ET398 LAB 6

“Flip-Flops in VHDL”

Flip-Flops

March 3, 2013

Tiffany Turner “ “
OBJECTIVE
The objectives of this lab are for you to begin the sequential and memory programming using flip flops in VHDL program. Emphasis will be place on how you implement the device in your code. Logical constructs, comments and demonstration will all be part of the grade (Kelly, 2013).

PROBLEM STATEMENT
Implement all four of the circuits below. It is only necessary to output Q to an LED (not Q̅). Each circuit should have its own logic file. Use the 1 second clock to run your flip flops and output it to any LED as a check.

✔ Circuit 1(D): Implement one flip-flop using the function table from 74LS74.
✔ Circuit 2 (JK): Implement one flip-flop using the function table for the 74LS112.
✔ Circuit 3: Design the complete D Quad Latch (7475) chip.
✔ Circuit 4: Implement the dual JK 74LS78 with a common clock, common clear and individual presets.

Demonstrate the implementation of each circuit to the instructor on the DE-1 board.

(Kelly, 2013)

DESIGN METHODOLOGY
For Lab 6, there were four different circuits that I was required to write code for and then simulate using my DE-1 board. All four circuits were flip-flops, just different types applied differently. The logic requirements for each design varied; below I have discussed the details of each circuit implementation.

Flip-flops; not the sandals, but the logic gates, are the fundamental building blocks of sequential logic. The D flip-flop is by far the most important of the clocked flip-flops as it ensures that inputs S and R are never equal to one at the same time. D-type flip-flops are constructed from a gated SR flip-flop with an inverter added between the S and the R inputs to allow for a single D (data) input. This single data input D is used in place of the "set" signal, and the inverter is used to generate the complementary "reset" input thereby making a level-sensitive D-type flip-flop from a level-sensitive SR-latch as now S = D and R = not D as shown in the diagram below.
A simple SR flip-flop requires two inputs, one to "SET" the output and one to "RESET" the output. By connecting an inverter (NOT gate) to the SR flip-flop you can "SET" and "RESET" the flip-flop using just one input as now the two input signals are complements of each other. This complement avoids the ambiguity inherent in the SR latch when both inputs are LOW, since that state is no longer possible.

Thus the single input is called the "DATA" input. If this data input is HIGH the flip-flop would be "SET" and when it is LOW the flip-flop would be "RESET". However, this would be rather pointless since the flip-flop's output would always change on every data input. To avoid this an additional input called the "CLOCK" or "ENABLE" input is used to isolate the data input from the flip-flop after the desired data has been stored. The effect is that D is only copied to the output Q when the clock is active. This then forms the basis of a D flip-flop.

The D flip-flop will store and output whatever logic level is applied to its data terminal so long as the clock input is HIGH. Once the clock input goes LOW the "set" and "reset" inputs of the flip-flop are both held at logic level "1" so it will not change state and store whatever data was present on its output before the clock transition occurred. In other words the output is "latched" at either logic "0" or logic "1". There are many different D flip-flop IC's available in both TTL and CMOS packages with the more common being the 74LS74 which is a Dual D flip-flop IC, which contains two individual D type bi-stables within a single chip enabling single or master-slave toggle flip-flops to be made.

**74LS74 Circuit Design**
D Flip-Flop Truth Table

For this circuit, two logic files were required, one for the actual JK application and the other for the one second clock. Being that there were two logic files, this meant that two components were needed.

