ECE327: PROJECT 3

November 3, 2019

Bradley Selee
Clemson University
Department of Electrical and Computer Engineering
bselee@g.clemson.edu

Abstract

Project 3 combined projects 1 and 2. The goal was to create a simple processor which performs four instructions: MV, MVI, Add, and Sub. This was accomplished by instantiating several registers, an adder, and a multiplexer all controlled by a Mealy Finite State-Machine (FSM). Upon completion of the lab, the processor successfully completed each instruction. Values were moved from the data input into register 0 and register 7 using the MVI instruction. Then, both registers were added with the Add instruction with the result stored in the X register. Next, the MV instruction copied the value of the addition into register 3. Finally, the Sub instruction subtracted the value in register 0, from the new value in register 3. That result was also stored in the X register [2].

1 Introduction

Project 3 capstones the previous two projects. In project 1, the goal was to create individual components, then instantiate them to make a functional circuit. In project 2, the goal to create a Moore Finite State-Machine (FSM), which was tested in OpenCL. In project 3, the goal was to create several components, instantiate them, and, finally, control the circuit with an FSM.The main goal of Lab 3 was to create a simple processor. This introduced several new components like registers, adders, and mealy type state machines. The A, G and R registers, the multiplexer, and the AddSub all take a 16-bit generic input. This allows for the user to easily change the width of the data input.

Each component of the lab required a test bench, this verifies that each component works properly, ultimately reducing the number of errors. Once each component was verified, the components were instantiated and connected with signals to create the entire processors. The final processor was verified in a test bench to show each instruction properly working. Finally, the processor was wrapped in a board file to map the functionality to the FPGA. Once the project was uploaded to the FPGA, a final test was run to ensure the board produced the proper output upon the user's request.

2 Components

The simple processor designed in lab3 required several different components. There consisted of 11 registers, an adder, a multiplexer, and a control FSM. This can be seen in the figure below:

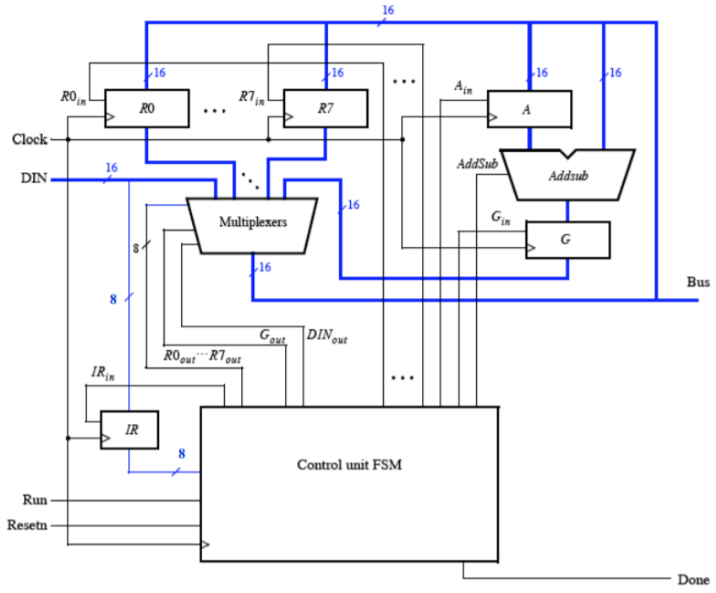

Figure 2.1: Lab 3 Simple Processor [2]

2.1 Registers

Registers R, G, and A are used to store a generic 16-bit data input. These registers store data until they are ready to be used. Registers R, store data from the input line until the multiplexer loads its value on the bus. Register A is used to load data from one of the R registers to perform the addition or subtraction to the adder. The A register is necessary because two values cannot be loaded on the bus in the same clock cycle, therefore, we need a place to store one of the values for the adder. Similar to the R register, the G register is used to store the result of the AddSub until the multiplexer selects its data to be put on the bus. These registers are generic, therefore, if the user requests, they can quickly change the data input width and add or subtract different size numbers.

Lastly, the instruction register (IR), takes an 8-bit data input to store the requested instruction. The IR takes the lower 8 bits from the data input, and sends this instruction to the control unit to be decoded. The 8 bits are encoded as such

> Your instructions will be encoded as **IIXXXYYY**, where **II** represents the instruction, **XXX** represents the **X** register, and **YYY** represents the **Y** register. [2]

Each register has inputs of a clock, an enable, and data input, and output of the data input. When ever the register is enabled and the it detects a rising clock edge, the register latches, storing the value on the data bus.

