Single Cycle CPU - First Group Project

Group 5
Ang Shen
Jordan Pounder
Tianyuan Zhao

Introduction

In computer architecture, a major component of the computer is the processor. This unit essentially is in charge of making computations and carrying out instructions for the computer. For this project, we were tasked with designing a central processing unit that completes instructions within a single cycle. To implement the given instructions, we had to design different components, the major ones being the control unit, the data memory, the instruction memory, a register file, a program counter, and an ALU. Along with these, different extenders and adders were used along the way to carry out certain instructions.

Control Unit Summary

Part I Design

Control unit is one of our biggest components that requires designing and testing. A control unit should take the *opcode* of the instructions as input and generate control signals for each *multiplexer,* D-mem, Register file and ALU. Because different instructions may have the same operation at ALU, we designed a component named ALU control unit, which takes operation signals from control unit and the *function* field of each instruction as inputs and generates the corresponding signals for ALU. For example, *addi* and *lw* will have the same operation signals (00) meaning adding, but different signals for the ALU. In such a way, it is much easier to debug because signal generation is divided into separate stages.

Before building the control unit and ALU control unit, we started by creating a table including all instruction *opcode* and control signals for our single cycle CPU. As shown in the table below:

- RegDst is the control bit to choose the write back register between Rd and Rt;
- ALUsrc is used to select signals between data from the register file and data from the extender;
- MemtoReg chooses the write back resource from either D-Mem or the ALU;
- NotBranch is set when the fetched instruction is *bne;*
- Branch is set when it is a branch instruction;
- ALUop are the operation signals such as addition, subtraction etc.
- func is the *function* field of the fetched instruction;
- ALUctr are directly connected to ALU for calculating.

insruction	opcode	RegDst	ALUsrc	MemtoReg	RegWrite	MemWrite	NotBranch	Branch	ALUop	ALUop	func	ALUctr
addi	001000	0	1	0	1	0	0	0	Add	00	XXX XXX	0010
lw	100011	0	1	1	1	0	0	0	Add	00	XXX XXX	0010
sw	101011	X	1	X	0	1	0	0	Add	00	XXX XXX	0010

beq	000100	X	0	X	0	0	0	1	Subtract	01	XXXXXX	0110
bne	000101	X	0	X	0	0	1	0	Subtract	01	XXXXXX	0110
add	000000	1	0	0	1	0	0	0	R-type	1X	100000	0010
addu	000000	1	0	0	1	0	0	0	R-type	1X	100001	0010
sub	000000	1	0	0	1	0	0	0	R-type	1X	100010	0110
subu	000000	1	0	0	1	0	0	0	R-type	1X	100011	0110
and	000000	1	0	0	1	0	0	0	R-type	1X	100100	0000
or	000000	1	0	0	1	0	0	0	R-type	1X	100101	0001
sll	000000	1	0	0	1	0	0	0	R-type	1X	000000	1000
slt	000000	1	0	0	1	0	0	0	R-type	1X	101010	0111
sltu	000000	1	0	0	1	0	0	0	R-type	1X	101011	1111

Part II Build

In the building stage, the main issue we were concerned about was the performance of our design. The circuit should not take too long delay at each gate and components. The CPU should easy to debug, to build with relatively low cost. We ended up by using PLA (Programmable Logic Array) for both control unit and ALU control unit, which using AND gates to decide which instruction it is, and OR gates to generate each control signal bit.

Part III Test

In the testing stage, we have to test the control unit and the ALU control unit. For the control unit, we provided the *opcode* for all the instructions and compared the waveform with data in the table above. For the ALU control unit, we tested it with all legit combinations of inputs. We conclude our design working well because all waveforms matched with the table.

