



Introduction to VLSI Design.

Design and Synthesis of Multi-Operand Adders.

Due to: 12/07/2012

***Adalberto Claudio.
A20294552.***

Index.

<u>ABSTRACTION.</u>	3
<u>INTRODUCTION.</u>	4
<u>BACKGROUND.</u>	5
<u>ARCHITECTURAL EXPLORATION OF ADDERS.</u>	7
<u>FUNCTIONAL VALIDATION AND VERIFICATION.</u>	12
<u>SYNTHESIS RESULTS.</u>	16
<u>CONCLUSIONS.</u>	19
<u>BIBLIOGRAPHY.</u>	20
<u>APENDIX.</u>	21

Abstraction.

The objective of this project has been to compare the different implementations of and adder [8:2], and how this determines the delay, area and power consumption of the designs. Also to familiarize to the Verilog design and the creating of testbenches, and how with the Verilog code an estimation of the delay, area and power consumption can be done to verify our design with Formality and to finish the creation of the layout using Virtuoso.

Introduction.

In this project we are going to design three different architectures of multi-operand adders. We are going to work with 8 operands of 4-bit width. To implement the multi-operand adders we are going to use Verilog and to test the correct behavior of our designs we will use different tools such as RTL Simulation, Logic Synthesis using Design Compiler, etc.

The three different ways in which we have implemented multi-operand adders are: conventional, linear and adder tree. We will study them in following sections. All implementations share modules such as the CSA module (Carry Save Adder) and the CPA module (Carry Ripple Adder). It is important to notice that, although the modules are the same for every single implementations, the way that the connections are realized are different.

Once we have finished the implementation of each design it is important to use tools that provide us enough information to determine the correct behavior of the design. Therefore, RTL Simulation, Logic Synthesis using Design Compiler, Place and Route using Encounter and Equivalence Checking using Formality are going to play an important role in our implementations.

Later, once we have finished the validation of our designs, we will study each model in terms of delay, power consumption and the area needed to implement the design and we compare all the designs.

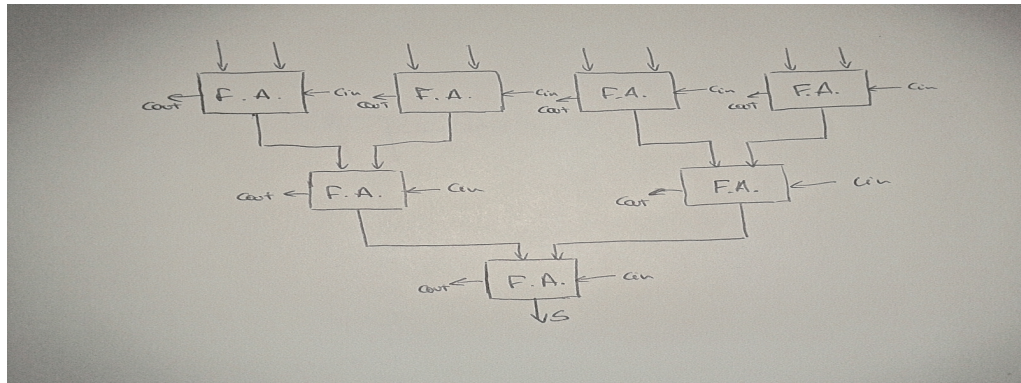
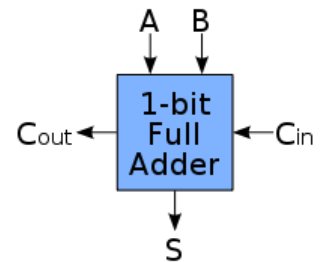
Background.

As we said in the previous section, we are going to implement three different multi-operand adders. We have to remember that we are going to work with 8 operands of 4-bit width each operand. First we are going to explain how each implementation works and later we will show how we implemented it. To understand how each multi-operand adder works we have use our class notes, the description of the final project and the book “*Digital Arithmetic* (Ercegovac and Lang 2004)”.

The first problem we found was the word length extension. When we are working with multi-operand we have to be careful with the length of the operands and the result. One solution is to extend the length of all operand by a factor p , where $p = \log_2(m)$, with m the number of operands. Thus, we are going to work with operands of 7-bit width.

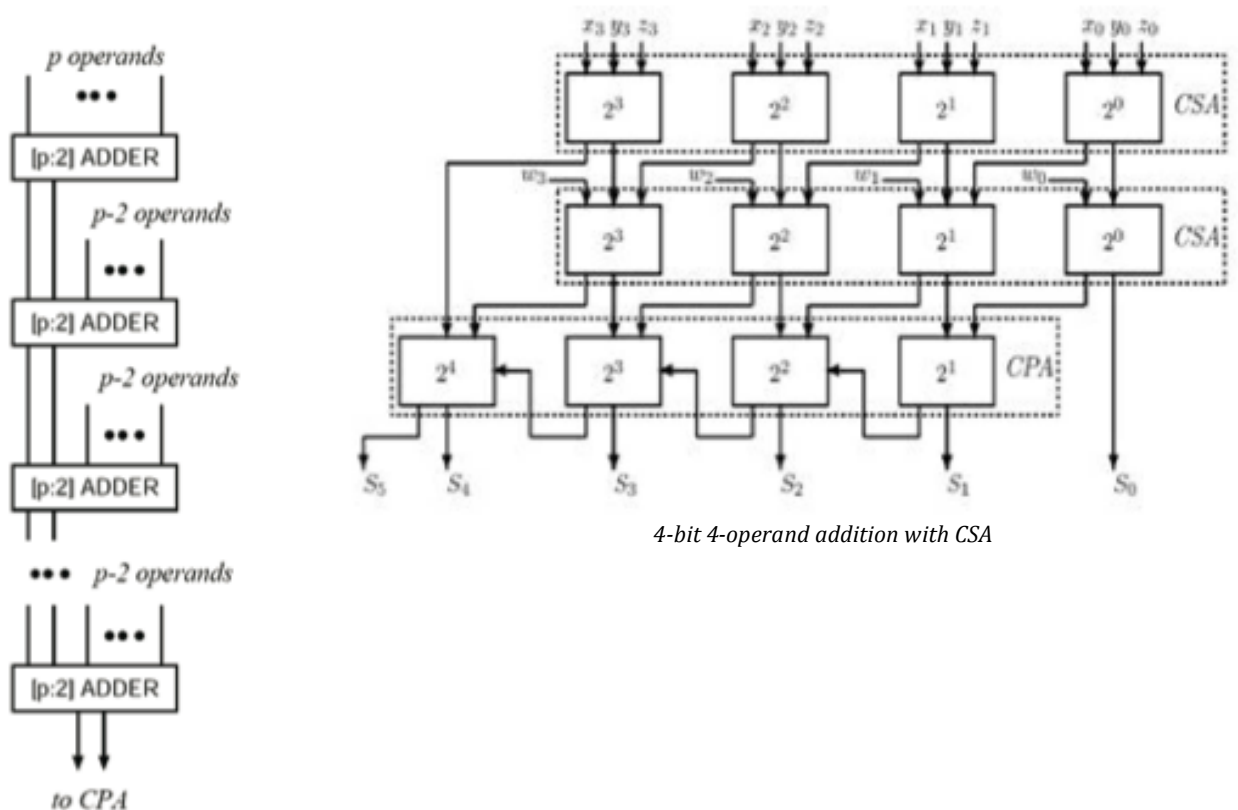
The designs that we are going to implement are:

1. Conventional implementation using full adders: this implementation consists in, using full adder, obtain the addition of all the operands. A full adder is as it is shown in the next figure. We have three inputs: two operands and the carry in, and two outputs, the sum and the carry out. To realize this implementation we decided to use 7 full adders with inputs and outputs of 8-bit width. We discarded the carry out of each block because the length of our operands is 4 bits, but it has been extended to solve the problem explained at the beginning of this section. Hence, all the carry outs are going to be zero so we won't consider them. In addition, we will equal all the carry in to zero. The next figure shows our implementation.



