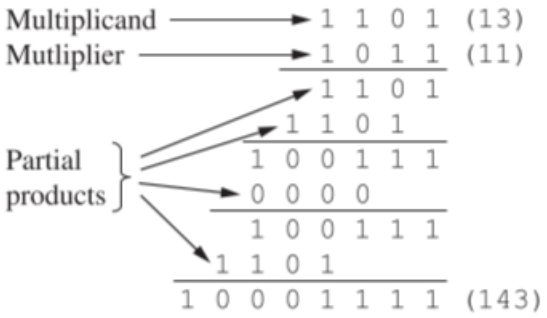
Multiplier

A multiplier is a digital circuit that calculates the product of two binary numbers. The process of multiplication is nothing but repeated addition of the given binary numbers. When we perform the product A * B, the first operand (A) is called the multiplicand and the second operand (B) is called the multiplier.



The simplest form of multiplication requires shifting and addition. From the above example we can say that each partial product is either the multiplicand (1101) shifted over by the appropriate number of places or zero. There are different types of multipliers that can be designed and used, they have their differences in their speed, area, complexity, etc.

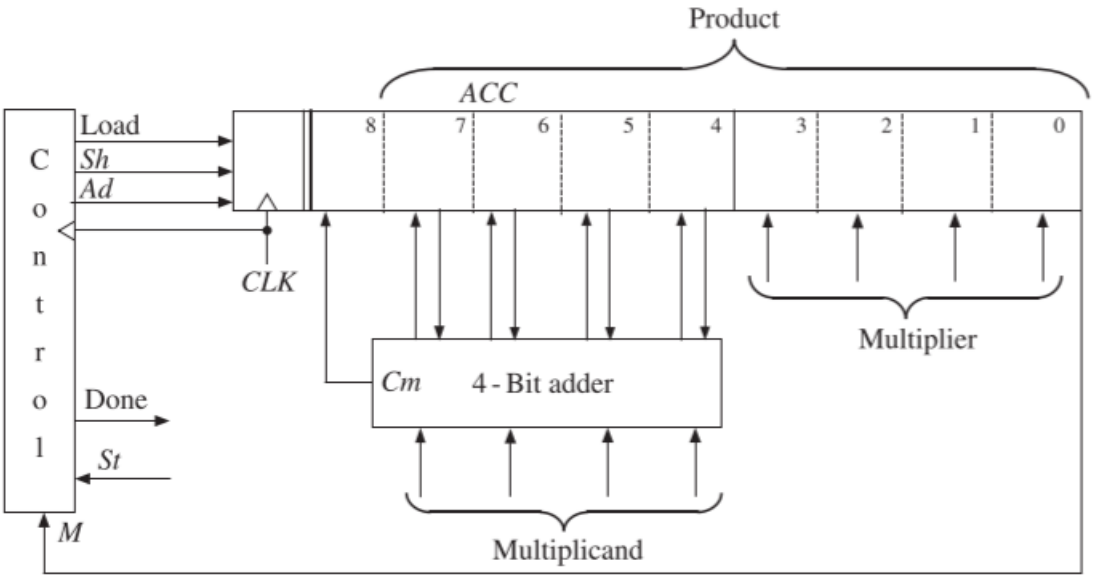
Types of adders:

i) Shift-and-add Multiplier
ii) Array multiplier

i) **Shift-and-Add Multiplier:**

In this method each partial product is either the multiplicand 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.

If the multiplicand is shifted to left each time, an 8-bit adder will be required for an 4-bit multiplier where as if we shift the multiplicand to right we required 4-bit adder only.



As mentioned before we are using 4-bit adder only and the inputs to the adder are 4bit multiplicand and upper 4-bit from the accumulator and the sum and carry is again sent back to the accumulator.

Whenever the first bit of the accumulator, that is the first bit of the multiplier (M) is '1' an add signal is generated. At this time the multiplicand is added to the upper 4bits of accumulator and stored in it. After the completion of the addition a shift signal is generated, at this time the bits in the accumulator are shifted to right by one bit. This process continues till it shifts 4-bits and for N bit multiplier till it shifts N-bits.

