SIMPLE PROCESSOR

Lab Report for ECE3270
Digital Systems Design

Submitted by

Ryan Barker

Department of Electrical & Computer Engineering
Clemson University

04/11/2015

Abstract

The goal of this experiment was to design, simulate, and export an eight-register processor to the Altera FPGAs. A dataflow diagram was provided as a starting point, and an ASM chart was drawn to facilitate design of the controller unit. The processor itself supported instructions for moving data between two registers, importing data from a data-in line into a register, and adding and subtracting two registers via an ALU. All instructions were encoded on the data-in line following the format "IIIXXXYYY", where "III" was the instruction (three bits allows for future expansion), "XXX" was the first register specified, and "YYY" was the second if applicable. Once the processor worked, a Quartus II wizard was used to build a memory circuit that ran with a counter. This showcased how the processor would behave in a typical computer. Quartus was also used to examine critical paths in the processor and produce recommendations to modify it to run at its maximum possible clocking frequency. The project as a whole stressed the ideas of preparation, organization, and modular design.

Introduction

The central focal point of this lab was to design the eight register processor shown in the dataflow diagram in Figure I.1 below. This goal was approached in pieces. Note that Figure I.1 is nearly identical to the provided diagram from the lab specification, the only difference is that the reset line is run to each register in the design as well as the controller unit, which was done so each could start with an initial value (all zeroes) when the system was initialized. Just like the last lab, the design was approached modularly and each entity was separated and tested independently from the rest of the circuit to ensure its functionality. The design process as a whole this time was much more straightforward, since ten of the eleven registers were identical, and all eleven were fairly simple to implement. Most of the challenge came in the state machine controller, but wrapper VHDL code was written to map the controller to its components and test its outputs independent of the rest of the circuit, which made troubleshooting it much easier.

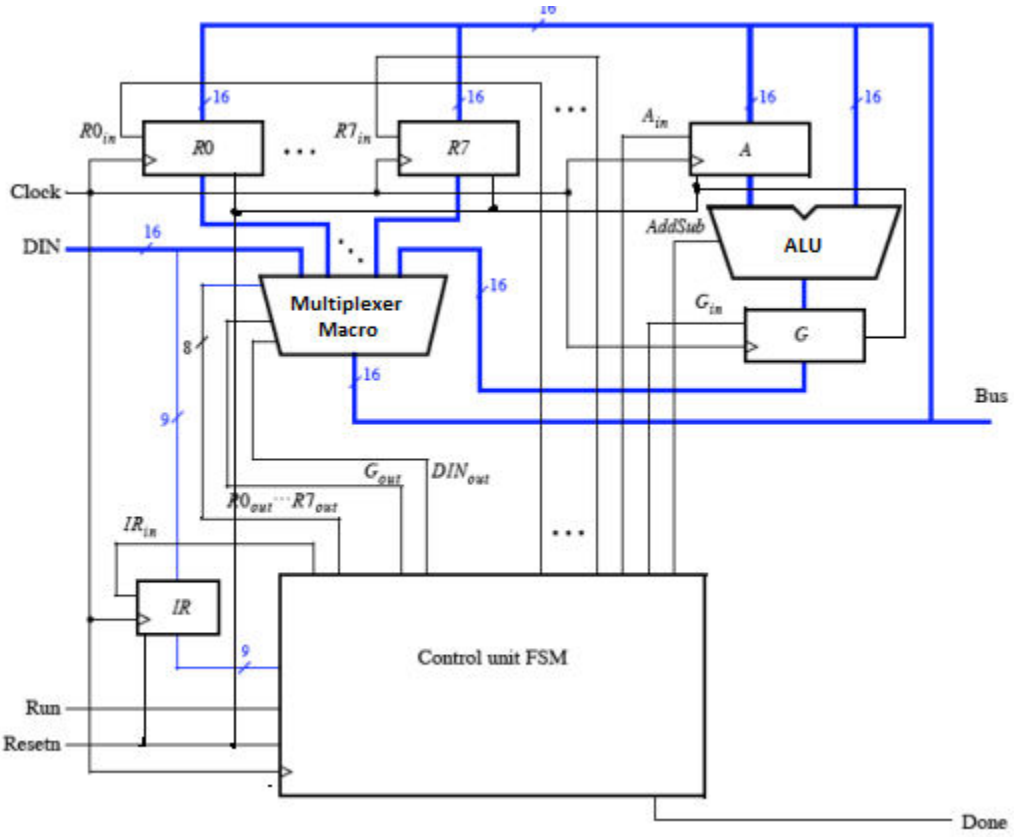


Figure I.1. Data Flow Diagram for Processor [1].

After the processor was tested and working, the Quartus II MegaWizard was used to design a thirty-two word, sixteen bit block of memory, initialized with a .mif file. A simple five bit counter was then constructed to drive the address input in this memory, and the main circuit in Figure I.2 was constructed. This showcased how the processor would run in a typical computer and allowed for much simpler use of the processor on the FPGA boards. Finally, TimeQuest Timing Analyzer was used to examine the processor, identify its critical paths, and produce recommendations to run the processor at its maximum possible clock speed.

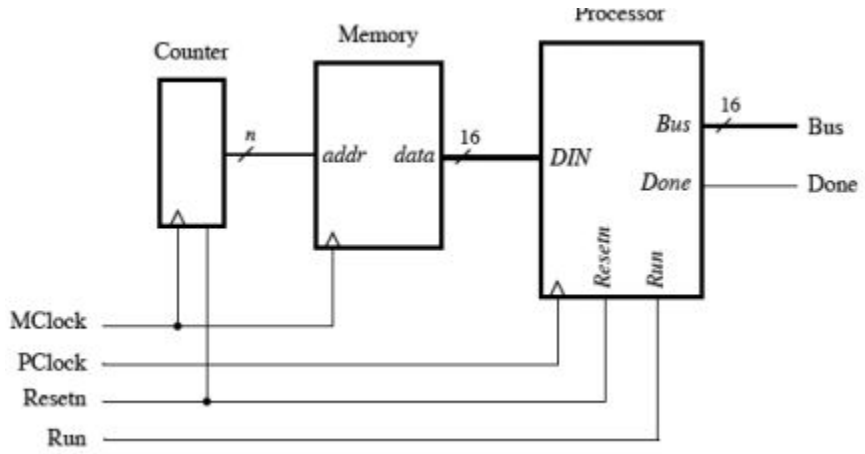


Figure I.2. Processor with Memory Circuit [1].

Section 1: The Registers

Section 1.A: Designing the Generic and Instruction Registers

One nice part about the overall design is that every register in it is simple and identical in functionality. The only unique register is the instruction register, which is nine bits instead of sixteen. This means only two VHDL entity and architecture pairs were necessary for designing all of the registers. The registers also follow the format of Register A from lab four very closely, so the generic design from lab four was used as a baseline for implementing all of the register VHDL code. Each register was set to be falling-edge activated for synchronization purposes. This way, the controller finite state machine could send instruction outputs on rising clock edges and they could be processed by the synchronous elements of the processor (namely, the registers) on falling edges. A low-active, asynchronous reset input was also added to every register in the design, which sets the output of each to zeroes and allows for the registers to be initialized when

the processor is reset. All VHDL code for the registers is seen in appendix A.1.1.1 and appendix A.1.1.2.

Section 1.B: Testing the Generic and Instruction Registers

Since their functionality was exactly the same, both register entity and architecture pairs could be tested with very similar test benches. Both tests began by setting an input value, setting the load signal high, setting the reset signal low, and waiting a clock cycle to verify the output was initialized to zero and did not change. Then, the reset was toggled high, and the test waited another clock cycle to observe the input being absorbed into the output of each register. After this, the input signal was changed and the load signal was toggled to ensure the input was only absorbed on falling clock edges when the load signal was high. Figure 1.1 shows this behavior for both registers. To enhance readability in Figure 1.1., the input and output lines in the Generic Register tests are show in hexadecimal and the input and output lines in the Instruction Register tests are shown in octal. Appendix A.1.2.1 and appendix A.1.2.2 show both test benches used in simulation.

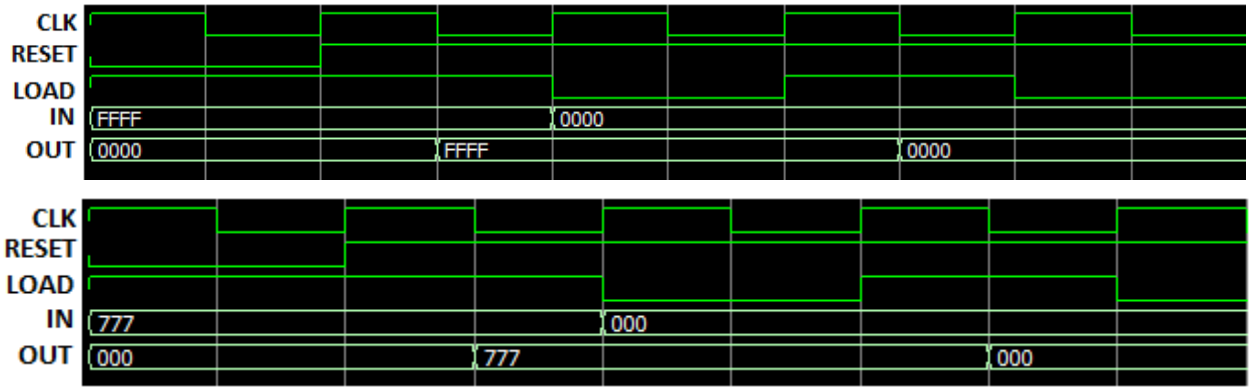


Figure 1.1. Generic and Instruction Register Simulation Waveforms (Top = Generic, Bottom = Instruction).

Section 2: The Multiplexer Macro

Section 2.A: Designing the Multiplexer Macro

The purpose of the multiplexer macro is to select one either of registers' 0 through 7 outputs, register G's output, or a sixteen bit data-in line and place it on the processor's bus. The

multiplexing is based on ten, single bit input lines from the controller as per Figure I.1. As will be discussed in the design of the controller unit, only up to one of these lines can be high at a time. If they are all low, the multiplexer macro puts all zeroes on the bus to prevent undefined bits on the bus. Most of the macro is implemented in a large IF-THEN-ELSIF-THEN-ELSE statement. All VHDL code for the multiplexer macro is seen in appendix A.2.1.

Section 2.B: Testing the Multiplexer Macro

Testing the multiplexer macro was very straightforward, since at its' core, it is simply a long chain of cascaded multiplexers. In the test bench, each input was declared to a unique value and the control lines were toggled once per a constant period of time to observe the multiplexer output switching to that input. Additionally, after every input was checked, the multiplexer control lines were left all low to observe the output of the multiplexer was zeroes. Figure 2.1 shows that the multiplexer macro exhibits this exact behavior in an easy to observe format. Again, for readability purposes, all inputs and outputs to the macro are shown in hexadecimal. Appendix A.2.2 shows the test bench used in simulation.

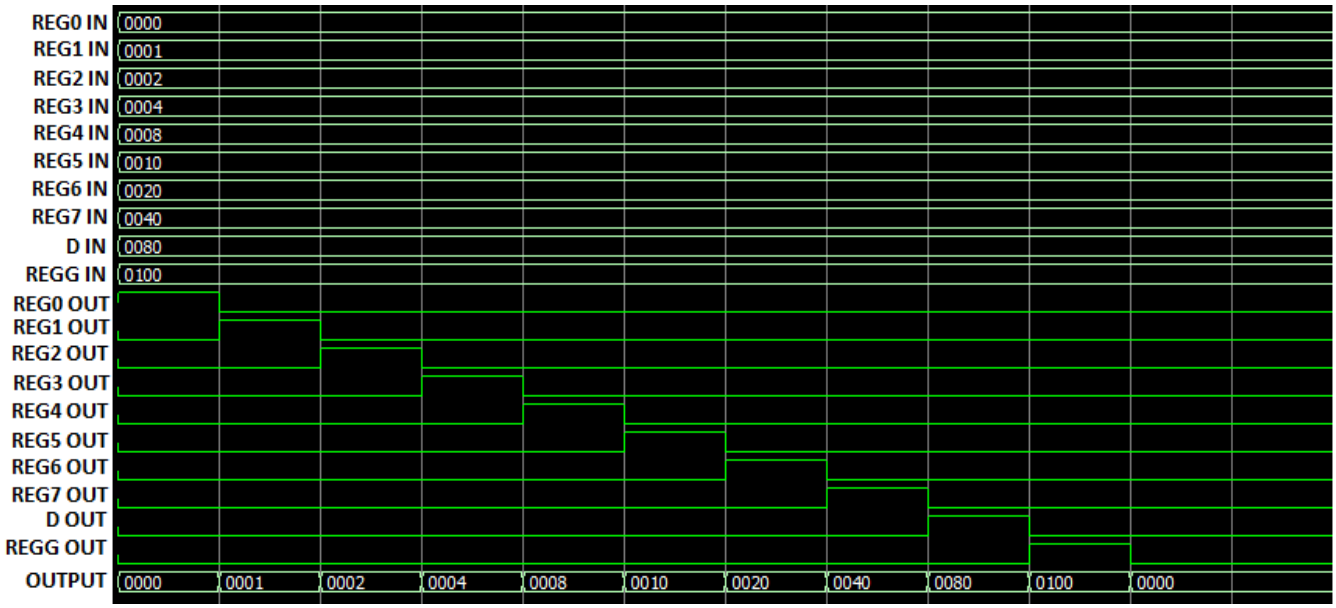


Figure 2.1. Multiplexer Macro Simulation Waveform.

Section 3: The Arithmetic Logic Unit (ALU)

Section 3.A: Designing the ALU

The ALU, along with registers A and G, is used by the processor to add or subtract the contents of two registers when an instruction beginning with "010" or "011" is processed. To perform arithmetic on two registers, the first register's content is first stored in Register A via the multiplexer macro and the bus. Then, the second register's content is placed on the bus, AddSub is asserted appropriately, the ALU performs the operation, and the result is stored in Register G. Lastly, the result is copied from Register G into the first register specified so it may be used in later computations.

Since it only needs to support addition and subtraction, the ALU in this design is nothing more than a sixteen bit ripple adder with a mode bit to toggle the operation into subtraction. To perform subtraction, the ALU takes advantage of the fact that subtracting two numbers A and B is the same as adding A to the two's complement of B. Since the two's complement of B is simply NOT B plus one and B XOR 1 = NOT(B), the two's complement of B is computed by placing XOR gates in between B and its input to the adder, tying each bit of B to one of each of the XOR gate inputs, tying the mode bit to the other, and tying the mode bit to the adder's carry in. Apart from this, nothing particularly interesting is happening in the ALU. All VHDL code for the ALU is seen in appendix A.3.1.

Section 3.B: Testing the ALU

The ALU was tested by loading several different values for the inputs, toggling the mode bit, and observing that the sum and carry out bits were accurate (though the carry out bit is ignored by the processor). The inputs started with simple examples, like 0x0001 + 0x0001 = 0x0002, and then tested boundary conditions likes 0x0001 – 0x0001 = 0x0000 and 0xFFFE + 0x0001 = 0xFFFF along with other examples. Note that toggling the mode bit makes each output in the circuit destabilize for an instantaneous moment in time, but this is not an issue, as the outputs stabilize rather quickly and the result of an operation is not stored until half a clock cycle after its inputs are set by the processor. Figure 3.1 shows the simulation waveform with the

desired behavior and the inputs and sum in hexadecimal. Appendix A.3.2 shows the test bench used in simulation.

Figure 3.1. ALU Simulation Waveform.

