Mike Bodzay
Jeff Grindel
Final Project: Design and Synthesis of Multi-Operand Adders

12/7/12

Signature: _____

Signature: _____

Contents

Abstract

Mike Bodzay worked on the [3:2] Linear CSA, the Functional Validation and verification, Synthesis Results, and Conclusion and Future Work Sections of the report.

Jeff Grindel worked on the Carry Propagation Adder, [4:2] Tree CSA, and the [5:2] CSA, the Introduction, Background, Architectural Exploration of Adders and Bonus Design Sections of the report.

Both members worked on the test bench and independently ran the entire test benches with the Verilog code to verify all implementation of the adders worked. Also both members ran the synthesis of the adders to verify that the timing, area, and power were correct.

Introduction

The purpose of this project was to design and implement three different multi-operand adders and to compare them against each other to determine which was the most efficient, based off of the power, timing, and area results. These adders will be designed and implemented using data path circuit design, standard cell based design flow, and design validation and verification through the construction of Verilog programs. The adders will be synthesized using commercial EDA tools from Synopsys and Cadence Design Systems.

The first adder is simply a Carry Ripple adder which is the basic implementation of tying a series of full adders together in a linear method and has the carry out of one full adder as an input to the next full adder. The second adder is a [3:2] Carry Save adder in a linear implementation. This is similar to the carry ripple adder, but in this implementation the full adder does not need to wait for the previous carry out bit to be calculated before it does its own calculation. The final adder is a [4:2] Carry Save adder in a tree implementation which utilizes a lot fewer adders than the linear implementations.

The major design challenges that implementing this project were to ensure that all the verilog code was written correctly before any of the synthesis could take place. As we will discuss later, this was overcome by carful layout of the adders and generalized code that could be used repeatedly.

Background

Carry Ripple Adder

A Carry Ripple Adder (CRA) or Carry Propagation Adder (CPA) is the basic implementation of a linear set of full adder strung together to form a complete adder. A Full Adder takes three inputs, and generates two outputs, a sum and a carry out. This can be seen below in Figure 1 [1].

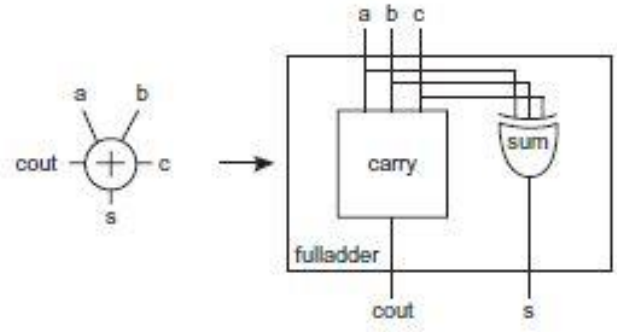

Figure 1: Full-Adder Implementation

In order to generate a full adder that took in multiple bit numbers, they were aligned so the Cout of the first Full Adder was the Cin for the second, etc. This generates a single-bit Cout, and a multiple bit sum. Seen below in Figure 2 [1].

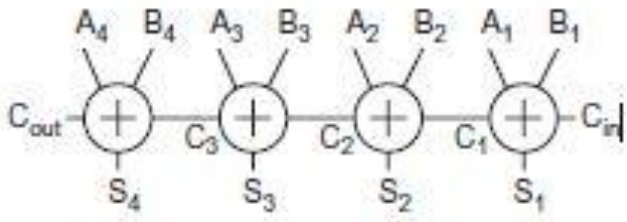

Figure 2: Multiple Bit Full Adder

Finally, the multiple bit full adders can be put together to form a complete multi-operand addition. This can be seen below in a 4-operand example in Figure 3. (A,B,C,D,Sum are multi-bit, and Ci and Co are just single-bit).

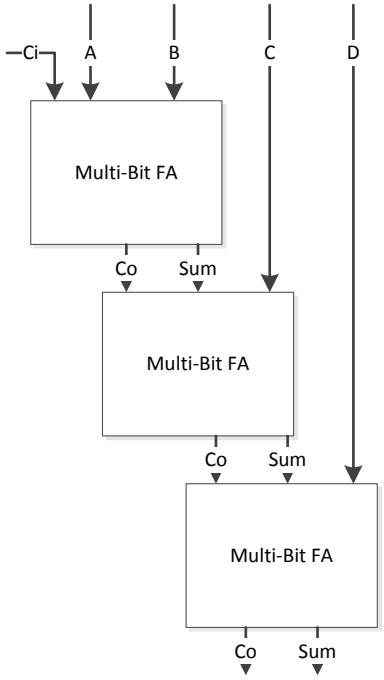

Figure 3: Multi-Operand Carry Propagation Adder

This can be expanded to p-operands by continually adding a Multi-Bit Full Adder to the next level, and bringing in a new operand. In total, for 8-Operand addition there needs to be seven multi-bit full adders.

[3:2] Linear Carry Save Adder

A [p:2] Carry Save Adder (CSA) is a way to take p bit-vectors and reduce them to just 2 bit-vectors(VS, VC). After they are reduced to the 2 bit-vectors they can be put through a full adder to get the sum and the carry out. A [3:2] Carry Save Adder (CSA) is simply a modified full adder, in our case to take in three 7-bit numbers and return two 7-bit numbers. The CSA allows for the three bits to be added together through a full adder and to generate a VS, and a VC that carries twice the weight. That is it is shifted to the left one bit than the generated sum. The picture below in Figure 4 shows the implementation of a [3:2] CSA [2].

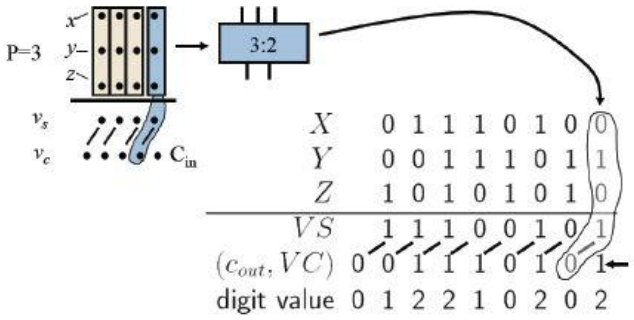

Figure 4: [3:2] Carry Save Adder Implementation

In our implementation of the [3:2] CSA, these adders will be put in a linear implementation. That is the first three numbers are summed together to obtain VS, and VC. These outputs are now the inputs for the next CSA along with the next number that needs to be added. After the final CSA takes place, the results are then fed to a Carry Propagation Adder (CPA) to obtain the Sum and Carry Bit; this can be seen below in Figure 5 [2].

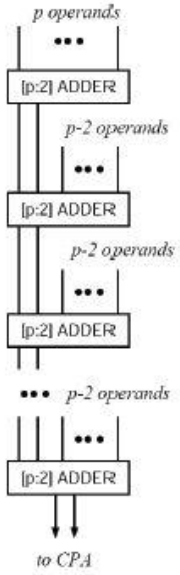

Figure 5: [p:2] Linear Implementation

[4:2] Tree Carry Save Adder

A [4:2] CSA is simply a modified full adder, in our case to take in four 7-bit numbers and return two 7-bit numbers. The CSA allows for the four bits to be added together through a full adder and to generate a VS, and a VC. In a [4:2] CSA there are two levels to it. These levels implement the [3:2] CSA, the first level takes 3 of the inputs, and generate the typical output for a [3:2] CSA. In the second level the fourth input is added to a sum from the previous level, and a carry from the previous bit of the previous level. The picture below in Figure 6 shows the implementation of a [4:2] CSA [2].

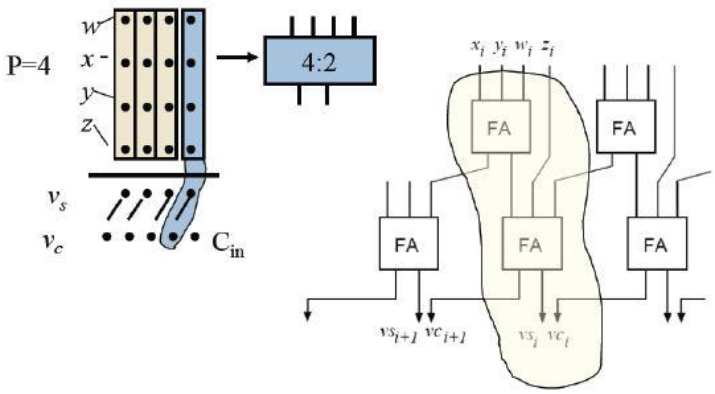

Figure 6: [4:2] Carry Save Adder Implementation

In our implementation of the [4:2] CSA, these adders will be put in a tree implementation. That is the first four numbers are summed together to obtain VS, and VC. And the last four number are summed together to obtain VS, and VC. The two sets of VS and VC then are inputted into a third [4:2] CSA on the second level. After the final CSA takes place, the results are then fed to a Carry Propagation Adder (CPA) to obtain the Sum and Carry Bit; this can be seen below in Figure 7 [2].

