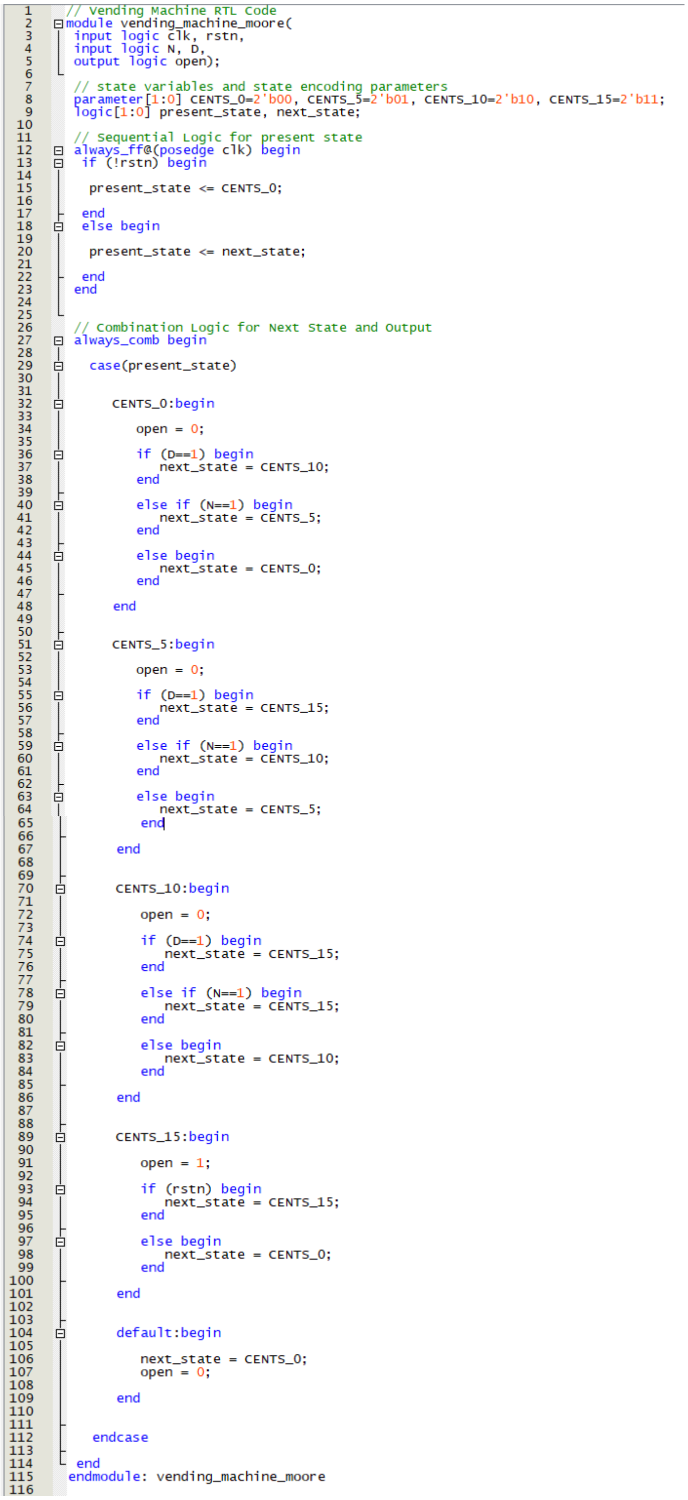
ECE 111 Winter 2022
HW6
Hao Le A15547504

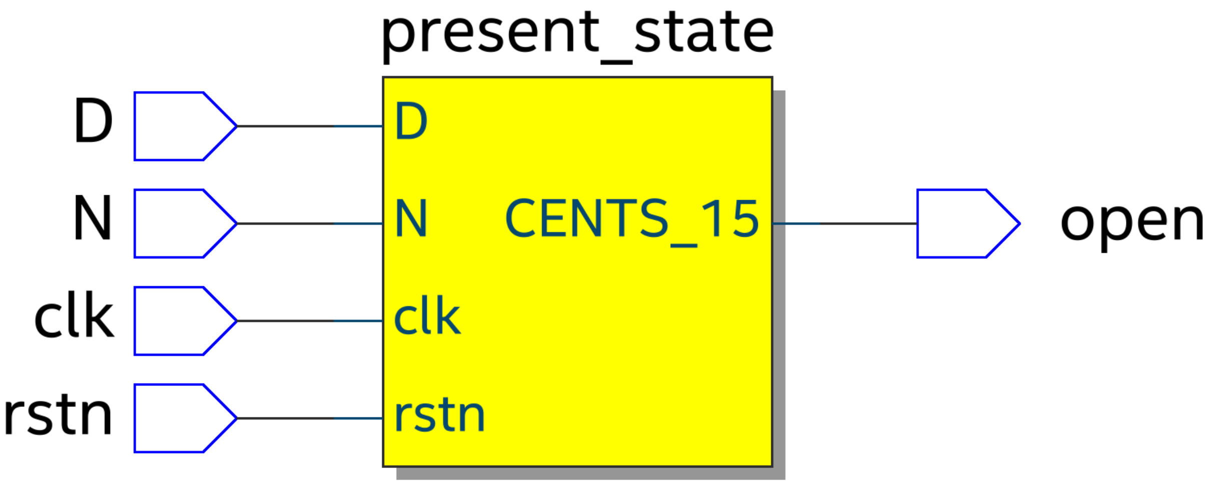
Moore Vending Machine

Code

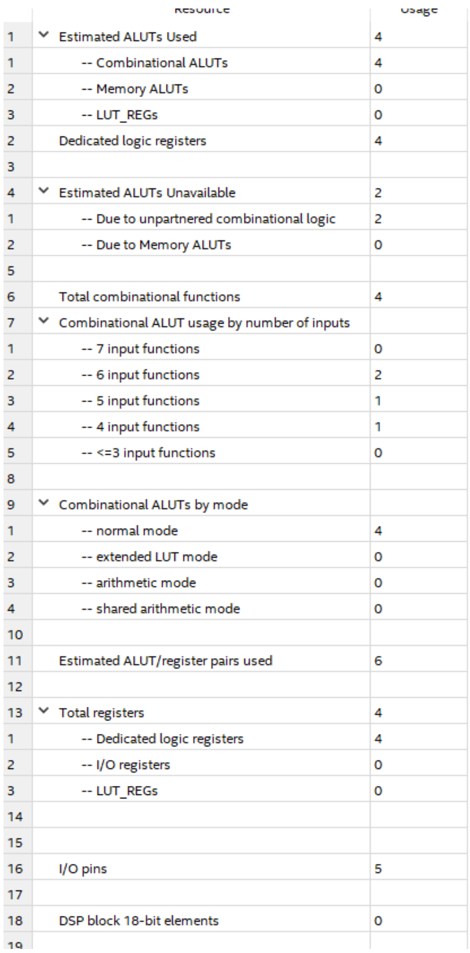


```
// Vending Machine RTL Code
module vending_machine_moore(
 input logic clk, rstn,
 input logic N, D,
 output logic open);

 // state variables and state encoding parameters
 parameter[1:0] CENTS_0=2'b00, CENTS_5=2'b01, CENTS_10=2'b10, CENTS_15=2'b11;
 logic[1:0] present_state, next_state;

 // Sequential Logic for present state
 always_ff@(posedge clk) begin
  if (!rstn) begin

   present_state <= CENTS_0;

  end
  else begin

   present_state <= next_state;

  end
 end


 // Combination Logic for Next State and Output
 always_comb begin

   case(present_state)


      CENTS_0:begin

         open = 0;

         if (D==1) begin
            next_state = CENTS_10;
         end

         else if (N==1) begin
            next_state = CENTS_5;
         end

         else begin
            next_state = CENTS_0;
         end

      end


      CENTS_5:begin

         open = 0;

         if (D==1) begin
            next_state = CENTS_15;
         end

         else if (N==1) begin
            next_state = CENTS_10;
         end

         else begin
            next_state = CENTS_5;
          end

       end


       CENTS_10:begin

          open = 0;

          if (D==1) begin
             next_state = CENTS_15;
          end

          else if (N==1) begin
             next_state = CENTS_15;
          end

          else begin
             next_state = CENTS_10;
          end

       end


       CENTS_15:begin

          open = 1;

          if (rstn) begin
             next_state = CENTS_15;
          end

          else begin
             next_state = CENTS_0;
          end

       end


       default:begin

          next_state = CENTS_0;
          open = 0;

       end


    endcase

  end
endmodule: vending_machine_moore
```

