Name: **Jonathan Lemarroy**

ECE 3544: Digital Design 1
Homework Assignment 7 (80 points)

When writing a Verilog module or naming a file, replace YOURPID with your Virginia Tech PID. Name the file according to the module name that you use.

For each design that you represent in a Verilog module, copy your source code into the document. For each design that you simulate in ModelSim, include waveforms displaying the correct operation of each module

In addition, submit the .v file containing each design and each test bench that you write. Your files should contain header information and be neatly formatted and commented. Submit your files separately on Canvas. Do not put them into an archive. *Make sure that you upload all of your files.*

Helpful Hints for Modeling Discrete Time Delays

When a model consists of a number of individual logic paths, modeling discrete delays is likely to be simpler than modeling `specify` blocks. To correctly model discrete delays in Verilog simulation, you must use *continuous assignment* and *dataflow operators*. The delay must appear on the left-hand side of the assignment.

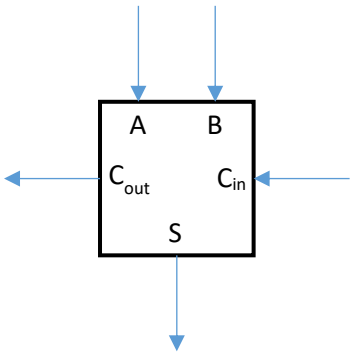
```
assign #10 out = in;
```

It is bad coding style to place delays into procedural models that represent hardware for the purpose of modeling behavior. Of course, placing delays into testbench modules to model the flow of time is fine.

Problem 1 (20 points)
To compare the timing behavior of the ripple-carry and carry-lookahead adders, implement an 8-bit adder of each variety in Verilog.

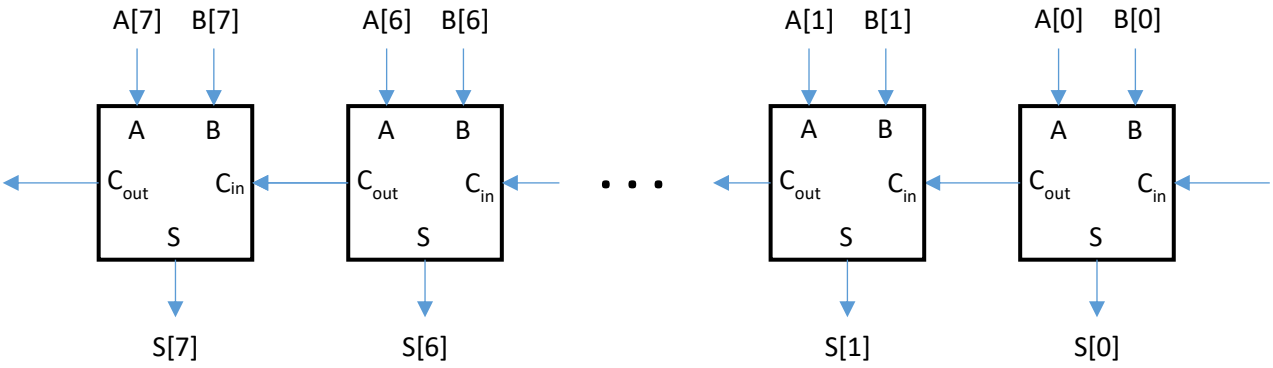
To implement the ripple-carry adder:

- Start by implementing a full adder. Use the general equations for sum and carry-out. Give each equation a delay of 20 nanoseconds. *Remember to include a timescale directive in your model*.




```
module full_adder_YOURPID(a, b, ci, s, co);
   input  a, b, ci;
   output s, co;
```

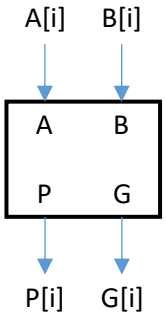
- Create the ripple-carry adder by **instantiating and linking** eight full adders. Since you introduced the delay into each full adder, you need not implement any delay in this module.




```
module p1_rca_YOURPID(A, B, S, Cout);

// Remember: C0 = 0.
// You can make this connection at this level of the model.

   input  [7:0] A, B;
   output [7:0] S;
   output       Cout;    // Which of the carry-out bits is last?
```

