EE417 Advanced Digital Logic with Verilog HDL Name: Ron Kalin / Lamin Jammeh Date: 06/26/24

Lesson 7 Data Path - Controller Design Structure

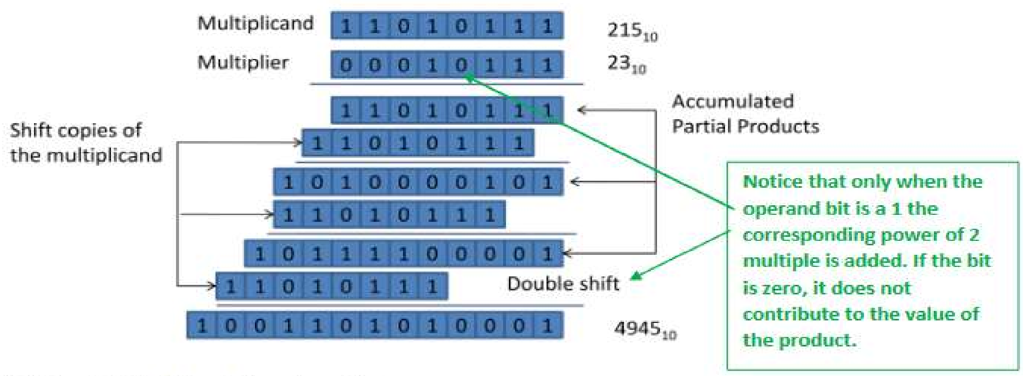
Multiplier Assignment EE 417.pdf

Design an optimized DataPath controller. The DataPath module and the top module are given in the attached file:

Design the controller that will complete the full design. Create a testbench and test the multiplier. Combine the code, StateGraph of the controller and the simulation results in one pdf file.

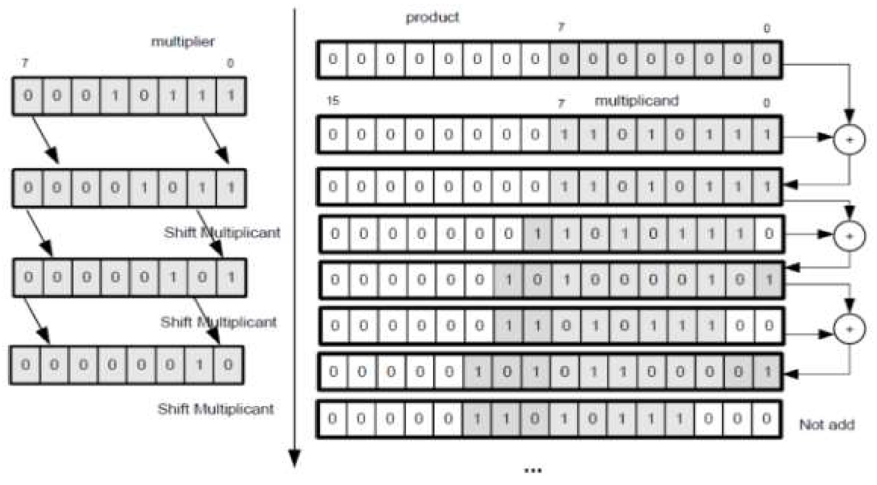
Assignment: DataPath Controller:

Consider the following technique used for combinational binary multiplying. If you want to multiply 215 by 23, you can add ($215*2^0 + 215*2^1 + 215*2^2 + 215*2^4$ = 215*1 + 215 *2 + 215*4 + 215*16 = 215(1+2+4+16) = 215(23))
To get the power 2 multiples of the multiplicand, all what we need to do is to shift it to the right as many positions as the exponent or power value. This is an optimized method compared to adding the multiplicand 23 times!
Design the Multiplier operation using a DataPath design layout. Test your design and verify the functionality.



Notice that only when the operand bit is a 1 the corresponding power of 2 multiple is added. If the bit is zero, it does not contribute to the value of the product.

