Layla Mohsen Fatemah Abdel Wahed Mariam Nousseir

Assembly RARS Simulator Report

RARS Simulator Description :

Our RARS Simlulator takes input from a file, written in assembly language RISC-V code.
We ask the name fo the file from the user, and open the file, with an error message if the file does not open. We add each line in vector of string called “inputcode”, using getline.
We then ask user the name of memory file, similarly with an error message and adding them to vector “memorycode”.
After we input the code, we begin parsing.
First, we begin with the memory file. In a for loop going through size of “memorycode”, we look for a comma or a space using find function. “string::npos” searches substring in the target string, “comma_index != string::npos” means if substring found, if comma and space are found we go in the if condition. A memory line will have the format “100, 4”, so the first left part of the string goes to the address variable, and the second right part goes to the value variable. The value turns to a binary to fit the memory, and is divided into parts of 8 bits each, to be added into their corresponding memory address, with each address holding 8 bits.
Secondly, we intiliaze the instructions in the “inputcode” file. In the beginning, we look for “:” to know if it’s a label, since they hold the format “Label:”. If it is a label, the label name is stored in the map called label <string, int>, with the string of the name of label and their address. The rest of the code is added again into the “inputcode” vector.
We ask the user of the initial state of the Program Counter, and change it accordingly by dividing by 4, to fit our format. In a while loop, the register x0 remains a constant and inside we add a for loop to go over the characters in the current line of the “inputcode” vector. If the current character is a comma, a space or a parenthesis, and not empty, we push back in vector the current character, else if the character is the end of the parenthesis, we skip the if loop, and else we add the to the string name the newest character. We attribute to the string function the first word of the line, which is the instruction to do. Depending on the instruction and format (if R, I, SB. UJ) , we add to a the corresponding struct the words inside the vector of strings. And inside he if loop of creating the struct, we also call the execute function with the Program Counter, and the specific struct Instruction format. This execution repeats until reaching the last line of the code. We output the PC and print the registers after each line.

Bonus features :

- Outputting all values in decimal, binary, and hexadecimal (instead of just decimal which is assumed to be the default)
- Including a larger set of test programs (at least 6 meaningful programs) and their equivalent C programs.

Design Decisions/Assumptions :

Types of registers:

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

The registers are stored in a map of <string,int>, with string being the name of the register, from x0 to x31, and int is the value stored in the register. They are all intilaized to 0. The registers are in the x format (x0, x1...).

Similarly, the memory takes the same format of a unsorted map <int, string>, of the first being the location in memory, and the second in the value stored. The memory is initialized from a file input.

We also created a map for the label <string, int> with the name of the label and its address.

We used 4 structs, “RFormat”, “IFormat”, “SBFormat”, “UJFormat” and each has their own execute functions.

Bugs/ Issues :

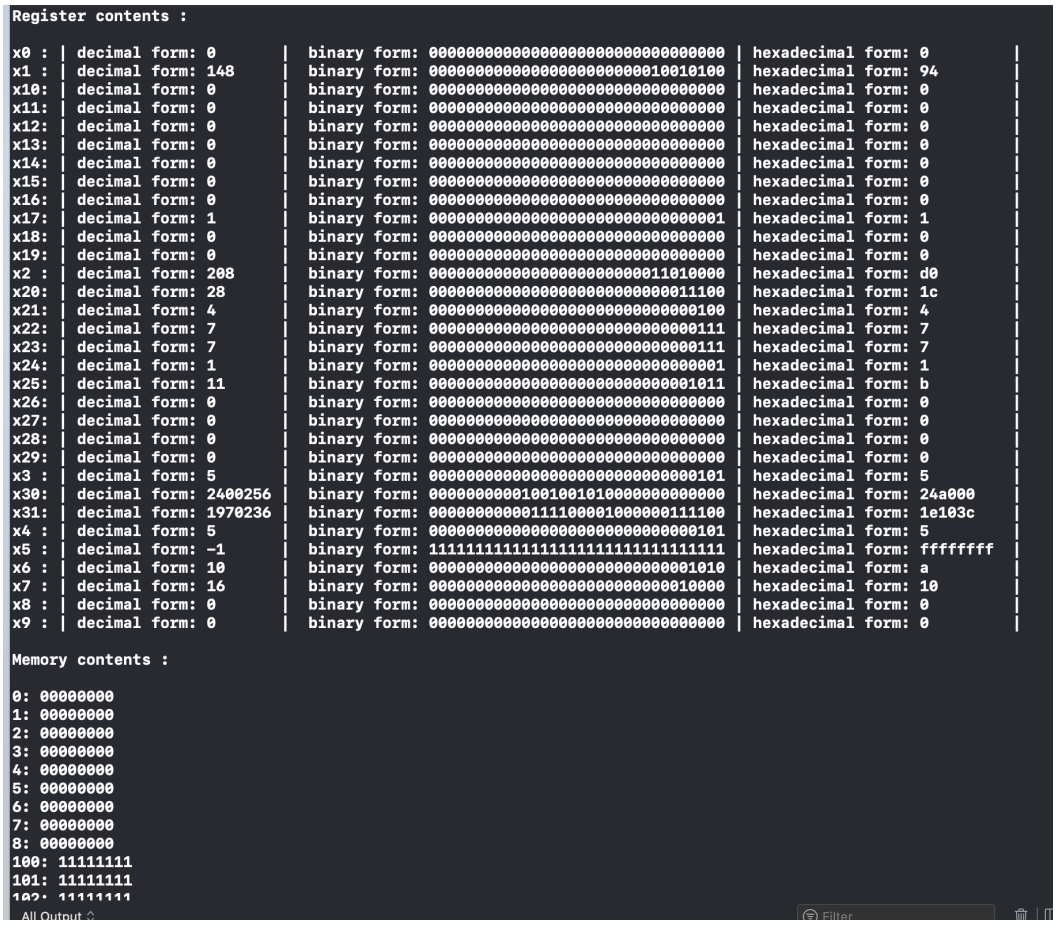
User Guide :

