

# **Design of High Speed Parallel Prefix Kogge Stone Adder using Tanner and Xilinx**

Bhargava U  
Dept of ECE  
1PI11EC026  
[Bhargavau6@gmail.com](mailto:Bhargavau6@gmail.com)

Chandra Kiran  
Dept of ECE  
1PI11EC029  
[ck.arsenal@gmail.com](mailto:ck.arsenal@gmail.com)

Mohammed Sahil Akbar  
Dept of ECE  
1PI11EC063  
[sahilakbar94@gmail.com](mailto:sahilakbar94@gmail.com)

**PROJECT PROPOSAL**  
submitted to the supervisor

**Mrs.Nagamani A**  
Assistant Professor of Low Power VLSI Design  
PES University,  
Bangalore,India  
[nagamani@pes.edu](mailto:nagamani@pes.edu)

**Abstract:**

The fundamental operations involved in any Digital systems are addition and multiplication. Addition is an indispensable operation in any Digital, Analog, or Control system. Fast and accurate operation of digital system depends on the performance of adders. The main function of adder is to speed up the addition of partial products generated during multiplication operation. Hence improving the speed by reduction in area is the main area of research in VLSI system design. Over the last decade many types of adder architectures were studied, such as carry ripple adders, carry skip adder, carry look ahead adder, parallel prefix tree adders etc. In tree adders, carries are generated in parallel and fast computation is obtained at the expense of increased area and power. The main advantage of the design is that the carry tree reduces the number of logic levels (N) by essentially generating the carries in parallel.

The parallel-prefix tree adders are more favorable in terms of speed due to the complexity  $O(\log_2 N)$  delay through the carry path compared to that of other adders. The prominent parallel prefix tree adders are Kogge-Stone, Brent-Kung, Han-Carlson, and Sklansky. Out of these, it was found from the literature that Kogge-stone adder is the fastest adder when compared to other adders. The adder priority in terms of worst-case delay is found to be Ripple-Carry, Carry-Look-Ahead, Carry-Select and Kogge-Stone. This is due to the number of “Reduced stages”. Kogge-Stone adder implementation is the most straightforward, and also it has one of the shortest critical paths of all tree adders. The drawback with the Kogge-Stone adder implementation is the large area consumed and the more complex routing (Fan-Out) of interconnects.

**Background information of KSA:**

The Kogge-Stone adder is a parallel prefix form of carry look-ahead adder. It generates the carry signals in  $O(\log_2 N)$  time, and is widely considered as the fastest adder design possible. It is the most common architecture for high-performance adders in industry. The Kogge-Stone adder concept was first developed by Peter M. Kogge and Harold S. Stone. In Kogge-stone adder, carries are generated fast by computing them in parallel at the cost of increased area. The Kogge Stone Adder (KSA) has regular layout which makes them favored adder in the electronic technology. Another reason the KSA is the favored adder is because of its minimum fan-out or minimum logic depth. As a result of that, the KSA becomes a fast adder but has a large area. The delay of KSA is equal to  $\log_2 n$  which is the number of stages for the “o” operator. The KSA has the area (number of “o” operators) of  $(n \cdot \log_2 n) - n + 1$  where n is the number of input bits

The complete functioning of KSA can be easily comprehended by analyzing it in terms of three distinct parts :

### 1. **Pre processing**

This step involves computation of generate and propagate signals corresponding too each pair of bits in A and B. These signals are given by the logic equations below:

$$p_i = A_i \text{ xor } B_i$$

$$g_i = A_i \text{ and } B_i$$

### 2. **Carry look ahead network**

This block differentiates KSA from other adders and is the main force behind its high performance. This step involves computation of carries corresponding to each bit. It uses group propagate and generate as intermediate signals which are given by the logic equations below:

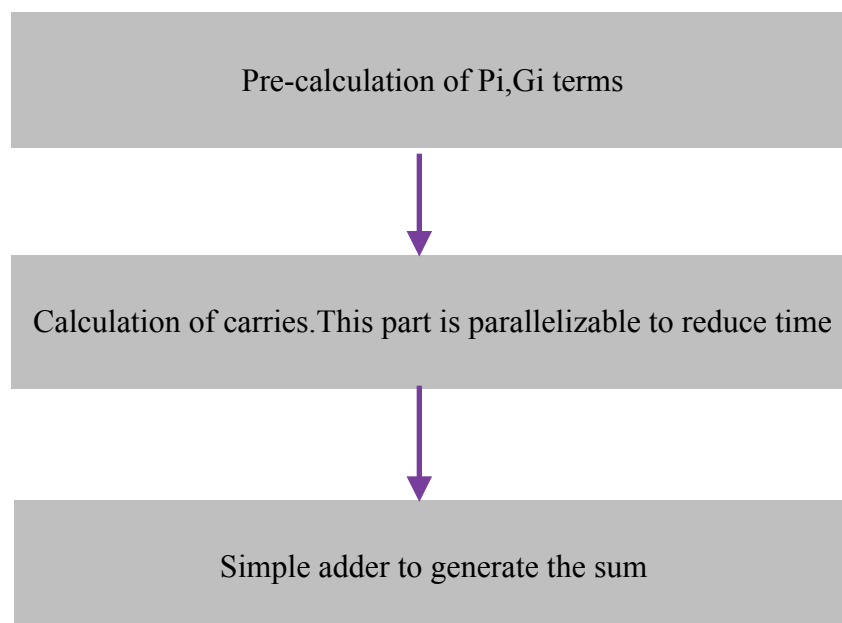
$$P_{i:j} = P_{i:k+1} \text{ and } P_{k:j}$$

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

### 3. **Post processing**

This is the final step and is common to all adders of this family (carry look ahead). It involves computation of sum bits. Sum bits are computed by the logic given below:

$$S_i = p_i \text{ xor } C_{i-1}$$



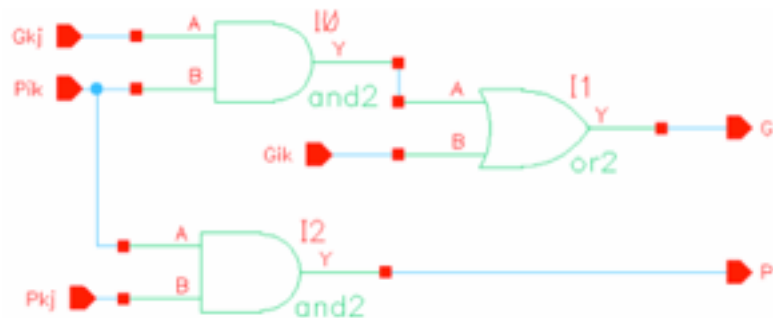
This example shows how the logic was verified of an 8 bit Kogge Stone Adder.

The major steps are:

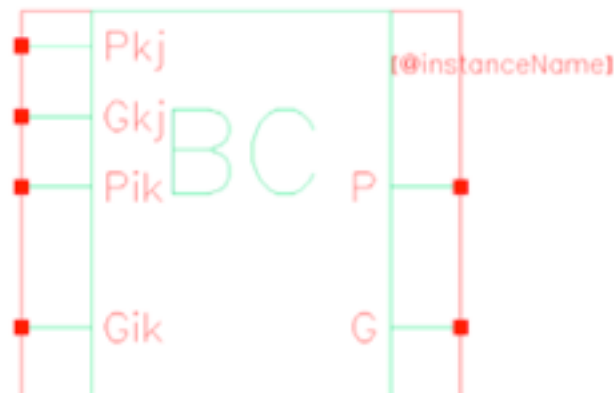
- 1.Convert gray and black cells to schematic .
- 2.Convert “tree” diagrams into a schematic.
- 3.Create a trusted adder.
- 4.Compare trusted adder with the basic ripple carry adder.

### **Black Cell:**

**Schematic**

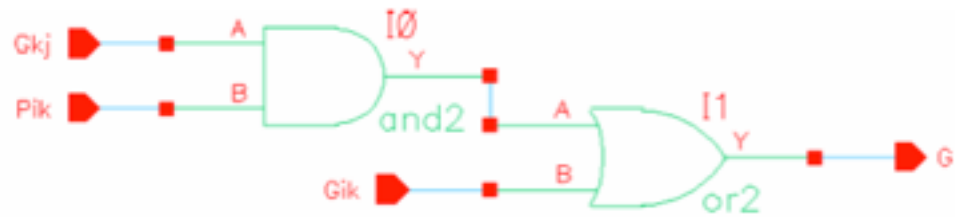


**Symbol**



Gray cell :

