.

sll-

Case 1: sll on all 0's

```
li  a0, 0x00000000
li  a1, 12
li  a2, 0xFFFFFFFF
sll a2, a0, a1
```

Expected Result: `a2 = 0x00000000`

Case 2: sll on all 1's

```
li  a0, 0xFFFFFFFF
li  a1, 12
li  a2, 0x00000000
sll a2, a0, a1
```

Expected Result: `a2 = 0xFFFFF000`

Case 3: sll on 1's and 0's

```
li  a0, 0xF0F0F0F0
li  a1, 16
li  a2, 0xFFFFFFFF
sll a2, a0, a1
```

Expected Result: `a2 = 0xF0F00000`

Case 4: sll on 1's until all gone

```
li  a0, 0xFFFFFFFF
li  a1, 32
li  a2, 0xFFFFFFFF
sll a2, a0, a1
```

Expected Result: `a2 = 0x00000000`

4.1.6 `srl & sra`

When the `func3` bits are `0x5`, there is a nested switch case that checks the func7 bits of the instruction. For 0x00 the `srl` instruction is selected, and for 0x20 the `sra` instruction is selected. These use the existing ¿¿ functions in C and exploit that C automatically uses logical shifts for unsigned numbers, and arithmetic shifts for signed numbers.

`srl-`

Case 1: srl on all 0's

```
li  a0, 0x00000000
li  a1, 12
li  a2, 0xFFFFFFFF
srl a2, a0, a1
```

Expected Result: `a2 = 0x00000000`

Case 2: srl on all 1's

```
li  a0, 0xFFFFFFFF
li  a1, 12
li  a2, 0x00000000
srl a2, a0, a1
```

Expected Result: `a2 = 0x000FFFFF`

Case 3: srl on 1's and 0's

```
li  a0, 0xF0F0F0F0
li  a1, 16
li  a2, 0xFFFFFFFF
srl a2, a0, a1
```

Expected Result: `a2 = 0x0000F0F0`

Case 4: srl on 1's until all gone

```
li  a0, 0xFFFFFFFF
li  a1, 31
li  a2, 0xFFFFFFFF
srl a2, a0, a1
```

Expected Result: `a2 = 0x00000001`

sra-

Case 1: sra on all 0's

```
li  a0, 0x00000000
li  a1, 12
li  a2, 0xFFFFFFFF
sra a2, a0, a1
```

Expected Result: `a2 = 0x00000000`

Case 2: sra on all 1's

```
li  a0, 0xFFFFFFFF
li  a1, 12
li  a2, 0x00000000
srl a2, a0, a1
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 3: sra on 1's and 0's

```
li  a0, 0xF0F0F0F0
li  a1, 16
li  a2, 0xFFFFFFFF
sra a2, a0, a1
```

Expected Result: `a2 = 0xFFFFF0F0`

Case 4: sra on 1's until all gone

```
li  a0, 0xFFFFFFFF
li  a1, 31
li  a2, 0x00000000
sra a2, a0, a1
```

Expected Result: `a2 = 0xFFFFFFFF`

4.1.7 `slt`

When the `func3` bits are `0x2`, the `slt` function is selected, which utilizes the already existing `<` function in C, with a ternary operator and signed copies of our register.

slt-

Case 1: slt negative vs positive (always true)

```
li  a0, -12
li  a1, 12
li  a2, 0xFFFFFFFF
slt a2, a0, a1
```

Expected Result: `a2 = 1`

Case 2: slt negative vs negative (true)

```
li  a0, -24
li  a1, -12
li  a2, 0x00000000
slt a2, a0, a1
```

Expected Result: `a2 = 1`

Case 3: slt negative vs negative (false)

```
li  a0, -12
li  a1, -24
li  a2, 0x00000000
slt a2, a0, a1
```

Expected Result: `a2 = 0`

Case 4: slt positive vs positive (true)

```
li  a0, 5
li  a1, 16
li  a2, 0xFFFFFFFF
slt a2, a0, a1
```

Expected Result: `a2 = 1`

Case 5: slt positive vs positive (false)

```
li  a0, 32
li  a1, 16
li  a2, 0xFFFFFFFF
slt a2, a0, a1
```

Expected Result: `a2 = 0`

Case 6: slt positive vs negative (always false)

```
li  a0, 32
li  a1, -32
li  a2, 0xFFFFFFFF
slt a2, a0, a1
```

Expected Result: `a2 = 0`

Case 7: slt equal (always false)

```
li  a0, 32
li  a1, 32
li  a2, 0xFFFFFFFF
slt a2, a0, a1
```

Expected Result: `a2 = 0`

4.1.8 sltu

When the `func3` bits are `0x3`, the `sltu` function is selected, which utilizes the already existing `<` function in C, with a ternary operator.

sltu-

Case 1: sltu negative vs positive (always false)

```
li  a0, -12
li  a1, 12
li  a2, 0xFFFFFFFF
sltu a2, a0, a1
```

Expected Result: `a2 = 0`

Case 2: sltu negative vs negative (true)

```
li  a0, -5
li  a1, -1
li  a2, 0x00000000
sltu a2, a0, a1
```

Expected Result: `a2 = 1`

Case 3: sltu negative vs negative (false)

```
li  a0, -1
li  a1, -5
li  a2, 0x00000000
sltu a2, a0, a1
```

Expected Result: `a2 = 0`

Case 4: sltu positive vs positive (true)

```
li  a0, 5
li  a1, 16
li  a2, 0xFFFFFFFF
sltu a2, a0, a1
```

Expected Result: `a2 = 1`

Case 5: sltu positive vs positive (false)

```
li  a0, 32
li  a1, 16
li  a2, 0xFFFFFFFF
sltu a2, a0, a1
```

Expected Result: `a2 = 0`

Case 6: sltu positive vs negative (always true)

```
li  a0, 32
li  a1, -32
li  a2, 0xFFFFFFFF
sltu a2, a0, a1
```

Expected Result: `a2 = 1`

Case 7: sltu equal (always false)

```
li  a0, 32
li  a1, 32
li  a2, 0xFFFFFFFF
sltu a2, a0, a1
```

Expected Result: `a2 = 0`

4.2 Immediate Format

Immediate Mode instructions are all sorted by their opcode (`0b0010011`) into the function `i_type`. This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rd`, `func3`, `rs1`, and `immediate`.

We ensured that this mode was correctly chosen through the use of our debugging mode and the print statement-

```
#ifdef DEBUG
fprintf(stderr, "I-Type instruction breakdown:\n    Opcode: 0x%02X\n
R_Des: 0x%02X\n    Func3: 0x%02X\n    R_S1: 0x%02X\n
Immediate: 0x%03X\n", opcode, rd, func3, rs1, imm);
#endif
```

-Which we've used to ensure every **I format** instruction is accurately chosen. In our `i_type` function exists a switch case based upon the `func3` bits.

4.2.1 addi

When the `func3` bits are `0x0`, the `addi` function is selected, which utilizes the already existing `+` function in C.

addi-

Case 1: Positive to 0

```
add a0, x0, x0
addi a0, a0, 32
```

Expected Result: `a0 = 32`

Case 2: Negative to 0

```
add a0, x0, x0
addi a0, a0, -32
```

Expected Result: `a0 = -32`

Case 3: 0 to 0

```
add a0, x0, x0
addi a0, a0, 0
```

Expected Result: `a0 = 0`

4.2.2 xori

When the `func3` bits are `0x4`, the `xori` function is selected, which utilizes the already existing `^` function in C.

xori-

Case 1: 0's with 0's

```
li   a0, 0x00000000
li   a2, 0xFFFFFFFF
xori a2, a0, 0x00000000
```

Expected Result: `a2 = 0`

Case 2: 1's with 1's

```
li   a0, 0xFFFFFFFF
li   a2, 0xFFFFFFFF
xori a2, a0, 0xFFFFFFFF
```

Expected Result: `a2 = 0`

Case 3: 1's with 0's

```
li   a0, 0x00000000
li   a2, 0x00000000
xori a2, a0, 0xFFFFFFFF
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 4: 1's with 1's and 0's

```
li   a0, 0x0F0F0F0F
li   a2, 0x00000000
xori a2, a0, 0xFFFFFFFF
```

Expected Result: `a2 = 0xF0F0F0F0`

Case 5: 1's with 1's and 0's with sign extension (MSB 0)

```
li   a0, 255
li   a2, 0x000000000
xori a2, a0, 0x00000000
```

Expected Result: `a2 = 255`

Case 6: 1's with 1's and 0's with sign extension (MSB 1)

```
li   a0, -255
li   a2, 0x00000000
xori a2, a0, 0x00000000
```

Expected Result: `a2 = -255`

4.2.3 `ori`

When the `func3` bits are `0x6`, the `ori` function is selected, which utilizes the already existing `|` function in C.

