ECE 3270

LAB 3 REPORT

June 13, 2022

Tim Driscoll
Clemson University
Department of Electrical and Computer Engineering
tdrisco@clemson.edu

Abstract

The main goal of this lab centered around creating a simple processor that could complete four different instructions. The processor was implemented using a combination of registers, a multiplexer, a add/subtraction unit and a general control unit. The general control unit was created using a mealy implementation of a finite state machine. Through the combination of the modular components the processor was created to perform two basic move instructions, an addition instruction and a subtraction instruction.

1 Introduction

Lab one consisted of four different parts that all combined together in order to develop a larger circuit which was implemented on the FPGA chip. Part one of the lab consisted of implementing a basic D flip-flop, which would be used as registers in the processor. The design of the D flip-flop was similar to the implementation used in lab one but included an enable bit. The second part of the lab involved creating a 10-to-1 multiplexer with a 10 bit selector that follows a one-hot encoding scheme. Each bit from the selector represents a different input to the multiplexer. In the final processor circuit the multiplexer functions as a way to determine which register's data gets placed onto the bus. The next step in the lab was to design the addsub unit. This logic component was used to carry out the add and subtract operations for two of the processors instructions. The last component that was created was the control unit which was implemented with a mealy finite state machine design. The control unit was used in the processor to assert all the necessary enable bits, multiplexer select bits and the add or subtract selection bit.These components were all combined in the final circuit to perform there designed instructions in Figure 1 and obtain the desired output. The final circuit was designed to obtain a 8 bit data input that was used to represent an instruction command. After the instruction command and based on some constraint the processor could carry out the desired command.

2 Design

2.1 Register Implementation

The registers were created using simple D flip-flops that included an enable bit. A D flip-flop is a relatively simple logic component that is used to update data from the input to the output based off of the rising or falling edge from a clock and the assertion of an enable bit. When the desired clock edge is detected and the enable bit is high the D flip-flop latches, setting the current input to the output. If the input is changed it will not update the output until the desired clock edge is received. In the lab the D flip-flop was implemented to latch on a rising edge if the enable bit was set high. This was done using a simple if statement that checked for a rising edge and the value of the enable bit. If both the constraints were detected then the input would be assigned to the output. When this design was implemented in the final processor circuit it was used to represent the instruction register, the 7 data registers, a register for the addsub input and a register for the addsub output. All these registers are implemented to hold data and are all tied to the same clock, for the board implementation the clock was mapped to a KEY on the FPGA. The enable bit inputs needed for the register are all mapped to outputs from the control unit. The control unit is used to determine when each necessary enable is to be asserted.

2.1.1 Test of Register

The testing that went into the register was relatively low level and abstract. Since the register was implemented after the D flip flop that was designed in lab 1 the basic functionality was already confirmed. For the implementation of the register in this lab the only update to the original D flip flop was the addition of an enable bit input. This enable bit only resulted in a slight change to the constraint for the flip flop to latch. Due to these circumstances the tests done on the register because they were done during the proof of the final circuit. Since there were no issues in the final circuit implementation there was no need to backtrack and test the sub-components, including the register.

2.2 Multiplexer Implementation

A multiplexer is another relatively simple logic device that was implemented in this lab. A multiplexer is used to output a single piece of data from many inputs. The input is determined by selector where each input has a corresponding selection code. The multiplexer implemented for the processor design was a 10-to-1 multiplexer. In the final processor the 10 inputs came from the data input of the final processor, the 8 available registers and the G register which is used to store the output from the addition or subtraction operation. There was a series of 9 when else statements to assign the output to one of the input values based off of the value of the selector. Instead of the selector being 4 bits to represent the minimum number of bits to encode the inputs in straight binary, 10 bits were used to implement a one-hot encoding scheme which can be observed in Table 1. A one in the lower eight bits corresponds to the general data registers and the upper to bits correspond to the G register and the direct data input. In the final processor implementation the multiplexer is used to select which data is to be passed to the data bus.

