The Chinese University of Hong Kong, Shenzhen

CSC3050

Computer Architecture

Project2

Author:
Shi Wenlan

Student Number:
119010265

April 11, 2021

1 Introduction

I wrote a Verilog program that can read some MIPS instructions of assembly language and simulate ALU excution. This program can execute 27 MIPS instructions including *add*, *beq* and so on.

2 Big picture thoughts and ideas

My program would execute MIPS instructions in three steps: parsing the instruction, calculation, and output.

In the first step, I assign each component of the instruction to reg type variables *opcode*, rs, rt, *shamt*, $func$ and *immediate* respectively. For example, opcode = instruction[31:26]. Then I assign the input register contents to reg type variables reg_A and reg_B respectively. I also prepared reg type variables reg_C and *tags* to store the values to be outout later, where the defult value of *tags* is 000. Besides, I set a reg type variable *extension*, which is used for sign-extension.

In the second step, I noticed that 27 instructions can be identified only by checking *opcode* and $func$. First, judge the value of *opcode*. If *opcode* = 0, then the combination of *opcode* and $func$ corresponds to the instructions of R-format in 27 instructions one by one. If *opcode* is not equal to 0, then *opcode* corresponds to the instructions of non-R-format in 27 instructions one by one. After identifying the instruction, the program would calculate with the input data and store the result in reg_C, and modify the value of *tags*.

In the third step, I assigned the values of reg_C and *tags* to *result* and $flags$ respectively to output.

About flags: The three bits of flag are [overflow, negative, zero].
Instructions that need to handle overflow: add, addi, sub.
Instructions that need to handle negative: slt, sltu, slti, sltiu.
Instructions that need to handle zero: beq, bne.
And the zero flag will be set to 1 when beq/bne instruction jumps.

About result: The result of lw and sw will be the calculated address, beq and bne will not update the result, and the result of other instructions will be the result of calculating the input register contents. And the result will not be updated if overflow flag is set to 1 during add, addi and sub.

3 A data flow chart

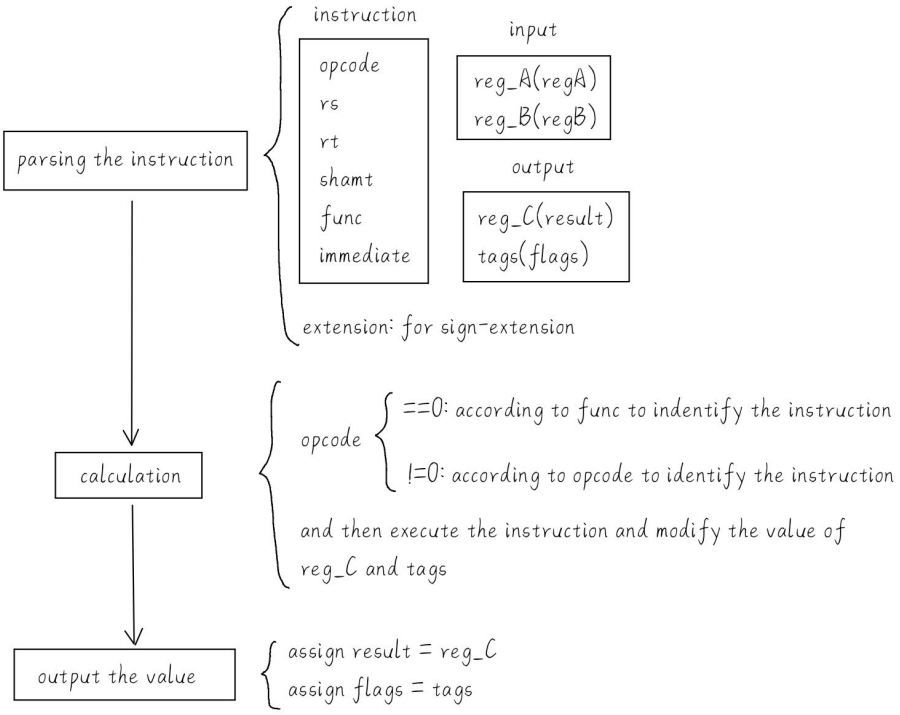

Figure 1: data flow chart

4 High level implementation ideas

About detecting overflow: Overflow is characterized by adding two positive numbers to obtain a negative number, or adding two negative numbers to obtain a positive number, or a positive number minus a negative number to obtain a negative number, or a negative number mines a positive number to obtain a positive number. Verilog has strict limits on the bit width of variables. If the calculated result exceeds the preset bit width, the high bit will be truncated. So I only need to check the sign bit of the input data and calculation result to determine whether overflow occurs.

