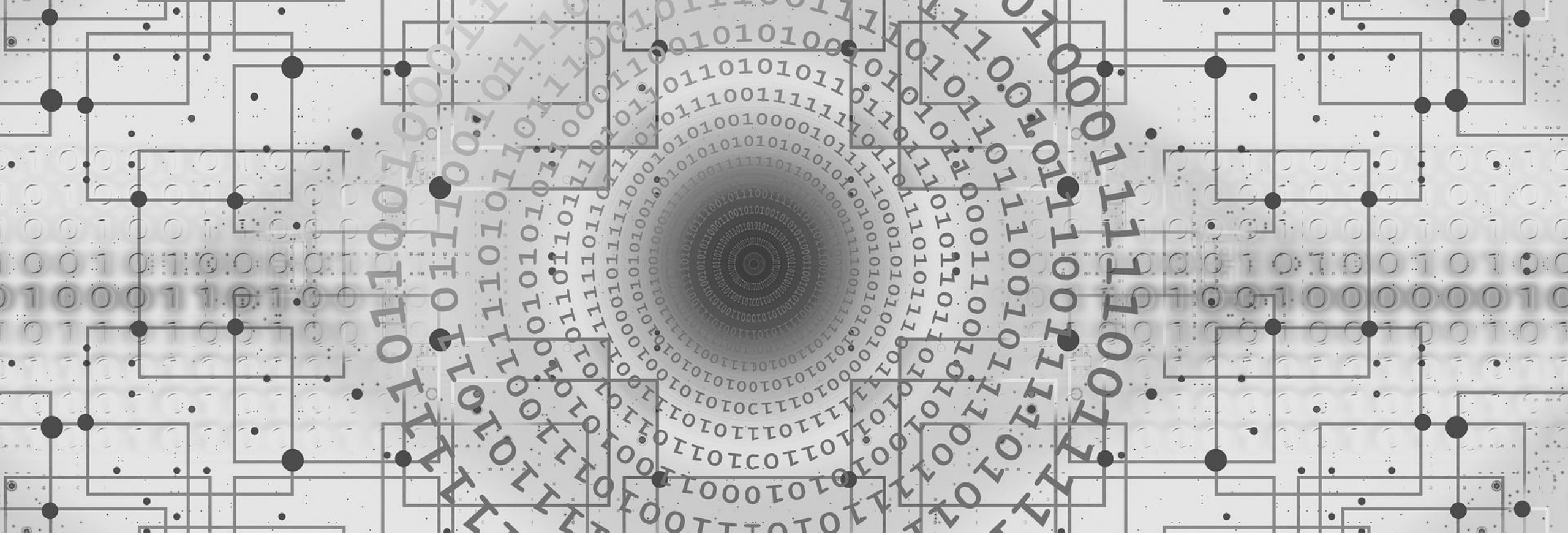


Booth Multiplier (Signed Integer Multiplication)

ECE-111 Advanced Digital Design Project

Brandon Saldanha

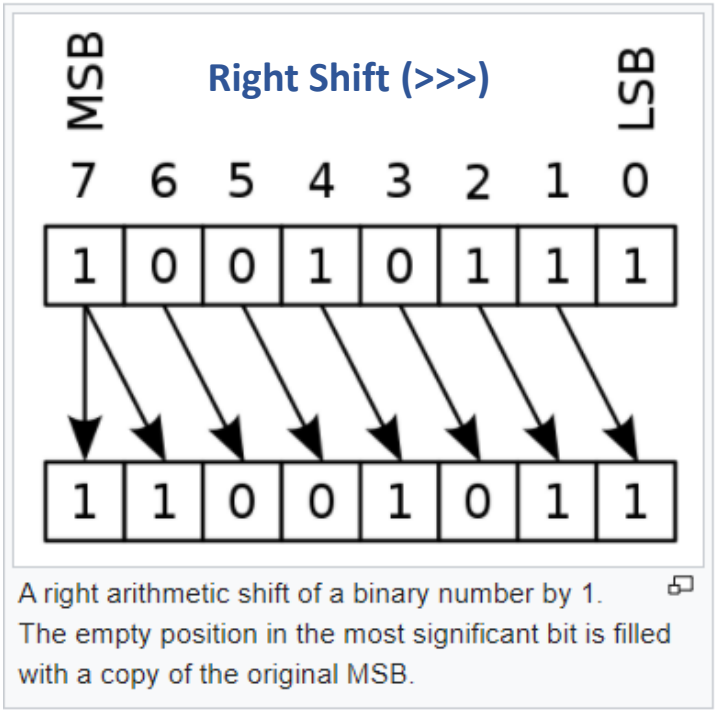
UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Booth Multiplication Algorithm

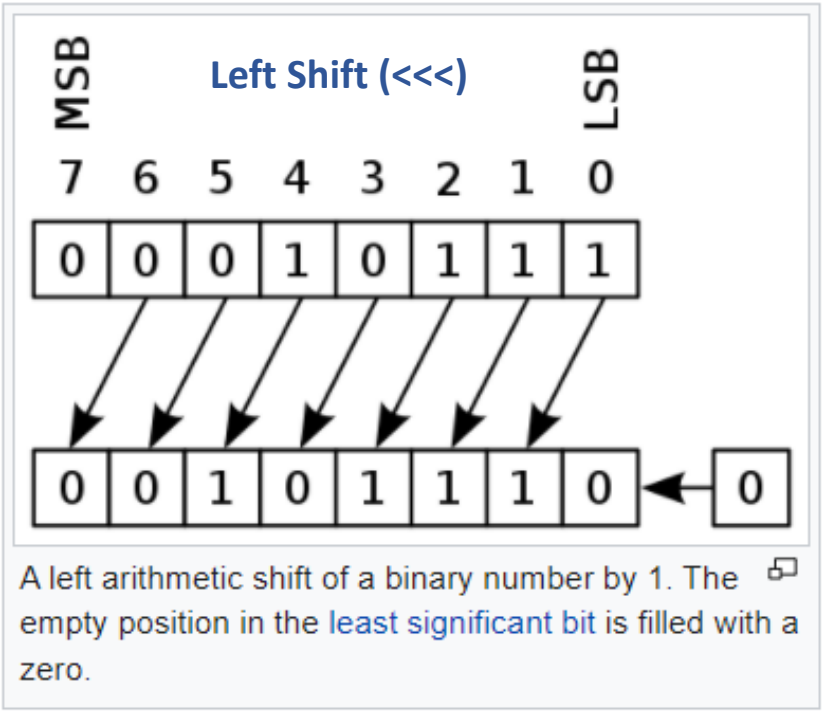
- **Same structure as the integer multiplier. Booth algorithm is extended to handle signed integer multiplication.**
- Let **m** and **r** be the multiplicand and multiplier, respectively; and let *x* represent the number of bits in **m** and **r**.
- **Determine the values of *A* and *S*, and the initial value of *P*. All of these numbers should have a length equal to (*x* + *y* + *2*).**
  - A: Fill the most significant (leftmost) bits with the value of **m**, extended with the sign. Fill the remaining (*x* + *1*) bits with zeros.
  - S: Fill the most significant bits with the value of (−**m**) in two's complement notation, extended with the sign. Fill the remaining (*y* + *1*) bits with zeros.
  - P: Fill the most significant *x* + *1* bits with zeros. To the right of this, append the value of **r**. Fill the least significant (rightmost) bit with a zero.
- **Determine the two least significant (rightmost) bits of *P*.**
  - If they are 01, find the value of *P* + *A*. Ignore any overflow.
  - If they are 10, find the value of *P* + *S*. Ignore any overflow.
  - If they are 00, do nothing. Use *P* directly in the next step.
  - If they are 11, do nothing. Use *P* directly in the next step.
- **Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let *P* now equal this new value.**
- **Repeat steps 2 and 3 until they have been done *x* times.**
- **Drop the least significant (rightmost) and most significant (leftmost) bit from *P*. This is the product of m and r.**