Table 1: Select Input Correspondence

Select Bits	Input Correspondence
0000000001	Register 0
0000000010	Register 1
0000000100	Register 2
0000001000	Register 3
0000010000	Register 4
0000100000	Register 5
0001000000	Register 6
0010000000	Register 7
0100000000	Register G
Any Other Input	DIN (Default Case)

2.2.1 Test of Multiplexer

Similar to the implementation of the of the register the multiplexer design was based off of the multiplexer that was previously designed in lab 1. The basic functionality of the multiplexer was already confirmed and the only adjustments that were made involved the encoding scheme and the addition of more inputs. Through the test of the final processor implementation the multiplexers functionality could be confirmed through the observation of the internal signals.

2.3 AddSub Implementation

The addSub unit is a simple logic device that is implemented in process. The addSub unit has a select bit that is used to determine which operation is used. Once the operation is selected the, the two data inputs into the addSub unit are used with the selected operation. For the implementation used for this lab the IEEE numeric_std library was used. This library allowed for casting of the inputs as signed values. After both inputs were cast as signed the operation could performed. Lastly the value that was obtained from the operation (output) needed to be casted back to a STD_LOGIC_VECTOR. The implementation of this adder did not involve a carry out bit and was simply truncated to the width of the output bits. When handling subtractions that would result in a negative number the output is represented by the 2's complement of the absolute value of the result from the subtraction.

2.3.1 AddSub Testing

The addSub was the first element of the processor that had not been previously tested. In order to confirm the functionality of the addSub unit a test bench was created to confirm functionality. This test bench is represented by Figure 2. In order to test the functionality with the test bench the and easily verify the outputs only four bit data inputs where used. To simple additions were tested and the second one shows how the carry out bit was simply truncated by the output. The sel bit was then updated to a one which indicated that subtraction was to be performed. Two cases of subtraction were then verified, one where the result was positive and one where the result was negative. The negative output correctly displayed the output in 2's complement form. The addSub unit was indirectly further tested in the final implementation of the processor.

2.4 Control Unit Implementation

The final sub-component of the circuit that was created was a control unit finite state machine. The purpose of this control unit was to assess the value of the current state and the value of the input and determine how to update the processor, resulting in the design of a mealy FSM. The processor was updated by the control unit through a serious of assertions, which would result in various registers being enabled and multiplexer select bits being set. The mealy finite state machine that was created can be represented by Figure 3. Then by looking at Figure 4 the assertions for each state can be observed

when looking at the corresponding instruction input. The only state that is not shown in Figure 4 is T0, state T0 is not shown because in this state all instructions will trigger a transition to T1 and the only assertion is the enable bit into the instruction register. The state machine would progress through state based on the instruction and if it needed multiple clock cycles to complete. The mv and mvi instruction only need a single clock cycle but the add and subtract instructions take three clock cycles. The design of a mealy finite state machine involved two separate processes, the first process updated the current state based on the clock and the reset. The second process was to update the output and it was based on the input and the current state. In VHDL the design included these two processes that were modeled very similarly. In both processes were case statements checking the current state, which all head nested if statements to determine the instruction inputted. From this point the processes would respectively update the current state and the outputs.

2.4.1 Control Unit Testing

The control unit was tested through the creation of various test benches to verify how the the state machine reacted to different instructions. When using the test benches to verify the design of the control unit it was critical to make sure that the correct state transitions were happening. It was also critical to verify that the correct outputs were being asserted based upon the given state and instruction. The first test bench which is represented by Figure 5 shows the outputs of the control unit for a "mvi" command. The test bench proves that the control until took the correct amount of clock cycles (1) to complete the instruction, and that the correct outputs were asserted. The second test bench which is represented Figure 6 shows the output of the control unit for a "sub" command. The test bench verified that the control unit asserted the correct outputs in each state and took the correct number of clock cycles to finish. The other instructions were further verified through the completion and testing of the final processor.

2.5 Final Processor Implementation

The final step of the lab was to create the overall simple processor that is comprised of all the components above described in the above sections. The final processor was designed to look like the circuit that is represented by Figure 7. Since all the components in the circuit were already created this final implementation involved initializing all the components, declaring instances, and mapping internal signals to connect the circuit. The only inputs that were needed for the final processor were clock, DIN, run and resent. Run and resetn were inputted directly into the control unit. Run represented an enable for the state machine to start transitioning and resetn was the resent for the state machine. The DIN input represented the data in value that was to be used to input any data into the processor. Lastly the clock was just used to cycle through all clock triggered components. The outputs to the final processor included a BUS vector and Done bit. The done bit was asserted by the control unit to represent when an instruction reached its final state. The Bus was used to feed data through the circuit and represented the last thing selected

by the multiplexer. Since all the components were modular this implementation was easy to complete when using the circuit diagram in Figure 7.

