Homework 4

Please turn in all verilog code and a hardcopy of your simulation results. You can demonstrate proper operation by either a printout of the "monitor" output (the table printed once the simulation has been run) or by a printout of the waveforms (the results of some problems are easier to see one way and the results of other problems are easier to see another). **Be sure to annotate your simulation results telling me how your results prove that you have met all specifications.**

Problem 1:

Design a 4-bit arithmetic block that performs the following functions. You may only use muxes, NOR gates, and 4 full adders. Assume all inputs are 4-bit binary numbers in 2's complement form. (Hint: This problem should sound very familiar! Be sure to think about the hardware before implementing this design to ensure an efficient design.)

S1	S0	Function
0	0	A+B
0	1	A-B
1	0	A+1
1	1	A-1

You must implement and test each module as distinct and separate entities. Tie the pieces together in a top-level module and design a test for it as well. Each functional unit should have its own module and file -- try to reuse modules as often as possible.

Top level module (ALU.v):

```
module ALU( A, B, S, out );

input [3:0] A, B;
input [1:0] S;

output [3:0] out;

wire [3:0] out;


// invert B for negation
wire [3:0] nB;

NOT4 invertB( B, nB );


// wire up mux
wire [3:0] muxOut;

mux4to1x4bit mux( B, nB, 4'b0001, 4'b1110, S, muxOut );

// carry-in depends on function:
//    S = 0 : 0
//    S = 1 : 1
//    S = 2 : 0
//    S = 3 : 1

wire ci;

assign ci = S[0];


// wire up adder

adder4 hexAdder( .A(A), .B(muxOut), .ci(ci), .S(out), .co() );


endmodule
```

NOT4.v:

```
module NOT4( in, out );

input [3:0] in;
output [3:0] out;

wire [3:0] out;

NOR4 norblock4( in, 4'b0000, out );

endmodule
```

NOR4.v:

```
module NOR4( in1, in2, out );

input [3:0] in1, in2;
output [3:0] out;

wire [3:0] out;

NOR1 n0( in1[0], in2[0], out[0] );
NOR1 n1( in1[1], in2[1], out[1] );
NOR1 n2( in1[2], in2[2], out[2] );
NOR1 n3( in1[3], in2[3], out[3] );


endmodule
```

NOR1.v:

```
module NOR1( in1, in2, out );

input in1, in2;
output out;

wire out;

assign out = ~(in1 | in2);


endmodule
```

mux4to1x4bit.v:

```
module mux4to1x4bit( in0, in1, in2, in3, s, out );

input [3:0] in0, in1, in2, in3;
input [1:0] s;

output [3:0] out;

reg [3:0] out;

always @ ( in0, in1, in2, in3, s )
   case( s )
      0: out <= in0;
      1: out <= in1;
      2: out <= in2;
      3: out <= in3;
   endcase


endmodule
```

adder4.v:

```
module adder4( A, B, ci, S, co );

input [3:0] A, B;
input ci;

output [3:0] S;
output co;

wire [3:0] S;
wire co, co0, co1, co2, co3;

fullAdder FA0( A[0], B[0], ci, S[0], co0 );
fullAdder FA1( A[1], B[1], co0, S[1], co1 );
fullAdder FA2( A[2], B[2], co1, S[2], co2 );
fullAdder FA3( A[3], B[3], co2, S[3], co3 );

assign co = co3;


endmodule
```

fullAdder.v:

```
module fullAdder( a, b, ci, s, co );

input a, b, ci;
output s, co;

wire s, co;

assign {co,s} = a+b+ci;

endmodule
```

ALU_TB.v:

```
module ALU_TB;

integer j, k, l;

reg [3:0] A, B;
reg [1:0] S;
wire [3:0] out;

ALU aluBlock( A, B, S, out );

initial begin
   $monitor( $time, " S=%b : A=%b, B=%b --> out=%b",
             S, A, B, out );

   A = 0;
   B = 0;
   S = 0;


   #5
   for( j = 0; j < 4; j = j + 1 )
   begin
      #5 S = j;
      for( k = 0; k < 16; k = k + 5 )
      begin
         #5 A = k;
         for( l = 0; l < 16; l = l + 5 )
         begin
            #5 B = l;
         end
      end
   end

   #5
   $finish;
end

endmodule
```