The components are the hardware assignments according to the various input/output requirements. For the D flip-flop there were 4 inputs and 1 output. As for the clock (CLK), I had 2 inputs and 1 output. The most important aspect of tying these two files together (in the hardware template) was the usage of a signal.

```vhdl
Component flipflopDLOGIC

Port (D, CLK, Preset, Clear, Q : in STD_LOGIC);
Q : out STD_LOGIC_VECTOR(0 DOWNTO 0));

End Component;

Component clockDIV

Port (clock, enableCLOCK, newCLOCK : in STD_LOGIC);

End Component;
```

The purpose of the signal implementation was to create a “bidirectional” data line in order to tie in the clock to the D flip-flop logic. Without the signal, this would not be possible because all other assignments were set to either “in” or “out”.

```vhdl
Signal clockDIVISION : STD_LOGIC;
```

The components are the hardware assignments according to the various input/output requirements. For the D flip-flop there were 4 inputs and 1 output. As for the clock (CLK), I had 2 inputs and 1 output. The most important aspect of tying these two files together (in the hardware template) was the usage of a signal.
Above is my port map application for each logic file. The code above clearly shows my signal utilization. My ‘D’ input was assigned to switch 9, Clear was assigned to switch 1 and preset was assigned to switch 0. My final output is simply assigned to one of the red LEDs; LEDR (0), while my clock pulse was demonstrated/checked using one of the green LEDs; LEDG(7).

    LEDG(7) <= clockDIVISION;

The actual logic statement used for this design was an if/else statement. It was pretty easy to implement and extremely straightforward as you can see below. All I did was follow the truth table logic.

```
Begin
  Process(CLK)
  Begin
    if (Preset = '0') then
      Q <= '1';
    elsif (clear = '0') then
      Q <= '0';
    elsif (rising_edge(CLK)) then
      Q <= D;
    end if;
  end process;
```

For this circuit not only was I to write the code for the actual decoder, but I was also to write code for the clock logic associated with it. I was able to utilize the same clock code from previous labs. The logic for this is shown below.

```
Begin
  Process(clock, enableCLOCK)
  Begin
    variable count : natural := 0;
    if(enableCLOCK = '0') then
      count := 0;
      clockDIV <= '0';
    elsif(clock'event AND clock = '1')
      then count := count + 1;
    if(count = 25000000) then
      clockDIV <= NOT clockDIV;
      count := 0;
    end if;
    newCLOCK <= clockDIV;
  end Process;
```

The clock pulse was outputted to a green LED; LEDG(7).

**74LS112 Circuit Design**
The JK flip-flop is a special digital circuit that can be set up to act like any type of flip-flop. It has a clock input that is set up to transfer data on either the leading or trailing edge of the clock.
pulse. The 74LS112 ICs simulated in this lab have an asserted low clock, meaning that the transfer of synchronous data from input to output occurs when the clock pulse goes from binary high to binary low. The inputs J and K are the flip-flops synchronous inputs. The inputs Preset and Clear are asynchronous inputs, meaning they can affect the output of the flip-flop at any time during the clock cycle. The asynchronous inputs are asserted low for the 74LS112.

The truth table for the JK flip-flop:

<table>
<thead>
<tr>
<th>Pre</th>
<th>Clr</th>
<th>J</th>
<th>K</th>
<th>Clk</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>UNUSED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>↓</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>↓</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>N/C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>↓</td>
<td>TOGGLE</td>
</tr>
</tbody>
</table>

The diagram above shows the entire structure for the actual 74LS112 IC, however, for Lab 6 I was only required to implement one of the JK flip-flops shown on the chip. For this design I implemented an if/else statement in my logic file. Although this type of logic statement seemed to make the most sense at the time, in hind sight I feel as though a case statement may have been a much simpler approach.

For this circuit I needed four inputs (denoted by the “in STD_LOGIC”) and one output (denoted by the “out STD_LOGIC”).
JK Logic Entity

Entity flipflopJKLOGIC is

Port (CLK : in STD_LOGIC;
     Preset : in STD_LOGIC;
     Clear : in STD_LOGIC;
     J, K : in STD_LOGIC;
     Q : out STD_LOGIC);

End flipflopJKLOGIC;

The JK logic entity is necessary because this is where all inputs/outputs are declared to be used in the actual logic statement (which is shown below).

