Experiment #3: Introduction to Hierarchical Design

Team Name
Steve Comer
Derek Tsui
Kevin Vece

Implementation Due: 3 Oct 2012
Report Due: 15 Oct 2012

_____
Steve Comer

_____
Derek Tsui

_____
Kevin Vece

Table of Contents

1.0 Introduction

This lab consisted of two distinct subsections. Both involved VHDL programming. VHDL is a nested acronym that stands for Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). The first task was to design and implement an Arithmetic Logic Unit (ALU) in VHDL. The second task was to design and implement a Program Counter (PC) in VHDL. The Hexadecimal Display Driver created in Lab 2 was used for displaying register, input and output values during this lab. The ALU and the PC are currently independent components, but they will be used together in future labs.

2.0 Method

2.1 Background Information

2.1.1 General

In VHDL, there are two types of data storage.
1. 'Variables' are temporary storage elements which only exist within a process. They are updated instantaneously when their values change.
2. 'Signals' are permanent data storage, which can be thought of as a physical wire. There is a delta timing delay before updates can be seen and used in a signal.

Parts typical of a VHDL file:

1. ENTITY is the entire program.
2. PORT is the input and outputs of the hardware component.
3. ARCHITECTURE is all the COMPONENTS and PROCESSES of an overall hardware component.
4. PROCESS is code which has a sensitivity list associated with it. This sensitivity list is a list of variables where, if any of them change, the process will be called. However, this introduced the concurrency, and one must proceed with caution.
5. COMPONENTS are ENTITIES initialized within a higher-level ENTITIY
6. MAPPING is establishing connections between a COMPONENT'S PORTS and signals or PORTS belonging to a higher-level COMPONENT.

A register is a memory element which stores bits of information.

2.1.2 Arithmetic Logic Unit

An arithmetic and logic unit (ALU) is a synchronous digital circuit used to perform arithmetic and logical operations. A typical ALU loads data from two input registers and performs a specific operation on one or both of the inputs based on the function-select multiplexer. Figure 2.2 lists the possible operations along with the required function-select inputs. The result is then stored into an output register. Our ALU implementation also includes an optional flag, Z, which sets when the output is zero. Figure 2.1 describes the specifics of the ALU implemented in this lab.

The "black box" diagram of our Arithmetic Logic Unit (ALU) is shown in Figure 2.1. There are a total of 15 input pins, 5 output pins, and a clock.

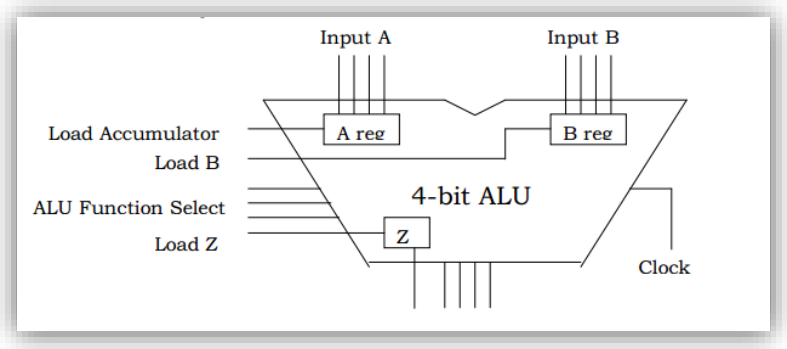


Figure 2.1 - ALU Diagram

Input A and Input B are 4-bit inputs, each of which feeds into a 4-bit register. Note that the register A is also referred to as Accumulator. Each 4-bit register is updated only when its respective load bit is high on a rising edge of the clock. The ALU Function Select is a 3-bit input that determines which of the eight possible operations to perform (see Figure 2.2). Z is a 1-bit register which functions as a zero flag. If and only if load Z is high, Z will update based on the 4-bit output. If load Z is high and the output is "0000", the output of Z will be high. If load Z is low, it will retain its current value regardless of the output and clock cycles.

0 - ADD	4 - NOT A
1 - SUB (A - B)	5 - Accumulator
2 - AND	6 - NOT B
3 - OR	7 - B Register

Figure 2.2 - ALU Function Select

The ALU function select is a 3-bit input which selects the operation to perform on the contents of the 4-bit registers A and B. The output is 4-bit, meaning that there is no overflow. Note that there is no option to perform multiplication or division. These operations can be performed with the addition of a shift register outside of the ALU. The operations performed by the ALU are not synchronous, meaning that the function select bits will immediately update the output regardless of the clock.

2.1.3 Program Counter

A Program Counter (PC) is a register that contains the address of the instruction being executed at the current time. As every instruction is fetched, the program counter increments its store value by one, and points to the next instruction in the sequence. The program counter can be cleared or incremented. The clear, increments, and load signals are all synchronous inputs. There also exists an 8-bit adder module which must produce the sum of the input pins and the X- register. Figure 2.4 illustrates the specifics of our PC.

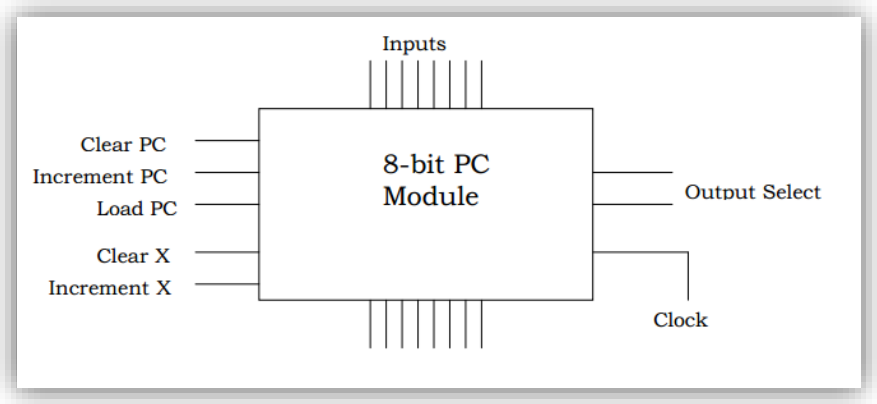


Figure 2.3 - Program Counter Diagram

The black box diagram of the Program Counter (PC) is shown in Figure 2.3. There are a total of 15 inputs. The PC is synchronous, thus all clear, increment, and load operations occur on the rising edge of the clock.

The PC can be cleared, loaded from input, or incremented. We designed the PC so that if any combination of clear, load, and increment are simultaneously high, then clear takes first priority, then load, and lastly increment. Clearing is the most important function because it causes the largest change to the value of the PC. It follows that loading causes larger changes than increment but less than clear, so they are prioritized in that order.

The X Register can be cleared or incremented, but not loaded. This X Register is used as a temporary data storage for the function call. This allows the PC to remember where it was in function-calling program so that it can be returned after a function call.

The Output Select is a 2-bit input that determines which of the four possible operations to perform on the input and register of the stored program count (see Figure 2.4). The output is an 8-bit value. This implies that programs which use this PC will be limited to a maximum of 256 instructions.

0 - Program Counter
1 - Sum, X + Inputs
2 - Inputs
3 - X register

Figure 2.4 - PC Output Select

2.2 Procedure

2.2.1 Design

Registers were used as the primary storage elements because of their ability to be accessed quickly. This prevents issues related to the delta delay caused by slower memory access. This issue exists because of concurrency, multiple processes executing simultaneously on shared variables.

Thus if Process 1 needs to update variable A, and Process 2 needs to read in the value of variable A after Process 1 updates it, one must ensure the order does not become compromised. An example of this would be if Process 2 reads the value of variable A before Process 1 has a chance to update it.

To increase the amount of parallelism in the components, we broke up the large projects into smaller distinct processes. The goal was to separate as many things as possible into separate processes, provided that they did not cause interdependency issues between processes. The processes were first separated between combinational and sequential logic. The sequential processes were then separated to the point that each independent register update operation had its own process.

2.2.2 ALU VHDL Coding

We used custom register files to implement parallelism and simplify the logic to decompose the project into smaller reusable pieces.

We created one process to help determine the output from the operation performed on Register A and Register B. This depends on the Function Select which uses a variable called "outFlag". Function Select determines the actual arithmetic or logical operations perform on register A and register B. The result of the operation is stored in the variable outFlag.

