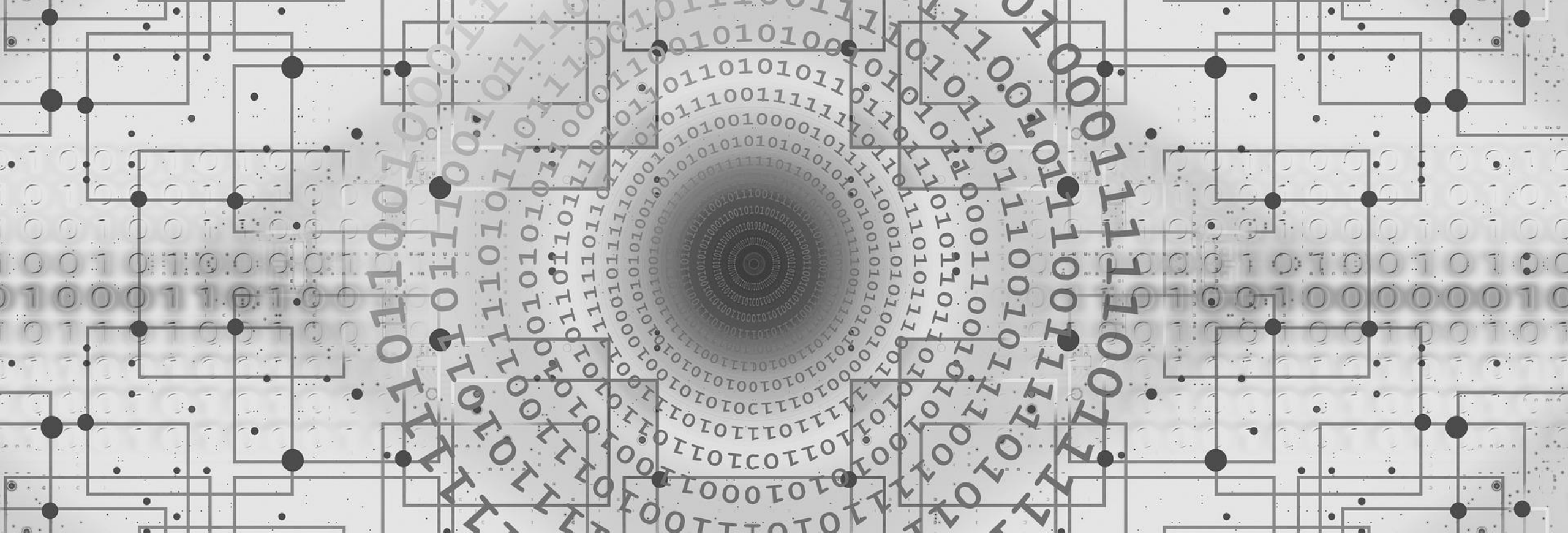


Integer Multiplier

ECE-111 Advanced Digital Design Project

Vishal Karna

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

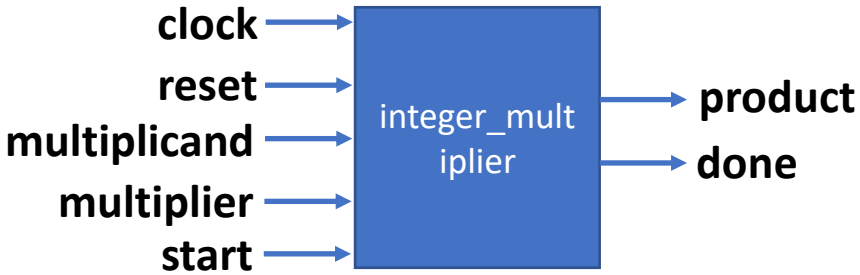
Homework-6b : Integer Multiplier

❑ Develop SystemVerilog RTL model for N-bit Integer Multiplier

- Develop Finite state machine and state transition diagram for SHIFT and ADD multiplier algorithm
- FSM coding style recommended : Single always block with non-blocking assignment statements within always block.
- Synthesize integer multiplier design for parameter N=4 and run simulation using testbench provided
- Review synthesis results (resource usage and RTL netlist/schematic)
- Review input and output signals in simulation waveform.
- Assume below mentioned primary port names and SystemVerilog RTL module **integer_multiplier.**
- **FSM code framework is provided in Lab folder with comments to help develop the code.**
- **Integer Multiplier testbench provided in Lab folder has built in checker to ensure design output is expected. See messages in Modelsim transcript window when performing simulation**

❑ Primary Ports for integer_multiplier module

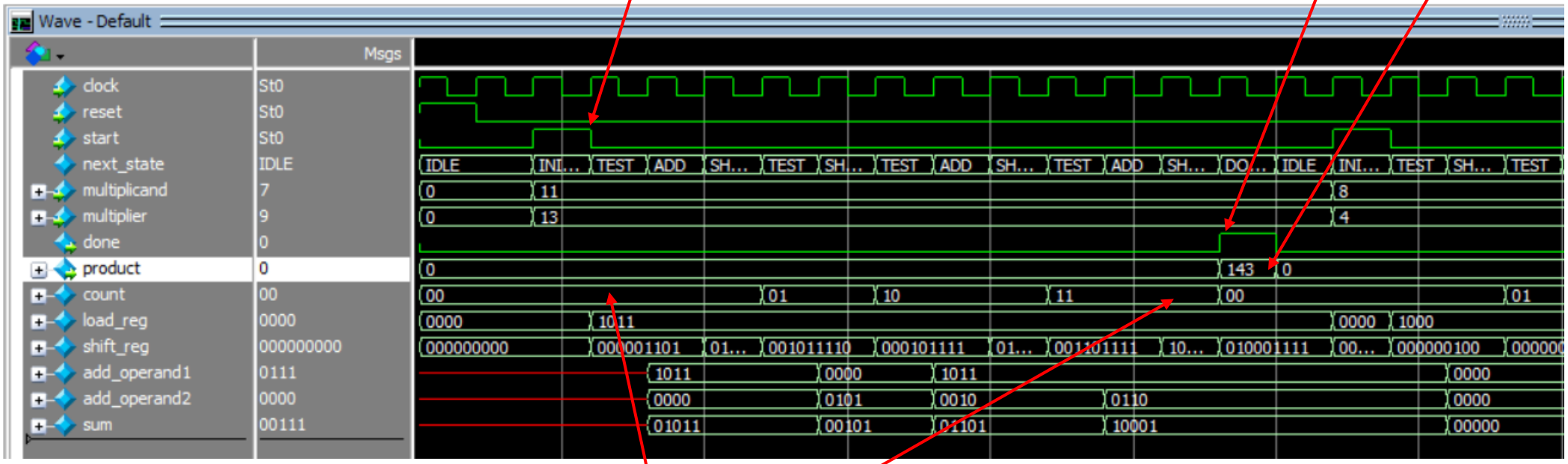
- **Input** clock, reset : asynchronous posedge reset
- **Input** start : 1 cycle pulse generated. Indicates to FSM to start multiplication operation
- **Input** logic[N-1:0] multiplicand, multiplier : Multiplicand and Multiplier inputs to Integer Multiplier
- **Output** logic[(2*N):0] product : Result of multiplication includes carry bit as MSB bit
- **Output** logic done : inidicates that product is available. This is one cycle pulse generated by FSM



