COMSATS University Islamabad (CUI), Lahore Campus
Department of Electrical and Computer Engineering

CPE343 – COMPUTER ORGANIZATION & ARCHITECTURE

Lab Manual for FALL 2022

Lab Resource Person

Engr. Moazzam Ali Sahi

Engr. Wajeeha Khan

Theory Resource Person

Engr. Moazzam Ali Sahi

Dr Naeem Awais

Name: _____ **Registration Number:** _____

Program: _____ **Batch:** _____

Semester _____

Revision History

S.No.	Update	Date (DD/MM/YYYY)	Performed by
1	Lab Manual Preparation	22/07/19	Moazzam Ali Sahi, Usman Rafique
2	Lab Manual Review	14/08/2022	Moazzam Ali Sahi
3	Lab Manual- Major Updates	20/12/2022	Moazzam Ali Sahi, Wajeeha Khan

Preface

Computer architecture is the science and art of selecting and interconnecting hardware components and designing the hardware/software interface to create a computer that meets functional, performance, and other specific goals. This lab introduces the basic hardware structure of a modern programmable computer, including the basic laws underlying performance evaluation. We will learn, for example, how to design the control and data path hardware for processor, how to make machine instructions execute simultaneously, and how to design fast memory and storage systems. The principles presented in the laboratory through the design, simulation and implementation of a MIPS-like processor in HDL (Verilog/VHDL).

Books

Text Book

1. Computer Organization and design: The hardware/software interface, by John L. Hennessy, David A. Patterson (5th edition) (H&P5)

Reference Books

1. Modern Processor Design; Fundamentals of Superscalar Processors, by John Paul Shen, Mikko Lapisti**.**
2. Computer organization and architecture designing for performance by William Stallings (8th edition)
3. Computer system organization and architecture by John D. Capinelli.
4. Computer organization and design: The hardware/software interface by David A Patterson & John. L. Hennessy (3rd edition)

Learning Outcomes

Theory CLOs:

After successful completion of this module, you will be able to:

CLO 1: **Describe** basic computer organization including interface between software and hardware, MIPS assembly, memory hierarchy, and I/O mechanisms using hardware/software abstraction layers based on the design principles of modern computer systems. (PLO1-C1)

CLO 2: **Explain** the principles and techniques used in implementing non-pipelined and pipelined implementations using various functional units to realize implementation of some instruction set like MIPS. (PLO1-C2)

CLO 3: **Solve** basic computer arithmetic using hardware and write assembly code for a high-level language code using its instruction set to lay the foundations of a simplified processor design. (PLO2-C3)

CLO 4: **Compute** the performance of computer programs, modern processors, cache systems using terms related to various hardware/software constituents to highlight tradeoffs between different design choices. (PLO3-C3)

Lab CLOs:

Lab CLO 1: **Follow** the procedure to assemble different parts of MIPS 32-bit microprocessor in HDL and reproduce its response using a software tool and hardware platform. (PLO5-P3)

CLOs – PLOs Mapping

PLO / CLO	PLO1	PLO2	PLO3	PLO5	Cognitive Domain	Affective Domain	Psychomotor Domain
CLO1	X				C1		
CLO2	X				C2		
CLO3		X			C3		
CLO4			X		C3		
Lab CLO1				X			P3

Lab CLOs – Lab Experiment Mapping

Lab / CLO	Lab 1	Lab 2	Lab 3	Lab 4	Lab 5	Lab 6	Lab 7	Lab 8	Lab 9	Lab 10	Lab 11	Lab 12
Lab CLO1	P2	P2	P2	P2	P3	P2	P3	P3	P3	P3	P3	P3

Grading Policy

Lab Assignments: i. Lab Assignment 1 Marks (Lab marks from Labs 1-3) ii. Lab Assignment 2 Marks (Lab marks from Labs 4-6) iii. Lab Assignment 3 Marks (Lab marks from Labs 7-9) iv. Lab Assignment 4 Marks (Lab marks from Labs 10-12)	25%
Lab Mid Term = 0.5*(Lab Mid Term exam) + 0.5*(average of lab evaluation of Lab 1-7)	25%
Lab Terminal = 0.5*(Lab Terminal exam) +0.375*(average of lab evaluation of Lab 7-12) + 0.125*(average of lab evaluation of Lab 1-6)	50%
Total (lab)	**100%**

List of equipment

Altera Cyclone IV E -DE2-115 FPGA
Altera Cyclone SoC V SE -DE1 FPGA

Software Resources

Quartus 13.1

Lab Instructions

- This lab activity comprises of three parts: Pre-lab, Lab Tasks and Viva session.
- The students should perform and demonstrate each lab task separately for step-wise evaluation.
- Only those tasks that are completed during the allocated lab time will be credited to the students.
- Students are however encouraged to practice on their own in spare time for enhancing their skills.

Safety Instructions

1. It is the duty of all concerned who use any electrical laboratory to take all reasonable steps to safeguard the health and safety of themselves and all other users and visitors.
2. Be sure that all equipment is properly working before using them for laboratory exercises. Any defective equipment must be reported immediately to the Lab. Instructors or Lab. Technical Staff.
3. Students are allowed to use only the equipment provided in the experiment manual.
4. Power supply terminals connected to any circuit are only energized with the presence of the Instructor or Lab. Staff.
5. Avoid any part of your body to be connected to the energized circuit and ground.
6. Switch off the equipment and disconnect the power supplies from the circuit before leaving the laboratory.
7. Observe cleanliness and proper laboratory housekeeping of the equipment and other related accessories.
8. Make sure that the last connection to be made in your circuit is the power supply and first thing to be disconnected is also the power supply.
9. Equipment should not be removed, transferred to any location without permission from the laboratory staffs.
10. Students are not allowed to use any equipment without proper orientation and actual hands on equipment operation.
11. Do not eat food, drink beverages, or chew gum in the laboratory.

Table of Contents

LAB # 1:
Explain VHDL and Altera Quartus tool with the design of combinational circuit

Objective

- How to program in hardware descriptive languages in general and VHDL in particular.
- How the software tool Quartus works and its interface
- How the FPGA design flow works from design specification to loading the bit file

Pre-Lab

Part 1 -Familiarize yourself with VHDL and Quartus:

Introduction to VHDL

VHDL is a hardware description language. It described the behavior of an electronic circuit or system, from which the physical circuit or system can then be attained (implemented). VHDL stands for VHSIC Hardware Description Language. VHSIC is itself an abbreviation for Very High Speed Integrated Circuits, an initiative funded by US DOD in the 1980s that led to the creation of VHDL. VHDL is intended for circuit synthesis as well as circuit simulation. However, though VHDL is fully simulated, not all constructs are synthesizable.

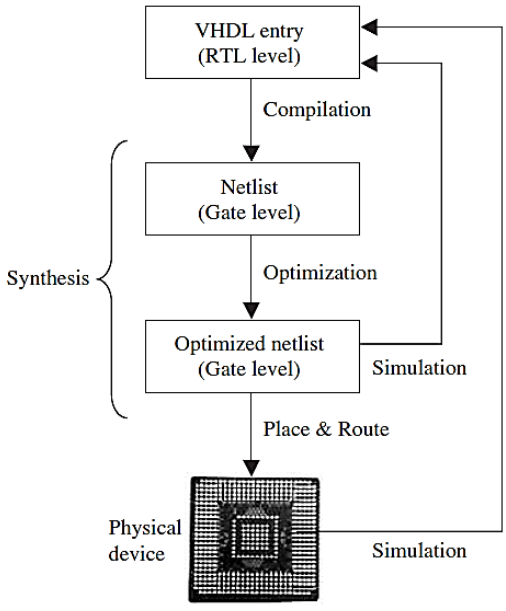


*Figure 1.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**-1 Summary of VHDL design flow*

A fundamental motivation to use VHDL (or its competitor, Verilog) is that VHDL is a standard, technology/vendor independent language, and is therefore portable and reusable. The two main immediate applications of VHDL are in the field of Programmable Logic Devices (including CPLDs and FPGAs) and ASICs. A final note regarding VHDL is that, contrary to regular computer programs which are sequential, its statements are inherently concurrent (parallel). In VHDL, only statements placed inside *a Process, Function or Procedure* are executed sequentially.

Fundamental VHDL Units:

A standalone piece of VHDL code is composed of at least three fundamental sections:

- **LIBRARY** declarations: Contains a list of all libraries to be used in the design. For example: *ieee, std, work, etc*.
- **ENTITY**: Specifies the I/O pins of the circuit.
- **ARCHITECTURE**: Contains the VHDL code proper, which describes how the circuit should behave (function)

A **LIBRARY** is a collection of commonly used pieces of code. Placing such pieces inside a library allows them to be reused or shared by other design.
The typical structure of a library is illustrated in figure 1.2. The code is usually written in the form of **FUNCTIONS**, **PROCEDURES**, or **COMPONENTS**, which are placed inside **PACKAGES**, and then compiled into the destination library.

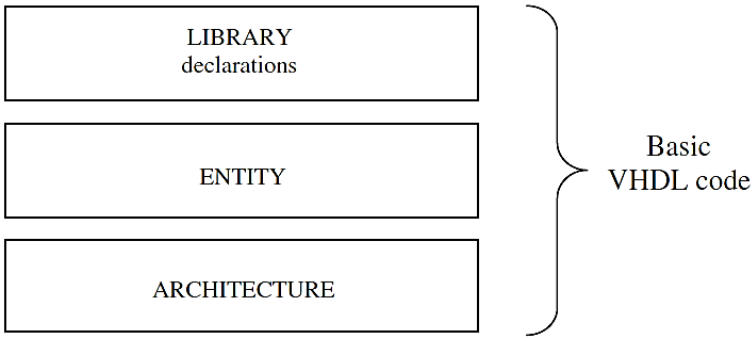


*Figure 1.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**-2 Fundamental sections of a basic VHDL code*

Entity declaration:

VHDL files are basically divided into two parts, the entity and the architecture. The entity is basically where the circuits (in & out) ports are defined. There is a multitude of I/O ports available, but this lab will only deal with the two most usual ones, the INput and OUTput ports. (Other types of ports are for example the INOUT and BUFFER ports.) The entity of the circuit in figure 1.3 should look something like below.

Please notice that comments in the code are made with a double-dash (--)

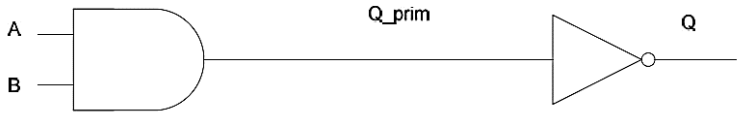


Figure 0 NAND gate using AND & NOT gate

Syntax:

```
entity entity_name is
        Port declaration;
 end entity_name;
```

For Example:

```
ENTITY nandgate IS
PORT(
        A: IN BIT;
        B: IN BIT;
        Q: OUT BIT                    -- Note! No ';' here!
);
END nandgate;
```

Now that we have defined the I/O interface to the rest of the world for the NAND gate, we should move on to the architecture of the circuit.

Architecture:

The entity told us nothing about how the circuit was implemented, this is taken care of by the architecture part of the VHDL code. The architecture of the NAND gate matching the entity above could then be written as something like this...

Syntax:

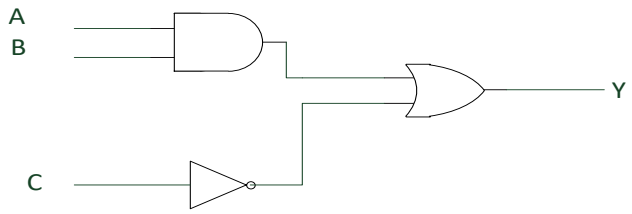
```
architecture architecture_name of entity_name
 architecture_declarative_part;
   begin
      Statements;
 end architecture_name;
```

For Example:

```
ARCHITECTURE struc  OF nandgate IS
        SIGNAL Q_prim: BIT;
BEGIN
        Q_prim <= A AND B; -- The AND-operation.
        Q <= NOT Q_prim; -- The inverter.
END struc;
```

In-Lab Tasks

Lab Task 1




```
Library IEEE;
Use IEEE.STD_LOGIC_1164.all;
-------------------------------------------------
Entity circuit is
 Port ( A, B , C: IN bit;       ( IEEE standard which defines a nine-value logic type)
        Y : OUT bit);
End circuit;
-------------------------------------------------
Architecture struct of circuit is
Begin
        Y<= (A AND B) OR (NOT C);
End struct;
```

Part 2 – Familiarize yourself with Quartus Tool

Introduction to Quartus ISE:

Altera Quartus II is design software produced by Altera. Quartus II enables analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Quartus includes an implementation of VHDL and Verilog for hardware description, visual editing of logic circuits, and vector waveform simulation.

Lab Task 2: Create a New Project

1. **Start the Quartus II software.**

   From the Windows Start Menu, select:

   All Programs → Other Apps →Altera → Quartus II 13.1 → Quartus II 13.1 (32-Bit)

2. **Start the New Project Wizard.**

   If the opening splash screen is displayed, select: **Create a New Project (New Project Wizard)**, otherwise from the Quartus II Menu Bar select: **File → New Project Wizard**.

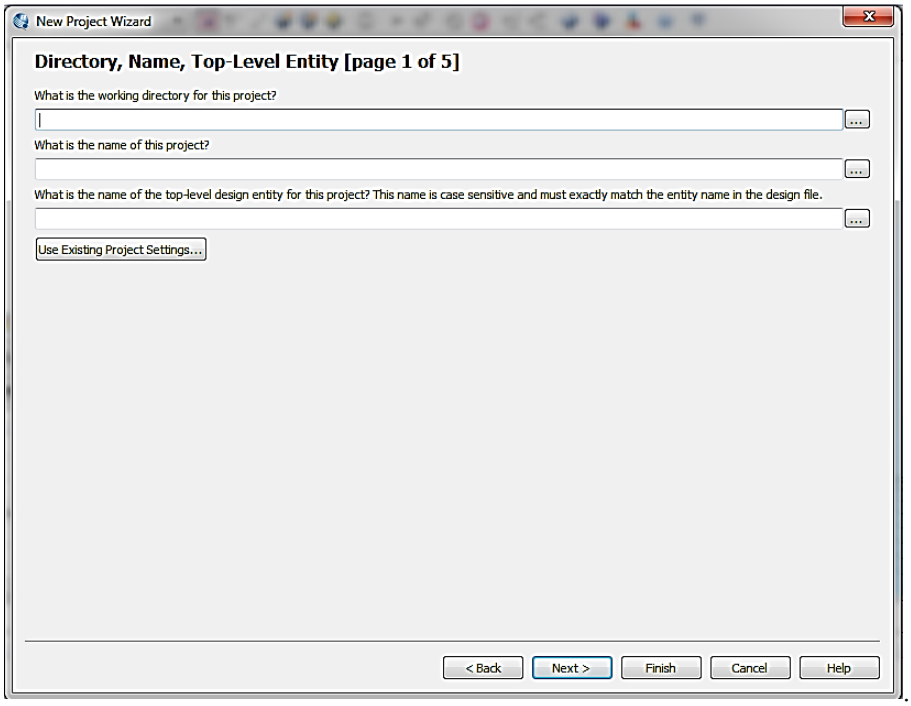
3. **Select the Working Directory and Project Name.**

Working Directory	H:\Altera_Training\Lab1
Project Name	Lab1
Top-Level Design Entity	Lab1

   Click **Next** to advance to page 2 of the New Project Wizard.

   Note: A window may pop up stating that the chosen working directory does not exist.

   *Click **Yes** to create it*



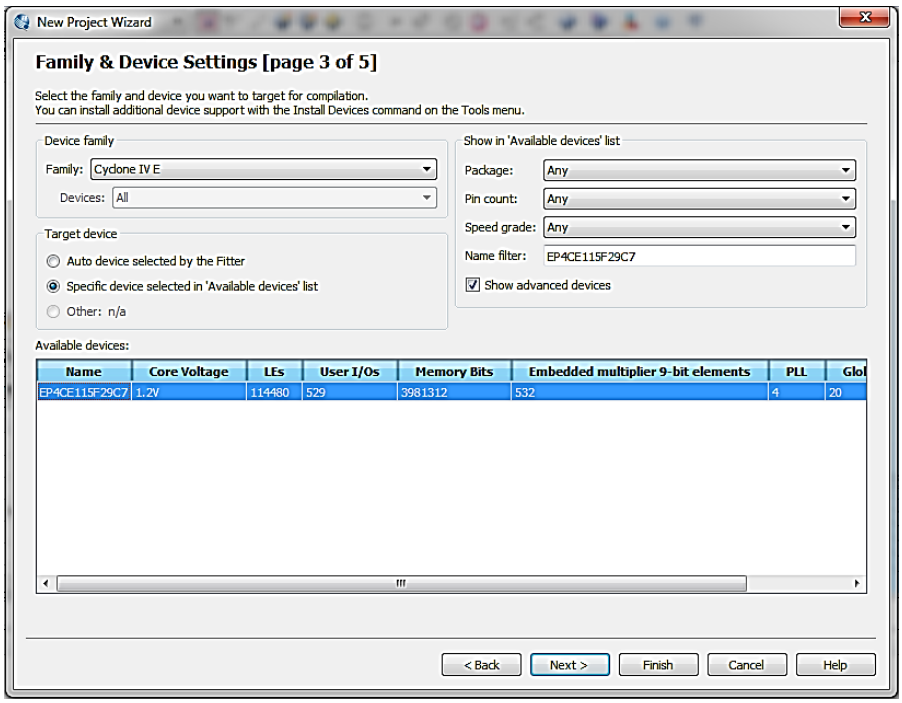
4. Click **Next** again as we will not be adding any preexisting design files at this time.

5. **Select the family and Device Settings.**

   From the pull-down menu labeled **Family**, select **Cyclone IV**.

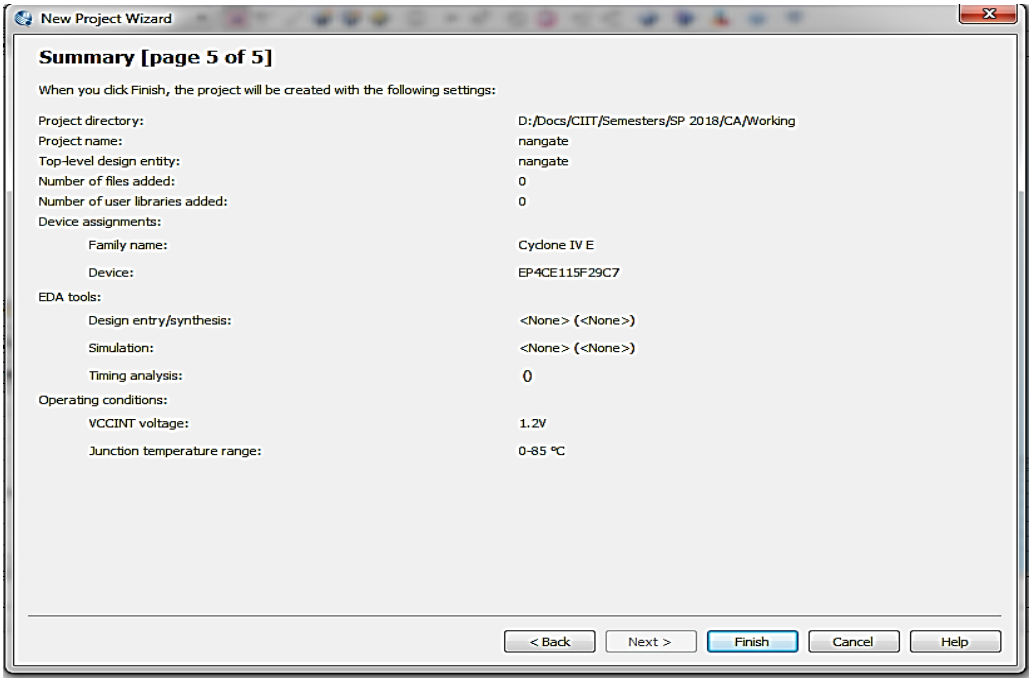
   In the list of available devices, select **EP4CE115F29C7**.

