# RISC-V ISA PIPELINED SIMULATOR : PHASE 2

## *Input*

1.) Input is received as a *.mc* file, which is basically divided into two parts. The first part is the instruction segment which contains PC(program counter), followed by instruction, in each line. The two parts of the file are separated by the means of a delimiter *0xffffc.* The second part is the data segment which contains memory address, followed by the data stored in that address, in each line.

For example:-

```
0x0 0x10000217
0x4 0x00020213
0x8 0x10000297
0xc 0x01428293
.
.
.
0xffffc
0x10000000 0x0A
0x10000004 0x2a
0x10000008 0x23
.
.
.
```

2.) The GUI interface will take the input from the user to choose if the pipeline is expected to work using data forwarding or has to stall.

## *Output*

Each instruction is read from the instruction memory one at a time, which is then decoded, executed, and the register file is updated respectively.

Given below is the set of supported instructions:-

| Format | Instructions |
|---|---|
| R format | add, sub, or, xor, and, rem, mul, div, sll, slt, sra, srl |
| I format | addi, andi, lw, lh, ld, lb, ori, jalr |
| S format | sw, sb, sd, sh |
| U format | lui, auipc |
| SB format | blt, beq,bne, bge |
| UJ format | jal |

The following output will be shown by the simulator:-

1. Memory (both instruction and data segments)
2. Register file (32 registers)
3. Total number of Cycles
4. Total instructions executed
5. Number of data transfer instructions executed
6. Number of ALU instructions executed
7. Number of control instructions executed
8. Number of stalls in the pipeline
9. Number of Data Hazards
10. Number of Control Hazards
11. Number of Branch Mispredictions
12. Number of Stalls due to Data Hazards
13. Number of Stalls due to Control Hazards
14. Details of the instruction being executed in each stage during that cycle

## *Working and Design of the Simulator*

**Main data structures used in implementation**:-

1. Array of size 32 to implement register file.
2. A dictionary of dictionaries to store all the variables for each instruction, which will simulate different hardware components.
3. A dictionary of dictionaries to store control path for each instruction.
4. Two dictionaries to simulate instruction memory and data memory respectively.
5. A dictionary to simulate branch target buffer.

**Handling of Data Hazards:-**

- To handle data hazards(mainly Read After Write), two hardware approaches are implemented: Pipeline Stalling and Data Forwarding. The user has been given the option to choose between the two.

**Handling of Control Hazards:-**

- To predict behaviour of branch instructions, a static branch predictor is implemented which will predict the branch as ‘not taken’.

- To predict behaviour of jump instructions, a branch target buffer is used to determine PC after jump is taken.

**Simulator Flow and its Working**:-

**init()**

This function will read the input file and correspondingly data memory and instruction memory are created using two separate dictionaries; *dict_data and dict_instructions*. Also, a dictionary of dictionaries is created to store all the variables for each instruction, and one to store control path for each instruction.

Now the following functions will be executed for each instruction:-

**fetch()**

- An instruction is fetched from the instruction memory and is stored in the instruction register (IR).
- If the encountered instruction is a jump instruction, then *jump_dict* (dictionary) is used, to determine PC after jump is taken, else incremented value of PC (PC+4), is stored as *pctemp*.

**decode()**

- First of all, the function extracts the *opcode* from the instruction register, according to which the instruction format and possible operations are determined.

Case 1: Opcode is 0x33

-> Possible operations are add, sub, or, xor, and, rem, mul, div, sll, slt, sra, srl.
-> The control type is set to ‘R’.
-> The other instruction fields *rd, func3, rs1, rs2, and func7,* are extracted.
-> Accordingly, by observing the *func3* and *func7*, the operation to be performed is determined, and the control operation is set.
-> The source registers and destination register is determined by *rs1, rs2* and *rd* respectively.

Case 2: Opcode is 0x13

-> Possible operations are addi, andi and ori.
-> The control type is set to ‘I’.
-> The other instruction fields *rd, func3, rs1, and imm,* are extracted.
-> Accordingly, by observing the *func3* ,the operation to be performed is determined, and the control operation is set.
-> The source register and destination register is determined by *rs1* and *rd* respectively.

Case 3: Opcode is 0x3

-> Possible operations are lb, lh, lw.
-> The control type is set to ‘I’.
-> The other instruction fields *rd, func3, rs1, and imm,* are extracted.
-> Accordingly, by observing the *func3* ,the operation to be performed is determined, and the control operation is set.
-> The source register and destination register is determined by *rs1* and *rd* respectively.

