Indian Institute of Information Technology, Guwahati

Electronics and Communication Department

Implementation of 16-Bit Kogge-Stone Adder

Ajit Singh

Contents

Problem Statement

Design and implement a 16-Bit Kogge Stone Adder in Verilog using Vivado Tool (Xilinx).

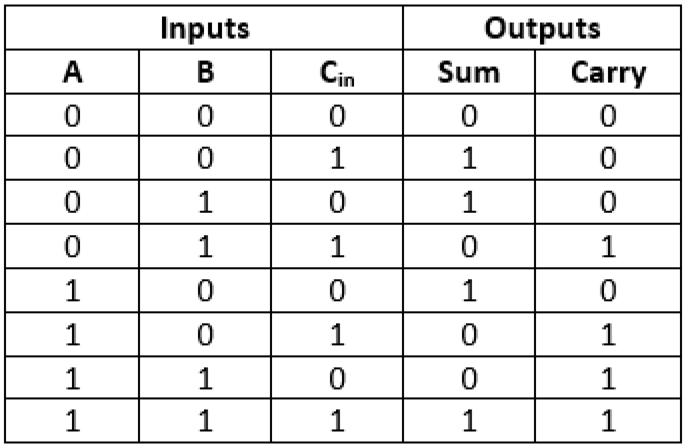
Need for Fast Adders:

2.1 Introduction

An adder is a digital circuit used in addition of 2 binary numbers. Adders are the most basic components of Arithmetic Logic Units and hence are at the core of almost all mathematical computations in computers and/or processors. Modern day computing requires number-crunching with large number of bits, like 64, 128, and so on. Hence designing adders which are quite fast with minimum time elapsed during and between computations is the key for powerful and efficient computing. The result of any adder involves a sum vector and a carry output. The challenge in designing adders is to limit the time delay involved in calculating the carry output. General adders like the basic Ripple Carry Adders compute carry with O(N) time delay. Thus, with increasing number of bits (N), there is a linear increase in time delay involved in carry computation (propagation delay). Thus, speed gets impacted terribly while performing high bit additions and to improve speed performance (i.e., reduce the propagation delay to the minimum), many different approaches are taken to calculate the carry output rather than just the linear propagation (Rippling) of carry bits from the LSB of Sum Vector to its MSB. One such family of fast performing adders are parallel prefix carry look ahead adders, of which we are studying Kogge-Stone adders. These adders sacrifice in terms of area and cost (more power consumed, greater complexity of circuit and complex wiring) to attain best speed for calculating carry out i.e., in O(log(N)) time delay.

2.2 Full Adder and Ripple Carry Adders

A typical full adder takes in three inputs, two numbers which have to be added together, and a carry-in which is generally zero (unless the adder is at an intermediate stage of many linked adders in which case carry-in may be 1 or 0 depending on the previous adder's carry out). The Truth Table (logic values) of a full adder is given by:



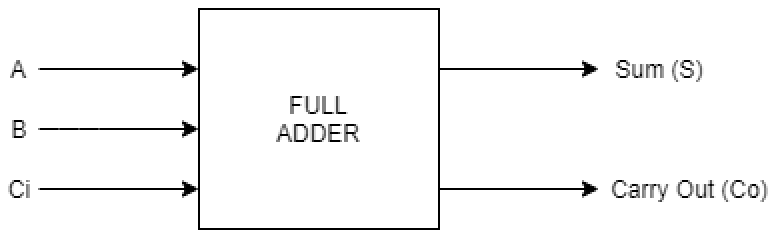
Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

From which we come to a simple conclusion that; the sum output is clearly:

$$S = A \textit{ xor } B \textit{ xor } C_{out}$$

And the carry output can be expressed as:

$$C_{out} = (A \textit{ and } B) \textit{ or } (C_{in} \textit{ and } (A \textit{ xor } B))$$



$$S = A \oplus B \oplus Cin$$

$$Cout = A.B + Cin.(A \oplus B)$$

Figure: Working of a Full Adder

A typical adder used for adding a greater number of bits together is a ripple-carry adder (RCA). In an RCA, for Calculating Sum at a particular bit position (S_i), we need A_i, B_i as well as the previous stage carry C_{i-1}. To calculate Carry out for a stage, i.e., C_i (Which is the input carry for next stage) we require to calculate the current sum which in turn cannot be calculated until and unless previous carry is fed into it. Thus, we can see that this system doesn't seem to be much effective in terms of speed.