What is Arithmetic Shift Operation

- An **arithmetic shift** is a shift operator, sometimes termed a **signed shift** (though it is not restricted to signed operands).
- The two basic types are the **arithmetic left shift (<<<)** and the **arithmetic right shift (>>>)**
  - For binary numbers is a bitwise operation that shifts all of the bits of its operand
  - Every bit in the operand is simply moved a given number of bit positions, and the vacant bit-positions are filled in
  - Instead of being filled with all 0s, as in logical shift, when shifting to the right, the leftmost bit (usually the sign bit in signed integer representations) is replicated to fill in all the vacant positions (this is a kind of sign extensions)



A right arithmetic shift of a binary number by 1. The empty position in the most significant bit is filled with a copy of the original MSB.



A left arithmetic shift of a binary number by 1. The empty position in the least significant bit is filled with a zero.

Courtesy Wikipedia : https://en.wikipedia.org/wiki/Arithmetic_shift

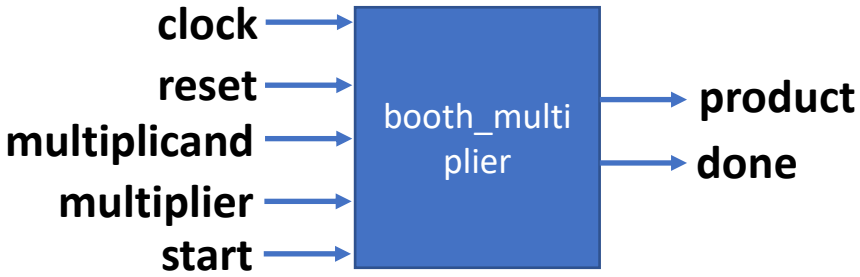
Homework-6c : Booth Multiplier

❑ Develop SystemVerilog RTL model for N-bit Booth Multiplier

- Develop Finite state machine and state transition diagram for Booth multiplier algorithm
- FSM coding style recommended : Single always block with non-blocking assignment statements within always block.
- Synthesize booth 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 **booth_multiplier.**
- **Note : FSM code framework is provided in Lab folder with comments to help develop the code.**
- **Testbench provided has built in checker to ensure design output is expected. See messages in Modelsim transcript window when performing simulation**

❑ Primary Ports for booth_multiplier module

- **Input** clock, reset : posedge clock and 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)-1:0] product : Result of multiplication includes carry bit as MSB bit
- **Output** logic done : Indicates that product is available. This is one cycle pulse generated by FSM.



Homework-6c : Booth Multiplier

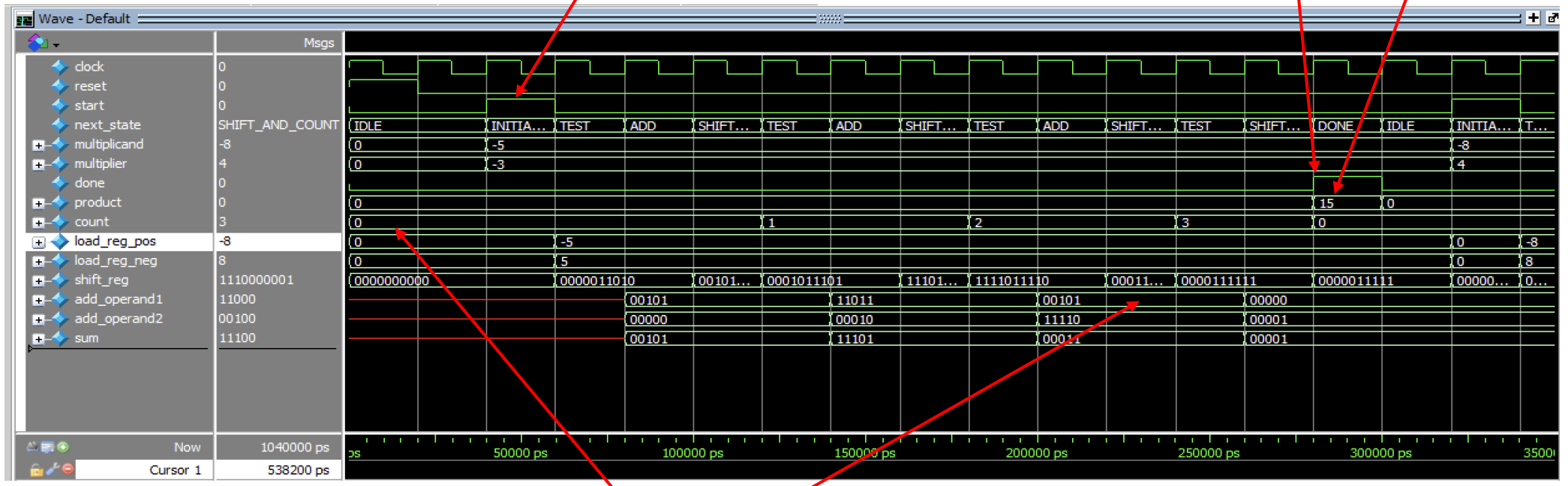
❑ Report should include :

- SystemVerilog FSM design code and State transition diagram (snapshot of hand drawn diagram or Quartus auto-generated diagram, either is acceptable)
- Synthesis resource usage and schematic generated from RTL netlist viewer
- Simulation snapshot and explain simulation result to confirm RTL model developed works as a integer multiplier

❑ Reference Output Snapshot

Once Start is '1' multiplicand value -3 and multipler value -5 and 5 are loaded in load_reg_pos and load_reg_neg respectively

Once 'done' signal is '1' for single clock cycle, 'product' value of 15 is available from multiplier FSM



Since N=4, ie. Inputs are 4 bit values, count value will increment from 0, 1, 2, 3 which indicates 4 stages of the Booth Algorithm

Booth Method to Perform Signed Integer Multiplier

- Multiplier : 4'b1101 (-3), Multiplicand : 4'b1011 (-5), Expected Product = -3 x -5 = 15 (8'b000_1111)
- N-bit Multiplier has N stages of SHIFT and ADD round of computation. Extra bit always initialized with 1'b0.
- If Multiplier LSB[1:0] = 01 Perform ADD of Accumulator + Multiplicand and store back to Accumulator and then perform Arithmetic Right Shift by 1
- If Multiplier LSB[1:0] = 10 Perform ADD of Accumulator + (– Multiplicand) and store back to Accumulator and then perform Arithmetic Right Shift by 1
- If Multiplier LSB[1:0] = 00 or 11 Perform Right Shift by 1

