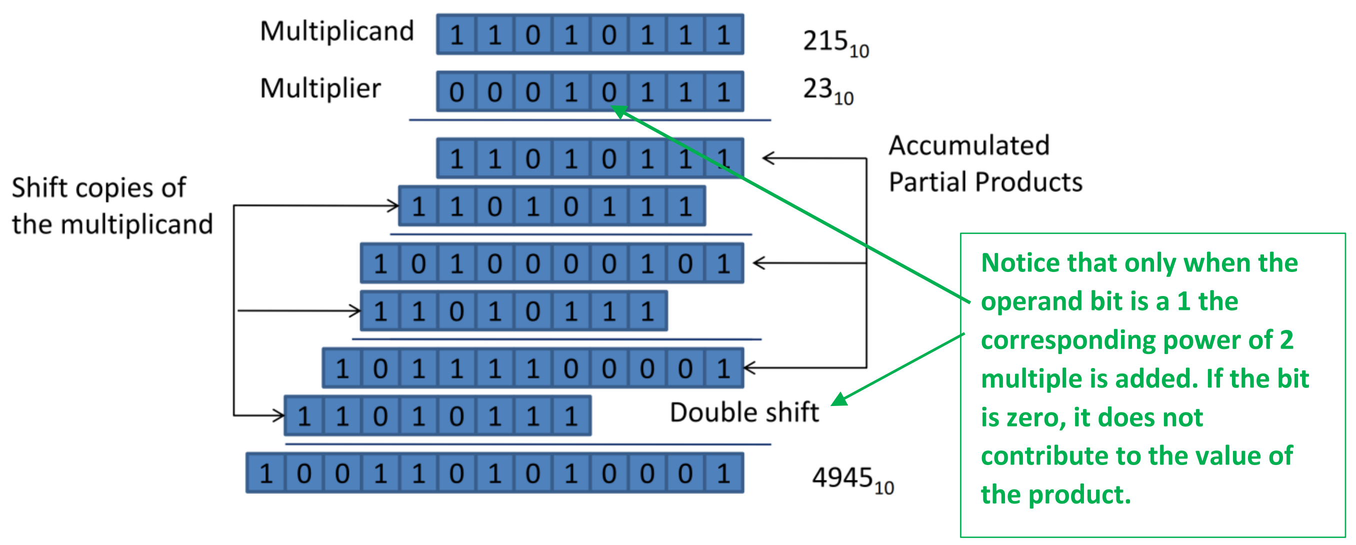
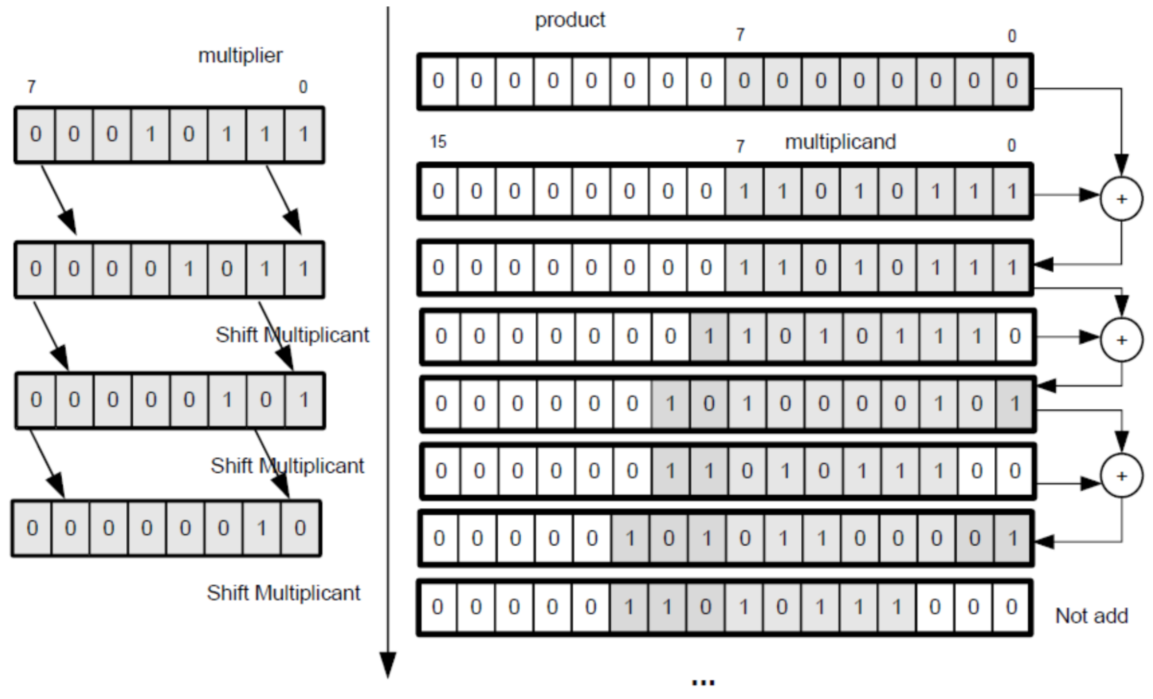
Problem 3: (25 points)

