Design Document: Functional Simulator for a Subset of ARM instruction set

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

# **Input/Output**

## **Input**

Input to the simulator is a .mc file that contains the encoded instruction i.e their machine codes and the corresponding address at which instruction is supposed to be stored, separated by space. The data segment, if present, starts after the code segment ends, i.e. after 0xEF000011 (swi 0x11) is read. For example:

0x0 0x00000113

0x4 0x00100193

0x8 0x00200213

0xC 0xEF000011

0x10000000 0xFFFFFFDD

0x10000004 0xFFFFFFE6

0x10000008 0x00000001

## **Functional Behavior and output**

The simulator reads the instruction from instruction memory, decodes the instruction, reads the register, executes the operation, access the memory when needed and writes back to the register file. The instruction set supported is the same as given in the phase 1 description.

The execution of instruction continues till it reaches the instruction “swi 0x11”. That is when the instruction reads “0xEF000011”, the simulator stops and writes the updated memory contents in the output file.

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

* Fetch prints:
  + 'FETCH: Fetch instruction 0x00100193 from address 0x4'
* Decode
  + 'DECODE: Operation is ADDI, first operand x0, destination register x3, immediate value is 1'
  + 'DECODE: Read registers x0 = 0'
* Execute
  + 'EXECUTE: ADD 0 and 1'
* Memory
  + 'MEMORY : No memory operation'
* Writeback
  + 'WRITEBACK: 1 in x3'

# **Design of Simulator**

## **Data structure**

Registers, memories are private variables. Thus these variables are encapsulated and are not visible outside memory and register files.

## **Simulator flow:**

There are two steps:

1. First memory is loaded with the input memory file.
2. Simulator executes instructions one by one.

For the second step, there is an infinite loop, which simulates all the instructions till the fetch step intercepts “0xEF000011” instruction.

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

1. FETCH -

(i) Instruction is fetched from the IR register using the current PC value.

(ii) PC is incremented by 4

1. DECODE -

In the decode the current value of instruction present in IR register is passed as parameter in 32 bit binary format. This instruction is decoded as follows:

(i) The last 7 bits are extracted to get the op code. By using the op code the instruction type or fmt is found.

(ii) Then the fmt is used to find what will be the instruction format and rs1, rs2, rd, immediate, func3 and func7 are extracted accordingly from their positions in the instruction format.

(iii) In R-type: func3 and func7 are extracted and used to find the mnemonic of the instruction. Rs1, rs2 and rd were extracted.

(iv) In I-type: func3 was extracted and was used to find the mnemonic along with opcode. Rs1, rd and immediate were extracted.

(v) In UJ-type: op code was used to find the mnemonic. Rd and immediate were extracted.

(vi) In U-type: op code was used to find the mnemonic. Rd and immediate were extracted.

(vii) In S-type: func3 was extracted and was used to find the mnemonic along with opcode. Rs1, rs2 and immediate were extracted.

(viii) In SB-type: func3 was extracted and was used to find the mnemonic along with opcode. Rs1, rs2 and immediate were extracted.

1. EXECUTE -

In execute output of decode stage is taken as parameter i.e.[ ins\_fmt, inst, registers, memory, imm ] .This instruction has been executed as follows.

1. R-Type : in R-Type instruction operation has been performed on rs1 and rs2 depending upon the type of the instruction and result is stored in a temporary register (ry) , which is then returned to store this value in rd.
2. I-Type: similar to R-Type operation is performed on rs1 and imm and result is stored in ry ,which then returned to store this value in rd.
3. S-Type: Effective address is returned as per the definition of S-Type of instructions to store the word,byte or halfword at destination.
4. SB-Type:These are conditional branching instructions, which updates the PC value when condition is satisfied
5. U-Type: lui->sets rd to a 32-bit value with the low 12 bits being 0 and the high 20 bits coming from the U-type immediate .

auipc-> sets rd to the sum of the current PC and a 32-bit value with the low 12 bits as 0 and the high 20 bits coming from the U-type immediate.

1. UJ-Type:Current PC value is returned in jal instruction and PC us updated as per the definition of jal instruction
2. MEMORY ACCESS

Memory access takes place when the control bit for memory non zero. Other parameters are memory module, register module and the address at which the operation is to be performed. For different values of the memory control bit, the instructions lw, lh, lb, sw, sh and sb are performed. In case of store operations, 0 is returned. In case of load operations, the value stored at the address is returned. The methods lw, lh, lb, sw, sh and sb are present in the memory class.

1. WRITE-BACK -

Write-back takes place when the control bit for write-back is non zero. Other parameters are the register module and the data to be written. If the write control is non-zero, then the data is written in the destination register. The “make0()” method ensures that the register x0 always contains the value 0.

1. Main Memory

The main memory initializes a dictionary which contains the address and the value stored at the address, byte by byte. The “code\_ends()” function writes back the final memory in the output file when the “Dump” is pressed. The load methods take address as their parameter and return the value stored at that address. The store methods take address and data as their parameters and they store the data at the address, i.e. they update the memory.

# **Test plan**

We test the simulator with following assembly programs:

* Bubble sort iterative
* Bubble sort recursive
* Factorial iterative
* Factorial recursive
* Fibonacci iterative
* Fibonacci recursive