RTL netlist

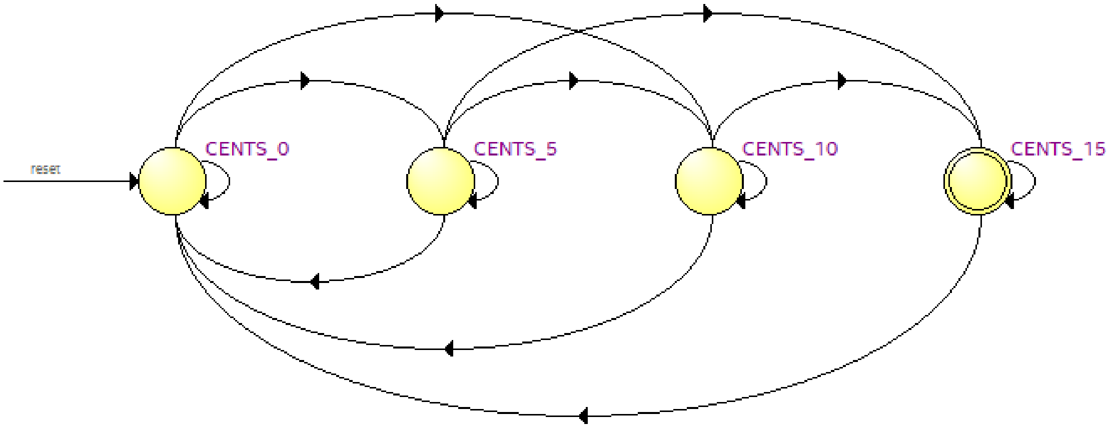


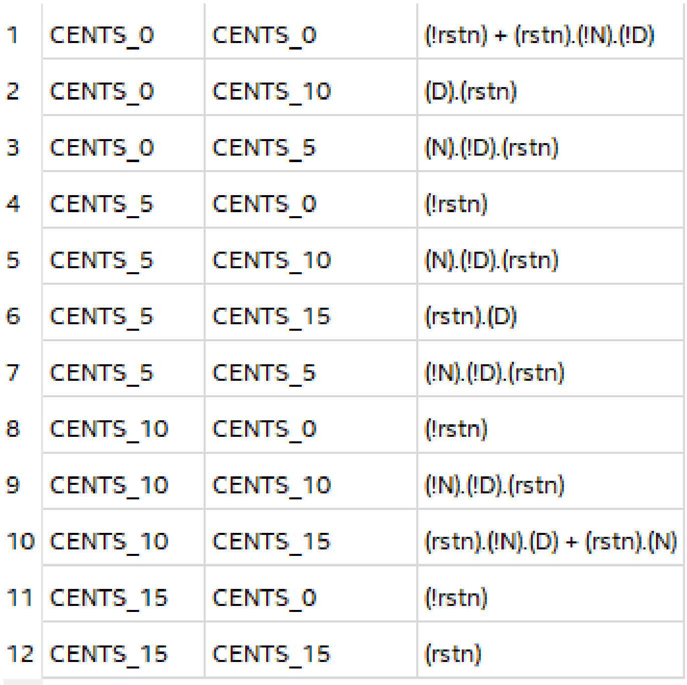
Resource usage



	Resource	Usage
1	Estimated ALUTs Used	4
1	-- Combinational ALUTs	4
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	4
3		
4	Estimated ALUTs Unavailable	2
1	-- Due to unpartnered combinational logic	2
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	4
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	2
3	-- 5 input functions	1
4	-- 4 input functions	1
5	-- <=3 input functions	0
8		
9	Combinational ALUTs by mode	
1	-- normal mode	4
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	6
12		
13	Total registers	4
1	-- Dedicated logic registers	4
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	5
17		
18	DSP block 18-bit elements	0
19		

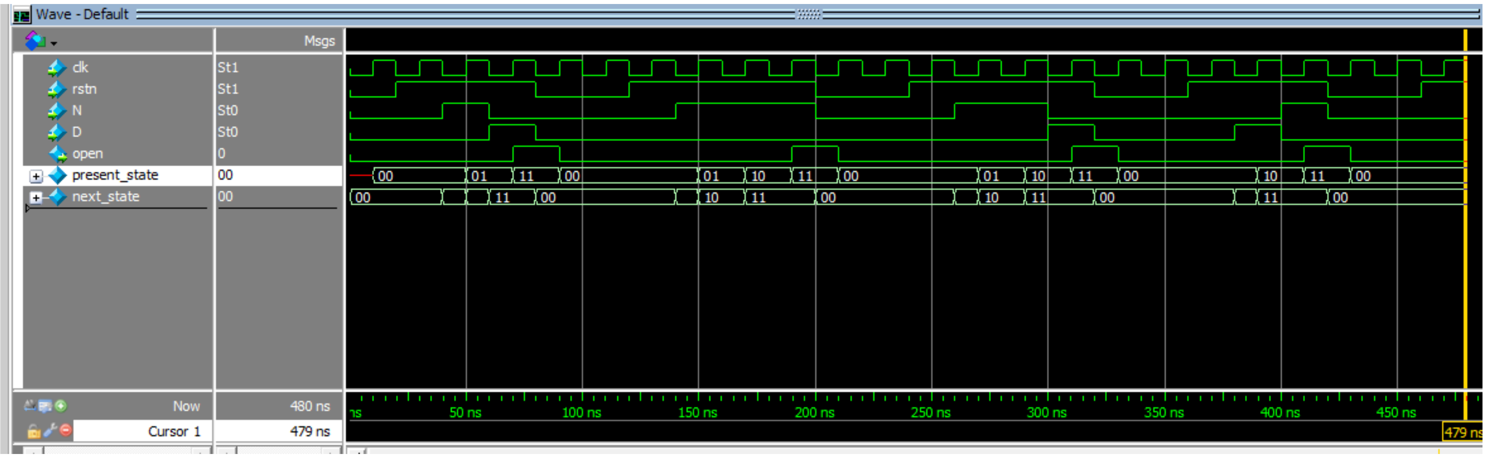
State diagram





1	CENTS_0	CENTS_0	(!rstn) + (rstn).(!N).(!D)
2	CENTS_0	CENTS_10	(D).(rstn)
3	CENTS_0	CENTS_5	(N).(!D).(rstn)
4	CENTS_5	CENTS_0	(!rstn)
5	CENTS_5	CENTS_10	(N).(!D).(rstn)
6	CENTS_5	CENTS_15	(rstn).(D)
7	CENTS_5	CENTS_5	(!N).(!D).(rstn)
8	CENTS_10	CENTS_0	(!rstn)
9	CENTS_10	CENTS_10	(!N).(!D).(rstn)
10	CENTS_10	CENTS_15	(rstn).(!N).(D) + (rstn).(N)
11	CENTS_15	CENTS_0	(!rstn)
12	CENTS_15	CENTS_15	(rstn)

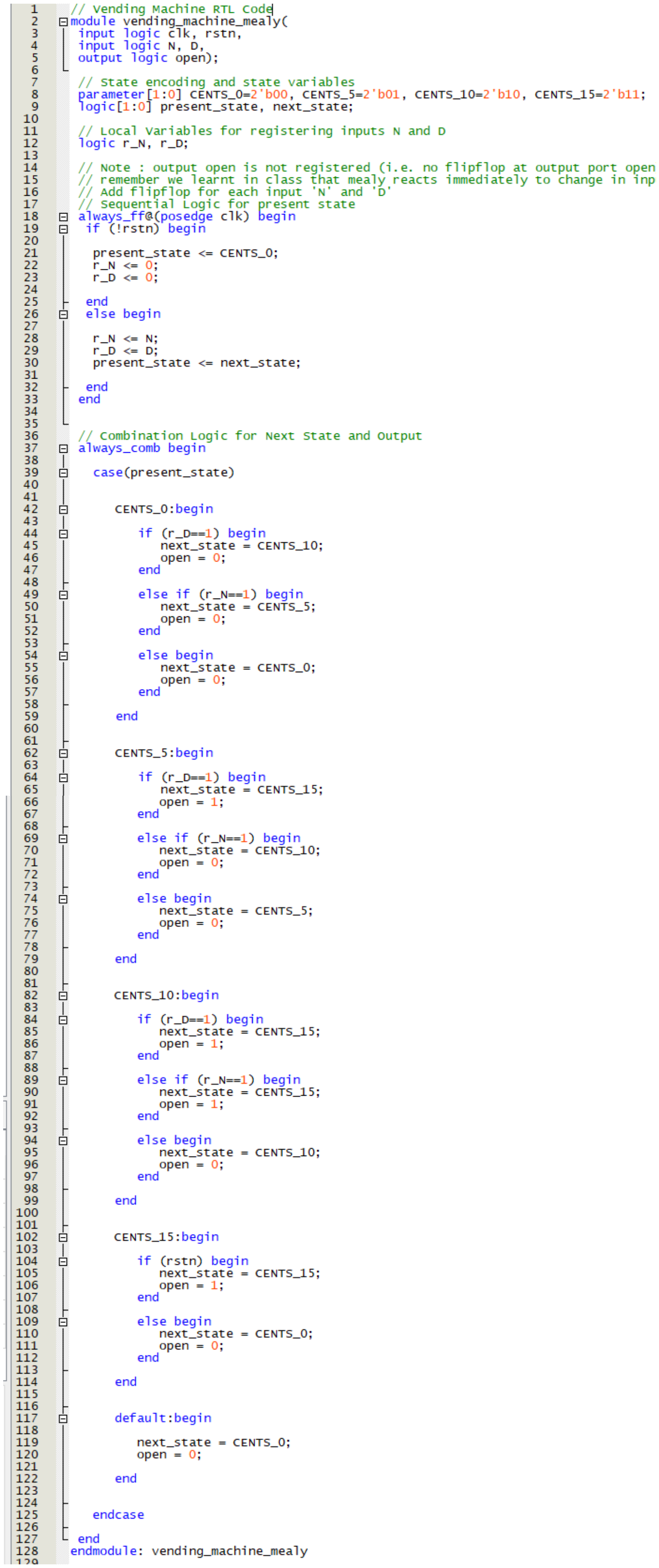
Testbench simulation waveform



- A Moore state machine generates an output based on the current state, regardless of the inputs
- We see this behavior in the waveform; for example, in the first operation starting at around 30ns, a nickel is inserted, but since it was inserted at the falling edge of the clock, the state does not change from 00 to 01 (5 cents) until the next rising clock edge. We see this for the proceeding dime, where upon the rising edge, the state changes from 5 cents to 15 cents, thus bringing the output high
- This is distinct from a Mealy state machine in that the output did not go high immediately after the dime was inserted
- Lastly, once the 15 Cents (11) state has been reached, rstn goes low, but only at the rising clock edge does the present state switch back to 0 Cents, and the output goes low
- Described above is the correct behavior