Section 4: The FSM Controller and its Decoder Components

Section 4.A: Designing the FSM Controller

Before design for the controller began, the ASM chart shown in appendix A.4.1.1 was drawn to obtain an idea of the structure of the code. This hardware is essentially the brain of the processor, so it was important to ensure it was designed correctly. The controller is designed with an asynchronous, low-active reset to reboot the processor immediately if it ever malfunctioned. The overall processor itself sits with the controller in its first state either while its' reset signal is low or its' reset is high and its' run signal is low, which causes it to load new instructions from the data-in line to the instruction register on every rising clock edge. Once the reset signal is high and the run signal is high, the state machine in the controller transitions into its next state and the processor begins executing the instruction stored in the instruction register. While it is executing an instruction, outputs corresponding to that instruction are asserted by the state machine on rising edges of the clock and are processed by the registers on falling edges so the processor itself is properly synchronized. The outputs asserted in each state and the transition pattern from each state to each state are shown in appendix A.4.1.1's ASM chart. Every sequence of outputs always ends with a state that asserts the Done signal for one clock cycle to indicate the instruction has finished. The controller currently supports move (III = "000"), move immediate (III = "001"), addition (III = "010"), and subtraction instructions (III = "011"). The controller macro itself was not tested until all of its components were built and tested, so it could be mapped to them, which will be discussed later in this section. For now, all VHDL code for the FSM controller is seen in appendix A.4.1.2.

Section 4.B: Designing the 3-to-8 Decoders

As a reminder, the instructions to the processor are all nine bits with the format "IIIXXXYYY", where "III" is the instruction to be performed, "XXX" represents the first register for the instruction, and "YYY" represents the second register for the instruction if applicable. Several instructions then use "$RX_{IN/OUT}$" and "RY_{OUT}" to specify which register to place on the bus and which register to store the bus's value in during computation. This presents a unique problem, since the outputs of the controller are not formatted as "$RX_{IN/OUT}$" and "RY_{OUT}", but rather $R0_{IN/OUT}$ through $R7_{IN/OUT}$. Thankfully, this problem is easily solved by sending the RX/RY bits of the instruction from the controller to two separate but identical and standard three-to-eight bit decoders, and taking input from each output line of the decoders. This feedback loop allows for the decoder outputs to be used any time "$RX_{IN/OUT}$" or "RY_{IN}" is output in a state. The decoders themselves are very simple and implemented in a straightforward Boolean architecture with outputs that are minterms of each's three bit inputs. All VHDL code for the 3-to-8 Decoders is seen in appendix A.4.1.3.

Section 4.C: Testing the 3-to-8 Decoders

Before the controller finite state machine itself was tested, a small and simple test bench was run on the decoders to ensure they behaved properly. Without their correct function, the state machine would never work. The test bench for the decoders simply runs through each value of the input and ensures the correct output line is the only output high. Figure 4.1 shows the simulation waveform with the input bits in octal and the desired behavior. The test bench for the decoders is seen in appendix A.4.2.1.

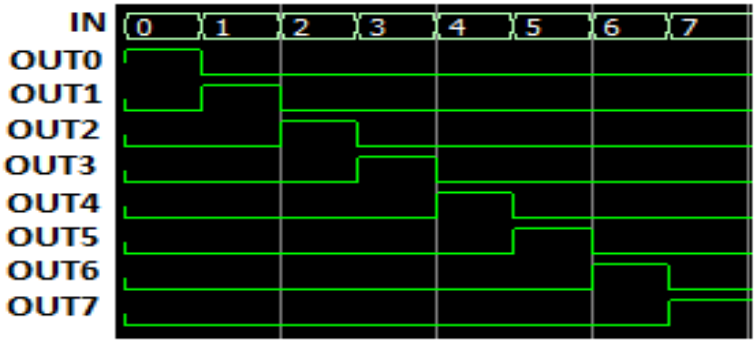


Figure 4.1. 3-to-8 Decoder Simulation Waveform.

Section 4.D: Testing the FSM Controller

Once the VHDL code used for both decoders was verified working, wrapper VHDL code was constructed for the FSM Controller, which mapped the controller to two decoders (One for RX, one for RY) and the instruction register. Each of the controllers' outputs were sent to an output on the simulation waveform, which allowed the designer to see each asserted output as the FSM Controller transitioned states. The test bench used to test this circuit verified reset functionality, and then sent one of each of the four different instructions to the state machine to ensure the outputs matched the ASM chart in appendix A.4.1.1 (utilizing the yellow cursor in ModelSim). The simulation waveform with the desired behavior is shown in Figure 4.2 with the instruction register output in octal. The VHDL wrapper code used is seen in appendix A.4.2.2 and the test bench for the wrapper code is seen in appendix A.4.2.3.

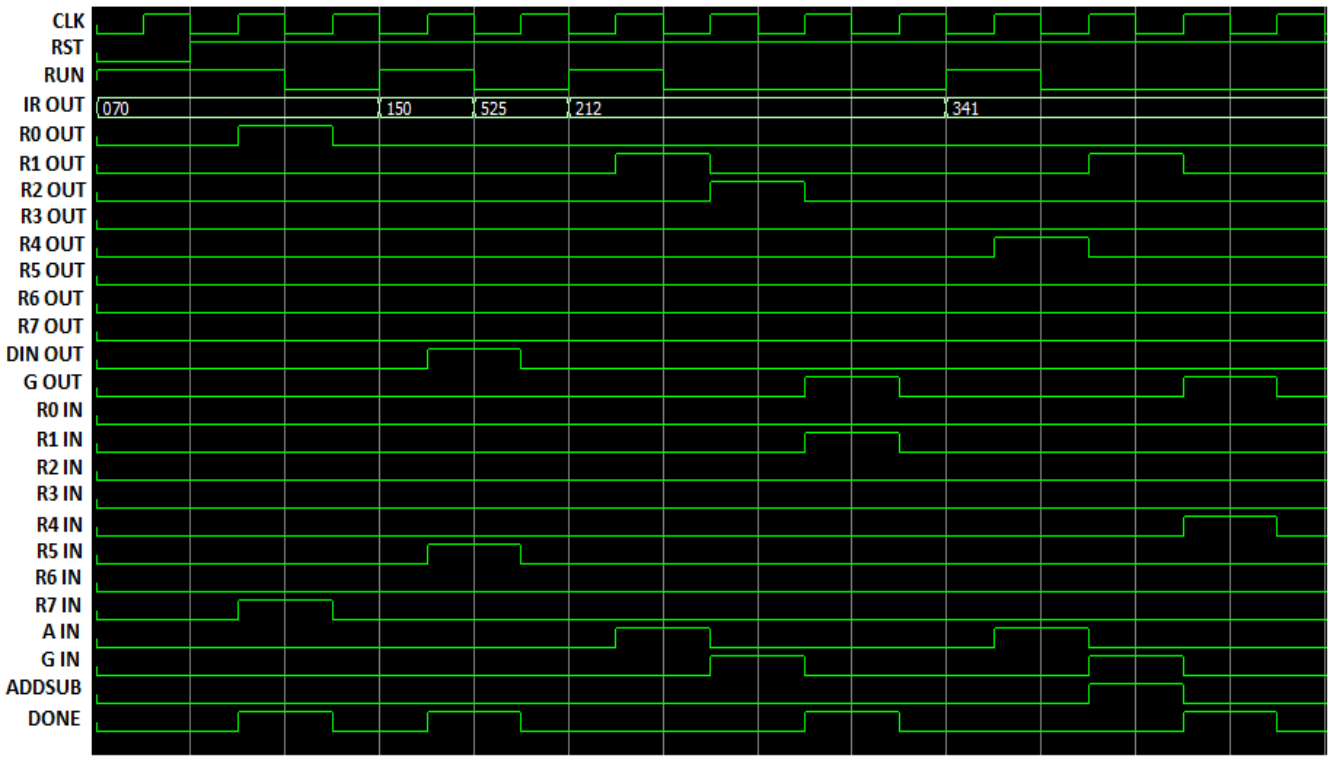


Figure 4.2. FSM Controller Wrapper Code Simulation Waveform.

Section 5: The Overall Processor

Section 5.A: Designing the Overall Processor

The overall processor itself contains a very large structural entity that port maps everything discussed so far into the circuit seen in appendix A.5.1.1. It would have been very difficult to test or troubleshoot components of the processor by themselves if not for the modular approach and through testing thus far. The processor still had to be tested to ensure proper top-level functionality, but this testing was made much more painless because of the previous tests. Also, so the processor could be verified as working during testing, outputs were added to its VHDL code that correspond to each register's output, so they may be observed in simulation. As specified in the lab manual, another VHDL file which port maps the processor to switches, keys,

and LEDs on the DE115 Altera boards was written so it could be exported and tested after simulation passed.

A very important thing to note is that the state machine and registers in the processor are clocked on opposite clock edges. This is done because it is impossible for the registers to read the control lines from the controller (A_{IN}, G_{IN}, and R0IN through $R7_{IN}$) on the same moment that the controller changes them. However, clocking the registers and state machine on opposite edges means their output signals are not synchronized. This means there is a limitation on the processor, which is that during operation, it will assert the done signal half a clock cycle before the value from the instruction it executed is loaded into the corresponding register (See Figure 5.1). The processor is still fully functional; this just means the hardware using it must be aware of this fact. If this processor were to be built physically and sold, this information would need to be included in its specifications. All VHDL code for the overall processor is seen in appendix A.5.1.2 and all VHDL code to map the processor to the board is seen in appendix A.5.1.3.

Section 5.B: Testing the Overall Processor

As mentioned above, all of the testing up to this point was to ensure the processor would function when the numerous macros in the design were hooked together, so testing here focused less on the ability of each macro to perform their purpose, and more on the synchronization and correctness of the outputs. The test bench used tested the processor's reset, and then stepped it through one of each of the four different kinds of instructions. The output waveform is shown in Figure 5.1, with each generic register's and the data bus's outputs in hexadecimal and the instruction register's output in octal. As seen the first instruction processed is 100_8, or 001000000_2. This translates to a move immediate into register zero. Notice at the next clock falling edge, the data at the DIN line, 0005_{16}, is placed into register zero and done is asserted for a clock cycle. Likewise, the next instruction is 010_8, or 000001000_2, and at the next clock falling edge the data in register zero (0005_{16}) is copied into register one and done is asserted for a clock cycle. Next comes 210_2, or 010001000_2. This is addition between registers one and zero. Notice in the next three clock cycles, register zero's output is stored in register A, the addition happens and is stored in register G, the value in register G is copied into register zero, and done is asserted for a clock cycle. Finally, the last instruction is 310_8, or 011001000_2. This is a subtraction of register zero from register one, stored in register one. This happens in the next

three clock cycles, very similarly to the previous instruction, and then done is asserted for a clock cycle. Once the simulation verified the circuit was functional, the code in A.5.1.3 was sent to the board and sample instructions were entered with no issue. The test bench for the processor is seen in appendix A.5.2.1.

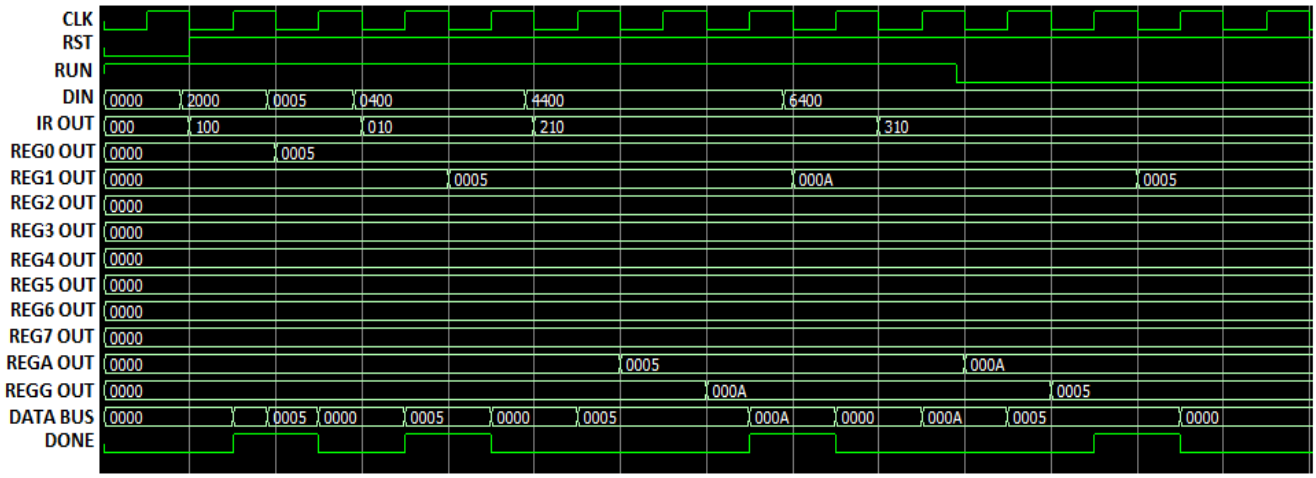


Figure 5.1. Processor Simulation Waveform.

Section 6: The Memory Circuit

Section 6.A: Designing the Remaining Components

Once the processor was deemed operational, the designer out of part one of the lab, into part two, and attention to the circuit in Figure I.2. Designing this circuit required the creation of two more VHDL files; one for a five bit counter and one for the memory component. Thankfully, both of these components were drastically simpler to implement than the processor.

Designing the five bit counter was relatively trivial, as it was almost the exact same as Register D from lab four, but even simpler. The counter was built with a low active, asynchronous reset, which initialized it to zero. Otherwise, the counter made use of an unsigned

signal in the architecture to simply add one to its' current count value and output it each clock cycle. A very tiny test bench was written to verify the counter effectively counted. All VHDL code for the five bit counter is seen in appendix A.6.1.1, the test bench is seen in appendix A.6.2.1, and the output waveform from simulation is seen in Figure 6.1. The output signal in Figure 6.1 is in binary.

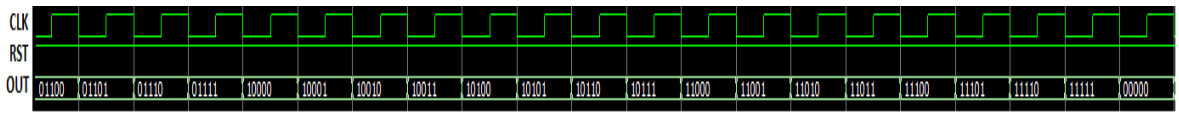


Figure 6.1. Counter Simulation Waveform.

Designing the memory component to the main circuit was as easy as following the steps in the lab five instructions to create a read-only-memory component that was sixteen bits wide and thirty-two words deep in VHDL with the MegaWizard and drafting a .mif file. Once this was done, everything was ready to be mapped together. All VHDL code for the memory in the circuit is shown in appendix A.6.1.2 and the associated .mif file, inst_mem.mif, is shown in appendix A.6.1.3.

Section 6.B: Designing the Overall Circuit

Thankfully, at this point in the lab, everything was built, so the only thing left to do was connect it all together. The remaining VHDL file, Lab5_Main.vhdl, does exactly that and outputs each register's value for verification in simulation. Just like in the processor VHDL code, one last VHDL file, Lab5b.vhdl, was created to map the main circuit to the specified switches, keys, and LEDs on the Altera Boards. Note that two clocks are used in the circuit to simplify testing the circuit on the board, but they are clocked at the same frequency in simulation. All VHDL code for the main circuit is seen in appendix A.6.1.4 and all VHDL code that maps the main circuit to the board is seen in appendix A.6.1.5.