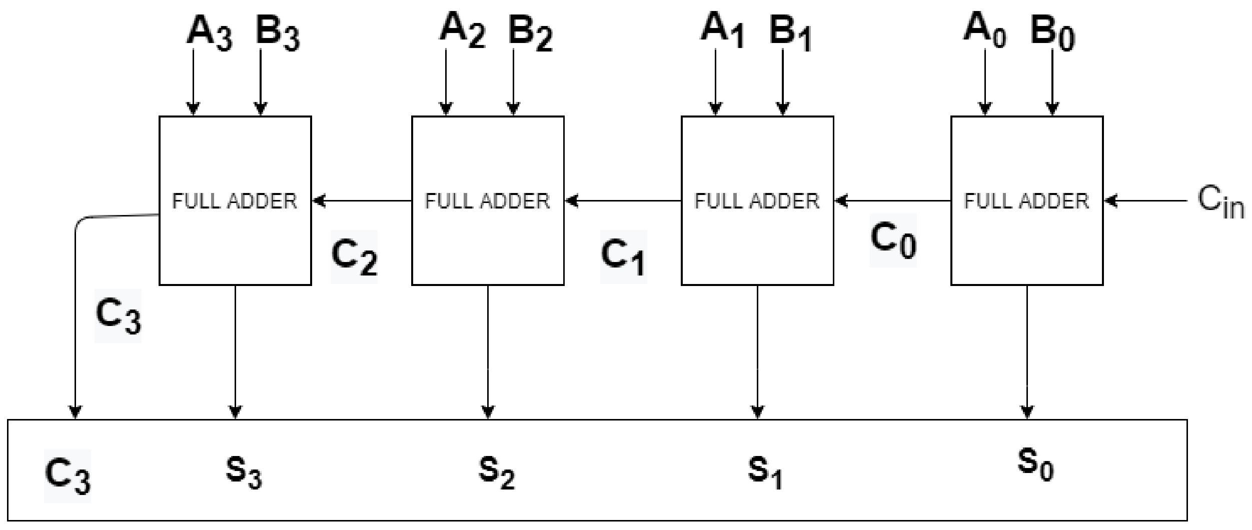
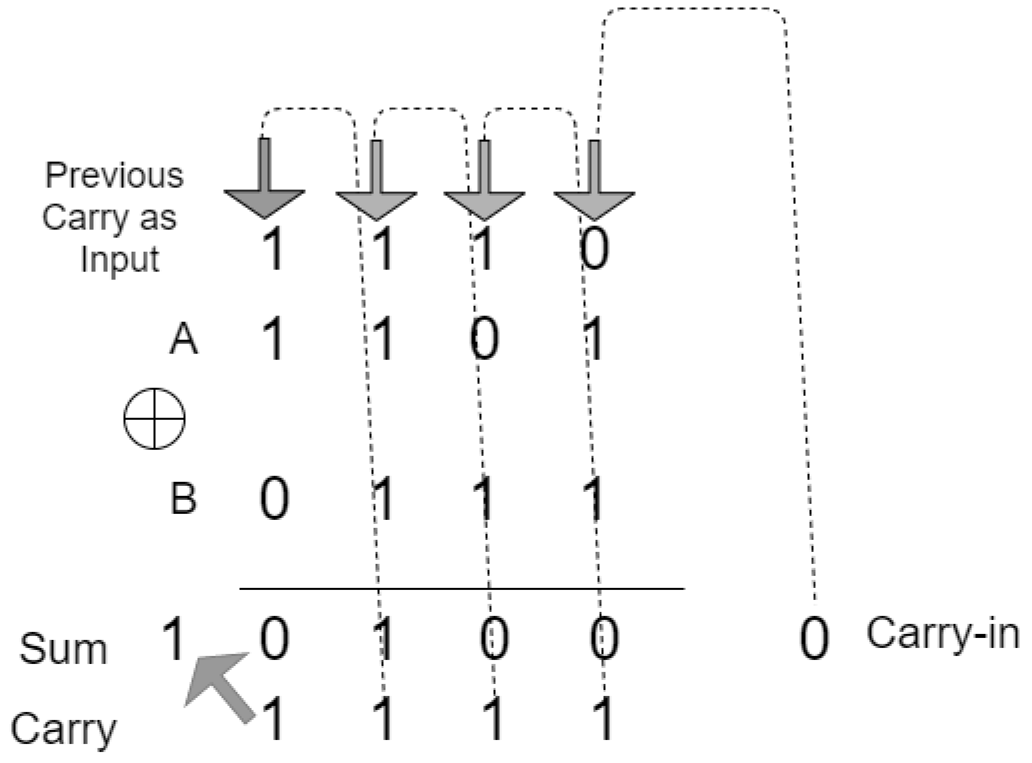


Figure: Working of 4-Bit Ripple Carry Adder

Final Sum generated for this adder – $C_3\ S_3S_2S_1S_0$



Propagation(Rippling) of Carry from the previous stage to the next stage in a RCA.

In this above example, two 4-bit numbers (13 and 7) are added normally (using a 4-bit Ripple Carry Adder) (Resultant Sum = 20 and C_{out} = 1). What is clearly visible that the carry generated for the ith stage propagates (ripples) to the (i+1) th stage as its input which is then used to compute both sum and carry for the (i+1) th stage. The carry then generated for this (i+1) th stage then propagates to (i+2) th stage. And this serial transmission (linear propagation) of carry bits continues, leading to performance slowdown in higher order adders.

2.3 Challenges involved in designing adders:

Thus, we saw that in a multi bit adder (Ripple Carry Adder) Carry bits from a previous stage (say i-1th stage) have to travel (propagate) to the next stage (ith stage) before the switching (computations) happens. Thus, the gate switching activity for a particular stage takes some additional time and this delay is propagated throughout all the stages and thus throughout the circuit. (Working numerical shown for more clarity). In this case, the since there is serial transmission of carry bits from one stage to another, there is a linear delay i.e., of order O(N). This is quite problematic for computing large number calculations.

Thus, the biggest challenge in designing adders is to reduce carry computation time and bring the delay from a linear delay to the smallest delay possible (logarithmic time computation). To design such fast systems, our task is to reduce the path of transmission of signals, which in our case is the propagating carry bits from LSB to MSB of the Sum vector. The Solution to this problem is provided by the Carry Look-Ahead (CLA) method, in which we do the carry computations ahead in time and feed the carries into the respective stages all at the same time (instead of having to wait for the carry to propagate).

This solution implementation needs extra hardware, complex wiring, requires more power consumption and spans over wider area but provides the fastest possible performance.

To compute carry ahead of time independent of propagation, we have to look at what are the possibilities of carry generation? To understand that, a simple glance at the truth table of Full Adder provides us with a possible solution to ‘predict’ or ‘look ahead/in advance’ the value of the carry. Two observations can be made:

Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

1. It is clearly seen that, the output carry is 1 if both A and B are 1, irrespective of what the value of C_{in} is.
2. The output carry is equal to the previous carry (C_{in}) when A *xor* B = 1.