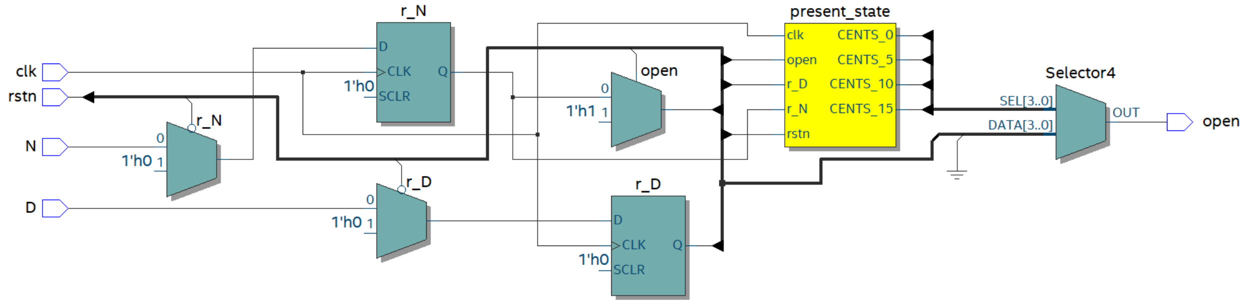
Mealy Vending Machine

Code

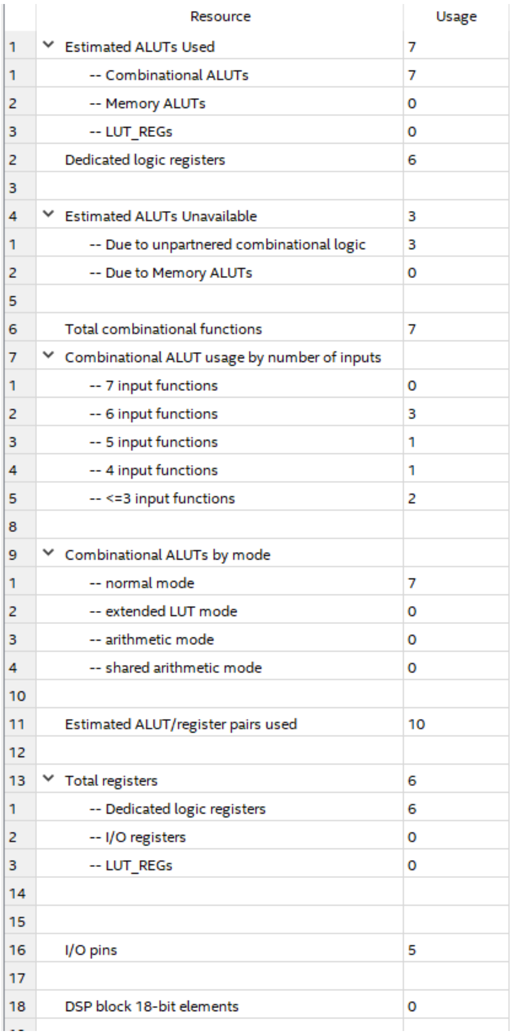


```
// Vending Machine RTL Code
module vending_machine_mealy(
 input logic clk, rstn,
 input logic N, D,
 output logic open);

 // State encoding and state variables
 parameter[1:0] CENTS_0=2'b00, CENTS_5=2'b01, CENTS_10=2'b10, CENTS_15=2'b11;
 logic[1:0] present_state, next_state;

 // Local Variables for registering inputs N and D
 logic r_N, r_D;

 // Note : output open is not registered (i.e. no flipflop at output port open
 // remember we learnt in class that mealy reacts immediately to change in inp
 // Add flipflop for each input 'N' and 'D'
 // Sequential Logic for present state
 always_ff@(posedge clk) begin
  if (!rstn) begin

   present_state <= CENTS_0;
   r_N <= 0;
   r_D <= 0;

  end
  else begin

   r_N <= N;
   r_D <= D;
   present_state <= next_state;

  end
 end


 // Combination Logic for Next State and Output
 always_comb begin

   case(present_state)


      CENTS_0:begin

         if (r_D==1) begin
            next_state = CENTS_10;
            open = 0;
         end

         else if (r_N==1) begin
            next_state = CENTS_5;
            open = 0;
         end

         else begin
            next_state = CENTS_0;
            open = 0;
         end

      end


      CENTS_5:begin

         if (r_D==1) begin
            next_state = CENTS_15;
            open = 1;
         end

         else if (r_N==1) begin
            next_state = CENTS_10;
            open = 0;
         end

         else begin
            next_state = CENTS_5;
            open = 0;
         end

      end


      CENTS_10:begin

         if (r_D==1) begin
            next_state = CENTS_15;
            open = 1;
         end

         else if (r_N==1) begin
            next_state = CENTS_15;
            open = 1;
         end

         else begin
            next_state = CENTS_10;
            open = 0;
         end

      end


      CENTS_15:begin

         if (rstn) begin
            next_state = CENTS_15;
            open = 1;
         end

         else begin
            next_state = CENTS_0;
            open = 0;
         end

      end


      default:begin

         next_state = CENTS_0;
         open = 0;

      end


   endcase

 end
endmodule: vending_machine_mealy
```

RTL netlist

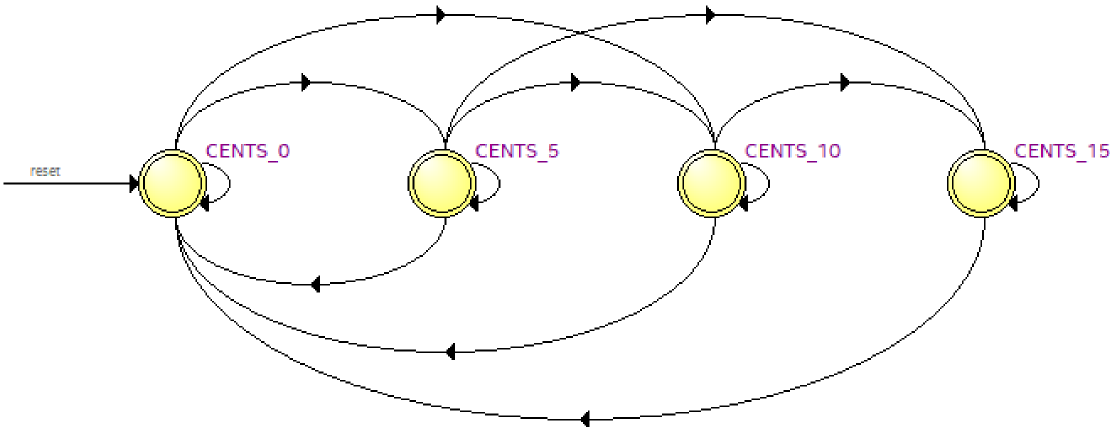


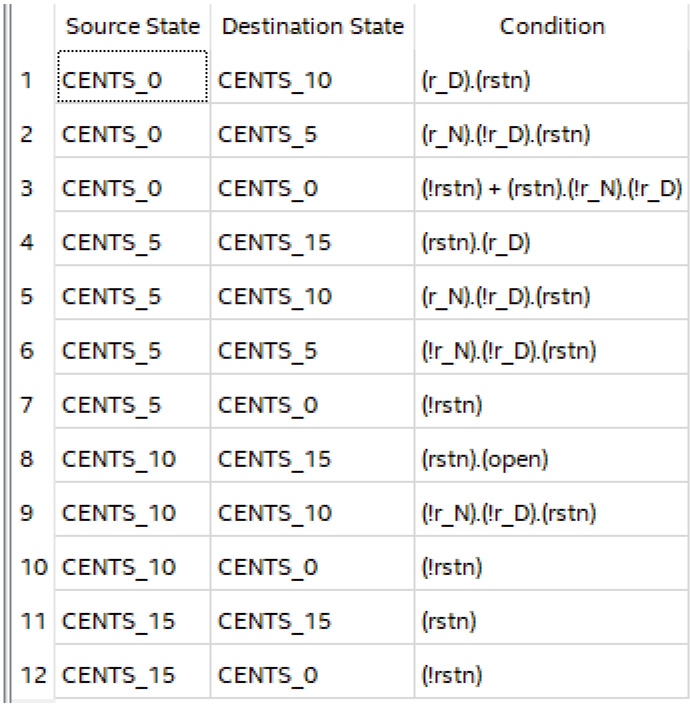
Resource usage



	Resource	Usage
1	⌄ Estimated ALUTs Used	7
1	-- Combinational ALUTs	7
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	6
3		
4	⌄ Estimated ALUTs Unavailable	3
1	-- Due to unpartnered combinational logic	3
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	7
7	⌄ Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	3
3	-- 5 input functions	1
4	-- 4 input functions	1
5	-- <=3 input functions	2
8		
9	⌄ Combinational ALUTs by mode	
1	-- normal mode	7
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	10
12		
13	⌄ Total registers	6
1	-- Dedicated logic registers	6
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	5
17		
18	DSP block 18-bit elements	0