   Click **Next**.



6. Click **Next** again as we will not be using any third party EDA tools

7. Click **Finish** to complete the New Project Wizard

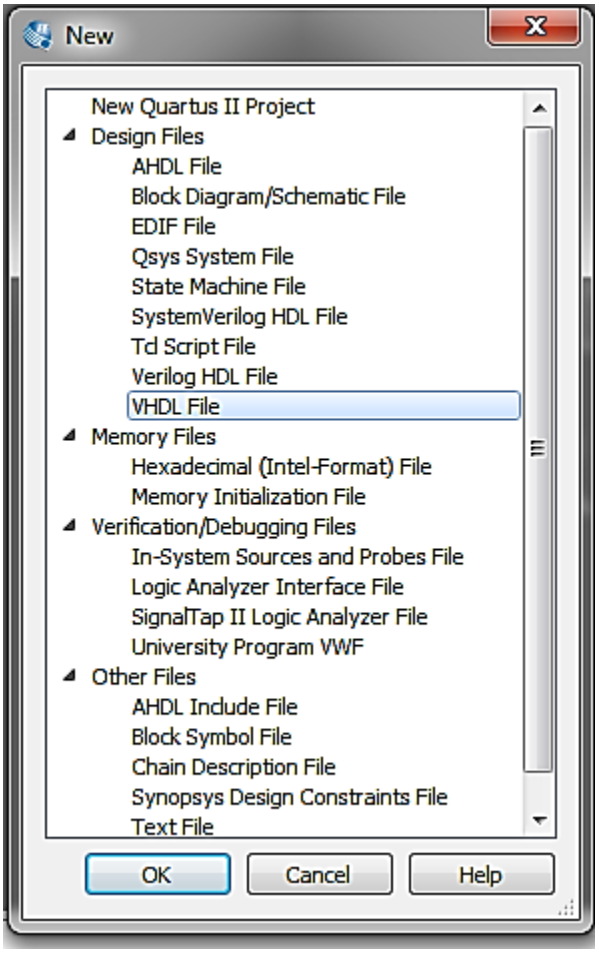


Task 2: Create, Add, and Compile Design Files

1. **Create a new Design File.**

   Select: **File→ New** from the Menu Bar.

Select: **VHDL File** from the **Design Files** list and click **OK**.



2. Copy and paste the following code into your new VHDL file, then save it by selecting **File →Save**.
   Name the file **nandgate** and click **Save** in the **Save As** dialog box.

```
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

entity nandgate is
port
        (
                A: in std_logic;
                B: in std_logic;
                cout : out std_logic
        );
end nandgate;

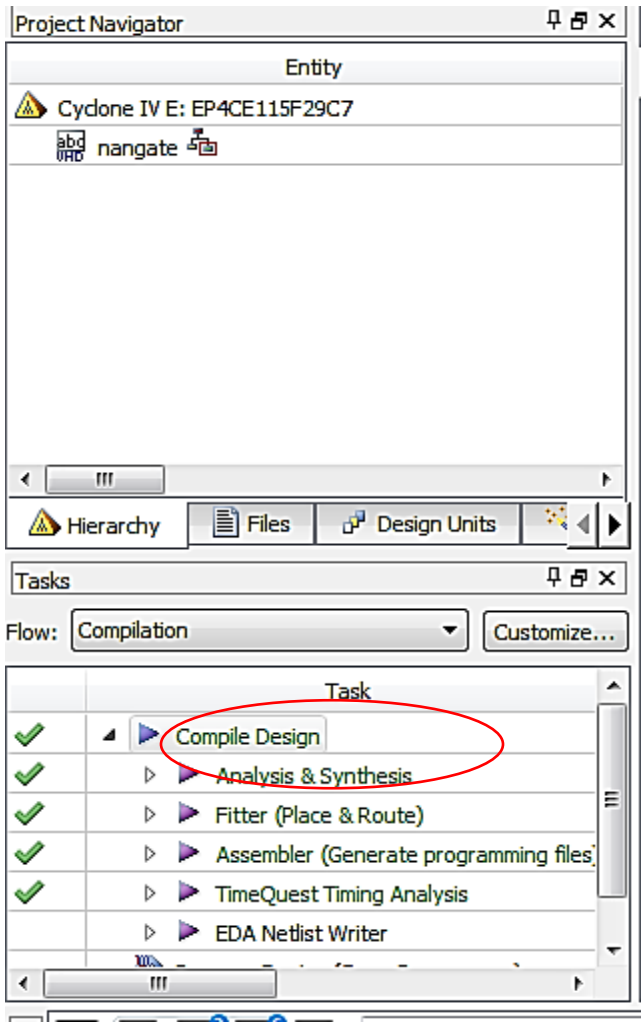
architecture struct of nandgate is
```

begin

cout<= not (A and B);

***end** struct;*

3. Now Click on **Compile Design** to look for errors in code

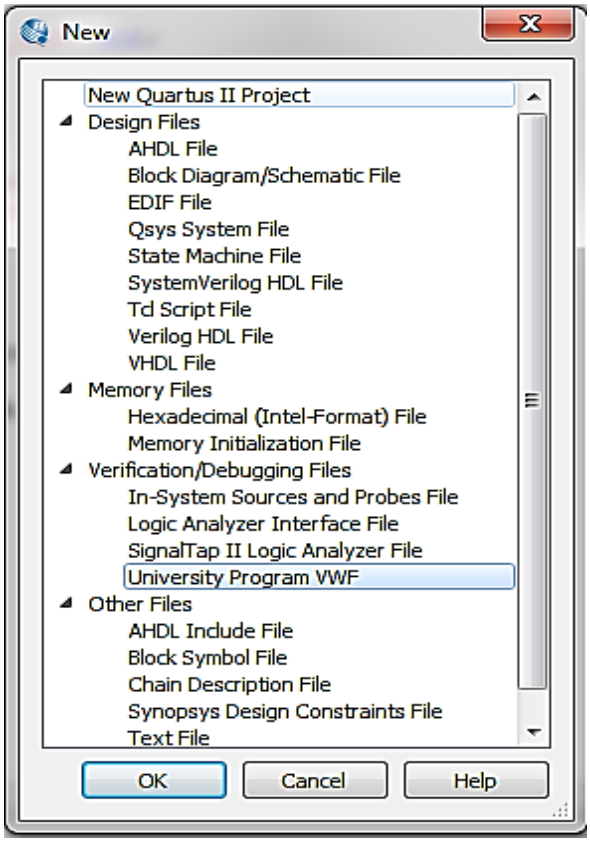


Task 3: Simulate Design using Quartus II

1. **Create a Vector Waveform File (vwf)**

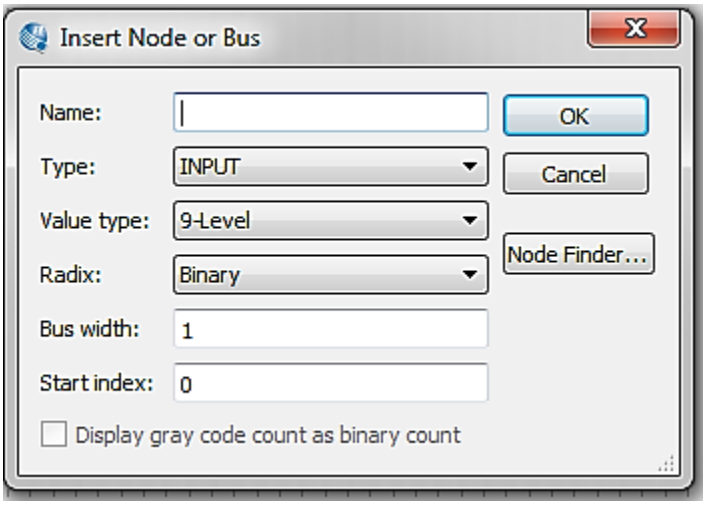
   Click the **New File** icon on the Menu Bar

   Select **Vector Waveform File** under the **Verification/Debugging Files** heading.



2. **Add Signals to the Waveform**

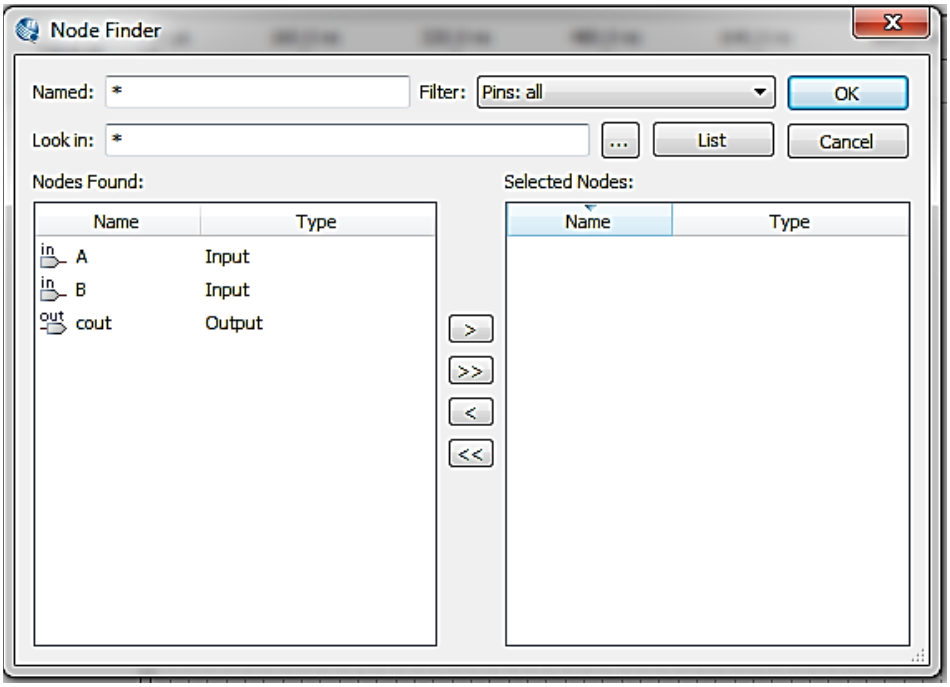
   Select **Edit → Insert → Insert Node or Bus…**to open the **Insert Node or Bus** window.



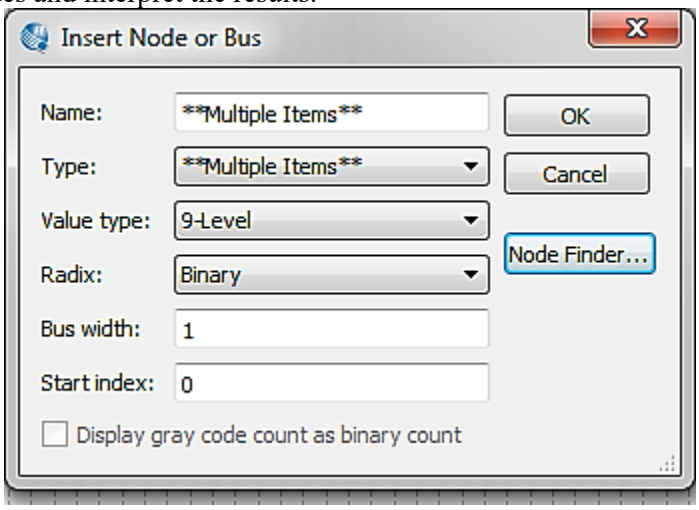
3. Click the **Node Finder…**button in the **Insert Node or Bus** window to open the **Node Finder** window.

From the **Filter** drop-down menu, select **Pins: all** and click the **List** button. This will display all the inputs and outputs to the circuit.

Highlight all the **Input Groups A, B**, and **Output Group** Sum in the **Nodes Found** window and click the right arrow  to select them. Then click **OK** to close the **Node Finder** window.

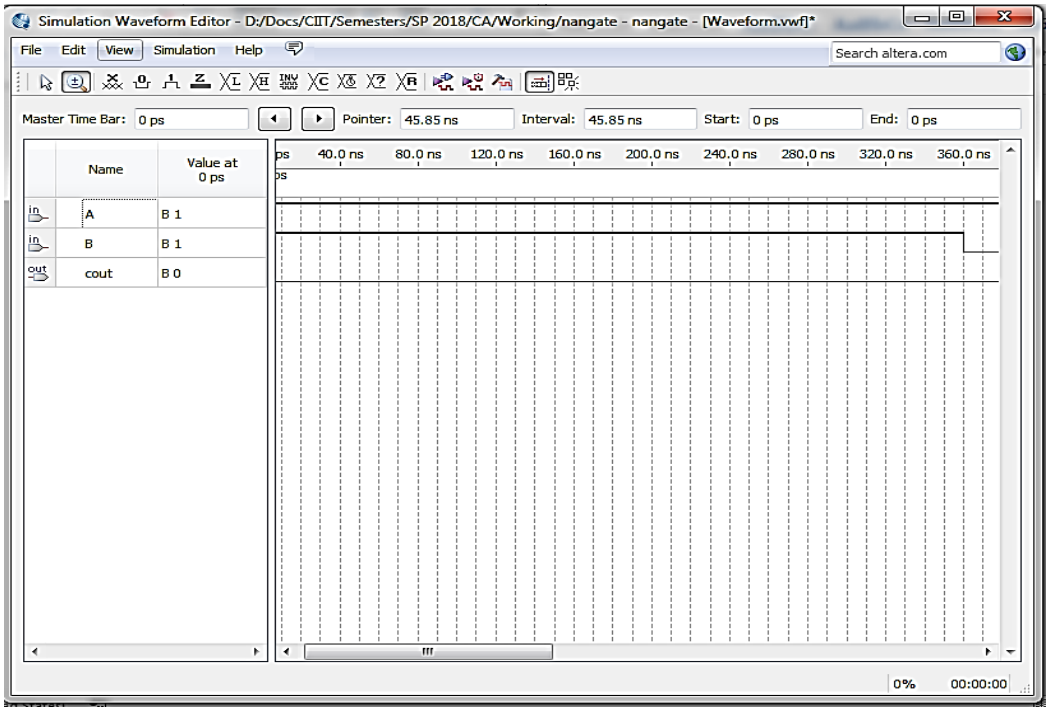


4. In the **Insert Node or Bus** window, change **Radix** to **Binary**. This will make it easier for us to enter values and interpret the results.



5. Input waveforms can be drawn in different ways. The most straightforward way is to indicate a specific time range and specify the value of a signal. To illustrate this

approach, click the mouse on the A waveform near the 0-ns point and then drag the mouse to the 400-ns point. The selected time interval will be highlighted in blue. Press **Ctrl + Alt + B** to open the **Arbitrary Value** window to enter the values manually. We will use this approach to create the waveform for B, which should change from 0 to 400 ns.

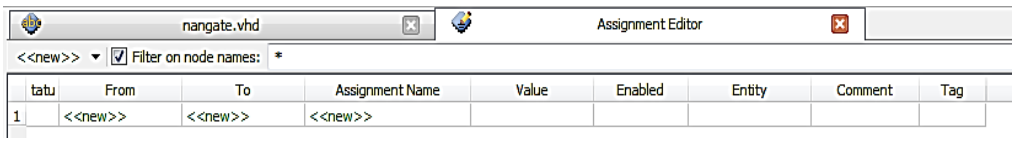


6. Leave **Sum** with a value of undefined (X), as values for this will be generated when the simulation runs. Click the **Save**  icon on the Menu bar and save the vwf file with the filename **nandgate.vwf**.
7. In the Main window, select *Simulation* | *Options* and then select *Quartus II Simulator*. Select *OK*.
8. In the Main window, select *Simulation* and then select *Run Functional Simulation*.
9. Now you should see your simulation output with the outputs defined.

   Note: The file will indicate "read-only" meaning you can't edit it.

Task 4: Implementing the Design on the DE2 Board

1. From the Menu Bar select: **Assignments → Assignment Editor**
2. Double-click **<<new>>** in the **To** column and select **Node Finder** button, List down all the input and output pins and click OK.



3. Double-click the adjacent cell **Assignment Name** and select **Location (Accepts wilds cards/groups).**

   Double-click the adjacent cell **Value** and assign the pin number.

   Continue to assign the pins as seen in the table below.

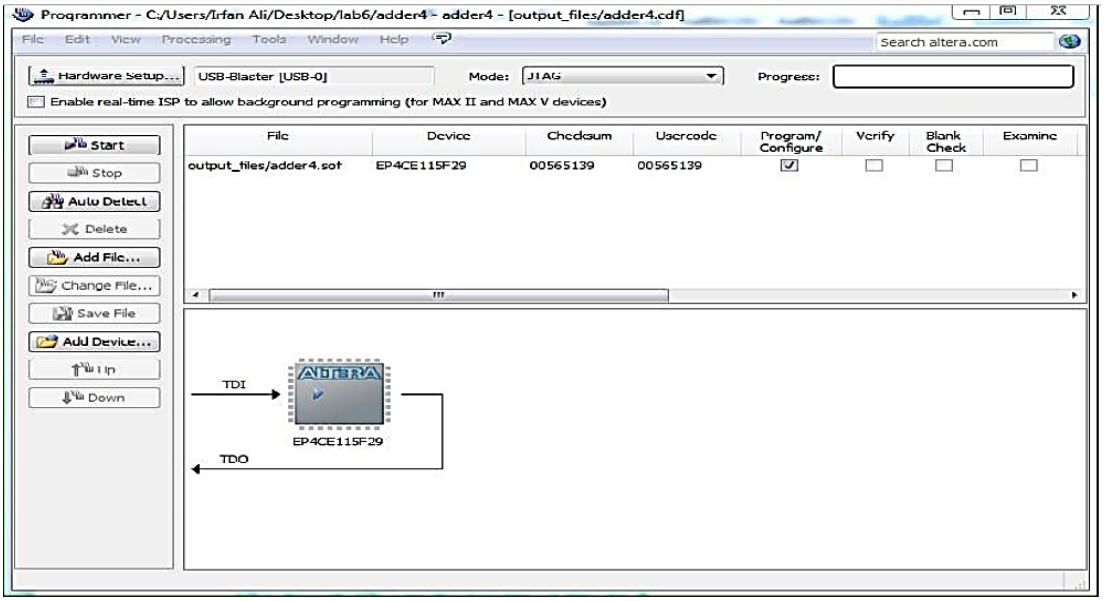
	To	Location	DE2 Board Description
1	A	PIN_AB28	SW0
2	B	PIN_AC28	SW1
3	Cout	PIN_G19	LED0

4. Save the pin assignments by selecting **File→ Save** from the Menu Bar, or by clicking the Save button  on the toolbar.
5. Compile the design again by clicking the Start Compilation button on the toolbar 
6. Plug in and power on the DE2 board. Make sure that the **RUN/PROG Switch for JTAG/AS Modes** is in the **RUN** position.
7. In the Quartus II window click the **Programmer** button on the Toolbar to open the Programmer window 

   The **Hardware Setup…**must be **USB-Blaster [USB-0]**. If not, click the **Hardware Setup…** button and select **USB-Blaster [USB-0]** from the drop-down menu for **currently selected hardware**.

