

**School of Engineering Science  
Simon Fraser University  
ENSC 350 – Digital Systems Design  
Spring 2018**

**Lab 2: Datapaths**

Working week: January 22-26, 2018  
Marking week: January 29-February 2, 2018

In this lab, you will implement a Roulette engine. Roulette is a game played in casinos around the world. Casinos like the game, because it has (among the) worst odds of all games on the casino floor. Designing this Roulette engine will help you understand how a simple datapath can be constructed and controlled by a state machine; this is the foundation of all large digital circuits.

Like all labs in this course, this lab will span two weeks. The first week is intended to be a work-week, where you spend the lab working on the tasks. A TA will be available to help you if you run into problems. The second week is primarily for marking (you should not expect help from the TA during the second week). You should expect that you must do some work on your own (at home or in the lab outside of lab hours) as well. You most certainly cannot finish the lab if you don't start until the marking week.

## PHASE ONE: UNDERSTANDING THE GAME

In this lab, you will design an electronic roulette system using your DE2 boards. Roulette is likely the most simple game you can find in casinos in Las Vegas, Macau, or other gambling meccas. The overall game is as follows: each player (we will assume only one player in our implementation) makes an arbitrary number of bets. A large wheel is then spun, resulting in one winning number. In Las Vegas, the wheel has 36 numbers (1 to 36) as well as a 0 and a 00 (so 38 possible outcomes). In Europe, there is no 00 (so 37 possible outcomes). We will consider the European version (no 00) in this lab. After the spin, the winning bets are paid off at a specified rate. If you have never seen roulette before, you can do an on-line search to learn more details.

In our implementation, we will consider only three types of bets: (1) straight-up, (2) colour, and (3) dozen. In a **straight-up bet**, the player bets on a single number from **0 to 36**. If that number is the result of the spin, then the player wins. Otherwise, the player loses. The payoff from this type of bet is **35:1** (i.e. if the player bets \$10 in a straight-up bet and wins, the player gets \$350 back plus the original \$10).

The second type of bet we will consider is a **colour bet**. Each number is associated with a colour (red or black). In a standard game, for numbers in the range **[1,10] or [19,28]**, **odd numbers are red** and **even numbers are black**. For numbers in the range **[11,18] or [29,36]**, the **odd numbers are black** and the **even numbers are red** (you can see this in the diagram to the right). The number 0 is green. The player can bet that either a red number will come up or a black number will come up. The payoff from this type of bet is **1:1** (i.e. if the player bets \$10 and correctly predicts the colour of the winning number, he or she will receive a payout of \$10 plus the original \$10 bet).

The third type of bet is called a **"dozen" bet**. The player can bet that the winning number will be in one of the following ranges: **[1,12]**, **[13,24]**, or **[25,36]**. If the player is right, the payoff for this kind of bet is **2:1** (so if the player bets \$10 and predicts the correct dozen, he or she will receive a payout of \$20 plus the original \$10 bet).

		0			
1 to 18	EVEN	1st 12	1	2	3
			4	5	6
			7	8	9
RED	2nd 12		10	11	12
			13	14	15
			16	17	18
BLACK	3rd 12		19	20	21
			22	23	24
			25	26	27
ODD	19 to 36		28	29	30
			31	32	33
			34	35	36
			2 to 1	2 to 1	2 to 1

There are all sorts of other bets available in the real game, but we will not consider them here.

Our implementation will contain circuitry that keeps track of the player's money, allows the user to bet, comes up with a "random" winning number (since a roulette wheel is unfortunately not part of a standard DE2 board, we will use a different technique to choose the winning number), calculates whether each bet wins or loses, and pays off the user appropriately. To make the lab achievable in two weeks, we will make the following simplifications:

1. As described earlier, we will assume the European version of the game, which does not have a 00 on the wheel. Thus, the **winning number can be any integer in the range [0,36]**.
2. We will assume only the three previously-mentioned types of bets. Further, we will assume that, before each spin, the user will make **exactly** three bets: bet #1 will **always** be a straight-up bet, bet #2 will **always** be a colour bet, and bet #3 will **always** be a dozen bet. Each bet is independent.
3. For each of the three bets, the **maximum amount that can be bet is \$7** (we are very low-rollers here in ENSC) and the **minimum bet is \$0** (effectively meaning no bet). Note that this means that the value of the bet can be represented by a **3-bit number**.
4. All inputs of our circuit (switches) will be in binary and all outputs will be in base-16 (on the hex digits). Although this may make it more challenging to use, it will make the coding a lot easier.
5. The user starts out with \$32.
6. You do not need to implement code that checks for the legality of a bet, or whether the user goes below \$0 (if you want to add code to do this, it is not hard, so go ahead).

To make the betting process more clear, consider the following example. Suppose the player starts with \$32 dollars (in base-16, that is 0x20). Before the spin, suppose the player makes the following three bets:

Bet 1: Straight-up Bet. The player bets \$3 that the number 10 will come up.

Bet 2: Colour Bet. The player bets \$4 that the winning number will be black.

Bet 3: Dozen Bet. The player bets \$2 that the winning number will be in the range [25,36].

Then the wheel is spun. Suppose the winning number (the number that comes up when the wheel is spun) is 10. In that case, the following is true:

Bet 1: The player wins this bet (lucky!) since he or she bet on 10, and a 10 came up. Since the original bet was \$3, and a straight-up bet pays off 35:1 as described on the first page of this handout, the player wins  $35 * \$3 = \$105$  from bet #1.

Bet 2: The player is extra lucky, since the winning number 10 is black (see the discussion on the previous page) and he or she had bet \$4 on black coming up. Since the payout of a colour bet is 1:1, the player wins \$4 from bet #2

Bet 3: In this case, the bet did not win, since the winning number 10 is not in the range [25, 36]. Therefore the player loses his or her \$2 bet.

After all the bets are counted, the player gets \$105 from bet 1, gets \$4 from bet 2, and loses \$2 from bet 3. Since the initial balance was \$32, the new balance is  $32 + 105 + 4 - 2 = \$139$ . Encouraged by his or her success, the player could then play another spin (and another and another...).

Since you have taken many math courses, you can use the payoffs from each bet as described above, along with the odds of each bet winning (which you should be able to determine) to estimate the expected payoff of this game, played over a long time. If you do this, you'll see that, over a long time, you'll lose. This game has among the worst return at the casino. In Las Vegas, someone has to pay for the simulated pirate fights, the exploding volcano, the lion farm, the "free" drinks, and the watershows... don't let it be you. =)

The overall design is as shown in the following diagram. There are two clocks in this design: **slow\_clock** is controlled by the user using **Key 0** (the rightmost push-button on your board). This **slow\_clock** is the main clock in your system (don't forget to **debounce** it). There is also a **fast\_clock**, which is *only* used to clock the **spinwheel** block. This **fast\_clock** is connected to **CLOCK\_50**, which is a 50MHz clock.

The operation of the system is as follows. The game starts when the player presses **resetb** (Key 1). When this happens, the **12-bit register (labeled 10 in the diagram)** is **loaded with \$32**, which is the initial amount of money the player starts with. **All other registers (labeled 2, 3, 4, 5, 7, 8, 9 in the diagram) are initialized to zero.**

The user places the following quantities on the switches:

Bet 0: Amount to bet (\$0-\$7): Switches 2-0  
Number to bet on (0-36): Switches 8-3

Bet 1: Amount to bet (\$0-\$7): Switches 9-11  
Colour to bet on (0-1): Switch 12 [red=1, black=0]

Bet 2: Amount to bet (\$0-\$7): Switches 15-13  
Dozen to bet on (0,1 or 2): Switches 17-16

Meanwhile, block **spinwheel** is continuously cycling through potential spin results, clocked at 50MHz (**fast\_clock**). The **6-bit bus, spin\_result, thus changes every 1/50MHz.**

On the next rising clock edge of **slow\_clock** (controlled using Key 0), the switch values are read into registers 3, 4, 5, 7, 8, 9. At the same time, the value on **spin\_result** (which is changing very quickly) is latched into the register labeled 2 (to produce signal **spin\_result\_latched**). Examine the diagram carefully to be sure you understand why this happens.

