Design Document: Functional Simulator for RISC V 32-Bit ISA

The document describes the functional simulator for RISC V 32-Bit ISA.

Input

Input to the simulator is .mc file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space. For example:

0x0 0xE3A0200A

0x4 0xE3A03002

0x8 0xE0821003

Functional Behavior and output

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

The execution of instruction continues till it reaches instruction “swi 0x11”. In other words as soon as instruction reads “0xEF000011”, the simulator stops and writes the updated memory contents onto a memory text file.

The simulator also prints messages for each stage, for example for the third instruction above following messages are printed.

- Fetch prints:
    - "FETCH:Fetch instruction 0xE3A0200A from address 0x0"
- Decode
    - "DECODE: Operation is ADD, first operand R2, Second operand R3, destination register R1"
    - "DECODE: Read registers R2 = 10, R3 = 2"
- Execute
    - "EXECUTE: ADD 10 and 2"
- Memory
    - "MEMORY:No memory operation"
- Writeback
    - "WRITEBACK: write 12 to R1"

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:
        - First memory is loaded with an input memory file.
        - Simulator executes instructions one by one.

For the second step, there is an infinite loop, which simulates all the instructions till the instruction sequence reads "SWI 0x11".

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

★ FETCH

The input file is opened in read mode and instructions and data are stored in different dictionaries(in_mry and dt_mry) as a string with the corresponding address as key. Using a global variable 'PC'(initialized to zero) instructions are fetched from the dictionary 'in_mry' using the 'get()' function. 'PC' is converted to hex format (pc=hex(PC)) and 'in_mry.get(pc)' will return the required instruction code. The instruction is stored in another global variable 'IR'. Once fetch, decode, execute, memory and write-back steps of an instruction is finished, 'PC' is incremented by 4 and can be used to fetch the next instruction.

★ DECODE

The global variable 'IR' contains instructions in hex format. It is converted to binary format. Then the opcode of the instruction (instruction[25:32]) is matched with the opcode of R, I, S, SB, U and UJ format instructions to find the instruction type. If the instruction is of R format, then the function7 (instruction[0:7]) followed by function3(instruction[17:20]) of the instruction is matched with function7 and function3 of different R format instructions(add, sub, and, or, etc.) using nested if statement. If the instruction is of I or S or SB format, then function3(instruction[17:20]) of the instruction is matched with function3 of different instructions of that format. If instruction is in U or UJ format, only the opcode is checked for determining the instruction(0010111→auipc(U), 0110111→lui(U),1101111→jal(UJ)). After this step, the required operands are extracted. Operands for all the instructions are given below.

rd=instruction[20:25]

rs1=instruction[12:17]

rs2=instruction[7:12]

imm=instruction[0:12] - for I format only

offset=instruction[0:7]+instruction[20:25] - for S format only

imm=instruction[0]+ instruction[2:8]+ instruction[20:24] + instruction[1] - for SB format only

imm=instruction[0:20] - for U format only

imm=instruction[0]+instruction[12:20]+instruction[11]+instruction[1:11] - for UJ format only

whichever is applicable for an instruction.

★ EXECUTE

After Decoding the instruction held in the instruction register, the next step is to execute the instruction. Execute function will receive the mnemonic of the instruction, data stored in registers and immediate or offset values from the decode function. Execution function has 6 parameters as its input namely s1, s2,s3,s4,drs1,dr2 . These parameters are passed to the execute function according to format of the instruction being read:

R format: s1- Mnemonic name; s2- Destination register s3- Register 1; s4- Register 2; drs1- Data of register 1; drs2- Data of register 2

I format: s1- Mnemonic name; s2- Destination register s3- Register 1; s4- Immediate value; drs1- Data of register 1

S format: s1- Mnemonic name; s2- Register 1; s3- Register 2; s4- Offset; drs1- Data of register 1; drs2- Data of register 2

SB format: s1- Mnemonic name; s2- Register 1; s3- Register 2; s4- Offset; drs1- Data of register 1; drs2- Data of register 2

U format: s1- Mnemonic name; s2- Destination register s3- Immediate value

UJ format: s1- Mnemonic name; s2- Destination register s3-Offset

With these above data inputs execution of instruction takes place as per their formats. For Example:

- Instruction: ADD x4, x5, x6

  In this instruction sum of data values stored in x5 and x6 will write back to x4 as a register update.

- Instruction: ANDI x4, x5 26

  In this instruction bitwise AND operation is performed between data value in x5 and immediate value, 26 and then the result will write back to register x4.

Similarly, Execution of other instructions takes place.

★ MEMORY

It does two tasks, either read from the data memory or write into the data memory. It takes three parameters as input.

r_w → read or write

src_des→ destination to which the data to be written or destination from where the data is t be read

value → the data

According to 'r_w' ('r' or 'w') the data is read from the memory(using 'get' function) or written into the memory using destination as key.

★ WRITE-BACK

It takes two parameters ‘rd’ (Register number) and ‘value’. It will update the Dictionary named ‘reg’ using ‘rd’ as the key and ‘value’ as the value.

Test plan

We test the simulator with following assembly programs:

- Fibonacci Program
- Factorial program
- Bubble Sort