Homework-6b : Integer Multiplier

- **Report should include :**
  - SystemVerilog FSM design code, State transition diagram (hand drawn snapshot or Snapshot from Quartus either way is fine) and Quartus auto-generated State transition table.
  - Synthesis resource usage and schematic generated from RTL netlist viewer
  - Simulation snapshot, simulation results transcript snapshot and explain simulation result to confirm RTL model developed works as a integer multiplier

- **Reference Output Snapshot**



SHIFT And ADD Method to Perform Integer Multiplier

- Multiplier : 4'b1101 (13), Multiplicand : 4'b1011 (11), Expected Product = 11 x 13 = 143 (9'b0_1000_0111)
- N-bit Multiplier has N stages of SHIFT and ADD round of computation
- If Multiplier LSB[0] = 1 Perform ADD of Accumulator + Multiplicand and store back to Accumulator and then perform Right Shift by 1
- If Multiplier LSB[0] = 0 Perform Right Shift by 1

acc + multiplicand
0 0 0 0 + 1011 = 01011

acc + multiplicand
0 0 1 0 + 1011 = 01101

acc + multiplicand
011 0 + 1011 =1000

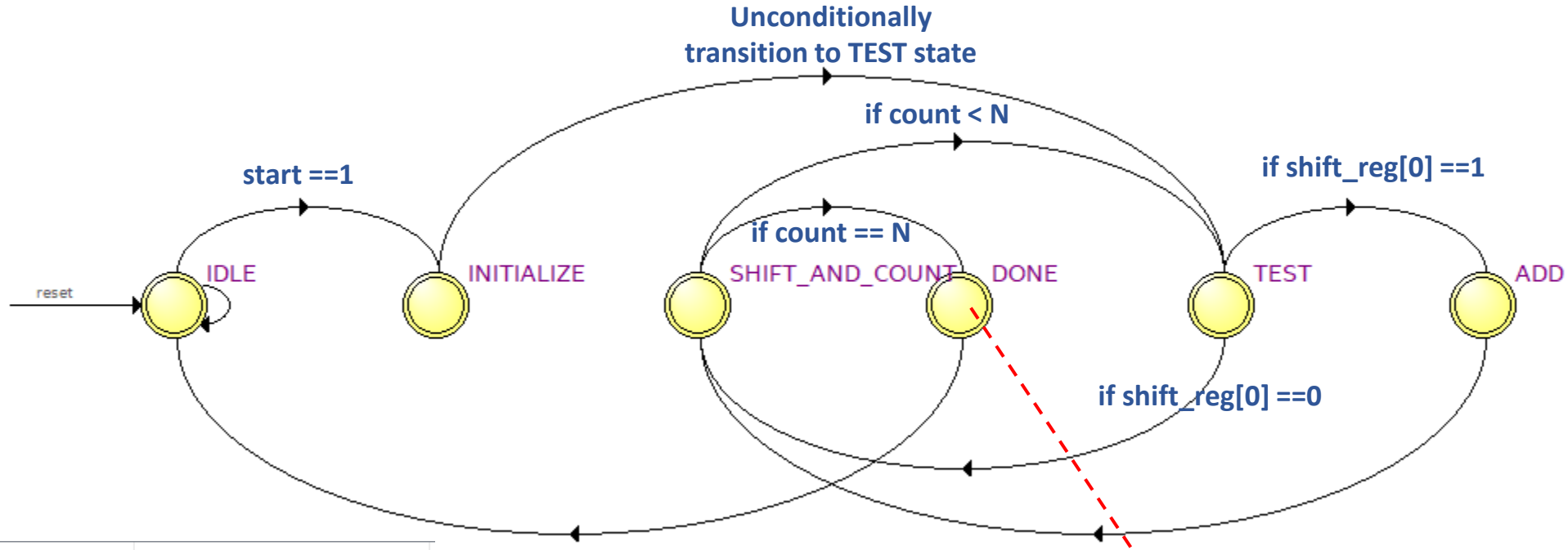
Stage-1	Carry	Accumulator	Multiplier	Operation Performed
0	0	0 0 0 0	1 1 0 1	Initialize
1	0	1 0 1 1	1 1 0 1	ADD
	0	0 1 0 1	1 1 1 0	SHIFT >> 1
2	0	0 0 1 0	1 1 1 1	SHIFT
3	0	1 1 0 1	1 1 1 1	ADD
	0	0 1 1 0	1 1 1 1	SHIFT >> 1
4	1	0 0 0 0	1 1 1 1	ADD
	0	1 0 0 0	0 1 1 1	SHIFT >> 1

SHIFT And ADD Integer Multiplier Algorithm Summary

- **Multiplication Process and FSM States :**
  - **IDLE:** Wait in this state until Start=1. Then move to INITIALIZE state if input signal Start==1
  - **INITIALIZE:** Multiplicand and multiplier are loaded into a load register and a shift register, respectively;
  - **TEST:** The LSB in the shift register which contains the multiplier is tested to decide the next state. If shift register LSB is '1', then next state is to **ADD** otherwise next sate is to **SHIFT_AND_COUNT**
  - **ADD:** the Adder adds previous stage add result with Multiplicand and stores the product to the accumulation result, back to shift register and then the state machine transits to **SHIFT_AND_COUNT** state
  - **SHIFT_AND_COUNT:** If shift register content is right shifted by 1 bit position. MSB of shift register '0' is entered
  - When the counter reaches to N, then next state is **DONE** stage otherwise next state is **TEST** state for next stage ADD/SHIFT operation
  - **DONE:** Done signal is asserted to '1' and shift register content is sent to 'product' output signal

