Lab 2

Binary-to-BCD

The purpose of this lab is to build a Binary-to-BCD converter using the double dabble algorithm (aka the shift-add-3 algorithm). You will only utilize combinational circuit components, verify the functionality of the converter using a testbench, and test the converter on the FPGA board by using it to display the output of a simple calculator.

Part 1 (Building a Binary-to-BCD Converter)

Binary Coded Decimal (BCD) is a numbering system used to represent decimal numbers using binary representations. Each BCD digit is composed of 4 bits and can represent a number from 0 to 9. Table 1 shows how to represent all decimal digits using BCD, the table also indicate the other six binary numbers (i.e. 1010, 1011, etc.) are not used.

Table 1: BCD Representation

Decimal Digit	BCD Representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Decimal numbers with multiple digits require 4-bit BCD numbers for each of the digits. Table 2 shows an example of using BCD to represent a 3 digits decimal number. The same concept can be applied to decimal numbers with more digits.

Table 2: Multidigit Decimal Number in BCD

	Hundreds	Tens	Ones
Decimal	3	9	7
BCD	0011	1001	0111

Many arithmetic operations are done in binary, then converted to BCD for display purposes (i.e. displaying the result on a seven-segment or LCD display). In this lab you will build a circuit that takes an 8-bit binary number and converts it to its 3 digits BCD representation.

When converting a binary number to its BCD reperesentation, more than one BCD digit locations might be required. For example, the binary number 1100 represent the decimal number 12, to convert this 4-bit binary number to its BCD representation, we first note that it is greater than 1001 (9 decimal), so we have to add 0110 (6 decimal), the result is 10010 which is equivalent to 0001 0010 the BCD representation of 12.

A similar computation could be done using shifts. Recall that a left shift moves all bit values one position to the left (toward the MSB). A left shift has the same function as multiplying the binary number by two. If we have the number 0101 (5 decimal), and we shift it left, we know the result will be 1010 (10 decimal), assuming we shift in a zero. If we then want to convert this number to a BCD, we will have to add six to it. Knowing that we will convert the number to BCD, we can check the original number before shifting it. If it is greather than or equal to five, then we can add three to that number (instead of adding 6 to the shifted number). Then when we shift it, it will automatically be in BCD form. Table 3 illustrates this concept using the number 0110.

Table 3: Preemptive add, then shift example

Operation	**Binary**
Original number	0110
Add 3	1001
Shift Left	0001 0010
Decimal	1 2

This concept can be used on a binary number with any number of bits. For each bit in the binary number, the shift-add-3 algorithm must be implemented. For a four bit number, four shifts must be performed. Before each shift is performed, if the number is greater than or equal to five, then it must be added to three. Once all four bits have been shifted, the final BCD number will be in the upper-most bits of the new number. See the example in Table 4.

Table 4: Example of converting 1111 to BCD

Operation	**Tens**	**Ones**	**Binary**
Start			1111
Shift		1	111
Shift		11	11
Shift		111	1
Add-3		1010	1
Shift	1	0101	
Decimal	1	5	

To perform a 4-bit binary conversion, 12 bits must be used. The upper four bits will be the tens decimal digit and the middle four will be the ones decimal digit. The lower four bits will be ignored. In the previous example, only the relevant bits were shown. In reality, the missing bit locations would contain zeros. This process can be extended for any size of binary number. Check Table 5 for another example using an 8- bit binary number.

Table 5: Example of converting 1111 1111 to BCD

Operation	Hundreds	Tens	Ones	Binary	
Start				1111	1111
Shift			1	1111	111
Shift			11	1111	11
Shift			111	1111	1
Add-3			1010	1111	1
Shift		1	0101	1111	
Add-3		1	1000	1111	
Shift		11	0001	111	
Shift		110	0011	11	
Add-3		1001	0011	11	
Shift	1	0010	0111	1	
Add-3	1	0010	1010	1	
Shift	10	0101	0101		
Decimal	2	5	5		

Implementation Steps

Table 6: Add-3 module

A[3:0]	S[3:0]
0000	0000
0001	0001
0010	0010
0011	0011
0100	0100
0101	1000
0110	1001
0111	1010
1000	1011
1001	1100
1010	XXXX
1011	XXXX
1100	XXXX
1101	XXXX
1110	XXXX
1111	XXXX

The double dabble algorithm can be used to convert binary numbers of arbiterary length. In fact, the double dabble's wikipedia article contains a parameterized verilog implementation that can be used with binary numbers of any length. You should not use that code. Instead you should develop a version that converts an 8-bit binary numbers into their equivalent BCD. Of course an 8-bit binary numbers should result in BCDs ranging from 000 to 255.

The double dabble can be completely implemented using combinational circuit elements, without using storage elements nor registers. Table 6 and figure one contain the details of the implementation.

The following steps should guide you through the implementation.

1. Create a module (call it `add_3`) to implement the add-3 part of the algorithm. Follow the truth table shown in Table 6. The module should have 4-bit input (`A`) and 4-bit output (`S`). The truth table shows that when the number in the input is less than 4, it is passed as is, when the number exceeds 4, the output will be the result of adding 3 to the input. The algorithm

will not allow any number above 9, so the implementation treats those inputs as don't care.