1. User should input path of file/name of file with the code and instructions

2. User should input path of file/name of file with the initialized memory values
3. User should input PC initial state

```
Enter program file name:
/Users/layla/Desktop/allinstructions.txt
Enter memory file name:
/Users/layla/Desktop/memory.txt
Enter PC initial state :
0
```

4. The instructions execute and are output.
   final register/memory contents (screenshot couldnt sustain all the memory contents)



```
Register contents :

x0 : | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x1 : | decimal form: 148     |  binary form: 00000000000000000000000010010100 | hexadecimal form: 94       |
x10: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x11: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x12: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x13: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x14: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x15: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x16: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x17: | decimal form: 1       |  binary form: 00000000000000000000000000000001 | hexadecimal form: 1        |
x18: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x19: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x2 : | decimal form: 208     |  binary form: 00000000000000000000000011010000 | hexadecimal form: d0       |
x20: | decimal form: 28      |  binary form: 00000000000000000000000000011100 | hexadecimal form: 1c       |
x21: | decimal form: 4       |  binary form: 00000000000000000000000000000100 | hexadecimal form: 4        |
x22: | decimal form: 7       |  binary form: 00000000000000000000000000000111 | hexadecimal form: 7        |
x23: | decimal form: 7       |  binary form: 00000000000000000000000000000111 | hexadecimal form: 7        |
x24: | decimal form: 1       |  binary form: 00000000000000000000000000000001 | hexadecimal form: 1        |
x25: | decimal form: 11      |  binary form: 00000000000000000000000000001011 | hexadecimal form: b        |
x26: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x27: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x28: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x29: | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x3 : | decimal form: 5       |  binary form: 00000000000000000000000000000101 | hexadecimal form: 5        |
x30: | decimal form: 2400256 |  binary form: 00000000001001001010000000000000 | hexadecimal form: 24a000   |
x31: | decimal form: 1970236 |  binary form: 00000000000111100001000000111100 | hexadecimal form: 1e103c   |
x4 : | decimal form: 5       |  binary form: 00000000000000000000000000000101 | hexadecimal form: 5        |
x5 : | decimal form: -1      |  binary form: 11111111111111111111111111111111 | hexadecimal form: ffffffff |
x6 : | decimal form: 10      |  binary form: 00000000000000000000000000001010 | hexadecimal form: a        |
x7 : | decimal form: 16      |  binary form: 00000000000000000000000000010000 | hexadecimal form: 10       |
x8 : | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |
x9 : | decimal form: 0       |  binary form: 00000000000000000000000000000000 | hexadecimal form: 0        |

Memory contents :

0: 00000000
1: 00000000
2: 00000000
3: 00000000
4: 00000000
5: 00000000
6: 00000000
7: 00000000
8: 00000000
100: 11111111
101: 11111111
102: 11111111
 All Output                                                                    Filter
```