`ori-`

Case 1: 0's with 0's

```
li  a0, 0x00000000
li  a2, 0xFFFFFFFF
ori a2, a0, 0x00000000
```

Expected Result: `a2 = 0x00000000`

Case 2: 1's with 1's

```
li  a0, 0xFFFFFFFF
li  a2, 0x00000000
ori a2, a0, 0xFFFFFFFF
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 3: 1's with 0's

```
li  a0, 0xFFFFFFFF
li  a2, 0x00000000
ori a2, a0, 0x00000000
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 4: 1's with 1's and 0's

```
li  a0, 0x0F0F0F0F
li  a2, 0x00000000
ori a2, a0, 0xFFFFFFFF
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 5: 1's with 1's and 0's with sign extension (MSB 0)

```
li  a0, 255
li  a2, 0x00
ori a2, a0, 0x00000000
```

Expected Result: `a2 = 255`

Case 6: 1's with 1's and 0's with sign extension (MSB 1)

```
li  a0, -255
li  a2, 0x00
ori a2, a0, 0x00000000
```

Expected Result: `a2 = -255`

4.2.4 andi

When the `func3` bits are `0x7`, the `andi` function is selected, which utilizes the already existing `&` function in C.

andi-

Case 1: 0's with 0's

```
li   a0, 0x00000000
li   a2, 0xFFFFFFFF
andi a2, a0, 0x00000000
```

Expected Result: `a2 = 0x00000000`

Case 2: 1's with 1's

```
li   a0, 0xFFFFFFFF
li   a2, 0x00000000
andi a2, a0, 0xFFFFFFFF
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 3: 1's with 0's

```
li   a0, 0xFFFFFFFF
li   a2, 0xFFFFFFFF
andi a2, a0, 0x00000000
```

Expected Result: `a2 = 0x00000000`

Case 4: 1's with 1's and 0's

```
li   a0, 0x0F0F0F0F
li   a2, 0x00000000
andi a2, a0, 0xFFFFFFFF
```

Expected Result: `a2 = 0x0F0F0F0F`

Case 5: 1's with 1's and 0's with sign extension (MSB 0)

```
li   a0, 255
li   a2, 0x00
andi a2, a0, 128
```

Expected Result: `a2 = 128`

Case 6: 1's with 1's and 0's with sign extension (MSB 1)

```
li   a0, -255
li   a2, 0x00
andi a2, a0, -512
```

Expected Result: `a2 = -512`

4.2.5 slli

When the `func3` bits are `0x1`, the `slli` function is selected, which utilizes the already existing `<<` function in C.

slli-

Case 1: slli on all 0's

```
li  a0, 0x00000000
li  a2, 0xFFFFFFFF
slli a2, a0, 12
```

Expected Result: `a2 = 0x00000000`

Case 2: slli on all 1's

```
li  a0, 0xFFFFFFFF
li  a2, 0x00000000
slli a2, a0, 12
```

Expected Result: `a2 = 0xFFFFF000`

Case 3: slli on 1's and 0's

```
li  a0, 0xF0F0F0F0
li  a2, 0xFFFFFFFF
slli a2, a0, 16
```

Expected Result: `a2 = 0xF0F00000`

Case 4: slli on 1's until all gone

```
li  a0, 0xFFFFFFFF
li  a2, 0xFFFFFFFF
slli a2, a0, 31
```

Expected Result: `a2 = 0x80000000`

4.2.6 srli & srai

When the `func3` bits are `0x5`, there is a nested switch case that checks the func7 bits of the instruction. For 0x00 the `srli` instruction is selected, and for 0x20 the `srai` instruction is selected. These use the existing ¿¿ functions in C and exploit that C automatically uses logical shifts for unsigned numbers, and arithmetic shifts for signed numbers.

srli-

Case 1: srli on all 0's

```
li  a0, 0x00000000
li  a2, 0xFFFFFFFF
srli a2, a0, 12
```

Expected Result: `a2 = 0x00000000`

Case 2: srli on all 1's

```
li  a0, 0xFFFFFFFF
li  a2, 0x00000000
srli a2, a0, 12
```

Expected Result: `a2 = 0x000FFFFF`

Case 3: srli on 1's and 0's

```
li  a0, 0xF0F0F0F0
li  a2, 0xFFFFFFFF
srli a2, a0, 16
```

Expected Result: `a2 = 0x0000F0F0`

Case 4: srli on 1's until all gone

```
li  a0, 0xFFFFFFFF
li  a2, 0xFFFFFFFF
srli a2, a0, 31
```

Expected Result: `a2 = 0x00000001`

`srai-`

Case 1: srai on all 0's

```
li  a0, 0x00000000
li  a2, 0xFFFFFFFF
srai a2, a0, 12
```

Expected Result: `a2 = 0x00000000`

Case 2: srai on all 1's

```
li  a0, 0xFFFFFFFF
li  a2, 0x00000000
srli a2, a0, 12
```

Expected Result: `a2 = 0xFFFFFFFF`

Case 3: srai on 1's and 0's

```
li  a0, 0xF0F0F0F0
li  a2, 0xFFFFFFFF
srai a2, a0, 16
```

Expected Result: `a2 = 0xFFFFF0F0`

Case 4: srai on 1's until all gone

```
li  a0, 0xFFFFFFFF
li  a2, 0x00000000
srai a2, a0, 31
```

Expected Result: `a2 = 0xFFFFFFFF`

4.2.7 `slti`

When the `func3` bits are `0x2`, the `slti` function is selected, which utilizes the already existing `<` function in C, with a ternary operator and signed copies of our register.

`slti-`

Case 1: slti negative vs positive (always true)

```
li  a0, -12
li  a2, 0xFFFFFFFF
slti a2, a0, 12
```

Expected Result: `a2 = 1`

Case 2: slti negative vs negative (true)

```
li  a0, -24
li  a2, 0x00000000
slti a2, a0, -12
```

Expected Result: `a2 = 1`

Case 3: slti negative vs negative (false)

```
li  a0, -12
li  a2, 0x00000000
slti a2, a0, -24
```

Expected Result: `a2 = 0`

Case 4: slti positive vs positive (true)

```
li  a0, 5
li  a2, 0xFFFFFFFF
slti a2, a0, 16
```

Expected Result: `a2 = 1`

Case 5: slti positive vs positive (false)

```
li  a0, 32
li  a2, 0xFFFFFFFF
slti a2, a0, 16
```

Expected Result: `a2 = 0`

Case 6: slti positive vs negative (always false)

```
li  a0, 32
li  a2, 0xFFFFFFFF
slti a2, a0, -32
```

Expected Result: `a2 = 0`

Case 7: slti equal (always false)

```
li  a0, 32
li  a2, 0xFFFFFFFF
slti a2, a0, 32
```

Expected Result: `a2 = 0`

4.2.8 `sltiu`

When the `func3` bits are `0x3`, the `sltiu` function is selected, which utilizes the already existing `<` function in C, with a ternary operator.

`sltiu-`

Case 1: sltiu negative vs positive (always false)

```
li  a0, -12
li  a2, 0xFFFFFFFF
sltiu a2, a0, 12
```

Expected Result: `a2 = 0`

Case 2: sltiu negative vs negative (true)

```
li  a0, -5
li  a2, 0x00000000
sltiu a2, a0, -1
```

Expected Result: `a2 = 1`

Case 3: sltiu negative vs negative (false)

```
li  a0, -1
li  a2, 0x00000000
sltiu a2, a0, -5
```

Expected Result: `a2 = 0`

Case 4: sltiu positive vs positive (true)

```
li  a0, 5
li  a2, 0xFFFFFFFF
sltiu a2, a0, 16
```

Expected Result: `a2 = 1`

Case 5: sltiu positive vs positive (false)

```
li  a0, 32
li  a2, 0xFFFFFFFF
sltiu a2, a0, 16
```

Expected Result: `a2 = 0`

Case 6: sltiu positive vs negative (always true)

```
li  a0, 32
li  a2, 0xFFFFFFFF
sltiu a2, a0, -32
```

Expected Result: `a2 = 1`

Case 7: sltiu equal (always false)

```
li  a0, 32
li  a2, 0xFFFFFFFF
sltiu a2, a0, 32
```

Expected Result: `a2 = 0`

4.3 Load

`Load` instructions are technically Immediate Mode instructions and are all sorted by their distinct opcode (`0b0000011`) in the function `i_type`. This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rd`, `func3`, `rs1`, and `immediate` like regular immediates.

We ensured that this mode was correctly chosen through the use of the same print statement-

```
#ifdef DEBUG
fprintf(stderr, "I-Type instruction breakdown:\n    Opcode: 0x%02X\n
R_Des: 0x%02X\n    Func3: 0x%02X\n    R_S1: 0x%02X\n
Immediate: 0x%03X\n", opcode, rd, func3, rs1, imm);
#endif
```

-Which we've used to ensure every **I format** instruction is accurately chosen. In our `i_type` function exists a switch case based upon the `opcode` bits. From there we move to a separate `load` function, which contains a switch case based upon the `func3` bits for the 5 load instructions: `lb`, `lh`, `lw`, `lbu`, `lhu`.

4.3.1 lb

`lb` (`func3 = 0x0`) uses our `readByte` function, which uses (address / 4), and (address % 4), to load the desired byte from the 4-byte word stored at the address. The value from `readByte` is then sign-extended based on the MSB bit to fill the full word, which is then stored in the register array.

