ES 215

Computer Organization and Architecture

Course Project: SDLX Simulator

Amey Rangari: 21110177
Hirva Patel: 21110154
Shrimay Shah: 21110204

Submitted on: 4th April 2023.
Instructor: Prof. Rajat Moona, Prof. Sameer Kulkarni

INDEX

1. **Project Brief:**

   A simulator for an SDLX processor can be considered a program that mimics the processor's working electronically. It takes an 8-bit program file as input and gives an output generated which is similar to the one generated by the processor. It helps us to predict the output of the processor. Also, it can be used to check the instruction code, as it helps to look at the registers' value after each instruction. Thus, the user can debug errors in the instruction code, if any. This project aims to develop a Python program that acts as a working simulator of the simplified DLX processor.

2. **Acknowledgments:**

   Even though we had some basic intuition about how the code should function, we were unsure about what exactly was expected from us in terms of input and output. We would like to thank Prof. Sameer G Kulkarni for helping us out at that moment. We would also like to thank our batchmate Naman Dharmani for helping us understand the project better. We have tried to meet the requirements of viewing the data stored at various locations at different times, as was expected, in our simulator.

3. **Work Distribution:**

   The initial architecture of the code was brainstormed by all the team members and was ultimately proposed by Hirva and Shrimay. Further, all three teammates contributed to writing the code, several versions of which were discarded. In the final submission, the ALU functions were written together by the entire team. The final user interface was designed by Hirva, and the report was compiled by the team together.

4. **Structure of the Code:**

   Our team has written the code for the SDLX processor in Python language. The processor has been defined as a class from which objects containing different instructions as programs are called. The execution of these objects is similar to the Verilog computation, except for some conventions of Python that affect data manipulation. As a result of this, some functions of the ALU needed modification. The Memory and the Reg file come predefined as an array of integer values (signed) with all elements initialized to zero when the SDLX processor class is called. The program counter can be traced by the global variable 'pc'.

5. **Functions:**

Apart from the main 'execute' function that processes the instructions, some others have been defined inside and outside the class. The 'load program' program stores the set of instructions, fed as input, in the memory. The 'invert' function just changes the zero from the given location to one. This function is used to change the state of the switch whenever said by the user. Memory addresses of LEDs are linked to the memory addresses of eight switches. Thus, the state of LEDs is also stored in memory. There is an 'input_user' function for an interactive input system wherein the user gives the values of some parameters, including the number of instructions that would get executed at once, switch state, etc.

6. **Defining Memory and RegFile:**

In our Python script, we have defined the memory of size 400 bytes. However, the code is essentially functional with any memory size. It functions in the same way as in an actual computer and stores values that can be written into and read from, but it doesn't change after every cycle (unless a Load or Store instruction is being executed). However, it can be viewed after every clock cycle.

The Regfile is an array of 32 integers (stores signed values), which dynamically changes and shows changes after every instruction (if specified by the user).

7. **ALU Instructions**

The same as our SDLX processor, our simulator executes instructions of 3 types based on the Opcodes, which are the first 6 bits of the instruction (line 58).

   a. ***R-type Triadic Instruction***

   When the Opcode is 0, an R-type Triadic Instruction is to be executed.In this instruction format, there are two fields called RS1 and RS2 that each uses 5 bits to indicate one of the 32 registers used as a source for the instruction. The RD field indicates which register will hold the result of the instruction. The specific operation that the arithmetic logic unit (ALU) will perform is indicated by a Func code. There are also five unused bits in the instruction format that should always be set to 0, failing which could possibly result in a runtime error.

   For unsigned comparisons, there is some manipulation required with the data stored in registers. As the data is stored in signed values, we need to

convert it to its unsigned value. All other instructions are performed in the usual way.

b. R-I type Triadic instruction

Triadic instructions of the R-I Type have three operands. One of the operands is a 16-bit signed immediate number, while the other source operand must be a register. The destination operand is also a register. A variety of instructions can be executed via this instruction type. These include: ADDI, SUBI, ORI, ANDI, XORI, SLLI, SRLI, SRAI, SLTI, SGTI, SLEI, SGEI, ULTI, UGTI, ULEI, UGEI and LHI. In the Shift Instructions, the shift count is provided in the signed immediate constant. In comparison instructions, the second operand is the immediate constant and is treated as a sign value for relevant instructions. Data transfer instructions like instructions: LB (load signed byte), LBU (load byte unsigned), LH, LHU, LW (load word), SB (store byte), SH (store half word), SW can also be executed via R-I type Triadic Instructions. For these instructions, the memory address is determined by adding the contents of register RS1 with a 16-bit immediate constant, which is treated as a signed number. This instruction type is very essential as it is highly versatile and can deal with memory-interactive instructions.

c. R-type Dyadic Instruction

R-type dyadic instructions involve only two operands and use only one source register. The bit field 16-20 is unused in this format and should be 0. This type of instruction is mainly used for control transfer instructions, namely Jump and Branch instructions. All control transfer instructions follow a notion known as *Delayed Branch.* As the name suggests, the instruction immediately following the instruction will be executed irrespective of whether the branch is taken or not. For this to function, we have defined a signal called “ini”, which holds the information on whether the instruction being performed is the delayed branch instruction or normal instruction. In case of a delayed branch instruction, it jumps to the required address by changing “self.pc” to “next_pc” instead of the usual “self.pc + 4”.

d. J-type instruction

J-type instructions are a type of instruction format in computer architecture used for jump instructions. These instructions are used to transfer control to a different part of the program by setting the program counter (PC) to a new address. J-type instructions have a 26-bit address field, which is used to specify the new target address for the PC. The remaining 6 bits are the corresponding opcode. The target address for the jump is calculated from the 26 bits.

