

Fast Multi-operand Addition using 7:3 Compressors

Speed and Cost Comparison

Jason A. Pamplin

Table of Contents

Abstract	1
Terminology	2
Compressor Notation	2
Input:Output	2
Delay	2
7:3/5- Compressor Design	3
Approach	3
4 - Bit Total Count (TC) Decoder	3
TC ₀ Computation	3
TC ₁ Computation	4
TC ₂ Computation	4
TC ₃ Computation	5
TC ₄ Computation	5
Combined TC	6
7 - Bit Computations	7
Sum	7
First Carry Bit (C ₁)	7
Second Carry Bit (C ₂)	7
7:3/6- Compressor	8
7/3:5- Compressor	9
Carry-Save Adder (CSA) Implementation	10
Wallace Tree	10
Pamplin Tree	10
Pamplin Tree Analysis	11
Performance	11
Wallace Tree	11
Pamplin Tree	11
Delay Comparison	12
Cost	12
Wallace Tree	12
Pamplin Tree	13

Total Gate Comparison	13
Conclusion	14
Future Work	14
Appendix A – Graph of Delay vs. Number of Inputs	15
Appendix B – 27-Input Tree Implementations	16
27-Input Wallace Tree	16
27-Input Pamplin Tree	17

Figures

Figure 1. Standard Full Adder	2
Figure 2. 4:3/3-Compressor	3
Figure 3. TC_0 Computation	3
Figure 4. TC_1 Computation	4
Figure 5. TC_2 Computation	4
Figure 6. TC_3 Computation	5
Figure 7. TC_4 Computation	5
Figure 8. TC Decoder	6
Figure 9. 7:3/6-Compressor	8
Figure 10. Full Adder with only 2 Delays	8
Figure 11. 7:3/5-Compressor	9
Figure 12. 7-Input Wallace Tree using CSA-3:2/3-compressors	10
Figure 13. 7-Input Pamplin Tree using CSA-7:3/5-compressors	10

Tables

Table 1. Truth Table for 7-Bit Addition	7
Table 2. Performance Comparison of CSA implementations	11
Table 3. Number of CSA-3:2/3's for arbitrary number of inputs	12
Table 4. Number of CSA-7:3/5's for arbitrary number of inputs	13

Abstract

As computers continue to be used for more and more mathematically intensive tasks, it is necessary to continue to improve the performance of adding a large number of binary numbers. In addition, modern VLSI techniques make the cost of circuit implementation increasingly less expensive even for more complex circuits. This paper demonstrates the practical implementation of a new compressor that adds 7 individual bits and outputs a sum bit and two carry bits. This is referred to as the 7:3-compressor. This paper then shows how this new compressor can be configured into a Wallace Tree structure. A mathematical proof is given that demonstrates that configuration of such an adder, while twice as costly, does give a greater than 20% improvement in the average gate delay time required to add an arbitrary number of inputs.

Terminology

Compressor Notation

Input:Output

For the purposes of this paper, a $x:y$ -compressor is an adding device that adds x number of inputs including any input carries and outputs a single sum bit and $y-1$ carry bits. Thus, the standard full adder as shown in figure 1 is, by our definition, a 3:2-compressor.

This terminology was chosen because it accurately reflects the actual number of inputs and outputs rather than ignoring the carry bits. This is particularly true in a carry-save adder (CSA) implementation which does not really consider the input carry as a carry bit from the previous stage. Instead, the carry input that would normally be used during standard carry propagation is used for a third operand. In addition, the carry-out bit is saved as a complete number and added to the sum in the next stage.

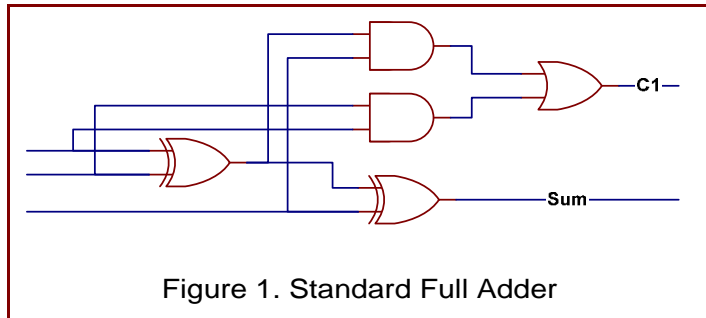


Figure 1. Standard Full Adder

Some papers refer to a 4:2-compressor as one that takes 4 inputs and a carry bit and produces 2 sum bits and a carry out. By our definition, however, this would be a 5:3-compressor.

Delay

Since this paper is primarily concerned with gate delay in the analysis, there is an additional notational element added to the representation of compressors. Thus, a $x:y/z$ -compressor is a compressor that accepts x inputs and outputs y values using z total gate delays. Using this notation a standard full adder as show in figure 1 is a 3:2/3-compressor.

7:3/5-Compressor Design

There are various papers outlining 4:3, 5:3, and even 6:3 compressors. But these fall short in that the output of all of these compressors is necessarily 3 bits. This is required because if all input bits happen to be 1's then multiple carries are always required even with as little as 4 input bits. Thus, it makes sense to reach for a 7:3 compressor as even with all 1's on the input no more than 3 outputs are required. The complexity of such a circuit obviously increases quickly as the number of inputs increases.

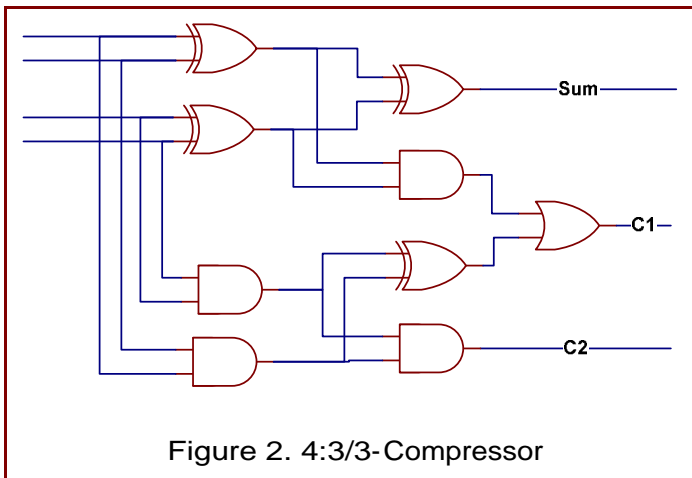


Figure 2. 4:3/3-Compressor

Approach

The basic approach taken was to split the 7 inputs into 4 and 3. Thus, we start with a 4:3/3-compressor as shown in figure 2 and a standard 3:2/3-compressor as shown in figure 1.