2.2 Multiplexer

The multiplexer serves to transfer data between registers by putting the data on the bus. By default, the multiplexer selects the data input of the system to be stored on the bus. All other values are decided by the control unit, driven by the instruction register.

The multiplexer takes 3 select inputs: one 8-bit vector to select the R registers, a single bit for the G register, and a single bit for the data input. It also contains 10 data inputs: eight 16-bit generic inputs for each R register, one 16-bit generic data input, and one 16-bit generic input for the result of the arithmetic. The mux outputted the generic width of the selected variable.

The 8-bit select was designed using one-hot encoding; however, the instruction encoding only uses 3 bits to decide which register to choose. Therefore, a function was created , *binary_to_oneHot()*, to convert the 3-bit binary value to an 8-bit one hot encoding. More so, this 8 bit vector allows for a *generate* statement to be used later on.

2.3 AddSub

The AddSub unit is essentially an adder, which can add or subtract two values. Typically, a ripple-carry adder is used to add and subtract binary numbers; however, VHDL is

smart enough to perform this mechanism in the background. As shown on page 229 of the text book including the library [1]

```
use ieee.std_logic_unsigned.all;
```

Allows the use of the + and - operators to perform the arithmetic. When an add instruction is sent, the AddSub will add the X_{data} with the Y_{data}. If a subtraction instruction is sent, the AddSub will subtract the Y_{data} from the X_{data}.

2.4 Control Unit FSM

The state machine controls the entire system. Due to the nature of the processor, the input being directly connected to the output, a mealy type state machine is the most logical and efficient state machine to use. Because mealy fsm's are typically less states than moore types, the processor FSM only required 4 states. This includes a loading state, and 3 states for each clock cycle

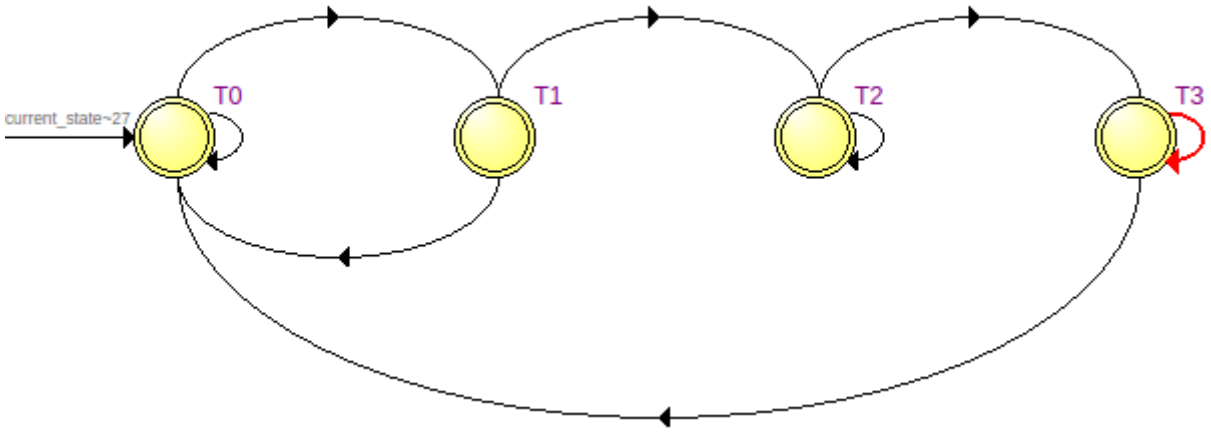

Figure 2.2: Quartus Generated Mealy State Machine

The FSM takes a 1-bit run, 1-bit reset, and 8-bit instruction for the inputs, and it outputs register selects, register enables, and 1-bit done signal. The FSM was created with two process statements, one to control the state and transitions, and another to assign the proper outputs based on the current state. The run bit was used to transition from state T_0 to T_1, then based off bits 7 and 6, **II** of the encoding, for the instruction vector, the states will transitions upon each clock cycle, according to the table in figure 2.3

The second process statement assigns the outputs of the system based on the current state. Similarly to the state transitions, the outputs were determined by figure 2.3. Before assigning the outputs, it was crucial that each output was cleared/disabled in order to prevent an implicit latch. Each state functioned very similar, regardless of the

instruction. For each state, a different register/data input was selected by the multiplexer to be loaded on to the bus, and the desired register to obtain the value was latched ONLY during that state. Due to the clock latency, the register would not necessarily latch the correct value on the bus; therefore, it was important to ensure that the state did not the register did not latch until the proper data was on the bus.