About reading register contents: This program only allows two register addresses, 00000 is the address of regA, and 00001 is the address of regB. The program branches according to the addrees of the called register. If the read address is 00000, the content of reg_A will be read for calculation.

About sign-extension: Signed extension requires copying the sign bit by 16 bits and connecting it in front of the number to be sign-extended. I took out these 16 bits and stored them in a reg type variable *extension*. This is mainly for the convenience of writing code.

About test bench: Every 10ns in test bench, *instruction*, *regA*, *regB*, and *name* (instruction name) are assigned without blocking. And the theoretical results are written in the comments in the code.

5 Implementation details

About signed number and unsigned number: Verilog defaults to unsigned numbers, so in sra and srav, the input number needs to be converted into signed numbers using $\$signed()$ before being arithmetically shifted to right (>>>). If it is not converted to a signed number first, the bits added will still be 0s, not sign bits.

About result update control: First of all, the value of *result* is updated with the value of *reg_C*, so the result cannot be directly assigned to *reg_C* before confirming the update. So I did not assign *reg_C* in beq and bne. And in the instructions that needs to consider the overflow, I stored the calculation result in another reg type variable *reg_D* first. And I would use the sign bit of *reg_D* to judge whether the overflow occurs, and assign the value of *reg_D* to *reg_C* only after confirming that the overflow does not occur.

6 Operation manual

I wrote a makefile(output name is ALU) , but I am not sure if it will work, so I suggest using the following method:
1. Install Icarus Verilog according to the websete tutorial:
https://zhuanlan.zhihu.com/p/95081329
2. Enter this folder in terminal and enter:
iverilog -o wave test_ALU.v ALU.v
3. Enter: vvp -n wave -lxt2
4. At this time, you should be able to see the output in the terminal(figure 2). If you want to see the waveform diagram(figure 3&4), continue to enter:
gtkwave wave.vcd