Begin
  Process(Preset, Clear, CLK, J, K)
  Begin
    if (Preset = '0') then
      Qoutput <= '1';
    elsif (Clear = '0') then
      Qoutput <= '0';
    elseif (falling_edge(CLK)) then
      if (J = '1' AND K = '0') then
        Qoutput <= '1';
      elsif (J = '0' AND K = '1') then
        Qoutput <= '0';
      elseif (J = '1' AND K = '1') then
        Qoutput <= NOT Qoutput;
      else Qoutput <= Qoutput;
    end if;
  end if;
  end process;

  Q <= Qoutput;

End Logic;

The preset and clear inputs override the synchronous inputs. When preset is asserted, the flip-flop is set and Q = 1. When clear is asserted, the flip-flop is cleared and Q = 0 (the Xs denote 'don't care' conditions).

When the asynchronous inputs are both high: the synchronous inputs can affect the input but only on the negative going clock edge, which is characterized by the down arrows in the truth table (and by the “falling_edge” statement in the code); when J is high and K is low Q = 1, when J is low and K is high Q = 0, for J and K both low the output will not change from any previous state and when J and K are both high the output will change states on each triggering clock edge.

The condition for both asynchronous inputs low is not used as the output will be unpredictable due to the nature of the flip-flop (both Q and Q would briefly be high and the feedback loop would no longer function).
The condition for which all inputs (except clock) are high is what makes the counter possible. A clock signal is applied to a flip-flop, all inputs are high and the output Q will alternate between states on the negative edge of each clock pulse. Q is then showing exactly one half of the clock frequency. This Q1 is delivered to the clock input of a second flip-flop with all other inputs tied high. The second clock is receiving half the frequency of the original clock and Q2 becomes one fourth the frequency of the original clock. This process can continue indefinitely and the resulting circuit can be described as a frequency divider. The frequency division = 2^n, n = number of flip-flops.

My logic implementation for the clock is shown below.

```
Begin
  Process(clock, enableCLOCK)
  
  Variable count : natural := 0;
  Begin
    if(enableCLOCK = '0') then
      count := 0;
      clockDIV <= '0';
    elsif(clock'event AND clock = '1') then
      count := count + 1;
    if(count = 25000000) then
      clockDIV <= NOT clockDIV;
      count := 0;
    end if;
  end if;
End Process;

newCLOCK <= clockDIV;
```

As for the clock logic entity, two inputs and one output were utilized.

```
Entity clockDIV is
  Port ( clock : in STD_LOGIC;
         enableCLOCK : in STD_LOGIC;
         newCLOCK : out STD_LOGIC);
End clockDIV;
```

A signal was also implemented in order to allow bidirectional data flow inside the “clock” execution. Without the signal, this would not be possible because all other assignments were set to either “in” or “out”.

```
--signal for my clock divider
Signal clockDIV :STD_LOGIC;
```

Seeing how there were two different logic files, one for the JK flip flop and the other for the clock, it was necessary that I implement two components in my hardware template. Keep in mind that the hardware file is what makes communication between the logic and the hardware
possible. The component files allow the hardware to know which inputs are to be called upon for certain “duties”, which are executed within these logic files.

Clock Logic Component:

```vhdl
Component clockDIV

Port (clock : in STD_LOGIC;
       enableCLOCK : in STD_LOGIC;
       newCLOCK : out STD_LOGIC);

End Component;
```

The component for each hardware file is the same as the Entity of the actual logic file utilized in each design. Therefore, I am not going to paste my component for the JK logic, however keep in mind that the two are implemented differently depending on the file it is used in. What I mean by this is that the “component” is part of the architecture (which is the logic of the circuit) of the hardware file and in the logic file, this same piece of code is implemented in the Entity (which describes the actual interface; assigns the I/Os).

