RISC-V ISA Simulator: Design Documentation

24.03.2020

Team:

Paras Goyal	2018CSB1111
Ekansh Mahendru	2018CSB1087
Hansin Ahuja	2018CSB1094
Sakshay Mahna	2018CSB1119
Atul Tiwari	2018CSB1077

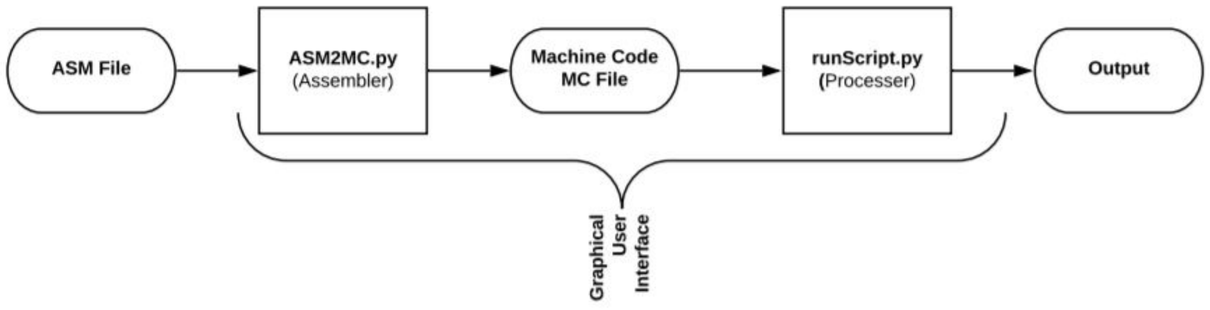
Goals

- The goal is to design a RISC-V instruction set simulator for educational purposes.
- The simulator must facilitate the RISC-V instructions taught in class and use the standard instruction formats to encode the aforementioned instructions. This task is implemented in Phase I of the project.
- The simulator must drive these instructions through the 5 cycles while maintaining the proper dataflow of these instructions. This task is implemented in Phase II of the project.
- The simulator is accompanied by an intuitive graphical user interface to tie the two phases together. The GUI helps visualise the current state of program execution, current status of registers, memory, program counter, etc.

Features

- **Simulator backend:** Python 3
- **GUI backend:** Flask
- **ISA:** RISC-V RV32IM
- **Size of instruction:** 32 bits
- **Number of registers:** 32
- **Size of registers:** 32 bits

Overview



The current version of the simulator facilitates the first two phases of the project:

I. Phase I: Conversion of assembly code to machine code

1) Input: .asm file

- The .asm file will have instructions as follows:
  add x1, x2, x3
  ori x2, x1, x4
  ...

- The instruction formats are:

R-Type:	ins rd rs1 rs2
I-Type (Except jalr):	ins rd rs1 immed
SB-Type:	ins rs1 rs2 label
S-Type:	ins rs1 immed(rs2)
U-Type:	ins rs1 immed
UJ-Type:	jal rd label
Jalr:	jalr rd immed(rs1)

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

- Supported assembler directives: .text, .data, .byte, .half, .word, .dword, .asciz.

- The following sample test case files have been included in our submission:

bubble.asm:	Implement bubble sort on the input array
fib.asm:	Find the nth fibonacci number
fact.asm:	Find factorial of a number

2) Output: .mc file

- .mc file has a format for each instruction:
  (<address-of-instr><delimiter-space><machine-code-of-instr>
- The code segment starts at 0x00000000, data segment starts at 0x10000000, stack segment at 0x7FFFFFFC and heap segment at 0x10007FE8.
- This output file would be fed to phase 2 of the project.

