ET398 LAB 3

“VHDL Implementation of Concurrent Statements”

Comparator/Multiplexer/Decoder/Encoder

February 09, 2013

Tiffany Turner

““
**OBJECTIVE:**
The objectives of this lab were for us to become more adept at creating VHDL code for basic digital design devices; while demonstrating a more in-depth knowledge using the DE-1 board (Kelly, 2013).

**PROBLEM STATEMENT:**
For the third lab I was to implement VHDL code for the four attached diagrams shown below. For each of the four programs it was required that I use concurrent statements only. There were also different types of logic required for each of the four sets of code. For the comparator I was to use combinational logic, the Multiplexer requirement was to implement a when/else statement, a with/select logic statement was to be used for the Decoder and for the Encoder I was able to choose the method of my choice. The other 3 requirements for Lab 3 are listed below:

1. Use logic files and component implementation for each problem (Kelly, 2013).
2. For each problem complete a waveform analysis using Qsim in Altera (Kelly, 2013).
3. Demonstrate the implementation of all problems on the Altera DE-1 Board; demonstrating one waveform analysis of my choice (Kelly, 2013).

The four circuit diagrams are shown below:

**2-2 Bit Comparator:**

![2-bit Comparator](image)

**4:1 Multiplexer:**

![4:1 MUX](image)
DESIGN METHODOLOGY:
The first design I began writing code for was the Comparator (2-2 bit Comparator). A comparator is a combinational circuit that compares two numbers and determines their relative magnitude. The output of a comparator is usually 3 binary variables indicating: $A > B$, $A = B$, or $A < B$.

In regards to a 2-bit comparator, there are four inputs $A_1$, $A_0$ and $B_1$, $B_0$ along with the same three outputs: $A > B$ (HIGH when $A$ is greater than $B$), $A = B$ (HIGH when $A$ is equal to $B$) and $A < B$ (HIGH when $A$ is less than $B$). The truth table for this 2-2bit comparator is shown below.
In order to adequately simulate the logic of this circuit on my DE-1 board I needed to design code which fit the criteria stated above. The template of my code remained the same as in Lab 2, with only the amount of inputs and outputs changing.

```
ENTITY comparator IS
    Port (CLOCK_50 : IN   STD_LOGIC;
          SW      : IN   STD_LOGIC_VECTOR(3 DOWNTO 0);
          LEDG    : OUT  STD_LOGIC_VECTOR(2 DOWNTO 0);
          KEY     : IN   STD_LOGIC_VECTOR(3 DOWNTO 0));
END comparator;
```

Since there are 4 inputs, I made sure to call switches 3 “downto” 0. As for my outputs, there were only 3 (greater than, equal to and less than) so I only called switches 2 “downto” 0. I made sure to name my inputs and outputs accordingly in order to better understand my code. The architecture (logic) portion of my code is shown below:

```
Component comparatorLOGIC
    Port (P1,P0,Q1,Q0 :in  STD_LOGIC;
          GT,EQ,LT   :out  STD_LOGIC);
End Component;
```

My port maps assignments are shown below. P1 was assigned to switch 3 because it was my most significant bit (MSB), while Q0 was assigned to switch 0 since it was my least significant bit (LSB). When assigning my LEDs I tried to match the order with the diagram given to me in the problem statement portion of the lab.
For the actual logic portion of my design, I was required to implement combinational logic. To me, the easiest way to implement this was to take my values directly from my truth table, without reducing my circuit (see below). For each HIGH under the P>Q column, I referred to the input values of P1, P0, Q1 and Q0 ORing the 6 possible outcomes together, thus giving me my logic statement for GT. I followed this same procedure for the four possible outcomes which triggered a HIGH for P=Q (my EQ logic statement), as well as for the six possible outcomes triggering a HIGH for P<Q (my LT logic statement).

\[
\begin{align*}
\text{GT} & \leftarrow ((\text{NOT } P_1 \text{ AND } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \text{ OR } (P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \\
& \text{ OR } (P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \text{ OR } (P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \\
& \text{ OR } (P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0))
\end{align*}
\]

\[
\begin{align*}
\text{EQ} & \leftarrow ((\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \text{ OR } (\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \\
& \text{ OR } (\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \text{ OR } (\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \\
& \text{ OR } (\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0))
\end{align*}
\]

\[
\begin{align*}
\text{LT} & \leftarrow ((\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND NOT } Q_1 \text{ AND NOT } Q_0) \text{ OR } (\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND Q_1 AND NOT Q_0}) \\
& \text{ OR } (\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND Q_1 AND NOT Q_0}) \text{ OR } (\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND Q_1 AND NOT Q_0}) \\
& \text{ OR } (\text{NOT } P_1 \text{ AND NOT } P_0 \text{ AND Q_1 AND NOT Q_0))}
\end{align*}
\]

