CSCE 2214

Lab Report

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

William Taylor

010840237

L 002

Lab 7

4/29/20

**PRELAB**

In order to complete this lab we will need to understand the function of the data memory block, instruction memory block, and the program counter. We should also have a pretty good understanding of how an entire basic CPU works.

**INTRODUCTION**

The objective of this lab is to implement and explain the basic CPU that we have been working on for the 2nd half of the semester. We are going to using arithmetic-logic units (do basic arithmetic problems), AND gates (which looks at two or more values and if they are both high, the gate will output a high signal), a control unit (decodes instructions, and controls the rest of the CPU), Full adders (takes in three 1-bit binary numbers, and produces a sum, as well as a carry out value), Instruction Memory Unit (contains and distributes information about the instructions the CPU will run), Data Memory Unit (contains and distributes necessary information about the data the CPU is operating on), 2 to 1 Multiplexers (takes in 2 input values, and uses a select value to decide which one to push forward), a 3 to 1 multiplexer (takes in 3 inputs and outputs based on select signal), and registers (stores data for other units to operate on).

**APPROACH**

In our CPU we only have two actual external inputs. These inputs are our clock signal and our clear signal. Our instructions are all contained in a text file called Instr.txt. In this section I will be going through the entire CPU and explaining exactly what each part does.

First, we have the clock signal. The clock signal is one of the most basic, but important signals. Our clock signal for this CPU changes every 50 Pico seconds. You could also look at this as having a rising edge every 100 Pico seconds. The clock’s main purpose is timing. Timing is important to make sure that all the different sub-systems in our CPU have a way of staying on the same page. Without the clock signal we would have some parts of the CPU trying to push new instructions, while others are still working. This can cause many different types of faults and exceptions (if you are lucky). All of out units that are more complex than ALU’s or multiplexers will use a clock signal.

The next signal is the clear signal. The clear signal is another signal that is very important and pretty easy to understand. If the clear signal is ever ‘0’ any system that uses the signal will revert back to its original state.

The first component I am going to talk about is the control unit. I would say this is the most important component, but in a CPU this basic every component is the most important. The control unit has only one input. This input is the OP code. The opcode is the first 4 bits of any instruction in our CPU. The opcode is the most important part of the instruction. It tells the control unit what type of instruction it is. The control unit takes this information and tell the other parts of the CPU what to do with the information they receive. The control unit does not even receive or interact in any way with the information the rest of the processor will receive and operate on. In our CPU all possible opcodes are hardcoded. There are 10 possible operations for out CPU. These are Add, subtract, AND, OR, add immediately, subtract immediately, load word, store word, set less than, and else. Each option will result in the control unit telling the ALU, registers, and memory unit to act differently.

Each choice contains 7 output signals. The first is another opcode. This one is for the ALU and will cause the ALU to go through the same process. Think of this as a recursive function. The second signal is the ALU’s source. This goes to a 2 to 1 multiplexer. This is a 1-bit signal. It is either a zero or a one. I will talk about these options in the 2 to 1 multiplexer section, but basically it decides which number the ALU will have for its ‘B’ option. The third output also goes to a 2 to 1 mux ( a different one) this signal is called register destination. This one decides between the last 4 bits of the instruction and the 1st 4 bits after the opcode. It pushes its decision to the register file. The fourth output signal is the register load signal it tells the register whether it should load in new information or not. The only time it will not be a ‘1’ is when the opcode does not match anything (else) or when the register is told to store a ‘word’. The fifth output signal is the register source. This goes to a 3 to 1 multiplexer that decides which input will go to Sout. I will take about the 3 to 1 multiplexor in its own section. The sixth output is the memory read signal. This is either ‘0’ or ‘1’ this tells the data memory unit whether to read in new data or not. The seventh output signal is the last one. It tells the data memory unit whether to write data out or not (1-bit). This is the completion of the functionality of the control unit this is crucially important, because if the other units are not told what to do correctly, the output will not have a chance of being correct.