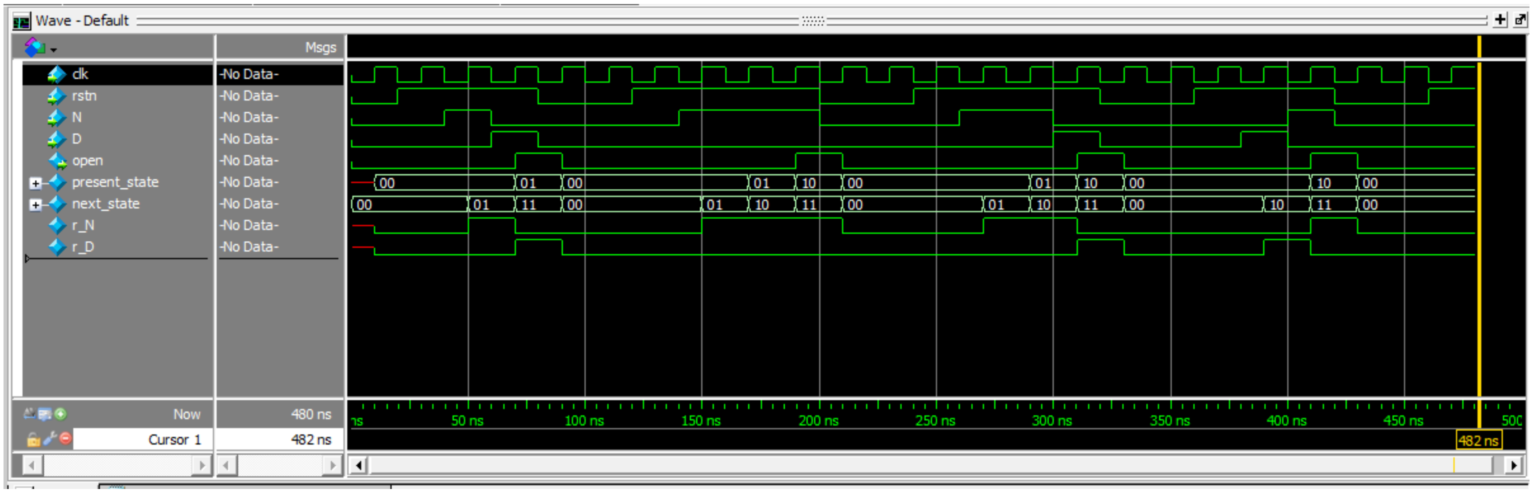
State diagram





	Source State	Destination State	Condition
1	CENTS_0	CENTS_10	(r_D).(rstn)
2	CENTS_0	CENTS_5	(r_N).(!r_D).(rstn)
3	CENTS_0	CENTS_0	(!rstn) + (rstn).(!r_N).(!r_D)
4	CENTS_5	CENTS_15	(rstn).(r_D)
5	CENTS_5	CENTS_10	(r_N).(!r_D).(rstn)
6	CENTS_5	CENTS_5	(!r_N).(!r_D).(rstn)
7	CENTS_5	CENTS_0	(!rstn)
8	CENTS_10	CENTS_15	(rstn).(open)
9	CENTS_10	CENTS_10	(!r_N).(!r_D).(rstn)
10	CENTS_10	CENTS_0	(!rstn)
11	CENTS_15	CENTS_15	(rstn)
12	CENTS_15	CENTS_0	(!rstn)

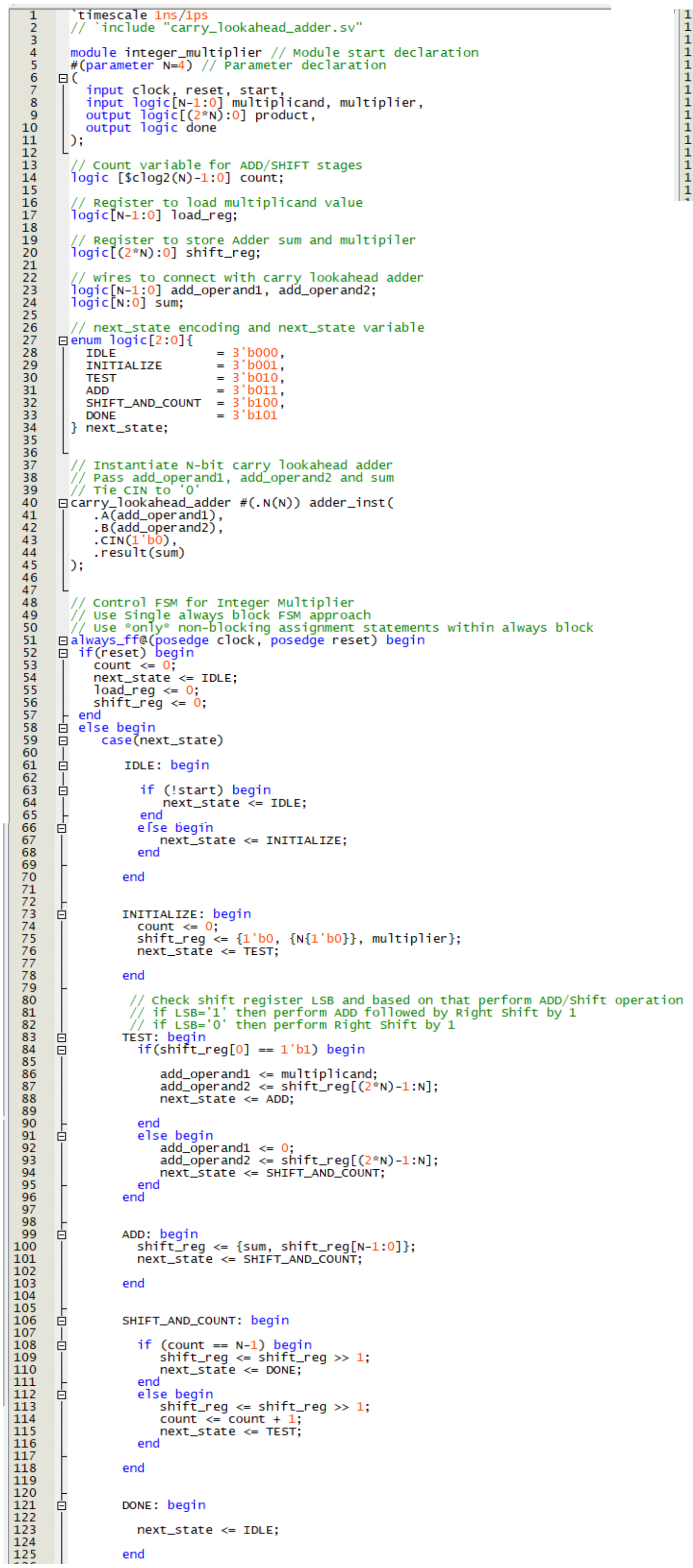
Testbench simulation waveform



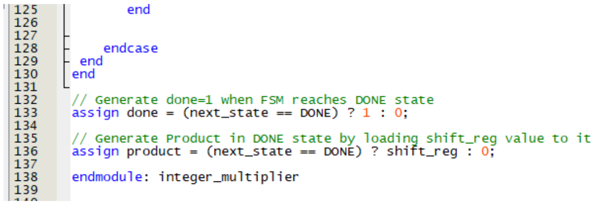
- A Mealy state machine generates an output based on the present state and the inputs; this differs from a Moore in that the output reacts immediately to a changing input
- In this implementation, flip flops were added to the inputs to delay them by half a clock cycle; this is to remedy the issue of double counting, especially during the third operation of the vending machine at around 260ns. As a side effect, the actual operation of this state machine is exactly like a Moore
- Without the flip flops, the inserted nickel would be counted three times, thus triggering the output before the dime even gets inserted which is incorrect behavior
- The “ideal” inputs would be the outputs of the registers
- Focusing on r_N and r_D as the inputs now, we see that at 50ns, the first nickel gets inserted, and the state is changed to 01 (5 Cents) at the positive clock edge
- Then, a dime is inserted, and the state changes to 11 (15 Cents) at the next positive clock edge
- We also observe that the output goes high as soon as the dime is inserted even though at that point, the present state is still 01 (5 Cents)
- This is because the Mealy machine knows the next state will be 15 Cents, so it triggers the output beforehand, saving a clock cycle.
- Described above is the correct behavior, assuming the inputs are timed correctly