II. Phase II: Five step instruction execution

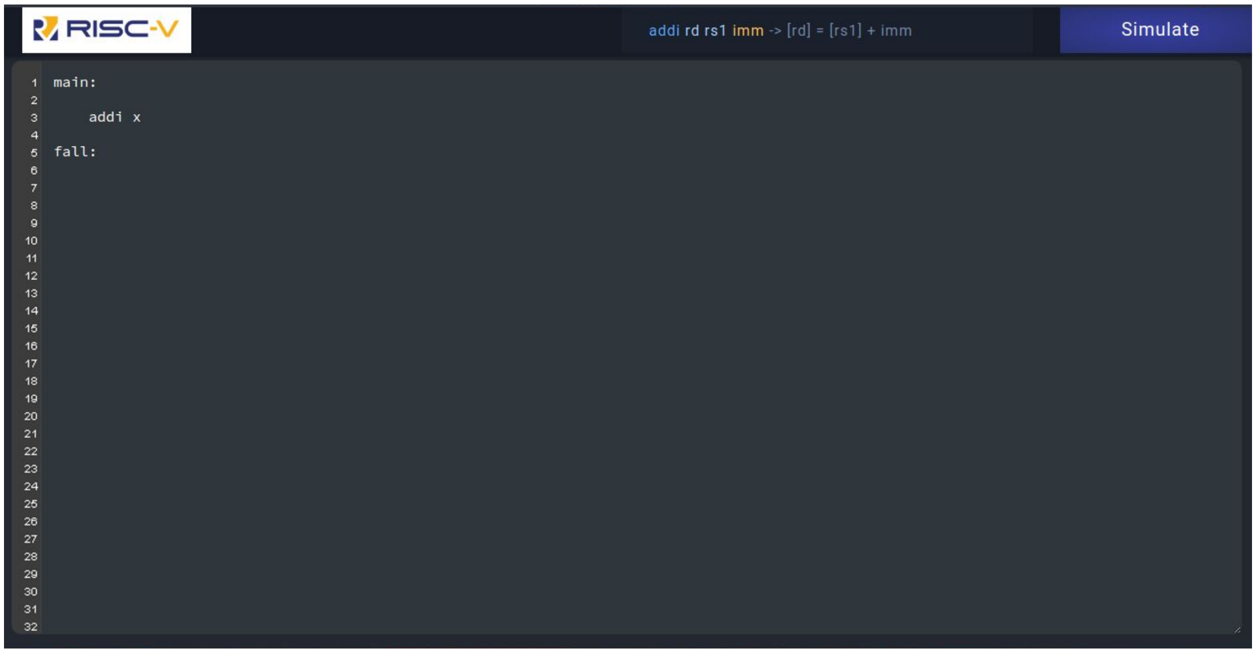
Input: .mc file generated in Phase I

- All the instructions in the given in the input .mc file is executed as per the functional behavior of the instructions. Each instruction will go through the following steps:
  Step 1: Fetch
  Step 2: Decode
  Step 3: Execute
  Step 4: Memory Access
  Step 5: Register Update or Writeback
- We include structures for all the registers here - PC, IR, Register File, temporary registers (like RM, RY, etc), etc., five steps of instruction execution as functions.

Graphical User Interface

1. A web app built on Google Chrome servers as the Graphical User Interface for the simulator.
2. The web app calls a local flask API to assemble code and simulate it. The API returns the register file, PC and memory dictionary after the instruction is executed, i.e. after the five-step execution process.
3. Some features:

- **Syntax Helper**: This section in the navbar displays the correct syntax for the command that is currently being entered in the editor.



