Sequential Binary Multiplier Design

16 June 2021 11:30 PM

A binary multiplier is used to multiply two binary numbers and produce a product. A n bit multiplicand and an m bit multiplier produces a m+n bit product

This binary multiplier when implemented in parallel form, where the multiplication Is performed in 1 clock cycle, though takes lesser time, it has more adders and hence more logic gates which trades off with the area and the power consumption of the circuit

On the other hand, a sequential multiplier has only one adder and one shift register to accomplish the same binary multiplication at a reduced area and power while trading off with the speed. (i.e more clock cycles are required to perform a multiplication)

Binary multiplication is adding of multiplicands and shifting based on the multiplier.

When done manually

1. Check the multiplier 1 or 0
2. If 1 then copy the multiplicand and shift the copied partial product no if positions to the left(initially multiplier is 0th position so no shift)
3. If 0 then copy all 0 s and shift the copied partial product no of positions of the current multiplier bit to the left
4. Repeat the procedure m times where m is the number of bits in the multiplier
5. Add the partial products (This addition can be done with a sequential adder)

When done with a hardware the process can be slightly altered so that the addition and the shifting can be performed in the same clock cycle instead of two different clock cycles

When we want the product in a single clock cycle a purely parallel combinational circuit will be synthesized

However when we want to implement it sequentially, the multiplicands based on the current bit focused in the multiplier, can be shifted and added to an accumulated partial product. This way we needn't do all the partial products (shifting) and then finally adding bit by bit as we did in the manual way.

The multiplier, multiplicand and the accumulated partial product are all stored in the register. The operation of shifting and adding is performed by the state machine

In a sequential design, there are several ways to distribute the effort of multiplication in multiple clock cycles

① In 1 clock cycle 1 partial product is formed and then accumulated

② In 1 clock cycle 2 partial products are formed and then accumulated (but this will take more logic gates & faster standard cells / slower clock)

So we opt for ①

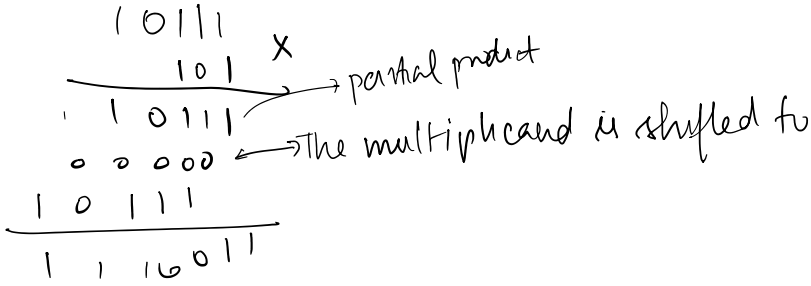
If there are p 1's in the multiplier, there will be p partial products
So we would need p-registers to store the p-partial products and a p-bit adder to add the partial products to get the final product

Instead of doing the exact equivalent of manual calculation as said above. only

① Adder to sum 2 binary numbers (1 Full Adder)

② Register to accumulate the partial product values (shift register)

Consider



Instead of shifting the multiplicand to the left, the partial product (accumulated in the shift register) can be shifted to the right

when the corresponding bit of the multiplier is 0, there is no need to add all 0's

REGISTER CONFIGURATION

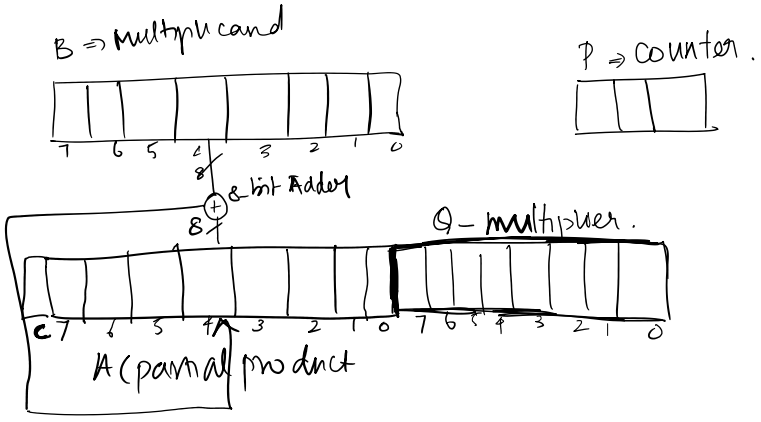
A ⇒ partial product

Q ⇒ Multiplier

B ⇒ Multiplicand

P ⇒ no of bits in the multiplier (counter) (down)