Homework Assignment : Integer Multiplier

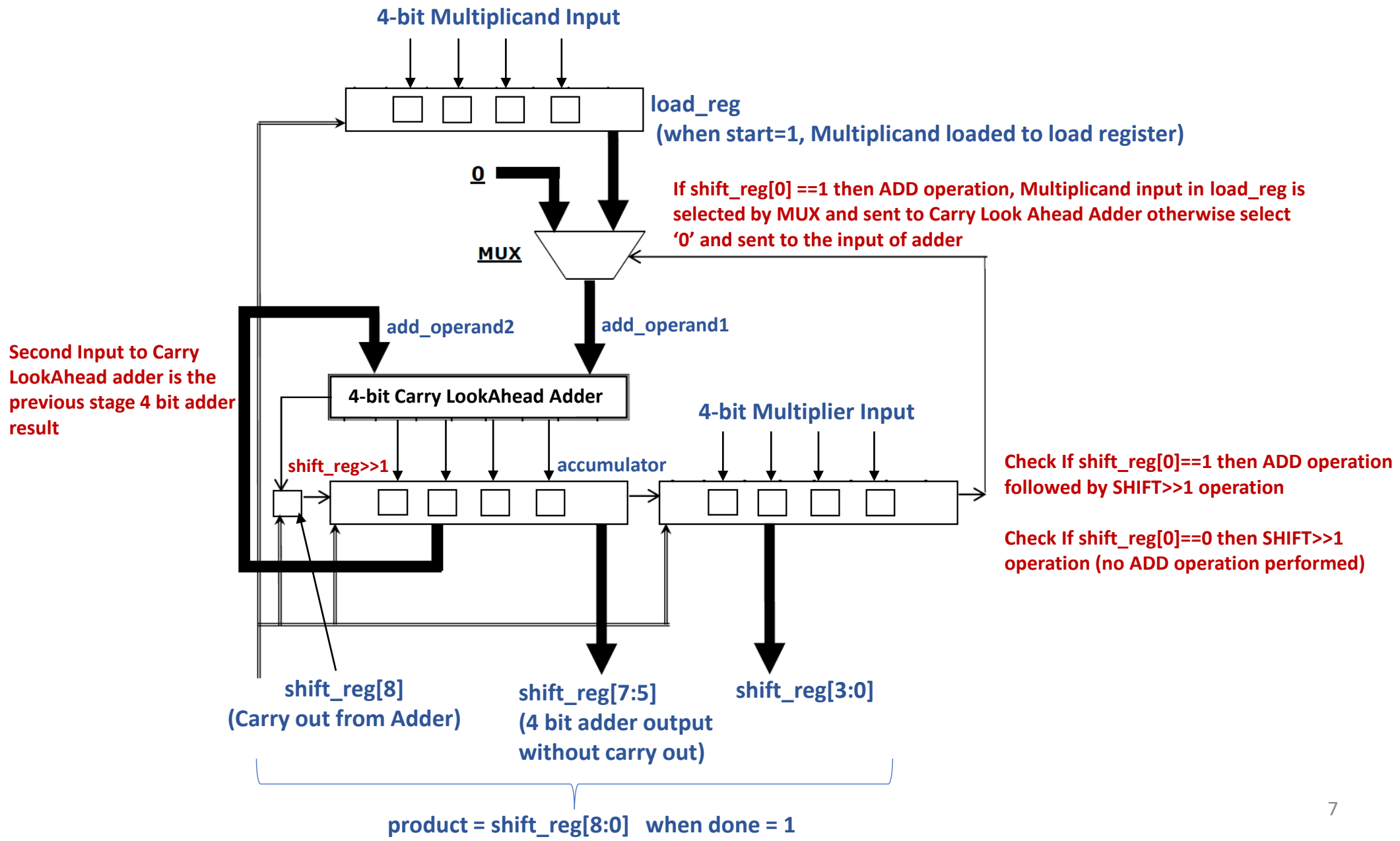
❑ Quartus Generated State Machine Diagram and State Table for reference purpose



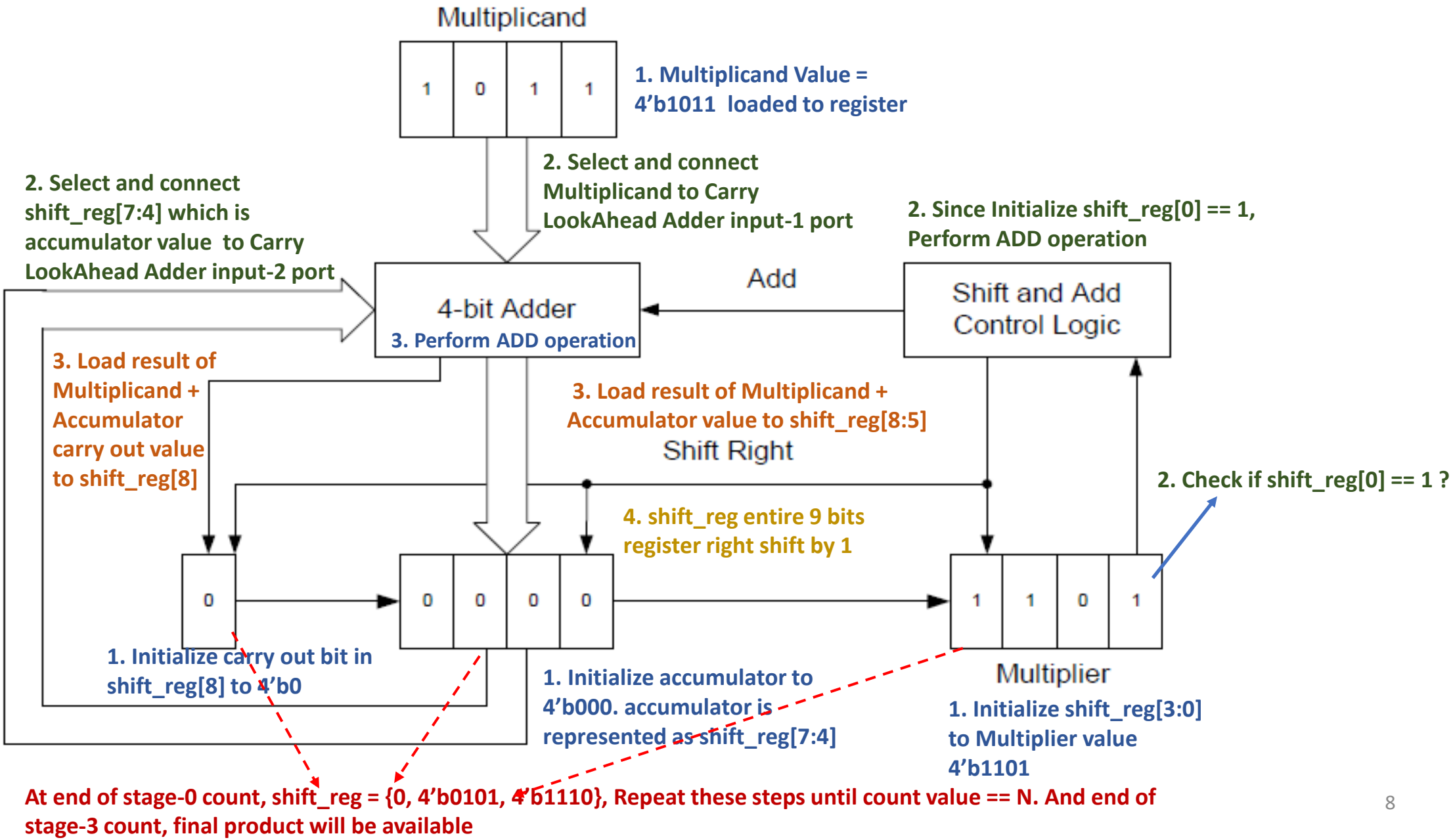
	Source State	Destination State	Condition
1	ADD	SHIFT_AND_COUNT	
2	DONE	IDLE	
3	IDLE	IDLE	(!start)
4	IDLE	INITIALIZE	(start)
5	INITIALIZE	TEST	
6	SHIFT_AND_COUNT	TEST	(!count[0]) + (count[0]).(!count[1])
7	SHIFT_AND_COUNT	DONE	(count[0]).(count[1])
8	TEST	SHIFT_AND_COUNT	(!shift_reg[0])
9	TEST	ADD	(shift_reg[0])

Note : When FSM transition to DONE state, load “shift_reg” value to “product”
And set done = 1 from separate logic using assign statement