2.5.1 Final Processor Testing

In order to test the simple processor an extensive test bench was used to test all the different possible instructions and verify the outputs. The first two instructions that were verified were the "mvi" and "mv" instructions. This test bench can be seen by Figure 8. Current_state, R0muxin, R1muxin and R2muxin are not input or outputs of the final processor but are shown in the test bench to enhance understanding and proper verification. This specific test bench shows that both the mvi and mv commands take one clock cycle. It can then be observed that the data which is being moved into R0 is coming from DIN. The value that is moved into R0 and saved can then be verified by the R0muxin signal. The same process can be traced for the mv command except that the data is being moved from register 0 and into register 1. The next test bench that was created was to verify the addition instruction, and is represented by Figure 9. This test bench is a continuation of Figure 8, explaining the values that are already in register 1 and 0. The test bench shows the use of another mvi command to initialize register 2. The value in register 1 is then added to the value in register and stored in register 2. The test bench shows that the addition produces a correct value and that the instruction takes the expected amount of clock cycles. Lastly the sub instruction was tested in the end of the original test bench. The sub portion of the test bench can be seen in Figure 10. The instruction subtracts register 1 from register 2 and stores the value in register 2. This operation restores register 2 to its original value prior to the addition. This test bench shows that the subtraction command executes correctly and in the correct number of clock cycles. The final processor was then mapped to I/O on the FPGA and further testing occurred in order to verify the final design. The final processor used the switches on the FPGA as data inputs, the KEY as a clock and the LEDs as outputs.

2.6 Clock Latency

One of the key design points in this lab was handling that clock latency of the different instructions, and how it varied from the first two instructions to the second two instructions. Both the mv and mvi instructions could complete execution in a single cycle whereas the add and sub instructions needed three clock cycles to complete execution. This variance in needed clock cycles needed was handled by the finite state machine as represented in Figure 3. Instructions mv and mvi would only cycle between states T0 and T1 whereas the add and sub instructions would cycle through T0-T3. Another issue that occurred as a result of timing and concurrent implementation were race conditions. In the final processor there was a few occurrences of race conditions leading to undefined behavior of the Bus. This occurred because all the bits were being set on the rising clock edge and causing a simultaneous transition and update to occur resulting in the bus being set prior to the data in the register was set. These race conditions were resolved by updating the data inputs on the falling edge of the same clock cycle.

3 Conclusion

The main goal of this lab was to design a simple processor that was to perform four simple operations. The goal was achieved by combining modular sub-components of the circuit together in a final implementation. Each component was individually created and tested to verify its correctness prior to its addition to the final circuit. The register and multiplexer were modeled off of previous designs and the control unit and addsub units were carefully created for this lab. Through testing of these sub-components and implementation in a final processor circuit the final goal was achieved. The final goal represented implementing the processor on the FPGA board using its I/O.

The lab provided a great learning experience and allowed for an application of many of the topics that were discussed during lecture. Some of the main lessons that were learned throughout this lab included how to implement a mealy version of a finite state machine in VHDL and how to better use modelsim to test and debug a circuit. This lab showed the effectiveness of taking time to plan out and design a modular circuit in order to work up to a larger circuit. Throughout this lab the only errors that occurred were minimal syntax bugs that were easily fixed, and the occasional issue of race conditions in the final processor test bench. Overall the lab provided a great learning experience through the process of implementing a simple processor.

4 References

M.Smith "Digital Computer Design" Clemson University Holcombe Department of Electrical and Computer Engineering, January 2020

M.Smith "ECE3270 Digital System Design Lab 3: Simple Processor" Clemson University Holcombe Department of Electrical and Computer Engineering, January 2020