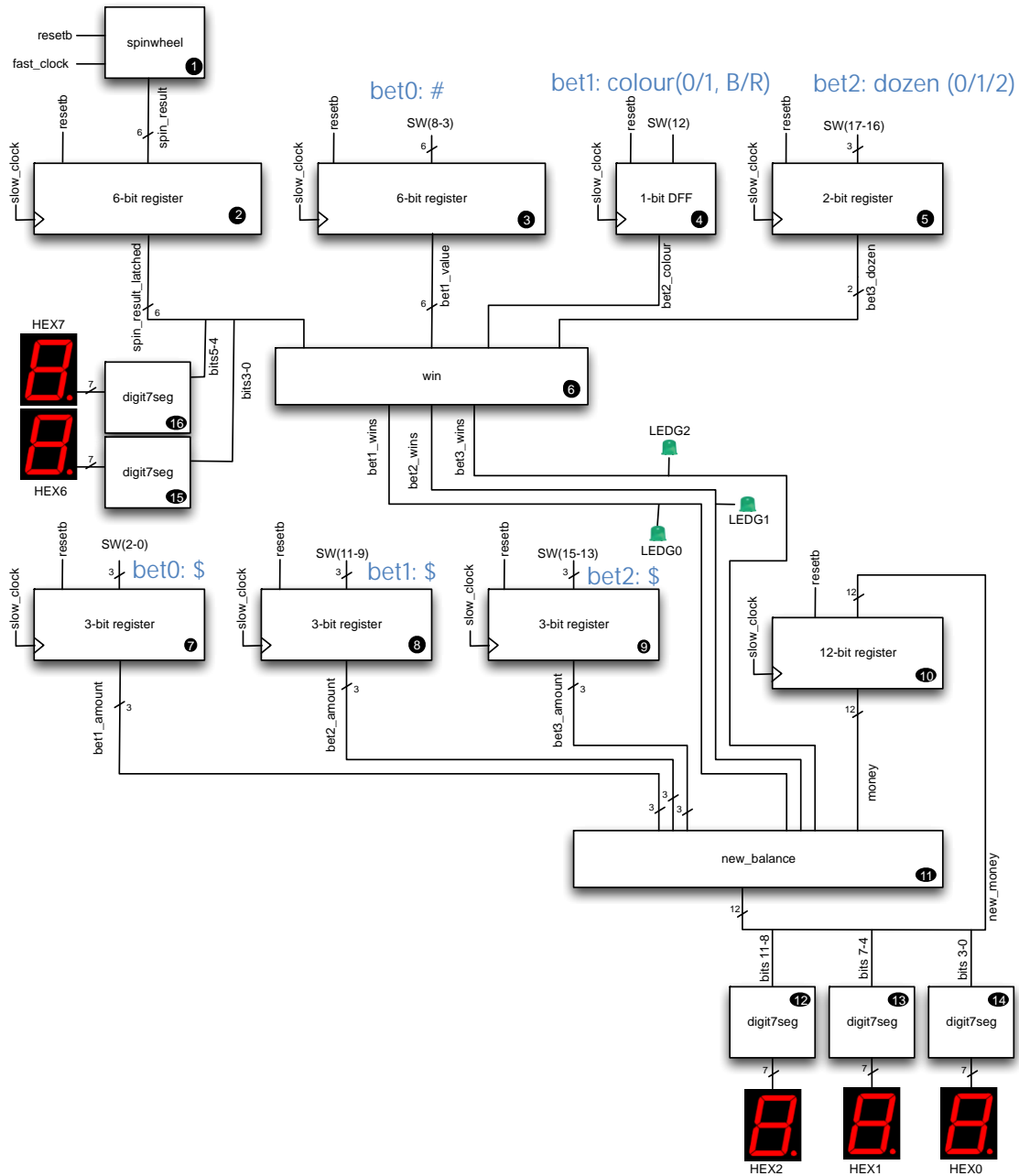
The signal **spin\_result\_latched** is the “winning number” and it is **held constant until the next rising edge of slow\_clock**. In this way, the “winning number” is random; the value on **spin\_result** changes very quickly, and the value latched depends on when exactly the user presses **slow\_clock**.