At this point, it is necessary to determine, using the 4:3/3-Compressor, how many 1's are in the input. If we know this answer, then we can modify the outputs of the 3:2/3-Compressor to be the appropriate values for our 7-bit addition. We use the term TC_x to represent boolean value that there are x number of 1's in the first 4 inputs of the data to be added. Thus, if TC_1 is true then there is only a single 1 in the first 4 bits of the input.

4-Bit Total Count (TC) Decoder

TC₀ Computation

Since I am attempting to minimize the number of gate delays in producing this value, I felt that the easiest way to compute this value is to add 2 OR gates and a NOR gate such that if all inputs are 0 then the output of the final NOR gate would be 1. However, during the computation of TC₂ it became necessary to know if both of 2 sets

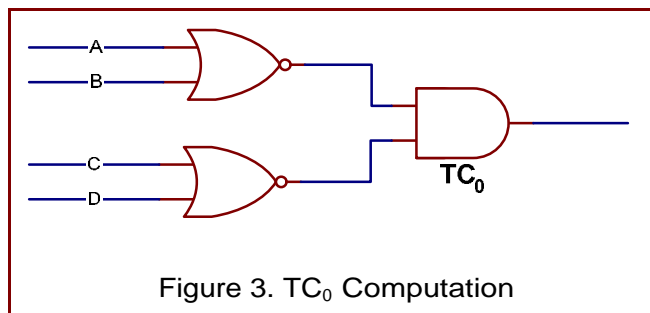


Figure 3. TC₀ Computation

of inputs were false by signifying a true. Thus, I replaced the 2 OR gates with 2 NOR gates and used an AND gate. These, are logically equivalent as:

$$TC_0 = \overline{(A + B) + (C + D)} = \overline{(A + B)} \bullet \overline{(C + D)}$$

The final circuit for calculating TC₀ is shown in figure 3.

TC₁ Computation

This computation takes advantage of the existing logic gates in the 4-bit adder of figure 2. The logic for this is based on the sum value. If the output sum of this circuit is true then we have either TC₃ or TC₁ equal to true. But, if TC₃ is true, then one of the 2 AND gates connected to the 2 input pairs must also be true. If they are both false, then TC₃ must be false and TC₁ is true. The equation for computing TC₁ in this way is:

$$TC_1 = ((A \oplus B) \oplus (C \oplus D)) \cdot \overline{(AB + CD)}$$

Figure 4 shows the circuit for calculating TC₁.

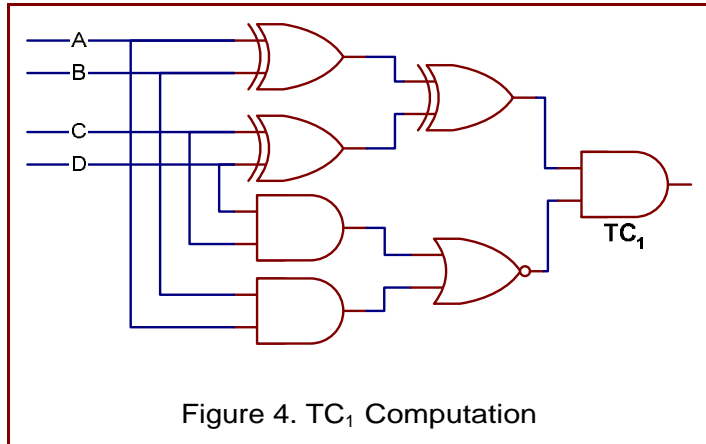


Figure 4. TC₁ Computation

TC₂ Computation

This is by far the hardest computation of the 5 required. The approach taken here is basically a brute force approach. TC₂ is true if either of 3 cases exists:

- A and B are both true but C and D are both false.
- C and D are both true but A and B are both false.
- Only one of A or B is true and only one of C or D is true

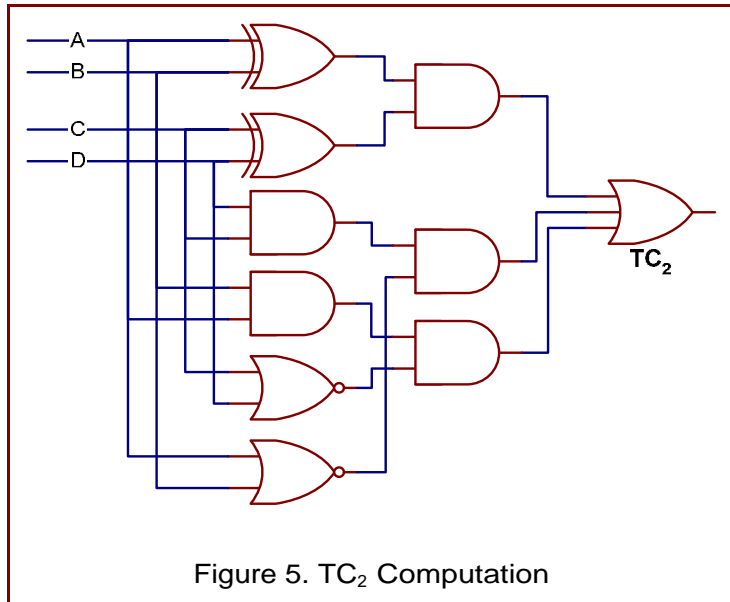


Figure 5. TC₂ Computation

The equation for computing TC₂ in this way is:

$$TC_2 = (AB\overline{(C + D)}) + (CD\overline{(A + B)}) + ((A \oplus B)(C \oplus D))$$

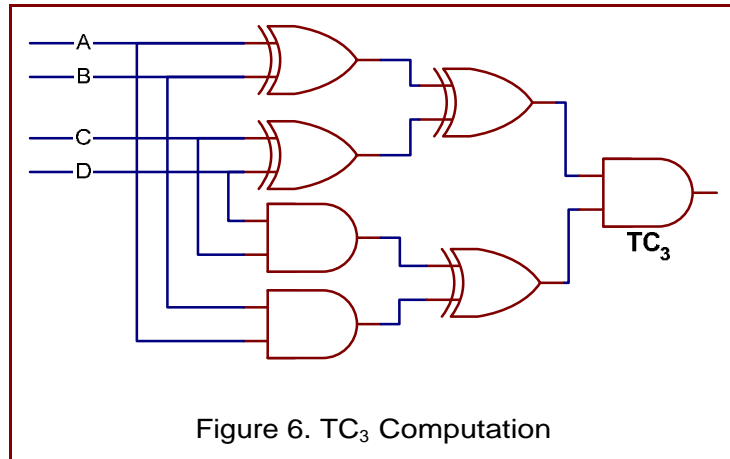
Figure 5 shows the circuit for computing TC₂. The choice to use NOR gates in the computation of TC₀ is apparent since we can use these same gates for this computation as well. This allows us to only add 2 AND gates and an OR gate to the current 4-bit adder after modification for TC₀ and TC₁.