W. Daily, R. Curtis Harting and T. Aamodt, Digital Design Using VHDL a systems approach. Cambridge University Press

5 Appendix

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

Figure 1: Processor Instructions

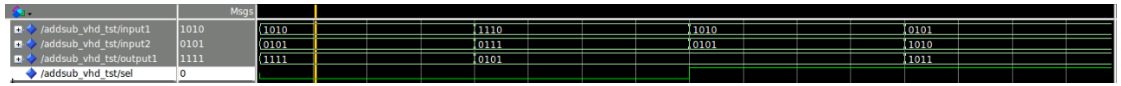

Figure 2: AddSub Test Bench

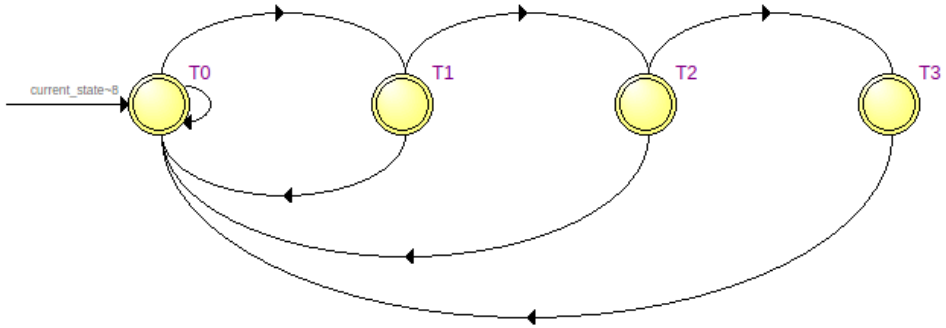

Figure 3: Mealy Finite State Machine

	T_1	T_2	T_3
(mv): I_0	$RY_{out}, RX_{in},$ *Done*		
(mvi): I_1	$DIN_{out}, RX_{in},$ *Done*		
(add): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	$G_{out}, RX_{in},$ *Done*
(sub): I_3	RX_{out}, A_{in}	$RY_{out}, G_{in},$ *AddSub*	$G_{out}, RX_{in},$ *Done*

