Zack Fravel

010646947

Lab 1 : M 4:10 - 5:55 PM

Lab 7 - Pipelined CPU

5/2/16

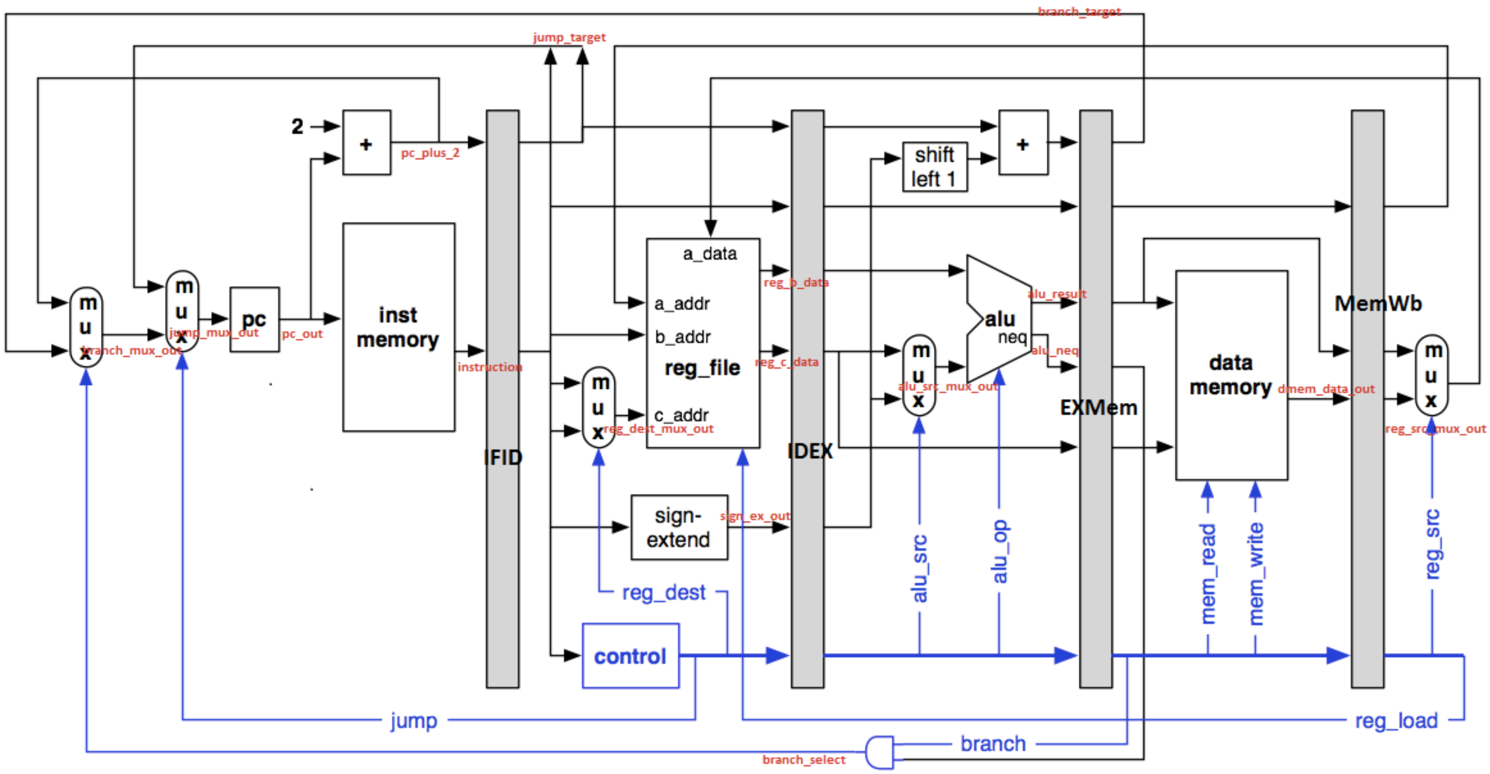
Introduction

The objectives of Lab 7 is to convert the Single-Cycle CPU we completed in lab 6 into a CPU that is implemented using pipelined instruction execution. What this allows us to do is get through our instructions much faster because we will have more than one instruction “in the pipeline” at any given time. This optimizes our CPU and allows us to execute more instructions in less time. However, it’s not all positive, with this pipelined design we also assume a few hazards, which we will have to take care of in our implementation.