My second design was based on the 4:1 Multiplexer. A multiplexer, or Mux, is an electronic device which selects from several input signals and transmits one or more output signals. Multiplexers are also referred to as Data Selectors. The simplest of the various Multiplexers is a two input Mux that has two signal inputs, one control input and one output. A very common example of an analog Mux would be the source control on a home stereo unit which allows a user to choose between the audio from a CD player, DVD player or cable television line.

Multiplexers also are used in building digital semiconductors such as central processing units (CPUs) as well as graphics controllers. In these applications, the number of inputs is generally a multiple of two, the number of outputs is either one or relatively small multiple of two, and the number of control signals is related to the combined number of inputs and outputs. For example, a two-input, one-output multiplexer requires only one control signal to select the input, and a 16-input, four-output multiplexer requires four control signals to select the input and two to select the output.

There are several truth tables that can be drawn up for any Multiplexer, for the 4:1 Mux simulated in Lab 3, however, I used a summarized version of all four possible truth tables (see below).

<table>
<thead>
<tr>
<th>S1</th>
<th>S0</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>D2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>D3</td>
</tr>
</tbody>
</table>
As you can see from the truth table, whenever the binary value for the SELECT line (S1 and S0) is “00”, only data from line D0 will be “selected”. When the binary value for the SELECT line is “01”, only data from line D1 will be “selected”. When the binary value for the SELECT line is “10” only data from line D2 will be “selected”. When the binary value for the SELECT line is “11” only data from line D3 will be “selected”.

In order to adequately simulate the logic of this circuit on my DE-1 board I needed to design code which fit the criteria stated above. The template of my code remained the same as in Lab 2, with only the amount of inputs and outputs changing.

```vhdl
Port (CLOCK_50 : IN STD_LOGIC;
    SW       : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    LEDG     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END MUX;
```

Even though there are only six inputs, typically requiring only six switches, I implemented 10 switches because I chose to assign my two select lines to switches 9 and 8 so that they would be separated from my data lines. I did this so that it would be easier visually for me to decipher which switches were used for the four data lines and which were my two select lines on the DE-1 board. Since I used switches 9 and 8 I needed to implement all switches in between 9 and 0 (even though I was not assigning any inputs to switches 7 through 4).

The entity of my design stayed the same as before, only changes made were the naming conventions of my inputs and outputs. Both inputs and outputs were named to match the naming scheme of the diagram in the problem statement, allowing for my code to be easier to understand.

```vhdl
Entity MuxLogic is
  Port (D3,D2,D1,D0,S1,S0 : in STD_LOGIC;
        F : out STD_LOGIC);
End MuxLogic;
```

Once again, my port map implementation was similar to my previous code, only the switch and LED assignments differ.

```vhdl
Port Map(S1=>SW(9), S0=>SW(8), D3=>SW(3), D2=>SW(2), D1=>SW(1), D0=>SW(0), F=>LEDG(7));
```

For the logic portion of my Mux design, I was required to implement a with/else statement. The structure of which is shown in my code below.

```vhdl
F <= D3 when (S1 = '1' and S0 = '1') else
    D2 when (S1 = '1' and S0 = '0') else
    D1 when (S1 = '0' and S0 = '1') else
    D0, when (S1 = '0' and S0 = '0') else
    '0';
```
The logic above follows my truth table precisely. It covers all four possible outcomes and is simple enough that even those who are new to VHDL design can read it and understand the outcome of each binary input possibility.

My third design was based on the Octal Decoder. Decoders are simply a collection of logic gates which are arranged in a specific way so as to breakdown any combination of inputs to a set of terms that are all set to '0' apart from one term. Therefore when one input changes, two output terms will change. Note that these terms are "minterms", remembering that minterms use a variable once, and once only. Let's say we have N inputs to a decoder, the number of outputs will be equal to \(2^n\). Thus there will be one line at the output for each possible input.

