ECE 429 Spring 2015

Final Project: Design and Synthesis of Carry Propagation Adders

Ming Chen A20314690
Xinyi Yu A20314655

Project Duration: the week of 04/08 to 05/04

Role and Contribution:

Carry Ripple Adder and Carry Skip Adder Design: Xinyi Yu

Carry Select Adder and Kogge-Stone Adder Design: Ming Chen

Synthesis and Simulation: Ming Chen

Functional alidation and verification: Xinyi Yu

Bonus Work- 16×16 array multiplier: Ming Chen and Xinyi Yu

Performance Analysis and report: Ming Chen and Xinyi Yu

Signature:

Ming Chen

Xinyi Yu

I. Abstraction

In this project, four categories of 8-bits and 32 bits adders with Verilog code are designed. They are Carry Ripple Adder, Carry Skip Adder, Carry Select Adder, and Prefix Adder-KoggeStone. In addition, their properties with EDA tools are also verified. A table including the timing, power and area are also presented in this paper, following by some analysis related to these result. Meanwhile, the bonus work is also implemented. The design and test result of 16×16 bit array multiplier is included in the bonus work section.

II. Introduction

This project aims at introducing VLSI design concepts including datapath circuit design, standard cell based design flow, and design validation and verification through construction of fast adder architectures in Verilog, to be synthesized using commercial EDA tools from Synopsys and Cadence Design Systems. [1]

It introduced measures to construct adder hardware with Verilog code, how to make design trade-offs during design, and how EDA tools transform the design implementation from higher abstraction levels.

During this project, several difficulties were encountered when compiling Verilog codes. Even though the Kogge-Stone is the most complex one, the most challenging one for us is the Carry Skip Adder. The Kogge-Stone is complex in logic, and took us a lot of time to design it, but we got the satisfied result at the end. However, the Carry Skip Adder confused us for a long time. The timing result for Carry Skip Adder was longer than the Carry Ripple Adder, which means there are some wrong in the Verilog code. But we checked the code many times and still cannot figure it out.

After reading the textbook and learning tutorial several times, and also consulted TA, we finally overcame this problem. We found out that we didn't understand the Carry Skip Adder thoroughly, even though we listened to professor's descriptions of this adder carefully in the class. The previous module we designed the adder was wrong. The "and", "or" gates in Verilog will wait till all the input signals to arrive. But we considered "and(out, in1, in2)" will skip waiting for "in2" to arrive if "in1" = 0 in our previous design. Then we looked for a different approach. Actually, when propagation for each single adder is 1, the output Cout should just related to initial carry in bit, Ci, and has no relationship with the carry out in the previous stage. Using this idea, we fixed this problem and finally got the satisfied timing result for the Carry Skip Adder.

III. Background

Addition can be divided into three parts, Bitwise PG logic, Group PG logic, and Sum logic.
In first logic, Bitwise PG logic, the inputs are A and B, the output is G (G=A•B) and P (P=A ⊕B). In third stage, inputs are Gi:j and C, the output is S (S=C ⊕Gi:j), and S is the result of this bit. The first and the third stage to four kinds of adders in this project are same. The only difference among these adders is the second stage. [2] Fig.1 is the three parts that professor introduced in class.

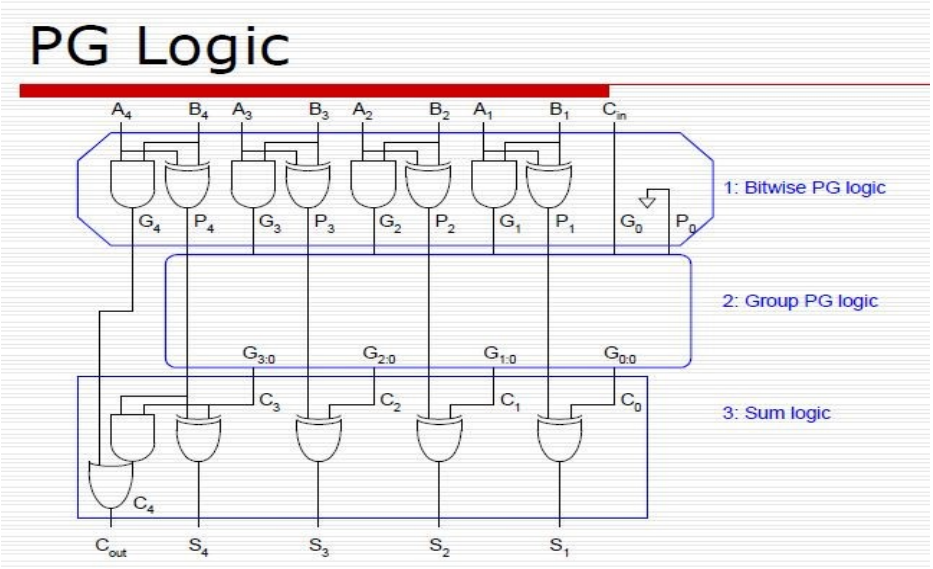


Figure 1

1. Carry Ripple Adder

Fig.2 shows an 8-bit Carry Ripple Adder Diagram, which is mentioned by professor in lecture.

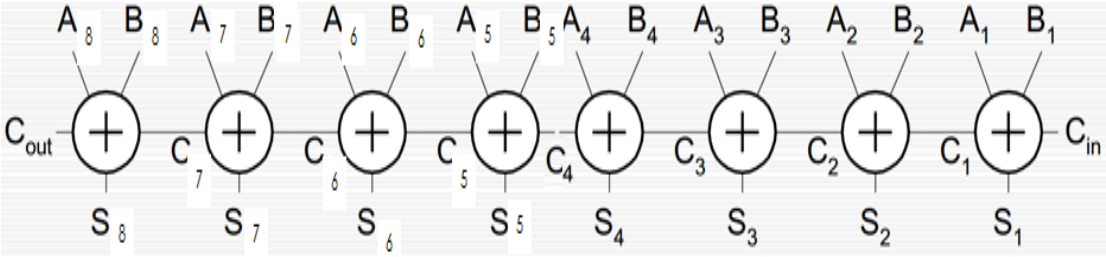


Figure 2 8-bit Carry Ripple Adder

Fig.3 shows a 16-bit Carry Ripple Adder PG Diagram, which is mentioned by professor in lecture. [5]

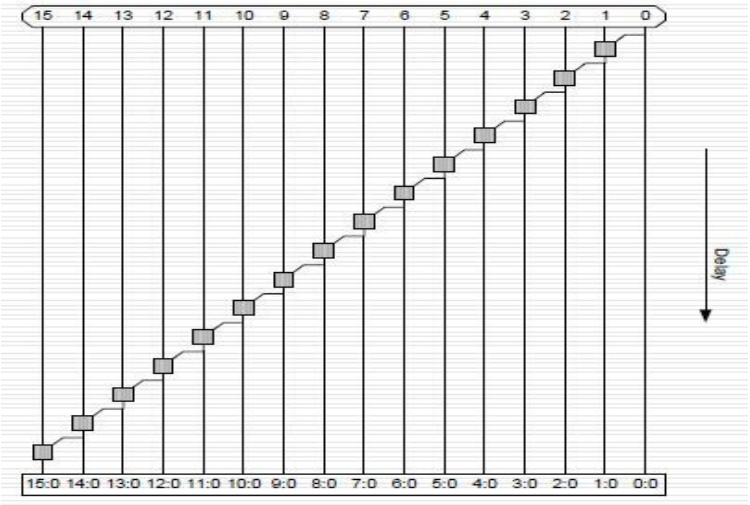


Figure 3 16-bit Carry Ripple Adder PG Diagram

Fig.2 demonstrates that for Carry Ripple Adder, the logic gate of each bit in Group PG logic is just a compound gate of an AND gate and an OR gate. And the architecture of each bit is just same to its previous one. That means each stage will receive carry in from previous stage and calculated and then sends carry out to the next stage and the next stage will not start to calculate until carry-in data came from last stage. Consequently, adder designed in this way has a large delay time. [2]

2. Carry Skip Adder.

Figure 3 is an 8-bit of Carry Skip Adder given by introduction of the project. And Figure 4 is the 16-bit of Carry Skip Adder. [5]

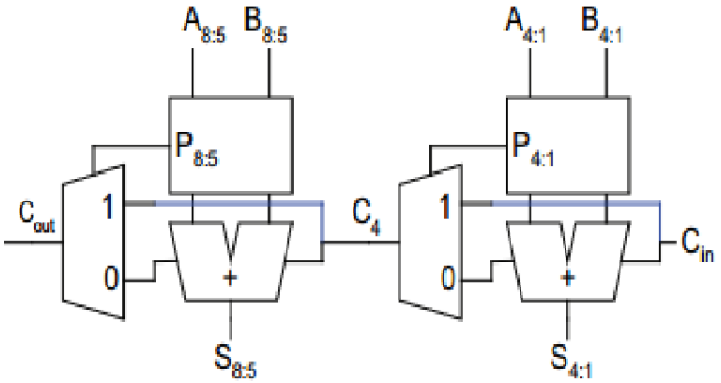


Figure4 8-bit Carry Skip Adder

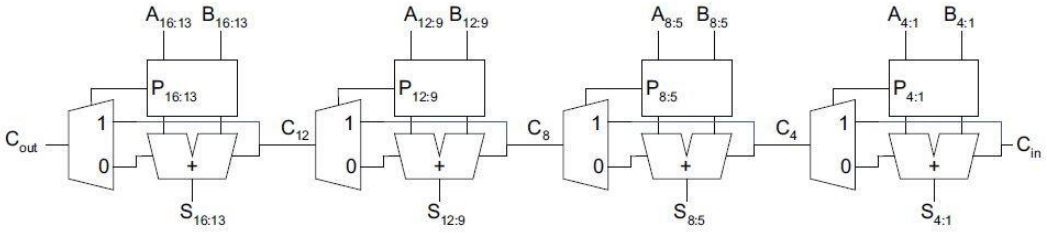


Figure5 16-bit Carry Skip Adder

This kind of adder is designed with a MUX to determine whether send the carry-in data from previous stage directly. The box P4:1, for example, is just a 4-input AND gate to calculate P4•P3•P2•P1. And all of P value of every bit had already calculated in the Bitwise PG logic. If P4:1 is 1, which means A+B=1111, the carry-in of this 4bits will be directly sent to the next stage. And box under box P4:1 is a 4-bit Carry Ripple Adder. If P P4:1 is 0, the carry-out of 4-bit Carry Ripple Adder will be sent as the carry-out of this stage. And the structure of next 4-bit stage is just fellow its previous one. That is to say, a 32-bit Carry Skip Adder can be designed in 8 stages and an 8-bit Carry Skip Adder can be assembled with 2 stages. This adder prevents the worst delay of Carry Ripple Adder. It is just like an improved Carry Ripple Adder.

3. Carry Select Adder

Fig.6 is an 8-bit Carry-Select Adder which is introduced in introduction of the project.

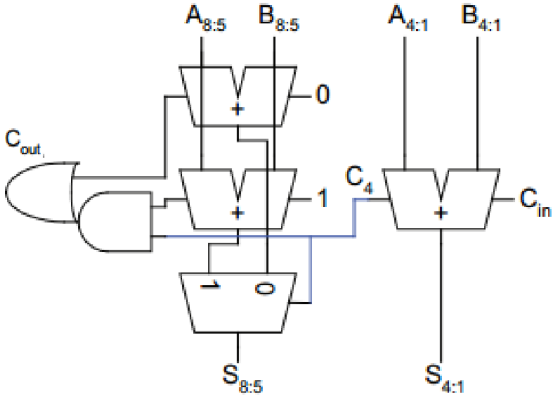


Figure 6 8-bit Carry-Select Adder

Fig.7 is a 16-bit Carry-Select Adder which is introduced in introduction of the project. [1]

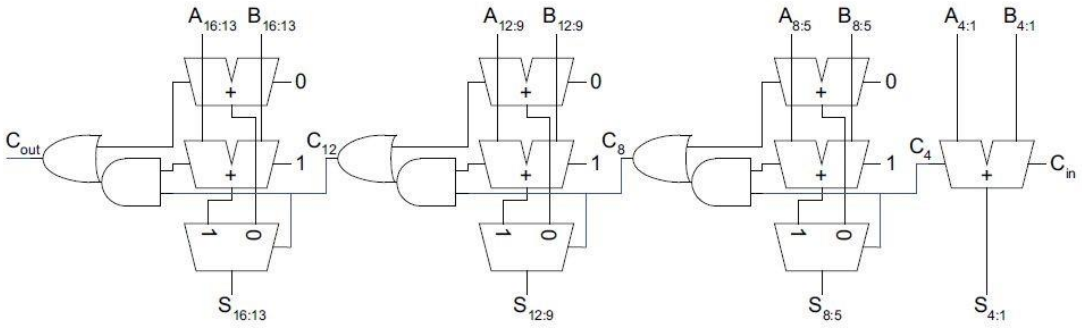


Figure 7 16-bit Carry-Select Adder

This adder is a bit different with former two adders. Fig.4 indicates that we can assemble 4-bits into one stage, and each stage (except the first stage) can be calculated at the same time. And at the same time, each stage (except the first stage) will be calculated twice, once calculated with an “assumed” carry-in “1” and the other one calculated with “assumed” carry-in “0”. And these two calculations are completed by two 4-bit Carry Ripple Adders. These two results will be selected by the “real” carry-in propagated by previous stage. And the selected result will be the out-put of this stage and the result of this stage. Start from the second 4-bit stage, the structure of next 4-bit stage is just fellow its previous one. [2] Carry-Select Adder can largely save delay time but at the same time it increase area and power consumption.

4. Prefix Adder: Kogge-Stone

For wide adders (roughly, N> 16 bits), the delay of carry-lookahead (or carry-skip or carry-select) adders becomes dominated by the delay of passing the carry through the lookahead stages. This delay can be reduced by looking ahead across the lookahead blocks. In general, a multilevel tree of look-ahead structures can be constructed to achieve delay that grows with log N. [2]

There are many ways to build the lookahead tree that offer trade-offs among the number of stages of logic, the number of logic gates, the maximum fanout on each gate,
and the amount of wiring between stages. Three fundamental trees are the Brent-Kung, Kogge-Stone, and Kogge-Stone architectures.

Fig.8 indicates an 8-bit Kogge-Stone Adder which mentioned in introduction of the project. [3]

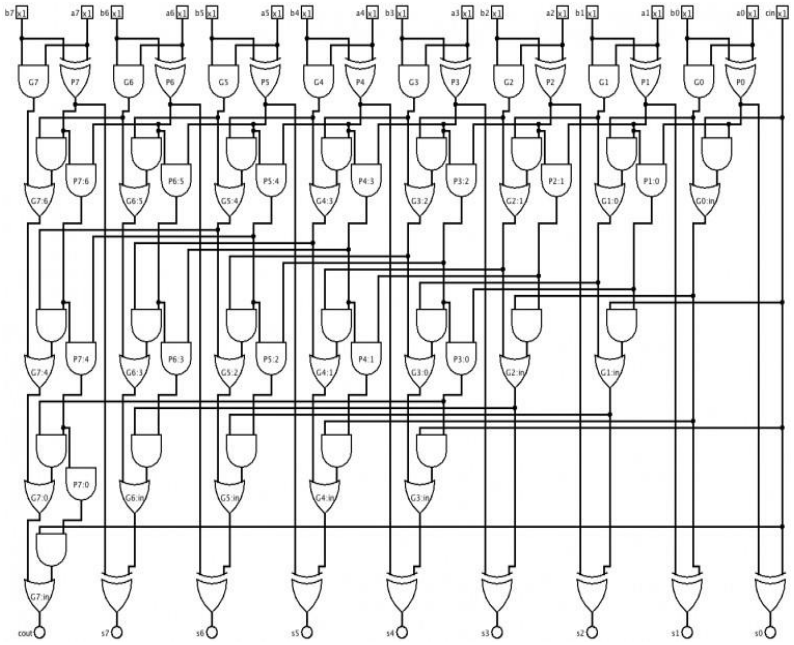


Figure 8 8-bit Kogge-Stone adder

The Kogge-Stone tree achieves both log2 N stages and fanout of 2 at each stage. This comes at the cost of many long wires that must be routed between stages. The tree also contains more PG cells; while this may not impact the area if the adder layout is on a regular grid, it will increase power consumption. Despite these costs, the Kogge-Stone tree is widely used in high-performance 32-bit and 64-bit adders.