Integer Multiplier

Code

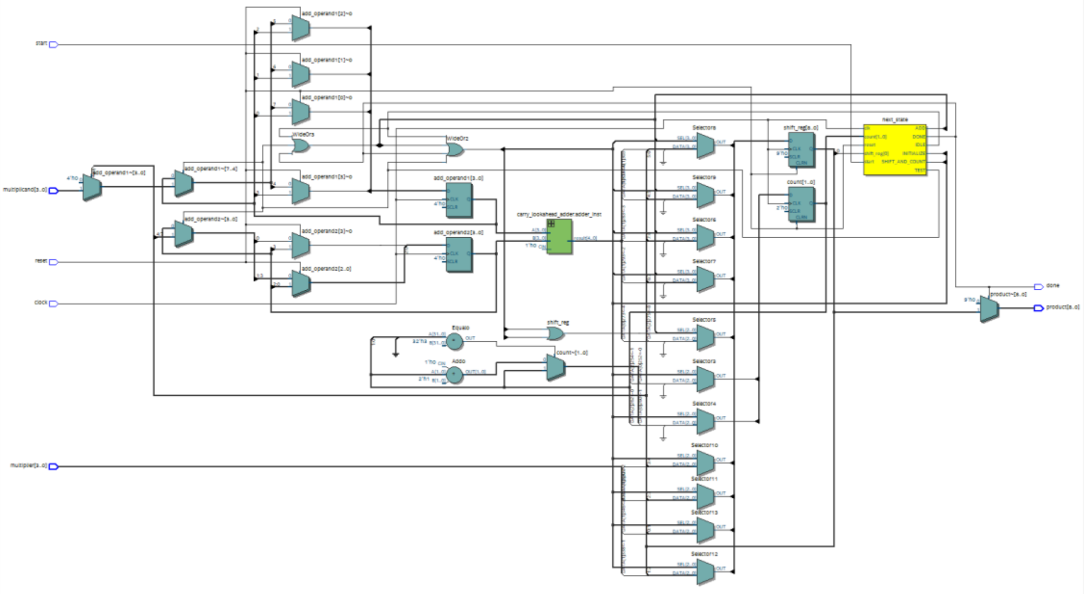


```
`timescale 1ns/1ps
// `include "carry_lookahead_adder.sv"

module integer_multiplier // Module start declaration
#(parameter N=4) // Parameter declaration
(
  input clock, reset, start,
  input logic[N-1:0] multiplicand, multiplier,
  output logic[(2*N):0] product,
  output logic done
);

// Count variable for ADD/SHIFT stages
logic [$clog2(N)-1:0] count;

// Register to load multiplicand value
logic[N-1:0] load_reg;

// Register to store Adder sum and multipiler
logic[(2*N):0] shift_reg;

// wires to connect with carry lookahead adder
logic[N-1:0] add_operand1, add_operand2;
logic[N:0] sum;

// next_state encoding and next_state variable
enum logic[2:0]{
  IDLE                = 3'b000,
  INITIALIZE          = 3'b001,
  TEST                = 3'b010,
  ADD                 = 3'b011,
  SHIFT_AND_COUNT     = 3'b100,
  DONE                = 3'b101
} next_state;


// Instantiate N-bit carry lookahead adder
// Pass add_operand1, add_operand2 and sum
// Tie CIN to '0'
carry_lookahead_adder #(.N(N)) adder_inst(
   .A(add_operand1),
   .B(add_operand2),
   .CIN(1'b0),
   .result(sum)
);


// Control FSM for Integer Multiplier
// Use Single always block FSM approach
// Use *only* non-blocking assignment statements within always block
always_ff@(posedge clock, posedge reset) begin
 if(reset) begin
   count <= 0;
   next_state <= IDLE;
   load_reg <= 0;
   shift_reg <= 0;
 end
 else begin
    case(next_state)

       IDLE: begin

         if (!start) begin
            next_state <= IDLE;
         end
         else begin
            next_state <= INITIALIZE;
         end

       end


       INITIALIZE: begin
         count <= 0;
         shift_reg <= {1'b0, {N{1'b0}}, multiplier};
         next_state <= TEST;

       end

        // Check shift register LSB and based on that perform ADD/Shift operation
        // if LSB='1' then perform ADD followed by Right Shift by 1
        // if LSB='0' then perform Right Shift by 1
       TEST: begin
         if(shift_reg[0] == 1'b1) begin

            add_operand1 <= multiplicand;
            add_operand2 <= shift_reg[(2*N)-1:N];
            next_state <= ADD;

         end
         else begin
            add_operand1 <= 0;
            add_operand2 <= shift_reg[(2*N)-1:N];
            next_state <= SHIFT_AND_COUNT;
         end
       end


       ADD: begin
         shift_reg <= {sum, shift_reg[N-1:0]};
         next_state <= SHIFT_AND_COUNT;

       end


       SHIFT_AND_COUNT: begin

         if (count == N-1) begin
            shift_reg <= shift_reg >> 1;
            next_state <= DONE;
         end
         else begin
            shift_reg <= shift_reg >> 1;
            count <= count + 1;
            next_state <= TEST;
         end

       end


       DONE: begin

         next_state <= IDLE;

       end
```

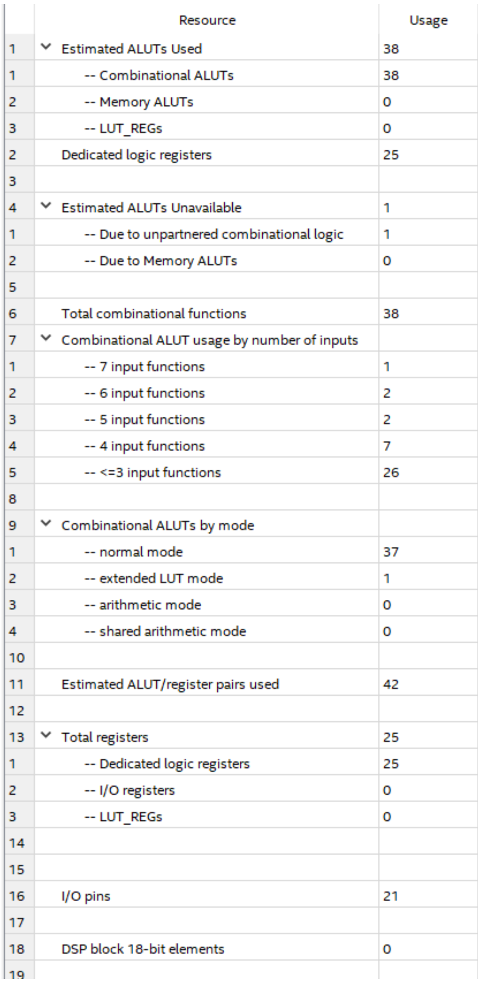


```
       end


    endcase
 end
end

// Generate done=1 when FSM reaches DONE state
assign done = (next_state == DONE) ? 1 : 0;

// Generate Product in DONE state by loading shift_reg value to it
assign product = (next_state == DONE) ? shift_reg : 0;

endmodule: integer_multiplier
```

RTL netlist

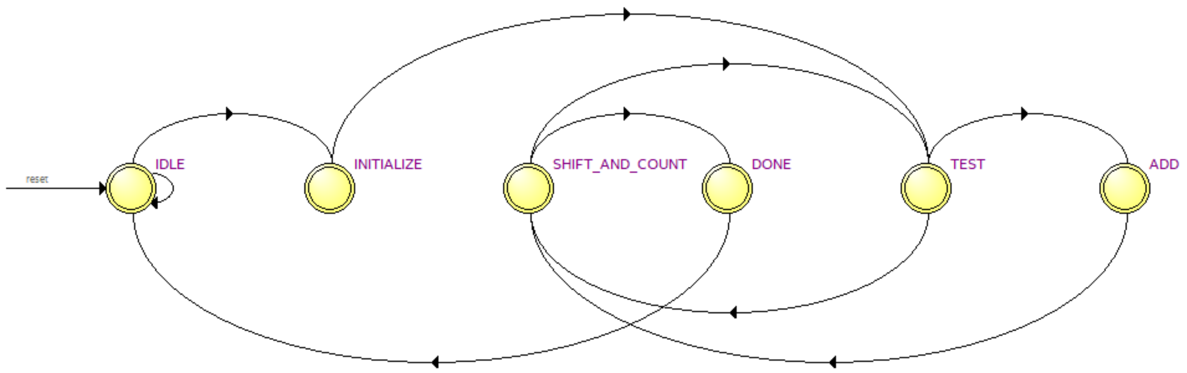


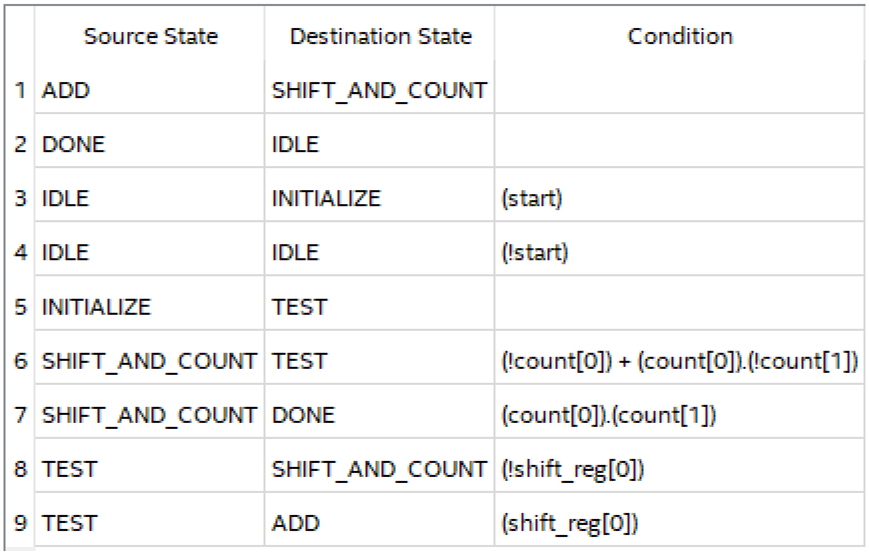
Resource usage



	Resource	Usage
1	⌄ Estimated ALUTs Used	38
1	-- Combinational ALUTs	38
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	25
3		
4	⌄ Estimated ALUTs Unavailable	1
1	-- Due to unpartnered combinational logic	1
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	38
7	⌄ Combinational ALUT usage by number of inputs	
1	-- 7 input functions	1
2	-- 6 input functions	2
3	-- 5 input functions	2
4	-- 4 input functions	7
5	-- <=3 input functions	26
8		
9	⌄ Combinational ALUTs by mode	
1	-- normal mode	37
2	-- extended LUT mode	1
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	42
12		
13	⌄ Total registers	25
1	-- Dedicated logic registers	25
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	21
17		
18	DSP block 18-bit elements	0
19		

