5-2004

The development of an innovative adder design evaluated using programmable logic.

James A. Haas 1971-
University of Louisville

THE DEVELOPMENT OF AN INNOVATIVE ADDER DESIGN EVALUATED USING PROGRAMMABLE LOGIC

By

James A. Haas
B.S.E.E., Purdue University, 1994

A Thesis
Submitted to the Faculty of the
Graduate School of the University of Louisville
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Electrical Engineering
University of Louisville

May, 2004

THE DEVELOPMENT OF AN INNOVATIVE ADDER DESIGN EVALUATED USING PROGRAMMABLE LOGIC

By

James A. Haas
B.S.E.E., Purdue University, 1994

A Thesis Approved on

April 12th, 2004

by the following Thesis Committee:

Dr. John F. Naber, Thesis Director

Dr. Thomas G. Cleaver

Dr. Rammohan K. Ragade

DEDICATION

This thesis is dedicated to my family

Mrs. Marcella M. Haas

and

Ms. Emily M. Haas

who have made this thesis all worthwhile.

ACKNOWLEDGEMENTS

I would like to thank my thesis director, Dr. John Naber, for his leadership and direction. I would also like to thank the other committee members, Dr. Thomas Cleaver and Dr. Rammohan Ragade, for their willingness to participate. I would also like to thank my wife, Marcie, and daughter, Emily, for their patience with me. They encouraged me and gave me a reason to finish. Also, many thanks are due to my family: George, Carol, Sue, Greg, and Katie. The loving care shown by each of you is what helped me to make it through the hard times. Finally, I would like to thank my close friends: Paul, Jim, Kent, and Wes. Their willingness to give their time to provide advice, proofreading, and engineering expertise was invaluable.

ABSTRACT

THE DEVELOPMENT OF AN INNOVATIVE ADDER DESIGN EVALUATED USING PROGRAMMABLE LOGIC

James A. Haas

April 23rd, 2004

This research evaluates an innovative binary adder design and compares it against five standard adder designs. It begins with an algorithmic description of the five standard designs followed by the innovative design. It uses two metrics, speed and size, to establish a fair comparison among the designs and draw conclusions about the performance and usability of the innovative design. The metrics are applied to theory, simulation, and implementation of the adder designs. The latter part of the research draws conclusions from the analysis of these metrics to establish a fair comparison between the innovative and existing designs.

The five standard designs are the carry-ripple, carry-complete, carry-lookahead, carry-select, and pyramid. The carry-ripple design is the fundamental and most straight-forward approach to addition. The carry-complete takes the carry-ripple design and adds a signal to detect when the addition is complete. The carry-lookahead design uses some intermediate signals to add multiple bits concurrently. The carry-select design is a brute force approach that allows high speed for a large gate count. Lastly, the pyramid design divides the addition into multiple stages, each calculating a single step of the addition process. The innovative design, called the carry-feedback, works by starting with the addends and iterating towards the solution, something unique from the other designs causing the sum to be latched by the adder. It's innovative approach provides a completion signal, similar to the carry-complete adder.

The research comes to the conclusion that the carry-feedback design is noteworthy deserving further attention. The carry-feedback design's performance along with its feature of latching the results and ability to signal completion make it an excellent candidate for asynchronous circuits, an area of continued interest in microprocessors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Concepts of Binary Addition

Binary addition is one of the most fundamental calculations performed by today's microprocessors (Patterson & Hennessy, 1994). Binary addition is performed in much the same way as decimal addition. Each digit of the vector is added, from right to left, producing a sum and a carry. The sum of two binary digits is very simple to calculate, and can be expressed in a simple truth table, often called a "half-adder" (Wakerly, 2000):

Table 1

Binary Half-Adder Truth Table

Input 1	*Input 2*	*Carry*	*Sum*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

In order to calculate the addition of a vector longer than one bit, this table must be expanded to include the addition of the carry output of the previous bit. This is expressed in the following truth table, often called a "full-adder" (Wakerly, 2000):

Table 2

Binary Full-Adder Truth Table

Carry-In	*Input 1*	*Input 2*	*Carry-Out*	*Sum*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

To provide a method for handling negative numbers, binary vectors are typically expressed in a form called the 2s complement. The 2s complement uses the most significant bit to indicate the sign (0 is positive, 1 is negative), and requires a specific bit-length to be chosen. The 2s complement is calculated by performing a 1s complement of the vector that describes the positive number, and adding 1. An example of this would be to model -7 in a 4-bit number would result in the following conversion. As a 4-bit binary number, '7' is expressed as '0111'. First, we perform a 1s complement to give '1000'. Finally we add one to give '1001'. By quick examination, we can determine that a 2s complement number of bit length N will result in a number in the range: $-2^{N-1} <=$ number $< 2^{N-1}$ (Omondi, 1994).