Section 6.C: Testing the Overall Circuit

Testing the overall circuit was even easier than testing the standalone processor, since all of the instructions were already contained in the memory component and automatically sent to the processor by the counter. The only thing necessary to do in the test bench was test the reset signal, set run and reset high, and watch the magic happen. Figure 6.2 shows the simulation

waveform, which can be stepped through in a very similar manner to Figure 5.1. For instance, the first instruction processed is 100_8, or 001000000_2. This is a move immediate into register zero. Notice at the next falling edge of the clock, data is loaded into register zero (from the data-in line, which is not visible in simulation). This process can be repeated for every instruction shown. The test bench for the main circuit is seen in appendix A.6.2.2.

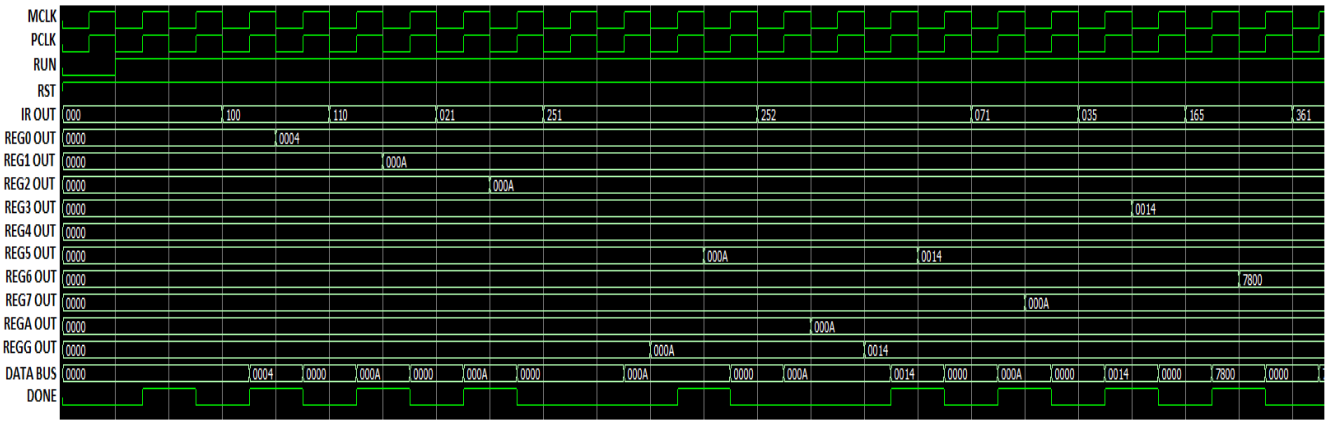


Figure 6.2. Main Circuit Simulation Waveform.

Part III and Conclusions

As a final step, the simple SDC file shown in appendix A.C.1 was written and used with TimeQuest to determine the critical path of the circuit. As seen in the appendix, the SDC file contains a clock operating at 100 MHz, set up ideally (Rises at 0 seconds, falls at 5 seconds) and minimal t_{su}. The generated TimeQuest slow setup report (to analyze setup time) is shown below. As seen below, the critical path in the circuit starts in the controller FSM, goes through the G_{OUT} output into the multiplexing circuit, through the bus, into the adder, and finally through register G to its output with a setup slack -8.800. Most of the worst-case paths start in the controller.

Slow 1200mV 85C Model Setup: 'clk'

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew
1	-8.800	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[15]	clk	clk	5.000	-0.049
2	-8.649	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[15]	clk	clk	5.000	-0.049
3	-8.565	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[13]	clk	clk	5.000	-0.036
4	-8.423	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[14]	clk	clk	5.000	-0.049
5	-8.414	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[13]	clk	clk	5.000	-0.036
6	-8.272	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[14]	clk	clk	5.000	-0.049
7	-8.083	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[14]~_Duplicate_1	clk	clk	5.000	0.003
8	-7.932	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[14]~_Duplicate_1	clk	clk	5.000	0.003
9	-7.786	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[11]	clk	clk	5.000	-0.040
10	-7.718	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[15]~_Duplicate_1	clk	clk	5.000	0.003
11	-7.695	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[13]~_Duplicate_1	clk	clk	5.000	0.016
12	-7.656	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[12]	clk	clk	5.000	-0.049
13	-7.635	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[11]	clk	clk	5.000	-0.040
14	-7.567	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[15]~_Duplicate_1	clk	clk	5.000	0.003
15	-7.544	Lab5_Ctrl:Ctrl|state.D	Lab5_Reg:RegG|output[15]	clk	clk	5.000	-0.049
16	-7.544	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[13]~_Duplicate_1	clk	clk	5.000	0.016
17	-7.505	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[12]	clk	clk	5.000	-0.049
18	-7.448	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[11]~_Duplicate_1	clk	clk	5.000	0.012
19	-7.309	Lab5_Ctrl:Ctrl|state.D	Lab5_Reg:RegG|output[13]	clk	clk	5.000	-0.036
20	-7.297	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[11]~_Duplicate_1	clk	clk	5.000	0.012
21	-7.167	Lab5_Ctrl:Ctrl|state.D	Lab5_Reg:RegG|output[14]	clk	clk	5.000	-0.049
22	-7.034	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[10]	clk	clk	5.000	-0.028
23	-6.883	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[10]	clk	clk	5.000	-0.028
24	-6.870	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[9]	clk	clk	5.000	-0.036
25	-6.827	Lab5_Ctrl:Ctrl|state.D	Lab5_Reg:RegG|output[14]~_Duplicate_1	clk	clk	5.000	0.003
26	-6.790	Lab5_Ctrl:Ctrl|state.B	Lab5_Reg:RegG|output[12]~_Duplicate_1	clk	clk	5.000	0.003
27	-6.719	Lab5_Ctrl:Ctrl|state.C	Lab5_Reg:RegG|output[9]	clk	clk	5.000	-0.036

Figure C.1. TimeQuest Output for Processing Circuit.

The time it takes to clear the combinational logic in the critical path is the bottleneck on how fast the circuit can be clocked on a physical board. As a critical path is nothing more than sequential combinational logic, and there is no feedback in this circuit's critical path from the controller to register G, the best way to eliminate it is to introduce D Flip-Flops periodically through the combinational logic to reduce the amount of sequential combinational logic. For instance, D Flip-Flops can be inserted between ever multiplexer in the multiplexer macro and every full adder in the ALU.

Overall, this lab reinforced the importance of modular testing and organization. Crafting the circuit in Figure I.2 would not have been possible without taking the design a component at a time. It taught a great deal about synchronization, as components that are controlled by one component cannot be clocked on the same clock edge as it without undesirable, random behavior. It allowed the designer to revisit basic circuits from introductory digital logic classes and see how they are used in higher-level circuits. It gave an introduction to using TimeQuest Timing Analyzer and tied into concepts from lecture dealing with critical paths through circuits. Lastly, and by no means least, this lab created a circuit that would be practical for real implementation in a computer, and gave a small taste of what higher-level circuit design is like, which is truly amazing.

References

[1] M. Smith. *ECE327 Digital System Design, Lab 5: Simple Processor.* [Online]. Available: https://bb.clemson.edu/bbcswebdav/pid-1909146-dt-content-rid-22214111_2/courses/smithmc-ece-327-DSD/Lab5_Processor%282%29.pdf

APPENDIX

Appendix A.1: The Registers

A.1.1.1: Generic Register VHDL Code

```vhdl
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare Generic Register Entity --
ENTITY Lab5_Reg IS
    GENERIC (N : INTEGER := 16);
    PORT (input  : IN std_logic_vector(N - 1 DOWNTO 0);
          reset  : IN std_logic;
          load   : IN std_logic;
          clk    : IN std_logic;
          output : OUT std_logic_vector(N - 1 DOWNTO 0));
END Lab5_Reg;

-- Architecture of Generic Register Entity --
ARCHITECTURE Lab5_Reg_B OF Lab5_Reg IS
BEGIN
    initialize: PROCESS (clk, reset)
    BEGIN
        IF(reset = '0') THEN
            -- Low active reset --
            zeroes: FOR i IN 0 TO N - 1 LOOP
                output(i) <= '0';
            END LOOP;
        ELSIF(falling_edge(clk) AND load = '1') THEN
            -- Load input into register --
            output <= input;
        END IF;
    END PROCESS initialize;
END Lab5_Reg_B;
```

A.1.1.2: Instruction Register VHDL Code

```vhdl
---- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare Generic Register Entity --
ENTITY Lab5_IR IS
    GENERIC (N : INTEGER := 9);
    PORT (input  : IN std_logic_vector(N - 1 DOWNTO 0);
          reset  : IN std_logic;
          load   : IN std_logic;
          clk    : IN std_logic;
          output : OUT std_logic_vector(N - 1 DOWNTO 0));
END Lab5_IR;

-- Architecture of Generic Register Entity --
ARCHITECTURE Lab5_IR_B OF Lab5_IR IS
BEGIN
    initialize: PROCESS (clk, reset)
    BEGIN
        IF(reset = '0') THEN
            -- Low active reset --
            zeroes: FOR i IN 0 TO N - 1 LOOP
                output(i) <= '0';
            END LOOP;
        ELSIF(falling_edge(clk) AND load = '1') THEN
            -- Load input into register --
            output <= input;
        END IF;
    END PROCESS initialize;
END Lab5_IR_B;
```

A.1.2.1: Generic Register Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab5_Reg_vhd_tst IS
END Lab5_Reg_vhd_tst;
ARCHITECTURE Lab5_Reg_arch OF Lab5_Reg_vhd_tst IS
-- constants
-- signals
SIGNAL clk : STD_LOGIC;
SIGNAL input : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL load : STD_LOGIC;
SIGNAL output : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL reset : STD_LOGIC;
COMPONENT Lab5_Reg
    PORT (
    clk : IN STD_LOGIC;
    input : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    load : IN STD_LOGIC;
    output : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    reset : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab5_Reg
    PORT MAP (
-- list connections between master ports and signals
    clk => clk,
    input => input,
    load => load,
    output => output,
    reset => reset
    );
init : PROCESS
-- variable declarations
BEGIN
        -- Initializations --
        input <= "1111111111111111";
        load <= '1';
        reset <= '0'; wait for 10 ps;
        reset <= '1'; wait for 10 ps;
        input <= "0000000000000000";
        load <= '0'; wait for 10 ps;
        load <= '1'; wait for 10 ps;
        load <= '0';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
falling_clock : PROCESS
BEGIN
    clk <= '1'; wait for 5 ps;
    clk <= '0'; wait for 5 ps;
END PROCESS falling_clock;
END Lab5_Reg_arch;
```

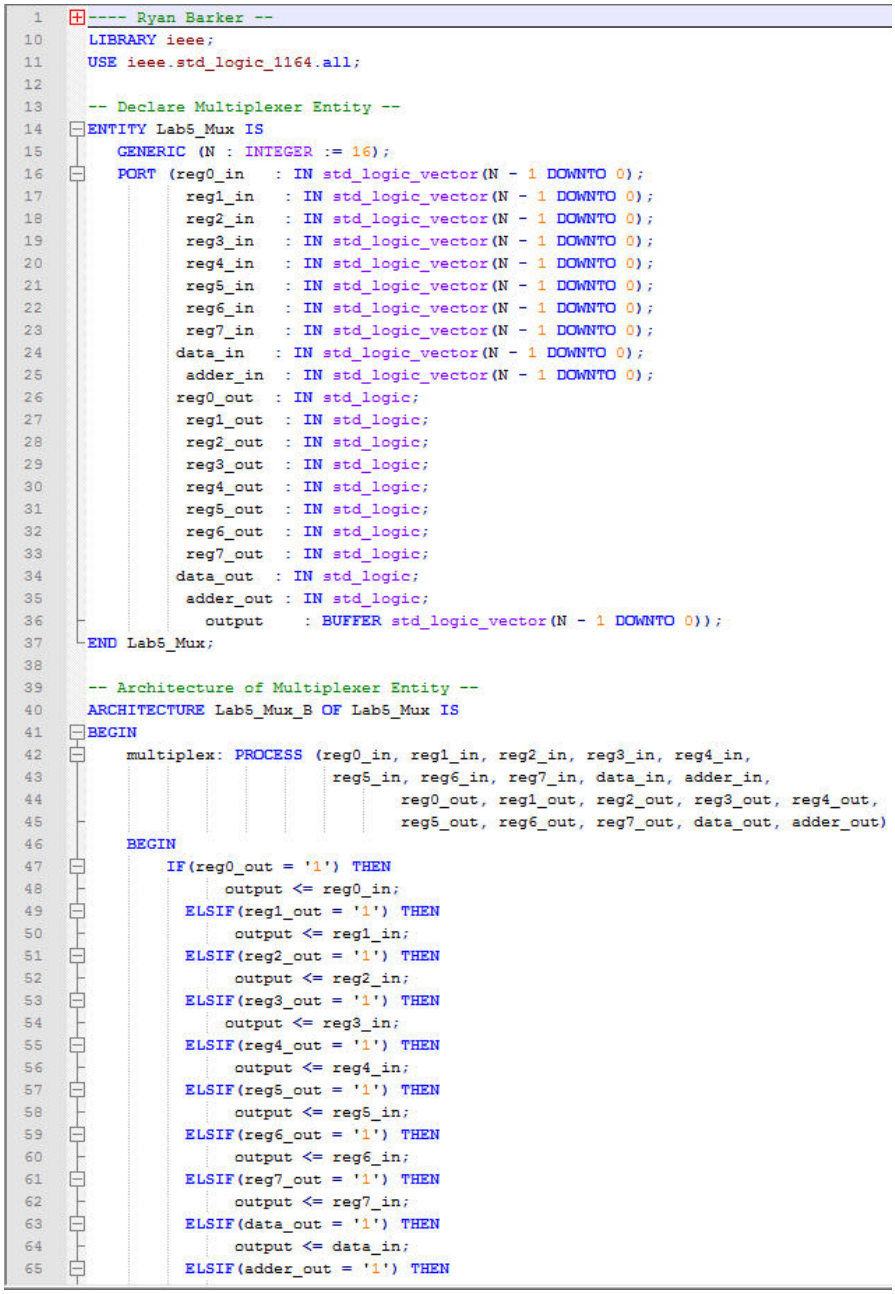
A.1.2.2: Instruction Register Test Bench

```
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31  ENTITY Lab5_IR_vhd_tst IS
32  END Lab5_IR_vhd_tst;
33  ARCHITECTURE Lab5_IR_arch OF Lab5_IR_vhd_tst IS
34  -- constants
35  -- signals
36  SIGNAL clk : STD_LOGIC;
37  SIGNAL input : STD_LOGIC_VECTOR(8 DOWNTO 0);
38  SIGNAL load : STD_LOGIC;
39  SIGNAL output : STD_LOGIC_VECTOR(8 DOWNTO 0);
40  SIGNAL reset : STD_LOGIC;
41  COMPONENT Lab5_IR
42      PORT (
43      clk : IN STD_LOGIC;
44      input : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
45      load : IN STD_LOGIC;
46      output : BUFFER STD_LOGIC_VECTOR(8 DOWNTO 0);
47      reset : IN STD_LOGIC
48      );
49  END COMPONENT;
50  BEGIN
51      i1 : Lab5_IR
52      PORT MAP (
53  -- list connections between master ports and signals
54      clk => clk,
55      input => input,
56      load => load,
57      output => output,
58      reset => reset
59      );
60  init : PROCESS
61  -- variable declarations
62  BEGIN
63          -- Initializations --
64          input <= "111111111";
65          load <= '1';
66          reset <= '0'; wait for 10 ps;
67          reset <= '1'; wait for 10 ps;
68          input <= "000000000";
69          load <= '0'; wait for 10 ps;
70          load <= '1'; wait for 10 ps;
71          load <= '0';
72  WAIT;
73  END PROCESS init;
74  always : PROCESS
75  -- optional sensitivity list
76  -- (        )
77  -- variable declarations
78  BEGIN
79          -- code executes for every event on sensitivity list
80  WAIT;
81  END PROCESS always;
```