There is another process that depends on outFlag. Thus, after the first process changes the value of outFlag, this process will run. Output is assigned to the value of outFlag. This was done to ensure the output would update before the Z Flag would be checked if it should be on or off. Then the program checks if the Z Flag should be on, which is when the output is zero and the Z Load is high.

The purpose of the last process is to clear the Z register so that the Z flag does not become updated anymore. This is because the Z flag is only updated on the first operation that the user performs, so once the operation changes, this process ensures that the Z flag will not continue to update without another loadZ input.

2.2.3 PC VHDL Coding

For the PC, we created a process to update the PC and X Register. This process will run whenever the clock is on its rising edge.

The other process will run whenever the Function Select changes. This checks the Function Select and performs the correct operation and set output to the appropriate result.

2.2.4 ALU Pin Assignments

The full list of pin assignments for the ALU is given in Appendix B.1. Pin Assignments are explained in the order in which they appear in the appendix.

A[3 DOWNTO 0]: These are four input switches used to select the 4-bit input value A. These switches can be toggled in different combinations to represent all possible values of the input A. Note that A[3 DOWNTO 0] corresponds to SW[17 DOWNTO 14].

B[3 DOWNTO 0]: These are four input switches used to select the 4-bit input value B. These switches can be toggled in different combinations to represent all possible values of the input B. Note that B[3 DOWNTO 0] corresponds to SW[13 DOWNTO 10].

clk: This is a push-button input to the circuit. Pushing this button down and then releasing it corresponds to one clock cycle for the circuit. The button gives a high signal when it is pressed. This corresponds to KEY[3] on the DE2 board.

clkOut: This is a green LED which shows the current value of the clock signal. A high clock signal turns on the LED. This LED was chosen because it is physically close to the push-button input used for clock. This corresponds to LEDG[6] on the DE2 board.

fSelect[2 DOWNTO 0]: These are three input switches used to select the function to be used on A and B. These switches can be toggled in different combinations to represent all possible functions. Note that A[2 DOWNTO 0] corresponds to SW[2 DOWNTO 0] on the DE2 board.

loadA: This input switch determines whether or not the value of the input A should be loaded into the register A. If the input switch is high during a rising clock edge, then A will be loaded into register A. This corresponds to SW[9] on the DE2 board.

loadA_out: This is a red LED which shows the current value of loadA. A high loadA signal turns on the LED. This corresponds to LEDR[9] on the DE2 board.

loadB: This input switch determines whether or not the value of the input B should be loaded into the register B. If the input switch is high during a rising clock edge, then B will be loaded into register B. This corresponds to SW[8] on the DE2 board.

loadB_out: This is a red LED which shows the current value of loadA. A high loadA signal turns on the LED. This corresponds to LEDR[8] on the DE2 board.

loadZ: This input switch determines whether or not the value of the Z flag will update on a given clock cycle. If the input switch is high during a rising clock edge, then the Z flag will update. This corresponds to SW[7] on the DE2 board.

loadZ_out: This is a red LED which shows the current value of the Z flag. A high loadZ signal turns on the LED. This corresponds to LEDR[7] on the DE2 board.

sevenSegmentOut0[6 DOWNTO 0]: These seven outputs are segments 'a' through 'g', forming one hexadecimal character on the DE2 board. These segments are active low because they turn on when given a value of 0. This displays the value of the input A. Note that sevenSegmentOut[x] corresponds to HEX7 [x] on the DE2 board.

sevenSegmentOut1[6 DOWNTO 0]: These seven outputs are segments 'a' through 'g', forming one hexadecimal character on the DE2 board. These segments are active low because they turn on when given a value of 0. This displays the value of the input B. Note that sevenSegmentOut1[x] corresponds to HEX5[x] on the DE2 board.

sevenSegmentOut2[6 DOWNTO 0]: These seven outputs are segments 'a' through 'g', forming one hexadecimal character on the DE2 board. These segments are active low because they turn on when given a value of 0. This is the value of the output. Note that sevenSegmentOut2[x] corresponds to HEX3[x] on the DE2 board.

zOut: This is a red LED which shows the current value of the Z flag. A high Z flag turns on the LED. This corresponds to LEDR[12].

2.2.5 PC Pin Assignments

The full list of pin assignments for the PC is given in Appendix B.2. Pin Assignments are explained in the order in which they appear in the appendix.

clearPC: This is a push-button input to the circuit. Pushing this button down and then releasing it corresponds to clearing the PC. The button gives a high signal when it is pressed. This corresponds to SW[9] on the DE2 board.

clearX: This is a push-button input to the circuit. Pushing this button down and then releasing it corresponds to clearing the X register.The button gives a high signal when it is pressed. This corresponds to SW[6] on the DE2 board.

clk: This is a push-button input to the circuit. Pushing this button down and then releasing it corresponds to one clock cycle for the circuit. The button gives a high signal when it is pressed. This corresponds to KEY[3] on the DE2 board.

clkOut: This is a green LED which shows the current value of the clock signal. A high clock signal turns on the LED. This LED was chosen because it is physically close to the push-button input used for clock. This corresponds to LEDG[6] on the DE2 board.

incPC: This is a switch input to the circuit. Pushing this button down and then releasing it corresponds incrementing the PC by one. The button gives a high signal when it is pressed. This corresponds to SW[8] on the DE2 board.

input[7 DOWNTO 0]: These switches are used for the input. This corresponds to SW[17 DOWNTO 10] on the DE2 board.

loadPC: This is a switch input to the circuit. Setting this high is used to load the PC. This corresponds to SW[7] on the DE2 board.

outSel[1 DOWNTO 0]: These are switch inputs to the circuit. Setting this to high is used to determine the output select. These correspond to SW[1 to 0] on the DE2 board.

sevenSegmentOut0[6 DOWNTO 0]: These seven outputs are segments 'a' through 'g', forming one hexadecimal character on the DE2 board. These segments are active low because they turn on when given a value of 0. This displays the value of the MSD of input. Note that sevenSegmentOut[x] corresponds to HEX7[x] on the DE2 board.

sevenSegmentOut1[6 DOWNTO 0]: These seven outputs are segments 'a' through 'g', forming one hexadecimal character on the DE2 board. These segments are active low because they turn on when given a value of 0. This displays the value of the LSD of input. Note that sevenSegmentOut1[x] corresponds to HEX6[x] on the DE2 board.

sevenSegmentOut2[6 DOWNTO 0]: These seven outputs are segments 'a' through 'g', forming one hexadecimal character on the DE2 board. These segments are active low because they turn on when given a value of 0. This is the value of the MSD of output. Note that sevenSegmentOut2[x] corresponds to HEX5[x] on the DE2 board.

sevenSegmentOut3[6 DOWNTO 0]: These seven outputs are segments 'a' through 'g', forming one hexadecimal character on the DE2 board. These segments are active low because they turn on when given a value of 0. This is the value of the LSD of the output. Note that sevenSegmentOut3[x] corresponds to HEX4[x] on the DE2 board.

2.3 Problems and Solutions

Problem: We were trying to use variables declared outside the process.
Solution: We discovered that *register* is a keyword and used that instead of a variable.

Problem: Our loads were not working correctly, A and B loads were independent of the respective loadA and loadB values.
Solution: Instead of using the function select in the sensitivity list, we used load

Problem: When the circuit was powered on, the Z flag would immediately be on.
Solution: Created registers to properly clear Z Flag when needed.

Problem: Certain conditions required an extra clock cycle to properly set the Z Flag to high.
Solution: We corrected the Z Flag to not update until the output updated first.

2.4 Tools and Materials Used

The materials and tools used in this experiment are as follows:

1. Quartus II Software
2. Altera DE 2 Board with Cyclone II – EP2C35F672C6 FPGA

3.0 Results

3.1 Observations

Because the project was essentially a programming assignment, the only observation was that of our finished results. Our completed project functioned correctly as we observed the project performed as expected.

3.2 Testing Procedure

We implemented two separate test plans, one for the ALU and one for the PC. Each plan is detailed below.

3.2.1 ALU

It is infeasible to demonstrate all possible values of input and output for the circuit. We designed our test plan to include border conditions that illustrate the operation of the circuit under conditions which could be problematic if designed incorrectly.

For the selected input values of A and B shown in Figure 3.1, demonstrate the correct operation of the circuit.

