RISC-V ISA SIMULATOR : PHASE 1

Input

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
.
.
.
```

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. Instruction Register (32 bits)
4. A print statement regarding completion of each execution step

Working and Design of the Simulator

Data Structures used in implementation:-

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

Simulator Flow and it's 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*.

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).
- 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’).

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.

Case 5: Control type is ‘SB’

-> Branch is evaluated.
-> If branch is true, PC is incremented according to the *imm.*
-> else, PC is incremented according to *pctemp*.

Case 6: Control type is ‘UJ’

-> PC is incremented according to the *imm*.

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.

Test Plan

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