Project 3

Yohandi (120040025)

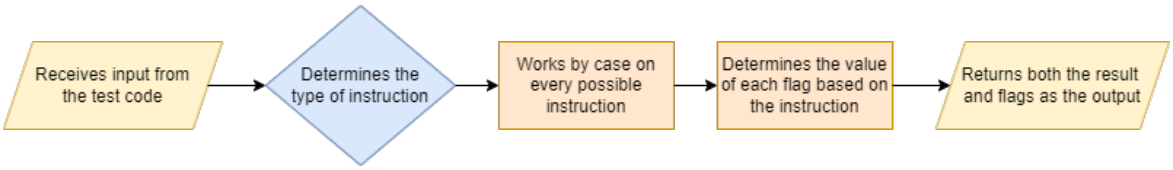
Design

Overview

This project is about simulating a module of ALU in a processor. From the given line of MIPS instruction, the program is expected to return a result with some notable flags: zero, overflow, and negative. The registers are restricted to only two registers, and those are `register_A` and `register_B`. `register_A` has `00000` value as its address, and `register_B` has `00001` value as its address.

Processing Logic

Flowchart



Work Summary

First, the module receives input from the test code. The input is some lines of instruction and register value. There are only two registers: `register_A` and `register_B`. The input is passed through the parameter of the module function.

Second, each MIPS instruction line has a different opcode and function code to describe a particular instruction specifically. Because of this reason, to determine the R-type instruction, the program will check whether the opcode is `000000` or not. If the opcode is `000000`, then the instruction is known to be an R-type instruction, which will later be determined based on its function code for the R-type instruction. Else, the instruction is known to be an I-type instruction and later will be determined based on its opcode. Note that there are only some considered instructions in this project, and J-type instructions are not included in this project.

Third, since each of the instructions has a different function, the work is done with the case-by-case method. This method is proved to be trivial yet the best solution to be worked on.

Lastly, in each of the considered instructions, some notable flags are required to be returned. For the overflow flag, the program is demanded to only consider `add`, `addi`, and `sub` instruction. The program is demanded only to consider both `beq` and `bne` instructions for the zero flag. For the negative flag, the program is demanded to only deal with `slt`, `slti`, `sltiu`, `sltu` instructions. With this flag's value and result, the output is returned.

Special Case

To solve the unsigned and signed problem, simply use `$signed` property provided by the Verilog. With this, it is not necessary to manually manipulate every bit according to the rules of each instruction. For example, in an `add` instruction, simply use the `rs + rt` code instead of adding each bit one by one. For the signed case, simply use `$signed(rs) + $signed(rt)`

Output

Running the Project

This project requires a Verilog compiler. Icarus Verilog is recommended as the compiler. After

installing it, the command `iverilog` can be used in the command prompt or the terminal. Note that it is required to have Icarus Verilog in the main path of the drive.

To run the make file, make sure `gnuwin32` is installed and is located in PATH. Then, for Windows users, open the command prompt, locate the destined folder, and run `make -f Make.txt` . Similarly, for Linux users, open the terminal, locate the destined folder, and run `./make.exe -f Make.txt` .

Custom Test Case

Of course, some particular test cases might not be good enough to represent the accuracy of the program. Because of this reason, the file `test_ALU.v` is allowed to be changed based on the user's desire. However, it is required to assign the input value based on the Verilog format.

Terminal Example

```
F:\Yohandi\OneDrive - CUHK-Shenzhen\Term 4\CSC3050\Projects\verilog>make -f Make.txt
iverilog -o test_ALU.vvp test_ALU.v
vvp test_ALU.vvp
instruction:op:func:  regA  :  regB  :   rs   :   rt   : result :z:n:o
   xxxxxxxx:xx: xx :xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:x:x:x
   00014020:00: 20 :40000000:40000000:40000000:40000000:80000000:0:0:1
   2009000d:08: 0d :00000003:40000000:00000003:00000000:00000010:0:0:0
   00015021:00: 21 :00000009:00000004:00000009:00000004:0000000d:0:0:0
   240b000d:09: 0d :00000003:00000004:00000003:00000000:00000010:0:0:0
   00014022:00: 22 :80000000:00000001:80000000:00000001:7fffffff:0:0:1
   00014823:00: 23 :00000008:00000006:00000008:00000006:00000002:0:0:0
   00014024:00: 24 :87654321:00006798:87654321:00006798:00004300:0:0:0
   3009000d:0c: 0d :0000000f:00006798:0000000f:00000000:0000000d:0:0:0
   00015027:00: 27 :87654321:69696969:87654321:69696969:10929496:0:0:0
   00015825:00: 25 :87654321:12345678:87654321:12345678:97755779:0:0:0
   340c000d:0d: 0d :00000182:12345678:00000182:00000000:0000018f:0:0:0
   00016826:00: 26 :87654321:12345678:87654321:12345678:95511559:0:0:0
   380e000d:0e: 0d :00000054:12345678:00000054:00000000:00000059:0:0:0
   1001ffff:04: 3f :00000008:00000006:00000008:00000006:00000002:0:0:0
   1401ffff:05: 3f :00000008:00000006:00000008:00000006:00000002:1:0:0
   0001402a:00: 2a :0000fec8:80000000:0000fec8:80000000:8000fec8:0:0:0
   2809000d:0a: 0d :80000008:80000000:80000008:00000000:7ffffffb:0:1:0
   2c0a000d:0b: 0d :80000008:80000000:80000008:00000000:7ffffffb:0:0:0
   0001582b:00: 2b :0000fec8:80000000:0000fec8:80000000:8000fec8:0:1:0
   8c290001:23: 01 :0000fec8:0000000e:0000000e:00000000:0000000f:0:0:0
   ac0a0001:2b: 01 :0000000e:0000000e:0000000e:00000000:0000000f:0:0:0
   00004340:00: 00 :00000005:0000000e:00000005:00000005:0000a000:0:0:0
   00204804:00: 04 :0000000f:0000000d:0000000d:0000000f:0001e000:0:0:0
   00005342:00: 02 :8ce00000:0000000d:8ce00000:8ce00000:00046700:0:0:0
   00205806:00: 06 :26158000:00000005:00000005:26158000:0130ac00:0:0:0
   00006343:00: 03 :ea618000:00000005:ea618000:ea618000:ffff530c:0:0:0
   00206807:00: 07 :ea618000:0000000d:0000000d:ea618000:ffff530c:0:0:0
test_ALU.v:143: $finish called at 280000 (1ps)
```