TC₃ Computation

If the sum is true and either pair of inputs is also both true then TC₃ is true. This computation is thus similar to TC₁ and like that computation we take advantage of the sum value which is already computed in the 4-bit adder. Thus, only a single XOR gate and a single AND gate need to be added to achieve the desired result. The equation for computing TC₃ is:

$$TC_3 = ((A \oplus B) \oplus (C \oplus D)) \cdot (AB \oplus CD)$$

Figure 6 shows the circuit for computing TC₃.

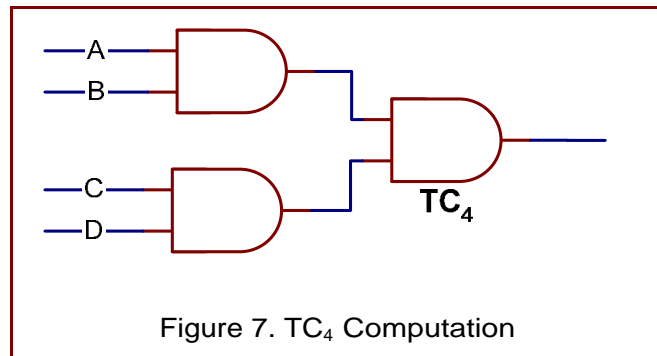


TC₄ Computation

Computation of TC₄ is simple in that both pairs of inputs must all be true. This requires no additional logic gates of the 4-bit adder of figure 2, since the C₂ value basically only occurs if all inputs are true. Thus, the equation for TC₄ is:

$$TC_4 = ABCD$$

Figure 7 shows the TC₄ Computation.



Combined TC

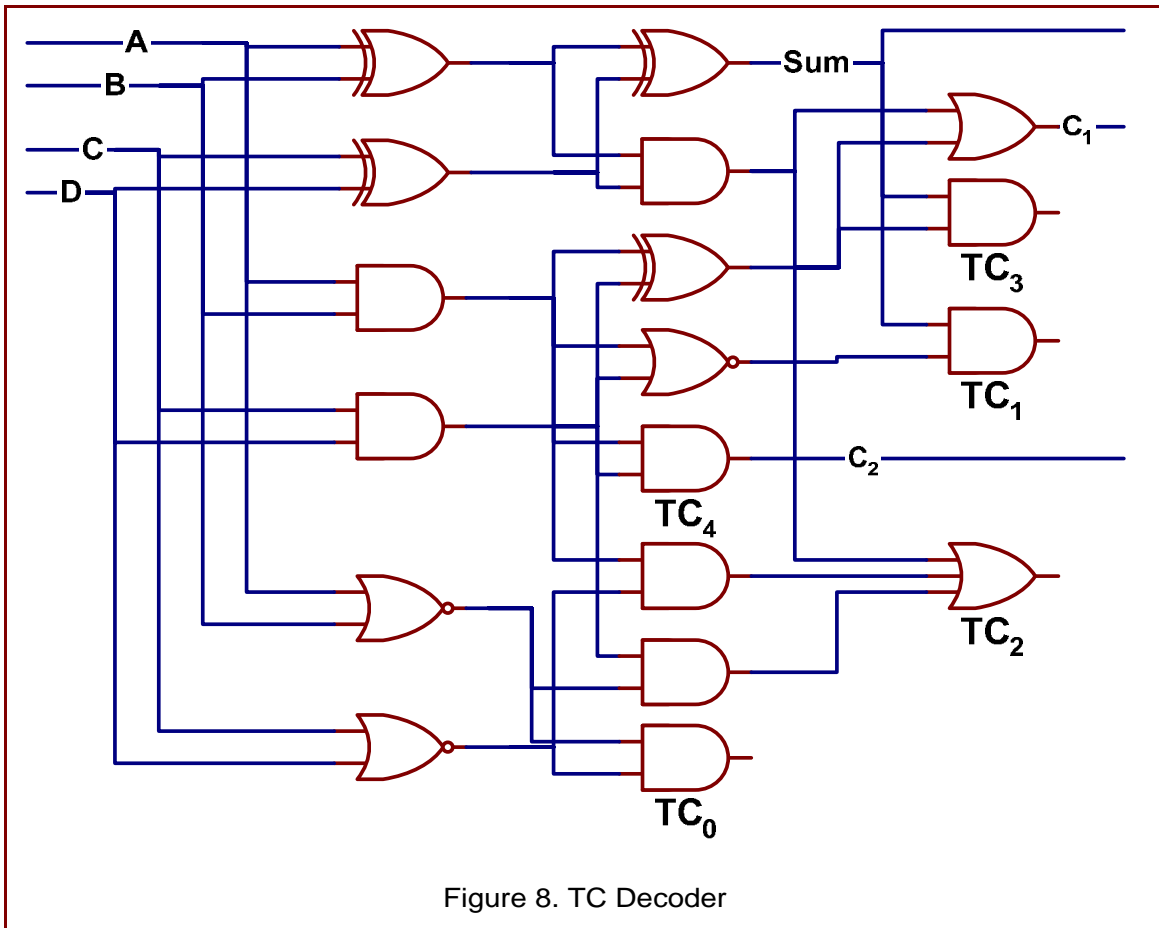


Figure 8. TC Decoder

By cleverly combining the individual TC_x computational elements, a 4-bit TC Decoder can be built such that a single output will be true based on the number of 1's in the 4-bit input. This is accomplished without adding any gate delays to the original 4-bit adder of figure 2 while doubling the cost in terms of logic gates. The TC Decoder still has all of the outputs of the original 4-bit adder allowing us to use them for the 7-bit computations.

7-Bit Computations

The TC Decoder allows us to base our final 7-bit output on the number of 1's in the first 4 bits.

Sum

The sum of the 7-bit adder is simply the sum of the 4-bit TC Decoder and the sum of the full adder (FA) of figure 1. The sum operation is a simple XOR operation thus the equation of the final sum is:

$$SUM = TC_{SUM} \oplus FA_{SUM}$$

First Carry Bit (C₁)

The TC Decoder allows us to generate the proper carry bit based on the number of input 1's in the first 4 bits, since the specific final result of the first 4 bits completely determines the final carry value based on the full adder.