`lb-`

Case 1: MSB 0 (byte 1 of 4)

```
0: lb s2, 8(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00000033`

Case 2: MSB 0 (byte 2 of 4)

```
0: lb s2, 9(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00000022`

Case 3: MSB 0 (byte 3 of 4)

```
0: lb s2, 10(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00000011`

Case 4: MSB 0 (byte 4 of 4)

```
0: lb s2, 11(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00000055`

Case 5: MSB 1 (byte 1 of 4)

```
0: lb s2, 8(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0xFFFFFFFF`

Case 6: MSB 1 (byte 2 of 4)

```
0: lb s2, 9(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0xFFFFFFDD`

Case 7: MSB 1 (byte 3 of 4)

```
0: lb s2, 10(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0xFFFFFFAA`

Case 8: MSB 1 (byte 4 of 4)

```
0: lb s2, 11(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0xFFFFFF88`

4.3.2 `lh`

`lh` (`func3 = 0x1`) uses our `readHalfWord` function, which uses (address / 4), and (address % 4), to load the desired half word from the 4-byte word stored at the address. The value from `readHalfWord` is then sign-extended based on the MSB bit to fill the full word, which is then stored in the register array.

lh-

Case 1: MSB 0 (byte 1 of 4)

```
0: lh, s2, 8(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00002233`

Case 2: MSB 0 (byte 3 of 4)

```
0: lh, s2, 10(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00005511`

Case 3: MSB 1 (byte 1 of 4)

```
0: lh, s2, 8(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0xFFFFDDFF`

Case 4: MSB 1 (byte 3 of 4)

```
0: lh, s2, 10(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0xFFFF88AA`

Case 5: Misaligned (byte 2 of 4)

```
0: lh, s2, 9(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `error(1)`

Case 6: Misaligned (byte 4 of 4)

```
0: lh, s2, 11(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `error(1)`

4.3.3 `lw`

`lw` (`func3 = 0x2`) uses our **`readWord`** function, which uses (address / 4) to load the desired word stored at the address. The value from **`readWord`** is then stored in the register array.

`lw-`

Case 1: Correct reference (byte 1 of 4)

```
0: lw, s2, 8(x0)
4: jr 0
8: 0xFEDCBA98
```

Expected Result: `s2 = 0xFEDCBA98`

Case 2: Misaligned reference (byte 2 of 4)

```
0: lw, s2, 9(x0)
4: jr 0
8: 0xFEDCBA98
```

Expected Result: `error(1)`

Case 3: Misaligned reference (byte 3 of 4)

```
0: lw, s2, 10(x0)
4: jr 0
8: 0xFEDCBA98
```

Expected Result: `error(1)`

Case 4: Misaligned reference (byte 4 of 4)

```
0: lw, s2, 11(x0)
4: jr 0
8: 0xFEDCBA98
```

Expected Result: `error(1)`

4.3.4 `lbu`

`lbu` (`func3 = 0x4`) uses our **`readByte`** function, which uses (address / 4), and (address % 4), to load the desired byte from the 4-byte word stored at the address. The value from **`readByte`** is then zero-extended based on the MSB bit to fill the full word, which is then stored in the register array.

`lbu-`

Case 1: MSB 0 (byte 1 of 4)

```
0: lbu s2, 8(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00000033`

Case 2: MSB 0 (byte 2 of 4)

```
0: lbu s2, 9(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00000022`

Case 3: MSB 0 (byte 3 of 4)

```
0: lbu s2, 10(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00000011`

Case 4: MSB 0 (byte 4 of 4)

```
0: lbu s2, 11(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00000055`

Case 5: MSB 1 (byte 1 of 4)

```
0: lbu s2, 8(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0x000000FF`

Case 6: MSB 1 (byte 2 of 4)

```
0: lbu s2, 9(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0x000000DD`

Case 7: MSB 1 (byte 3 of 4)

```
0: lbu s2, 10(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0x000000AA`

Case 8: MSB 1 (byte 4 of 4)

```
0: lbu s2, 11(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0x00000088`

4.3.5 `lhu`

`lhu` (`func3 = 0x5`) uses our `readHalfWord` function, which uses (address / 4), and (address % 4), to load the desired half word from the 4-byte word stored at the address. The value from `readHalfWord` is then zero-extended based on the MSB bit to fill the full word, which is then stored in the register array.

`lhu-`

Case 1: MSB 0 (byte 1 of 4)

```
0: lhu, s2, 8(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00002233`

Case 2: MSB 0 (byte 3 of 4)

```
0: lhu, s2, 10(x0)
4: jr 0
8: 0x55112233
```

Expected Result: `s2 = 0x00005511`

Case 3: MSB 1 (byte 1 of 4)

```
0: lhu, s2, 8(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0x0000DDFF`

Case 4: MSB 1 (byte 3 of 4)

```
0: lhu, s2, 10(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `s2 = 0x000088AA`

Case 5: Misaligned (byte 2 of 4)

```
0: lhu, s2, 9(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `error(1)`

Case 6: Misaligned (byte 4 of 4)

```
0: lhu, s2, 11(x0)
4: jr 0
8: 0x88AADDFF
```

Expected Result: `error(1)`

4.4 jalr

The `jalr` instruction is technically an Immediate Format instruction and is sorted by the distinct opcode (`0b1100111`) in the function `i_type`. This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rd`, `func3`, `rs1`, and `immediate` like regular immediates. Using the same switch case based on opcode in the `i_type` function, the `jalr` will simply store `pc + 4` into `rd`, then load `(((contents of rs1) + immediate) & 0xFFFFFFFE)` into the `pc`.

jalr-

Case 1: Basic Functionality

```
li    s1, 0x00000000
li    s2, 0x00000000
jalr  s2, s1, 0x100
```

Expected Result: `s2 = pc + 4`, `pc = 0x100`

Case 2: LSB → Zero Verification

```
li    s1, 0x0000000F
li    s2, 0x00000000
jalr  s2, s1, 0x100
```

Expected Result: `s2 = pc + 4`, `pc = 0x10E`

Case 3: Return to previous `pc`

```
li    s2, 0x00000000
jalr  s2, x0, 0x0
```

Expected Result: `s2 = pc + 4`, `pc = 0x0`

4.5 Store Format

`Store` instructions use the Store Format and are all sorted by their distinct opcode (`0b0100011`) in the function `s_type`. This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rs1`, `rs2`, `func`, and `immediate`.

We ensured that this mode was correctly chosen through the use of the same print statement-

```
#ifdef DEBUG
fprintf(stderr, "S-Type instruction breakdown:\n    Opcode: 0x%02X\n    Func3: 0x%02X\n
R_S1: 0x%02X\n    R_S2: 0x%02X\n    Immediate: 0x%04X\n", opcode, func3, rs1, rs2, imm);
#endif
```

-Which we've used to ensure every **S format** instruction is accurately chosen. In our `s_type` function exists a switch case based upon the `func3` bits, which chooses between the 3 store instructions: `sb`, `sh`, `sw`.

4.5.1 `sb`

`sb` (`func3 = 0x0`) uses our `writeByte` function, which uses (address / 4), and (address % 4), to store the desired byte from the 4-byte word stored in the selected register, which is then stored in memory.

`sb-`

Case 1: MSB 0 (Byte 1 of 4)

```
0: li s2, 0x11223344
4: sb s2, 80(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x00000044`

Case 2: MSB 0 (Byte 2 of 4)

```
0: li s2, 0x11223344
4: sb s2, 81(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x00004400`

Case 3: MSB 0 (Byte 3 of 4)

```
0: li s2, 0x11223344
4: sb s2, 82(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x00440000`

Case 4: MSB 0 (Byte 4 of 4)

```
0: li s2, 0x11223344
4: sb s2, 83(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x44000000`

Case 5: MSB 1 (Byte 1 of 4)

```
0: li s2, 0xFFEEDDCC
4: sb s2, 80(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x000000CC`

Case 6: MSB 1 (Byte 2 of 4)

```
0: li s2, 0xFFEEDDCC
4: sb s2, 81(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x0000CC00`

Case 7: MSB 1 (Byte 3 of 4)

```
0: li s2, 0xFFEEDDCC
4: sb s2, 82(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x00CC0000`

Case 8: MSB 1 (Byte 4 of 4)

```
0: li s2, 0xFFEEDDCC
4: sb s2, 83(x0)
8: jr 0
```

Expected Result: `mem(80) = 0xCC000000`

4.5.2 sh

`sh` (`func3 = 0x1`) uses our `writeHalfWord` function, which uses (address / 4), and (address % 4), to store the desired half word from the 4-byte word stored in the selected register, which is then stored in memory.

`sh-`

Case 1: MSB 0 (byte 1 of 4)

```
0: li s2, 0x11223344
4: sh s2, 80(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x00003344`

Case 2: MSB 0 (byte 3 of 4)

```
0: li s2, 0x11223344
4: sb s2, 82(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x33440000`

Case 3: MSB 1 (byte 1 of 4)

```
0: li s2, 0xFFEEDDCC
4: sb s2, 80(x0)
8: jr 0
```

Expected Result: `mem(80) = 0x0000DDCC`

Case 4: MSB 1 (byte 3 of 4)

```
0: li s2, 0xFFEEDDCC
4: sb s2, 82(x0)
8: jr 0
```

Expected Result: `mem(80) = 0xDDCC0000`

Case 5: Misaligned (byte 2 of 4)

```
0: li s2, 0xFFEEDDCC
4: sb s2, 81(x0)
8: jr 0
```

Expected Result: `error(1)`

Case 6: Misaligned (byte 4 of 4)

```
0: li s2, 0xFFEEDDCC
4: sb s2, 83(x0)
8: jr 0
```

Expected Result: `error(1)`

4.5.3 `sw`

`sw` (`func3 = 0x2`) uses our `writeWord` function, which uses (address / 4) to store the desired word from the selected register, which is then stored in memory.

`sw-`

Case 1: Correct Reference (byte 1 of 4)

```
0: li s2, 0xFFEEDDCC
4: sw s2, 80(x0)
8: jr 0
```

Expected Result: `mem(80) = 0xFFEEDDCC`

Case 2: Misaligned Reference (byte 2 of 4)

```
0: li s2, 0xFFEEDDCC
4: sw s2, 81(x0)
8: jr 0
```

Expected Result: `error(1)`

Case 3: Misaligned Reference (byte 3 of 4)

```
0: li s2, 0xFFEEDDCC
4: sw s2, 82(x0)
8: jr 0
```

Expected Result: `error(1)`

Case 4: Misaligned Reference (byte 4 of 4)

