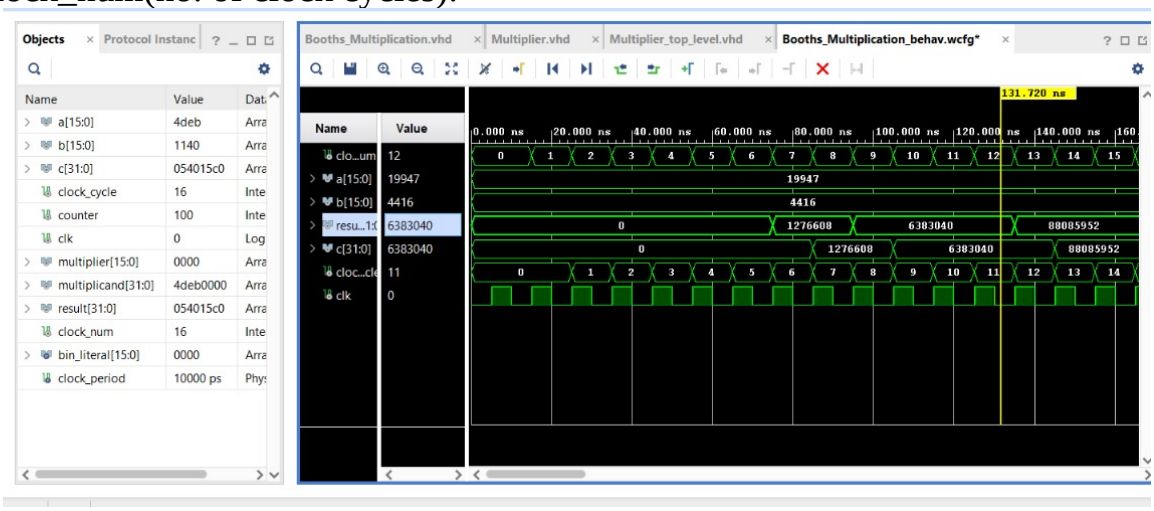
COL216

Fixed Latency Multiplication

This is a fairly algorithm to multiply two unsigned binary numbers. We declare a standard logic vector called result and initialize it to the 32 bit unsigned representation of '*0*'. We concatenate a 16 bit unsigned representatiion of '*0*' to the multiplicand. This prevents overflow during lef shift of multiplicand. We run a loop for each digit of the multiplier. If the current bit is *'1'*,we add the multiplicand to the current result and update the result. Then we perform left shift on the multiplicand. In case if the current bit is *'0'*, we only perform the left shift. The result is the product of the input numbers.

The design :-
a,b : std_logic_vector(15 downto 0) – inputs
c : std_logic_vector(31 downto 0) – output
clock_cycle : integer – output(no of clock cycles required)
Intermediates :-
result : std_logic_vector(31 downto 0) – temporary calculations (intially "00000000000000000000000000000000")
multiplier : std_logic_vector(15 downto 0) – stores the multiplier
multiplicand : std_logic_vector(31 downto 0) – stores the multiplicand
clock_num : integer – stores the number of performed cycles
counter : integer – number of bits covered

The following screenshot shows 500 test-cases being run and their respective clock_num(no. of clock cycles):



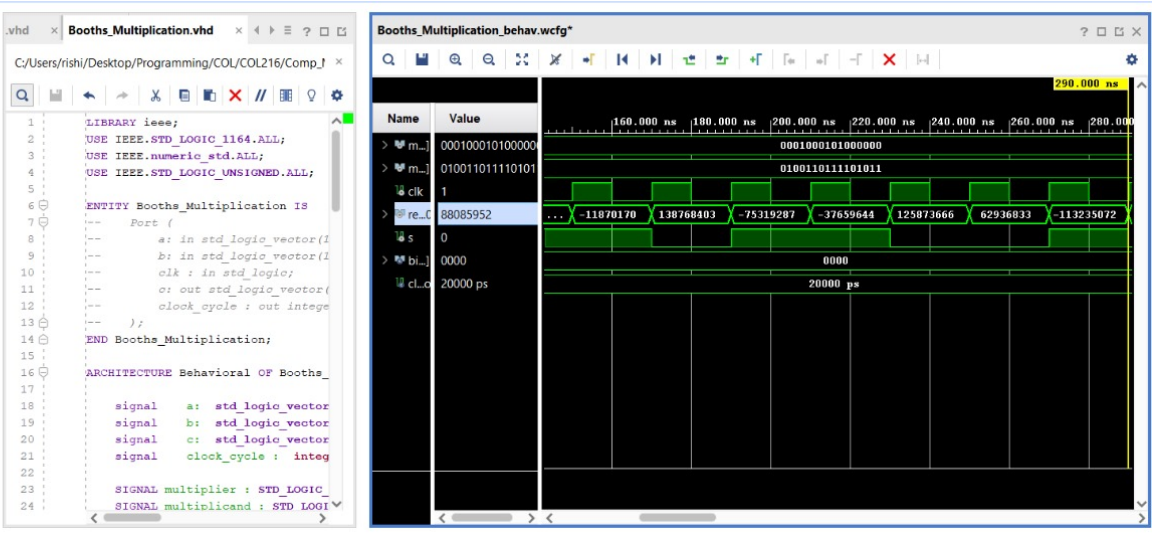
Variable Latency Multiplier

This program is an improvement over the classic fixed latency multiplier. We use the Booth's algorithm to calculate the product of two signed binary numbers. We initialize the multiplier and multiplicand. We declare a signal s which represents the value multipler[-1]. We initialize s to *'0'*. Now we check s and multiplier[0] and do the following according to different cases :-

Multiplier[0]	Multiplier[-1]	Action
0	0	Right shift the multiplier
0	1	Add multiplicand to current product and right shift the multiplier
1	0	Subtract multiplicand from current product and right shift the multiplier
1	1	Right shift the multiplier

The design :-
a,b : std_logic_vector(15 downto 0) – inputs
c : std_logic_vector(31 downto 0) – output
clock_cycle : integer – output(no of clock cycles required)
Intermediates :-
arr : std_logic_vector(31 downto 0) – temporary calculations (intially "0000000000000000" & a)
s : std_logic – multiplier[-1]
counter : integer – number of bits covered
clock_num : integer – stores the number of performed cycles

The following screenshot shows 500 test-cases being run and their respective clock_num(no. of clock cycles):



In our testing, we found some test-cases, in which fixed latency performs better. Some of them are:-
"1010101010101010", "0101010101010101"
"0010010010010010", "1001001001001001"
"1010010100101001", "0100101001010010"
"1010100101010010", "1001010101001001"
"0101010101010101", "1010010100101001"
"1011100100101001", "1000001110001101"
"0101101010010101", "1100000110000001"
"0100110101100101", "0011011110111001"
"1011010110101001", "1111011011100010"
"1001011011010101", "1010011000100111"
"0011010101010101", "1010001001100000"
"1011010101100101", "0111101101100110"
"1010110010101011", "0111011100010011"

As per the overall testing, the variavle latency approach seems to be noticibly faster. This can also be seen in some screenshots attached.

Rishit Jakharia
2022CS11621
Aditya jha
2022CS11102