Table 1 shows the truth table value of the carry bit for each of the 5 different values of the TC Decoder as well as the output of the full adder. From this table it is relatively easy to see the logical operations required to calculate the first carry out of the 7-bit adder. This allows us to derive a Boolean expression for computing C₁ as:

$$C_1 = (TC_0 + TC_4)FA_c + TC_1(FA_{SUM} \oplus FA_C) + TC_2 \overline{FA_c} + TC_3 \overline{(FA_{SUM} \oplus FA_C)}$$

TC₀ and TC₄ have the same truth values for C₁ thus they are combined in this equation. Each product in this equation has at most 3 terms. Thus, we can compute the first carry while only adding at most 3 gate delays, for a total of 6.

Second Carry Bit (C₂)

Table 1 also shows us the proper values for the second carry out bit. This carry computation appears at first to be more complicated than the first in that no pattern is readily apparent in the truth table. However, with some cleverness we can compute C₂ while still only adding 3 gate delays.

We start by looking at the outside values. If TC₀ is true then we can basically ignore the computation for C₂ as no matter what the full adder produces C₂ will always be zero. In addition, if TC₄ is true then C₂ will always be true independent of what the full adder produces.

The next step is to compare the output carries of both adders. If the full adder produces an output carry and the TC decoder produces a first-order output carry, then both adders have at least 2 1's each. Regardless of exactly how many 1's they have, there are at least 4 1's total in the 7-bit input. Thus, C₂ must be true.

The only combinations unaccounted for are the 3 to 1 combinations. In other words, if TC₃ is true and the FASUM is true then there are 4 ones total and thus C₂ must be true. Likewise, if TC₁ is true and both the FA_{SUM} and FA_C are true then we still have 4 ones and C₂ must be true.

Thus the equation for C₂ is:

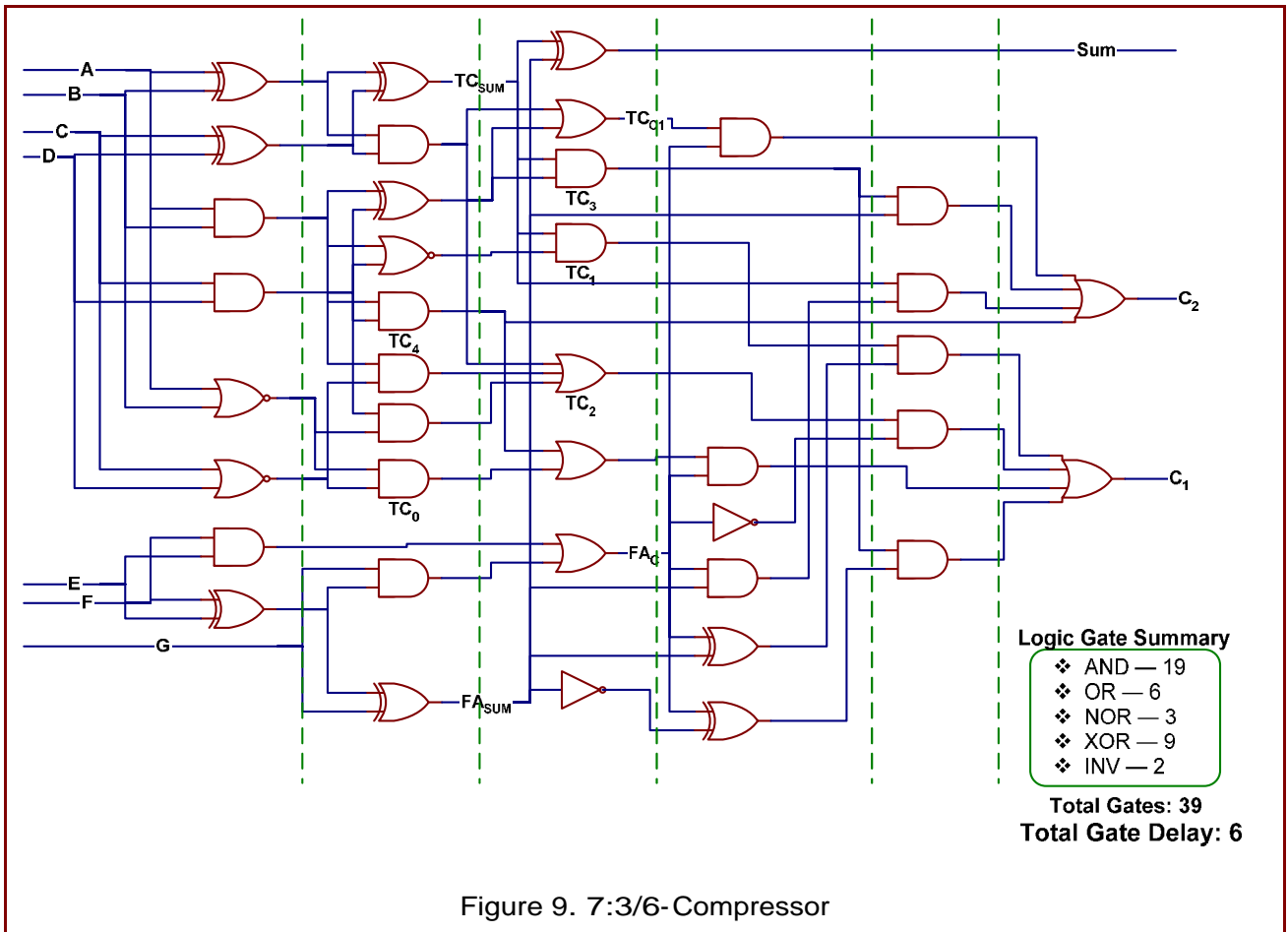
	FA _{SUM}	FA _C	C ₁	C ₂
TC ₀	0	0	0	0
	1	0	0	0
	0	1	1	0
	1	1	1	0
TC ₁	0	0	0	0
	1	0	1	0
	0	1	1	0
	1	1	0	1
TC ₂	0	0	1	0
	1	0	1	0
	0	1	0	1
	1	1	0	1
TC ₃	0	0	1	0
	1	0	0	1
	0	1	0	1
	1	1	1	1
TC ₄	0	0	0	1
	1	0	0	1
	0	1	1	1
	1	1	1	1

Table 1. Truth Table for 7-Bit Addition

$$C_2 = TC_4 + TC_{C1}FA_C + TC_3FA_{SUM} + TC_1(FA_{SUM}FA_C)$$

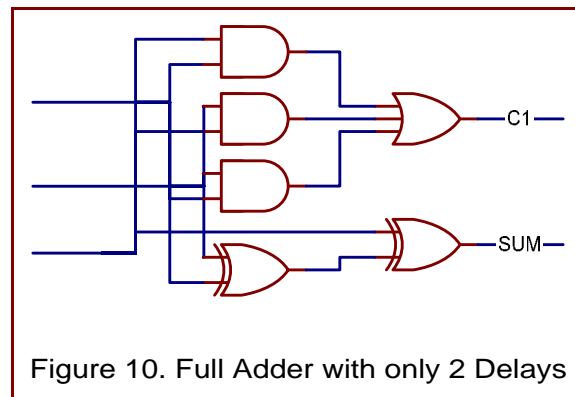
7:3/6-Compressor

Figure 8 shows the combined 7:3/6-Compressor. This circuit uses 39 logic gates and has a total gate delay of 6. The main problem with this compressor is that the gate



delay is too long to offer much in the way of performance improvements. Intuitively, 3 3:2/3-Compressors could be configured to reduce 6 numbers to 3 in 6 gate delays. Since the 7:3/6 compressor can reduce 7 numbers to 3 in the same number of delays, it represents only a marginal improvement over former and probably not worth the increased complexity of the circuit.

