8-Bit Multiplier

An multiplier is a digital circuit or component that is specifically designed to perform multiplication operations on two binary numbers. It takes two binary numbers as input and produces output. There are so many ways to perform multiplication between two numbers. Here I present one of the method i.e Shift and Add multiplication.

Shift-and-add multiplication is similar to the multiplication performed by paper and pencil. This method adds the multiplicand X to itself Y times, where Y denotes the multiplier. To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.

As an example, consider the multiplication of two 4-bit numbers, 13 (1101) and 11 (1011).

Multiplicand	1101 ×
Multiplier	1011
	1101
	1101
	0000
	1101
Product	10001111 (143)

Now that each partial product is either the multiplicand (1101) shifted over by the appropriate number of places or zero. Instead of forming all the partial products first and then adding, each new partial product is added in as soon as it is formed, which eliminates the need for adding more than two binary numbers at a time.

Multiplication of two 4-bit numbers requires a 4-bit multiplicand register, a 4-bit multiplier register, a 4-bit full adder, and an 8-bit register for the product. The product register serves as an accumulator to accumulate the sum of the partial products. If the multiplicand were shifted left each time before it was added to the accumulator, as was done in the previous example, an 8-bit adder would be needed. Therefore, it is better to shift the contents of the product register to the right each time, as shown in the block diagram below ;

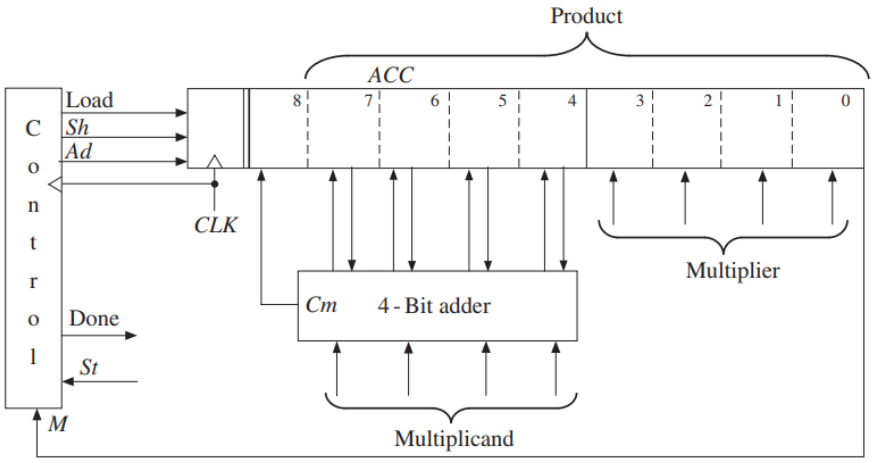


Figure : Block Diagram for Binary Multiplier

This type of multiplier is sometimes referred to as a serial-parallel multiplier, since the multiplier bits are processed serially, but the addition takes place in parallel. As indicated by the arrows on the diagram, 4 bits from the accumulator (ACC) and 4 bits from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator. When an add signal (Ad) occurs, the adder outputs are transferred to the accumulator by the next clock pulse, thus causing the multiplicand to be added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry that is generated when the multiplicand is added to the accumulator. When a shift signal (Sh) occurs, all 9 bits of ACC are shifted right by the

next clock pulse. Since the lower 4 bits of the product register are initially unused, we will store the multiplier in this location instead of in a separate register. As each multiplier bit is used, it is shifted out the right end of the register to make room for additional product bits. A shift signal (Sh) causes the contents of the product register (including the multiplier) to be shifted right one place when the next clock pulse occurs. The control circuit puts out the proper sequence of add and shift signals after a start signal (St 5 1) has been received. If the current multiplier bit (M) is 1, the multiplicand is added to the accumulator followed by a right shift; if the multiplier bit is 0, the addition is skipped and only the right shift occurs. The multiplication example (13 × 11) is reworked as follows showing the location of the bits in the registers at each clock time:

initial contents of product register	0 0 0 0 0 | 1 0 1 1 ← M (11)
(add multiplicand since M = 1)	1 1 0 1 M (13)
after addition	0 1 1 0 1 | 1 0 1 1
after shift	0 0 1 1 0 1 | 1 0 1 ← M
(add multiplicand since M = 1)	1 1 0 1
after addition	1 0 0 1 1 1 | 1 0 1
after shift	0 1 0 0 1 1 1 | 1 0 ← M
(skip addition since M = 0)	
after shift	0 0 1 0 0 1 1 1 | 1 ← M
(add multiplicand since M = 1)	1 1 0 1
after addition	1 0 0 0 1 1 1 1 | 1
after shift (final answer)	0 1 0 0 0 1 1 1 | 1 (143)