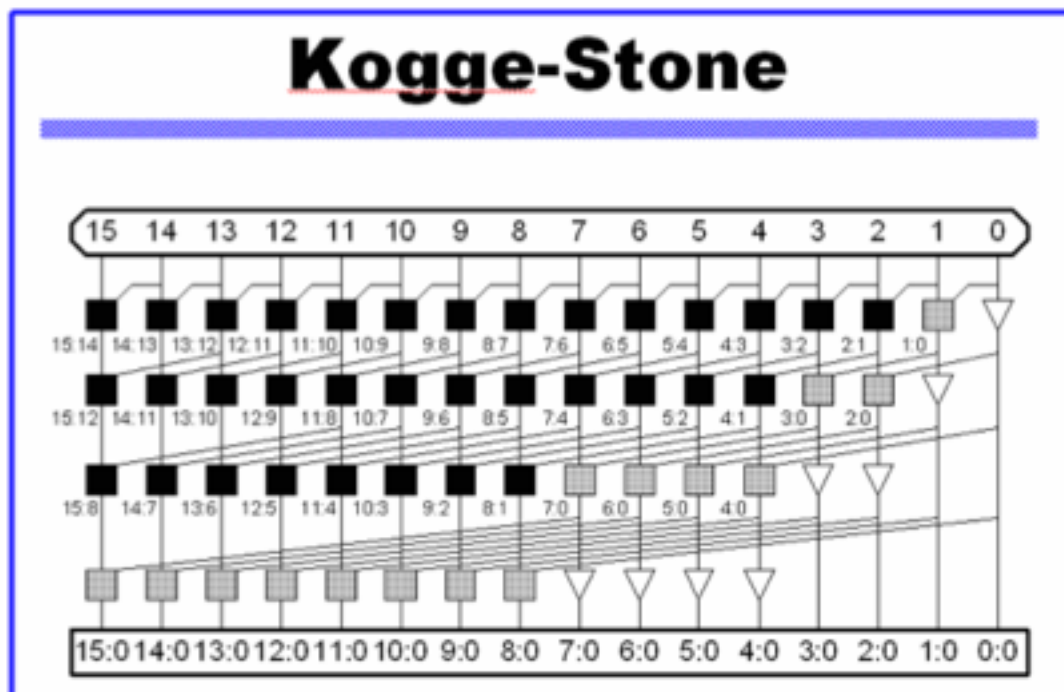
## Schematic

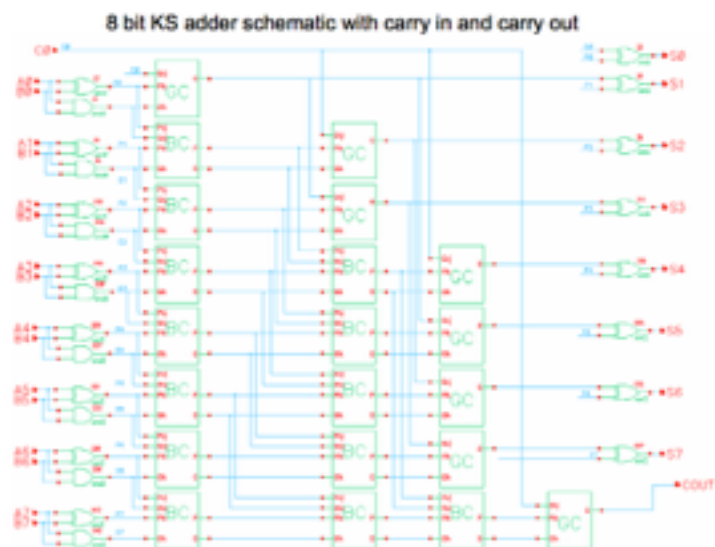


## Symbol



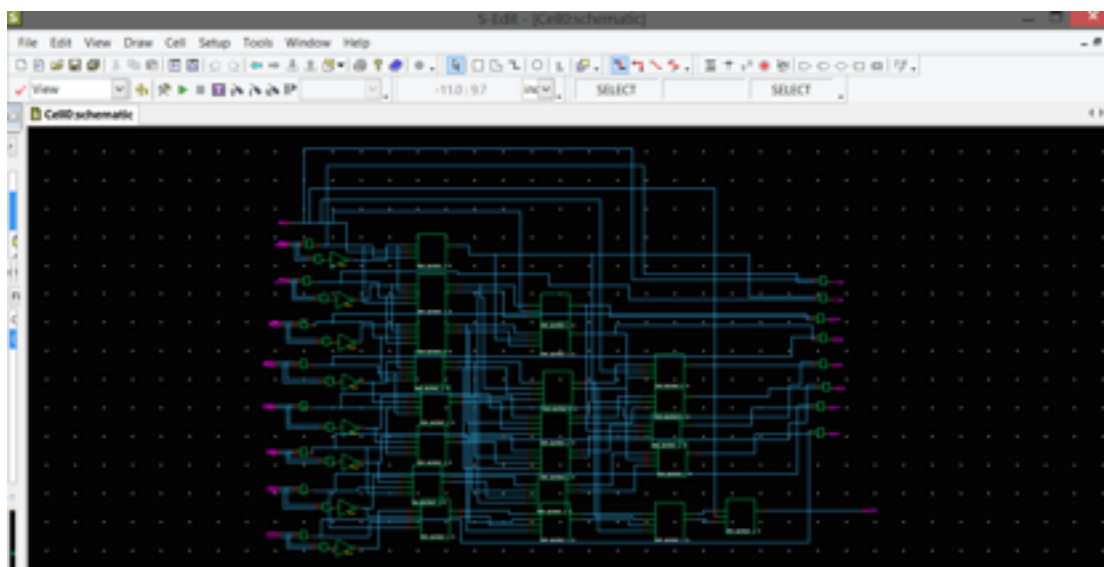
*A Gray cell is a Black cell without the group propagate.*