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	(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)

dividing line between product and multiplier

Above example describes the process of multiplying 11 and 13 and the result is 143

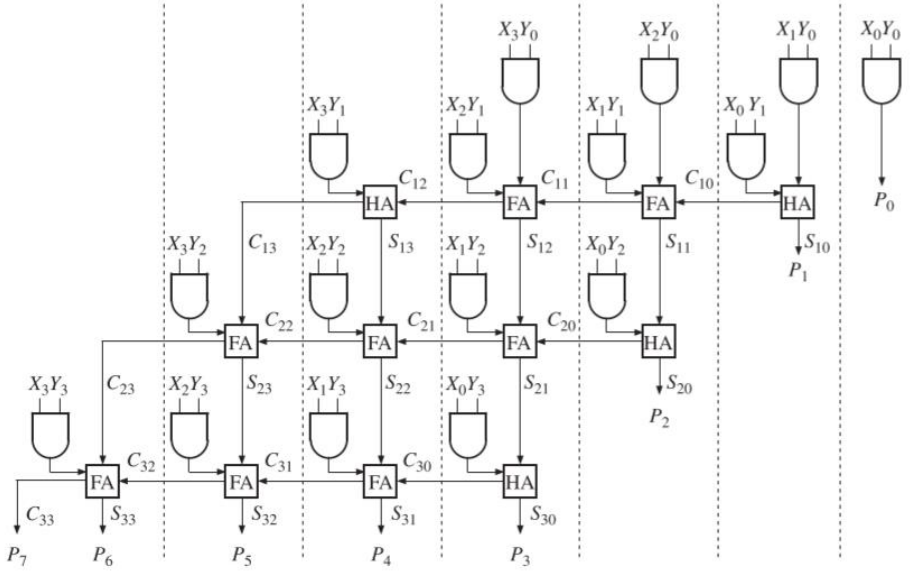
Improvements: As mentioned above we are shifting the bits to right, we can also shift to left but we are not doing it because when shift the bits to left we will be requiring more number of adder due to this the data path increases and the slack value is decrease.

ii) Array multiplier:

An array multiplier is a parallel multiplier that generates the partial products in a parallel fashion. The various partial products are added as soon as they are available. Let's take two 4-bit unsigned numbers $X_3X_2X_1X_0$ and$Y_3Y_2Y_1Y_0$, they are multiplied to generate a product of 8bit.

			X_3	X_2	X_1	X_0		Multiplicand
			Y_3	Y_2	Y_1	Y_0		Multiplier
			X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0		partial product 0
		X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1			partial product 1
		C_{12}	C_{11}	C_{10}				1st row carries
	C_{13}	S_{13}	S_{12}	S_{11}	S_{10}			1st row sums
	X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2				partial product 2
	C_{22}	C_{21}	C_{20}					2nd row carries
C_{23}	S_{23}	S_{22}	S_{21}	S_{20}				2nd row sums
X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3					partial product 3
C_{32}	C_{31}	C_{30}						3rd row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				3rd row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	final product

Here each partial product X_iY_i is generated by performing AND operation of $X_i\ and\ Y_i$. Each partial product can be added to the previous sum of partial products using a row of adders. If there are two inputs to the adder then we use Half adder and if there are three inputs we use Full adders.



Above diagram shows the array of AND gates and adders to perform multiplication.

For an N bit multiplier we need N^2 - AND gates

N (N-2) - FULL adders

N - HALF adders.

The worst-case time to complete the multiplication of the above mentioned circuit will be $8t_{ad} + t_g$ where t_{ad} is the adder delay and t_g is the AND gate delay. For the N-bit the worst case delay will be $(3N - 4)t_{ad} + t_g$. When N<=4 the delay will not change but for higher values of N the delay can be **improved to $2Nt_{ad} + t_g$ by forwarding carry from each adder to the diagonally lower adder rather than the adder on the left side. The delay can also be decreased by using Carry Look-Ahead adders instead of normal half or full adders. For smaller value of N there may not be a significant change in the delay but when the N increases the delay value of CLA will increase logarithmically, while the array multiplier's delay grows linearly.**

Comparison of array multiplier over shift and add multiplier:

1) Shift-and-add multiplier:

a) **Working principle:** of shift and add multiplier is simple which is, we shift the partial product and add them to form the product, array multiplier they us parallelism in which bits of multiplier and multiplicand are operated at a time.

b) **Operation steps:** Generation of partial products is done by shifting the multiplier bit by bit and these partial products are accumulated by addition to produce the final result.

c) **Speed:** Shift-and-Add Multipliers are relatively slow, especially for large operands, for N-bit multiplier this design requires 2n clocks to complete the multiply in the worst case.

d) **Circuit Complexity:** They have a relatively simple and straightforward hardware implementation, making them suitable for low-speed and low-power applications.

2) Array multiplier:

a) **Working principle:** they us parallelism in which bits of multiplier and multiplicand are operated at a time.

