**DOCUMENTATION**

**FUNCTIONAL SIMULATOR FOR SUBSET OF RISC-V INSTRUCTION SET**

This document describes the design aspect of RISC-V Simulator, a GUI-based functional simulator for a subset of ARM instruction set.

**Input Mechanism**

Input to the RISC-V Simulator is a Mem file that contains the contents of Instruction memory and the Data memory separated by a dollar sign.

It contains the machine code of the instructions and their corresponding address separated by space.

The Data Memory is stored as the data and its corresponding address separated by space.

**Example :**

0x0 0xE3A0200A

0x4 0xE3A0300A

0x8 0xE0821003

0xc $

0x10000000 0x2

0x10000004 0x8

**Note :** 0xc $ is used to distinguish between Instruction Memory and Data Memory.

**Simulator Design**

**DATA STRUCTURE**

The Registers, Instruction Memory, Data Memory and the control signals are declared as global variables.

The Registers are maintained in an array of size 32 where each element is an integer.

The Instruction Memory and the Data Memory are maintained in python dictionaries where the key is the address of the instruction (or the data) and value is the instruction (or the data) itself.

**SIMULATOR FLOW**

There are two broad 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 **WriteBack.**

**FETCH**

In this stage, firstly the PC is converted into its hexadecimal equivalent and stored in a variable. Using this variable as key, Instruction is fetched from the Instruction Memory.

PC\_Temp is updated to store PC + 4.

**DECODE**

In this stage, firstly the opcode and the funct3 fields of the instruction is extracted using bit masking technique.

(Even if there is no funct3 field in the instruction, extracting those bits does not harm.)

1. **Extraction**

There are five different types of instructions in the instruction set that we are asked to support. So based on the opcode, the instruction is classified into one of those categories i.e. **R** type, **I** type, **S** type, **SB** type, **U** type and **UJ** type.

After identifying the type of the instruction, the values of each of its fields are extracted and stored in desired variables.

There are multiple instructions in these categories as well. So to identify the instruction particularly, the values of funct3 and funct7 fields are used if necessary.

Based on the type of instruction, the values of **rs1** and **rs2** registers are stored in **RA** and **RB** respectively.

If it is a store instruction then **RM** is updated to store **RB**.

The value of the **immediate field** is sign extended.

**Note :** In case of UJ and SB format instruction, immediate value is doubled after sign extension.

**Note :** It is essential to identify the particular instruction to figure out the type of operation to be performed.

1. **Operation Identification**

An array ALUOp of size 15 is created. Each index of this array represents a unique operation. For example index 0 represents addition, 1 represents subtraction and so on.

In the Execute stage, the type of operation to be performed is determined by checking the index which is on (i.e. which index has value 1).

**Example :** Suppose index 0 has value 1, then addition will be performed in the ALU.

After the instruction is particularly identified, corresponding to the type of operation which should be performed in this instruction, the particular index in the array ALUOp is turned on and the rest are turned off.

1. **Control Signals Generation**

The necessary control signals needed for the smooth operation of this instruction is generated in the Decode Stage itself.

There are a total of **9** control signals in this RISC-V Simulator which are listed below.

**Control Signals :**

**MuxB\_Select MuxC\_Select MuxINC\_Select MuxY\_Select MuxPC\_Select MuxMA\_Select RF\_Write Mem\_Write**

**Mem\_Read**

After identifying the instruction, the values of each of these control signals are properly set.

**EXECUTE**

In this stage, two variables are created namely **InA** and **InB** to store the final operands. InA is set to RA. Based on the value of **MuxB\_Select**, InB stores either RB or immediate value calculated in Decode Stage.

Now, the index of the array ALUOp which is turned on is found. Based on the type of operation it represents, that operation is performed between InA and InB and the result is stored in **RZ**.

In case of comparison operation, RZ stores 1 if the type of comparison performed is true otherwise 0.

**MEMORY ACCESS**

In this stage, based on the value of **MuxY\_Select**, three different operations can be performed.

**Case 1 : MuxY\_Select == 0**

In this case, the result of Execute Stage i.e. **RZ** is stored in **RY**.

**Case 2 : MuxY\_Select == 1**

The hexadecimal equivalent of **RZ** stored in **MAR** (Memory Address Register). Now if **Mem\_Read** is on, then using MAR as key, data is fetched from the Data Memory and stored in **RY.**

Otherwise if **Mem\_Write** is on, **MDR** is updated to store **RM** and this value is updated in the desired address of Data Memory.

**Case 3 : MuxY\_Select == 2**

**RY** is updated to store **PC\_Temp**.

**IAG Module**

This function implementing this module is called in Execute Stage. This module is used to update the value of the PC.

If **MuxPC\_Select** is off, then the PC is updated to store the value RA.

Otherwise if **MuxINC\_Select** is on then the PC is incremented by immediate value.

If **MuxINC\_Select** is off then the PC is incremented by four.

**WRITE BACK**

If **RF\_Write** is on and RD is not equal to zero then the value of **RD** register is updated to **RY**.

**Credits :**

1. Fetch : Apurv, Aman, Vasu, Jaglike, Anant
2. Decode : Apurv, Aman, Vasu, Jaglike, Anant
3. Execute : Apurv, Aman, Vasu, Jaglike, Anant
4. Memory Access : Apurv, Aman, Vasu, Jaglike, Anant
5. WriteBack : Apurv, Aman, Vasu, Jaglike, Anant
6. GUI : Apurv, Aman, Vasu, Jaglike, Anant