The following is the datapath idea of operation:



Experimentation

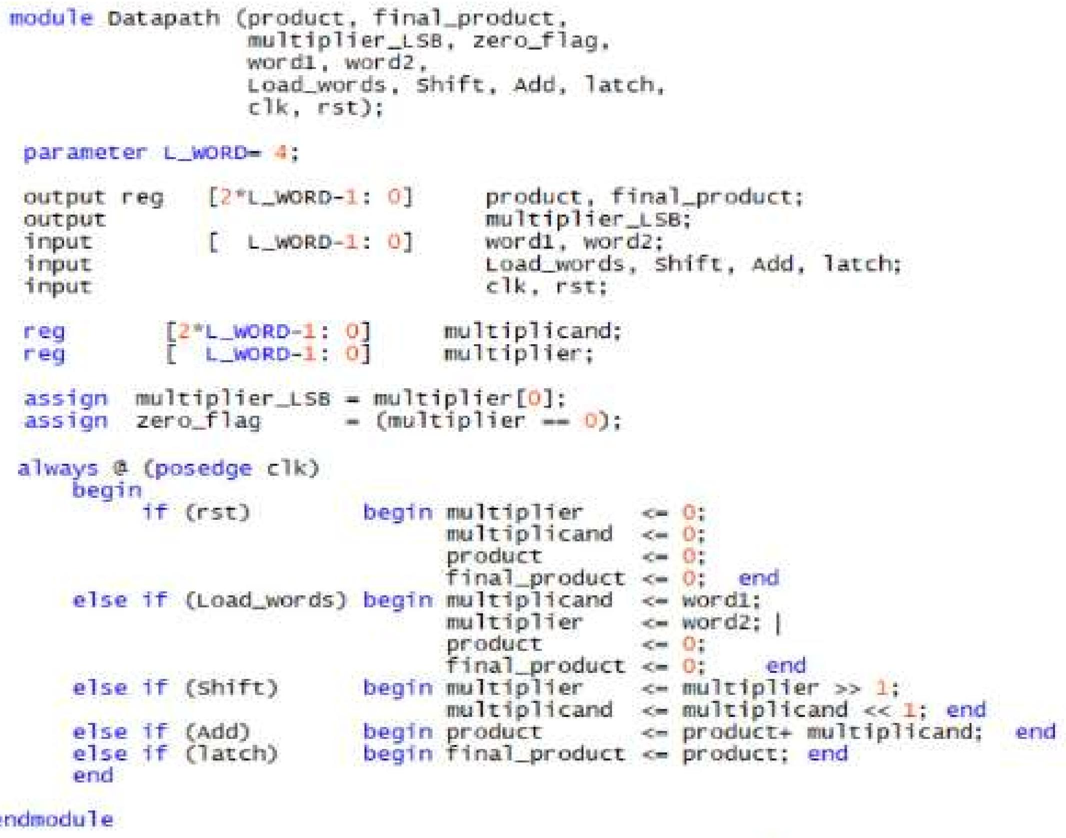
Built a quick calculator which demostrates better than example how this system works.

	Result	195					
		Value Dec	Value Bin	LSB Mult	State	Dec Prod	Bin Prod
Multiplicand	Word 1	15	1111				
Multiplier	Word 2	13	1101	1	Add	15	1111
					Shift	Shift must follow add	
Multiplicand	Word 1	30	11110				
Multiplier	Word 2	6	110	0	Shift	15	1111
Multiplicand	Word 1	60	111100				
Multiplier	Word 2	3	11	1	Add	75	1001011
					Shift	Shift must follow add	
Multiplicand	Word 1	120	1111000				
Multiplier	Word 2	1	1	1	Add	195	11000011
					Shift	Shift must follow add	
Multiplicand	Word 1	240	11110000				
Multiplier	Word 2	0			Latch	195	11000011