In the first case, an output carry is generated whenever A and B = 1 irrespective of previous carry value and thus, A and B is said to generate carry and the carry is said to be a generate signal or simply, a generate (G).

In the second case, when A xor B = 1, the carry output is equal to the previous carry. If C_{in} was 1, C_{out} will be 1. If it was 0, C_{out} would be 0. Thus, in this case, carry is 'propagated' from previous carry to new carry and carry is said to be a propagate signal or simply, a propagate (P).

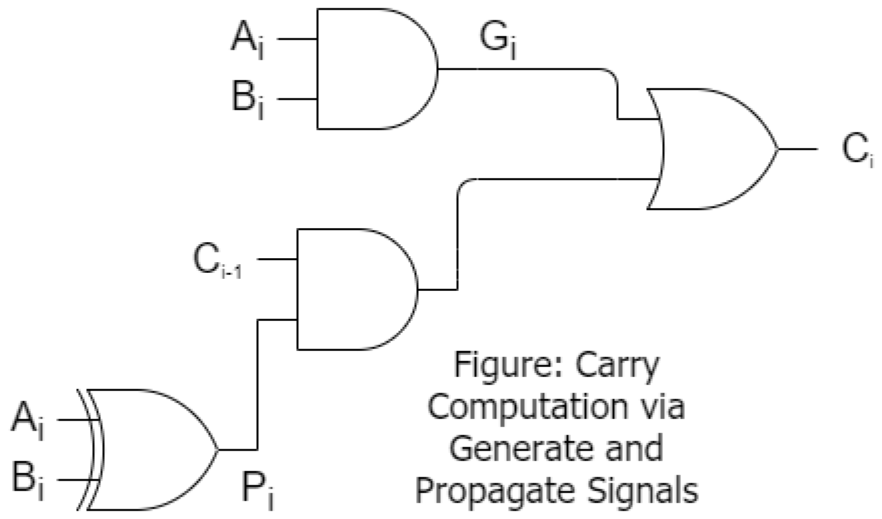


Figure: Carry Computation via Generate and Propagate Signals

Therefore,

C_{out} = (A *and* B) *or* (C_{in} *and* (A *xor* B)), which can be re written as:

$$C_{out} = G \textit{ or } (P \textit{ and } C_i)$$

Thus, by computing generates and propagates (which can be simply computed for each step in constant time and hence has minimal delay) we can look-ahead the carries in advance and save us a ton of time. We can further improve speed performance by doing these computations parallelly in a prefix tree manner, which is what we do in Kogge-Stone adder (hence the name Parallel Prefix Carry Look-Ahead Adder).

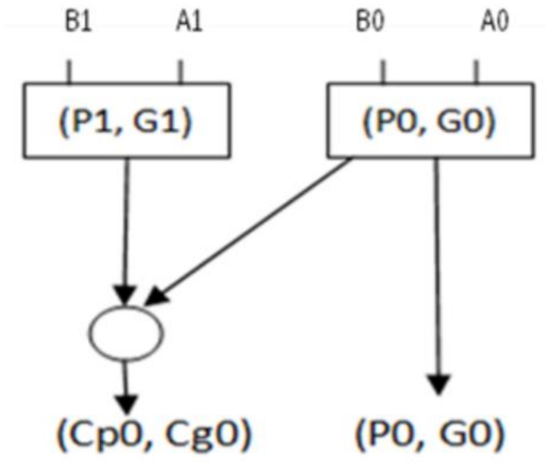
Kogge Stone Adder (KSA)

<u>3.1 Introduction</u>

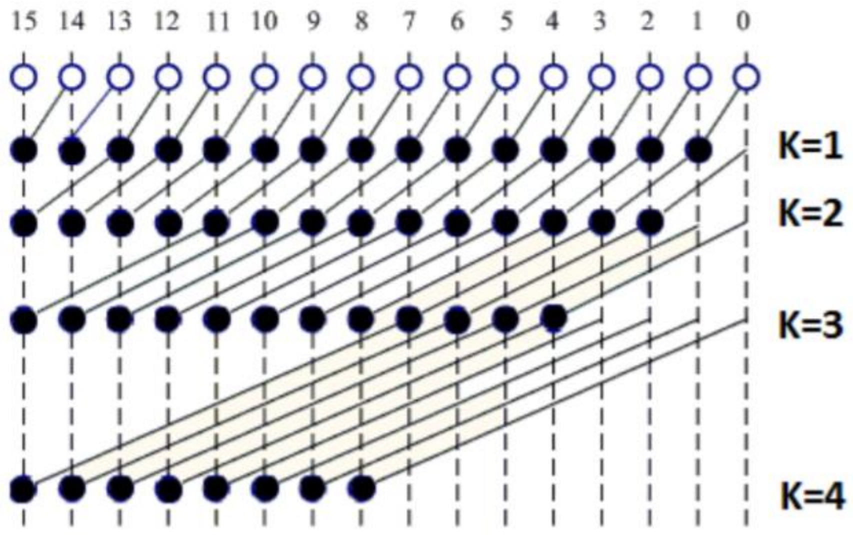
The KSA introduced above is a parallel prefix form carry look ahead adder, i.e., the carry computation takes place via prefix computation simultaneously in parallel with other operations, unlike other adders in which we have to wait for S_{i-1} to be computed to calculate the C_i (serial computation of carry). Therefore, it generates carry in O (log(N)) time and thus is widely considered as the fastest adder and is widely used in the industry for high performance arithmetic circuits and ALU operations. Kogge-Stone Adder is marked with low Fan-Out (2) at each stage, which increases performance for typical CMOS process nodes. (Fan-Out refers to the number of outputs that we can take from a gate and feed it to similar gates without degradation in performance of the output.)
Thus, basically in KSA carries are computed fast by computing them in parallel at the cost of increased area, as what is the cost with parallel prefix computation.