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 left as many positions as the exponent or power value. This is an optimized method compared to adding the multiplicand 23 times!



The following is the datapath idea of operation:

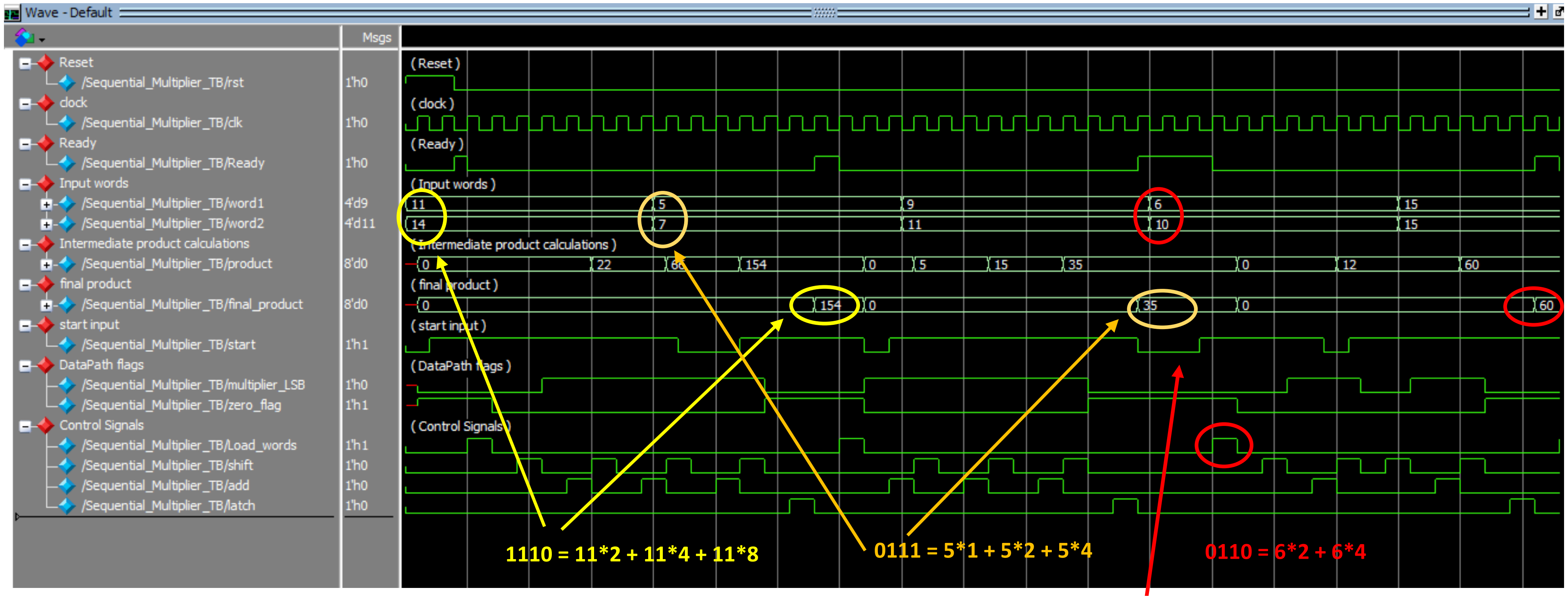


```
module Datapath (product, final_product,
                 multiplier_LSB, zero_flag,
                 word1, word2,
                 Load_words, Shift, Add, latch,
                 clk, rst);

 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



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
```

