Design Document: Functional Simulator for Subset of RISC-V instruction set

**Group Members:**

|  |  |
| --- | --- |
| ***Hardik Aggarwal*** | ***2021CSB1173*** |
| ***Komalpreet Singh*** | ***2021CSB1237*** |
| ***Patil Ritesh Mohan*** | ***2021CSB1120*** |
| ***Edgar Aditya Thorpe*** | ***2021CSB1169*** |

The document describes the design aspect of RISC-V, a functional simulator for subset of RISC-V instruction set.

# Input/Output

## Input

Input to the simulator is .mc file that contains the contents of Instruction memory and the Data memory separated by ‘0x7FFFFFFF’sign. For example:

0x0 E3A0200A

0x4 E3A0300A

0x8 E0821003

0xc 7FFFFFFF

## Functional Behavior and output

The simulator reads the instruction from instruction memory, decodes the instruction, reads the register, execute the operation, and write back to the register file. The instruction set supported is same as given in the lecture notes.

The execution of instruction continues till it reaches instruction “0x7FFFFFFF”. After this, the simulator stops and writes the updated memory contents on to a memory text file.

The simulator also prints messages for each cycle describing all the information regarding the tasks performed in each of the stages namely, Fetch, Decode, Execute, Memory Access and Write Back.

The **functions** used in the program are:

1. run\_riscvsim(): The main function that runs the simulation. It contains a loop that repeatedly fetches, decodes, executes, accesses memory, and writes back the results of each instruction.
2. reset\_proc(): Resets all registers and memory content to 0.
3. load\_program\_memory(): Reads the input memory and populates the instruction memory.
4. write\_data\_memory(): Writes the data memory to a file.
5. swi\_exit(): Called when the instruction is swi\_exit.
6. fetch(): Reads from the instruction memory and updates the instruction register.
7. decode(): Reads the instruction register, reads operand1 and operand2 from the register file, and decides the operation to be performed in the execute stage.
8. execute(): Performs the operation on the operands based on the decoded instruction.
9. mem(): Accesses memory if necessary.
10. write\_back(): Writes the result back to the register file if necessary.

The **variables** used in the program are:

1. instruction\_word: A variable representing the current instruction being executed.
2. operand1 and operand2: Variables representing the operands of the current instruction.
3. alu\_result: A variable representing the result of the current ALU operation.
4. pc and nextpc: Variables representing the current and next program counters.
5. clk: A variable representing the clock cycle of the simulation.

The **Arrays, maps and structs** used are:

1. X: An array representing the register file.
2. instruction\_memory: An unordered map representing the instruction memory.
3. MEM: An unordered map representing the data memory.
4. instruction: A struct representing the decoded instruction. It contains the opcode, source registers, destination register, immediate value, function code, instruction type, and instruction name.

# Design of Simulator

## Data structure

Registers, memories, intermediate output for each stage of instruction execution are declared as global static. Being static, the variables are not visible outside the file, thus, make the data encapsulated in the myARMSim.cpp.

## Simulator flow:

There are two steps:

1. First, the input file is loaded and the Instruction Memory and Data Memory are filled up with the instructions and data provided.
2. Secondly, the Simulator executes the instructions one by one.

The second step is executed by using a while loop. While the PC points to a valid instruction, the body of the while loop is executed otherwise the while loop is terminated.

The body of the while loop lines up the execution of the five stages of Instruction Execution i.e. Fetch, Decode, Execute, Memory Access and Write Back.

Now, we describe the implementation of fetch, decode, execute, memory, and write-back function.

## *FETCH*

The **fetch()** function is part of a RISC-V simulator program and its main purpose is to fetch an instruction from the instruction memory and set the **instruction\_word** variable to the value of the instruction at the current program counter (pc). The function also sets the **nextpc** variable to the value of the program counter incremented by 4, as each RISC-V instruction is 4 bytes in size.

The function first reads the instruction word from the instruction memory at the address specified by pc using the instruction\_memory map, which maps memory addresses to instruction words. It then checks if the fetched instruction is equal to 0xffffffff, which is used to signal the end of the program. If the instruction is equal to 0xffffffff, the program calls the **swi\_exit()** function, which writes the data memory to a file and terminates the program.

If there is no instruction at the current program counter, the function prints an error message and calls **swi\_exit()** to terminate the program.

The function then calls **set\_instruction\_bin()** to set the **instruction.instruction\_bin variable**, which is an array of characters representing the binary representation of the instruction word. Finally, the function prints a message to indicate that it has fetched the instruction.

## *DECODE*

The **decode()** function reads the instruction register and extracts different fields of the instruction, such as opcode, destination register, source registers, immediate value, function fields, and instruction type.