RESULTS:

Expected output matches actual output:

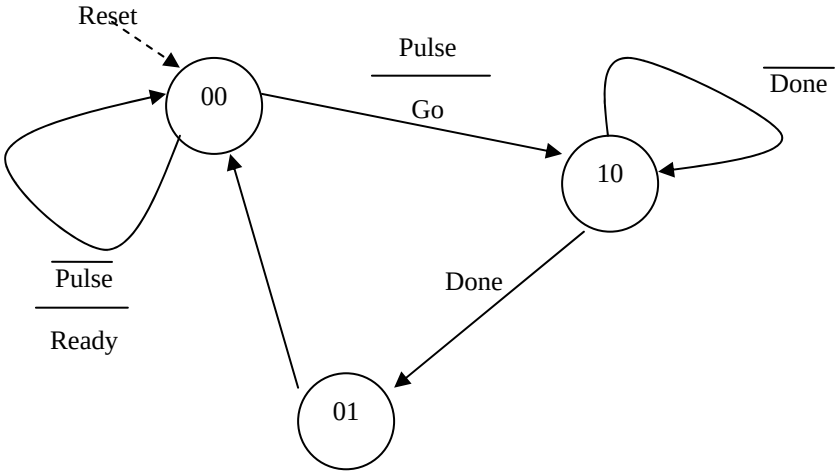
	Time	Select	Input A	Input B	Output	**Expected**
A+B	0	S=00	A=0000,	B=0000	out=0000	**0000**
	25	S=00	A=0000,	B=0101	out=0101	**0101**
	30	S=00	A=0000,	B=1010	out=1010	**1010**
	35	S=00	A=0000,	B=1111	out=1111	**1111**
	40	S=00	A=0101,	B=1111	out=0100	**0100**
	45	S=00	A=0101,	B=0000	out=0101	**0101**
	50	S=00	A=0101,	B=0101	out=1010	**1010**
	55	S=00	A=0101,	B=1010	out=1111	**1111**
	60	S=00	A=0101,	B=1111	out=0100	**0100**
	65	S=00	A=1010,	B=1111	out=1001	**1001**
	70	S=00	A=1010,	B=0000	out=1010	**1010**
	75	S=00	A=1010,	B=0101	out=1111	**1111**
	80	S=00	A=1010,	B=1010	out=0100	**0100**
	85	S=00	A=1010,	B=1111	out=1001	**1001**
	90	S=00	A=1111,	B=1111	out=1110	**1110**
	95	S=00	A=1111,	B=0000	out=1111	**1111**
	100	S=00	A=1111,	B=0101	out=0100	**0100**
	105	S=00	A=1111,	B=1010	out=1001	**1001**
	110	S=00	A=1111,	B=1111	out=1110	**1110**
A-B	115	S=01	A=1111,	B=1111	out=0000	**0000**
	120	S=01	A=0000,	B=1111	out=0001	**0001**
	125	S=01	A=0000,	B=0000	out=0000	**0000**
	130	S=01	A=0000,	B=0101	out=1011	**1011**
	135	S=01	A=0000,	B=1010	out=0110	**0110**
	140	S=01	A=0000,	B=1111	out=0001	**0001**
	145	S=01	A=0101,	B=1111	out=0110	**0110**
	150	S=01	A=0101,	B=0000	out=0101	**0101**
	155	S=01	A=0101,	B=0101	out=0000	**0000**
	160	S=01	A=0101,	B=1010	out=1011	**1011**
	165	S=01	A=0101,	B=1111	out=0110	**0110**
	170	S=01	A=1010,	B=1111	out=1011	**1011**
	175	S=01	A=1010,	B=0000	out=1010	**1010**
	180	S=01	A=1010,	B=0101	out=0101	**0101**
	185	S=01	A=1010,	B=1010	out=0000	**0000**
	190	S=01	A=1010,	B=1111	out=1011	**1011**
	195	S=01	A=1111,	B=1111	out=0000	**0000**
	200	S=01	A=1111,	B=0000	out=1111	**1111**
	205	S=01	A=1111,	B=0101	out=1010	**1010**
	210	S=01	A=1111,	B=1010	out=0101	**0101**
	215	S=01	A=1111,	B=1111	out=0000	**0000**
A+1	220	S=10	A=1111,	B=1111	out=0000	**0000**
	225	S=10	A=0000,	B=1111	out=0001	**0001**
	230	S=10	A=0000,	B=0000	out=0001	**0001**