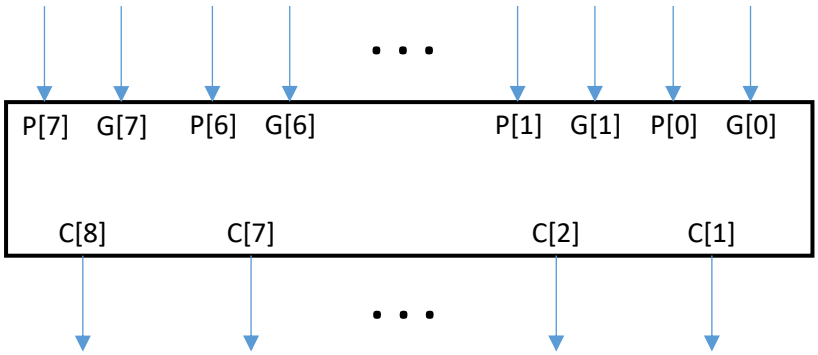
To implement the carry-lookahead adder:

- Start by implementing a propagate-generate block. Use the general equations for propagate and generate. Give each equation a delay of 20 nanoseconds.



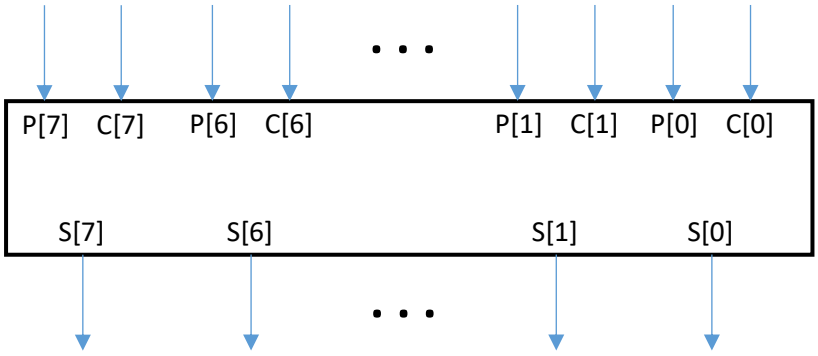

```
module prop_gen_YOURPID(a, b, p, g);
   input  a, b;
   output p, g;
```

- Next, implement the carry-lookahead block. It should accept propagate and generate signals as inputs and produce carry bits as outputs. Give each carry output equation a delay of 20 nanoseconds. *Make certain to represent the carry-out equations as parallel functions – they should be functions of P and G only*.




```
module cl_block_YOURPID(P, G, C);
   input  [7:0] P, G;
   output [8:1] C;       // C0 = 0. It doesn't need an equation.
```

- Next, implement the sum block. It should accept the propagate and carry terms as inputs and produce the sum as outputs. Give each equation a delay of 20 nanoseconds.




```
module sum_block_YOURPID(P, C, S);

// Remember, C0 is still 0.
// You can make this connection at this level of the model.

   input  [7:0] P;
   input  [7:1] C;
   output [7:0] S;
```

- Finally, create the carry-lookahead adder by **instantiating and linking** *eight* propagate-generate blocks, the carry-lookahead block, and the sum block. Since you introduced the delay into each block of the carry-lookahead adder, you need not implement any delay in this module.

```
module p1_cla_YOURPID(A, B, S, Cout);
   input  [7:0] A, B;
   output [7:0] S;
   output       Cout;    // Which of the carry-out bits is last?
```

As a simple test of one worst-case scenario, write a test bench (`tb_p1_YOURPID`) that simulates the addition of 8'b11111111 and 8'b00000001 using both adders. Show the waveform results in your solution. Also, write a comparison of the time behavior of both implementations, and comment on whether or not the behavior exhibited by both adders was expected or unexpected.

```
module full_adder_jdl25175(a, b, ci, s, co);
    input  a, b, ci;
    output s, co;

    assign #20 s = a ^ b ^ ci;
    assign #20 co = (a & b) | (b & ci) | (a & ci);

endmodule
```