Shift Register

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

acc + (– multiplicand)
0_0000 + 0_0101 = 0_0101

acc + (multiplicand)
0_0101 + 1_1011 = 1_1101

acc + (– multiplicand)
1_1110+ 0_0101 = 0_0011

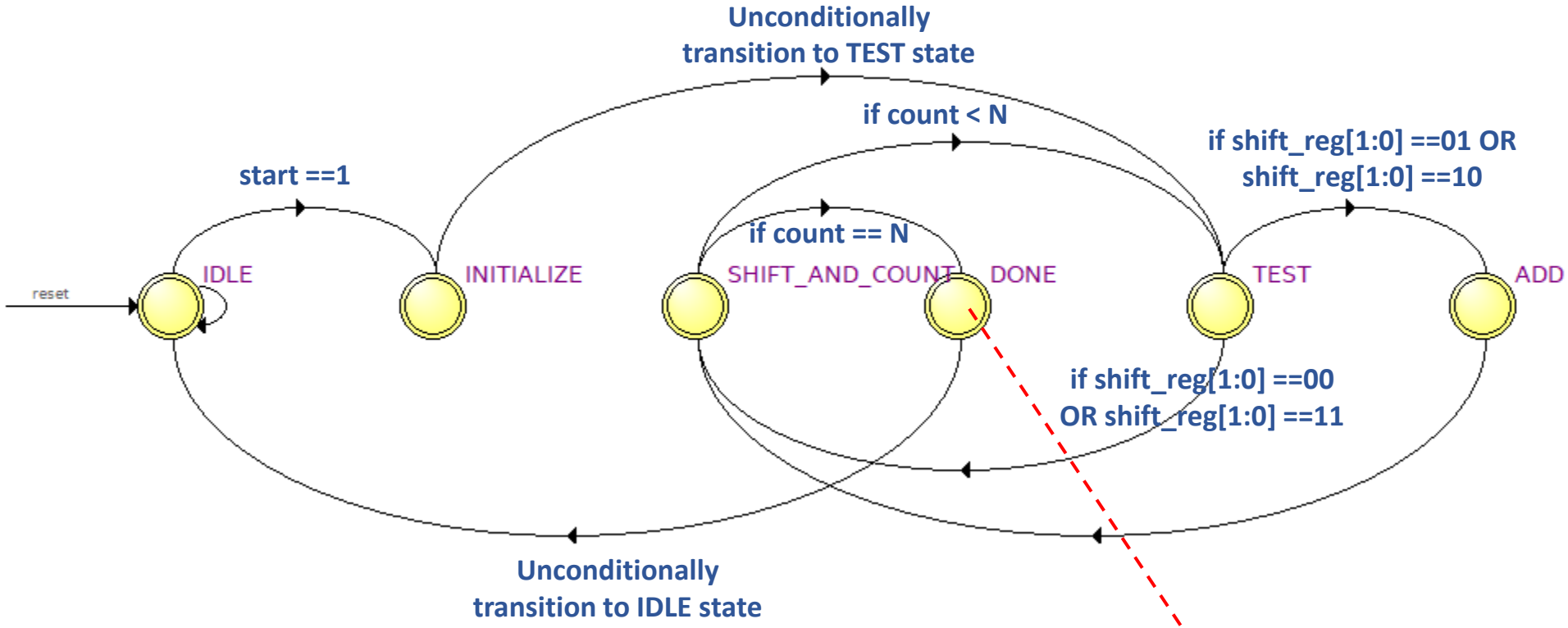
Product

Booth 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, –Multiplicand and Multiplier are loaded into a positive load register, negative 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[1:0] is '01' or '10', then next state is to **ADD** otherwise next sate is to **SHIFT_AND_COUNT**
  - **ADD:** If LSB[1:0] is '01', the adder adds previous stage add result with Multiplicand, if LSB[1:0] is '10', the Adder adds previous stage add result with –Multiplicand. The result is stored 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 is sign extended (Read about **>>>** and **<<<** operators in previous pages of this document)
  - 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 specific bits of shift register content is sent to 'product' output signal

Homework-6b : Signed Integer Multiplier

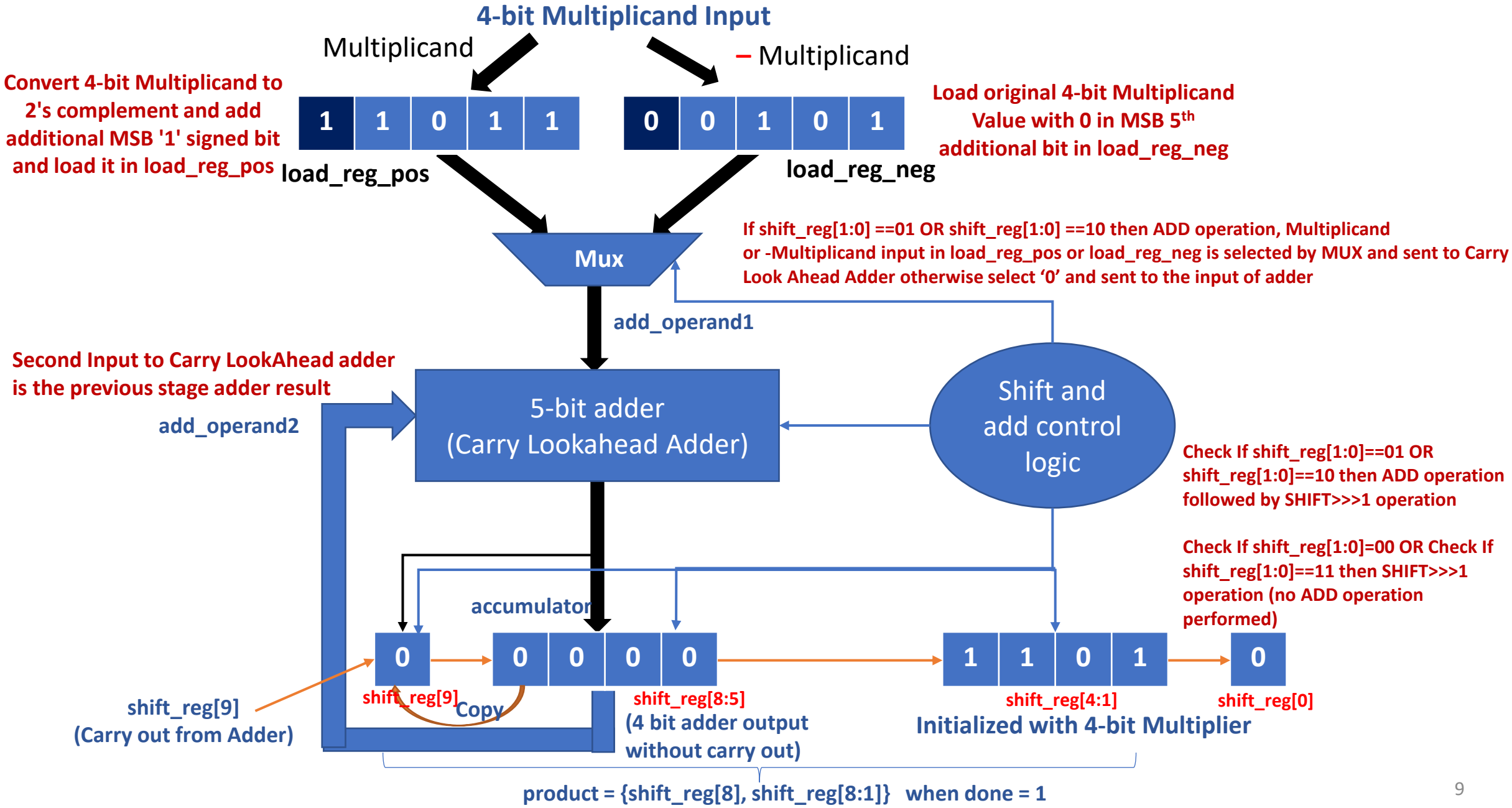
❑ Quartus Generated State Machine Diagram and State Table for reference purpose