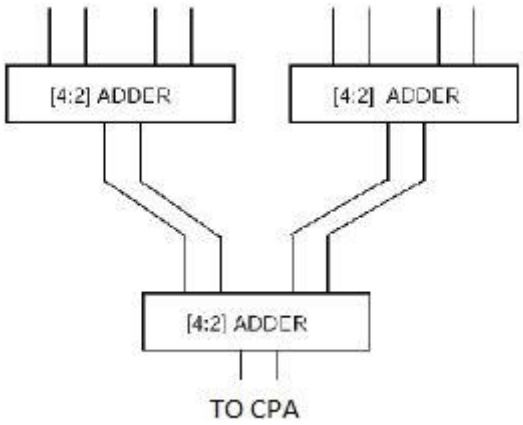

Figure 7: 8-operand Tree Implementation utilizing [4:2] CSA

Architectural Exploration of Adders

With the CPA, in order for the next iteration of adder to be completed it needs to wait until the previous adder is completely done. This adds a whole series of delays in the adder where even though the sum may be calculated it is still waited for the carry out to be calculated. With the CPA design it is going to be the slowest implementation because it is waiting for the propagation of the previous carry out to be used as a carry input. As you can see in the Verilog code below for the CPA, there is no carry forwarding, thus causing a propagation delay.

```
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], co, a[6], b[6], c6);
```

For the CSA the sum and carry vectors are calculated separately, thus not having to wait for a certain adder to be completed before the next iteration is calculated. In the [3:2] adder this was simply done by shifting the carry out vector to the left by one position. This gave each carry out twice the weight, since in binary a shift left instruction is the same a multiply by 2 instruction. At this level, all of the [3:2] adders were then implemented in a linear method similar to the CPA adder, but this time as each operand was being factored into the calculation there was not a significant amount of time spent on waiting for the previous adder to finish with its computations before moving on. Simply with this bit shift, it increased the performance of the adder. Once all of the VS and VC are calculated they are put through a CPA, having a single CPA is a lot less than a series of seven of them. Unlike the CPA, the [3:2] CSA utilized this concept of port forwarding, this can be seen in the Verilog code below. Notice that the carry out is shifted to the left by one (the first bit outputs to co[1]).

```
adder a0(s[0], co[1], a[0], b[0], c[0] );
adder a1(s[1], co[2], a[1], b[1], c[1]);
adder a2(s[2], co[3], a[2], b[2], c[2]);
adder a3(s[3], co[4], a[3], b[3], c[3]);
adder a4(s[4], co[5], a[4], b[4], c[4]);
adder a5(s[5], co[6], a[5], b[5], c[5]);
adder a6(s[6], c1, a[6], b[6], c[6]);
```

In the [4:2] Implementation it had a similar design as the [3:2], but added an additional operand into the equation. The first three operands were added together, and the carry shifted over, then the sum, another operand, and the previous carry are summed together. In this implementation the carry is not going to be a big problem because instead of waiting for the carry of that adder it is utilizing the carry of the previous one, which has had plenty of time to be able to calculate. On top of this, the [4:2] CSA implemented a tree structure. This means that while the first four operands are being calculated, the last four can be calculated simultaneously since there is absolutely no dependency on the operands. This greatly reduces the time. They then go through a second level [4:2] adder, but since all 8-operands are calculated simultaneously in the first level, there is minimal delay on the second level. Once all of the VS

and VC are calculated they are put through a CPA, having a single CPA is a lot less than a series of seven of them. Finally with the implementation of the [4:2] CSA, there are 2 layers (seen in Figure 6). These only depend on the current sum and the previous carry. This can be seen in the Verilog code below.

```
adder s00(ts[0],tc[1],a[0],b[0],c[0]);
adder s01(ts[1],tc[2],a[1],b[1],c[1]);
adder s02(ts[2],tc[3],a[2],b[2],c[2]);
adder s03(ts[3],tc[4],a[3],b[3],c[3]);
adder s04(ts[4],tc[5],a[4],b[4],c[4]);
adder s05(ts[5],tc[6],a[5],b[5],c[5]);
adder s06(ts[6],c1,   a[6],b[6],c[6]);

adder s10(s[0],co[1],ts[0],d[0],tc[0]);
adder s11(s[1],co[2],ts[1],d[1],tc[1]);
adder s12(s[2],co[3],ts[2],d[2],tc[2]);
adder s13(s[3],co[4],ts[3],d[3],tc[3]);
adder s14(s[4],co[5],ts[4],d[4],tc[4]);
adder s15(s[5],co[6],ts[5],d[5],tc[5]);
adder s16(s[6],c2,  ts[6],d[6],tc[6]);
```

Theoretically, [4:2] Tree implementation should be the fastest design due to carry forwarding and the simultaneous calculation of four operands at a time. This will be discussed further be discussed in the Synthesis Results section.

Bonus Design

The bonus was to construct a [5:2] CSA and to implement 10-operand addition. This was done utilizing a [5:2] CSA, as seen in the Figure 8 below [2].

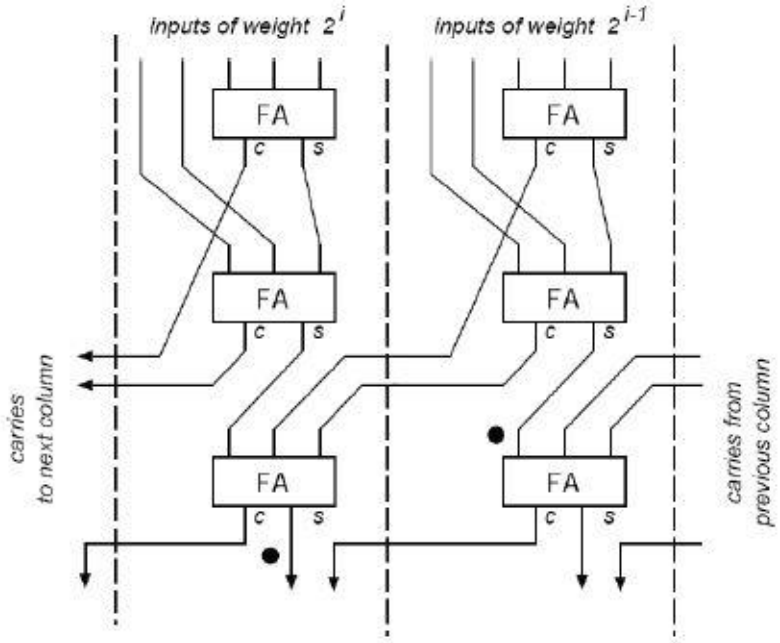

Figure 8: [5:2] adder block

The [5:2] adder was very similar to the other CSA, with this implementation it had three levels of multi-bit full adder described earlier in the report. Utilizing these Carry Outs of each level could be manipulated to become Carry Inputs for the other levels of Full Adders. As seen in the figure above. The first level of full adders takes in the first three operands and generates a sum and carry, the sum is fed directly to the next level, and the carry is sent to the input of the third level full adder but shifted to the left by one. The second level takes the sum from the first level, and the other two operands. This generates another sum and carry. The sum is sent to the input of the third level full adder, and the carry is sent to the input of the third level full adder but shifted to the left by one. Finally the third level takes in the sum from level two, and the two carries from the level one and level two from the right section. This final level generates the VS and VC that can then be inputted into a simple CPA to generate the sum and carry out.

In order to account for all the 10 operands, a tree structure where the first 5 operands are inputted into a [5:2] CSA, and the last five operands are inputted into another [5:2] CSA. The 4 total outputs of the [5:2] (VS1, VC1, VS2, VC2) are then inputted into a [4:2] CSA to generate VS3 and VC3. These outputs are then inputted into a CPA where the Ci could then be factored in. This generates the single bit Cout and the multi-bit Sum. This implementation of 10-operand addition can be seen below in Figure 9.