Block Diagram of Integer Multiplier Using SHIFT and ADD



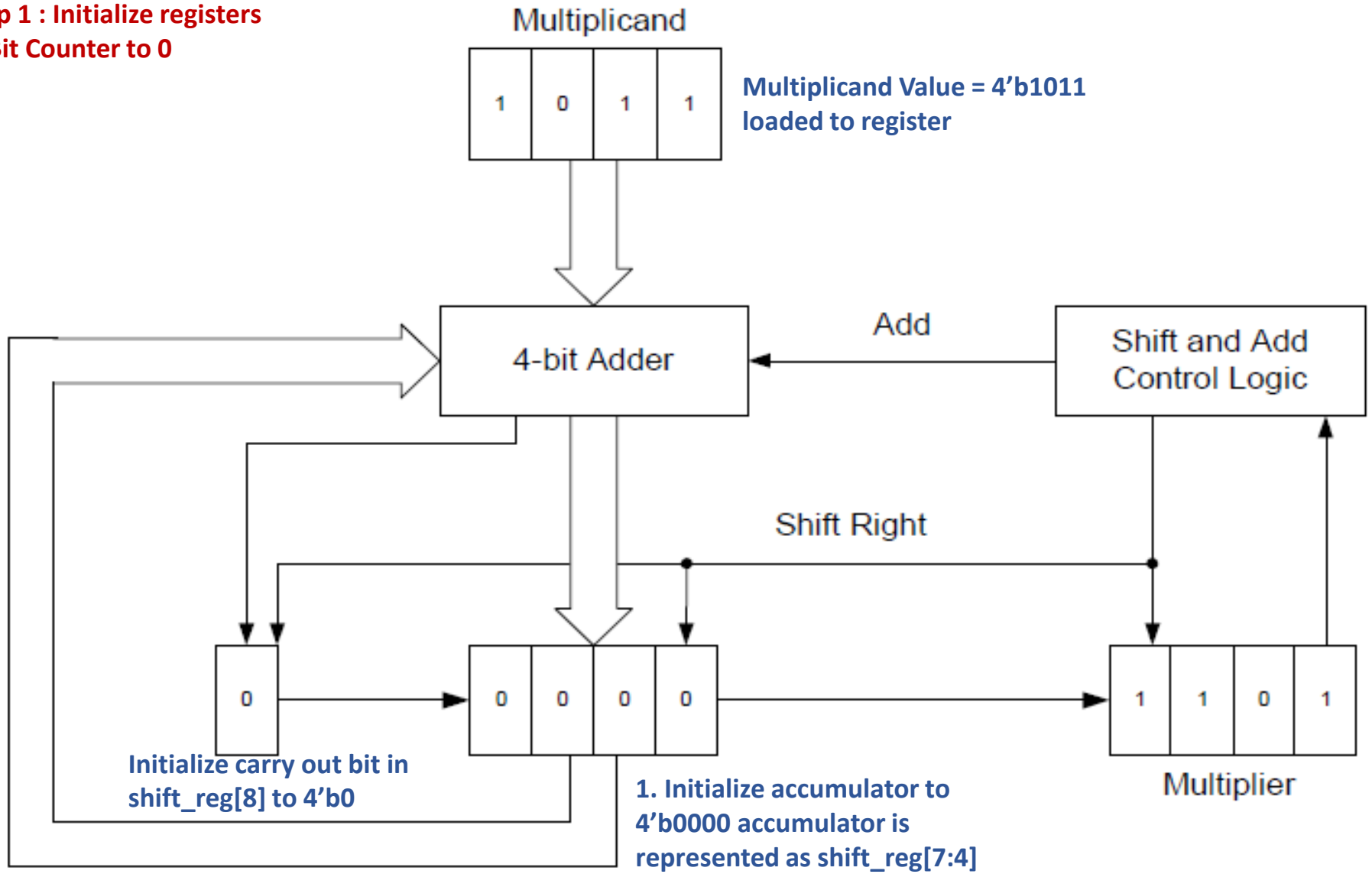
SHIFT And ADD Integer Multiplier Algorithm Steps Summary



Let’s us do simulation of 4-bit Integer Multiplier using SHIFT and ADD Multiplier Algorithm

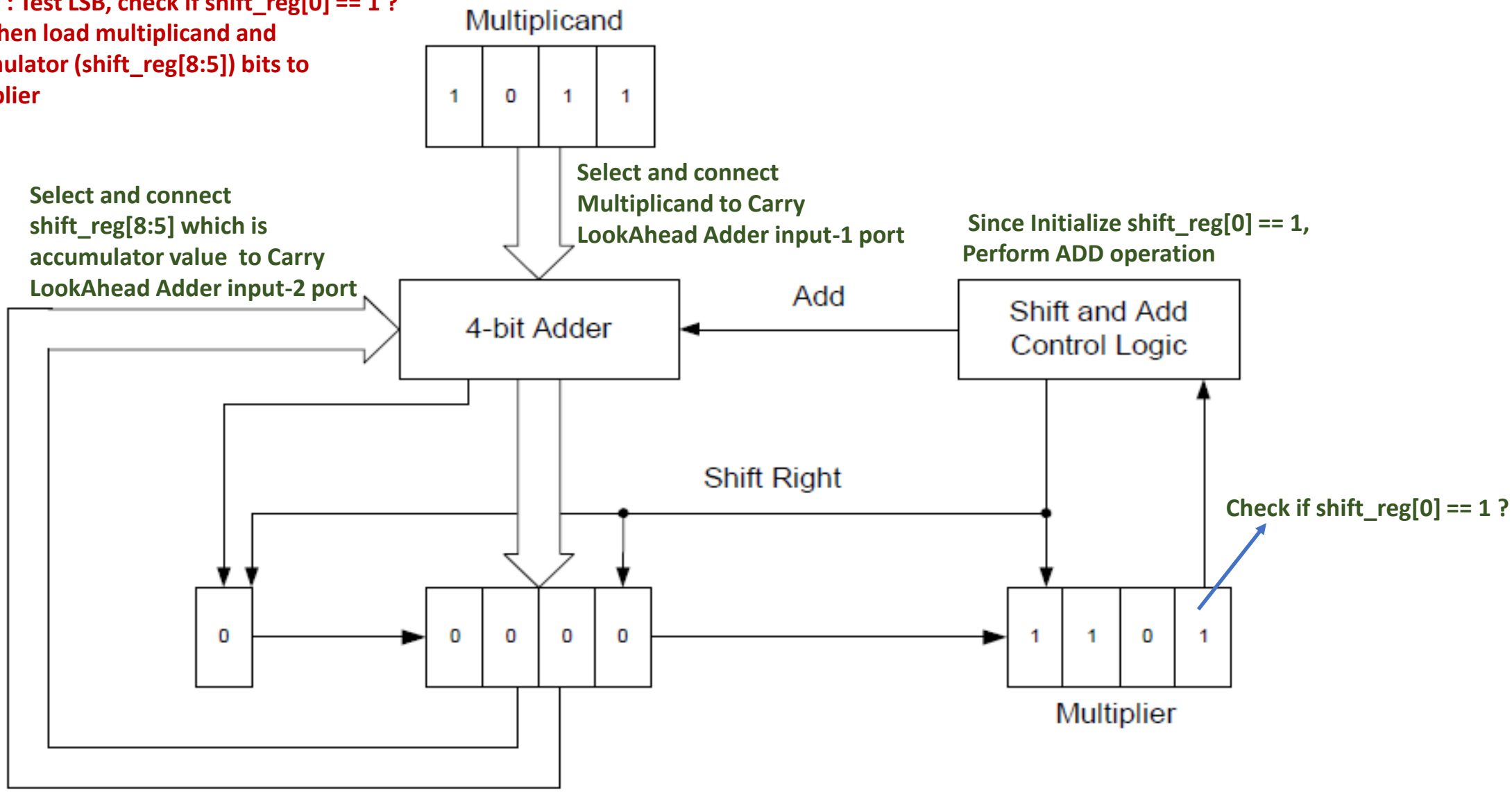
SHIFT And ADD Integer Multiplier Algorithm Example Steps