Tree Diagram:



## Screenshots of our work:

### 1. Creation of the KSA schematic in S-Edit



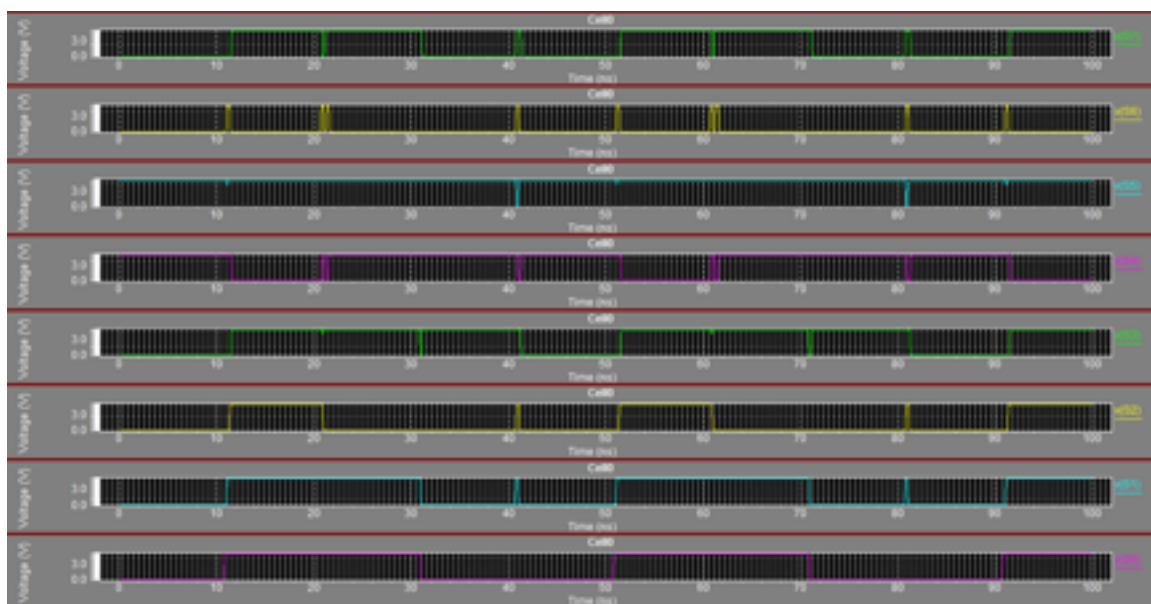
### 2.Generation of Netlist

```

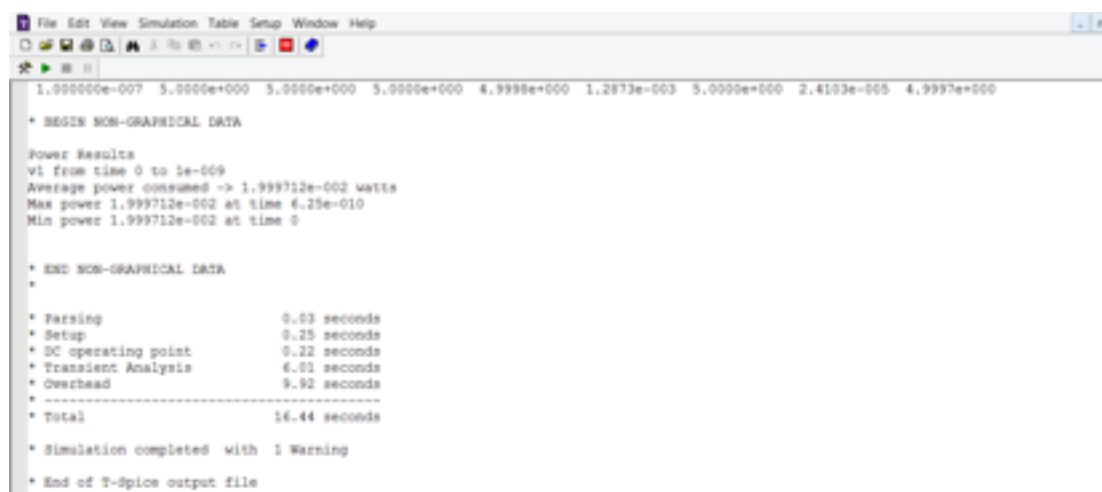
lib "C:\T\SPICE\Tanner EDA Libraries\Tanner Tools v13.0\Library\Models\Generic_025.lib"
v1 VDD GND dc 5
.vtran 10n 500m
v2 A0 GND dc 5 BIT ((0010))
v3 A1 GND dc 5 BIT ((1010))
v4 A2 GND dc 5 BIT ((1010))
v5 A3 GND dc 5 BIT ((0100))
v6 A4 GND dc 5 BIT ((1010))
v7 A5 GND dc 5 BIT ((1011))
v8 A6 GND dc 5 BIT ((1010))
v9 A7 GND dc 5 BIT ((0010))
v10 B0 GND dc 5 BIT ((1001))
v11 B1 GND dc 5 BIT ((1100))
v12 B2 GND dc 5 BIT ((0110))
v13 B3 GND dc 5 BIT ((1001))
v14 B4 GND dc 5 BIT ((1001))
v15 B5 GND dc 5 BIT ((1100))
v16 B6 GND dc 5 BIT ((0000))
v17 B7 GND dc 5 BIT ((1110))
v18 C0 GND dc 5 BIT ((0000))
.print tran v180
.print tran v181
.print tran v182
.print tran v183
.print tran v184
.print tran v185
.print tran v186
.print tran v187