C ⇒ carry of the parallel addition



Working

① Initially the system is in the idle state, till Start becomes 1

② when start is 1, the system loads A to 0, C to 0, B ⇐ multiplicand
Q ⇐ multiplier
P ⇐ n (no of multiplier bits)

③ Depending on the value of Q[0], multiplicant is added and shifted of
if Q[0]=0 then only shift happens

④ Now since we have started to perform the multiplication on multiplicand 1st bit P--

⑤ Depending on Q[0], always ↓ happens. whether to add or not dependson Q[0]=1 or Q[0]=0
{C, A, Q} >>1 (Right shift by 1). Q[0] is taken.

If Q[0] is 1, multiplicand is added to the shifted partial product (content

of reg A)

⑥ On this addition a carry C can/cannot be produced, but the value of this carry is stored in C, so as to be added to the next partial product when it is shifted into A

⑦ Now P is checked if it is 0, if not then goto step 4

When P==0, the content in the combined register A and Q is the product

CONTROL PATH DESIGN

Three control signals are required to data path

① load registers	load_regs	(Interface signals)
② Shift register	shift_regs	
③ Add registers	add_regs	
④ decrement the counter P	decr_P	

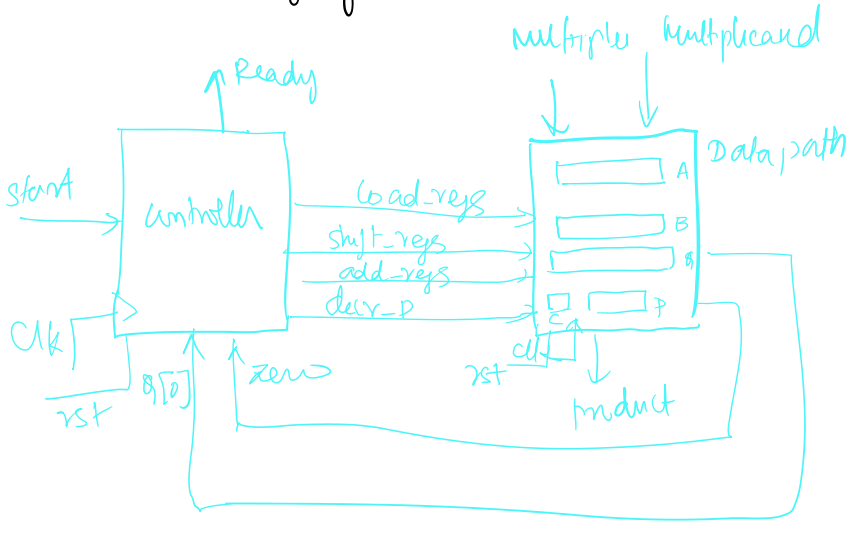
Feedback from data path.

⑤ Q[0] (Current last bit of multiplier)