Fig.9 indicates an 16-bit Kogge-Stone Adder which mentioned in introduction of the project. [1]

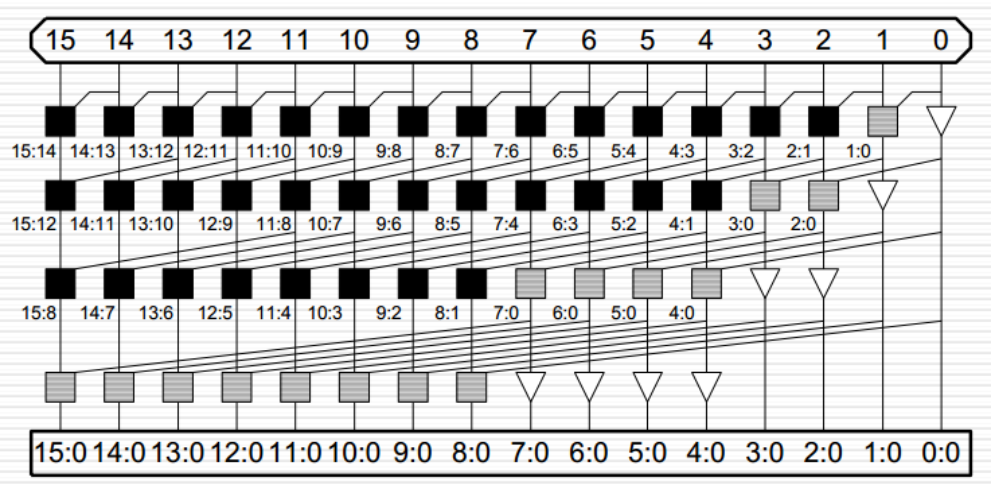


Figure 9 16-bit Kogge-Stone adder

The black cell and gray cell and buffer elements in Fig.8 and Fig.9 are used to assemble the group G and P functions. Fig.10 demonstrates the compound gates that black and grey cell stand for.

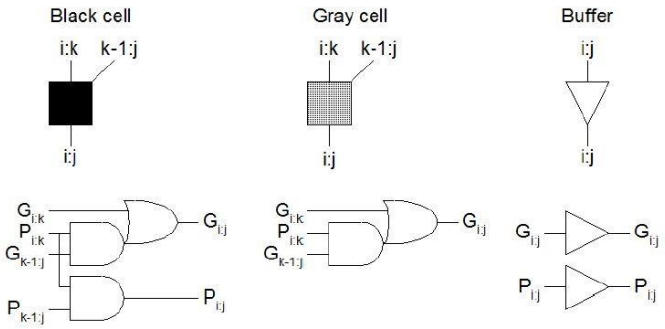


Figure 10

In black cell, Gi:k and Pi:k means the input G and P of this cell from the vertical line. Gk-1:j and Pk-1:j means the input G and P of this cell from the oblique line. Gi:j and P i:j means the output G and P of this cell. Professor mentioned the structure of 32-bit Kogge-Stone Adder is like cascade in the class. There are 3 lines components in 8-bit Kogge-Stone Adder. In 32-bit Adder, there will be 5 lines.

IV. Architectural Exploration of Adders

1) Carry Ripple Adder

a) 8-bit Carry Ripple Adder

The Verilog code is posted on blackboard. Firstly, it defines a module for 1 bit to get the result and carry-out.

With the top module:

```
module adder8(s, co, a, b, ci);
```

Since the architecture of each bit is just same to its previous one so it can be used 8 times to get the result and carry-out for 8-bit Carry Ripple Adder.

b) 32-bit Carry Ripple Adder

We used the 8-bit Carry Ripple Adder which already designed before to build our 32-bit Carry Ripple Adder. Therefore adder8 module are used 4 times to get the result and carry-out for 32-bit Carry Ripple Adder. This design is better than using the 4-bit adder module 32 times to build the 32-bit Carry Ripple Adder.

The core of 32-bit Carry Ripple Verilog file is:

```
adder8 a0(s[7:0], c7, a[7:0], b[7:0], ci);
adder8 a1(s[15:8], c15, a[15:8], b[15:8], c7);
adder8 a2(s[23:16], c23, a[23:16], b[23:16], c15);
adder8 a3(s[31:24], co, a[31:24], b[31:24], c23);
```

2) Carry Skip Adder

a) 8-bit Carry Skip Adder

We divided 8-bit Adder into two stages, and each stage has 4-bit. The 4-bit Carry Skip Adder module is built by using the module of 4-bit Carry Ripple Adder. For each stage, it should determine propagation first, the P3:0 from bit 0 to bit 3. If the propagation is 1, then just let the initial carry bit pass to the next stage. While, if the propagation is 0, the carry bit should be taken to the sum group, then go to next stage.

Following the logic function of 4-bit adder group:

c4=G3+G2*P3+G1*P2*P3+G0*P1*P2*P3+c0*P0*P1*P2*P3,

we defined the module with code:

```
wire [3:0] g,p;
wire c41, c42, c43, c44;
base ba0(g[0],p[0],a[0],b[0]);
    base ba1(g[1],p[1],a[1],b[1]);
    base ba2(g[2],p[2],a[2],b[2]);
    base ba3(g[3],p[3],a[3],b[3]);

and(c41, g[2], p[3]);
and(c42, g[1], p[2], p[3]);
and(c43, g[0], p[1], p[2], p[3]);
and(c44, ci, p[0], p[1], p[2], p[3]);

or(co, g[3], c41, c42, c43, c44);
```

b) 32-bit Carry Skip Adder

Since we have defined the module for 8-bit Carry Skip Adder, we used that module 4 times to build the 32-bit Carry Skip Adder which is faster than using 4-bit Carry Skip Adder 8 times to accomplish the 32-bit Carry Skip Adder.

3) Carry-Select Adder

a) 8-bit Carry-Select Adder

For the Carry Select Adder, we firstly created two different 4 bit adder modules. The first 4 bit adder module is the ordinary carry ripple one. The second one is the special 4 bit adder for Carry-select. We called it “adder4_csl”. It computes two possible outputs for fixed carry in 0 and 1 using “adder4”. Then it waits for the “real” carry in to arrive and then its sum bits are selected by the mux2 module bit by bit. The fundamental part is shown below:

```
adder4 a0(s0[3:0], co0, a[3:0], b[3:0], 1'b0);
adder4 a1(s1[3:0], co1, a[3:0], b[3:0], 1'b1);

and(o0, co1, ci);
or(co, o0, co0);

mux2 m0(ci, s[0], s1[0], s0[0]);
mux2 m1(ci, s[1], s1[1], s0[1]);
mux2 m2(ci, s[2], s1[2], s0[2]);
mux2 m3(ci, s[3], s1[3], s0[3]);
```

In the top module “adder8_csl”- module adder8_csl(s, co, a, b, ci), it calls the following two modules:

```
adder4 a0(s[3:0], c4, a[3:0], b[3:0], ci);
adder4_csl as0(s[7:4], co, a[7:4], b[7:4], c4);
```

b) 32-bit Carry-Select Adder

The 32 bit carry select adder is somewhat different from the previous two 32 bit adders. Previously, we simply connected 4 of 8 bit modules into a 32 bit adder. But this time, we connect 8 of 4 bit modules. In the top module “adder32_csl”, the core is shown below:

```
adder4 a0(s[3:0], c4, a[3:0], b[3:0], ci);
adder4_csl as1(s[7:4], c8, a[7:4], b[7:4], c4);
adder4_csl as2(s[11:8], c12, a[11:8], b[11:8], c8);
adder4_csl as3(s[15:12], c16, a[15:12], b[15:12], c12);
adder4_csl as4(s[19:16], c20, a[19:16], b[19:16], c16);
adder4_csl as5(s[23:20], c24, a[23:20], b[23:20], c20);
adder4_csl as6(s[27:24], c28, a[27:24], b[27:24], c24);
adder4_csl as7(s[31:28], co, a[31:28], b[31:28], c28);
```

4) Kogge-stone Adder

a) 8-bit Kogge-stone Adder Firstly, we created the modules for black and grey cells. In case of occupying too much space, we attached the code in the Appendix A. Since Kogge-Stone has a “cascade” structure, we placed the components line by line.

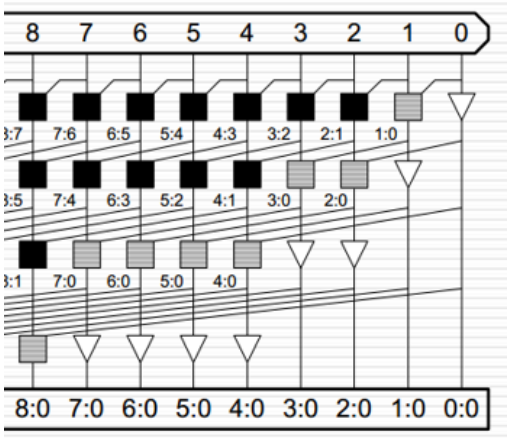


Figure 11

Fig.11 shows an 8-bit Kogge-Stone Adder.

On the base level, we computed all the P0-P7 and G0-G7. Partial code of this level is shown below:

```
base bs0(P[0],G[0],a[0],b[0]);
```

On the first level, we computed G0:in,G1:0, G2:1...G7:6 and P1:0,P2:1...P7:6, takomg base G0-G7 and P0-P7 and ci. There are 1 grey cell and 7 black cells. Partial code of this level is shown below:

```
grey l1_0(G1[0],G[0],P[0],ci);
black l1_1(G1[1],P1[1],G[1],P[1],G[0],P[0]);
```

On the second level, we computed G1:in, G2:in, G3:0, G4:1, G5:2,G6:3,G7:4 and P3:0, P4:1, P5:2, P6:3, P7:4 with 2 grey cells and 5 black cells.

On the third level, we computed G3:in, G4:in, G5:in, G6:in, G7:0 and P7:0 with 4 grey cells and 1 black cell.

The last level is to compute the carry out. Code is shown below:

```
grey l4_7(co,G3[7],P3[7],ci);
```

Finally, we have the block for sum bits. S0 and s1 computation is shown below:

```
xor s0(s[0],P[0],ci);
xor s1(s[1],P[1],G1[0]);
```

b) 32-bit Kogge-Stone Adder

32-bit Kogge-stone adder is an extended version of 8 bit Kogge-stone adder. There is one base level, 5 internal logic levels to group G and P, and one level to compute carry out.

The black cells in each level are decreasing and the grey cells are increasing. Following this trend and the specific number of black cells, grey cells, we created the 32-bit Kogge-Stone Adder Verilog code from modifying the 8 bit one.

Please see more detail in our original Verilog code in Appendix A. Both 8 bit and 32 bit KoggeStone adder files are in full commented.

V. Multiplier Design

a) Introduction of the multiplier

Fig.12 shows the 4×4 array multiplier. It makes up with CSA Array and CPA.

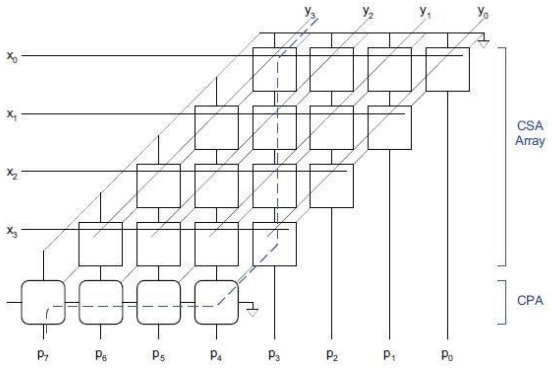
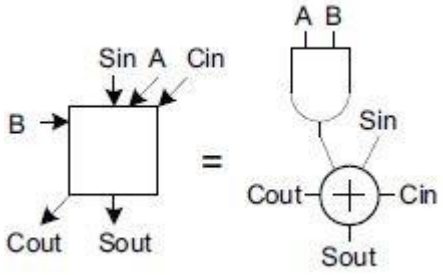


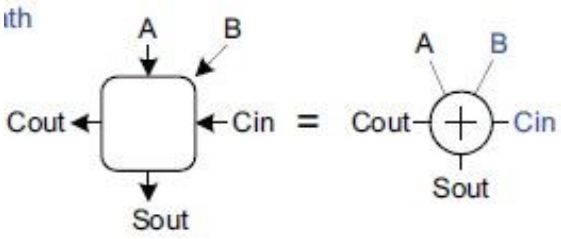
Figure 12

Firstly, I wrote the code for two basic cells for 16×16 array multiplier. Fig.13 and Fig.14 indicate the CSA cell and CPA cell.




```
module CSA(so,co,A,B,si,ci);
```

Figure 13




```
module CPA(s, co, a, b, ci);
```

Figure 14

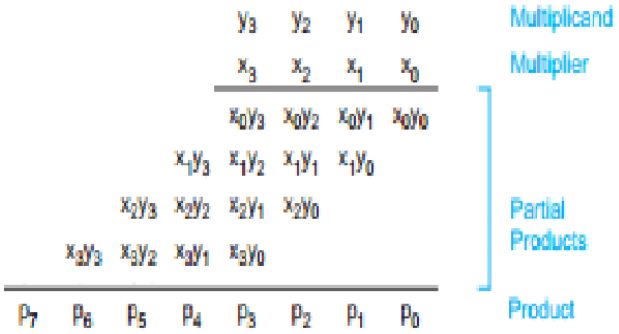


Figure 15 4×4 Partial Products in a 4 × 4-bit Multiplier

b) Implementation of 16×16 array multiplier

For each CSA, inputs A and B are x and y in corresponding line and row. Carry-out in previous line is the carry-in for next line. And the result p for each bit is addition of the S-outs of CSAs in that row. But what should be paid attention to is that the S-in and carry-in of the first line (from up to down) in the array multiplier is 0. And the S-in of the last row (from right to left) in array multiplier is also 0. In order to module the 16×16 array multiplier, firstly we tried to make a 4×4 array multiplier and modified the "test.v" document, which is used to test adder, to test whether it can operate correctly. Figure 16 shows the main part of our 4x4bit multiplier. There are CSA modules for 0-3 levels and the fourth level is consist of all CPAs. [2]

```
CSA m00(s[0],c0[0],a[0],b[0],1'b0,1'b0);
CSA m01(s0[0],c0[1],a[1],b[0],1'b0,1'b0);
CSA m02(s0[1],c0[2],a[2],b[0],1'b0,1'b0);
CSA m03(s0[2],c0[3],a[3],b[0],1'b0,1'b0);
CPA m44(s[4],c4[0],c3[0],s3[0],1'b0);
CPA m45(s[5],c4[1],c3[1],s3[1],c4[0]);
CPA m46(s[6],c4[2],c3[2],s3[2],c4[1]);
CPA m47(s[7],co,c3[3],1'b0,c4[2]);
```

Figure 16 Core of 4x4 bit array multiplier

After successfully made 4×4 array multiplier, we got the idea of how a multiplier works. 16x16 bit multiplier is simply an extended version of 4x4, with 16 time's larger size of matrix than the 4x4 one. Vertically, we extended the 4 bit one into 16 inputs. Horizontally, we assigned CSA computation to 0-15 levels and CPA to the level 16.

The Verilog code of 4x4 bit and 16×16 array multiplier are attached in Appendix B.

c) Test bench

```
a = 65535;
b = 65535;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
a = 13;
b = 10;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
a = 5;
b = 10;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
a = 9;
b = 3;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
a = 12;
b = 7;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
a = 5;
b = 14;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
a = 4;
b = 11;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
a = 10;
b = 15;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
a = 12;
b = 12;
#100 $display("At Time: %d Sum = %d Carry = %d", $time, s,co);
```

d) Functional Validation and verification

The functional validation and verification for 16x16 array multiplier screenshots are attached in Appendix C.

VI. Functional Validation and verification

After we enter code in Verilog file, we are on the Register Transfer Level (RTL). Then we created a proper Verilog testbench for our circuit. Command "Verilog test.v adder.v" will perform the RTL simulation so that we can validate the functionality of our design on the first level.

Then we do logic synthesis and place & route stages, and we have the “adder.vh” and “final.v” files generated correspondingly. Post-synthesis simulation can be performed at this point. “Verilog gscl45nm.v test.v adder.vh” is the command. This can further test the functionality of the “vh” file. Similarly, a post-P&R simulation can be performed using command “Verilog gscl45nm.v test.v final.v”.

Finally, to verify the “final.v” file matches to our design, we used Formality ESP to perform equivalence checking. Formality takes our “adder.v” file, which is created in RTL step, and “final.v” to check if they actually match.

After running RTL simulations and Equivalence Checking, it can be guaranteed that all the adders designed can run correct functions and there are no bugs in the design. The Equivalence checking is also success. The screenshots of testing results are shown in Appendix C.

VII. Synthesis Results

All the models are built based on the standard cell based ASIC design flow using tools and libraries from various vendors. We used the OSU standard cell library from FreePDK45 to implement the designs. In the first step, we did RTL simulation between our testbench and designed circuit in Verilog. Then we synthesized the design using the Synopsys Design Compiler and then performed place and route using the Cadence Encounter Digital Implementation System.

a) RTL simulation

This is the initial level of our design. We come up with idea and enter code in Verilog. This level is called the Register Transfer Level (RTL). We model our design using clocked registers, datapath elements and control elements. We use Cadence VerilogXL to simulate our design. We also need to create a testbench to test our circuit. This level validates the functionality of our design but just in an abstract way.

b) Logic synthesis

Once we verify that your Verilog RTL code is working correctly we can synthesize it into standard cells.