The next component I am going to talk about is the Register file. The register contains the information that all of the multiplexors and ALUs will operate on. The register has seven inputs and two outputs.

The first input is the clock signal. This is used to ensure that the register is in sync with the other components. The second input is the clear signal. If the clear signal is ever ‘0’ all sixteen of the registers will be replaced with a value of “0000”. The third input is a 4-bit binary number. This is the address of ‘a’. This is the register destination. At this address a 16- bit integer is stored; this is the data of ‘a’. Data of ‘a’ is the fourth input. The fifth input is the load signal. This is the only signal that comes directly from the control unit. As mentioned in the control unit section, it decides whether or not to load new data into the registers. The sixth and seventh inputs are addresses for ‘b’ and ‘c’ address of ‘b’ is the register source. And the address of ‘c’ is either the register destination or the register target (depends on the opcode). These both map to data of ‘b’ and ‘c’. However, unlike data of ‘a’, data of ‘b’ and ‘c’ are outputs. Data of ‘b’ goes into the 16-bit ALU. Data of ‘c’ goes directly into data memory. It also goes into a 2 to 1 mux as “Input 1”. If selected out of this mux, it also goes into the 16-bit ALU for calculations.

The next component I am going to talk about is the Instruction memory unit. This unit has six inputs and one output. The first input is the clock. This is used to make sure that the instruction memory unit is in sync with the other major units. This unit does not have a clear input. The second input is the “read enable” input. This is a 1-bit signal, which means it can only be a 1 or a 0. In the lab instruction manual we are told that we will always read in instructions from a file. So “read\_en” will always be ‘1’. The third signal is the write enable signal. The lab instruction manual also told us that this will always be a ‘0’. The fourth input signal is the address vector (16-bits). This comes directly from the program counter and it is the instruction itself. The fifth input signal is the data in vector. This always contains “0000”. The sixth and final input is the memory dump signal, this is a 1-bit signal. It is basically the equivalent of a clear signal. It will always be ‘0’, meaning the instruction memory will never be dumped.

The only output of the instruction memory is the instruction. This output gets sent to the control unit, register unit, a 2 to 1 mux, and the sign extension unit. Only the control unit reads the opcode (1st four bits). The register unit takes in the register destination (2nd four bits), the register source (3rd four bits), and the register target (last four bits). It uses them differently for each opcode. The sign extension unit uses the register target.

The next component I am going to discuss is the Data memory unit. The data memory is very similar to the instruction memory. It has six inputs and one output.

The first input into this system is the clock. The clock (“clk”) is used to make sure that the Data memory is in sync with the rest of the CPU. This makes sure it does not do anything it should not be doing at any specific tick. The second input is the read enable signal. This signal is 1-bit wide, meaning it is either on or off. The read enable signal comes directly from the control unit. If it is high (‘1’) the data memory unit will read in new data and if it is low (‘0’) the data memory unit will not read in new information. The third input is the write enable signal. This signal is 1-bit wide, meaning it is either on or off. The write enable signal comes directly from the control unit. If it is high (‘1’) the data memory unit will write out the current data it has and if it is low (‘0’) the data memory unit will not write out the current data it has. The fourth input is the address. The address is 16-bits and it comes directly from the 16-bit ALU. The result from the ALU is determined by the opcode, so the address that the data memory receives is hardcoded for each of the possible 10 instructions. The fifth input is the data\_in input this data come directly from the registers and is stored in data memory until the data memory unit is told to pass it along to the 3 to 1 multiplexer. The data that the data memory unit passes out is the one and only output of this system. This output is 16-bits as well and goes directly to input1 of the 3 to 1 multiplexer.