```
0: li s2, 0xFFEEDDCC
4: sw s2, 83(x0)
8: jr 0
```

Expected Result: `error(1)`

4.6 Branch Format

`Branch` instructions use the Branch Format and are all sorted by their distinct opcode (`0b1100011`) in the function `b_type`. This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rs1`, `rs2`, `func`, and `immediate`.

We ensured that this mode was correctly chosen through the use of the same print statement-

```
#ifdef DEBUG
fprintf(stderr, "B-Type instruction breakdown:\n    Opcode: 0x%02X\n    Func3: 0x%02X\n
R_S1: 0x%02X\n    R_S2: 0x%02X\n    Immediate: 0x%04X\n", opcode, func3, rs1, rs2, imm);
#endif
```

-Which we've used to ensure every **B format** instruction is accurately chosen. In our `b_type` function exists a switch case based upon the `func3` bits, which chooses between the 6 store instructions: `beq`, `bne`, `blt`, `bge`, `bltu`, and `bgeu`.

4.6.1 beq

`beq` (`func3 = 0x0`) uses the already existing == comparison in C to check for equality between the registers. If true, the pc is incremented by the immediate.

`beq-`

Case 1: Equal (Taken Forwards)

```
li  s1, 5
li  s2, 5
beq s1, s2, 10
```

Expected Result: `pc += 10`

Case 2: Equal (Taken Backwards)

```
li  s1, 5
li  s2, 5
beq s1, s2, -10
```

Expected Result: `pc -= 10`

Case 3: Not Equal

```
li  s1, 10
li  s2, 5
beq s1, s2, 10
```

Expected Result: `pc += 4`

Case 4: Equal but Opposite Signs (Not Equal)

```
li  s1, -5
li  s2, 5
beq s1, s2, 10
```

Expected Result: `pc += 4`

4.6.2 bne

`bne` (`func3 = 0x1`) uses the already existing != comparison in C to check for equality between the registers. If true, the pc is incremented by the immediate.

bne-

Case 1: Equal

```
li  s1, 5
li  s2, 5
bne s1, s2, 10
```

Expected Result: `pc += 4`

Case 2: Equal but Opposite Signs (Not Equal)

```
li  s1, -5
li  s2, 5
bne s1, s2, 10
```

Expected Result: `pc += 10`

Case 3: Not Equal (Taken Forwards)

```
li  s1, 10
li  s2, 5
bne s1, s2, 10
```

Expected Result: `pc += 10`

Case 4: Not Equal (Taken Backwards)

```
li  s1, 10
li  s2, 5
bne s1, s2, -10
```

Expected Result: `pc -= 10`

4.6.3 `blt`

`blt` (`func3 = 0x4`) uses the already existing ¡ comparison in C to check for equality between the registers. If true, the pc is incremented by the immediate.

`blt-`

Case 1: less than (positive & positive)

```
li  s1, 1
li  s2, 5
blt s1, s2, 10
```

Expected Result: `pc += 10`

Case 2: less than (negative & positive)

```
li  s1, -10
li  s2, 5
blt s1, s2, 10
```

Expected Result: `pc += 10`

Case 3: less than (negative & negative)

```
li  s1, -10
li  s2, -5
blt s1, s2, 10
```

Expected Result: `pc += 10`

Case 4: Equal

```
li  s1, 5
li  s2, 5
blt s1, s2, 10
```

Expected Result: `pc += 4`

Case 5: greater than (negative & negative)

```
li  s1, -1
li  s2, -5
blt s1, s2, 10
```

Expected Result: `pc += 4`

Case 6: greater than (positive & negative)

```
li  s1, 10
li  s2, -5
blt s1, s2, 10
```

Expected Result: `pc += 4`

Case 7: greater than (positive & positive)

```
li  s1, 10
li  s2, 5
blt s1, s2, 10
```

Expected Result: `pc += 4`

4.6.4 bge

`bge` (`func3 = 0x5`) uses the already existing ¿= comparison in C to check for equality between the registers. If true, the pc is incremented by the immediate.

`bge-`

Case 1: less than (positive & positive)

```
li  s1, 1
li  s2, 5
bge s1, s2, 10
```

Expected Result: `pc += 4`

Case 2: less than (negative & positive)

```
li  s1, -10
li  s2, 5
bge s1, s2, 10
```

Expected Result: `pc += 4`

Case 3: less than (negative & negative)

```
li  s1, -10
li  s2, -5
bge s1, s2, 10
```

Expected Result: `pc += 4`

Case 4: Equal (Both Positive)

```
li  s1, 5
li  s2, 5
bge s1, s2, 10
```

Expected Result: `pc += 10`

Case 5: Equal (Both Negative)

```
li  s1, -12
li  s2, -12
bge s1, s2, 10
```

Expected Result: `pc += 10`

Case 6: greater than (negative & negative)

```
li  s1, -1
li  s2, -5
bge s1, s2, 10
```

Expected Result: `pc += 10`

Case 7: greater than (positive & negative)

```
li  s1, 10
li  s2, -5
bge s1, s2, 10
```

Expected Result: `pc += 10`

Case 8: greater than (positive & positive)

```
li  s1, 10
li  s2, 5
bge s1, s2, 10
```

Expected Result: `pc += 10`

4.6.5 `bltu`

`bltu` (`func3 = 0x6`) uses the already existing ¡ comparison in C to check for equality between the registers with unsigned versions of them. If true, the pc is incremented by the immediate.

`bltu-`

Case 1: less than (positive & positive)

```
li  s1, 1
li  s2, 5
bltu s1, s2, 10
```

Expected Result: `pc += 10`

Case 2: less than (negative & positive)

```
li  s1, -10
li  s2, 5
bltu s1, s2, 10
```

Expected Result: `pc += 4`

Case 3: less than (negative & negative)

```
li  s1, -10
li  s2, -5
bltu s1, s2, 10
```

Expected Result: `pc += 10`

Case 4: Equal

```
li  s1, 5
li  s2, 5
bltu s1, s2, 10
```

Expected Result: `pc += 4`

Case 5: greater than (negative & negative)

```
li  s1, -1
li  s2, -5
bltu s1, s2, 10
```

Expected Result: `pc += 4`

Case 6: greater than (positive & negative)

```
li  s1, 10
li  s2, -5
bltu s1, s2, 10
```

Expected Result: `pc += 10`

Case 7: greater than (positive & positive)

```
li  s1, 10
li  s2, 5
bltu s1, s2, 10
```

Expected Result: `pc += 4`

4.6.6 bgeu

`bgeu` (`func3 = 0x7`) uses the already existing ¿= comparison in C to check for equality between the registers with unsigned versions of them. If true, the pc is incremented by the immediate.

bgeu-

Case 1: less than (positive & positive)

```
li  s1, 1
li  s2, 5
bgeu s1, s2, 10
```

Expected Result: `pc += 4`

Case 2: less than (negative & positive)

```
li  s1, -10
li  s2, 5
bgeu s1, s2, 10
```

Expected Result: `pc += 10`

Case 3: less than (negative & negative)

```
li  s1, -10
li  s2, -5
bgeu s1, s2, 10
```

Expected Result: `pc += 4`

Case 4: Equal (Both Positive)

```
li  s1, 5
li  s2, 5
bgeu s1, s2, 10
```

Expected Result: `pc += 10`

Case 5: Equal (Both Negative)

```
li  s1, -12
li  s2, -12
bgeu s1, s2, 10
```

Expected Result: `pc += 10`

Case 6: greater than (negative & negative)

```
li  s1, -1
li  s2, -5
bgeu s1, s2, 10
```

Expected Result: `pc += 10`

Case 7: greater than (positive & negative)

```
li  s1, 10
li  s2, -5
bgeu s1, s2, 10
```

Expected Result: `pc += 4`

Case 8: greater than (positive & positive)

```
li  s1, 10
li  s2, 5
bgeu s1, s2, 10
```

Expected Result: `pc += 10`

4.7 Upper Format

Upper Format contains two different instructions, `auipc`(`0b0010111`) and `lui` (`0b0110111`). This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rd` and `immediate`.

We ensured that this mode was correctly chosen through the use of our debugging mode and the print statement-

```
#ifdef DEBUG
fprintf(stderr, "U-Type instruction breakdown:\n    Opcode: 0x%02X\n
R_Des: 0x%02X\n    Immediate: 0x%08X\n", opcode, rd, imm);
#endif
```

-Which we've used to ensure both **U format** instructions are accurately chosen. A switch case in our `u_type` function exists based upon the `opcode` bits.

4.7.1 auipc

`auipc` takes the destination register and sets it equal to the `pc` plus the `immediate << 12`.

Case 1: Positive Immediate

```
li s1, 0xFFFFFFFF
auipc s1, 0xEFDCB
```

Expected Result: `s1 = 0xEFDCB000 + pc`

Case 2: Negative Immediate

```
li s1, 0x10101010
auipc s1, 0xF0F0F
```

Expected Result: `s1 = 0xF0F0F000 + pc`

Case 3: Immediate Value of 0

```
li s1, 0x22222222
auipc s1, 0x00000
```

Expected Result: `s1 = pc`

4.7.2 `lui`

`lui` takes the destination register and sets it equal to the `immediate << 12`.

Case 1: Positive Immediate

```
li s1, 0xFFFFFFFF
lui s1, 0xEFDCB
```

Expected Result: `s1 = 0xEFDCB000`

Case 2: Negative Immediate

```
li s1, 0x10101010
lui s1, 0xF0F0F
```

Expected Result: `s1 = 0xF0F0F000`

Case 3: Immediate Value of 0