In most addition schemes, there is no direct way to perform subtraction. To bypass this limitation, the complement of the number to be subtracted is added. The 2s complement is a quick and easy method to implement to perform subtraction. For example, instead of subtracting '55' from '77', '-55' is added to '77' (Patterson & Hennessy, 1994)

Because of the limitations of the specific bit sizes in microprocessors, specific tests must be added to watch for overflow conditions, or conditions that result in an incorrect result. These conditions occur whether the addition is signed or not. This is because any addition of numbers of bit sized N can potentially result in an answer that needs (N+1) bits to describe. For example, if you add the unsigned 3-bit numbers of '111' and '001' together, you end up with '1000', a 4-bit number (Omondi, 1994).

Microprocessor complexity has grown quickly, and the width of the binary vectors used has increased from 8-bits to 32-bits, 64-bits, and beyond. Although the sum may be calculated by a binary adder quickly, the inability of most binary adders to determine whether the addition is complete requires the worst case to be assumed. The worst case addition is based upon the length of time for the carry calculation to propagate across the entire width of the sum vector. As such, addition of vectors of width N can require N operations to calculate despite the fact that the average case has but $\log_2 N$ carry propagations (Omondi, 1994). Therefore, tremendous research has gone into reducing the time required to perform that addition.

The research presented here investigates an innovative implementation of a binary adder, hereafter called a carry-feedback adder. The carry-feedback adder is compared against five traditional binary adder designs. The following five designs were chosen because they are either fundamental or highly optimized: carry-ripple, carry-completion, carry-lookahead, carry-select, and pyramid adder schemes. The design comparison metrics are speed and size (Omondi, 1994) that are combined to determine overall performance.

There is tremendous difficulty in establishing fairness in regard to these two metrics, as the exact implementation of a design often depends on the transistor technology being used. In an effort to maintain simplicity, factors such as 'fan-out' (a single output attached to too many gate inputs) and 'fan-in' (too many inputs to logic gate) are ignored. As such, speed is measured in terms of gate delays, and the gates are assumed to all require the same amount of time, represented as t. Likewise, size is measured in gate count, and gates are all assumed to be the same size. Adder variables are chosen such that the values regarding fan-in and fan-out are reasonable.

The number of gate delays is calculated by determining the worst-case path through the adder. For example, given the equation AB + CD, the number of gate delays is two: one to determine the outputs of the gate (A AND B) and (C AND D), which are calculated simultaneously, another to determine the output of the OR gate that combines them. Likewise, the gate count is determined by summing up the total number of gates in the adder (XOR is counted as a single gate). For example, given the equation (AB + CD), the number of gates is three: one for AB, one for BC, and one for AB + CD.

Adder performance is measured by multiplying speed and size. Adders with lowers values for this calculated parameter indicate a higher performance. This research weights speed and size equally in regards to performance, but it is a simple change to weight one metric more important than another.

For all adder designs and comparisons, the vectors **A** and **B** denote the addends, the vector **C** denotes the carries, and the vector **S** denotes the sum. All of the vectors are of width N. A subscript of n on a vector, such as A_n, represents the single bit of vector **A** at position n, where n ranges from 0 to (N – 1). C_{in} is a single bit input to the adder representing the carry input, sometimes represented as C_0. C_{out} is a single bit output of the adder representing the carry output, sometimes represented as C_N.

Where latching is required, two technologies are compared. The first is a traditional D flip-flop; the second is an SRAM cell. D flip-flops have 6 gates and require 3t gate delays to latch the output (Wakerly, 2000). SRAM cells have 1 gate equivalent and require 1t gate delays to latch the output. Both of these technologies are used to provide an adequate comparison for implementations where SRAM cells, the obviously better choice, are not available.

CHAPTER 2

BINARY ADDERS

Carry Ripple Adder

The carry-ripple design is the least complex binary adder design. It is constructed by connecting a large number of single-bit full-adders together in a chain. The outputs are dependent on inputs given at the beginning of the calculation, as well as outputs from the previous significant bit. Thus, this design is aptly named, as the output of the addition is not complete until the carry has rippled through each bit of the vector until completion (Omondi, 1994). The following block diagram illustrates this concept:

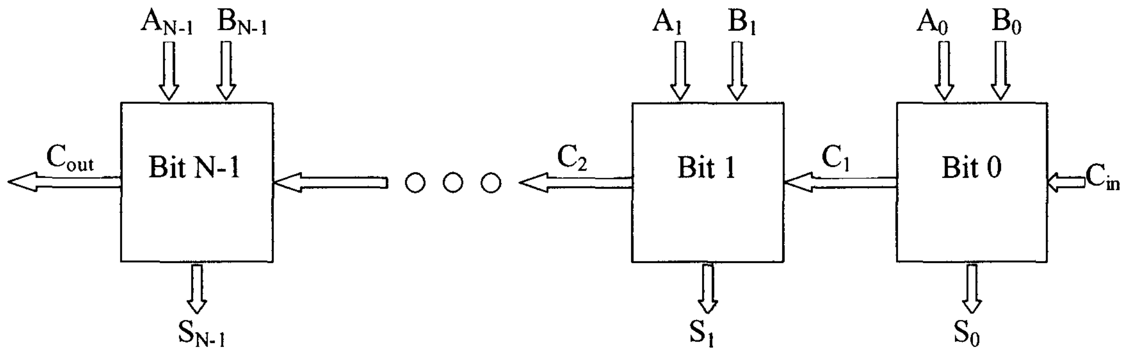


Figure 1. Block Diagram for the Carry-Ripple Algorithm.

Full-adders are chained together to perform addition on vectors.

The equations describing an implementation of the aforementioned block diagram for chained full-adders are as follows.

$$S_n = A_n \oplus B_n \oplus C_{n-1}$$

$$C_n = A_nC_{n-1} + B_nC_{n-1} + A_nB_n$$

The size of this adder is 6N gates: 2N for the **S** vector and 4N for the **C** vector. Since the carry-ripple algorithm has no indication of when addition is complete, the time required to complete an addition is N times the number of gate delays of the carry-out calculation, or in this case 2Nτ.

Carry-Completion Adder

The carry-completion design is a simple modification to the carry-ripple design, adding an output signal that indicates when the summation is complete. The basic idea is to modify the carry-ripple design to have two carry chains, one to detect the 0-carries, the other to detect the 1-carries. It is known that the addition is complete when every bit has either generated a carry or is known to not generate a carry (Omondi, 1994). In the following equations, C_n^0 will denote the 0-carry from stage n, and C_n^1 will denote the 1-carry from stage n. Also, the equation for the DONE signal, the signal that indicates the addition is complete is included:

$$C_n^1 = A_nB_n + (A_n \oplus B_n)\, C_{n-1}^1$$

$C_n^0 = \overline{A_n}\ \overline{B_n} + (A_n \oplus B_n)\overline{C_{n-1}^0}$

$S_n = A_n \oplus B_n \oplus C_{n-1}^1$

$\text{DONE} = (C_0^0+C_0^1)(C_1^0+C_1^1) \ldots (C_{n-1}^0+C_{n-1}^1)$

The size of this adder is (9N + 1) gates: 4N for the $\mathbf{C^1}$ vector, 3N for the $\mathbf{C^0}$ vector, N for the **S** vector, and (N + 1) for the DONE signal. The time required to complete an addition of bit length N is summed up by the following three conditions (two gate delays are added to every condition for initialization): best case 7τ, average case $(2\log_2 N + 4)\tau$, worst case $(2N + 4)\tau$. Since the carry-completion algorithm has a signal to indicate when addition is complete, the average case of $(2\log_2 N + 4)\tau$ is used.

Carry-Lookahead Adder

The carry-lookahead design, often called a “Propagate/Generate (PG) Adder”, is an improvement to the carry-ripple design. This design implements additional logic to speed up the addition by determining the carry without waiting for the summation to finish. This is accomplished with the addition of intermediate signals called propagate and generate. The generate signal describes the condition of a single digit addition that will generate a carry regardless of the C_{in} signal. The propagate signal describes the condition of a single addition that will propagate a carry only if the C_{in} signal is a one (Omondi, 1994).

A bit will have a carry if either it generates a carry or it propagates a carry and the previous bit will generate a carry. This can be implemented through the following equations:

$$S_n = A_n \oplus B_n \oplus C_{n-1}$$

$$P_n = A_n+B_n$$

$$G_n = A_nB_n$$

$$C_n = G_n + P_nC_{n-1}$$

The formula for C_n can be expanded to determine the equation for the nth carry bit. For example, suppose you wanted to calculate the equation for the 4th carry.

$$C_4 = G_4 + P_4C_3$$

$$C_4 = G_4 + P_4(G_3 + P_3C_2)$$

$$C_4 = G_4 + P_4(G_3 + P_3(G_2 + P_2C_1))$$

$$C_4 = G_4 + P_4(G_3 + P_3(G_2 + P_2(G_1 + P_1C_0)))$$

$$C_4 = G_4 + P_4(G_3 + P_3(G_2 + P_2(G_1 + P_1(G_0 + P_0C_{in})))$$

C_{in} can be described as G_{-1}, or the carry generated as input to the adder. The equation becomes summarized as:

$$C_4 = G_4 + P_4G_3 + P_4P_3G_2 + P_4P_3P_2G_1 + P_4P_3P_2P_1G_0 + P_4P_3P_2P_1P_0G_{-1}$$

An important observation from the resulting equation for C_4 that can easily be seen is that in theory, any arbitrarily chosen C_n will result in 3-layer logic. This implies that any length addition can be solved in three gate delays. However, the AND and OR gates quickly grow in complexity because they require increasing number of inputs. Typically, the number of inputs is limited to some value determined by the capabilities of the hardware regarding fan-in.