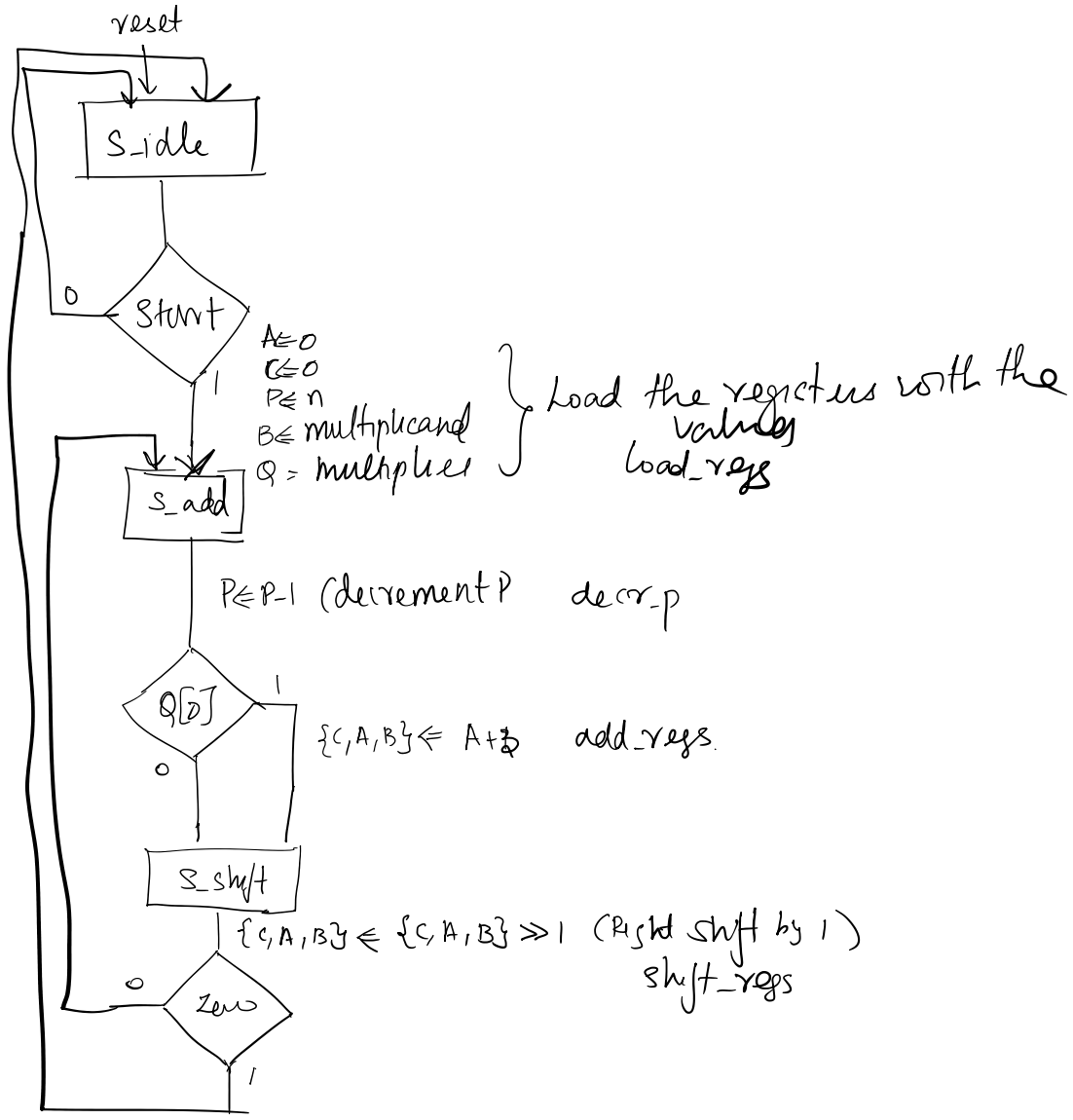
⑥ Zero (zero==1, when P=0 else zero==0)

⑦ Start signal to the Controller

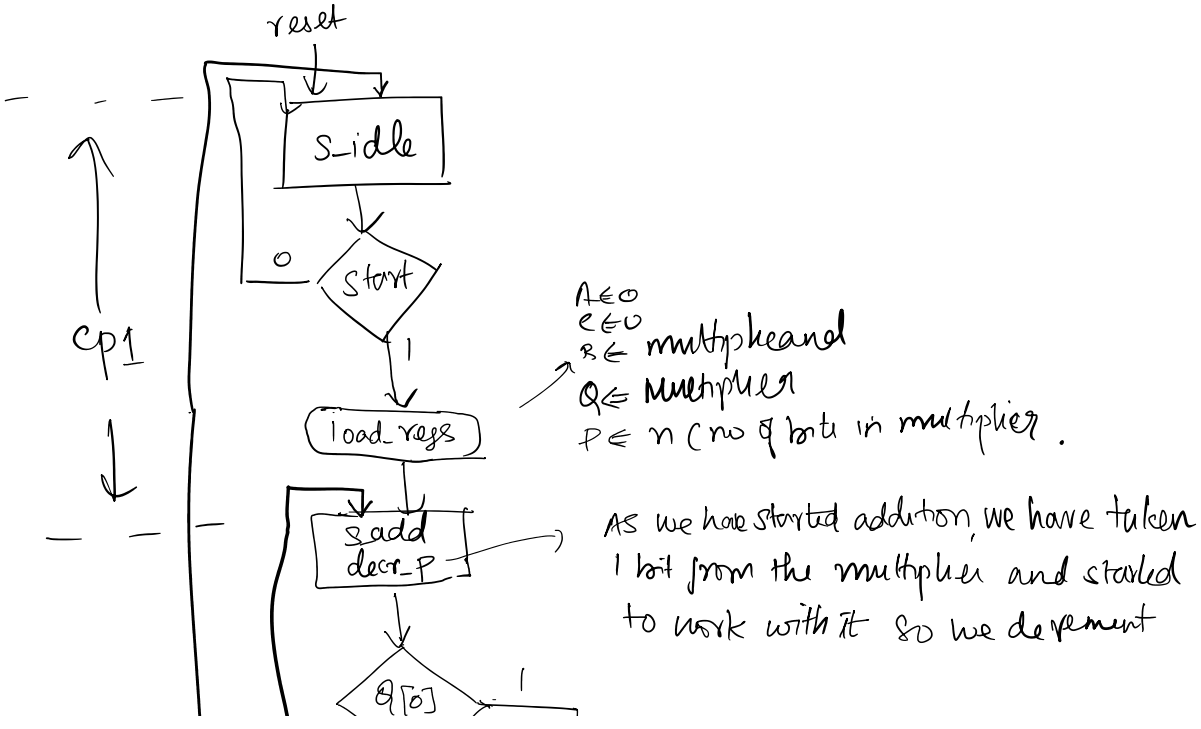
⑧ Ready signal (since the contents of the product register are changing we need to know when the multiplier is ready