Product= Product+Multiplicand

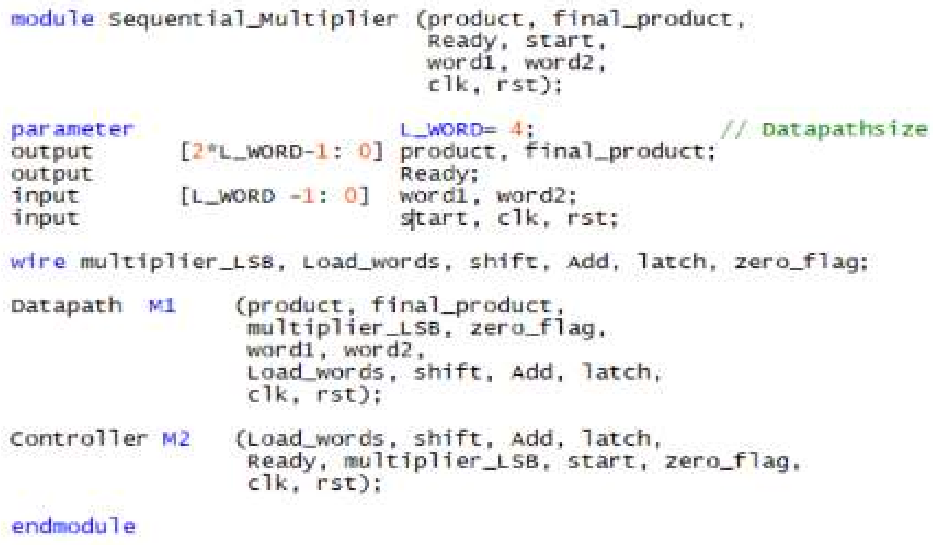
Design Methodology

The top level module and Datapath were given below. A couple additions to code given were made in order for it to be synthesized.



```
module Datapath (product, final_product,
                 multiplier_LSB, zero_flag,
                 word1, word2,
                 Load_words, Shift, Add, latch,
                 clk, rst);

 parameter L_WORD= 4;

 output reg   [2*L_WORD-1: 0]     product, final_product;
 output                           multiplier_LSB;
 input        [  L_WORD-1: 0]     word1, word2;
 input                            Load_words, Shift, Add, latch;
 input                            clk, rst;

 reg       [2*L_WORD-1: 0]     multiplicand;
 reg       [  L_WORD-1: 0]     multiplier;

 assign  multiplier_LSB = multiplier[0];
 assign  zero_flag      = (multiplier == 0);

 always @ (posedge clk)
     begin
         if (rst)        begin multiplier    <= 0;
                               multiplicand  <= 0;
                               product       <= 0;
                               final_product <= 0;  end
     else if (Load_words) begin multiplicand  <= word1;
                               multiplier    <= word2;
                               product       <= 0;
                               final_product <= 0;    end
     else if (Shift)      begin multiplier    <= multiplier >> 1;
                               multiplicand  <= multiplicand << 1; end
     else if (Add)        begin product       <= product+ multiplicand;   end
     else if (latch)      begin final_product <= product; end
     end
endmodule
```

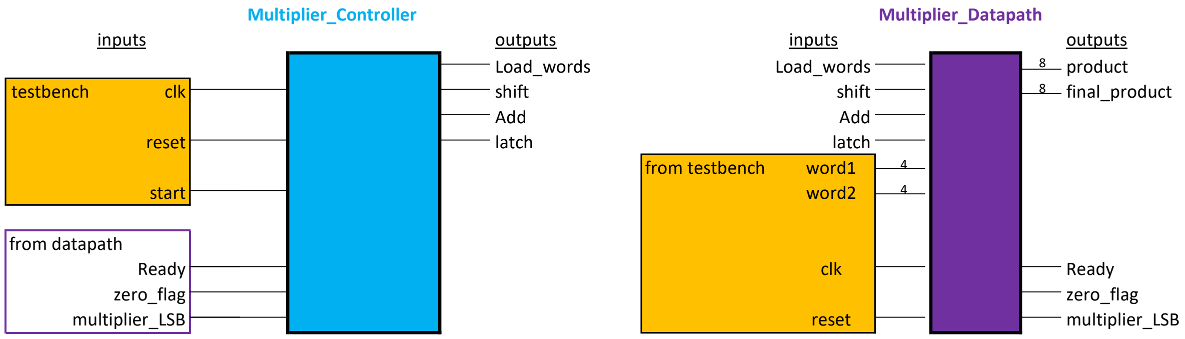
The following is the suggested top module. The module should raise a ready flag when it is ready to load new input words. The user should activate a start input to indicate that a new multiplication operation needs to be started.