2. Shifting numbers in combinational circuits is done by connecting the inputs to the outputs while shifting by one position. For example, a left shift can be achieved by connecting the least significant bit to the second least significant bit, and the same for the rest of the wires. You should also connect 0 to the least signficiant bit of the output.
3. Create a module (call it `bin2bcd`) that implements an 8-bit binary to BCD converter. Figure 1 shows the diagram for this module. The module should have 8-bit input (`bin`) and 12-bit output (`bcd`). You should instantiate as many instances of `add_3` as necessary.
4. Verify the functionality of `bin2bcd` by writing a testbench (`bin2bcd_tb`) to test its functionality .
5. Include a screenshot of the simulation output with your submission, you should embed the screenshot in your README.md file.

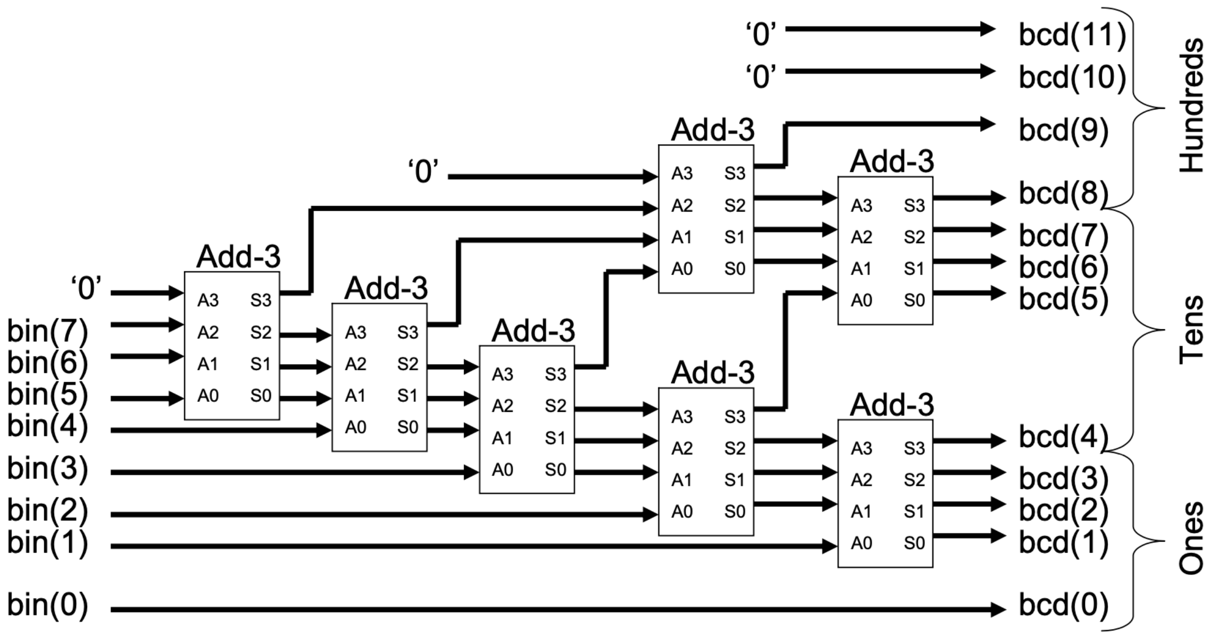

Figure 1: 8-bit implementation of the double dabble binary to BCD converter

Part 2 (Building a simple calculator with BCD outputs)

In this part, you will modify the `simple_calc` module you developed in a previous lab so that it displays its output as BCD instead of binary. Recall, the calculator should perform 4-bit addition, subtraction, and multiplication.

1. Import `simple_calc` along with all necessary dependencies (i.e. the adder/subtractor, csa multiplier,etc.)

2. Modify the module so it displays its result on 12 LEDs as BCD. You should utilize the `bin2bcd` module you developed in part 1.
   As a reminder
   a. The module should accept two 4-bit inputs `X, Y`
   b. The module should accept a 2-bit operator select input (`op_sel`).
      i. `op_sel` = 00 (add)
      ii. `op_sel` = 01 (subtract)
      iii. `op_sel` = 1x (multiply)
   c. The module should output a 12-bit signal (`result`) represented in BCD
3. The `bin2bcd` module you developed in part 1 assumes positive numbers only. You should figure out what needs to be done if the answer of the `simple_calc` is being interpreted as a signed number. For example, you might need to take its 2's complement before passing it to the converter and use one of the LEDs to indicate the answer is negative.
4. Verify the functionality of your simple calculator by implementing in on the FPGA board using the following IO specifications:
      i. SW3 ← SW0 for the input `X`
      ii. SW7 ← SW4 for the input `Y`
      iii. SW15 ← SW14 for the `op_sel`
      iv. LED11 ← LED0 for the output `result` (displayed as BCD)
      v. LED13 to indicate the result displayed is negative
      vi. LED14 to display the `carry_out` of the adder/subtractor
      vii. LED15 to display the `overflow` of the adder/subtractor

Submission check list:

[ ] All Verilog code you generated or modified
[ ] All testbenches written
[ ] Include a screenshot of your testbench output, Embedded in your README.md
[ ] Include a sketch of the block diagram used in Part 2, no need to include the details of the simple calc, just draw it as a blackbox along with its inputs and outputs. Embedded in your README.md
[ ] Short video demonstrating a working calculator