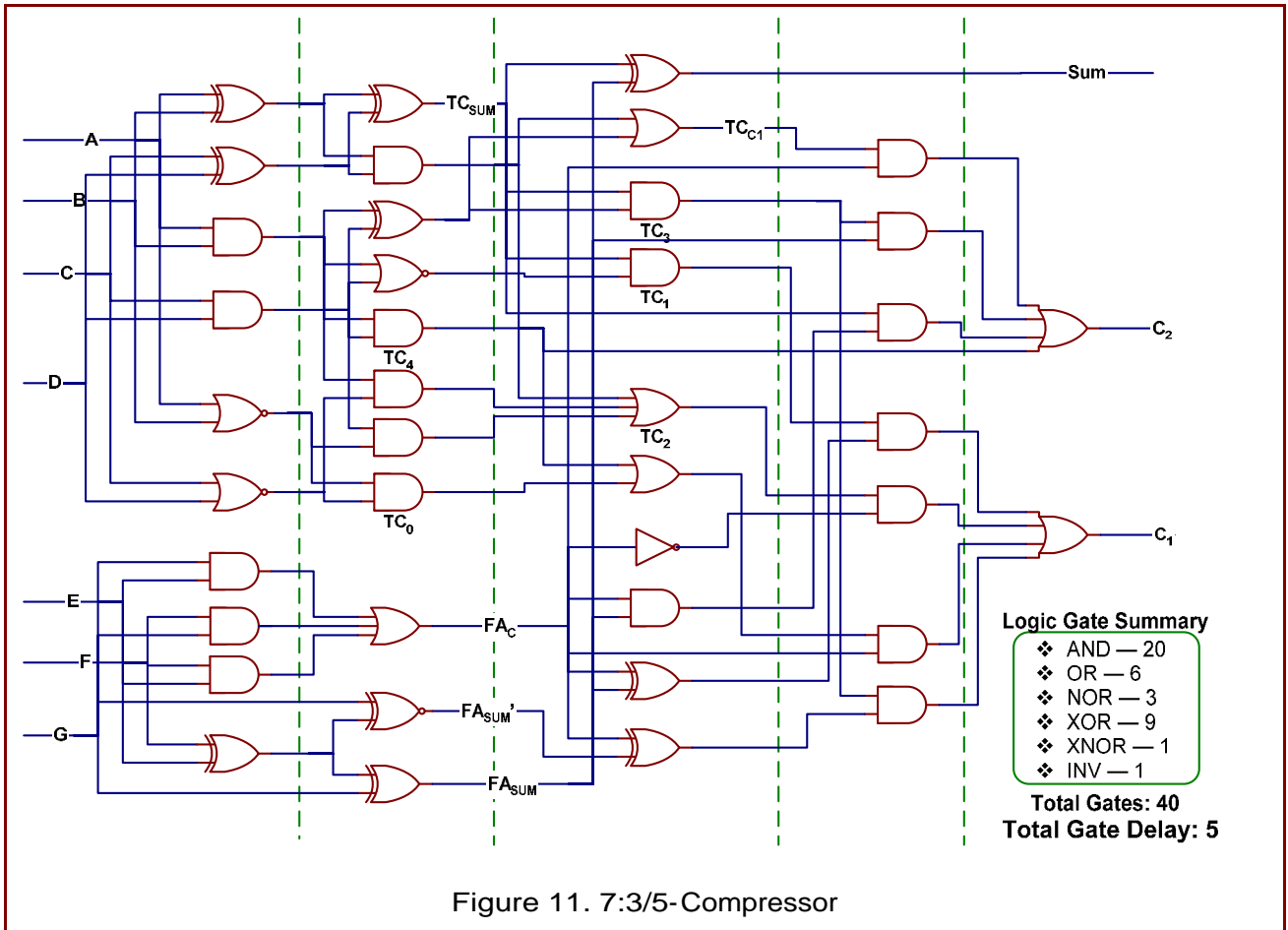
However, inspection of figure 9 will show that the only reason we have a gate delay of 6 is that the full adder takes 3 gate delays to produce its carry. Thus, the decisions that require the carry from the full adder must wait one gate delay for it to be ready. Figure 10 shows how the full



adder can be modified to produce its carry in only 2 gate delays. This modification can be done while only adding one logic gate to the circuit. Integration of this idea with the 7/3:6-Compressor allows for the removal of a delay.

7/3:5-Compressor

Figure 11 shows the final 7/3:5-Compressor.



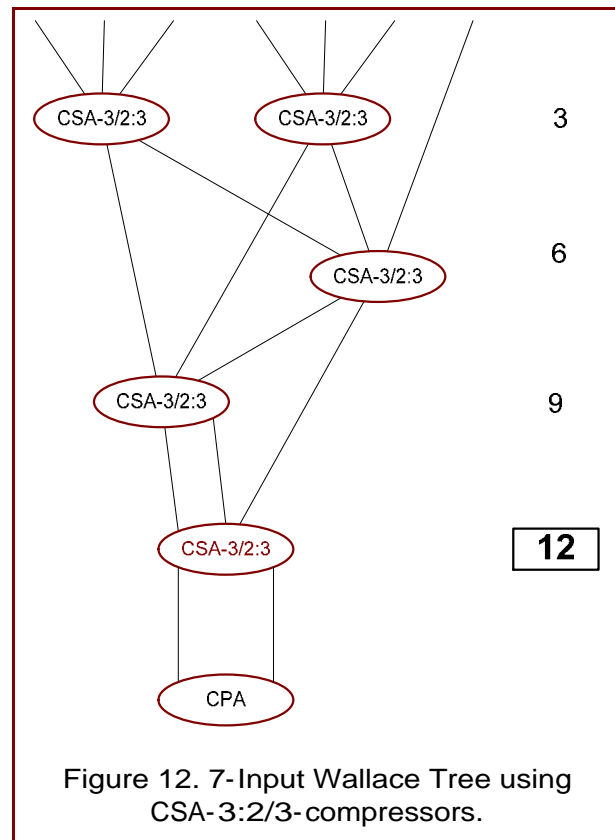
Carry-Save Adder (CSA) Implementation

Wallace Tree

Compressors can be configured in such a way that the carry bits are not rippled through to the next stage, but are saved in order to be used as an input into the next compressor. This allows for the parallel processing of multi-operand addition. One of the most common ways of doing this is by constructing a Wallace Tree with a set of 3:2/3-compressors. Figure 12 shows an example of such a configuration with 3:2/3-compressors. A carry save adder using 3:2/3-compressors is referred to here as a CSA-3:2/3.