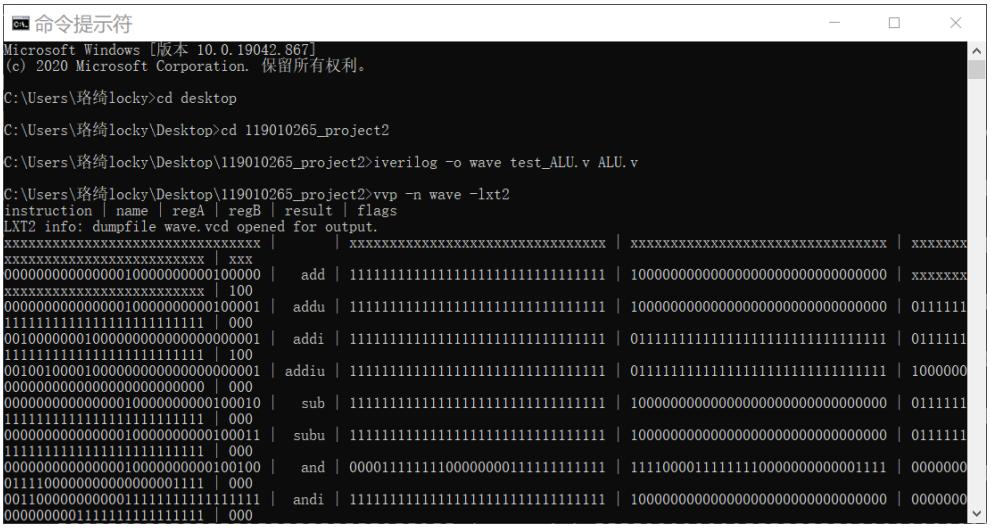

Figure 2: after step 3

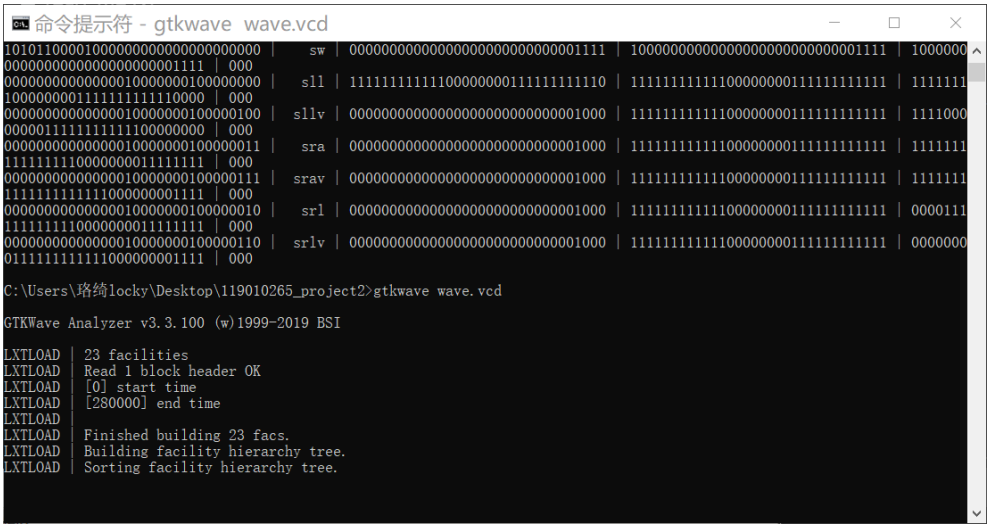

Figure 3: step 4

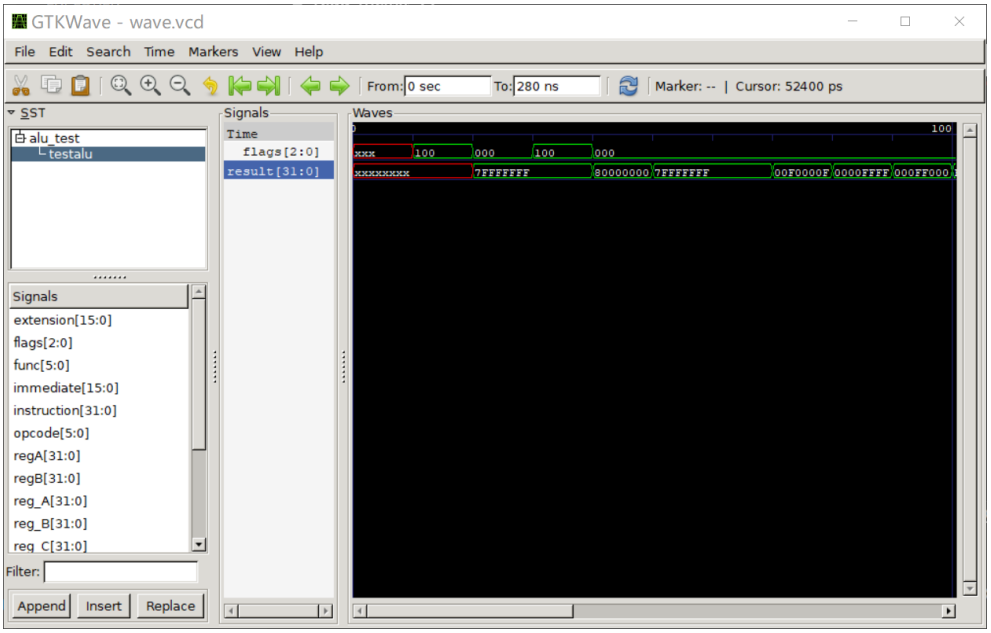

Figure 4: waveform diagram

7 Conclusion

This project has made clear the simple steps of ALU excution and has made me master the simple use of Verilog.