	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 != N-1}
7	SHIFT_AND_COUNT	DONE	{count == N-1}
8	TEST	SHIFT_AND_COUNT	{shift_reg[1:0] == 00 || shift_reg[1:0] == 11}
9	TEST	ADD	{shift_reg[1:0] == 01 || shift_reg[1:0] == 10}

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

Booth Multiplier Block Diagram

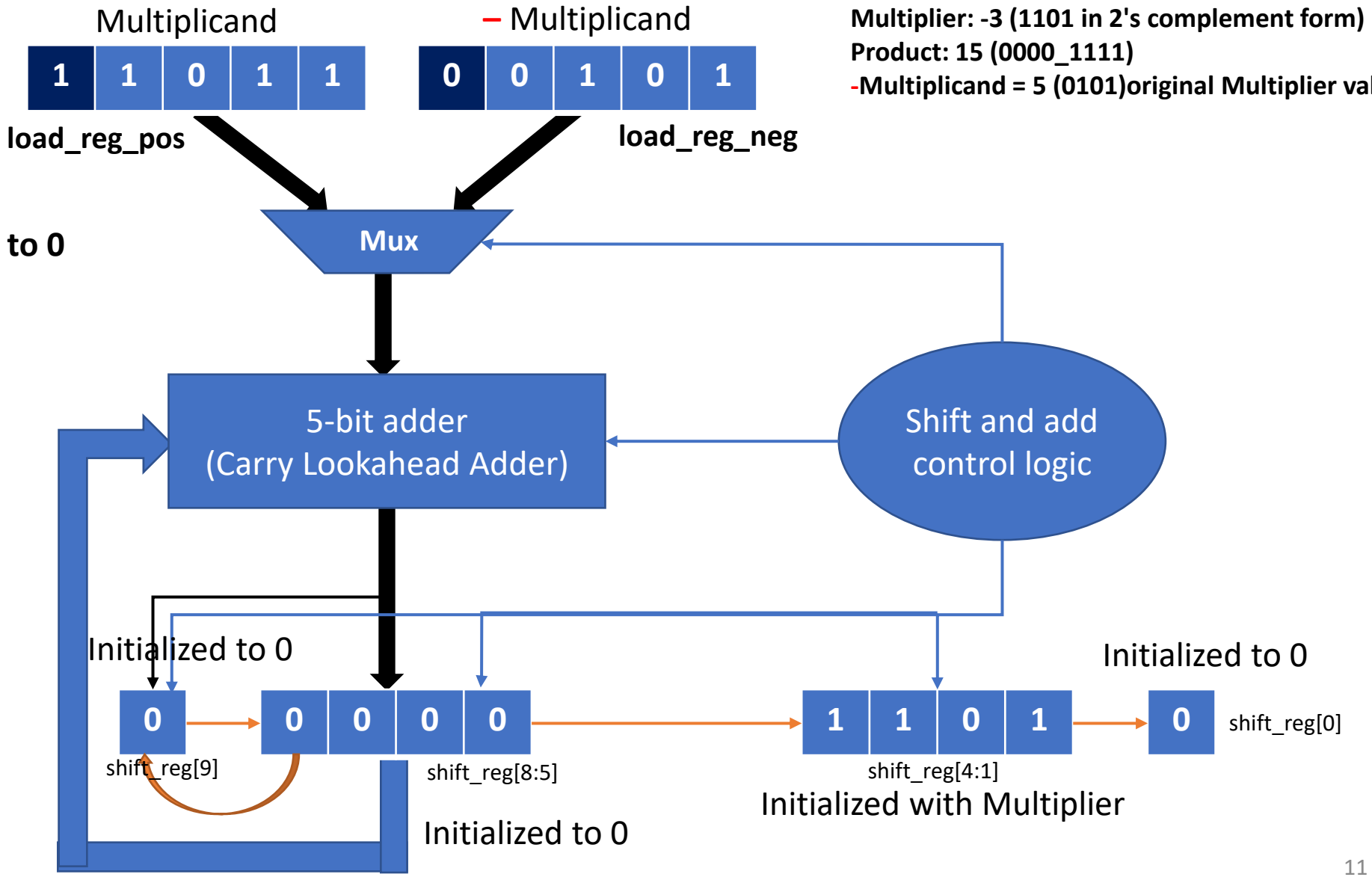


Let’s do simulation of 4-bit Booth Multiplier using SHIFT and ADD Multiplier Algorithm

Booth Multiplier Algorithm Example Steps

1. Initialize

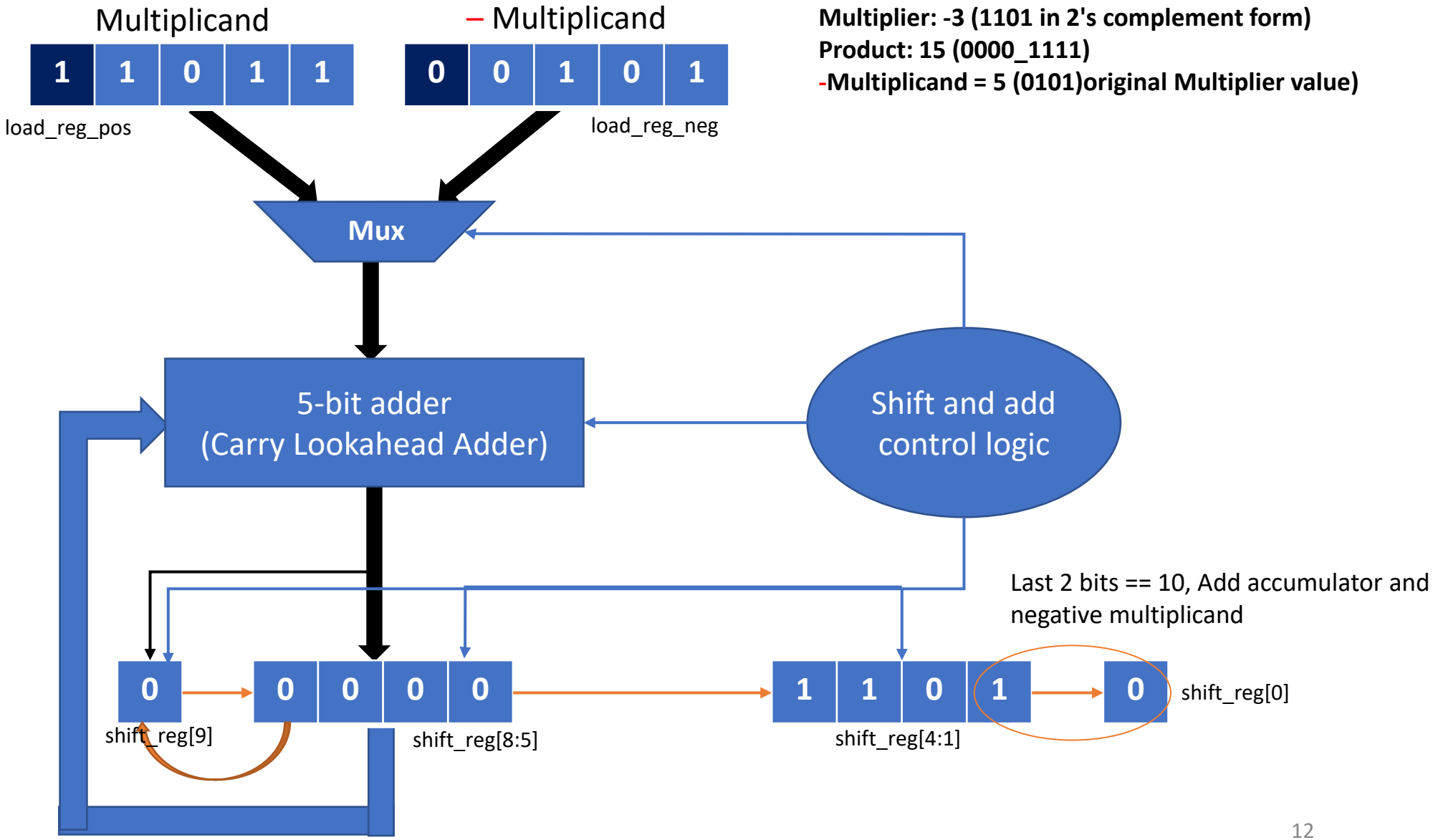
Multiplicand: -5 (1011 in 2's complement form)
Multiplier: -3 (1101 in 2's complement form)
Product: 15 (0000_1111)
-Multiplicand = 5 (0101)original Multiplier value)