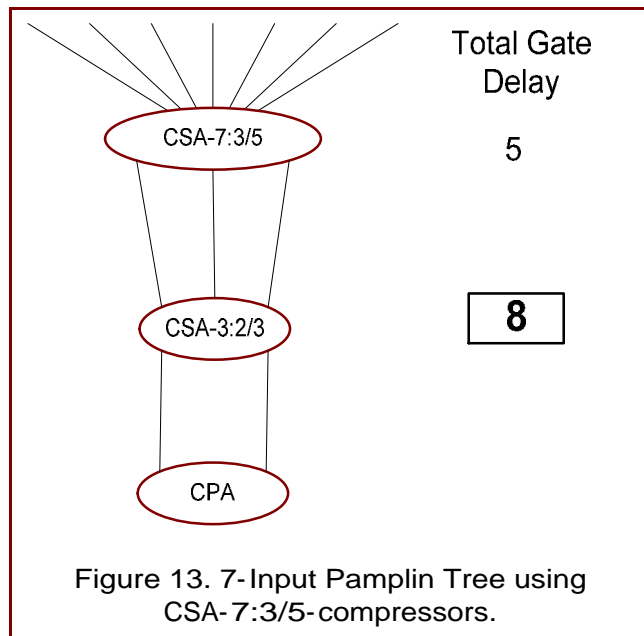
By using the outputs of one level of the tree as the inputs into the next level, we reduce 7 inputs down to 2 in 4 levels. The big advantage here is that the gate delay is independent of the word size. In other words, whether 32-bit numbers or 64-bit numbers need be added, the overall gate delay will remain unchanged.

The addition of the final 2 numbers is done here by a standard carry propagation adder (CPA).



Pamplin Tree

The Pamplin Tree, as shown in Figure 13, is in fact a derivation of the original Wallace Tree; instead of using standard full adders, the new 7:3/5-compressor is used. In this way, seven inputs can be reduced to 3 with just a single reduction. The final stage is reduce to 2 operands using a standard CSA-3:2/3 since the new compressor is not practical with just 3 inputs. Thus, the performance comparisons made in the following section only include the reduction to the final 3 inputs. Appendix B shows a Wallace tree and a Pamplin tree with 27 inputs each.



Pamplin Tree Analysis

It has yet to be shown that creating a CSA tree using the 7:3/5-compressors will actually improve performance as the number of inputs increases.

Performance

The performance of both CSA implementations is directly proportional to the height of the tree produced by both methods. Thus a calculation of the tree height for a given number of inputs is the first step in performance analysis. Each level of the tree will work in parallel at the delay of the individual CSA used for that level. Thus, the height of the tree multiplied by the total gate delay for the CSA will give the total delay to reduce the numbers to 3 inputs.

Tree Adder	Wallace		Pamplin		Pamplin	
	CSA-3:2/3		CSA-7:3/6		CSA-7:3/5	
No of Inputs (n)	h(n)	g(h)	h(n)	g(h)	h(n)	g(h)
7	4	12	2	9	2	8
25	7	21	3	15	3	13
100	10	30	5	27	5	23
250	12	36	6	33	6	28
500	14	42	7	39	7	33
750	15	45	7	39	7	33
1,000	16	48	8	45	8	38
2,500	18	54	9	51	9	43
5,000	20	60	10	57	10	48
7,500	21	63	10	57	10	48
10,000	22	66	11	63	11	53
50,000	25	75	12	69	12	58
100,000	27	81	13	75	13	63
1,000,000	33	99	16	93	16	78
10,000,000	39	117	19	111	19	93
100,000,000	44	132	21	123	21	103
1,000,000,000	50	150	24	141	24	118

Table 2. Performance Comparison of CSA implementations

Wallace Tree

The overall height of the Wallace tree can be computed by looking at the number of inputs required to reduce the tree down to 3 inputs using a 3:2 ratio for reduction. The value $h(n)$ represents the height of the tree with n inputs. The recursive equation for calculating the height of a CSA-3:2/3 Wallace Tree $h(n)$ and the associated delay cost $g(n)$ is:

$$h(n) = 1 + h\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) \approx O(\log_{1.5} n)$$

$$g(n) = 3 \bullet h(n)$$

Table 2 shows the calculations of $h(n)$ and $g(n)$ for various values of n .

Pamplin Tree

The overall height of the Pamplin tree can be computed by looking at the number of inputs required to reduce the tree down to 3 inputs using a 7:3 ratio for reduction. The recursive equation for calculating the height of a CSA-7:3/5 Pamplin Tree $h(n)$ and the associated delay cost $g(n)$ is:

$$h(n) = 1 + h\left(\left\lfloor \frac{3n}{7} \right\rfloor\right) \approx O(\log_{2.3} n)$$

$$g(n) = 5 \bullet h(n)$$

Delay Comparison

Table 2 shows the overall tree height $h(n)$ and associated cost $g(n)$ for CSA tree implementations using the various CSAs discussed in this paper. Notice that our intuitive assumption, made earlier in this paper, that the CSA-7:3/6 implementation would only show marginal performance improvements is correct. The average theoretical performance improvement for a CSA tree using the 7:3/6-compressor is 9.57%. However, the modifications to the adder to create a 7:3/5-compressor is more than justified by the drastic improvement in performance of 23.5% over the standard CSA-3:2/3 tree implementation. Appendix A shows a graph of the delay in terms of logic gates for various number of inputs.

Cost

At first glance the cost calculations would appear to be a landslide in favor of the old implementation, as the CSA-3:2/3 has only 5 logic gates and the CSA-7:3/5 has 8 times more than that. However, it should be noted that the number of CSA's required to perform the same calculation is not the same for each implementation. However, the calculation of the total number of CSA's is fairly straightforward.