	235	S=10	A=0000,	B=0101	out=0001	**0001**
	240	S=10	A=0000,	B=1010	out=0001	**0001**
	245	S=10	A=0000,	B=1111	out=0001	**0001**
	250	S=10	A=0101,	B=1111	out=0110	**0110**
	255	S=10	A=0101,	B=0000	out=0110	**0110**
	260	S=10	A=0101,	B=0101	out=0110	**0110**
	265	S=10	A=0101,	B=1010	out=0110	**0110**
	270	S=10	A=0101,	B=1111	out=0110	**0110**
	275	S=10	A=1010,	B=1111	out=1011	**1011**
	280	S=10	A=1010,	B=0000	out=1011	**1011**
	285	S=10	A=1010,	B=0101	out=1011	**1011**
	290	S=10	A=1010,	B=1010	out=1011	**1011**
	295	S=10	A=1010,	B=1111	out=1011	**1011**
	300	S=10	A=1111,	B=1111	out=0000	**0000**
	305	S=10	A=1111,	B=0000	out=0000	**0000**
	310	S=10	A=1111,	B=0101	out=0000	**0000**
	315	S=10	A=1111,	B=1010	out=0000	**0000**
	320	S=10	A=1111,	B=1111	out=0000	**0000**
A-1	325	S=11	A=1111,	B=1111	out=1110	**1110**
	330	S=11	A=0000,	B=1111	out=1111	**1111**
	335	S=11	A=0000,	B=0000	out=1111	**1111**
	340	S=11	A=0000,	B=0101	out=1111	**1111**
	345	S=11	A=0000,	B=1010	out=1111	**1111**
	350	S=11	A=0000,	B=1111	out=1111	**1111**
	355	S=11	A=0101,	B=1111	out=0100	**0100**
	360	S=11	A=0101,	B=0000	out=0100	**0100**
	365	S=11	A=0101,	B=0101	out=0100	**0100**
	370	S=11	A=0101,	B=1010	out=0100	**0100**
	375	S=11	A=0101,	B=1111	out=0100	**0100**
	380	S=11	A=1010,	B=1111	out=1001	**1001**
	385	S=11	A=1010,	B=0000	out=1001	**1001**
	390	S=11	A=1010,	B=0101	out=1001	**1001**
	395	S=11	A=1010,	B=1010	out=1001	**1001**
	400	S=11	A=1010,	B=1111	out=1001	**1001**
	405	S=11	A=1111,	B=1111	out=1110	**1110**
	410	S=11	A=1111,	B=0000	out=1110	**1110**
	415	S=11	A=1111,	B=0101	out=1110	**1110**
	420	S=11	A=1111,	B=1010	out=1110	**1110**
	425	S=11	A=1111,	B=1111	out=1110	**1110**

Problem 2:

You have been given a finite state machine that is to accomplish the functionality in the following figure. The compiled module that implements this machine is available online. Download and uncompress the compressed folder problem2.rar. You should set your working design directory within NC-SIM to the problem2 folder extracted from the archive. (Note: you will not be able to ready the actual module code. However, the image of the FSM will appear in your worklib.) The interface to the machine is **Problem2FSM(clk, Reset, Pulse, Done, Go, Ready)**.