The control circuit must be designed to output the proper sequence of add and shift signals. Figure 1 shows a state graph for the control circuit. In Figure , S0 is the reset state, and the circuit stays in S0 until a start signal (St 5 1) is received. This generates a Load signal, which causes the multiplier to be loaded into the lower 4 bits of the accumulator (ACC) and the upper 5 bits of the accumulator to be cleared. In state S1 , the low-order bit of the multiplier (M) is tested. If M 5 1, an add signal is generated, and if M 5 0, a shift signal is generated. Similarly, in states S3 , S5 , and S7 , the current multiplier bit (M) is tested to determine whether to generate an add or shift signal. A shift signal is always generated at the next clock time following an add signal (states S2 , S4 , S6 , and S8). After four shifts have been generated, the control network goes to S9 and a done signal is generated before returning to S0 .

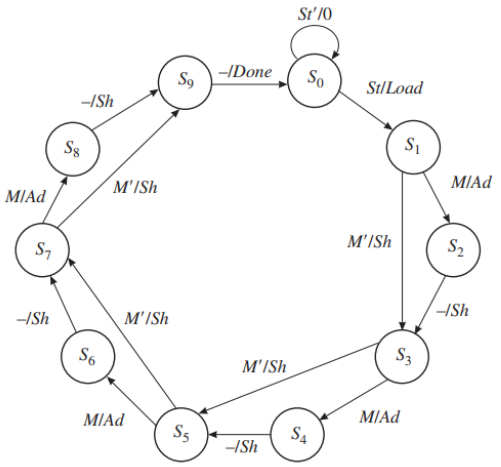
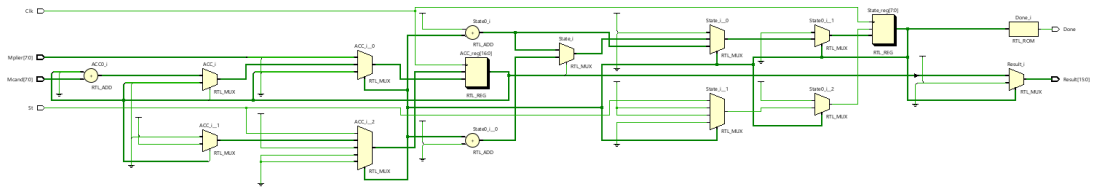