The truth table I used for the octal decoder is shown below.

<table>
<thead>
<tr>
<th>Octal</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>Y7</th>
<th>Y6</th>
<th>Y5</th>
<th>Y4</th>
<th>Y3</th>
<th>Y2</th>
<th>Y1</th>
<th>Y0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

As you can see from the truth table above, as the binary input value changes from 0 to 7, the corresponding output value is displayed. It was much clearer once implemented on the DE-1 board because you can see the corresponding LED light up as the input values change.

In order to adequately simulate the logic of this circuit on my DE-1 board I needed to design code which fit the criteria stated above. The template of my code remained the same as in Lab 2, with only the amount of inputs and outputs changing.

```vhdl
Entity octalDECODERlogic is
    Port (inputs :in STD_LOGIC_VECTOR(2 downto 0);
            Y     :out STD_LOGIC_VECTOR(7 downto 0));
End octalDECODERlogic;
```

The biggest difference between this code and my previous codes is, rather than assigning each individual input its own name, I chose to implement all inputs together using a vector. A vector is merely a data bus. Since all three of my inputs combined together are one binary input value, it made more sense to set up my code this way. The same theory applied to my output which I named "Y" rather than "Y7-Y0".

As for my architecture, or logic, I was required to implement a with/select statement for this design. The following snapshot shows this implementation from my logic file.
From the code above you see that my output “Y” changes according to my variable “inputs”. The column on the left is what is displayed on the LEDs as the input column on the right changes its value. The inputs are assigned to switches 2 through 0, so as we flip these switches on the DE-1 board we see the corresponding LEDs light up. This code is fairly straightforward and easy to interpret.

For the fourth and final design was based on the Decimal Encoder. This encoder generates a three-bit binary code corresponding to the switch position (one out of eight positions). The truth table for the 8-3 binary encoder (8 inputs and 3 outputs) is shown in the table below. It is assumed that only one input has a value of 1 at any given time, otherwise the output has some undefined value and the circuit is meaningless.

<table>
<thead>
<tr>
<th>A7</th>
<th>A6</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>Y2</th>
<th>Y1</th>
<th>Y0</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

As you can see from the truth table above, as you cycle through the switches (one at a time), the corresponding binary output is displayed. It was much clearer once implemented on the DE-1 board because you can see the corresponding LEDs light up as the input values change.

In order to adequately simulate the logic of this circuit on my DE-1 board I needed to design code which fit the criteria stated above. The template of my code remained the same as in the previous three designs, with only the amount of inputs and outputs changing.

```
with inputs select
  Y <= "00000001" when "000",
       "00000010" when "001",
       "00000100" when "010",
       "00001000" when "011",
       "00010000" when "100",
       "00100000" when "101",
       "01000000" when "110",
       "10000000" when "111";
End Logic;
```
The biggest difference between this code and my previous codes is, rather than assigning each individual input its own name, I chose to implement all inputs together using a vector. A vector is merely a data bus. Since all three of my inputs combined together are one binary input value, it made more sense to set up my code this way. The same theory applied to my output which I named “Y” rather than “Y3-Y0”.

As for my architecture, or logic, I was allowed to implement any method of my choice, so I chose to stick with the with/select statement for this design. The following snapshot shows this implementation from my logic file.

```vhdl
with inputs select
  Y <=  "0000" when "0000000001",
       "0001" when "0000000010",
       "0010" when "0000000100",
       "0011" when "0000001000",
       "0100" when "0000010000",
       "0101" when "0000100000",
       "0110" when "0001000000",
       "0111" when "0010000000",
       "1000" when "0100000000",
       "1001" when "1000000000",
       "0000" when others;
```

End Logic;

From the code above you see that my output “Y” changes according to my variable “inputs”. The column on the left is what is displayed on the LEDs as the input column on the right changes its value. The inputs are assigned to switches 9 through 0, so as we flip these switches on the DE-1 board we see the corresponding LEDs light up. This code is fairly straightforward and easy to interpret as well.

**IMPLEMENTATION, TEST CRITERIA and RESULTS:**

Once the code for each design was written and compiled it was time to implement them utilizing the DE-1 boards. After each successful compilation, I would burn the board in order to test my code. For each simulated circuit I would go through the truth tables making sure that my input and output values each corresponded accordingly. My input values were implemented by using the various switches assigned to them, while my outputs were displayed using the green LEDs.