We used the Synopsys Design Compiler for logic synthesis. Since a hardware design requires not only the Verilog descriptions but also the specifications, we used a script file to automate the synthesis task. [4] The template file is provided as 'compile_dc.tcl'.

It generates a gatelevel Verilog netlist that only contains interconnected standard cells. From the outside it is exactly the same circuit as we coded on the RTL level. But on the inside all

functionality is now expressed only in terms of standard cells. A post-synthesis simulation can be performed by including Verilog models of the standard cells available from 'gscl45nm.v'.

c) Place & Route

Since standard cells' layouts are provided, we can now place and route them for the final layout. We first use Encounter to place the standard cells and to route the interconnects. The output is the placement of the cells and the metal interconnects.

Similar to logic synthesis using DC, we use a set of script files to automate the placement and route task.

After we run Encounter, "encounter init encounter.tcl", a bunch of files are generated. "final.v" contains an equivalent Verilog model the placed circuit. From the outside it is exactly the same circuit as we coded on the RTL level. But on the inside additional buffers are introduced to reduce interconnect delays and enhance clocking robustness. A post-P&R simulation can also be performed.

d) Performance Result

After all the above steps, we collected performance results from all the adders. Their delay time, power consumption and area information are assembled into the table below.

	Carry Ripple Adder		Carry Skip Adder		Carry Select Adder		Kogge-Stone Adder	
	8-bit	32-bit	8-bit	32-bit	8-bit	32-bit	8-bit	32-bit
Delay (ns)	1.173	3.983	0.974	3.065	0.869	2.247	0.879	1.296
Power (mW)	0.03154	0.1497	0.04475	0.2013	0.05109	0.2976	0.06385	0.3987
Area (um^2)	131.4	525.6	184.0	735.9	192.4	952.7	245.0	1378.8

As can be seen from the data above, the delay time for 32-bit adder are obvious greater than the 8-bit. Moreover, in terms of power consumption and area, the 32-bit adders are also much larger the 8-bit adders, especially for the Carry Select Adder and Kogge-Stone Adder.

When comparing the four types of 8-bit adders, there are not much difference in terms of delay time. However, when we increase the bit number to 32, there are much difference in the delay time. The Carry Ripple is the simplest design, but also the slowest one. Even though Carry Skip Adder and Carry Select Adder are the improved design for the Carry Ripple Adder, but still cannot decrease the delay time efficiently. However, the Kogge-Stone can realize it. Although the bit numbers increasing from 8 bit to 32 bit, the delay time doesn't increase much significantly, actually, it is smaller than the delay time of 8 bit Carry Ripple Adder. Therefore, if we just look at the delay time, the Kogge-Stone is the fastest design among these types. But the Kogge-Stone

also has its own disadvantages, the problem for this design is the long wire tack. It will more serious as the bit number increasing dramatically.

16x16 bit multiplier performance result is shown as below:

Multiplier	Delay (ns)	Power (mW)	Area (um2)
16x16 bit	4.452	3.597	4564.9

VIII. Conclusion and Future Work

From the performance result table, we can conclude that for a certain bit number, Kogge-Stone Adder has the shortest delay time than other adders, and its area and power consumption is acceptable when compared to the other adder. In addition, when the input bit number is small, like 8 bits adder, there are not much difference between different types of adder, especially in terms of delay time. However, when the bit number is large, like 32 bits, the differences are obvious. The delay time for 32-bit Kogge-Stone Adder are much less than the 32-bit Carry Ripple Adder.

In addition, to design the 32-bit adder, we find that using the 8-bit adder is 4 times better than using the 4-bit adder 8 times. It is faster to use the 8-bit adder, so all our 32-bit adders are based on the design of 8-bit adders.

Since there are still some other types of adder design, like Sklansky and Brent Kung, we can also try to design a 32-bit Sklansky adder or 32-bit Brent Kung adder, and compare them with the ones we already finished to find the most efficient one. We believe, with the development of the technology, the performance of adders design can be improved, and a model close to the most ideal one can be created in the future.

IX. References

[1] Introduction of the project
[2] Textbook: CMOS VLSI Design a Circuits and Systems Perspective
[3] 8-bit Kogge-Stone Adder Diagram:
http://venividiwiki.ee.virginia.edu/mediawiki/index.php/ClassECE6332Fall12Group-Fault-Tolerant_Reconfigurable_PPA
[4] Tutorial IV
[5] Lecture PPT adders

X. Appendix A

a) Verilog Code of the 8-bit adders:

1) 8 bit Carry Ripple Adder

```
module adder(s, co, a, b, ci);

        output s,co;
        input a, b, ci;

        wire o0, o1, o2;
        xor(s, a, b, ci);

        or(o0, a, b);
        or(o1, b, ci);
        or(o2, ci, a);
        and(co, o0, o1, o2);

endmodule

module adder8(s, co, a, b, ci);
        output [7:0] s;
        output co;
        input [7:0] a,b;
        input ci;

        wire c1, c2, c3, c4, c5, c6, c7;

        adder a0(s[0], c1, a[0], b[0], ci);
        adder a1(s[1], c2, a[1], b[1], c1);
        adder a2(s[2], c3, a[2], b[2], c2);
        adder a3(s[3], c4, a[3], b[3], c3);
        adder a4(s[4], c5, a[4], b[4], c4);
        adder a5(s[5], c6, a[5], b[5], c5);
        adder a6(s[6], c7, a[6], b[6], c6);
        adder a7(s[7], co, a[7], b[7], c7);
endmodule
```

2) 8 bit Carry Skip Adder

```
module base(g,p,a,b);
        output g,p;
```

```
        input a,b;

        and(g, a,b);

    xor(p, a, b);

endmodule

module adder_cs(s, co, a, b, ci);

    output s, co;
    input a, b, ci;

    wire o0, o1, o2;
    xor(s, a, b, ci);

    or(o0, a, b);
    or(o1, b, ci);
    or(o2, ci, a);
    and(co, o0, o1, o2);

endmodule

module adder4_cs(s, co, a, b, ci);
    output [3:0] s;
    output co;
    input [3:0] a,b;
    input ci;

    wire [3:0] g,p;
    wire c41, c42, c43, c44;

    wire c1, c2, c3, c4;

    base ba0(g[0],p[0],a[0],b[0]);
        base ba1(g[1],p[1],a[1],b[1]);
        base ba2(g[2],p[2],a[2],b[2]);
        base ba3(g[3],p[3],a[3],b[3]);

    and(c41, g[2], p[3]);
```

```
    and(c42, g[1], p[2], p[3]);
    and(c43, g[0], p[1], p[2], p[3]);
    and(c44, ci, p[0], p[1], p[2], p[3]);

    or(co, g[3], c41, c42, c43, c44);

    adder_cs a0(s[0], c1, a[0], b[0], ci);
    adder_cs a1(s[1], c2, a[1], b[1], c1);
        adder_cs a2(s[2], c3, a[2], b[2], c2);
        adder_cs a3(s[3], c4, a[3], b[3], c3);

endmodule

module adder8_cs(s, co, a, b, ci);
        output [7:0] s;
        output co;
        input [7:0] a,b;
        input ci;

        wire c4;

        adder4_cs a0(s[3:0], c4, a[3:0], b[3:0], ci);
        adder4_cs a1(s[7:4], co, a[7:4], b[7:4], c4);

endmodule
```

3) 8 bit Carry Select Adder

```
module adder(s, co, a, b, ci);

        output s, co;
        input a, b, ci;

        wire o0, o1, o2;
        xor(s, a, b, ci);

        or(o0, a, b);
        or(o1, b, ci);
        or(o2, ci, a);
        and(co, o0, o1, o2);
endmodule
```

```
module mux2(select, co, c1, c2);
	output co;
	input select, c1, c2;

	wire o1;

	and(o1, select, c1);

	or(co, o1, c2);

endmodule

module adder4_csl(s, co, a, b, ci);
	output [3:0] s;
	output co;
	input [3:0] a,b;
	input ci;

	wire [3:0] s0,s1;
	wire o0;

	adder4 a0(s0[3:0], co0, a[3:0], b[3:0], 1'b0);
	adder4 a1(s1[3:0], co1, a[3:0], b[3:0], 1'b1);

	and(o0, co1, ci);
	or(co, o0, co0);

	mux2 m0(ci, s[0], s1[0], s0[0]);
	mux2 m1(ci, s[1], s1[1], s0[1]);
	mux2 m2(ci, s[2], s1[2], s0[2]);
	mux2 m3(ci, s[3], s1[3], s0[3]);
endmodule

module adder4(s, co, a, b, ci);
	output [3:0] s;
	output co;
	input [3:0] a,b;
	input ci;
```

```
        wire c1, c2, c3;

        adder a0(s[0], c1, a[0], b[0], ci);
        adder a1(s[1], c2, a[1], b[1], c1);
        adder a2(s[2], c3, a[2], b[2], c2);
        adder a3(s[3], co, a[3], b[3], c3);

endmodule

module adder8_csl(s, co, a, b, ci);
        output [7:0] s;
        output co;
        input [7:0] a,b;
        input ci;

        wire c4;

        adder4 a0(s[3:0], c4, a[3:0], b[3:0], ci);
        adder4_csl as0(s[7:4], co, a[7:4], b[7:4], c4);

endmodule
```

4) 8 bit Kogge-Stone Adder

```
//main module, a 8 bit kogge-stone adder. very fast
module adder8_ks(s, co, a, b, ci);
        output [7:0] s;
        output co;
        input [7:0] a,b;
        input ci;

        wire [7:0] P,G,P1,G1,P2,G2,P3,G3;

        //base level computation, to compute P0-P7, G0-G7

        base bs0(P[0],G[0],a[0],b[0]);
        base bs1(P[1],G[1],a[1],b[1]);
        base bs2(P[2],G[2],a[2],b[2]);
        base bs3(P[3],G[3],a[3],b[3]);
        base bs4(P[4],G[4],a[4],b[4]);
```

```
base bs5(P[5],G[5],a[5],b[5]);
base bs6(P[6],G[6],a[6],b[6]);
base bs7(P[7],G[7],a[7],b[7]);

//level 1 computation, compute G0:in,G1:0, G2:1...G7:6 as G1[0],G1[1]...G1[7]
//and P1:0,P2:1...P7:6 as P1[1],P1[2],...P1[7]
//takes base G0-G7 and P0-P7 and ci

grey l1_0(G1[0],G[0],P[0],ci);
black l1_1(G1[1],P1[1],G[1],P[1],G[0],P[0]);
black l1_2(G1[2],P1[2],G[2],P[2],G[1],P[1]);
black l1_3(G1[3],P1[3],G[3],P[3],G[2],P[2]);
black l1_4(G1[4],P1[4],G[4],P[4],G[3],P[3]);
black l1_5(G1[5],P1[5],G[5],P[5],G[4],P[4]);
black l1_6(G1[6],P1[6],G[6],P[6],G[5],P[5]);
black l1_7(G1[7],P1[7],G[7],P[7],G[6],P[6]);

//level 2 computation,
// compute G1:in, G2:in, G3:0, G4:1, G5:2,G6:3,G7:4 as G2[1],G2[2]...G2[7]
// compute P3:0,P4:1,P5:2,P6:3,P7:4 as P2[3],P2[4],P2[5],P2[6],P2[7]
// takes ....
grey l2_1(G2[1],G1[1],P1[1],ci);
grey l2_2(G2[2],G1[2],P1[2],G1[0]);
black l2_3(G2[3],P2[3],G1[3],P1[3],G1[1],P1[1]);
black l2_4(G2[4],P2[4],G1[4],P1[4],G1[2],P1[2]);
black l2_5(G2[5],P2[5],G1[5],P1[5],G1[3],P1[3]);
black l2_6(G2[6],P2[6],G1[6],P1[6],G1[4],P1[4]);
black l2_7(G2[7],P2[7],G1[7],P1[7],G1[5],P1[5]);

//level 3 computation
// compute G3:in, G4:in, G5:in, G6:in G7:0 as G3[3],G3[4],G3[5],G3[6],G3[7]
// compute P7:0 as P3[7];
// takes ....

grey l3_3(G3[3],G2[3],P2[3],ci);
grey l3_4(G3[4],G2[4],P2[4],G1[0]);
grey l3_5(G3[5],G2[5],P2[5],G2[1]);
grey l3_6(G3[6],G2[6],P2[6],G2[2]);
black l3_7(G3[7],P3[7],G2[7],P2[7],G2[3],P2[3]);
```

```
	//level 4 computation, compute G7:in as co, takes G7:0 as G3[7], and P7:0 as P3[7]
and ci
	grey l4_7(co,G3[7],P3[7],ci);

	//sum bit
	xor s0(s[0],P[0],ci);
	xor s1(s[1],P[1],G1[0]);
	xor s2(s[2],P[2],G2[1]);
	xor s3(s[3],P[3],G2[2]);
	xor s4(s[4],P[4],G3[3]);
	xor s5(s[5],P[5],G3[4]);
	xor s6(s[6],P[6],G3[5]);
	xor s7(s[7],P[7],G3[6]);

endmodule

//to compute base p and g
module base(p,g,a,b);
	output p,g;
	input a,b;

	and(g,a,b);
	xor(p,a,b);

endmodule

//grey cell, takes Gi:k as Gik, Pi:k as Pik, Gk-1:j as Gk1j,output Gi:j as Gij
module grey(Gij,Gik,Pik,Gk1j);
	output Gij;
	input Gik,Pik,Gk1j;

	wire i0;

	and(i0,Pik,Gk1j);
	or(Gij,Gik,i0);

endmodule
//Black cell, takes Gi:k as Gik, Pi:k as Pik, Gk-1:j as Gk1j,Pk-1:j as Pk1j, output Gi:j as Gij, Pi:j as
Pij
module black(Gij,Pij,Gik,Pik,Gk1j,Pk1j);
```

```
        output Gij,Pij;
        input Gik,Pik,Gk1j,Pk1j;

        wire i0;

        and(i0,Pik,Gk1j);
        or(Gij,Gik,i0);

        and(Pij,Pik,Pk1j);

endmodule
```

b) Verilog Code of the 32-bit adders:

1) 32 bit Carry Ripple Adder

```
module adder(s, co, a, b, ci);

        output s,co;
        input a, b, ci;

        wire o0, o1, o2;
        xor(s, a, b, ci);

        or(o0, a, b);
        or(o1, b, ci);
        or(o2, ci, a);
        and(co, o0, o1, o2);

endmodule

module adder8(s, co, a, b, ci);
        output [7:0] s;
        output co;
        input [7:0] a,b;
        input ci;

        wire c1, c2, c3, c4, c5, c6, c7;

        adder a0(s[0], c1, a[0], b[0], ci);
        adder a1(s[1], c2, a[1], b[1], c1);
        adder a2(s[2], c3, a[2], b[2], c2);
```

```
        adder a3(s[3], c4, a[3], b[3], c3);
        adder a4(s[4], c5, a[4], b[4], c4);
        adder a5(s[5], c6, a[5], b[5], c5);
        adder a6(s[6], c7, a[6], b[6], c6);
        adder a7(s[7], co, a[7], b[7], c7);
endmodule

module adder32(s, co, a, b, ci);
        output [31:0] s;
        output co;
        input [31:0] a,b;
        input ci;

        wire c7, c15, c23 ;

        adder8 a0(s[7:0], c7, a[7:0], b[7:0], ci);
        adder8 a1(s[15:8], c15, a[15:8], b[15:8], c7);
        adder8 a2(s[23:16], c23, a[23:16], b[23:16], c15);
        adder8 a3(s[31:24], co, a[31:24], b[31:24], c23);

endmodule
```

