Learning Objectives:

- Practice logic simulation using *ModelSim*
- Practice coding combinational logic with gate-level models
- Practice coding combinational logic with dataflow models

Homework 1

Assignment: Coding Combinational Logic in Verilog

The purpose of this assignment is to practice gate-level modeling and creating and instantiating modules in Verilog. You will also gain experience creating hierarchical Verilog models and running a test bench to check the functionality of your design. The project consists of completing several small design projects resulting in a 4-bit BCD (binary coded decimal) adder.

This assignment will be graded.

Functional Specification

EXCERPTS ARE TAKEN ALMOST VERBATIM FROM WIKIPEDIA.ORG. THE FULL ARTICLE IS INCLUDED IN THE RELEASE PACKAGE FOR THE ASSIGNMENT

The functional description for the assignment is easy to state. Implement a 4-bit BCD adder. The inputs to the BCD adder are two unsigned 4-bit BCD digits. The output of the adder is a single byte, packed BCD result. Since the largest sum of two 4-bit unsigned BCD numbers is 18 (9 + 9) the result should fit nicely into a single byte (largest packed BCD number that can be represented in 8-bits is 99). Your adder design should include an `out-of-range` output that is asserted high whenever either of the two input numbers is greater than 9. You will design the 4-bit BCD adder so that wider BCD words can be processed by instantiating additional BCD adder blocks.

BCD Addition

In computing and electronic systems, **binary-coded decimal** (**BCD**) is a binary encoding of decimal numbers where each decimal digit is represented by a fixed number of bits, usually four or eight. BCD's main virtue is ease of conversion between machine- and human-readable formats, as well as a more precise machine-format representation of decimal quantities. As compared to typical binary formats, BCD's principal drawbacks are a small increase in the complexity of the circuits needed to implement basic mathematical operations and less efficient usage of storage facilities.

As most computers store data in 8-bit bytes, it is possible to use one of the following methods to encode a BCD number:

- **Uncompressed**: each numeral is encoded into one byte, with four bits representing the numeral and the remaining bits having no significance.
- **Packed**: two numerals are encoded into a single byte, with one numeral in the least significant nibble (bits 0-3) and the other numeral in the most significant nibble (bits 4-7).

As an example, encoding the decimal number **91** using uncompressed BCD results in the following binary pattern of two bytes:

```
Decimal:        9              1
Binary :  0000 1001  0000 0001
```

In packed BCD, the same number would fit into a single byte:

```
Decimal:    9    1
Binary :  1001 0001
```

Hence the numerical range for one uncompressed BCD byte is zero through nine inclusive, whereas the range for one packed BCD is zero through ninety-nine inclusive.

To represent numbers larger than the range of a single byte any number of contiguous bytes may be used. For example, to represent the decimal number **12345** in packed BCD, using big-endian format, a program would encode as follows:

```
Decimal:    0      1        2       3        4      5
Binary :  0000   0001    0010    0011    0100   0101
```

Note that the most significant nibble of the most significant byte is zero, implying that the number is in actuality **012345**. Also note how packed BCD is more efficient in storage usage as compared to uncompressed BCD; encoding the same number in uncompressed format would consume 100 percent more storage (3 bytes for packed BCD, 6 bytes for unpacked BCD).

Addition with BCD

It is possible to perform addition in BCD by first adding in binary, and then converting to BCD afterwards. Conversion of the simple sum of two digits can be done by adding 6 (that is, 16 – 10) when the result has a value greater than 9. For example:

```
1001 + 1000 = 10001 = 0001   0001
  9   +   8   =   17   =    1       1
```

In this example the result of the binary add is 17 (10001) which in BCD would be 11 which is incorrect. To correct this, 6 (0110) is added to that sum to get the correct first two digits:

```
0001 0001 + 0000 0110    = 0001 0111
  1      1   +   0      6    =   1      7
```

which gives two nibbles, 0001 and 0111, which correspond to the digits "1" and "7". This yields "17" in BCD, which is the correct result. This technique can be extended to adding multiple digits, by adding in groups from right to left, propagating the second digit as a carry, always comparing the 5-bit (4-bit sum and a carry out) result of each digit-pair sum to 9.

Hardware

There is no hardware required for this project. For this assignment you will have to limit your satisfaction to watching your skillfully crafted Verilog model simulate correctly.

Design Software

ModelSim PE Student Edition or QuestaSim or another logic simulator. Please indicate which simulator (product name and version) you used if you are not using ModelSim PE Student Edition

Project Tasks

NOTE: THIS ASSIGNMENT ASSUMES THAT YOU HAVE ALREADY INSTALLED MODELSIM AND HAVE WORKED THROUGH THE ASSIGNED CHAPTERS IN THE MODELSIM TUTORIAL. IF YOU HAVE NOT DONE SO, AND HAVE LITTLE OR NO EXPERIENCE WITH LOGIC SIMULATION IT WOULD BE WORTHWHILE TO DO SO BEFORE STARTING THIS ASSIGNMENT.