Approach

The high level implementation of converting our Single-Cycle datapath into a Pipelined datapath is fairly simple. We want to break up our datapath into five phases: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write-back (WB). Each one of these phases in our datapath is broken up by adding a simple generic-width register that was given to us in the lab description. This generic width allows us to set the width of the register in our system entity, since each of our buffers take in different amounts of data.

One thing we need to take into account when dealing with a pipelined datapath is different kinds of hazards. There are three different types of hazards: structural, data, and control hazards. In our lab, we have a few data and control hazards we need to take care of. First of all, because of the way our pipeline is set up, it is possible for the write-back address that we give to the register to complete an operation to change as another instruction moves along the pipeline, causing us to write to the wrong registers. Another thing that can happen is a branch cannot properly execute because we have the next 3 instructions after it already in the pipeline. What we need to do to take care of some of these hazards is add 3 nop instructions (x”0000”) after the instructions where we have these issues. The way you figure this out is by looking where we have RAW (read-after-write) hazards, which would be an instruction using the destination register as an operand or a comparator immediately following the computation of that register. This is why we need to add nops, because we have actual hardware buffers between instructions. Below is a diagram of the datapath we are implementing.



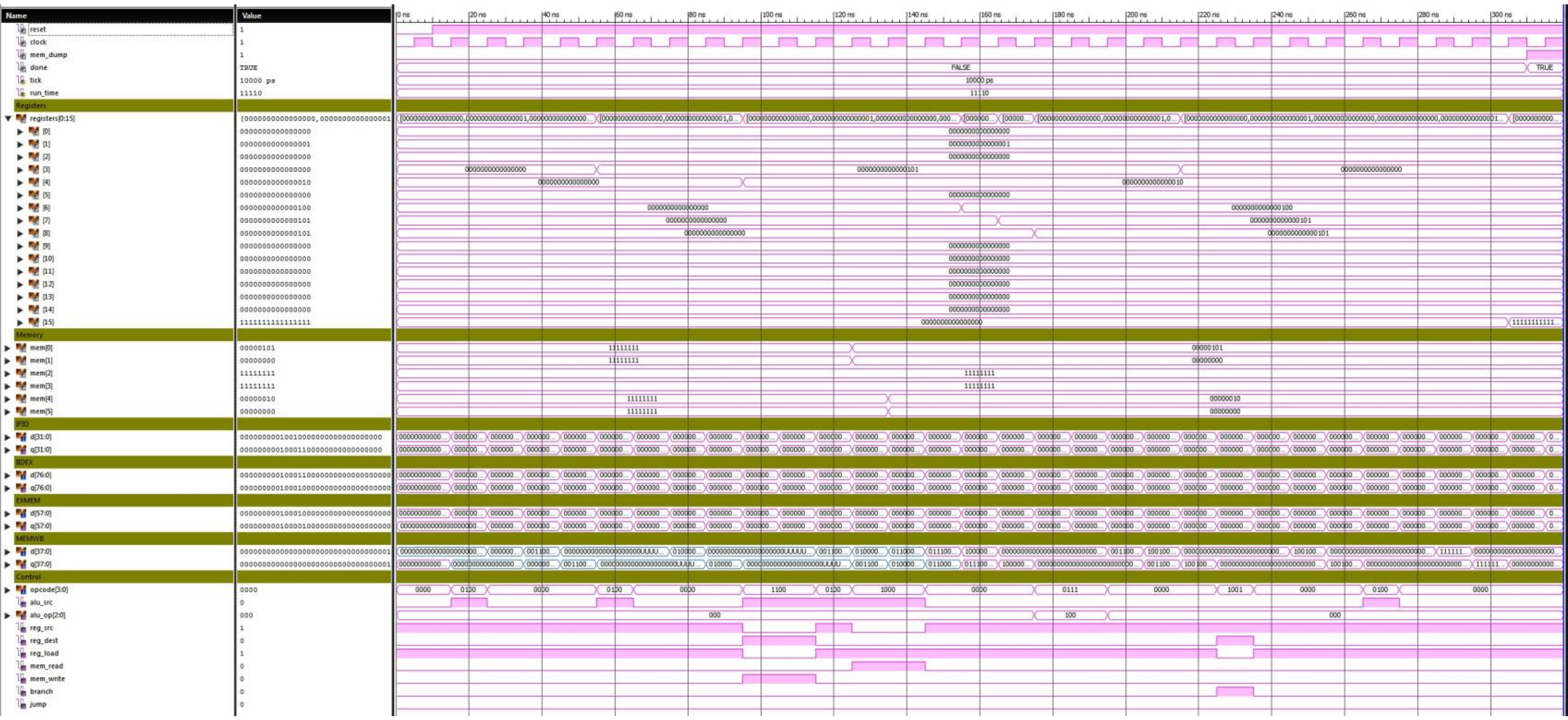
Since we are given the buffer VHDL code, the only implementation we need to do is in the system entity as well as changing the instruction_in memory file to accommodate for hazards. The main work was checking the diagram, going through and figuring out which signals I needed to replace with a buffer output signal. With these buffer output signals, the way they're constructed is by concatenating multiple signals together. For example, the first buffer has an input of d <= PC_count_plus_2 & instruction. So, in order to keep track of the signals, I constructed a table that accurately keeps track of which signal goes where within the buffer so it was easier to connect everything in the system entity. I have attached this table to the end of the report with the instruction_in file. Other than that, the implementation was as simple as going through and changing any signals, for example anywhere I had an "instruction" signal as an input, it needed to be changed to IFID_out(15 downto 0) or whatever bits are necessary. After going through the whole datapath doing this and adding the necessary nops to avoid hazards, that was all that was needed to be done to complete the pipelined datapath.