   Mode should be set to **JTAG**.

   Make sure that the **File** is **adder4.sof**, **Device** is **EP4CE115F29C7** and the **Program/Configure** box is checked.



Then click the **Start** button to program the DE2 board.

When the progress bar reaches 100%, programming is complete.

8. You can now test the program on the DE2 board by using the toggle switches located along the bottom of the board.

   SW0 bit of data for A.

   SW1 bit of data for B.

   The output of the operation is displayed on the first LED (LED0) located above the blue push buttons on the DE2 board.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date: _____

LAB # 2:

Explain VHDL programming concepts to design combinational circuit using Altera Quartus tool and implementation on FPGA

Objective

- To understand different architectures in VHDL language
- To understand the designing of combinational circuit
- To understand the implementation of a truth table

Pre-Lab

ARCHITECTURE contains a description of how the circuit should function, from which the actual circuit is inferred. A simplified syntax is shown below.

```
ARCHITECTURE architecture_name OF entity_name IS
  [architecture_declarative_part]
BEGIN
  architecture_statements_part
END [ARCHITECTURE] [architecture_name];
```

The architecture body defines, the internal working, or behavior, of the entity. We can choose between one of three "ways" to describe the architecture body:

i. Structural (see Lab 1)
ii. Dataflow
iii. Behavioral descriptions.

The main difference between these approaches is the level of detail required (or, conversely, the level of abstraction allowed).

Consider an example of a black box,

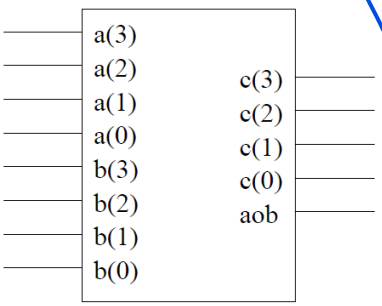


Figure 2.1 Black Box Description of the circuit

The entity declaration for the black box is

```
entity bbox is port(
     a,b : in std_logic_vector(3 downto 0);
     c   : out std_logic_vector(3 downto 0);
     aob : out std_logic);
end bbox;
```

for single bit (vector X)

1. Behavioral Description

Consider the architecture body shown in Figure 1. This behavior description is quite reminiscent of a C-language programmed in many ways. Behavioral descriptions are high-level, just as C is a high-level language. The architectural description is bounded by the architecture and end **arch_bbox** statements. When declaring the architecture **arch_bbox.** We define which entity the architecture belongs to (of bbox is). The **process** statement is used to enclose an algorithm.

The process in Figure 2 is named **comp** and the sensitivity list of **comp** is declared as (a,b). The sensitivity list identifies the signals that will cause the process to execute. In this case, the circuit is sensitive to changes in the two input signals, **a** and **b**. This means that whenever a or b changes, the **comp** process changes

```
architecture arch_bbox of bbox is
begin
  comp: process (a,b) begin
    c <= b;
    if a = b then
        aob <= '1';
     else
       aob <= '0';
     end if
  end process comp;
end arch_bbox;
```

Figure 2.2 Behavioral Description of Black Box Circuit

```
architecture arch_bbox of bbox is
begin
  comp: process (a,b) begin
    c <= b;
    aob <= '0';
    if a = b then
        aob <= '1';
    end if
  end process comp;
end arch_bbox;
```

name of architecture entity

sensitivity list

process is sequential in nature body outside is concurrent

Figure 2.3 An Alternate Behavioral Description of Black Box Circuit

2. DataFlow Description concurrent

A dataflow description is very similar to a behavioral description. In fact, the two are often both referred to as behavioral. The main difference is that a dataflow description does not use the **process** construct. Clearly the dataflow description is easy to understand for a simple example, such as the one we are looking at. With a more complicated algorithm is required, such as one with nested sequential statements, a behavioral description will probably make more sense.

```
architecture dataflow of bbox is
begin
   aob <= '1' when (a=b) else '0';
end dataflow;
```

Figure 2.4: Dataflow Description of Black Box

The big difference between behavioral and dataflow can be seen when we consider a circuit where the inputs may change at any time, but where we only want these (possibly changed) inputs to be noticed when a clock pulse triggers the circuit. We can easily describe this using a behavioral description where **process(clk)** identifies the clock signal as causing the circuit to activate. With a dataflow description, we cannot as easily or neatly control "when" the circuit activates.

3. Combinational logic

Combinational logic can be written with both concurrent and. sequential statements. Con- current statements may be executed in parallel (concurrently) and are found in dataflow and structural descriptions of a circuit. Sequential statements must be executed in a given sequential order and are used in behavioral descriptions (hint: what is the big difference between behavioral and dataflow descriptions?)

3.1 Concurrent Statements

Concurrent statements fall outside of the process statement (and hence fit nicely with dataflow descriptions).

3.1.1 Boolean Statements

The most "obvious" of concurrent statements are Boolean statements. As an example, suppose we wish to build a circuit that will produce the logical-and and logical-or of two bits. We can accomplish this using Boolean statements, as shown in Figure 5.

concurrently execute all and it is dataflow method

```
library ieee;
use ieee.std_logic_1164.all;
entity cct1 is port(
   a,b      : in std_logic;
   land,lor : out std_logic);
end cct1;
architecture archcct1 of cct1 is
begin
   land <= a and b;
   lor  <= a or b;
end archcct1;
```

Figure 2.5: Concurrent Boolean Statement Circuit

The output of this circuit is the two signals, land and lor, produced concurrently (simultaneously).

The Boolean statements that are available in the **ieee 1164** library are: **and**, **or**, **nand**. **not**, **xor** and **xnor**

These data types can be used with bit and Boolean variables (std logic) and with one-dimensional arrays of bits and Boolean variables (such as std logic vector(3 downto 0)), where both variables have the same length.

If you have an equation with multiple Boolean operations, you must use parenthesis to force VHDL into order of operations (otherwise you will get a compile-time error).

3.1.2 With-Select-When Statements

There may be cases where a signal value is assigned based on the value of another signal (a selection signal). In this case, the **with-select-when** statements come in handy.

For example, consider a circuit where the output, **z**. will be assigned the value of signals **a** or **b**, depending on the value of a selection signal. **s**. We can represent this in VHDL as shown in Figure 6.

```
ARCHITECTURE cct OF testcct IS
BEGIN
with s select
  z <= a when '0',
       b when '1';
END cct;
```

s is variable having value 0 and 1

Figure 2.6: Code Fragment: With-Select-When Statements

3.1.3 With-Select-When-When Others

Because of how std logic is defined, a single bit does not necessarily have only two values (unless it is explicitly Boolean). Other possible values include high impedance, unspecified, low impedance, and so on. For this reason, we have the choice of specifying the case when others as a catch-all for all other, not already specified, values of the selection signal. This idea is similar to

the use of default in a C-language case statement. Figure 7 shows the when-others "equivalent" of Figure 6,

```
ARCHITECTURE cct OF testcct IS
BEGIN
with s select
  z <= a when '0',
       b when '1',
default case'0' when others;
```

Figure 2.7: Code: With-Select-When-When Others Statements

The code in Figure 7 states that for any value of **s** other than "0", the signal **z** will have the same value as the signal **b**.

3.1.4 When-Else Statements

The **when-else** statements are a version of the when-select statements where assignment is based on a condition that may or may not revolve around a single signal. The condition evaluated in this type of statement may be based on a single signal, like the when-else statements. or on multiple signals, or multiple conditions involving different signals. For this reason, there is an order of preference within a when-else statement; once a successful, or true, condition is encountered, the assignment specified by the when-else statement is executed and the entire clause is "exited".

For example, the **when-else** equivalent of the code of Figures 6 and 7 is shown in Figure 8.

```
ARCHITECTURE cct OF testcct IS
BEGIN
  z <= a when (s='0') else
       b;
END cct;
```

Figure 2.8: Code Fragment: When-Else Statements

Suppose we only want **z** to be assigned the value of **a** or **b** based on the select signal **s** and some other condition. We can create compound conditional statements within a when-else statement. All that we have to do enclose the compound statement in a set of parentheses, to "create" a simple statement, as seen in Figure 9.

```
ARCHITECTURE cct OF testcct IS
BEGIN
  z <= a when (s='0' and ocond1=true) else
       b when (s='1' and ocond2=true) else
       z;
END cct;
```

*Figure **Error! Use the Home tab to apply 0 to the text that you want to appear here.**-39: Code Fragment: When-Else Statements*

Task 1: Implementation of Truth Table: Step-by-Step Practice

Let's start by creating a truth table which is the basic information for any combinatorial logic circuit. We will choose a 2-to-4 binary decoder, which would be described by the following table.

Input		Output			
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

A binary decoder is a circuit that simply translates binary to a position of bit information. Using the syntax given at the intro, this table can be coded as below, assuming input as a two-bit vector and the output as a four-bit vector:

```
output <= "0011" when (input = "00") else
          "0110" when (input = "01") else
          "1100" when (input = "10") else
          "1001";
```

Next, we are going to use this code to implement the **2-to-4 binary decoder**.
Please follow the steps provided in lab # 1
You should see the code window that contains the basic template of a VHDL module which should look like below, excluding the comments.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity dec2to4 is
end dec2to4;

architecture Behavioral of dec2to4 is

begin

end Behavioral;
```

Next, complete the entity and architecture code as shown below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dec2to4 is
  port ( input :in std_logic_vector(1 downto 0);
         output :out std_logic_vector(3 downto 0));
end dec2to4;

architecture Behavioral of dec2to4 is
begin
  output <= "0011" when (input = "00") else
            "0110" when (input = "01") else
            "1100" when (input = "10") else
            "1001";

end Behavioral;
```

Assign the pins as seen in the table below.

Input		Output			
Input (1)	Input (0)	Output (3)	Output (2)	Output (1)	Output (0)
SW1	**SW0**	**LED3**	**LED2**	**LED1**	**LED0**
PIN_AC28	PIN_AB28	PIN_F21	PIN_E19	PIN_F19	PIN_G19

In-Lab Tasks

When you test the implementation using slide switches as the input like **2-to-4 binary decoder** example, always gingerly move the switch handles. Because it is a mechanical switch, it generates bouncing signals (noise) when you move the switch from on-to-off or vice versa, often giving errors or damaging input ports. The best contact condition of slide switches is achieved when you apply just a right amount of force; if you push too hard, you will get bad contact conditions.

Lab Task 1: Implement the 3-to-8 decoder

Complete the truth table of 3-to-8 decoder and show its results on the FPGA

input	output
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

Lab Task 2: Seven Segment Display (SSD)

The DE2-115 Board has eight 7-segment displays. These displays are arranged into two pairs and a group of four, behaving the intent of displaying numbers of various sizes. As indicated in the schematic in Figure 10, the seven segments (common anode) are connected to pins on Cyclone IV E FPGA. Applying a low logic level to a segment will light it up and applying a high logic level turns it off.

Each segment in a display is identified by an index from 0 to 6, with the positions given in Figure 10.

Figure

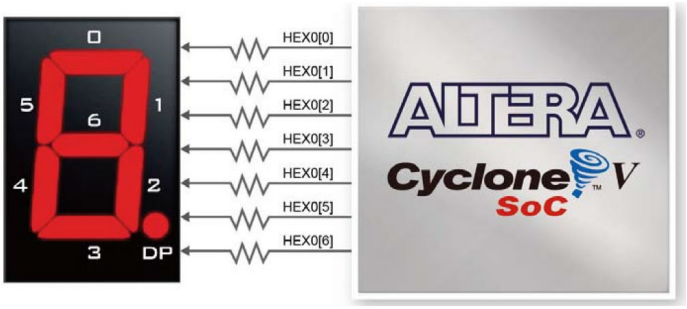


*Figure **Error! Use the Home tab to apply 0 to the text that you want to appear here.**-4: Connections between the 7-segment display HEX0 and Cyclone IV E FPGA*

Next you are going to write a VHDL code to control the SSD. Please follow the steps provided below.

Step 1: Create a new project

Step 2: Using a New Source wizard, create a VHDL module

Step 3: Write the entity

```
library ieee;
use ieee.std_logic_1164.all;
entity seven is
port(
        cathodes: out std_logic_vector(6 downto 0);
        bininput: in std_logic_vector(3 downto 0)
);
end seven;
```

The port "**cathodes**" is a seven-bit vector that would be connected to the cathodes of LED segments. The port **binInput** provides binary input patterns and connected to the four slide switches.

Step 4: Complete the binary to SSD conversion table and finish the decoder that can display all 4bit binaries to SSD, i.e., display of patterns, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. and write the architecture code.

bininput	cathodes
0000	6543210 1000000
0001	1111001
0010	0100100
0011	
0100	
0101	
0110	
0111	
1000	

1001	
1010	
1011	
1100	
1101	
1110	
1111	

Step 5: Now write the dataflow architecture for the truth table generated in step 4,

```
architecture dataflow of seven is
begin
        cat <= "1000000"    when (bininput ="0000") else
               "1111001"    when (bininput ="0001") else
               "0100100"    when (bininput ="0010") else
               "0110000"    when (bininput ="0011") else
```

*Hint: Write the remaining patterns for **bininput** variable*

Step 6: Assign the pins as seen in the table below

Pin Assignments for bininput	
SW0	PIN_AB28
SW1	PIN_AC28
SW2	PIN_AC27
SW3	PIN_AD27

Pin Assignments for 7-segment Displays	
HEX0[0]	PIN_G18
HEX0[1]	PIN_F22
HEX0[2]	PIN_E17
HEX0[3]	PIN_L26
HEX0[4]	PIN_L25
HEX0[5]	PIN_J22
HEX0[6]	PIN_H22

Step 7: Generate the bit file, download it to the board, and test the circuit. Remember that by changing the **bininput** slide switches, the value of 7-segment will change.

Challenge:

a) Can you put the same digit on another 7-segment? How

b) Lets say your switches are set to **ON OFF OFF ON** , then digit "9" should be displayed on one of the 7-segment, now you want to put digit "8" (one less than the digit on 7-segment) on another 7-segment. What changes do you need to do in your code in order to achieve this?

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:**_____

LAB # 3:
Display the output of combinational circuit using library building technique of VHDL programming

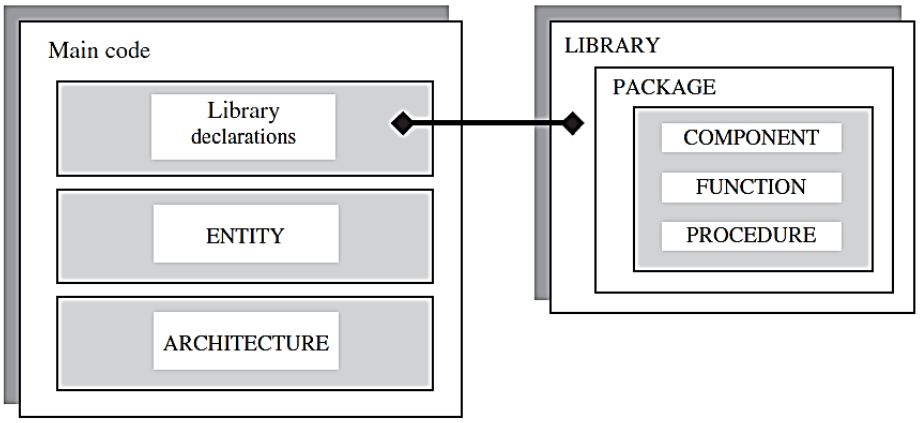
Objective

- To learn to utilize the library functions in VHDL
- To learn how to create own library in VHDL

Pre-Lab

Creating your own library components

Just like other programing languages, VHDL also provide a functionality to create your own library called package. The package contains components, function, procedures, constants and types. In this lab, we will learn how to create our own library.



We will learn this concept by walking you through an example (as pre-lab) and later on asking you; the student, to create another library to solve the in-lab task.

Pre-Lab Tasks

Task 1: Creating your own library: Step-by-Step Practice

Consider the combinational circuit given in the figure 1.

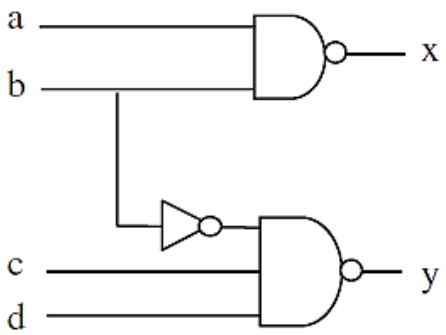


Figure ***Error! Use the Home tab to apply 0 to the text that you want to appear here.****-5 : Combinational circuit*

The circuit in figure 1 can be implemented as

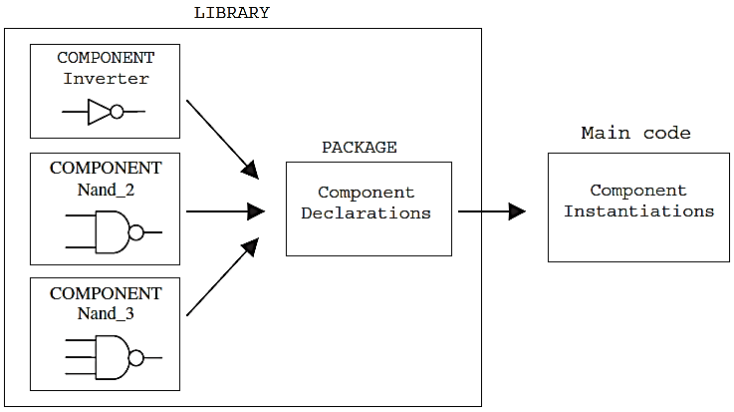


Figure 3.2: contents of user-defined library

1) Entity

The firsts part in creating a library is to define the entities that needs to be grouped together as a package. The entities required are

i. An inverter

```
-- Code for myinverter.vhd file

library IEEE;
use ieee.std_logic_1164.all;
entity myinverter is
port(
        inv_in: in std_logic;
        inv_out: out std_logic
);
end myinverter;
architecture bhv of myinverter is
begin
    inv_out <= not inv_in;
end bhv;
```

ii. A 2-input NAND

```
----------------------Code for nand_2.vhd

library IEEE;
use ieee.std_logic_1164.all;
entity nand_2 is
port(
        nand2_in1: in std_logic;
        nand2_in2: in std_logic;
        nand2_out: out std_logic
);
end nand_2;
architecture nand_2 of nand_2 is
begin
    nand2_out <= NOT (nand2_in1 AND nand2_in2);
end nand_2;
```

iii. A 3-input NAND

```
----------------------Code for nand_3.vhd
library IEEE;
use ieee.std_logic_1164.all;
entity nand_3 is
port(
        nand3_in1: in std_logic;
        nand3_in2: in std_logic;
        nand3_in3: in std_logic;
        nand3_out: out std_logic
);
end nand_3;
architecture nand_3 of nand_3 is
begin
    nand3_out <= NOT (nand3_in1 AND nand3_in2 AND nand3_in3);
end nand_3;
```

2) Package

Once the entities are created now, we need to create the package (library in our case)

```
----------------------Code for my_components.vhd
library ieee;
use ieee.std_logic_1164.all;

package my_components is
component myinverter is
port( inv_in: in std_logic; inv_out: out std_logic);
end component;

component nand_2 is
port    (   nand2_in1: in std_logic;
            nand2_in2: in std_logic;
            nand2_out: out std_logic
        );
end component;

component nand_3 is
port(
        nand3_in1: in std_logic;
        nand3_in2: in std_logic;
        nand3_in3: in std_logic;
        nand3_out: out std_logic
);
end component;
end my_components;
```

3) The top-entity (wrapper file)

Now we need to define another vhdl file that will be called a wrapper file. This file will integrate components from our package and create a circuit.
Remember to set this file as top-level entity.

```
-- -- Code for circuit1, wrapper file

library ieee;
use ieee.std_logic_1164.all;
use work.my_components.all; -- declaring our Library

entity circuit1 is
port( a,b,c,d : in std_logic;
      x,y: out std_logic
    );
end circuit1;

architecture struct of circuit1 is
signal w: std_logic;
begin
    --calling the components
    U1: myinverter PORT MAP
                  (inv_in  => b,
                   inv_out => w);
    U2: nand_2 PORT MAP
                  (nand2_in1 => a,
                   nand2_in2 => b,
                   nand2_out => x);
    U3: nand_3 PORT MAP(w,c,d,y);
end struct;
```

iv. Mapping Options

A ***port map*** is used to define the interconnection between instances.
PORT MAP is simply a list relating the ports of the actual circuit to the ports of the predesigned circuit (component being instantiated). Such a mapping can be positional or nominal, as illustrated in the example below, which employs the nand3 circuit again.

```
-------- Component instantiation: ----------------------
nand3_1: nand3 PORT MAP (x1, x2, x3, y); --positional mapping
nand3_2: nand3 PORT MAP (a1=>x1, a2=>x2, a3=>x3, b=>y); --nominal mapping
```

Use the following pin assignments to test the code on FPGA.

a	b	c	d	x	y
PIN_AD27	PIN_AC27	PIN_AC28	PIN_AB28	PIN_G19	PIN_F19

In-Lab Tasks

Lab Task 1: ALU implementation using your own library components

Using the techniques, you learned from prelab, implement a single bit ALU shown in figure 3 using VHDL codes and components in your library.