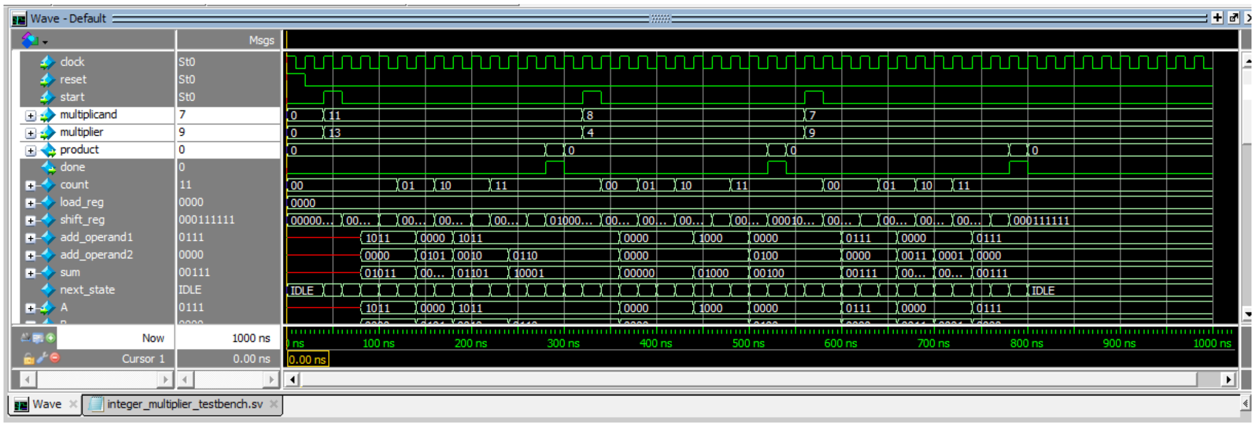
State diagram

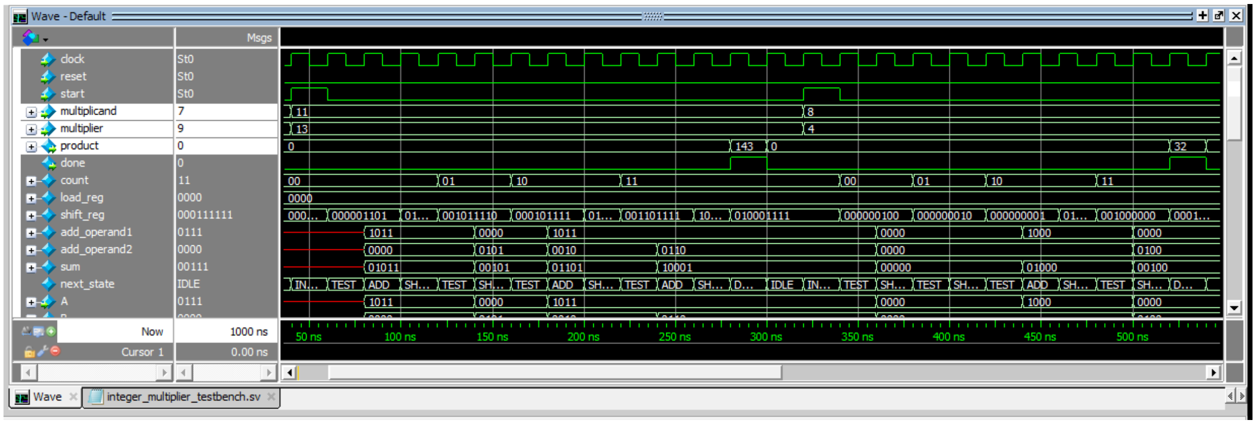


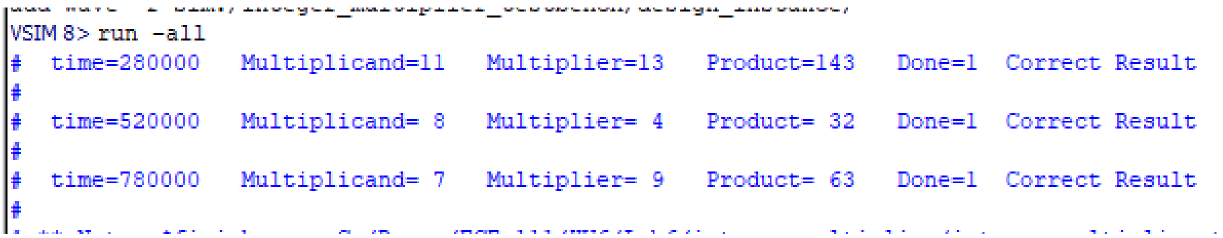


	Source State	Destination State	Condition
1	ADD	SHIFT_AND_COUNT	
2	DONE	IDLE	
3	IDLE	INITIALIZE	(start)
4	IDLE	IDLE	(!start)
5	INITIALIZE	TEST	
6	SHIFT_AND_COUNT	TEST	(!count[0]) + (count[0]).(!count[1])
7	SHIFT_AND_COUNT	DONE	(count[0]).(count[1])
8	TEST	SHIFT_AND_COUNT	(!shift_reg[0])
9	TEST	ADD	(shift_reg[0])

Testbench simulation waveform





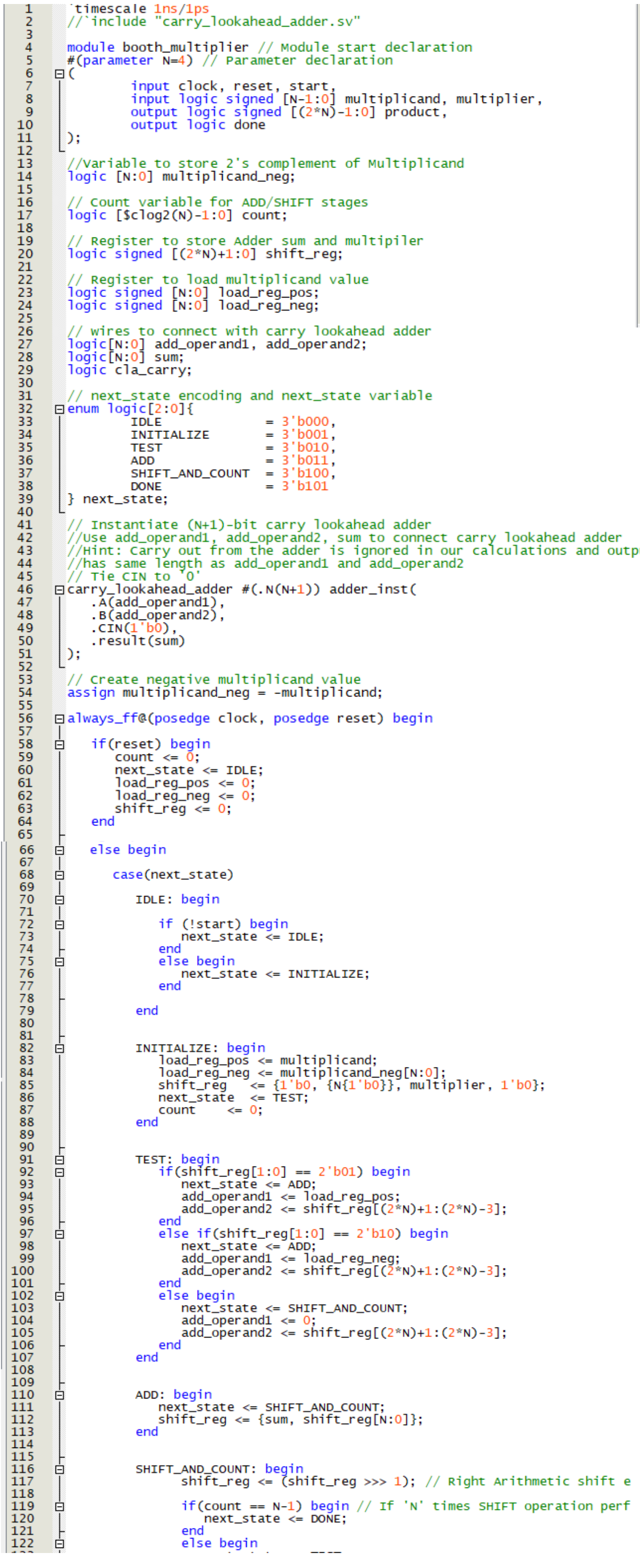


```
VSIM 8> run -all
#  time=280000   Multiplicand=11   Multiplier=13   Product=143   Done=1  Correct Result
#
#  time=520000   Multiplicand= 8   Multiplier= 4   Product= 32   Done=1  Correct Result
#
#  time=780000   Multiplicand= 7   Multiplier= 9   Product= 63   Done=1  Correct Result
#
```

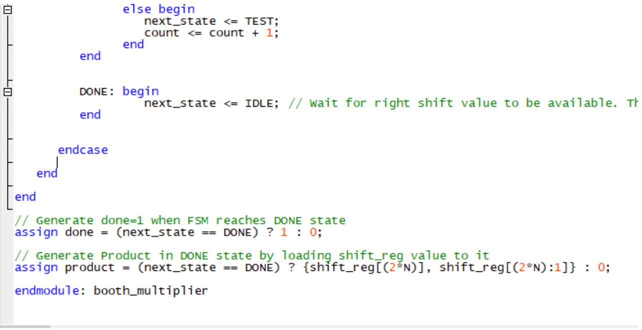
- The waveforms shown are of 3 views: the overall waveform, a zoomed in section to show the correct results from input pairs, and the transcript to demonstrate that each case yielded the correct result
- The multiplier behaves correctly because upon loading two numbers to be multiplied, and by setting start to high, this starts the internal cycle of states, and once the final product has been calculated, it is outputted with the done signal going high
- Then upon the next clock cycle, product resets to 0, and the multiplier idles until a new pair of numbers is loaded along with the pull up of the start input