```
li s1, 0x22222222
lui s1, 0x00000
```

Expected Result: `s1 = 0x00000000`

4.8 Jump Format

Jump Format contains only one instruction, `jal(0b1101111)`. This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rd` and `immediate`.

We ensured that this mode was correctly chosen through the use of our debugging mode and the print statement-

```
#ifdef DEBUG
fprintf(stderr, "J-Type instruction breakdown:\n    Opcode: 0x%02X\n
R_Des: 0x%02X\n    Immediate: 0x%06X\n", opcode, rd, imm);
#endif
```

-Which we've used to ensure the **J format** instruction is accurately chosen.

4.8.1 `jal`

`jal` takes the destination register and sets it equal to the `pc + 4`, then increments the `pc` by the value of the `immediate`.

Case 1: Positive Jump

`jal s1, 24`

Expected Result: `s1 = pc + 4`, `pc += 24`

Case 2: Negative Jump

`jal s1, -24`

Expected Result: `s1 = pc + 4`, `pc -= 24`

Case 3: Loop

`jal x0, 0`

Expected Result: `pc += 0`

Case 4: Maximum Jump Forward

`jal s1, 2^20 - 1`

Expected Result: `s1 = pc + 4`, `pc +=`

Case 5: Maximum Jump Backward

`jal s1, -2^20`

Expected Result: `s1 = pc + 4`, `pc +=`

5 Extra Credit

5.1 RV32M Multiply Extension

The multiply instructions are built off our previous register format switch case based on the `func3` bits. Then, we added nested switch cases based on the `func7` bits to select the correct function.

5.1.1 `mul`

The `mul` instruction is selected when `func3` is `0x0` and `func7` is `0x01`.

`mul-`

Case 1: Positive by Positive (small #'s) (positive sign extension)

```
addi s1, x0, 5
addi s2, x0, 5
mul  s3, s1, s2
```

Expected Result: `s3 = 25`

Case 2: Positive by Negative (small #'s) (negative sign extension)

```
addi s1, x0, 5
addi s2, x0, -5
mul  s3, s1, s2
```

Expected Result: `s3 = -25`

Case 3: Negative by Positive (small #'s) (negative sign extension)

```
addi s1, x0, -5
addi s2, x0, 5
mul  s3, s1, s2
```

Expected Result: `s3 = -25`

Case 4: Negative by Negative (small #'s) (positive sign extension)

```
addi s1, x0, -5
addi s2, x0, -5
mul  s3, s1, s2
```

Expected Result: `s3 = 25`

Case 5: Multiplication by 0

```
addi s1, x0, -5
addi s2, x0, 0
mul  s3, s1, s2
```

Expected Result: `s3 = 0`

Case 6: "Overflow"

```
addi s1, x0, 0x00000004
addi s2, x0, 0x7FFFFFFF
mul  s3, s1, s2
```

Expected Result: `s3 = 0xFFFFFFFC`

5.1.2 mulh

The `mulh` instruction is selected when `func3` is `0x1` and `func7` is `0x01`.

mulh-

Case 1: Positive by Positive (positive sign extension)

```
addi s1, x0, 0x7FFFFFFF
addi s2, x0, 0x00000010
mulh  s3, s1, s2
```

Expected Result: `s3 = 0x00000007`

Case 2: Positive by Negative (negative sign extension)

```
addi s1, x0, 0x7FFFFFFF
addi s2, x0, 0xFFFFFFF0
mulh  s3, s1, s2
```

Expected Result: `s3 = 0xFFFFFFF8`

Case 3: Negative by Positive (negative sign extension)

```
addi s1, x0, 0x80000001
addi s2, x0, 0x00000010
mulh  s3, s1, s2
```

Expected Result: `s3 = 0xFFFFFFF8`

Case 4: Negative by Negative (positive sign extension)

```
addi s1, x0, 0xFFFFFFFF
addi s2, x0, 0xFFFFFFFF
mulh  s3, s1, s2
```

Expected Result: `s3 = 0x0000000`

Case 5: Multiplication by 0

```
addi s1, x0, 0x7FFFFFFF
addi s2, x0, 0
mulh  s3, s1, s2
```

Expected Result: `s3 = 0`

5.1.3 `mulhsu`

The `mulhs u` instruction is selected when `func3` is `0x2` and `func7` is `0x01`.

`mulhsu-`

Case 1: Positive by Positive (actually positive) (positive sign extension)

```
addi s1, x0, 0x7FFFFFFF
addi s2, x0, 0x7FFFFFFF
mulhsu  s3, s1, s2
```

Expected Result: `s3 = 0x3FFFFFFF`

Case 2: Positive by Positive (actually negative) (positive sign extension)

```
addi s1, x0, 0x7FFFFFFF
addi s2, x0, 0xFFFFFFFF
mulhsu  s3, s1, s2
```

Expected Result: `s3 = 0x7FFFFFFE`

Case 3: Negative by Positive (actually positive) (negative sign extension)

```
addi s1, x0, 0xF0000000
addi s2, x0, 0x7FFFFFFF
mulhsu  s3, s1, s2
```

Expected Result: `s3 = 0xF8000000`

Case 4: Negative by Positive (actually negative) (negative sign extension)

```
addi s1, x0, 0xF0000000
addi s2, x0, 0xFFFFFFFF
mulhsu  s3, s1, s2
```

Expected Result: `s3 = 0xF0000000`

Case 5: Multiplication by 0

```
addi s1, x0, 0xFFFFFFFF
addi s2, x0, 0
mulhsu  s3, s1, s2
```

Expected Result: `s3 = 0`

5.1.4 mulhu

The `mulh u` instruction is selected when `func3` is `0x3` and `func7` is `0x01`.

mulhu-

Case 1: Positive (actually positive) by Positive (actually positive)

```
addi s1, x0, 0x7FFFFFFF
addi s2, x0, 0x7FFFFFFF
mulhu  s3, s1, s2
```

Expected Result: `s3 = 0x3FFFFFFF`

Case 2: Positive (actually positive) by Positive (actually negative)

```
addi s1, x0, 0x7FFFFFFF
addi s2, x0, 0xFFFFFFFF
mulhu  s3, s1, s2
```

Expected Result: `s3 = 0x7FFFFFFE`

Case 3: Positive (actually negative) by Positive (actually positive)

```
addi s1, x0, 0xFFFFFFFF
addi s2, x0, 0x07FFFFFF
mulhu  s3, s1, s2
```

Expected Result: `s3 = 0x07FFFFFE`

Case 4: Positive (actually negative) by Positive (actually negative)

```
addi s1, x0, 0xFFFFFFFF
addi s2, x0, 0xFFFFFFFF
mulhu  s3, s1, s2
```

Expected Result: `s3 = 0xFFFFFFFE`

Case 5: Multiplication by 0

```
addi s1, x0, 0xFFFFFFFF
addi s2, x0, 0
mulhu  s3, s1, s2
```

Expected Result: `s3 = 0`

5.1.5 `div`

The `div` instruction is selected when `func3` is `0x4` and `func7` is `0x01`.

`div-`

Case 1: Positive by Positive

```
li  s1, 10
li  s2, 5
div s3, s1, s2
```

Expected Result: `s3 = 2`

Case 2: Positive by Negative

```
li  s1, 10000
li  s2, -50
div s3, s1, s2
```

Expected Result: `s3 = -200`

Case 3: Negative by Positive

```
li  s1, -1024
li  s2, 64
div s3, s1, s2
```

Expected Result: `s3 = -16`

Case 4: Negative by Negative

```
li  s1, -56230
li  s2, -11246
div s3, s1, s2
```

Expected Result: `s3 = 5`

Case 5: Divide by 1

```
li  s1, 250
li  s2, 1
div s3, s1, s2
```

Expected Result: `s3 = 250`

Case 6: Divide by -1

```
li  s1, 250
li  s2, -1
div s3, s1, s2
```

Expected Result: `s3 = -250`

Case 7: Divide by 0

```
li  s1, 7
li  s2, 0
div s3, s1, s2
```

Expected Result: `Error (raise exception)`

Case 8: Rounding

```
li  s1, 7
li  s2, 3
div s3, s1, s2
```

Expected Result: `s3 = 2`

5.1.6 `divu`

The `divu` instruction is selected when `func3` is `0x5` and `func7` is `0x01`.

`divu-`

Case 1: Positive by Positive

```
li  s1, 10
li  s2, 5
divu s3, s1, s2
```

Expected Result: `s3 = 2`

Case 2: Positive by Negative

```
li  s1, 10000
li  s2, -1
divu s3, s1, s2
```

Expected Result: `s3 = 0`

Case 3: Negative by Positive

```
li  s1, -1024
li  s2, 64
divu s3, s1, s2
```

Expected Result: `s3 = 67108848`

Case 4: Negative by Negative

```
li  s1, -56230
li  s2, -11246
divu s3, s1, s2
```

Expected Result: `s3 = 0`

Case 5: Divide by 1

```
li  s1, 250
li  s2, 1
divu s3, s1, s2
```

Expected Result: `s3 = 250`

Case 6: Divide by -1

```
li  s1, 250
li  s2, -1
divu s3, s1, s2
```

Expected Result: `s3 = 0`

Case 7: Divide by 0

```
li  s1, 7
li  s2, 0
divu s3, s1, s2
```

Expected Result: `Error (raise exception)`

Case 8: Rounding

```
li  s1, 7
li  s2, 3
divu s3, s1, s2
```

Expected Result: `s3 = 2`

5.1.7 rem

The `rem` instruction is selected when `func3` is `0x6` and `func7` is `0x01`.

rem-

Case 1: Positive by Positive