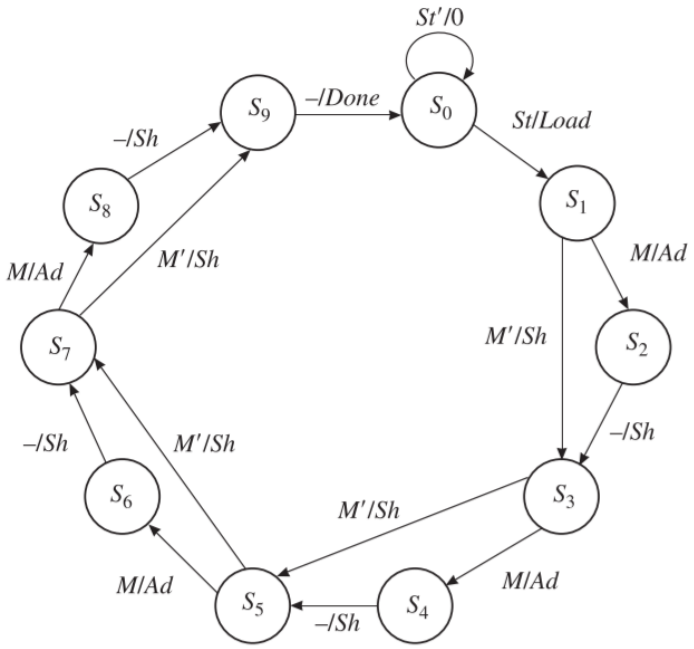
b) **Operation steps:** Each bit of the multiplier is processed in parallel, generating partial products for all bits simultaneously. The partial products are then aligned and added together to produce the final result.

c) **Speed:** They are significantly faster than Shift-and-Add Multipliers, especially for large operands. They provide high-speed multiplication due to parallel processing. For N-bit multipliers this design requires $(3N - 4)t_{ag} + t_g$ delay.

d) **Circuit Complexity:** They are more complex in terms of hardware implementation due to the parallel processing and the need for carry-look-ahead logic. The added complexity results in faster multiplication times but may consume more power and require larger chip area.

Implementation of 8-bit Shift-and-add multiplier

In the implementation of 8-bit shift-and-add multiplier, a control circuit must be designed to output the proper sequence of add and shift signals for this purpose I used FSM.



Above state graph is for 4-bit multiplier which is similar to 8-bit multiplier.

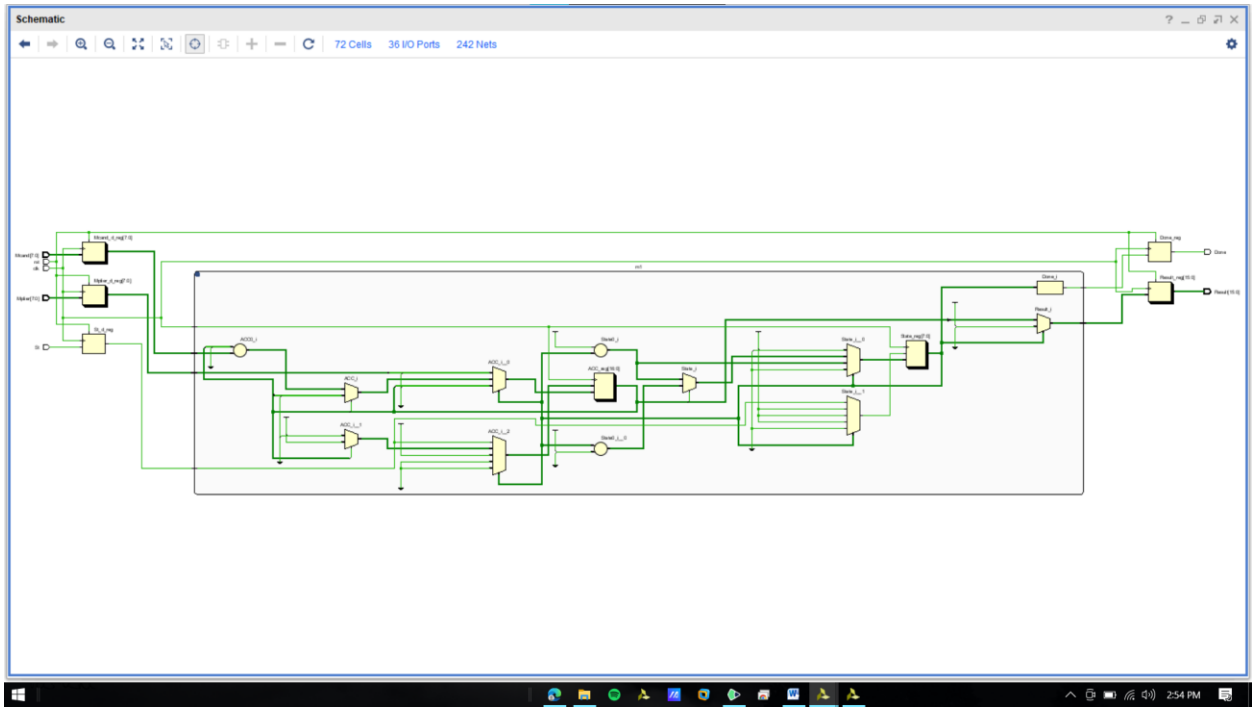
Where S_0 is the reset state and the circuit stays in S0 until a start signal is received. This generates a Load signal, which loads the multiplier into the lower 4 bits of the accumulator (ACC) while clearing the upper 5 bits of the accumulator.