Booth Multiplier Algorithm Example Steps

2. Test last 2 bits of shift register

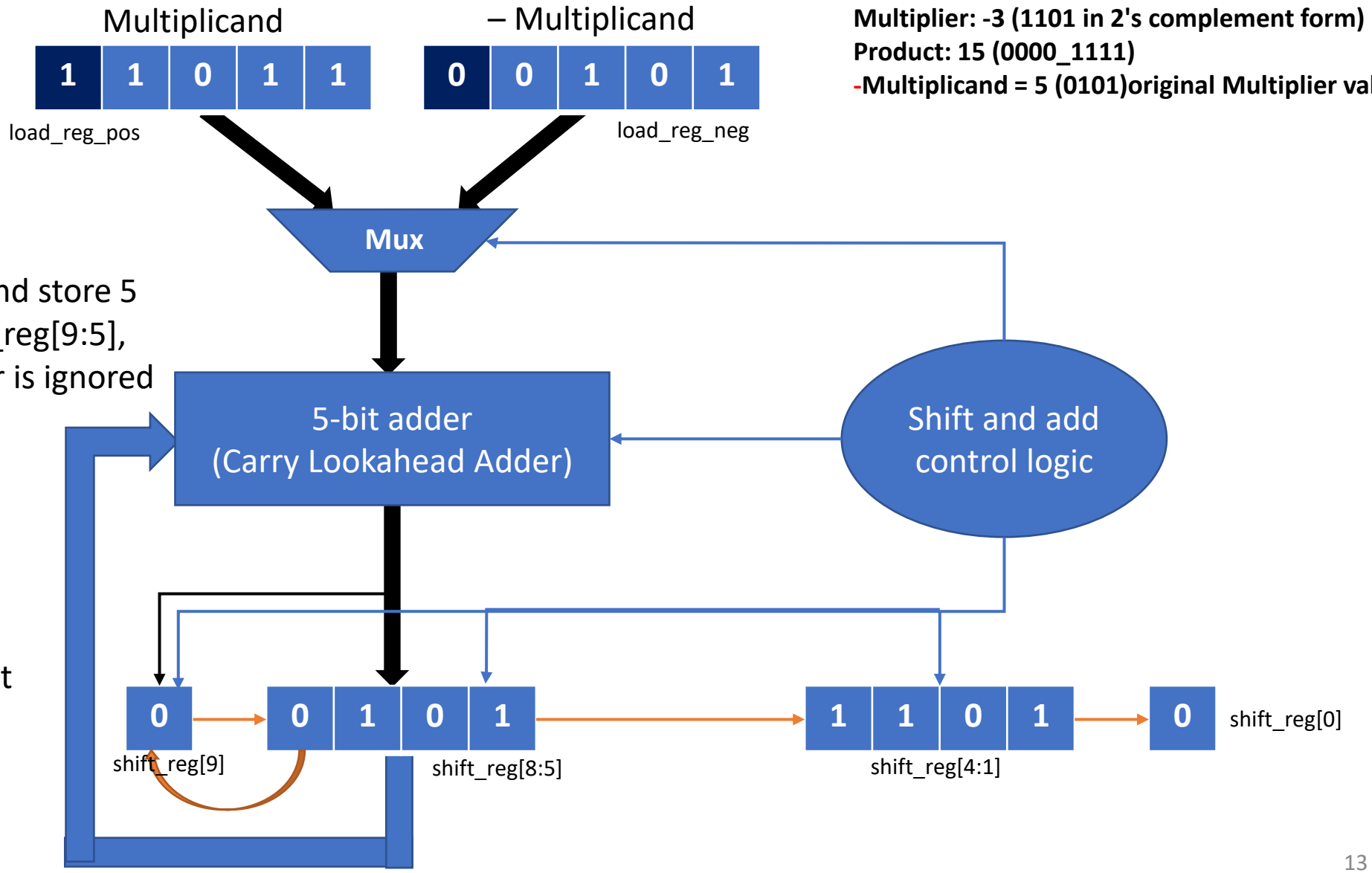
Multiplicand: -5 (1011 in 2's complement form)
Multiplier: -3 (1101 in 2's complement form)
Product: 15 (0000_1111)
-Multiplicand = 5 (0101)original Multiplier value)



Booth Multiplier Algorithm Example Steps

3. ADD

Multiplicand: -5 (1011 in 2's complement form)
Multiplier: -3 (1101 in 2's complement form)
Product: 15 (0000_1111)
-Multiplicand = 5 (0101)original Multiplier value)