STEP 1: Download the Homework #1 release package

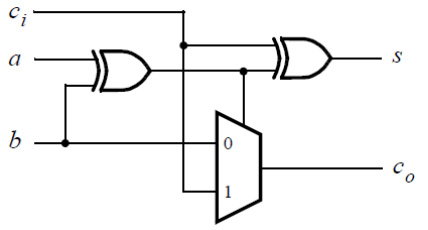
Download and unzip the release from the course website. This .zip file contains this document, copies of the relevant Wikipedia article(s) and an hdl directory containing *`hw1_starter.v`*. This file contains some module definitions and includes a testbench (*`stimulus.v`*) to verify your design. While most of the modules have no code in their body, there are two modules that you will find useful. `compare` takes compares two unsigned numbers and produces greater than (gt) and equal (eq) outputs. `mux2to1_nbits` implements an n-bit 2 to 1 multiplexer. Both modules accept a single parameter, `SIZE`, so you can specify the number of bits to compare or multiplex. Both of these modules make use of Dataflow modeling constructs which we have not discussed in class…yet.

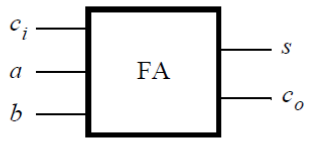
You may want to take a few minutes and separate each of the modules in *`hw1_starter.v`* into individual files with a single module per file (same file name as the module) since that is the convention with Verilog code.

STEP 2: Implement a 1-bit full adder module

We will implement our BCD adder by building a 1-bit full adder, building a 4-bit adder by instantiating four instances of the 1-bit adder and creating a third module with the logic to do the conversion to BCD. Since this design is not speed-constrained we will pick the simplest of the adder configurations – a ripple-carry adder. Palnitkar has an example of this (p 75-78) but there's not much learning in simply copying text from a book or cutting/pasting from the CD that comes with the book…so we will implement a variation.

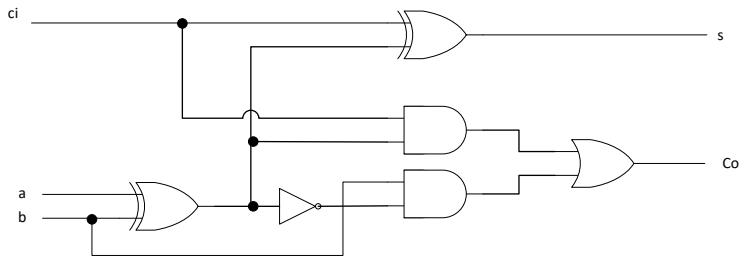
The schematic (courtesy Intel PSG) for a single bit full adder is shown below:





b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

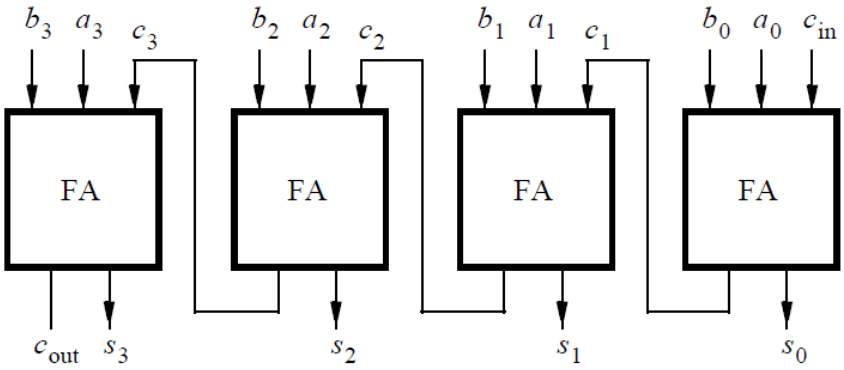
The trapezoid represents a 2 to 1 multiplexer which can be implemented in your design. A gate model of the one-bit adder cell is shown below:



Your first task is to implement a Verilog module `fulladd()` containing a single-bit full adder. Use Gate-level modeling as discussed in class. In this circuit `a` and `b` are the two bits to be summed, `ci` is the carry-in, `s` is the sum and `co` is the carry-out.