In state S_1, the LSB bit of multiplier (M[0]) is tested. If M[0] =1, an add signal is generated and if M[0]=0, a shift signal is generated. Stages S_3, S_5, S_7, S_9, S_{11}, S_{13} and S_{15} all this states have the same operation as S_1.

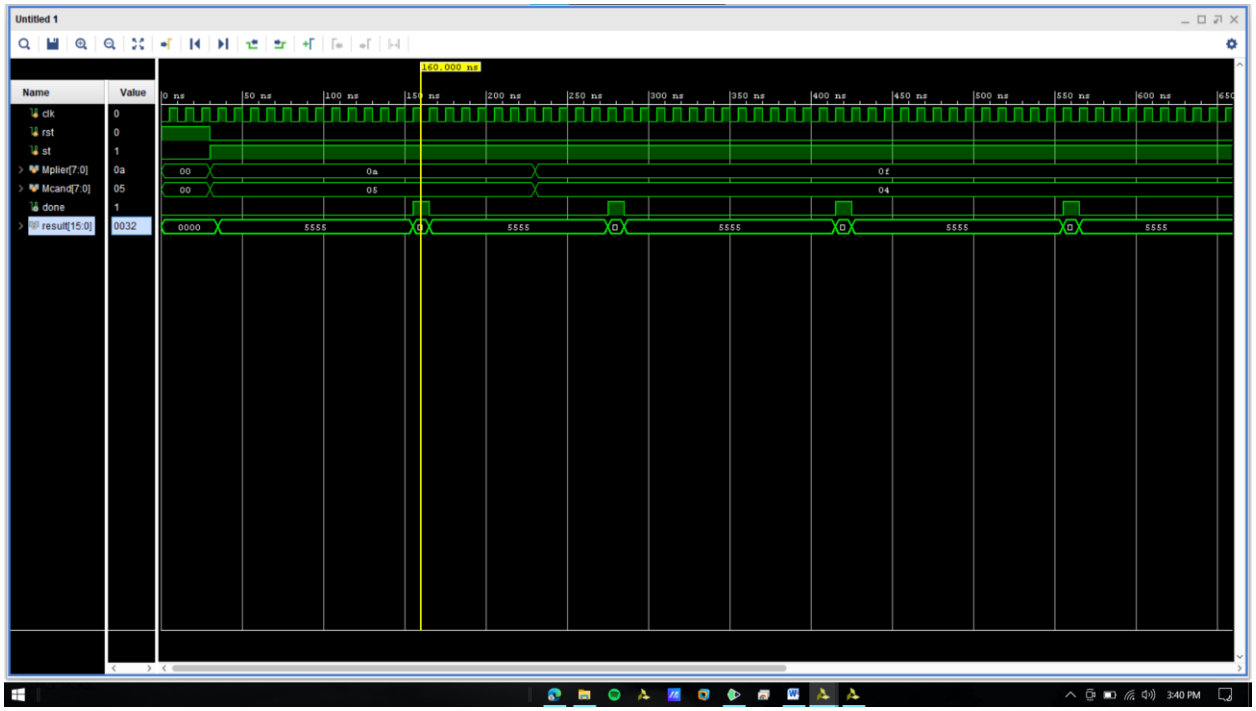
A shift signal is always generated at the next clock time following an add signal (states S_2, S_4, S_6, S_8, S_{10}, S_{12}, S_{14}, S_{16}).