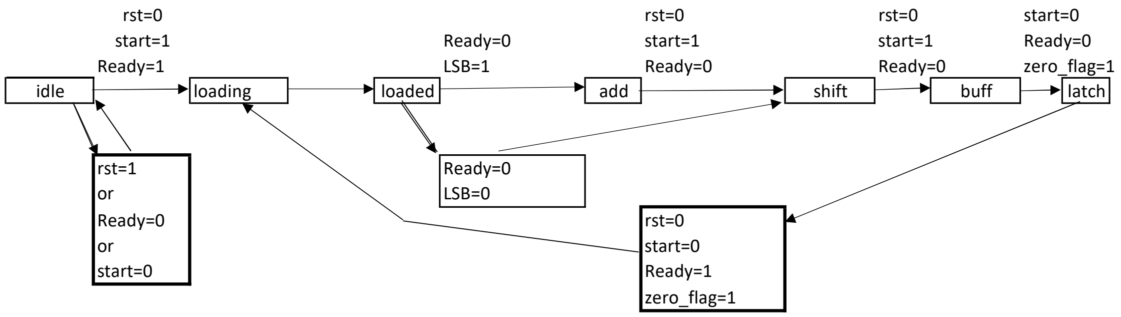
```
module Sequential_Multiplier (product, final_product,
                              Ready, start,
                              word1, word2,
                              clk, rst);

parameter                         L_WORD= 4;              // Datapathsize
output      [2*L_WORD-1: 0] product, final_product;
output                      Ready;
input       [L_WORD -1: 0]  word1, word2;
input                       start, clk, rst;

wire multiplier_LSB, Load_words, Shift, Add, latch, zero_flag;

Datapath  M1     (product, final_product,
                  multiplier_LSB, zero_flag,
                  word1, word2,
                  Load_words, Shift, Add, latch,
                  clk, rst);

Controller M2    (Load_words, Shift, Add, latch,
                  Ready, multiplier_LSB, start, zero_flag,
                  clk, rst);

endmodule
```

A block diagram and Finite State Machine (FSM) were developed from the Datapath in order to design the Controller code.
Ready was added as an output to the Datapath and as an input to the Controller.