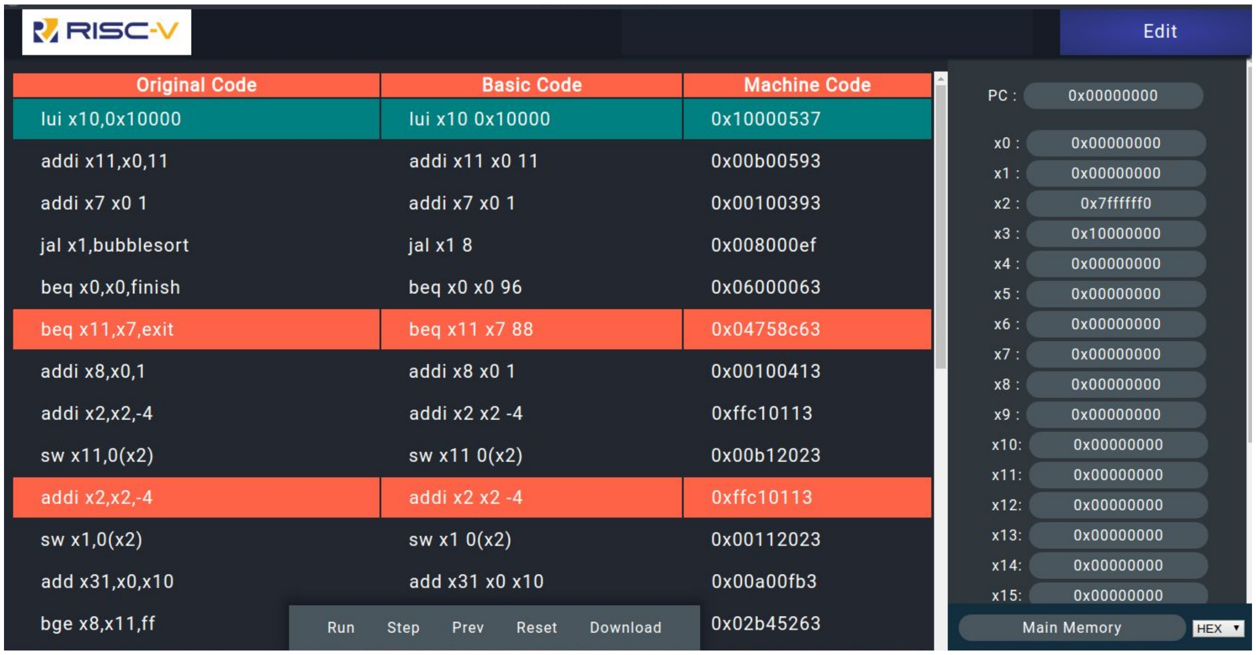
addi rd rs1 imm -> [rd] = [rs1] + imm

```
main:

    addi x

fall:
```

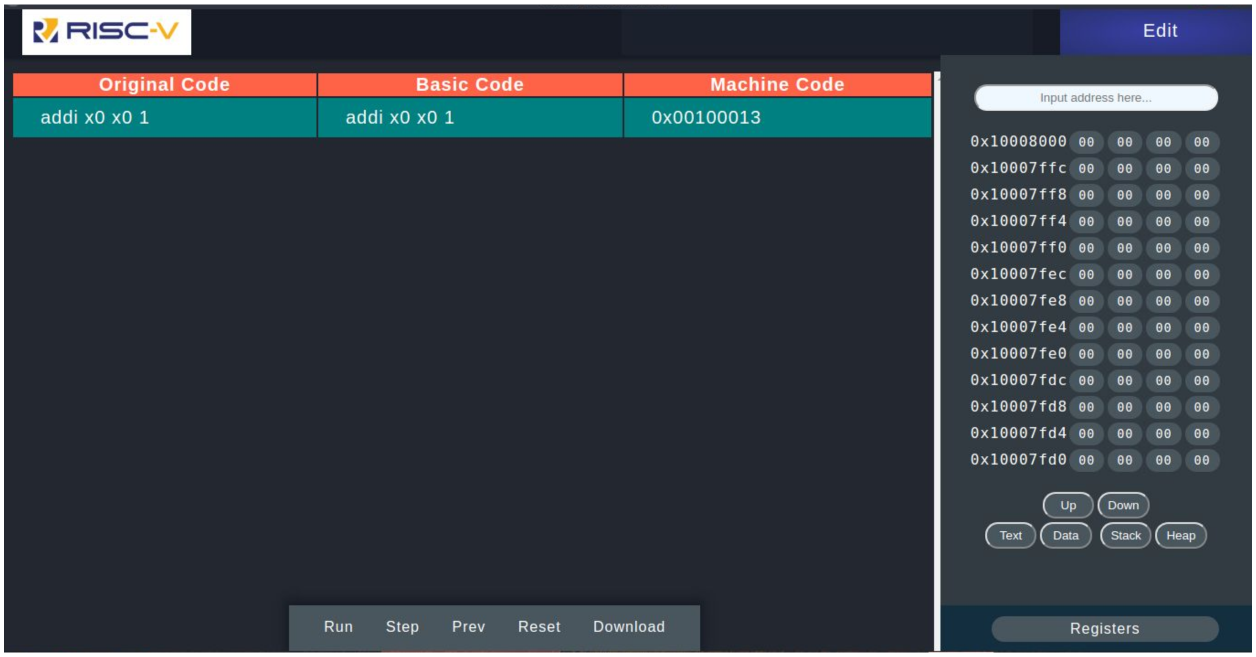
- **Breakpoints:** Add/Remove breakpoints in the program by clicking on the command in the simulator tab.