1. Power on the board, and load the program. The Z flag will not be on.
2. Use the input switches to set the desired input values for A and B.
3. Use the input switches to set load A and load B to high. Cycle the clock. Register A and Register B now contain the desired input values.
4. Demonstrate the correctness of each of the eight functions of A and B using the function select switches.
5. Repeat Step 4 with load Z high.
6. Repeat Steps 2-5 for remaining combinations of A and B.

A	B
0000	0000
1000	0001
1100	0011
1110	0111
1111	1111

Figure 3.1 ALU Inputs

3.2.2 Program Counter

It is infeasible to demonstrate all possible values of input and output for the circuit. We designed our test plan to include border conditions that illustrate the operation of the circuit under conditions which could be problematic if designed incorrectly.

Figure 3.2 shows the functions that should be performed based on the 2-bit function select input. Figure 3.3 shows the priority of the functions.

Select	Function
00	PC
01	X + Input
10	Inputs
11	X Register

Figure 3.2 PC Functions

High Priority		Low Priority
Clear	>	Load
Clear	>	Increment
Clear	>	(Load + Increment)
Load	>	Increment

Figure 3.3 PC Priority

For the selected input values shown in Table 3.2, demonstrate the correct operation of the circuit.

1. Power on the board, and load the program.
2. Use the input switches to set the desired input value.
3. Cycle the clock.
4. Demonstrate the correctness of each of the four output using the function select switches.
5. Demonstrate the correctness of each of possible combinations Clear, Load, and Increment which follows the Figure 3.3.
6. Repeat Steps 2-5 for remaining input.

3.3 Experiment Evaluation

After fixing the Z Flag error, our results satisfy the objectives from the lab. The PC and ALU worked as intended.

3.4 Warning Explanations

Quartus gives four main types of messages when compiling code: info, warnings, critical warnings, and errors.

3.4.1 Errors

Errors are the most important type of message. They explicitly tell you where in the code or design Quartus was unable to understand. It is essential that all errors are removed from a project, since the code will not compile correctly unless the errors are fixed. It's also important to realize that just because all the errors are removed from a project doesn't mean the code works.

3.4.2 ALU Critical Warnings

The second most important message type is a critical warning. While these don't explicitly mean that there's something wrong with the code, there still exist a high chance that something isn't right.

Critical Warning 1:
Critical Warning (332012): Synopsys Design Constraints File file not found: 'alu.sdc'. A Synopsys Design Constraints File is required by the TimeQuest Timing Analyzer to get proper timing constraints. Without it, the Compiler will not properly optimize the design.

Reason:
We did not supply a timing constraint file, so Quartus was unable to test the timing of our circuit to see if it'll work in its environment.

Critical Warning 2:
Critical Warning (332148): Timing requirements not met

Reason:
Critical warning 2 is related to critical warning 1, in that since a timing constraints file was not supplied, Quartus was unable to do timing analysis, so the timing requirements were not met.

3.4.3 ALU Warnings

Warnings are fairly common in Quartus projects. Due to the fact that Quartus compiles to an industry standard, it is very particular about little details. Quartus warnings can be considered gentle reminders that inform the programmer that it finds something peculiar. A common example in programming languages is unused variables. Declaring a variable and not using it generally doesn't cause any harm, but at the same time, the variable has no use and should be removed or used at some point.

We received fifteen different types of warnings. They are described below.

Warning 1:
Warning (10492): VHDL Process Statement warning at alu.vhd(122): signal "regZ" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

Reason:
Quartus is stating that the sensitivity list for the main process in the counter module did not contain regZ, even though regZ was used inside of the process. This warning is superfluous due to our design. Since the Z flag is updated after the output has been updated, this process just needs to check if the user wants the Z flag to be updated and is not changing the value of it.

Warning 2:
Warning (10631): VHDL Process Statement warning at alu.vhd(119): inferring latch(es) for signal or variable "zOut", which holds its previous value in one or more paths through the process

Reason:
Quartus is stating that the signal zOut will be treated as a latch because it only changes value in certain cases during a process. This is the desired behavior because we only need zOut to change when the user desires the Z flag to be updated for the current output. In all other instances, the value of zOut should remain the same.

Warning 3:
Warning (10492): VHDL Process Statement warning at alu.vhd(135): signal "clear" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

Reason:
Quartus is stating is that the clear signal is not in the sensitivity list for the Z register clearing Process. This is desired because we want the Z register to be cleared whenever the output is done updating. This ensures that the Z flag does not continue updating unless the user reloads the Z register.

Warning 4:
Warning (10631): VHDL Process Statement warning at alu.vhd(132): inferring latch(es) for signal or variable "clear", which holds its previous value in one or more paths through the process

Reason:
Quartus is stating that the signal clear will be treated as a latch because it only changes its value in certain cases during a process. This is desired because we do not always need to clear the Z register if we already know that it is cleared.

Warning 5:
Warning (10492): VHDL Process Statement warning at reg.vhd(43): signal "load" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

Reason:
Quartus is stating that the load signal is not in the Process Statement's sensitivity list for the register component. This is desired because we do not want the process to continuously rerun every time the load signal changes. Instead, we prefer it to only check load when the clock signal changes because the register loads synchronously.

Warning 6:
Warning (10492): VHDL Process Statement warning at reg1.vhd(44): signal "reset" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

Reason:
Quartus is stating that the reset signal is not in the Process Statement's sensitivity list for the z register component. This is desired because the reset signal is not updated except inside the process statement so we do not need to worry about the reset signal triggering an event.

Warning 7:
Warning (10492): VHDL Process Statement warning at reg1.vhd(48): signal "A" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

Reason:
Quartus is stating that the A signal is not in the Process Statement's sensitivity list for the Z register component. This is desired because we do not want the process to continuously rerun every time the A signal changes. Instead, we prefer it to only check load when the clock signal changes because the register loads synchronously.

Warning 8:
Warning (10631): VHDL Process Statement warning at reg1.vhd(41): inferring latch(es) for signal or variable "B", which holds its previous value in one or more paths through the process

Reason:
Quartus is stating that the signal B will be treated as a latch because it will hold it value through the main process in certain cases. This is desired because B is used to represent the value the register holds. Since the register is synchronous on the rising edge of the clock, we only want this value to update when the clock hits its rising edge. Otherwise it should hold its value.

Warning 9:
Warning (10631): VHDL Process Statement warning at reg1.vhd(41): inferring latch(es) for signal or variable "reset", which holds its previous value in one or more paths through the process

Reason:
Quartus is stating that the signal reset will be treated as a latch because it will hold it value through the main process in certain cases. This is desired because reset is used to represent the value the clear input was and compare it to what is currently is. We want this value to remain constant during load statements because that means that clear has not changed. The only time we want this signal to change its value is when the clear input has changed so that it can update itself to the new value.

Warning 10:
Warning (13013): Ports D and ENA on the latch are fed by the same signal alu:alu0|clear
Warning (332125): Found combinational loop of 2 nodes
Warning (332126): Node "alu0|clear~0|combout"
Warning (332126): Node "alu0|clear~0|datad"

Reason:
Quartus is stating that the latch for the clear signal in the ALU is fed by its own output. This is desired because we want the value to change so we say that the new clear value is not the old clear value. Whenever clear changes the Z register knows to clear its value.

Warning 11:
Warning (335093): TimeQuest Timing Analyzer is analyzing 3 combinational loops as latches.

Reason:
Quartus is stating that it will change three combinational loops into latches. These three loops that it will view as latches are the clear signal, the reset signal, and the B signal from the bitreg component. This is because these signals act more as memory elements and the VHDL compiler is just converting them as such.

Warning 12:
Warning (15705): Ignored locations or region assignments to the following nodes…

Reason:
Quartus is stating that the parameters to our seven segment display driver are not specifically bound to anything physical. This is due to the fact that we are treating those specific variables as signals.

Warning 13:
Warning (306006): Found 26 output pins without output pin load capacitance assignment

Reason:
This warning states that the reported timing of these pins will be faster than in reality because load capacitance is unknown.

Warning 14:
Warning (171167): Found invalid Fitter assignments. See the Ignored Assignments panel in the Fitter Compilation Report for more information.

Reason:
Since we tested each component individually with specific pin assignments, and then used different pin assignments late when combining all the modules together, some filters will be invalid depending on the context.