Waveforms of the Traces

Trace I *unsigned_sum.dat*:

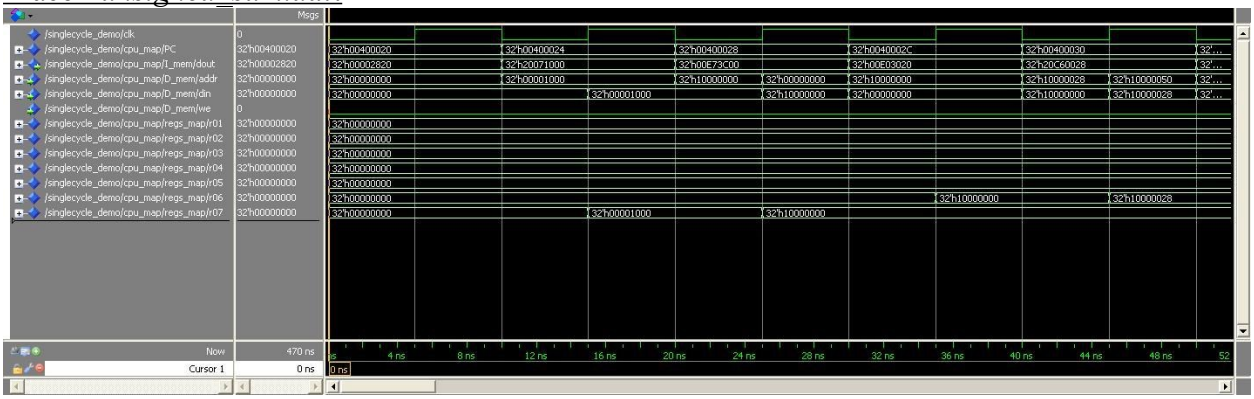

Figure 1. Start of waveform for *unsigned_sum.dat*

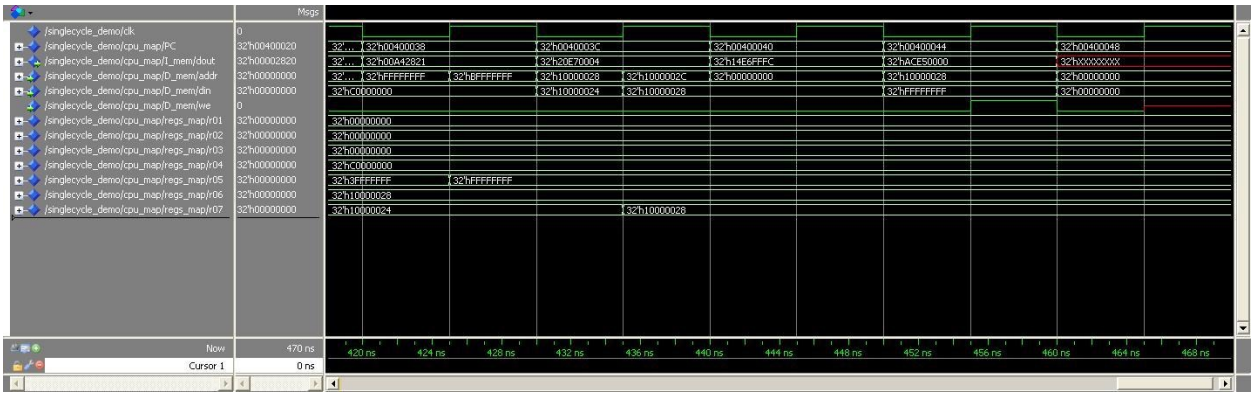

Figure 2. End of waveform for *unsigned_sum.dat*

Trace II *bills_branch.dat*:

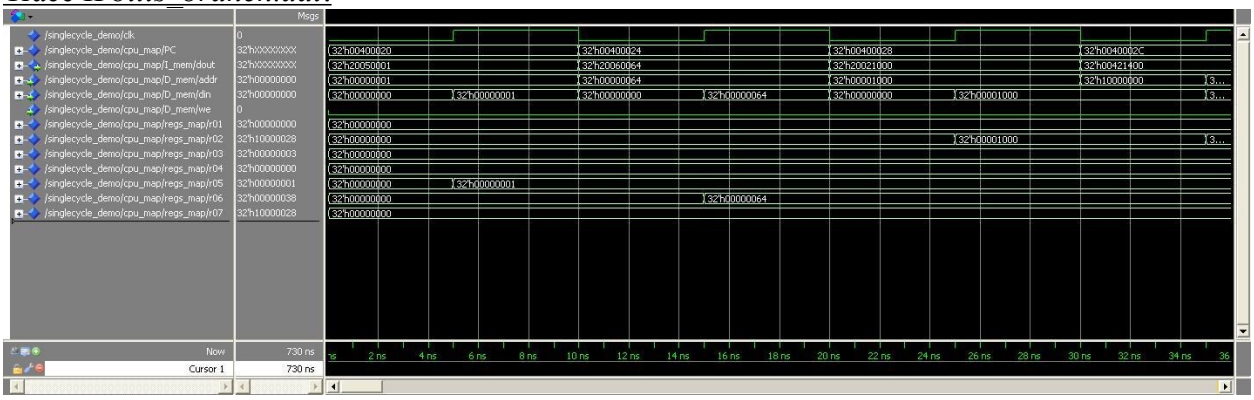

Figure 3. Start of waveform for *bills_branch.dat*

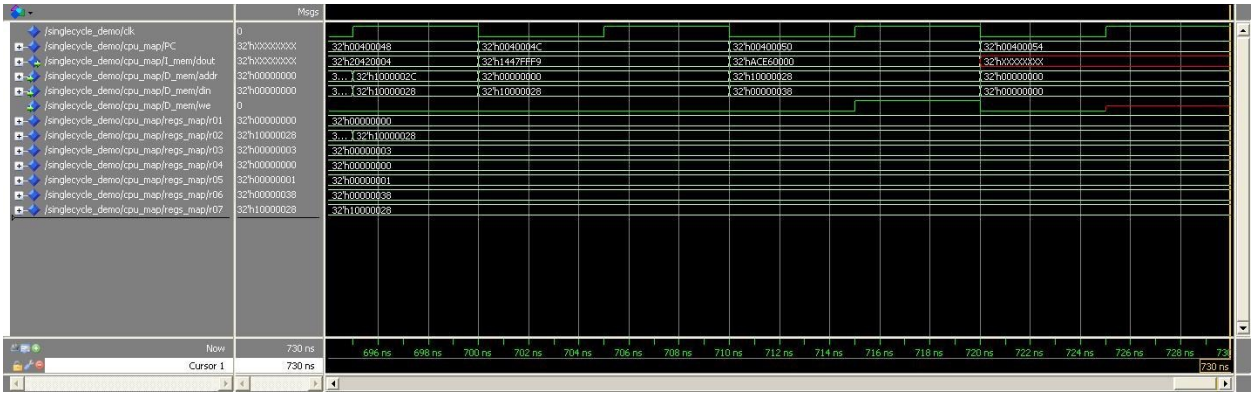

Figure 4. End of waveform for *bills_branch.dat*

Trace III *sort_corrected_branch.dat*:

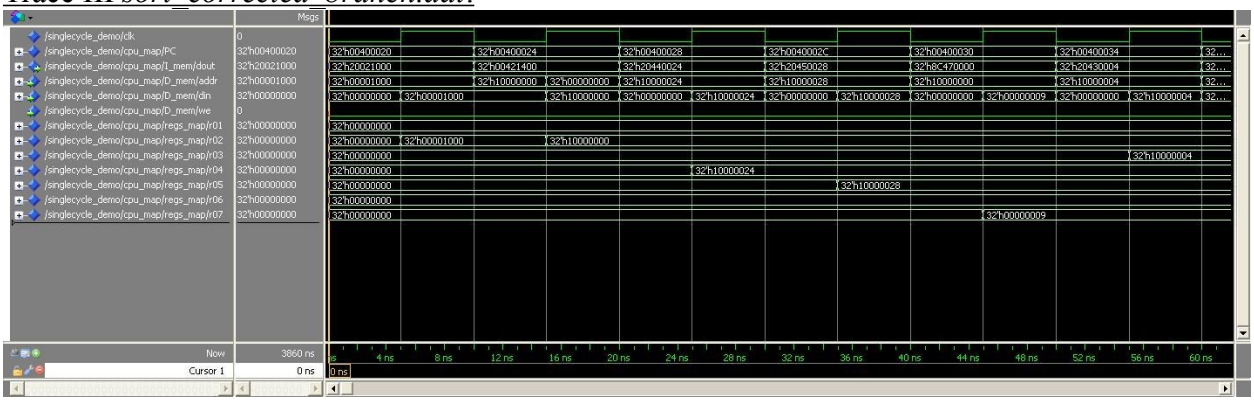

Figure 5. Start of waveform for *sort_corrected_branch.dat*

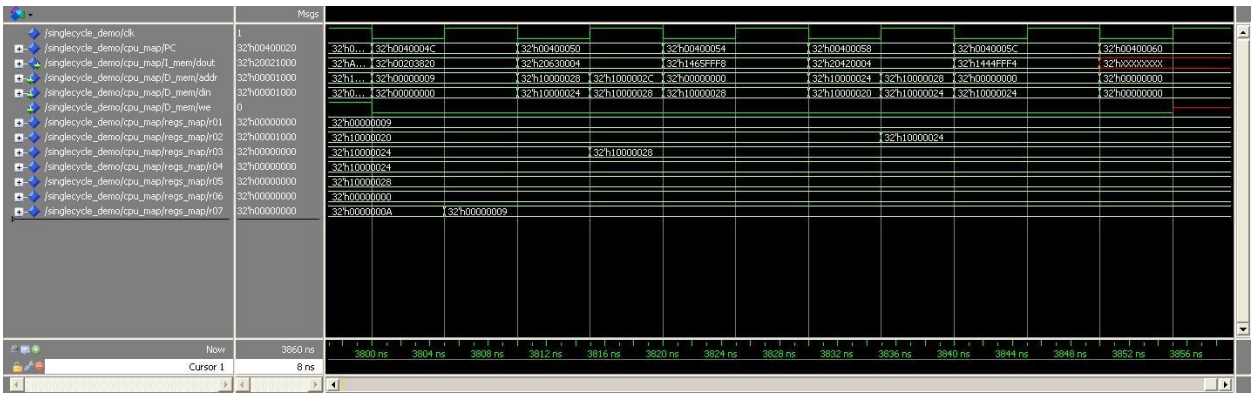

Figure 6. End of waveform for *sort_corrected_branch.dat*

Problems/Issue/Challenges

In the approach to our design, we started by brainstorming and listing the components we needed, based on the instructions we needed to implement. The design layout in the lecture notes helped us for this, and gave us the basic model from which we based our overall processor design from, making changes where necessary. For each component, we drew the smaller pre-

existing components that would combine to make them up, linking each port to each other, ensuring that the correct output and input signals were eventually generated. By designing the processor component by component, this allowed us to troubleshoot and debug each component individually, making it much easier to debug and troubleshoot when putting the overall processor together. However, we still of course ran into some challenges that we had to overcome.

One of these obstacles was the issue of clocking, and ensuring that all of our calculations were calculated and data stored back in the correct places at the end of a single clock cycle. To do this, we implemented different memory components, with write and read signals being triggered depending on different control signals, sometimes being the clock. Our register files consisted of dffr_a components, d-flip-flops that were triggered at the rising edge of the clock signal. If our a_init signal was raised, this was used as the arst signal for our register file, setting all the registers to zero, which is what we want at the beginning of a program. Our PC was also a dffr_a component; however this was triggered at the falling edge of the clock signal. The a_init signal in this case operated as the a_load signal for the PC, which loaded in the a_data value which corresponded to the starting address of our program's first instruction.

For our memory, we used sram components. Sram components are triggered by a write enable signal, and are independent of the clock. However, we passed the clock signal and the memory write signal from our control unit through an AND gate, the output was our write enable signal for our data memory unit. In such way, we built our own synchronized data memory which writes only at rising edge, and reads independently on clock. This is because consecutive store instructions would result in a memory write signal being raised for two entire clock cycles, which could result in data being stored in the wrong memory address. Therefore, passing it through an AND gate with the clock signal turns off the write enable signal for enough time to ensure the data is written to the correct address. Our Instruction Memory also was a sram component, however its write enable signal was always zero, since we only need to read from Instruction Memory. Once we had our clocking system working correctly, and the data flow was timed correctly to ensure that there were no hazards along the way, we did not run into many other problems. We had a fully functional ALU from the previous project, so the main challenge was designing the memory and clocking systems.

Conclusion

In summary, our team was able to successfully design a single cycle processor that implements the instructions that were required of us. Taking in the clock signal and an initial reset signal, our processor is able to reset its state (so that the PC starts at the first instruction of the program, and all the registers start off empty), and then run through the program, executing the instructions in the proper sequence. Our design process was successful because each problem that arose during the project was easily solved through debugging our components. By analyzing the waveforms of the signals after running a test bench, we were able to troubleshoot based on the known behaviors of our individual components. Our instructions were/are executing within one clock

cycle, and our memory / clock system was timed so that the data flowed in the correct manner, creating no hazards along the way.