Step 1 : Initialize registers
N-Bit Counter to 0



SHIFT And ADD Integer Multiplier Algorithm Example Steps

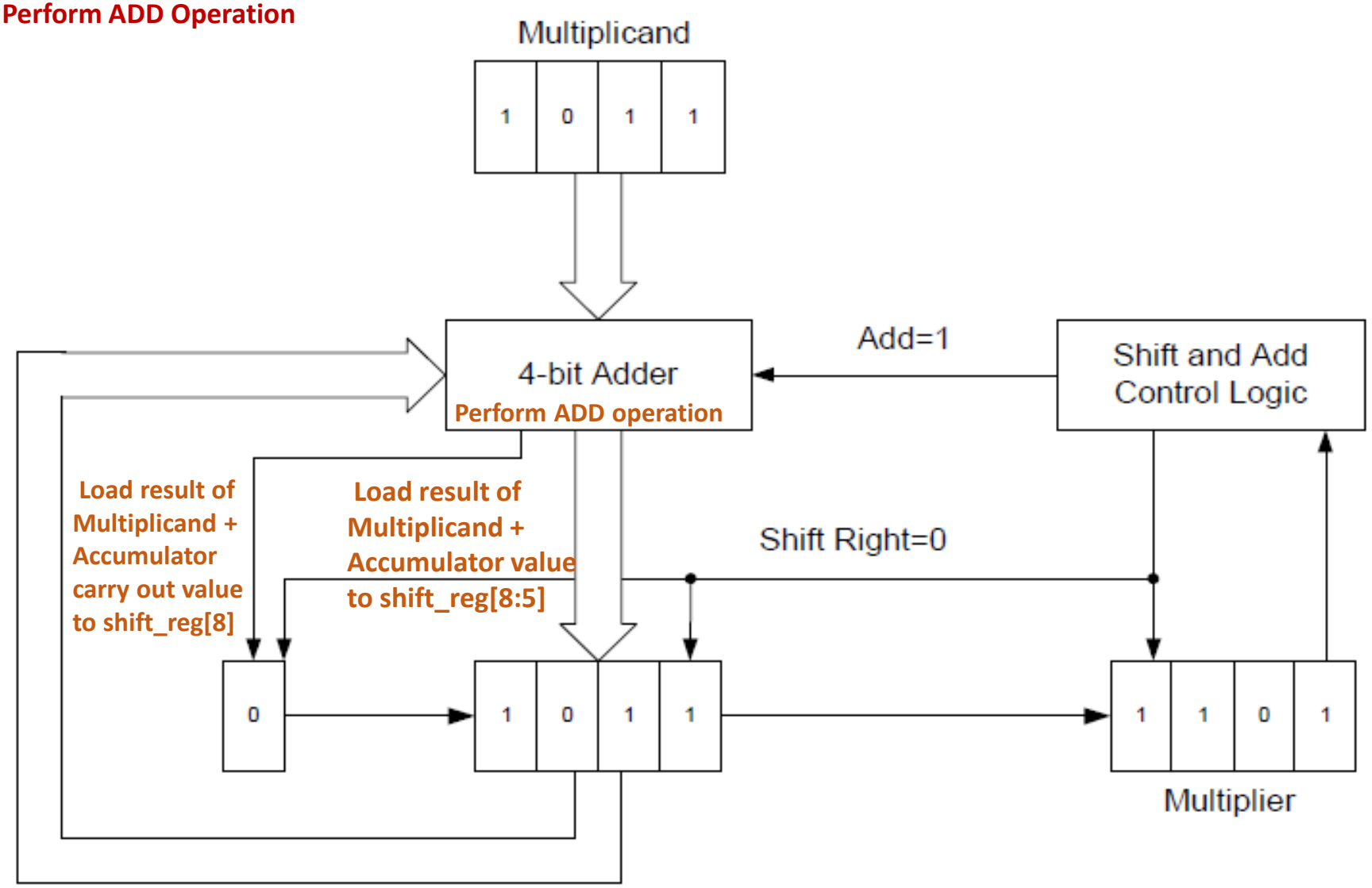
Step 2 : Test LSB, check if shift_reg[0] == 1 ?
If '1' then load multiplicand and accumulator (shift_reg[8:5]) bits to multiplier