Warning 15:
Warning (169174): The Reserve All Unused Pins setting has not been specified, and will default to 'As output driving ground'.

Reason:
Since it was not specified what to do with the unused pins, they will be grounded.

3.4.4 PC Critical Warnings
The second most important message type is a critical warning. While these don't explicitly mean that there's something wrong with the code, there still exist a high chance that something isn't right.

Critical Warning 1:
Critical Warning (332012): Synopsys Design Constraints File file not found: 'pc.sdc'. A Synopsys Design Constraints File is required by the TimeQuest Timing Analyzer to get proper timing constraints. Without it, the Compiler will not properly optimize the design.

Reason:
We did not supply a timing constraint file, so Quartus was unable to test the timing of our circuit to see if it'll work in its environment.

Critical Warning 2:
Critical Warning (332148): Timing requirements not met

Reason:
Critical warning 2 is related to critical warning 1, in that since a timing constraints file was not supplied, Quartus was unable to do timing analysis, so the timing requirements were not met.

3.4.5 PC Warnings

Warnings are fairly common in Quartus projects. Due to the fact that Quartus compiles to an industry standard, it is very particular about little details. Quartus warnings can be considered gentle reminders that inform the programmer that it finds something peculiar. A common example in programming languages is unused variables. Declaring a variable and not using it generally doesn't cause any harm, but at the same time, the variable has no use and should be removed or used at some point.

We received seven different types of warnings. They are described below.

Warnings 1-4:
Warning (10492): VHDL Process Statement warning at pc.vhd(64): signal "pcREG" is read inside the Process Statement but isn't in the Process Statement's sensitivity list
Warning (10492): VHDL Process Statement warning at pc.vhd(66): signal "xREG" is read inside the Process Statement but isn't in the Process Statement's sensitivity list
Warning (10492): VHDL Process Statement warning at pc.vhd(66): signal "input" is read inside the Process Statement but isn't in the Process Statement's sensitivity list
Warning (10492): VHDL Process Statement warning at pc.vhd(70): signal "xREG" is read inside the Process Statement but isn't in the Process Statement's sensitivity list

Reason:
Quartus is stating the the main process in the Program Counter does not have any of the input values in the sensitivity list. This is desired because we do not want the process to run every time these values change. Instead, since the Program Counter in completely synchronous. We only want these values to update when a clock input is given.

Warning 5:
Warning (306006): Found 26 output pins without output pin load capacitance assignment

Reason:
This warning states that the reported timing of these pins will be faster than in reality because load capacitance is unknown.

Warning 6:
Warning (169174): The Reserve All Unused Pins setting has not been specified, and will default to 'As output driving ground'.

Reason:
Since it was not specified what to do with the unused pins, they will be grounded.

3.4.6 Info

Info messages are purely for the programmers benefit. They let the programmer know what Quartus is doing with the project during compile time. They do not need to be checked every compile cycle.

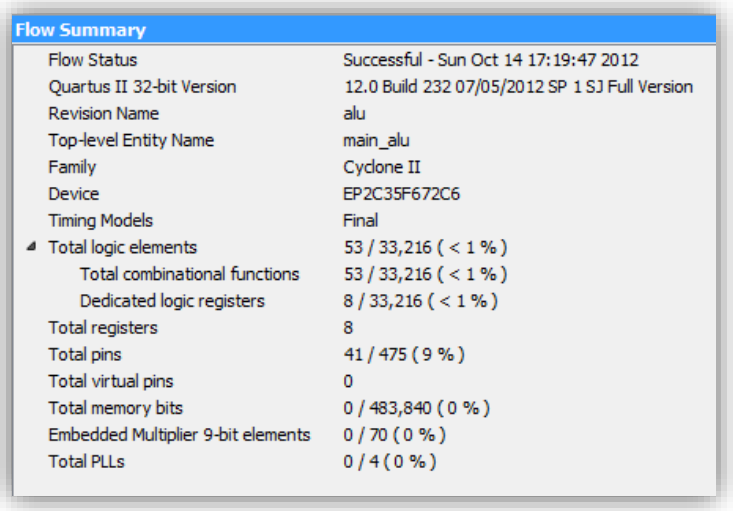
3.5 Quartus Flow Summary

Compared to Lab 2, the Flow Summaries are not as useful because we have no physical implementation to compare the Quartus-generated circuit against.

3.5.1 ALU Flow Summary

The Flow Summary for the ALU is shown in Figure 3.4. The final design includes 53 combinational logic elements and 8 dedicated logic registers. There are 41 total pins used for input and output. The 8

dedicated logic registers were used to store the A and B values of 4 bits each. For the combinational logic elements, we devised an estimate as follows: 4 elements for the NOT functions, 4 elements for AND functions, 4 elements for OR functions, 16 elements for a simple 4-bit adder/subtractor, 9 elements for a function select multiplexer, and 18 elements for the three HexDrivers, for a total estimate of 55 elements. This is close to the 53 dedicated logic elements that Quartus derived.

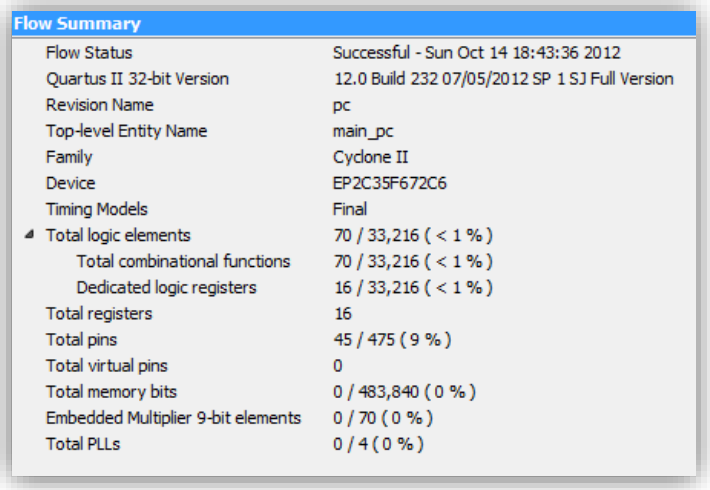


Flow Summary	
Flow Status	Successful - Sun Oct 14 17:19:47 2012
Quartus II 32-bit Version	12.0 Build 232 07/05/2012 SP 1 SJ Full Version
Revision Name	alu
Top-level Entity Name	main_alu
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	53 / 33,216 (< 1 %)
Total combinational functions	53 / 33,216 (< 1 %)
Dedicated logic registers	8 / 33,216 (< 1 %)
Total registers	8
Total pins	41 / 475 (9 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 3.4 ALU Flow Summary

3.5.2 PC Flow Summary

The Flow Summary for the PC is shown in Figure 3.5. The final design includes 70 combinational logic elements and 16 dedicated logic registers. The 16 dedicated logic registers were used to store the PC and X registers, each 8 bits long. For the combinational logic elements, we devised an estimate as follows: 32 elements for a simple 8-bit adder, 5 elements for a multiplexer to select the output, 15 elements for the encoder to update the PC register, and 9 elements for the encoder to update the X register. This gives a total of 61 logic elements, which is close to the 70 produced by Quartus circuit. This may be a case where the human implementation is more efficient than the implementation quickly generated by Quartus.



Flow Summary	
Flow Status	Successful - Sun Oct 14 18:43:36 2012
Quartus II 32-bit Version	12.0 Build 232 07/05/2012 SP 1 SJ Full Version
Revision Name	pc
Top-level Entity Name	main_pc
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	70 / 33,216 (< 1 %)
Total combinational functions	70 / 33,216 (< 1 %)
Dedicated logic registers	16 / 33,216 (< 1 %)
Total registers	16
Total pins	45 / 475 (9 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 3.5 PC Flow Summary

4.0 Conclusion

The objective of this experiment was to implement a Program Counter and an Arithmetic Logic Unit in VHDL. The components were created independently from one another in this lab. In the future, they will be combined with other components to form a microprocessor. Both components were implemented successfully. This lab report and future lab reports are very important because they must clearly spell out the operation of the components to prevent confusion when designing the microprocessor. If these reports are not clear, it will be difficult to effectively use these components in conjunction with those designed by others. This concept applies to any project that we may work on in the future, whether that be in industry or in the military.