3.2 Computing carry using the Parallel Prefix CLA method:

Using prefix sum problem which involves the use of prefix sum operator (◦) and Recursive doubling algorithm the carry generation network is constructed in the form of trees. The basic tree is constructed in prefix-2 form, where each node processes 2 inputs. The number of nodes at first stage (as shown in figure below) is equal to the number of input bits. At first stage generate(g_i) (A_i *and* B_i) and propagate(p_i) (A_i *xor* B_i) signals corresponding to each bit is calculated. Now, if the tree levels are numbered as k, with first level as k=0, for all further tree stages every column shifts its value to 2k-1 greater than their own. In the third stage (as shown in figure below) the sum and carry are generated. All the nodes in 2nd stage (figure below) are prefix sum operator working on 2 inputs and generating outputs. Below shown are the prefix-tree structures used for doing carry computations (as mentioned already, this computation is loosely based on the recursive doubling technique) for 2 bit-Kogge Stone adder and the 16-bit Kogge Stone adder.



(2 bit-KSA)



(16-bit KSA, depicting the Parallel-Prefix based carry computation (dot operator across levels. It can be seen as when bit levels are increased by a factor of N, complexity increases by log(N) (lines joining 2nd last layer to last layer, i.e., from k=3 to k=4 shows this.)

3.3 Algorithm/Steps for calculating sum:

Given two unsigned i+1 bit binary numbers A and B ($A_iA_{i-1}...A_2A_1A_0$ and $B_iB_{i-1}...B_2B_1B_0$),

Calculating sum using Kogge Stone Adder involves mainly 4 steps:

1. Computing Initial Sum Vector ($S_{initial}$) as (A *xor* B) and keeping it aside. $S_{initial}$ can also be interpreted as the overall propagate (p_i).
2. Performing pre-processing, that is computing generate and propagate signals corresponding to each pair of bits in A and B. These signals are given by the logic equations below:
   (p – propagate signal, g – generate signal)

$$p_i = A_i \textit{ xor } B_i \text{ (i}^{th}\text{ propagate).}$$
$$g_i = A_i \textit{ and } B_i \text{ (i}^{th}\text{ generate).}$$

3. Computation of carry corresponding to each bit is done by the Carry Look-Ahead network by performing parallel-prefix carry calculation, which involves using group propagates and generates as intermediate signals which are given by the logic equations below:

$$P_{i:j} = P_{i:k+1} \textit{ and } Pk_{:j}$$
$$G_{i:j} = G_{i:k+1} \textit{ or } (P_{i:k+1} \textit{ and } G_{k:j})$$

4. Computing the Sum Vector Si: The final result of the addition, i.e., the sum is given by xor between initial sum vector ($S_{initial}$ or p_i) and C_{i-1}, that is the final Carry vector.

$$S_i = p_i \textit{ xor } Ci\text{-}1$$

Implementation:

4.1 16-Bit Kogge Stone Adder in Verilog:

The Kogge Stone Adder module consists of various generate and propagate computing operations (prefix tree carry computation) divided into a number of different blocks and further sub-modules which combine to perform the whole task of calculating the sum together in one module which is my top module for this computation.

4.2 Testbench:

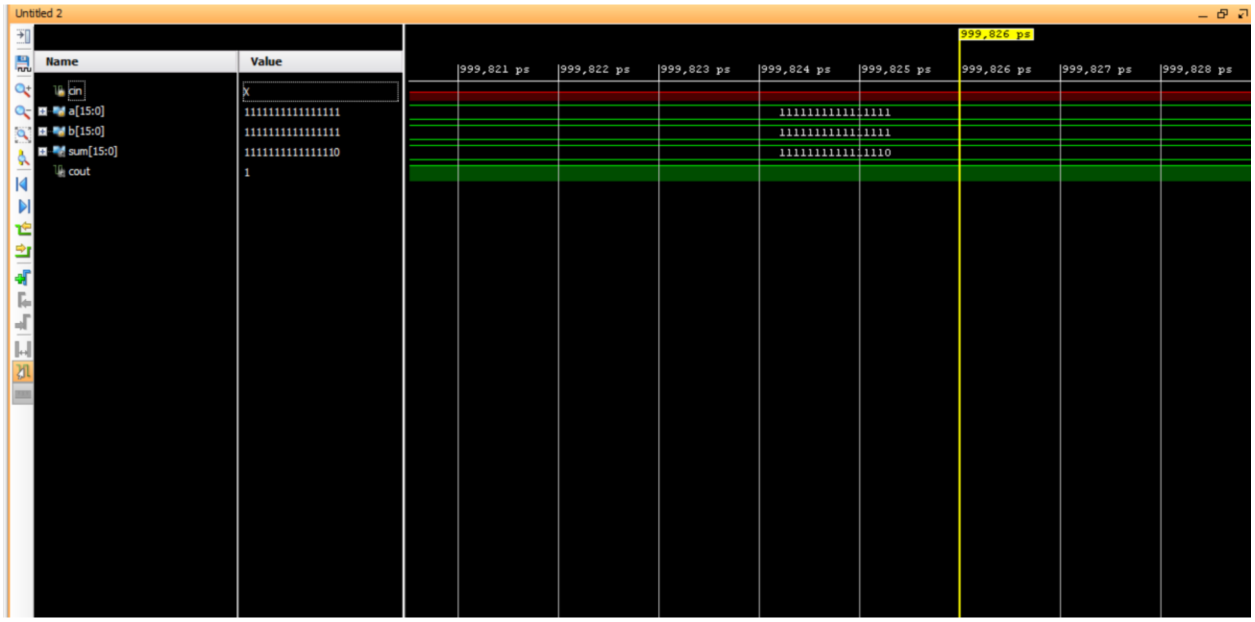
To test the proper functioning of the adder, we need to provide different 16-bit numbers to see whether our adder performs the task it is created to do: Addition. I have taken a variety of test cases to cover all cases of carry output generated as well as not generated. I have set Carry in as 0 for all times since my 16-bit KSA is a single entity and is not between multiple 16-bit KSAs so there are no chances of it receiving an input carry in the beginning.