To address this issue, several approaches have been taken to form a hierarchy of carry-lookahead calculation. Some of these include rippling the carries of several smaller PG Adders connected in series. Another approach is to connect a series of several smaller carry-ripple adders together, and then use a PG Adder to generate the C_{in} signal for each block (Omondi, 1994). However, the fastest model is to use a pyramid of PG Adders to generate the C_{in} signal for blocks of PG Adders. This is commonly called a PG-PG Adder. The following block diagram illustrates how a PG-PG adder would be connected:

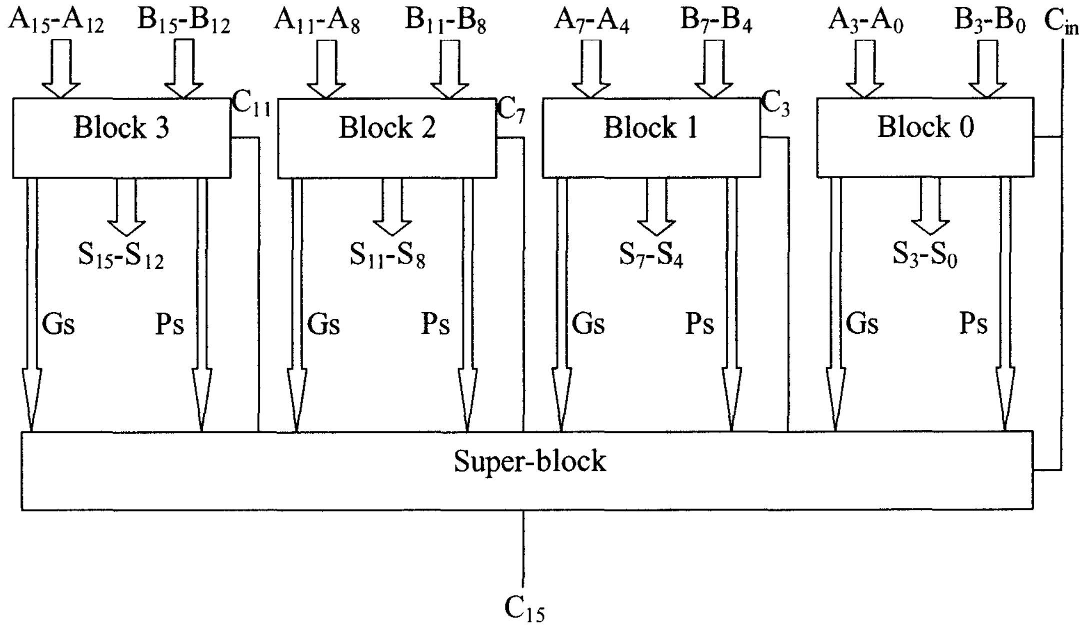


Figure 2. Block Diagram for a 16-bit PG-PG Carry-Lookahead Algorithm. Lookahead blocks perform addition on a subset of the data. The super-block takes propagate/generate data from the lookahead blocks and calculates intermediate carries. The number of levels can be increased to improve efficiency for larger width vectors. Each block and super-block is M units wide, in this example, four.

The super-blocks work much the same way as the smaller blocks. Each super-block takes propagate and generate outputs from the blocks beneath, combining them to form block propagate and block generate signals. The block propagate and block generate vectors are then combined in the same way as the propagate and generate vectors at the level below to determine the carry output for the super-block layer. For example, the super-block will have a C_{out} of 1 if the last block generates a carry, or if it will propagate a carry and the previous block generate a carry.

The size of this adder is given by the following formula of N/M(4M + M(M+1)/2 – 1) + ($\log_M N(\log_M N - 1)/2$)(5M + M(M+1)/2 – 1) + (M + 1) gates: N/M adder blocks of size (4M + M(M+1)/2 – 1) gates, ($\log_M N(\log_M N - 1)/2$) superblocks of size (5M + M(M+1)/2 – 1), and (M+1) gates for the final carry-out calculation. Since the PG-PG carry-lookahead algorithm has no indication of when addition is complete, the time required to complete addition with group size M is the gate delay of each block times the number of levels, or ($5\log_M N$)t.