A. Appendix A – Code

A.1 ALU Code

A.1.1 main_alu.vhd

```
--------------------------------------------------------------------------------
-- Lab 3                                                                      --
-- ALU                                                                        --
-- Steve Comer                                                                --
-- Derek Tsui                                                                 --
-- Kevin Vece                                                                 --
-- Updated 14 Oct 2012                                                        --
--------------------------------------------------------------------------------

-- Import the necessary libraries
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

--------------------------------------------------------------------------------

-- Create the ALU entity which loads values and does calculations based on those values
ENTITY alu IS

PORT(A          : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);         -- The A input, or accumulator
     B          : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);   -- The B input
     loadA      : IN  STD_LOGIC;                            -- If a clock signal is given and loadA
                                                            --      is set then the ALU with load the
                                                            --      input value from A into registerA
     loadB      : IN  STD_LOGIC;                            -- If a clock signal is given and loadB
                                                            --      is set then the ALU with load the
                                                            --      input value from B into registerB
     loadZ      : IN  STD_LOGIC;                            -- If a clock signal is given and loadZ
                                                            --      is set then the ALU with load the
                                                            --      Z flag is the output is zero
     fSelect    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);         -- The function select input is defined
                                                            --      as follows:
                                                            --      0: ADD (registerA + registerB)
                                                            --      1: SUBTRACT (registerA - registerB)
                                                            --      2: AND (registerA & registerB)
                                                            --      3: OR (registerA | registerB)
                                                            --      4: NOT A (~registerA)
                                                            --      5: A (registerA)
                                                            --      6: NOT B (~registerB)
                                                            --      7: B (registerB)
     clk        : IN  STD_LOGIC;                            -- The clock input used to synchronize
                                                            --      loads and operations
     zOut       : OUT STD_LOGIC := '0';                     -- The Z flag, shows if output is zero
     output     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)          -- The 4 bit result from the desired
                                                            --      operation
);

END alu;

--------------------------------------------------------------------------------

ARCHITECTURE Behavioral OF alu IS

    -- This is a simple four bit register which updates is value if a clock input is given
    --      while load = '1'
    COMPONENT reg
      PORT(A        : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);     -- the register's input
           clk      : IN  STD_LOGIC;                        -- an input clock signal
           load     : IN  STD_LOGIC;                        -- the load check
           B        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)      -- the value of the register
      );
```

```
    END COMPONENT;

    -- This is a one bit register with a clock and clear signal. If a clock input is given
    --      the register will load the input. If the clear signal changes then the register
    --      will reset the register back to '0'
    COMPONENT bitreg IS
      PORT(A            : IN  STD_LOGIC;                    -- the register's input
           clk          : IN  STD_LOGIC;                    -- an input clock signal
           clear        : IN  STD_LOGIC;                    -- the input reset signal
           B            : OUT STD_LOGIC                     -- the value of the register
      );
    END COMPONENT;

    -- Create wires for processes to communicate with other
    SIGNAL regA    : STD_LOGIC_VECTOR(3 DOWNTO 0);-- holds the value of registerA
    SIGNAL regB    : STD_LOGIC_VECTOR(3 DOWNTO 0);-- holds the value of registerB
    SIGNAL regZ  : STD_LOGIC;                      -- holds the value of registerZ
    SIGNAL outFlag : STD_LOGIC_VECTOR(3 DOWNTO 0);     -- holds the result of the ALU operation
    SIGNAL clear  : STD_LOGIC := '0';                  -- used to clear registerZ

---------------------------------------------------------------------------------------------

BEGIN

      -- Create the three registers to store A, B, and Z
      registerA : reg PORT MAP(A, clk, loadA, regA);
      registerB : reg PORT MAP(B, clk, loadB, regB);
      registerZ : bitreg PORT MAP(loadZ, clk, clear, regZ);

      -- This process calculates the desired ALU operation using the stored values in
      --          registerA and registerB
      PROCESS(fSelect, regA, regB)
      BEGIN
          CASE fSelect IS
              WHEN "000" =>                          -- ADD (registerA + registerB)

                  outFlag <= regA + regB;
              WHEN "001" =>                          -- SUBTRACT (registerA - registerB)

                  outFlag <= regA - regB;
              WHEN "010" =>                          -- AND (registerA & registerB)
                  outFlag <= regA and regB;
              WHEN "011" =>                          -- OR (registerA | registerB)
                  outFlag <= regA or regB;
              WHEN "100" =>                          -- NOT A (~registerA)
                  outFlag <= not regA;
              WHEN "101" =>                          -- A (registerA)
                  outFlag <= regA;
              WHEN "110" =>                          -- NOT B (~registerB)
                  outFlag <= not regB;
              WHEN "111" =>                          -- B (registerB)
                  outFlag <= regB;
              WHEN OTHERS =>                             -- safe value, just set output as zero
                  outFlag <= "0000";
          END CASE;
      END PROCESS;

      -- This process updates the calculated result to the output
      PROCESS (outFlag)
      BEGIN
          output <= outFlag;                     -- set the output to the ALU's result
          IF (regZ = '1') THEN                   -- check if registerZ is set
              IF outFlag = "0000" THEN           --          if it is and the output is zero then
                  zout <= '1';                   --                  set the z flag on
              ELSE                                           --          if output is nonzero then
                  zout <= '0';                   --                  set the z flag off
              END IF;                                    --    if registerZ is not set then ignore the z flag
          END IF;
      END PROCESS;
```

```
        -- Whenever an operation is performed then the z flag does not need to be updated anymore
        PROCESS (fSelect)
        BEGIN
            IF regZ = '1' THEN              -- If registerZ was set when a function was performed then
                clear <= not clear;         --          clear the value of registerZ after the function
            END IF;                                 -- otherwise disregard registerZ
        END PROCESS;


END Behavioral;
```