Figure 4: State Assertions Table

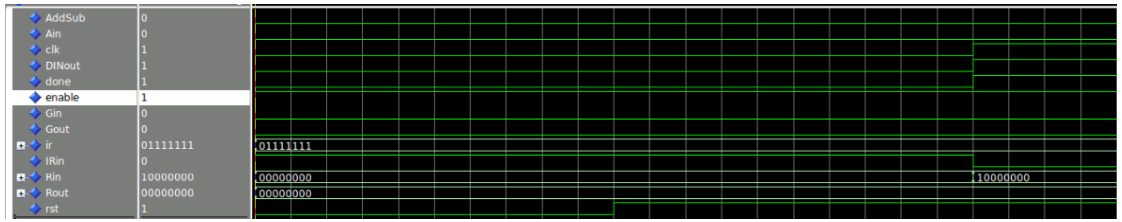

Figure 5: Control Unit "mvi" Test Bench

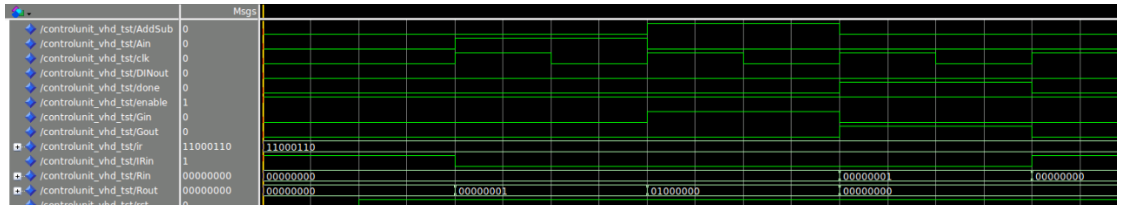

Figure 6: Control Unit "sub" Test Bench

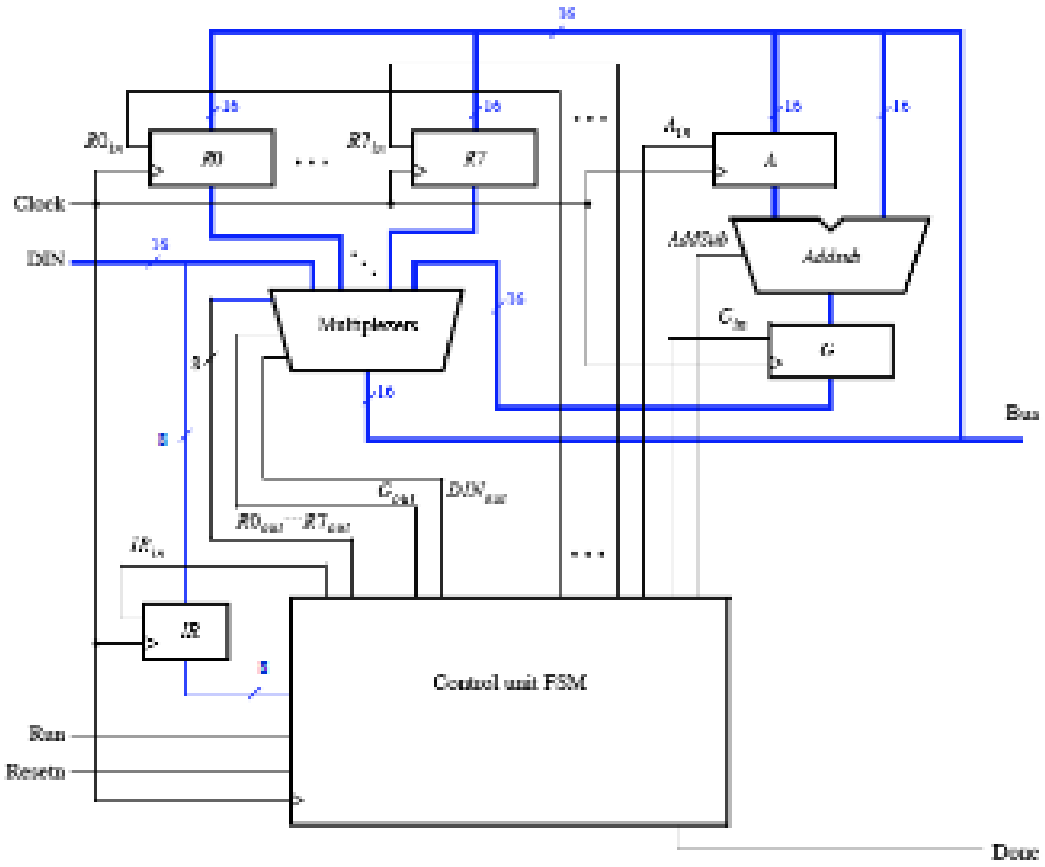

Figure 7: Final Processor Circuit

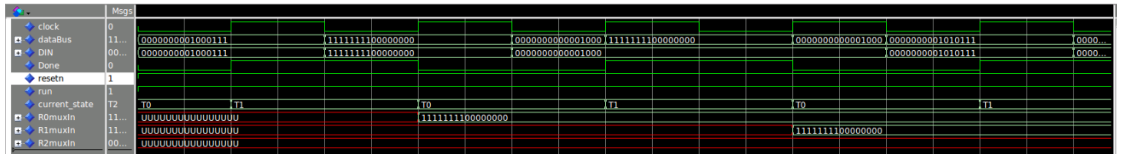

Figure 8: Final Processor "mvi" and "mv" Instruction

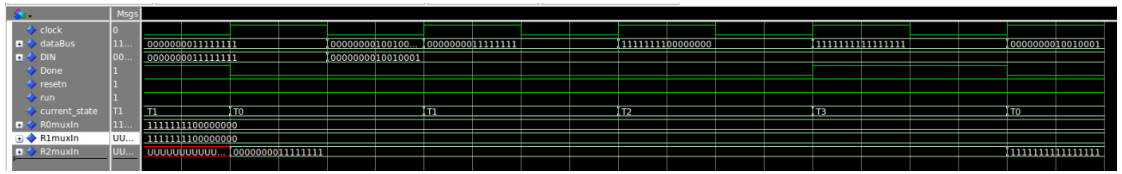

Figure 9: Final Processor "add" Instruction

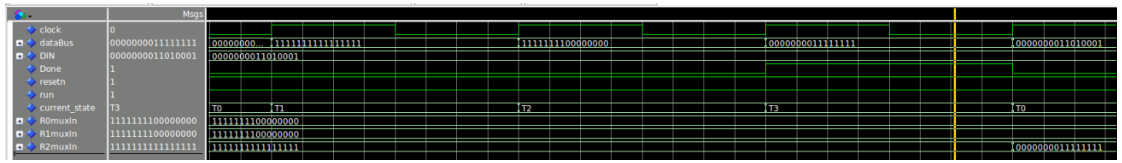

Figure 10: Final Processor "sub" Instruction