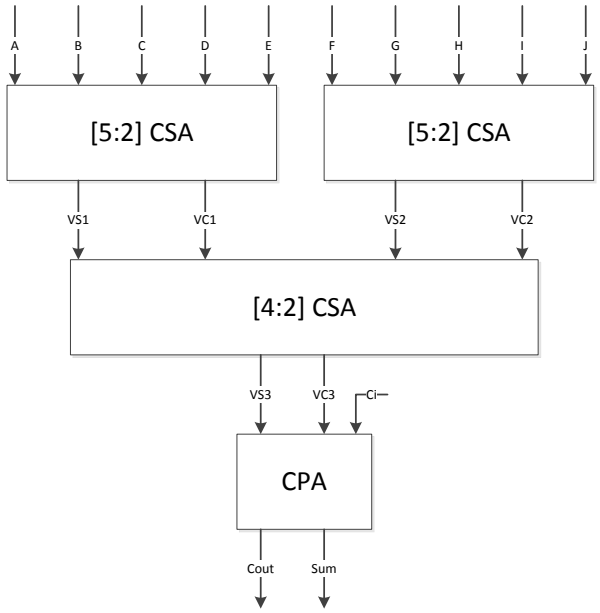

Figure 9: 10-operand Tree Implementation

A test program was written to account for 10 operands. There were 7 test cases and when the program was run it generated a verilog.log file (found in the appendix). From the verilog.log and the test case the following can be summarized in Table X below.

Table 1: 10 Operand Adder Test Results

A	B	C	D	E	F	G	H	I	J	Ci	Sum	Co
1	1	1	1	1	1	1	1	1	1	0	10	0
1	2	3	4	5	6	7	8	9	10	0	55	0
12	12	12	12	12	12	12	12	12	12	1	121	0
0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	6	1	1	44	0
15	15	15	15	15	15	15	15	1	1	1	123	0
10	15	15	15	7	12	13	15	15	12	1	0	1

Functional Validation and Verification

In order to ensure proper functionality throughout the design process of our 8 operand adders, we employed different verification techniques. First, the original verilog code for each adder was tested using a test bench. The test bench simulated several different input combinations to verify that the code produced the correct sum for input values between 0 and 15 for each operand. In addition, the carry-in and carry-out bits were tested by running a simulation with a carry-in value of 1 and a simulation with a sum of 128 respectively. Having a sum of 128 verifies the carry-out bit because the sum output is only 7 bits long and 128 is an 8 bit number. The list of simulations run can be seen in Table 2. The same test bench was used to verify each of the three different designs, thus ensuring that the carry ripple adder, 3:2 linear, and 4:2 tree adder produced the same results as can be seen in Table 3.

Table 2: Test Bench Simulations

Simulation Number	A	B	C	D	E	F	G	H	Ci
1	1	1	1	1	1	1	1	1	0
2	1	2	3	4	5	6	7	8	0
3	15	15	15	15	15	15	15	15	0
4	10	14	15	0	4	6	9	13	0
5	1	2	3	4	5	6	7	8	1
6	16	15	15	15	15	15	15	15	1
7	16	16	16	16	16	16	16	15	1

Validation after gate level synthesis (.vh file), and placement and routing (final.v file) was achieved by using the same test bench as the original code. This allowed us to check the results of the simulations in two different ways. We could verify that each simulation gave the correct sum and we could compare the results to those obtained from the test bench of the original code (see Table 3 below).

Although using the test bench shows proper functionality during different steps of the design process, in order to guarantee the correctness of our models, we must use formal equivalence checking. To do this we use Synopsys Formality. This software can verify the original verilog code to the standard cell layout that has been synthesized and optimized [3]. The successful equivalence reports for the carry ripple design, 3:2 linear adder design, and 4:2 tree adder design can been seen in figures 10, 11, and 12 respectively.

Table 3: Simulation Results for Each Case of Each Design

		Simulation Number													
		1		2		3		4		5		6		7	
		Sum	Co	Sum	Co	Sum	Co	Sum	Co	Sum	Co	Sum	Co	Sum	Co
Carry Ripple Adder	Verilog Code	8	0	36	0	120	0	71	0	37	0	122	0	0	1
	Standard Cell Netlist	8	0	36	0	120	0	71	0	37	0	122	0	0	1
	Cell Layout	8	0	36	0	120	0	71	0	37	0	122	0	0	1
[3:2] Linear Adder	Verilog Code	8	0	36	0	120	0	71	0	37	0	122	0	0	1
	Standard Cell Netlist	8	0	36	0	120	0	71	0	37	0	122	0	0	1
	Cell Layout	8	0	36	0	120	0	71	0	37	0	122	0	0	1
[4:2] Tree Adder	Verilog Code	8	0	36	0	120	0	71	0	37	0	122	0	0	1
	Standard Cell Netlist	8	0	36	0	120	0	71	0	37	0	122	0	0	1
	Cell Layout	8	0	36	0	120	0	71	0	37	0	122	0	0	1