A.1.2 alu.vhd

```
--------------------------------------------------------------------------------
-- Lab 3                                                                      --
-- ALU                                                                        --
-- Steve Comer                                                                --
-- Derek Tsui                                                                 --
-- Kevin Vece                                                                 --
-- Updated 14 Oct 2012                                                        --
--------------------------------------------------------------------------------

-- Import the necessary libraries
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

--------------------------------------------------------------------------------
-- Create the ALU entity which loads values and does calculations based on those values
ENTITY alu IS

PORT(A          : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);         -- The A input, or accumulator
     B          : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);   -- The B input
     loadA      : IN  STD_LOGIC;                            -- If a clock signal is given and loadA
                                                            --      is set then the ALU with load the
                                                            --      input value from A into registerA
     loadB      : IN  STD_LOGIC;                            -- If a clock signal is given and loadB
                                                            --      is set then the ALU with load the
                                                            --      input value from B into registerB
     loadZ      : IN  STD_LOGIC;                            -- If a clock signal is given and loadZ
                                                            --      is set then the ALU with load the
                                                            --      Z flag is the output is zero
     fSelect    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);         -- The function select input is defined
                                                            --      as follows:
                                                            --      0: ADD (registerA + registerB)
                                                            --      1: SUBTRACT (registerA - registerB)
                                                            --      2: AND (registerA & registerB)
                                                            --      3: OR (registerA | registerB)
                                                            --      4: NOT A (~registerA)
                                                            --      5: A (registerA)
                                                            --      6: NOT B (~registerB)
                                                            --      7: B (registerB)
     clk        : IN  STD_LOGIC;                            -- The clock input used to synchronize
                                                            --      loads and operations
     zOut       : OUT STD_LOGIC := '0';                     -- The Z flag, shows if output is zero
     output     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)          -- The 4 bit result from the desired
                                                            --      operation
);

END alu;

--------------------------------------------------------------------------------
ARCHITECTURE Behavioral OF alu IS

    -- This is a simple four bit register which updates is value if a clock input is given
    --      while load = '1'
    COMPONENT reg
      PORT(A        : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);     -- the register's input
           clk      : IN  STD_LOGIC;                        -- an input clock signal
           load     : IN  STD_LOGIC;                        -- the load check
           B        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)      -- the value of the register
      );
    END COMPONENT;

    -- This is a one bit register with a clock and clear signal. If a clock input is given
    --      the register will load the input. If the clear signal changes then the register
    --      will reset the register back to '0'
    COMPONENT bitreg IS
      PORT(A        : IN  STD_LOGIC;                        -- the register's input
           clk      : IN  STD_LOGIC;                        -- an input clock signal
```

```
            clear       : IN  STD_LOGIC;                    -- the input reset signal
            B           : OUT STD_LOGIC                     -- the value of the register
        );
    END COMPONENT;

    -- Create wires for processes to communicate with other
    SIGNAL regA    : STD_LOGIC_VECTOR(3 DOWNTO 0);-- holds the value of registerA
    SIGNAL regB    : STD_LOGIC_VECTOR(3 DOWNTO 0);-- holds the value of registerB
    SIGNAL regZ  : STD_LOGIC;                              -- holds the value of registerZ
    SIGNAL outFlag : STD_LOGIC_VECTOR(3 DOWNTO 0);      -- holds the result of the ALU operation
    SIGNAL clear  : STD_LOGIC := '0';                   -- used to clear registerZ

-------------------------------------------------------------------------------------------
BEGIN
    -- Create the three registers to store A, B, and Z
    registerA : reg PORT MAP(A, clk, loadA, regA);
    registerB : reg PORT MAP(B, clk, loadB, regB);
    registerZ : bitreg PORT MAP(loadZ, clk, clear, regZ);

    -- This process calculates the desired ALU operation using the stored values in
    --          registerA and registerB
    PROCESS(fSelect, regA, regB)
    BEGIN
        CASE fSelect IS
            WHEN "000" =>                           -- ADD (registerA + registerB)

                outFlag <= regA + regB;
            WHEN "001" =>                           -- SUBTRACT (registerA - registerB)

                outFlag <= regA - regB;
            WHEN "010" =>                           -- AND (registerA & registerB)
                outFlag <= regA and regB;
            WHEN "011" =>                           -- OR (registerA | registerB)
                outFlag <= regA or regB;
            WHEN "100" =>                           -- NOT A (~registerA)
                outFlag <= not regA;
            WHEN "101" =>                           -- A (registerA)
                outFlag <= regA;
            WHEN "110" =>                           -- NOT B (~registerB)
                outFlag <= not regB;
            WHEN "111" =>                           -- B (registerB)
                outFlag <= regB;
            WHEN OTHERS =>                              -- safe value, just set output as zero
                outFlag <= "0000";
        END CASE;
    END PROCESS;

    -- This process updates the calculated result to the output
    PROCESS (outFlag)
    BEGIN
        output <= outFlag;                      -- set the output to the ALU's result
        IF (regZ = '1') THEN                    -- check if registerZ is set
            IF outFlag = "0000" THEN            --          if it is and the output is zero then
                zout <= '1';                    --                  set the z flag on
            ELSE                                --          if output is nonzero then
                zout <= '0';                    --                  set the z flag off
            END IF;                                 --  if registerZ is not set then ignore the z flag
        END IF;
    END PROCESS;

    -- Whenever an operation is performed then the z flag does not need to be updated anymore
    PROCESS (fSelect)
    BEGIN
        IF regZ = '1' THEN          -- If registerZ was set when a function was performed then
            clear <= not clear;     --          clear the value of registerZ after the function
        END IF;                         -- otherwise disregard registerZ
    END PROCESS;


END Behavioral;
```

A.1.3 reg.vhd

```
--------------------------------------------------------------------------------
-- Lab 3                                                                      --
-- reg                                                                        --
-- Steve Comer                                                                --
-- Derek Tsui                                                                 --
-- Kevin Vece                                                                 --
-- Updated 14 Oct 2012                                                        --
--    Implements a four bit synchronous register with a load bit.             --
--------------------------------------------------------------------------------

-- Import the necessary libraries.
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

--------------------------------------------------------------------------------

-- Create an Entity that acts as a four bit register.
ENTITY reg IS

-- Input and output for the Register
PORT(A              : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);     -- The input value for the register
     clk            : IN  STD_LOGIC;                        -- A clock input used for a synchronous load
     load           : IN  STD_LOGIC;                        -- Load bit, the register will only load
                                                                --    the value in A if this bit is set
     B              : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)      -- The output, the value of the register
);

END reg;

--------------------------------------------------------------------------------


ARCHITECTURE Behavioral OF reg IS

BEGIN

    -- This is the synchronous load command. When the clock is high and load is set, then the value
    --        of the register is updated
    PROCESS (clk)
    BEGIN
        IF (rising_edge(clk) and load = '1') THEN       -- If the clock is on its rising edge and load is set,
            B <= A;                                     --        then store the value of A into B
        END IF;
    END PROCESS;

END Behavioral;
```

A.1.4 reg1.vhd

```
----------------------------------------------------------------------------------------------
-- Lab 3                                                                                    --
-- reg1                                                                                     --
-- Steve Comer                                                                              --
-- Derek Tsui                                                                               --
-- Kevin Vece                                                                               --
-- Updated 14 Oct 2012                                                                      --
--    Implements a one bit synchronous register with a load bit and a clear input.   --
----------------------------------------------------------------------------------------------

-- Import the necessary libraries.
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

----------------------------------------------------------------------------------------------

-- Create and Entity which acts as a one bit register.
ENTITY bitreg IS

PORT(A              : IN  STD_LOGIC;                    -- The input value for the register
     clk            : IN  STD_LOGIC;                    -- Clock input used for synchrnous load
     clear          : IN  STD_LOGIC;                    -- Clears the value in the register whenever this
                                                        --          value changes
     B              : OUT STD_LOGIC := '0'      -- The output value of the register
);
END bitreg;

----------------------------------------------------------------------------------------------


ARCHITECTURE Behavioral OF bitreg IS

    -- Declare a signal used to store what value that the clear input is.
    SIGNAL reset : STD_LOGIC := '0';

BEGIN

    -- This process activates whenever the clock or clear changes
    PROCESS (clk, clear)
    BEGIN

        IF clear = not reset THEN       -- If the value of clear changed
            B <= '0';                   --    then set the output to zero
            reset <= clear;             --          and update the new value of clear
        ELSIF (clk = '1') THEN          -- Otherwise, if the clock is high
            B <= A;                     --          update the value of the register to A
        END IF;

    END PROCESS;

END Behavioral;
```

A.1.5 display_driver.vhd

```
-- Lab 3
-- Hex Display Driver
-- Steve Comer
-- Stuart Larsen
-- Team A-Shred
-- Updated 14 Oct 2012
--    HexDriver takes a binary number and outputs the equivalent signal for a 7-seg display.

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY HexDriver IS

PORT(numberToDisplay  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
      sevenSegmentOut     : OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
END HexDriver;

ARCHITECTURE Behavioral OF HexDriver IS
    BEGIN

        HexDriver : PROCESS(numberToDisplay)
            BEGIN
                CASE numberToDisplay IS                                                 -- Disp
                    WHEN "0000" => sevenSegmentOut <= "1000000";        --    0
                    WHEN "0001" => sevenSegmentOut <= "1111001";        --    1
                    WHEN "0010" => sevenSegmentOut <= "0100100";        -- 2
                    WHEN "0011" => sevenSegmentOut <= "0110000";        -- 3
                    WHEN "0100" => sevenSegmentOut <= "0011001";        -- 4
                    WHEN "0101" => sevenSegmentOut <= "0010010";        -- 5
                    WHEN "0110" => sevenSegmentOut <= "0000010";        -- 6
                    WHEN "0111" => sevenSegmentOut <= "1111000";        -- 7
                    WHEN "1000" => sevenSegmentOut <= "0000000";        --    8
                    WHEN "1001" => sevenSegmentOut <= "0010000";        -- 9
                    WHEN "1010" => sevenSegmentOut <= "0001000";        -- A
                    WHEN "1011" => sevenSegmentOut <= "0000011";        -- b
                    WHEN "1100" => sevenSegmentOut <= "1000110";        -- C
                    WHEN "1101" => sevenSegmentOut <= "0100001";        --    d
                    WHEN "1110" => sevenSegmentOut <= "0000110";        -- E
                    WHEN "1111" => sevenSegmentOut <= "0001110";        -- F
                    WHEN OTHERS => sevenSegmentOut <= "1111111";          --    null
                END CASE;

            END PROCESS HexDriver;

    END Behavioral;
```

