ET398 LAB 2

“Intro to Altera Board and VHDL Implementation”

Adders

February 02, 2013

Tiffany Turner
OBJECTIVE:
The overall objective of our first lab was for us to become acquainted with the ET398 laboratory environment, equipment, requirements, terminology and most importantly to gain design and application skills in circuit design and analysis using Altera software and DE1 boards (Kelly, 2013).

PROBLEM STATEMENT:
We were given two circuit diagrams and one truth table, and then asked to come up with a design which would simulate each of the implied circuits on our DE1 boards. The three problems are shown below:

HALF-ADDER CIRCUIT:

FULL-ADDER CIRCUIT:
2-2 BIT ADDER TRUTH TABLE:

<table>
<thead>
<tr>
<th>A1</th>
<th>A0</th>
<th>B1</th>
<th>B0</th>
<th>COUT</th>
<th>S1</th>
<th>S0</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

DESIGN METHODOLOGY:
The most basic adder circuit is the half-adder. It has two inputs and two outputs. Each input delivers a bit to the circuit and the sum bit (signified by Σ), and outputs the corresponding sum from 0 to 1. The sum bit (signified by Σ), shows a 1 on its output for the case of 1 + 1 = 01. The output Σ follows exclusive-OR logic and the carry bit follows the logic for AND. The truth table and circuit schematic for the half-adder are shown below:
The main difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs. The first two inputs are A and B and the third input is an input carry designated as $C_I$. Essentially the full-adder is two half-adders that have been electrically connected. Two bits are delivered to the first half-adder and their sum inputs into the second half-adder along with any external carry in bit $C_I$. The $\Sigma$ from the second half-adder equals the first $\Sigma$ XOR’ED with the external carry bit. The final $C_O$ is AND’ED with the $\Sigma$ from the original two inputs and then OR’ED with their product. The circuit schematic is shown below:

![Circuit Diagram](image)

The truth table for the full adder follows the laws of addition:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>$C_I$</th>
<th>$C_O$</th>
<th>$\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2

Once I understood the basic functionality of the half-adder and full-adder, I was then able to better understand the ripple carry adder; which is what was simulated in the third problem. First of all, a ripple carry adder allows you to add two n-bit numbers together. A simple ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the carry output from each full adder connected to the carry input of the next full adder in the chain. The input is from the right side because the first cell traditionally represents the least significant bit (LSB). The main problem with this type of adder is the delays needed to produce the carry out signal and the most significant bits. These delays increase with the increase in the number of bits to be added.

This delay between the time the inputs are sent to the circuit, and the time the output is computed is due to the fact that combinational logic circuits can't compute the outputs instantaneously. Since an n-bit ripple carry adder consists of n adders, there will be a delay of nT. This is $O(n)$ delay. While the adders are working in parallel, the carries must "ripple" their way from the least significant bit and work their way to the most significant bit. It takes T units for the carry out of
the rightmost column to make it as an input to the adder in the next to rightmost column. Thus, the carries slow down the circuit, making the addition linear with the number of bits in the adder.

![Figure 3](image1.png)

Shown below is the basic schematic for a 4 bit adder using a half-adder and full-adder:

![Figure 4](image2.png)

**TEST CRITERIA:**
My test criteria for this lab consisted of the three waveforms shown below:

**HALF-ADDER:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value at 0 ps</th>
<th>0 ps</th>
<th>10.0 ns</th>
<th>20.0 ns</th>
<th>30.0 ns</th>
<th>40.0 ns</th>
<th>50.0 ns</th>
<th>60.0 ns</th>
<th>70.0 ns</th>
<th>80.0 ns</th>
<th>90.0 ns</th>
<th>100.0 ns</th>
<th>110.0 ns</th>
<th>120.0 ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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 half-adder waveform above the following arithmetic applies:

When \( A=0 \) and \( B=0 \), the result is \( 0 \)
\( (0 + 0 = 0) \)

When \( A=0 \) and \( B=1 \), the result is \( 1 \)
\( (0 + 1 = 1) \)

When \( A=1 \) and \( B=0 \), the result is \( 1 \)
\( (1 + 0 = 1) \)

When \( A=1 \) and \( B=1 \), the result is \( 2 \)
\( (1 + 1 = 2) \)

The half-adder is simply performing basic math.

**FULL-ADDER:**

As you can see from the full-adder waveform above the following arithmetic applies:

When \( A=0, B=0 \) and \( Cin=0 \), the result is \( 0 \)
\( (0 + 0 = 0) \)

When \( A=0, B=0 \) and \( Cin=1 \), the result is \( 1 \)
\( (0 + 1 = 1) \)

When \( A=0, B=1 \) and \( Cin=1 \), the result is \( 2 \)
\( (1 + 1 = 2) \)

When \( A=1, B=1 \) and \( Cin=1 \), the result is \( 3 \)
\( (2 + 1 = 3) \)

When \( A=0, B=1 \) and \( Cin=0 \), the result is \( 1 \)
\( (1 + 0 = 1) \)
The difference between the half-adder and the full-adder waveforms is that the half-adder uses only two bits to calculate the sum. However, the full-adder uses 3 bits to calculate the sum, where bits A and B serve as one number combined, and are then added to the carry-in bit.