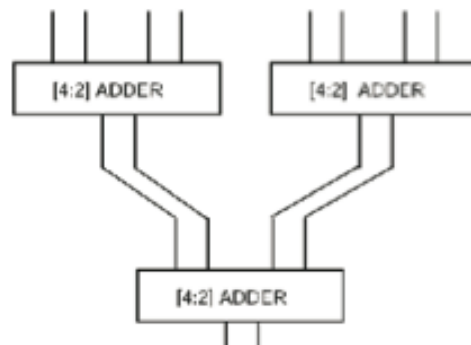
2. Linear implementation with CSA adder ([3:2]): we are going to need 7 adders. This kind of adders is used, for the addition of m operands the array consists of $\lceil (m-2)/(p-2) \rceil$ adders, since the first adder now receives p operands and the rest adders receives $p-2$. The scheme provided in the description of the final project is very useful to understand how a linear works. Once we have finished the addition with the carry save adders

[3:2] we will use the two outputs as inputs for a carry propagate adder. To implement the carry save adder we have used the next figure:



Linear implementation with [3:2] adders

3. Adder tree implementation with [4:2] adders: the idea of implement adders as a tree becomes from the possibility to reduce the number of levels of the linear adder. However, the number of adder required is the same than in the linear case.



8-operand addition with tree structure

Architectural Exploration of Adders.

In this section will be described how the implementation of the adders has been done, and how the modules have been built.

For the conventional adder Cadder.v, at first an adder of two 8-bit numbers has been implemented in *module adder8(s, co, a, b, ci)*, and with this module construct the combinational adder as shown in the following code, and in the background section an image helps to understand the design:

```
adder8 a0(w1, co1, m, n, w);
adder8 a1(w2, co2, o, p, w);
adder8 a2(w3, co3, q, r, w);
adder8 a3(w4, co4, s, t, w);
adder8 a4(w5, co5, w1, w2, w);
adder8 a5(w6, co6, w3, w4, w);
adder8 a6(u, co7, w5, w6, w);
```

Next design is the adder tree, as the following image shows it has been implemented with three [4:2] Adders, and this is the code, the module of [4:2] adder will be explain after the Linear Adder:

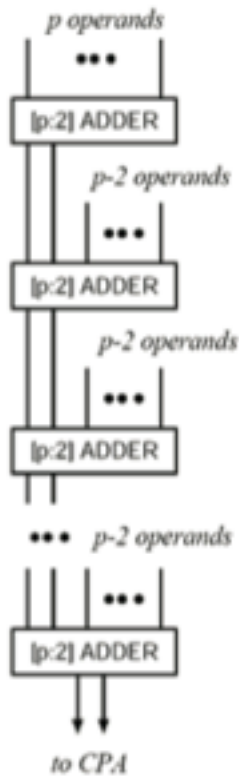
```
module addertree(u, m, n, o, p, q, r, s, t);

    output [8:0] u;
    input [6:0] m, n, o, p, q, r, s, t;
    wire [6:0] x, y, z, w;

    adder4to2 adder0(x, y, m, n, o, p);
    adder4to2 adder1(z, w, q, r, s, t);
    adder4to21 adder2(u, x, y, z, w);

endmodule //addertree
```

And for the linear adder, the implementation has been the following:



Using the $[3:2]$ CSA, a number of 6 and then to finish using a CPA to give the final number:

```
module linearadder(u, m, n, o, p, q, r, s, t);
```

```
output [8:0] u;
```

```
input [6:0] m, n, o, p, q, r, s, t;
```

```
wire [6:0] y, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14;
```

```
wire co;
```

```
csa c0(w1, w2, m, n, o);
```

```
csa2 c1(w3, w4, w1, w2, p);
```

```
csa2 c2(w5, w6, w3, w4, q);
```

```
csa2 c3(w7, w8, w5, w6, r);
```

```
csa2 c4(w9, w10, w7, w8, s);
```

```
csa2 c5(w11, w12, w9, w10, t);
```

```
cpa cp0(y, co, w11, w12);
```

```
assign u[0]=w11[0];
```

```
assign u[1]=y[0];
```

```
assign u[2]=y[1];
```

```
assign u[3]=y[2];
```

```
assign u[4]=y[3];
```



```

assign u[5]=y[4];
assign u[6]=y[5];
assign u[7]=y[6];
assign u[8]=co;

```

```

endmodule

```

The module of the [4:2] adder implementation as been the following, two CSA, first one with the 3 first inputs and carry in zero, and second one with the sum of the previous one and the fourth input, also with the carry in zero, then to finish a CPA, that takes the carry out of the first CSA and the sum of the second one, but only the bits most significant bits until 1. The following code shows this:

```

module csa1(s, co, x, y, z);

```

```

    output [6:0] s;
    output [6:0] co;
    input [6:0] x;
    input [6:0] y;
    input [6:0] z;

    adder a0(s[0], co[0], x[0], y[0], z[0]);
    adder a1(s[1], co[1], x[1], y[1], z[1]);
    adder a2(s[2], co[2], x[2], y[2], z[2]);
    adder a3(s[3], co[3], x[3], y[3], z[3]);
    adder a4(s[4], co[4], x[4], y[4], z[4]);
    adder a5(s[5], co[5], x[5], y[5], z[5]);
    adder a6(s[6], co[6], x[6], y[6], z[6]);

```

```

endmodule // CSA1

```

```

module csa2(s, co, x, y, cin);

```

```

    output [6:0] s;
    output [6:0] co;
    input [6:0] x;
    input [6:0] y;
    input [6:0] cin;
    wire w;

    assign w =0;
    adder a0(s[0], co[0], x[0], y[0], w);
    adder a1(s[1], co[1], x[1], y[1], cin[0]);
    adder a2(s[2], co[2], x[2], y[2], cin[1]);

```

```

    adder a3(s[3], co[3], x[3], y[3], cin[2]);
    adder a4(s[4], co[4], x[4], y[4], cin[3]);
    adder a5(s[5], co[5], x[5], y[5], cin[4]);
    adder a6(s[6], co[6], x[6], y[6], cin[5]);

```

```

endmodule // CSA2

```

```

module cpa(s, co, x, cin);

```

```

    output [6:0] s;
    output co;
    input [6:0] x;
    input [6:0] cin;
    wire w, w1, w2, w3, w4, w5, w6;

    assign w = 0;
    adder a0(s[0], w1, x[0], cin[0], w);
    adder a1(s[1], w2, x[1], cin[1], w1);
    adder a2(s[2], w3, x[2], cin[2], w2);
    adder a3(s[3], w4, x[3], cin[3], w3);
    adder a4(s[4], w5, x[4], cin[4], w4);
    adder a5(s[5], w6, x[5], cin[5], w5);
    adder a6(s[6], co, x[6], cin[6], w6);

```

```

endmodule // CPA

```

```

module adder4to2(sf1, sf0, x, y, z, w);