Figure 1: State Graph for Binary Multiplier Control

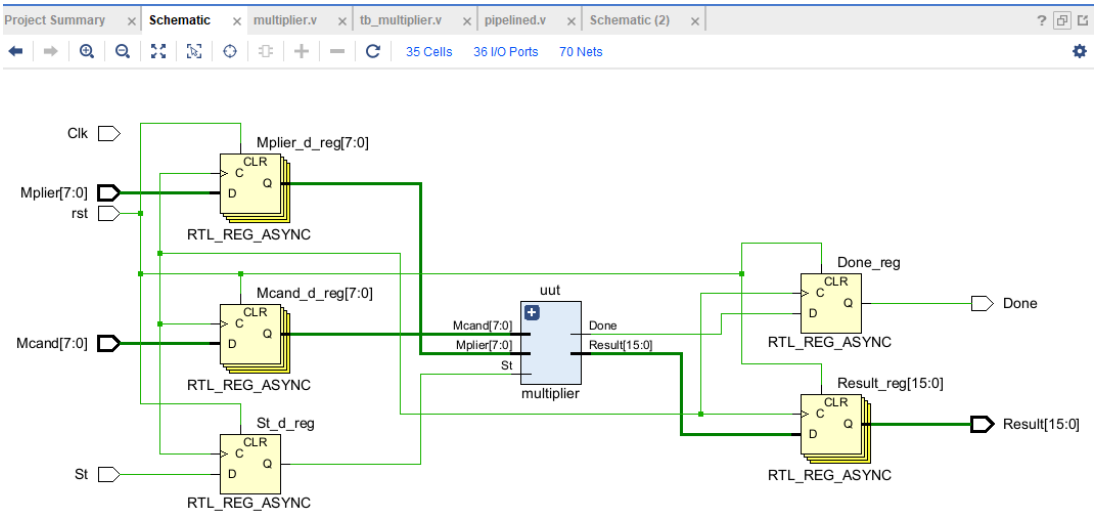
Implementation of 8 bit multiplier:

```
`define M ACC[0]
module multiplier (Clk, St, Mplier, Mcand, Done, Result);
 input Clk;
 input St;
 input[7:0] Mplier;
 input[7:0] Mcand;
 output Done;
 output[15:0] Result;
 reg[7:0] State;
 reg[16:0] ACC;
 initial
 begin
 State = 0;
 ACC = 0;
 end
 always @(posedge Clk)
 begin
 case (State)
 0 :
 begin
 if (St ==1'b1)
 begin
 ACC[16:8] <= 5'b00000 ;
 ACC[7:0] <= Mplier ;
 State <= 1 ;
 end
 end
 1, 3, 5, 7, 9, 11, 13, 15:
     begin
 if (`M == 1'b1)
 begin
 ACC[16:8] <= {1'b0, ACC[15:8]} + Mcand ;
 State <= State + 1 ;
 end
 else
 begin
 ACC <= {1'b0, ACC[16:1]} ;
State <= State + 2 ;
 end
 end
 2, 4, 6, 8, 10, 12, 14, 16 :
 begin
 ACC <= {1'b0, ACC[16:1]} ;
 State <= State + 1 ;
 end
 17 :
 begin
 State <= 0 ;
 end
 endcase
 end
 assign Done = (State == 17) ? 1'b1 : 1'b0 ;
 assign Result = (State == 17) ? ACC[15:0] : 16'b0101010100000 ;
endmodule
```



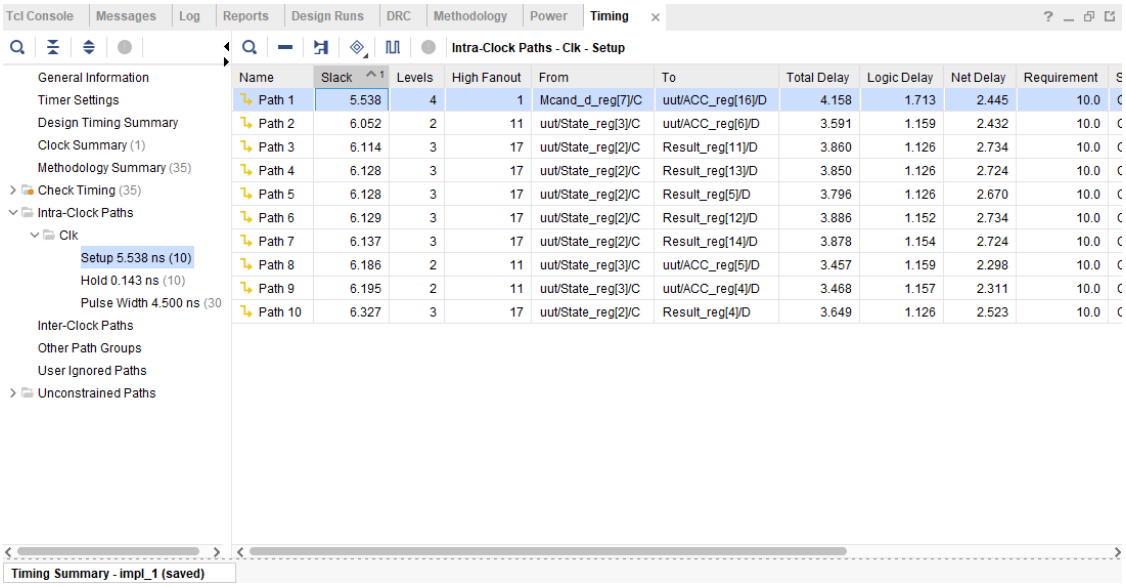
Implementation of 8 bit pipeline multiplier:

```
module pipelined(
   input Clk,
   input rst,
   input St,
   input [7:0]Mplier,
   input [7:0]Mcand,
   output reg Done,
   output reg [15:0]Result
   );
   reg St_d;
   reg [7:0]Mplier_d, Mcand_d;
   wire done_d;
   wire [15:0]result_d;
   multiplier uut(.Clk(Clk),.St(St_d),.Mplier(Mplier_d),.Mcand(Mcand_d),.Done(done_d),.Result(result_d));
   always @(posedge Clk or posedge rst) begin
   if(rst) begin
   St_d<=0;
   Mplier_d<=0;
   Mcand_d<=0;
   Done<=0;
   Result<=0;
   end
   else begin
   St_d<=St;
   Mplier_d<=Mplier;
   Mcand_d<=Mcand;
   Done<=done_d;
   Result<=result_d;
   end
   end
endmodule
```