After generating eight shifts, the control network transitions to S_{17} and generates a "done" signal before returning to S_0.

1. Schematic:



2. Waveform:



The output is see at 12th positive clock edge after the input is changed

3. Timing report:

The clock used in this design has a time period of 10ns. There are a total of 20 timing paths in this design, comprising 10 setup timing paths and 10 hold timing paths. The following table lists 10 setup timing paths, including their source flip-flops, destination flip-flops, and respective slack values.

Name	Slack ^1	Levels	High Fanout	From	To
Path 1	6.815	1	36	m1/State_reg[0]/C	m1/ACC_reg[0]/CE
Path 2	6.815	1	36	m1/State_reg[0]/C	m1/ACC_reg[1]/CE
Path 3	6.815	1	36	m1/State_reg[0]/C	m1/ACC_reg[2]/CE
Path 4	6.815	1	36	m1/State_reg[0]/C	m1/ACC_reg[3]/CE
Path 5	6.815	1	36	m1/State_reg[0]/C	m1/ACC_reg[4]/CE
Path 6	6.815	1	36	m1/State_reg[0]/C	m1/ACC_reg[5]/CE
Path 7	6.815	1	36	m1/State_reg[0]/C	m1/ACC_reg[6]/CE
Path 8	6.815	1	36	m1/State_reg[0]/C	m1/ACC_reg[7]/CE
Path 9	6.913	4	4	m1/ACC_reg[9]/C	m1/ACC_reg[13]/D
Path 10	6.957	4	1	Mcand_d_reg[6]/C	m1/ACC_reg[16]/D

As mentioned above there are different values of slacks for different paths. The path which is having the lowest value of slack will be the critical path, because the value of slack is the time that is left out with us and if this time is less it means that the time taken by this path is more which means it the critical path.

3.1. Critical path:

Source : m1/State_reg[0]/C

Destination : m1/ACC_reg[0]/CE

Source clock path : clock rise edge(0ns) + net(0ns) + IBUF(0.938ns) + net(1.972ns) + BUFG(0.096ns) + net(1.627ns) **=4.633ns**

Data path : FDRE(0.456ns) + net(1.869ns) + LUT5(0.124ns) + net(0.474ns) **=2.923ns**

Destination clock path : clock rise edge(10ns) + net(0ns) + IBUF(0.805ns) + net(1.868ns) + BUFG(0.091ns) + net(1.511ns) + clock pessimism(0.336ns) + clock uncertainty(-0.035ns) + FDRE(-0.205ns) **=14.371ns**

Arrival Time = source clock path + Data path
= 4.633 + 2.923
= 7.556ns.
Required Time = Destination clock path
= 14.371ns.
Slack = Required time – Arrival time
= 14.371 – 7.556
= 6.815ns.

3.2. Maximum frequency of operation

To find the maximum frequency of operation we need to know the minimum time required by this circuit to produce the output, which can nothing but the total delay in the critical path.

We know time period and the value of slack for the critical path, using these both we can calculate minimum time required and the formula is

Min time = time period – slack

Min time = 10 – 6.815 = 3.185ns

Max frequency of operation = $\frac{1}{\min time\ required} = \frac{1}{3.185*10^{-9}}$ **= 313.9MHz**

Name	Slack ^1	Levels	High Fanout	From	To
Path 11	0.197	1	36	m1/State_reg[0]/C	Result_reg[15]/D
Path 12	0.229	1	2	m1/ACC_reg[1]/C	Result_reg[1]/D
Path 13	0.231	1	2	m1/ACC_reg[2]/C	Result_reg[2]/D
Path 14	0.238	1	2	m1/ACC_reg[5]/C	m1/ACC_reg[4]/D
Path 15	0.250	1	1	m1/ACC_reg[16]/C	m1/ACC_reg[15]/D
Path 16	0.253	1	4	m1/ACC_reg[12]/C	Result_reg[12]/D
Path 17	0.257	1	2	m1/ACC_reg[3]/C	m1/ACC_reg[2]/D
Path 18	0.276	1	36	m1/State_reg[0]/C	Result_reg[14]/D
Path 19	0.277	1	27	m1/State_reg[4]/C	m1/State_reg[0]/D
Path 20	0.279	1	36	m1/State_reg[0]/C	Result_reg[4]/D

Implementation of 8-bit CLA Array Multiplier:

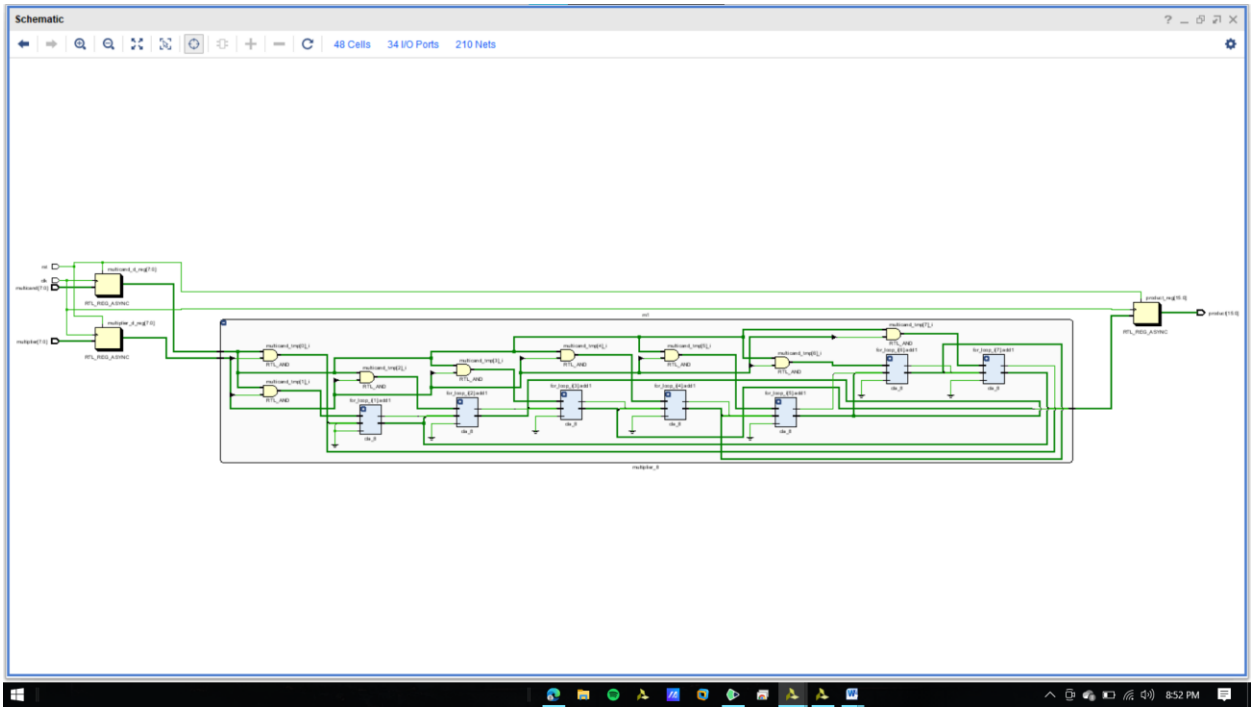
In the implementation of 8bit CLA multiplier, I have first designed 8-bit carry look ahead adder.

Then I have generated particle products for the 8bit inputs.

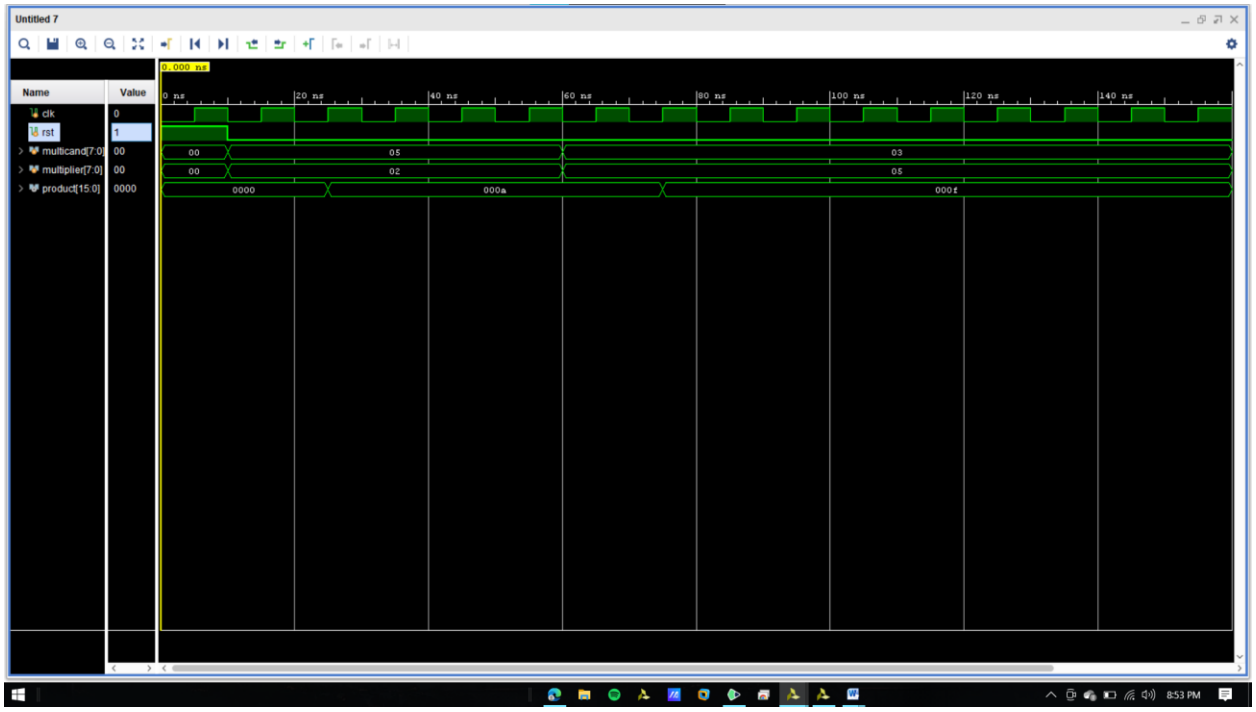
These particle products are feed to 8-bit carry look ahead adder.