The next system to discuss is the program counter unit. This unit has three input and one output. This system is used to help the CPU keep track of how many instructions it has ran. The first input is clock signal. This signal is a 1-bit signal, meaning it is either a ‘1’ or a ‘0’. The clock ensures that the program counter is in sync with all of the other systems in the CPU. The second input is the reset signal. The reset signal is a 1-bit signal that is the exactly the same as a clear signal, just with a different name. To reset something back to its default is the same thing as clearing whatever was inside of it. The only output of this system is the “program counter output” this is a 16-bit signal. This goes directly to the instruction memory unit as the “address”. Every time this gets passed to instruction memory it tells it to get the next instruction at the next line. This is because every time it the signal goes through the system it adds a value of ‘2’. Adding two to the address indicates the next line. In order to implement this correctly we must use our third input “program counter input” to act as a register holding the next instruction address. The value of “program counter input” is the value of “program counter output” + ‘2’. These two signals loop until there are no more valid instructions to be read. The program counter is only useful because our instructions are stored in a text file. If our instructions were coming from an external source the program counter unit would be dead weight. Something to note about the program counter is that if it ever resets it will start again from the first instruction. There is no way to repeat an instruction, go back to a previous instruction, or skip one or more instructions. It can be predicted exactly what is going to happen in the CPU from the beginning.

The next system I am going to explain is the sign extension unit. This unit has one 16-bit input and one 16-bit output. The input is the 16-bit instruction, however only the last 4-bits are needed. But the overall result still needs to be 16-bits. The sign extension unit does this by turning the first 12 bits into the same value as the 13th bit, while keeping the 14th, 15th, and 16th bits the same. The result is a 16-bit representation of a 4-bit signal, while still having the same sign.

The next component is a 2 to 1 multiplexer. This 2 to 1 multiplexer is the register destination multiplexer. This multiplexer has 3 inputs and 1 output. The first input is “Input1”, it is 16-bits and contains the last 4 bits of the instruction (the register target (“rt”)). The second input is “Input2” and It contains bits 5 to 8 (the register destination (“rd”)). The third input is the select signal. It is one bit large, meaning it is either on, or it is off. On is represented by a ‘1’ and off is represented by ‘0’. If the select bit is ‘0’ Input1 will be pushed past the multiplexer. If the select bit is a ‘1’ then Input2 will be pushed past the multiplexer. The select signal comes from the register destination bit out of the control unit. Whichever one that is pushed past will be “Sout”. Sout is the same as whichever input was selected, except it is an output instead of an input. Sout is the address of ‘c’ in the register file.

The next component is another a 2 to 1 multiplexer. This multiplexer’s select bit is the alu\_source bit from the control unit. The first input is “Input1”, it is 16-bits and contains the data of ‘c’ from the register file. The second input is “Input2” and it contains the sign extended version of the last 4 bits of the instruction. The third input is the select signal. It is one bit large, meaning it is either on, or it is off. On is represented by a ‘1’ and off is represented by ‘0’. If the select bit is ‘0’ Input1 will be pushed past the multiplexer. If the select bit is a ‘1’ then Input2 will be pushed past the multiplexer. The select signal comes from the alu\_source bit out of the control unit. Whichever one that is pushed past will be “Sout”. Sout is the same as whichever input was selected, except it is an output instead of an input. Sout goes on to be the ‘B’ input for the 16-bit ALU.

The next component is another multiplexer. This one is a 3 to 1 multiplexer. It has 4 inputs and 1 output. The first input is “Input1”. Input1 is 16 bits wide and it comes from the output of data memory. The second input is “Input2” and it is the result of the 16-bit ALU. Input2 is also 16 bits wide. The third input is “input3”. Input3 is also 16 bits wide. It is the result of 16 0s ANDed together with the result from the 16-bit ALU. The main difference between a 2 to 1 multiplexer and a 3 to 1 multiplexer is the number of inputs. However you may notice that the select input should have 4 options (00, 01, 10, 11) but only three possible outputs. The solution is to make one of the options irrelevant. In this case we made the second bit irrelevant if the first bit is a ‘1’. S is the select bit, the final (fourth) input. S is 2 bits wide (4 possible scenarios). It comes register source bit directly out of the control unit. If select is “00” then Input1 will be selected. If select “01” then Input2 is selected. If select is “10” or “11” then Input3 is selected.