Block Diagram



FSM for Controller



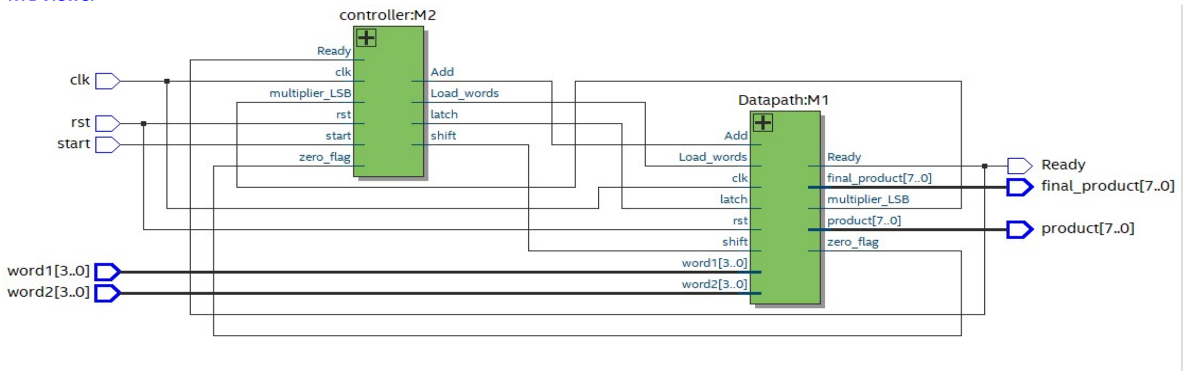
Design Verilog HDL Code -

```
// ee417 lesson 7 Assignment 1 L7A1
// Name: Ron Kalin, Date: 06-25-24  Group: Kalin/Jammeh
// Design: Sequential_multiplier
// top level module raises a ready flag when ready to load new input words
// user should activate a start input to indicate a new multiplication operation starts
module Sequential_multiplier  (product, final_product,
                               Ready, start,
                               word1, word2,
                               clk, rst);
parameter                     L_WORD =4;  //Datapathsize
output    [2*L_WORD-1: 0] product, final_product;
output                    Ready;
input     [L_WORD -1: 0]  word1, word2;
input                     start, clk, rst;

reg [L_WORD-1:0] mcand, mult;

wire multiplier_LSB , Load_words, shift, Add, latch, zero_flag;
wire [L_WORD-2:0] state;

Datapath M1   (product, final_product,
               Ready, multiplier_LSB , zero_flag,
               word1, word2,
               Load_words, shift, Add, latch,
               clk, rst);

controller M2 (Load_words, shift, Add, latch, state,
               Ready, multiplier_LSB , start, zero_flag,
               clk, rst);

endmodule

//Datapath
module Datapath (product, final_product,
                 Ready, multiplier_LSB , zero_flag,
                 word1, word2,
                 Load_words, shift, Add, latch,
                 clk, rst);

parameter L_WORD= 4;  //declare parameter values

//declare outputs and input
output reg [2*L_WORD-1:0] product, final_product;
output reg                Ready;
output                    multiplier_LSB , zero_flag;
input      [ L_WORD-1:0]  word1, word2;
input                     Load_words, shift, Add, latch;
input                     clk, rst;
//declare internal wires
reg    [2*L_WORD-1: 0]    multiplicand;
reg    [  L_WORD-1: 0]    multiplier;
//assign values
assign multiplier_LSB  = multiplier[0];    //least significant bit of multiplier
assign zero_flag       = (multiplier == 0);//if multiplier=all zeros, zero_flag=1

//create always block
always @ (posedge clk)
 begin
       if (rst)        begin multiplier    <= 0;//if reset =1 then zero
                             multiplicand  <= 0;
                             product       <= 0;
                             final_product <= 0;
                             Ready         <= 1; end//ready high= accept input words
  else if (Load_words) begin multiplicand  <= word1;//Load_words=1
                             multiplier    <= word2;//then m..cand gets value of word1
                             product       <= 0;    //mult gets value word2
                             final_product <= 0;    //prod and final_prod=0
                             Ready         <= 0; end//ready low means calculate
  else if (shift)      begin multiplier    <= multiplier >> 1;   //shift right 1
                             multiplicand  <= multiplicand << 1; //shift left 1
```

```
                             Ready         <= 0; end
  else if (Add)        begin product       <= product + multiplicand;
                             Ready         <= 0; end
  else if (latch)      begin final_product <= product; //adding done
                             Ready         <= 1; end   //ready for new input
  else                 begin end
 end

endmodule


//controller
module controller (Load_words, shift, Add, latch, state,
                 Ready, multiplier_LSB , start, zero_flag,
                 clk, rst);
//parameter L_WORD= 4;  //declare parameter values

//declare outputs/inputs, control unit only handles single bit inputs and outputs
output reg Load_words, shift, Add, latch;
output reg [2:0] state;
input       Ready, multiplier_LSB , start, zero_flag;
input       clk, rst;

reg [2:0]  next_state; //3 bits for up to 8 states
//build code from FSM diagram

//declare states from FSM
parameter idle    = 3'b000;
parameter loading = 3'b001;
parameter loaded  = 3'b010;
parameter add     = 3'b011;
parameter shft    = 3'b100;
parameter buff    = 3'b101;
parameter ltch    = 3'b110;

always @ (posedge clk)
if (rst) state <= idle;
   else  state <= next_state;

always @ *
  //assign probe <= state;
  case (state)
    idle: begin
            Load_words  = 1'b0;
            latch       = 1'b0;
            Add         = 1'b0;
            shift       = 1'b0;
            if (Ready && start) next_state = loading;
                         else   next_state = idle;    end
    loading: begin
            Load_words  = 1'b1;
            latch       = 1'b0;
            Add         = 1'b0;
            shift       = 1'b0;
            next_state  = loaded;   end
    loaded: begin  //2nd load stage is needed because input data changes
            Load_words  = 1'b0;  //1 cycle is needed to look at the change
            latch       = 1'b0;
            Add         = 1'b0;
            shift       = 1'b0;
            if (multiplier_LSB ) next_state = add;
                         else   next_state = shft;    end
    add: begin
            Load_words  = 1'b0;
            latch       = 1'b0;
            Add         = 1'b1;    //output changes
            shift       = 1'b0;
            next_state  = shft;     end  //shift always follows after an add
    shft: begin
            Load_words  = 1'b0;
            latch       = 1'b0;
```