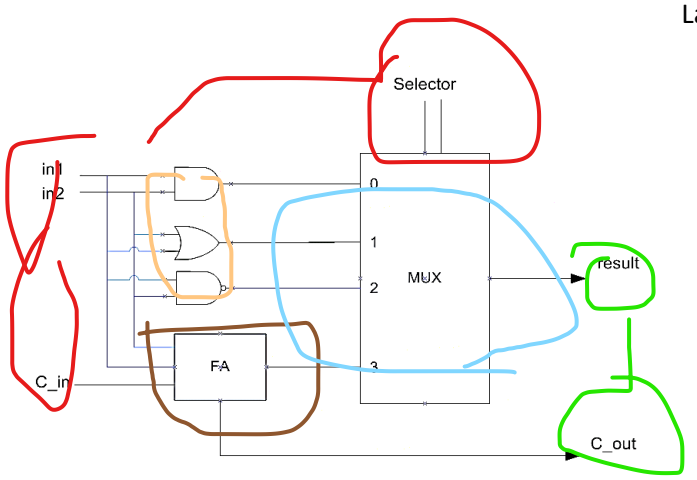


Figure ***Error! Use the Home tab to apply 0 to the text that you want to appear here.****-6: A simple ALU implementation*

Your library should include

- ~~Orgate~~
- ~~Andgate~~
- ~~Nandgate~~
- Halfadder
- ~~Fulladder~~
- ~~mux.~~

Use the following pin assignments to test your code on FPGA

S1	S0	C_in	In2	In1	x	y
PIN_AB27	PIN_AD27	PIN_AC27	PIN_AC28	PIN_AB28	PIN_G19	PIN_F19

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:**_____

LAB # 4:
Display the output of the sequential circuit using VHDL programming techniques

Objective

- To learn to design sequential circuits in VHDL
- To learn sequential circuit constructs in VHDL
- To observe the behaviour of D-Latch and Flip-Flop

Pre-Lab

Familiarize yourself with Sequential Circuit

Introduction:

A sequential circuit is a circuit with *memory*, which forms the internal state of the circuit. Unlike a combinational circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state. The synchronous design methodology is the most commonly used practice in designing a sequential circuit. In this methodology, all storage elements are controlled (i.e., synchronized) by a global ***clock*** signal and the data is sampled and stored at the ***rising*** or ***falling edge*** of the clock signal. It allows designers to separate the storage components from the circuit and greatly simplifies the development process. This methodology is the most important principle in developing a large, complex digital system and is the foundation of most synthesis, verification, and testing algorithms.

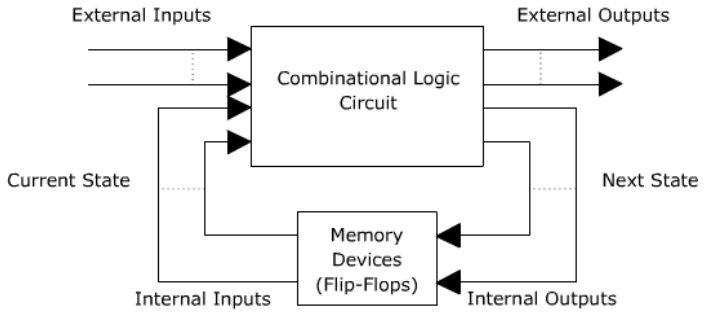


Figure 4.1 Sequential Circuit

Types of Sequential Circuit:

There are two types of sequential circuit, **synchronous** and **asynchronous**.

Synchronous Circuit:

Synchronous types use pulsed or level inputs and a clock input to drive the circuit (with restrictions on pulse width and circuit propagation).

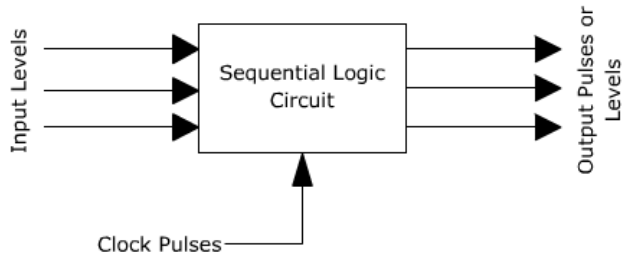


Figure 4.2 Synchronous circuit

Asynchronous Circuit:

Asynchronous sequential circuits do not use a clock signal as synchronous circuits do. Instead, the circuit is driven by the pulses of the inputs. You will not need to know any more about asynchronous circuits for this course.

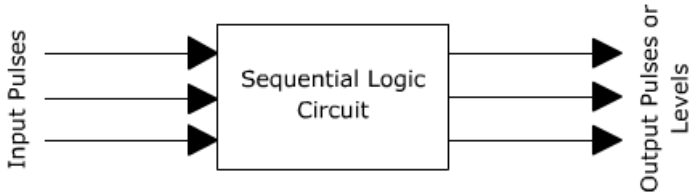


*Figure **Error! Use the Home tab to apply 0 to the text that you want to appear here.**-7 Asynchronous Circuit*

Sequential circuit coding in VHDL

In VHDL, concurrent code is intended only for the design of combinational circuits, while sequential code can be used indistinctly to design both sequential and combinational circuits. The statements intended only for completely concurrent code, referred to as concurrent statements, are WHEN, SELECT, and GENERATE (Lab 1 to Lab 3), while those for sequential code, referred to as sequential statements, are IF, WAIT, LOOP, and CASE.
In VHDL, there are three kinds of sequential code:

i. PROCESS
ii. FUNCTION (subprograms)
iii. PROCEDURE (subprograms)

PROCESS:

PROCESS is intended for the architecture body (main code, for example), while subprograms, being intended mainly for libraries (Lab 5), which deals with system-level code. ***Remember*** that a PROCESS or a subprogram call is a concurrent statement.

PROCESS is a sequential section of VHDL code, located in the statements part of an architecture. Inside it, only sequential statements (IF, WAIT, LOOP, CASE) are allowed. A simplified syntax is shown below

```
[label:] PROCESS [(sensitivity_list)] [IS]
   [declarative_part]
BEGIN
  sequential_statements_part
END PROCESS [label];
```

- The **label**, whose purpose is to improve readability in long codes, is optional.
- The **sensitivity list** is mandatory (but is forbidden when WAIT is used), and causes the process to be run every time a signal in the list changes (or the condition associated with WAIT is fulfilled).
- The **declarative part** of PROCESS can contain the following:
    - subprogram declaration
    - subprogram body
    - type declaration
    - subtype declaration
    - constant declaration
    - variable declaration
    - file declaration
    - alias declaration
    - attribute declaration
    - attribute specification
    - use clause
    - group template declaration
    - group declaration.
    - ***Signal declaration is not allowed*, while variable is by far the most common declaration**
- The **statements part** of PROCESS, only sequential statements are allowed (besides operators as they can go in any kind of code).

A crucial point when dealing with sequential code is to fully understand the difference between SIGNAL and VARIABLE.

Main properties of SIGNAL:

- A signal can only be declared outside sequential code (though it can be used there).
- A signal is not updated immediately (when a value is assigned to a signal inside sequential code, the new value will only be ready after the conclusion of that run).
- A signal assignment, when made at the transition of another signal, will cause the inference of registers (given that the signal affects the design entity).
- Only a single assignment is allowed to a signal in the whole code (even though the compiler might accept multiple assignments to the same signal in PROCESS or subprograms, only the last one will be effective, so again it is just one assignment).

Main properties of VARIABLE:

- A variable can only be declared and used inside a PROCESS or subprogram (if it is a shared variable, then the declaration is made elsewhere, but it still should only be modified inside a sequential unit).

- A variable is updated immediately (hence the new value can be used/tested in the next line of code).
- A variable assignment, when made at the transition of another signal, will cause the inference of registers (assuming that the variable's value affects a signal, which in turn affects the design entity).
- Multiple assignments are fine.

Sequential statements

a) If-Then-Else statements

The if-then-else statements have the same meaning in VHDL as they do in the C-language. By comparison with the combinational statements, these statements are the sequential equivalents of the ***with-select-when*** and ***when-else statements***.

```
-- 2x1 MUX using Sequential if-else implementation
    ARCHITECTURE cct OF testcct IS BEGIN
    test process(a,b,s) begin
            if  s= 0 then
                z  = a;
            elsif (s= 1 ) then z = b;
            end if;
        end process;
    END cct;
```

FIGURE ERROR! USE THE HOME TAB TO APPLY 0 TO THE TEXT THAT YOU WANT TO APPEAR HERE.-8 2X1 MUX USING IF-ELSE STATEMENT

```
-- 2x1 MUX using Concurrent with-select-when implementation
    ARCHITECTURE cct OF testcct IS BEGIN
        with s select
            z  = a when  0 ,
                 b when  1 ,
                 0 when others;
```

Figure 4.4 4x1 MUX concurrent implementation

b) Case-When Statements

A **case-when** statement is the sequential equivalent of a **with-select-when** statement. Figure 6 shows the case-when equivalent of the simple, single-bit multiplexor-type circuit of Figures 4 and 5.

Note that this code must account for the other possible values of the signal s.

```
-- 2x1 MUX using Sequential case-when statement
    ARCHITECTURE cct OF testcct IS BEGIN
        test process(s)
            begin
                case s is
                    when 0  =>  z  =  a;
                    when 1  =>  z  =  b;
                    when others => z = b;
                end case;
            end process;
```

Figure 4.5 2x1 MUX using case-when statement

Pre-Lab Tasks

Latches and Flip-Flops

D Latch:

Latch is an electronic device that can be used to store one bit of information. The D latch is used to capture, or 'latch' the logic level which is present on the Data line when the clock input is high. If the data on the D line changes state while the clock pulse is high, then the output, Q, follows

the input, D. When the CLK input falls to logic 0, the last state of the D input is trapped and held in the latch.

Timing diagram

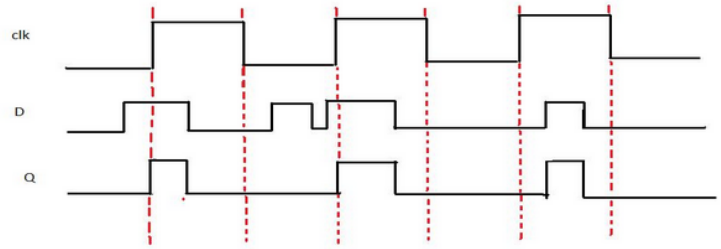


Figure 4.6 Timing of D Latch

From Figure 7, we can see that the input D is transferred to output Q when the clock remains at logic '1'. This is the true behaviour of a level triggered circuit.

Pre-Lab Task1: Create and observe the waveform of D-latch

For pre-lab task1, you are required to make a waveform of the code given in Figure 8 and compare it with waveform of Figure 7.

```
library ieee;
use ieee.std_logic_1164.all;

entity mydlatch is port(
    clk : in std_logic;
    D   : in std_logic;
    Q   : out std_logic
);

end mydlatch;

architecture df of mydlatch is

begin
    with clk select
        q <= d when '1',
             '0' when '0';
end df;
```

Figure 4.7 Code for D Latch

D Flip-Flop:

The working of D flip flop is similar to the D latch except that the output of D Flip Flop takes the state of the D input at the moment of a positive edge at the clock pin (or negative edge if the clock input is active low) and delays it by one clock cycle.

Timing diagram

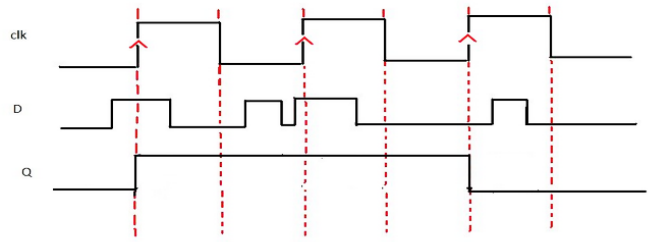


Figure 4.8 Timing of D Flip-Flop

Pre-Lab Task2: Create and observe the waveform of D Flip-Flop

For pre-lab task2, you are required to make a waveform of the code given in Figure 10 and compare it with waveform of Figure 9.

```
library ieee;
use ieee.std_logic_1164.all;

entity mydlatch is port(
    clk : in std_logic;
    D   : in std_logic;
    Q   : out std_logic
);

end mydlatch;

architecture df of mydlatch is

begin
    process (clk)
        begin
            if clk='1' then
                Q <= D;
            end if;
    end process;
end df;
```

Figure 4.9 Code for D Flip-Flop

In-Lab Tasks

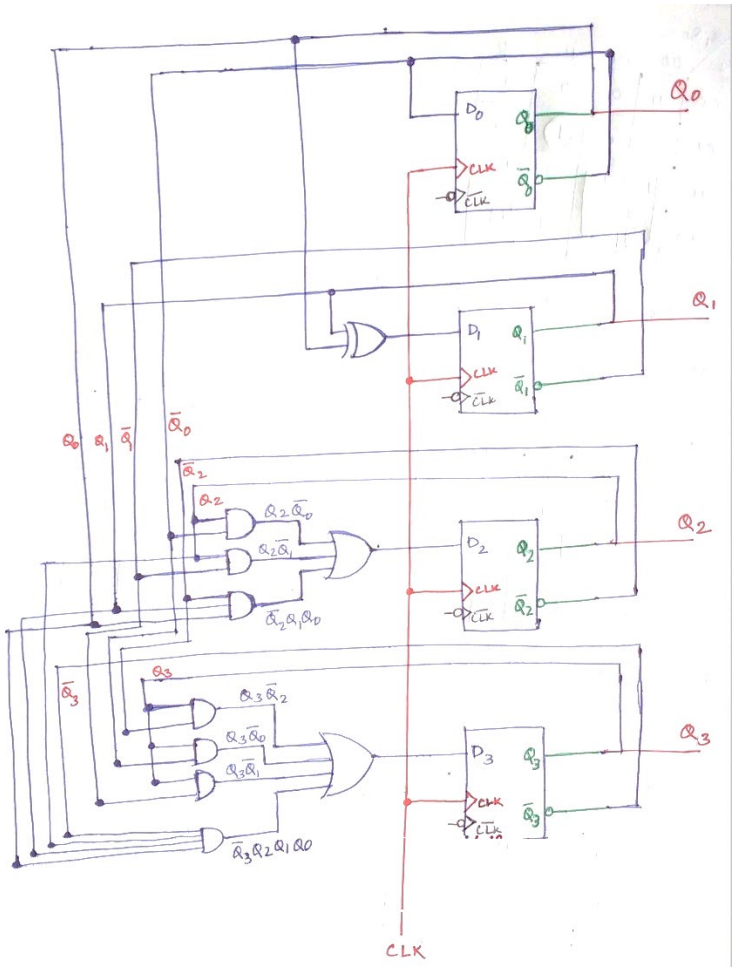
Lab Task 1: Binary counter using D Flip-Flop

For first lab task, you are to design a free-running binary counter that circulates through a binary sequence repeatedly. For example, a 4-bit binary counter counts from "0000", "0001", "0010"… "1111" and wraps around. When the code starts you will display "0000" on LEDs. After every clock pulse, the LEDs display the next number according to the truth table given below

package me just sub module ki port declaration wala hisa copy paste karna ha

Present State (Q3 Q2 Q1 Q0)	Next State (Q3+ Q2+ Q1+ Q0+)	D3	D2	D1	D0
0000	0001	0	0	0	1
0001	0010	0	0	1	0
0010	0011	0	0	1	1
0011	0100	0	1	0	0
0100	0101	0	1	0	1
0101	0110	0	1	1	0
0110	0111	0	1	1	1
0111	1000	1	0	0	0
1000	1001	1	0	0	1
1001	1010	1	0	1	0
1010	1011	1	0	1	1
1011	1100	1	1	0	0
1100	1101	1	1	0	1
1101	1110	1	1	1	0
1110	1111	1	1	1	1
1111	0000	0	0	0	0

The schematic of the binary counter is given below



Implement this schematic using packages, show the waveforms and test it on FPGA.
To test the functionality of sequential circuits we will assign a push button for clock signal, after testing is done, we will assign the original or slow version of clock available on FPGA.
For Pin Assignment, consult the Altera handout.
The state diagram of 4-bit binary counter is

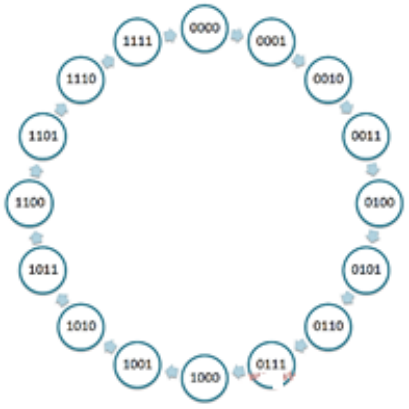


Figure 4.10 State diagram of 4-bit counter

Lab Task 2: Up/Down Binary counter

A 3-bit binary up/down counter is more versatile. It can count, up or down depending upon selection. If selection is ZERO then the output should go from 000,001 to 111 on rising edge of the clock and if selection is ONE, then the output should go from 111,110 to 000.
Your second lab task is to modify the design from task 1 and add the functionality of up/down counting. Remember that you might need to do the same working for the down counter as it is done for you previously.