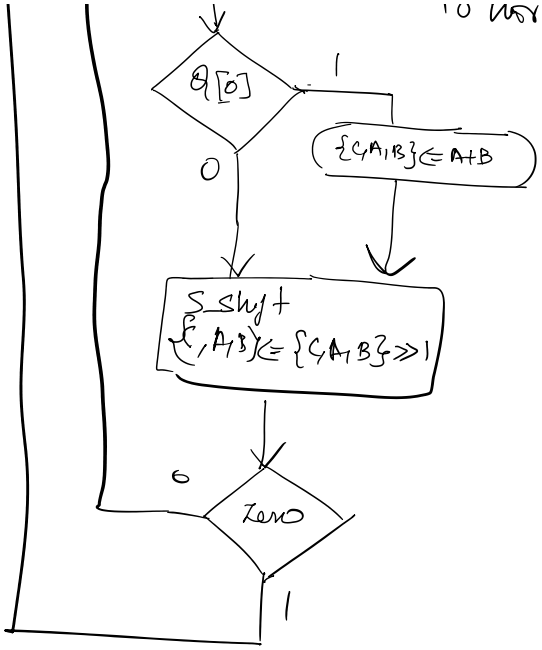


ASMD chart



ASMD chart with control signals





to work with it so we decrement

From the above chart we have 3 states

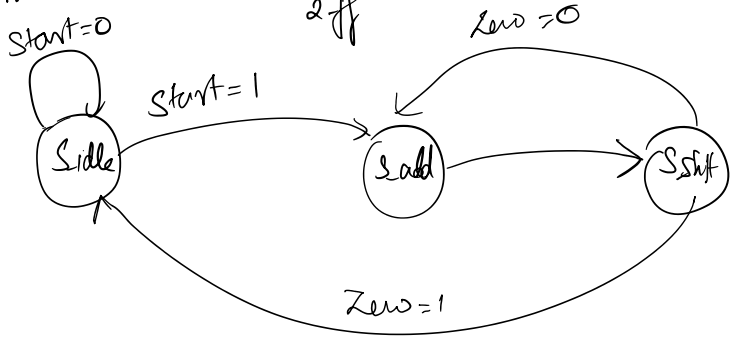
Binary
00 Gray code assignment
01 with 1 bit
10 change to simplify logic assignment

Graycode
00 =
01
11

2 ff

S_idle
S_add
S_shift
} we need 2 bits to represent 3 states so 2 flip flops.

One hot
001
010
100
3 flip flops



State diagram

S_idle → S_add

A ∈ 0 Q ∈ 0 P ∈ n
B ∈ multiplicand
Q ∈ multiplier

S_add → S_shift

P ∈ P-1

if Q[0]=1, {C,A,B} = A+B
(i.e) A ∈ A+B
C ∈ Cout

S_shift

{C,A,B} ∈ {C,A,B} >> 1
[C becomes 0]

multiplicand multiplier

C becomes 0

Example multiplicand $B = 10111_2 = 17_H = 23_{10}$ multiplier $Q = 10011_2 = 13_H = 19_{10}$