**RIPPLE CARRY ADDER (2-2 BIT ADDERS):**

As you can see from the waveform above, the carry-in bit is not used in the sum function of this particular design. The carry-in bit in the ripple carry adder is used to “cascade” the half-adder and the full-adder together in order for the 2 bits from each chip to be summed. This adder uses 2-2 bit binary numbers in order to calculate a sum. The waveform does not adequately show it, but the four values being added are A1, B1, A0 and B0; A1 being the MSB (most significant) and B0 being the LSB (least significant) bits. A1 and A0 act together as the first number, while B1 and B0 act together as the second number in the equation. I have carefully annotated my waveform in order to show that the following applies:

When A1=0, A0=0, B1=0, B0=0, the result is 0  
(0 + 0 = 0)

When A1=0, A0=0, B1=0, B0=1, the result is 1  
(0 + 1 = 1)

When A1=0, A0=0, B1=1, B0=0, the result is 2  
(0 + 2 = 2)

When A1=0, A0=0, B1=1, B0=1, the result is 3  
(0+ 3 = 3)

When A1=0, A0=1, B1=0, B0=0, the result is 1  
(1+ 0 = 1)

When A1=0, A0=1, B1=0, B0=1, the result is 2  
(1 + 1 = 2)
When \(A_1=0, A_0=1, B_1=1, B_0=0\), the result is 3
\((1 + 2 = 3)\)

When \(A_1=0, A_0=1, B_1=1, B_0=1\), the result is 4
\((1 + 3 = 4)\)

When \(A_1=1, A_0=0, B_1=0, B_0=0\), the result is 2
\((1 + 2 = 3)\)

When \(A_1=1, A_0=0, B_1=0, B_0=1\), the result is 3
\((2 + 1 = 3)\)

When \(A_1=1, A_0=0, B_1=1, B_0=0\), the result is 4
\((2 + 2 = 4)\)

When \(A_1=1, A_0=0, B_1=1, B_0=1\), the result is 5
\((2 + 3 = 5)\)

When \(A_1=1, A_0=1, B_1=0, B_0=0\), the result is 3
\((3 + 0 = 3)\)

When \(A_1=1, A_0=1, B_1=0, B_0=1\), the result is 4
\((3 + 1 = 4)\)

When \(A_1=1, A_0=1, B_1=1, B_0=0\), the result is 5
\((3 + 2 = 5)\)

When \(A_1=1, A_0=1, B_1=1, B_0=1\), the result is 6
\((3 + 3 = 6)\)

**IMPLEMENTATION and RESULTS**

The implementation of these three designs consisted of three sets of code which built upon each other. I began with the simplest of the three circuits, the half-adder. The two basic building blocks of any VHDL code are the entity and the architecture. The entity consists of the hardware, while the architecture is the logic of the code.

```vhdl
entity halfADDER is
    PORT (SW : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
           LEDR : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
end halfADDER;
```

As you can see the entity is merely assigning which switches and LEDs are to be used. In the portion of code above you can see that the red LEDs were assigned as the designated outputs and
only two switches were needed as the inputs, (1 downto 0) implies this. As for the logic portion of the code, the architecture calls this from our logic file, see below:

architecture LOGIC of halfADDER is

component halfADDERLogic
  Port(A,B,Cin :in std_logic;
       S,Cout :out std_logic);

End component;

The component “halfADDERLogic” is the actual logic file created which contains the actual design of the simulated chip. In the case of the half-adder, I used combinational logic:

    S    <= A XOR B;
    Cout <= A AND B;

One other portion of the code not yet discussed is the port map. The port map is implemented at the end of the template in order to assign inputs to switches and outputs to LEDs. The highest switch number utilized tells you what number to start with when calling switches in your entity portion of the code, example:

  myfullAdderPortMap:   halfADDERLogic port map(A=>SW(1), B=>SW(0), Cin=>'0',
                                           Cout=>LEDR(1), S=>LEDR(0));

The highest switch number used in the port map is one, which is why (1 downto 0) was used in the entity portion of the code.

For the full-adder, the code was similar; the only differences were the number of inputs (switches), as well as my actual logic used. I still used combinational logic, however, instead of concurrent statements, I used sequential logic:

  Process (A,B,Cin)
      Variable TEMP : std_logic;
      Begin

      TEMP := (A XOR B);
      S    <= TEMP XOR Cin;
      Cout <= (A AND B) OR (TEMP AND Cin);

  End Process;

Implementing the “Process” statement allows each command to be run independently, rather than concurrently. Using the code above, each step was executed one at a time.
With the third and final circuit I used the same code from my full-adder changing only a couple of things. The number of inputs and outputs changed; 4 inputs and 3 outputs. The port map was setup differently as well due to the “cascading” affect required to sum the two, two bit numbers from each “chip”.

adderLSBs: adderCASCADELogic port map(A => SW(2), B => SW(0), Cin => '0', S => LEDG(0), Cout => CarryOut);

adderMSBs: adderCASCADELogic port map(A => SW(3), B => SW(1), Cin => CarryOut, S => LEDG(1), Cout => LEDG(2));

As you can see, the carry-out from the half-adder is used to cascade the two chips. The carry-out from the half-adder connects directly to the carry-in of the full-adder creating this “cascade” affect. Also, a signal was used for the carry-out of the half-adder.

Signal CarryOut: std_logic;

Once my code had been designed and implemented, it was time to run the code and make sure that it compiled. Pleasingly enough, all three of my codes compiled successfully.

CONCLUSION:
Overall, I would have to say that this lab was a success. I was able to successfully demo all three designs in lab on Friday with no complications. The only issue that I did encounter was with the usage of a “process” in my code. We were supposed to use concurrent statements in all of our code, and I did not realize that by implementing a process I was essentially executing my logic sequentially, rather than concurrently. Other than that everything went smoothly and I now know the differences in code implementations, when it comes to sequential and concurrent logic statements.
REFERENCES:

1. **DIGITAL FUNDAMENTALS**, Floyd, Prentice Hall, Chapters 1-4 plus appendices

2. **TTL DATA BOOK** Vol. II, Texas Instruments

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