```

```

    output [6:0] sf1, sf0;
    input [6:0] x,y,z,w;
    wire [6:0] cp, sp, cs, ss, s, incpa;
    wire co;

    csa1 c1(sp, cp, x, y, z);
    csa2 c2(ss, cs, w, sp, cp);
    assign incpa[0] = ss[1];
    assign incpa[1] = ss[2];
    assign incpa[2] = ss[3];
    assign incpa[3] = ss[4];
    assign incpa[4] = ss[5];
    assign incpa[5] = ss[6];
    assign incpa[6] = cp[6];
    cpa cpa0(s, co, cs, incpa);
    assign sf0[0] = ss[0];

```

```

    assign sf0[1] = s[0];
    assign sf0[2] = s[1];
    assign sf0[3] = s[2];
    assign sf0[4] = 0;
    assign sf0[5] = 0;
    assign sf0[6] = 0;
    assign sf1[0] = 0;
    assign sf1[1] = 0;
    assign sf1[2] = 0;
    assign sf1[3] = 0;
    assign sf1[4] = s[3];
    assign sf1[5] = s[4];
    assign sf1[6] = 0;

```

```

endmodule // adder4to2

```

```

module adder4to21(sf, x, y, z, w);

```

```

    output [8:0] sf;
    input  [6:0] x,y,z,w;
    wire  [6:0] cp, sp, cs, ss, s, incpa;
    wire      co;

```

```

    csa1 c1(sp, cp, x, y, z);
    csa2 c2(ss, cs, w, sp, cp);
    assign incpa[0] = ss[1];
    assign incpa[1] = ss[2];
    assign incpa[2] = ss[3];
    assign incpa[3] = ss[4];
    assign incpa[4] = ss[5];
    assign incpa[5] = ss[6];
    assign incpa[6] = cp[6];
    cpa cpa0(s, co, cs, incpa);
    assign sf[0] = ss[0];
    assign sf[1] = s[0];
    assign sf[2] = s[1];
    assign sf[3] = s[2];
    assign sf[4] = s[3];
    assign sf[5] = s[4];
    assign sf[6] = s[5];
    assign sf[7] = s[6];
    assign sf[8] = co;

```

```

endmodule // adder4to21

```

Functional Validation and Verification.

The three implementations of the adder [8:2] have been tested with the same testbench, but changing the module used for the test, adder tree, conventional and linear implementations.

As we can see in the next piece of code of the test bench, the only line that needs to be changed for the test will be:

```
linearadder adder1(u, m, n, o, p, q, r, s, t);
```

As it can be noticed in this particular case the module tested is the linear implementation.

Four cases have been tested, three regular sums and a the case of the maximum number that can be obtained for the implementation, to test if the overflow is working fine.

```
`timescale 1ns/10ps
```

```
module stimulus;
```

```
//reg clk;
```

```
reg ci;
```

```
reg [6:0] m, n, o, p, q, r, s, t;
```

```
wire [8:0] u;
```

```
linearadder adder1(u, m, n, o, p, q, r, s, t);
```

```
initial begin
```

```
//      clk = 1'b0;
```

```
//      forever begin #5 clk = ~clk;
```

```
//      $display("At Time: %d u=%d", $time, u);      end
```

```
end
```

```
initial begin
```

```
    $shm_open("shm.db",1); // Opens a waveform database
```

```
    $shm_probe("AS"); // Saves all signals to database
```

```
    #1000 $finish;
```

```
    $shm_close(); // Closes the waveform database
```

```
end
```

```
// Stimulate the Input Signals
```

```
initial begin
```

```
    m = 0;
```

```
    n = 1;
```

```
    o = 2;
```

```

    p = 3;
    q = 4;
    r = 5;
    s = 6;
    t = 7;
    #100 $display("At Time: %d Result=%d ",$time,u);

    m = 1;
    n = 1;
    o = 1;
    p = 1;
    q = 1;
    r = 1;
    s = 1;
    t = 1;
    #100 $display("At Time: %d Result=%d ",$time,u);

    m = 3;
    n = 1;
    o = 2;
    p = 3;
    q = 4;
    r = 5;
    s = 6;
    t = 7;
    #100 $display("At Time: %d Result=%d ",$time,u);

    m = 15;
    n = 15;
    o = 15;
    p = 15;
    q = 15;
    r = 15;
    s = 15;
    t = 15;
    #100 $display("At Time: %d Result=%d ",$time,u);

end

endmodule // stimulus

```

The following are the verification that has been made to check the design of three implementation of the [8:2] adder, all with Formality.

- **Adder Tree.**

Formality (R) Console - Synopsys Inc.

File Edit View Designs Run Window Help

Reference: r:/WORK/addertree
Implementation: i:/WORK/addertree

0. Guidance 1. Reference 2. Implementation 3. Setup 4. Match 5. Verify 6. Debug

Failing Points	Passing Points	Aborted Points	Unverified Points	Probe Points	Analyses
Type	Reference	Size	Implementation	Size	+/-
1	Port u[0]		u[0]		
2	Port u[1]		u[1]		
3	Port u[2]		u[2]		
4	Port u[3]		u[3]		
5	Port u[4]		u[4]		
6	Port u[5]		u[5]		
7	Port u[6]		u[6]		
8	Port u[7]		u[7]		
9	Port u[8]		u[8]		

Number of Passing Points: 9 Display names: Original Mapped

Filter:

Analyze Analyze Selected Points

Failing (not equivalent) 0 0 0 0 0 0 0 0 0

1

Log Errors Warnings History Last Command

Formality (verify)>

Ready Shell State: verify

- **Linear Adder.**

Formality (R) Console - Synopsys Inc.

File Edit View Designs Run Window Help

Reference: r:/WORK/linearadder
Implementation: i:/WORK/linearadder

0. Guidance 1. Reference 2. Implementation 3. Setup 4. Match 5. Verify 6. Debug

Failing Points	Passing Points	Aborted Points	Unverified Points	Probe Points	Analyses
Type	Reference	Size	Implementation	Size	+/-
1	Port u[0]		u[0]		
2	Port u[1]		u[1]		
3	Port u[2]		u[2]		
4	Port u[3]		u[3]		
5	Port u[4]		u[4]		
6	Port u[5]		u[5]		
7	Port u[6]		u[6]		
8	Port u[7]		u[7]		
9	Port u[8]		u[8]		

Number of Passing Points: 9 Display names: Original Mapped

Filter:

Analyze Analyze Selected Points

Failing (not equivalent) 0 0 0 0 0 0 0 0 0

1

Log Errors Warnings History Last Command

Formality (verify)>

- **Conventional Adder:**

Formality (R) Console - Synopsys Inc.

File Edit View Designs Run Window Help

Verification Succeeded

Reference: r:/WORK/cadder
Implementation: i:/WORK/cadder

0. Guidance 1. Reference 2. Implementation 3. Setup 4. Match 5. Verify 6. Debug

Failing Points		Passing Points	Aborted Points	Unverified Points	Probe Points	Analyses
	Type	Reference		Size	Implementation	Size +/-
1	Port	u[0]			u[0]	
2	Port	u[1]			u[1]	
3	Port	u[2]			u[2]	
4	Port	u[3]			u[3]	
5	Port	u[4]			u[4]	
6	Port	u[5]			u[5]	
7	Port	u[6]			u[6]	
8	Port	u[7]			u[7]	

Number of Passing Points: 8 Display names: Original Mapped

Filter:

Analyze Analyze Selected Points

Failing (not equivalent) 0 0 0 0 0 0 0 0 0

1

Log Errors Warnings History Last Command

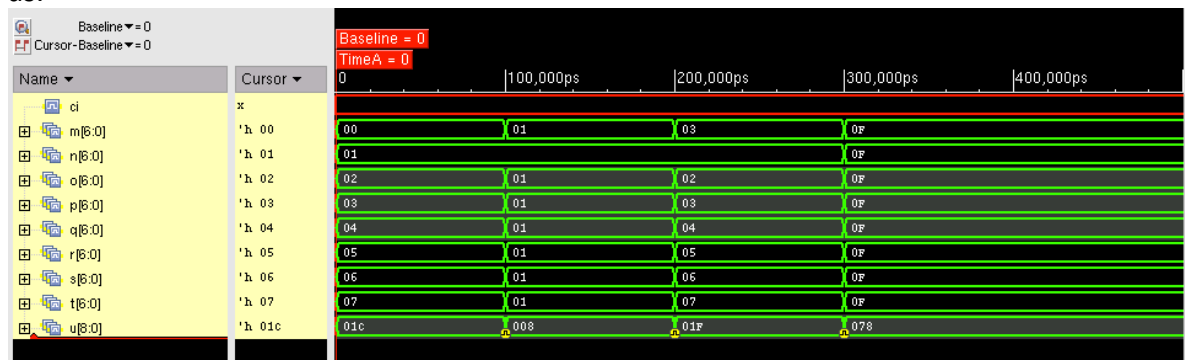
Formality (verify)>

Ready Shell State: verify

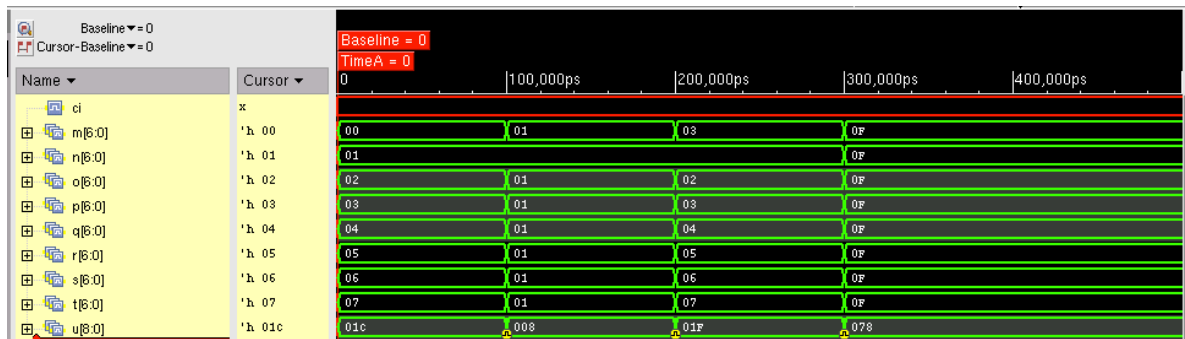
Synthesis Results.

The results that we have obtained in each verification are going to be showed in the following images:

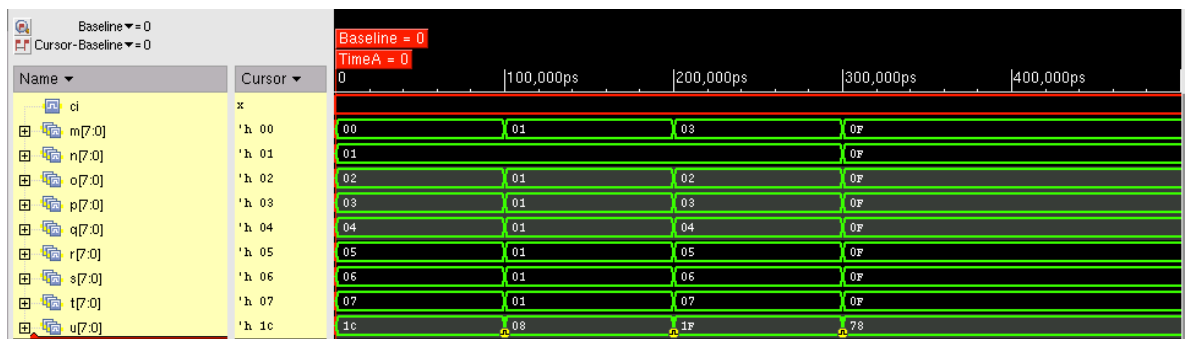
- RTL Simulation:** we are going to use the command `“verilog accu_test.v accu.v”` but instead of `accu_test.v` and `accu.test` we are going to use the Verilog code write by us.



Adder tree simvision simulation.



Linear adder simvision simulation.



Carry adder simvision simulation.

The Verilog code that has been used to test the three adders has been the same:

```
At Time:          100 Result= 28
At Time:          200 Result=  8
At Time:          300 Result= 31
At Time:          400 Result=120
```


This is what we obtained for all the simulations. Therefore, all adder added what was supposed to be added.

- **Logic Synthesis using Design Compiler:** in this case we are going to use the compiler to obtain the results. Therefore, the command needed is *"dc_shell -f compile_dc.tcl"*. The results are (we are going to show only one image, the other two are in the appendix):

```

INVX4
INVX8
LATCH
MUX2X1
NAND2X1
NAND3X1
NOR2X1
NOR3X1
OAI22X1
OR2X1
OR2X2
TBUF1
TBUF2
stimulus

At Time:          100 Result= 28
At Time:          200 Result=  8
At Time:          300 Result= 31
At Time:          400 Result=120
L22 "caddertest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 1724 accelerated events + 70 timing check events
CPU time: 0.1 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:  VERILOG-XL      08.20.001-p   Nov 30, 2012  12:56:14
aclaudio@saturn.ece.iit.edu:~% █

```

As it can be appreciate the adders add each time what is supposed. Then, the first addition is a 28, the second 8, the third 31 and the last 120.

- **Place and Route using Encounter:** as in the previous cases, we expect to obtain 28, 8, 31 and 120 in each addition implemented in our designs. All three adders provide us the same window with the same results; therefore, we won't show any figure because the results are exactly the same. The commands we have used are *"encounter -init encounter.tcl"* and *"verilog gsc145nm.v accu_test.v final.v"*.

In the following lines we are going to compare the different adders that have been implemented in terms of costs and performance. For this comparison we are going help us with tables:

	Power consumption (mW)	Area	Delay
Carry adder	0.6929	858.349675	--
Linear adder	8.0549 e-2	850.840876	0.53
Tree adder	0.1484	636.840081	2.88

In the previous table we have colored the better combinations for each field of study. We have studied the power consumption, the area and the delay of each adder. As

it is showed, the adder with less power consumption is the linear adder. In addition the faster adder is the linear adder again. On the other hand, the smallest adder is the tree adder; this is because, however linear and tree adder have the same number of [p: 2] adders, in the tree adder the number of levels is smaller than in the linear. Finally we had some problems because we obtain a "0" at the carry adder delay. This is impossible because it is supposed to be the greater value compared with tree adder and linear adder. Thus we decided to don't put any value in the table for carry adder delay, but we didn't find the error or we didn't understand why it became zero.

It would be very useful to know the cost of the area and the cost of the power consumption to compare which adder could be better, the adder tree or the linear adder. The area of the tree adder is a 25.16% slower than the linear adder and the linear adder consumes 45.22% less energy than the adder tree. Therefore, if we should pick one between these two adders, because the carry save adder has been discarded, we will select the linear adder because is faster and power consumption is smaller, so the total cost (including an hypothetic amortization) will be slower than in the tree adder case.

Conclusions.

As it has been shown through this research, we have different ways to implement a multi-operand adder: carry ripple adder, linear adder and tree adder. The first possibility always, carry ripple adder, is going to be discarded always because is the basic one and it doesn't offer us any strength: it is the slowest, with the higher power consumption and the one that has the biggest area. Thus, it is slow and expensive. Hence we have to select between the other two options that are the adder tree and the linear adder. The linear adder will have more levels, so the area will be bigger, but the other two important fields (delay and power consumption) we don't know firstly which one is going to be better. This will depend on the number of operands and the number of bits of each operand.

