EE 469 Lab 3 Report

Jiarong Qian

In this lab, I implemented a 5 stage pipeline CPU based on the previous labs.

In lab 2, I implemented a single-cycle CPU without using pipeline, so the first thing I needed to do was to break down my lab 2 into 5 stages. I divided the lab 2 files into different modules and replaced most of the blocking assignments with non-blocking assignments. The CPU now worked like a multi-staged one. However, I was expecting the CPU would behave more like a pipeline one, since I set the input of the previous module to go directly into the next module and the CPU is always reading the next PC value from the registers. Later I found that the CPU was simply reading the same PC value over and over again until the PC is updated at the last stage. In order to implement the pipeline, I would have to make sure the PC updates on its own each clock cycle and writes that into the PC register instead of reading the latest PC value from it.

However, always updating the PC creates a new problem. If the CPU needs to do the branch, it would have to wait until stage 3 execution finishes so that it knows where to branch to, but that would be too late since the PC has already moved forwarded and some instructions I may not want to execute would be taken in as well.

In order to solve that problem, I added a function to the CPU that it would stop updating PC and taking new instructions until the branch instruction is over. By implementing this feature, the CPU could achieve branch as well as pipeline.

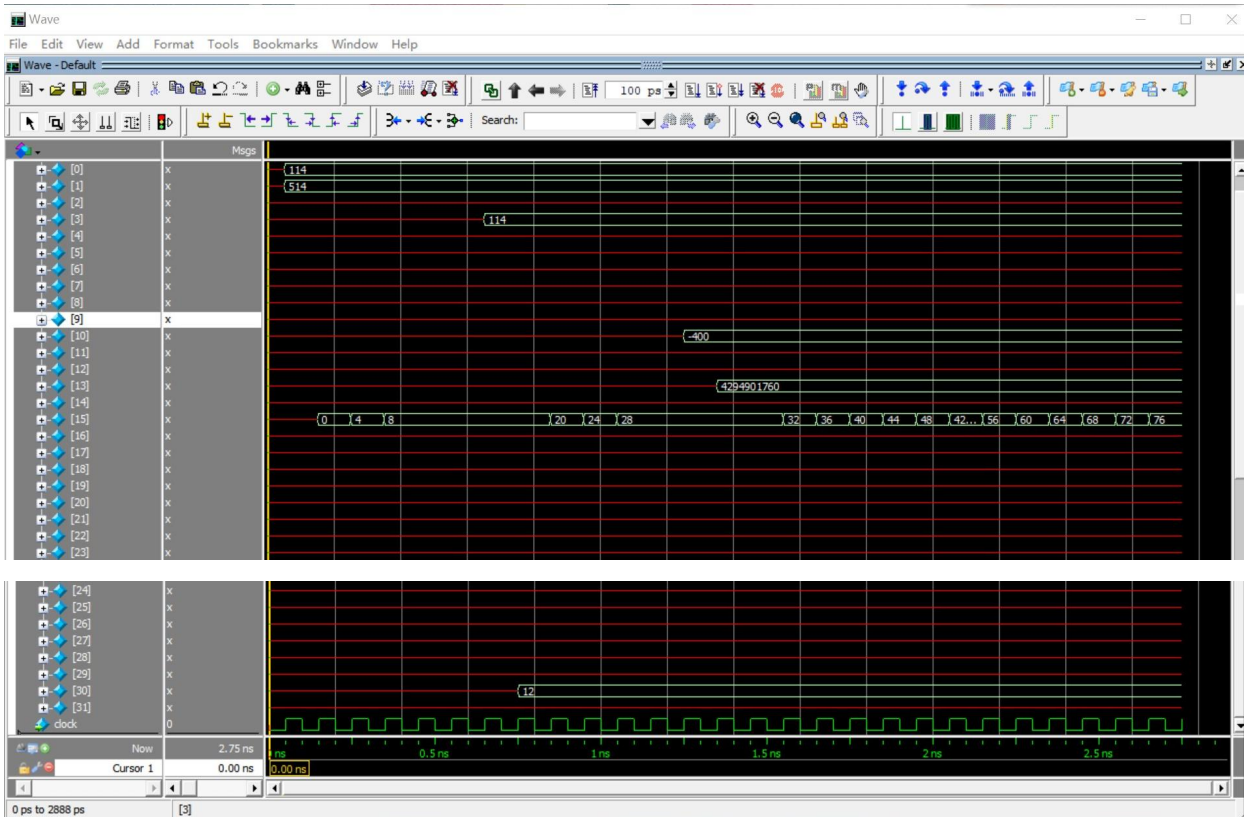
Another issue I had with pipeline was that the instructions did not produce the same output as in the single cycle design in lab 2. Then I found that the new instructions fetched from the code memory would override the old instruction while it was still updating so the result would be off. It is a classical pipeline problem and I solved that by adding more registers to hold the old values.