```
82  falling_clock : PROCESS
83  BEGIN
84      clk <= '1'; wait for 5 ps;
85      clk <= '0'; wait for 5 ps;
86  END PROCESS falling_clock;
87  END Lab5_IR_arch;
88
```

Appendix A.2: The Multiplexer Macro

A.2.1: Multiplexer Macro VHDL Code



```
1   ---- Ryan Barker --
10  LIBRARY ieee;
11  USE ieee.std_logic_1164.all;
12
13  -- Declare Multiplexer Entity --
14  ENTITY Lab5_Mux IS
15      GENERIC (N : INTEGER := 16);
16      PORT (reg0_in   : IN std_logic_vector(N - 1 DOWNTO 0);
17            reg1_in   : IN std_logic_vector(N - 1 DOWNTO 0);
18            reg2_in   : IN std_logic_vector(N - 1 DOWNTO 0);
19            reg3_in   : IN std_logic_vector(N - 1 DOWNTO 0);
20            reg4_in   : IN std_logic_vector(N - 1 DOWNTO 0);
21            reg5_in   : IN std_logic_vector(N - 1 DOWNTO 0);
22            reg6_in   : IN std_logic_vector(N - 1 DOWNTO 0);
23            reg7_in   : IN std_logic_vector(N - 1 DOWNTO 0);
24           data_in    : IN std_logic_vector(N - 1 DOWNTO 0);
25            adder_in  : IN std_logic_vector(N - 1 DOWNTO 0);
26           reg0_out   : IN std_logic;
27            reg1_out  : IN std_logic;
28            reg2_out  : IN std_logic;
29            reg3_out  : IN std_logic;
30            reg4_out  : IN std_logic;
31            reg5_out  : IN std_logic;
32            reg6_out  : IN std_logic;
33            reg7_out  : IN std_logic;
34           data_out   : IN std_logic;
35            adder_out : IN std_logic;
36              output    : BUFFER std_logic_vector(N - 1 DOWNTO 0));
37  END Lab5_Mux;
38
39  -- Architecture of Multiplexer Entity --
40  ARCHITECTURE Lab5_Mux_B OF Lab5_Mux IS
41  BEGIN
42      multiplex: PROCESS (reg0_in, reg1_in, reg2_in, reg3_in, reg4_in,
43                          reg5_in, reg6_in, reg7_in, data_in, adder_in,
44                                  reg0_out, reg1_out, reg2_out, reg3_out, reg4_out,
45                                  reg5_out, reg6_out, reg7_out, data_out, adder_out)
46      BEGIN
47          IF(reg0_out = '1') THEN
48              output <= reg0_in;
49          ELSIF(reg1_out = '1') THEN
50              output <= reg1_in;
51          ELSIF(reg2_out = '1') THEN
52              output <= reg2_in;
53          ELSIF(reg3_out = '1') THEN
54              output <= reg3_in;
55          ELSIF(reg4_out = '1') THEN
56              output <= reg4_in;
57          ELSIF(reg5_out = '1') THEN
58              output <= reg5_in;
59          ELSIF(reg6_out = '1') THEN
60              output <= reg6_in;
61          ELSIF(reg7_out = '1') THEN
62              output <= reg7_in;
63          ELSIF(data_out = '1') THEN
64              output <= data_in;
65          ELSIF(adder_out = '1') THEN
66              output <= adder_in;
67          ELSE
68              -- If an output signal is not asserted --
69              -- by the controller, output zeroes  --
70              zeroes: FOR i IN 0 TO N - 1 LOOP
71              output(i) <= '0';
72            END LOOP;
73        END IF;
74      END PROCESS multiplex;
75  END Lab5_Mux_B;
```

A.2.2: Multiplexer Macro Test Bench

```
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31  ENTITY Lab5_Mux_vhd_tst IS
32  END Lab5_Mux_vhd_tst;
33  ARCHITECTURE Lab5_Mux_arch OF Lab5_Mux_vhd_tst IS
34  -- constants
35  -- signals
36  SIGNAL adder_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
37  SIGNAL adder_out : STD_LOGIC;
38  SIGNAL data_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
39  SIGNAL data_out : STD_LOGIC;
40  SIGNAL output : STD_LOGIC_VECTOR(15 DOWNTO 0);
41  SIGNAL reg0_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
42  SIGNAL reg0_out : STD_LOGIC;
43  SIGNAL reg1_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
44  SIGNAL reg1_out : STD_LOGIC;
45  SIGNAL reg2_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
46  SIGNAL reg2_out : STD_LOGIC;
47  SIGNAL reg3_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
48  SIGNAL reg3_out : STD_LOGIC;
49  SIGNAL reg4_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
50  SIGNAL reg4_out : STD_LOGIC;
51  SIGNAL reg5_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
52  SIGNAL reg5_out : STD_LOGIC;
53  SIGNAL reg6_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
54  SIGNAL reg6_out : STD_LOGIC;
55  SIGNAL reg7_in : STD_LOGIC_VECTOR(15 DOWNTO 0);
56  SIGNAL reg7_out : STD_LOGIC;
57  COMPONENT Lab5_Mux
58      PORT (
59      adder_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
60      adder_out : IN STD_LOGIC;
61      data_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
62      data_out : IN STD_LOGIC;
63      output : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
64      reg0_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
65      reg0_out : IN STD_LOGIC;
66      reg1_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
67      reg1_out : IN STD_LOGIC;
68      reg2_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
69      reg2_out : IN STD_LOGIC;
70      reg3_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
71      reg3_out : IN STD_LOGIC;
72      reg4_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
73      reg4_out : IN STD_LOGIC;
74      reg5_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
75      reg5_out : IN STD_LOGIC;
76      reg6_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
77      reg6_out : IN STD_LOGIC;
78      reg7_in : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
79      reg7_out : IN STD_LOGIC
80      );
81  END COMPONENT;
```

```
82  BEGIN
83      i1 : Lab5_Mux
84      PORT MAP (
85  -- list connections between master ports and signals
86      adder_in => adder_in,
87      adder_out => adder_out,
88      data_in => data_in,
89      data_out => data_out,
90      output => output,
91      reg0_in => reg0_in,
92      reg0_out => reg0_out,
93      reg1_in => reg1_in,
94      reg1_out => reg1_out,
95      reg2_in => reg2_in,
96      reg2_out => reg2_out,
97      reg3_in => reg3_in,
98      reg3_out => reg3_out,
99      reg4_in => reg4_in,
100     reg4_out => reg4_out,
101     reg5_in => reg5_in,
102     reg5_out => reg5_out,
103     reg6_in => reg6_in,
104     reg6_out => reg6_out,
105     reg7_in => reg7_in,
106     reg7_out => reg7_out
107     );
108 init : PROCESS
109 -- variable declarations
110 BEGIN
111         -- Initializations --
112         reg0_in <=  "0000000000000000";
113         reg1_in <=  "0000000000000001";
114         reg2_in <=  "0000000000000010";
115         reg3_in <=  "0000000000000100";
116         reg4_in <=  "0000000000001000";
117         reg5_in <=  "0000000000010000";
118         reg6_in <=  "0000000000100000";
119         reg7_in <=  "0000000001000000";
120           data_in <=  "0000000010000000";
121           adder_in <= "0000000100000000";
122
123           -- Initial Control Lines --
124           reg0_out <= '1';
125           reg1_out <= '0';
126           reg2_out <= '0';
127           reg3_out <= '0';
128           reg4_out <= '0';
129           reg5_out <= '0';
130           reg6_out <= '0';
131           reg7_out <= '0';
132           data_out <= '0';
133           adder_out <= '0'; wait for 10 ps;
134
135           -- Toggle Control Lines --
136           reg0_out <= '0';
137           reg1_out <= '1';
138           reg2_out <= '0';
```

```
139             reg3_out <= '0';
140             reg4_out <= '0';
141             reg5_out <= '0';
142             reg6_out <= '0';
143             reg7_out <= '0';
144             data_out <= '0';
145             adder_out <= '0'; wait for 10 ps;
146
147             reg0_out <= '0';
148             reg1_out <= '0';
149             reg2_out <= '1';
150             reg3_out <= '0';
151             reg4_out <= '0';
152             reg5_out <= '0';
153             reg6_out <= '0';
154             reg7_out <= '0';
155             data_out <= '0';
156             adder_out <= '0'; wait for 10 ps;
157
158             reg0_out <= '0';
159             reg1_out <= '0';
160             reg2_out <= '0';
161             reg3_out <= '1';
162             reg4_out <= '0';
163             reg5_out <= '0';
164             reg6_out <= '0';
165             reg7_out <= '0';
166             data_out <= '0';
167             adder_out <= '0'; wait for 10 ps;
168
169             reg0_out <= '0';
170             reg1_out <= '0';
171             reg2_out <= '0';
172             reg3_out <= '0';
173             reg4_out <= '1';
174             reg5_out <= '0';
175             reg6_out <= '0';
176             reg7_out <= '0';
177             data_out <= '0';
178             adder_out <= '0'; wait for 10 ps;
179
180             reg0_out <= '0';
181             reg1_out <= '0';
182             reg2_out <= '0';
183             reg3_out <= '0';
184             reg4_out <= '0';
185             reg5_out <= '1';
186             reg6_out <= '0';
187             reg7_out <= '0';
188             data_out <= '0';
189             adder_out <= '0'; wait for 10 ps;
190
191             reg0_out <= '0';
192             reg1_out <= '0';
193             reg2_out <= '0';
194             reg3_out <= '0';
195             reg4_out <= '0';
```

```
196             reg5_out <= '0';
197             reg6_out <= '1';
198             reg7_out <= '0';
199             data_out <= '0';
200             adder_out <= '0'; wait for 10 ps;
201
202             reg0_out <= '0';
203             reg1_out <= '0';
204             reg2_out <= '0';
205             reg3_out <= '0';
206             reg4_out <= '0';
207             reg5_out <= '0';
208             reg6_out <= '0';
209             reg7_out <= '1';
210             data_out <= '0';
211             adder_out <= '0'; wait for 10 ps;
212
213             reg0_out <= '0';
214             reg1_out <= '0';
215             reg2_out <= '0';
216             reg3_out <= '0';
217             reg4_out <= '0';
218             reg5_out <= '0';
219             reg6_out <= '0';
220             reg7_out <= '0';
221             data_out <= '1';
222             adder_out <= '0'; wait for 10 ps;
223
224             reg0_out <= '0';
225             reg1_out <= '0';
226             reg2_out <= '0';
227             reg3_out <= '0';
228             reg4_out <= '0';
229             reg5_out <= '0';
230             reg6_out <= '0';
231             reg7_out <= '0';
232             data_out <= '0';
233             adder_out <= '1'; wait for 10 ps;
234
235             reg0_out <= '0';
236             reg1_out <= '0';
237             reg2_out <= '0';
238             reg3_out <= '0';
239             reg4_out <= '0';
240             reg5_out <= '0';
241             reg6_out <= '0';
242             reg7_out <= '0';
243             data_out <= '0';
244             adder_out <= '0';
245     WAIT;
246     END PROCESS init;
247     always : PROCESS
248     -- optional sensitivity list
249     -- (        )
250     -- variable declarations
```

```
251     BEGIN
252             -- code executes for every event on sensitivity list
253     WAIT;
254     END PROCESS always;
255     END Lab5_Mux_arch;
256
```

Appendix A.3: The ALU

A.3.1: ALU VHDL Code

```
1   -- Ryan Barker --
11
12  LIBRARY ieee;
13  USE ieee.std_logic_1164.all;
14
15  -- Declare One Bit Full Adder Entity --
16  ENTITY One_Bit_Full_Adder IS
17      PORT (a,b    : IN std_logic;
18            c_in   : IN std_logic;
19            sum    : OUT std_logic;
20            c_out  : OUT std_logic);
21  END One_Bit_Full_Adder;
22
23  -- Architecture of One Bit Full_Adder Entity --
24  ARCHITECTURE One_Bit_Full_Adder_B OF One_Bit_Full_Adder IS
25  BEGIN
26      sum <= a XOR b XOR c_in;
27      c_out <= (a AND b) OR (c_in AND (a XOR b));
28  END One_Bit_Full_Adder_B;
29
30  LIBRARY ieee;
31  USE ieee.std_logic_1164.all;
32
33  -- Declare Adder Entity --
34  ENTITY Lab5_AddSub IS
35      GENERIC (N : INTEGER := 16);
36      PORT (in1    : IN std_logic_vector(N - 1 DOWNTO 0);
37            in2    : IN std_logic_vector(N - 1 DOWNTO 0);
38            mode   : IN std_logic;
39            sum    : OUT std_logic_vector(N - 1 DOWNTO 0);
40            c_out  : OUT std_logic);
41  END Lab5_AddSub;
42
43  -- Architecture of Adder Entity --
44  ARCHITECTURE Lab5_AddSub_B OF Lab5_AddSub IS
45      SIGNAL carries : std_logic_vector(N DOWNTO 0);
46      SIGNAL xors    : std_logic_vector(N - 1 DOWNTO 0);
47  COMPONENT One_Bit_Full_Adder
48      PORT (a,b    : IN std_logic;
49            c_in   : IN std_logic;
50            sum    : OUT std_logic;
51            c_out  : OUT std_logic);
52  END COMPONENT;
53  BEGIN
54      carries(0) <= mode;
55      c_out <= carries(N);
56
57      Gen_Adder : FOR i IN 0 to N - 1 GENERATE
58        xors(i) <= in2(i) XOR mode;
59
60          AdderX : One_Bit_Full_Adder
61              PORT MAP(a=>in1(i), b=>xors(i), c_in=>carries(i), sum=>sum(i), c_out=>carries(i + 1));
62      END GENERATE Gen_Adder;
63  END Lab5_AddSub_B;
```

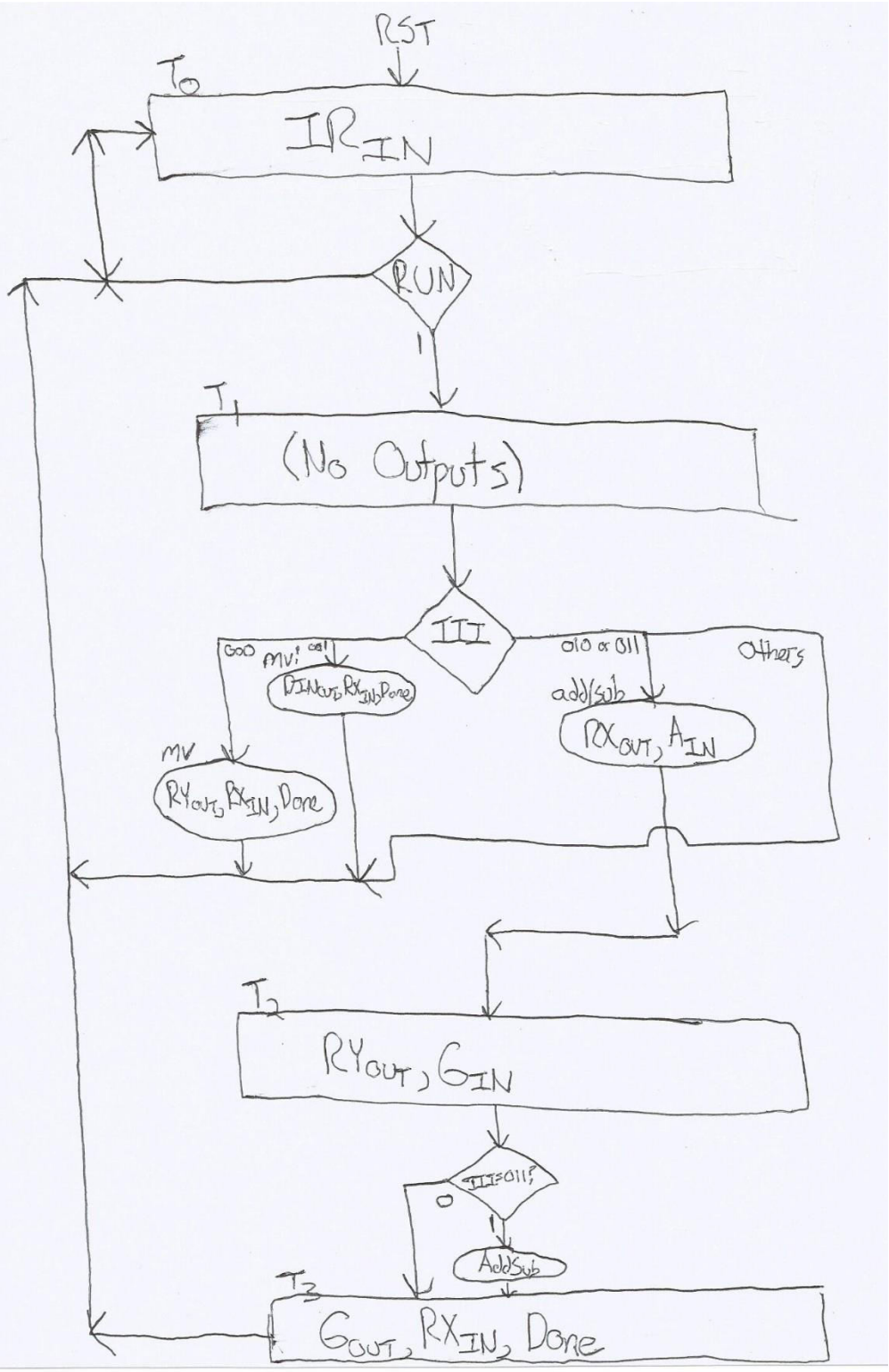
A.3.2: ALU Test Bench