Write a test bench for this machine and determine whether or not it is working to specifications. Be sure to annotate your work.



Problem2FSM.v:

```
module Problem2FSM( clk, Reset, Pulse, Done, Go, Ready );

input clk, Reset, Pulse, Done;
output Go, Ready;

reg [1:0] State, NextState;
reg Go, Ready;

always @ ( posedge clk or posedge Reset )
   if( Reset )
      State <= 0;
   else
      State <= NextState;

always @ ( State or Pulse or Done )
   case( State )
      2'b00 : NextState <= Pulse ? 2'b10 : 2'b00;
      2'b01 : NextState <= 2'b00;
      2'b10 : NextState <= Done ? 2'b01 : 2'b10;
      default : NextState <= 2'b00;
   endcase

always @ ( State or Pulse or Done )
   case( State )
      2'b00 : {Go, Ready} <= Pulse ? 2'b10 : 2'b01;
      2'b01 : {Go, Ready} <= 2'b00;
      2'b10 : {Go, Ready} <= 2'b00;
      default : {Go, Ready} <= 2'b00;
   endcase

endmodule
```

Problem2FSM_TB.v:

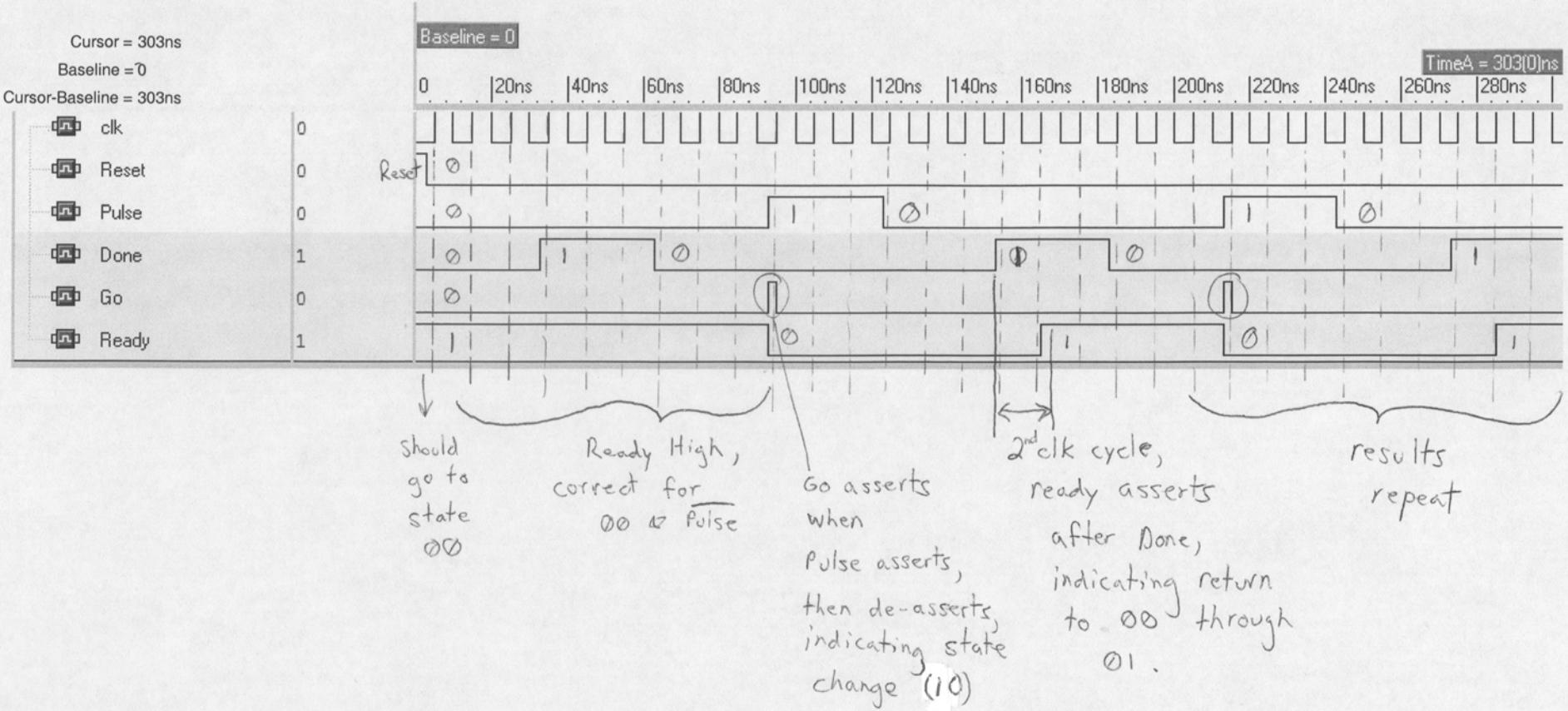
```
module Problem2FSM_TB;

// declarations
reg clk, Reset, Pulse, Done;
wire Go, Ready;

// instantiate block
Problem2FSM testMachine( clk, Reset, Pulse, Done, Go, Ready );

// define clock
always #5 clk <= ~clk;

// test ...
initial begin

   // waveform probe
   $shm_open( "Problem2FSM.waves" );
   $shm_probe( "AC" );

   // initial conditions
   clk = 0;
   Reset = 1;
   Pulse = 0;
   Done = 0;

   // finish reseting
   #3 Reset = 0;        // wait 3 delay units so that inputs are off
clk

   // - initially, the state machine should be set in the 00 state
   // - as long as Pulse is 0, there should be no state change
   // - try flexing done and see what happens
   #30
   Done = 1;
   #30
   Done = 0;
   #30

   // set Pulse to one
   // the FSM should assert Go while traveling to state 10
   Pulse = 1;
   #30

   // while Done is 0, no change...
   // flex Pulse
   Pulse = 0;
   #30

   // set Done to 1, expect migration of state twice (01->00)
   // Ready should be asserted in second clock cycle
   Done = 1;
   #30

   // repeat test to check against additional functionality
   Done = 0;
```

```
    #30
    Pulse = 1;
    #30
    Pulse = 0;
    #30
    Done = 1;
    #30


    $finish;

end

endmodule
```