2) 32 bit Carry Skip Adder

```
module base(g,p,a,b);
        output g,p;
        input a,b;

        and(g, a,b);

    xor(p, a, b);

endmodule

module adder_cs(s, co, a, b, ci);

    output s, co;
    input a, b, ci;

    wire o0, o1, o2;
    xor(s, a, b, ci);
```

```
    or(o0, a, b);
    or(o1, b, ci);
    or(o2, ci, a);
    and(co, o0, o1, o2);

endmodule

module adder4_cs(s, co, a, b, ci);
    output [3:0] s;
    output co;
    input [3:0] a,b;
    input ci;

    wire [3:0] g,p;
    wire c41, c42, c43, c44;

    wire c1, c2, c3, c4;

    base ba0(g[0],p[0],a[0],b[0]);
        base ba1(g[1],p[1],a[1],b[1]);
        base ba2(g[2],p[2],a[2],b[2]);
        base ba3(g[3],p[3],a[3],b[3]);

    and(c41, g[2], p[3]);
    and(c42, g[1], p[2], p[3]);
    and(c43, g[0], p[1], p[2], p[3]);
    and(c44, ci, p[0], p[1], p[2], p[3]);

    or(co, g[3], c41, c42, c43, c44);

    adder_cs a0(s[0], c1, a[0], b[0], ci);
    adder_cs a1(s[1], c2, a[1], b[1], c1);
        adder_cs a2(s[2], c3, a[2], b[2], c2);
        adder_cs a3(s[3], c4, a[3], b[3], c3);
endmodule

module adder8_cs(s, co, a, b, ci);
        output [7:0] s;
        output co;
```

```
        input [7:0] a,b;
        input ci;

        wire c4;

        adder4_cs a0(s[3:0], c4, a[3:0], b[3:0], ci);
        adder4_cs a1(s[7:4], co, a[7:4], b[7:4], c4);

endmodule

module adder32_cs(s, co, a, b, ci);
        output [31:0] s;
        output co;
        input [31:0] a,b;
        input ci;

        wire c8,c16,c24;

        adder8_cs a0(s[7:0], c8, a[7:0], b[7:0], ci);
        adder8_cs a1(s[15:8], c16, a[15:8], b[15:8], c8);
        adder8_cs a2(s[23:16], c24, a[23:16], b[23:16], c16);
        adder8_cs a3(s[31:24], co, a[31:24], b[31:24], c24);

endmodule
```

3) 32 bit Carry Select Adder

```
module adder(s, co, a, b, ci);

        output s, co;
        input a, b, ci;

        wire o0, o1, o2;
        xor(s, a, b, ci);

        or(o0, a, b);
        or(o1, b, ci);
        or(o2, ci, a);
        and(co, o0, o1, o2);

endmodule
```

```
module mux2(select, co, c1, c2);
	output co;
	input select, c1, c2;

	wire o1;

	and(o1, select, c1);

	or(co, o1, c2);

endmodule

module adder4_csl(s, co, a, b, ci);
	output [3:0] s;
	output co;
	input [3:0] a,b;
	input ci;

	wire [3:0] s0,s1;
	wire o0;

	adder4 a0(s0[3:0], co0, a[3:0], b[3:0], 1'b0);
	adder4 a1(s1[3:0], co1, a[3:0], b[3:0], 1'b1);

	and(o0, co1, ci);
	or(co, o0, co0);

	mux2 m0(ci, s[0], s1[0], s0[0]);
	mux2 m1(ci, s[1], s1[1], s0[1]);
	mux2 m2(ci, s[2], s1[2], s0[2]);
	mux2 m3(ci, s[3], s1[3], s0[3]);
endmodule
module adder4(s, co, a, b, ci);
	output [3:0] s;
	output co;
	input [3:0] a,b;
	input ci;
```

```
        wire c1, c2, c3;

        adder a0(s[0], c1, a[0], b[0], ci);
        adder a1(s[1], c2, a[1], b[1], c1);
        adder a2(s[2], c3, a[2], b[2], c2);
        adder a3(s[3], co, a[3], b[3], c3);

endmodule

module adder32_csl(s, co, a, b, ci);
        output [31:0] s;
        output co;
        input [31:0] a,b;
        input ci;

        wire c4,c8,c12,c16,c20,c24,c28;

        adder4 a0(s[3:0], c4, a[3:0], b[3:0], ci);
        adder4_csl as1(s[7:4], c8, a[7:4], b[7:4], c4);
        adder4_csl as2(s[11:8], c12, a[11:8], b[11:8], c8);
        adder4_csl as3(s[15:12], c16, a[15:12], b[15:12], c12);
        adder4_csl as4(s[19:16], c20, a[19:16], b[19:16], c16);
        adder4_csl as5(s[23:20], c24, a[23:20], b[23:20], c20);
        adder4_csl as6(s[27:24], c28, a[27:24], b[27:24], c24);
        adder4_csl as7(s[31:28], co, a[31:28], b[31:28], c28);

endmodule
```

4) 32 bit KoggeStone Adder

```
//main module, a 32 bit kogge-stone adder. very fast
module adder32_ks(s, co, a, b, ci);
        output [31:0] s;
        output co;
        input [31:0] a,b;
        input ci;

        wire [31:0] P,G,P1,G1,P2,G2,P3,G3,P4,G4,P5,G5; //just for simplexity

        //base level computation, to compute P0-P31, G0-G31
```

```
base bs0(P[0],G[0],a[0],b[0]);
base bs1(P[1],G[1],a[1],b[1]);
base bs2(P[2],G[2],a[2],b[2]);
base bs3(P[3],G[3],a[3],b[3]);
base bs4(P[4],G[4],a[4],b[4]);
base bs5(P[5],G[5],a[5],b[5]);
base bs6(P[6],G[6],a[6],b[6]);
base bs7(P[7],G[7],a[7],b[7]);
base bs8(P[8],G[8],a[8],b[8]);
base bs9(P[9],G[9],a[9],b[9]);
base bs10(P[10],G[10],a[10],b[10]);
base bs11(P[11],G[11],a[11],b[11]);
base bs12(P[12],G[12],a[12],b[12]);
base bs13(P[13],G[13],a[13],b[13]);
base bs14(P[14],G[14],a[14],b[14]);
base bs15(P[15],G[15],a[15],b[15]);
base bs16(P[16],G[16],a[16],b[16]);
base bs17(P[17],G[17],a[17],b[17]);
base bs18(P[18],G[18],a[18],b[18]);
base bs19(P[19],G[19],a[19],b[19]);
base bs20(P[20],G[20],a[20],b[20]);
base bs21(P[21],G[21],a[21],b[21]);
base bs22(P[22],G[22],a[22],b[22]);
base bs23(P[23],G[23],a[23],b[23]);
base bs24(P[24],G[24],a[24],b[24]);
base bs25(P[25],G[25],a[25],b[25]);
base bs26(P[26],G[26],a[26],b[26]);
base bs27(P[27],G[27],a[27],b[27]);
base bs28(P[28],G[28],a[28],b[28]);
base bs29(P[29],G[29],a[29],b[29]);
base bs30(P[30],G[30],a[30],b[30]);
base bs31(P[31],G[31],a[31],b[31]);

//level 1 computation, compute G0:in,G1:0, G2:1...G31:30 as G1[0],G1[1]...G1[31]
//and P1:0,P2:1...P31:30 as P1[1],P1[2],...P1[31]
//takes base G0-G31 and P0-P31 and ci

grey l1_0(G1[0],G[0],P[0],ci);
black l1_1(G1[1],P1[1],G[1],P[1],G[0],P[0]);
black l1_2(G1[2],P1[2],G[2],P[2],G[1],P[1]);
```

```
        black l1_3(G1[3],P1[3],G[3],P[3],G[2],P[2]);
        black l1_4(G1[4],P1[4],G[4],P[4],G[3],P[3]);
        black l1_5(G1[5],P1[5],G[5],P[5],G[4],P[4]);
        black l1_6(G1[6],P1[6],G[6],P[6],G[5],P[5]);
        black l1_7(G1[7],P1[7],G[7],P[7],G[6],P[6]);
        black l1_8(G1[8],P1[8],G[8],P[8],G[7],P[7]);
        black l1_9(G1[9],P1[9],G[9],P[9],G[8],P[8]);
        black l1_10(G1[10],P1[10],G[10],P[10],G[9],P[9]);
        black l1_11(G1[11],P1[11],G[11],P[11],G[10],P[10]);
        black l1_12(G1[12],P1[12],G[12],P[12],G[11],P[11]);
        black l1_13(G1[13],P1[13],G[13],P[13],G[12],P[12]);
        black l1_14(G1[14],P1[14],G[14],P[14],G[13],P[13]);
        black l1_15(G1[15],P1[15],G[15],P[15],G[14],P[14]);
        black l1_16(G1[16],P1[16],G[16],P[16],G[15],P[15]);
        black l1_17(G1[17],P1[17],G[17],P[17],G[16],P[16]);
        black l1_18(G1[18],P1[18],G[18],P[18],G[17],P[17]);
        black l1_19(G1[19],P1[19],G[19],P[19],G[18],P[18]);
        black l1_20(G1[20],P1[20],G[20],P[20],G[19],P[19]);
        black l1_21(G1[21],P1[21],G[21],P[21],G[20],P[20]);
        black l1_22(G1[22],P1[22],G[22],P[22],G[21],P[21]);
        black l1_23(G1[23],P1[23],G[23],P[23],G[22],P[22]);
        black l1_24(G1[24],P1[24],G[24],P[24],G[23],P[23]);
        black l1_25(G1[25],P1[25],G[25],P[25],G[24],P[24]);
        black l1_26(G1[26],P1[26],G[26],P[26],G[25],P[25]);
        black l1_27(G1[27],P1[27],G[27],P[27],G[26],P[26]);
        black l1_28(G1[28],P1[28],G[28],P[28],G[27],P[27]);
        black l1_29(G1[29],P1[29],G[29],P[29],G[28],P[28]);
        black l1_30(G1[30],P1[30],G[30],P[30],G[29],P[29]);
        black l1_31(G1[31],P1[31],G[31],P[31],G[30],P[30]);

        //level 2 computation,
        // compute G1:in, G2:in, G3:0, G4:1, G5:2,G6:3,G7:4,...G30:27,G31:28 as
G2[1],G2[2]...G2[7]...G2[30],G2[31]
        // compute P3:0,P4:1,P5:2,P6:3,P7:4...P30:27,P31:28 as
P2[3],P2[4],P2[5],P2[6],P2[7]...P2[30],P2[31]
        // takes ....
        grey l2_1(G2[1],G1[1],P1[1],ci);
        grey l2_2(G2[2],G1[2],P1[2],G1[0]);
        black l2_3(G2[3],P2[3],G1[3],P1[3],G1[1],P1[1]);
        black l2_4(G2[4],P2[4],G1[4],P1[4],G1[2],P1[2]);
```

```
        black l2_5(G2[5],P2[5],G1[5],P1[5],G1[3],P1[3]);
        black l2_6(G2[6],P2[6],G1[6],P1[6],G1[4],P1[4]);
        black l2_7(G2[7],P2[7],G1[7],P1[7],G1[5],P1[5]);
        black l2_8(G2[8],P2[8],G1[8],P1[8],G1[6],P1[6]);
        black l2_9(G2[9],P2[9],G1[9],P1[9],G1[7],P1[7]);
        black l2_10(G2[10],P2[10],G1[10],P1[10],G1[8],P1[8]);
        black l2_11(G2[11],P2[11],G1[11],P1[11],G1[9],P1[9]);
        black l2_12(G2[12],P2[12],G1[12],P1[12],G1[10],P1[10]);
        black l2_13(G2[13],P2[13],G1[13],P1[13],G1[11],P1[11]);
        black l2_14(G2[14],P2[14],G1[14],P1[14],G1[12],P1[12]);
        black l2_15(G2[15],P2[15],G1[15],P1[15],G1[13],P1[13]);
        black l2_16(G2[16],P2[16],G1[16],P1[16],G1[14],P1[14]);
        black l2_17(G2[17],P2[17],G1[17],P1[17],G1[15],P1[15]);
        black l2_18(G2[18],P2[18],G1[18],P1[18],G1[16],P1[16]);
        black l2_19(G2[19],P2[19],G1[19],P1[19],G1[17],P1[17]);
        black l2_20(G2[20],P2[20],G1[20],P1[20],G1[18],P1[18]);
        black l2_21(G2[21],P2[21],G1[21],P1[21],G1[19],P1[19]);
        black l2_22(G2[22],P2[22],G1[22],P1[22],G1[20],P1[20]);
        black l2_23(G2[23],P2[23],G1[23],P1[23],G1[21],P1[21]);
        black l2_24(G2[24],P2[24],G1[24],P1[24],G1[22],P1[22]);
        black l2_25(G2[25],P2[25],G1[25],P1[25],G1[23],P1[23]);
        black l2_26(G2[26],P2[26],G1[26],P1[26],G1[24],P1[24]);
        black l2_27(G2[27],P2[27],G1[27],P1[27],G1[25],P1[25]);
        black l2_28(G2[28],P2[28],G1[28],P1[28],G1[26],P1[26]);
        black l2_29(G2[29],P2[29],G1[29],P1[29],G1[27],P1[27]);
        black l2_30(G2[30],P2[30],G1[30],P1[30],G1[28],P1[28]);
        black l2_31(G2[31],P2[31],G1[31],P1[31],G1[29],P1[29]);


        //level 3 computation
        // compute G3:in, G4:in, G5:in, G6:in G7:0, G8:1, G9:2...G30:23,G31:24 as
G3[3],G3[4],G3[5],G3[6],G3[7],G3[8],G3[9]...G3[30],G3[31]
        // compute P7:0,P8:0,...P30:23,P31:24 as P3[7],P3[8],P3[9]...P3[30],P3[31];
        // takes ....

        grey l3_3(G3[3],G2[3],P2[3],ci);
        grey l3_4(G3[4],G2[4],P2[4],G1[0]);
        grey l3_5(G3[5],G2[5],P2[5],G2[1]);
        grey l3_6(G3[6],G2[6],P2[6],G2[2]);
        black l3_7(G3[7],P3[7],G2[7],P2[7],G2[3],P2[3]);
```

```
        black l3_8(G3[8],P3[8],G2[8],P2[8],G2[4],P2[4]);
        black l3_9(G3[9],P3[9],G2[9],P2[9],G2[5],P2[5]);
        black l3_10(G3[10],P3[10],G2[10],P2[10],G2[6],P2[6]);
        black l3_11(G3[11],P3[11],G2[11],P2[11],G2[7],P2[7]);
        black l3_12(G3[12],P3[12],G2[12],P2[12],G2[8],P2[8]);
        black l3_13(G3[13],P3[13],G2[13],P2[13],G2[9],P2[9]);
        black l3_14(G3[14],P3[14],G2[14],P2[14],G2[10],P2[10]);
        black l3_15(G3[15],P3[15],G2[15],P2[15],G2[11],P2[11]);
        black l3_16(G3[16],P3[16],G2[16],P2[16],G2[12],P2[12]);
        black l3_17(G3[17],P3[17],G2[17],P2[17],G2[13],P2[13]);
        black l3_18(G3[18],P3[18],G2[18],P2[18],G2[14],P2[14]);
        black l3_19(G3[19],P3[19],G2[19],P2[19],G2[15],P2[15]);
        black l3_20(G3[20],P3[20],G2[20],P2[20],G2[16],P2[16]);
        black l3_21(G3[21],P3[21],G2[21],P2[21],G2[17],P2[17]);
        black l3_22(G3[22],P3[22],G2[22],P2[22],G2[18],P2[18]);
        black l3_23(G3[23],P3[23],G2[23],P2[23],G2[19],P2[19]);
        black l3_24(G3[24],P3[24],G2[24],P2[24],G2[20],P2[20]);
        black l3_25(G3[25],P3[25],G2[25],P2[25],G2[21],P2[21]);
        black l3_26(G3[26],P3[26],G2[26],P2[26],G2[22],P2[22]);
        black l3_27(G3[27],P3[27],G2[27],P2[27],G2[23],P2[23]);
        black l3_28(G3[28],P3[28],G2[28],P2[28],G2[24],P2[24]);
        black l3_29(G3[29],P3[29],G2[29],P2[29],G2[25],P2[25]);
        black l3_30(G3[30],P3[30],G2[30],P2[30],G2[26],P2[26]);
        black l3_31(G3[31],P3[31],G2[31],P2[31],G2[27],P2[27]);

        //level 4 computation, compute G7:in,G8:in,G9:in,G10:in,G11:in...G14:in,
G15:0,G16:1...G31:16 as G4[7], G4[8]...G[31]
        // compute P15:0,P16:1...P31:16 as P4[15]..P4[31]
        grey l4_7(G4[7],G3[7],P3[7],ci);
        grey l4_8(G4[8],G3[8],P3[8],G1[0]);
        grey l4_9(G4[9],G3[9],P3[9],G2[1]);
        grey l4_10(G4[10],G3[10],P3[10],G2[2]);
        grey l4_11(G4[11],G3[11],P3[11],G3[3]);
        grey l4_12(G4[12],G3[12],P3[12],G3[4]);
        grey l4_13(G4[13],G3[13],P3[13],G3[5]);
        grey l4_14(G4[14],G3[14],P3[14],G3[6]);
        black l4_15(G4[15],P4[15],G3[15],P3[15],G3[7],P3[7]);
        black l4_16(G4[16],P4[16],G3[16],P3[16],G3[8],P3[8]);
        black l4_17(G4[17],P4[17],G3[17],P3[17],G3[9],P3[9]);
        black l4_18(G4[18],P4[18],G3[18],P3[18],G3[10],P3[10]);
```

```
        black l4_19(G4[19],P4[19],G3[19],P3[19],G3[11],P3[11]);
        black l4_20(G4[20],P4[20],G3[20],P3[20],G3[12],P3[12]);
        black l4_21(G4[21],P4[21],G3[21],P3[21],G3[13],P3[13]);
        black l4_22(G4[22],P4[22],G3[22],P3[22],G3[14],P3[14]);
        black l4_23(G4[23],P4[23],G3[23],P3[23],G3[15],P3[15]);
        black l4_24(G4[24],P4[24],G3[24],P3[24],G3[16],P3[16]);
        black l4_25(G4[25],P4[25],G3[25],P3[25],G3[17],P3[17]);
        black l4_26(G4[26],P4[26],G3[26],P3[26],G3[18],P3[18]);
        black l4_27(G4[27],P4[27],G3[27],P3[27],G3[19],P3[19]);
        black l4_28(G4[28],P4[28],G3[28],P3[28],G3[20],P3[20]);
        black l4_29(G4[29],P4[29],G3[29],P3[29],G3[21],P3[21]);
        black l4_30(G4[30],P4[30],G3[30],P3[30],G3[22],P3[22]);
        black l4_31(G4[31],P4[31],G3[31],P3[31],G3[23],P3[23]);

        //level 5 computation, compute G15:in,G16:in...G30:in, G31:0 as G5[15],
G5[16]...G5[31]
        // compute P31:0 as P5[31]
        grey l5_15(G5[15],G4[15],P4[15],ci);
        grey l5_16(G5[16],G4[16],P4[16],G1[0]);
        grey l5_17(G5[17],G4[17],P4[17],G2[1]);
        grey l5_18(G5[18],G4[18],P4[18],G2[2]);
        grey l5_19(G5[19],G4[19],P4[19],G3[3]);
        grey l5_20(G5[20],G4[20],P4[20],G3[4]);
        grey l5_21(G5[21],G4[21],P4[21],G3[5]);
        grey l5_22(G5[22],G4[22],P4[22],G3[6]);
        grey l5_23(G5[23],G4[23],P4[23],G4[7]);
        grey l5_24(G5[24],G4[24],P4[24],G4[8]);
        grey l5_25(G5[25],G4[25],P4[25],G4[9]);
        grey l5_26(G5[26],G4[26],P4[26],G4[10]);
        grey l5_27(G5[27],G4[27],P4[27],G4[11]);
        grey l5_28(G5[28],G4[28],P4[28],G4[12]);
        grey l5_29(G5[29],G4[29],P4[29],G4[13]);
        grey l5_30(G5[30],G4[30],P4[30],G4[14]);
        black l5_31(G5[31],P5[31],G4[31],P4[31],G4[15],P4[15]);

        //level 6 computation, compute G31:in as co
        grey l6_31(co,G5[31],P5[31],ci);

        //sum bit
        xor s0(s[0],P[0],ci);
```

```
        xor s1(s[1],P[1],G1[0]);
        xor s2(s[2],P[2],G2[1]);
        xor s3(s[3],P[3],G2[2]);
        xor s4(s[4],P[4],G3[3]);
        xor s5(s[5],P[5],G3[4]);
        xor s6(s[6],P[6],G3[5]);
        xor s7(s[7],P[7],G3[6]);
        xor s8(s[8],P[8],G4[7]);
        xor s9(s[9],P[9],G4[8]);
        xor s10(s[10],P[10],G4[9]);
        xor s11(s[11],P[11],G4[10]);
        xor s12(s[12],P[12],G4[11]);
        xor s13(s[13],P[13],G4[12]);
        xor s14(s[14],P[14],G4[13]);
        xor s15(s[15],P[15],G4[14]);
        xor s16(s[16],P[16],G5[15]);
        xor s17(s[17],P[17],G5[16]);
        xor s18(s[18],P[18],G5[17]);
        xor s19(s[19],P[19],G5[18]);
        xor s20(s[20],P[20],G5[19]);
        xor s21(s[21],P[21],G5[20]);
        xor s22(s[22],P[22],G4[21]);
        xor s23(s[23],P[23],G4[22]);
        xor s24(s[24],P[24],G4[23]);
        xor s25(s[25],P[25],G4[24]);
        xor s26(s[26],P[26],G5[25]);
        xor s27(s[27],P[27],G5[26]);
        xor s28(s[28],P[28],G5[27]);
        xor s29(s[29],P[29],G5[28]);
        xor s30(s[30],P[30],G5[29]);
        xor s31(s[31],P[31],G5[30]);
endmodule

//to compute base p and g
module base(p,g,a,b);
        output p,g;
        input a,b;

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