	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 2.3: State Transition and Instruction Table [2]

3 Testing and Verification

Once each component was created for the processor, it was crucial to ensure that each component worked as designed. This was accomplished by creating test benches for each component, then verifying the proper output was displayed on the wave form. Below shows the wave forms of each component

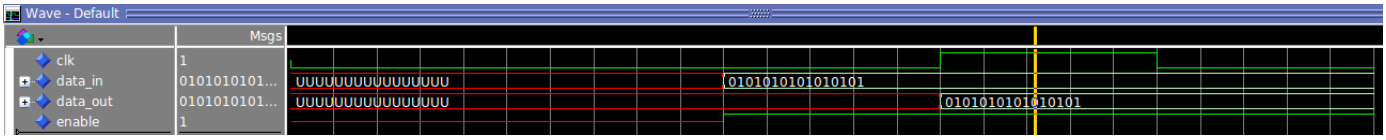

Figure 3.1: Register component wave form

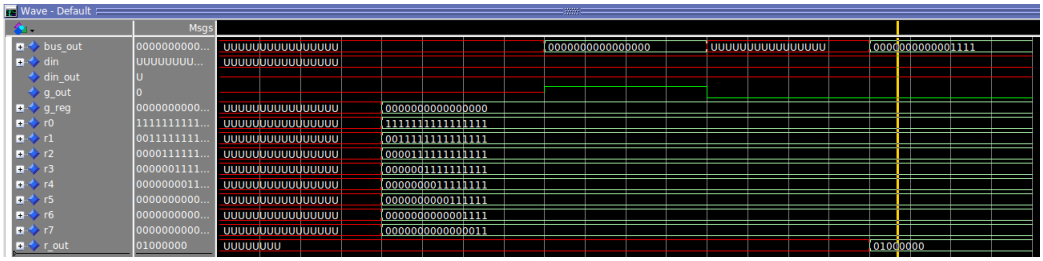

Figure 3.2: Multiplexer component wave form

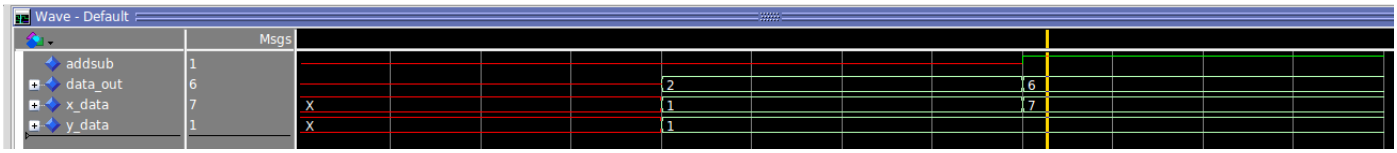

Figure 3.3: AddSub component wave form

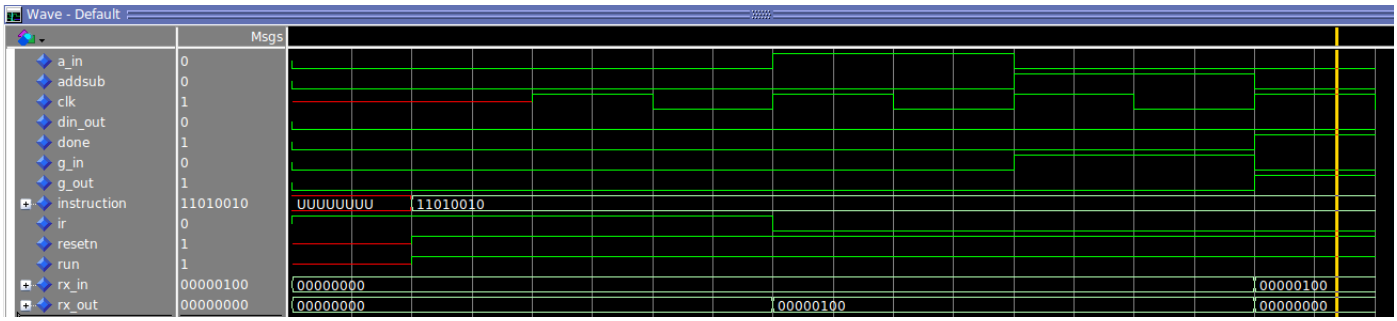

Figure 3.4: Mealy FSM wave form

4 Processor

Once all of the individual components were created and verified, the components were connected through signals to form the processor.