Experimentation

The difficulty in this lab was not the concepts or the theory involved in the implementation, it was the confusion that could come up very easily because of all these loose signals. Building a table for all the signals was essential in getting this lab correct; without that to keep track of which signal was where, this lab would have been very difficult to complete. For a long time I was trying to run the simulation with the same instruction set we had in Lab 6, which was obviously going to give me some errors as it had not accounted for hazards.

Once I started going through the instruction_in file and changed up the instructions is where I began to actually see the success of the implementation. I went through the code and found where we had RAW data dependancies and added 3 nops. Along with that, I also needed to add some nops after any branch instructions. Once I was here, I still could not figure out why my code was not working, however the final step had not been completed. To finally get everything working, I also had to go through and recount the instructions and change where the branch and jump instructions are pointing to, since adding the nops changed up the instruction addressing a good amount. After changing the branch and jump to their correct destinations, the code executed as it did in Lab 6.

Results



Below is the simulation waveform diagram for my Pipelined CPU.

It can be seen that, just as in Lab 6, we have all the registers and memory locations set to the values we desire. I included the buffer signals as well, not to keep track of each exact instruction, but to demonstrate that they have different lengths. It can be seen in the MEM/WB buffer there is a 16 bit portion that is undefined until a certain point. This makes sense, as the data_out on the data memory is undefined until the two SW instructions. We have register 7 and 8 = 5 and register 9 = 0.

Conclusion

In conclusion, the pipelined datapath works as intended and we are now able to execute instructions with a little more optimization. This lab was great at teaching how to modify an existing project was well as what goes into a pipelined datapath. I didn't think I'd have to spend as much time messing with the instruction_in file as I did, but turns out that was more than half the battle! Another useful piece of information I gained was how to do generic implementations, allowing the actual VHDL component creation to be made for many different uses.

Appdx

IF / ID (32 bits)
31 - 16 : PC + 2
15 - 0 : instruction

ID / EX (77 bits)

76 - 61 : PC + 2
60 - 57 : a_addr
56 - 41 : b_data
40 - 25 : c_data
24 - 9 : SE_output
8 : alusrc
7 - 5 : aluop
4 : branch
3 : readmem
2 : writemem
1 : regsrc
0 : regload

EX / MEM (58 bits)

57 - 42 : branch target
41 - 38: a_addr
37 - 22 : alu_result
21 : alu_neq
20 - 5 : reg_c_data
4 : branch
3 : readmem
2 : writemem
1 : regsrc
0 : regload

MEM / WB (38 bits)
37 - 34 : a_addr
33 - 18 : alu_result
17 - 2 : data_mem_out
1 : regsrc
0 : regload

```
00000101
01000011
00000000
00000000
00000000
00000000
00000000
00000000
00000010
01000100
00000000
00000000
00000000
00000000
00000000
00000000
00000000
11000011
00000100
11000100
00000100
01000110
01100000
10000111
00000000
10001000
00000000
00000000
00000000
00000000
00000000
00000000
10000111
01110011
01111000
01111001
00000000
00000000
00000000
00000000
00000000
00000000
00010111
10011001
00000000
00000000
00000000
00000000
00000000
00000000
01110001
01000111
00010001
10110000
00000100
10001010
00000000
00000000
00001111
01001111
```