Carry-Select Adder

The carry-select algorithm derives its name from the method in which the algorithm uses the carry to select which addition is correct. Adder blocks are duplicated, and the carry out selects which block is the correct one. By creating a pyramid of these blocks, the addition can be performed very quickly (Omondi, 1994). However, as the width of the addition vectors increases relative to the size of the adder blocks, the number of adder blocks increases exponentially. One implementation of this algorithm is illustrated in the following block diagram:

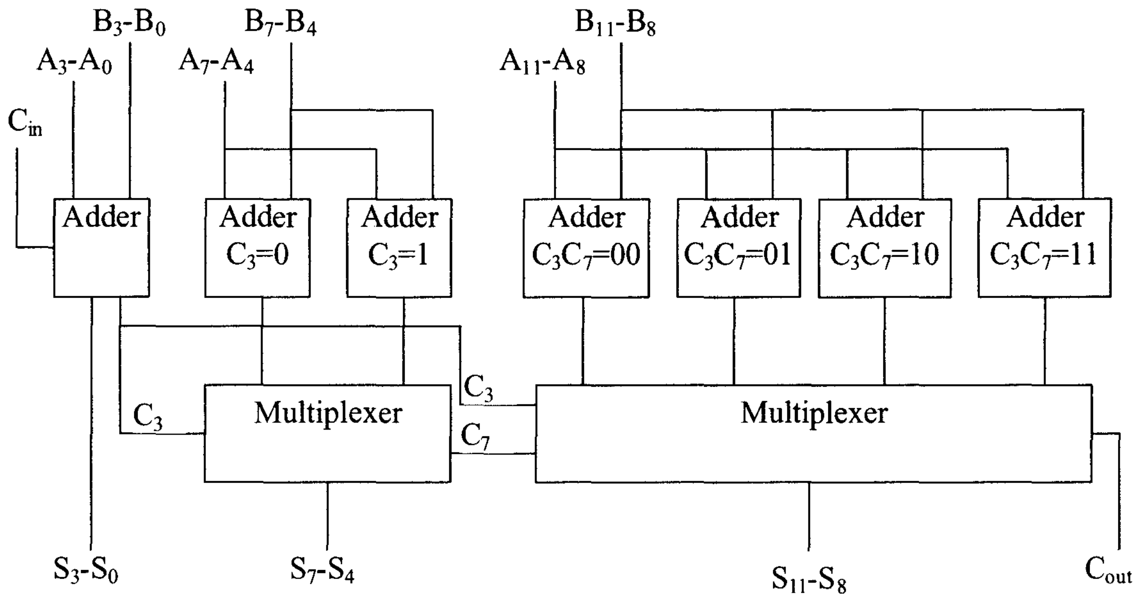


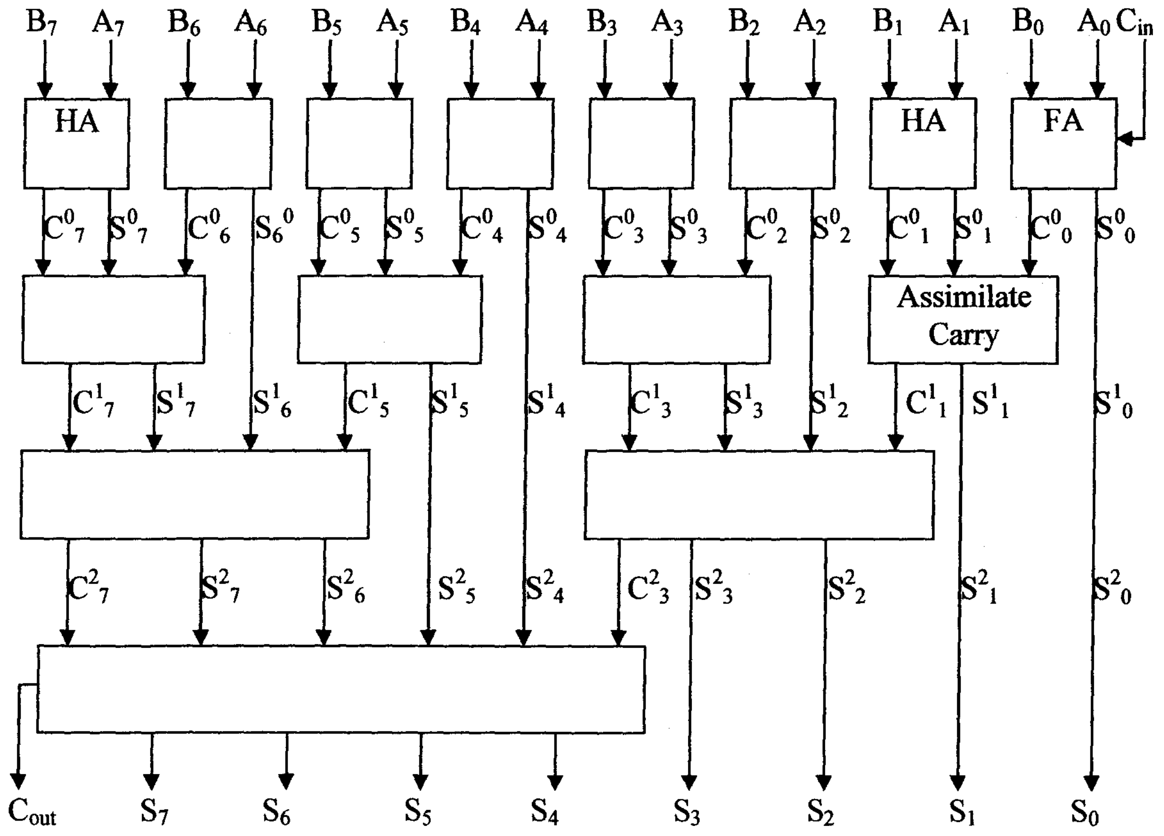
Figure 3. Block Diagram for a 12-bit adder using the Carry-Select Algorithm. Adder blocks perform addition on a subset of the data. The first adder unit performs like a normal adder. The other adder units have their carry inputs set to 0 or 1. The multiplexers take carry outputs from the adder blocks and select the proper sums.

