Computer Architecture Lab Assignment 4

Divya Rajparia ES22BTECH11013
Ahmik Virani ES22BTECH11001

Building a RISC V Simulator

October 3, 2024

This report has 3 main sections:

1. **Program Design:** Which explains our high level approach to the problem, and the procedure followed to simulate an assembly code.
2. **Assumptions Made:** Which explains the assumptions our simulator will make about the input assembly code
3. **Testing of code:** This section describes all testing conducted to ensure correctness of our simulator.

Program Design

Similar to the assembler, here also we have modularized our code to handle each instruction with a separate function.

3 pass method:

We have implemented a 3 pass method in our simulator.

1. On doing load input.S, the first pass happens, wherein we first detect labels, store them in a map, and then store the entire assembly code in a 2D string vector - to help us in execution later.
2. In the second pass, we go through all of these instructions, convert them to their machine code representation using the assembler logic, and then we store it in the **text** section of the memory.
3. Now, we are all set to step/run the code. On every step/run command, the instruction is sent to the function for its specific instruction type, and it runs - making the changes in memory, registers, and program counter.

General Step function design for all formats:

1) The argument for step function of each instruction is a vector, which contains the operands of the instruction. We first identify the operands - including registers and immediate values(if any) on which we have to execute this instruction
Please note, we have also implemented the alias remover function here - which checks if any of the register operands are referenced by alias names, in which case they are replaced by their original names - to maintain uniformity.

2) Now once we have identified the registers/immediate values we have to work on, we create an if else block which executes the required operation - and stores the result in either the registers, memory, or the control branches to a different line - based on what the instruction demands.

3) After this, we print the executed instruction, along with the program counter - as specified in the question, and then we finally update the globally declared program counter, to send it to the next instruction to be executed. Please note, that in case of B format and J format instructions, we change the value of PC not by 4, but by the value specified in the operands.

Logic behind memory operations - load and store:

1) Loading unsigned values: Since our memory is byte addressable, we deal with values in the same way - byte by byte. We wish to explain loading unsigned values with the help of lw. lw takes a 32 bit i.e. 4 byte data, and stores it into a register. Now here's the thing, the little endian format used denies us the simplicity of storing the bytes in a simple order. We need the least significant byte to come at the least memory location. Hence, for an instance if we consider:
ld x2, 0(x4), where the memory location starting from x4 contains the .word 0x12345678, byte0 would be 78, byte 1 would be 56, byte 2 would be 34, and byte 3 would be 12. Thus, to accomplish this, we meed to left shift the byte 3 by 24, byte 2 by 16, byte 1 by 8, and use "or" between these 4 bytes - to yield our desired result.

2) Loading signed values: The process here is similar to above, except one thing - the data type we use to store these bytes becomes changes to signed integer. The program automatically treats the numbers as signed, and loads them accordingly.

3) Storing values: Here, we want to store data into the memory. We follow a similar approach as load, but now we right shift the number to extract each byte, and to store it in the memory location one by one.
Example: sw x3, 0(x4) where x3 contains 0x12345678 and x4 is 0x10000. In this case, byte0 becomes 78, byte1 56, byte2 34 , and byte3 12, where we get each of these by right shifting the word and extracting the first byte.

Assumptions Made:

input.S format(data and text section)

1. If the input.S has a data section, both the data and text sections have to be mentioned with their heading of **.data** and **.text**. If only text section is present, it is not necessary to mention .text header.

2. In the header lines, no data/instruction are there on the same line. Data and instructions start from the next line.
   For example:
   .data .dword 1234 - incorrect.
   .data
   .dword 1234 – correct

3. In case of multiple data elements of a particular type, they should be **comma separated.**
   For example:
   .dword 1234, 4333, 2344 – correct
   .dword 1234 –incorrect
   33454
   32343

Immediate value format followed

1. For the data section, all numbers **must be in decimal.**
   Example:
   .dword –correct
   17
   .dword
   0x11 –incorrect

2. Immediate values for all I format instruction **must be in decimal.**
   Example:
   addi x2, x3, 17 –correct
   addi x2, x3, 0x11 –incorrect

3. Immediate value for the lui instruction **can be in either representation - decimal, or hexadecimal.**

4. Memory indexing **must be in hexadecimal.**
   Example:
   mem 0x10000 8 –correct
   mem 65536 8 –incorrect

5. Memory count value **must be in decimal.**
   Example:

mem 0x10000 8 –correct
mem 0x10000 0x8 –incorrect

6. Breakpoint line number **must be in decimal.**
   Example:
   break 12 –correct
   break 0xc –incorrect

Whitespacing followed for input.S

1. We have assumed that there is only one space between instruction and next operand, and only one space between the comma of the preceeding operand and next operand in line. No spaces between comma and its preceeding operand.

2. It is mandatory to have **the instruction after the label to be in the same line as the label.**
   Example:

```
L1: addi x5, x5, 2 --correct

L1:
    addi x5, x5, 2 --incorrect
```

3. We have supported instructions inside a label/function to be spaced as well as non-spaced.
   Example:

```
L1: addi x5, x5, 2
    add x2, x2, x3
    sub x4, x4, x5

L1: addi x5, x5, 2
add x2, x2, x3
sub x4, x4, x5
```

   Both of these formats for input.S will be accepted by our code.

Format for jalr

We have assumed the following bracketing and format for jalr instruction:

```
jalr rd, imm(rs1)
not
jalr rd ,(imm)rs1
```

Testing and Correctness of Code:

We created 4 test case files which cover **all instructions**, **all register names**, **all alias names**, as well as **all edge-cases**.
There are different files for different instruction types - one for I, one for R, one for S, and a combined one for B, J and U. These files are attached with our submission.