```
endmodule

//grey cell, takes Gi:k as Gik, Pi:k as Pik, Gk-1:j as Gk1j,output Gi:j as Gij
module grey(Gij,Gik,Pik,Gk1j);
	output Gij;
	input Gik,Pik,Gk1j;

	wire i0;

	and(i0,Pik,Gk1j);
	or(Gij,Gik,i0);

endmodule

//Black cell, takes Gi:k as Gik, Pi:k as Pik, Gk-1:j as Gk1j,Pk-1:j as Pk1j, output Gi:j as Gij, Pi:j as Pij
module black(Gij,Pij,Gik,Pik,Gk1j,Pk1j);
	output Gij,Pij;
	input Gik,Pik,Gk1j,Pk1j;

	wire i0;

	and(i0,Pik,Gk1j);
	or(Gij,Gik,i0);

	and(Pij,Pik,Pk1j);

endmodule
```

XI. Appendix B

Verilog Code of bonus work

1) 4 bit x 4bit multiplier:

```
module mul4(s,co,a,b);
        output [7:0] s;
        output co;
        input [3:0] a,b;

        wire [3:0] c0,c1,c2,c3,c4;
        wire [2:0] s0,s1,s2,s3;

        CSA m00(s[0],c0[0],a[0],b[0],1'b0,1'b0);
        CSA m01(s0[0],c0[1],a[1],b[0],1'b0,1'b0);
        CSA m02(s0[1],c0[2],a[2],b[0],1'b0,1'b0);
        CSA m03(s0[2],c0[3],a[3],b[0],1'b0,1'b0);

        CSA m11(s[1],c1[0],a[0],b[1],s0[0],c0[0]);
        CSA m12(s1[0],c1[1],a[1],b[1],s0[1],c0[1]);
        CSA m13(s1[1],c1[2],a[2],b[1],s0[2],c0[2]);
        CSA m14(s1[2],c1[3],a[3],b[1],1'b0,c0[3]);

        CSA m22(s[2],c2[0],a[0],b[2],s1[0],c1[0]);
        CSA m23(s2[0],c2[1],a[1],b[2],s1[1],c1[1]);
        CSA m24(s2[1],c2[2],a[2],b[2],s1[2],c1[2]);
        CSA m25(s2[2],c2[3],a[3],b[2],1'b0,c1[3]);

        CSA m33(s[3],c3[0],a[0],b[3],s2[0],c2[0]);
        CSA m34(s3[0],c3[1],a[1],b[3],s2[1],c2[1]);
        CSA m35(s3[1],c3[2],a[2],b[3],s2[2],c2[2]);
        CSA m36(s3[2],c3[3],a[3],b[3],1'b0,c2[3]);

        CPA m44(s[4],c4[0],c3[0],s3[0],1'b0);
        CPA m45(s[5],c4[1],c3[1],s3[1],c4[0]);
        CPA m46(s[6],c4[2],c3[2],s3[2],c4[1]);
        CPA m47(s[7],co,c3[3],1'b0,c4[2]);

endmodule

module CSA(so,co,A,B,si,ci);
```

```
        output so,co;
        input A,B,si,ci;

        wire o0;

        and(o0,A,B);
        CPA a0(so,co,o0,si,ci);
endmodule

module CPA(s, co, a, b, ci);
        output s,co;
        input a, b, ci;

        wire o0, o1, o2;
        xor(s, a, b, ci);

        or(o0, a, b);
        or(o1, b, ci);
        or(o2, ci, a);
        and(co, o0, o1, o2);
endmodule
```

2) 16 bit x 16bit multiplier:

```
module mul16(s,co,a,b);
        output [31:0] s;
        output co;
        input [15:0] a,b;

        wire [15:0] c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;
        wire [14:0] s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15;

        CSA m00(s[0],c0[0],a[0],b[0],1'b0,1'b0);
        CSA m01(s0[0],c0[1],a[1],b[0],1'b0,1'b0);
        CSA m02(s0[1],c0[2],a[2],b[0],1'b0,1'b0);
        CSA m03(s0[2],c0[3],a[3],b[0],1'b0,1'b0);
        CSA m04(s0[3],c0[4],a[4],b[0],1'b0,1'b0);
        CSA m05(s0[4],c0[5],a[5],b[0],1'b0,1'b0);
        CSA m06(s0[5],c0[6],a[6],b[0],1'b0,1'b0);
        CSA m07(s0[6],c0[7],a[7],b[0],1'b0,1'b0);
        CSA m08(s0[7],c0[8],a[8],b[0],1'b0,1'b0);
```

```
CSA m09(s0[8],c0[9],a[9],b[0],1'b0,1'b0);
CSA m010(s0[9],c0[10],a[10],b[0],1'b0,1'b0);
CSA m011(s0[10],c0[11],a[11],b[0],1'b0,1'b0);
CSA m012(s0[11],c0[12],a[12],b[0],1'b0,1'b0);
CSA m013(s0[12],c0[13],a[13],b[0],1'b0,1'b0);
CSA m014(s0[13],c0[14],a[14],b[0],1'b0,1'b0);
CSA m015(s0[14],c0[15],a[15],b[0],1'b0,1'b0);

CSA m1_1(s[1],c1[0],a[0],b[1],s0[0],c0[0]);
CSA m1_2(s1[0],c1[1],a[1],b[1],s0[1],c0[1]);
CSA m1_3(s1[1],c1[2],a[2],b[1],s0[2],c0[2]);
CSA m1_4(s1[2],c1[3],a[3],b[1],s0[3],c0[3]);
CSA m1_5(s1[3],c1[4],a[4],b[1],s0[4],c0[4]);
CSA m1_6(s1[4],c1[5],a[5],b[1],s0[5],c0[5]);
CSA m1_7(s1[5],c1[6],a[6],b[1],s0[6],c0[6]);
CSA m1_8(s1[6],c1[7],a[7],b[1],s0[7],c0[7]);
CSA m1_9(s1[7],c1[8],a[8],b[1],s0[8],c0[8]);
CSA m1_10(s1[8],c1[9],a[9],b[1],s0[9],c0[9]);
CSA m1_11(s1[9],c1[10],a[10],b[1],s0[10],c0[10]);
CSA m1_12(s1[10],c1[11],a[11],b[1],s0[11],c0[11]);
CSA m1_13(s1[11],c1[12],a[12],b[1],s0[12],c0[12]);
CSA m1_14(s1[12],c1[13],a[13],b[1],s0[13],c0[13]);
CSA m1_15(s1[13],c1[14],a[14],b[1],s0[14],c0[14]);
CSA m1_16(s1[14],c1[15],a[15],b[1],1'b0,c0[15]);

CSA m2_1(s[2],c2[0],a[0],b[2],s1[0],c1[0]);
CSA m2_2(s2[0],c2[1],a[1],b[2],s1[1],c1[1]);
CSA m2_3(s2[1],c2[2],a[2],b[2],s1[2],c1[2]);
CSA m2_4(s2[2],c2[3],a[3],b[2],s1[3],c1[3]);
CSA m2_5(s2[3],c2[4],a[4],b[2],s1[4],c1[4]);
CSA m2_6(s2[4],c2[5],a[5],b[2],s1[5],c1[5]);
CSA m2_7(s2[5],c2[6],a[6],b[2],s1[6],c1[6]);
CSA m2_8(s2[6],c2[7],a[7],b[2],s1[7],c1[7]);
CSA m2_9(s2[7],c2[8],a[8],b[2],s1[8],c1[8]);
CSA m2_10(s2[8],c2[9],a[9],b[2],s1[9],c1[9]);
CSA m2_11(s2[9],c2[10],a[10],b[2],s1[10],c1[10]);
CSA m2_12(s2[10],c2[11],a[11],b[2],s1[11],c1[11]);
CSA m2_13(s2[11],c2[12],a[12],b[2],s1[12],c1[12]);
CSA m2_14(s2[12],c2[13],a[13],b[2],s1[13],c1[13]);
CSA m2_15(s2[13],c2[14],a[14],b[2],s1[14],c1[14]);
```

```
CSA m2_16(s2[14],c2[15],a[15],b[2],1'b0,c1[15]);

CSA m3_1(s[3],c3[0],a[0],b[3],s2[0],c2[0]);
CSA m3_2(s3[0],c3[1],a[1],b[3],s2[1],c2[1]);
CSA m3_3(s3[1],c3[2],a[2],b[3],s2[2],c2[2]);
CSA m3_4(s3[2],c3[3],a[3],b[3],s2[3],c2[3]);
CSA m3_5(s3[3],c3[4],a[4],b[3],s2[4],c2[4]);
CSA m3_6(s3[4],c3[5],a[5],b[3],s2[5],c2[5]);
CSA m3_7(s3[5],c3[6],a[6],b[3],s2[6],c2[6]);
CSA m3_8(s3[6],c3[7],a[7],b[3],s2[7],c2[7]);
CSA m3_9(s3[7],c3[8],a[8],b[3],s2[8],c2[8]);
CSA m3_10(s3[8],c3[9],a[9],b[3],s2[9],c2[9]);
CSA m3_11(s3[9],c3[10],a[10],b[3],s2[10],c2[10]);
CSA m3_12(s3[10],c3[11],a[11],b[3],s2[11],c2[11]);
CSA m3_13(s3[11],c3[12],a[12],b[3],s2[12],c2[12]);
CSA m3_14(s3[12],c3[13],a[13],b[3],s2[13],c2[13]);
CSA m3_15(s3[13],c3[14],a[14],b[3],s2[14],c2[14]);
CSA m3_16(s3[14],c3[15],a[15],b[3],1'b0,c2[15]);

CSA m4_1(s[4],c4[0],a[0],b[4],s3[0],c3[0]);
CSA m4_2(s4[0],c4[1],a[1],b[4],s3[1],c3[1]);
CSA m4_3(s4[1],c4[2],a[2],b[4],s3[2],c3[2]);
CSA m4_4(s4[2],c4[3],a[3],b[4],s3[3],c3[3]);
CSA m4_5(s4[3],c4[4],a[4],b[4],s3[4],c3[4]);
CSA m4_6(s4[4],c4[5],a[5],b[4],s3[5],c3[5]);
CSA m4_7(s4[5],c4[6],a[6],b[4],s3[6],c3[6]);
CSA m4_8(s4[6],c4[7],a[7],b[4],s3[7],c3[7]);
CSA m4_9(s4[7],c4[8],a[8],b[4],s3[8],c3[8]);
CSA m4_10(s4[8],c4[9],a[9],b[4],s3[9],c3[9]);
CSA m4_11(s4[9],c4[10],a[10],b[4],s3[10],c3[10]);
CSA m4_12(s4[10],c4[11],a[11],b[4],s3[11],c3[11]);
CSA m4_13(s4[11],c4[12],a[12],b[4],s3[12],c3[12]);
CSA m4_14(s4[12],c4[13],a[13],b[4],s3[13],c3[13]);
CSA m4_15(s4[13],c4[14],a[14],b[4],s3[14],c3[14]);
CSA m4_16(s4[14],c4[15],a[15],b[4],1'b0,c3[15]);

CSA m5_1(s[5],c5[0],a[0],b[5],s4[0],c4[0]);
CSA m5_2(s5[0],c5[1],a[1],b[5],s4[1],c4[1]);
CSA m5_3(s5[1],c5[2],a[2],b[5],s4[2],c4[2]);
CSA m5_4(s5[2],c5[3],a[3],b[5],s4[3],c4[3]);
```

```
CSA m5_5(s5[3],c5[4],a[4],b[5],s4[4],c4[4]);
CSA m5_6(s5[4],c5[5],a[5],b[5],s4[5],c4[5]);
CSA m5_7(s5[5],c5[6],a[6],b[5],s4[6],c4[6]);
CSA m5_8(s5[6],c5[7],a[7],b[5],s4[7],c4[7]);
CSA m5_9(s5[7],c5[8],a[8],b[5],s4[8],c4[8]);
CSA m5_10(s5[8],c5[9],a[9],b[5],s4[9],c4[9]);
CSA m5_11(s5[9],c5[10],a[10],b[5],s4[10],c4[10]);
CSA m5_12(s5[10],c5[11],a[11],b[5],s4[11],c4[11]);
CSA m5_13(s5[11],c5[12],a[12],b[5],s4[12],c4[12]);
CSA m5_14(s5[12],c5[13],a[13],b[5],s4[13],c4[13]);
CSA m5_15(s5[13],c5[14],a[14],b[5],s4[14],c4[14]);
CSA m5_16(s5[14],c5[15],a[15],b[5],1'b0,c4[15]);