NOTE: Testbenches may differ

Problem #2, Homework #4, test

Laflen
RHIT



Problem 3

Write a verilog description of the FSM in Problem #2. Run your test from Problem #2 on it to verify that it works.

See above code. It is correct.

Problem 4

Write a verilog description of the FSM in Problem #2, but implement it as a Moore Machine. Run your test from Problem #2 on it to verify that it works.

The state machine needs to be changed slightly to facilitate this operation. **Go** and **Ready** can only be functions of the state. **Go** should be output by state **10** and **Ready** should be output by state **00**.

Problem4FSM.v:

```
module Problem2FSM( clk, Reset, Pulse, Done, Go, Ready );

input clk, Reset, Pulse, Done;
output Go, Ready;

reg [1:0] State, NextState;
reg Go, Ready;

always @ ( posedge clk or posedge Reset )
   if( Reset )
      State <= 0;
   else
      State <= NextState;

always @ ( State or Pulse or Done )
   case( State )
      2'b00 : NextState <= Pulse ? 2'b10 : 2'b00;
      2'b01 : NextState <= 2'b00;
      2'b10 : NextState <= Done ? 2'b01 : 2'b10;
      default : NextState <= 2'b00;
   endcase

always @ ( State or Pulse or Done )
   case( State )
      2'b00 : {Go, Ready} <= 2'b01;
      2'b01 : {Go, Ready} <= 2'b10;
      2'b10 : {Go, Ready} <= 2'b00;
      default : {Go, Ready} <= 2'b00;
   endcase

endmodule
```