1) Complete the state transition table

Input	**Current State**			**Next State**			**Flip Flop inputs**		
updown	Q_2	Q_1	Q_0	Q_2	Q_1	Q_0	D_2	D_1	D_0
0	0	0	0	0	0	1	0	0	1
0	0	0	1	0	1	0	0	1	0
0	0	1	0	0	1	1	0	1	1
0	0	1	1	1	0	0			
0	1	0	0	1	0	1			
0	1	0	1	1	1	0			
0	1	1	0	1	1	1			
0	1	1	1	0	0	0			
1	0	0	0	1	1	1			
1	0	0	1	0	0	0			
1	0	1	0	0	0	1			
1	0	1	1	0	1	0			
1	1	0	0	0	1	1			
1	1	0	1	1	0	0			
		1	0	1	0	1			
		1	1	1	1	0			

2) the minimize state equation.
3) aflow modeling in VHDL

Note: A for handling such problems is to use the idea of state machine irly an easy approach once it comes to VHDL design. We will revi

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date:_____

LAB #5
Display the output of sequential circuit using VHDL programming techniques

Objective

- To learn how to make procedures and functions in VHDL
- To learn to behavioural modelling of sequential circuits in VHDL
- To learn how to design finite state machines in VHDL

Pre-Lab

Lab 5 is divided into two parts, in part one we will learn about two new constructs in VHDL programming, procedures and functions. In second part we will learn about how state machines are implemented in VHDL.

Part-1 Procedures and Functions

FUNCTION and PROCEDURE (called subprograms) are very similar to PROCESS, in the sense that these three are the only sections of VHDL code that are interpreted sequentially. Consequently, only sequential statements (IF, WAIT, LOOP, CASE) are allowed (plus operators, of course, because these can be used in any kind of code).

On the other hand, contrary to PROCESS, which is intended for the ARCHITECTURE body (regular codes), subprograms can be constructed in a PACKAGE, ENTITY, ARCHITECTURE, or PROCESS. Because PACKAGE is the most common location, with which the VHDL libraries are built, in our context subprograms are considered system-level units (along with PACKAGE and COMPONENT).

FUNCTION

FUNCTION is a section of sequential VHDL code whose main purpose is to allow the creation and storage in libraries of solutions for commonly encountered problems, like data-type conversions, logical and arithmetic operations, etc.

FUNCTION is similar to PROCESS in the sense that it too is sequential and therefore can only use the same statements (IF, WAIT, LOOP, and CASE). A simplified syntax for the construction of functions is shown below.

```
[PURE | IMPURE] FUNCTION function_name [(input_list)]
RETURN return_value_type IS
   [declarative_part]
BEGIN
   statement_part
  [label:] RETURN expression;
END [FUNCTION] [function_name];
```

Let's explain each thing separately,

PURE: A function is said to be pure when it can only modify its own variables. If left unspecified, the default value (PURE) is assumed.

IMPURE: An impure function, on the other hand, may also modify signals or variables from the architecture, process, or subprogram where it is declared.

The input list can contain any number of parameters (including zero), which are all of mode IN (that is, all parameters are inputs to the function). The list can only contain the objects CONSTANT (default), SIGNAL, and FILE (VARIABLE is not allowed).

Example The function below, named ***positive_edge***, receives a signal called s, returning TRUE when a positive transition occurs on s.
A function can be declared inside an ARCHITECTURE, but we will always be declaring FUNCTION and PROCEDURE in a PACKAGE like the example shown below

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

PACKAGE my_components is
    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
END PACKAGE;

PACKAGE BODY my_components IS

    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
        BEGIN
            RETURN (s'EVENT AND s='1');
    END FUNCTION positive_edge;

END PACKAGE BODY;
```

Once the function is declared then the other point is how to **CALL** the function in our ARCHITECURE block.
Three equivalent function calls are shown below.

```
-----Function declaration:----------------------------
FUNCTION my_function (SIGNAL a, b: BIT) RETURN BIT;
-----Equivalent function calls:----------------------
y <= my_function (x1, x2); --positional mapping
y <= my_function (a=>x1, b=>x2); --nominal mapping
y <= my_function (b=>x2, a=>x1); --nominal mapping
-----------------------------------------------------
```

PROCEDURE

The purpose, construction, and usage of PROCEDURE are similar to those of FUNCTION. Their main difference is that a PROCEDURE can return more than one value. A syntax for the construction of procedures is shown below.

```
PROCEDURE procedure_name (input_output_list) IS
   [declarative_part]
BEGIN
   statement_part
END [PROCEDURE] [procedure_name]
```

The **input-output list** can contain CONSTANT, SIGNAL, and VARIABLE. Their mode can be IN, OUT, or INOUT; if it is IN, then CONSTANT is the default object, while for OUT and INOUT the default is VARIABLE. Their declarations are as follows:

```
CONSTANT constant_name: mode constant_type;
SIGNAL signal_name: mode signal_type;
VARIABLE variable_name: mode variable_type;
```

Both FUNCTION and PROCEDURE are sequential codes, so only sequential statements are allowed. PROCEDURE can be constructed and used in the same way, with PACKAGE (plus the corresponding PACKAGE BODY) as the most common location (for libraries).

Like function calls, procedure calls can be made basically anywhere (in sequential as well concurrent code, in subprograms, etc.). However, the former is called as part of an expression, while the latter is a statement on its own. Examples of procedure **CALLS** are shown below.

```
-------------------------------------------------
sort (a1, a2, a3, b1, b2, b3);
-------------------------------------------------
divide (dividend, divisor, quotient, remainder);
-------------------------------------------------
IF (x>y) THEN get_max (x1, x2, x3, x4, y1, y2);
-------------------------------------------------
```

Here, the procedure names are sort, divide and get_max. Another important reason for using PROCEDURE and FUNCTION over PORT MAP is that the formers can be used inside a sequential code block(process).

Example: The function, named ***positive_edge***, can be declared using procedures as

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

PACKAGE my_components is
    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
    PROCEDURE pos_edge (SIGNAL s: in std_logic; SIGNAL sout : out std_logic);
END PACKAGE;

PACKAGE BODY my_components IS

    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
        BEGIN
            RETURN (s'EVENT AND s='1');
    END FUNCTION positive_edge;

    PROCEDURE pos_edge (SIGNAL s: in std_logic; SIGNAL sout : out std_logic) IS
    BEGIN
        if (s'event and s='1') then
            sout <= '1';
        else
            sout <='0';
        end if;
    END PROCEDURE pos_edge;
END PACKAGE BODY;
END my_components;
```

The code for our package (my_components) given above has a function named positive_edge and a procedure named pos_edge. As a beginner student, you should pay close attention to how both functions and procedures are declared and later **CALLed** in an architecture.

```
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.my_package.all;
entity fastcounter is port(
    clk : in std_logic;
    binout : out std_logic_vector(3 downto 0)); end fastcounter;

ARCHITECTURE behav of fastcounter is
signal cout: std_logic_vector(3 downto 0):="0000";
SIGNAL sclk: std_logic;
BEGIN
    binout <= cout;
    PROCESS(clk)
    begin
        --if clk='1' then
    ------------ Using FUNCTION------------
        if (positive_edge(clk)) then

            cout <= cout + '1';
        end if;

    ------------ Using PROCEDURE------------
        pos_edge( clk, sclk);
        if (sclk ='1') then
            cout <= cout + '1';
        end if;
    end process;
end behav;
```

While testing the above code try to comment either FUNCTION or PROCEDURE part of the PROCESS. As both are doing the same thing.

Pre-Lab Tasks

Implement 4-bit Binary counter implemented in Lab 4 using procedures and functions

You are to design the 4-bit binary counter and this time you have to show the output on 7-segment display. In order to do that work, you need to make a function/procedure for 7-segment display. You can use the code given below for counter

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity fastcounter is port(
    clk : in std_logic;
    binout : out std_logic_vector(3 downto 0)

); end fastcounter;

architecture behav of fastcounter is
signal cout: std_logic_vector(3 downto 0):="0000";
begin

    binout <= cout;
    process(clk)
    begin
        if clk='1' then
            cout <= cout + '1';
        end if;

    end process;
end behav;
```

Figure 5.1 Up counter using behavioral modelling in VHDL

Part-2 Implementing State Machines in VHDL

A Finite State Machine (FSM) is a circuit/machine/system that goes through a fixed number of states and has fixed numbers of input/output combinations. FSMs are typically used to control, monitor, calculate a circuit or to implement a communication protocol etc.
To implement FSM in VHDL, we need to know about three important things

i. **Next State Logic (NSL)**
   It will help us decide which state we will go under what condition
ii. **Output Function Logic (OFL)**
   What to do in what state under what condition
iii. **State Memory (SM)**
   Holds the current state

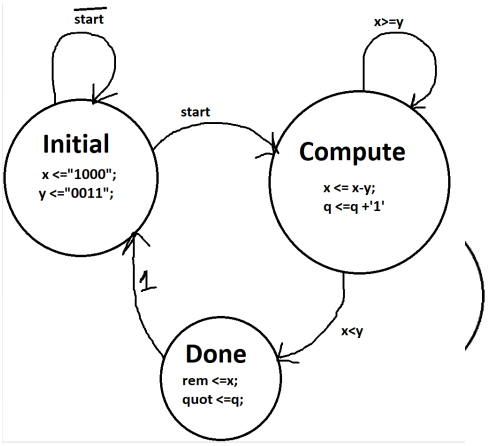
Writing an FSM

- We will model NSL using if-else statement
- Each state will be modeled using case statement
- State variables will be defined as constants (using one-hot encoding)
- OFL will be modeled using conditional assignments

In-Lab Tasks

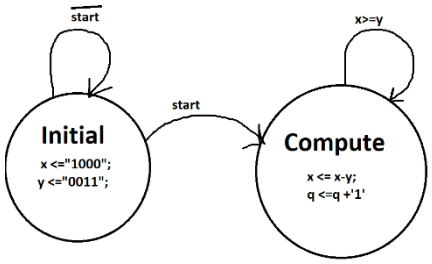
Lab Task 1: Implement the FSM on FPGA

In Lab Task 1, we will learn how to implement a divider circuit using FSM. We will be implementing division using subtraction. The state diagram shown has three states namely, initial, compute and done.

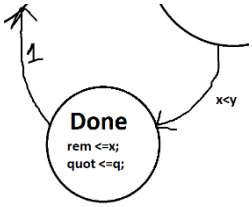


In state **initial**, we will initialize the dividend (x as 1000_2) and divisor (y as 0011_2), this will act as our OFL. Our state machine will stay in initial state until a start button is turned ON, so the NSL will be, go to state compute if start is high otherwise stay in state initial.

In state **Compute**, we will start the actual division process. At every clock positive edge, we will compute x <= x-y; and increment "q", this will act as our OFL. For NSL, our state will change to state done when x goes less than y.



In state **done**, we will simply assign x to our remainder and q to our quotient and unconditionally move to state initial.



Now your task is to test the functionality of the code on FPGA.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity FSMdiv is port(
    clk : in std_logic;
    letstart : in std_logic;
    remaind : out std_logic_vector(3 downto 0);
    quot    : out std_logic_vector(3 downto 0)
);end FSMdiv;

architecture behav of FSMdiv is
signal x: std_logic_vector(3 downto 0) :="1000";  -- Dividend is 8
signal y: std_logic_vector(3 downto 0) := "0011";  -- Divisor is 3
signal q: std_logic_vector(3 downto 0) := "0000"; -- For Quotient

-- defining state using one-hot coding
constant initial : std_logic_vector(2 downto 0) :="001";
constant compute : std_logic_vector(2 downto 0) :="010";
constant done : std_logic_vector(2 downto 0)    :="100";
-- This is our State Memory SM
signal mystate : std_logic_vector(2 downto 0) := "001";
begin
    process(clk)
```

```
begin
    if (clk'event and clk='1') then

        case mystate is
            when initial =>
                -- OFL
                x <= "1000";  -- 8
                y <= "0011";  -- 3

        -- NSL
                if (letstart ='1') then
                    mystate <= compute;
                else
                    mystate <= initial;
                end if;

            when compute =>
             -- OFL
             if (x>=y) then
                x <= x-y;
                q <= q+1;

             end if;

             --NSL

                if (x>=y) then
                    mystate <= compute;
                else
                    mystate <= done;

                end if;

            when done =>
            --OFL binding input signals with top-level entity signals
                remaind <= x;
                quot <= q;
            --NSL
                mystate <=initial;
      when others =>
          mystate <=initial;

      end case;
      end if;  -- end if of clk'event

  end process;
end behav;
```

Lab Task 2: Implement a 4-bit sequence detector

In Lab Task 2, you are to design a FSM that detects the input pattern of 1011. Create its state diagram and implement it using the technique learnt in previous lab task.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date:_____

LAB # 6:

Follow the steps to reproduce the sequential circuit using advanced VHDL programming techniques

Objective

- To understand the working of clock in FPGA design
- To generate slow clocks from a given clock
- To understand the function of a RESET signal in digital circuits.
- To design an electronic wheel of the fortune, utilizing the package and procedure constructs

Pre-Lab

Clocks and reset signals in digital circuits

a) Clock signal

A process statement easily handles synchronous logic, by allowing us to specify a clock signal as one of the triggers to the circuit. Figure 1 shows a circuit that is sensitive only to changes in the clock signal.

```
ARCHITECTURE cct OF testing IS BEGIN
process (clk)
begin
...
end process;
END cct;
```

Figure 6.1: Code Fragment: A Clock-Sensitive Circuit

Let's think about what a clock signal, or any waveform for that matter, looks like. There will be a *rising edge* and a *falling edge*. Do we want our circuit to be triggered by any change in the clock signal, or *{positive1negative} edge-triggered*? How do we represent edge- triggering in VHDL?

Edge-triggering requires two conditions to be true:

1) the clock signal must change
2) it must change in the positive (negative) direction, as required.

Representing edge-triggering in VHDL therefore requires a compound statement, specifying each of these conditions.

To represent the change in a clock signal, we need some way to record or recognize that an event (corresponding to the change in clock value) has occurred. To do this, we consider the event attribute of the clock signal, given by clk'event .

Unfortunately, all that the clk'event attribute tells us is that we have encountered an edge in the clock signal, and not whether it was a rising or falling edge. To further specify the type of edge encountered, we also specify the value of the clock after the edge has "completed". Thus, a value of clk='1' would indicate that a rising edge had just occurred, and a value of clk='0' would indicate that a falling edge had just occurred.

Figure 2 shows the clock-sensitive circuit of Figure 1 re-written to specify a negative-edge triggered circuit.

```
ARCHITECTURE cct OF testing IS BEGIN
    process (clk) begin
        if (clk event and clk= 0 ) then
            ...
        end if; end process;
    END cct;
```

Figure 6.2: Code Fragment: A Clocked, Negative-Edge Triggered Circuit

A typical digital circuit usually have more than one clock in a circuit. These clocks are generated using the standard clock available in the circuit. The Altera DE2-115 FPGA board includes one oscillator that produces 50 MHz clock signal. A clock buffer is used to distribute 50 MHz clock signal with low jitter to FPGA. The distributing clock signals are connected to the FPGA that are used for clocking the user logic. A typical clock waveform is shown in Figure 3

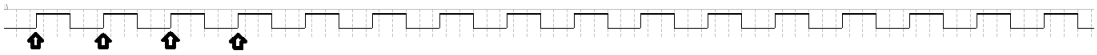


Figure 6.3: Clock waveform

The small arrows in Figure 3 indicates the positive edge of original clock. To generate slow clocks from original clock, we will use the analogy of a counter. At every rising-edge, the counter will increment, see the code in Figure 4.

```
library ieee;
use ieee.std_logic_1164.all;

entity slowclocks is port(
    clk: in std_logic;
    sclk: out std_logic
); end slowclocks;

architecture behv of slowclocks is
    signal count: std_logic_vector(31 downto 0) := X"00000000";

begin

    process(clk)

    begin

        if (clk'event and clk='1') then
            count <= count + '1';
        end if;

    end process;

    sclk <= count(20);

end behv;
```

count chal raha hum nay apni marzi k count ko clk me dal dain gay

-- count high > clk slow
-- count low > clk fasat

Figure 6.4: Code for generating slow clocks

The key thing to note here is that the bit count (1) is half of the input clock frequency (25 MHz), count (2) will be 12.5MHz and so on. If we want to generate a fast clock then the bit value of count should be closer to zero and for slow clocks, it should be closer to bit number 31.

In-Lab Tasks

Lab Task 1: LED blinking

For our first lab task, you need to design a system that blinks 4 LEDs depending upon 3 available frequencies. We will use 3 slide switches to control the blinking frequency of LEDs. LEDs will blink according to the table given below

Selection Switches			Blinking frequency of LEDs
SW2	SW1	SW0	toggles cycles = 2^n+1
OFF	OFF	OFF	Sclk <= count (20) 2^21 = 2,097,152 cycles
ON	X	X	Sclk <= count(31)
OFF	ON	X	Sclk <= count(15)
OFF	OFF	ON	Sclk <= count(2) 2^3 = 8 cycles

low latency → slow

high latency → fast

Please remember that all LEDs will turn ON and OFF together. Clk signal in our entity will use the clock available on our FPGA board, the pin number is PIN_Y2

b) Reset Signal

Resets are a generally very useful thing. A reset signal in a circuit can be used to restore initial conditions, such as resetting a counter to its initial count value. The problem with resets is that they are generally asynchronous signals. How do we work a reset into asynchronous circuit?

This is actually quite easy if we remember a couple of things about VHDL. The first is that we can specify in the process sensitivity a list of all signals that affect the circuit outputs. So, we can specify a sensitivity list that includes both the (synchronous) clock signal and the (asynchronous) reset signal.

The second thing that we must remember that in the specification of a sequential statement such as an if-then-else statement, there is an order of preference that is followed. The first conditions that are encountered are of higher precedence, even if subsequent conditions are also true. So, as long as the reset conditions are tested first, we should be okay.

Figure 5 shows how to incorporate an asynchronous reset signal into a synchronous circuit.

```
ARCHITECTURE cct OF testing IS BEGIN
    process (clk, reset)
    begin
        if (reset = 1 ) then
            -- do (asynch) reset actions
        elsif rising_edge(clk) then
            -- do (synch) rising edge clock triggered actions
        end if;
    end process;
END cct;
```

Figure 6.5: Code Fragment: Asynchronous Resets in Synchronous Circuits

The sensitivity list tells us that the circuit outputs are sensitive to changes in the ***clk*** signal and the ***reset*** signal. The first condition included in the if statement concerns the reset signal; if this (asynchronous) signal is set, then the reset actions will be executed. If the (asynchronous) reset signal is not set, and the rising edge of the clock has occurred (note that rising edge == (clk'event and clk='1')) then the synchronous rising edge triggered actions occur.

Lab Task 2: Implementation of an LED wheel-of-fortune using a package/procedure

The LED wheel of fortune you are designing works as follows.

At the startup, the LED lights continuously rotate around using a bit pattern "00000001", and at the same time the seven-segment display (SSD) digits continuously increments the four-digit hex number (16 bit) at a fast speed. The digit starts from 0000 and increments to FFFF and repeats. When a user presses a button, the LED rotation stops, and the SSD displays the number caught at the positive transition of the button press. When the user presses the button again, it starts the whole process again, i.e., the LED lights rotates around, and SSD increments the four-digit hex number at a fast speed. The LED movements should be slow enough to be visible. However, the numbers on the SSD should be fast enough so that it is hard to recognize except when it is stopped.

In order to accomplish this, several things must happen.