```
li  s1, 6
li  s2, 4
rem s3, s1, s2
```

Expected Result: `s3 = 2`

Case 2: Positive by Negative

```
li  s1, 80
li  s2, -12
rem s3, s1, s2
```

Expected Result: `s3 = 8`

Case 3: Negative by Positive

```
li  s1, -400
li  s2, 100
rem s3, s1, s2
```

Expected Result: `s3 = 0`

Case 4: Negative by Negative

```
li  s1, -86
li  s2, -3
rem s3, s1, s2
```

Expected Result: `s3 = -2`

Case 5: Rem 1

```
li  s1, 45454
li  s2, 1
rem s3, s1, s2
```

Expected Result: `s3 = 0`

Case 6: Rem -1

```
li  s1, 424232323
li  s2, -1
rem s3, s1, s2
```

Expected Result: `s3 = 0`

Case 7: Rem 0

```
li  s1, 6996420
li  s2, 0
rem s3, s1, s2
```

Expected Result: `error (raise exception)`

5.1.8 `remu`

The `remu` instruction is selected when `func3` is `0x7` and `func7` is `0x01`.

`remu-`

Case 1: Positive by Positive

```
li  s1, 6
li  s2, 4
remu s3, s1, s2
```

Expected Result: `s3 = 2`

Case 2: Positive by Negative

```
li  s1, 800000
li  s2, -12
remu s3, s1, s2
```

Expected Result: `s3 = 80000`

Case 3: Negative by Positive

```
li  s1, -400
li  s2, 100
remu s3, s1, s2
```

Expected Result: `s3 = 96`

Case 4: Negative by Negative

```
li  s1, -3
li  s2, -9
remu s3, s1, s2
```

Expected Result: `s3 = 6`

Case 5: Rem 1

```
li  s1, 45454
li  s2, 1
remu s3, s1, s2
```

Expected Result: `s3 = 0`

Case 6: Rem -1

```
li  s1, 424232323
li  s2, -1
remu s3, s1, s2
```

Expected Result: `s3 = 424232323`

Case 7: Rem 0

```
li  s1, 6996420
li  s2, 0
remu s3, s1, s2
```

Expected Result: `error (raise exception)`

5.2 RV32F / D Floating-Point Extensions

Floating Mode instructions are all sorted by their opcodes (`0b0000111`, `0b0100111`, `0b1000011`, `0b1000111`, `0b1001011`, `0b1001111`, `0b1010011`) into the functions, `flt_round`, `f1_type`, `f2_type`, `f3_type`. This function breaks down instructions and functions as part of our decode, by masking and shifting the instruction into `rd`, `rm`, `func3`, `rs1`, `rs2`, `func7`.

The correct function is chosen based on the opcode, and then cases are selected based on the function bits containing the correct operations for the given instruction.

5.2.1 `flw`

Case 1: Correct reference (byte 1 of 4)

```
flw f1, x0, 16
(Mem 16 = 0xF0F0D0D0)
```

Expected Result: `f1 = F0F0D0D0`

Case 2: Misaligned reference (byte 2 of 4)

```
flw f1, x0, 17
```

Expected Result: `Misaligned`

Case 3: Misaligned reference (byte 3 of 4)

```
flw f1, x0, 18
```

Expected Result: `Misaligned`

Case 4: Misaligned reference (byte 4 of 4)

```
flw f1, x0, 19
```

Expected Result: `Misaligned`

5.2.2 `fsw`

Case 1: Correct Reference (byte 1 of 4)

```
0: fld f2, memory containing(0xFFEEDDCC)
4: fsw f2, 80(x0)
8: jr 0
```

Expected Result: `mem(80) = 0xFFEEDDCC`

Case 2: Misaligned Reference (byte 2 of 4)

```
0: fld f2, memory containing(0xFFEEDDCC)
4: fsw f2, 81(x0)
8: jr 0
```

Expected Result: `error(1)`

Case 3: Misaligned Reference (byte 3 of 4)

```
0: fld f2, memory containing(0xFFEEDDCC)
4: fsw f2, 82(x0)
8: jr 0
```

Expected Result: `error(1)`

Case 4: Misaligned Reference (byte 4 of 4)

```
0: fld, f2, memory containing(0xFFEEDDCC)
4: fsw  f2, 83(x0)
8: jr 0
```

Expected Result: `error(1)`

5.2.3 `fmadd.s`

Case 1: All Positive

```
li a0, 2
li a1, 2
li a2, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 6`

Case 2: Multiply Positive Add Negative

```
li a0, 2
li a1, 2
li a2, -2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.w  f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 2`

Case 3: Multiply Positive & Negative (way 1) Add Negative

```
li a0, 4
li a1, -4
li a2, -4
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -20`

Case 4: Multiply Positive & Negative (way 1) Add Positive

```
li a0, 4
li a1, -4
li a2, 4
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -12`

Case 5: Multiply Positive & Negative (way 2) Add Negative

```
li a0, -4
li a1, 4
li a2, -4
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -20`

Case 6: Multiply Positive & Negative (way 2) Add Positive

```
li a0, -4
li a1, 4
li a2, 4
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -12`

Case 7: Multiply Negative Add Negative

```
li a0, -2
li a1, -2
li a2, -2
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 2`

Case 8: Multiply Negative Add Positive

```
li a0, -2
li a1, -2
li a2, 2
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 6`

Case 9: Multiply Zero Add Positive

```
li a0, 0
li a1, 5
li a2, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 2`

Case 10: Multiply Zero Add Negative

```
li a0, 6
li a1, 0
li a2, -68
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -68`

Case 11: Multiply Zero Add Zero

```
li a0, 0
li a1, 12
li a2, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 0`

5.2.4 `fmsub.s`

Case 1: All Positive

```
li a0, 2
li a1, 2
li a2, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 2`

Case 2: Multiply Positive Sub Negative

```
li a0, 2
li a1, 2
li a2, -2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.w  f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 6`

Case 3: Multiply Positive & Negative (way 1) Sub Negative

```
li a0, 4
li a1, -4
li a2, -4
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -12`

Case 4: Multiply Positive & Negative (way 1) Sub Positive

```
li a0, 4
li a1, -4
li a2, 4
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -20`

Case 5: Multiply Positive & Negative (way 2) Sub Negative

```
li a0, -4
li a1, 4
li a2, -4
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -12`

Case 6: Multiply Positive & Negative (way 2) Sub Positive

```
li a0, -4
li a1, 4
li a2, 4
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -20`

Case 7: Multiply Negative Sub Negative

```
li a0, -2
li a1, -2
li a2, -2
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 6`

Case 8: Multiply Negative Sub Positive

```
li a0, -2
li a1, -2
li a2, 2
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 2`

Case 9: Multiply Zero Sub Positive

```
li a0, 0
li a1, 5
li a2, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -2`

Case 10: Multiply Zero Sub Negative

```
li a0, 6
li a1, 0
li a2, -68
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 68`

Case 11: Multiply Zero Sub Zero

```
li a0, 0
li a1, 12
li a2, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 0`

5.2.5 `fnmadd.s`

Case 1: All Positive

```
li a0, 2
li a1, 2
li a2, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -2`

Case 2: Multiply Positive Add Negative

```
li a0, 2
li a1, 2
li a2, -2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.w  f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -6`

Case 3: Multiply Positive & Negative (way 1) Add Negative

```
li a0, 4
li a1, -4
li a2, -4
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 12`

Case 4: Multiply Positive & Negative (way 1) Add Positive

```
li a0, 4
li a1, -4
li a2, 4
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 20`

Case 5: Multiply Positive & Negative (way 2) Add Negative

```
li a0, -4
li a1, 4
li a2, -4
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 12`

Case 6: Multiply Positive & Negative (way 2) Add Positive

```
li a0, -4
li a1, 4
li a2, 4
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 20`

Case 7: Multiply Negative Add Negative

```
li a0, -2
li a1, -2
li a2, -2
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -6`

Case 8: Multiply Negative Add Positive

```
li a0, -2
li a1, -2
li a2, 2
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -2`

Case 9: Multiply Zero Add Positive

```
li a0, 0
li a1, 5
li a2, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 2`

Case 10: Multiply Zero Add Negative

```
li a0, 6
li a1, 0
li a2, -68
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = -68`

Case 11: Multiply Zero Add Zero

```
li a0, 0
li a1, 12
li a2, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmadd.s f4, f1, f2, f3
```

Expected Result: `f4 = 0`

5.2.6 fnmsub.s

Case 1: All Positive

```
li a0, 2
li a1, 2
li a2, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -6`

Case 2: Multiply Positive Sub Negative

```
li a0, 2
li a1, 2
li a2, -2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.w  f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -2`

Case 3: Multiply Positive & Negative (way 1) Sub Negative

```
li a0, 4
li a1, -4
li a2, -4
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 20`

Case 4: Multiply Positive & Negative (way 1) Sub Positive

```
li a0, 4
li a1, -4
li a2, 4
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 12`

Case 5: Multiply Positive & Negative (way 2) Sub Negative

```
li a0, -4
li a1, 4
li a2, -4
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 20`

Case 6: Multiply Positive & Negative (way 2) Sub Positive

```
li a0, -4
li a1, 4
li a2, 4
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 12`

Case 7: Multiply Negative Sub Negative

```
li a0, -2
li a1, -2
li a2, -2
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -2`

Case 8: Multiply Negative Sub Positive

```
li a0, -2
li a1, -2
li a2, 2
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -6`

Case 9: Multiply Zero Sub Positive

```
li a0, 0
li a1, 5
li a2, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = -2`

Case 10: Multiply Zero Sub Negative

```
li a0, 6
li a1, 0
li a2, -68
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.w f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 68`

Case 11: Multiply Zero Sub Zero