For this concrete case, 8 operands of 4 bits width, we have studied the different cases and finally we have decided that the linear adder is the best option to be implemented.

Bibliography.

Ercegovic, Milos D., y Tomas Lang. *Digital Arithmetic*. San Francisco: Morgan Kaufman Publishers, 2004.

Class notes. Ptof. Oruklu, E. ECE 429 Illinois Institute of Technology, Fall Semester, 2012.

Appendix.

Linearadder.v

```
// 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 // adder

module csa(s, co, x, y, z);

    output [6:0] s;
    output [6:0] co;
    input [6:0] x;
    input [6:0] y;
    input [6:0] z;

    adder a0(s[0], co[0], x[0], y[0], z[0]);
    adder a1(s[1], co[1], x[1], y[1], z[1]);
    adder a2(s[2], co[2], x[2], y[2], z[2]);
    adder a3(s[3], co[3], x[3], y[3], z[3]);
    adder a4(s[4], co[4], x[4], y[4], z[4]);
    adder a5(s[5], co[5], x[5], y[5], z[5]);
    adder a6(s[6], co[6], x[6], y[6], z[6]);

endmodule // CSA

module csa2(s, co, x, y, z);

    output [6:0] s;
    output [6:0] co;
    input [6:0] x;
    input [6:0] y;
    input [6:0] z;
    wire w;
```

```

assign w=0;
adder a0(s[0], co[0], x[0], w, z[0]);
adder a1(s[1], co[1], x[1], y[0], z[1]);
adder a2(s[2], co[2], x[2], y[1], z[2]);
adder a3(s[3], co[3], x[3], y[2], z[3]);
adder a4(s[4], co[4], x[4], y[3], z[4]);
adder a5(s[5], co[5], x[5], y[4], z[5]);
adder a6(s[6], co[6], x[6], y[5], z[6]);

```

```

endmodule // CSA

```

```

module cpa(s, co, x, cin);

```

```

output [6:0] s;
output co;
input [6:0] x;
input [6:0] cin;
wire w, w1, w2, w3, w4, w5, w6;

```

```

assign w =0;
adder a0(s[0], w1, x[1], cin[0], w);
adder a1(s[1], w2, x[2], cin[1], w1);
adder a2(s[2], w3, x[3], cin[2], w2);
adder a3(s[3], w4, x[4], cin[3], w3);
adder a4(s[4], w5, x[5], cin[4], w4);
adder a5(s[5], w6, x[6], cin[5], w5);
adder a6(s[6], co, w, cin[6], w6);

```

```

endmodule // CPA

```

```

module linearadder(u, m, n, o, p, q, r, s, t);

```

```

output [8:0] u;
input [6:0] m, n, o, p, q, r, s, t;
wire [6:0] y, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14;
wire co;

```

```

csa c0(w1, w2, m, n, o);
csa2 c1(w3, w4, w1, w2, p);
csa2 c2(w5, w6, w3, w4, q);
csa2 c3(w7, w8, w5, w6, r);
csa2 c4(w9, w10, w7, w8, s);
csa2 c5(w11, w12, w9, w10, t);
cpa cp0(y, co, w11, w12);

```

```

assign u[0]=w11[0];
assign u[1]=y[0];
assign u[2]=y[1];
assign u[3]=y[2];
assign u[4]=y[3];
assign u[5]=y[4];
assign u[6]=y[5];
assign u[7]=y[6];
assign u[8]=co;

```

```

endmodule // linear adder

```

Linearaddertest.v

```

`timescale 1ns/10ps

```

```

module stimulus;

```

```

//reg clk;
reg ci;
reg [6:0] m, n, o, p, q, r, s, t;

```

```

wire [8:0] u;

```

```

linearadder adder1(u, m, n, o, p, q, r, s, t);

```

```

initial begin
//    clk = 1'b0;
//    forever begin #5 clk = ~clk;
//    $display("At Time: %d u=%d,$time,u);  end
end

```

```

initial begin
    $shm_open("shm.db",1); // Opens a waveform database
    $shm_probe("AS"); // Saves all signals to database
    #1000 $finish;
    $shm_close(); // Closes the waveform database
end

```

```

// Stimulate the Input Signals

```

```

initial begin
    m = 0;
    n = 1;
    o = 2;
    p = 3;
    q = 4;

```

```

    r = 5;
    s = 6;
    t = 7;
    #100 $display("At Time: %d Result=%d ",$time,u);

    m = 1;
    n = 1;
    o = 1;
    p = 1;
    q = 1;
    r = 1;
    s = 1;
    t = 1;
    #100 $display("At Time: %d Result=%d ",$time,u);

    m = 3;
    n = 1;
    o = 2;
    p = 3;
    q = 4;
    r = 5;
    s = 6;
    t = 7;
    #100 $display("At Time: %d Result=%d ",$time,u);

    m = 15;
    n = 15;
    o = 15;
    p = 15;
    q = 15;
    r = 15;
    s = 15;
    t = 15;
    #100 $display("At Time: %d Result=%d ",$time,u);

end

endmodule // stimulus

Addertree.v

// 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 // adder

module csa1(s, co, x, y, z);

    output [6:0] s;
    output [6:0] co;
    input [6:0] x;
    input [6:0] y;
    input [6:0] z;

    adder a0(s[0], co[0], x[0], y[0], z[0]);
    adder a1(s[1], co[1], x[1], y[1], z[1]);
    adder a2(s[2], co[2], x[2], y[2], z[2]);
    adder a3(s[3], co[3], x[3], y[3], z[3]);
    adder a4(s[4], co[4], x[4], y[4], z[4]);
    adder a5(s[5], co[5], x[5], y[5], z[5]);
    adder a6(s[6], co[6], x[6], y[6], z[6]);

endmodule // CSA1

module csa2(s, co, x, y, cin);

    output [6:0] s;
    output [6:0] co;
    input [6:0] x;
    input [6:0] y;
    input [6:0] cin;
    wire w;

    assign w = 0;
    adder a0(s[0], co[0], x[0], y[0], w);
    adder a1(s[1], co[1], x[1], y[1], cin[0]);
    adder a2(s[2], co[2], x[2], y[2], cin[1]);
    adder a3(s[3], co[3], x[3], y[3], cin[2]);
    adder a4(s[4], co[4], x[4], y[4], cin[3]);
    adder a5(s[5], co[5], x[5], y[5], cin[4]);
    adder a6(s[6], co[6], x[6], y[6], cin[5]);

endmodule // CSA2

```

```

module cpa(s, co, x, cin);

    output [6:0] s;
    output co;
    input [6:0] x;
    input [6:0] cin;
    wire w, w1, w2, w3, w4, w5, w6;

    assign w = 0;
    adder a0(s[0], w1, x[0], cin[0], w);
    adder a1(s[1], w2, x[1], cin[1], w1);
    adder a2(s[2], w3, x[2], cin[2], w2);
    adder a3(s[3], w4, x[3], cin[3], w3);
    adder a4(s[4], w5, x[4], cin[4], w4);
    adder a5(s[5], w6, x[5], cin[5], w5);
    adder a6(s[6], co, x[6], cin[6], w6);

endmodule // CPA

```

```

module adder4to2(sf1, sf0, x, y, z, w);