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.
These are the initial statements for the controller code. Complete the code and include the FSM state graph and include a testbench to test the top module.

```
module Controller (Load_words, shift, Add, latch,
                   Ready, multiplier_LSB, start, zero_flag,
                   clk, rst);

output      reg    Load_words, shift, Add, latch;
input              multiplier_LSB, zero_flag;
input              start, clk, rst;
output             Ready;       // The multiplier is ready and the reset is not active

reg                ready;       // The multiplier is in the idle state

assign Ready = ready & (~rst);


parameter       S_idle          = 3'b000;
parameter       S_load          = 3'b001;
parameter       S_add           = 3'b010;
parameter       S_shift         = 3'b011;
parameter       S_latch         = 3'b100;
parameter       S_zeroCheck     = 3'b101;

reg [2:0]       state, next_state;


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

always @ *
begin
   Load_words = 1'b0;
   shift      = 1'b0;
   Add        = 1'b0;
   ready      = 1'b0;
   latch      = 1'b0;


   case (state)
      S_idle:       begin
                    ready = 1'b1;
                    if(start)  next_state = S_load;
                    else       next_state = S_idle;   end
      S_load:       begin
                    Load_words = 1'b1; next_state = S_zeroCheck; end
      S_zeroCheck:  begin
                         if (zero_flag)      next_state = S_latch;
                    else if (multiplier_LSB) next_state = S_add;
                    else                     next_state = S_shift;    end
      S_shift:      begin
                    shift = 1'b1;
                    next_state = S_zeroCheck;    end
      S_add:        begin
                    Add = 1'b1;
                    next_state = S_shift;    end
      S_latch:      begin
                    latch = 1'b1;
                    next_state = S_idle;     end
      default:      next_state = S_idle;
   endcase
end

endmodule
```

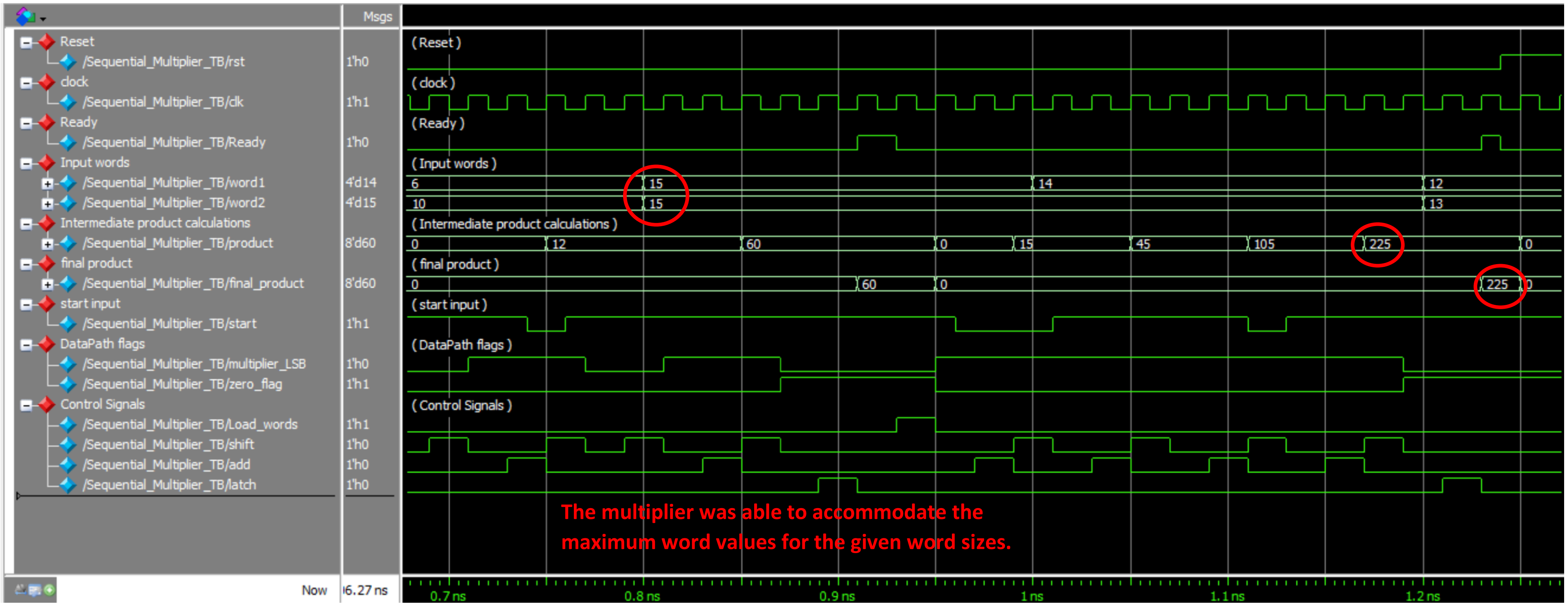
(10 points)