After the rising clock edge, combinational block **win** (labeled 6 in the diagram) looks at information from all the bets (**bet1\_value**, **bet2\_colour**, **bet3\_dozen**) as well as the winning number (**spin\_result\_latched**) and determines whether each bet 1-3 is a winner. This block asserts **bet1\_wins** if bet 1 is a winner, **bet2\_wins** if bet 2 is a winner, and **bet3\_wins** if bet 3 is a winner (clearly, there may be more than one winner so multiple output signals may go high at the same time, or there may be no winner at all).

The combinational logic block **new\_balance** (labeled 11) then uses that information to adjust the amount of money the player has (which is stored in register 10). For each bet, if the bet is a loser, the bet amount is subtracted from the balance. If the bet is a winner, the payout amount is added to the balance. The updated balance appears on bus **new\_money** (and will be stored in register labeled 10 at the next rising edge of **slow\_clock**).

Notice that the value of **new\_money** is displayed on HEX2, HEX1 and HEX0 using three instances of combinational block **digit7seg** (blocks 12, 13, and 14). Note that each digit is driven by 4 bits of **new\_money**; 4 bits corresponds to one hex digit. This means that the output will be in base 16 (hexadecimal). In the challenge task, you can change this to base 10 output if you want. Similarly, the value of **spin\_result\_latched** is displayed on HEX7 and HEX6 (note that this quantity is only 6 bits, so **the higher-order two bits of the 4-bit bus that is input to block 16 should be tied to 0**).

In the following tasks, you will first design and test the combinational logic blocks in isolation. Then, you will combine the blocks with registers and the **spinwheel** block (which we will give you) to create a working system. Note that we are providing a lot of details in this lab writeup. This may be the first big datapath circuit you have designed, so we are purposely making it easy for you. In later labs, there will be less detail in the writeup.



## PHASE TWO: BUILDING A WIN DETECTOR

**TASK 2.1:** We will start out easy. In this task, you will build and test the **win** sub-block. Given the set of bets placed by the player, and the winning number that has come up on the wheel, this combinational block (no clock) determines whether each of the three bets was a winner.

The entity part of the block showing the inputs and outputs is as follows (a skeleton file is provided as part of this lab folder):

```
ENTITY win IS
    PORT (spin_result_latched : in unsigned(5 downto 0); -- result of the spin
          bet1_value : in unsigned(5 downto 0); -- value for bet 1
          bet2_colour : in std_logic; -- colour for bet 2
          bet3_dozen : in unsigned(1 downto 0); -- dozen for bet 3
          bet1_wins : out std_logic; -- whether bet 1 is a winner
          bet2_wins : out std_logic; -- whether bet 2 is a winner
          bet3_wins : out std_logic); -- whether bet 3 is a winner
END win;
```

Notice that there are four inputs: the **spin\_input\_latched** encodes the result of the latest spin (since the result can be between 0 and 36, a 6-bit number is needed). The next three inputs describe the three bets the player made. Recall that bet 1 is a “Straight up” bet in which the player has to predict the exact number that will come up. The input **bet1\_value** is the value that the user predicted. Since this value can be between 0 and 36, a 6-bit number is needed. Recall that bet 2 is a “colour” bet in which the user has to predict the colour of the number that will come up. The input **bet2\_colour** indicates the colour that the player has predicted. Since there are only two colours the user can bet on, this is a one-bit signal, with ‘1’ meaning ‘red’ and ‘0’ meaning black. Recall that bet 3 is a “dozen” bet in which the user has to predict in which of three ranges the winning number will lie. The input **bet3\_dozen** indicates which range the user has predicted. Since there are three ranges possible, this input is 2-bits wide, and is encoded as follows: “00” means the user has predicted the winning number will be in the range [1,12], “01” means the user has predicted the winning number will be in the range [13,24], and “10” means the user has predicted the winning number will be in the range [25,36]. The value “11” should never happen.