N-bit counter value is still '0' at end of this step

SHIFT And ADD Integer Multiplier Algorithm Example Steps

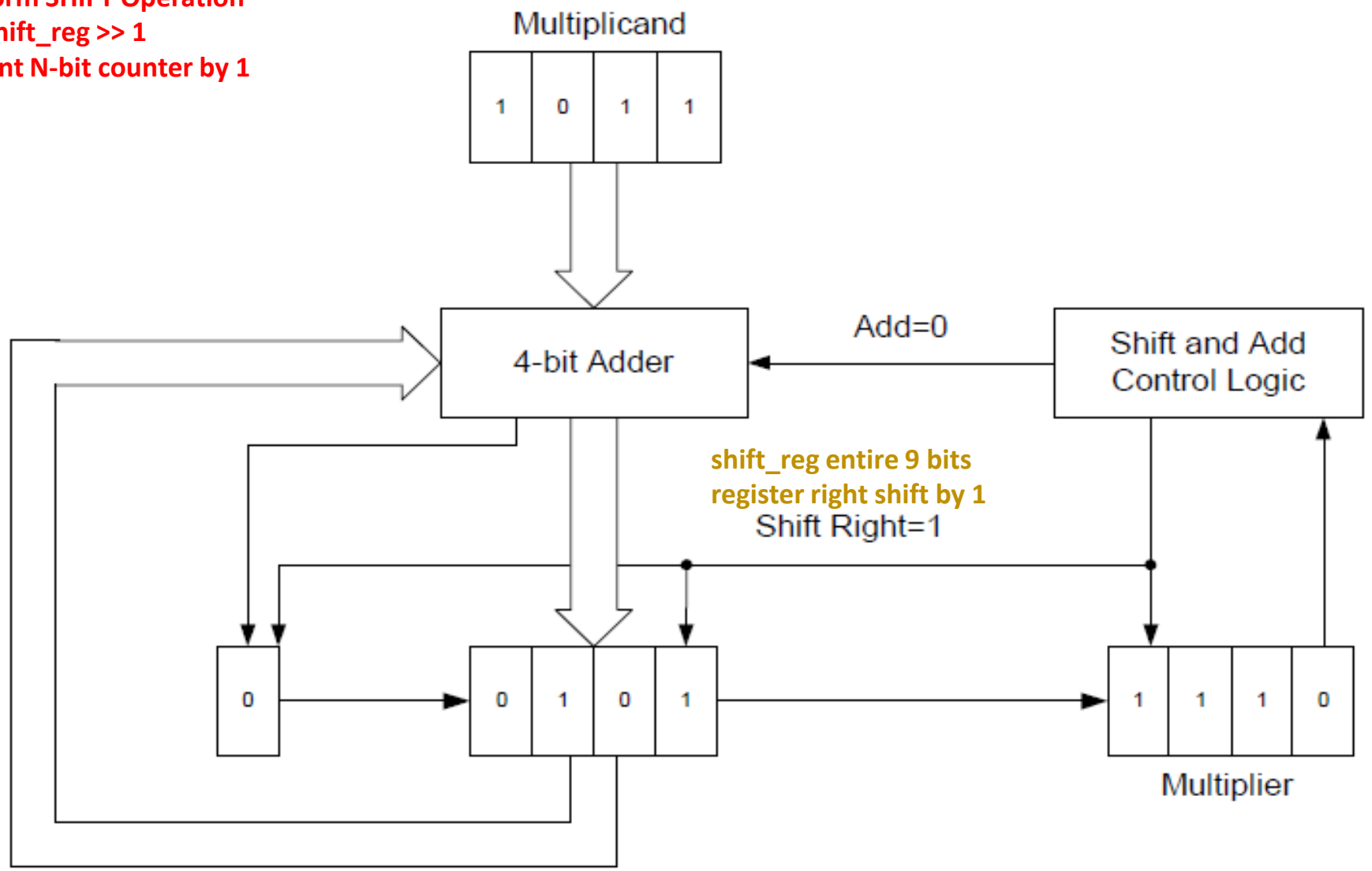
Step 3 : Perform ADD Operation



N-bit counter value is still '0' at end of this step

SHIFT And ADD Integer Multiplier Algorithm Example Steps

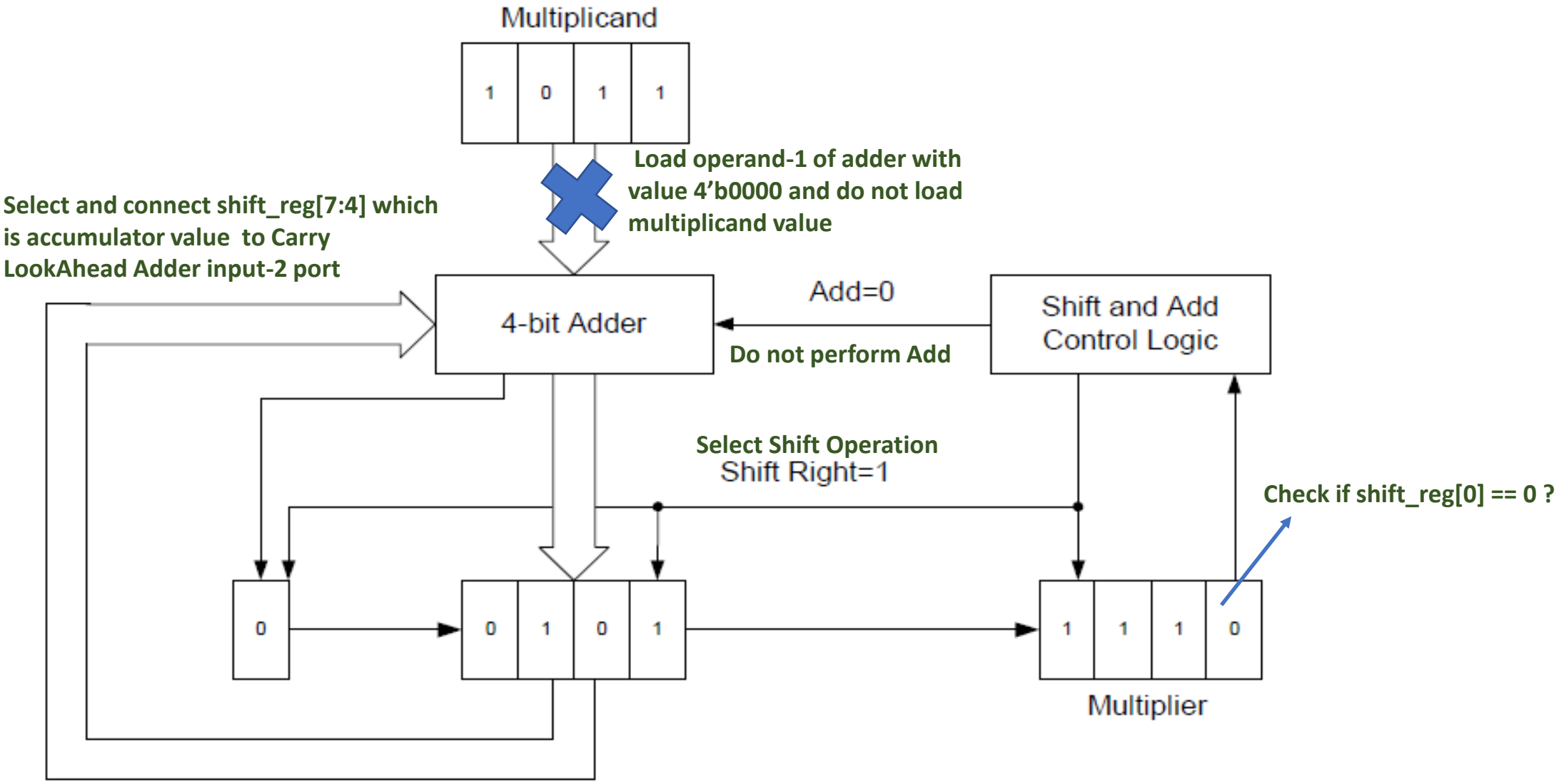
Step 4 : Perform SHIFT Operation
shifg_reg = shift_reg >> 1
And increment N-bit counter by 1



