"An Introduction to Verilog: Adders & Counters"

Points: 2% of class grade
Goal: Provide an introduction to the Verilog language as well as providing an opportunity to learn hierarchical design in Verilog.

In this project you will design three successive Verilog modules, each using the previous as a component. The first is 1-bit bit-sliced ALU. The second is a 4-bit bit-sliced ALU, while the last is a 4-bit counter/shift register. You are to follow the design rules provided on the course website. These will also be covered in discussion . Even if you already know Verilog it is *strongly* suggested you look over our tutorials and general Verilog directions.

The website also includes directions for the electronic hand-in of this assignment in as well as a testbench or the one-bit ALU.

One-bit ALU

You are to write a module using the following declaration. You must use this declaration exactly as written:

```
module alu1(func, opa, opb, cin, result, cout);
        input func;              // function code
        input opa;               // operand A
        input opb;               // operand B
        input cin;               // carry in
        output result;           // result output
        output cout;             // carry out
```

This module takes three inputs (opa, opb, and cin) and generates two outputs (result and cout) depending upon the value of func as follows:

func's value	result	cout
0	LSB of opa+opb+cin	MSB of opa+opb+cin
1	cin	opa

On the website you will find a Verilog "testbench" for this one-bit ALU. Use it to insure that your code works correctly. You may only use AND(&), OR(|), NOT(~), and XOR(^) operations to write your ALU.

Four-bit ALU

You are to write a module using the following declaration. You must use this declaration exactly as written:

```
module alu4(func, opa, opb, cin, result, cout);
        input func;                // function code
        input [3:0] opa;           // operand A
        input [3:0] opb;           // operand B
        input cin;                 // carry in
        output [3:0] result;       // result output
        output cout;               // carry out
```

Using your one-bit ALU you are to implement a 4-bit ALU that performs the following functions:

func	**Operation**
0	Add opa+opb+cin, having the 4 LSBs of the result appear in result while the MSB is cout.
1	left-shift: result[0]=cin, result[3:1]=opa[2:0], cout=opa[3].

You ***must*** use your 1-bit ALU to implement this 4-bit ALU! You also will need to write a testbench, modeled on the one provided for the 1-bit ALU, which tests your code. You will be graded on how well your testbench actually tests.

Four-bit counter/shift-register

You are to write a module using the following declaration. You must use this declaration exactly as written:

```
      module csr4(func, ld, in, clock, enable, result, out);
              input [1:0] func;          // function code
              input [3:0]ld;             // register load
              input in;                  // 1-bit input
              input clock;
              input enable;
              output [3:0] result;       // result output
              output out;                // output
```

On every rising clock edge when enable is asserted *result* and *out* are updated as follows:

func[1:0]	**Operation**
00	result=result +1. If this goes from 1111 → 0000 out=1, else out=0
01	result=result -1. If this goes from 0000 → 1111 out=1, else out=0
10	left shift Result[3:1]=Result[2:0],Result[0]=in, out=Result[3]
11	load: Result=ld. out=0

(Out is being used to indicate overflow of the counter or the bit that was shifted out of the register.)

You **must** use your 4-bit ALU to implement this (at least for func=00, 01 and 10). You also will need to write a testbench, modeled on the one provided for the 1-bit ALU, which tests your code. You will be graded on how well your testbench actually tests. Notice that this device has state, so testing this could be quite different.

You may assume the clock period will be at least 10 time units.