First, the function calls the **bin2dec()** function to extract the different fields from the instruction register. It extracts the opcode, rd (destination register), rs1 (source register 1), rs2 (source register 2), func3 (function field 3), and func7 (function field 7) fields of the instruction.

Then, the function determines the type of the instruction based on its opcode and the immediate is calculated if present. If the opcode is 51, the instruction is of type R (register-register) instruction. If the opcode is 19, 3, or 103, the instruction is of type I (immediate) instruction. If the opcode is 99, the instruction is of type SB (branch) instruction. If the opcode is 35, the instruction is of type S (store) instruction. If the opcode is 55, the instruction is of type U (upper immediate) instruction. If the opcode is 23, the instruction is of type UJ (jump) instruction.

The instruction type and other fields are stored in a global structure **instruction**.

## *EXECUTE*

The **execute()** function is used to perform the execution of the instruction fetched in the **fetch()** function and the decoded information from the instruction is used to perform arithmetic or logical operations on the data present in the register file or the memory.

The **execute()** function uses the opcode field of the **instruction\_set** struct and depending on the opcode, the operation to be performed is determined and executed using **if-else** conditions.

For example, for an **R** and **I** type instructions (excluding the **load**), the **execute()** function performs arithmetic or logical operations using the data present in the registers. Similarly, for a **S and load** type instruction, it stores the address that is the sum of **op1 and immediate** .

Except for **jal** instruction , in which the **nextpc** is updated, once the operation is performed, the result is stored in the **alu\_result** variable. This **alu\_result** is used by the **mem()** function to either store the result in the memory or fetch the data from the memory depending on the type of instruction.

## *MEMORY ACCESS*

The **mem()** It takes in an instruction and an ALU result and performs the appropriate memory access operation based on the instruction.

If the instruction's opcode is 3, it's a load instruction, and it performs one of three memory accesses based on the func3 field of the instruction. For func3=0 and func3=1, it retrieves data from memory at the ALU result, and extends the sign bit to 32 bits. Then, it converts the resulting number into binary and places it in an array of size 32. Finally, it converts the first 7 or 15 bits of the array into a 7 or 15-bit signed integer, and stores the result in the **MEM\_result** variable.

If the instruction's opcode is 35, it's a store instruction, and it performs one of three memory accesses based on the func3 field of the instruction. For func3=0 and func3=1, it retrieves data from memory at the ALU result, and extends the sign bit to 32 bits. It then retrieves data from register rs2, and extends the sign bit to 32 bits. The two numbers are concatenated and placed in an array of size 32, with the first 7 or 15 bits being filled with the rs2 data. Finally, the resulting binary number, with the first 31 bits being the concatenated value of **MEM[alu\_result]** and **X[instruction.rs2]**, is stored in memory at the ALU result. For func3=2, it stores the value of rs2 in memory at the ALU result.

## *WRITE BACK*

**The write\_back()** function that executes the writeback stage of a computer's instruction pipeline. The writeback stage is responsible for writing the result of the instruction execution back to the register file.

The function initializes a variable **wb\_result** that will hold the value to be written back to the register file. It also initializes a boolean variable **write** to 0, which will be set to 1 if a writeback operation needs to be performed.

The function then checks if the instruction's destination register **rd** is not 0 (x0 register is hardwired to 0), which means that the instruction requires a writeback operation. If **rd** is not 0, the function examines the instruction's opcode to determine what operation needs to be performed.

If the instruction is an arithmetic or logic instruction (**opcode == 19** or **opcode == 51**), the function sets **wb\_result** to the result of the arithmetic or logic operation stored in the **alu\_result** register. If the instruction is a load instruction (**opcode == 3**), the function sets **wb\_result** to the data retrieved from memory and stored in the **MEM\_result** register. If the instruction is a jump and link instruction (**opcode == 111**), the function sets **wb\_result** to the address of the next instruction after the jump instruction. If the instruction is a jump and link register instruction (**opcode == 103**), the function sets **wb\_result** to the address of the next instruction after the jump instruction. If the instruction is a load upper immediate instruction (**opcode == 55**) or an add upper immediate to PC instruction (**opcode == 23**), the function sets **wb\_result** to the result of the arithmetic operation stored in the **alu\_result** register.

If a writeback operation is required, the function writes the value stored in **wb\_result** to the register file at the location specified by the **rd** field of the instruction. The function then increments the clock cycle counter **clk**, updates the program counter **pc** to the next instruction address, sets the x0 register to 0, and prints a message indicating whether a writeback operation was performed or not.

# Test plan

We test the simulator with following assembly programs:

* Fibonacci Program
* Sum of the array of N elements. Initialize an array in first loop with each element equal to its index. In second loop find the sum of this array, and store the result at Arr[N].
* Bubble sort