```
li a0, 0
li a1, 12
li a2, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, a2
fcvt.s.wu f4, x0
fnmsub.s f4, f1, f2, f3
```

Expected Result: `f4 = 0`

5.2.7 fadd.s

Case 1: Positive to Positive

```
li a0, 4
li a1, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fadd.s f3, f1, f2
```

Expected Result: `f3 = 6`

Case 2: Positive to Negative (Way 1)

```
li a0, 4
li a1, -2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fadd.s f3, f1, f2
```

Expected Result: `f3 = 2`

Case 3: Positive to Negative (Way 2)

```
li a0, -4
li a1, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fadd.s f3, f1, f2
```

Expected Result: `f3 = -2`

Case 4: Negative to Negative

```
li a0, -4
li a1, -2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fadd.s f3, f1, f2
```

Expected Result: `f3 = -6`

Case 5: Add 0

```
li a0, 10
li a1, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fadd.s f3, f1, f2
```

Expected Result: `f3 = 10`

5.2.8 `fsub.s`

Case 1: Positive from Positive

```
li a0, 5
li a1, 4
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsub.s f3, f1, f2
```

Expected Result: `f3 = 1`

Case 2: Positive from Negative (Way 1)

```
li a0, 5
li a1, -4
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsub.s f3, f1, f2
```

Expected Result: `f3 = 9`

Case 3: Positive from Negative (Way 2)

```
li a0, -5
li a1, 4
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsub.s f3, f1, f2
```

Expected Result: `f3 = -9`

Case 4: Negative from Negative

```
li a0, -8
li a1, -10
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsub.s f3, f1, f2
```

Expected Result: `f3 = 2`

Case 5: Sub 0

```
li a0, 66
li a1, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsub.s f3, f1, f2
```

Expected Result: `f3 = 66`

5.2.9 `fmul.s`

Case 1: Positive by Positive

```
li a0, 6
li a1, 6
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmul.s f3, f1, f2
```

Expected Result: `f3 = 36`

Case 2: Positive by Negative (Way 1)

```
li a0, 6
li a1, -9
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmul.s f3, f1, f2
```

Expected Result: `f3 = -54`

Case 3: Positive by Negative (Way 2)

```
li a0, -9
li a1, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmul.s f3, f1, f2
```

Expected Result: `f3 = -18`

Case 4: Negative by Negative

```
li a0, -5
li a1, -9
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmul.s f3, f1, f2
```

Expected Result: `f3 = 45`

Case 5: Mul 0

```
li a0, 33333
li a1, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmul.s f3, f1, f2
```

Expected Result: `f3 = 0`

5.2.10 `fdiv.s`

Case 1: Positive by Positive

```
li a0, 6
li a1, 6
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fdiv.s f3, f1, f2
```

Expected Result: `f3 = 1`

Case 2: Positive by Negative (Way 1)

```
li a0, 6
li a1, -2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fdiv.s f3, f1, f2
```

Expected Result: `f3 = -3`

Case 3: Positive by Negative (Way 2)

```
li a0, -9
li a1, 2
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fdiv.s f3, f1, f2
```

Expected Result: `f3 = -4.5`

Case 4: Negative by Negative

```
li a0, -5
li a1, -9
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fdiv.s f3, f1, f2
```

Expected Result: `f3 = 0.55555`

Case 5: Div 0

```
li a0, 33333
li a1, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fdiv.s f3, f1, f2
```

Expected Result: `error(1)`

Case 6: Div 1

```
li a0, 505
li a1, 1
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fdiv.s f3, f1, f2
```

Expected Result: `f3 = 505`

Case 7: Div -1

```
li a0, 432
li a1, -1
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fdiv.s f3, f1, f2
```

Expected Result: `f3 = -432`

5.2.11 `fsqrt.s`

Case 1: Evenly Roots

```
li a0, 9
fcvt.s.wu f1, a0
fcvt.s.wu f2, x0
fsqrt.s f2, f1
```

Expected Result: `f2 = 3`

Case 2: Unevenly Roots

```
li a0, 5
fcvt.s.wu f1, a0
fcvt.s.wu f2, x0
fsqrt.s f2, f1
```

Expected Result: `f2 = 2.2360679775`

Case 3: 0

```
li a0, 0
fcvt.s.wu f1, a0
fcvt.s.wu f2, x0
fsqrt.s f2, f1
```

Expected Result: `f2 =` ∞

Case 4: 1

```
li a0, 1
fcvt.s.wu f1, a0
fcvt.s.wu f2, x0
fsqrt.s f2, f1
```

Expected Result: `f2 = 1`

Case 5: Large Case

```
li a0, 90000
fcvt.s.wu f1, a0
fcvt.s.wu f2, x0
fsqrt.s f2, f1
```

Expected Result: `f2 = 300`

Case 6: Negative Number

```
li a0, -90000
fcvt.s.wu f1, a0
fcvt.s.wu f2, x0
fsqrt.s f2, f1
```

Expected Result: `f2 = NAN`

5.2.12 `fsgnj.s`

Case 1: Both Positive

```
li a0, 85000
li a1, 1
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsgnj.s f3, f1, f2
```

Expected Result: `f3 = 85000`

Case 2: Positive & Negative (Way 1)

```
li a0, 85000
li a1, -1
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, x0
fsgnj.s f3, f1, f2
```

Expected Result: `f3 = -85000`

Case 3: Positive & Negative (Way 2)

```
li a0, -85000
li a1, 1
fcvt.s.w  f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsgnj.s f3, f1, f2
```

Expected Result: `f3 = 85000`

Case 4: Both Negative

```
li a0, -85000
li a1, -1
fcvt.s.w  f1, a0
fcvt.s.w  f2, a1
fcvt.s.wu f3, x0
fsgnj.s f3, f1, f2
```

Expected Result: `f3 = -85000`

5.2.13 fsgnjn.s

Case 1: Both Positive

```
li a0, 85000
li a1, 1
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsgnjn.s f3, f1, f2
```

Expected Result: `f3 = -85000`

Case 2: Positive & Negative (Way 1)

```
li a0, 85000
li a1, -1
fcvt.s.wu f1, a0
fcvt.s.w  f2, a1
fcvt.s.wu f3, x0
fsgnjn.s f3, f1, f2
```

Expected Result: `f3 = 85000`

Case 3: Positive & Negative (Way 2)

```
li a0, -85000
li a1, 1
fcvt.s.w  f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsgnjn.s f3, f1, f2
```

Expected Result: `f3 = -85000`

Case 4: Both Negative

```
li a0, -85000
li a1, -1
fcvt.s.w  f1, a0
fcvt.s.w  f2, a1
fcvt.s.wu f3, x0
fsgnjn.s f3, f1, f2
```

Expected Result: `f3 = 85000`

5.2.14 `fsgnjx.s`

Case 1: Both Positive

```
li a0, 1000
li a1, 25
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsgnjx.s f3, f1, f2
```

Expected Result: `f3 = 1000`

Case 2: Positive & Negative (Way 1)

```
li a0, 1000
li a1, -1000
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, x0
fsgnjx.s f3, f1, f2
```

Expected Result: `f3 = -1000`

Case 3: Positive & Negative (Way 2)

```
li a0, -1000
li a1, 1000
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fsgnjx.s f3, f1, f2
```

Expected Result: `f3 = -1000`

Case 4: Both Negative

```
li a0, -1000
li a1, -1000
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, x0
fsgnjx.s f3, f1, f2
```

Expected Result: `f3 = 1000`

5.2.15 `fmin.s`

Case 1: Both Positive (rs1 larger)

```
li a0, 25000
li a1, 3500
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmin.s f3, f1, f2
```

Expected Result: `f3 = 3500`

Case 2: Both Positive (rs2 larger)

```
li a0, 5767
li a1, 5777
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmin.s f3, f1, f2
```

Expected Result: `f3 = 5767`

Case 3: Positive & Negative (rs1 larger)

```
li a0, 1
li a1, -200
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmin.s f3, f1, f2
```

Expected Result: `f3 = -200`

Case 4: Positive & Negative (rs2 larger)

```
li a0, -86000
li a1, 8323
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmin.s f3, f1, f2
```

Expected Result: `f3 = -86000`

Case 5: Both Negative (rs1 larger)

```
li a0, -1
li a1, -5
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, x0
fmin.s f3, f1, f2
```

Expected Result: `f3 = -5`

Case 6: Both Negative (rs2 larger)

```
li a0, -50320
li a1, -542
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, x0
fmin.s f3, f1, f2
```

Expected Result: `f3 = -50320`

Case 7: Zero w/ Positive

```
li a0, 0
li a1, 100
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmin.s f3, f1, f2
```

Expected Result: `f3 = 0`

Case 8: Zero w/ Negative

```
li a0, 0
li a1, -100
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmin.s f3, f1, f2
```

Expected Result: `f3 = -100`

5.2.16 `fmax.s`

Case 1: Both Positive (rs1 larger)

```
li a0, 25000
li a1, 903
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmax.s f3, f1, f2
```

Expected Result: `f3 = 25000`

Case 2: Both Positive (rs2 larger)

```
li a0, 502
li a1, 54602
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmax.s f3, f1, f2
```

Expected Result: `f3 = 54602`

Case 3: Positive & Negative (rs1 larger)

```
li a0, 5
li a1, -1
fcvt.s.wu f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, x0
fmax.s f3, f1, f2
```

Expected Result: `f3 = 5`

Case 4: Positive & Negative (rs2 larger)