```
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31  ENTITY Lab5_AddSub_vhd_tst IS
32  END Lab5_AddSub_vhd_tst;
33  ARCHITECTURE Lab5_AddSub_arch OF Lab5_AddSub_vhd_tst IS
34  -- constants
35  -- signals
36  SIGNAL c_out : STD_LOGIC;
37  SIGNAL in1 : STD_LOGIC_VECTOR(15 DOWNTO 0);
38  SIGNAL in2 : STD_LOGIC_VECTOR(15 DOWNTO 0);
39  SIGNAL mode : STD_LOGIC;
40  SIGNAL sum : STD_LOGIC_VECTOR(15 DOWNTO 0);
41  COMPONENT Lab5_AddSub
42      PORT (
43      c_out : BUFFER STD_LOGIC;
44      in1 : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
45      in2 : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
46      mode : IN STD_LOGIC;
47      sum : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0)
48      );
49  END COMPONENT;
50  BEGIN
51      i1 : Lab5_AddSub
52      PORT MAP (
53  -- list connections between master ports and signals
54      c_out => c_out,
55      in1 => in1,
56      in2 => in2,
57      mode => mode,
58      sum => sum
59      );
60  init : PROCESS
61  -- variable declarations
62  BEGIN
63          -- Initializations --
64          in1  <= "0000000000000001";
65          in2  <= "0000000000000001";
66
67           -- Toggle mode --
68          mode <= '0'; wait for 10 ps;
69          mode <= '1'; wait for 10 ps;
70
71          in1  <= "1111111111111111";
72          in2  <= "0000000000000001";
73          mode <= '0'; wait for 10 ps;
74          mode <= '1'; wait for 10 ps;
75
76          in1  <= "0101010101010100";
77          in2  <= "0000000000000001";
78          mode <= '0'; wait for 10 ps;
79          mode <= '1';
80  WAIT;
81  END PROCESS init;
```

```
82  always : PROCESS
83  -- optional sensitivity list
84  -- (        )
85  -- variable declarations
86  BEGIN
87          -- code executes for every event on sensitivity list
88  WAIT;
89  END PROCESS always;
90  END Lab5_AddSub_arch;
91
```

Appendix A.4: FSM Controller and Remaining Components

A.4.1.1: Controller ASM Chart



A.4.1.2: Controller VHDL Code

```
---- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare controller (state machine) --
ENTITY Lab5_Ctrl IS
    PORT (IR        : IN std_logic_vector(8 DOWNTO 0);
          Run       : IN std_logic;
          Resetn    : IN std_logic;
            Clk     : IN std_logic;
            DecX0   : IN std_logic;
            DecX1   : IN std_logic;
            DecX2   : IN std_logic;
            DecX3   : IN std_logic;
            DecX4   : IN std_logic;
            DecX5   : IN std_logic;
            DecX6   : IN std_logic;
            DecX7   : IN std_logic;
            DecY0   : IN std_logic;
            DecY1   : IN std_logic;
            DecY2   : IN std_logic;
            DecY3   : IN std_logic;
            DecY4   : IN std_logic;
            DecY5   : IN std_logic;
            DecY6   : IN std_logic;
            DecY7   : IN std_logic;
           IR_in    : OUT std_logic;
            R0_out  : OUT std_logic;
            R1_out  : OUT std_logic;
            R2_out  : OUT std_logic;
            R3_out  : OUT std_logic;
            R4_out  : OUT std_logic;
            R5_out  : OUT std_logic;
            R6_out  : OUT std_logic;
            R7_out  : OUT std_logic;
           G_out    : OUT std_logic;
            DIN_out : OUT std_logic;
            R0_in   : OUT std_logic;
            R1_in   : OUT std_logic;
            R2_in   : OUT std_logic;
            R3_in   : OUT std_logic;
            R4_in   : OUT std_logic;
            R5_in   : OUT std_logic;
            R6_in   : OUT std_logic;
            R7_in   : OUT std_logic;
           A_in     : OUT std_logic;
            G_in    : OUT std_logic;
          AddSub    : OUT std_logic;
            Done    : OUT std_logic;
           DecX_out : OUT std_logic_vector(2 DOWNTO 0);
           DecY_out : OUT std_logic_vector(2 DOWNTO 0));
END Lab5_Ctrl;
```

```
-- Architecture of controller (state machine) --
ARCHITECTURE Lab5_Ctrl_B OF Lab5_Ctrl IS
    TYPE fsm_state IS (A, B, C, D);
    SIGNAL state : fsm_state;
BEGIN
    -- Send RX Value and RY Value to Decoders --
    DecX_out <= IR(5 DOWNTO 3);
    DecY_out <= IR(2 DOWNTO 0);

    next_state: PROCESS (Resetn, Clk)
    BEGIN
        IF (Resetn = '0') THEN
            -- Asynchronous Reset --
            state <= A;
        ELSIF (rising_edge(Clk)) THEN
            CASE state IS
                WHEN A =>
                    -- Idle state (T0) --
                    IF(Run = '1') THEN state <= B;
                    ELSE state <= A;
                    END IF;
                WHEN B =>
                    -- Processing state 1 (T1) --
                    IF(IR(8 DOWNTO 6) = "010" OR
                     IR(8 DOWNTO 6) = "011") THEN
                        state  <= C;
                     ELSE state <= A;
                    END IF;
                WHEN C =>
                    -- Processing state 2 (T2) --
                    state <= D;
                 WHEN D =>
                    -- Processing state 3 (T3) --
                    state <= A;
            END CASE;
        END IF;
    END PROCESS next_state;

    outputs: PROCESS (state, IR(8 DOWNTO 6), DecX0,
                      DecX1, DecX2, DecX3, DecX4,
                             DecX5, DecX6, DecX7, DecY0,
                             DecY1, DecY2, DecY3, DecY4,
                             DecY5, DecY6, DecY7)
    BEGIN
        CASE state IS
            WHEN A =>
                -- Idle state outputs --
                IR_in   <= '1';
               R0_out  <= '0';
               R1_out  <= '0';
               R2_out  <= '0';
               R3_out  <= '0';
               R4_out  <= '0';
               R5_out  <= '0';
               R6_out  <= '0';
               R7_out  <= '0';
```

```
                G_out   <= '0';
                DIN_out <= '0';

                R0_in   <= '0';
                R1_in   <= '0';
                R2_in   <= '0';
                R3_in   <= '0';
                R4_in   <= '0';
                R5_in   <= '0';
                R6_in   <= '0';
                R7_in   <= '0';
                A_in    <= '0';
                G_in    <= '0';

                AddSub  <= '0';
                Done    <= '0';
            WHEN B =>
                -- The below are never asserted in this state --
                IR_in  <= '0';
                G_out  <= '0';
                G_in   <= '0';
                AddSub <= '0';

                -- Processing state 1 outputs--
                IF (IR(8 DOWNTO 6) = "000") THEN
                -- Command is mv --

                    -- Use RY values --
                    R0_out  <= DecY0;
                    R1_out  <= DecY1;
                    R2_out  <= DecY2;
                    R3_out  <= DecY3;
                    R4_out  <= DecY4;
                    R5_out  <= DecY5;
                    R6_out  <= DecY6;
                    R7_out  <= DecY7;
                    DIN_out <= '0';

                    -- Use RX values --
                    R0_in   <= DecX0;
                    R1_in   <= DecX1;
                    R2_in   <= DecX2;
                    R3_in   <= DecX3;
                    R4_in   <= DecX4;
                    R5_in   <= DecX5;
                    R6_in   <= DecX6;
                    R7_in   <= DecX7;
                    A_in    <= '0';

                    Done    <= '1';
                ELSIF(IR(8 DOWNTO 6) = "001") THEN
                -- Command is mvi --
                    R0_out  <= '0';
                    R1_out  <= '0';
                    R2_out  <= '0';
                    R3_out  <= '0';
                    R4_out  <= '0';
```

```
                    R5_out  <= '0';
                    R6_out  <= '0';
                    R7_out  <= '0';
                    DIN_out <= '1';

                    -- Use RX Values--
                    R0_in   <= DecX0;
                    R1_in   <= DecX1;
                    R2_in   <= DecX2;
                    R3_in   <= DecX3;
                    R4_in   <= DecX4;
                    R5_in   <= DecX5;
                    R6_in   <= DecX6;
                    R7_in   <= DecX7;
                    A_in    <= '0';

                    Done    <= '1';
                ELSIF(IR(8 DOWNTO 6) = "010" OR
                      IR(8 DOWNTO 6) = "011") THEN
                    -- Command is add/subtract --

                    -- Use RX Values --
                    R0_out  <= DecX0;
                    R1_out  <= DecX1;
                    R2_out  <= DecX2;
                    R3_out  <= DecX3;
                    R4_out  <= DecX4;
                    R5_out  <= DecX5;
                    R6_out  <= DecX6;
                    R7_out  <= DecX7;
                    DIN_out <= '0';

                    R0_in   <= '0';
                    R1_in   <= '0';
                    R2_in   <= '0';
                    R3_in   <= '0';
                    R4_in   <= '0';
                    R5_in   <= '0';
                    R6_in   <= '0';
                    R7_in   <= '0';
                    A_in    <= '1';

                    Done    <= '0';
                ELSE
                    -- Command is unsupported --
                    R0_out  <= '0';
                    R1_out  <= '0';
                    R2_out  <= '0';
                    R3_out  <= '0';
                    R4_out  <= '0';
                    R5_out  <= '0';
                    R6_out  <= '0';
                    R7_out  <= '0';
                    DIN_out <= '0';
                    R0_in   <= '0';
                    R1_in   <= '0';
                    R2_in   <= '0';
```

```
                R3_in   <= '0';
                R4_in   <= '0';
                R5_in   <= '0';
                R6_in   <= '0';
                R7_in   <= '0';
                A_in    <= '0';
                Done    <= '0';
            END IF;
        WHEN C =>
            -- Processing state 2 outputs
            -- (for add/subtract) --

            -- Use RY Values --
            IR_in   <= '0';
          R0_out  <= DecY0;
          R1_out  <= DecY1;
          R2_out  <= DecY2;
          R3_out  <= DecY3;
          R4_out  <= DecY4;
          R5_out  <= DecY5;
          R6_out  <= DecY6;
          R7_out  <= DecY7;
         G_out   <= '0';
          DIN_out <= '0';

            R0_in   <= '0';
          R1_in   <= '0';
          R2_in   <= '0';
          R3_in   <= '0';
            R4_in   <= '0';
          R5_in   <= '0';
          R6_in   <= '0';
          R7_in   <= '0';
         A_in    <= '0';
          G_in    <= '1';

            -- AddSub on if doing subtraction --
          IF(IR(8 DOWNTO 6) = "011") THEN
              AddSub  <= '1';
          ELSE AddSub <= '0';
          END IF;
          Done    <= '0';
        WHEN D =>
            -- Processing state 3 outputs --
            -- (for add/subtract) --
        IR_in   <= '0';
          R0_out  <= '0';
          R1_out  <= '0';
          R2_out  <= '0';
          R3_out  <= '0';
          R4_out  <= '0';
          R5_out  <= '0';
          R6_out  <= '0';
          R7_out  <= '0';
         G_out   <= '1';
          DIN_out <= '0';

            -- Use RX Values --
            R0_in   <= DecX0;
          R1_in   <= DecX1;
          R2_in   <= DecX2;
          R3_in   <= DecX3;
            R4_in   <= DecX4;
          R5_in   <= DecX5;
          R6_in   <= DecX6;
          R7_in   <= DecX7;
         A_in    <= '0';
          G_in    <= '0';

        AddSub  <= '0';
          Done    <= '1';
    END CASE;
  END PROCESS outputs;
END Lab5_Ctrl_B;
```

A.4.1.3: 3-to-8 Decoder VHDL Code

```
1  ---- Ryan Barker --
11
12 LIBRARY ieee;
13 USE ieee.std_logic_1164.all;
14
15 -- Declare Decoder Entity --
16 ENTITY Lab5_Decoder IS
17    PORT (input  : IN std_logic_vector(2 DOWNTO 0);
18          out0   : OUT std_logic;
19          out1   : OUT std_logic;
20          out2   : OUT std_logic;
21          out3   : OUT std_logic;
22          out4   : OUT std_logic;
23          out5   : OUT std_logic;
24          out6   : OUT std_logic;
25          out7   : OUT std_logic);
26 END Lab5_Decoder;
27
28 -- Architecture of Decoder Entity --
29 ARCHITECTURE Lab5_Decoder_B OF Lab5_Decoder IS
30 BEGIN
31 Decode : PROCESS (input)
32       BEGIN
33          out0 <= NOT(input(2)) AND NOT(input(1)) AND NOT(input(0));
34          out1 <= NOT(input(2)) AND NOT(input(1)) AND     input(0);
35          out2 <= NOT(input(2)) AND     input(1)  AND NOT(input(0));
36          out3 <= NOT(input(2)) AND     input(1)  AND     input(0);
37          out4 <=     input(2)  AND NOT(input(1)) AND NOT(input(0));
38          out5 <=     input(2)  AND NOT(input(1)) AND     input(0);
39          out6 <=     input(2)  AND     input(1)  AND NOT(input(0));
40          out7 <=     input(2)  AND     input(1)  AND     input(0);
41    END PROCESS Decode;
42 END Lab5_Decoder_B;
```

A.4.2.1: 3-to-8 Decoder Test Bench