```

```

    output [6:0] sf1, sf0;
    input [6:0] x,y,z,w;
    wire [6:0] cp, sp, cs, ss, s, incpa;
    wire      co;

    csa1 c1(sp, cp, x, y, z);
    csa2 c2(ss, cs, w, sp, cp);
    assign incpa[0] = ss[1];
    assign incpa[1] = ss[2];
    assign incpa[2] = ss[3];
    assign incpa[3] = ss[4];
    assign incpa[4] = ss[5];
    assign incpa[5] = ss[6];
    assign incpa[6] = cp[6];
    cpa cpa0(s, co, cs, incpa);
    assign sf0[0] = ss[0];
    assign sf0[1] = s[0];
    assign sf0[2] = s[1];
    assign sf0[3] = s[2];
    assign sf0[4] = 0;
    assign sf0[5] = 0;
    assign sf0[6] = 0;
    assign sf1[0] = 0;
    assign sf1[1] = 0;

```

```

        assign sf1[2] = 0;
        assign sf1[3] = 0;
        assign sf1[4] = s[3];
        assign sf1[5] = s[4];
        assign sf1[6] = 0;

endmodule // adder4to2

module adder4to21(sf, x, y, z, w);

    output [8:0] sf;
    input  [6:0] x,y,z,w;
    wire  [6:0] cp, sp, cs, ss, s, incpa;
        wire      co;

        csa1 c1(sp, cp, x, y, z);
        csa2 c2(ss, cs, w, sp, cp);
        assign incpa[0] = ss[1];
    assign incpa[1] = ss[2];
        assign incpa[2] = ss[3];
        assign incpa[3] = ss[4];
        assign incpa[4] = ss[5];
        assign incpa[5] = ss[6];
        assign incpa[6] = cp[6];
    cpa cpa0(s, co, cs, incpa);
        assign sf[0] = ss[0];
    assign sf[1] = s[0];
        assign sf[2] = s[1];
        assign sf[3] = s[2];
        assign sf[4] = s[3];
        assign sf[5] = s[4];
        assign sf[6] = s[5];
        assign sf[7] = s[6];
        assign sf[8] = co;

endmodule // adder4to21

module addertree(u, m, n, o, p, q, r, s, t);

    output [8:0] u;
    input  [6:0] m, n, o, p, q, r, s, t;
    wire  [6:0] x, y, z, w;

        adder4to2 adder0(x, y, m, n, o, p);
        adder4to2 adder1(z, w, q, r, s, t);
        adder4to21 adder2(u, x, y, z, w);

```

```
endmodule //addertree
```

Addertreetest.v

```
`timescale 1ns/10ps
```

```
module stimulus;
```

```
    //reg clk;
```

```
    reg ci;
```

```
    reg [6:0] m, n, o, p, q, r, s, t;
```

```
    wire [8:0] u;
```

```
    addertree adder1(u, m, n, o, p, q, r, s, t);
```

```
    initial begin
```

```
        //    clk = 1'b0;
```

```
        //    forever begin #5 clk = ~clk;
```

```
        //    $display("At Time: %d u=%d", $time, u);    end
```

```
    end
```

```
    initial begin
```

```
        $shm_open("shm.db", 1); // Opens a waveform database
```

```
        $shm_probe("AS"); // Saves all signals to database
```

```
        #1000 $finish;
```

```
        $shm_close(); // Closes the waveform database
```

```
    end
```

```
    // Stimulate the Input Signals
```

```
    initial begin
```

```
        m = 0;
```

```
        n = 1;
```

```
        o = 2;
```

```
        p = 3;
```

```
        q = 4;
```

```
        r = 5;
```

```
        s = 6;
```

```
        t = 7;
```

```
        #100 $display("At Time: %d Result=%d", $time, u);
```

```
        m = 1;
```

```
        n = 1;
```

```
        o = 1;
```

```
        p = 1;
```

```

    q = 1;
    r = 1;
    s = 1;
    t = 1;
    #100 $display("At Time: %d Result=%d ",$time,u);

    m = 3;
    n = 1;
    o = 2;
    p = 3;
    q = 4;
    r = 5;
    s = 6;
    t = 7;
    #100 $display("At Time: %d Result=%d ",$time,u);

end

endmodule // stimulus

```

Cadder.v (Conventional Adder)

```

// 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 // adder

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 // adder8

module cadder(u, m, n, o, p, q, r, s, t);

output [7:0] u;
input  [7:0] m, n, o, p, q, r, s, t;
wire  [7:0] w1, w2, w3, w4, w5, w6;
wire      w, co1, co2, co3, co4, co5, co6, co7;

assign w=0;
adder8 a0(w1, co1, m, n, w);
adder8 a1(w2, co2, o, p, w);
adder8 a2(w3, co3, q, r, w);
adder8 a3(w4, co4, s, t, w);
adder8 a4(w5, co5, w1, w2, w);
adder8 a5(w6, co6, w3, w4, w);
adder8 a6(u, co7, w5, w6, w);

endmodule // cadder

```

Caddertest.v

```

`timescale 1ns/10ps

module stimulus;

//reg clk;
reg ci;
reg [7:0] m, n, o, p, q, r, s, t;

wire [7:0] u;

```

```
cadder adder1(u, m, n, o, p, q, r, s, t);
```

```
initial begin
```

```
//    clk = 1'b0;
```

```
//    forever begin #5 clk = ~clk;
```

```
//    $display("At Time: %d u=%d", $time, u);    end
```

```
end
```

```
initial begin
```

```
    $shm_open("shm.db", 1); // Opens a waveform database
```

```
    $shm_probe("AS"); // Saves all signals to database
```

```
    #1000 $finish;
```

```
    $shm_close(); // Closes the waveform database
```

```
end
```

```
// Stimulate the Input Signals
```

```
initial begin
```

```
    m = 0;
```

```
    n = 1;
```

```
    o = 2;
```

```
    p = 3;
```

```
    q = 4;
```

```
    r = 5;
```

```
    s = 6;
```

```
    t = 7;
```

```
    #100 $display("At Time: %d Result=%d ", $time, u);
```

```
    m = 1;
```

```
    n = 1;
```

```
    o = 1;
```

```
    p = 1;
```

```
    q = 1;
```

```
    r = 1;
```

```
    s = 1;
```

```
    t = 1;
```

```
    #100 $display("At Time: %d Result=%d ", $time, u);
```

```
    m = 3;
```

```
    n = 1;
```

```
    o = 2;
```

```
    p = 3;
```

```
    q = 4;
```

```
    r = 5;
```

```
    s = 6;
```