Now the actual pin assignments, or port map, implementation for this particular design was as follows:

```vhdl
myFlipflopJXLogicPortMap: flipflopJXLogic Port Map(J => Sw(9), K => Sw(8), Preset => Sw(0), Clear => Sw(1), CLK => clockDIVISION, Q => LEDR(0));
myClockDIVPortMap: clockDIV Port Map(clock => CLOCK_0, enableCLOCK => '1', newCLOCK => clockDIVISION);
```

As with the components, I also needed two port maps due to the two logic files used in my design. The most important aspect of tying these two files together (in the hardware template) was the usage of a signal. The same signal implemented in the JK logic file was also implemented in each of my port maps.

I used switches 9 and 8 for my J and K so that I could keep them separate from my Clear and Preset. Having the four inputs separated by six switches made it easier for me to differentiate between them when demoing in lab. I assigned my only output (Q) to LEDR(0) and my clock pulse to LEDG(7).

```vhdl
LEDG(7) <= clockDIVISION;
```

This assignment was placed outside of the individual port maps but still inside of the actual port map specifications.

For my third circuit I was to design the code for the 7475 IC (D Quad Latch). The difference between a D-type latch and a D-type flip-flop is that a latch does not have a clock signal to change state whereas a flip-flop always does. The D flip-flop is an edge triggered device which transfers input data to Q on clock rising or falling edge. Data Latches are level sensitive devices such as the data latch and the transparent latch.

The 7475 is a TTL MSI circuit that contains four D latches, also known as a 'quad bi-stable
latch.’ These latches are ideally suited for use as temporary storage for binary information between processing units and input/output or indicator units. Information present at a data (D) input is transferred to the Q input when the enable (G) is high, and the Q output will follow the data input as long as the enable remains high. When the enable goes low, the information (that was present at the data input at the time the transition occurred) is retained at the Q output until the enable is permitted to go high. The logic diagram and truth table for the 7475 IC is shown below.

**7475 Circuit Design**

![Logic Diagram](image)

<table>
<thead>
<tr>
<th>Enable</th>
<th>D</th>
<th>Q</th>
<th>Q̅</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>Q₀</td>
<td>Q̅₀</td>
<td>No change</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Reset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Set</td>
</tr>
</tbody>
</table>

The logic for this circuit was fairly straight forward making it the easiest of the four designs to implement. Basically, as you can see from the truth table above, as long as Enable is HIGH, ‘Q’ follows ‘D’. When Enable is LOW, there is no change. My logic statement for this design is shown below.
I applied two simple 'if' statements in order to simulate the D QuadLatch. The code above is fairly straightforward as it follows the truth table for this circuit; Q follows D as long as Enable is HIGH.

For this circuit there was only one logic file required. While the logic file is what creates the actual simulation of this circuit, it would not be able to perform its functions had it not been for the hardware template; which is used to communicate with the DE-1 board. For the architecture of this circuit, only one component was needed rather than two, since this circuit did not have a clock.

```
BEGIN
  Process(D4,D3,D2,D1,enable2,enable1)
  Begin
    -Latch_1-2
    if(enable1 = '1') then
      Q1 <= D1;
      Q2 <= D2;
    end if;
    -Latch_3-4
    if(enable2 = '1') then
      Q3 <= D3;
      Q4 <= D4;
    end if;
  end process;
END
```

The components are the hardware assignments according to the various input/output requirements. For the D Latch there were 6 inputs (D1 through D4 and Enable1, Enable2) and 4 outputs (Q1 through Q4).

```
Component DLatchLogic
Port (D4,D3,D2,D1 :in STD_LOGIC;
      enable2,enable1 :in STD_LOGIC;
      Q4,Q3,Q2,Q1 :out STD_LOGIC);
End Component;
```

Above is my port map application for my logic file. My inputs are set to switches 3 through 0, with D4 (MSB) being assigned to switch 3 and D1 (LSB) being assigned to switch 0. My output assignments were set up similar; Q4 (MSB) was assigned to LEDG 3 and Q1 (LSB) was assigned to LEDG0. I assigned my two enables to switches 9 (enable2) and 8 (enable1) so that I could more easily differentiate between them and my inputs.