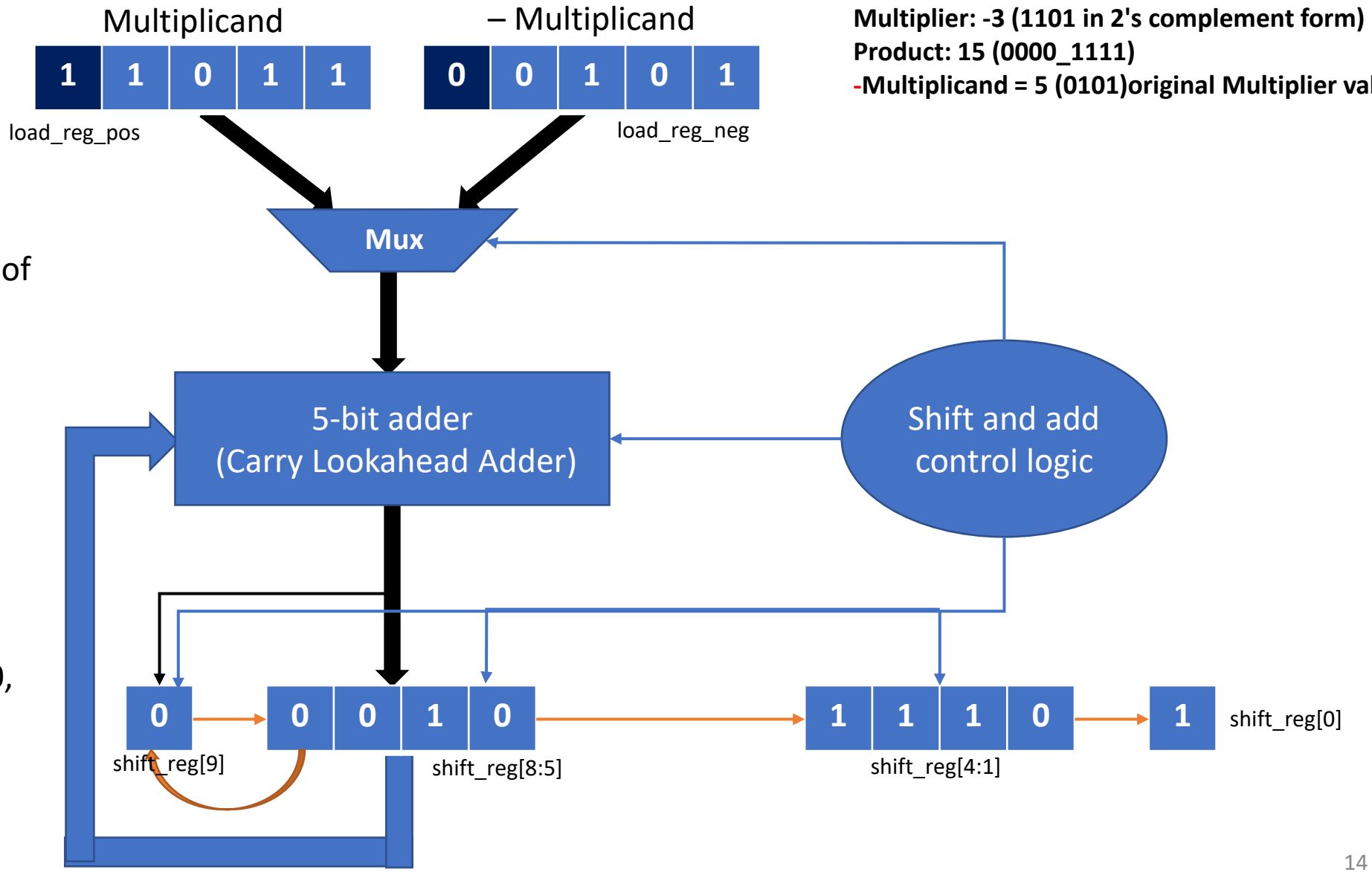
Booth Multiplier Algorithm Example Steps

4. Shift shift_reg to the right arithmetically (shift_reg >>> 1)

Multiplicand: -5 (1011 in 2's complement form)
Multiplier: -3 (1101 in 2's complement form)
Product: 15 (0000_1111)
-Multiplicand = 5 (0101)original Multiplier value)

Increment the value of the Count register

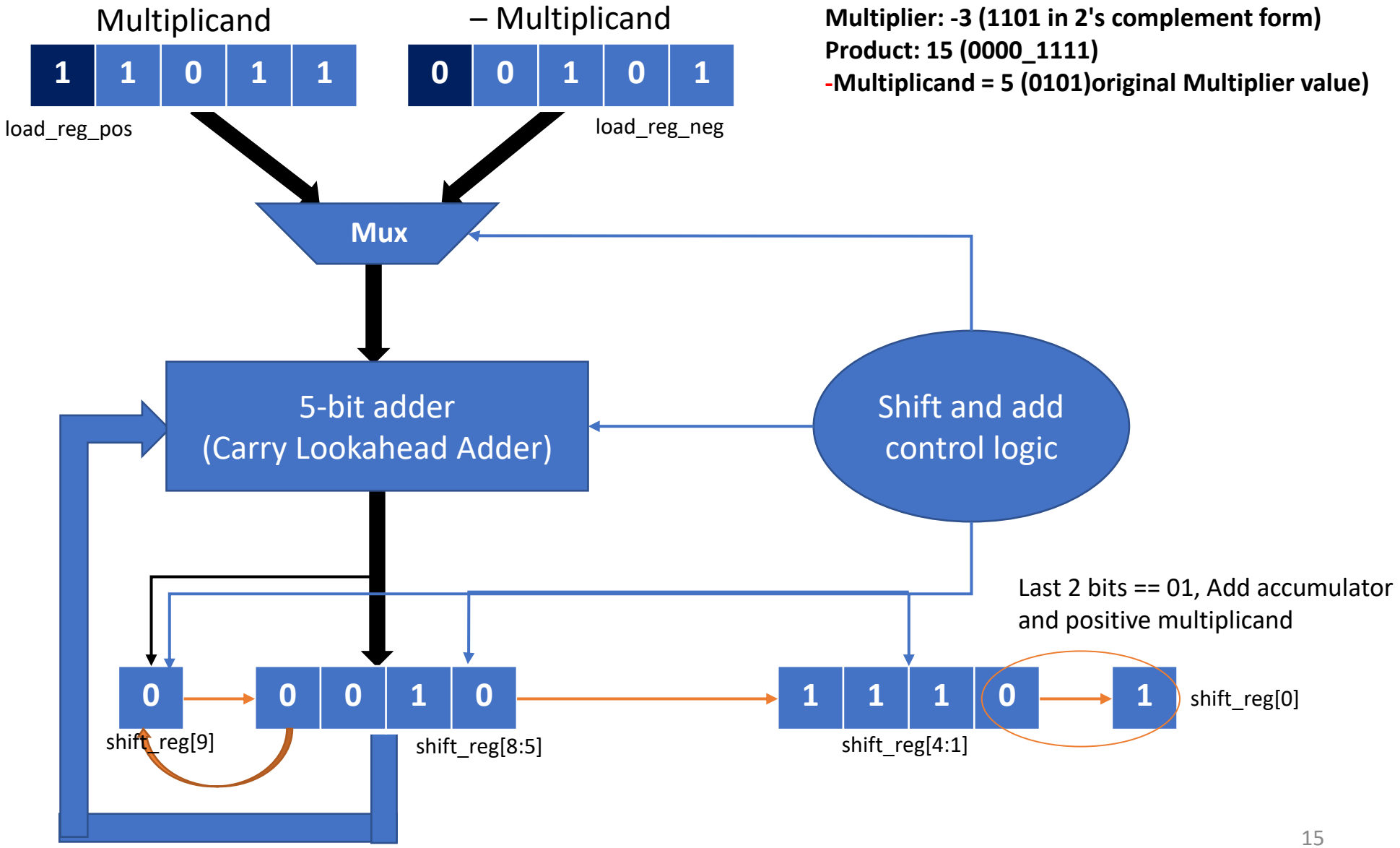
*At the end of count == 0, value of shift_reg = 0_0010_1110_1



Booth Multiplier Algorithm Example Steps

5. Test last 2 bits of shift_reg

Multiplicand: -5 (1011 in 2's complement form)
Multiplier: -3 (1101 in 2's complement form)
Product: 15 (0000_1111)
-Multiplicand = 5 (0101)original Multiplier value)