<u>**(Simulation source- testbench):**</u>

```
module tb_KSA16;
  wire [15:0] sum;
  wire cout;
  reg [15:0] a, b;
  reg cin;

  KSA16 ksa16(sum [15:0], cout, a [15:0], b [15:0]);

  initial
  begin
    $display ("a|b ||cout|sum");
  end

  initial
  begin
    $monitor ("%b|%b||%b   |%b", a [15:0], b [15:0], cout, sum [15:0]);
  end
  initial
  begin
  ;
    #10 a=16'b0101100011110100; b=16'b1111010011110100;
    #10 a=16'b0000111100111101; b=16'b0000111100001111;
    #10 a=16'b1111111111111111; b=16'b1111111111111111;

  end
endmodule
```

Implementation Results:

5.1 Working of the Kogge-Stone adder (Simulation Results)

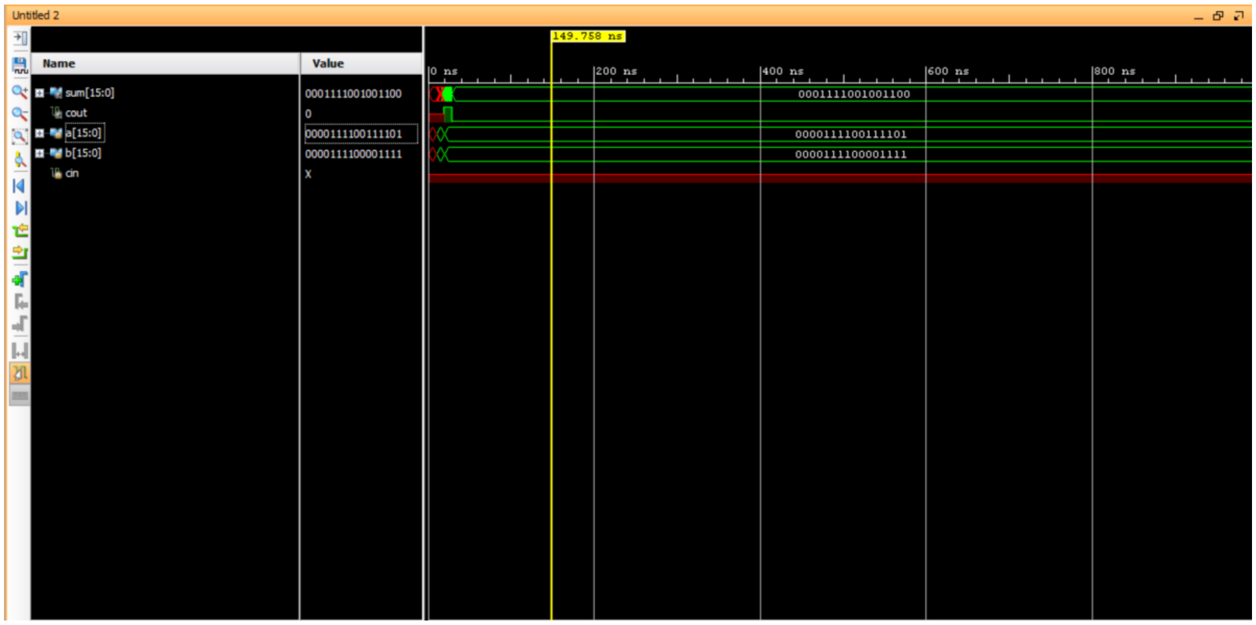


```
Untitled 2
                                    999,826 ps
Name          Value               999,821 ps  999,822 ps  999,823 ps  999,824 ps  999,825 ps  999,826 ps  999,827 ps  999,828 ps
cin           x
a[15:0]       1111111111111111                                  1111111111111111
b[15:0]       1111111111111111                                  1111111111111111
sum[15:0]     1111111111111110                                  1111111111111110
cout          1
```

In this case, I have provided the two 16-bit inputs as:

A = 1111111111111111, B = 1111111111111111.

Since C_{in} = 0, Sum should be A xor B and Carry out has to be generated subsequently. The expected result is 11111111111111110 which matches our received result, just place the value of C_{out} in front of S vector received.