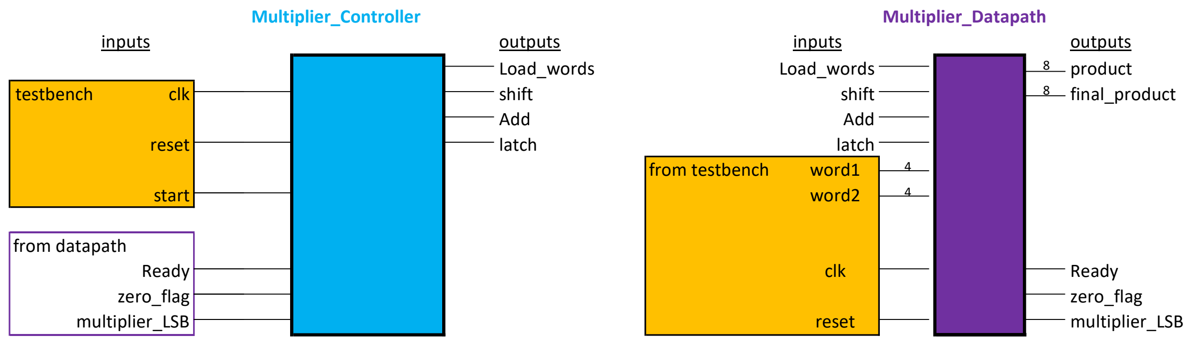
```
            Add         = 1'b0;
            shift       = 1'b1;
            next_state  = buff;     end
    buff: begin //buffer state needed after shift because input data changes
            Load_words  = 1'b0; //1 cycle is needed to take a look at the change
            latch       = 1'b0;
            Add         = 1'b0;
            shift       = 1'b0;
            if (multiplier_LSB ) next_state  = add;
            else if (zero_flag) next_state  = ltch;
                         else   next_state  = shft;
end
    ltch: begin
            Load_words  = 1'b0;
            Add         = 1'b0;
            shift       = 1'b0;
            latch       = 1'b1;
            next_state  = loading;    end  //cycle back to loading stage

//below is initial pseudocode
//if rst=1 then load_words=0, shift=add=latch=0
//if rst=0 and start=1 and ready=1 then load_words=1, shift=0, add=0, latch=0
//if rst=0 and start=1 and ready=0 then
  //if multiplier_LSB=0 then load_words=0, shift=1, add=0, latch=0
  //
  //else if multiplier_LSB=1 then load_words=0, shift=0, add=1, latch=0
  //
  //else if zero_flag=1      then load_words=0, shift=0, Add=0, latch=1, ready=1 ...done
  endcase
endmodule
```

RTL Viewer



As you can see the actual block diagram looks remarkably similar to the one that was conceptualized at the beginning of the project copied below.