I tested my CPU with the following instructions:

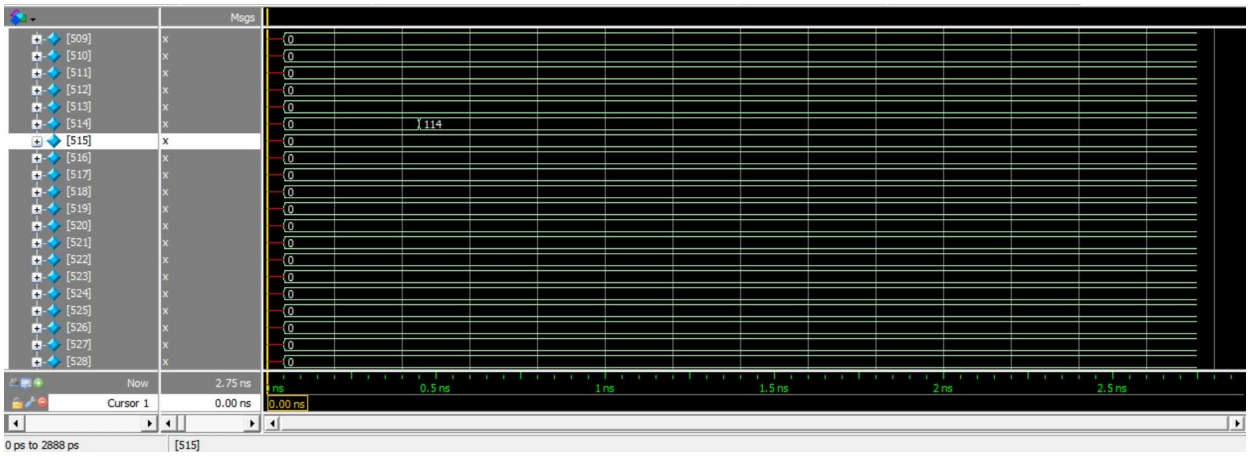
// PC = 0
// STR [r1, 0], r0
11111000_00000000_00000100_00000001
// PC = 4
// LDR r3, [r1, 0]
11111100_01000000_00000100_00100011
// PC = 8
// B # 12
10010100_00000000_00000000_00000011
// PC = 12
// ADDS r0, r1, r0 (skiped)
10101011_00000000_00000000_00100000

// PC = 16
// ADDS r1, r1, r1 (skipped)
10101011_00000001_00000000_00100001
// PC = 20
// SUBS r10, r1, r3
11101011_00000001_00000000_01101010
// PC = 24
// MOV r13, 65535 (<< 16)
11010010_10111111_11111111_11101101

The waveform and the debug console output is given below:



From the waveform of the registers we can see that the PC register is being updated, and some registers are written during the execution.



Here is a screenshot of the memory waveform. The value 114 from r0 is written into 514, which is pointed by the value stored in r1 after the first instruction.

```
add wave -position end  sim:/code_execution_testbench/c/mem/memory
VSIM 7> restart -f
VSIM 8> run -all
# rm:                    x, rn:                    x, pc:                    x, cond: xxxx, res:                    x, NZCV: xxxx
# rm:                    x, rn:                    x, pc:                    x, cond: xxxx, res:                    x, NZCV: 0000
# rm:                    x, rn:                    x, pc:                    x, cond: xxxx, res:                    0, NZCV: 0000
# rm:                    x, rn:                  114, pc:                    0, cond: xxxx, res:                  514, NZCV: 0000
# rm:                    x, rn:                  514, pc:                    4, cond: xxxx, res:                  114, NZCV: 0000
# rm:                    x, rn:                    x, pc:                    8, cond: xxxx, res:                   12, NZCV: 0000
# rm:                    x, rn:                    x, pc:                    8, cond: xxxx, res:                    x, NZCV: 0000
# rm:                  514, rn:                  114, pc:                   20, cond: xxxx, res:                 -400, NZCV: 0000
# rm:                    x, rn:                    x, pc:                   24, cond: xxxx, res:           4294901760, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   28, cond: 1111, res:                    x, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   28, cond: xxxx, res:                    x, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   32, cond: xxxx, res:                    x, NZCV: 1010
# rm:                    x, rn:           4294901760, pc:                   36, cond: xxxx, res:           4294901768, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   40, cond: xxxx, res:           4294901768, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   44, cond: xxxx, res:           4294901768, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   48, cond: xxxx, res:           4294901768, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   52, cond: xxxx, res:           4294901768, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   56, cond: xxxx, res:           4294901768, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   60, cond: xxxx, res:           4294901768, NZCV: 1010
# rm:                    x, rn:                    x, pc:                   64, cond: xxxx, res:           4294901768, NZCV: 1010
# ** Note: $stop    : C:/Users/Jiarong Qian/Documents/Course Materials/wi2020/ee469/lab3/cpu.sv(155)
#    Time: 2750 ps  Iteration: 1  Instance: /code_execution_testbench
# Break in Module code_execution_testbench at C:/Users/Jiarong Qian/Documents/Course Materials/wi2020/ee469/lab3/cpu.sv line 155
```

Here is a screenshot of the debug console, which prints the rm, rn values of the instruction, current PC value, computation result, execution condition and current flags. Since the code used in the test is only up to PC 24, the following output are generated by the code used in the test of lab 2. They can be neglected since the required registers might not be set and some instructions are undefined as well.