Often, this algorithm is used in conjunction with other algorithms to improve efficiency. Each individual adder block can be of any type. This allows incredible flexibility in implementation, and provides multiple methods of optimization. A common choice is to make each adder block a carry-lookahead adder.

The size of a carry-select adder divided into carry-lookahead adder groups of size M is $(2^{N/M}-1)(4M + (M+1)(M+2)/2 - 1) + (2^{N/M} + N/M - 3)$ gates: $(2^{N/M}-1)$ adder blocks of size $(4M + (M+1)(M+2)/2 - 1)$ gates, and $(N/M - 1)$ multiplexer blocks that sum to a size of $(2^{N/M} + N/M - 3)$ gates. Since the carry-select algorithm does not require any rippling of the addition, the time required for addition is a constant 7t. However, an adder of small width is only practical for this application.

Pyramid Adder

The last algorithm discussed is the pyramid adder algorithm. The pyramid adder consists of two main parts. One part is made up of half-adders that calculate the partial sum and partial carry bits for all the stages. The other part is a pyramid structure whose base adjoins the other part, and where the partial carries are assimilated with the partial sums. The assimilation occurs over a number of stages in which carries are propagated over a predefined width, no more than twice the width of the previous stage (Omondi, 1994). This is illustrated in the following block diagram:



B7 A7 B6 A6 B5 A5 B4 A4 B3 A3 B2 A2 B1 A1 B0 A0 Cin
HA
HA
FA
Assimilate Carry
Cout S7 S6 S5 S4 S3 S2 S1 S0

Figure 4. Block diagram for an 8-bit adder using the pyramid algorithm. Partial carries are assimilated in blocks of twice the previous block size until the addition is complete.

The pyramid algorithm can be implemented through the following series of equations:

Stage 0: The partial sums and partial carries are produced.

$$S^0_0 = A_0 \oplus B_0 \oplus C_{in}$$

$$C^0_0 = A_0B_0 + (A_0 \oplus B_0 \oplus C_{in})$$

$S^0_j = A_j \oplus B_j$

$C^0_j = A_jB_j$

Stage 1: The stages are grouped in twos, and carries are assimilated within each group.

$S^1_{j+1} = S^0_{j+1} \oplus C^0_j$, j=0,2,4,...,n-2

$C^1_{j+1} = C^0_{j+1} + S^0_{j+1}C^0_j$

$S^1_j = S^0_j$, for all other j

Stage 2: The stages are grouped in fours and carries are propagated and assimilated within each group.

$S^2_{j+1} = S^1_{j+1} \oplus C^1_j$, j=1,5,9,...,n-3

$S^2_{j+2} = S^1_{j+2} \oplus S^1_{j+1}C^1_j$

$C^2_{j+2} = C^1_{j+2} + S^1_{j+2}S^1_{j+1}C^0_j$

$S^2_j = S^1_j$, for all other j

Stage 3: The same procedure is repeated with groups of eight.

$S^3_{j+1} = S^2_{j+1} \oplus C^2_j$, j=3,11,19,...,n-5

$S^3_{j+2} = S^2_{j+2} \oplus S^2_{j+1}C^2_j$

$S^3_{j+3} = S^2_{j+3} \oplus S^2_{j+2}S^2_{j+1}C^2_j$

$S^3_{j+4} = S^2_{j+4} \oplus S^2_{j+3}S^2_{j+2}S^2_{j+1}C^2_j$

$C^3_{j+4} = C^2_{j+4} + S^2_{j+4}S^2_{j+3}S^2_{j+2}S^2_{j+1}C^2_j$

$S^3_j = S^2_j$, for all other j

Stage k: Similar, with the groups size doubled at each step. In general, the group size at step k is 2k, and the assimilation is expressed by the following equations (j=2^k-1, 3*2^k-1, 5*2^k-1, ..., n-1-2^k-1).