```
28 LIBRARY ieee;
29 USE ieee.std_logic_1164.all;
30
31 ENTITY Lab5_Decoder_vhd_tst IS
32 END Lab5_Decoder_vhd_tst;
33 ARCHITECTURE Lab5_Decoder_arch OF Lab5_Decoder_vhd_tst IS
34 -- constants
35 -- signals
36 SIGNAL input : STD_LOGIC_VECTOR(2 DOWNTO 0);
37 SIGNAL out0 : STD_LOGIC;
38 SIGNAL out1 : STD_LOGIC;
39 SIGNAL out2 : STD_LOGIC;
40 SIGNAL out3 : STD_LOGIC;
41 SIGNAL out4 : STD_LOGIC;
42 SIGNAL out5 : STD_LOGIC;
43 SIGNAL out6 : STD_LOGIC;
44 SIGNAL out7 : STD_LOGIC;
45 COMPONENT Lab5_Decoder
46    PORT (
47    input : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
48    out0 : OUT STD_LOGIC;
49    out1 : OUT STD_LOGIC;
50    out2 : OUT STD_LOGIC;
51    out3 : OUT STD_LOGIC;
52    out4 : OUT STD_LOGIC;
53    out5 : OUT STD_LOGIC;
54    out6 : OUT STD_LOGIC;
55    out7 : OUT STD_LOGIC
56    );
57 END COMPONENT;
```

```
BEGIN
    i1 : Lab5_Decoder
    PORT MAP (
-- list connections between master ports and signals
    input => input,
    out0 => out0,
    out1 => out1,
    out2 => out2,
    out3 => out3,
    out4 => out4,
    out5 => out5,
    out6 => out6,
    out7 => out7
    );
init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
        input <= "000"; wait for 10 ps;
        input <= "001"; wait for 10 ps;
        input <= "010"; wait for 10 ps;
        input <= "011"; wait for 10 ps;
        input <= "100"; wait for 10 ps;
        input <= "101"; wait for 10 ps;
        input <= "110"; wait for 10 ps;
        input <= "111";
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
END Lab5_Decoder_arch;
```

A.4.2.2: Controller Wrapper VHDL Code

```
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare Controller Test --
ENTITY Lab5_Ctrl_tst IS
PORT (IR_bits  : IN std_logic_vector(8 DOWNTO 0);
      Run      : IN std_logic;
      Resetn   : IN std_logic;
      Clk      : IN std_logic;
       R0_out   : OUT std_logic;
       R1_out   : OUT std_logic;
       R2_out   : OUT std_logic;
       R3_out   : OUT std_logic;
       R4_out   : OUT std_logic;
       R5_out   : OUT std_logic;
       R6_out   : OUT std_logic;
       R7_out   : OUT std_logic;
       G_out    : OUT std_logic;
       DIN_out  : OUT std_logic;
       R0_in    : OUT std_logic;
       R1_in    : OUT std_logic;
       R2_in    : OUT std_logic;
       R3_in    : OUT std_logic;
       R4_in    : OUT std_logic;
       R5_in    : OUT std_logic;
       R6_in    : OUT std_logic;
       R7_in    : OUT std_logic;
       A_in     : OUT std_logic;
       G_in     : OUT std_logic;
       AddSub   : OUT std_logic;
       Done     : OUT std_logic);
END Lab5_Ctrl_tst;
```

```
-- Architecture of Controller Test --
ARCHITECTURE Lab5_Ctrl_tst_B OF Lab5_Ctrl_tst IS
    SIGNAL IR_to_ctrl : std_logic_vector(8 DOWNTO 0);
     SIGNAL IR_in : std_logic;
     SIGNAL ctrl_to_DecX : std_logic_vector(2 DOWNTO 0);
     SIGNAL ctrl_to_DecY : std_logic_vector(2 DOWNTO 0);
     SIGNAL decX0, decX1, decX2, decX3, decX4, decX5,
            decX6, decX7, decY0, decY1, decY2, decY3,
              decY4, decY5, decY6, decY7 : std_logic;

    COMPONENT Lab5_Ctrl
    PORT (IR       : IN std_logic_vector(8 DOWNTO 0);
          Run      : IN std_logic;
          Resetn   : IN std_logic;
           Clk     : IN std_logic;
           DecX0   : IN std_logic;
           DecX1   : IN std_logic;
           DecX2   : IN std_logic;
           DecX3   : IN std_logic;
           DecX4   : IN std_logic;
           DecX5   : IN std_logic;
           DecX6   : IN std_logic;
           DecX7   : IN std_logic;
           DecY0   : IN std_logic;
           DecY1   : IN std_logic;
           DecY2   : IN std_logic;
           DecY3   : IN std_logic;
           DecY4   : IN std_logic;
           DecY5   : IN std_logic;
           DecY6   : IN std_logic;
           DecY7   : IN std_logic;
          IR_in    : OUT std_logic;
           R0_out  : OUT std_logic;
           R1_out  : OUT std_logic;
           R2_out  : OUT std_logic;
           R3_out  : OUT std_logic;
           R4_out  : OUT std_logic;
           R5_out  : OUT std_logic;
           R6_out  : OUT std_logic;
           R7_out  : OUT std_logic;
          G_out    : OUT std_logic;
           DIN_out : OUT std_logic;
           R0_in   : OUT std_logic;
           R1_in   : OUT std_logic;
           R2_in   : OUT std_logic;
           R3_in   : OUT std_logic;
           R4_in   : OUT std_logic;
           R5_in   : OUT std_logic;
           R6_in   : OUT std_logic;
           R7_in   : OUT std_logic;
          A_in     : OUT std_logic;
           G_in    : OUT std_logic;
         AddSub    : OUT std_logic;
           Done    : OUT std_logic;
          DecX_out : OUT std_logic_vector(2 DOWNTO 0);
           DecY_out : OUT std_logic_vector(2 DOWNTO 0));
```

```
    END COMPONENT;

     COMPONENT Lab5_Decoder
    PORT (input : IN std_logic_vector(2 DOWNTO 0);
          out0  : OUT std_logic;
          out1  : OUT std_logic;
          out2  : OUT std_logic;
          out3  : OUT std_logic;
          out4  : OUT std_logic;
          out5  : OUT std_logic;
          out6  : OUT std_logic;
          out7  : OUT std_logic);
    END COMPONENT;

     COMPONENT Lab5_IR
    PORT (input  : IN std_logic_vector(8 DOWNTO 0);
           reset  : IN std_logic;
          load   : IN std_logic;
            clk    : IN std_logic;
            output : OUT std_logic_vector(8 DOWNTO 0));
    END COMPONENT;
BEGIN
    Ctrl: Lab5_Ctrl
    PORT MAP (IR       => IR_to_ctrl,
              Run      => Run,
              Resetn   => Resetn,
               Clk     => Clk,
               DecX0   => decX0,
               DecX1   => decX1,
               DecX2   => decX2,
               DecX3   => decX3,
               DecX4   => decX4,
               DecX5   => decX5,
               DecX6   => decX6,
               DecX7   => decX7,
               DecY0   => decY0,
               DecY1   => decY1,
               DecY2   => decY2,
               DecY3   => decY3,
               DecY4   => decY4,
               DecY5   => decY5,
               DecY6   => decY6,
               DecY7   => decY7,
              IR_in    => IR_in,
               R0_out  => R0_out,
               R1_out  => R1_out,
               R2_out  => R2_out,
               R3_out  => R3_out,
               R4_out  => R4_out,
               R5_out  => R5_out,
               R6_out  => R6_out,
               R7_out  => R7_out,
              G_out    => G_out,
               DIN_out => DIN_out,
               R0_in   => R0_in,
               R1_in   => R1_in,
               R2_in   => R2_in,
```

```
                R3_in    => R3_in,
                R4_in    => R4_in,
                R5_in    => R5_in,
                R6_in    => R6_in,
                R7_in    => R7_in,
                A_in     => A_in,
                G_in     => G_in,
                AddSub   => AddSub,
                Done     => Done,
                DecX_out => ctrl_to_DecX,
                DecY_out => ctrl_to_DecY);

DecX: Lab5_Decoder
    PORT MAP (input => ctrl_to_DecX,
              out0  => decX0,
              out1  => decX1,
              out2  => decX2,
              out3  => decX3,
              out4  => decX4,
              out5  => decX5,
              out6  => decX6,
              out7  => decX7);

DecY: Lab5_Decoder
    PORT MAP (input => ctrl_to_DecY,
              out0  => decY0,
              out1  => decY1,
              out2  => decY2,
              out3  => decY3,
              out4  => decY4,
              out5  => decY5,
              out6  => decY6,
              out7  => decY7);

IR: Lab5_IR
    PORT MAP (input => IR_bits,
              reset => Resetn,
              load  => IR_in,
                clk   => Clk,
                output => IR_to_ctrl);
END Lab5_Ctrl_tst_B;
```

A.4.2.3: Controller Wrapper Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab5_Ctrl_tst_vhd_tst IS
END Lab5_Ctrl_tst_vhd_tst;
ARCHITECTURE Lab5_Ctrl_tst_arch OF Lab5_Ctrl_tst_vhd_tst IS
-- constants
-- signals
SIGNAL A_in : STD_LOGIC;
SIGNAL AddSub : STD_LOGIC;
SIGNAL Clk : STD_LOGIC;
SIGNAL DIN_out : STD_LOGIC;
SIGNAL Done : STD_LOGIC;
SIGNAL G_in : STD_LOGIC;
SIGNAL G_out : STD_LOGIC;
SIGNAL IR_bits : STD_LOGIC_VECTOR(8 DOWNTO 0);
SIGNAL R0_in : STD_LOGIC;
SIGNAL R0_out : STD_LOGIC;
SIGNAL R1_in : STD_LOGIC;
SIGNAL R1_out : STD_LOGIC;
SIGNAL R2_in : STD_LOGIC;
SIGNAL R2_out : STD_LOGIC;
SIGNAL R3_in : STD_LOGIC;
SIGNAL R3_out : STD_LOGIC;
SIGNAL R4_in : STD_LOGIC;
SIGNAL R4_out : STD_LOGIC;
SIGNAL R5_in : STD_LOGIC;
SIGNAL R5_out : STD_LOGIC;
SIGNAL R6_in : STD_LOGIC;
SIGNAL R6_out : STD_LOGIC;
SIGNAL R7_in : STD_LOGIC;
SIGNAL R7_out : STD_LOGIC;
SIGNAL Resetn : STD_LOGIC;
SIGNAL Run : STD_LOGIC;
COMPONENT Lab5_Ctrl_tst
    PORT (
    A_in : OUT STD_LOGIC;
    AddSub : OUT STD_LOGIC;
    Clk : IN STD_LOGIC;
    DIN_out : OUT STD_LOGIC;
    Done : OUT STD_LOGIC;
    G_in : OUT STD_LOGIC;
    G_out : OUT STD_LOGIC;
    IR_bits : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
    R0_in : OUT STD_LOGIC;
    R0_out : OUT STD_LOGIC;
    R1_in : OUT STD_LOGIC;
    R1_out : OUT STD_LOGIC;
    R2_in : OUT STD_LOGIC;
    R2_out : OUT STD_LOGIC;
    R3_in : OUT STD_LOGIC;
    R3_out : OUT STD_LOGIC;
    R4_in : OUT STD_LOGIC;
    R4_out : OUT STD_LOGIC;
    R5_in : OUT STD_LOGIC;
    R5_out : OUT STD_LOGIC;
    R6_in : OUT STD_LOGIC;
```

```
    R6_out : OUT STD_LOGIC;
    R7_in : OUT STD_LOGIC;
    R7_out : OUT STD_LOGIC;
    Resetn : IN STD_LOGIC;
    Run : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab5_Ctrl_tst
    PORT MAP (
-- list connections between master ports and signals
    A_in => A_in,
    AddSub => AddSub,
    Clk => Clk,
    DIN_out => DIN_out,
    Done => Done,
    G_in => G_in,
    G_out => G_out,
    IR_bits => IR_bits,
    R0_in => R0_in,
    R0_out => R0_out,
    R1_in => R1_in,
    R1_out => R1_out,
    R2_in => R2_in,
    R2_out => R2_out,
    R3_in => R3_in,
    R3_out => R3_out,
    R4_in => R4_in,
    R4_out => R4_out,
    R5_in => R5_in,
    R5_out => R5_out,
    R6_in => R6_in,
    R6_out => R6_out,
    R7_in => R7_in,
    R7_out => R7_out,
    Resetn => Resetn,
    Run => Run
    );
init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
          Run <= '1';
        IR_bits <= "000111000"; -- move 0 into 7
        Resetn <= '0'; wait for 10 ps;
          Resetn <= '1'; wait for 10 ps;
        Run <= '0';    wait for 10 ps;
        IR_bits <= "001101000"; -- mvi into 5
          Run <= '1';    wait for 10 ps;
          IR_bits <= "101010101";
          Run <= '0';    wait for 10 ps;
          IR_bits <= "010001010"; -- add 1 and 2. Store 1.
          Run <= '1';    wait for 10 ps;
          Run <= '0';    wait for 30 ps;
          IR_bits <= "011100001"; -- subtract 4 and 1. Store 4.
          Run <= '1';    wait for 10 ps;
          Run <= '0';
```

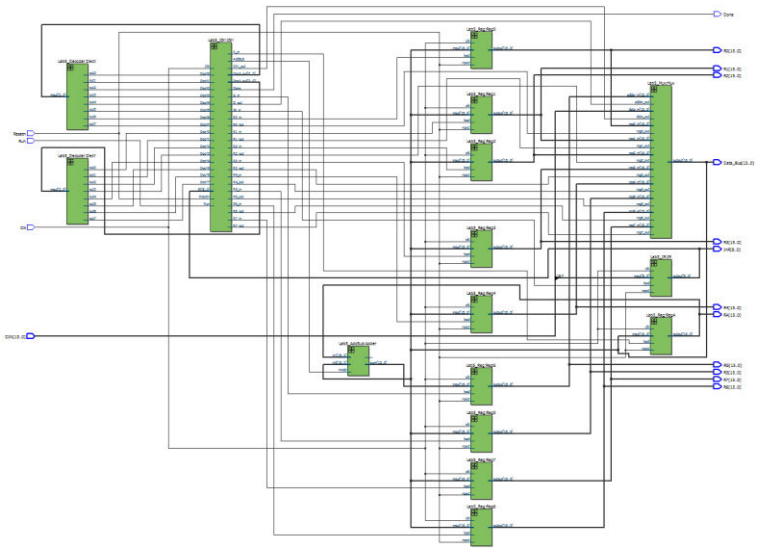
```
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list

WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    Clk <= '0'; wait for 5 ps;
    Clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab5_Ctrl_tst_arch;