```

### 3.Verification using W-Edit:



### 4.Power Analysis:



**Final Observation and Analysis :**

	Kogge Stone Adder	Ripple Carry Adder
<b>No.of Bits(N)</b>	8	8
<b>Logic Levels</b>	$\log N(\text{base}2) = 3$	$N-1 = 7$
<b>Cells</b>	$N \log N(\text{base}2) = 24$	$N = 8$
<b>Max Fanout</b>	2	1
<b>Power (Simulation Time=1ns)</b>	1.999712e-002 watts	12.334e-002 watts

**Part 2:**

We have also performed the KSA analysis using Xilinx(Verilog HDL).The observations are as follows:

**Verilog code for grey cell :**

```

module gray_cell(Gkj, Pik, Gik, G);
//gray cell
input Gkj, Pik, Gik;
output G;
wire Y;

and(Y, Gkj, Pik);
or(G, Y, Gik);

endmodule

```

**Verilog code for black cell :**

```

module black_cell(Gkj, Pik, Gik, Pkj, G, P);
//black cell
input Gkj, Pik, Gik, Pkj;
output G, P;
wire Y;

and(Y, Gkj, Pik);
or(G, Gik, Y);

```



```
and(P, Pkj, Pik);
```

```
endmodule
```

Verilog code for XOR block:

```
module and_xor(a, b, p, g);
```

```
//very first inputs - and/xor
```

```
input a, b;
```

```
output p, g;
```

```
xor(p, a, b);
```

```
and(g, a, b);
```

```
endmodule
```