There are three outputs of the block: **bet1\_wins** indicates whether bet 1 is a win (i.e. if **bet1\_value** is the same as **spin\_result**). The output **bet2\_wins** indicates whether bet 2 is a win, and **bet3\_wins** indicates whether bet 3 is a win.

Using this information as well as the information on the first page that describes the board setup, complete this block, starting the skeleton **win.vhd** you can find in this lab folder.

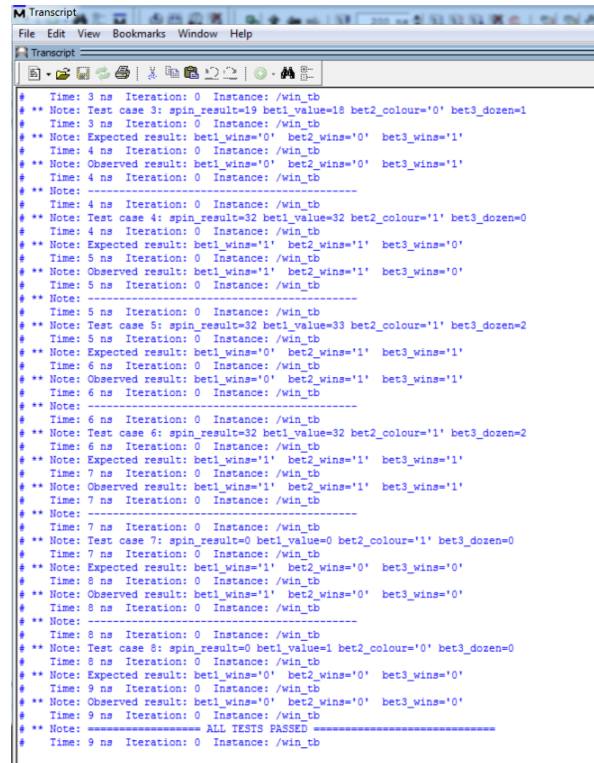
**TASK 2.2:** In this task, you will test your **win** implementation using Modelsim. Although you may be tempted to jump right to Quartus II, this task will introduce you to text-based debugging, which will make your life much easier. Trust me, don’t skip this step.

Download the testbench **win\_tb.vhd** from the folder. Look at this file carefully. You will see an array defined near the top of the file that describes each of the test vectors. The body of the testbench cycles through all elements of this array, applying input test vectors, waiting a while for the circuit to respond (arbitrarily 1ns) and then checking the outputs. Notice that the output is printed to the transcript window (you are not going to need to look at waveforms when you run this simulation).

Once you understand how the testbench works, create a new Modelsim project containing your **win** block and the testbench (you do *not* have to modify this testbench unless you want to add more test vectors). Compile both designs. Don’t be surprised if your win block doesn’t compile perfectly the first time; when working with numbers, it is easy to get type casts wrong (understanding this is one of the learning objectives of this lab!). Don’t continue until both files compile without errors.

This testbench sends its outputs to the transcript window. Normally, the transcript window is at the bottom of your Modelsim window, and is quite small (too small to be useful). You can expand the transcript window to a separate window using the  button. Be sure to do this before running the simulation.