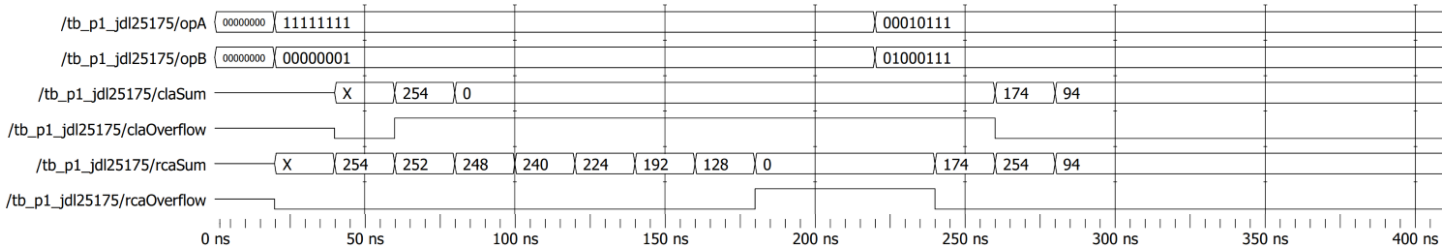
```
module p1_rca_jdl25175(A, B, S, Cout);
// Remember: C0 = 0.
// You can make this connection at this level of the model.
    input  [7:0] A, B;
    output [7:0] S;
    output       Cout;  // Which of the carry-out bits is last?
    wire   [6:0] c;
    full_adder_jdl25175 adder0(A[0], B[0], 1'b0, S[0], c[0]),
                        adder1(A[1], B[1], c[0], S[1], c[1]),
                        adder2(A[2], B[2], c[1], S[2], c[2]),
                        adder3(A[3], B[3], c[2], S[3], c[3]),
                        adder4(A[4], B[4], c[3], S[4], c[4]),
                        adder5(A[5], B[5], c[4], S[5], c[5]),
                        adder6(A[6], B[6], c[5], S[6], c[6]),
                        adder7(A[7], B[7], c[6], S[7], Cout);
endmodule
```

```
module prop_gen_jdl25175(a, b, p, g);
    input  a, b;
    output p, g;
    assign #20 g = a & b;
    assign #20 p = a ^ b;
endmodule
```

```
module cl_block_jdl25175(P, G, C);
    input  [7:0] P, G;
    output [8:1] C;     // C0 = 0. It doesn't need an equation.
    assign #20 C[1] = G[0];
    assign #20 C[2] = G[1] | (P[1] & G[0]);
    assign #20 C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]);
    assign #20 C[4] = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] & P[1] & G[0
]);
    assign #20 C[5] = G[4] | (P[4] & G[3]) | (P[4] & P[3] & G[2]) | (P[4] & P[3] & P[2] & G[1
]) | (P[4] & P[3] & P[2] & P[1] & G[0]);

    assign #20 C[6] = G[5] | (P[5] & G[4]) | (P[5] & P[4] & G[3]) | (P[5] & P[4] & P[3] & G[2
])
    | (P[5] & P[4] & P[3] & P[2] & G[1]) | (P[5] & P[4] & P[3] & P[2] & P[1] & G[0]);

    assign #20 C[7] = G[6] | (P[6] & G[5]) | (P[6] & P[5] & G[4]) | (P[6] & P[5] & P[4] & G[3
])
    | (P[6] & P[5] & P[4] & P[3] & G[2]) | (P[6] & P[5] & P[4] & P[3] & P[2] & G[1]) | (P[6]
& P[5] & P[4] & P[3] & P[2] & P[1] & G[0]);

    assign #20 C[8] = G[7] | (P[7] & G[6]) | (P[7] & P[6] & G[5]) | (P[7] & P[6] & P[5] & G[4
]) | (P[7] & P[6] & P[5] & P[4] & G[3])
    | (P[7] & P[6] & P[5] & P[4] & P[3] & G[2]) | (P[7] & P[6] & P[5] & P[4] & P[3] & P[2] &
G[1]) | (P[7] & P[6] & P[5] & P[4] & P[3] & P[2] & P[1] & G[0]);

endmodule
```

```
module sum_block_jdl25175(P, C, S);
    input  [7:0] P;
    input  [7:1] C;
    output [7:0] S;
    assign #20 S[0] = P[0];
    assign #20 S[1] = P[1] ^ C[1];
    assign #20 S[2] = P[2] ^ C[2];
    assign #20 S[3] = P[3] ^ C[3];
    assign #20 S[4] = P[4] ^ C[4];
    assign #20 S[5] = P[5] ^ C[5];
    assign #20 S[6] = P[6] ^ C[6];
    assign #20 S[7] = P[7] ^ C[7];
endmodule
```

```
module p1_cla_jdl25175(A, B, S, Cout);
    input  [7:0] A, B;
    output [7:0] S;
    output       Cout;  // Which of the carry-out bits is last?
    wire [7:0] P;
    wire [7:0] G;
    wire [8:1] C;
    prop_gen_jdl25175 propGen0(A[0], B[0], P[0], G[0]),
                      propGen1(A[1], B[1], P[1], G[1]),
                      propGen2(A[2], B[2], P[2], G[2]),
                      propGen3(A[3], B[3], P[3], G[3]),
                      propGen4(A[4], B[4], P[4], G[4]),
                      propGen5(A[5], B[5], P[5], G[5]),
                      propGen6(A[6], B[6], P[6], G[6]),
                      propGen7(A[7], B[7], P[7], G[7]);

    cl_block_jdl25175 clBlock(P, G, C);
    sum_block_jdl25175 sumBlock(P, C[7:1], S);
    assign Cout = C[8];
endmodule
```