Block Diagram

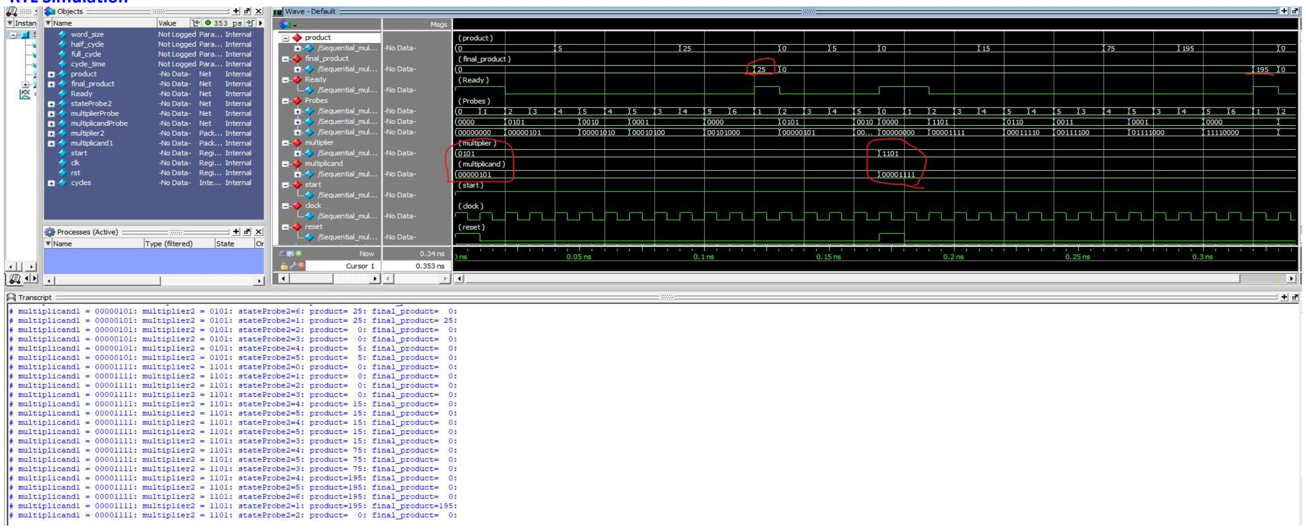


Testbench Verilog HDL Code

```
/*---------------------------------------------------------------------------------
Name Ron Kalin
CLass: EE417 Summer 2024
Lesson 07 HW Question 01
Group: Ron Kalin/ Lamin Jammeh
Project Description: test-bench for sequential multiplier
  ---------------------------------------------------------------------------------*/
module Sequential_multiplier_tb ();

//set the parameters wires and registers
parameter      word_size = 4;  //bit length of word inputs
parameter      half_cycle = 5; //half cycle time of clock
parameter      full_cycle = 10;//full cycle time of clock
parameter      cycle_time = 160;//number of cycles before next cycle
/*top level module under test original declaration
module Sequential_multiplier (product, final_product,
                             Ready, start,
                              word1, word2,
                              clk, rst);*/
//define outputs as wires, inputs as registers
wire [2*word_size-1: 0] product, final_product;
wire                    Ready;
wire [word_size-2:0]    stateProbe2; //internal probe wire for troubleshooting
wire [word_size-1:0]    multiplierProbe;
wire [2*word_size-1: 0] multiplicandProbe;
reg  [word_size -1: 0]  multiplier2;
reg  [2*word_size-1: 0] multiplicand1;
reg                     start, clk, rst;
integer cycles;
//define the unit under test UUT
Sequential_multiplier UUT (product, final_product,
                           Ready, start,
                           multiplicand1, multiplier2,
                           clk, rst);
//internal probes to track logic and troubleshoot
//assign stateProbe = UUT.state;
assign stateProbe2 = UUT.M2.state; //UUT=top module, M2 =submodule instance name, state is
register
assign multiplierProbe    = UUT.M1.multiplier;
assign multiplicandProbe  = UUT.M1.multiplicand;

//instantiate clock
initial
    begin: clock_loop
      clk = 1'b1;
      cycles = 0;
      forever begin
         #half_cycle     clk = ~clk;
         cycles = cycles + 1;
      end
      if (cycles == cycle_time) disable clock_loop;
    end

//define input words and observe the outputs
initial begin
  start = 1;  //start to high means start process
  rst = 1;    //reset high will set everything to zero and ready to high
  multiplicand1  = 8'b0000_0101; //initialize both words random test values
  multiplier2    = 4'b0101;
  #10 rst = 0;//reset low
  #cycle_time
  rst =1;
  multiplicand1  = 8'b0000_1111; //initialize both words random test values
  multiplier2    = 4'b1101;
  #10 rst = 0;
  #cycle_time //disable clock_loop;
//end
  multiplier2  = 4'b0000;
  multiplicand1  = 8'b0000_0000;
  forever begin: input_loop
```

```
   multiplier2 = multiplier2 + 1'b1;  //increment multiplier
   if (multiplier2 == 4'b1111) begin      //if multiplier reaches max value
       multiplier2 = 4'b0001;              //reset multiplier to one
       multiplicand1  = multiplicand1 + 1'b1;//incr multiplicand
   end
   if (multiplicand1  == 8'b1111_1111  && multiplier2  == 4'b1111) begin //both reach max value
      disable input_loop; disable clock_loop; end
   #cycle_time;
  end
end

////monitor and display the output
initial begin
$monitor("multiplicand1 = %b: multiplier2 = %b: stateProbe2=%d: product=%d:
final_product=%d:",multiplicand1, multiplier2,
   stateProbe2,product, final_product);
   end
endmodule
```