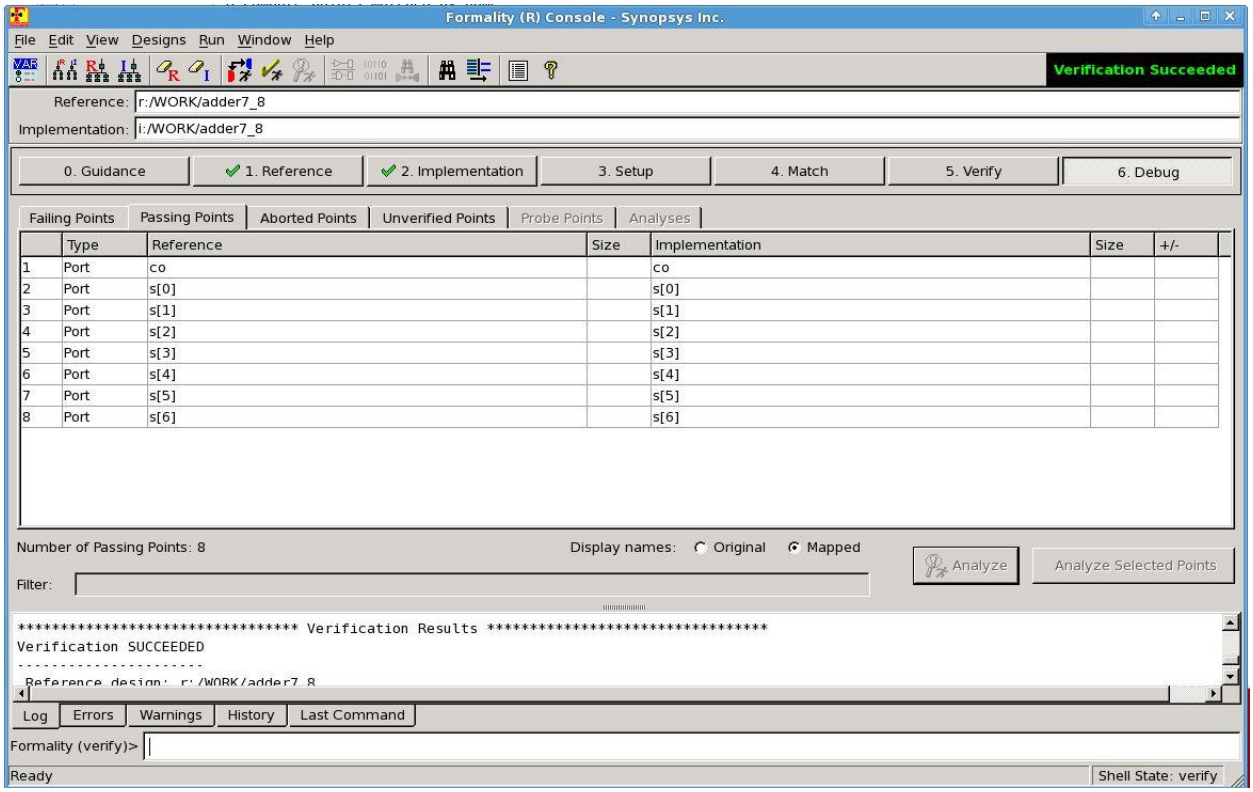

Figure 10: Carry Ripple Synopsys Formality Results

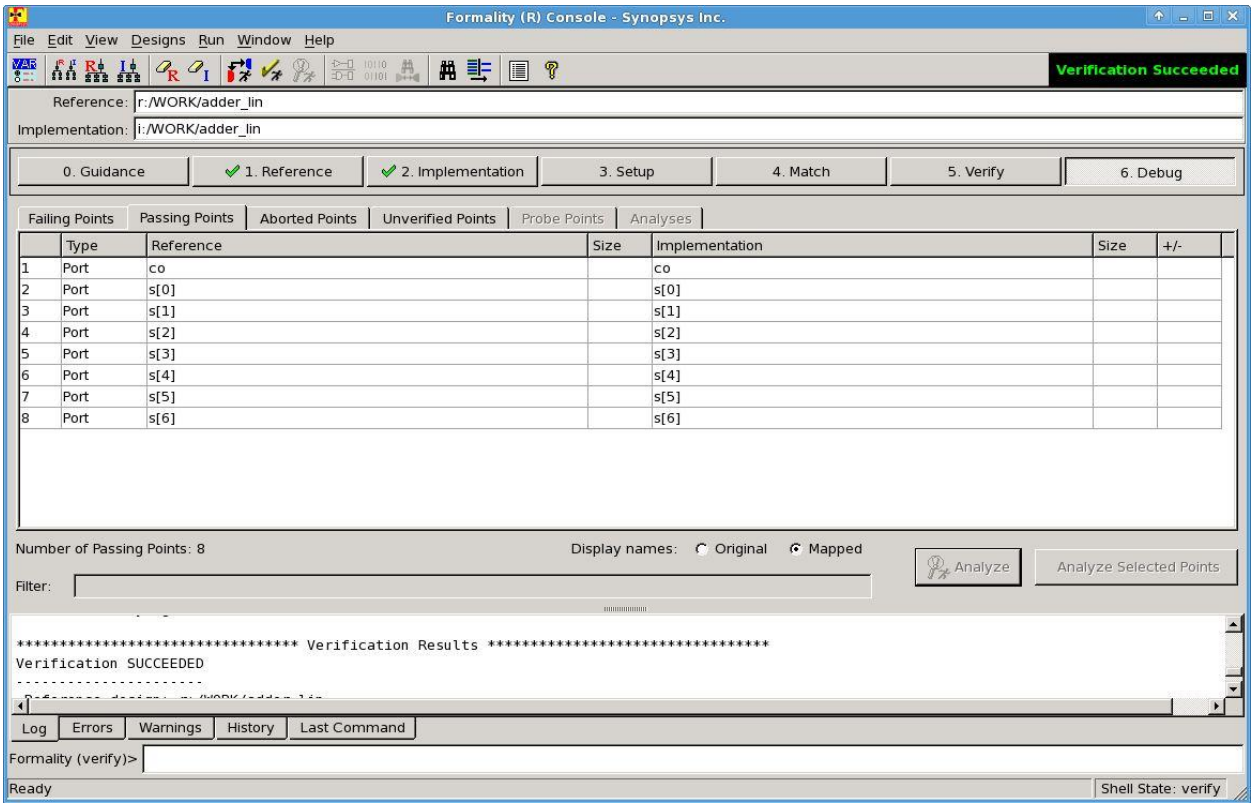

Figure 11: [3:2] Linear Adder Synopsys Formality Results

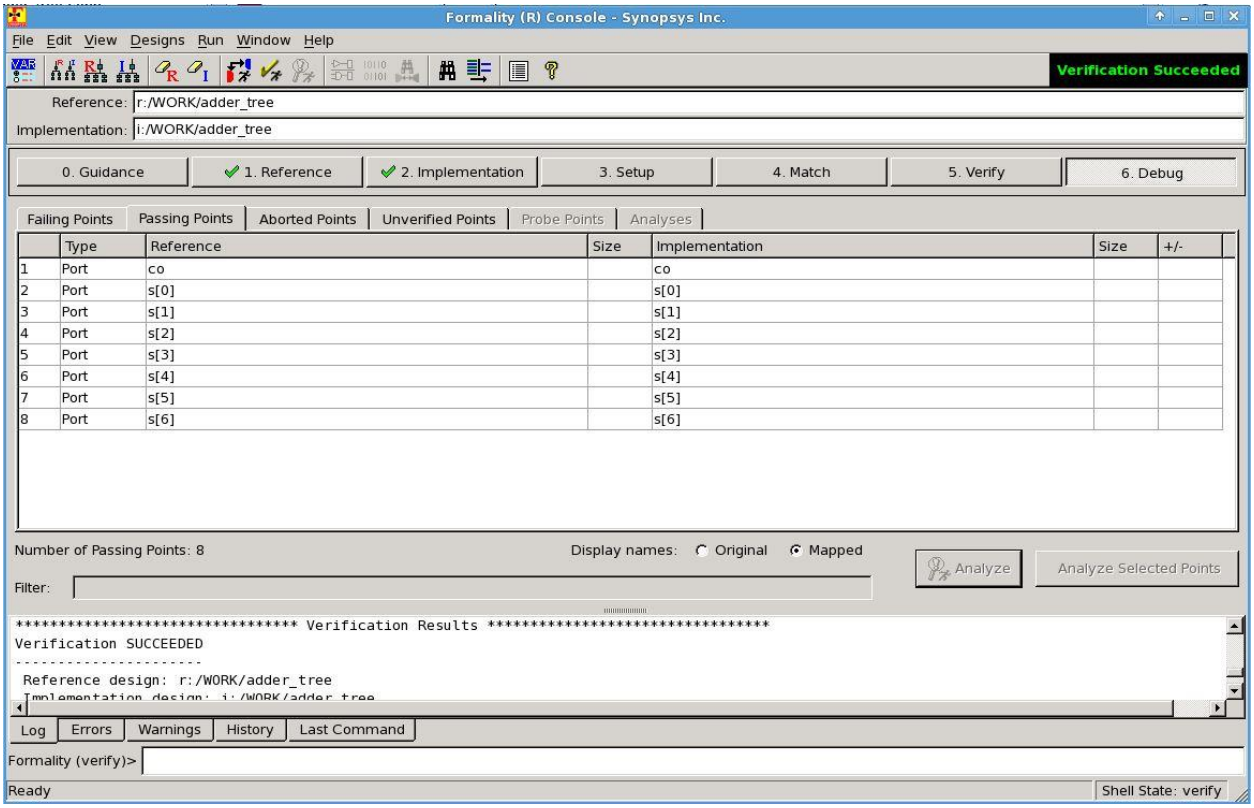

Figure 12: [4:2] Tree Adder Synopsys Formality Results

Synthesis Results

The carry ripple adder, 3:2 linear, and 4:2 tree adder were synthesized using the same design process. First, the preliminary verilog code was created using structure verilog code. Structural verilog code allowed us to easily follow the execution of the code and debug possible errors.

Next, the verilog code was synthesized into standard cells. This process was completed using Synopsys Design Compiler [3]. The compiler requires two pieces of information into order to properly function: the verilog code to be synthesized and specifications for the synthesis. The specification file from lab 9 was used with minor modifications. The list of verilog files was modified to include our code to be synthesized and the clock was changed to 250 MHz as specified in the project descriptions. The output of the compiler is a gate level netlist of the interconnected standard cells saved as a .vh file.

After creating the netlist of standard cells, we need to lay them out and connect them. Since the dimensions, timing, and power consumption for all standard cells are known, we can use that information to optimally place each cell and connect it to the other cells. This process is completed using Encounter [3]. As with the Design Compiler, Encounter requires two pieces of information: the gate level netlist to be laid out and a template file. The template file used was the same one used for lab 9. It contains information regarding the placement and routing of the cells. We modified it to include the netlist obtained from the Design Compiler. Encounter produces a GDSII file (final.gds2) which contains the optimized cell layout, an equivalent verilog model (final.v) of the circuit, and many files regarding timing and power. The GDSII file is used for the final layout design and the verilog model is used for verification. The timing and power files are used for comparison and analysis.

At this stage we had all the information necessary to analyze the three different adder designs. In order to accurately compare the different designs we looked at three different factors: timing, power, and size. All these factors are important to chip design because we wish to have the fastest logic possible for the lowest price and the lowest power consumption. Using the information output by Encounter after the placement and routing of the standard cells, we were able to determine the size of each design, the power consumption, and the propagation delay. The results are shown below in Table 4, as well as in the output files from Encounter in Appendix 3.