The timing matches what would be expected as the rcaSum takes 160ns for the carry bit to propagate (8 times) down the line. Whereas the claSum only takes 60ns to propagate (3 times). In reality the cl_block wouldn't stay at a constant 20ns to compute each, but in worst case since each of the AND/OR gates can be calculated in parallel it would take a logarithmic time complexity relative to the number of operand bits.

Problem 2 (30 points)
Implement a "mini-ALU" having the following characteristics:

- First, implement a *combinational circuit* to act as the ALU core. This core performs operations on two 8-bit operands according to the following function table. You may use existing dataflow operators to represent the operations.

Opcode	Operation
00	`core_out = opA + opB`
01	`core_out = opA - opB`
10	`core_out = opA & opB`
11	`core_out = opA | opB`

The module declaration for the core is as follows:

```
module alu_core_YOURPID(opcode, opA, opB, core_out);
   input  [3:0] opcode;
   input  [7:0] opA, opB;
   output [7:0] core_out;
```

Even though the ALU only has four operations, the opcode is four bits wide. This is done on purpose to make the ALU extensible.

- Next, implement 8-bit registers for the ALU's use. An active-low asynchronous `reset` should clear the register output (`outs`) when it is asserted. An active-high `load` should cause `outs` to take on the value of `ins` following a clock pulse when `enable` = 1. The module declaration for the register is as follows:

```
module reg_8bit_YOURPID(clock, reset, load, ins, outs);
   input        clock;
   input        reset;
   input        load;
   input  [7:0] ins;
   output [7:0] outs;
```

Even though the opcode is only four bits wide, we're using an 8-bit register to hold its value. That way, we don't have to write a distinct 4-bit register.

- If the ALU had access to multiple inputs we wouldn't need registers. Instead, we could perform every ALU operation as a micro-operation. This ALU only has one 8-bit input to supply the opcode and operand values.

  In this ALU architecture, each operation takes place over four clock cycles:
  - After a reset, the input carries the value of the opcode, which must be registered on the next clock edge.
  - The input then carries the value of operand A, which must be registered on the next clock edge.
  - The input then carries the value of operand B, which must be registered on the next clock edge.
  - The value of the result must be registered on the next clock edge. The cycle then repeats; *the circuit does not have to be reset after each operation*.

The controller is a *state machine* having the following module declaration:

```
module alu_controller_YOURPID(clock, reset, loadOp, loadA, loadB,
                              loadOut);
   input  clock;
   input  reset;
   output loadOp, loadA, loadB, loadOut;
```

- The top-level module is the ALU. The top-level module is the only one that receives the 8-bit ALU input `alu_in`. Your top-level module should instantiate 8-bit registers to act as the opcode, operand, and result registers, an ALU core to perform the operations, and an ALU controller to govern when `alu_in` is visible to the components of the ALU as described above. The top-level module has the following module declaration:

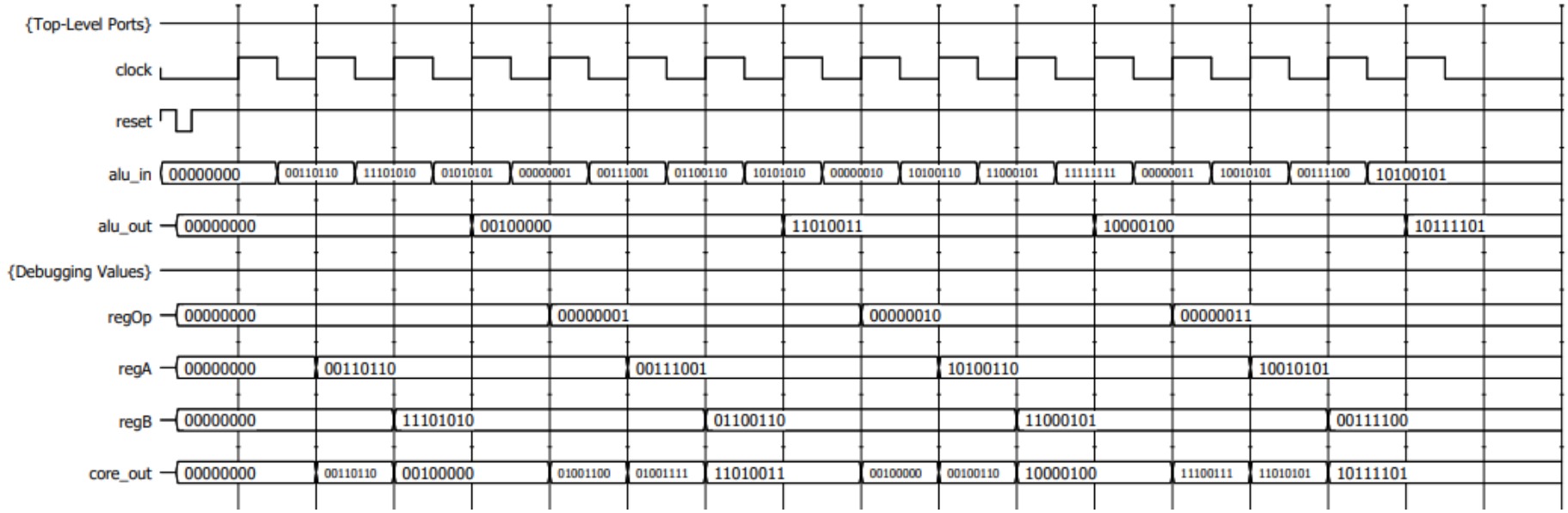
```
module p2_alu_YOURPID(clock, reset, alu_in, alu_out,
                      regOp, regA, regB, core_out);
   input        clock, reset;
   input  [7:0] alu_in;
   output [7:0] alu_out;
   output [7:0] regOp, regA, regB, core_out;
```

`alu_out` *is the primary output of the module. The other outputs are being made visible for simulation and debugging purposes.*

Write a test bench (`tb_p2_YOURPID`) that simulates the behavior of your ALU. Show the waveform results in your solution.

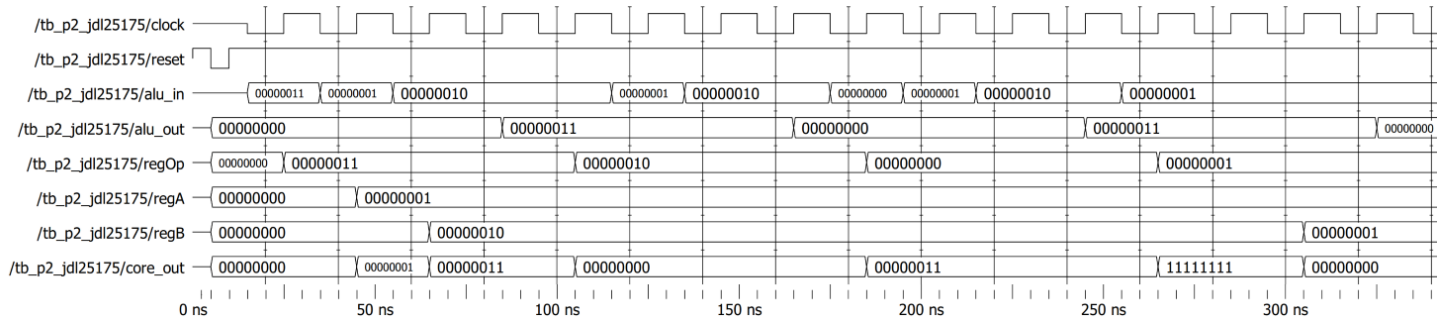
The sample timing waveform shown below demonstrates the behavior of the ALU through four 4-clock period cycles. Note the way – after a reset – `regOp` takes on the value of `alu_in` after the first `clock` trigger, `regA` takes on the value of `alu_in` after the second `clock` trigger, `regB` takes on the value of `alu_in` after the third clock cycle, and `alu_out` takes on the value of `core_out` after the fourth `clock` trigger. The second four-clock period cycle begins on the first `clock` trigger after the first cycle ends, and so forth.