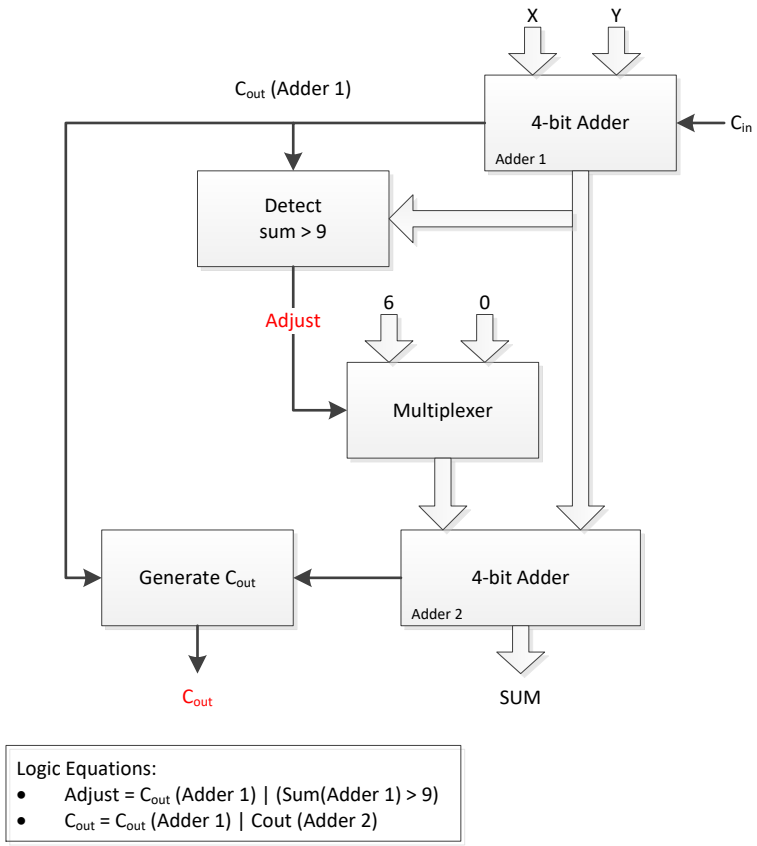
STEP 3: Implement a 4-bit binary ripple-carry adder

Create a new module `fulladd4()`that has as inputs `a[3:0]`, `b[3:0]`, and a single bit input `c_in`. The module produces a 4-bit sum `s[3:0]` and a single bit `c_out`. You may use the sample code in Palnitkar (p. 77) as the starting point for your implementation. A block diagram for this adder is:



STEP 4: Implement a 4-bit BCD adder

As the Wikipedia article points out, it is possible to adjust the results from a binary adder to produce BCD. Doing so requires detecting when the result of adding the two BCD digits is greater than 9 (4'b1001), and if so, adding 6 (4'b0110) to the result. A block diagram for this functionality is shown below:



Create a new module called `bcd_adder4()` that implements the block diagram. You can do this by instantiating two instances of `fulladd4()` and creating new logic for the sum > 9 detection, multiplexer, and C_{out} generation logic. The input to `bcd_adder4()`should be two 4-bit BCD values `X` and `Y`. The output of this module should be the carry-out from the block and an 8-bit packed BCD number representing the sum of `X` and `Y`. You can create the packed BCD output with the Verilog concatenation operation and a dataflow assign as shown:

`assign result = {3'b000, Cout, SUM}.`

This adder architecture only works with BCD numbers (0 to 9) so your module should flag illegal operations. You can do this by checking that both X and Y are <= 9 and if either of them is > 9, asserting an `out_of_range` output.

The Adjust logic and the out_of_range logic can be implemented by instantiating several compare modules and Verilog or gate primitives. The multiplexer that adds either 6 or 0 to perform the BCD correction can be implemented with an instance of mux2to1_nbits. If you have read ahead in Palnitkar or Bhasker you can also use the dataflow modeling constructs.

STEP 5: Simulate your design

Now that you have completed your design you're 60% of the way done (this is a straightforward application of Roy's rule ☺). Use the provided top level stimulus module to test whether your BCD adder works correctly. Study the test bench code because it is a pretty good way to write test benches for simple combinatorial logic. Note that this test bench is not of the "self-checking" variety where your implementation is compared to a functional "golden model." You are the "golden model" in that you should check the console log to make sure the BCD add is being performed correctly.

STEP 6: Submit your deliverables to your D2L Dropbox

One you have your simulation working and you've verified that all of your results are correct it's time for us to grade your work. Please put all of the Verilog source code that you write and a console log (a transcript) showing your successful simulation run into a single .rar or .zip to your Homework #1 Dropbox. Name the file something like `vlgwkspf19_rkravitz.zip`, using your name instead of mine. Please double check that you are submitting all of the files that you intended to and that they are the correct versions of the file. It seems like there are always a few students who want "do-overs" because they accidentally submitted the wrong files. With about 55 students enrolled in the course this term it is going to be challenging enough to grade your assignments and get them back to you in a reasonable time without having to deal with additional/different submissions.

References

[1] Wikipedia.org. Article on binary coded decimal (http://en.wikipedia.org/wiki/Binary_coded_decimal)

[2] Altera laboratory exercise 6
ftp://ftp.altera.com/up/pub/Altera_Material/Laboratory_Exercises/Digital_Logic/DE0/verilog/lab6_Verilog.pdf

[3] Palnitkar Chapter 6