Set the maximum run-time to at least 10ns (nothing wrong with making it longer). Now run the simulation. You do *not* need to add signals to the waveform window, since you are not going to use the waveform window in this task. Instead, you will use the transcript window to determine whether your design works. If it works, you should see something like:



```

# Time: 3 ns Iteration: 0 Instance: /win_tb
# ** Note: Test case 3: spin_result=19 bet1_value=18 bet2_colour='0' bet3_dozen=1
# Time: 3 ns Iteration: 0 Instance: /win_tb
# ** Note: Expected result: bet1_wins='0' bet2_wins='0' bet3_wins='1'
# Time: 4 ns Iteration: 0 Instance: /win_tb
# ** Note: Observed result: bet1_wins='0' bet2_wins='0' bet3_wins='1'
# Time: 4 ns Iteration: 0 Instance: /win_tb
# ** Note: -----
# Time: 4 ns Iteration: 0 Instance: /win_tb
# ** Note: Test case 4: spin_result=32 bet1_value=32 bet2_colour='1' bet3_dozen=0
# Time: 4 ns Iteration: 0 Instance: /win_tb
# ** Note: Expected result: bet1_wins='1' bet2_wins='1' bet3_wins='0'
# Time: 5 ns Iteration: 0 Instance: /win_tb
# ** Note: Observed result: bet1_wins='1' bet2_wins='1' bet3_wins='0'
# Time: 5 ns Iteration: 0 Instance: /win_tb
# ** Note: -----
# Time: 5 ns Iteration: 0 Instance: /win_tb
# ** Note: Test case 5: spin_result=32 bet1_value=33 bet2_colour='1' bet3_dozen=2
# Time: 5 ns Iteration: 0 Instance: /win_tb
# ** Note: Expected result: bet1_wins='0' bet2_wins='1' bet3_wins='1'
# Time: 6 ns Iteration: 0 Instance: /win_tb
# ** Note: Observed result: bet1_wins='0' bet2_wins='1' bet3_wins='1'
# Time: 6 ns Iteration: 0 Instance: /win_tb
# ** Note: -----
# Time: 6 ns Iteration: 0 Instance: /win_tb
# ** Note: Test case 6: spin_result=32 bet1_value=32 bet2_colour='1' bet3_dozen=2
# Time: 6 ns Iteration: 0 Instance: /win_tb
# ** Note: Expected result: bet1_wins='1' bet2_wins='1' bet3_wins='1'
# Time: 7 ns Iteration: 0 Instance: /win_tb
# ** Note: Observed result: bet1_wins='1' bet2_wins='1' bet3_wins='1'
# Time: 7 ns Iteration: 0 Instance: /win_tb
# ** Note: -----
# Time: 7 ns Iteration: 0 Instance: /win_tb
# ** Note: Test case 7: spin_result=0 bet1_value=0 bet2_colour='1' bet3_dozen=0
# Time: 7 ns Iteration: 0 Instance: /win_tb
# ** Note: Expected result: bet1_wins='1' bet2_wins='0' bet3_wins='0'
# Time: 8 ns Iteration: 0 Instance: /win_tb
# ** Note: Observed result: bet1_wins='1' bet2_wins='0' bet3_wins='0'
# Time: 8 ns Iteration: 0 Instance: /win_tb
# ** Note: -----
# Time: 8 ns Iteration: 0 Instance: /win_tb
# ** Note: Test case 8: spin_result=0 bet1_value=1 bet2_colour='0' bet3_dozen=0
# Time: 8 ns Iteration: 0 Instance: /win_tb
# ** Note: Expected result: bet1_wins='0' bet2_wins='0' bet3_wins='0'
# Time: 9 ns Iteration: 0 Instance: /win_tb
# ** Note: Observed result: bet1_wins='0' bet2_wins='0' bet3_wins='0'
# Time: 9 ns Iteration: 0 Instance: /win_tb
# ** Note: ----- ALL TESTS PASSED -----
# Time: 9 ns Iteration: 0 Instance: /win_tb

```

Each test vector is separated by “-----” lines, and for each, the applied inputs, expected outputs, and actual outputs are shown (see the testbench **win\_tb.vhd** to see the code that prints this information). The testbench is set up to fail (and stop) as soon as the first failing test vector is encountered, and it will print an error message. If you see the “ALL TESTS PASSED” statement, that means the simulation has made it to the end, and you *likely* have a working design (although, to be really sure, you need a lot more test vectors). If you see an error message, you have some debugging to do.

## PHASE THREE: BUILDING THE NEW\_BALANCE SUBBLOCK

**TASK 3.1:** In this task, you will build and test the **new\_balance** sub-block. This block is a combinational block that, given the current balance (amount of money the player has), along with information about each bet that has just been played, computes the new balance (the new amount of money the player has).