Since `core_out` is **combinational**, it changes frequently during the operation of the ALU. But since it takes three clock cycles for the opcode and operands to become correct, `core_out` is not valid before the third clock cycle of each operation cycle. It is this value that is registered on the fourth clock cycle.



```
module alu_core_jdl25175(opcode, opA, opB, core_out);
   input  [3:0] opcode;
   input  [7:0] opA, opB;
   output [7:0] core_out;

   assign core_out = (opcode == 4'd0) ? (opA + opB) :
                     (opcode == 4'd1) ? (opA - opB) :
                     (opcode == 4'd2) ? (opA & opB) :
                     (opcode == 4'd3) ? (opA | opB) :
                     4'd0;

endmodule

module reg_8bit_jdl25175(clock, reset, load, ins, outs);
   input        clock;
   input        reset;
   input        load;
   input  [7:0] ins;
   output [7:0] outs;
   reg    [7:0] outs;

    always @(posedge clock or negedge reset) begin
        if(reset == 1'b0)
            outs <= 8'b0;
        else if(load == 1'b1)
            outs <= ins;
    end

endmodule


module alu_controller_jdl25175(clock, reset, loadOp, loadA, loadB, loadOut);
    input  clock;
    input  reset;
    output loadOp, loadA, loadB, loadOut;
    reg    loadOp, loadA, loadB, loadOut;

    reg [1:0] cycles;

    wire out;

    always @(posedge clock or negedge reset) begin
        if(reset == 1'b0) begin
            cycles <= 2'b00;
        end
        else begin
            cycles <= cycles + 2'b1;
        end
    end
```

```
    always @(cycles) begin
        if(cycles == 2'b00) begin
            loadOp = 1'b1;
            loadA = 1'b0;
            loadB = 1'b0;
            loadOut = 1'b0;
        end
        else if(cycles == 2'b01) begin
            loadOp = 1'b0;
            loadA = 1'b1;
            loadB = 1'b0;
            loadOut = 1'b0;
        end
        else if(cycles == 2'b10) begin
            loadOp = 1'b0;
            loadA = 1'b0;
            loadB = 1'b1;
            loadOut = 1'b0;
        end
        else begin
            loadOp = 1'b0;
            loadA = 1'b0;
            loadB = 1'b0;
            loadOut = 1'b1;
        end
    end
endmodule

module p2_alu_jdl25175(clock, reset, alu_in, alu_out, regOp, regA, regB, core_out);
    input        clock, reset;
    input  [7:0] alu_in;
    output [7:0] alu_out;
    output [7:0] regOp, regA, regB, core_out;
    wire loadOp, loadA, loadB, loadOut;
    alu_controller_jdl25175 aluCtrl(clock, reset, loadOp, loadA, loadB, loadOut);
    reg_8bit_jdl25175 opReg(clock, reset, loadOp, alu_in, regOp),
                      aReg(clock, reset, loadA, alu_in, regA),
                      bReg(clock, reset, loadB, alu_in, regB),
                      outReg(clock, reset, loadOut, core_out, alu_out);
    alu_core_jdl25175 aluCore(regOp[3:0], regA, regB, core_out);
endmodule
```



Problem 3 (30 points)
Follow the model presented on Slides 6 and 7 of Section 13.C (Multipliers) to implement an 8-bit multiplier.

- Your multiplier only has one 8-bit input (`bus_in`) which you must use to initialize the multiplicand (an 8-bit register) and the multiplier (the lowest 8 bits of a 17-bit register).
  - On the first clock cycle after reset, `bus_in` carries the value of the multiplicand.
  - On the second clock cycle after reset, `bus_in` carries the value of the multiplier.
  - Your control unit must enable the correct register at each clock edge.
- Multiplication is a multi-cycle operation, so your control unit must also be able to sequence and control the multiplication process.
  - On the clock cycle after the multiplier is loaded, your multiplier should begin the eight "instruction cycle" process of producing the product through a process of adds (selective-loads) and shifts.
  - The control unit should use the current LSB of the multiplier to tell the product register when to store the result of the adder, and when to shift.
- In this implementation, you must determine the appropriate number of clock cycles for the multiplier control unit to take when performing the serial multiplication. The registers and control unit must operate on the same clock. Use the sequence of multiplier operations described in the notes to determine how many clock cycles is appropriate.