	C	A	Q	P
	0	00000	10011	101
$Q_0=1$, add B	0	10111		100
		10111		
shift right CAQ	0	01011	11001	
$Q_0=1$ add B		10111		
	1	00010		011
Shift CAQ	0	10001	01100	
$Q_0=0$, shift CAQ	0	01000	10110	010
$Q_0=0$, shift CAQ	0	00100	01011	001
$Q_0=1$, add B		10111		
	0	11011		000
shift CAQ	0	01101101 01		000 Stop

Product 00110110101_2

STATE TABLE

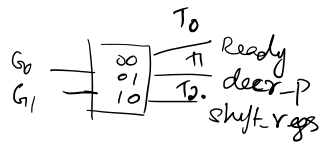
[] ⇒ moore type o/p

State Symbol	Present state G_1 G_0	Input start	Q[0]	zero	Next state $Q_{t+1}=D$ G_1 G_0	Output T_0 ready	load_reg	T_1 decr_p	add_reg	T_2 shift_reg
S_idle	0 0	0	X	X	0 0	1	0	0	0	0
S_idle	0 0	1	X	X	0 1	1	1	0	0	0
S_add.	0 1	X	0	X	1 0	0	0	1	0	0
	0 1	X	1	X	1 0	0	0	1	1	0
	1 0	X	X	0	0 1	0	0	0	0	1
	1 0	X	X	1	0 0	0	0	0	0	1

We realise with DFF

These more type o/p which is 1 at a given time is determined by

These more type o/p which is 1 at a given time is determined by the state value.

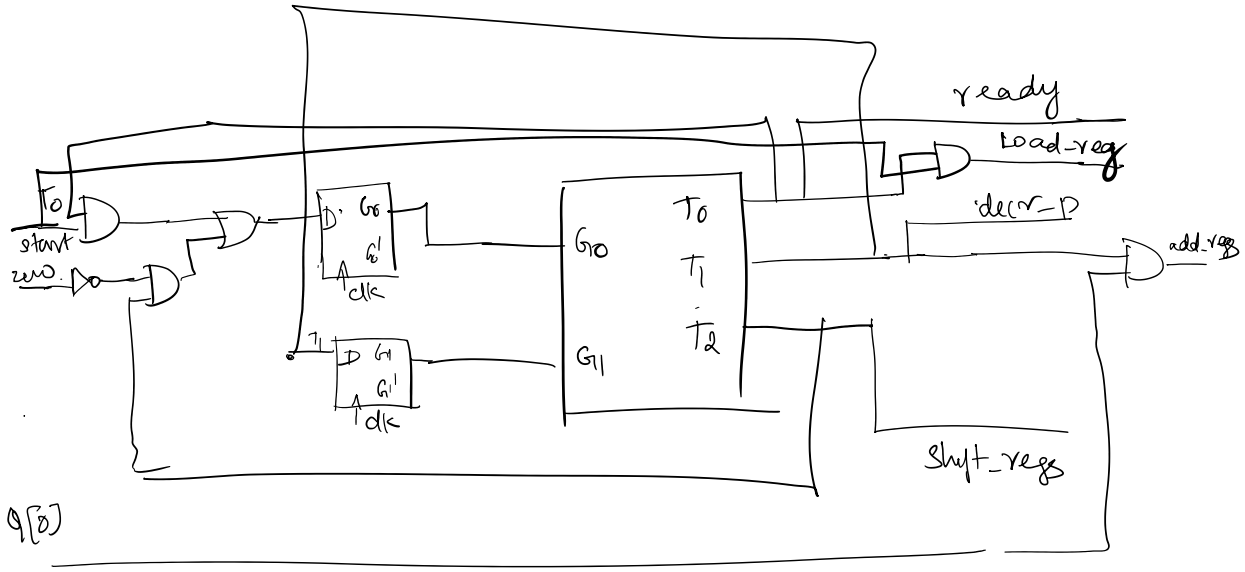


$D_{G_1} = T_1$

$D_{G_0} = start\ T_0 + zero'\ T_2.$

$Load\_reg = T_0.start$

$add\_regs = T_1.Q[0]$



Control path

Ref: Morris Mano
One hot design

One-Hot Design (One Flip-Flop per State)

Another method of control logic design is the one-hot assignment, which results in a sequential circuit with one flip-flop per state. Only one of the flip-flops contains a 1 at any time; all others are reset to 0. The single 1 propagates from one flip-flop to another under the control of decision logic. In such a configuration, each flip-flop represents a state that is present only when the control bit is transferred to it.