Wave - Default
Msgs
Reset
/Sequential_Multiplier_TB/rst 1'h0
clock
/Sequential_Multiplier_TB/clk 1'h0
Ready
/Sequential_Multiplier_TB/Ready 1'h0
Input words
/Sequential_Multiplier_TB/word1 4'd9
/Sequential_Multiplier_TB/word2 4'd11
Intermediate product calculations
/Sequential_Multiplier_TB/product 8'd0
final product
/Sequential_Multiplier_TB/final_product 8'd0
start input
/Sequential_Multiplier_TB/start 1'h1
DataPath flags
/Sequential_Multiplier_TB/multiplier_LSB 1'h0
/Sequential_Multiplier_TB/zero_flag 1'h1
Control Signals
/Sequential_Multiplier_TB/Load_words 1'h1
/Sequential_Multiplier_TB/shift 1'h0
/Sequential_Multiplier_TB/add 1'h0
/Sequential_Multiplier_TB/latch 1'h0
11
14
5
7
9
11
6
10
15
15
0
22
66
154
0
5
15
35
0
12
60
0
154
0
35
0
60
1110 = 11*2 + 11*4 + 11*8
0111 = 5*1 + 5*2 + 5*4
0110 = 6*2 + 6*4

The latency for calculating the product is data dependent. Numbers with a smaller number of ones finish faster, and smaller numbers with higher bits equal to zero latch faster as well.

