Design Document: Functional Simulator for Subset of RISC-V instruction set

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

Input

PHASE-I

Input to the simulator is an **“asmbl.asm”** file that contains the assembly language instruction of the code to be executed along with an optional data segment.

Example-

addi x2 x0 2

addi x1 x2 1

sub x3 x1 x2

PHASE-II

Input is a “**mcode.mc**” file (which is the output of phase 1) with encoded instruction (machine code) and the corresponding address at which instruction is supposed to be stored, (program counter)separated by space.

- 0x0 0xE3A0200A
- 0x4 0xE3A03002
- 0x8 0xE0821003

At the end of this segment, the file contains the data segment which starts at **0x10000000** and contains the memory locations and data stored , separated by a space.

- 0x10000000 0x08
- 0x10000001 0x00
- 0x10000002 0x00
- 0x10000003 0x07

Functional Behavior and Output

PHASE-1

The program(PHASE1.cpp) reads the RISC-V instruction from an assembly file (say asmbl.asm).The above program will generate a machine code file (**mcode.mc**) .

.mc file will have the following format:

- 0x0 0xE3A0200A
- 0x4 0xE3A03002
- 0x8 0xE0821003 (end of text segment)

It contains the address of the instruction(program counter) and the machine code of the instruction ,separated by a space.

At the end of this segment, the file contains the data segment which starts at **0x10000000** and contains the memory locations and data stored , separated by a space.

- 0x10000000 0x08
- 0x10000001 0x00
- 0x10000002 0x00
- 0x10000003 0x07

The terminal also prints the memory loaded from the .data segment.

PHASE-2

The program(PHASE2.cpp) reads the machine code line by line from the output file of Phase 1, decodes the instruction, reads 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 the end of the machine code (mcode.mc) file. In other words as soon as the end of mcode.mc is reached, the program stops and writes the updated memory contents onto a memory text file.

Corresponding to each instruction executed, it's program counter, type of instruction, operation performed, values stored in registers and other information such as memory and register update is printed. The no. of clock cycles are also printed after each instruction.

The program also prints messages for each stage, for example for the instruction(addi x10,x10,-1) following messages are printed.

- Fetch instruction 0xFFF50513 from address 0x34 ##
- Decode: Operation is addi
- First operand: x10
- Immediate value:-1
- Destination register: x10

- Read registers x10 =1
- Execute addi 1 and -1
- Memory: No memory operation
- Writeback: 0 to x10
- Number of clock cycles= 24

At the end of phase 2, 2 files named data_mem.mc and data_r.mc are generated which contain the memory locations next to stored data and registers with corresponding values respectively.

Design of Simulator

Data structure

Register file, memory array,stack are declared as global static. Being static, the variables are not visible outside the file, thus, make the data encapsulated in the PHASE2.cpp.

The memory array is of size 6000, with data segment starting from **0x10000000** and the stack segment growing downwards from **0x10001770.**

Simulator flow:

There are two steps:

1. First memory is loaded with input memory file, both .text and .data segments being loaded into the memory .

2. Simulator executes instructions one by one according to the logic flow, starting at PC=0X0.The pc is updated according to the instruction executed and accordingly, the next instruction is fetched. This continues until the pc is less than equal to the last instruction of the input machine code.

   Finally, the memory array is written to the **“data_mem.mc”** file and the register values are written to the **“data_r.mc”** file.

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

- FETCH: In this step, the machine code file is read line by line and machine code and program counter of the instruction is obtained. Further program counter for next instruction is obtained by incrementing the previous PC by 4 (except for the instructions of SB ,UJ format and jalr).In addition to this ,the machine code(hexadecimal) obtained in this step is converted to binary and passed to decode function.

- DECODE: The instruction passed on from fetch is decoded further. The opcode and funct 7, funct 3 (if any) are used to predict the operation type and name (using operation.txt). Next according to the instruction type and format, we decode the register, immediate and/or offset values stored in the instruction and return this decoded information. RA and RB (if they exist for the instruction format) are returned at this stage. The values returned are then passed on to execute( ).
- EXECUTE: First of all we identify the operation and next we store the result in rregno_rd and each and every operation type has its own execution formula. Further what is done in execution whatever it be add or addi or mo execution step, it is clearly mentioned in cout. All the arithmetic operations are done in this segment of code. And finally the rregno_rd (value calculated), regno_rd (the destination register) and the operation type is passed to the Memory access function .
- MEMORY: In this step we compare the operations with the instructions , if the instructions are add, sub , mul, div, or, etc. then there is no memory access.If the operations are load, store etc then we have a memory access procedure and if the operation doesn't match the instruction then we have an invalid input.
- WRITE-BACK: In this step we read the value to be written in the register and then write it back to the register. If there is no write operation then a message indicating that no writeback operation is displayed. After this the control goes to fetch the next instruction according to the pc to writeback until the machine code terminates.

Test plan

We successfully tested the simulator with following assembly programs:

- Fibonacci Program
- Bubble Sort
- Factorial Program