Table 4: Size, Power, and Timing Results

	Size (um)	Power Consumption (mW)	Propagation Delay (ns)
Carry Ripple Adder	804.8	0.8269	5.786
[3:2] Linear Adder	717.6	0.5668	1.845
[4:2] Tree Adder	686.6	0.3393	1.464

The carry ripple adder is by far the slowest design out of the three. This is because the carry-out is propagated from one bit to the next, thus sum of the current bit has to be completed before he sum of the next bit can start. The linear adder and tree adder, however, are much faster because the carry is saved instead of propagated. The tree adder is faster than the linear adder

because it has 2 fewer levels; the tree adder has 4 levels and the final addition, whereas the linear adder has 6 levels and the final addition.

The size of the designs is determined by the number of standard cells required to construct it. Again, the carry ripple adder is much larger than the other two. The propagation of the carry-out requires there to be more internal adders, thus increasing the size. The linear adder and the tree adder are much more comparable due to carry saving. The tree adder is slightly smaller due to the fewer levels required to implement it.

The power consumption is related to the number of transistors required for each design. As previously discussed, the carry ripple adder is much larger meaning it has more transistors. Since it has more transistors, the carry ripple adder will have a larger power requirement. However, the linear adder and tree adder have similar sizes but the tree adder consumes almost half the power. This is due to the activity factor of the transistors. The activity factor of the tree adder transistors is likely to be lower than those of the linear adder, resulting in a lower power requirement.

Comparing the three designs as a whole it is clear that the tree adder is a much better design because it is smaller, faster, and has lower power requirements than the others.

Finally, we need to complete the layout design. The GDSII file produced by Encounter contains the layout of the placement and routing of the standard cells, but it does not contain the individual cell layouts. To incorporate the individual layouts into the placement layout, we use Virtuoso [3]. Virtuoso accepts the GDSII file as an input stream and uses a standard cell library to import the layouts of the standard cells. The resulting layout design for the carry ripple adder, linear adder, and tree adder can be seen in figures 13, 14, and 15 respectively.