### **Verilog code for KSA:**

```
//8-bit Kogge-Stone adder
```

```
module kogge_stone (x, y, sum, cin, cout);
```

```
//kogge stone structural model
```

```
input [7:0] x, y; //input
```

```
output [7:0] sum; //output
```

```
input cin; //carry-in
```

```
output cout; //carry-out
```

```
wire [7:0] G_Z, P_Z, //wires
```

```
    G_A, P_A,
```

```
    G_B, P_B,
```

```
    G_C, P_C;
```

```
//level 1
```

```
gray_cell level_0A(cin, P_Z[0], G_Z[0], G_A[0]);
```

```
black_cell level_1A(G_Z[0], P_Z[1], G_Z[1], P_Z[0], G_A[1], P_A[1]);
```

```
black_cell level_2A(G_Z[1], P_Z[2], G_Z[2], P_Z[1], G_A[2], P_A[2]);
```

```
black_cell level_3A(G_Z[2], P_Z[3], G_Z[3], P_Z[2], G_A[3], P_A[3]);
```

```
black_cell level_4A(G_Z[3], P_Z[4], G_Z[4], P_Z[3], G_A[4], P_A[4]);
```

```
black_cell level_5A(G_Z[4], P_Z[5], G_Z[5], P_Z[4], G_A[5], P_A[5]);
```

```
black_cell level_6A(G_Z[5], P_Z[6], G_Z[6], P_Z[5], G_A[6], P_A[6]);
```

```
black_cell level_7A(G_Z[6], P_Z[7], G_Z[7], P_Z[6], G_A[7], P_A[7]);
```

```
//level 2
```

```
gray_cell level_1B(cin, P_A[1], G_A[1], G_B[1]);
```

```
gray_cell level_2B(G_A[0], P_A[2], G_A[2], G_B[2]);
```

```
black_cell level_3B(G_A[1], P_A[3], G_A[3], P_A[1], G_B[3], P_B[3]);
```

```
black_cell level_4B(G_A[2], P_A[4], G_A[4], P_A[2], G_B[4], P_B[4]);
```

```
black_cell level_5B(G_A[3], P_A[5], G_A[5], P_A[3], G_B[5], P_B[5]);
```

```
black_cell level_6B(G_A[4], P_A[6], G_A[6], P_A[4], G_B[6], P_B[6]);
```

```
black_cell level_7B(G_A[5], P_A[7], G_A[7], P_A[5], G_B[7], P_B[7]);
```

```
//level 3
```

```
gray_cell level_3C(cin, P_B[3], G_B[3], G_C[3]);
```

```
gray_cell level_4C(G_A[0], P_B[4], G_B[4], G_C[4]);
```

```
gray_cell level_5C(G_B[1], P_B[5], G_B[5], G_C[5]);
```

```
gray_cell level_6C(G_B[2], P_B[6], G_B[6], G_C[6]);
```

```
black_cell level_7C(G_B[3], P_B[7], G_B[7], P_B[3], G_C[7], P_C[7]);
```

```
//level 4
```

```

gray_cell level_7D(cin, P_C[7], G_C[7], cout);

//xor with and
and_xor level_Z0(x[0], y[0], P_Z[0], G_Z[0]);
and_xor level_Z1(x[1], y[1], P_Z[1], G_Z[1]);
and_xor level_Z2(x[2], y[2], P_Z[2], G_Z[2]);
and_xor level_Z3(x[3], y[3], P_Z[3], G_Z[3]);
and_xor level_Z4(x[4], y[4], P_Z[4], G_Z[4]);
and_xor level_Z5(x[5], y[5], P_Z[5], G_Z[5]);
and_xor level_Z6(x[6], y[6], P_Z[6], G_Z[6]);
and_xor level_Z7(x[7], y[7], P_Z[7], G_Z[7]);

//outputs
xor(sum[0], cin, P_Z[0]);
xor(sum[1], G_A[0], P_Z[1]);
xor(sum[2], G_B[1], P_Z[2]);
xor(sum[3], G_B[2], P_Z[3]);
xor(sum[4], G_C[3], P_Z[4]);
xor(sum[5], G_C[4], P_Z[5]);
xor(sum[6], G_C[5], P_Z[6]);
xor(sum[7], G_C[6], P_Z[7]);

endmodule

```

### **Test Bench:**

```

module tb2lf; // testbench for the 8-bit unsigned Kogge-Stone adder
    // exhaustive checking of all 256*256*2 possible cases

    reg [7:0] a, b; // 8-bit operands
    reg c0; // carry input
    wire [7:0] s; // 8-bit sum output
    wire c8; // carry output
    reg [8:0] check; // 9-bit value used to check correctness
    integer i, j, k; // loop variables
    integer num_correct; // counter to keep track of the number correct
    integer num_wrong; // counter to keep track of the number wrong

    // instantiate the 8-bit Kogge-Stone adder
    kogge_stone ks1(a, b, s, c0, c8);

    // exhaustive checking
    initial begin
        // initialize the counter variables
        num_correct = 0; num_wrong = 0;

        // loop through all possible cases and record the results
        for (i = 0; i < 256; i = i + 1) begin
            a = i;
            for (j = 0; j < 256; j = j + 1) begin
                b = j;
                for (k = 0; k < 2; k = k + 1) begin
                    c0 = k;
                    check = a + b + c0;

                    // compute and check the product
                    #10 if ({c8, s} == check)