```
li a0, -92
li a1, 1
fcvt.s.w f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmax.s f3, f1, f2
```

Expected Result: `f3 = 1`

Case 5: Both Negative (rs1 larger)

```
li a0, -232
li a1, -990
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, x0
fmax.s f3, f1, f2
```

Expected Result: `f3 = -232`

Case 6: Both Negative (rs2 larger)

```
li a0, -664542
li a1, -920
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.wu f3, x0
fmax.s f3, f1, f2
```

Expected Result: `f3 = -920`

Case 7: Zero w/ Positive

```
li a0, 0
li a1, 3
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmax.s f3, f1, f2
```

Expected Result: `f3 = 3`

Case 8: Zero w/ Negative

```
li a0, 0
li a1, -10
fcvt.s.wu f1, a0
fcvt.s.wu f2, a1
fcvt.s.wu f3, x0
fmax.s f3, f1, f2
```

Expected Result: `f3 = 0`

5.2.17 `fcvt.s.w`

Case 1: Zero

```
li a0, 0
fcvt.s.w f1, a0
```

Expected Result: `f1 = 0`

Case 2: Negative

```
li a0, -25000
fcvt.s.w f1, a0
```

Expected Result: `f1 = -25000`

Case 3: Positive

```
li a0, 25000
fcvt.s.w f1, a0
```

Expected Result: `f1 = 25000`

5.2.18 `fcvt.s.wu`

Case 1: Zero

```
li a0, 0
fcvt.s.wu f1, a0
```

Expected Result: `f1 = 0`

Case 2: Negative

```
li a0, -25000
fcvt.s.wu f1, a0
```

Expected Result: `f1 = 4294942296`

Case 3: Positive

```
li a0, 25000
fcvt.s.wu f1, a0
```

Expected Result: `f1 = 25000`

5.2.19 `fcvt.w.s`

Case 1: Zero

```
li a0, 0
fcvt.s.w f1, a0
li a0, 94203
fcvt.w.s a0, f1
```

Expected Result: `a0 = 0`

Case 2: Negative

```
li a0, -5000
fcvt.s.w f1, a0
li a0, 94203
fcvt.w.s a0, f1
```

Expected Result: `a0 = -5000`

Case 3: Positive

```
li a0, 42321
fcvt.s.w f1, a0
li a0, 94203
fcvt.w.s a0, f1
```

Expected Result: `a0 = 42321`

5.2.20 `fcvt.wu.s`

Case 1: Zero

```
li a0, 0
fcvt.s.w f1, a0
li a0, 94203
fcvt.wu.s a0, f1
```

Expected Result: `a0 = 0`

Case 2: Negative

```
li a0, -50230
fcvt.s.w f1, a0
li a0, 94203
fcvt.wu.s a0, f1
```

Expected Result: `a0 = 12218` (Unsure about this result)

Case 3: Positive

```
li a0, 5656
fcvt.s.w f1, a0
li a0, 94203
fcvt.wu.s a0, f1
```

Expected Result: `a0 = 5656`

5.2.21 `fmv.x.w`

Case 1: Positive (w/o Rounding)

```
li a0, 55
fcvt.s.w f1, a0
li a0, 0
fmv.x.w  a0, f1
```

Expected Result: `a0 = 1113325568`

Case 2: Positive (w/ Rounding)

```
li a0, 55
li a1, 10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fdiv.s   f3, f1, f2
li a0, 0
fmv.x.w  a0, f3
```

Expected Result: `a0 = 1085276160`

Case 3: Negative (w/o Rounding)

```
li a0, -55
fcvt.s.w f1, a0
li a0, 0
fmv.x.w  a0, f1
```

Expected Result: `a0 = 3260809216`

Case 4: Negative (w/ Rounding)

```
li a0, 55
li a1, -10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fdiv.s   f3, f1, f2
li a0, 0
fmv.x.w  a0, f3
```

Expected Result: `a0 = 3232759808`

Case 5: Zero

```
li a0, 0
fcvt.s.w f1, a0
li a0, 2025
fmv.x.w  a0, f1
```

Expected Result: `a0 = 0`

5.2.22 `fmv.w.x`

Case 1: Positive (w/o Rounding)

```
li a0, 55
fmv.w.x  f1, a0
```

Expected Result: `f1 = 7.7e-44`

Case 2: Negative (w/o Rounding)

```
li a0, -55
fmv.w.x  f1, a0
```

Expected Result: `f1 = 0xFFFFFFC9`

Case 3: Zero

```
li a0, 0
fmv.w.x  f1, a0
```

Expected Result: `f1 = 0`

5.2.23 `feq.s`

Case 1: Equal

```
li a0, -10
li a1, -10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
feq.s a0, f1, f2
```

Expected Result: `f3 = 1`

Case 2: Greater Than

```
li a0, 10
li a1, -10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
feq.s a0, f1, f2
```

Expected Result: `f3 = 0`

Case 3: Less Than

```
li a0, -10
li a1, 10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
feq.s a0, f1, f2
```

Expected Result: `f3 = 0`

Case 4: NAN

```
li a0, 10
li a1, -10
li a2, 0
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.w f4, a2
fdiv.s f1, f1, f4
feq.s a0, f1, f2
```

Expected Result: `f3 = 0 (Invalid Operand Exception)`

5.2.24 `flt.s`

Case 1: Equal

```
li a0, -10
li a1, -10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
flt.s a0, f1, f2
```

Expected Result: `f3 = 0`

Case 2: Greater Than

```
li a0, 10
li a1, -10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
flt.s a0, f1, f2
```

Expected Result: `f3 = 0`

Case 3: Less Than

```
li a0, -10
li a1, 10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
flt.s a0, f1, f2
```

Expected Result: `f3 = 1`

Case 4: NAN

```
li a0, 10
li a1, -10
li a2, 0
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.w f4, a2
fdiv.s f1, f1, f4
flt.s a0, f1, f2
```

Expected Result: `f3 = 0 (Invalid Operand Exception)`

5.2.25 `fle.s`

Case 1: Equal

```
li a0, -10
li a1, -10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fle.s a0, f1, f2
```

Expected Result: `f3 = 1`

Case 2: Greater Than

```
li a0, 10
li a1, -10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fle.s a0, f1, f2
```

Expected Result: `f3 = 0`

Case 3: Less Than

```
li a0, -10
li a1, 10
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fle.s a0, f1, f2
```

Expected Result: `f3 = 1`

Case 4: NAN

```
li a0, 10
li a1, -10
li a2, 0
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.w f4, a2
fdiv.s f1, f1, f4
fle.s a0, f1, f2
```

Expected Result: `f3 = 0 (Invalid Operand Exception)`

5.2.26 fclass.s

Case 1: -∞

```
li a0, 10
li a1, 0
li a2, -1
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fcvt.s.w f3, a2
fdiv.s f1, f1, f2
fmul.s f1, f1, f3
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000001`

Case 2: - Normal Number

```
li a0, -10
fcvt.s.w f1, a0
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000002`

Case 3: - Subnormal Number

```
li a0, 2147483649
fmv.w.x f1, a0
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000004`

Case 4: -0

```
li a0, 0
li a1, -1
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fmul.s f1, f1, f2
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000008`

Case 5: +0

```
li a0, 0
li a1, 1
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fmul.s f1, f1, f2
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000010`

Case 6: + Subnormal Number

```
li a0, 1
fmv.w.x f1, a0
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000020`

Case 7: + Normal Number

```
li a0, 10
fcvt.s.w f1, a0
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000040`

Case 8: $+\infty$

```
li a0, 10
li a1, 0
fcvt.s.w f1, a0
fcvt.s.w f2, a1
fdiv.s f1, f1, f2
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000080`

Case 9: Signaling NAN

```
li a0, -1
fcvt.s.w f1, a0
fsqrt.s f1, f1
fclass.s a0, f1
```

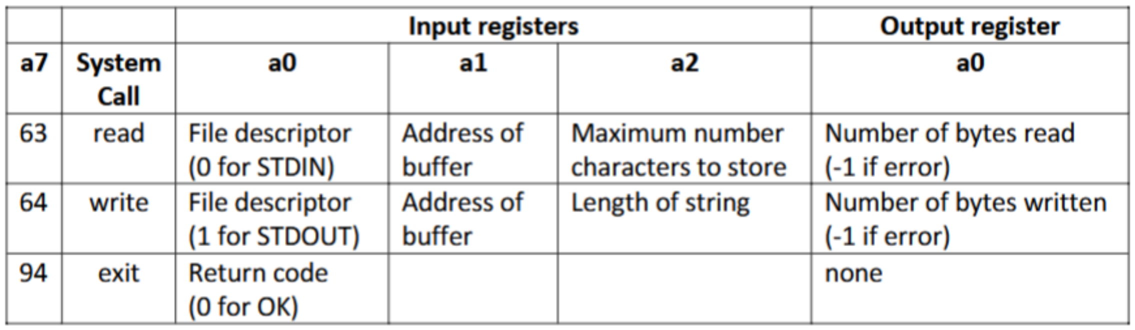
Expected Result: `a0 = 0x00000100`

Case 10: Quiet NAN

```
li a0, 0
fcvt.s.w f1, a0
fdiv.s f1, f1, f1
fclass.s a0, f1
```

Expected Result: `a0 = 0x00000200`

5.3 `ecall`



		Input registers			**Output register**
a7	**System Call**	**a0**	**a1**	**a2**	**a0**
63	read	File descriptor (0 for STDIN)	Address of buffer	Maximum number characters to store	Number of bytes read (-1 if error)
64	write	File descriptor (1 for STDOUT)	Address of buffer	Length of string	Number of bytes written (-1 if error)
94	exit	Return code (0 for OK)			none