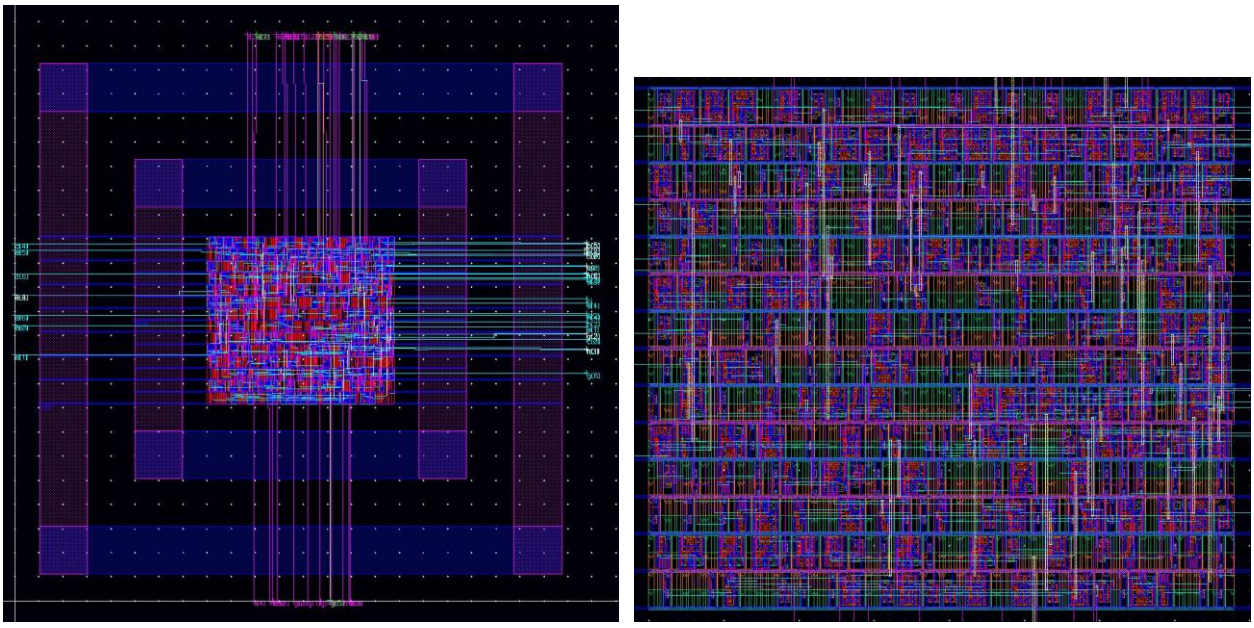

Figure 13: Carry Ripple Adder Layout

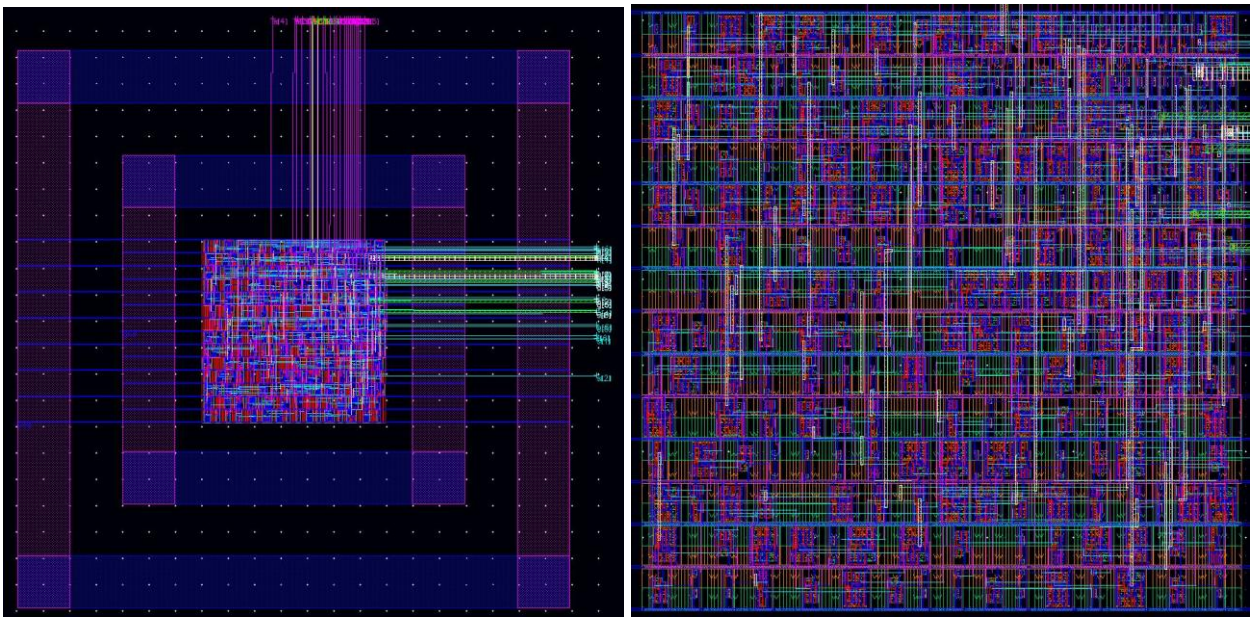

Figure 14: [3:2] Linear Adder Layout

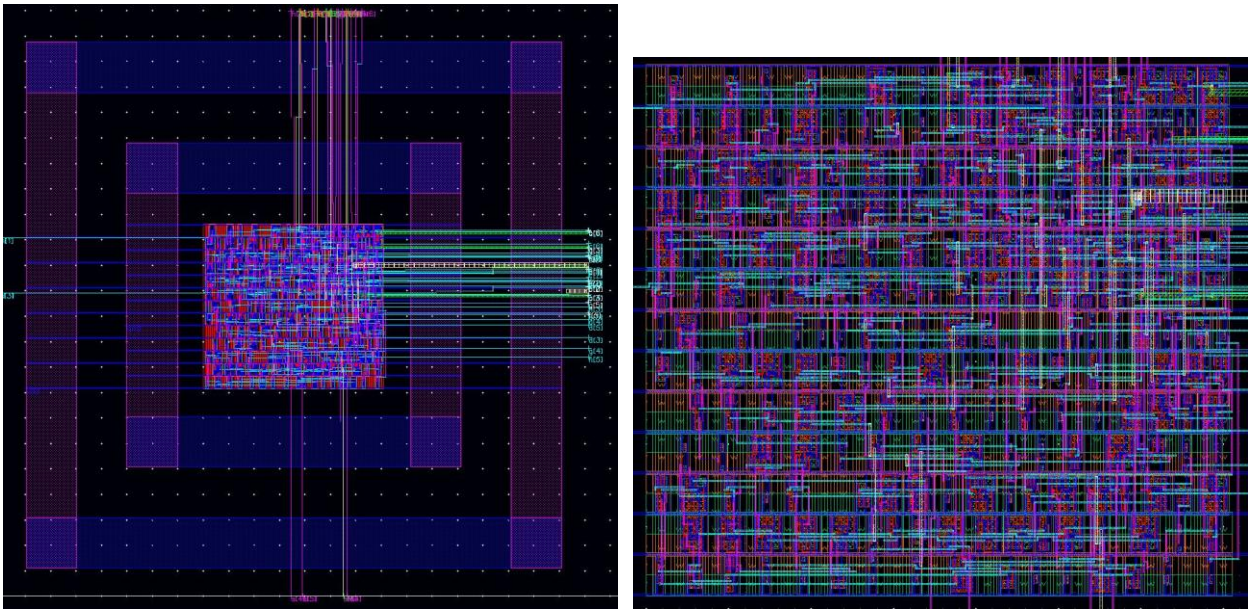

Figure 15: [4:2] Tree Adder Layout

Conclusion and Future Work

Three different 4-bit, 8 operand adders were designed as part of this project: a carry ripple adder, a [3:2] linear adder, and a [4:2] tree adder. The same design process was used for all three. First, the verilog code was written. Then, the code was synthesized into a netlist of standard cells using Synopsys Design Compiler. After that, the layout of the placement and routing of the standard cells was created using Encounter. Finally, the individual standard cell layouts were added to the layout of the placement of the cells using Virtuoso.

Throughout the design process, the results were validated using a verilog test bench. The test bench simulated several input combinations for the adders so they could be verified to be operating correctly for all input cases. The same test bench was used for all three designs and at each stage in the process so the results could be compared to ensure proper functionality. Additionally, once the placement and routing was completed, Synopsys Formality was used to formally verify that the interconnected cell layout was equivalent to the original verilog code.

Once the design was complete, it was found that the [4:2] tree adder was the most ideal out of the tree designs. It was the smallest, the fastest, and had the lowest power requirements of all three designs. This is due to the fact that the carry ripple adder uses carry propagation, greatly slowing down the logic operation, and the [3:2] linear adder had more levels, requiring more space and more power.

In addition to the three 4-bit, 8 operand adders, one 4-bit, 10 operand adder was also designed. This adder used two [5:2] carry save adders and one [4:2] carry save adder in a tree format. This adder was designed using verilog code and was tested using a new test bench that implemented two extra operands. The verilog code was tested to be fully functional, however it was designed as a proof of concept and the layout was not synthesized.

Future work that could be done on this project would be to fully synthesize the 4-bit, 10 operand adder. We could also modify the 4-bit, 8 operand adder to handle signed numbers as opposed to unsigned. This would effectively allow the adders to perform addition and subtraction.

References

1. CMOS VLSI Design: A Circuits and Systems Perspective. Neil Weste, David Harris. 2011.
2. ECE 429, Fall 2012 Final Project Guide. Dr. Erdal Oruklu. 11/20/2012
3. ECE 429- Tutorial IV: Standard Cell Based ASIC Design Flow. Dr. Jia Wang. August 2011.

Appendix 1: Verilog Code

Carry Ripple Adder

```
// Adder

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

   output s, co; //Outputs of Sum and Carry Out
   input  a, b, ci;   //Inputs of A, b and Carry In

   wire   o0, o1, o2; //Internal wiring

   xor(s, a, b, ci);  //Calculation of Sum

   or(o0, a, b); //Calculation of Carry Out
   or(o1, b, ci);
   or(o2, ci, a);
   and(co, o0, o1, o2);

endmodule // adder

//adder 7
module adder7(s, co, a, b, ci);

   output [6:0] s;    //7-bit Sum output
   output co;         //Output bit of Carry out

   input [6:0]  a, b; //7-bit Input A and B
   input ci;          //Input bit of Carry in

   wire    c1, c2, c3, c4, c5, c6;//wire [6:0] c

   adder a0(s[0], c1, a[0], b[0], ci);//(s[0], c[1]
   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], co, a[6], b[6], c6);

endmodule // adder8

//7-operand adder
module adder7_8(s, co, a, b, c, d, e, f, g, h, ci);

      output [6:0] s;
      output co;
      input  [6:0] a, b, c, d, e, f, g, h;
      input ci;
```

```
      wire [6:0] s1,s2,s3,s4,s5,s6;
      wire co1, co2, co3, co4, co5, co6;

      adder7 a0(s1, co1, a, b, ci);
      adder7 a1(s2, co2, c, s1, co1);
      adder7 a2(s3, co3, d, s2, co2);
      adder7 a3(s4, co4, e, s3, co3);
      adder7 a4(s5, co5, f, s4, co4);
      adder7 a5(s6, co6, g, s5, co5);
      adder7 a6(s, co, h, s6, co6);

endmodule //7-operand adder
```

[3:2] Linear Carry Save Adder

```
// 1-bit Adder
module adder(s, co, a, b, ci);

   output s, co; //Outputs of Sum and Carry Out
   input  a, b, ci;   //Inputs of A, b and Carry In

   wire   o0, o1, o2; //Internal wiring

   xor(s, a, b, ci);  //Calculation of Sum

   or(o0, a, b); //Calculation of Carry Out
   or(o1, b, ci);
   or(o2, ci, a);
   and(co, o0, o1, o2);

endmodule // adder

//7-bit adder used in the final calculation after the 3:2 adders have
done there thing
module adder7(s, co, a, b, ci);

   output [6:0] s;    //7-bit Sum output
   output co;         //Output bit of Carry out

   input [6:0]  a, b; //7-bit Input A and B
   input ci;               //Input bit of Carry in

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

   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], co, a[6], b[6], c6);
```

```
endmodule // adder8

//3:2 adder with all inputs and outputs 7-bit wide
module adder3_2(s, co, a, b, c);

   output [6:0] s,co; //7-bit Sum output and 7-bit carry in where co[0]
= 0 (Carry In will be added in later)
   input [6:0]  a, b, c;   //7-bit Input A and B

   wire    c1;        //the last carry bit is not used so assign to
bogus entiry

   assign co[0] = 0;  //assingment of co[0] to 0, this causes the Carry
vector to be shifted over to left by 1

   adder a0(s[0], co[1], a[0], b[0], c[0] );
   adder a1(s[1], co[2], a[1], b[1], c[1]);
   adder a2(s[2], co[3], a[2], b[2], c[2]);
   adder a3(s[3], co[4], a[3], b[3], c[3]);
   adder a4(s[4], co[5], a[4], b[4], c[4]);
   adder a5(s[5], co[6], a[5], b[5], c[5]);
   adder a6(s[6], c1, a[6], b[6], c[6]);



endmodule // adder8

//7-operand adder utilizing the 3:2 adders
module adder_lin(s, co, a, b, c, d, e, f, g, h, ci);

     output [6:0] s;
     output co;
     input  [6:0] a, b, c, d, e, f, g, h;
     input ci;

     wire [6:0] s1,s2,s3,s4,s5,s6;
     wire [6:0]co1, co2, co3, co4, co5, co6;

     //3:2 adders in linear fashion
     adder3_2 a0(s1, co1, a, b, c);
     adder3_2 a1(s2, co2, s1, co1, d);
     adder3_2 a2(s3, co3, s2, co2, e);
     adder3_2 a3(s4, co4, s3, co3, f);
     adder3_2 a4(s5, co5, s4, co4, g);
     adder3_2 a5(s6, co6, s5, co5, h);

     adder7 a6(s, co, s6, co6, ci);        //normal FA implementation,
adds in ci

endmodule //7-operand adder
```