At end of N-bit count==0 stage, value of shift_reg = {0, 4'b0101, 4'b1110}

SHIFT And ADD Integer Multiplier Algorithm Example Steps

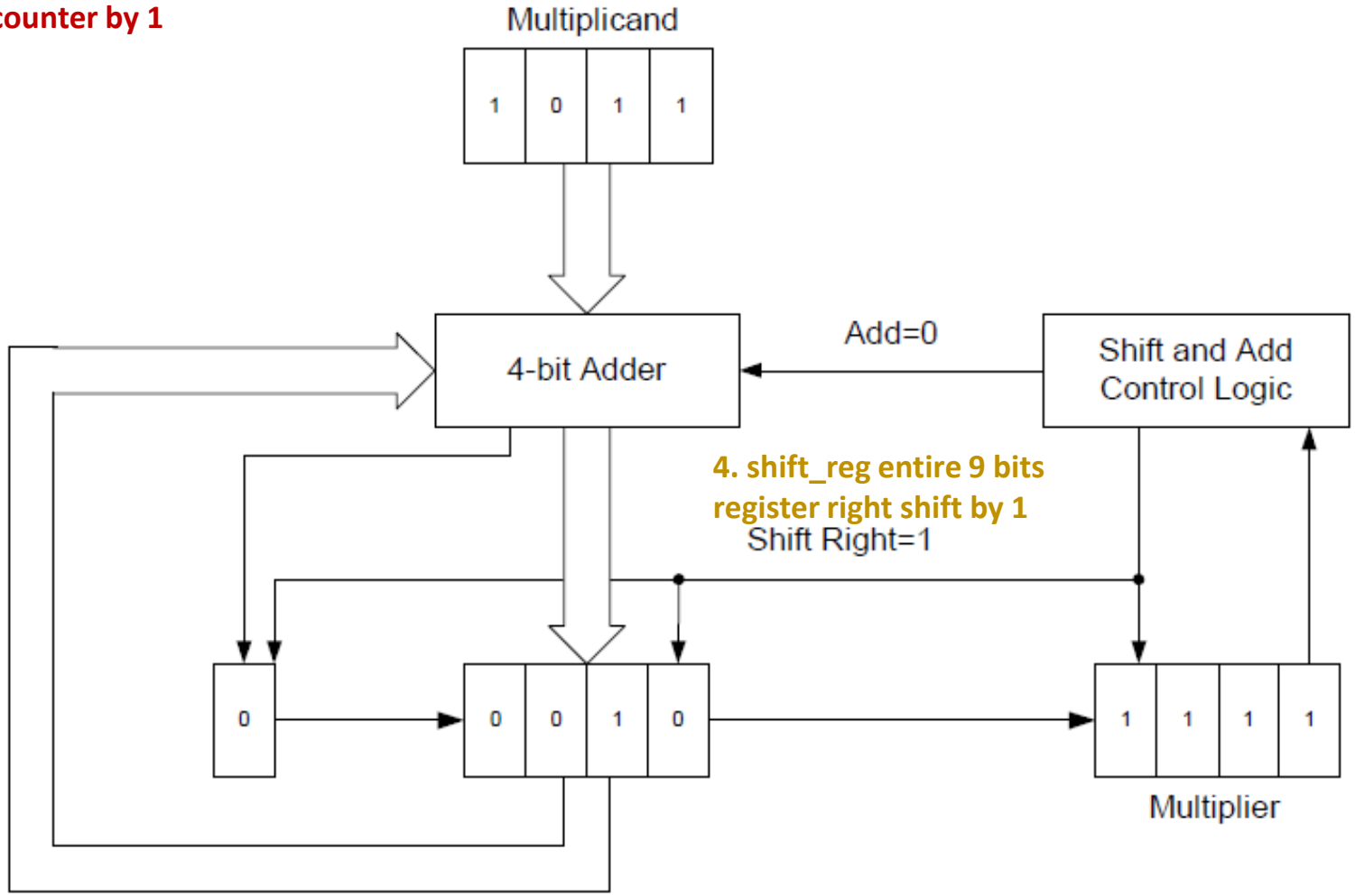
Step 5 : Check if shift_reg[0] == 0 or 1 ?



N-bit Counter value is '1'

SHIFT And ADD Integer Multiplier Algorithm Example Steps

Step 6 : Perform SHIFT Operation
shifg_reg = shift_reg >> 1
And increment N-bit counter by 1



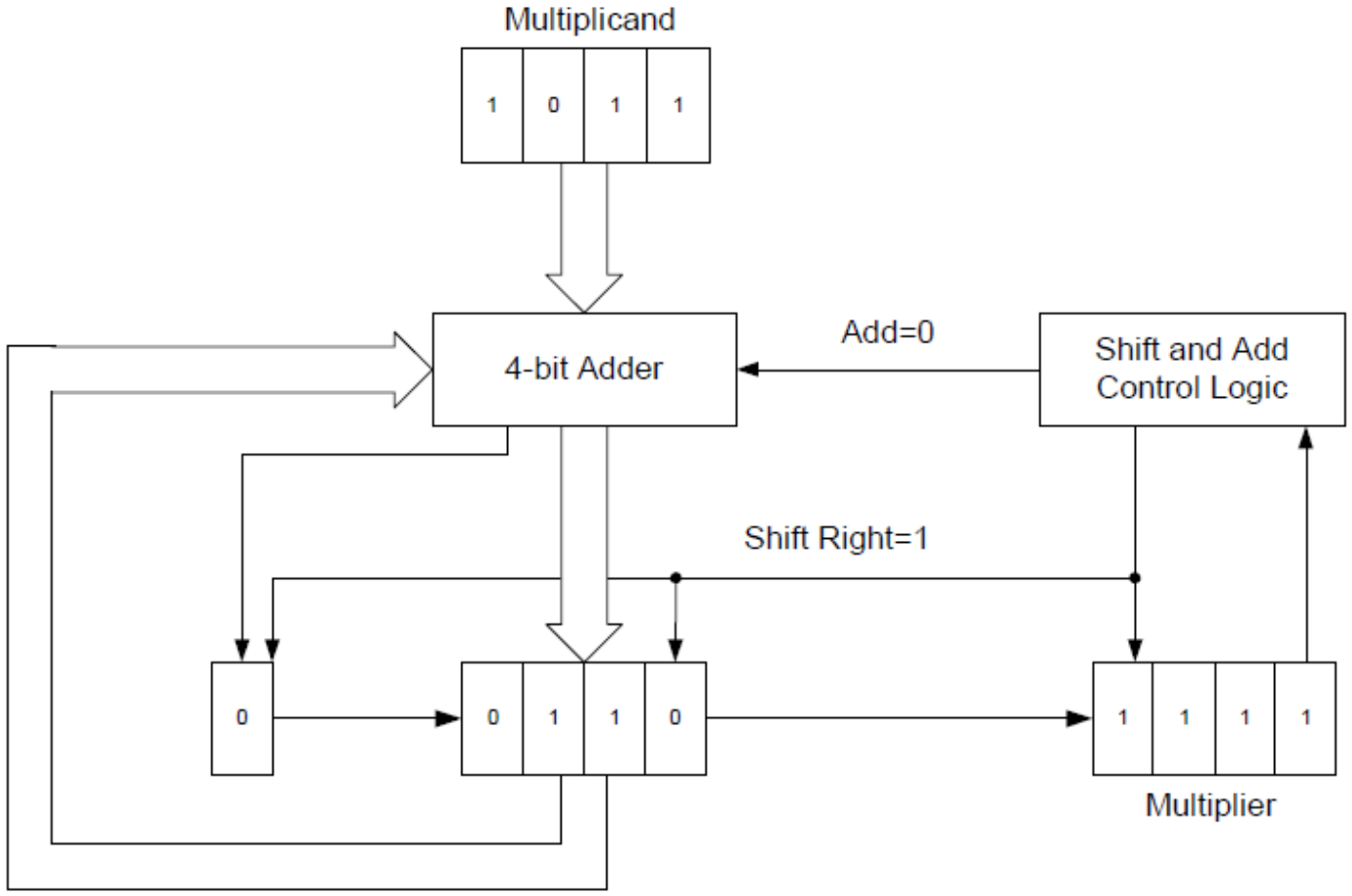
At end of N-bit count==1 stage, value of shift_reg = {0, 4'b0010, 4'b1111}

SHIFT And ADD Integer Multiplier Algorithm Example Steps

Step 7 : Check shift_reg[0] == 1 or 0 ?