The entity part of the description (which shows the inputs and outputs) is below. A skeleton file is available in the lab folder.



```

ENTITY new_balance IS
    PORT (money : in unsigned(11 downto 0);
          value1 : in unsigned(2 downto 0);
          value2 : in unsigned(2 downto 0);
          value3 : in unsigned(2 downto 0);
          bet1_wins : in std_logic;
          bet2_wins : in std_logic;
          bet3_wins : in std_logic;
          new_money : out unsigned(11 downto 0));
END new_balance;

```

The algorithm that this block should implement is as follows (*this is pseudo-code, not VHDL*).

```

if bet1_wins = '1' then
    balance := balance + 35*value1
else
    balance := balance - value1
end if

if bet2_wins = '1' then
    balance := balance + value2
else
    balance := balance - value2
end if

if bet3_wins = '1' then
    balance := balance + 2*value3
else
    balance := balance - value3
end if

```

Starting with the skeleton file **new\_balance.vhd**, which you can find in the lab folder, write a VHDL description of the block as described above. Some hints that might help you:

1. Remember that assignments to signals only happen at the end of a process. You likely want to use variables here (remember slide set 4; use that to figure this out).
2. In my implementation of **new\_balance**, I had only one process. If you prefer (I would not), you can instead create a structural description with adders, multipliers, subtractors, and multiplexers.
3. Remember that you can use the **+** operator to add signals or variables (or the **-** operator to subtract signals or variables) of type **unsigned**. For example, if **a**, **b**, and **c** are each 4 bit unsigned values, you can write the following to create a 4-bit adder:

```
c := a + b;
```

You need to make sure that **a**, **b**, and **c** have the same length (Quartus II will let you get away with being lazy, but Modelsim won't, and you want portable code).

4. You can do the same with the **\*** operator, however, there is something to be aware of. Consider the operation

```
c := a * b;
```

If all quantities are unsigned, then the width (in bits) of **c** must be the width of **a** *plus* the width of **b**. So for example, if **a** and **b** are each defined with a width of 4 bits, then the width of **c** must be defined as exactly 8 bits. This might seem strange at first, but work out a few examples on paper, and you will see that it is always sufficient to have the width of a multiplication output to be equal to the sum of the widths of its operands.

5. If one of the operands is a literal, things are a bit different. Suppose you have

```
c := 36 * b
```

In this case, 36 is a literal (a number), not a signal. In that case, the width of **c** is *equal* to the width of **b**. That might seem very strange, and it is. This can cause problems because suppose **b** is 3 bits (the amount of bet1) and **c** is the expected payoff. In this case, it would expect **c** to have a width of only 3 bits, which is not large enough to store the product, so you will get truncation. This is bad. One way of dealing with this is to convert the constant 36 to an unsigned value of the

appropriate width. In the above example, suppose **b** is defined to be 3 bits wide, and **c** is defined to be 12 bits wide. In that case, you could write:


```
c := to_unsigned(36, 9) * b
```

The call to **to\_unsigned** returns an unsigned representation for the number 36 that is 9 bits long.

If you don't like hardcoding bitwidths, you could be even more fancy:

```
c := to_unsigned(36, c'length - b'length)
```

Quartus II is somewhat forgiving, but Modelsim requires you to get it exactly right. Since portable code is your goal, you should make sure that Modelsim compiles and simulates your designs correctly. In the next task, you will simulate using Modelsim to make sure you are not making any of these mismatch or truncation errors.

**TASK 3.2:** Open the testbench **new\_balance\_tb.vhd** from your lab folder. If you examine the testbench, you will see that, like Task 2, this testbench sends it outputs to the transcript window. Create a new Modelsim project containing **new\_balance\_tb.vhd** (you do not have to change this file unless you want to add more tests) and your **new\_balance.vhd**. Set the maximum run-time to at least 10ns (nothing wrong with making it longer). Compile and simulate the project, and observe the results in the transcript window, remembering to open the transcript window as wide as possible (you can expand the transcript window to a separate window using the  button). As in Task 2, you will not be viewing waveforms in this task... text is easier. The testbench we provide first performs a number of tests. If any test fails, the simulation stops. Examine the output and make sure every test passes, and the simulation ends with the message "ALL TESTS PASSED".