My fourth and final design was the 74LS78 JK Flip-Flop. This chip contains two negative-edge
triggered flip-flops with individual J-K, preset inputs and common clock and common clear inputs. The logic levels at the J and K inputs may be allowed to change while the clock pulse is HIGH and the flip-flop will perform according to the function table as long as minimum setup and hold times are observed. The preset and clear are asynchronous active-low inputs. When LOW, they override the clock and data inputs forcing the outputs to the steady-state levels as shown in the truth table below.

### 74LS78 Circuit Design

![74LS78 Circuit Diagram](image)

### JK Truth Table

<table>
<thead>
<tr>
<th>Pre</th>
<th>Clr</th>
<th>J</th>
<th>K</th>
<th>Clk</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>UNUSED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Q₀</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>↓</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>↓</td>
</tr>
</tbody>
</table>
The diagram above shows the entire structure for the actual 74LS78 IC, for this design I was required to simulate both JKS, unlike with the 74LS112 design. For this design I implemented an if/else statement in my logic file. Although this type of logic statement seemed to make the most sense at the time, in hind sight (once again) I feel as though a case statement may have been a much simpler approach.

For this circuit I needed seven inputs (denoted by the “in STD_LOGIC”) and two outputs (denoted by the “out STD_LOGIC”).

JK Logic Entity

Entity flipflop78LOGIC is
    Port (J2,K2,J1,K1 :IN STD_LOGIC;
          preset2, preset1 :IN STD_LOGIC;
          Clear, CLK :IN STD_LOGIC;
          Q2,Q1 :OUT STD_LOGIC);
End flipflop78LOGIC;

The JK logic entity is necessary because this is where all inputs/outputs are declared to be used in the actual logic statement (which is shown below).
The preset and clear inputs override the synchronous inputs. When preset is asserted, the flip-flop is set and Q = 1. When clear is asserted, the flip-flop is cleared and Q = 0 (the Xs denote 'don't care' conditions).

When the asynchronous inputs are both high: the synchronous inputs can affect the input but only on the negative going clock edge, which is characterized by the down arrows in the truth table (and by the “falling_edge” statement in the code); when J is high and K is low Q = 1, when J is low and K is high Q = 0, for J and K both low the output will not change from any previous state and when J and K are both high the output will change states on each triggering clock edge.

The condition for both asynchronous inputs low in not used as the output will be unpredictable due to the nature of the flip-flop (both Q and Q would briefly be high and the feedback loop would no longer function).

The condition for which all inputs (except clock) are high is what makes the counter possible. A clock signal is applied to a flip-flop, all inputs are high and the output Q will alternate between
states on the negative edge of each clock pulse. Q is then showing exactly one half of the clock frequency. This Q1 is delivered to the clock input of a second flip-flop with all other inputs tied high. The second clock is receiving half the frequency of the original clock and Q2 becomes one fourth the frequency of the original clock. This process can continue indefinitely and the resulting circuit can be described as a frequency divider. The frequency division \(2^n, n = \) number of flip-flops.

My logic implementation for the clock is shown below.

```vhdl
Begin
  Process (clock, enableCLOCK)
  Variable count : natural := 0;
  Begin
  if(enableCLOCK = '0') then
    count := 0;
    ClockDIV <= '0';
  elsif(clock'event AND clock = '1') then
    count := count + 1;
  if(count = 25000000) then
    ClockDIV <= NOT ClockDIV;
    count := 0;
  end if;
  end if;
  End Process;

  newCLOCK <= ClockDIV;
End
```

As for the clock logic entity, two inputs and one output were utilized.

```vhdl
Entity clockDIV is
  Port (clock :in STD_LOGIC;
        enableCLOCK :in STD_LOGIC;
        newCLOCK :out STD_LOGIC);
End clockDIV;
```

A signal was also implemented in order to allow bidirectional data flow inside the “clock” execution. Without the signal, this would not be possible because all other assignments were set to either “in” or “out”.

```vhdl
Signal ClockDIV :STD_LOGIC;
```