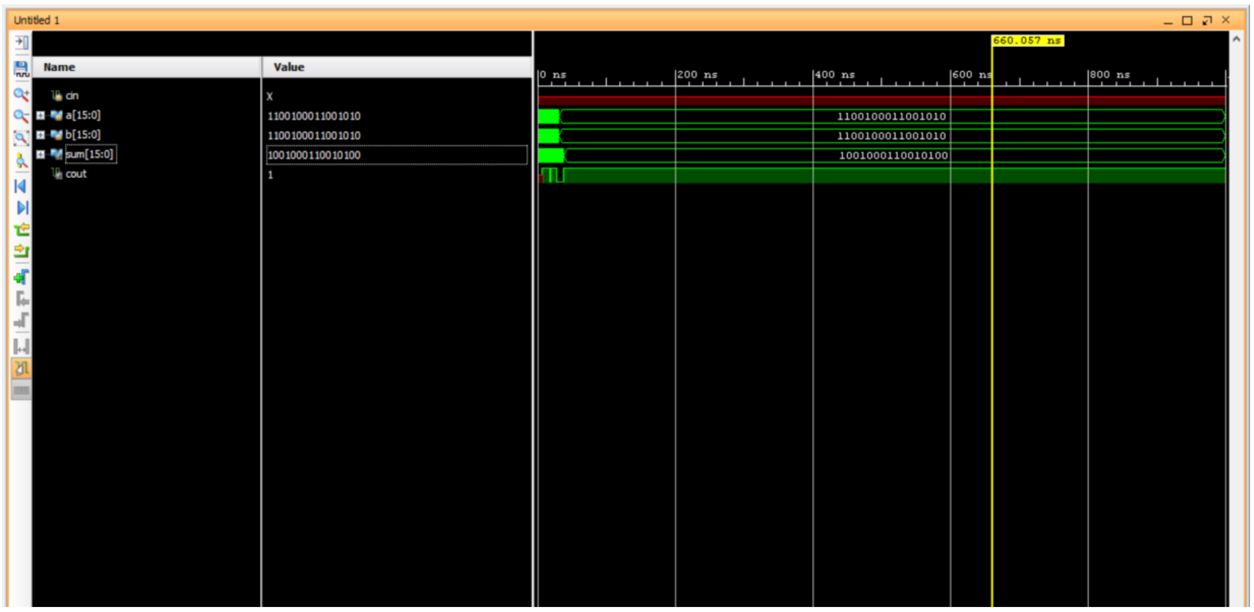


```
Untitled 2
                                149.758 ns
Name          Value               0 ns      200 ns      400 ns      600 ns      800 ns
sum[15:0]     0001111001001100                      0001111001001100
cout          0
a[15:0]       0000111100111101                      0000111100111101
b[15:0]       0000111100001111                      0000111100001111
cin           x
```

In this case, I have provided the two 16-bit inputs as:

A = 0000111100111101; B=0000111100001111.

Since C_{in} = 0, Sum should be A xor B and Carry out has to be generated subsequently. The expected result is 0001111001001100 which matches our received result.



Untitled 1

Name	Value
cin	X
a[15:0]	1100100011001010
b[15:0]	1100100011001010
sum[15:0]	1001000110010100
cout	1

In this case, I have provided the two 16-bit inputs to be equal, with both

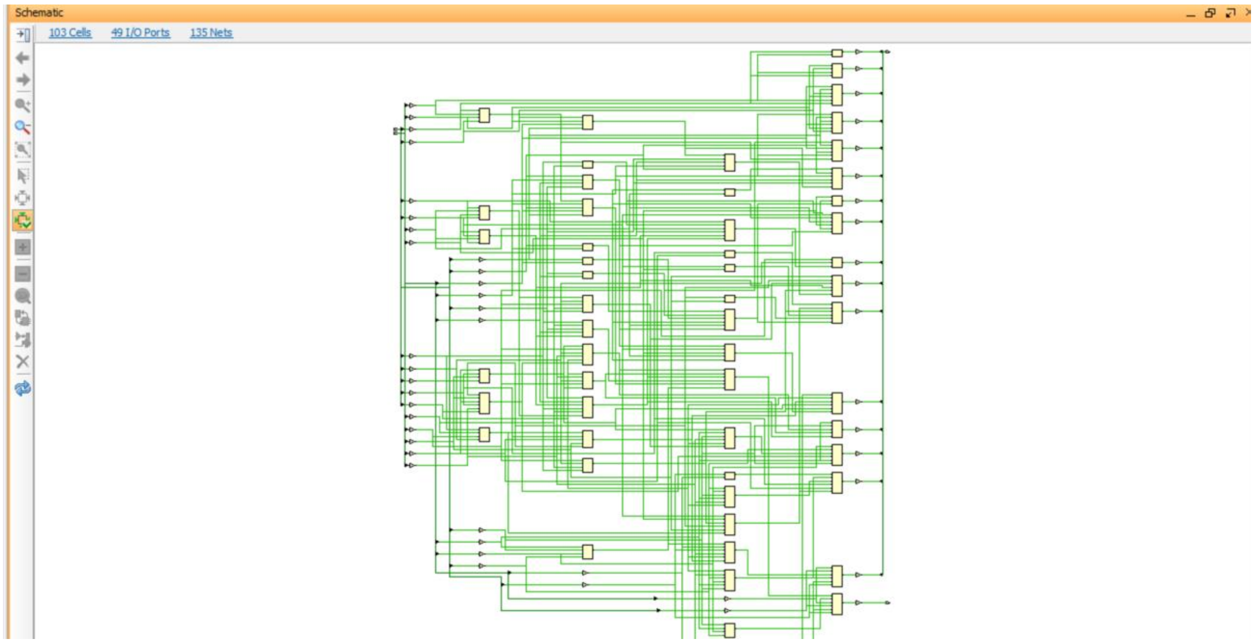
A = 1100100011001010 and B = 1100100011001010.

Since C_{in} = 0, Sum should be A xor B and Carry out has to be generated subsequently. The expected result is 11001000110010100 which matches our received result, just place the value of C_{out} in front of S vector received.