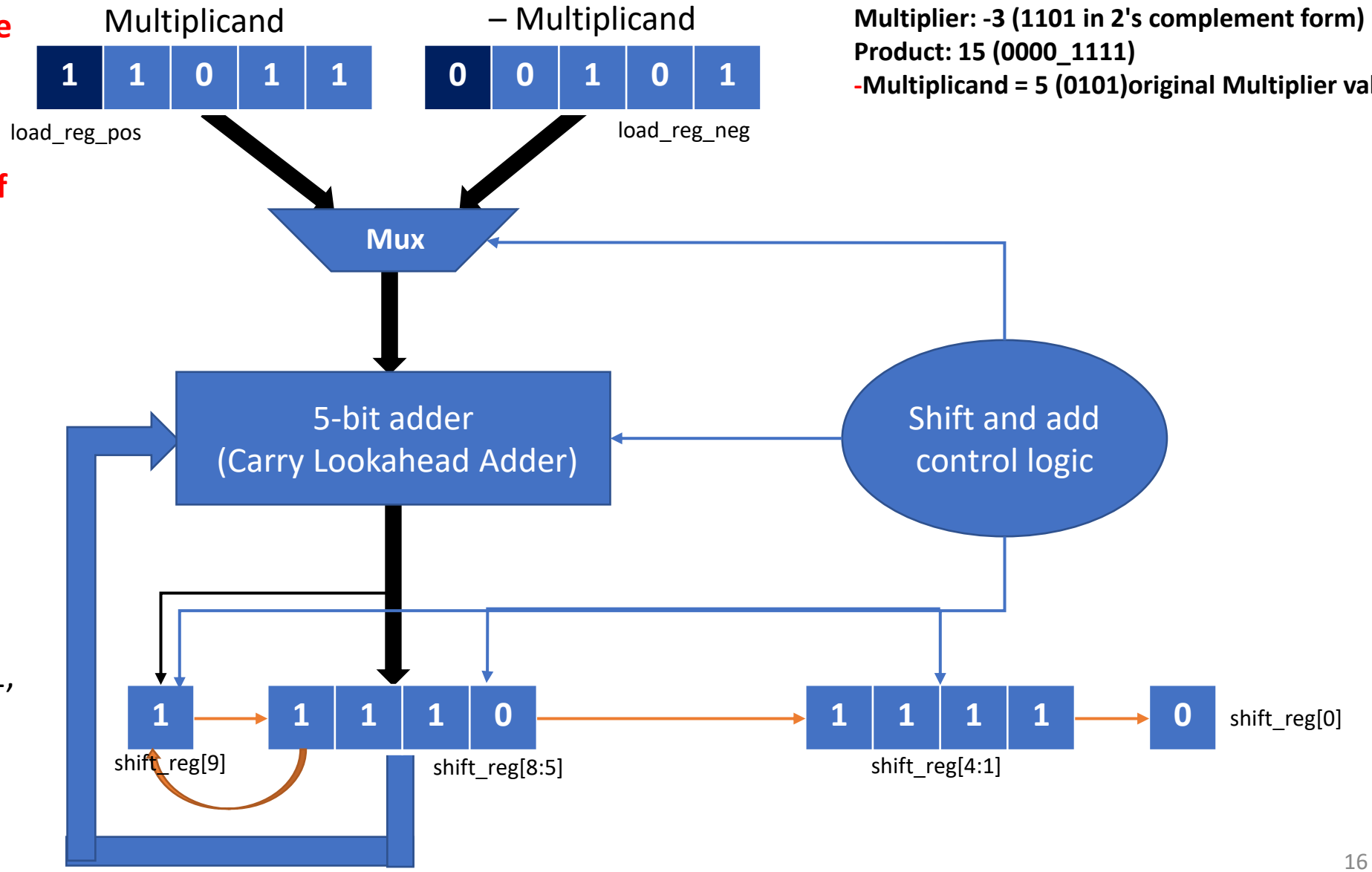


Booth Multiplier Algorithm Example Steps

6. Shift shift_reg to the right arithmetically (shift_reg >>> 1)

Increment the value of the Count register

Multiplicand: -5 (1011 in 2's complement form)
Multiplier: -3 (1101 in 2's complement form)
Product: 15 (0000_1111)
-Multiplicand = 5 (0101)original Multiplier value)



*At the end of count == 1, value of shift_reg = 1_1110_1111_0

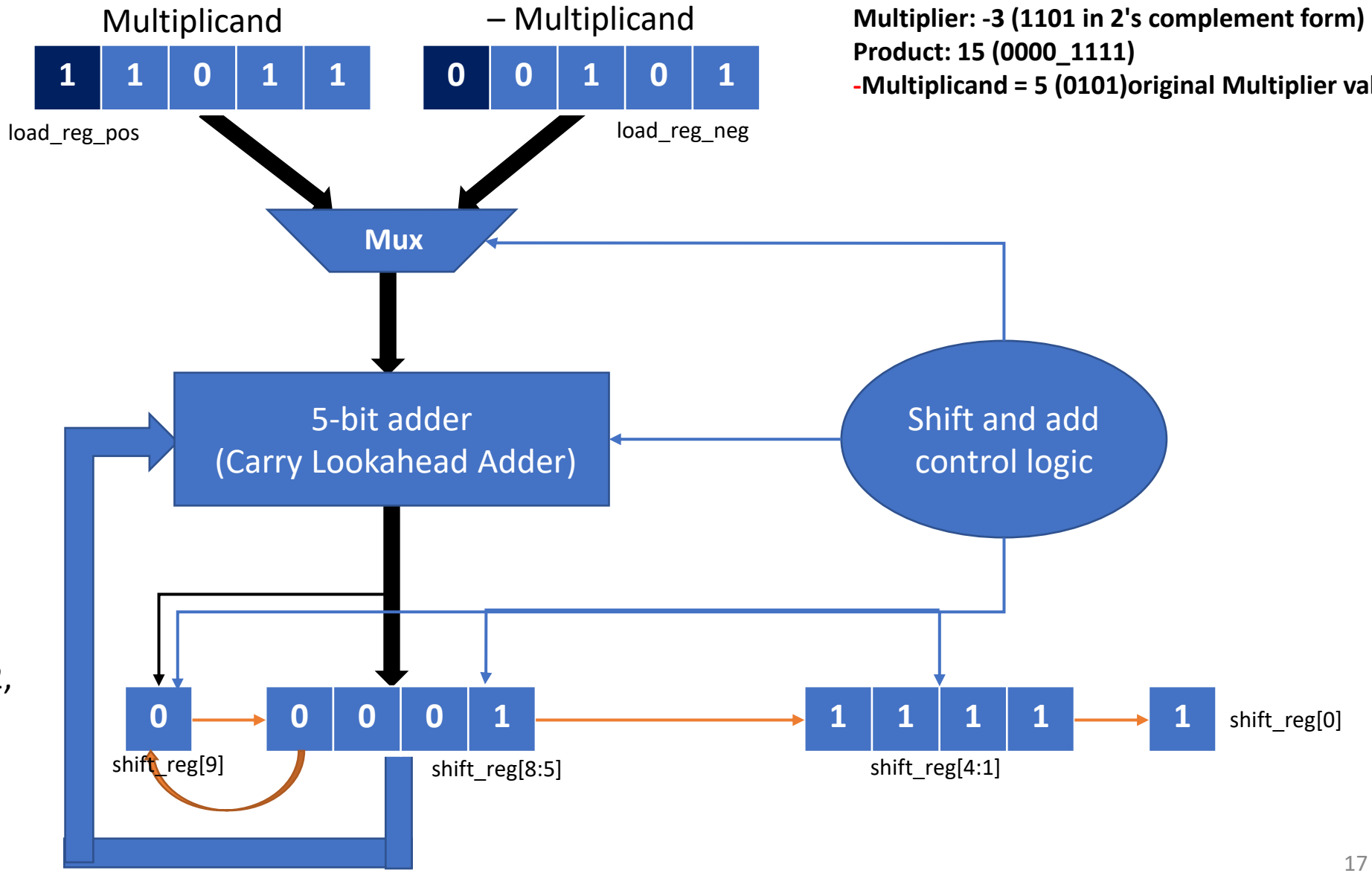
Booth Multiplier Algorithm Example Steps