Timing Summary :

There are total 10 setup timing paths and 10 hold timing paths. For the designed shown above the clock is kept constant that is 10ns.(**clock =10ns**)



Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	5.538	4	1	Mcand_d_reg[7]/C	uut/ACC_reg[16]/D	4.158	1.713	2.445	10.0
Path 2	6.052	2	11	uut/State_reg[3]/C	uut/ACC_reg[6]/D	3.591	1.159	2.432	10.0
Path 3	6.114	3	17	uut/State_reg[2]/C	Result_reg[11]/D	3.860	1.126	2.734	10.0
Path 4	6.128	3	17	uut/State_reg[2]/C	Result_reg[13]/D	3.850	1.126	2.724	10.0
Path 5	6.128	3	17	uut/State_reg[2]/C	Result_reg[5]/D	3.796	1.126	2.670	10.0
Path 6	6.129	3	17	uut/State_reg[2]/C	Result_reg[12]/D	3.886	1.152	2.734	10.0
Path 7	6.137	3	17	uut/State_reg[2]/C	Result_reg[14]/D	3.878	1.154	2.724	10.0
Path 8	6.186	2	11	uut/State_reg[3]/C	uut/ACC_reg[5]/D	3.457	1.159	2.298	10.0
Path 9	6.195	2	11	uut/State_reg[3]/C	uut/ACC_reg[4]/D	3.468	1.157	2.311	10.0
Path 10	6.327	3	17	uut/State_reg[2]/C	Result_reg[4]/D	3.649	1.126	2.523	10.0

1. Path-1 :

Source(Launch FF): Mcand_d_reg[7]/C
Destination (Capture FF): uut/ACC_reg[16]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.208ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 4.158ns.
Arrival time : 5.208 + 4.158 = **9.366ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **14.903ns.**

Slack = Required time - Arrival time
= 14.903 – 9.366
= **5.538.067ns**

2. **Path-2 :**

Source(Launch FF): uut/State_reg[3]/C
Destination (Capture FF): uut/ACC_reg[6]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.591ns.
Arrival time : 5.206 + 3.591 = **8.797ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **14.849ns.**

Slack = Required time - Arrival time
= 14.849 – 8.797
= **6.052ns**

3. **Path-3 :**

Source(Launch FF): uut/State_reg[2]/C
Destination (Capture FF): Result_reg[11]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.860ns.
Arrival time : 5.206 + 3.860 = **9.065ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **15.179ns.**

Slack = Required time - Arrival time
= 15.179 – 9.065
= **6.114ns**

4. **Path-4** :

Source(Launch FF): uut/State_reg[2]/C
Destination (Capture FF): Result_reg[13]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.850ns.
Arrival time : 5.206 + 3.850 = **9.055ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **15.183ns.**

Slack = Required time - Arrival time
= 15.183 – 9.055
= **6.128ns**

5. **Path-5** :

Source(Launch FF): uut/State_reg[2]/C
Destination (Capture FF): Result_reg[5]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.796ns.
Arrival time : 5.206 + 3.796 = **9.002ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **15.130ns.**

Slack = Required time - Arrival time
= 15.130 – 9.002
= **6.128ns**

6. **Path-6** :

Source(Launch FF): uut/State_reg[2]/C
Destination (Capture FF): Result_reg[12]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.886ns.
Arrival time : 5.206 + 3.886 = **9.091ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **15.220ns.**

Slack = Required time - Arrival time
= 15.220 – 9.091
= **6.129ns**

7. **Path-7** :

Source(Launch FF): uut/State_reg[2]/C
Destination (Capture FF): Result_reg[14]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.878ns.
Arrival time : 5.206 + 3.878 = **9.083ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **15.220ns.**

Slack = Required time - Arrival time
= 15.220 – 9.083
= **6.137ns**

8. **Path-8** :