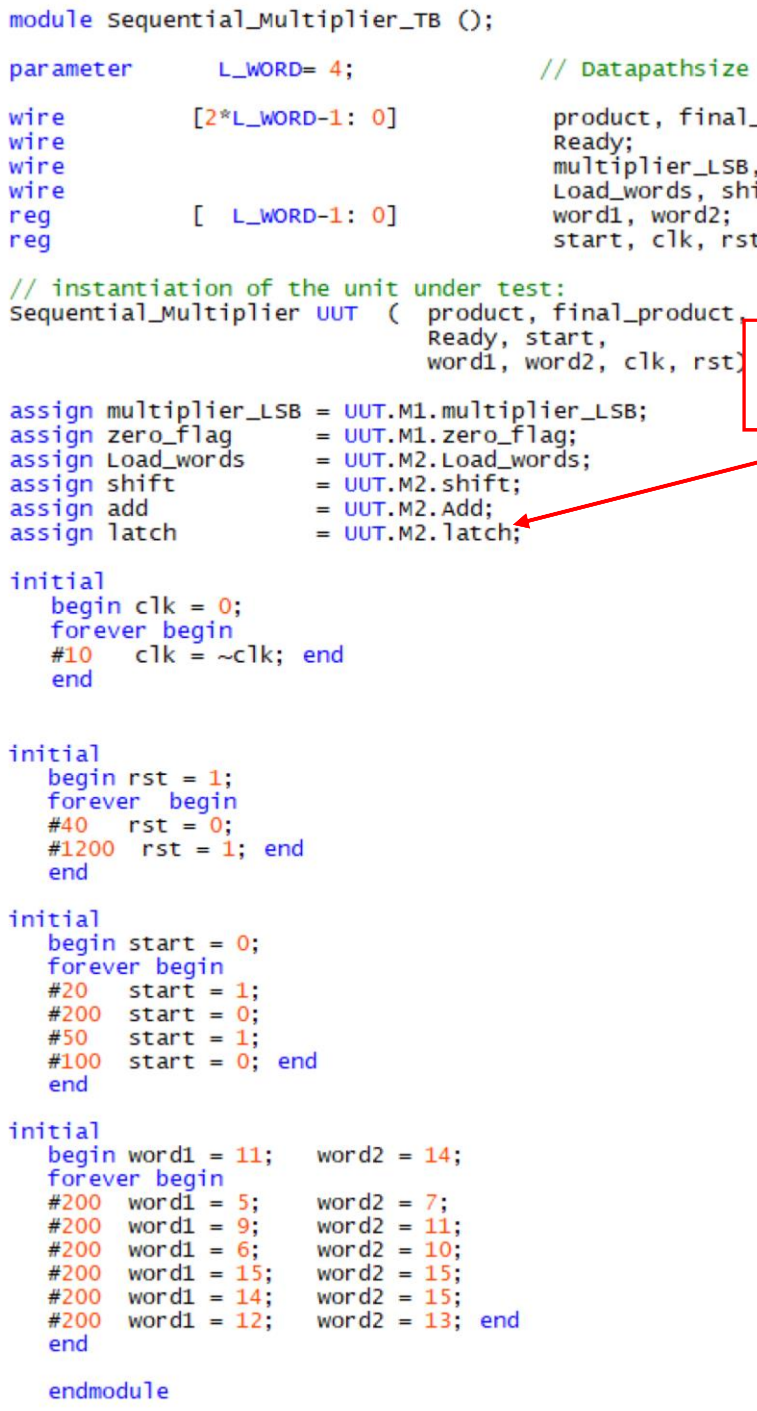
When the start is inactive the multiplier stays in the idle state. When the multiplier is ready and the start is activated, the word loading starts.

(5 points)



Msgs
Reset
/Sequential_Multiplier_TB/rst
1'h0
clock
/Sequential_Multiplier_TB/clk
1'h1
Ready
/Sequential_Multiplier_TB/Ready
1'h0
Input words
/Sequential_Multiplier_TB/word1
4'd14
/Sequential_Multiplier_TB/word2
4'd15
Intermediate product calculations
/Sequential_Multiplier_TB/product
8'd60
final product
/Sequential_Multiplier_TB/final_product
8'd60
start input
/Sequential_Multiplier_TB/start
1'h1
DataPath flags
/Sequential_Multiplier_TB/multiplier_LSB
1'h0
/Sequential_Multiplier_TB/zero_flag
1'h1
Control Signals
/Sequential_Multiplier_TB/Load_words
1'h1
/Sequential_Multiplier_TB/shift
1'h0
/Sequential_Multiplier_TB/add
1'h0
/Sequential_Multiplier_TB/latch
1'h0
Now
16.27 ns
(Reset)
(clock)
(Ready)
(Input words)
6
15
14
12
10
15
13
(Intermediate product calculations)
0
12
60
0
15
45
105
225
0
(final product)
0
60
0
225
0
(start input)
(DataPath flags)
(Control Signals)
0.7 ns
0.8 ns
0.9 ns
1 ns
1.1 ns
1.2 ns

The multiplier was able to accommodate the maximum word values for the given word sizes.