CSA m6_1(s[6],c6[0],a[0],b[6],s5[0],c5[0]);
CSA m6_2(s6[0],c6[1],a[1],b[6],s5[1],c5[1]);
CSA m6_3(s6[1],c6[2],a[2],b[6],s5[2],c5[2]);
CSA m6_4(s6[2],c6[3],a[3],b[6],s5[3],c5[3]);
CSA m6_5(s6[3],c6[4],a[4],b[6],s5[4],c5[4]);
CSA m6_6(s6[4],c6[5],a[5],b[6],s5[5],c5[5]);
CSA m6_7(s6[5],c6[6],a[6],b[6],s5[6],c5[6]);
CSA m6_8(s6[6],c6[7],a[7],b[6],s5[7],c5[7]);
CSA m6_9(s6[7],c6[8],a[8],b[6],s5[8],c5[8]);
CSA m6_10(s6[8],c6[9],a[9],b[6],s5[9],c5[9]);
CSA m6_11(s6[9],c6[10],a[10],b[6],s5[10],c5[10]);
CSA m6_12(s6[10],c6[11],a[11],b[6],s5[11],c5[11]);
CSA m6_13(s6[11],c6[12],a[12],b[6],s5[12],c5[12]);
CSA m6_14(s6[12],c6[13],a[13],b[6],s5[13],c5[13]);
CSA m6_15(s6[13],c6[14],a[14],b[6],s5[14],c5[14]);
CSA m6_16(s6[14],c6[15],a[15],b[6],1'b0,c5[15]);

CSA m7_1(s[7],c7[0],a[0],b[7],s6[0],c6[0]);
CSA m7_2(s7[0],c7[1],a[1],b[7],s6[1],c6[1]);
CSA m7_3(s7[1],c7[2],a[2],b[7],s6[2],c6[2]);
CSA m7_4(s7[2],c7[3],a[3],b[7],s6[3],c6[3]);
CSA m7_5(s7[3],c7[4],a[4],b[7],s6[4],c6[4]);
CSA m7_6(s7[4],c7[5],a[5],b[7],s6[5],c6[5]);
CSA m7_7(s7[5],c7[6],a[6],b[7],s6[6],c6[6]);
CSA m7_8(s7[6],c7[7],a[7],b[7],s6[7],c6[7]);
CSA m7_9(s7[7],c7[8],a[8],b[7],s6[8],c6[8]);
CSA m7_10(s7[8],c7[9],a[9],b[7],s6[9],c6[9]);
```

```
CSA m7_11(s7[9],c7[10],a[10],b[7],s6[10],c6[10]);
CSA m7_12(s7[10],c7[11],a[11],b[7],s6[11],c6[11]);
CSA m7_13(s7[11],c7[12],a[12],b[7],s6[12],c6[12]);
CSA m7_14(s7[12],c7[13],a[13],b[7],s6[13],c6[13]);
CSA m7_15(s7[13],c7[14],a[14],b[7],s6[14],c6[14]);
CSA m7_16(s7[14],c7[15],a[15],b[7],1'b0,c6[15]);

CSA m8_1(s[8],c8[0],a[0],b[8],s7[0],c7[0]);
CSA m8_2(s8[0],c8[1],a[1],b[8],s7[1],c7[1]);
CSA m8_3(s8[1],c8[2],a[2],b[8],s7[2],c7[2]);
CSA m8_4(s8[2],c8[3],a[3],b[8],s7[3],c7[3]);
CSA m8_5(s8[3],c8[4],a[4],b[8],s7[4],c7[4]);
CSA m8_6(s8[4],c8[5],a[5],b[8],s7[5],c7[5]);
CSA m8_7(s8[5],c8[6],a[6],b[8],s7[6],c7[6]);
CSA m8_8(s8[6],c8[7],a[7],b[8],s7[7],c7[7]);
CSA m8_9(s8[7],c8[8],a[8],b[8],s7[8],c7[8]);
CSA m8_10(s8[8],c8[9],a[9],b[8],s7[9],c7[9]);
CSA m8_11(s8[9],c8[10],a[10],b[8],s7[10],c7[10]);
CSA m8_12(s8[10],c8[11],a[11],b[8],s7[11],c7[11]);
CSA m8_13(s8[11],c8[12],a[12],b[8],s7[12],c7[12]);
CSA m8_14(s8[12],c8[13],a[13],b[8],s7[13],c7[13]);
CSA m8_15(s8[13],c8[14],a[14],b[8],s7[14],c7[14]);
CSA m8_16(s8[14],c8[15],a[15],b[8],1'b0,c7[15]);

CSA m9_1(s[9],c9[0],a[0],b[9],s8[0],c8[0]);
CSA m9_2(s9[0],c9[1],a[1],b[9],s8[1],c8[1]);
CSA m9_3(s9[1],c9[2],a[2],b[9],s8[2],c8[2]);
CSA m9_4(s9[2],c9[3],a[3],b[9],s8[3],c8[3]);
CSA m9_5(s9[3],c9[4],a[4],b[9],s8[4],c8[4]);
CSA m9_6(s9[4],c9[5],a[5],b[9],s8[5],c8[5]);
CSA m9_7(s9[5],c9[6],a[6],b[9],s8[6],c8[6]);
CSA m9_8(s9[6],c9[7],a[7],b[9],s8[7],c8[7]);
CSA m9_9(s9[7],c9[8],a[8],b[9],s8[8],c8[8]);
CSA m9_10(s9[8],c9[9],a[9],b[9],s8[9],c8[9]);
CSA m9_11(s9[9],c9[10],a[10],b[9],s8[10],c8[10]);
CSA m9_12(s9[10],c9[11],a[11],b[9],s8[11],c8[11]);
CSA m9_13(s9[11],c9[12],a[12],b[9],s8[12],c8[12]);
CSA m9_14(s9[12],c9[13],a[13],b[9],s8[13],c8[13]);
CSA m9_15(s9[13],c9[14],a[14],b[9],s8[14],c8[14]);
CSA m9_16(s9[14],c9[15],a[15],b[9],1'b0,c8[15]);
```

```
CSA m10_1(s[10],c10[0],a[0],b[10],s9[0],c9[0]);
CSA m10_2(s10[0],c10[1],a[1],b[10],s9[1],c9[1]);
CSA m10_3(s10[1],c10[2],a[2],b[10],s9[2],c9[2]);
CSA m10_4(s10[2],c10[3],a[3],b[10],s9[3],c9[3]);
CSA m10_5(s10[3],c10[4],a[4],b[10],s9[4],c9[4]);
CSA m10_6(s10[4],c10[5],a[5],b[10],s9[5],c9[5]);
CSA m10_7(s10[5],c10[6],a[6],b[10],s9[6],c9[6]);
CSA m10_8(s10[6],c10[7],a[7],b[10],s9[7],c9[7]);
CSA m10_9(s10[7],c10[8],a[8],b[10],s9[8],c9[8]);
CSA m10_10(s10[8],c10[9],a[9],b[10],s9[9],c9[9]);
CSA m10_11(s10[9],c10[10],a[10],b[10],s9[10],c9[10]);
CSA m10_12(s10[10],c10[11],a[11],b[10],s9[11],c9[11]);
CSA m10_13(s10[11],c10[12],a[12],b[10],s9[12],c9[12]);
CSA m10_14(s10[12],c10[13],a[13],b[10],s9[13],c9[13]);
CSA m10_15(s10[13],c10[14],a[14],b[10],s9[14],c9[14]);
CSA m10_16(s10[14],c10[15],a[15],b[10],1'b0,c9[15]);

CSA m11_1(s[11],c11[0],a[0],b[11],s10[0],c10[0]);
CSA m11_2(s11[0],c11[1],a[1],b[11],s10[1],c10[1]);
CSA m11_3(s11[1],c11[2],a[2],b[11],s10[2],c10[2]);
CSA m11_4(s11[2],c11[3],a[3],b[11],s10[3],c10[3]);
CSA m11_5(s11[3],c11[4],a[4],b[11],s10[4],c10[4]);
CSA m11_6(s11[4],c11[5],a[5],b[11],s10[5],c10[5]);
CSA m11_7(s11[5],c11[6],a[6],b[11],s10[6],c10[6]);
CSA m11_8(s11[6],c11[7],a[7],b[11],s10[7],c10[7]);
CSA m11_9(s11[7],c11[8],a[8],b[11],s10[8],c10[8]);
CSA m11_10(s11[8],c11[9],a[9],b[11],s10[9],c10[9]);
CSA m11_11(s11[9],c11[10],a[10],b[11],s10[10],c10[10]);
CSA m11_12(s11[10],c11[11],a[11],b[11],s10[11],c10[11]);
CSA m11_13(s11[11],c11[12],a[12],b[11],s10[12],c10[12]);
CSA m11_14(s11[12],c11[13],a[13],b[11],s10[13],c10[13]);
CSA m11_15(s11[13],c11[14],a[14],b[11],s10[14],c10[14]);
CSA m11_16(s11[14],c11[15],a[15],b[11],1'b0,c10[15]);

CSA m12_1(s[12],c12[0],a[0],b[12],s11[0],c11[0]);
CSA m12_2(s12[0],c12[1],a[1],b[12],s11[1],c11[1]);
CSA m12_3(s12[1],c12[2],a[2],b[12],s11[2],c11[2]);
CSA m12_4(s12[2],c12[3],a[3],b[12],s11[3],c11[3]);
CSA m12_5(s12[3],c12[4],a[4],b[12],s11[4],c11[4]);
```

```
CSA m12_6(s12[4],c12[5],a[5],b[12],s11[5],c11[5]);
CSA m12_7(s12[5],c12[6],a[6],b[12],s11[6],c11[6]);
CSA m12_8(s12[6],c12[7],a[7],b[12],s11[7],c11[7]);
CSA m12_9(s12[7],c12[8],a[8],b[12],s11[8],c11[8]);
CSA m12_10(s12[8],c12[9],a[9],b[12],s11[9],c11[9]);
CSA m12_11(s12[9],c12[10],a[10],b[12],s11[10],c11[10]);
CSA m12_12(s12[10],c12[11],a[11],b[12],s11[11],c11[11]);
CSA m12_13(s12[11],c12[12],a[12],b[12],s11[12],c11[12]);
CSA m12_14(s12[12],c12[13],a[13],b[12],s11[13],c11[13]);
CSA m12_15(s12[13],c12[14],a[14],b[12],s11[14],c11[14]);
CSA m12_16(s12[14],c12[15],a[15],b[12],1'b0,c11[15]);

CSA m13_1(s[13],c13[0],a[0],b[13],s12[0],c12[0]);
CSA m13_2(s13[0],c13[1],a[1],b[13],s12[1],c12[1]);
CSA m13_3(s13[1],c13[2],a[2],b[13],s12[2],c12[2]);
CSA m13_4(s13[2],c13[3],a[3],b[13],s12[3],c12[3]);
CSA m13_5(s13[3],c13[4],a[4],b[13],s12[4],c12[4]);
CSA m13_6(s13[4],c13[5],a[5],b[13],s12[5],c12[5]);
CSA m13_7(s13[5],c13[6],a[6],b[13],s12[6],c12[6]);
CSA m13_8(s13[6],c13[7],a[7],b[13],s12[7],c12[7]);
CSA m13_9(s13[7],c13[8],a[8],b[13],s12[8],c12[8]);
CSA m13_10(s13[8],c13[9],a[9],b[13],s12[9],c12[9]);
CSA m13_11(s13[9],c13[10],a[10],b[13],s12[10],c12[10]);
CSA m13_12(s13[10],c13[11],a[11],b[13],s12[11],c12[11]);
CSA m13_13(s13[11],c13[12],a[12],b[13],s12[12],c12[12]);
CSA m13_14(s13[12],c13[13],a[13],b[13],s12[13],c12[13]);
CSA m13_15(s13[13],c13[14],a[14],b[13],s12[14],c12[14]);
CSA m13_16(s13[14],c13[15],a[15],b[13],1'b0,c12[15]);

CSA m14_1(s[14],c14[0],a[0],b[14],s13[0],c13[0]);
CSA m14_2(s14[0],c14[1],a[1],b[14],s13[1],c13[1]);
CSA m14_3(s14[1],c14[2],a[2],b[14],s13[2],c13[2]);
CSA m14_4(s14[2],c14[3],a[3],b[14],s13[3],c13[3]);
CSA m14_5(s14[3],c14[4],a[4],b[14],s13[4],c13[4]);
CSA m14_6(s14[4],c14[5],a[5],b[14],s13[5],c13[5]);
CSA m14_7(s14[5],c14[6],a[6],b[14],s13[6],c13[6]);
CSA m14_8(s14[6],c14[7],a[7],b[14],s13[7],c13[7]);
CSA m14_9(s14[7],c14[8],a[8],b[14],s13[8],c13[8]);
CSA m14_10(s14[8],c14[9],a[9],b[14],s13[9],c13[9]);
CSA m14_11(s14[9],c14[10],a[10],b[14],s13[10],c13[10]);
```

```
        CSA m14_12(s14[10],c14[11],a[11],b[14],s13[11],c13[11]);
        CSA m14_13(s14[11],c14[12],a[12],b[14],s13[12],c13[12]);
        CSA m14_14(s14[12],c14[13],a[13],b[14],s13[13],c13[13]);
        CSA m14_15(s14[13],c14[14],a[14],b[14],s13[14],c13[14]);
        CSA m14_16(s14[14],c14[15],a[15],b[14],1'b0,c13[15]);

        CSA m15_1(s[15],c15[0],a[0],b[15],s14[0],c14[0]);
        CSA m15_2(s15[0],c15[1],a[1],b[15],s14[1],c14[1]);
        CSA m15_3(s15[1],c15[2],a[2],b[15],s14[2],c14[2]);
        CSA m15_4(s15[2],c15[3],a[3],b[15],s14[3],c14[3]);
        CSA m15_5(s15[3],c15[4],a[4],b[15],s14[4],c14[4]);
        CSA m15_6(s15[4],c15[5],a[5],b[15],s14[5],c14[5]);
        CSA m15_7(s15[5],c15[6],a[6],b[15],s14[6],c14[6]);
        CSA m15_8(s15[6],c15[7],a[7],b[15],s14[7],c14[7]);
        CSA m15_9(s15[7],c15[8],a[8],b[15],s14[8],c14[8]);
        CSA m15_10(s15[8],c15[9],a[9],b[15],s14[9],c14[9]);
        CSA m15_11(s15[9],c15[10],a[10],b[15],s14[10],c14[10]);
        CSA m15_12(s15[10],c15[11],a[11],b[15],s14[11],c14[11]);
        CSA m15_13(s15[11],c15[12],a[12],b[15],s14[12],c14[12]);
        CSA m15_14(s15[12],c15[13],a[13],b[15],s14[13],c14[13]);
        CSA m15_15(s15[13],c15[14],a[14],b[15],s14[14],c14[14]);
        CSA m15_16(s15[14],c15[15],a[15],b[15],1'b0,c14[15]);

        CPA m16_1(s[16],c16[0],c15[0],s15[0],1'b0);
        CPA m16_2(s[17],c16[1],c15[1],s15[1],c16[0]);
        CPA m16_3(s[18],c16[2],c15[2],s15[2],c16[1]);
        CPA m16_4(s[19],c16[3],c15[3],s15[3],c16[2]);
        CPA m16_5(s[20],c16[4],c15[4],s15[4],c16[3]);
        CPA m16_6(s[21],c16[5],c15[5],s15[5],c16[4]);
        CPA m16_7(s[22],c16[6],c15[6],s15[6],c16[5]);
        CPA m16_8(s[23],c16[7],c15[7],s15[7],c16[6]);
        CPA m16_9(s[24],c16[8],c15[8],s15[8],c16[7]);
        CPA m16_10(s[25],c16[9],c15[9],s15[9],c16[8]);
        CPA m16_11(s[26],c16[10],c15[10],s15[10],c16[9]);
        CPA m16_12(s[27],c16[11],c15[11],s15[11],c16[10]);
        CPA m16_13(s[28],c16[12],c15[12],s15[12],c16[11]);
        CPA m16_14(s[29],c16[13],c15[13],s15[13],c16[12]);
        CPA m16_15(s[30],c16[14],c15[14],s15[14],c16[13]);
        CPA m16_16(s[31],co,c5[15],1'b0,c6[14]);
endmodule
```

```
module CSA(so,co,A,B,si,ci);
	output so,co;
	input A,B,si,ci;

	wire o0;

	and(o0,A,B);
	CPA a0(so,co,o0,si,ci);
endmodule

module CPA(s, co, a, b, ci);
	output s,co;
	input a, b, ci;

	wire o0, o1, o2;
	xor(s, a, b, ci);

	or(o0, a, b);
	or(o1, b, ci);
	or(o2, ci, a);
	and(co, o0, o1, o2);
endmodule
```

XII. Appendix C

a) Screenshots of RTL Simulation

8-bit Carry Ripple Adder

```
Compiling source file "adder8test.v"
Compiling source file "adder8.v"
Highest level modules:
stimulus