8. The Program Counter

The program counter or PC is required to keep track of the instructions performed. Our design of the simulator can execute multiple instructions or a single instruction as specified by the user. This helps the user to dynamically view the values in memory and registers and figure out errors, if any. This can be helpful to students who write instruction code, as they can view the result before feeding it to the processor. We figured out that there was a requirement for a global variable that continues the count of instructions after a new set of instructions is given, which is maintained by "total_instr_count". An extra variable called 'ini' has been added to take care of branch and jump instructions because of the delayed branching in SDLX, according to which the instruction after the branch or jump instruction is expected to be executed first before the branching (a jump of the pc) takes place. To handle this in our simulator, whenever these instructions appear, the execute function is called for one instruction at "PC+4" with 'ini' changed to 1. The value of the next instruction where the pc must jump is already updated in another variable called 'next_pc'. The next instruction to be processed comes from 'next_pc' if the earlier instruction had 'ini=1'.

9. The User Interface

A simulator is expected to show how the contents of the memory and the reg file change over each cycle/instruction so that it is possible to analyse the working of the processor better. Our simulator is able to cater to this by the 'spec_count' variable, whose value taken as an input decides the number of instructions that would be executed before an output is displayed. Some specific memory locations (200 to 207 and 208 to 215 respectively) are reserved for switches and LEDs as a part of the memory-mapped IO. The user can choose to turn on/off any switch or LED and view the states of each switch and LED at every instance of the output. In our simulator, the user needs to end the program by requesting the execution of 0 instructions. Whenever this happens, the simulator outputs the number of RAW

hazards that could have occurred and the cycles per instruction (CPI) that it keeps computing throughout, using the 'stall_count' variable.

10. Example Code

1) 00000100 00000001 00000000 00001110
   01010100 00000000 00000000 00000100
   00001000 00100001 00000000 00000111
   00000100 00100001 00000000 00011000
   00000100 00100011 00000000 00000111

   The instructions are:
   ADDI: R1 ← R0 + $14
   JALR: R0 ($4)
   SUBI: R1 ← R1 - $7
   ADDI: R1 ← R1 + $24
   ADDI: R3 ← R1 + $7

   Here, Jump instruction jumps to 5th instruction after executing 3rd.
   Thus, the result is:
   Registers: R1 = 7, R3 = 14

   In case, the 4th instruction was executed the output would be:
   Registers: R1 = 31, R3 = 38.

   The output generated is:
   How many instructions would you like to execute?
   5
   Any switch you wish to turn on or off? (Enter a number from 1 to 8, 0 otherwise)
   3
   Program counter = 5.0

   Memory
   4 , 1 , 0 , 14 , 84 , 0 , 0 , 4 , 8 , 33 , 0 , 7 , 4 , 33 , 0 , 24
   4 , 35 , 0 , 7 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
   0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0

0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0
0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0

Registers
[0, 7, 0, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12]

Switch states
Switch 1 = 0, Switch 2 = 0, Switch 3 = 1, Switch 4 = 0, Switch 5 = 0, Switch 6 = 0, Switch 7 = 0, Switch 8 =
0,
LED states
LED 1 = 0, LED 2 = 0, LED 3 = 1, LED 4 = 0, LED 5 = 0, LED 6 = 0, LED 7 = 0, LED 8 = 0,
How many instructions would you like to execute?
0
Program Stopped
RAW Hazard = 1

stall_count = 1
CPI = 1.125
PS C:\Users\HP\OneDrive\Documents\programming>

Explanation:
Program counter will be at the 5th instruction. The memory and registers values can be viewed. As we changed the state of 3rd switch, the switch state and corresponding LED has been turned to 1. Also, we can see these values stored in memory too at address: 202 and 210.

When we input 0, the program is stopped and hazards count, stall count and CPI are displayed. As seen in the input, we see one consecutive useof register R1 which will cause 1 RAW hazard.

2) 00000100 00000001 00000000 00001110
00001000 00100001 00000000 00000111
00000100 00100001 00000000 00011000
00000100 00100011 00000000 00000111

The instructions are:
ADDI: R1 ← R0 + $14
SUBI: R1 ← R1 - $7
ADDI: R1 ← R1 + $24
ADDI: R3 ← R1 + $7

Here, the output will be:
Registers: R1 = 31, R3 = 38.

The output generated is:
How many instructions would you like to execute?
5
Any switch you wish to turn on or off? (Enter a number from 1 to 8, 0 otherwise)
2
Program counter = 4.0

Memory

4 , 1 , 0 , 14 , 8 , 33 , 0 , 7 , 4 , 33 , 0 , 24 , 4 , 35 , 0 , 7
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , 0
0 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0
0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0

Registers
[0, 31, 0, 38, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Switch states
Switch 1 = 0, Switch 2 = 1, Switch 3 = 0, Switch 4 = 0, Switch 5 = 0, Switch 6 = 0, Switch 7 = 0, Switch 8 =
0,
LED states
LED 1 = 0, LED 2 = 1, LED 3 = 0, LED 4 = 0, LED 5 = 0, LED 6 = 0, LED 7 = 0, LED 8 = 0,
How many instructions would you like to execute?

```
0
Program Stopped
RAW Hazard =  3
stall_count = 3
CPI =  1.375
PS C:\Users\HP\OneDrive\Documents\programming>
```

Explanation:
As the program had only four instructions, the program counter shows 4.0 at the end of execution. The memory and registers values can be viewed. As we changed the state of 2nd switch, the switch state and corresponding LED has been turned to 1. Also, we can see these values stored in memory too at address: 201 and 209.

When we input 0, the program is stopped and hazards count, stall count and CPI are displayed. As seen in the input, we see three consecutive uses of register R1 which will cause 3 RAW hazards.