[4:2] Tree Carry Save Adder

```
// 1-bit Adder
module adder(s, co, a, b, ci);

   output s, co; //Outputs of Sum and Carry Out
   input  a, b, ci;   //Inputs of A, b and Carry In

   wire   o0, o1, o2; //Internal wiring

   xor(s, a, b, ci);  //Calculation of Sum

   or(o0, a, b); //Calculation of Carry Out
   or(o1, b, ci);
   or(o2, ci, a);
   and(co, o0, o1, o2);

endmodule // adder

//7-bit adder used in the final calculation after the 3:2 adders have
done there thing
module adder7(s, co, a, b, ci);

   output [6:0] s;    //7-bit Sum output
   output co;         //Output bit of Carry out

   input [6:0]  a, b; //7-bit Input A and B
   input ci;               //Input bit of Carry in

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

   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], co, a[6], b[6], c6);

endmodule // adder8

//4:2 adder with all inputs and outputs 7-bit wide
module adder4_2(s, co, a, b, c, d);

   output [6:0] s,co; //7-bit Sum output and 7-bit carry in where co[0]
= 0 (Carry In will be added in later)
   input [6:0]  a, b, c, d ;     //7-bit Input A, B, C, D

   wire [6:0] ts;
   wire [6:0] tc;
   wire c1,c2;//crap 1 and 2
```

```
  assign co[0] = 0;
  assign tc[0] = 0;

   adder s00(ts[0],tc[1],a[0],b[0],c[0]);
   adder s01(ts[1],tc[2],a[1],b[1],c[1]);
   adder s02(ts[2],tc[3],a[2],b[2],c[2]);
   adder s03(ts[3],tc[4],a[3],b[3],c[3]);
   adder s04(ts[4],tc[5],a[4],b[4],c[4]);
   adder s05(ts[5],tc[6],a[5],b[5],c[5]);
   adder s06(ts[6],c1,   a[6],b[6],c[6]);

   adder s10(s[0],co[1],ts[0],d[0],tc[0]);
   adder s11(s[1],co[2],ts[1],d[1],tc[1]);
   adder s12(s[2],co[3],ts[2],d[2],tc[2]);
   adder s13(s[3],co[4],ts[3],d[3],tc[3]);
   adder s14(s[4],co[5],ts[4],d[4],tc[4]);
   adder s15(s[5],co[6],ts[5],d[5],tc[5]);
   adder s16(s[6],c2,  ts[6],d[6],tc[6]);

endmodule // adder8

//7-operand adder utilizing the 3:2 adders
module adder_tree(s, co, a, b, c, d, e, f, g, h, ci);

     output [6:0] s;
     output co;
     input  [6:0] a, b, c, d, e, f, g, h;
     input ci;

     wire [6:0] ts1, ts2, ts3;
     wire [6:0] tc1, tc2, tc3;

     //3:2 adders in linear fashion
     adder4_2 a1(ts1, tc1, a, b, c, d);
     adder4_2 a2(ts2, tc2, e, f, g, h);
     adder4_2 a3(ts3, tc3, ts1, tc1, ts2, tc2);

     adder7 a4(s, co, ts3, tc3, ci);       //normal FA implementation,
adds in ci

endmodule //7-operand adder
```

Appendix 2: Test Bench Results

Carry Ripple Adder

```
Host command: /apps/cadence/IES82/IUS82/tools/verilog/bin/verilog.exe
Command arguments:
    gscl45nm.v
    adder7_8test.v
    adder7_8.v

Tool: VERILOG-XL 08.20.001-p log file created Nov 17, 2012  14:21:43
Tool: VERILOG-XL 08.20.001-p   Nov 17, 2012  14:21:43

Copyright (c) 1995-2004 Cadence Design Systems, Inc.  All Rights
Reserved.
Unpublished -- rights reserved under the copyright laws of the United
States.

Copyright (c) 1995-2004 UNIX Systems Laboratories, Inc.  Reproduced
with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL
INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC.  USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN
PERMISSION OF
CADENCE DESIGN SYSTEMS, INC.
RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(1) and (2) of Commercial Computer Software --
Restricted
Rights at 48 CFR 52.227-19, as applicable.

               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 "gscl45nm.v"
Compiling source file "adder7_8test.v"
Compiling source file "adder7_8.v"
Highest level modules:
```

```
stimulus

At Time:                   100  Sum=  8 Carry=0
At Time:                   200  Sum= 36 Carry=0
At Time:                   300  Sum=120 Carry=0
At Time:                   400  Sum= 71 Carry=0
At Time:                   500  Sum= 37 Carry=0
At Time:                   600  Sum=122 Carry=0
At Time:                   700  Sum=  0 Carry=1
L36 "adder7_8test.v": $finish at simulation time 400000
0 simulation events (use +profile or +listcounts option to count) +
2029 accelerated events + 70 timing check 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   Nov 17, 2012  14:21:45
```

[3:2] Linear Carry Save Adder

```
Host command: /apps/cadence/IES82/IUS82/tools/verilog/bin/verilog.exe
Command arguments:
    adder_lintest.v
    adder_lin.v

Tool: VERILOG-XL 08.20.001-p log file created Nov 17, 2012  14:43:51
Tool: VERILOG-XL 08.20.001-p   Nov 17, 2012  14:43:51

Copyright (c) 1995-2004 Cadence Design Systems, Inc.  All Rights
Reserved.
Unpublished -- rights reserved under the copyright laws of the United
States.

Copyright (c) 1995-2004 UNIX Systems Laboratories, Inc.  Reproduced
with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL
INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC.  USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN
PERMISSION OF
CADENCE DESIGN SYSTEMS, INC.
RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(1) and (2) of Commercial Computer Software --
Restricted
Rights at 48 CFR 52.227-19, as applicable.

                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 "adder_lintest.v"
Compiling source file "adder_lin.v"
Highest level modules:
stimulus

At Time:                  100  Sum=  8 Carry=0
At Time:                  200  Sum= 36 Carry=0
At Time:                  300  Sum=120 Carry=0
At Time:                  400  Sum= 71 Carry=0
At Time:                  500  Sum= 37 Carry=0
At Time:                  600  Sum=122 Carry=0
At Time:                  700  Sum=  0 Carry=1
L36 "adder_lintest.v": $finish at simulation time 400000
0 simulation events (use +profile or +listcounts option to count) +
1564 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   Nov 17, 2012  14:43:52
```

[4:2] Tree Carry Save Adder

```
Host command: /apps/cadence/IES82/IUS82/tools/verilog/bin/verilog.exe
Command arguments:
    adder_treetest.v
    adder_tree.v

Tool: VERILOG-XL 08.20.001-p log file created Nov 17, 2012  13:22:39
Tool: VERILOG-XL 08.20.001-p   Nov 17, 2012  13:22:39

Copyright (c) 1995-2004 Cadence Design Systems, Inc.  All Rights
Reserved.
Unpublished -- rights reserved under the copyright laws of the United
States.

Copyright (c) 1995-2004 UNIX Systems Laboratories, Inc.  Reproduced
with Permission.

THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL
INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC.  USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN
PERMISSION OF
```

CADENCE DESIGN SYSTEMS, INC.
RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software -- Restricted
Rights at 48 CFR 52.227-19, as applicable.

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 "adder_treetest.v"
Compiling source file "adder_tree.v"
Highest level modules:
stimulus

At Time: 100 Sum= 8 Carry=0
At Time: 200 Sum= 36 Carry=0
At Time: 300 Sum=120 Carry=0
At Time: 400 Sum= 71 Carry=0
At Time: 500 Sum= 37 Carry=0
At Time: 600 Sum=122 Carry=0
At Time: 700 Sum= 0 Carry=1
L36 "adder_treetest.v": $finish at simulation time 400000
0 simulation events (use +profile or +listcounts option to count) +
1181 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   Nov 17, 2012 13:22:39