```

```

        num_correct = num_correct + 1;
    else
        num_wrong = num_wrong + 1;

    // following line is for debugging
    // $display($time, " %d + %d + %d = %d (%d)", a, b, c0, {c8, s}, check);

    end
end
end

// print the final counter values
$display("num_correct = %d, num_wrong = %d", num_correct, num_wrong);

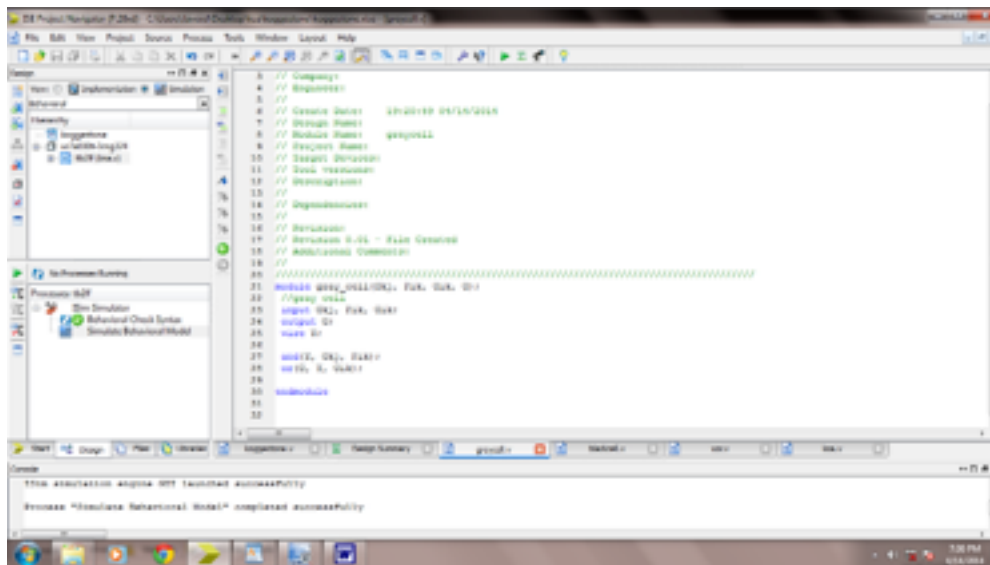
end

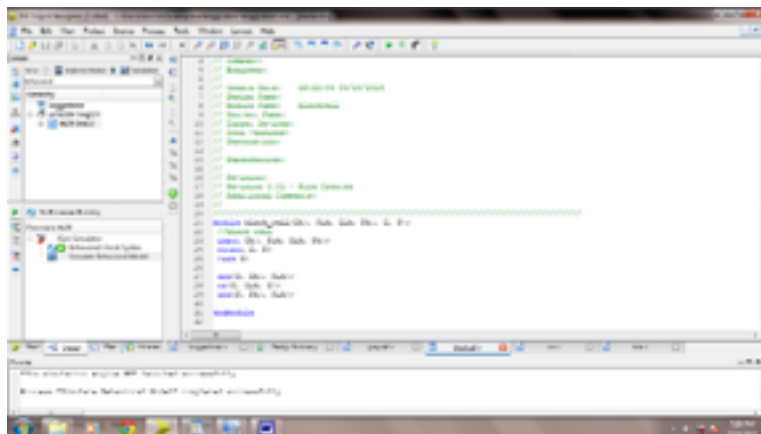
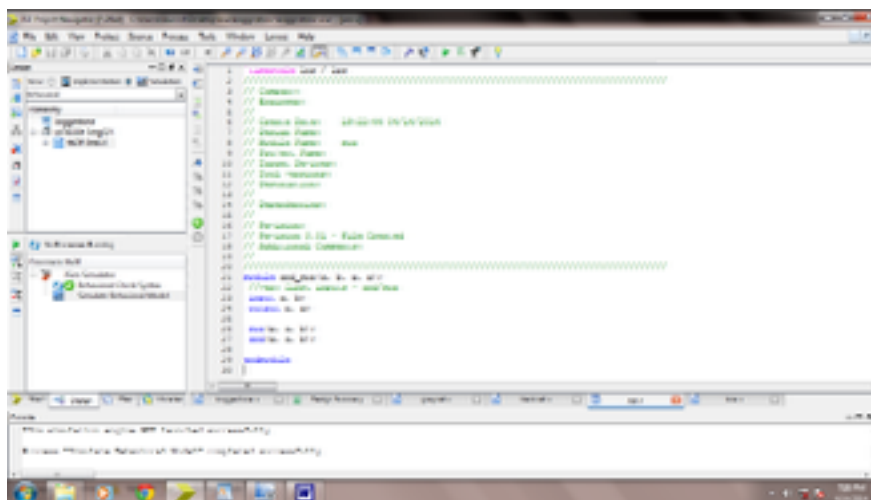
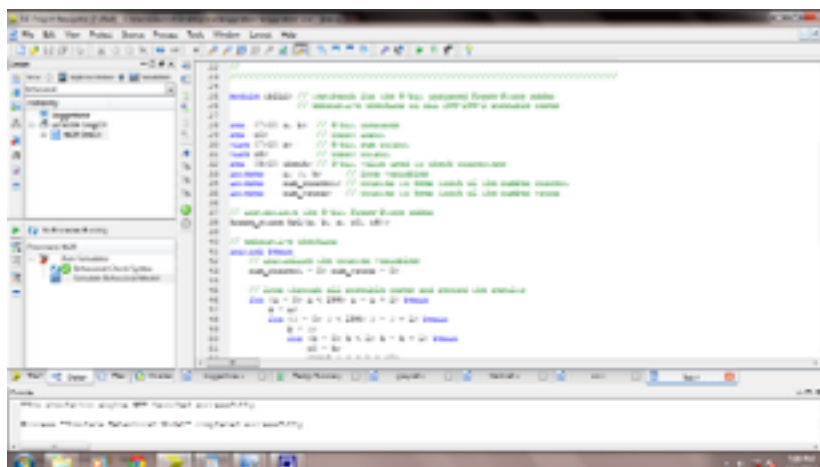
endmodule