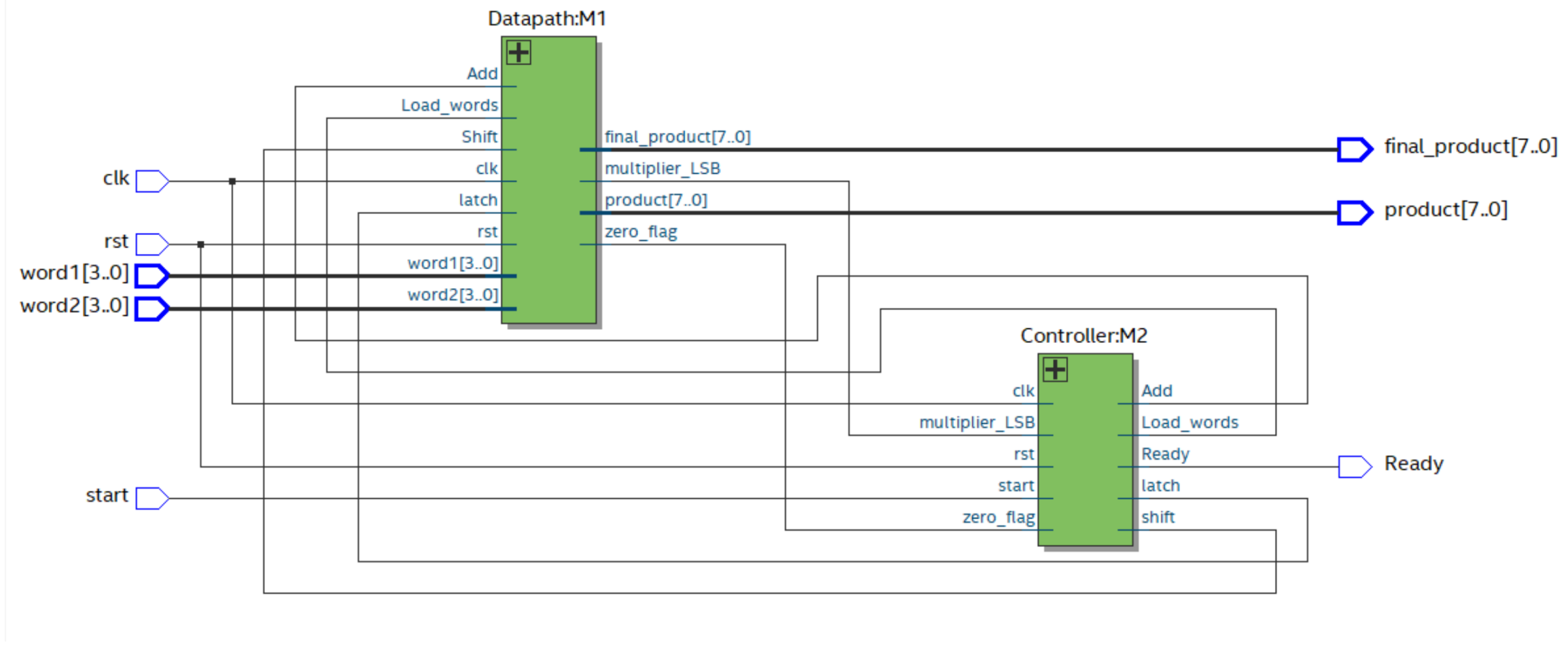


```
module Sequential_Multiplier_TB ();

parameter      L_WORD= 4;              // Datapathsize

wire        [2*L_WORD-1: 0]            product, final_product;
wire                                   Ready;
wire                                   multiplier_LSB, zero_flag;
wire                                   Load_words, shift,add, latch;
reg         [  L_WORD-1: 0]            word1, word2;
reg                                    start, clk, rst;

// instantiation of the unit under test:
Sequential_Multiplier UUT  (  product, final_product,
                              Ready, start,
                              word1, word2, clk, rst)

assign multiplier_LSB = UUT.M1.multiplier_LSB;
assign zero_flag      = UUT.M1.zero_flag;
assign Load_words     = UUT.M2.Load_words;
assign shift          = UUT.M2.shift;
assign add            = UUT.M2.Add;
assign latch          = UUT.M2.latch;

initial
   begin clk = 0;
   forever begin
   #10   clk = ~clk; end
   end


initial
   begin rst = 1;
   forever  begin
   #40   rst = 0;
   #1200  rst = 1; end
   end

initial
   begin start = 0;
   forever begin
   #20   start = 1;
   #200  start = 0;
   #50   start = 1;
   #100  start = 0; end
   end

initial
   begin word1 = 11;    word2 = 14;
   forever begin
   #200  word1 = 5;     word2 = 7;
   #200  word1 = 9;     word2 = 11;
   #200  word1 = 6;     word2 = 10;
   #200  word1 = 15;    word2 = 15;
   #200  word1 = 14;    word2 = 15;
   #200  word1 = 12;    word2 = 13;  end
   end

   endmodule
```