Wallace Tree

The number of CSA's required to perform a reduction of n inputs to 3 output is given by the recursive function as show in Table 3 where $c(n)$ is the number of CSA's. The starting equation is based on the fact that the number of CSA's is equal to the number of CSA's on the current level of the tree ($n \bmod 3$) plus the number of CSA's for all subsequent levels of the tree ($c(2n/3)$). By ignoring the floor and ceiling operations we can easily solve this recursion to demonstrate that the number of CSA-3:2's required is exactly proportional to the number of inputs.

Appendix B demonstrates this relationship on a small scale in that a 27 input CSA-3:2/3 Wallace Tree requires approximately 24 CSA's in order to reduce to 2 inputs.\

$$c(n) = \left\lfloor \frac{n}{3} \right\rfloor + c\left(\left\lceil \frac{2n}{3} \right\rceil\right)$$

$$c\left(\frac{2n}{3}\right) = \frac{2n/3}{3} + c\left(\frac{2(2n/3)}{3}\right) = \frac{2n}{9} + c\left(\frac{4n}{9}\right)$$

$$c\left(\frac{4n}{9}\right) = \frac{4n/9}{3} + c\left(\frac{2(4n/9)}{3}\right) = \frac{4n}{27} + c\left(\frac{8n}{27}\right)$$

$$c(n) = \frac{n}{3} + \frac{2n}{9} + \frac{4n}{9} + \dots + 0$$

$$c(n) = \frac{2^0 n}{3^1} + \frac{2^1 n}{3^2} + \frac{2^2 n}{3^3} + \dots + 0$$

$$c(n) = \sum_{i=0}^n \frac{2^i}{3^{i+1}} n = n \sum_{i=0}^n \frac{2^i}{3^{i+1}} = n \sum_{i=0}^n \frac{1}{3} \cdot \frac{2^i}{3^i}$$

$$c(n) = \frac{1}{3} n \sum_{i=0}^n \left(\frac{2}{3}\right)^i \approx \frac{1}{3} n \cdot \frac{1}{1 - 2/3} = \frac{1}{3} n \cdot \frac{3}{1}$$

$$c(n) \approx n$$

Table 3. Number of CSA-3:2/3's for arbitrary number of inputs

Pamplin Tree

The starting equation for the Pamplin tree only changes in terms of the reductions. The number of CSA's for a level is $n \bmod 7$ and the number of additional levels is $c(3n/7)$. Thus, we can solve for the total number of actual CSA-7:3/5 circuits required for n inputs in a similar way. Table 4 shows that the final number of CSA's is directly proportional to $\frac{1}{4}n$. This means that for the same number of inputs the CSA-7:3/5 tree implementation uses only $\frac{1}{4}$ the CSA circuits as its CSA-3:2/3 counterpart.

Total Gate Comparison

This analysis reveals that the overall cost difference from the standard CSA-3:2/3 implementation to a CSA-7:3/5 implementation is NOT a factor of 8 since it requires 75% fewer CSA's for the 7:3 reduction tree. Thus, the actual difference in cost is a factor of 2.

$$c(n) = \left\lfloor \frac{n}{7} \right\rfloor + c\left(\left\lceil \frac{3n}{7} \right\rceil\right)$$

$$c\left(\frac{3n}{7}\right) = \frac{3n/7}{7} + c\left(\frac{3(3n/7)}{7}\right) = \frac{3n}{49} + c\left(\frac{9n}{49}\right)$$

$$c\left(\frac{9n}{49}\right) = \frac{9n/49}{7} + c\left(\frac{3(9n/49)}{7}\right) = \frac{9n}{343} + c\left(\frac{27n}{343}\right)$$

$$c(n) = \frac{n}{7} + \frac{3n}{49} + \frac{9n}{343} + \dots + 0$$

$$c(n) = \frac{3^0 n}{7^1} + \frac{3^1 n}{7^2} + \frac{3^2 n}{7^3} + \dots + 0$$

$$c(n) = \sum_{i=0}^n \frac{3^i}{7^{i+1}} n = n \sum_{i=0}^n \frac{3^i}{7^{i+1}} = n \sum_{i=0}^n \frac{1}{7} \cdot \frac{3^i}{7^i}$$

$$c(n) = \frac{1}{7} n \sum_{i=0}^n \left(\frac{3}{7}\right)^i \approx \frac{1}{7} n \cdot \frac{1}{1 - 3/7} = \frac{1}{7} n \cdot \frac{7}{4}$$

$$c(n) \approx \frac{1}{4} n$$

Table 4. Number of CSA-7:3/5's for arbitrary number of inputs

Conclusion

A new combinatorial circuit compressor is designed that will take as input 7 numbers and add them while also outputting 2 carry bits. This adder can be created with a gate delay time of 5 as opposed to the gate delay of 3 for the standard full adder. This circuit is called a 7:3/5-compressor.