All of the resets are asynchronous and active-low.

Use the following module declaration as a guide. You may add outputs to it as needed if you want to observe their behavior in specification; the sample simulation waveform may offer some hint as to internal values that are worth observing as outputs.

```
module p3_multiplier_YOURPID(clock, reset, bus_in, multiplicand, product);
   input         clock;
   input         reset;
   input   [7:0] bus_in;
   output  [7:0] multiplicand;
   output [15:0] product;
```

`product` *is the primary output of the module. The output* `multiplicand` *is being made visible for simulation and debugging purposes.*

Your multiplier should consist of the following subunits:

- An 8-bit parallel load register, to hold the multiplicand:

```
module reg_8bit_YOURPID(clock, reset, load, ins, outs);
```

- A 17-bit shift register with parallel load, to hold the multiplier (at the start) and the product (at the end):

```
module shiftreg_17bit_YOURPID(clock, reset, load, shift_right,
                              ins, outs);
```

- An 8-bit adder. You don't have to implement an adder using the techniques of problem 1. You may use the addition dataflow operator.

  ```
  module adder_8bit_YOURPID(a, b, cout, s);
  ```

- A 17-bit 2-to-1 multiplexer. During the "initialization" cycles, you have to load the 17-bit register with the value on `bus_in`. But during the selective-load cycles, you have to load the register using the adder outputs and the register's own values. The multiplexer allows you to select the right value at the right times.

  ```
  module mux2to1_17bit_YOURPID(select, in0, in1, out);
  ```

- A control circuit. In the appropriate order and at the appropriate times, you must control the multiplicand load, the product register load and shift, and the multiplexer select.

  ```
  module multiply_controller_YOURPID(clock, reset, lsb, loadA, loadB,
                                     select, shift);
  ```

  Remember what the complete system cycle looks like:
  - On the first clock cycle, `bus_in` should be stored in the multiplicand register.
  - On the second clock cycle, `bus_in` should be stored in the multiplier register.
  - On the next sixteen clock cycles, the multiplier should alternate between selective-load and shift operations.

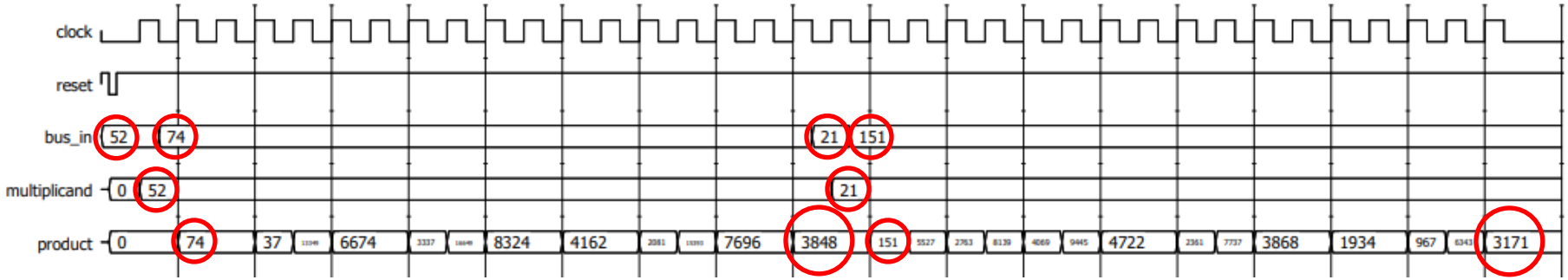
Write a test bench (`tb_p3_YOURPID`) that simulates your multiplier carrying out a sequence of operations. Show the waveform results in your solution.

Here is a waveform for a multiplier that meets the requirements specified above.

Part of the “trick” is knowing when the right time to read the circuit’s output is. Since the operation is serial, the product is not always valid. The “right” number of cycles must pass before the user tries to read the correct value from product.