At Time:                  100 Sum =  16 Carry = 0
At Time:                  200 Sum =  85 Carry = 0
At Time:                  300 Sum = 236 Carry = 0
At Time:                  400 Sum =  18 Carry = 1
At Time:                  500 Sum = 134 Carry = 0
At Time:                  600 Sum = 247 Carry = 0
At Time:                  700 Sum =  94 Carry = 0
At Time:                  800 Sum =  44 Carry = 1
At Time:                  900 Sum = 255 Carry = 0
L17 "adder8test.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 364 accelera
ted events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL      08.20.001-p   May  4, 2015  15:31:42
mchen@uranus.ece.iit.edu:~%
```

8-bit Carry-Skip Adder

```
Compiling source file "adder8_cstest.v"
Compiling source file "adder8_cs.v"
Highest level modules:
stimulus

At Time:                  100 Sum =  16 Carry = 0
At Time:                  200 Sum =  85 Carry = 0
At Time:                  300 Sum = 236 Carry = 0
At Time:                  400 Sum =  18 Carry = 1
At Time:                  500 Sum = 134 Carry = 0
At Time:                  600 Sum = 247 Carry = 0
At Time:                  700 Sum =  94 Carry = 0
At Time:                  800 Sum =  44 Carry = 1
At Time:                  900 Sum = 255 Carry = 0
L17 "adder8_cstest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 530 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL      08.20.001-p   May  4, 2015  15:39:33
mchen@uranus.ece.iit.edu:~%
```

8-bit Carry Select Adder

```
Compiling source file "adder8_csltest.v"
Compiling source file "adder8_csl.v"
Highest level modules:
stimulus

At Time:                  100 Sum =  16 Carry = 0
At Time:                  200 Sum =  85 Carry = 0
At Time:                  300 Sum = 252 Carry = 0
At Time:                  400 Sum =  18 Carry = 1
At Time:                  500 Sum = 246 Carry = 0
At Time:                  600 Sum = 247 Carry = 0
At Time:                  700 Sum =  94 Carry = 0
At Time:                  800 Sum =  44 Carry = 1
At Time:                  900 Sum = 255 Carry = 0
L17 "adder8_csltest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 538 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL     08.20.001-p   May  4, 2015  15:54:01
```

8-bit Kogge-Stone Adder

```
Compiling source file "adder8_kstest.v"
Compiling source file "adder8_ks.v"
Highest level modules:
stimulus

At Time:                  100 Sum =  16 Carry = 0
At Time:                  200 Sum =  85 Carry = 0
At Time:                  300 Sum = 236 Carry = 0
At Time:                  400 Sum =  18 Carry = 1
At Time:                  500 Sum = 134 Carry = 0
At Time:                  600 Sum = 247 Carry = 0
At Time:                  700 Sum =  94 Carry = 0
At Time:                  800 Sum =  44 Carry = 1
At Time:                  900 Sum = 255 Carry = 0
L17 "adder8_kstest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 476 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL     08.20.001-p   May  4, 2015  15:55:56
```

32-bit Carry Ripple Adder

```
Compiling source file "adder32_test.v"
Compiling source file "adder32.v"
Highest level modules:
stimulus

At Time:                  100 Sum =         16 Carry = 0
At Time:                  200 Sum =         85 Carry = 0
At Time:                  300 Sum =        236 Carry = 0
At Time:                  400 Sum =        274 Carry = 0
At Time:                  500 Sum =        134 Carry = 0
At Time:                  600 Sum =        247 Carry = 0
At Time:                  700 Sum =         94 Carry = 0
At Time:                  800 Sum =        300 Carry = 0
At Time:                  900 Sum =        255 Carry = 0
L17 "adder32_test.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 550 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL     08.20.001-p   May  4, 2015  16:07:35
```

32-bit Carry-Skip Adder

```
Compiling source file "adder32_cstest.v"
Compiling source file "adder32_cs.v"
Highest level modules:
stimulus

At Time:                  100 Sum =          16 Carry = 0
At Time:                  200 Sum =          85 Carry = 0
At Time:                  300 Sum =         236 Carry = 0
At Time:                  400 Sum =         274 Carry = 0
At Time:                  500 Sum =         134 Carry = 0
At Time:                  600 Sum =         247 Carry = 0
At Time:                  700 Sum =          94 Carry = 0
At Time:                  800 Sum =         300 Carry = 0
At Time:                  900 Sum =         255 Carry = 0
L17 "adder32_cstest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 854 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL     08.20.001-p   May  4, 2015  16:16:34
```

32-bit Carry Select Adder

```
Compiling source file "adder32_csltest.v"
Compiling source file "adder32_csl.v"
Highest level modules:
stimulus

At Time:                  100 Sum =          16 Carry = 0
At Time:                  200 Sum =          85 Carry = 0
At Time:                  300 Sum =         252 Carry = 0
At Time:                  400 Sum =         274 Carry = 0
At Time:                  500 Sum =         246 Carry = 0
At Time:                  600 Sum =         247 Carry = 0
At Time:                  700 Sum =          94 Carry = 0
At Time:                  800 Sum =         300 Carry = 0
At Time:                  900 Sum =         255 Carry = 0
L17 "adder32_csltest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 906 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL     08.20.001-p   May  4, 2015  16:18:49
```

32-bit Kogge-Stone Adder

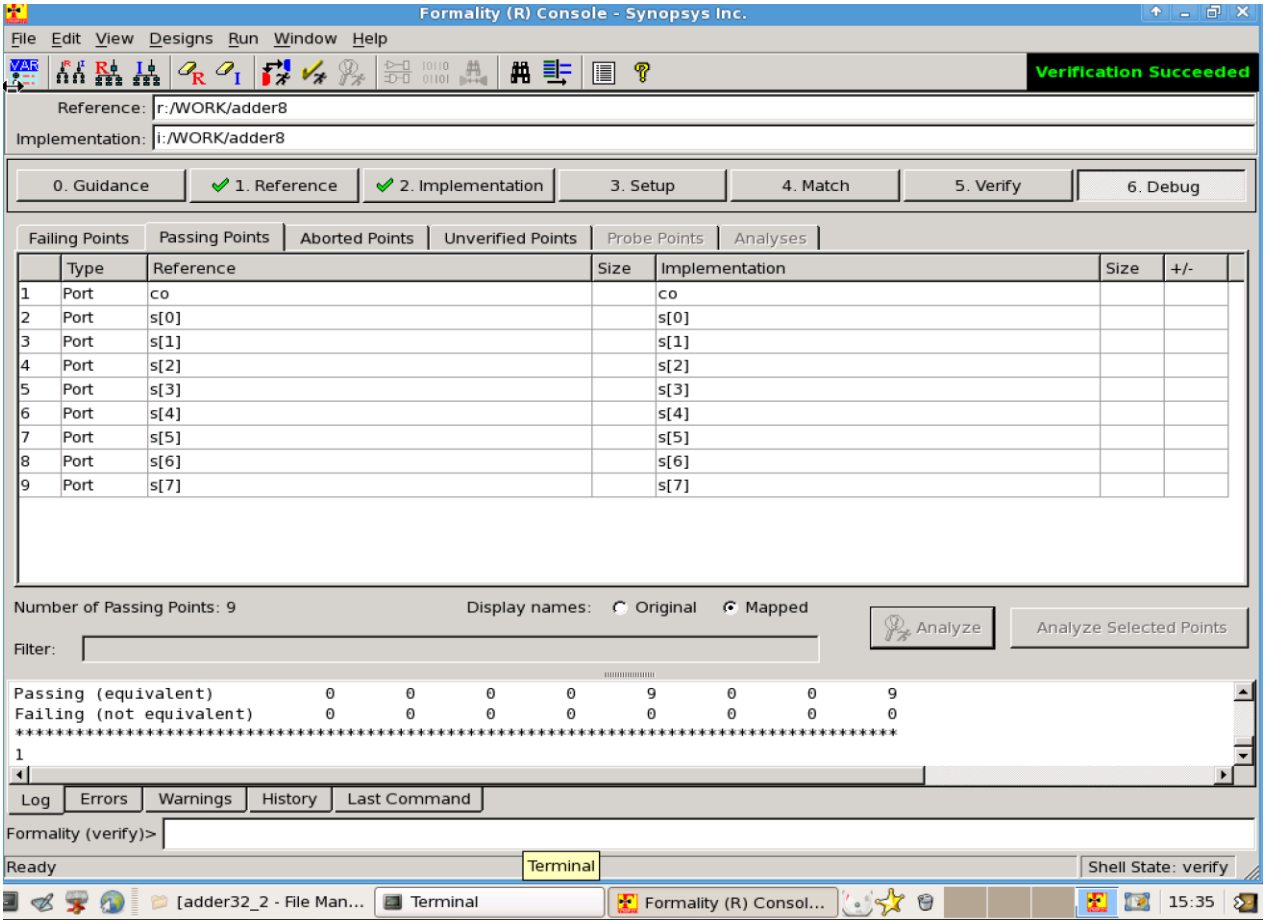
```
Compiling source file "adder32_kstest.v"
Compiling source file "adder32_ks.v"
Highest level modules:
stimulus

At Time:                  100 Sum =          16 Carry = 0
At Time:                  200 Sum =          85 Carry = 0
At Time:                  300 Sum =         236 Carry = 0
At Time:                  400 Sum =         274 Carry = 0
At Time:                  500 Sum =         134 Carry = 0
At Time:                  600 Sum =         247 Carry = 0
At Time:                  700 Sum =          94 Carry = 0
At Time:                  800 Sum =         300 Carry = 0
At Time:                  900 Sum =         255 Carry = 0
L17 "adder32_kstest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 918 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL     08.20.001-p   May  4, 2015  16:24:36
```

16×16 array multiplier

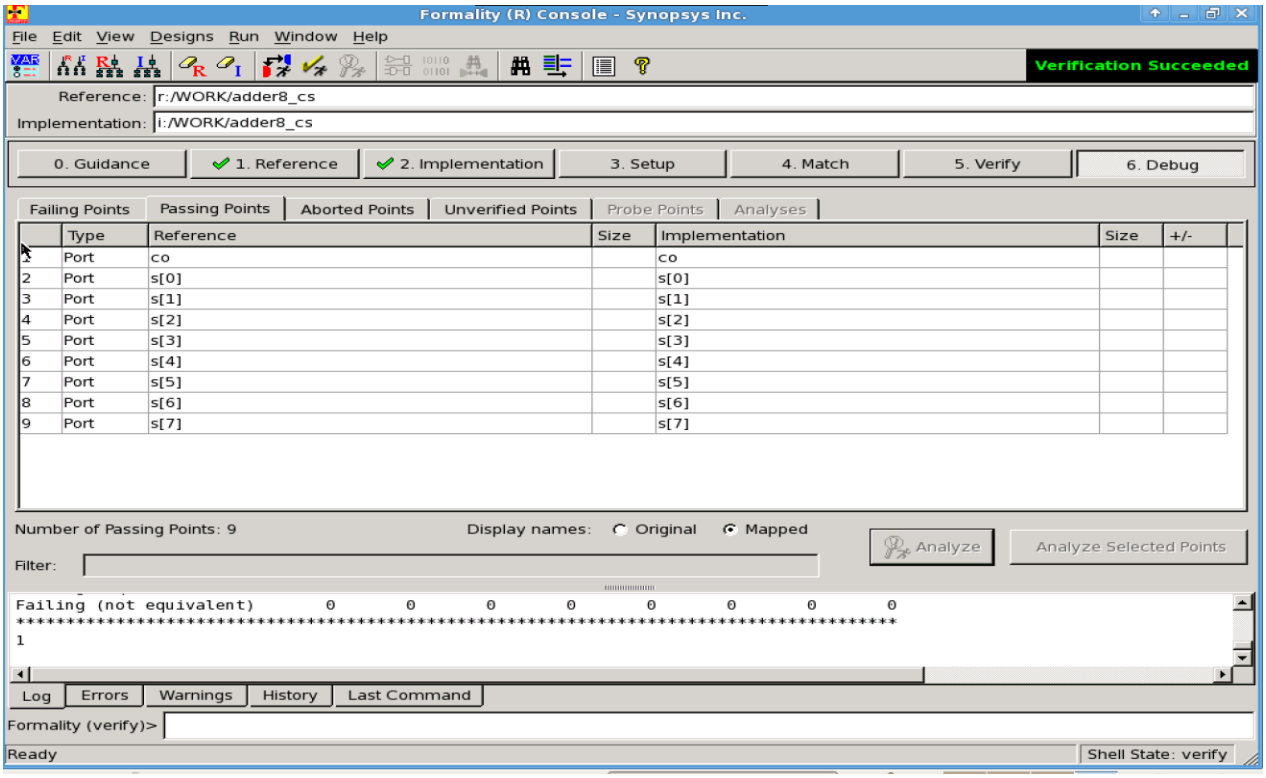
```
Compiling source file "mul16_test.v"
Compiling source file "mul16.v"

Warning!  Port sizes differ in port connection (port 2)     [Verilog-PCDPC]
          "mul16_test.v", 11: a

Warning!  Port sizes differ in port connection (port 3)     [Verilog-PCDPC]
          "mul16_test.v", 11: b
Highest level modules:
stimulus