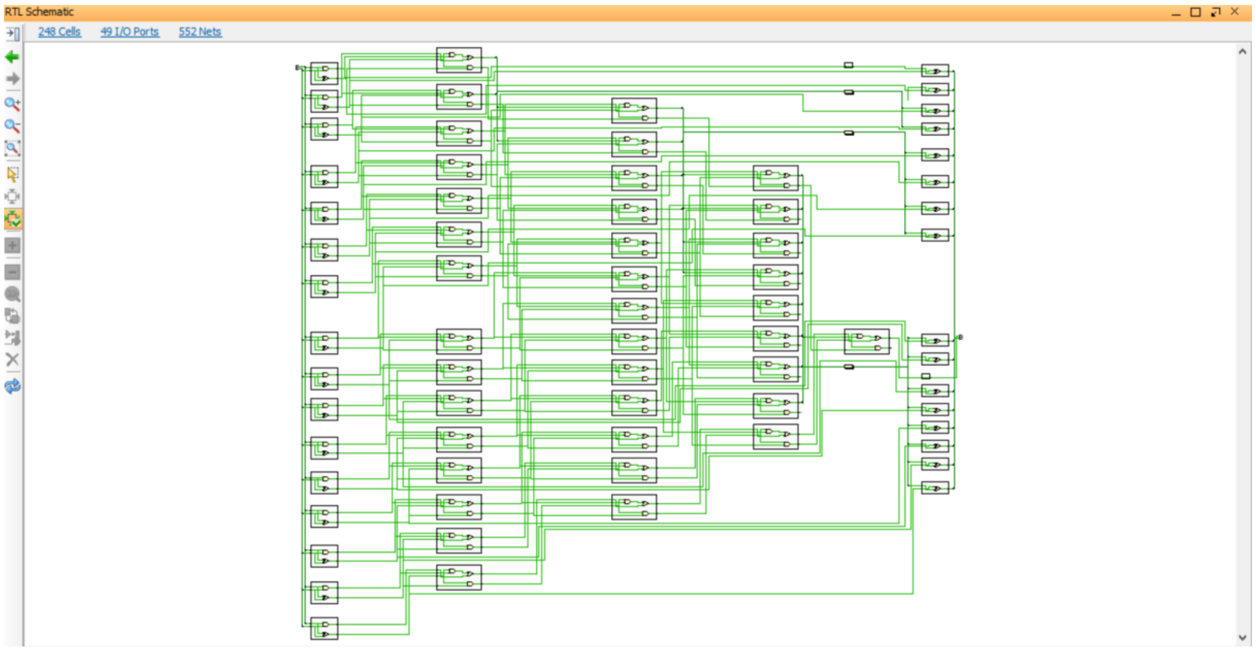
Thus, we have seen that our adder works perfectly and calculates accurately the sum and output carry in fastest time possible.

5.2 RTL Schematic Generated:

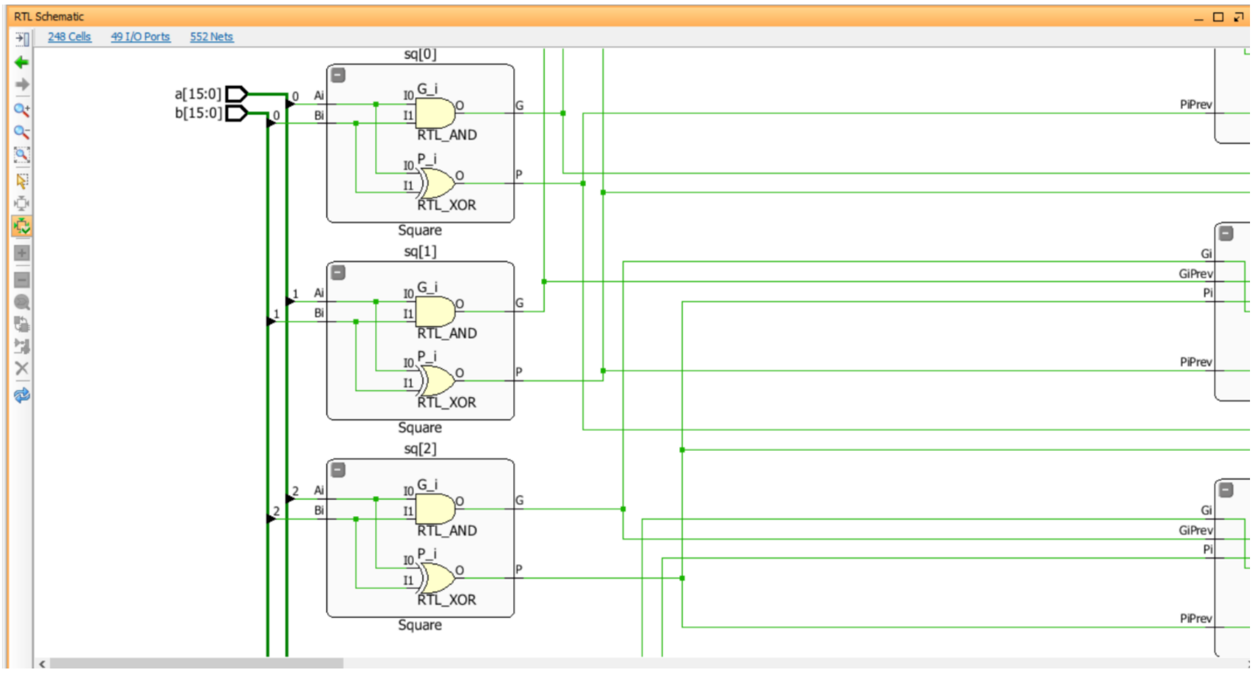
Register Transfer Level Schematic shows us the the practical gate level circuit modelled by our program. The 16-bit Kogge Stone Adder Schematic generated by our implementation is:



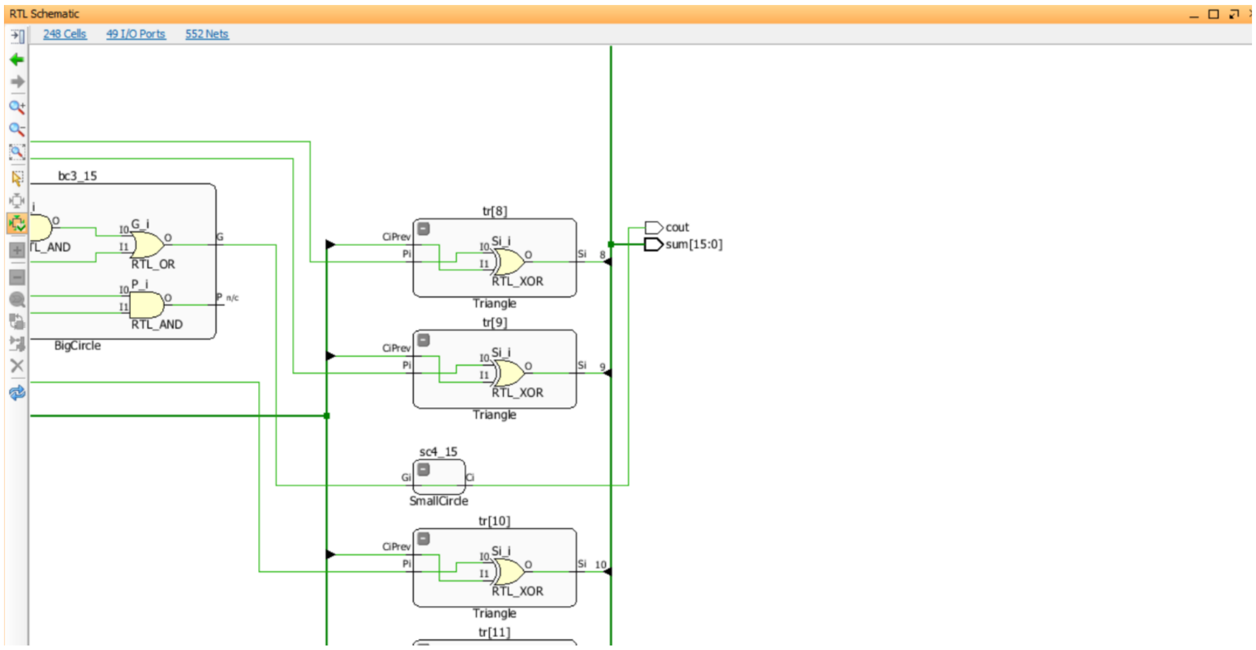
Generated RTL Schematic for 16-bit KSA



Generated RTL Schematic, expanded view with logic gates



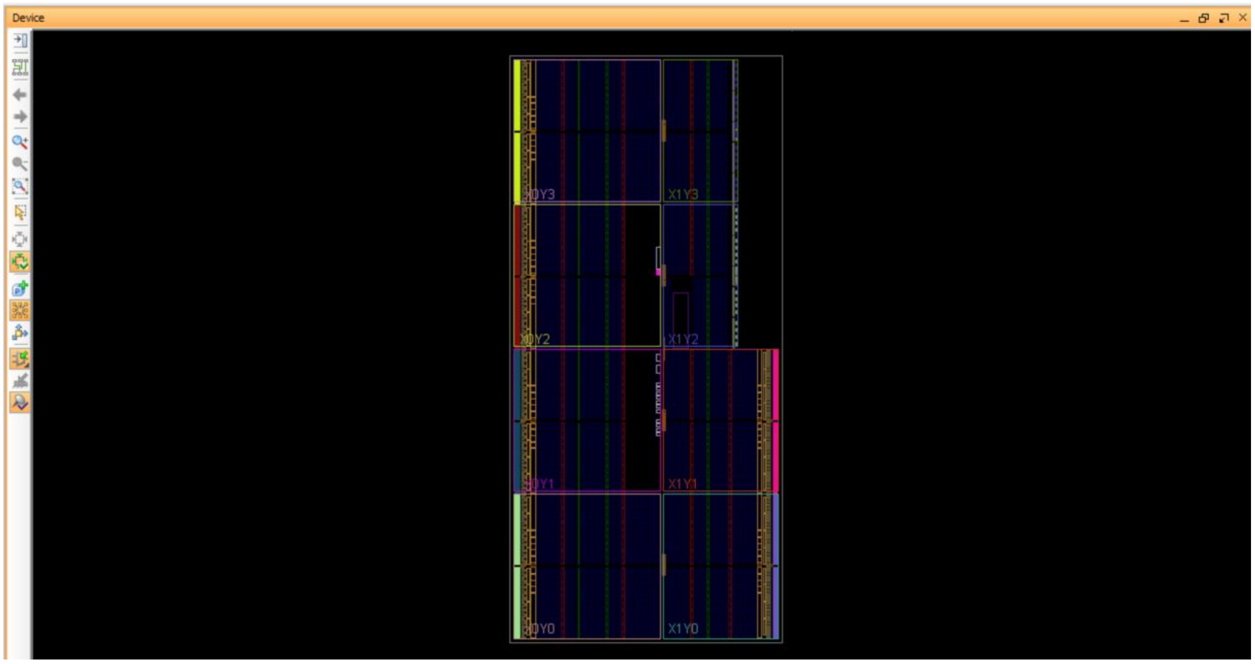
Input side section of the schematic. A and B are our 2 16-bit inputs, as is clearly visible.



Output side section of the generated schematic. The output is a 17-bit number, with 16 Sum bits and 1 Carry-Out bit (C_{out} here).

5.3 Synthesis Result:

The final synthesized result for the VLSI implementation of 16-bit Kogge Stone adder is shown:



5.4 Conclusion:

Kogge-Stone adder sacrifices complexity and area for achieving highest computational speed; Its parallel prefix-tree based look-ahead computation of carry provides a significant speed-up as compared to normal adders like the Ripple Carry Adder (O(log(N) vs O(N) delay in computing carry).

Successfully implemented a 16-bit Kogge Stone adder using Vivado tool (Xilinx). Coded architecture in Verilog.