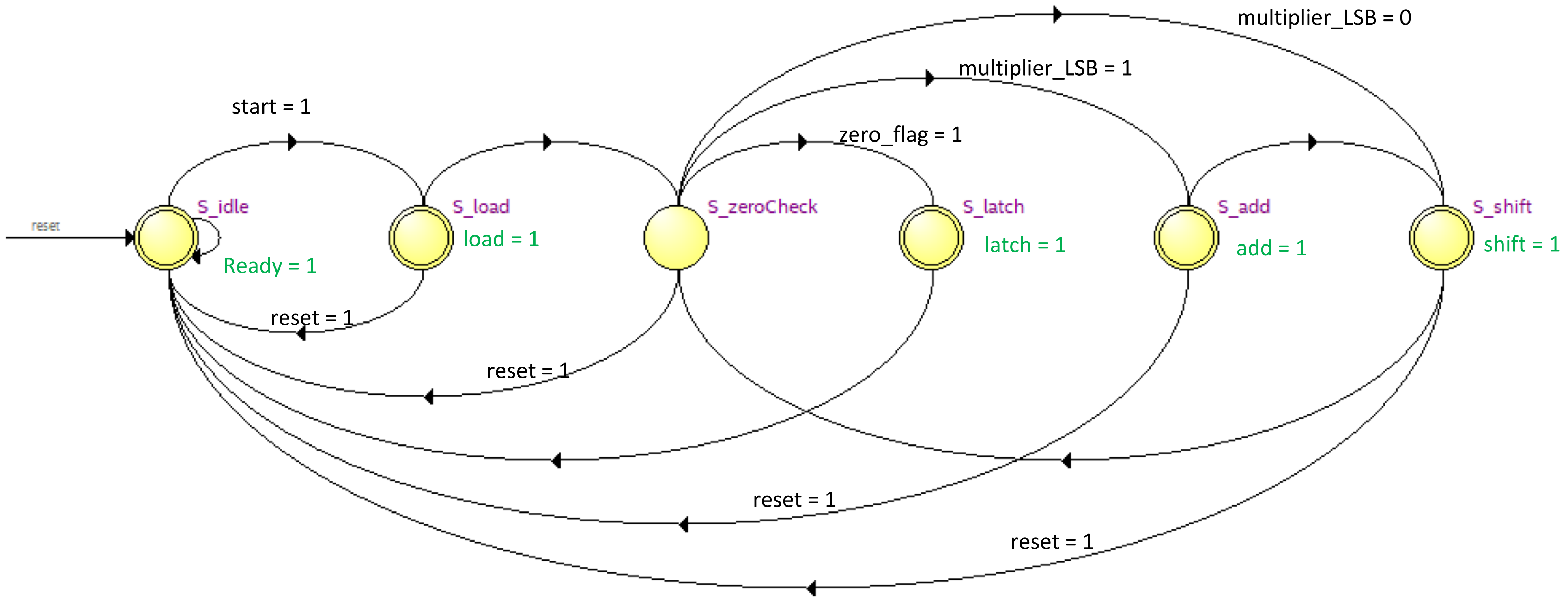
Probes to observe the internal control signals and verify the FSM operation.

(5 points)

Netlist Viewer



Finite State Machine State Graph



(5 points)