Booth Multiplier

Code

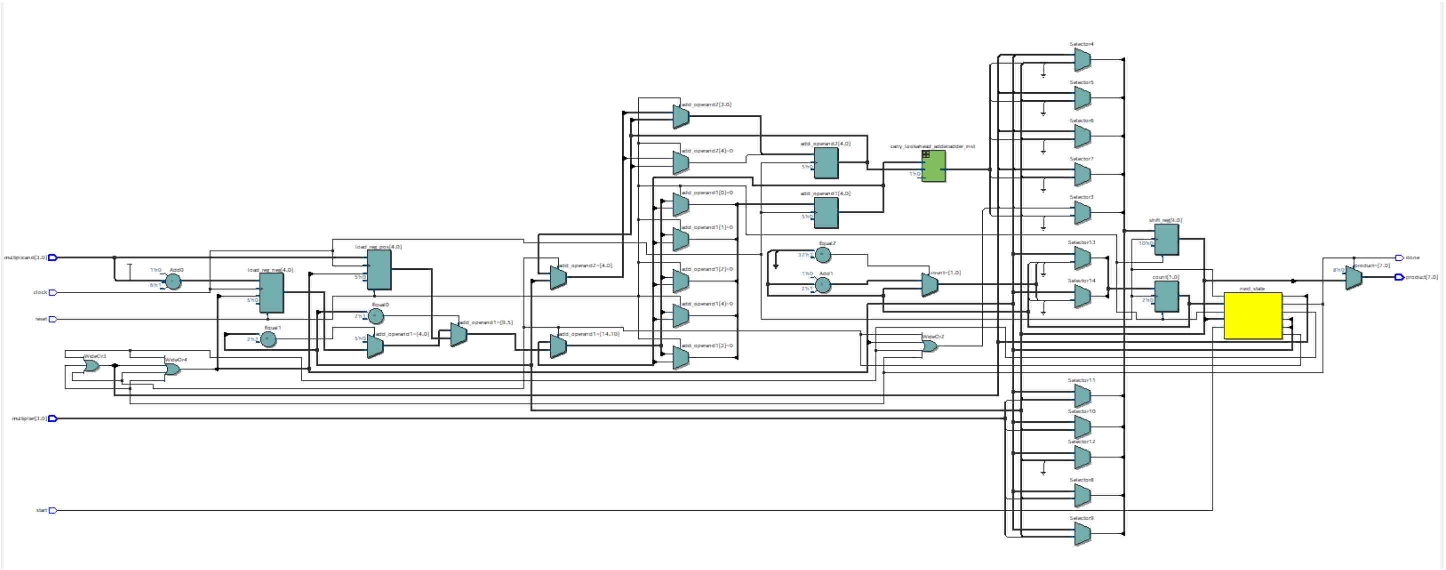


```
`timescale 1ns/1ps
//`include "carry_lookahead_adder.sv"

module booth_multiplier // Module start declaration
#(parameter N=4) // Parameter declaration
(
        input clock, reset, start,
        input logic signed [N-1:0] multiplicand, multiplier,
        output logic signed [(2*N)-1:0] product,
        output logic done
);

//Variable to store 2's complement of Multiplicand
logic [N:0] multiplicand_neg;

// Count variable for ADD/SHIFT stages
logic [$clog2(N)-1:0] count;

// Register to store Adder sum and multipiler
logic signed [(2*N)+1:0] shift_reg;

// Register to load multiplicand value
logic signed [N:0] load_reg_pos;
logic signed [N:0] load_reg_neg;

// wires to connect with carry lookahead adder
logic[N:0] add_operand1, add_operand2;
logic[N:0] sum;
logic cla_carry;

// next_state encoding and next_state variable
enum logic[2:0]{
        IDLE                = 3'b000,
        INITIALIZE          = 3'b001,
        TEST                = 3'b010,
        ADD                 = 3'b011,
        SHIFT_AND_COUNT     = 3'b100,
        DONE                = 3'b101
} next_state;

// Instantiate (N+1)-bit carry lookahead adder
//Use add_operand1, add_operand2, sum to connect carry lookahead adder
//Hint: Carry out from the adder is ignored in our calculations and output sum
//has same length as add_operand1 and add_operand2
// Tie CIN to '0'
carry_lookahead_adder #(.N(N+1)) adder_inst(
   .A(add_operand1),
   .B(add_operand2),
   .CIN(1'b0),
   .result(sum)
);

// Create negative multiplicand value
assign multiplicand_neg = -multiplicand;

always_ff@(posedge clock, posedge reset) begin

   if(reset) begin
      count <= 0;
      next_state <= IDLE;
      load_reg_pos <= 0;
      load_reg_neg <= 0;
      shift_reg <= 0;
   end

   else begin

      case(next_state)

         IDLE: begin

            if (!start) begin
               next_state <= IDLE;
            end
            else begin
               next_state <= INITIALIZE;
            end

         end


         INITIALIZE: begin
            load_reg_pos <= multiplicand;
            load_reg_neg <= multiplicand_neg[N:0];
            shift_reg   <= {1'b0, {N{1'b0}}, multiplier, 1'b0};
            next_state  <= TEST;
            count    <= 0;
         end


         TEST: begin
            if(shift_reg[1:0] == 2'b01) begin
               next_state <= ADD;
               add_operand1 <= load_reg_pos;
               add_operand2 <= shift_reg[(2*N)+1:(2*N)-3];
            end
            else if(shift_reg[1:0] == 2'b10) begin
               next_state <= ADD;
               add_operand1 <= load_reg_neg;
               add_operand2 <= shift_reg[(2*N)+1:(2*N)-3];
            end
            else begin
               next_state <= SHIFT_AND_COUNT;
               add_operand1 <= 0;
               add_operand2 <= shift_reg[(2*N)+1:(2*N)-3];
            end
         end


         ADD: begin
            next_state <= SHIFT_AND_COUNT;
            shift_reg <= {sum, shift_reg[N:0]};
         end


         SHIFT_AND_COUNT: begin
               shift_reg <= (shift_reg >>> 1); // Right Arithmetic shift e

               if(count == N-1) begin // If 'N' times SHIFT operation perf
                  next_state <= DONE;
               end
               else begin
```

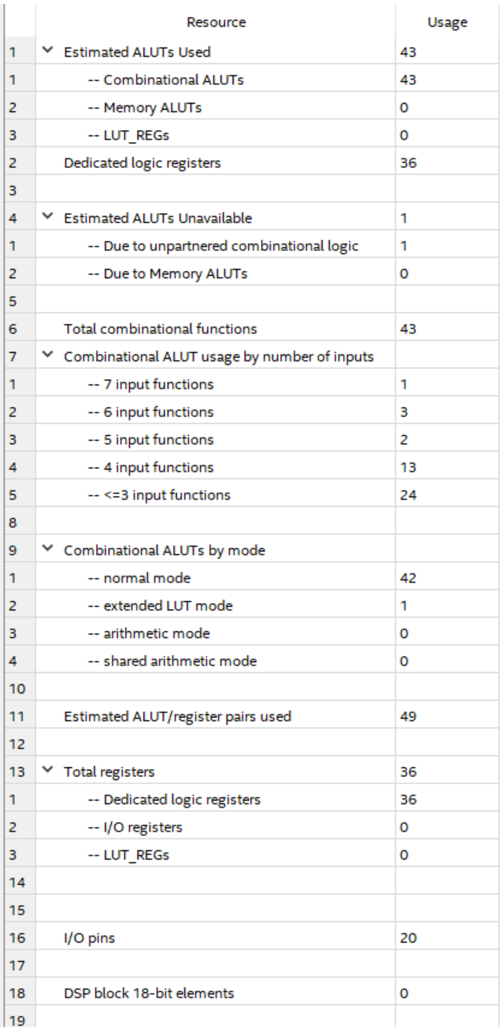


```
               else begin
                  next_state <= TEST;
                  count <= count + 1;
               end
         end


         DONE: begin
                  next_state <= IDLE; // Wait for right shift value to be available. Th
         end


      endcase

   end

end

// Generate done=1 when FSM reaches DONE state
assign done = (next_state == DONE) ? 1 : 0;

// Generate Product in DONE state by loading shift_reg value to it
assign product = (next_state == DONE) ? {shift_reg[(2*N)], shift_reg[(2*N):1]} : 0;

endmodule: booth_multiplier
```