Appendix 3: Encounter Results

Carry Ripple Adder

```
<CMD> reportGateCount -limit 0
Gate area 2.8158 um^2
[0] adder7_8 Gates=285 Cells=294 Area=804.8 um^2


Finished Calculating power
2012-Nov-17 14:18:55 (2012-Nov-17 20:18:55 GMT)
*------------------------------------------------------------------------
-------------------
*     Encounter 10.13-s292_1 (32bit) 08/15/2012 15:12 (Linux 2.6)
*
*
*     Date & Time:     2012-Nov-17 14:18:55 (2012-Nov-17 20:18:55 GMT)
*
*------------------------------------------------------------------------
-------------------
*
*     Design: adder7_8
*
*     Liberty Libraries used:
*             /apps/FreePDK45/osu_soc/lib/files/gscl45nm.tlf
*
*     Power Domain used:
*          Rail:        vdd      Voltage:        1.1
*
*     Power Units = 1mW
*
*     Time Units = 1e-09 secs
*
*       report_power
*
------------------------------------------------------------------------
-------------------


Total Power
------------------------------------------------------------------------
-------------------
Total Internal Power:      0.4233       51.2%
Total Switching Power:      0.3986      48.21%
Total Leakage Power:    0.004872     0.5893%
Total Power:      0.8269
------------------------------------------------------------------------
-------------------


############################################################
```

```
#  Generated by:      Cadence Encounter 10.13-s292_1
#  OS:                Linux x86_64(Host ID saturn.ece.iit.edu)
#  Generated on:      Sat Nov 17 14:17:41 2012
#  Design:            adder7_8
#  Command:           report_timing -nworst  5 -net >
timing.rep.5.final
#####################################################################
Path 1: VIOLATED Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: b[0] (v) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  5.786
= Slack Time                   -1.886
     Clock Rise Edge                         0.000
     + Input Delay                           0.100
     + Drive Adjustment                      0.038
     = Beginpoint Arrival Time               0.138

Path 2: VIOLATED Late External Delay Assertion
Endpoint:   s[6] (v) checked with  leading edge of 'vclk'
Beginpoint: b[0] (v) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  5.773
= Slack Time                   -1.873
     Clock Rise Edge                         0.000
     + Input Delay                           0.100
     + Drive Adjustment                      0.038
     = Beginpoint Arrival Time               0.138

Path 3: VIOLATED Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: a[0] (v) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  5.770
= Slack Time                   -1.870
     Clock Rise Edge                         0.000
     + Input Delay                           0.100
     + Drive Adjustment                      0.033
     = Beginpoint Arrival Time               0.133

Path 4: VIOLATED Late External Delay Assertion
Endpoint:   s[6] (v) checked with  leading edge of 'vclk'
Beginpoint: a[0] (v) triggered by  leading edge of 'vclk'
```

```
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  5.757
= Slack Time                   -1.857
     Clock Rise Edge                          0.000
     + Input Delay                            0.100
     + Drive Adjustment                       0.033
     = Beginpoint Arrival Time                0.133

Path 5: VIOLATED Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: b[0] (^) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  5.743
= Slack Time                   -1.843
     Clock Rise Edge                          0.000
     + Input Delay                            0.100
     + Drive Adjustment                       0.060
     = Beginpoint Arrival Time                0.160
```

[3:2] Linear Carry Save Adder

```
<CMD> reportGateCount -limit 0
Gate area 2.8158 um^2
[0] adder_lin Gates=254 Cells=258 Area=717.6 um^2

Finished Calculating power
2012-Nov-17 15:07:31 (2012-Nov-17 21:07:31 GMT)
*------------------------------------------------------------------------
-------------------
*     Encounter 10.13-s292_1 (32bit) 08/15/2012 15:12 (Linux 2.6)
*
*
*     Date & Time:     2012-Nov-17 15:07:31 (2012-Nov-17 21:07:31 GMT)
*
*------------------------------------------------------------------------
-------------------
*
*     Design: adder_lin
*
*     Liberty Libraries used:
*             /apps/FreePDK45/osu_soc/lib/files/gscl45nm.tlf
*
*     Power Domain used:
*          Rail:        vdd      Voltage:        1.1
*
*     Power Units = 1mW
```

```
*
*     Time Units = 1e-09 secs
*
*       report_power
*
--------------------------------------------------------------------------------
-------------------

Total Power
--------------------------------------------------------------------------------
-------------------
Total Internal Power:         0.3      52.93%
Total Switching Power:      0.2624       46.3%
Total Leakage Power:     0.00438     0.7727%
Total Power:      0.5668
--------------------------------------------------------------------------------
-------------------

#############################################################
#  Generated by:      Cadence Encounter 10.13-s292_1
#  OS:                Linux x86_64(Host ID saturn.ece.iit.edu)
#  Generated on:      Sat Nov 17 15:06:38 2012
#  Design:            adder_lin
#  Command:           report_timing -nworst  10 -net >
timing.rep.5.final
#############################################################
Path 1: MET Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: a[0] (v) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.845
= Slack Time                    2.056
     Clock Rise Edge                         0.000
     + Input Delay                           0.100
     + Drive Adjustment                      0.047
     = Beginpoint Arrival Time               0.147

Path 2: MET Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: b[0] (v) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.834
= Slack Time                    2.066
     Clock Rise Edge                         0.000
     + Input Delay                           0.100
     + Drive Adjustment                      0.031
```

```
     = Beginpoint Arrival Time            0.131

Path 3: MET Late External Delay Assertion
Endpoint:   s[6] (v) checked with  leading edge of 'vclk'
Beginpoint: a[0] (v) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.830
= Slack Time                    2.070
     Clock Rise Edge                         0.000
     + Input Delay                           0.100
     + Drive Adjustment                      0.047
     = Beginpoint Arrival Time               0.147

Path 4: MET Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: a[0] (v) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.824
= Slack Time                    2.076
     Clock Rise Edge                         0.000
     + Input Delay                           0.100
     + Drive Adjustment                      0.047
     = Beginpoint Arrival Time               0.147

Path 5: MET Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: a[0] (v) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.823
= Slack Time                    2.077
     Clock Rise Edge                         0.000
     + Input Delay                           0.100
     + Drive Adjustment                      0.047
     = Beginpoint Arrival Time               0.147
```

[4:2] Tree Carry Save Adder

```
<CMD> reportGateCount -limit 0
Gate area 2.8158 um^2
[0] adder_tree Gates=243 Cells=239 Area=686.6 um^2

Finished Calculating power
```

```
2012-Nov-17 15:28:39 (2012-Nov-17 21:28:39 GMT)
*------------------------------------------------------------------------
-------------------
*     Encounter 10.13-s292_1 (32bit) 08/15/2012 15:12 (Linux 2.6)
*
*
*     Date & Time:     2012-Nov-17 15:28:39 (2012-Nov-17 21:28:39 GMT)
*
*------------------------------------------------------------------------
-------------------
*
*     Design: adder_tree
*
*     Liberty Libraries used:
*             /apps/FreePDK45/osu_soc/lib/files/gscl45nm.tlf
*
*     Power Domain used:
*          Rail:        vdd      Voltage:        1.1
*
*     Power Units = 1mW
*
*     Time Units = 1e-09 secs
*
*       report_power
*
------------------------------------------------------------------------
-------------------

Total Power
------------------------------------------------------------------------
-------------------
Total Internal Power:      0.1769      52.14%
Total Switching Power:      0.1581      46.58%
Total Leakage Power:    0.004355      1.284%
Total Power:      0.3393
------------------------------------------------------------------------
-------------------

#############################################################
#  Generated by:      Cadence Encounter 10.13-s292_1
#  OS:                Linux x86_64(Host ID saturn.ece.iit.edu)
#  Generated on:      Sat Nov 17 15:27:27 2012
#  Design:            adder_tree
#  Command:           report_timing -nworst  10 -net >
timing.rep.5.final
#############################################################
Path 1: MET Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: a[1] (^) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
```

```
= Required Time                 3.900
- Arrival Time                  1.464
= Slack Time                    2.436
     Clock Rise Edge                          0.000
     + Input Delay                            0.100
     + Drive Adjustment                       0.077
     = Beginpoint Arrival Time                0.177

Path 2: MET Late External Delay Assertion
Endpoint:   s[6] (v) checked with  leading edge of 'vclk'
Beginpoint: a[1] (^) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.452
= Slack Time                    2.448
     Clock Rise Edge                          0.000
     + Input Delay                            0.100
     + Drive Adjustment                       0.077
     = Beginpoint Arrival Time                0.177

Path 3: MET Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: b[1] (^) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.449
= Slack Time                    2.451
     Clock Rise Edge                          0.000
     + Input Delay                            0.100
     + Drive Adjustment                       0.055
     = Beginpoint Arrival Time                0.155

Path 4: MET Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
Beginpoint: a[1] (^) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.444
= Slack Time                    2.456
     Clock Rise Edge                          0.000
     + Input Delay                            0.100
     + Drive Adjustment                       0.077
     = Beginpoint Arrival Time                0.177

Path 5: MET Late External Delay Assertion
Endpoint:   s[6] (^) checked with  leading edge of 'vclk'
```

```
Beginpoint: a[1] (^) triggered by  leading edge of 'vclk'
Other End Arrival Time          0.000
- External Delay                0.100
+ Phase Shift                   4.000
= Required Time                 3.900
- Arrival Time                  1.442
= Slack Time                    2.458
     Clock Rise Edge                          0.000
     + Input Delay                            0.100
     + Drive Adjustment                       0.077
     = Beginpoint Arrival Time                0.177
```