$S^k_{j+1} = S^{k-1}_{j+1} \oplus C^{k-1}_j$

$S^k_{j+2} = S^{k-1}_{j+2} \oplus S^{k-1}_{j+1}C^{k-1}_j$

.

.

.

$S^k_{j+2\wedge(k-1)} = S^{k-1}_{j+2\wedge(k-1)} \oplus S^{k-1}_{j+2\wedge(k-1)-1}\, S^{k-1}_{j+2\wedge(k-1)-2}\ldots S^{k-1}_{j+1}C^{k-1}_j$

$C^k_{j+2\wedge(k-1)} = C^{k-1}_{j+2\wedge(k-1)} + S^{k-1}_{j+2\wedge(k-1)}\, S^{k-1}_{j+2\wedge(k-1)-1}\ldots S^{k-1}_{j+1}C^{k-1}_j$

$S^k_j = S^{k-1}_j$

The size of this adder is (2N+2) + N($\log_2$N($\log_2$N+1)/2) + N(1-$2^{-\log_2 N}$) gates: (2N+2) for the first stage, and then (N/2^i)(2^i+1) for each stage i afterwards. Since the pyramid algorithm has no indication of when addition is complete, the time required to complete an addition is the number of stages ($\log_2$N + 1) times the number of gate delays of each stage (2), plus an additional gate delay for the 0th stage, or (3+2$\log_2$N)τ.

Carry-Feedback Adder

The innovative adder design presented in this paper from this point forward is called the "carry-feedback" adder. The carry-feedback adder design is an unsophisticated design developed by the author while examining ways to speed up the average case addition. Typical additions performed by a human take advantage of the cases where there is no carry for a particular digit during addition. For example, 336 + 122 can typically be added more quickly than 938 + 239. As the lengths of the addend vectors increase, the percentage of "worst case" additions (additions that have a lot of digits that generate a carry) decrease. Mathematically, this is expressed by the following three cases for all binary addition for adding vectors of width N (Omondi, 1994):

1. The best case is no carry propagation.
2. The average case is $\log_2 N$ carries propagating.
3. The worst case is N carries propagating.

The author developed the carry-feedback algorithm to work iteratively to solve the addition rather than by directly calculating. The carry-feedback algorithm works by calculating two terms from the inputs: a partial-sum and a partial-carry. The partial-sum is calculated by performing a half-adder addition on all the digits. The partial-carry is calculated by performing a half-adder carry calculation on all the digits, and left shifting this vector one bit. Whenever a '1' is shifted out, it is latched as the 'carry-out' for the addition. Initially, the inputs are the two addends. Then, the partial-sum and partial carry are fed back in as the new inputs. The addition is complete when the partial-carry vector is all zeroes. This is illustrated in the following block diagram:

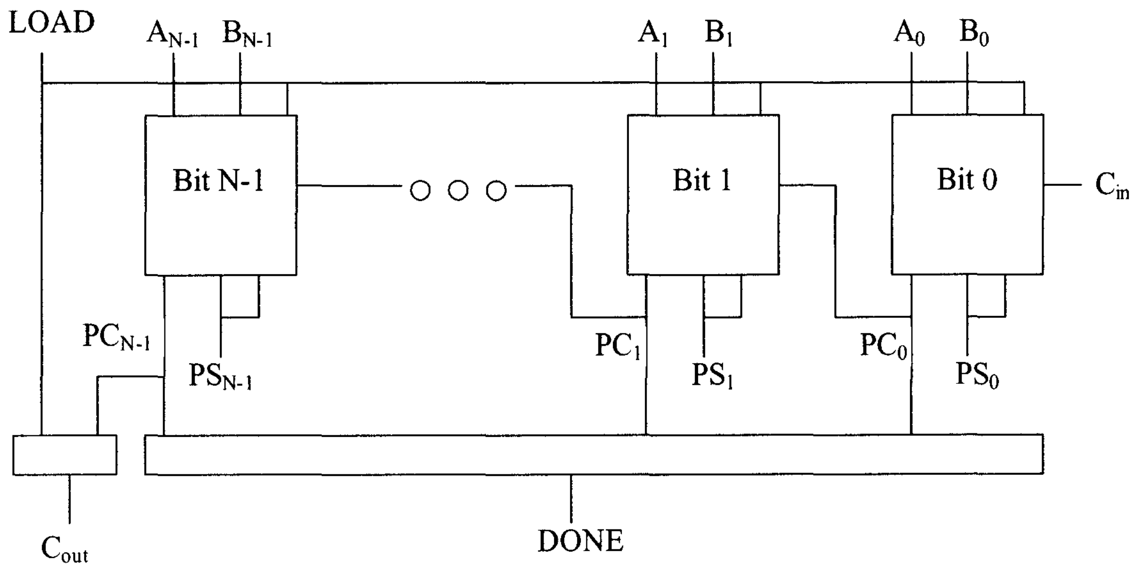


Figure 5. Block diagram for an adder using the carry-feedback algorithm.