Original Code	Basic Code	Machine Code
lui x10,0x10000	lui x10 0x10000	0x10000537
addi x11,x0,11	addi x11 x0 11	0x00b00593
addi x7 x0 1	addi x7 x0 1	0x00100393
jal x1,bubblesort	jal x1 8	0x008000ef
beq x0,x0,finish	beq x0 x0 96	0x06000063
beq x11,x7,exit	beq x11 x7 88	0x04758c63
addi x8,x0,1	addi x8 x0 1	0x00100413
addi x2,x2,-4	addi x2 x2 -4	0xffc10113
sw x11,0(x2)	sw x11 0(x2)	0x00b12023
addi x2,x2,-4	addi x2 x2 -4	0xffc10113
sw x1,0(x2)	sw x1 0(x2)	0x00112023
add x31,x0,x10	add x31 x0 x10	0x00a00fb3
bge x8,x11,ff		0x02b45263

PC :	0x00000000
x0 :	0x00000000
x1 :	0x00000000
x2 :	0x7ffffff0
x3 :	0x10000000
x4 :	0x00000000
x5 :	0x00000000
x6 :	0x00000000
x7 :	0x00000000
x8 :	0x00000000
x9 :	0x00000000
x10:	0x00000000
x11:	0x00000000
x12:	0x00000000
x13:	0x00000000
x14:	0x00000000
x15:	0x00000000

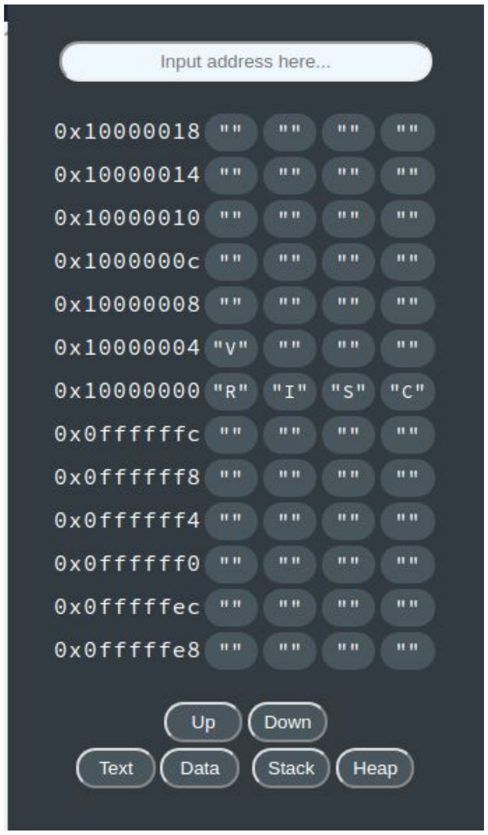
- **Memory Pane:** Memory pane allows the user to view the contents of different memory locations either by clicking the segment or by entering the address manually.



Original Code	Basic Code	Machine Code
addi x0 x0 1	addi x0 x0 1	0x00100013

```
0x10008000 00 00 00 00
0x10007ffc 00 00 00 00
0x10007ff8 00 00 00 00
0x10007ff4 00 00 00 00
0x10007ff0 00 00 00 00
0x10007fec 00 00 00 00
0x10007fe8 00 00 00 00
0x10007fe4 00 00 00 00
0x10007fe0 00 00 00 00
0x10007fdc 00 00 00 00
0x10007fd8 00 00 00 00
0x10007fd4 00 00 00 00
0x10007fd0 00 00 00 00
```

- **Datatype Selector:** User can select between DEC, HEX and ASCII to display in registers and memory pane.



```
0x10000018 ""  ""  ""  ""
0x10000014 ""  ""  ""  ""
0x10000010 ""  ""  ""  ""
0x1000000c ""  ""  ""  ""
0x10000008 ""  ""  ""  ""
0x10000004 "V" ""  ""  ""
0x10000000 "R" "I" "S" "C"
0x0ffffffc ""  ""  ""  ""
0x0ffffff8 ""  ""  ""  ""
0x0ffffff4 ""  ""  ""  ""
0x0ffffff0 ""  ""  ""  ""
0x0fffffec ""  ""  ""  ""
0x0fffffe8 ""  ""  ""  ""
```