The test criterion for each circuit was the waveforms. All waveforms were simulated accurately and are shown below with brief explanations describing all input and output values, as well as switch and LED assignments for each.

**Comparator**

For the Comparator switches 3 through 0 were assigned to inputs P1 (switch 3) through Q0 (switch 0); P1 being the MSB and Q0 being the LSB. Switch assignments were as follows: P1 => switch 3, P0 => switch 2, Q1 => switch 1 and Q0 => switch 0. The outputs were demonstrated by assigning them to LEDGs 2 through 0. LED assignments were as follows: GT (greater than) => LEDG (2), EQ (equal to) => LEDG (1) and LT (less than) => LEDG (0).
GT (P > Q)

EQ (P = Q)

LT (P < Q)

Multiplexer

For the Multiplexer switches 3 through 0 were assigned to inputs D3 (switch 3) through D0 (switch 0); D3 being the MSB and D0 being the LSB. The output was demonstrated by assigning it to LEDG 7. The reason I chose 7 rather than 0 is because I wanted my output to be closer to my switches; visually it made more sense to me. My two selectors S1 and S0 were assigned to switches 9 and 8. S1 (switch 9) was my MSB and S0 (switch 8) was my LSB. I
chose these two switches for my selectors to keep them separate from my data switches as to not confuse them while demonstrating my design.

As you can see from the waveform above when my selectors are “00”, the data line that is chosen is D0. I have grouped the other three data lines together so that it is more clearly visible which input is chosen.

As you can see from the waveform above when my selectors are “01”, the data line that is chosen is D1. I have grouped the other three data lines together so that it is more clearly visible which input is chosen.

As you can see from the waveform above when my selectors are “10”, the data line that is chosen is D2. I have grouped the other three data lines together so that it is more clearly visible which input is chosen.
As you can see from the waveform above when my selectors are “11”, the data line that is chosen is D3. I have grouped the other three data lines together so that it is more clearly visible which input is chosen.

**Octal Decoder**

For the octal decoder switches 3 through 0 were assigned to inputs 3 (switch 3) through 0 (switch 0); inputs 3 being the MSB and inputs 0 being the LSB. The outputs were demonstrated by assigning them to LEDGs 9 through 0; Y9 being the MSB and Y0 being the LSB.

The waveform clearly shows that as the “inputs” value changes, the Y (output) value changes its value accordingly. For example, when the “inputs” binary value is “1”, the output (Y1) is HIGH. The same goes for every other corresponding input/output values. In lab, as the binary input value changed, the corresponding LED would light up; example: binary value 4, LEDG(4) would light up and so on.

**Decimal to Binary Encoder**

For the decimal to binary encoder switches 9 through 0 were assigned to inputs 9 (switch 9) through 0 (switch 0); inputs 9 being the MSB and inputs 0 being the LSB. The outputs were demonstrated by assigning them to LEDGs 3 through 0; Y3 being the MSB and Y0 being the LSB. Essentially this circuit used all the same syntax as the previous one (octal decoder), only the input and output values are swapped.

Rather than an input binary value triggering the corresponding LED as the output; as you flip each individual switch, the corresponding binary output value is illuminated through the LEDs. Unfortunately I had trouble with the waveform for this circuit, so I do not have one to show my results. Basically, the waveform for this circuit should look similar to the one above (octal decoder), only the output and input values are reversed.

For each design my code was implemented successfully, with the exception of the decimal to BCD encoder. I misunderstood what that circuit was supposed to do and therefore had set up my logic file incorrectly. My input values were setup incorrectly, therefore causing my output values to display the incorrect LED sequences.
CONCLUSION:
Lab 3 did not go as smoothly as the first two, however I still consider it a success seeing that I was able to get all four designs working correctly by the end of the lab. I was having some difficulty compiling my last two pieces of code, the octal decoder and the decimal to BCD encoder. After some debugging I was able to figure out the issue and compile successfully. Although debugging was extremely frustrating during lab due to the time constraints, I feel that I learned a lot from my errors and am able to better understand VHDL design as well as the logic of each of the four circuits simulated.
REFERENCES:

1. *Digital Fundamentals*, Floyd, Prentice Hall, Chapters 1-4 plus appendices

2. *Shock and Awe VHDL tutorial*, Bryan Mealy


4. In class lecture and notes, Kelly, L., ET398