The final component to mention is the 16-bit ALU. The 16-bit ALU has three inputs and two outputs. The first input is ‘A’ and it is 16 bits. This is the data stored in register ‘b’ from the register file. The second input is ‘B’ and it is also 16 bits. This can be either the data stored in register ‘c’ from the register file, or the sign extended version of the last 4 bits of the instruction. Which one it is gets decided by the ALU source 2 to 1 multiplexer. The final input is the select input. It is 2 bits wide, meaning the 16-bit ALU can do four things. The 16-bit ALU is made up of 16 1-bit ALUs. Each ALU is wired up in succession to the next. Meaning the data from ALU 00 is used in the calculations of ALU 01 all the way through ALU 15. These port maps were made and mapped in lab 4. Depending on the select bits these ALUs all act different to do anything from simply add to comparing two different values.

**EXPERIMENTATION**

My code successfully did what it was supposed to. I have screenshots of the code I changed for each TODO.

A screenshot of a cell phone

Description automatically generated**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**A close up of a screen

Description automatically generatedRESULTS**

**A picture containing sitting, building

Description automatically generated**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instruction | Op | Rd | Rs | Rt | value |
| ADDI R3, R0, 5 | 4 | 3 | 0 | 5 | 5 |
| ADDI R4, R0, 2 | 4 | 4 | 0 | 2 | 2 |
| SLT, R11, R3, R4 | 7 | B | 3 | 4 | 4 |
| SW R3, 0(R0) | C | 3 | 0 | 0 | 0 |
| SW R4, 4(R0) | C | 4 | 0 | 4 | 4 |
| ADDI R6, R0, 4 | 4 | 6 | 0 | 4 | 4 |
| LW R7, 0(R6) | 8 | 7 | 6 | 0 | 0 |
| LW R8, 0(R0) | 8 | 8 | 0 | 0 | 0 |
| ADD R9, R7, R8 | 0 | 9 | 7 | 8 | 8 |
| SLT R10, R0, R1 | 7 | A | 0 | 1 | 1 |
| SLT R10, R1, R0 | 7 | A | 1 | 0 | 0 |
| OR R5, R10, 7 | 3 | 5 | A | 9 | 9 |
| SUBI R10, R5, 7 | 5 | A | 5 | 7 | 7 |
| SUB R11, R10, R7 | 1 | B | A | 7 | 7 |
| SW R11, 5(R8) | C | B | 8 | 5 | 5 |

My design is correct, because it displays the correct results based on the inputted instructions and information. There was one bug with my results. In the waveform section my waveform would not go past 1000 nano seconds. After trying lots of different things to fix the problem, I decided to change the tick time from 100ns to 50ns, so that all of the instructions would run in under 1000ns. Other than this one bug my program produced all the correct results.

For the graph above the previous paragraph, I looked at the data\_out line in the waveform. Each digit corresponds to a Column in the graph. For value, I copied the digit used for “rt”.

A screen shot of a computer

Description automatically generated

**CONCLUSION**

In these last couple of labs I learned lots of valuable information. From understanding how a simple ALU works all the way to. Understanding how to code and map an entire functional CPU. This is evident by the correct results in the waveform above. I learned lots of skills both with VHDL and computer hardware in general that I am sure I will be using in just about all of my future computer engineering classes, and even beyond them into the real world.

Despite understanding how the CPU works, there is still lots of improvement that can be made. Such as finding a way to have the CPU run instructions that are not hard coded in a text file. There are also a lot more systems and units that can be added to our CPU to make it more than just a very basic calculator.

Throughout these labs I have run into numerous bugs and problems. What I found to be the most efficient was working on a little piece at a time and debugging that before moving on. Another helpful tool is being able to take a step back and try to think about what the CPU was supposed to be doing. By simply taking a step back and not over complicating things these labs actually were not too difficult.