```

Appendix A.5: The Processor

A.5.1.1: The Processor in the RTL Viewer



A.5.1.1.2: Processor VHDL Code

```
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare Controller Test --
ENTITY Lab5_Processor IS
GENERIC (N : INTEGER := 16);
PORT (DIN      : IN std_logic_vector(N - 1 DOWNTO 0);
      Run      : IN std_logic;
      Resetn   : IN std_logic;
      Clk      : IN std_logic;
      Data_Bus : BUFFER std_logic_vector(N - 1 DOWNTO 0);
        InR      : OUT std_logic_vector(8 DOWNTO 0);
      R0       : OUT std_logic_vector(N - 1 DOWNTO 0);
      R1       : OUT std_logic_vector(N - 1 DOWNTO 0);
      R2       : OUT std_logic_vector(N - 1 DOWNTO 0);
      R3       : OUT std_logic_vector(N - 1 DOWNTO 0);
       R4       : OUT std_logic_vector(N - 1 DOWNTO 0);
      R5       : OUT std_logic_vector(N - 1 DOWNTO 0);
      R6       : OUT std_logic_vector(N - 1 DOWNTO 0);
        R7       : OUT std_logic_vector(N - 1 DOWNTO 0);
      RA       : OUT std_logic_vector(N - 1 DOWNTO 0);
      RG       : OUT std_logic_vector(N - 1 DOWNTO 0);
        Done     : OUT std_logic);
END Lab5_Processor;

-- Architecture of Controller Test --
ARCHITECTURE Lab5_Processor_B OF Lab5_Processor IS
    SIGNAL IR_to_ctrl : std_logic_vector(8 DOWNTO 0);
     SIGNAL IR_in : std_logic;
     SIGNAL ctrl_to_DecX : std_logic_vector(2 DOWNTO 0);
     SIGNAL ctrl_to_DecY : std_logic_vector(2 DOWNTO 0);
     SIGNAL decX0, decX1, decX2, decX3, decX4, decX5,
            decX6, decX7, decY0, decY1, decY2, decY3,
              decY4, decY5, decY6, decY7 : std_logic;
    SIGNAL R0_out, R1_out, R2_out, R3_out, R4_out, R5_out, R6_out,
           R7_out, G_out, DIN_out, R0_in, R1_in, R2_in, R3_in, R4_in,
             R5_in, R6_in, R7_in, A_in, G_in, AddSub : std_logic;
    SIGNAL R0_to_mux, R1_to_mux, R2_to_mux, R3_to_mux, R4_to_mux,
           R5_to_mux, R6_to_mux, R7_to_mux, Adder_to_mux :
             std_logic_vector(N - 1 DOWNTO 0);
     SIGNAL RA_to_Adder, Adder_to_RG,
            RG_to_mux : std_logic_vector(N - 1 DOWNTO 0);

    COMPONENT Lab5_Ctrl
    PORT (IR       : IN std_logic_vector(8 DOWNTO 0);
          Run      : IN std_logic;
          Resetn   : IN std_logic;
           Clk      : IN std_logic;
            DecX0    : IN std_logic;
            DecX1    : IN std_logic;
            DecX2    : IN std_logic;
            DecX3    : IN std_logic;
            DecX4    : IN std_logic;
            DecX5    : IN std_logic;
            DecX6    : IN std_logic;
```

```
            DecX7    : IN std_logic;
            DecY0    : IN std_logic;
            DecY1    : IN std_logic;
            DecY2    : IN std_logic;
            DecY3    : IN std_logic;
            DecY4    : IN std_logic;
            DecY5    : IN std_logic;
            DecY6    : IN std_logic;
            DecY7    : IN std_logic;
            IR_in    : OUT std_logic;
            R0_out   : OUT std_logic;
            R1_out   : OUT std_logic;
            R2_out   : OUT std_logic;
            R3_out   : OUT std_logic;
            R4_out   : OUT std_logic;
            R5_out   : OUT std_logic;
            R6_out   : OUT std_logic;
            R7_out   : OUT std_logic;
            G_out    : OUT std_logic;
            DIN_out  : OUT std_logic;
            R0_in    : OUT std_logic;
            R1_in    : OUT std_logic;
            R2_in    : OUT std_logic;
            R3_in    : OUT std_logic;
            R4_in    : OUT std_logic;
            R5_in    : OUT std_logic;
            R6_in    : OUT std_logic;
            R7_in    : OUT std_logic;
            A_in     : OUT std_logic;
            G_in     : OUT std_logic;
            AddSub   : OUT std_logic;
            Done     : OUT std_logic;
            DecX_out : OUT std_logic_vector(2 DOWNTO 0);
            DecY_out : OUT std_logic_vector(2 DOWNTO 0));
    END COMPONENT;

    COMPONENT Lab5_Decoder
    PORT (input : IN std_logic_vector(2 DOWNTO 0);
          out0  : OUT std_logic;
          out1  : OUT std_logic;
          out2  : OUT std_logic;
          out3  : OUT std_logic;
          out4  : OUT std_logic;
          out5  : OUT std_logic;
          out6  : OUT std_logic;
          out7  : OUT std_logic);
    END COMPONENT;

    COMPONENT Lab5_IR
    PORT (input  : IN std_logic_vector(8 DOWNTO 0);
          reset  : IN std_logic;
          load   : IN std_logic;
          clk    : IN std_logic;
          output : OUT std_logic_vector(8 DOWNTO 0));
    END COMPONENT;

    COMPONENT Lab5_Reg
```

```
    PORT (input  : IN std_logic_vector(N - 1 DOWNTO 0);
          reset  : IN std_logic;
          load   : IN std_logic;
          clk    : IN std_logic;
          output : OUT std_logic_vector(N - 1 DOWNTO 0));
    END COMPONENT;

    COMPONENT Lab5_Mux
    PORT (reg0_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          reg1_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          reg2_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          reg3_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          reg4_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          reg5_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          reg6_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          reg7_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          data_in   : IN std_logic_vector(N - 1 DOWNTO 0);
          adder_in  : IN std_logic_vector(N - 1 DOWNTO 0);
          reg0_out  : IN std_logic;
          reg1_out  : IN std_logic;
          reg2_out  : IN std_logic;
          reg3_out  : IN std_logic;
          reg4_out  : IN std_logic;
          reg5_out  : IN std_logic;
          reg6_out  : IN std_logic;
          reg7_out  : IN std_logic;
          data_out  : IN std_logic;
          adder_out : IN std_logic;
          output    : BUFFER std_logic_vector(N - 1 DOWNTO 0));
    END COMPONENT;

    COMPONENT Lab5_AddSub
    PORT (in1   : IN std_logic_vector(N - 1 DOWNTO 0);
          in2   : IN std_logic_vector(N - 1 DOWNTO 0);
          mode  : IN std_logic;
          sum   : OUT std_logic_vector(N - 1 DOWNTO 0);
          c_out : OUT std_logic);
    END COMPONENT;
BEGIN
    -- Set reg outputs for view in simulation --
    InR <= IR_to_ctrl;
    R0  <= R0_to_mux;
    R1  <= R1_to_mux;
    R2  <= R2_to_mux;
    R3  <= R3_to_mux;
    R4  <= R4_to_mux;
    R5  <= R5_to_mux;
    R6  <= R6_to_mux;
    R7  <= R7_to_mux;
    RA  <= RA_to_Adder;
    RG  <= RG_to_mux;

    Ctrl: Lab5_Ctrl
    PORT MAP (IR       => IR_to_ctrl,
              Run      => Run,
              Resetn   => Resetn,
              Clk      => Clk,
```

```
                    DecX0    => decX0,
                    DecX1    => decX1,
                    DecX2    => decX2,
                    DecX3    => decX3,
                    DecX4    => decX4,
                    DecX5    => decX5,
                    DecX6    => decX6,
                    DecX7    => decX7,
                    DecY0    => decY0,
                    DecY1    => decY1,
                    DecY2    => decY2,
                    DecY3    => decY3,
                    DecY4    => decY4,
                    DecY5    => decY5,
                    DecY6    => decY6,
                    DecY7    => decY7,
                   IR_in    => IR_in,
                    R0_out   => R0_out,
                    R1_out   => R1_out,
                    R2_out   => R2_out,
                    R3_out   => R3_out,
                    R4_out   => R4_out,
                    R5_out   => R5_out,
                    R6_out   => R6_out,
                    R7_out   => R7_out,
                   G_out    => G_out,
                    DIN_out  => DIN_out,
                    R0_in    => R0_in,
                    R1_in    => R1_in,
                    R2_in    => R2_in,
                    R3_in    => R3_in,
                    R4_in    => R4_in,
                     R5_in    => R5_in,
                     R6_in    => R6_in,
                     R7_in    => R7_in,
                     A_in     => A_in,
                     G_in     => G_in,
                     AddSub   => AddSub,
                     Done     => Done,
                     DecX_out => ctrl_to_DecX,
                     DecY_out => ctrl_to_DecY);

 DecX: Lab5_Decoder
     PORT MAP (input => ctrl_to_DecX,
               out0  => decX0,
               out1  => decX1,
               out2  => decX2,
               out3  => decX3,
               out4  => decX4,
               out5  => decX5,
               out6  => decX6,
               out7  => decX7);

 DecY: Lab5_Decoder
     PORT MAP (input => ctrl_to_DecY,
               out0  => decY0,
               out1  => decY1,
```

```
               out2  => decY2,
               out3  => decY3,
               out4  => decY4,
               out5  => decY5,
               out6  => decY6,
               out7  => decY7);

 IR: Lab5_IR
     PORT MAP (input => DIN(N - 1 DOWNTO N - 9),
                reset => Resetn,
               load  => IR_in,
                  clk   => Clk,
                  output => IR_to_ctrl);

 Reg0: Lab5_Reg
     PORT MAP (input  => Data_Bus,
                reset => Resetn,
               load   => R0_in,
                  clk    => Clk,
                  output => R0_to_mux);

 Reg1: Lab5_Reg
     PORT MAP (input  => Data_Bus,
                reset => Resetn,
               load   => R1_in,
                  clk    => Clk,
                  output => R1_to_mux);

 Reg2: Lab5_Reg
     PORT MAP (input  => Data_Bus,
                reset => Resetn,
               load   => R2_in,
                  clk    => Clk,
                  output => R2_to_mux);

 Reg3: Lab5_Reg
     PORT MAP (input  => Data_Bus,
                reset => Resetn,
               load   => R3_in,
                  clk    => Clk,
                  output => R3_to_mux);

 Reg4: Lab5_Reg
     PORT MAP (input  => Data_Bus,
                reset => Resetn,
               load   => R4_in,
                  clk    => Clk,
                  output => R4_to_mux);

 Reg5: Lab5_Reg
     PORT MAP (input  => Data_Bus,
                reset => Resetn,
               load   => R5_in,
                  clk    => Clk,
                  output => R5_to_mux);

 Reg6: Lab5_Reg
```

```
Reg6: Lab5_Reg
    PORT MAP (input  => Data_Bus,
              reset => Resetn,
              load  => R6_in,
                clk   => Clk,
                output => R6_to_mux);

Reg7: Lab5_Reg
    PORT MAP (input  => Data_Bus,
              reset => Resetn,
              load  => R7_in,
                clk   => Clk,
                output => R7_to_mux);

RegA: Lab5_Reg
    PORT MAP (input  => Data_Bus,
              reset => Resetn,
              load  => A_in,
                clk   => Clk,
                output => RA_to_Adder);

RegG: Lab5_Reg
    PORT MAP (input  => Adder_to_RG,
              reset => Resetn,
              load  => G_in,
                clk   => Clk,
                output => RG_to_mux);

Mux: Lab5_Mux
    PORT MAP (reg0_in   => R0_to_mux,
              reg1_in   => R1_to_mux,
              reg2_in   => R2_to_mux,
              reg3_in   => R3_to_mux,
              reg4_in   => R4_to_mux,
              reg5_in   => R5_to_mux,
              reg6_in   => R6_to_mux,
              reg7_in   => R7_to_mux,
             data_in   => DIN,
                adder_in  => RG_to_mux,
                reg0_out  => R0_out,
                reg1_out  => R1_out,
                reg2_out  =>  R2_out,
                reg3_out  =>  R3_out,
                reg4_out  => R4_out,
                reg5_out  => R5_out,
                reg6_out  => R6_out,
                reg7_out  => R7_out,
                data_out  => DIN_out,
                adder_out => G_out,
                output    => Data_Bus);

Adder: Lab5_AddSub
    PORT MAP (in1   => RA_to_Adder,
              in2   => Data_Bus,
                mode  => AddSub,
                sum   => Adder_to_RG);
END Lab5_Processor_B;
```

A.5.1.1.3: Processor VHDL Code to Board

```
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare Entity for Part 1 --
ENTITY Lab5a IS
GENERIC(N : INTEGER := 16);
PORT (SW   : IN std_logic_vector(17 DOWNTO 0);
      KEY  : IN std_logic_vector(1 DOWNTO 0);
      LEDR : OUT std_logic_vector(17 DOWNTO 0));
END Lab5a;

-- Architecture of Entity for Part 1 --
ARCHITECTURE Lab5a_B OF Lab5a IS
    COMPONENT Lab5_Processor
    PORT (DIN      : IN std_logic_vector(N - 1 DOWNTO 0);
          Run      : IN std_logic;
            Resetn   : IN std_logic;
            Clk      : IN std_logic;
            Data_Bus : BUFFER std_logic_vector(N - 1 DOWNTO 0);
            Done     : OUT std_logic);
    END COMPONENT;
BEGIN
LEDR(16) <= '1'; -- Keep off
Proc: Lab5_Processor
    PORT MAP (DIN      => SW(15 DOWNTO 0),
              Run      => SW(17),
                Resetn   => KEY(0),
                Clk      => KEY(1),
                Data_Bus => LEDR(15 DOWNTO 0),
                Done     => LEDR(17));
END Lab5a_B;
```

A.5.1.2.1: Processor Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab5_Processor_vhd_tst IS
END Lab5_Processor_vhd_tst;
ARCHITECTURE Lab5_Processor_arch OF Lab5_Processor_vhd_tst IS
-- constants
-- signals
SIGNAL Clk : STD_LOGIC;
SIGNAL Data_Bus : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL DIN : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Done : STD_LOGIC;
SIGNAL InR : STD_LOGIC_VECTOR(8 DOWNTO 0);
SIGNAL R0 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL R1 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL R2 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL R3 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL R4 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL R5 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL R6 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL R7 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL RA : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Resetn : STD_LOGIC;
SIGNAL RG : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Run : STD_LOGIC;
COMPONENT Lab5_Processor
    PORT (
    Clk : IN STD_LOGIC;
    Data_Bus : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    DIN : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    Done : OUT STD_LOGIC;
    InR : OUT STD_LOGIC_VECTOR(8 DOWNTO 0);
    R0 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    R1 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    R2 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    R3 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    R4 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    R5 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    R6 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    R7 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    RA : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    Resetn : IN STD_LOGIC;
    RG : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    Run : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab5_Processor
    PORT MAP (
-- list connections between master ports and signals
    Clk => Clk,
    Data_Bus => Data_Bus,
    DIN => DIN,
    Done => Done,
    InR => InR,
    R0 => R0,
    R1 => R1,
    R2 => R2,
    R3 => R3,
    R4 => R4,
    R5 => R5,
    R6 => R6,
    R7 => R7,
    RA => RA,
    Resetn => Resetn,
    RG => RG,
    Run => Run
    );
init : PROCESS
-- variable declarations
BEGIN
        -- Test 1: Reset --
        DIN <= x"0000";
        Resetn <= '0';
        Run <= '1'; wait for 9 ps;

          -- Test 2: mvi(Reg0) --
          DIN <= x"2000"; wait for 1 ps;
          Resetn <= '1';  wait for 9 ps;
          DIN <= x"0005"; wait for 10 ps;

          -- Test 3: mv(Reg1, Reg0) --
          DIN <= x"0400"; wait for 20 ps;

          -- Test 4: add(Reg1, Reg0) --
          DIN <= x"4400"; wait for 30 ps;

          -- Test 5: sub(Reg1, Reg0) --
          DIN <= x"6400"; wait for 20 ps;
          Run <= '0';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    Clk <= '0'; wait for 5 ps;
    Clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab5_Processor_arch;
```

Appendix A.6: The Memory Circuit

A.6.1.1: Five Bit Counter VHDL Code

```
-- Ryan Barker --

  LIBRARY ieee;
  USE ieee.numeric_std.all;
  USE ieee.std_logic_1164.all;

  -- Declare Counter --
ENTITY Lab5_Counter IS
    GENERIC (CNT : INTEGER := 5);
    PORT (clk     : IN std_logic;
          reset   : IN std_logic;
            output  : OUT std_logic_vector(CNT - 1 DOWNTO 0));
END Lab5_Counter;

  -- Architecture of Counter --
ARCHITECTURE Lab5_Counter_B OF Lab5_Counter IS
    SIGNAL counter : unsigned (CNT - 1 DOWNTO 0);
BEGIN
    cntr: PROCESS(clk, reset)
     BEGIN
         IF(reset = '0') THEN
             zeroes: FOR i IN 0 TO CNT - 1 LOOP
               counter(i) <= '0';
           END LOOP;
          ELSIF(rising_edge(clk)) THEN
               -- Increment counter --
              counter <= counter + 1;
          END IF;

          -- Set output to counter value --
          output <= std_logic_vector(counter);
     END PROCESS cntr;
END Lab5_Counter_B;
```

A.6.1.2: Memory Component VHDL Code

```
-- megafunction wizard: %ROM: 1-PORT%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: altsyncram

-- ============================================================
-- File Name: Lab5_Mem.vhd
-- Megafunction Name(s):
--          altsyncram
--
-- Simulation Library Files(s):
--          altera_mf
-- ============================================================
-- ************************************************************
-- THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
--
-- 14.1.0 Build 186 12/03/2014 SJ Web Edition
-- ************************************************************


--Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
--Your use of Altera Corporation's design tools, logic functions
--and other software and tools, and its AMPP partner logic
--functions, and any output files from any of the foregoing
--(including device programming or simulation files), and any
--associated documentation or information are expressly subject
--to the terms and conditions of the Altera Program License
--Subscription Agreement, the Altera Quartus II License Agreement,
--the Altera MegaCore Function License Agreement, or other
--applicable license agreement, including, without limitation,
--that your use is for the sole purpose of programming logic
--devices manufactured by Altera and sold by Altera or its
--authorized distributors.  Please refer to the applicable
--agreement for further details.


LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY Lab5_Mem IS
    PORT
    (
        address     : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        clock       : IN STD_LOGIC  := '1';
        q       : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
    );
END Lab5_Mem;


```

```
    q    <= sub_wire0(15 DOWNTO 0);

    altsyncram_component : altsyncram
    GENERIC MAP (
        address_aclr_a => "NONE",
        clock_enable_input_a => "BYPASS",
        clock_enable_output_a => "BYPASS",
        init_file => "inst_mem.mif",
        intended_device_family => "Cyclone IV E",
        lpm_hint => "ENABLE_RUNTIME_MOD=NO",
        lpm_type => "altsyncram",
        numwords_a => 32,
        operation_mode => "ROM",
        outdata_aclr_a => "NONE",
        outdata_reg_a => "CLOCK0",
        widthad_a => 5,
        width_a => 16,
        width_byteena_a => 1
    )
    PORT MAP (
        address_a => address,
        clock0 => clock,
        q_a => sub_wire0
    );


END SYN;

-- ============================================================
```

A.6.1.3: Memory Initialization File

```
-- Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
-- Your use of Altera Corporation's design tools, logic functions
-- and other software and tools, and its AMPP partner logic
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Altera Program License
-- Subscription Agreement, the Altera Quartus II License Agreement,
-- the Altera MegaCore Function License Agreement, or other
-- applicable license agreement, including, without limitation,
-- that your use is for the sole purpose of programming logic
-- devices manufactured by Altera and sold by Altera or its
-- authorized distributors.  Please refer to the applicable
-- agreement for further details.

-- Quartus II generated Memory Initialization File (.mif)

WIDTH=16;
DEPTH=32;
ADDRESS_RADIX= HEX;
DATA_RADIX= BIN;
CONTENT
BEGIN
    00  : 0010000001111111; -- mvi R0
    01  : 0000000000000100; -- data
    02  : 0010010001010101; -- mvi R1
    03  : 0000000000001010; -- data
    04  : 0000100010101010; -- mv R2, R1
    05  : 0001010001111111; -- mv R5, R0
    06  : 0101010010000000; -- add R5, R1
    07  : 0110010000000000; -- sub R1, R0
    08  : 0100000010000000; -- add R0, R1
    09  : 0011010000000000; -- mvi R5
    0A  : 0101010101010101; -- data
    0B  : 0000101010000000; -- mv R4, R5
    0C  : 0110010000000000; -- sub R1, R0
    0D  : 0110101010000000; -- sub R4, R5
    0E  : 0001110010000000; -- mv R7, R1
    0F  : 0100101110000000; -- add R2, R7
    10  : 0000111010000000; -- mv R3, R5
    11  : 0011100000000000; -- mvi R6
    12  : 0011101010101110; -- data
    13  : 0111100000000000; -- sub R6, R0
    14  : 0111100010000000; -- sub R6, R1
    15  : 0101010100101100; -- add R5, R4
    16  : 0000100010101010; -- mv R2, R1
    17  : 0110101010000000; -- sub R4, R5
    18  : 0100000010000000; -- add R0, R1
    19  : 0000100010101010; -- mv R2, R1
    1A  : 0100101110000000; -- add R2, R7
    1B  : 0110101010000000; -- sub R4, R5
    1C  : 0100000010000000; -- add R0, R1
    1D  : 0000100010101010; -- mv R2, R1
    1E  : 0100101110000000; -- add R2, R7
    1F  : 0100000010000000; -- add R0, R1
END;
```

A.6.1.4: Memory Circuit VHDL Code

```
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare Main Circuit --
ENTITY Lab5_Main IS
GENERIC (N : INTEGER := 16);
PORT (Run      : IN std_logic;
      Resetn   : IN std_logic;
      PClk     : IN std_logic;
      MClk     : IN std_logic;
      Data_Bus : BUFFER std_logic_vector(N - 1 DOWNTO 0);
        InRg     : OUT std_logic_vector(8 DOWNTO 0);
      Rg0      : OUT std_logic_vector(N - 1 DOWNTO 0);
      Rg1      : OUT std_logic_vector(N - 1 DOWNTO 0);
      Rg2      : OUT std_logic_vector(N - 1 DOWNTO 0);
      Rg3      : OUT std_logic_vector(N - 1 DOWNTO 0);
       Rg4      : OUT std_logic_vector(N - 1 DOWNTO 0);
      Rg5      : OUT std_logic_vector(N - 1 DOWNTO 0);
      Rg6      : OUT std_logic_vector(N - 1 DOWNTO 0);
        Rg7      : OUT std_logic_vector(N - 1 DOWNTO 0);
      RgA      : OUT std_logic_vector(N - 1 DOWNTO 0);
      RgG      : OUT std_logic_vector(N - 1 DOWNTO 0);
        Done     : OUT std_logic);
END Lab5_Main;

-- Architecture of Main Circuit --
ARCHITECTURE Lab5_Main_B OF Lab5_Main IS
    SIGNAL count_to_mem : std_logic_vector(4 DOWNTO 0);
     SIGNAL mem_to_proc  : std_logic_vector(N - 1 DOWNTO 0);

     COMPONENT Lab5_Counter
        PORT (clk     : IN std_logic;
              reset   : IN std_logic;
                output  : OUT std_logic_vector(4 DOWNTO 0));
    END COMPONENT;

     COMPONENT Lab5_Mem
        PORT (address : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
              clock   : IN STD_LOGIC;
              q       : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
    END COMPONENT;

    COMPONENT Lab5_Processor
        PORT (DIN      : IN std_logic_vector(N - 1 DOWNTO 0);
              Run      : IN std_logic;
              Resetn   : IN std_logic;
                Clk      : IN std_logic;
                Data_Bus : BUFFER std_logic_vector(N - 1 DOWNTO 0);
                InR      : OUT std_logic_vector(8 DOWNTO 0);
                R0       : OUT std_logic_vector(N - 1 DOWNTO 0);
                R1       : OUT std_logic_vector(N - 1 DOWNTO 0);
                R2       : OUT std_logic_vector(N - 1 DOWNTO 0);
                R3       : OUT std_logic_vector(N - 1 DOWNTO 0);
                R4       : OUT std_logic_vector(N - 1 DOWNTO 0);
                R5       : OUT std_logic_vector(N - 1 DOWNTO 0);
                R6       : OUT std_logic_vector(N - 1 DOWNTO 0);
                R7       : OUT std_logic_vector(N - 1 DOWNTO 0);
                RA       : OUT std_logic_vector(N - 1 DOWNTO 0);
                RG       : OUT std_logic_vector(N - 1 DOWNTO 0);
                Done     : OUT std_logic);
    END COMPONENT;
BEGIN
    Counter: Lab5_Counter
    PORT MAP (clk     => MClk,
              reset   => Resetn,
                output  => count_to_mem);

     Mem: Lab5_Mem
    PORT MAP (address => count_to_mem,
                clock    => MClk,
                q        => mem_to_proc);

    Proc: Lab5_Processor
    PORT MAP (DIN      => mem_to_proc,
              Run      => Run,
              Resetn   => Resetn,
                Clk      => PClk,
                Data_Bus => Data_Bus,
                InR      => InRg,
                R0       => Rg0,
                R1       => Rg1,
                R2       => Rg2,
                R3       => Rg3,
                R4       => Rg4,
                R5       => Rg5,
                R6       => Rg6,
                R7       => Rg7,
                RA       => RgA,
                RG       => RgG,
                Done     => Done);
END Lab5_Main_B;--
```

A.6.1.5: Memory Circuit VHDL Code to Board

```
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare Main Circuit --
ENTITY Lab5b IS
PORT (SW       : IN std_logic_vector(17 DOWNTO 0);
      KEY      : IN std_logic_vector(2 DOWNTO 0);
        LEDR     : OUT std_logic_vector(17 DOWNTO 0));
END Lab5b;

-- Architecture of Main Circuit --
ARCHITECTURE Lab5b_B OF Lab5b IS
    COMPONENT Lab5_Main
        PORT (Run      : IN std_logic;
                Resetn   : IN std_logic;
                  PClk     : IN std_logic;
                  MClk     : IN std_logic;
                  Data_Bus : BUFFER std_logic_vector(15 DOWNTO 0);
                  Done     : OUT std_logic);
    END COMPONENT;
BEGIN
    LEDR(16) <= '1'; -- Keep Off
    Main : Lab5_Main
    PORT MAP (Run      => SW(17),
                Resetn   => KEY(0),
                  PClk     => KEY(2),
                  MClk     => KEY(1),
                  Data_Bus => LEDR(15 DOWNTO 0),
                  Done     => LEDR(17));
END Lab5b_B;
```

A.6.2.1: Five Bit Counter Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab5_Counter_vhd_tst IS
END Lab5_Counter_vhd_tst;
ARCHITECTURE Lab5_Counter_arch OF Lab5_Counter_vhd_tst IS
-- constants
-- signals
SIGNAL clk : STD_LOGIC;
SIGNAL output : STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL reset : STD_LOGIC;
COMPONENT Lab5_Counter
    PORT (
    clk : IN STD_LOGIC;
    output : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
    reset : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab5_Counter
    PORT MAP (
-- list connections between master ports and signals
    clk => clk,
    output => output,
    reset => reset
    );
init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
        reset <= '0'; wait for 10 ps;
        reset <= '1';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
clock : PROCESS
BEGIN
    Clk <= '0'; wait for 5 ps;
    Clk <= '1'; wait for 5 ps;
END PROCESS clock;
END Lab5_Counter_arch;
```

A.6.2.2: Memory Circuit Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Lab5_Main_vhd_tst IS
END Lab5_Main_vhd_tst;
ARCHITECTURE Lab5_Main_arch OF Lab5_Main_vhd_tst IS
-- constants
-- signals
SIGNAL Data_Bus : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Done : STD_LOGIC;
SIGNAL InRg : STD_LOGIC_VECTOR(8 DOWNTO 0);
SIGNAL MClk : STD_LOGIC;
SIGNAL PClk : STD_LOGIC;
SIGNAL Resetn : STD_LOGIC;
SIGNAL Rg0 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Rg1 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Rg2 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Rg3 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Rg4 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Rg5 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Rg6 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Rg7 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL RgA : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL RgG : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL Run : STD_LOGIC;
COMPONENT Lab5_Main
    PORT (
    Data_Bus : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Done : BUFFER STD_LOGIC;
    InRg : BUFFER STD_LOGIC_VECTOR(8 DOWNTO 0);
    MClk : IN STD_LOGIC;
    PClk : IN STD_LOGIC;
    Resetn : IN STD_LOGIC;
    Rg0 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Rg1 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Rg2 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Rg3 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Rg4 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Rg5 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Rg6 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Rg7 : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    RgA : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    RgG : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
    Run : IN STD_LOGIC
    );
END COMPONENT;
BEGIN
    i1 : Lab5_Main
    PORT MAP (
-- list connections between master ports and signals
    Data_Bus => Data_Bus,
    Done => Done,
    InRg => InRg,
    MClk => MClk,
    PClk => PClk,
    Resetn => Resetn,
    Rg0 => Rg0,
    Rg1 => Rg1,
    Rg2 => Rg2,
    Rg3 => Rg3,
    Rg4 => Rg4,
    Rg5 => Rg5,
    Rg6 => Rg6,
    Rg7 => Rg7,
    RgA => RgA,
    RgG => RgG,
    Run => Run
    );
init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
        Resetn <= '0';
        Run <= '1'; wait for 10 ps;
        Resetn <= '1';
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (        )
-- variable declarations
BEGIN
        -- code executes for every event on sensitivity list
WAIT;
END PROCESS always;
Mclock : PROCESS
BEGIN
    MClk <= '0'; wait for 5 ps;
    MClk <= '1'; wait for 5 ps;
END PROCESS Mclock;
Pclock : PROCESS
BEGIN
    PClk <= '0'; wait for 5 ps;
    PClk <= '1'; wait for 5 ps;
END PROCESS Pclock;
END Lab5_Main_arch;
```

Appendix A.C: TimeQuest

A.C.1: TimeQuest SDC File

```
#**************************************************************
# THIS IS A WIZARD-GENERATED FILE.
#
# Version 13.0.1 Build 232 06/12/2013 Service Pack 1 SJ Full Version
#
#**************************************************************

# Copyright (C) 1991-2013 Altera Corporation
# Your use of Altera Corporation's design tools, logic functions
# and other software and tools, and its AMPP partner logic
# functions, and any output files from any of the foregoing
# (including device programming or simulation files), and any
# associated documentation or information are expressly subject
# to the terms and conditions of the Altera Program License
# Subscription Agreement, Altera MegaCore Function License
# Agreement, or other applicable license agreement, including,
# without limitation, that your use is for the sole purpose of
# programming logic devices manufactured by Altera and sold by
# Altera or its authorized distributors.  Please refer to the
# applicable agreement for further details.



# Clock constraints

create_clock -name "clk" -period 10.000ns [get_ports {Clk}] -waveform {1.000 6.000}


# Automatically constrain PLL and other generated clocks
derive_pll_clocks -create_base_clocks

# Automatically calculate clock uncertainty to jitter and other effects.
derive_clock_uncertainty


# tsu/th constraints

set_input_delay -clock "clk" -max 10ns [get_ports {Clk}]
set_input_delay -clock "clk" -min 0.000ns [get_ports {Clk}]


# tco constraints

# tpd constraints


```