```

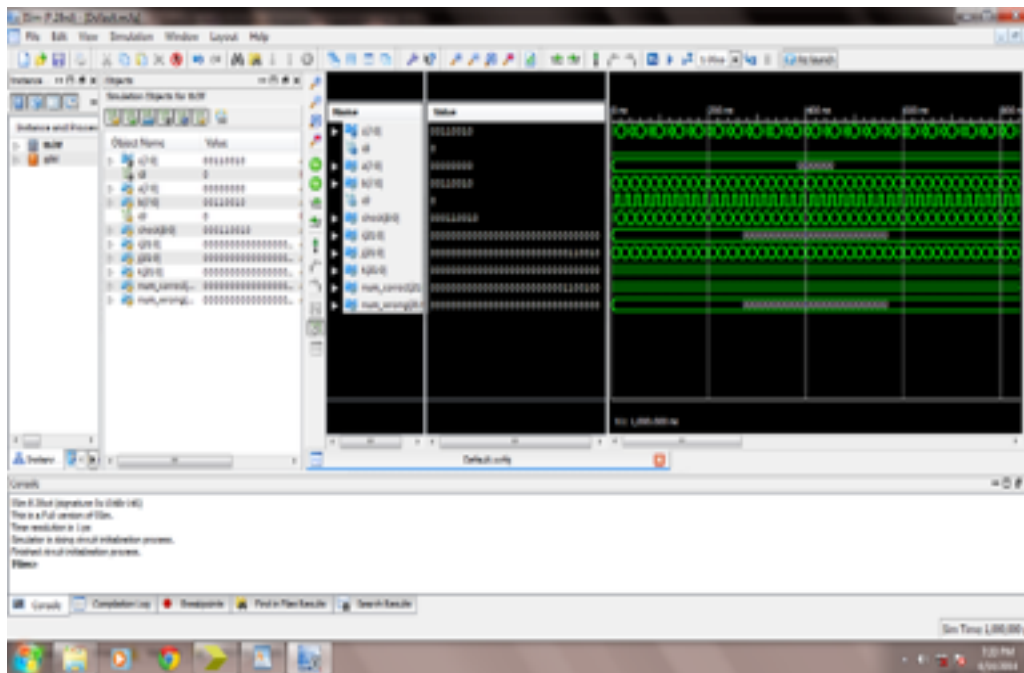
### Screenshots from Xilinx tool:

#### 1.Grey cell:



**2.Black cell:****3.XOR block:****4.KSA:**

#### 4.Waveforms:



The above analysis is performed for the operands  $a=00000000$  and  $b=00100110$  which yields sum  $s=00100110$

and carry out  $c_0=0$ .

Hence KSA is also verified using Xilinx.

#### Conclusion:

1. An 8 bit KSA was verified using Tanner and waveforms were observed.
2. Power analysis was carried out and was compared with the classical 8 bit ripple carry adder and the results were very impressive. It was found out that KSA has better power optimization than the ripple adder but there is a tradeoff with the area as the KSA consumed large amount of chip area.
3. The number of stages consumed by KSA is very impressive as it just uses 3 stages as compared to the ripple adder which uses 7 stages. Hence the delay is reduced.
4. The speed of computation is very high in KSA compared to the ripple adder.
5. The 8 bit KSA was also verified using Xilinx Verilog Tool and waveforms observed with a particular example.

References:

1. <http://www.cl.cam.ac.uk/research/srg/han/ACS-P35/8-bit-KoggeStone-Adder.pdf>
2. [http://www.engr.sjsu.edu/~dparent/ee166/adder\\_verification.pdf](http://www.engr.sjsu.edu/~dparent/ee166/adder_verification.pdf)
3. <http://www.ijese.org/attachments/File/v1i4/D0181021413.pdf>
4. <http://www.slideshare.net/peeyushpashine/kogge-stone-adder>
5. <http://georgeblog.nyarangi.com/2010/07/8-bit-kogge-stone-adder.html>
6. [“VLSI Design - by Weste and Harris” text book for the tree diagram](#)