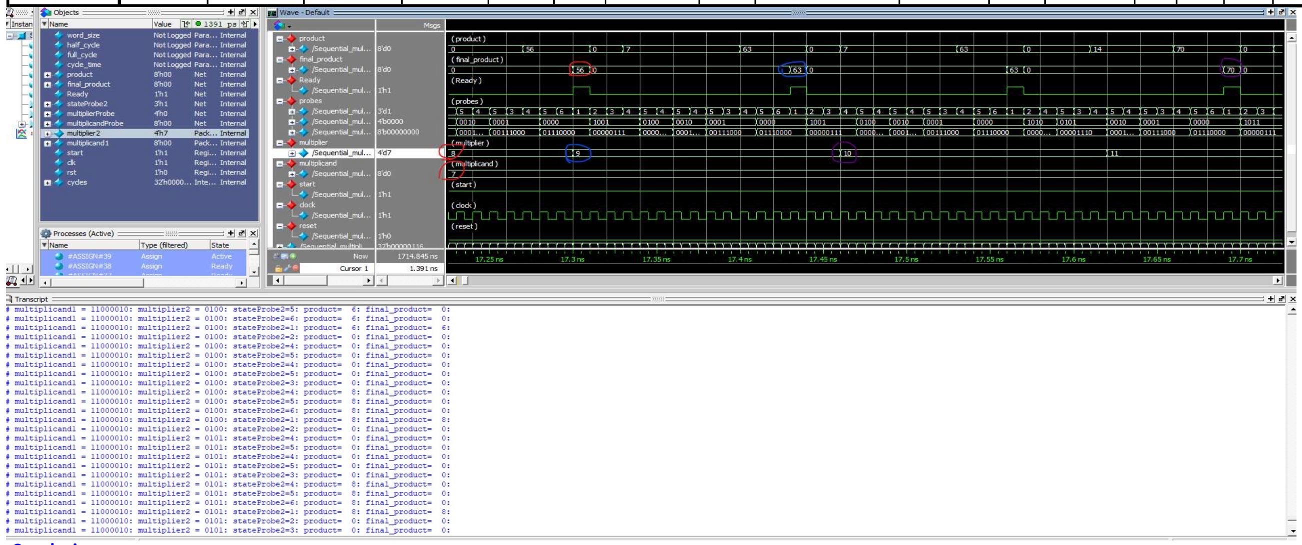
RTL Simulation



```
# multiplicand1 = 00000101: multiplier2 = 0101: stateProbe2=6: product= 25: final_product=  0:
# multiplicand1 = 00000101: multiplier2 = 0101: stateProbe2=1: product= 25: final_product= 25:
# multiplicand1 = 00000101: multiplier2 = 0101: stateProbe2=2: product=  0: final_product=  0:
# multiplicand1 = 00000101: multiplier2 = 0101: stateProbe2=3: product=  0: final_product=  0:
# multiplicand1 = 00000101: multiplier2 = 0101: stateProbe2=4: product=  5: final_product=  0:
# multiplicand1 = 00000101: multiplier2 = 0101: stateProbe2=5: product=  5: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=0: product=  0: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=1: product=  0: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=2: product=  0: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=3: product=  0: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=4: product= 15: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=5: product= 15: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=4: product= 15: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=5: product= 15: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=3: product= 15: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=4: product= 75: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=5: product= 75: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=3: product= 75: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=4: product=195: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=5: product=195: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=6: product=195: final_product=  0:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=1: product=195: final_product=195:
# multiplicand1 = 00001111: multiplier2 = 1101: stateProbe2=2: product=  0: final_product=  0:
```

Times table below for a 4-bit multiplicand and multiplier

multiplicand1	multiplier2															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
3	0	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45
4	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
5	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
6	0	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90
7	0	7	14	21	28	35	42	49	56	63	70	77	84	91	98	105
8	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120
9	0	9	18	27	36	45	54	63	72	81	90	99	108	117	126	135
10	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150
11	0	11	22	33	44	55	66	77	88	99	110	121	132	143	154	165
12	0	12	24	36	48	60	72	84	96	108	120	132	144	156	168	180
13	0	13	26	39	52	65	78	91	104	117	130	143	156	169	182	195
14	0	14	28	42	56	70	84	98	112	126	140	154	168	182	196	210
15	0	15	30	45	60	75	90	105	120	135	150	165	180	195	210	225



```
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=5: product=  6: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=6: product=  6: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=1: product=  6: final_product=  6:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=2: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=4: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=5: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=4: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=5: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=3: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=4: product=  8: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=5: product=  8: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=6: product=  8: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=1: product=  8: final_product=  8:
# multiplicand1 = 11000010: multiplier2 = 0100: stateProbe2=2: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=4: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=5: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=4: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=5: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=3: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=4: product=  8: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=5: product=  8: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=6: product=  8: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=1: product=  8: final_product=  8:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=2: product=  0: final_product=  0:
# multiplicand1 = 11000010: multiplier2 = 0101: stateProbe2=3: product=  0: final_product=  0:
```

Conclusion

When reset is low, this simulation shows that the two words, multiplicand and multiplier are multiplied together to give the result final_product.
When the reset goes high the process resets.
Random samples of the final_product can be correlated to the times table given above.