Outputs of this adders are brought together to form final multiplied product which is a 16-bit product.

1. Schematic:



2. **Waveform:**



The output is produced after one clock pulse.

3. Timing Report:

The clock used in this design has a time period of 10ns.

The total number of timing paths in this design is 20, in which 10 paths are setup timing paths and another 10 paths are hold timing paths.

The below table 10 setup timing paths, it contain source flip flop, destination flip flop and slack value of that respective path.

Name	Slack ^1	Levels	High Fanout	From	To
Path 1	0.073	12	19	multiplier_d_reg[1]/C	product_reg[13]/D
Path 2	0.121	12	19	multiplier_d_reg[1]/C	product_reg[14]/D
Path 3	0.188	12	19	multiplier_d_reg[1]/C	product_reg[15]/D
Path 4	0.405	11	19	multiplier_d_reg[1]/C	product_reg[12]/D
Path 5	1.237	11	19	multiplier_d_reg[1]/C	product_reg[11]/D
Path 6	1.668	10	19	multiplier_d_reg[1]/C	product_reg[10]/D
Path 7	1.877	10	19	multiplier_d_reg[1]/C	product_reg[9]/D
Path 8	2.453	9	19	multiplier_d_reg[1]/C	product_reg[8]/D
Path 9	2.662	7	19	multiplier_d_reg[1]/C	product_reg[6]/D
Path 10	3.273	8	19	multiplier_d_reg[1]/C	product_reg[7]/D

As mentioned previously, various paths have different slack values. The path with the lowest slack value is considered the critical path. A lower slack value indicates that less time is available, suggesting that this path consumes more time, making it the critical path.

3.1. Critical path:

Source :- multiplier_d_reg[13]/D

Destination:- product_reg[13]/D

Source clock path :- clock rise edge + net(0ns) + buffer(0.938ns) + net(1.972ns) + BUFG(prop_bufg_i_O)(0.096ns) + net(1.639ns) =**4.645ns**

Data path:- FDCE(prop_fdce_C_Q)(0.456ns) + LUT5(0.124ns) + LUT5(0.15ns) + LUT6(0.326ns) + LUT6(0.124ns) + LUT5(0.15) + LUT6(0.326ns) + LUT6(0.124ns) + LUT6(0.124ns) + LUT4(0.15ns) + LUT5(0.326ns) + LUT4(0.124ns) + LUT4(0.124ns) + net(0ns) = **9.880ns**

Destination clock path:- clock rise edge(10ns) + net(0ns) + buffer(0.805ns) + net(1.868ns) + BUFG(0.091ns) + net(1.517ns) + clock pessimism(0.322ns) + clock uncertainty(-0.035ns) + FDCE(0.031ns) = **14.599ns**

Arrival Time = source clock path + Data path
= 4.645 + 9.88
= 14.525ns.

Required Time = Destination clock path
= 14.599ns.

Slack = Required time – Arrival time
= 14.599 – 14.525
= 0.073ns.

3.2. Maximum frequency of operation

To find the maximum frequency of operation we need to know the minimum time required by this circuit to produce the output, which can nothing but the total delay in the critical path.