During your simulation, you may see errors or warnings related to operator width mismatch or truncation. You need to fix these. Have a look at the discussion at the end of Task 3.1 to help find your errors. I anticipate that most people will spend some time debugging at this point. Do not go on if your design does not pass our Modelsim testbench. Be able to tell your TA what additional Tests you might add to the testbench to verify your circuit.

## PHASE FOUR: BUILD THE digit7seg BLOCK:

As shown in the diagram earlier in this document, each instance of digit7seg converts a 4-bit input to a 7-bit bus that can drive one of the HEX digits. This is a combinational decoder, similar to what you designed in Lab 1. If the value of the four-bit input is 0-9, the corresponding digit should be displayed on the seven segment display. If the value of the four bit input is 10, 11, 12, 13, 14, or 15, the seven-segment display should show A, b, C, d, E, F respectively.

The entity description of this block is as follows:

```
ENTITY digit7seg IS
    PORT (
        digit : IN  UNSIGNED(3 DOWNTO 0);
        seg7  : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
    );
END;
```

Using the skeleton file **digit7seg.vhd** found in the folder, complete this block. First, create a testbench with a series of tests using Modelsim. We aren't giving you a testbench for this module, but you can use the testbenches from the previous tasks as a template. Your testbench should provide *complete* coverage for all possible input combinations (but not all the possible sequences of input combinations). After your design passes all the tests, you should verify it using the DE2 board.



## PHASE FIVE: PUT IT ALL TOGETHER

Go back and look at the schematic earlier in this document. You have now designed all the combinational blocks. You can download **spinwheel.vhd** from the lab1 folder (be sure to study it so you know how it works). Combine these blocks along with the required registers (you know how to make a register from class) in a structural description. You can start with the skeleton file **roulette.vhd** that you can also find in the same *task5* subfolder as **spinwheel.vhd**.

Import pin assignments, compile and download your design to the DE2, and test it. Remember: When you are testing, remember that the HEX displays are displaying the balance and winning number in base-16.

### CHALLENGE TASK:

Challenge tasks are tasks that you should only perform if you have extra time, are keen, and want to show off a little bit. This challenge task is only worth 1 mark, but is more work than the other tasks. If you don't demo the challenge task, the maximum score you can get on this lab is 9/10 (which is still an A+).

Playing the game in base-16 is fine for engineers, but it won't fly in Vegas. Change the code so that it outputs both the winning number and the balance in base 10. (NOTE: You will also need to add an additional base-10 seven-segment display if you want to do this as newbalance stores 4096 different values).

### MARKING OF LAB 2:

During the second week of the lab (Jan 29-Feb 2, 2017), you will be marked by the TA. There are two components to the marking:

1. In-person demonstration: you must demonstrate your working circuit(s) (see below). Your TA will ask you questions to assess your knowledge of the material from this lab. You will be evaluated during the same demo time slot you signed up for from Lab 1's marking week
2. After your demo, you must submit your code to the Lab 2 submission page on Canvas. Your code must be submitted *no later than 24 hours after your demo*. You must ensure you submit exactly the code that you demoed; submitting code other than the code you demo is considered academic misconduct. You should submit all of the .vhd files that you have written.

If you got the entire design working (with or without the challenge task), demonstrate that to the TA and submit all VHDL files. Do not submit temporary files, or .vhd files that you did not modify. Please do not zip up your entire working directory and submit it; if everyone does this, it takes too much space. Only submit the .vhd files.

If you didn't get the whole thing working, part marks will be given based on the following table:

Demonstrating that the **win** block works correctly: 3  
Demonstrating that the **new\_balance** block works correctly: 2  
Demonstrating that the **digit7seg** block works correctly: 1

This means that if you get the parts working in isolation, but nothing else, you can get 6/10 on the lab. If you can demonstrate part of the datapath working, you might get more part marks for that.

You are doing this lab in pairs; however, both people need to submit the code on Canvas. Both members of the group must also be present during the in-person demonstration (*if one person is not present, that person will not receive marks for this lab*).