1) Incorporate a reset signal in the design
2) Make a procedure/function for hex to ssd that can display 0-9 plus A-F on 7-segment display.
3) You will need to generate a fast clock ($1/60^{th}$ seconds) for SSDs and slow clock for LEDs
4) Rotate bits and display the bits on LED. Synchronize the LED pattern to a slow clock, so that the rotation would be visible. Rotating LED can be done using concatenation.
5) Write a push button code to detect positive transition and to take a proper action.
6) Please note that this design can be easily turned into a stopwatch or a timer.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date:_____

LAB # 7:
Follow the steps to reproduce the Fetch module of MIPS 32-bit microprocessor using VHDL and implementation on FPGA

Objective

- To design and implement the Fetch module of a single-cycle MIPS 32-bit processor

Pre-Lab

Fetch Module (fetch.vhd)

Program Counter ***(PC)*** provides the address of an instruction to be fetched from the instruction memory. For simplicity, we will implement the instruction memory using an array of 32-bit words, which means that your PC will use word addresses, instead of byte addresses as in a real MIPS processor. For this lab, we will limit the size of the instruction memory to 16 words, which would result in using only the least significant 4-bits of the address bus.

The address of an instruction memory word is selected based on three options:

1) Normal PC+1,
2) Branch address, or
3) Jump address.

To incorporate the three options of the PC choices, the fetch module includes branch and jump addresses as a part of its inputs, as illustrated in Figure 7.1. The jump address is supplied by the decoding unit, while the branch address is supplied by the execution unit, which would be implemented in the future labs. The ***PC_out*** output is determined based on two input signals; that is, ***branch_decision*** and ***jump_decision***. If both of these signals are zero, normal PC+1 is sent to ***PC_out***. Otherwise, ***PC_out*** would be the ***branch*** or ***jump target address*** plus one.

The ***PC_out*** is later used by the execution unit (execute.vhd) to compute the branch address. Since ***branch_addr*** is computed from the execute module, it is not a relative address but an absolute address. For this lab, it is only used to test and confirm instruction fetch operations.

We also need a ***Reset*** signal that should set the PC to 0, so the computer can perform initialization of the system. In order to fetch one instruction per clock, a ***clock*** signal (generated by a button) is used as one of the inputs of the fetch module.

Summarizing these needs, a block diagram of the fetch module is shown in Figure 7-1 which includes all required inputs and outputs.

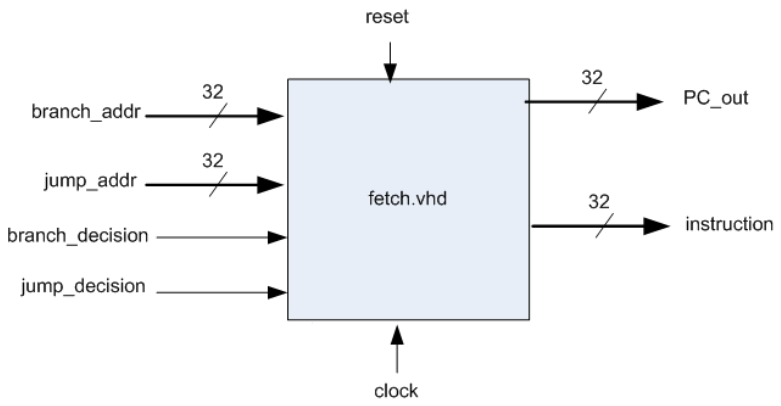


Figure 7-9: Entity for Fetch Module

For implementation, the PC is implemented as an internal variable. The internal ***PC*** is normally incremented by one and then outputted to ***PC_out***, i.e., ***PC_out <= PC+1***. For branch and jump instructions, the ***branch_addr*** or ***jump_addr*** is loaded to ***PC*** when ***jump_decision=1*** or ***branch_decision=1***, respectively. The instruction fetched from the PC location is sent out to the instruction output, while PC+1 is sent to the ***PC_out*** for the next instruction fetch.

Pre-Lab Task

Task 1: Entity of Fetch Module

The entity of fetch.vhd according to the above block diagram is:

```
entity fetch is
port (
    PC_out                                          : out STD_LOGIC_VECTOR (31 downto 0);
    instruction                                     : out STD_LOGIC_VECTOR (31 downto 0);
    branch_addr, jump_addr                          : in STD_LOGIC_VECTOR (31 downto 0);
    branch_decision, jump_decision, reset, clock    : in std_logic
);
end fetch;
```

Task 2: Writing the Fetch Module

```
architecture bhv of fetch is
--instruction memory is created as an array of 32bits and has 16 locations
type mem_array is array(0 to 15) of std_logic_vector(31 downto 0);16 rows X 31 columns
begin
process
variable mem: mem_array :=  ( --initialize the instruction memory
            X"8c220000",        --L:    lw $2, 0($1)     -- make your own machine codes here
            X"8c640001",        --      lw $4, 1($3)        -- to check the PC changes
            X"00822022",        --      sub $4, $4, $3
            X"ac640000",        --      sw $4, 0($3)
            X"1022fffa",        --      beq $1, $2, L
            X"00612064",        --      and $4, $3, $1
            X"08000000",        --      j L
            X"00000000".
            …
            );                  --
variable PC : std_logic_vector(31 downto 0);        -- PC is defined here
variable index : integer := 0;
begin
--- fetch process code begins here.
--- below describes the logic, not the code. It is your responsibility to code it.
--- wait until start of a cycle, i.e., wait until (clock'event and clock='1')
--- if reset = '1' then
--- set the PC to zero, i.e., PC := x"00000000";
--- set instruction to zero, i.e., instruction <= x"00000000"
--- this takes care of the reset
--- Check if PC should be normal increment, branch, or jump address
--- if (branch_decision = '1') then PC := branch_addr;
--- if (jump_decision ='1') then PC := jump_addr
--- Since we are using only four least significant bits,
index := to_integer(PC(3 downto 0));
--- Increment the PC by one, i.e.,
PC := PC + X"1";
remember that your PC uses word address in our design
-- Please remember that variables retain their values until the next time this process is executed.
--- At the end of the process, you need to output the results.
PC_out <= PC;
instruction <= mem(index); -- Output the fetched instruction
end process;
end bhv;
```

Note from the sample MIPS assembly code that the registers are now expressed simply $0, $1, ..$7. For simplicity, we will only implement eight registers, and the registers are simply expressed as $0 - $7. This syntax is accepted by the Mars assembler. Therefore, you can still use the Mars assembler to generate the machine codes for this MIPS implementation, except that the PC values have to be modified to word addresses.

In-Lab Tasks

You may start implementing the **fetch.vhd** as the top module to debug it, but it must be turned to a module later to test interfaces. Once debugging is completed, an interface test module must be written as a top module to test and verify the operations of the **fetch.vhd**.

Lab Task 1: Write codes to test the functionality

- Design a wrapper file that calls (port map) fetch module created in pre-lab tasks.
- Set 8 seven segments to display the data (we will call it display unit).

- Connect ***branch_decision, jump_decision***, and ***reset*** to slide switches for testing.
- Connect the ***clock*** signal to one of the push buttons
- Depending on the conditions, show the respective PC value and instruction on display unit
- Test and verify whether ***PC_out*** increments by one for normal (i.e., ***branch_decision=0*** and ***jump_decision=0***) instructions when a clock period is applied. Verify if the correct instruction is fetched to the instruction output, and if **PC** retains the correct value.
- Show the operations of branch and jump. If ***branch_decision='1'*** or ***jump_decision='1'***, the ***PC*** display (LEDs) should show the correct address changes. Later, the control unit should be designed.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date: _____

LAB # 8:

Follow the steps to reproduce the Decode module of MIPS 32-bit Microprocessor using VHDL and implementation on FPGA

Objective

- To design the decode module of a single-cycle MIPS 32-bit processor

Pre-Lab

In previous lab, you designed the fetch module (fetch.vhd) and tested PC changes and instruction fetches. In this lab, you will design the decode module that decodes the fetched instruction. The decode module is marked using a blue circle in Figure 9.1. The objective is to design the decode.vhd and test it along the fetch.vhd.

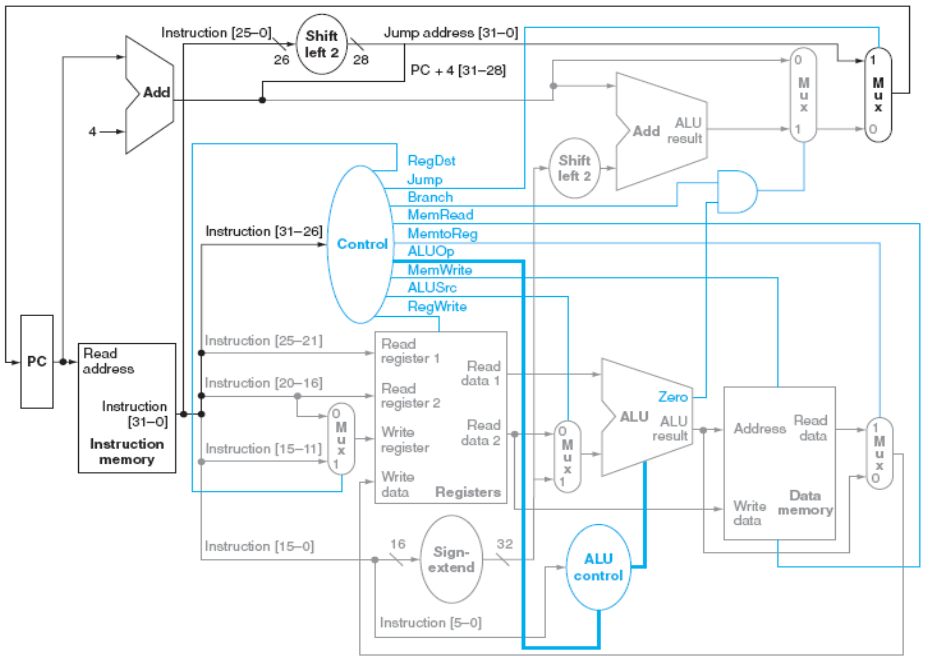


Figure 8-1: Single Cycle MIPS Datapath

Decode module (decode.vhd)

The decode module in the overall data path diagram is shown in a blue circle in Figure 8-1. The functional block diagram of the decode module is depicted in Figure 8-2. In this design, the registers will be placed inside the decode module. Note from the diagram that the decode module includes output signals denoted ***register_rs*** and ***register_rt***. These are two register values read from the register file pointed by the rs and rt fields of the instruction, which will be sent to ALU. It also includes ***jump_addr*** and immediate signals that are directly decoded from the instruction. The job

of decode module is then to decode the register addresses and immediate values from the fetched instruction and then to send out the values to respective output signals. Since the registers reside inside the decode module, the values of destination register, ***register_rd***, is not allocated as the output.

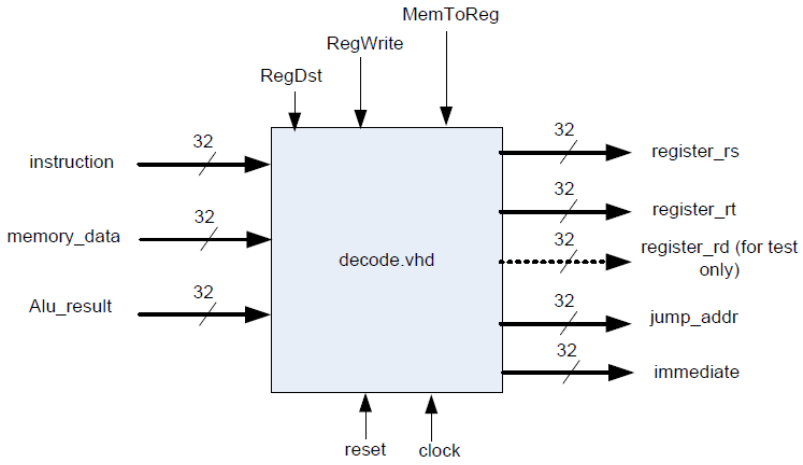


Figure 8-2: Decode Module

- There are two ways of implementing the internal registers. The first approach is to use an array of 32-bit values, similarly to the memory implementation.

  type reg_array is array(0 to 31) of std_logic_vector(31 downto 0);

  shared variable RegFile: reg_array;

  The second approach is to implement the register file as signals, i.e.,

  signal reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7 : std_logic_vector(31 downto 0);

  The second approach makes the code longer and looks not efficient. However, if we look at in a hardware point of view, it should actually run faster since it does not require decoding of the array index. For this lab, either approach should work fine, but we will limit the number of registers to only eight (8). Real MIPS has 32 registers.

  It is recommended to use first approach as it is easier and simpler to extend and test.

- The next to consider are the internal multiplexers. Internally there are two multiplexers. The first multiplexer determines the source of the destination register address. For R-type instructions, the destination register is specified in the bits of Instruction[15:11]. However, for the load and store instructions the destination register is specified in Instruction[20:16].

The signal ***RegDst*** controls this choice. Figure 9.3 illustrates implementation of theses relations using a multiplexer.

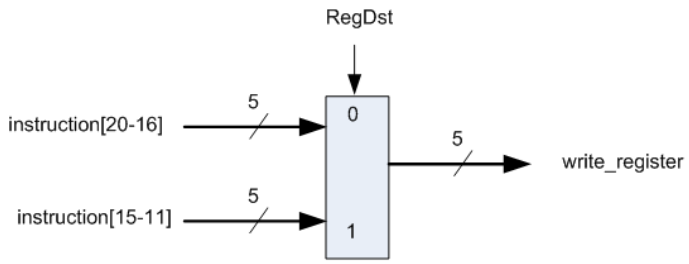


Figure 8-3: Destination register selector multiplexer

- The second multiplexer determines whether the data to be stored at the destination register is coming from ***Alu_out*** or ***memory_data*** (***read_data*** in Figure 8-1). This relationship is shown in Figure 8-4

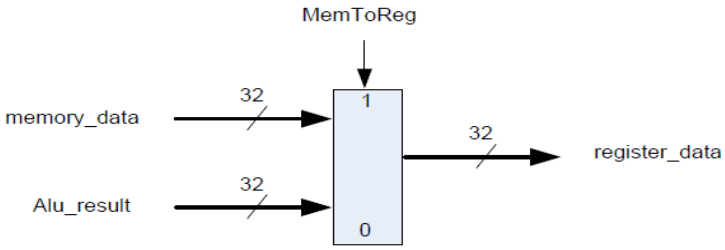


Figure 8-4: Multiplexer that selects the source of data to be sent to the destination register

- Another output must be taken care of is the immediate output. Immediate data is always specified by the bits in Instruction[15:0], but it has to be sign extended to 32 bit. More specifically, Instruction(15) determines the sign and can be coded as:

```
immediate(15 downto 0) <= instruction(15 downto 0);
if instruction(15) = '1' then
    immediate (31 downto 16) <= x"ffff";
else
    immediate (31 downto 16) <= x"0000";
end if;
```

- The output ***jump_addr*** is determined from Instruction[25:0]. Since we are using word addresses, concatenation of "00" is not required. We also do not use jump based on the current PC but directly. It should be simply,

- ```
  jump_addr(31 downto 0) <= "000000" & instruction(25 downto 0);
  ```

- Another consideration to simplify the overall program is to create two processes: one for writing registers and another for reading registers. This approach simplifies the overall implementation.

- Last, but not least important aspect is that ***reg0*** (or register(0)) must have the constant value 0 because it is called the ***$zero*** register and the value of this register must always be 0.

In-Lab Tasks

Lab Task 1: Entity of Decode Module

The entity of ***decode.vhd*** according to the above block diagram is:

```
entity decode is
    port (  instruction : in STD_LOGIC_VECTOR (31 downto 0);
    …
    ….
    immediate : out STD_LOGIC_VECTOR (31 downto 0);
    reset : in STD_LOGIC);
    end decode;
```

Lab Task 2: Writing the Decode Module

```
architecture Behavioral of decode is
type reg_array is array(0 to 31) of std_logic_vector(31 downto 0);
shared variable RegFile: reg_array :=   ( --initialize the Register File
            X"00000000",        -- Register ZERO
            X"11111111",        --  Register One
            X"22222222",        --
            X"33333333",        --
            X"44444444",        --
        others =>
            X"01010101");

begin
```
```

```
    -- Process to write the register file when required
    ------------------------------------------------------------------
    reg_write: process(reset, memory_data, alu_result, clock)
    variable write_value : std_logic_vector (31 downto 0);
    --variable addr1, addr2, addr3 : std_logic_vector(4 downto 0);
    variable index : integer := 0;
    begin
    -- on reset initialize the registers
    if reset = '1' then
    RegFile :=  (                   --  Reset the Register File
                X"00000000",        --  Register ZERO
                X"11111111",        --  Register One
                X"22222222",        --
                X"33333333",        --
                X"44444444",        --
              others =>
                X"01010101");
    else
    --- determine the address of the register to be written , use Figure 8-3 to implement
        if RegDst = '0' then
            --value should be written to register number availble at instruction(20 downto 16)
        else
            --value should be written to register number availble at instruction instruction(15 downto 11)
        end if;
--Remember for Approach 1, we need to have the integer index to access our Register File, use conv_integer
--to convert std logic vector to integer value

--- Determine the source operand to be written, i.e., memory or alu result
if RegWrite = '1' then
--- if MemToReg = '1' then
--- write_value := memory_data;
---else
--- write_value := alu_result;
---end if;

--- Store write_value to the destination register (need to disable this when you test other
functions)


   -- Process to read register file and pass the operands to the execute module
   ------------------------------------------------------------
   reg_read: process(instruction)
   variable rt, rs, imm, rd : std_logic_vector(31 downto 0);
   variable addr1, addr2 : integer := 0;
   variable addr_rd : integer := 0;
   begin
   -- register addresses for reading the registers
-- addr1 will contain the index for Register File to read register rs, availabe at instruction(25 downto 21)
-- addr2 will contain the index for Register File to read register rt, availabe at instruction(20 downto 16));
-- addr_rd will contain the index for Register File to read register rd, availabe at instruction(15 downto 11));
    ---read the register
        rs := RegFile(addr1);
        rt := RegFile(addr2);
        rd := RegFile(addr_rd);

   --- access immediate from instruction and perform sign extension
   imm(15 downto 0) := instruction(15 downto 0);
   if instruction(15) = '1' then
       imm(31 downto 16) := x"ffff";
   else
       imm(31 downto 16) := x"0000";
   end if;
--compute the jump address.
jump_addr(31 downto 0) <= …..;
--- bring out signals to the ports of the module