A.2 PC Code

A.2.1 main_pc.vhd

```
-- Lab 3
-- Master File (Uses PC & Display Driver)
-- Steve Comer
-- Derek Tsui
-- Kevin Vece
-- Updated 14 Oct 2012
--		Combines both PC and Display Driver to create a Program Counter

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY main_pc IS

PORT(input	: IN  STD_LOGIC_VECTOR(7 DOWNTO 0);			-- input value on the PC
		clearPC	: IN  STD_LOGIC;					-- clear PC register
		incPC		: IN  STD_LOGIC;					-- increment PC register
		loadPC		: IN  STD_LOGIC;					-- load PC register
		clearX		: IN  STD_LOGIC;					-- clear X register
		incX		: IN  STD_LOGIC;					-- increment X register
		outSel		: IN  STD_LOGIC_VECTOR(1 DOWNTO 0);		-- select which output to display
		clk		: IN  STD_LOGIC;					-- clock input
		clkOut		: OUT STD_LOGIC;					-- clock output, used for testing
												-- SEVEN SEGMENT DISPLAYS:
		sevenSegmentOut0	: OUT STD_LOGIC_VECTOR(6 DOWNTO 0); -- first digit on input
		sevenSegmentOut1	: OUT STD_LOGIC_VECTOR(6 DOWNTO 0); -- second digit on input
		sevenSegmentOut2	: OUT STD_LOGIC_VECTOR(6 DOWNTO 0); -- first digit on output
		sevenSegmentOut3	: OUT STD_LOGIC_VECTOR(6 DOWNTO 0)  -- second digit on input
);

END main_pc;

ARCHITECTURE Behavioral OF main_pc IS

	-- Declare the Hex driver component to convert hex values into a format for the
	--	seven segment displays on the board.
	COMPONENT HexDriver
	  PORT(numberToDisplay  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
	       sevenSegmentOut	: OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
	END COMPONENT;

	-- Declare the Program Counter component to do the grunt of the work.
	COMPONENT pc IS
	  PORT(input	: IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
	       clearPC	: IN  STD_LOGIC;
	       incPC	: IN  STD_LOGIC;
	       loadPC	: IN  STD_LOGIC;
	       clearX	: IN  STD_LOGIC;
	       incX	: IN  STD_LOGIC;
	       outSel	: IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
	       clk	: IN  STD_LOGIC;
	       output	: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
	END COMPONENT;

	-- Create wires
	SIGNAL output : STD_LOGIC_VECTOR(7 DOWNTO 0); -- wire used to show output on the PC
	SIGNAL clkNeg : STD_LOGIC;				-- Reverse of the clock input

BEGIN

	-- Create the program counter and link it to the seven segment displays.
	pc0 : pc PORT MAP(input, clearPC, incPC, loadPC, clearX, incX, outSel, clkNeg, output);
	hexDriver0 : HexDriver PORT MAP(input(7 DOWNTO 4), sevenSegmentOut0);
	hexDriver1 : HexDriver PORT MAP(input(3 DOWNTO 0), sevenSegmentOut1);
	hexDriver2 : HexDriver PORT MAP(output(7 DOWNTO 4), sevenSegmentOut2);
```

```
        hexDriver3 : HexDriver PORT MAP(output(3 DOWNTO 0), sevenSegmentOut3);

        -- Reverse the clock input so that button pressed = clock high
        PROCESS(clk)
        BEGIN
           clkNeg <= NOT clk;
           clkOut <= NOT clk;
        END PROCESS;

END Behavioral;
```

A.2.2 pc.vhd

```
-- Lab 3
-- ALU
-- Steve Comer
-- Derek Tsui
-- Kevin Vece
-- Updated 14 Oct 2012

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY pc IS

PORT(input      : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);      -- input value
     clearPC    : IN  STD_LOGIC;                         -- clear input for counter
     incPC      : IN  STD_LOGIC;                         -- increment input for counter
     loadPC     : IN  STD_LOGIC;                         -- load input for counter
     clearX     : IN  STD_LOGIC;                         -- clear input for x
     incX       : IN  STD_LOGIC;                         -- increment input for x
     outSel     : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);      -- output select value
     clk        : IN STD_LOGIC;                          -- clock input for synchronous load
     output     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));     -- output value of the PC

END pc;

ARCHITECTURE Behavioral OF pc IS

    -- Create two registers for both the PC and X
    SIGNAL pcREG : STD_LOGIC_VECTOR(7 DOWNTO 0) REGISTER;
    SIGNAL xREG : STD_LOGIC_VECTOR(7 DOWNTO 0) REGISTER;

BEGIN

    -- Change values on a high clock edge
    PROCESS(clk)
    BEGIN

        IF (RISING_EDGE(clk)) THEN              -- check fro rising edge of clock

            IF clearPC = '1' THEN               -- update the PC, priority is:
                pcREG <= "00000000";            --    clear, load, incrememnt
            ELSIF loadPC = '1' THEN
                pcREG <= input;
            ELSIF incPC = '1' THEN
                pcREG <= pcREG + '1';
            END IF;

            IF clearX = '1' THEN                -- update the X register, priority is:
                xREG <= "00000000";             --    clear, increment
            ELSIF incX = '1' THEN
                xREG <= xREG + '1';
            END IF;

        END IF;

    END PROCESS;

    -- Update the output
```

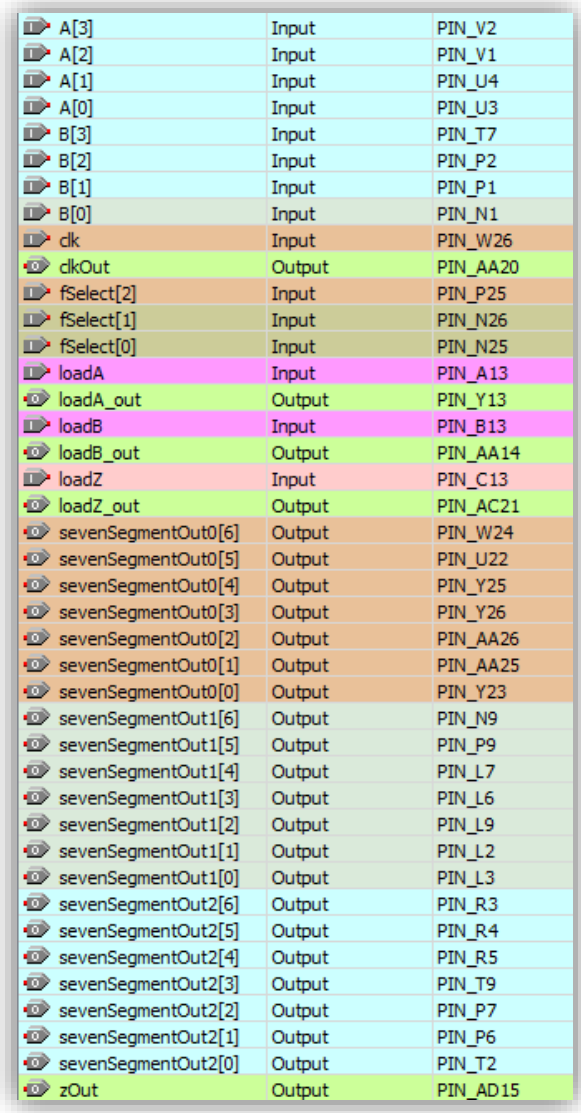
```
    PROCESS(outSel)
    BEGIN
        CASE (outSel) IS
            WHEN "00" =>                        -- 00: PC
                output <= pcREG;
            WHEN "01" =>                        -- 01: X + input
                output <= xREG + input;
            WHEN "10" =>                        -- 10: input
                output <= input;
            WHEN "11" =>                        -- 11: X
                output <= xREG;
            WHEN OTHERS => NULL;                    -- safe case, do nothing
        END CASE;
    END PROCESS;

END Behavioral;
```