This method uses the maximum number of flip-flops for the sequential circuit. For example, a sequential circuit with 12 states requires a minimum of four flip-flops. By contrast, with the method of one flip-flop per state, the circuit requires 12 flip-flops, one for each state. At first glance, it may seem that this method would increase system cost, since more flip-flops are used. But the method offers some advantages that may not be apparent. One advantage is the simplicity with which the logic can be designed by inspection of the ASMD chart or the state diagram. No state or excitation tables are needed if *D*-type flip-flops are employed. The one-hot method offers a savings in design effort, an increase in operational simplicity, and a possible decrease in the total number of gates, since a decoder is not needed.

The design procedure for a one-hot state assignment will be demonstrated by obtaining the control circuit specified by the state diagram of Fig. 8.16(a). Since there are three states in the state diagram, we choose three *D* flip-flops and label their outputs G_0, G_1, and G_2, *corresponding to S_idle, S_add*, and *S_shift*, respectively. The input equations for setting each flip-flop to 1 are determined from the present state and the input conditions along the corresponding directed lines going into the state. For example, D_{G0}, the input to flip-flop G_0, is set to 1 if the machine is in state G_0 and *Start* is not asserted, or if the machine is in state G_2 and *Zero* is asserted. These conditions are specified by the input equation:

$$D_{G0} = G_0 Start' + G_2 Zero$$

In fact, the condition for setting a flip-flop to 1 is obtained directly from the state diagram, from the condition specified in the directed lines going into the corresponding flip-flop state ANDed with the previous flip-flop state. If there is more than one directed line going into a state, all conditions must be ORed. Using this procedure for the other three flip-flops, we obtain the remaining input equations:

$$D_{G1} = G_0 Start + G_2 Zero'$$
$$D_{G2} = G_1$$

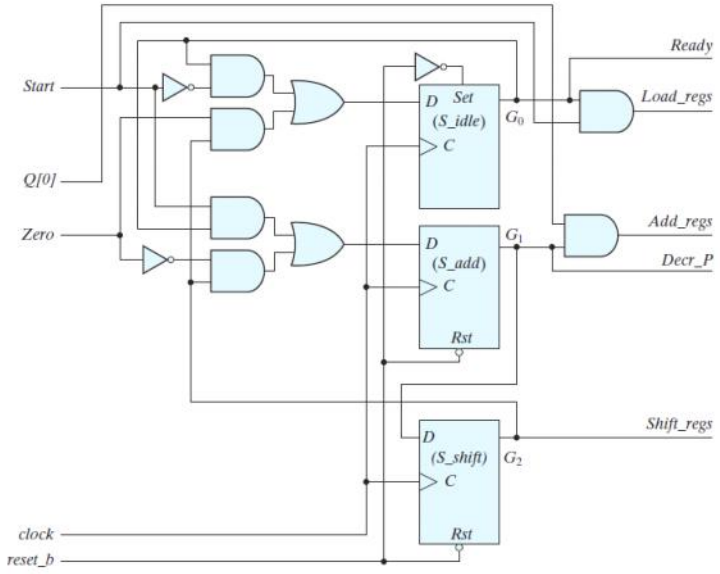


FIGURE 8.18
Logic diagram for one-hot state controller

The logic diagram of the one-hot controller (with one flip-flop per state) is shown in Fig. 8.18. The circuit consists of three *D* flip-flops labeled G_0 through G_2, together with the associated gates specified by the input equations. Initially, flip-flop G_0 must be set to 1 and all other flip-flops must be reset to 0, so that the flip-flop representing the initial state is enabled. This can be done by using an asynchronous preset on flip-flop G_0 and an asynchronous clear for the other flip-flops. Once started, the controller with one flip-flop per state will propagate from one state to the other in the proper manner. Only one flip-flop will be set to 1 with each clock edge; all others are reset to 0, because their *D* inputs are equal to 0.