RTL netlist

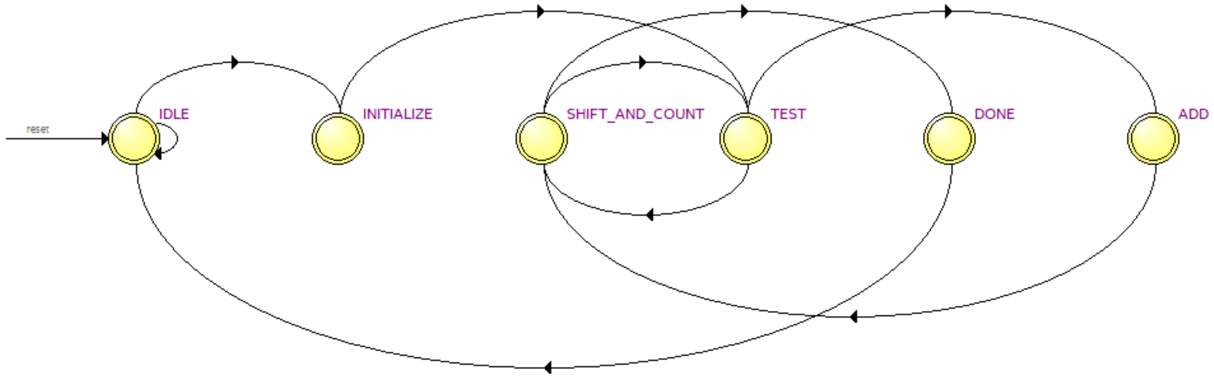


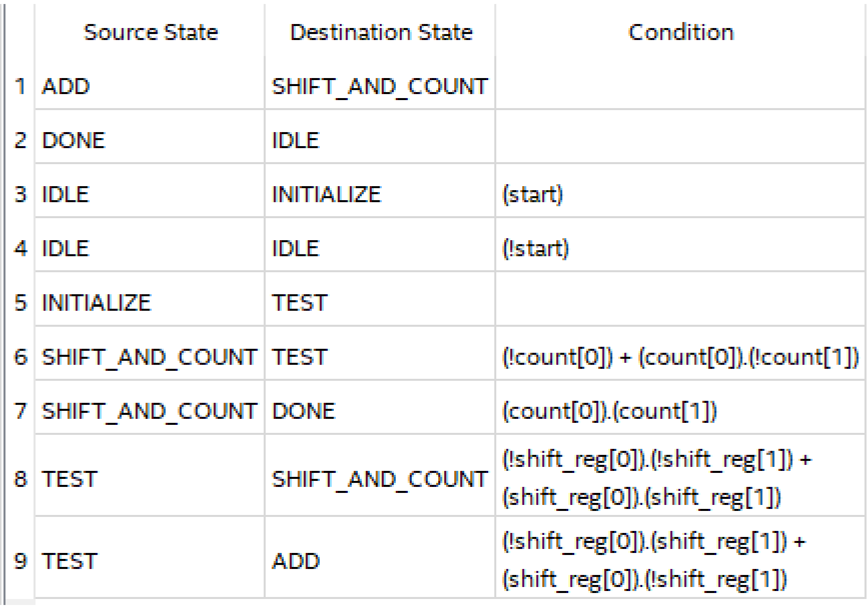
Resource usage



	Resource	Usage
1	Estimated ALUTs Used	43
1	-- Combinational ALUTs	43
2	-- Memory ALUTs	0
3	-- LUT_REGs	0
2	Dedicated logic registers	36
3		
4	Estimated ALUTs Unavailable	1
1	-- Due to unpartnered combinational logic	1
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	43
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	1
2	-- 6 input functions	3
3	-- 5 input functions	2
4	-- 4 input functions	13
5	-- <=3 input functions	24
8		
9	Combinational ALUTs by mode	
1	-- normal mode	42
2	-- extended LUT mode	1
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	49
12		
13	Total registers	36
1	-- Dedicated logic registers	36
2	-- I/O registers	0
3	-- LUT_REGs	0
14		
15		
16	I/O pins	20
17		
18	DSP block 18-bit elements	0
19		

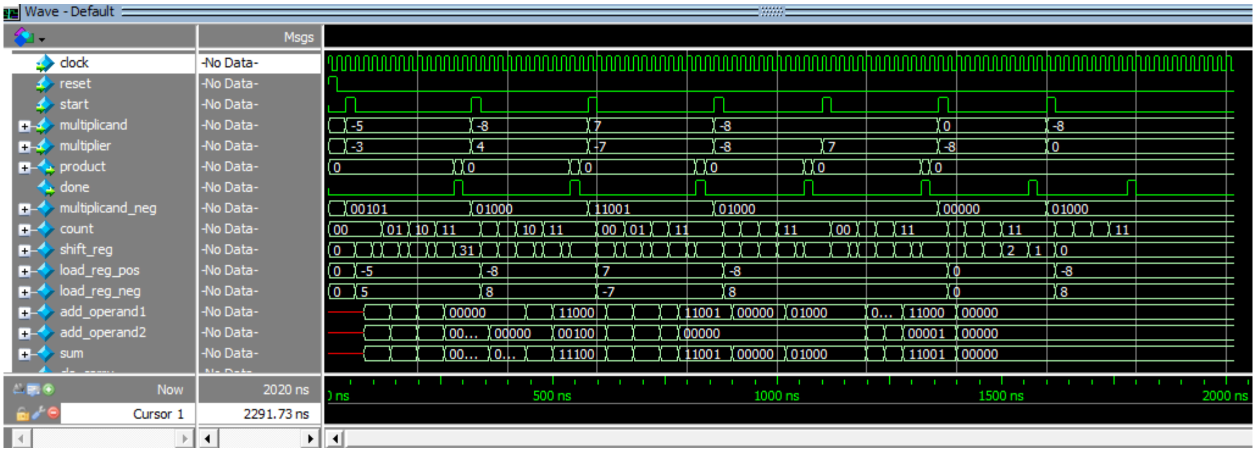
State diagram

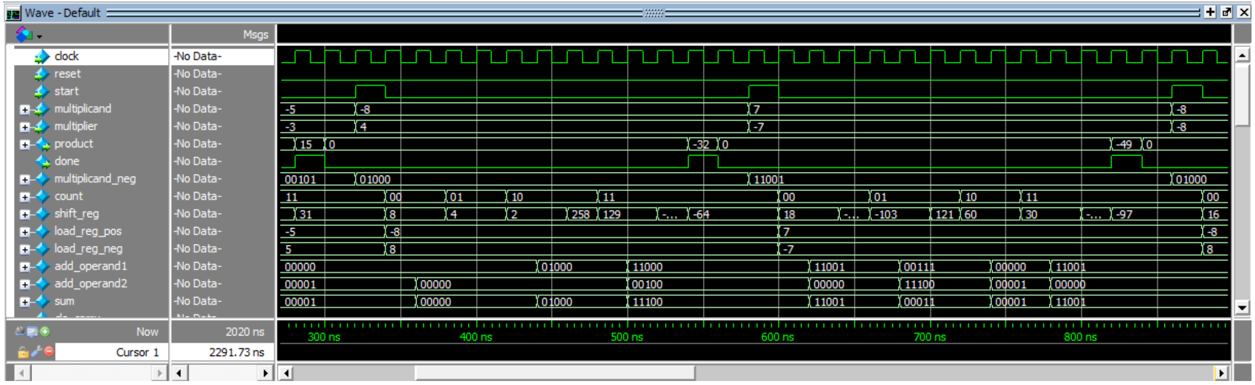


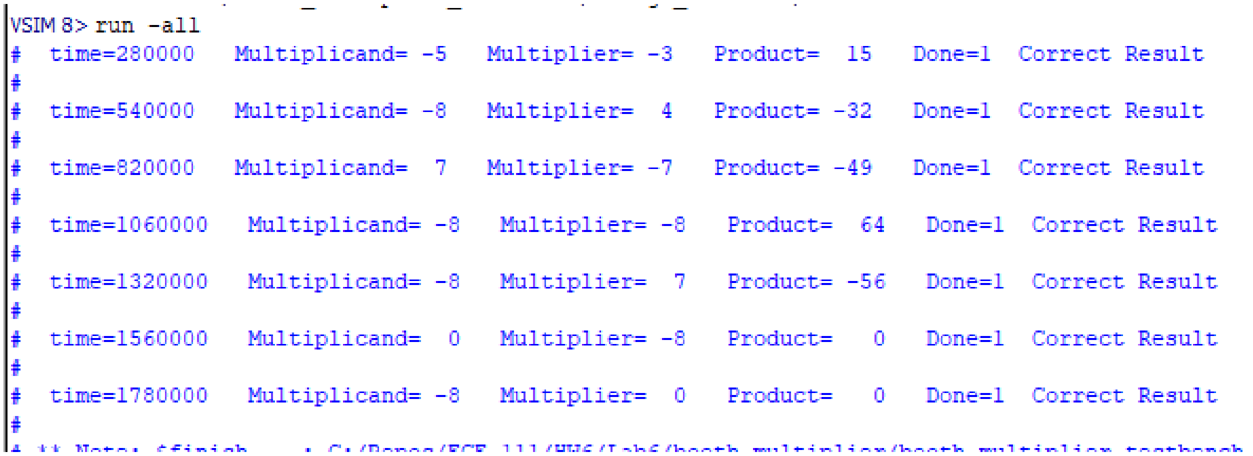


	Source State	Destination State	Condition
1	ADD	SHIFT_AND_COUNT	
2	DONE	IDLE	
3	IDLE	INITIALIZE	(start)
4	IDLE	IDLE	(!start)
5	INITIALIZE	TEST	
6	SHIFT_AND_COUNT	TEST	(!count[0]) + (count[0]).(!count[1])
7	SHIFT_AND_COUNT	DONE	(count[0]).(count[1])
8	TEST	SHIFT_AND_COUNT	(!shift_reg[0]).(!shift_reg[1]) + (shift_reg[0]).(shift_reg[1])
9	TEST	ADD	(!shift_reg[0]).(shift_reg[1]) + (shift_reg[0]).(!shift_reg[1])

Testbench simulation waveform







```
VSIM 8> run -all
#  time=280000   Multiplicand= -5   Multiplier= -3   Product=  15   Done=1  Correct Result
#
#  time=540000   Multiplicand= -8   Multiplier=  4   Product= -32   Done=1  Correct Result
#
#  time=820000   Multiplicand=  7   Multiplier= -7   Product= -49   Done=1  Correct Result
#
#  time=1060000   Multiplicand= -8   Multiplier= -8   Product=  64   Done=1  Correct Result
#
#  time=1320000   Multiplicand= -8   Multiplier=  7   Product= -56   Done=1  Correct Result
#
#  time=1560000   Multiplicand=  0   Multiplier= -8   Product=   0   Done=1  Correct Result
#
#  time=1780000   Multiplicand= -8   Multiplier=  0   Product=   0   Done=1  Correct Result
#
# ** Note: $finish    : C:/Repos/ECE 111/HW6/Lab6/booth_multiplier/booth_multiplier_testbench
```

- The waveforms shown are of 3 views: the overall waveform, the zoomed in waveform to show 3 cases and their outputs, and the transcript to demonstrate that each case yielded the correct result
- The multiplier behaves correctly because upon loading two numbers to be multiplied, and by setting start to high, this starts the internal cycle of states, and once the final product has been calculated, it is outputted with the done signal going high
- Then upon the next clock cycle, product resets to 0, and the multiplier idles until a new pair of numbers is loaded along with the pull up of the start input