A.2.3 display_driver.vhd

```
-- Lab 3
-- Hex Display Driver
-- Steve Comer
-- Stuart Larsen
-- Team A-Shred
-- Updated 14 Oct 2012
--    HexDriver takes a binary number and outputs the equivalent signal for a 7-seg display.

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY HexDriver IS

PORT(numberToDisplay  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
      sevenSegmentOut     : OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
END HexDriver;

ARCHITECTURE Behavioral OF HexDriver IS
    BEGIN

        HexDriver : PROCESS(numberToDisplay)
            BEGIN
                CASE numberToDisplay IS                                                 -- Disp
                    WHEN "0000" => sevenSegmentOut <= "1000000";        --    0
                    WHEN "0001" => sevenSegmentOut <= "1111001";        --    1
                    WHEN "0010" => sevenSegmentOut <= "0100100";        -- 2
                    WHEN "0011" => sevenSegmentOut <= "0110000";        -- 3
                    WHEN "0100" => sevenSegmentOut <= "0011001";        -- 4
                    WHEN "0101" => sevenSegmentOut <= "0010010";        -- 5
                    WHEN "0110" => sevenSegmentOut <= "0000010";        -- 6
                    WHEN "0111" => sevenSegmentOut <= "1111000";        -- 7
                    WHEN "1000" => sevenSegmentOut <= "0000000";        --    8
                    WHEN "1001" => sevenSegmentOut <= "0010000";        -- 9
                    WHEN "1010" => sevenSegmentOut <= "0001000";        -- A
                    WHEN "1011" => sevenSegmentOut <= "0000011";        -- b
                    WHEN "1100" => sevenSegmentOut <= "1000110";        -- C
                    WHEN "1101" => sevenSegmentOut <= "0100001";        --    d
                    WHEN "1110" => sevenSegmentOut <= "0000110";        -- E
                    WHEN "1111" => sevenSegmentOut <= "0001110";        -- F
                    WHEN OTHERS => sevenSegmentOut <= "1111111";          --    null
                END CASE;

            END PROCESS HexDriver;

    END Behavioral;
```

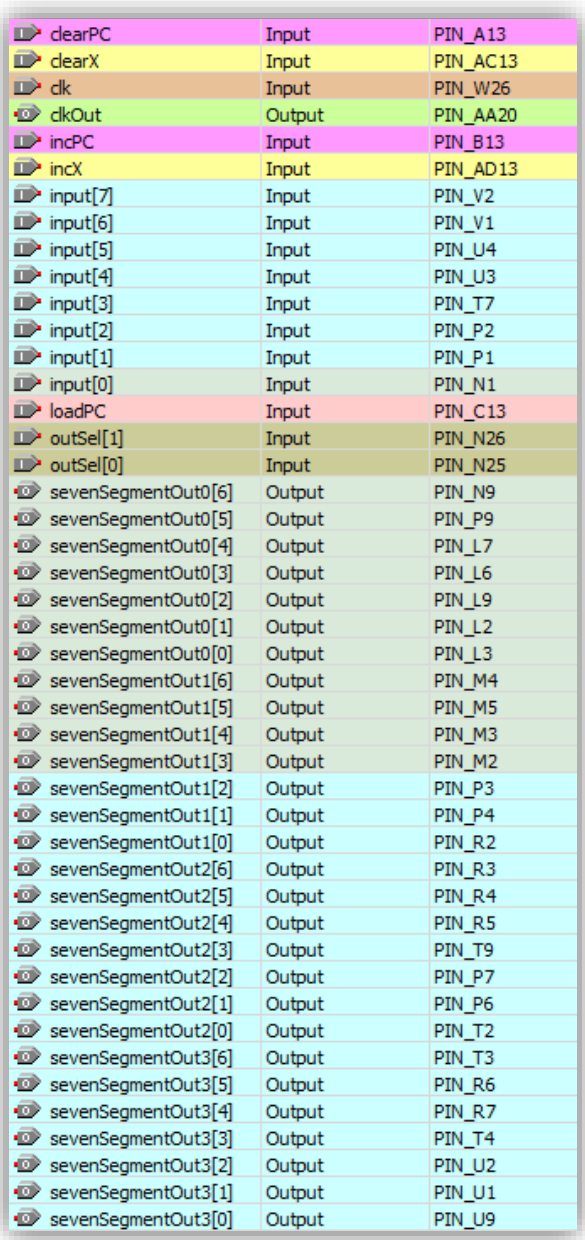
B. Appendix B – Quartus II Pinouts

B.1 ALU Pinout



A[3]	Input	PIN_V2
A[2]	Input	PIN_V1
A[1]	Input	PIN_U4
A[0]	Input	PIN_U3
B[3]	Input	PIN_T7
B[2]	Input	PIN_P2
B[1]	Input	PIN_P1
B[0]	Input	PIN_N1
clk	Input	PIN_W26
clkOut	Output	PIN_AA20
fSelect[2]	Input	PIN_P25
fSelect[1]	Input	PIN_N26
fSelect[0]	Input	PIN_N25
loadA	Input	PIN_A13
loadA_out	Output	PIN_Y13
loadB	Input	PIN_B13
loadB_out	Output	PIN_AA14
loadZ	Input	PIN_C13
loadZ_out	Output	PIN_AC21
sevenSegmentOut0[6]	Output	PIN_W24
sevenSegmentOut0[5]	Output	PIN_U22
sevenSegmentOut0[4]	Output	PIN_Y25
sevenSegmentOut0[3]	Output	PIN_Y26
sevenSegmentOut0[2]	Output	PIN_AA26
sevenSegmentOut0[1]	Output	PIN_AA25
sevenSegmentOut0[0]	Output	PIN_Y23
sevenSegmentOut1[6]	Output	PIN_N9
sevenSegmentOut1[5]	Output	PIN_P9
sevenSegmentOut1[4]	Output	PIN_L7
sevenSegmentOut1[3]	Output	PIN_L6
sevenSegmentOut1[2]	Output	PIN_L9
sevenSegmentOut1[1]	Output	PIN_L2
sevenSegmentOut1[0]	Output	PIN_L3
sevenSegmentOut2[6]	Output	PIN_R3
sevenSegmentOut2[5]	Output	PIN_R4
sevenSegmentOut2[4]	Output	PIN_R5
sevenSegmentOut2[3]	Output	PIN_T9
sevenSegmentOut2[2]	Output	PIN_P7
sevenSegmentOut2[1]	Output	PIN_P6
sevenSegmentOut2[0]	Output	PIN_T2
zOut	Output	PIN_AD15

B.2 PC Pinout



clearPC	Input	PIN_A13
clearX	Input	PIN_AC13
clk	Input	PIN_W26
clkOut	Output	PIN_AA20
incPC	Input	PIN_B13
incX	Input	PIN_AD13
input[7]	Input	PIN_V2
input[6]	Input	PIN_V1
input[5]	Input	PIN_U4
input[4]	Input	PIN_U3
input[3]	Input	PIN_T7
input[2]	Input	PIN_P2
input[1]	Input	PIN_P1
input[0]	Input	PIN_N1
loadPC	Input	PIN_C13
outSel[1]	Input	PIN_N26
outSel[0]	Input	PIN_N25
sevenSegmentOut0[6]	Output	PIN_N9
sevenSegmentOut0[5]	Output	PIN_P9
sevenSegmentOut0[4]	Output	PIN_L7
sevenSegmentOut0[3]	Output	PIN_L6
sevenSegmentOut0[2]	Output	PIN_L9
sevenSegmentOut0[1]	Output	PIN_L2
sevenSegmentOut0[0]	Output	PIN_L3
sevenSegmentOut1[6]	Output	PIN_M4
sevenSegmentOut1[5]	Output	PIN_M5
sevenSegmentOut1[4]	Output	PIN_M3
sevenSegmentOut1[3]	Output	PIN_M2
sevenSegmentOut1[2]	Output	PIN_P3
sevenSegmentOut1[1]	Output	PIN_P4
sevenSegmentOut1[0]	Output	PIN_R2
sevenSegmentOut2[6]	Output	PIN_R3
sevenSegmentOut2[5]	Output	PIN_R4
sevenSegmentOut2[4]	Output	PIN_R5
sevenSegmentOut2[3]	Output	PIN_T9
sevenSegmentOut2[2]	Output	PIN_P7
sevenSegmentOut2[1]	Output	PIN_P6
sevenSegmentOut2[0]	Output	PIN_T2
sevenSegmentOut3[6]	Output	PIN_T3
sevenSegmentOut3[5]	Output	PIN_R6
sevenSegmentOut3[4]	Output	PIN_R7
sevenSegmentOut3[3]	Output	PIN_T4
sevenSegmentOut3[2]	Output	PIN_U2
sevenSegmentOut3[1]	Output	PIN_U1
sevenSegmentOut3[0]	Output	PIN_U9