HDL Code

```
module Sequential_Binary_Multiplier (Product, Ready, Multiplicand, Multiplier, Start,clock, reset_b);
// Default configuration: five-bit datapath
parameter dp_width = 5; // Set to width of datapath
output [2*dp_width -1: 0] Product;
output Ready;
input [dp_width -1: 0] Multiplicand, Multiplier;
input Start, clock, reset_b;
parameter BC_size = 3; // Size of bit counter
parameter S_idle = 3'b001, // one-hot code
S_add = 3'b010,
S_shift = 3'b100;
reg [2: 0] state, next_state;
reg [dp_width -1: 0] A, B, Q; // Sized for datapath
reg C;
reg [BC_size -1: 0] P;
reg Load_regs, Decr_P, Add_regs, Shift_regs;
// Miscellaneous combinational logic
assign Product = {A, Q};
wire Zero = (P == 0); // counter is zero
// Zero = ~|P; // alternative

wire Ready = (state == S_idle); // controller status

// control unit
always @ ( posedge clock, negedge reset_b)
if (~reset_b) state <= S_idle; else state <= next_state;

always @ (state, Start, Q[0], Zero) begin
next_state = S_idle;
Load_regs = 0;
Decr_P = 0;
Add_regs = 0;
Shift_regs = 0;
```

```
case (state)
S_idle: begin if (Start) next_state = S_add; Load_regs = 1; end
S_add: begin next_state = S_shift; Decr_P = 1; if ( Q[0]) Add_regs = 1; end
S_shift: begin Shift_regs = 1; if (Zero) next_state = S_idle;
else next_state = S_add; end
default : next_state = S_idle;
endcase
end


// datapath unit
always @ ( posedge clock) begin
if (Load_regs) begin
P <= dp_width;
A <= 0;
C <= 0;
B <= Multiplicand;
Q <= Multiplier;
end
if (Add_regs) {C, A} <= A + B;
if (Shift_regs) {C, A, Q} <= {C, A, Q} >> 1;
if (Decr_P) P <= P -1;
end
endmodule
```

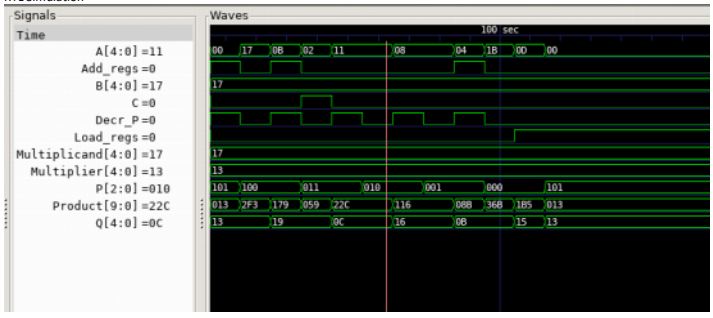
Test bench

```
// Test bench for the binary multiplier
module t_Sequential_Binary_Multiplier;
parameter dp_width = 5; // Set to width of datapath
wire [2*dp_width -1: 0] Product; // Output from multiplier
wire Ready;
reg [dp_width -1: 0] Multiplicand, Multiplier; // Inputs to multiplier
reg Start, clock, reset_b;
// Instantiate multiplier
Sequential_Binary_Multiplier M0 (Product, Ready, Multiplicand, Multiplier, Start, clock,
reset_b);
// Generate stimulus waveforms
initial #200 $finish ;
initial
begin
Start = 0;
reset_b = 0;
#2 Start = 1; reset_b = 1;
Multiplicand = 5'b10111; Multiplier = 5'b10011;
#10 Start = 0;
end
initial
begin
clock = 0;
repeat (26) #5 clock = ~clock;
end
// Display results and compare with Table 8.5
always @ ( posedge clock)
begin
$dumpfile("tb_sbm.vcd");
$dumpvars;
$strobe ("C=%b A=%b Q=%b P=%b time=%0d",M0.C,M0.A,M0.Q,M0.P, $time );
end
endmodule
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

┌──(root💀shriharipc)-[/d/RISC-V/verilog/Sequential Binary Multiplier]
└─# iverilog sbm.v tb_sbm.v
┌──(root💀shriharipc)-[/d/RISC-V/verilog/Sequential Binary Multiplier]
└─# ./a.out
C=0 A=00000 Q=10011 P=101 time=5
C=0 A=10111 Q=10011 P=100 time=15
C=0 A=01011 Q=11001 P=100 time=25
C=1 A=00010 Q=11001 P=011 time=35
C=0 A=10001 Q=01100 P=011 time=45
C=0 A=10001 Q=01100 P=010 time=55
C=0 A=01000 Q=10110 P=010 time=65
C=0 A=01000 Q=10110 P=001 time=75
C=0 A=00100 Q=01011 P=001 time=85
C=0 A=11011 Q=01011 P=000 time=95
C=0 A=01101 Q=10101 P=000 time=105
C=0 A=00000 Q=10011 P=101 time=115
C=0 A=00000 Q=10011 P=101 time=125
┌──(root💀shriharipc)-[/d/RISC-V/verilog/Sequential Binary Multiplier]
└─#
```