Case 4: Opcode is 0x67

-> Possible operation is jalr.
-> The control type is set to ‘I’.
-> The other instruction fields *rd, func3, rs1, and imm,* are extracted.
-> Accordingly, by observing the *func3* ,the operation to be performed is determined, and the control operation is set.
-> The source register and destination register is determined by *rs1* and *rd* respectively.

Case 5: Opcode is 0x23

-> Possible operations are sb, sw, and sh.
-> The control type is set to ‘S’.
-> The other instruction fields *func3, rs1, rs2, and imm,* are extracted.
-> Accordingly, by observing the *func3* ,the operation to be performed is determined, and the control operation is set.
-> The source registers *rs1 and rs2* are determined.

Case 6: Opcode is 0x63

-> Possible operations are beq, bne, bge, blt.
-> The control type is set to ‘SB’.
-> The other instruction fields *func3, rs1, rs2, and imm,* are extracted.
-> Accordingly, by observing the *func3* ,the operation to be performed is determined, and the control operation is set.
-> The source registers *rs1 and rs2* are determined.

Case 7: Opcode is 0x17

-> Possible operation is auipc.
-> The control type is set to ‘U’.
-> The other instruction fields *rd and imm,* are extracted.
-> The control operation is determined (set to ‘auipc’).

Case 8: Opcode is 0x37

-> Possible operation is lui.
-> The control type is set to ‘U’.
-> The other instruction fields *rd and imm,* are extracted.

-> The control operation is determined (set to 'lui').

Case 9: Opcode is 0x6f

-> Possible operation is jal.
-> The control type is set to 'UJ'.
-> The other instruction fields *rd and imm,* are extracted.
-> The control operation is determined (set to 'jal').

- If user has chosen the option of stalling, then the pipeline is stalled accordingly to tackle data hazards and branch instructions(flushing).

- If user has chosen the option of Data Forwarding instead, then the required values are forwarded from previous decode, execute or memory buffers to decode buffer according to the requirement. Two dictionaries *rd_use* and *forward* are created, for implementing data forwarding.

- If the decoded instruction is a jump instruction and is encountered for the first time, then the entry is stored in branch target buffer by indexing via its PC, and PC is updated according to target address. If any jump instruction has already been encountered before, then the branch target buffer is looked up to get the target address and the PC is updated accordingly.

- To handle branch instructions, flushing and static branch predictor are used.

**execute()**

Case 1: Control type is 'R'

-> ALU performs the operation on the two source register values, and will update the result in buffer *rz*.
-> PC is updated.

Case 2: Control type is 'I'

-> If the control operation is either addi, andi or ori, then the ALU performs the source register and the immediate value, and updates the result in the buffer *rz*. Also, PC is updated.
-> If the control operation is a load instruction, then the ALU calculates the effective address of the corresponding memory cell, which is then stored in *mar*. Also, PC is updated.
-> If control operation is jalr, then the ALU calculates the effective address using the source register and immediate, and updates the PC with this value.

Case 3: Control type is 'S'

-> ALU performs the operation on the source register rs1 and the immediate value to calculate the effective address of the corresponding memory, which is then stored in *mar.*
-> Value of source register *rs2* is stored in *mdr*.
-> PC is updated.

Case 4: Control type is 'U'

-> ALU performs the specified operation (lui or auipc), and updates the buffer *rz* with the result.
-> PC is updated.

Note: Branch instructions and jump instructions have been already handled.

**memory_read_write()**

Case 1: Control type is 'R' or 'U'

-> Value of buffer *rz* is forwarded to buffer *ry*.

Case 2: Control type is 'I'

-> If the control operation is either addi, andi or ori, value of buffer *rz* is forwarded to buffer *ry*.
-> If the control operation is a load instruction, the value of *mdr* is updated by the data at the memory cell specified by *mar.* Then the *mdr* value is forwarded to buffer *ry*.
-> If control operation is jalr, the value of *pctemp* is forwarded to buffer *ry*.

Case 3: Control type is 'S'

-> The value at *mdr* is stored at the location specified by *mar*.

Case 4: Control type is 'UJ'

-> The value of *pctemp* is forwarded to buffer *ry*.

**write_back()**

- If value of buffer *ry* has been updated during the instruction execution, then the value is forwarded to the register file to update the destination register.

- In case both read and write are being done in same clock cycle, then write back to the destination register is prioritised over read, but both are performed in the same cycle.

## *Test Plan*

The simulator is tested by assembly programs of *fibonacci series*, *bubble sort*, and *recursive factorial*.