4.1 Architecture

Within the processor VHDL file, the I/O ports were created for the system. The processor has inputs of clock, data input, run, and reset, and outputs of the data bus, and a done signal. Then, each component was instantiated and port mapped to each other. The Mux, AddSub, FSM, and the A, G, and IR registers were all instantiated individually. The R registers were instantiated and port mapped using a generate statement to generate 8 registers; this allows for less code and errors when port mapping each component.

4.2 Testing and Verification

Once all components were mapped, Quartus was used to generate the block diagram to ensure that the components were properly connected show in figure 4.1

Once the components were properly connected, the entire processor was rigorously tested for each instruction to ensure the proper output was given on the bus. One test bench was created for the processor. This test bench begins by performing two MVI instructions to move the value 10_{10} into register 7 and the value 11_{10} into register 0. Then, the processor adds register 7 with register 0 and stores the result, 21_{10}, into register 7. Then the value in register 7 was copied to register 3. Finally, register 3 was subtracted from register 0 and the result, 10_{10}, was stored in register 3 (figure 4.2).

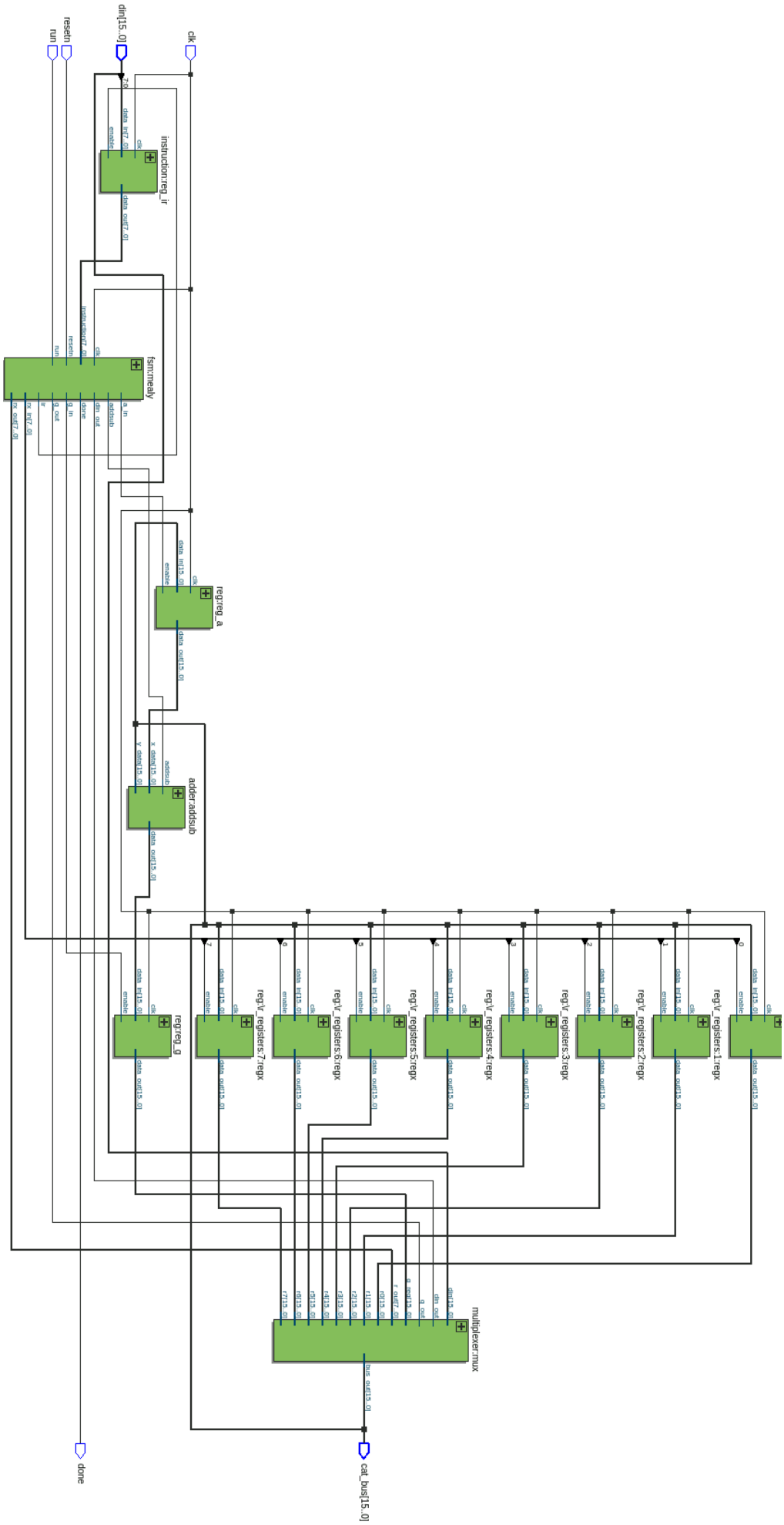

Figure 4.1: Processor created with individual components and signals

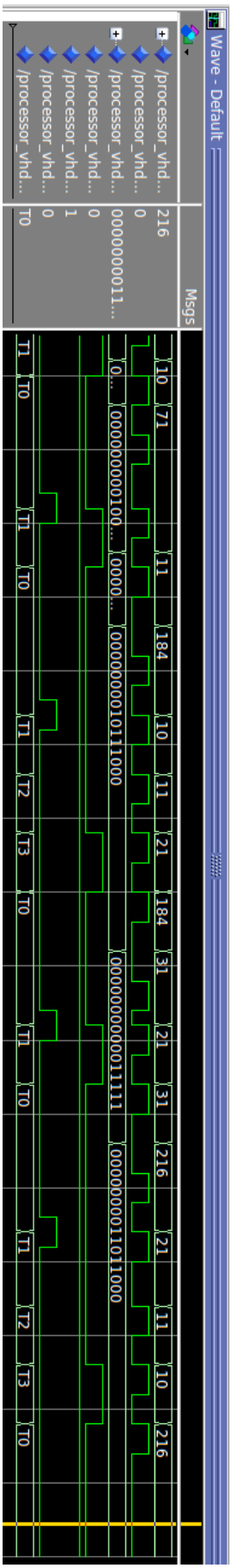

Figure 4.2: Processor wave form demonstrating its functionality

5 Discussion and Results

Lab 3 combines every concept from lab 1 and lab 2. The simple processor was built by creating individual components controlled by a state machine, and connecting every component through signals. This lab taught the importance of verification and testing. Many times throughout this lab I thought I had achieved the correct output, however, after testing different values there were small