We know time period and the value of slack for the critical path, using these both we can calculate minimum time required and the formula is

Min time = time period – slack

Min time = 10 – 0.073 = 9.927ns

Max frequency of operation = $\frac{\mathbf{1}}{\mathbf{min}\ \boldsymbol{time\ required}} = \frac{\mathbf{1}}{\mathbf{9.927 * 10^{-9}}}$ =100.7**MHz**

Name	Slack ^1	Levels	High Fanout	From	To
Path 11	0.418	1	28	multicand_d_reg[0]/C	product_reg[2]/D
Path 12	0.445	1	21	multiplier_d_reg[3]/C	product_reg[3]/D
Path 13	0.447	1	17	multiplier_d_reg[7]/C	product_reg[8]/D
Path 14	0.447	1	26	multicand_d_reg[6]/C	product_reg[15]/D
Path 15	0.457	1	17	multiplier_d_reg[7]/C	product_reg[7]/D
Path 16	0.513	1	17	multiplier_d_reg[7]/C	product_reg[12]/D
Path 17	0.518	1	27	multicand_d_reg[3]/C	product_reg[10]/D
Path 18	0.536	1	17	multiplier_d_reg[7]/C	product_reg[9]/D
Path 19	0.550	1	24	multiplier_d_reg[4]/C	product_reg[4]/D
Path 20	0.569	1	17	multiplier_d_reg[7]/C	product_reg[14]/D

Improvements in speed:

- **Pipelined**: Speed can be improved by implementing pipeline stages in the multiplier to break down the multiplication into multiple stages. This reduces the critical path length and allows for higher clock frequencies. Their frequencies should be in the ration

$$\frac{\textbf{frequenc of normal}}{\textbf{frequenc of pipelined}} = \frac{\textbf{no of clocks for result of normal}}{\textbf{no of clocks for result for pipelined}}$$

- **Hierarchical Design:** Use a hierarchical structure, where smaller blocks are combined to form a larger multiplier. This reduces the complexity of each block and can improve performance.

Application of Multipliers:

1. **Arithmetic Operations:** Multipliers are fundamental for performing multiplication operations in digital arithmetic, which are required for a wide range of applications.
2. **Digital Signal Processing (DSP):** In DSP systems, multipliers are extensively used for operations like filtering, convolution, correlation, and Fourier transformations.
3. **Graphics Processing Units (GPUs):** GPUs are designed to perform parallel processing and are equipped with a large number of multipliers. Multipliers in GPUs are used for tasks like pixel shading, texture mapping, and 3D graphics rendering.
4. **Microcontrollers and Microprocessors:** Multipliers are included in the instruction set of microcontrollers and microprocessors, enabling them to perform efficient arithmetic calculations. They are used for various tasks, including data processing, control algorithms, and mathematical computations.
5. **Wireless Communications:** Multipliers play a critical role in wireless communication systems, including modulation and demodulation. They are used in algorithms for encoding and decoding digital signals.
6. **Scientific Computing:** In scientific simulations and computations, multipliers are used for numerical analysis, solving differential equations, and complex mathematical modeling.
7. **Cryptography:** Cryptographic algorithms often involve large integer arithmetic, where multipliers are used for encryption and decryption. Public-key encryption algorithms like RSA heavily rely on efficient modular multiplication.
8. **Artificial Intelligence (AI):** In machine learning and neural networks, multipliers are used in the training and execution of deep learning models, which involve numerous matrix multiplications.
9. **Image and Video Processing:** Image and video processing applications require multipliers for tasks like image enhancement, compression, and pattern recognition.

10. **Control Systems:** In control systems, multipliers are used for modeling, simulation, and real-time control algorithms. They help implement proportional-integral-derivative (PID) controllers and other control strategies.

Conclusion:

In this report, I have explained what are multipliers, what are their types and implementation of 8-bit Shift-and-add and 8-bit carry look ahead array multipliers. Propagation delay of the shift-and-add in more than the propagation delay of the carry look ahead adder. The complexity of the circuit of the CLA array multiplier is high and the area required is also more, whereas for shift-and-add multiplier the complexity is less and the area required is also less.
The value of slack of shift-and-add is more because they are more number of registers along the path from the input to output, where the slack value of the CLA array multiplier is less because there are no registers in between input and output ports.

References:

- Digital system design using Verilog by Charles Roth
- Fpga4students : Verilog Code for Multiplier using Carry-Look-Ahead Adders - FPGA4student.com
- ChatGpt