Step 8 : since since shift_reg[0] == 1 then Perform Add Operation

Step 9 : Perform Shift operation shift_reg >> 1 and then increment N-bit counter value



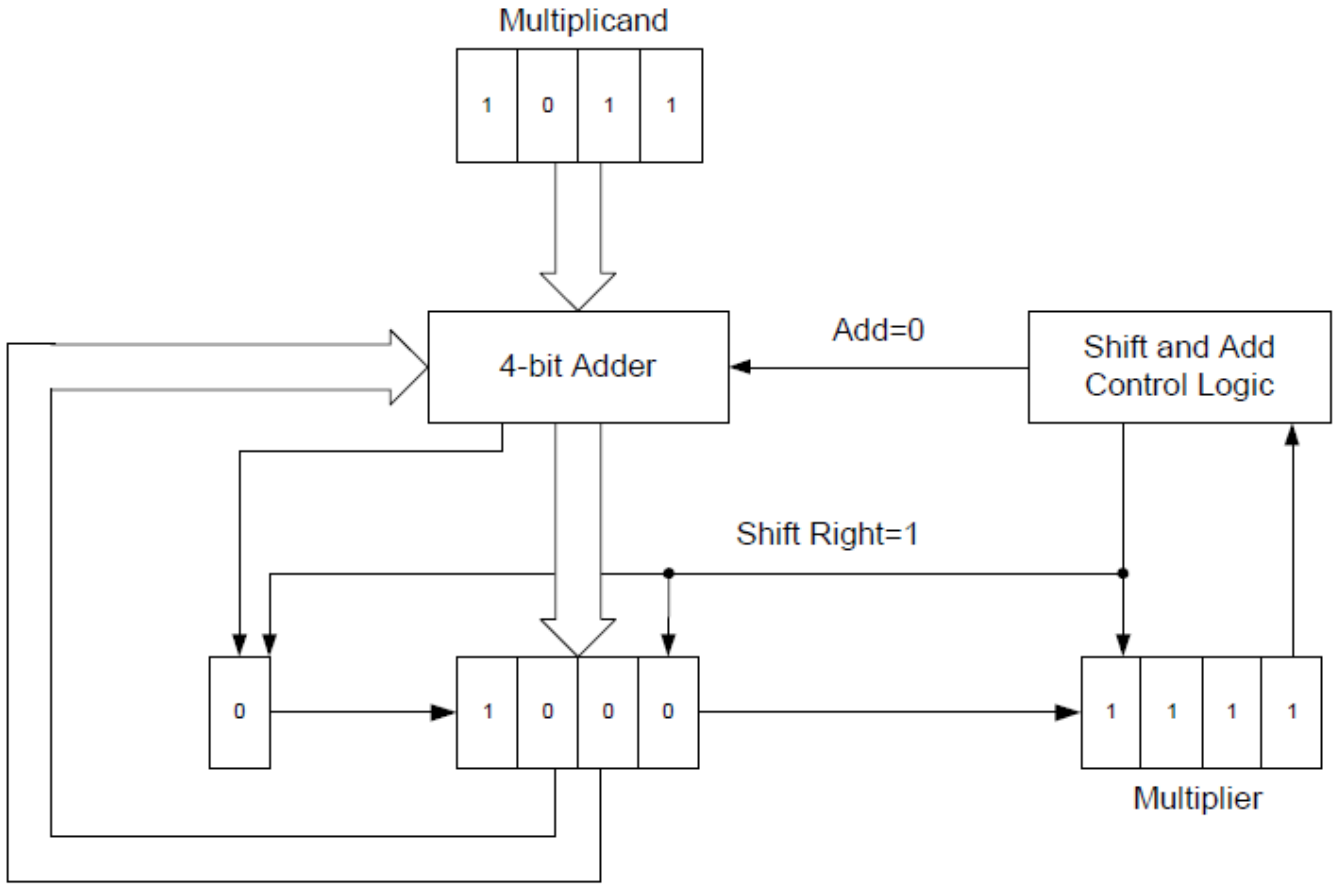
At end of N-bit count==2 stage, value of shift_reg = {0, 4'b0110, 4'b1111}

SHIFT And ADD Integer Multiplier Algorithm Example Steps

Step 10 : Check shift_reg[0] == 1 or 0 ?

Step 11 : since since shift_reg[0] == 1 then Perform Add Operation

Step 12 : Perform Shift operation shift_reg >> 1 and then increment N-b counter value



At end of N-bit count==3 stage, value of shift_reg = {0, 4'b1000, 4'b1111}
Since count == 3, set done = 1 and product = shift_reg (9'b0_1000_1111) = 143

Integer Multiplier Code Development Hint

- **Define 6 FSM States**

```
// 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;
```

- **For N-bit Integer Divider, ensure size of shift register is = N + N + 1 bits**
  - N bits to store adder output without carry out in shift register **(this is also know as accumulator)**
  - N bits to store multiplier value
  - 1 bit to store carryout bit from adder
  - shift register format = {carryout, N-bit Adder output, N-bit Multiplier}

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

- **Use Carry Look Ahead Adder and Full Adder Module implementation from previous homework assignment**

```
// Instantiate N-bit carry lookahead adder
// Pass add_operand1, add_operand2 and sum
// Tie CIN to '0'
carry_lookahead_adder #(.N(N)) adder_inst(

// Student to add code here
// use add_operand1, add_operand2, sum logic (wires) to connect to carry lookahead adder inputs

);
```

Integer Multiplier Code Development Hint

- **Develop FSM code using single always block approach with non-blocking assignment statements within**

```
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)
       // Intialized count, load_reg, shift reg to 0
       // Wait for start signal. if start is '1' then move to INITIALIZE otherwise say in IDLE state
       IDLE: begin

       end
       // Load Multiplicand and Multiplier in a load register and a shift register
       // Initialize count to 0 and then set next_state to TEST
       INITIALIZE: begin


       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

           end
           else begin

           end
       end
```

Integer Multiplier Code Development Hint

- **FSM code framework…. Continued**

```
        // Perform ADD operation
        ADD: begin
           // Load shift register : Output sum from Adder which includes carry and retain previous lower bit of shift register
           // move to shift and increment count state (SHIFT_AND_COUNT)

        end
        // Perform Right Shift by 1 on shift_reg and check if count == N
        SHIFT_AND_COUNT: begin
            // Right shift entire shift_reg by 1 position and store result in shift_reg
            // Increment count
            if(count == N-1) begin // If 'N' times SHIFT operation performed then move to Done state else go back to Test state

            end
            else begin

            end
        end
        // Staty in DONE state for final product value to be available and done = 1
        DONE: begin
            next_state <= IDLE; // Wait for right shift value to be available. THis is the final product value.
        end
    endcase
end
```

- **Using assign statement generate output ‘done’ and final ‘product’ value when FSM transitions to DONE state**

```
// 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;
```