Source(Launch FF): uut/State_reg[3]/C
Destination (Capture FF): uut/ACC_reg[4]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.457ns.
Arrival time : 5.206 + 3.457 = **8.662ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **14.848ns.**

Slack = Required time - Arrival time
= 14.848 – 8.662
= **6.186ns**

9. **Path-9** :

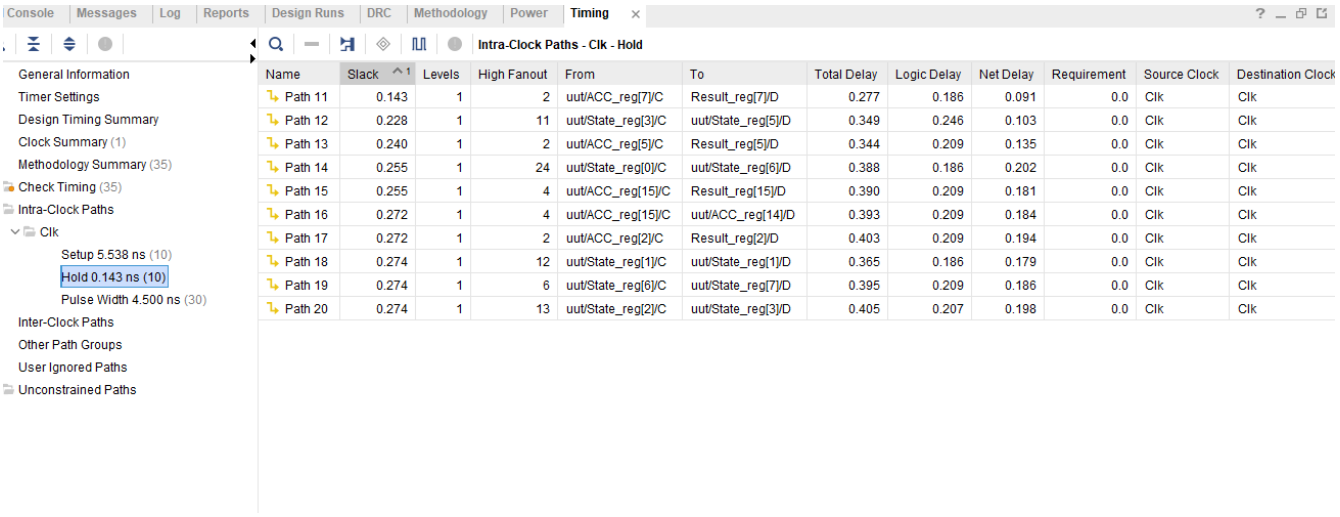
Source(Launch FF): uut/State_reg[3]/C
Destination (Capture FF): uut/ACC_reg[4]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.468ns.
Arrival time : 5.206 + 3.468 = **8.673ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **14.868ns.**