In this case, on the first two clock cycles, the multiplier registers `bus_in` as the multiplicand and the multiplier. This simulation doesn’t show the value of the multiplicand register, but note that on the second clock edge, the value of the product register equals the value on `bus_in` prior to that clock edge. After 16 more clock cycles, the product is correct.

On the next clock edge, the multiplier is ready to perform again. After a total of 18 cycles, the product is again correct.



To change the radix for a field so that it appears as an unsigned value, highlight the field name, then right-click and choose Radix > Unsigned.

READ THIS: Unfortunately, I could not get this to function properly in time for this assignment. All the modules <u>besides</u> the control circuit and the top-level multiplier do work correctly tho.

```
module reg_8bit_jdl25175(clock, reset, load, ins, outs);
    input        clock;
    input        reset;
    input        load;
    input  [7:0] ins;
    output [7:0] outs;
    reg    [7:0] outs;
```

```
    always @(posedge clock or negedge reset) begin
        if(reset == 1'b0)
            outs <= 8'b0;
        else if(load == 1'b1)
            outs <= ins;
    end
endmodule

module adder_8bit_jdl25175(a, b, cout, s);
    input  [7:0] a, b;
    output [7:0] s;
    output cout;
    assign {cout, s} = {1'b0, a} + {1'b0, b};
endmodule

module shiftreg_17bit_jdl25175(clock, reset, load, shift_right, ins, outs);
    input clock, reset, load, shift_right;
    input [16:0] ins;
    output reg [16:0] outs;
    always @(posedge clock, negedge reset) begin
        if(reset == 1'b0)
            outs <= 17'b0;
        else if(load & shift_right)
            outs <= ins >> 1;
        else if(load)
            outs <= ins;
        else if(shift_right)
            outs <= outs >> 1'b1;
    end

endmodule
module mux2to1_17bit_jdl25175(select, in0, in1, out);
    input select;
    input [16:0] in0, in1;
    output [16:0] out;
    assign out = (select == 1'b0) ? in0 : in1;
endmodule
```

```
module multiply_controller_jdl25175(clock, reset, lsb, loadA, loadB, select, shift);
    input clock, reset, lsb;
    output reg loadA, loadB, select, shift;
    reg [4:0] cycles;
    always @(posedge clock, negedge reset) begin
        if(reset == 1'b0)
            cycles <= 5'd0;
        else if(cycles >= 5'd8)
            cycles <= 5'd0;
        else
            cycles <= cycles + 1;
    end
    always @(cycles) begin
        if(cycles == 5'd0) begin
            loadA = 1'b1;
            loadB = 1'b0;
            select = 1'b0;
            shift = 1'b0;
        end
        else if(cycles == 5'd1) begin
            loadA = 1'b0;
            loadB = 1'b1;
            select = 1'b0;
            shift = 1'b0;
        end
        else begin
            loadA = 1'b0;
            loadB = 1'b0;
            if(lsb == 1'b0) begin
                select = 1'b0;
                shift = 1'b1;
            end
            else begin
                select = 1'b1;
                shift = 1'b1;
            end
        end
```

```
    end
endmodule

module p3_multiplier_jdl25175(clock, reset, bus_in, multiplicand, product);
    input         clock;
    input         reset;
    input   [7:0] bus_in;
    output  [7:0] multiplicand;
    output [15:0] product;

    wire loadA, loadB, select, shift, carry;
    wire [16:0] shiftRegOut;
    wire [7:0] sum;
    wire [16:0] mux1Out, mux2Out;

    adder_8bit_jdl25175 adder(multiplicand, shiftRegOut[15:8], carry, sum);
    reg_8bit_jdl25175 multpcnd(clock, reset, loadA, bus_in, multiplicand);

    mux2to1_17bit_jdl25175 mux1(select, {shiftRegOut}, {carry, sum, shiftRegOut[7:0]}, mux1Out),
                           mux2(loadB, mux1Out, {9'b0, bus_in}, mux2Out);

    shiftreg_17bit_jdl25175 shiftReg(clock, reset, 1'b1, shift, mux2Out, shiftRegOut);

    multiply_controller_jdl25175 cntrl(clock, reset, mux2Out[0], loadA, loadB, select, shift);

    assign product = shiftRegOut[15:0];

endmodule
```