--- register_rs <= rs ;
--- register_rt <= rt ;
--- immediate <= imm ;
--- you may add rd as well for testing
end process reg_read;
end Behavioral;
```

Please understand that above codes are only provided as a description or pseudo code, not the actual code.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date: _____

LAB # 9:
Follow the steps to test the Decode module of MIPS 32-bit Microprocessor using VHDL and implementation on FPGA

Objective

- To test the decode module that decodes the fetched instruction of a single-cycle MIPS 32-bit processor

Introduction

Testing the decode module is a lot more complicated than the previous module. There are two ways that could be done. The first is to test the fetch and decode modules independently by supplying appropriate signals to the input signals. The second method is to test two modules by connecting the decode module to the fetch module. You will use the second approach to ensure that your fetch module works correctly.

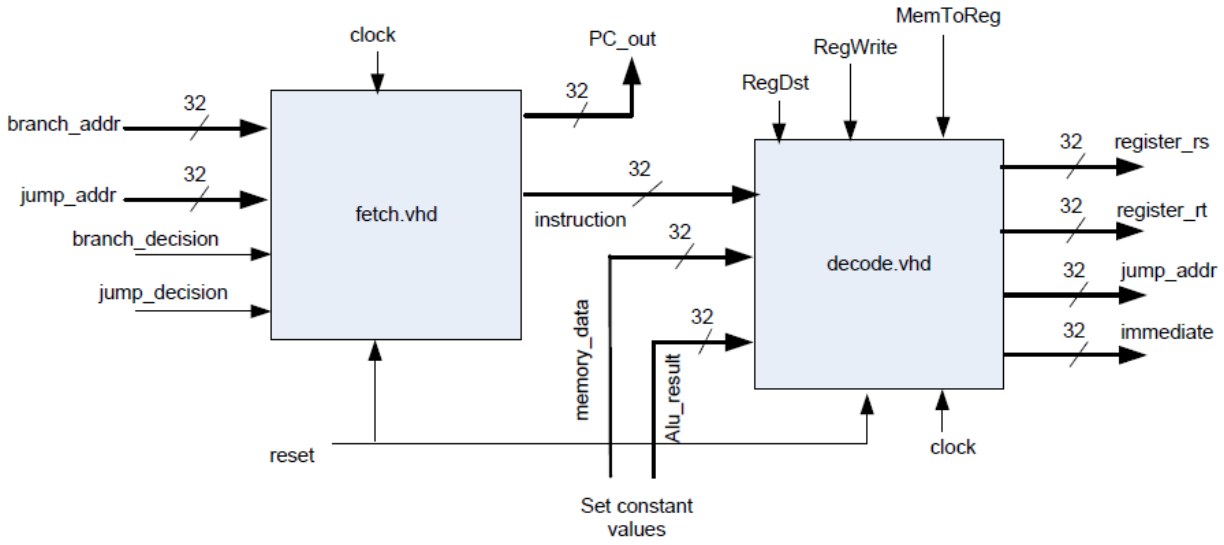
In-Lab Tasks

Lab Task 1: To test the functionality follow the steps

1) **Modify the codes in the fetch module for testing *(You may try different codes)***

```
--initialize the instruction memory
variable mem: mem_array := (
X"8c220000",            --L: lw $2, 0($1) ; $2 <= mem[0+$1]
X"8c640001",            -- lw $4, 1($3) ; $4 <= mem[1+$3]
X"00622022",            -- sub $4, $3, $2 ; $4 <= $3 - $2
X"ac640000",            -- sw $4, 0($3) ; mem[0+$3] <=$4
X"1022fffb",            -- beq $1, $2, L ; if ($1=$2), branch_addr<=L
X"00612064",            -- and $4, $3, $1 ; $4 <= $3 and $4
X"08000000",            -- j L ; jump_addr <= L
X"00000000"); --
```

2) **Make a port map with the *fetch.vhd* and *decode.vhd* module**



3) **Initialize the registers in *decode.vhd* as follows**

```
-- on reset initialize the registers
if reset = '1' then
RegFile :=  (                       --  Reset the Register File
            X"00000000",            --  Register ZERO
            X"11111111",            --  Register One
            X"22222222",            --
            X"33333333",            --
            X"44444444",            --
            X"55555555",            --
            X"66666666",            --
            X"77777777",
            X"88888888",
            X"AAAAAAAA",
            X"BBBBBBBB",
            X"CCCCCCCC",
            x"DDDDDDDD",
            x"EEEEEEEE",
            X"FFFFFFFF",
        others =>
            X"01010101");
```

4) **Display and switch setup**

Note that there are four 32 bit outputs that must be verified. Use the four switches to control the selection of the outputs, i.e.

Switch status	To Display
0001	Contents of register_rs
0010	Contents of register_rt
0100	Jump Address
1000	32 bit Immediate value
Otherwise	Instruction

5) **RegDst, RegWrite, MemToReg control**

At the top test module, RegDst, RegWrite and MemToReg signals are generated according to the instruction op code. It follows the following table. The complete version of this table can be found from the textbook

	R-type	lw	sw	beq
RegDst	1	0	x	x
RegWrite	1	1	0	0
MemToReg	0	1	x	x

```
--For the jump instruction,
if (opcode = jump) then
    jump_decision< = '1';
else
    jump_decision< = '0';
end if;
```

6) **Instruction-by-instruction Verification**

Consider the first instruction:

```
X"8c220000", --L: lw $2, 0($1) ; $2 <= mem[0+$1]
```

In this case, the source register is $1, the target register is $2, and the immediate is 0. Hence, you should see assuming you initialize the registers as in Step 3.

register_rs = x"11111111"
register_rt = x"22222222" -- disable storing to see the target reg value
immediate = x"00000000"
jump_addr = don't care

Consider another example which is an R-type instruction:

```
X"00622022", -- sub $4, $3, $2 ; $4 <= $3 - $2
```

In this case, since the sub instruction has a format “sub rd, rs, rt”, you should expect to see the following:

register_rs = x”33333333”
register_rt = x”22222222”
immediate = don’t care
jump_addr = don’t care

Similarly, ***beq*** and ***j*** instructions can be verified. Create a list of expected output values of the decode module for each instruction and verify. Unfortunately, writing to registers cannot be tested easily unless we bring out the destination registers as another test output. One way of easy testing is to modify the reg_write process to replace the register_rt output with the register_rd signal and then you can use the same testing method as above to verify the decoding. Another way is to add one more interface signal register_rd as a part of the port and display it on the terminal. But you should remember that this interface has to be removed later because MIPS does not send this signal outside the decode module.

This lab is probably the most complex part of the MIPS-32 design. After this step is completed, the subsequent labs for MIPS-32 implementation will become easier.

7) **Demo**

Show the four outputs of the decode module using buttons, switches, and the Hyper terminal for each type of instruction.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date: _____

LAB # 10:

Follow the procedure to reproduce the Control unit and Data Memory of MIPS 32-bit microprocessor using VHDL and implementation on FPGA

Objective

- To design the Control and Data memory module of a single-cycle MIPS 32-bit processor

Pre-Lab

In this lab you will be designing the control unit and data memory of MIPS-32 processor. The control module and its control signals are shown in light blue color.

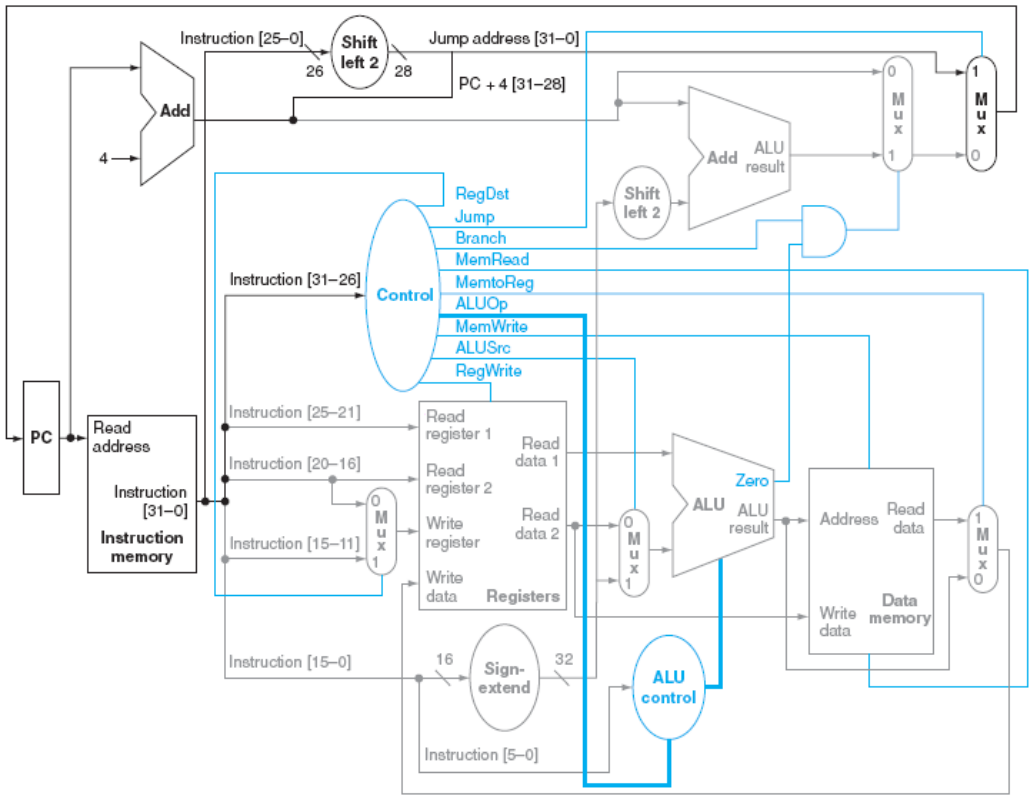


Figure 10-1 Single cycle MIPS datapath

The control module (control.vhd)

The control module simply generates the control signals based on the instruction op code.

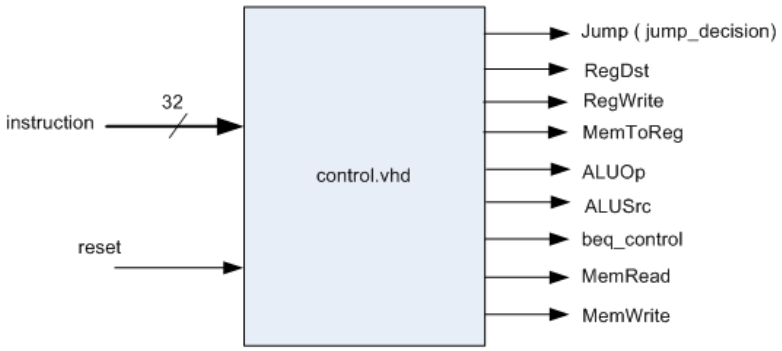


Figure 10.2 The control module

The implementation of the control module basically follows the table given in the textbook. It is relatively simple to implement this module.

Pre-Lab Task

Task 1: Control Module Functionality

To implement control unit, please refer to Lab 2 of the manual, which teaches who to implement a truth table. Below gives a partially implemented example:

```
process (instruction, reset)
    variable rformat, lw, sw, beq, jmp, addi : std_logic;   -- 1 if the instruction is identified
    variable opcode : std_logic_vector(5 downto 0);
begin
if reset = '1' then
    RegDst <= '0';
    ALUSrc <= '0';
    MemToReg <= '0';
    RegWrite <= '0';
    MemRead <= '0';
    MemWrite <= '0';
    ALUOp <= "00";
    Jump <= '0';
    beq_control <= '0';
else
    opcode(5 downto 0) := instruction(31 downto 26);
```

```
else
    opcode := instruction(31 downto 26);

case opcode is
    when "000000" =>          --RType
        jump            <='0';
        RegDst      <='1';
        RegWrite        <='1';
        MemToReg        <='0';
        ALUOp           <="10";
        ALUSrc      <='0';
        beq_control <='0';
        MemRead     <='0';
        MemWrite        <='0';
    when "100011"   =>  --Load Word lw
```

Figure 10.3 shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion.

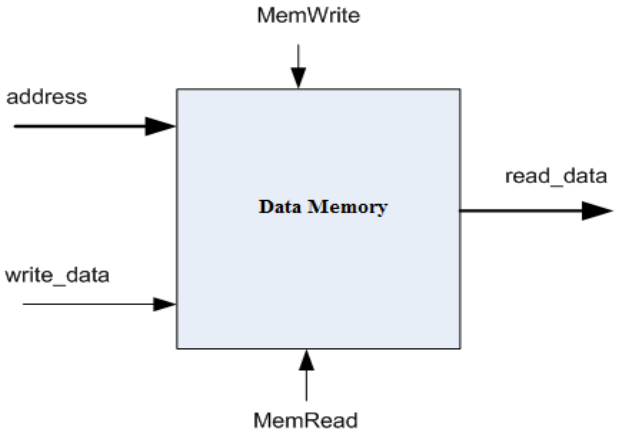
Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figure 10.3 Control function for the simplified function

For pre-lab task 1, implement the table given in figure 10.3.

Task 2: Data Memory module

Data memory module implementation is nearly identical to the instruction memory. Please refer to your previous program. The block diagram is shown in Figure 10.4 and implemented using an array of 32bits



10.4 Data Memory Module

In-Lab Tasks

Testing will be done by connecting all of the modules you have constructed. Your understanding on the data path is important for debugging

Lab Task 1: Make a port map and test the Control and Memory Unit

Make module to module connections

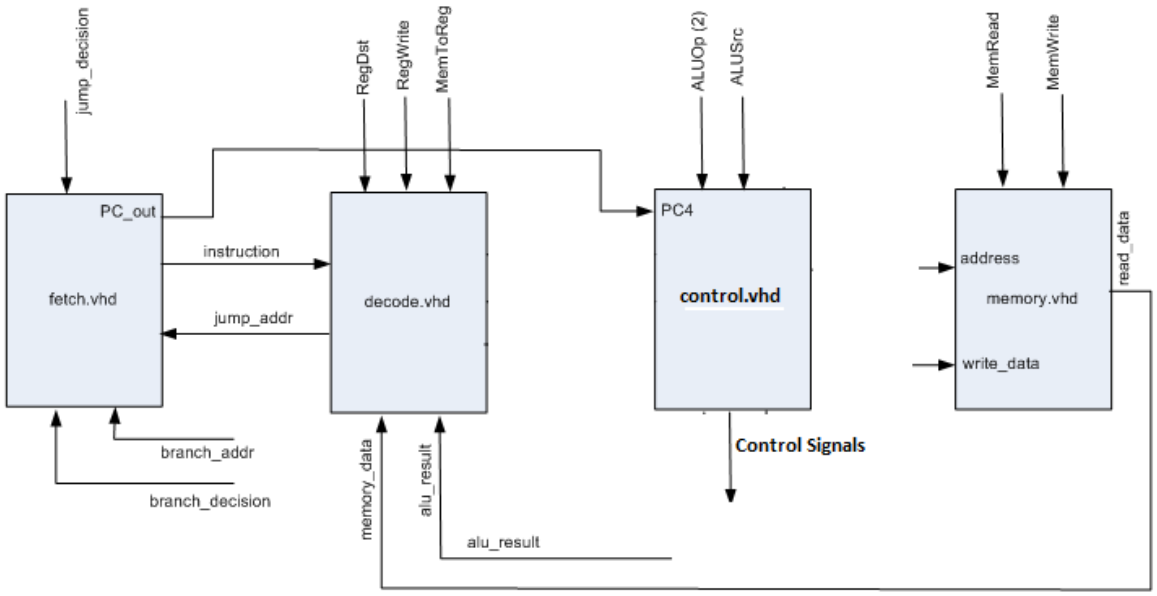


Figure 10.5 Control and Memory Unit

Create a top module file by port mapping all the modules.

Lab Task 2: Implement the Data Memory module

```
entity memory is
generic (module_delay : time :=10 ns);
port(
    address             : in std_logic_vector(31 downto 0);
    write_data          : in std_logic_vector(31 downto 0);
    MemWrite,MemRead    : in std_logic;
    read_data           : out std_logic_vector(31 downto 0)

); end memory;
architecture behavioral of memory is
    Type mem_array is array(0 to 7) of std_logic_vector(31 downto 0);
shared variable data_mem : mem_array :=(
            X"00000001",
            X"00000001",
            X"00000002",
            X"00000003",
            X"00000004",
            X"00000005",
            X"00000006",
            X"00000007");

begin
    Memory_Write: process (address,MemWrite)
    begin
        if MemWrite ='1' then
            data_mem(conv_integer(address(2 downto 0))) := write_data;
        end if;

    end process;

    Memory_read: process (address,MemRead)
    begin
        if MemRead ='1' then
            read_data <= data_mem(conv_integer(address(2 downto 0))) after module_delay;
        end if;

    end process;

            addr := conv_integer(address(2 downto 0)); --since there are only 8 words
            mem_content := write_data;
            if MemWrite = '1' then
                data_mem(addr) := mem_content;
            elsif MemRead = '1' then
                mem_content := data_mem(addr);
            read_data <= mem_content after module_delay;
            end if;
        end process ReadWrite1;
end Behavioral;
```

Lab Task 3: Demo

Show the control module outputs on FPGA LEDs according to the following table

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ Date: _____

LAB # 11:

Follow the steps to reproduce the Execution unit of MIPS 32-bit Microprocessor using VHDL and implementation on FPGA

Objective

- To design the Execution module of a single-cycle MIPS 32-bit processor

Pre-Lab

In this lab you will complete the MIPS-32 processor design by adding the final module: the execution unit. The execution module is marked using a red polygon in Figure 11.1.

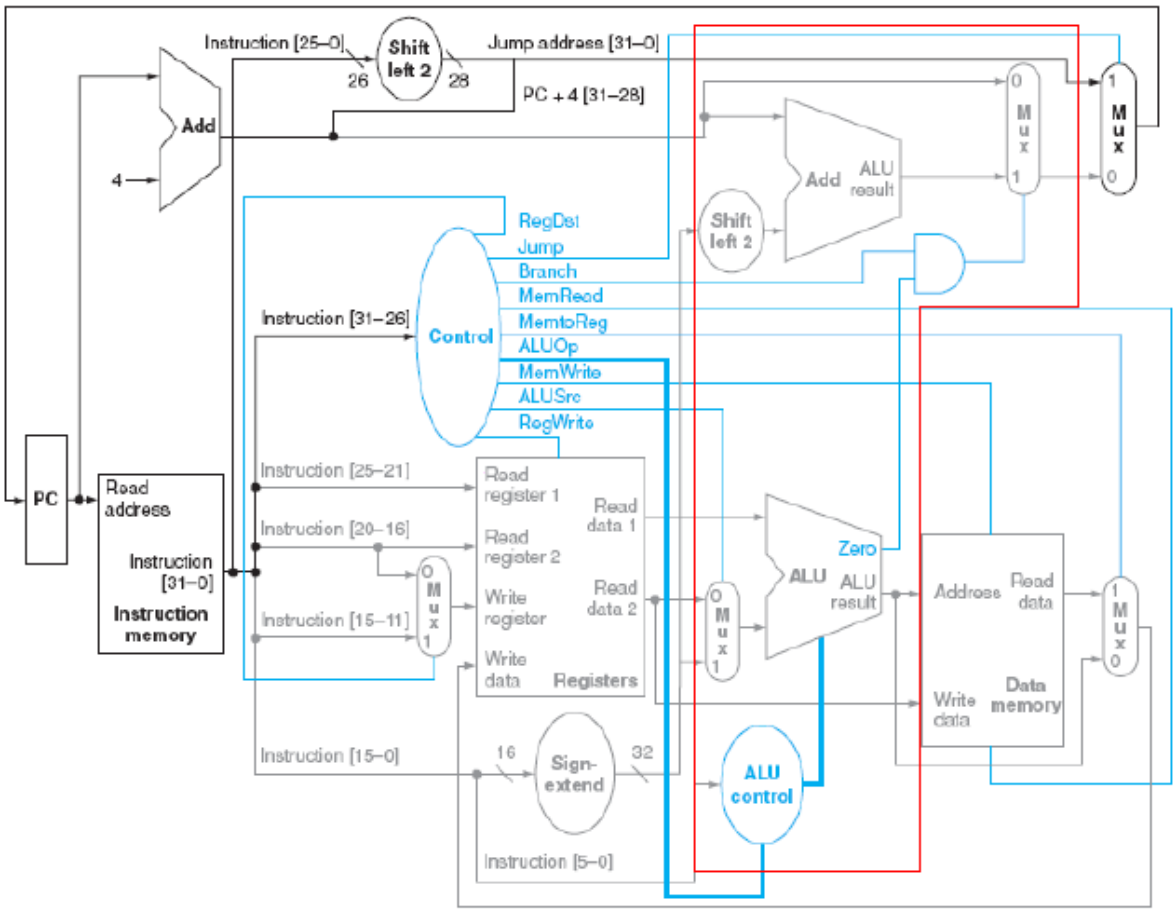


Figure 11.1 Execution Unit

The execution module (execute.vhd)

The execution module takes its inputs from the decode module and performs ALU operations. It consists of six inputs which are

1. register_rs (32bits)
2. register_rt (32bits)
3. immediate (32bits)
4. ALUOp (2 bits)
5. ALUSrc (1bit)
6. beq_control (1 bit).

These inputs produce three outputs:

1. alu_result (32 bits)
2. branch_addr (32 bits)
3. branch_decision (1 bit).

The block diagram is shown in Figure 11.2

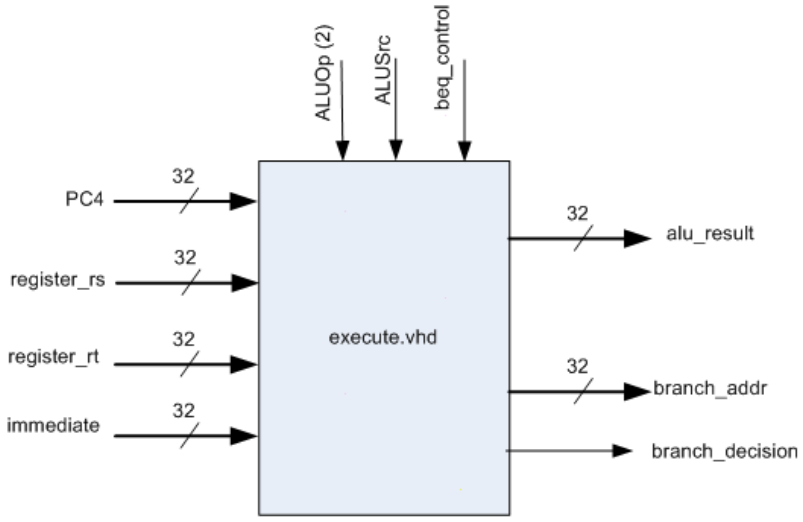


Figure 11.2 The execute module

For the design, understanding of the ALU control is imperative. Please refer to the textbook. Assume that you defined two temporary local variables ***alu_output***(32bit) and zero (1 bit). The zero variable is later passed to ***branch_decision*** output. The function of ALU is controlled by two bits of ALUOp as summarized below.