RTL Simulation



Synthesis

```
yosys> read_liberty -lib /d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
1. Executing Liberty frontend.
Imported 428 cell types from liberty file.

yosys> read_verilog sbm.v

2. Executing Verilog-2005 frontend: sbm.v
Parsing Verilog input from `sbm.v' to AST representation.
Generating RTLIL representation for module `\Sequential_Binary_Multiplier'.
Note: Assuming pure combinatorial block at sbm.v:28 in
compliance with IEC 62142(E):2005 / IEEE Std. 1364.1(E):2002. Recommending
use of @* instead of @(...) for better match of synthesis and simulation.
Successfully finished Verilog frontend.

yosys> synth -top Sequential_Binary_Multiplier
```

```
=== Sequential_Binary_Multiplier ===

   Number of wires:                 79
   Number of wire bits:            128
   Number of public wires:          15
   Number of public wire bits:      50
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                104
     $_ANDNOT_                      17
     $_AND_                          1
     $_AOI3_                         2
     $_AOI4_                         1
     $_DFF_PN0_                      2
     $_DFF_PN1_                      1
     $_DFF_P_                       19
     $_MUX_                         29
     $_NAND_                         6
     $_NOR_                          3
     $_NOT_                          1
     $_OAI3_                         3
     $_OR_                           8
     $_XNOR_                         4
     $_XOR_                          7
```

```
yosys> abc -liberty /d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
```

```
4.1.2. Re-integrating ABC results.
ABC RESULTS:   sky130_fd_sc_hd__a21o_1 cells:        2
ABC RESULTS:   sky130_fd_sc_hd__a21oi_1 cells:       5
ABC RESULTS:   sky130_fd_sc_hd__a221o_1 cells:       1
ABC RESULTS:   sky130_fd_sc_hd__a31oi_1 cells:      10
ABC RESULTS:   sky130_fd_sc_hd__and2_0 cells:        3
ABC RESULTS:   sky130_fd_sc_hd__clkinv_1 cells:      2
ABC RESULTS:   sky130_fd_sc_hd__lpflow_inputiso1p_1 cells:       1
ABC RESULTS:   sky130_fd_sc_hd__lpflow_isobufsrc_1 cells:       1
ABC RESULTS:   sky130_fd_sc_hd__nand2_1 cells:      38
ABC RESULTS:   sky130_fd_sc_hd__nand2b_1 cells:      1
ABC RESULTS:   sky130_fd_sc_hd__nand3_1 cells:       2
ABC RESULTS:   sky130_fd_sc_hd__nor2_1 cells:       12
ABC RESULTS:   sky130_fd_sc_hd__nor3_1 cells:        3
ABC RESULTS:   sky130_fd_sc_hd__o211ai_1 cells:      1
ABC RESULTS:   sky130_fd_sc_hd__o21ai_0 cells:       6
ABC RESULTS:   sky130_fd_sc_hd__o22ai_1 cells:       1
ABC RESULTS:   sky130_fd_sc_hd__o31ai_1 cells:       1
ABC RESULTS:   sky130_fd_sc_hd__or3b_1 cells:        2
ABC RESULTS:   sky130_fd_sc_hd__xnor2_1 cells:       3
ABC RESULTS:   sky130_fd_sc_hd__xor2_1 cells:        1
ABC RESULTS:        internal signals:       59
ABC RESULTS:           input signals:       33
ABC RESULTS:          output signals:       23
Removing temp directory.
```

```
yosys> dfflibmap -liberty /d/RISC-V/sky130RTLDesignAndSynthesisWorkshop/my_lib/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
```

```
Mapping DFF cells in module `\Sequential_Binary_Multiplier':
  mapped 2 $_DFF_PN0_ cells to \sky130_fd_sc_hd__dfrtp_1 cells.
  mapped 1 $_DFF_PN1_ cells to \sky130_fd_sc_hd__dfstp_2 cells.
  mapped 19 $_DFF_P_ cells to \sky130_fd_sc_hd__dfxtp_1 cells.
```

```
yosys> opt_clean -purge

6. Executing OPT_CLEAN pass (remove unused cells and wires).
Finding unused cells or wires in module \Sequential_Binary_Multiplier..
Removed 0 unused cells and 118 unused wires.
<suppressed ~4 debug messages>
```

Yosys>show