At Time:                  100 Sum = 4294836225 Carry = 0
At Time:                  200 Sum =        130 Carry = 0
At Time:                  300 Sum =         50 Carry = 0
At Time:                  400 Sum =         27 Carry = 0
At Time:                  500 Sum =         84 Carry = 0
At Time:                  600 Sum =         70 Carry = 0
At Time:                  700 Sum =         44 Carry = 0
At Time:                  800 Sum =        150 Carry = 0
At Time:                  900 Sum =        144 Carry = 0
L17 "mul16_test.v": $finish at simulation time 100000
2 warnings
0 simulation events (use +profile or +listcounts option to count) + 5016 accelerated events
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.1 secs in simulation
End of Tool:    VERILOG-XL     08.20.001-p   May  4, 2015  16:28:14
```

b) Equivalence Checking using Formality

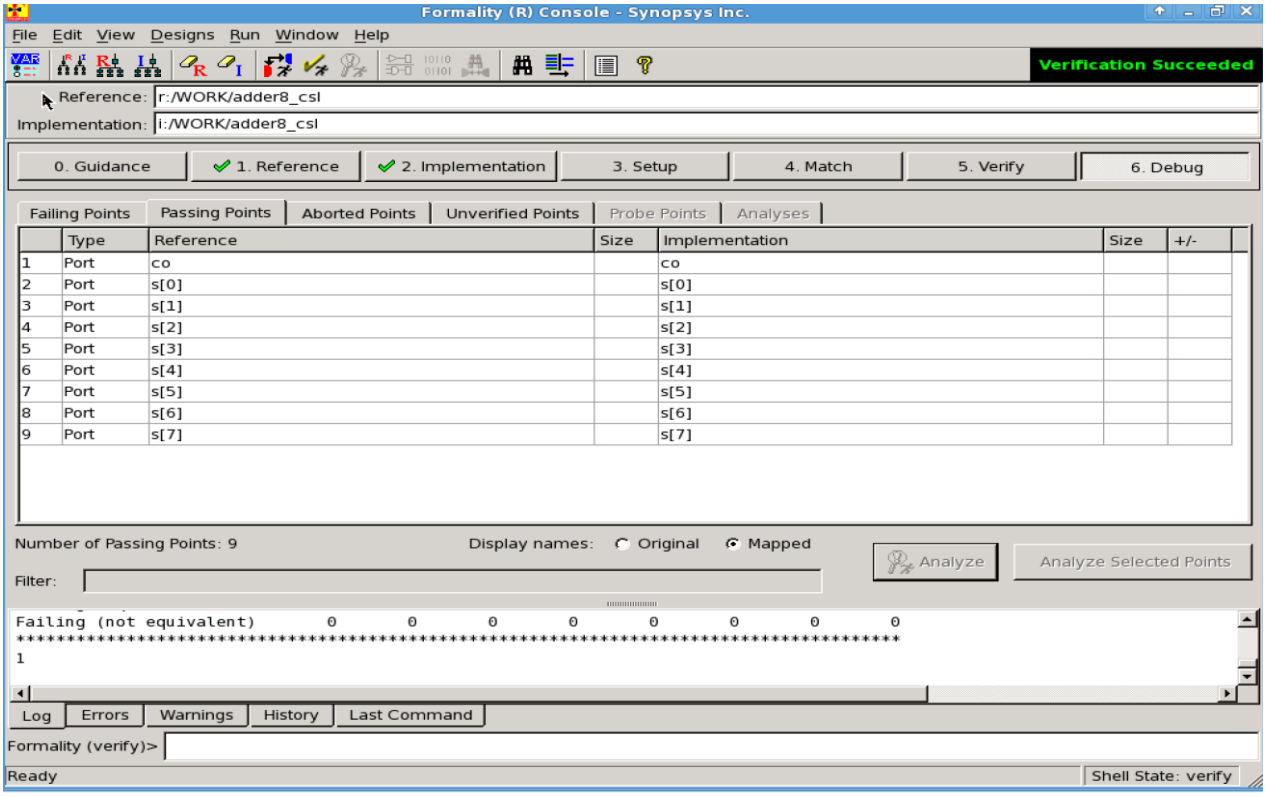
8-bit Carry Ripple Adder



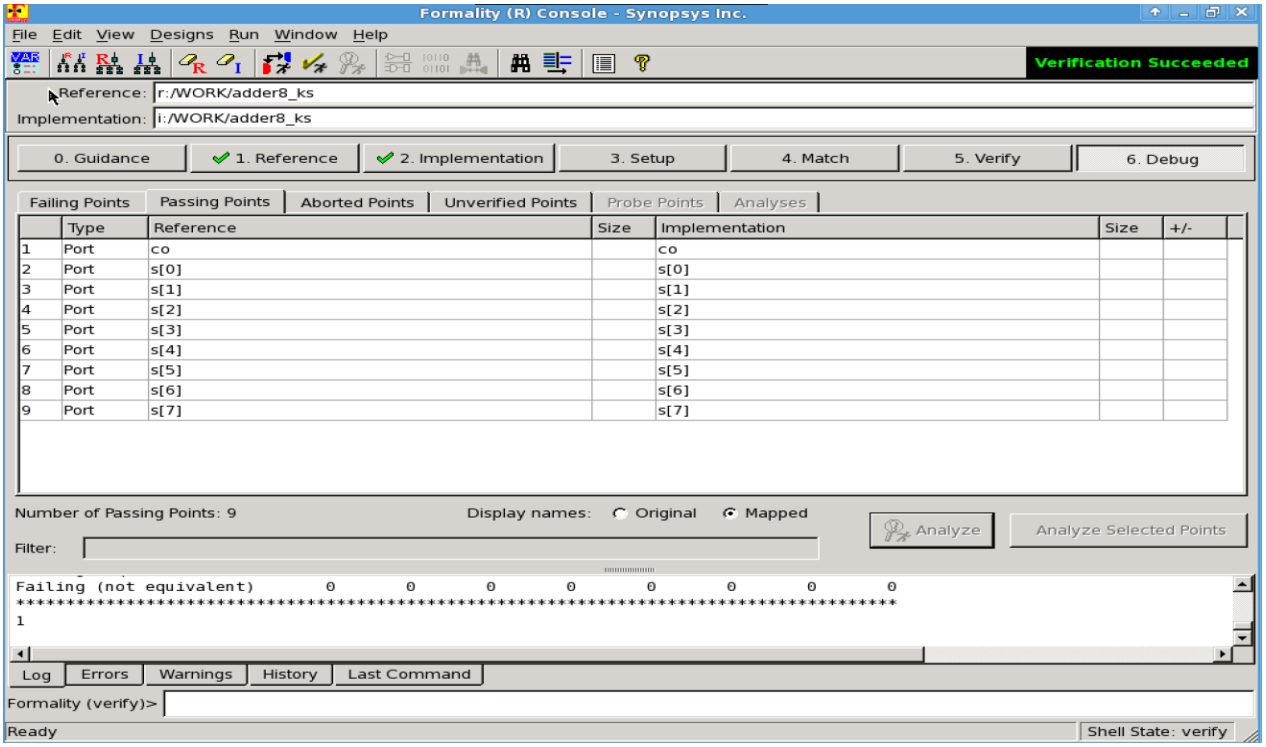
8-bit Carry-Skip Adder



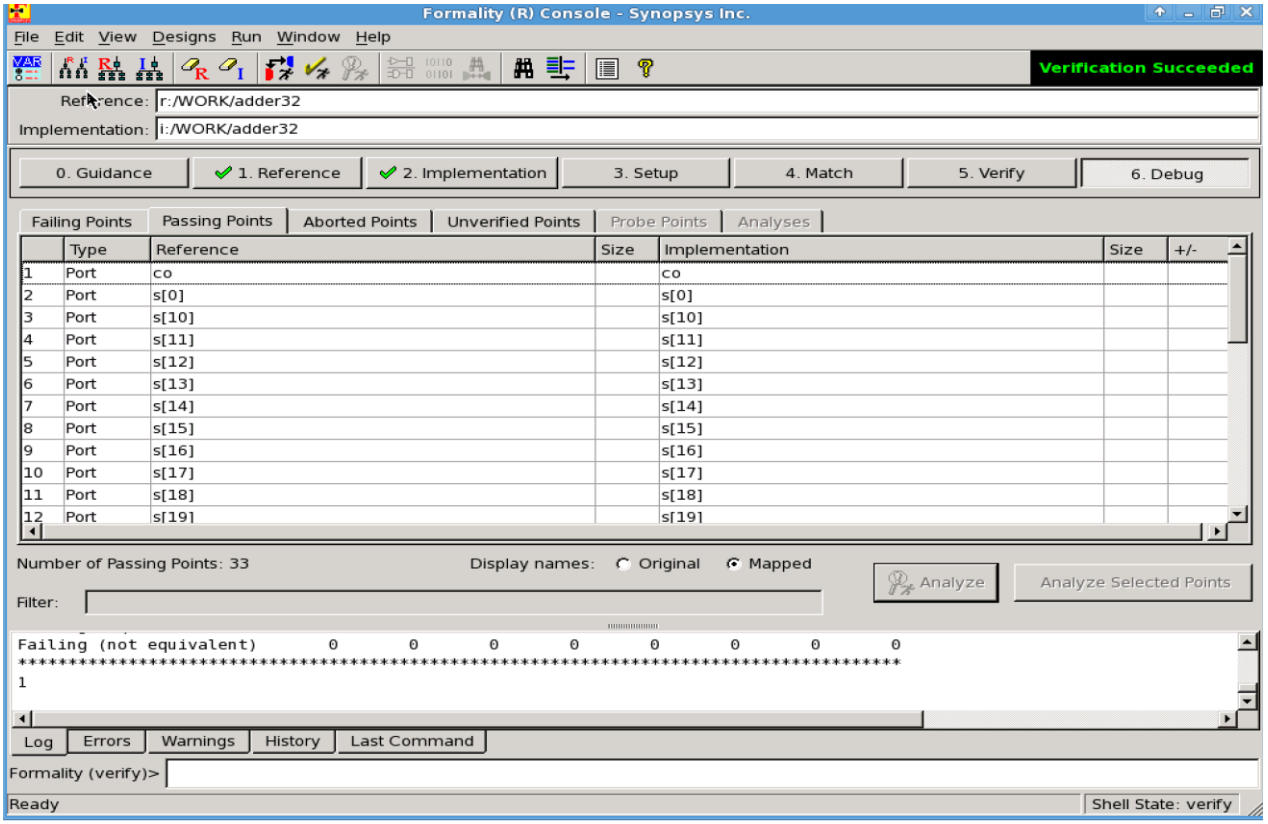
8-bit Carry Select Adder



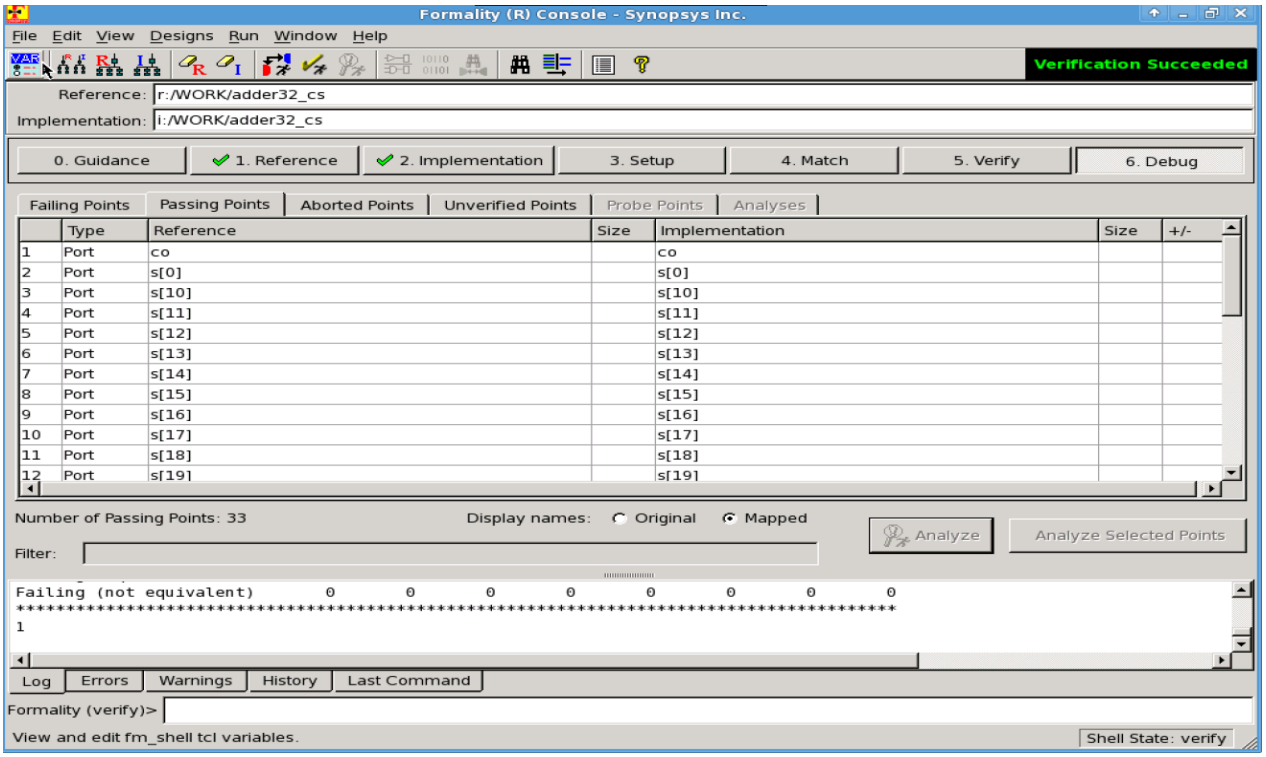
8-bit Kogge-Stone Adder



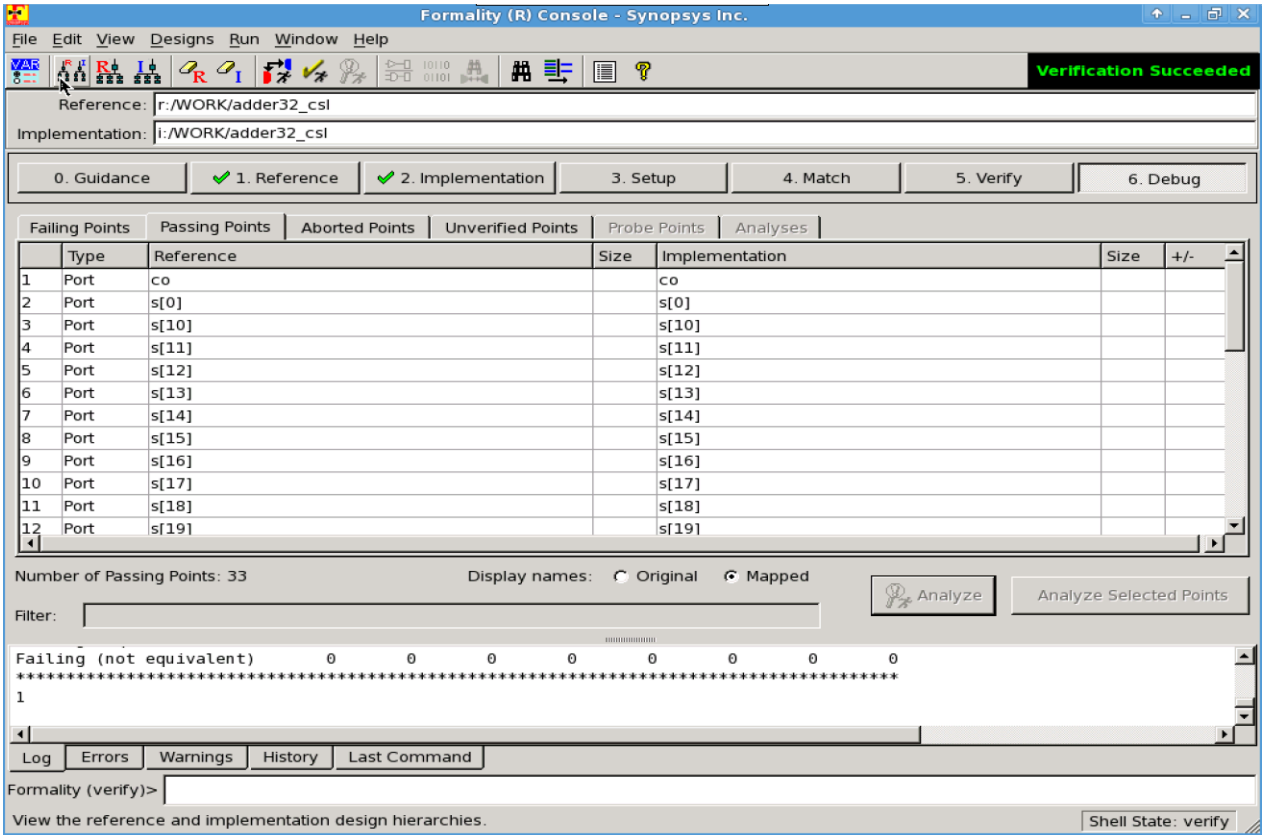
32-bit Carry Ripple Adder



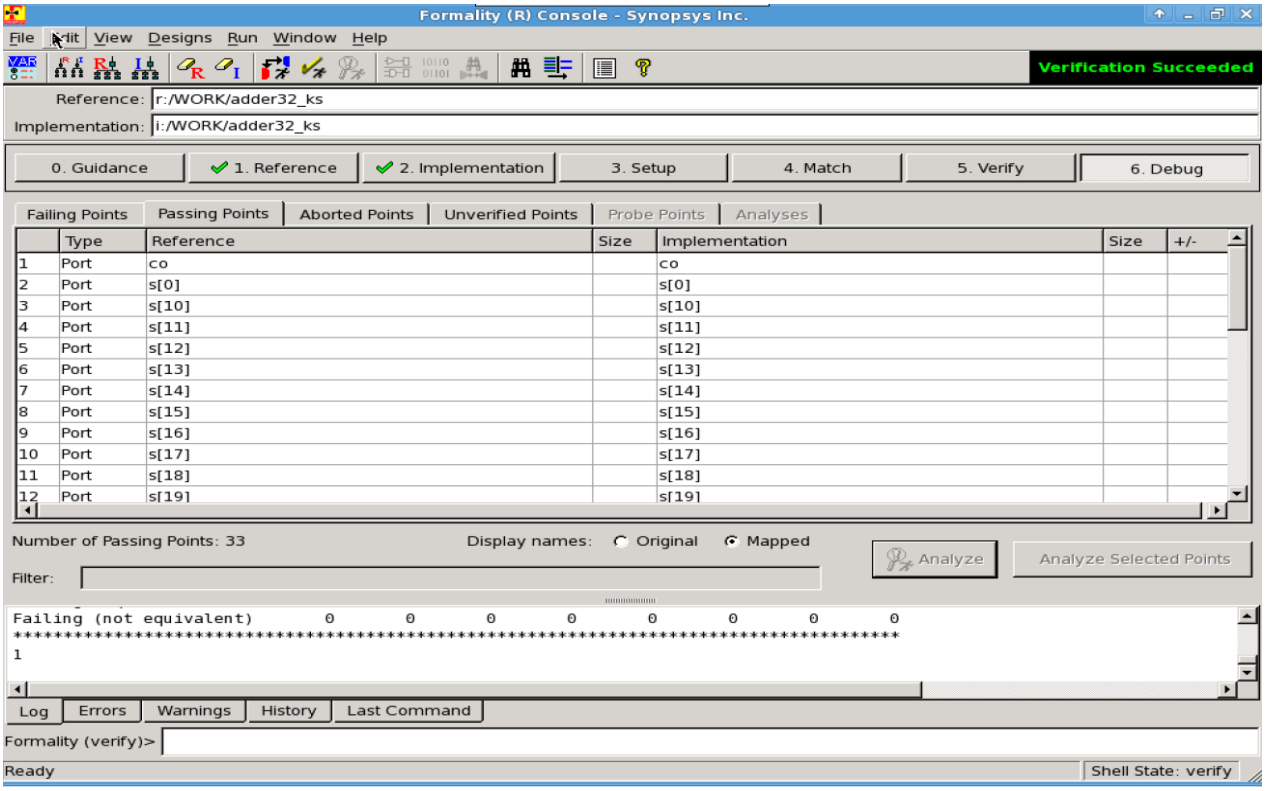
32-bit Carry-Skip Adder



32-bit Carry Select Adder



32-bit Kogge-Stone Adder



16×16 array multiplier