```
When ALUOp = "00"
--This is a memory instruction, so memory address is computed and set to the alu_output,
--i.e.,
--      alu_output := register_rs + immediate
-- In addition, zero must be set according to the condition.
When ALUOp = "01"
-- This is a branch instruction.
-- alu_output := register_rs - register_rt
-- The result of this operation, that is whether it is 0 or not,
-- must set the variable named zero to 0 or 1.
When ALUOp = "10"
-- This is an r-type instruction, so the operation is determined by the 6 least
-- significant bits in the instruction. A couple examples are:
case immediate(5 downto 0) is
    when "100000" => --add
        alu_output := register_rs + register_rt;
    when "100010" => -- subtract
        alu_output := register_rs - register_rt.
--      and so on.

--      and so on.
-- For undefined or unimplemented instructions (when others), write the error indications as,
        alu_output := x"ffffffff";
-- The branch address is computed using:
        branch_offset := immediate;
-- recall that we are using a word address, so no need for shift by 2
        temp_branch_addr := PC4 + branch_offset;
-- The final synchronized outputs are then given as:
        branch_decision <= (beq_control and zero);
        branch_addr <= temp_branch_addr;
        alu_result <= alu_output;
--The execute module is implemented as a process in a vhdl module.
```

Task 1: Entity of Execute Module

The entity of ***execute.vhd*** according to the above block diagram is:

```
entity execute is
    port( register_rs, register_rt: in std_logic_vector(31 downto 0);
          PC4, immediate: in std_logic_vector(31 downto 0);
          ALUOp: in std_logic_vector(1 downto 0);
          ALUSrc: in std_logic;
          beq_control,clock: in std_logic;
          alu_result, branch_addr: out std_logic_vector(31 downto 0);
          branch_decision: out std_logic);
end execute;
```

Task 2: Writing the Execute Module

```
architecture Behavioral of execute is
begin
    process
    variable alu_output: std_logic_vector(31 downto 0);
        variable zero: std_logic;
        variable branch_offset, temp_branch_addr: std_logic_vector(31 downto 0);
    begin
            if(clock'event and clock='1') then
                case ALUOp is
                    when "00" =>
                        --logic comes here
                    when "01" =>
                        --logic comes here
                    when "10" =>
                        --logic comes here

                end case;
                branch_offset:=immediate;
                temp_branch_addr:=PC4+branch_offset;
                branch_decision <= (beq_control and zero);
                branch_addr <= temp_branch_addr;
                alu_result <= alu_output;
            end if;
    end process;
end Behavioral;
```

In-Lab Tasks

Testing will be done by connecting all the modules you have constructed. Your understanding on the data path is important for debugging.

Lab Task 2: Make module-to-module connections as shown

For all connections, a signal should be defined. This allows testing of the module-to-module functionality to be done conveniently by connecting them to LEDs, 7-segment displays, and the PC Hyper terminal.

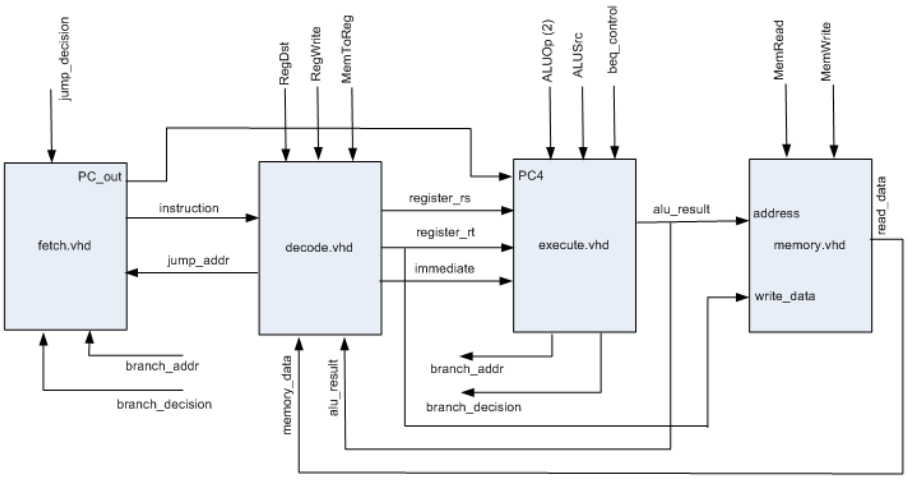


Figure 11.3 Module to Module connection

This diagram does not show the control module, neither the reset signal, to avoid the clutter. It is not shown but the 32bit instruction must be fed to the control module. All of the outputs of the control module are connected to the specified locations in the diagram.

Lab Task 3: Connections to switches, LEDs, and 7-segment

From the decode module, assign register_rd as an additional output port and modify the code so that its value can be displayed. Assign the switches as

Switch status	To Display
0001	Displays **PC_out** from fetch.vhd
0010	Displays **register_rs** from decode.vhd
0011	Displays **register_rt** from decode.vhd
0100	Displays **Immediate** from decode.vhd
0101	Displays **alu_result** from execute.vhd
0110	Displays **read_data** from memory.vhd
0111	Display **branch_address**
Otherwise	Displays **Instruction** from fetch.vhd

Lab Task 4: Examine the registers, PC, alu_result, and memory values instruction-by-instruction

```
--instruction memory
variable mem: mem_array := (
X"8c220000",          --L: lw $2, 0($1) ; $2 <= mem[0+$1]
X"8c640001",          -- lw $4, 1($3) ; $4 <= mem[1+$3]
X"00622022",          -- sub $4, $3, $2 ; $4 <= $3 - $2
X"ac640000",          -- sw $4, 0($3) ; mem[0+$3] <=$4
X"1022fffb",          -- beq $1, $2, L ; if ($1=$2), branch_addr<=L
X"00612064",          -- and $4, $3, $1 ; $4 <= $3 and $4
X"08000000",          -- j L ; jump_addr <= L
X"00000000"); --
```

Lab Task 5: Demo

- Show the registers, PC, immediate, read_data, alu_result of the following test code. This code simply tests loading of two values, add them, and store the result.
- Initialize the register and memory values as:
  reg0=0, reg1=1, reg2=2, reg3=3, reg4=4, reg5=5, reg6=6, reg7=7

  mem[0]=0, mem[1]=1, mem[2]=2, mem[3]=3, mem[4]=4, mem[5]=5, mem[6]=6, mem[7]=7

- Test the following five lines of codes and provide a demo:Show the four outputs of the decode module using buttons, switches, and the Hyperterminal for each type of instruction.

	Machine Code	**Assembly Code**	**Register, imm, ALU, and mem signals, i.e., contents**
0	8c22 0000	lw $2,0($1)	rs=1, rt=1, rd=, imm=0, ALU=1, mem=1
1	8c23 0005	lw $3,5($1)	rs=1, rt=6, rd=, imm=5, ALU=6, mem=6
2	0062 2020	add $4,$3,$2	rs=6, rt=1, rd=7, imm=2020, ALU=7, mem=6 (x)
3	ac24 0000	sw $4, 0($1)	rs=1, rt=7, rd= , imm=0, ALU=1, mem=6 (x)
4	8c22 0000	lw $2, 0($1)	rs=1, rt=7, rd= , imm=0, ALU=1, mem=7

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____

LAB # 12:
Follow the steps to reproduce the single cycle MIPS 32-bit microprocessor using VHDL

Objective

- To design the single-cycle MIPS 32-bit processor
- To test the R-Type instructions
- To test LW, SW and ADDI instructions
- To test BEQ instruction
- To test Jump instruction

Pre-Lab

Congratulations on your hard work and dedication to complete your own MIPS-32 processor implementation! Implementing a working processor is not a trivial task, but you have done it. You should be proud of yourself that you came this far, considering that you had no experience in either VHDL or processor design at the beginning of the class. In this lab you will complete the MIPS-32 processor design by adding all the modules together as shown in Figure 12.1

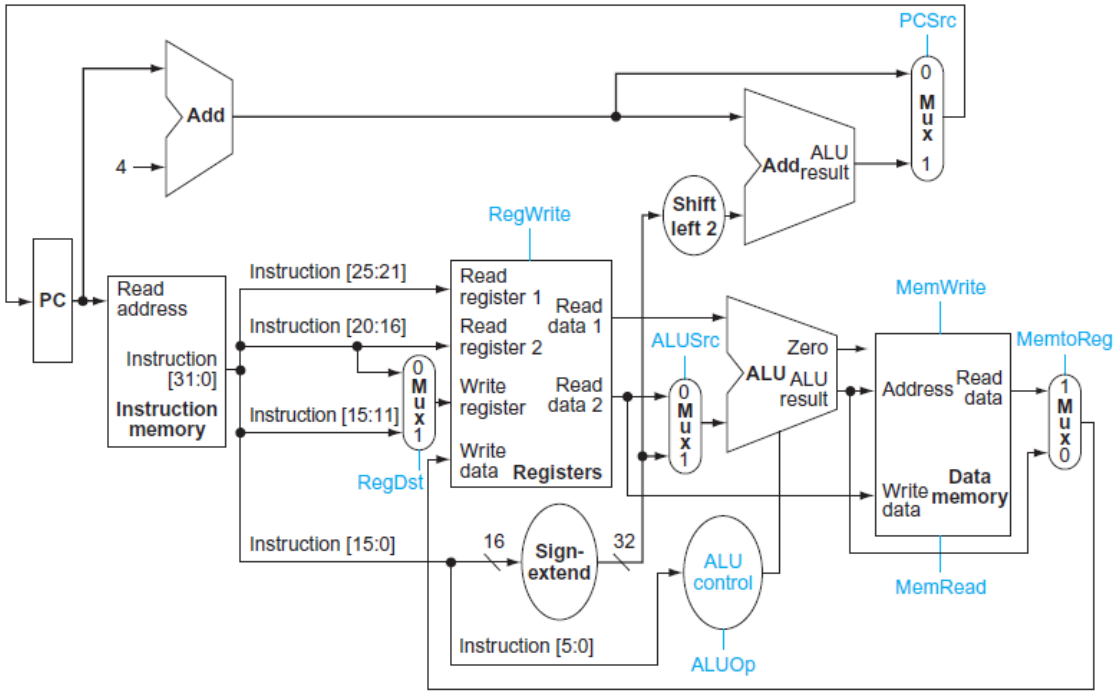


Figure 12.1 Complete MIPS microprocessor

Task 1: Make Module to Module connections

For all connections, a signal should be defined. This allows testing of the module-to-module functionality to be done conveniently by connecting them to LEDs and 7-segment displays.

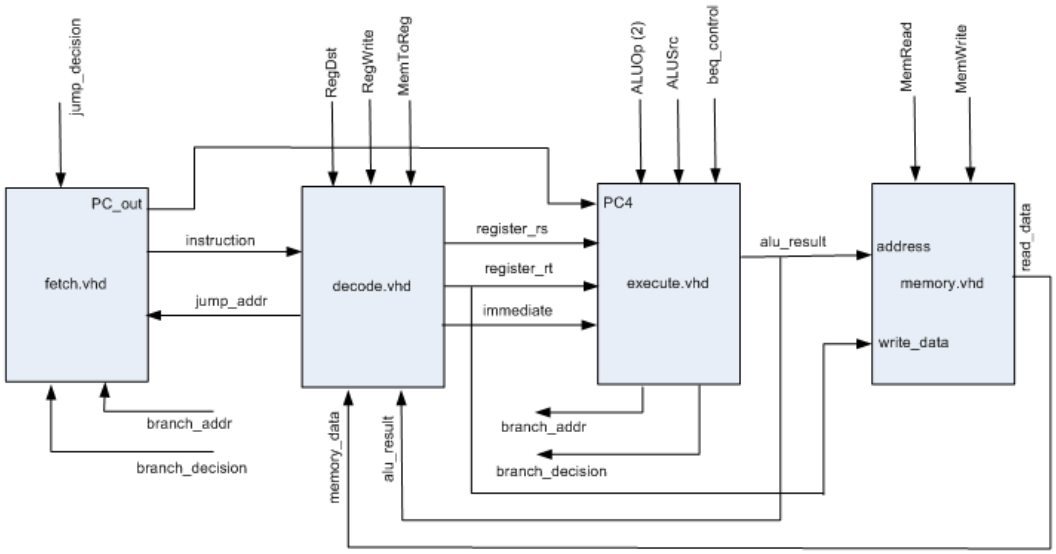


Figure 12.2 Module Connection

Task 2: Connections to switches, LEDs, and 7-segment

The Display unit will have 4 switches, eight 7-segment displays, a button (clock) and a 50MHz clock.

Switch status	To Display
0001	Displays **PC_out** from fetch.vhd
0010	Displays **register_rs** from decode.vhd
0011	Displays **register_rt** from decode.vhd
0100	Displays **Immediate** from decode.vhd
0101	Displays **alu_result** from execute.vhd
0110	Displays **read_data** from memory.vhd
0111	Display **branch_address**
Otherwise	Displays **Instruction** from fetch.vhd

In-Lab Tasks

We will be testing our MIPS 32-bit Single Cycle Microprocessor by completing the lab tasks incrementality (like we designed the Datapath). In order for the processor to work as a single cycle design, we need to incorporate some changes in our individual created modules. The single cycle ideology recommends that every instruction must complete in one clock cycle. To do that, do the following changes in already created modules

1) At every rising edge of clock, pick a new instruction from Instruction memory (fetch module)
2) The decode module has two processes, one for register read and other for register write. Register read process will be sensitive on instruction and reset only (no clock signal here), while the Register write process will write the values in registers at the end of clock signal, that is when clock'event and clock ='0'.
3) The Execute module should not be sensitive on clock, rather it should be sensitive on the values of register_rs, register_rt or immediate.
4) Memory process should also be exempted from clock sensitivity rather it should be sensitive on memory_address or memory signals.

Once the changes are incorporated, move on with Lab tasks.

Lab Task 1: Create the top-level entity

Create the wrapper file (MIPS.vhd) that has the following pins

Input

a) Clock
b) Button (will be used as clock, for testing only)
c) Reset switch

Outputs

a) Eight 7-segment displays

Lab Task 2: Test R-Type and ADDI functionality

This last lab is to test the MIPS machine codes as the final step. Update the instruction memory with the machine code of following instructions. Use green sheet to convert the assembly code into machine instructions

```
addi $1,$0,10
addi $2,$0, 20
add $3,$1,$2
sub $4,$2,$1
```

Lab Task 3: Test LW, SW commands

```
addi $1,$0,10
addi $2,$0, 20
add $3,$1,$2
sw $3,1($0)      --- Store at Data Memory location 1
lw $4,1($0)      --- Read the content of Memory location
```

Lab Task 4: Test BEQ and Jump instructions

```
      addi $1,$0,10
      addi $2,$0, 20
      add   $3,$1,$2
      beq   $1,$2,lblyes
      j lblend
lblyes:
      sub   $3,$1,$2
lblend:

      sw $3,1($0)      --- Store at Data Memory location 1
      lw $4,1($0)      --- Read the content of Memory location
```

Lab Task 5: Make module-to-module connections as shown

This last lab task is to test the MIPS machine codes as the final step. Write and test a MIPS assembly code that adds from 1 to 200 (C8hex) and then stores the result at a memory location on your own processor. This time, connect the clock to the system clock (50MHz) or a divided slower clock. In order to display the final result, use an infinitive loop of loads. This can be achieved through displaying the signal ***read_data*** to the 7-segment display output. The 7-segment display is used because it is a faster display technology.

Your assembly code should look like
MIPS code that computes 1+2+3+, …., + 200=20100=4e84hex.

```
L1: ….
beq   $1,$2, L2  --- "bne" can be implemented by beq and j combination
j     L1
L2:   [write your code here]
                      --- Assume that the addition result is in $5
      …

      sw    $5, 3($0)  --- Save the addition result at mem[3]
L3:   lw     $5, 3($0)
--- Create an infinitive loop to display the result stored in Memory
      j     L3               --- to the seven segment display.
```

Above method should allow you to test your processor's loop (branch) capability. One common mistake students make is not recognizing the instructions implemented. Remember that you only implemented lw, sw, rformat, beq, j, and addiu. Your program can use only these instructions. Another important feature you should test is the reset capability. Resetting your processor should produce the same result, that is, it should be able to repeat the execution of the addition program. This is because the processor starts from PC=0 when a reset is applied, and thus your code ends up executing the infinitive loop again.

Rubric for Lab Assessment

The student performance for the assigned task during the lab session was:			
Excellent	The student completed assigned tasks without any help from the instructor and showed the results appropriately.	4	
Good	The student completed assigned tasks with minimal help from the instructor and showed the results appropriately.	3	
Average	The student could not complete all assigned tasks and showed partial results.	2	
Worst	The student did not complete assigned tasks.	1	

Instructor Signature: _____ **Date:** _____