errors. More so, it was crucial that a solid understanding of the processor was obtained before creating and testing the components. Originally, I had the wrong concept on how the processor operated and how certain inputs, such as *run*, were supposed to be used. After many tests and code reworking, the processor was functional and outputting the proper values. Upon the completion of the lab, the user is able to add, subtract, and move any values (up to 16 bits) within the registers. The user is able to adjust the generic data input from 16 bits to 8 bits, depending on the value that needs to be added or subtracted. For a MV and MVI instruction, the *done* LED goes high during state T_1, and for an ADD and SUB instruction, the LED goes high during state T_3. Whenever the reset switch is a logic 0, the processor resets (active low).

6 Conclusion

Combining labs 1 and 2 greatly improved my understanding logical circuits to implement a functional design. My understanding of register latches and bus lines improved a lot from this, mostly because getting the time correct and knowing what was on the bus was essential. The biggest struggle I encountered in this lab was dealing with clock latency and latching the incorrect values from the data bus. I overcame this by physically drawing a timing diagram on paper, in order to understand the concept of when the registers needed to latch. I believe this lab was successful because my processor was able to perform all four instructions correctly. If I could redo this lab, I would have made sure I understood the concept of disabling the instruction register, in order to feed any value into the data input.

References

[1] W. J. Dally, R. C. Harting, and T. M. Aamodt. *Digital Design Using VHDL: A Systems Approach.* Cambridge University Press, Cambridge, 2015.

[2] M. Smith. *Lab 3: Lab 3: Simple Processor.* Clemson University, 2019.

7 Appendix

No further references were needed for this project.