7. Test last 2 bits of shift_reg

8. ADD with negative multiplicand

9. Shift >>> 1

Multiplicand: -5 (1011 in 2's complement form)
Multiplier: -3 (1101 in 2's complement form)
Product: 15 (0000_1111)
-Multiplicand = 5 (0101)original Multiplier value)



*At the end of count == 2, value of shift_reg = 0_0001_1111_1

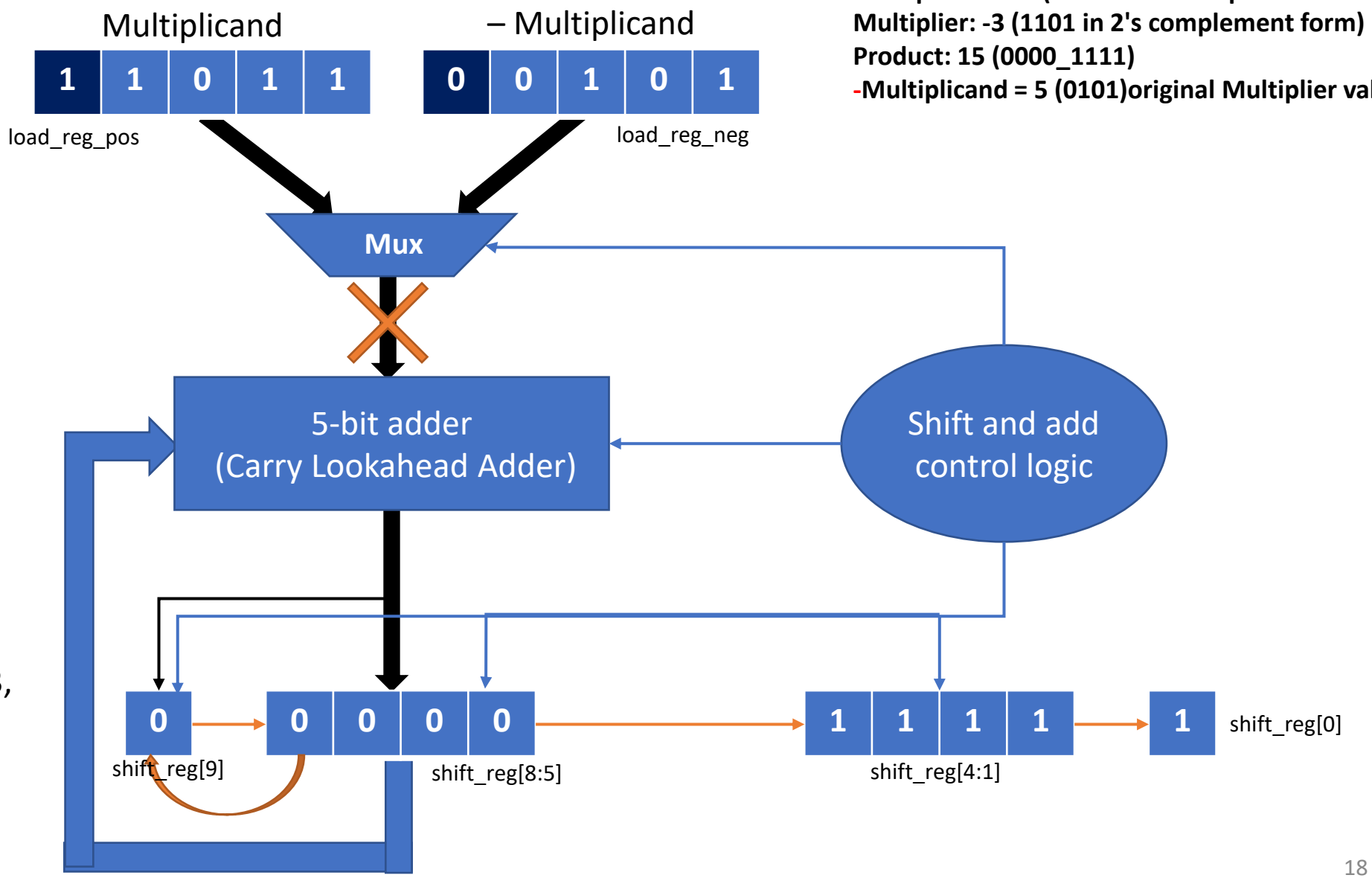
Booth Multiplier Algorithm Example Steps

7. Test last 2 bits of shift_reg

8. ADD with zero

9. Shift >>> 1

Multiplicand: -5 (1011 in 2's complement form)
Multiplier: -3 (1101 in 2's complement form)
Product: 15 (0000_1111)
-Multiplicand = 5 (0101)original Multiplier value)



*At the end of count == 3, value of shift_reg = 0_0000_1111_1

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 +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
  - 1 extra bit required for Booth Algorithm
  - shift register format = {carryout, N-bit Adder output, N-bit Multiplier, extra bit}

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

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

```
// Instantiate (N+1)-bit carry lookahead adder
carry_lookahead_adder #(.N(N+1)) adder_inst(i
//Student to add code here
//
//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
);
```

Integer Multiplier Code Development Hint

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

```
55 always_ff@(posedge clock, posedge reset) begin
56         if(reset) begin
57                 count <= 0;
58                 next_state <= IDLE;
59                 load_reg_pos <= 0;
60                 load_reg_neg <= 0;
61         end
62         else begin
63         case(next_state)
64         // Wait for start signal
65         IDLE: begin
66                 count <= 0;
67                 load_reg_pos <= 0;
68                 load_reg_neg <= 0;
69                 shift_reg <= 0;
70                 if(start == 1'b1) begin
71                         next_state <= INITIALIZE;
72                 end
73                 else begin
74                         next_state <= IDLE;
75                 end
76         end
77
78         // Load Multiplicand and Multiplier in a load register and a shift register
79         INITIALIZE: begin
80                 //**Add code
81         end
82
83         // Check shift register LSB and based on that perform ADD/Shift operation
84         // if last 2 LSB='01' then perform ADD with positive multiplicand followed by Right Shift by 1
85         // if last 2 LSB='10' then perform ADD with negative multiplicand followed by Right Shift by 1
86         // if last 2 LSB='00' then perform Right Shift by 1
87         // if last 2 LSB='11' then perform Right Shift by 1
```

Booth Multiplier Code Development Hint

- FSM code framework…. Continued

```
88          TEST: begin
89                  if(shift_reg[1:0] == 2'b01) begin
90                          //**Add code
91                  end
92                  else if(shift_reg[1:0] == 2'b10) begin
93                          //**Add code
94                  end
95                  else begin
96                          //**Add code
97                  end
98          end
99
100         ADD: begin
101                 //**Add code // Load shift register : Output sum from Adder which includes carry and retain previous lower bit of shift register
102                 //**Add code // Move to shift and increment count state
103         end
104
105         // Perform right shift by 1 on shift_reg and check if count == N-1
106         SHIFT_AND_COUNT: begin
107                 //**Add code // Right shift entire shift register by 1 position
108                 //**Add code // Increment count
109                 if(count == N-1) begin // If 'N' times SHIFT operation performed then move to Done state else go back to Test state
110                         //**Add code
111                 end
112                 else begin
113                         //**Add code
114                 end
115         end
116
117         DONE: begin
118                 next_state <= IDLE; // Wait for right shift value to be available. This is the final product value.
119         end
120         endcase
121         end
-- INSERT --
```

- 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[(2*N)], shift_reg[(2*N):1]} : 0;
```