Seeing how there were two different logic files, one for the JK flip-flop and the other for the clock, it was necessary that I implement two components in my hardware template. Keep in mind that the hardware file is what makes communication between the logic and the hardware possible. The component files allow the hardware to know which inputs are to be called upon for certain “duties”, which are executed within these logic files.
Clock Logic Component:

```vhdl
Component clockDIV
    Port (clock :in STD_LOGIC;
         enableCLOCK :in STD_LOGIC;
         newCLOCK :out STD_LOGIC);
End Component;
```

The component for each hardware file is the same as the Entity of the actual logic file utilized in each design. Therefore, I am not going to paste my component for the JK logic, however keep in mind that the two are implemented differently depending on the file it is used in. What I mean by this is that the “component” is part of the architecture (which is the logic of the circuit) of the hardware file and in the logic file, this same piece of code is implemented in the Entity (which describes the actual interface; assigns the I/Os).

Now the actual pin assignments, or port map, implementation for this particular design was as follows:

```vhdl
myFlipFlop78LogicPortMap: flipFlop78Logic Port Map(Clear => Sw(9), preset2 => Sw(3), preset1 => Sw(2), CLK => clockDIVISION, J2 => Sw(4),
                                                     K2 => Sw(3), I1 => Sw(1), I1 => Sw(0), Q2 => LED(1), Q1 => LED(0));
myClockDIVPortMap: clockDIV Port Map(clock => CLOCK_50, enableCLOCK => '1', newCLOCK => clockDIVISION);
```

As with the components, I also needed two port maps due to the two logic files used in my design. The most important aspect of tying these two files together (in the hardware template) was the usage of a signal. The same signal implemented in the JK logic file was also implemented in each of my port maps.

I used switch 9 for my clear, switches 5 and 2 for my presets, switches 4 and 3 for JK2 and switches 1 and 0 for JK1. Once again I kept my Clear separate from all other inputs in order to help differentiate between them in lab. I assigned my two outputs (Q2 and Q1) to LEDR(1) and LEDR(0), while my clock pulse was assigned to LEDG(7).

```vhdl
LEDG(7) <= clockDIVISION;
```

This assignment was placed outside of the individual port maps but still inside of the actual port map specifications.

**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 board. 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 red and green LEDs.
For this lab, we were only required to test the first two circuits using waveforms. All of my circuits were successfully demoed. The two waveforms that were required are shown below with related annotations.

**D Flip-Flop**

- All inputs are HIGH at Rising Edge of CLOCK, Q goes HIGH
- When Preset, Clear = HIGH and D = LOW (at rising edge), Q goes LOW
- When CLEAR is LOW, Q is LOW

**J K Flip-Flop (74LS112)**

- When all inputs are HIGH, Q TOGGLES at falling edge of CLOCK
- When all inputs EXCEPT ‘J’ are HIGH, Q = LOW at falling edge of CLOCK
- When all inputs EXCEPT ‘K’ are HIGH, Q=HIGH at falling edge of CLOCK

**CONCLUSION**

Aside from the fact that I was unable to demo all four of my circuits in lab on Friday, I would consider this lab a success. I was able to successfully write and compile my code later that day in order to upload all of my files and prepare for my next opportunity to demo. Although I had to debug syntax errors on 2 out of 4 of my designs, I feel as though I am finally getting the hang of VHDL after the last couple of labs. Especially since my last two designs were written in notepad, opened up in Quartus and then successfully compiled without any errors.

I credit my surplus of syntax errors to my lack of preparation prior to lab. I realize how important it is to have all of my code written and debugged prior to lab. I hope to be completely caught up in all my classes by the end of this week, and am determined to keep it that way so that
I will not be turning in any more late assignments. With that said, I really enjoyed this lab and all its challenges, and look forward to the next.

REFERENCES

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

2. Shock and Awe VHDL tutorial, Bryan Mealy

3. TTL Data Book Vol. II, Texas Instruments

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