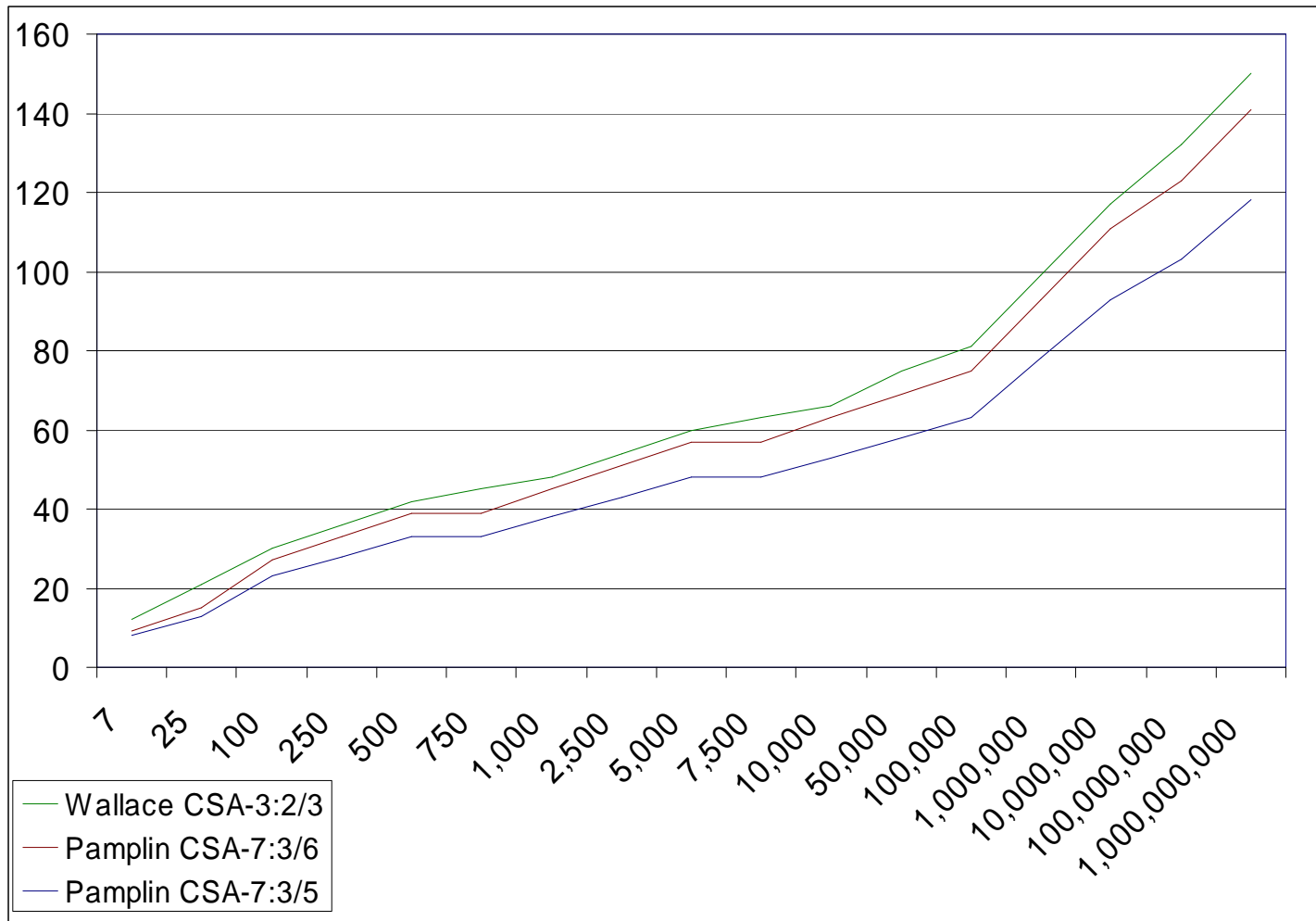
This new compressor allows for the creation of arbitrary word length adders that can add a large number of operands more quickly than the traditional full adder. By configuring this adder for use as a carry save adder in a tree structure, computation of large numbers of operands can be reduced by an average of 23.5%. This performance improvement comes at a cost of approximately twice the number of logic gates as the traditional Wallace Tree implementation.

Future Work

The actual transistor implementation of the new 7:3/5-compressor has not been designed. It is very likely that the overall delay cost can be reduced in terms of transistor delay since a large number of gates can take advantage of pass-logic implementations, such, all of the AND gates located in the 4th gate delay region of Figure 11.

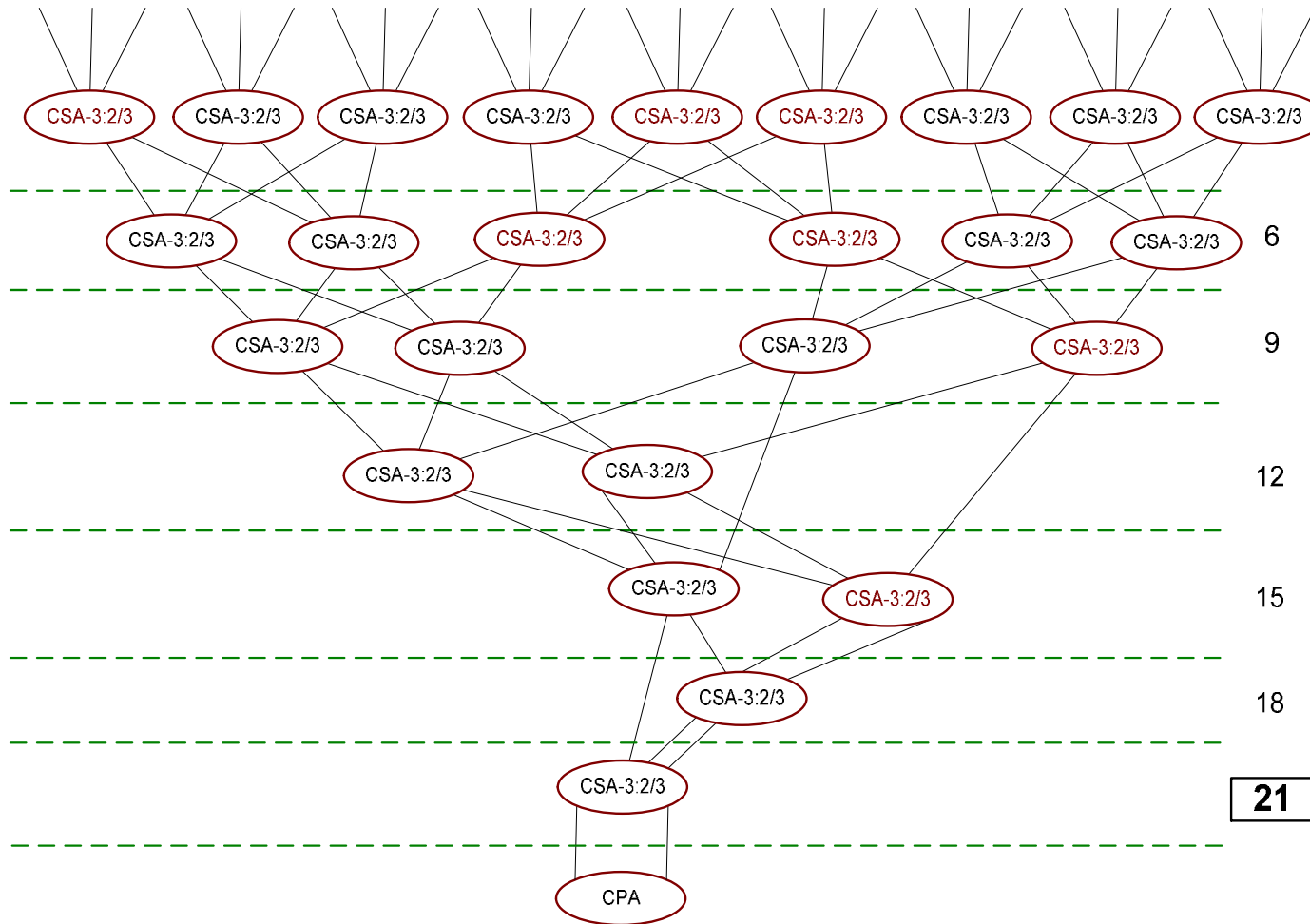
In addition, actual creation of a VLSI chip to perform multi-operand addition and subsequent input simulations could be done to verify the mathematical results presented in this paper.

Appendix A – Graph of Delay vs. Number of Inputs



Appendix B – 27-Input Tree Implementations

27-Input Wallace Tree



27-Input Pamplin Tree