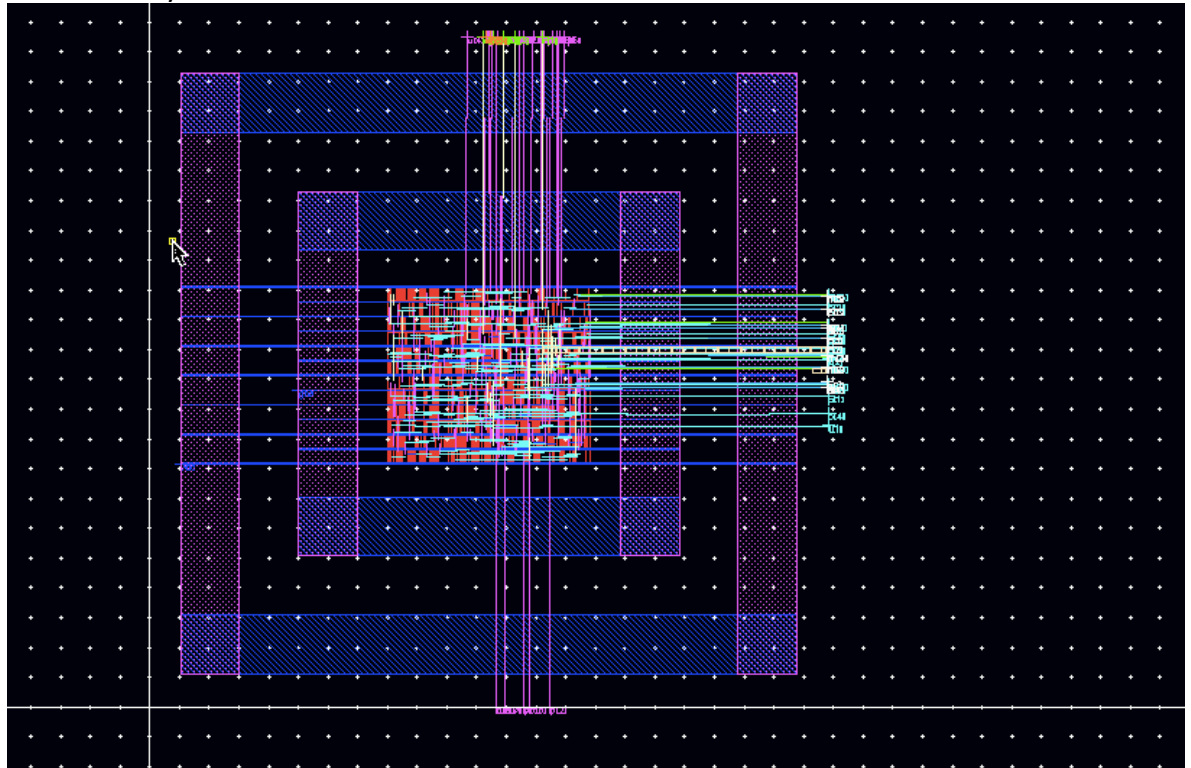
```
    t = 7;
```

```
    #100 $display("At Time: %d Result=%d ", $time, u);
```

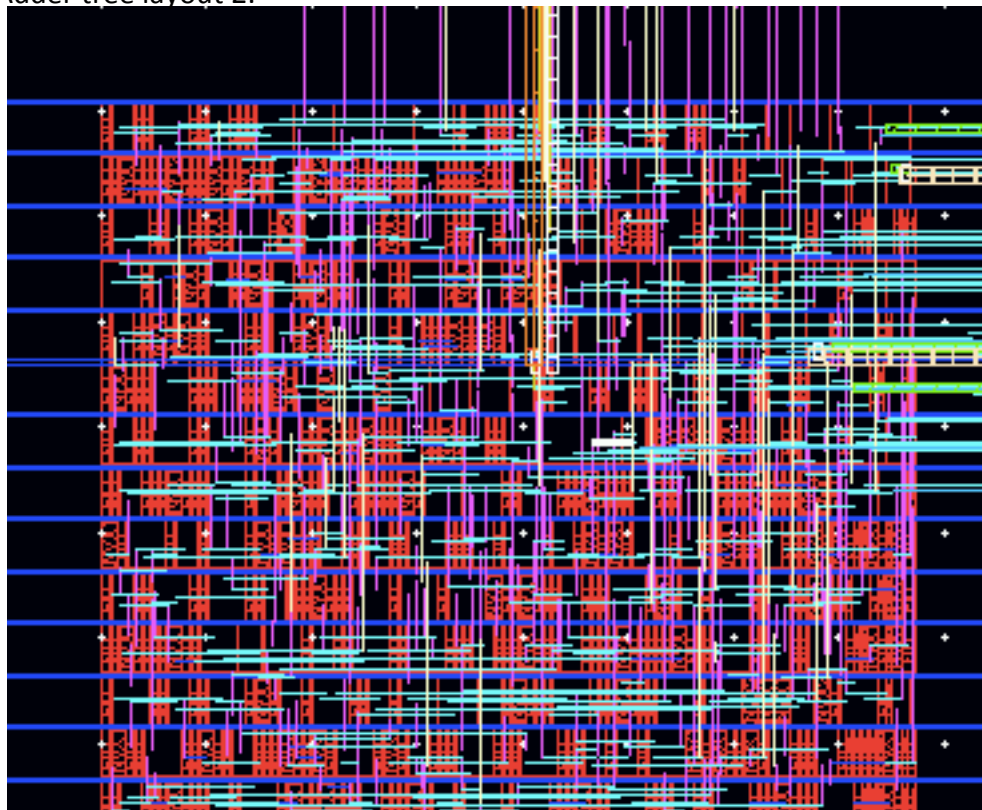
```
end  
endmodule // stimulus
```

Other figures obtained through the research:

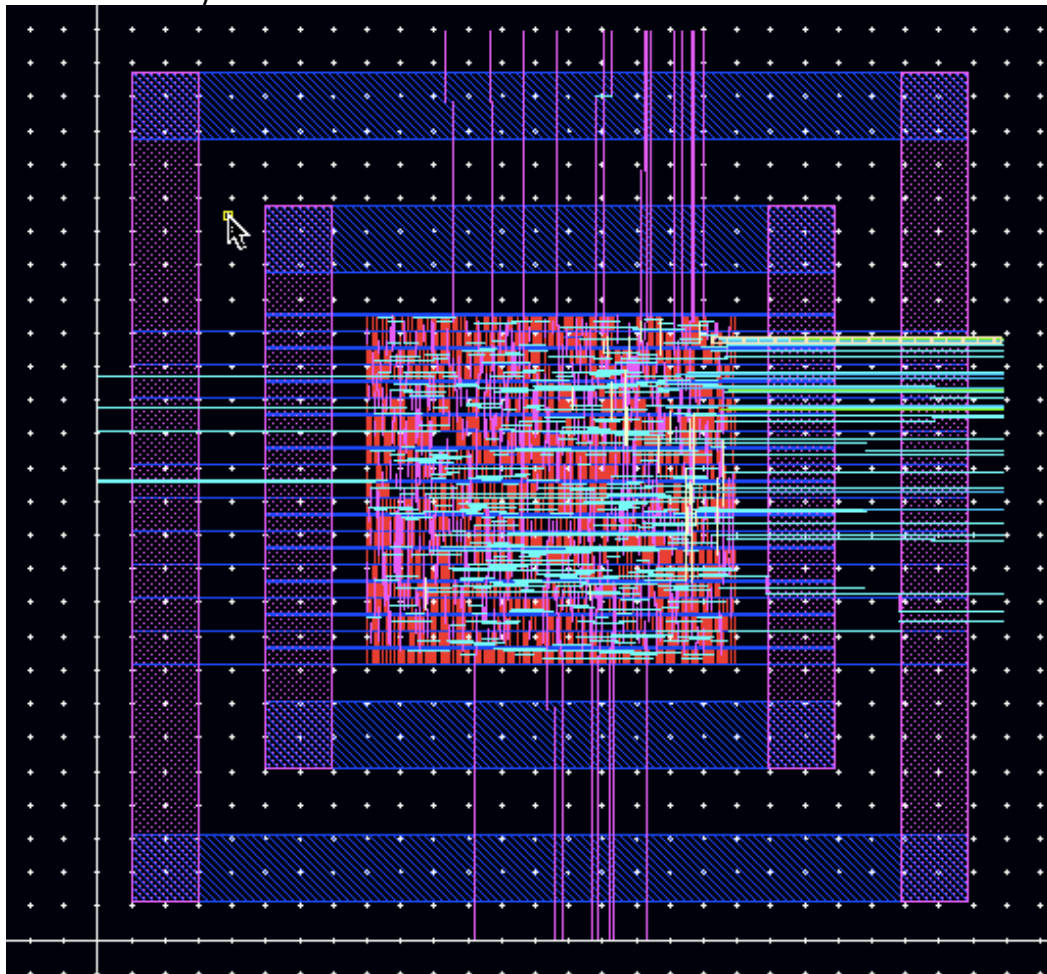
Adder tree layout:



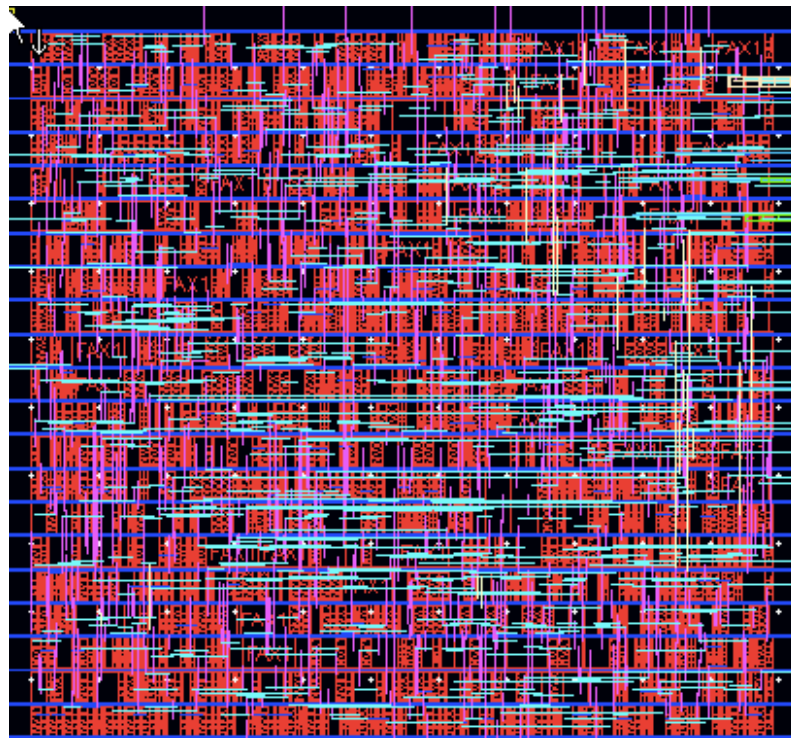
Adder tree layout 2:



Linear adder layout 1:



Linear adder layout2:



Carry adder Verilog test.

```
Terminal
File Edit View Terminal Tabs Help

Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134

For technical assistance please contact the Cadence Response Center at
1-877-CDS-4911 or send email to support@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkv@cadence.com

Compiling source file "caddertest.v"
Compiling source file "cadder.v"
Highest level modules:
stimulus

At Time:          100 Result= 28
At Time:          200 Result= 8
At Time:          300 Result= 31
At Time:          400 Result=120
L22 "caddertest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 754 accelerated events
CPU time: 0.1 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool: VERILOG-XL 08.20.001-p Nov 30, 2012 12:40:35
acclaudio@saturn.ece.iit.edu:~%
```

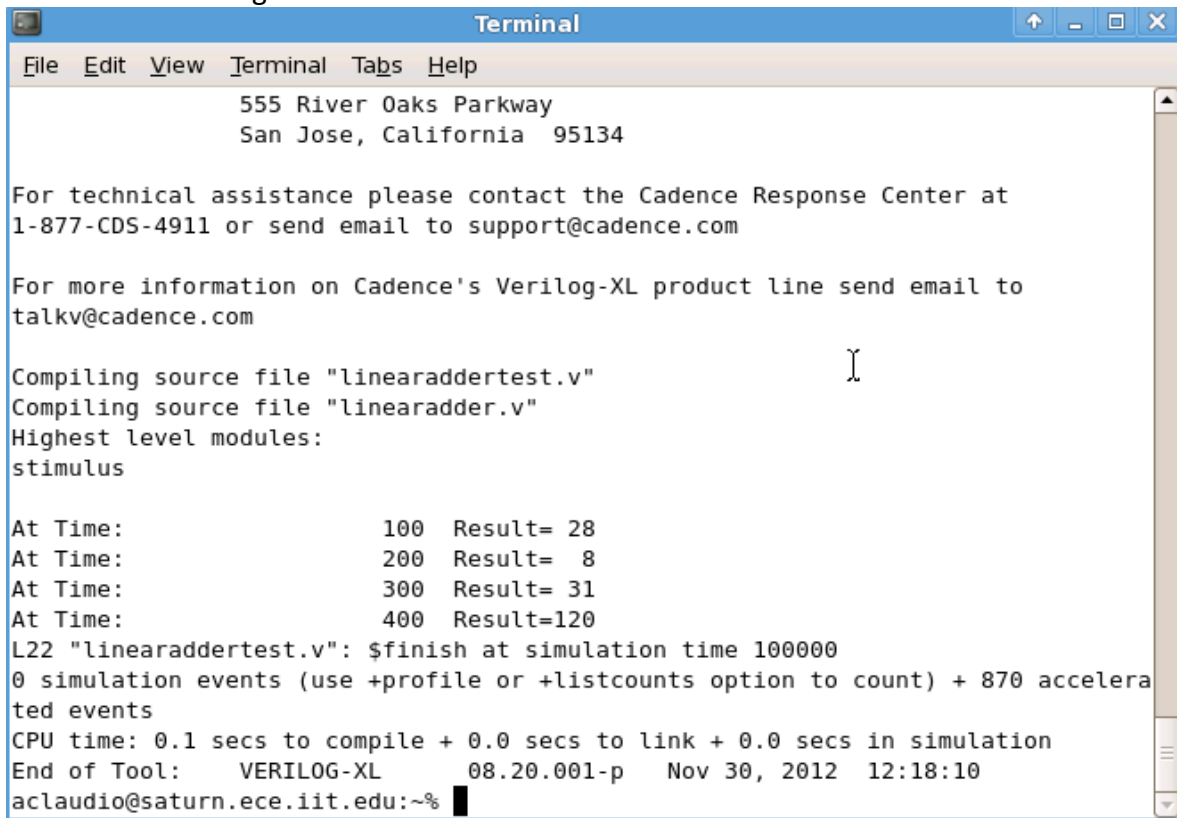
Linear adder gscl45nm test

```
Terminal
File Edit View Terminal Tabs Help

CLKBUF2
CLKBUF3
DFFNEGX1
DFFPOSX1
DFFSR
HAX1
LATCH
MUX2X1
NAND2X1
NOR2X1
OAI22X1
TBUF1
TBUF2
stimulus

At Time:          100 Result= 28
At Time:          200 Result= 8
At Time:          300 Result= 31
At Time:          400 Result=120
L22 "linearaddertest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 7872 accelerated events + 70 timing check events
CPU time: 0.1 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool: VERILOG-XL 08.20.001-p Nov 30, 2012 12:27:36
acclaudio@saturn.ece.iit.edu:~%
```

Linear adder Verilog test



```
Terminal
File Edit View Terminal Tabs Help
555 River Oaks Parkway
San Jose, California 95134

For technical assistance please contact the Cadence Response Center at
1-877-CDS-4911 or send email to support@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkv@cadence.com

Compiling source file "linearaddertest.v"
Compiling source file "linearadder.v"
Highest level modules:
stimulus

At Time:          100 Result= 28
At Time:          200 Result= 8
At Time:          300 Result= 31
At Time:          400 Result=120
L22 "linearaddertest.v": $finish at simulation time 100000
0 simulation events (use +profile or +listcounts option to count) + 870 accelera
ted events
CPU time: 0.1 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool: VERILOG-XL 08.20.001-p Nov 30, 2012 12:18:10
aclaudio@saturn.ece.iit.edu:~%
```