Slack = Required time - Arrival time
= 14.868 – 8.673
= **6.195ns**

10. Path-10 :

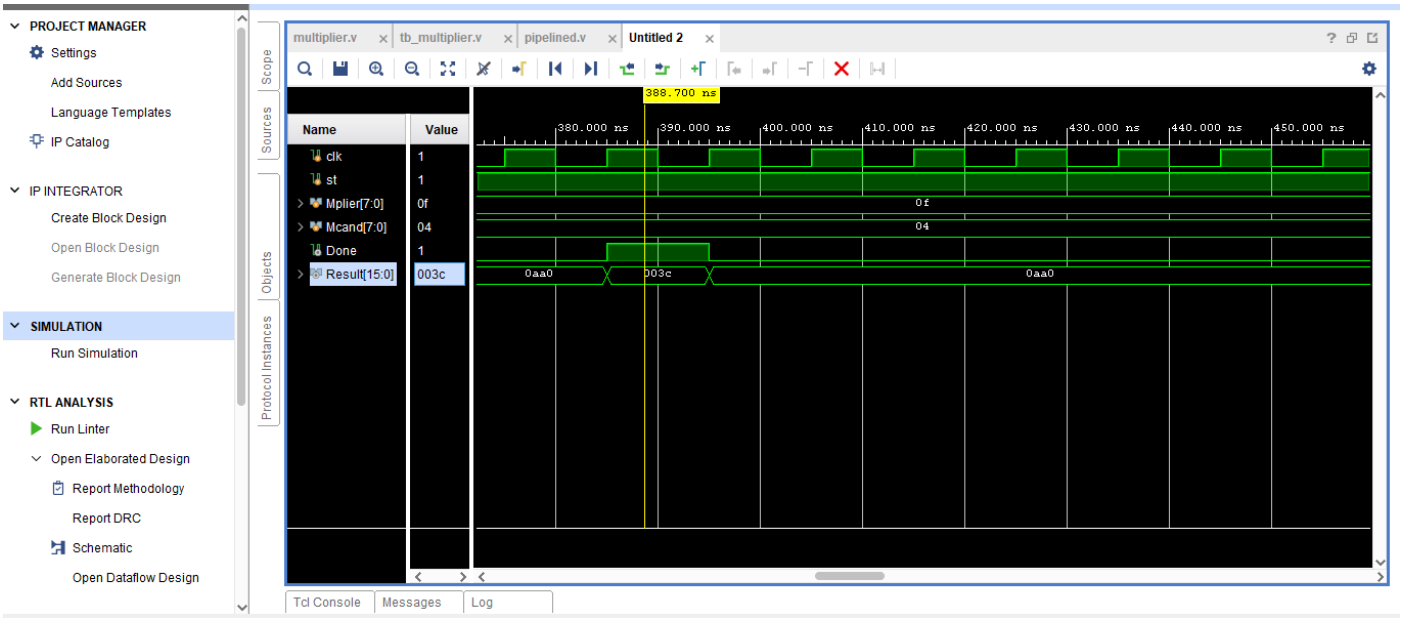
Source(Launch FF): uut/State_reg[2]/C
Destination (Capture FF): Result_reg[4]/D
Source clock path: clock clk rise edge + net + buffer + FDCE (Prop_fdce_C) = 5.206ns
Data path : FDCE (Prop_fdce_C_Q) + net + buffer + LUT + SU = 3.649ns.
Arrival time : 5.206 + 3.649 = **8.855ns.**
Required time : clock clk rise edge + net + buff + cp + cu = **15.182ns.**

Slack = Required time - Arrival time
= 15.182 – 8.855
= **6.327ns**



Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 11	0.143	1	2	uut/ACC_reg[7]/C	Result_reg[7]/D	0.277	0.186	0.091	0.0	Clk	Clk
Path 12	0.228	1	11	uut/State_reg[3]/C	uut/State_reg[5]/D	0.349	0.246	0.103	0.0	Clk	Clk
Path 13	0.240	1	2	uut/ACC_reg[5]/C	Result_reg[5]/D	0.344	0.209	0.135	0.0	Clk	Clk
Path 14	0.255	1	24	uut/State_reg[0]/C	uut/State_reg[6]/D	0.388	0.186	0.202	0.0	Clk	Clk
Path 15	0.255	1	4	uut/ACC_reg[15]/C	Result_reg[15]/D	0.390	0.209	0.181	0.0	Clk	Clk
Path 16	0.272	1	4	uut/ACC_reg[15]/C	uut/ACC_reg[14]/D	0.393	0.209	0.184	0.0	Clk	Clk
Path 17	0.272	1	2	uut/ACC_reg[2]/C	Result_reg[2]/D	0.403	0.209	0.194	0.0	Clk	Clk
Path 18	0.274	1	12	uut/State_reg[1]/C	uut/State_reg[1]/D	0.365	0.186	0.179	0.0	Clk	Clk
Path 19	0.274	1	6	uut/State_reg[6]/C	uut/State_reg[7]/D	0.395	0.209	0.186	0.0	Clk	Clk
Path 20	0.274	1	13	uut/State_reg[2]/C	uut/State_reg[3]/D	0.405	0.207	0.198	0.0	Clk	Clk

Waveform:



Conclusion :

The goal of the project, to design an 8-bit multiplier using the add and shift method, was achieved. The multiplier was designed, coded in VHDL and simulated using the Xilinx Vivado tools. As an added value to the project, several designs were implemented in order to compare speed and area. The design can also expanded to a 32-bit version .

References:

1. Digital Systems Design Using Verilog Charles H. Roth, Jr., Lizy Kurian John, and Byeong Kil Lee
2. Digital Systems Design A.P. Godse , Dr.D.A.Godse
3. https://users.utcluj.ro/~baruch/book_ssce/SSCE-Shift-Mult.pdf