Partial sums are fed back into each block, whereas partial carries are fed forward into the next block. Partial sums iterate to the sum whereas partial carries iterate to zero.

The carry-feedback adder received its name because of the way the outputs are fed back as inputs. The sequence of partial-sum vectors will approach and reach the actual sum. The sequence of partial-carry vectors will converge to zero. The following decimal addition illustrates the algorithm:

```
        54963
+       18265
```

```
-----------------
        62128 (partial sum vector, step 1)
+       11100 (partial carry vector shifted left, step 1)
-----------------
        73228 (partial sum vector, step 2)
        00000 (partial carry vector, step 2 and final step since 0)
```

In Boolean algebra, the equations are much simpler. One bit additions without regard to carry can be expressed using a logical XOR gate. Likewise, one bit carry calculations can be expressed using a logical AND gate. The output of the XOR gates will approach and reach the sum very quickly. This is illustrated in the following 8-bit addition example:

Table 3

Carry-Feedback Addition Example

Stage	*Input 1*	*Input 2*	*Carry-Out*	*Partial-Sum*	*Partial-Carry*
1	11101111 (-17d)	01101011 (107d)	0	10000100 (-124d)	11010110 (-42d)
2	10000100 (-124d)	11010110 (-42d)	1	01010010 (82d)	00001000 (8d)
3	01010010 (82d)	00001000 (8d)	1	01011010 (90d)	00000000

The carry-feedback adder design is simple and uses very few logic gates for large vector additions. Likewise, it can be very fast for certain additions. For the following equations, **A** and **B** denote binary vectors of length N. Let $\mathbf{A}_t$ denote the partial-sum vector at iteration t, and $\mathbf{A}_0$ is the first addend. Let $\mathbf{B}_t$ denote the partial-carry vector at iteration t, and $\mathbf{B}_0$ is the second addend. Let DONE denote the output indicating the addition is complete. The equations are given by:

$\mathbf{A}_t = \mathbf{A}_{t-1} \oplus \mathbf{B}_{t-1}$

$\mathbf{B}_t = (\mathbf{A}_{t-1} \bullet \mathbf{B}_{t-1}) << 1$; (Where '<< 1' represents left shifting the vector 1 bit)

$\text{DONE} = \overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \bullet \ldots \bullet \overline{B_{N-1}}$

The algorithm requires a clock to synchronize the feedback of signals. However, the clock is not required to be synchronized with anything else. The outputs are determined asynchronously, allowing the addition to be calculated at the fastest speed possible. The DONE signal allows nearly instantaneous reaction to the addition being complete.

When D flip-flops are used, the size of this adder is 13N+9 gates, 7N for the partial-sum and partial-carry calculation, 6N for latching these values, 4 for setting up Cin, 4 for setting up Cout, and 1 for calculating DONE. A single iteration of the algorithm requires 5t time, two for the logic and three to latch the output. Additionally, the DONE signal requires t time. When SRAM cells are used, the size reduces to 8N + 9 and the gate delay reduces to 3t. From the carry-propagation equations, the time required to complete an addition of bit length N is expressed in the following table (in all three cases, two gate delays are included for initialization):

Table 4

Addition Time for Carry-Feedback in Gate Delays

Type	*D flip-flops*	*SRAM*
Best Case	7t	5t
Average Case	$(5\log_2 N + 2)t$	$(3\log_2 N + 2)t$
Worst Case	$(5N + 2)t$	$(3N + 2)t$

For comparison with the other designs, the average time will be used. It is assumed that whatever device the adder is placed in will take advantage of the design's ability to notify when complete.

CHAPTER 3

ADDERS SUMMARY

Summary of Adder Gate Delays and Gate Counts

To provide a theoretical basis for comparison, the addition times and sizes for the six adder designs were computed. Addition times are represented through a count of gate delays with the assumption that each gate in the design has the same delay. Likewise, sizes are represented through a gate count, with the assumption that each gate in the design has the same size. The reader should understand that these are estimations used solely for magnitude comparisons. An exact calculation requires specifying the transistor technology used and is beyond the scope of this paper. The theoretical speed and size are based upon the optimal implementation of the design, not on programmable logic. Also, specific information regarding the maximum fan-in and fan-out is required.

Table 5 displays the theoretical addition times for each of the six adders at various bit widths. Some items of note: as expected on larger adders, the carry-ripple adder has the largest number of gate delays. Also as expected, the carry-select adder has the fewest number of gate delays, being a constant regardless of the adder size. The carry-complete and carry-feedback designs use the average case addition time instead of worst case because of their inclusion of a "DONE" signal. The other adders have gate delays on the same rough order of magnitude.

Table 5

Theoretical Addition Times of the Six Adders in Gate Delays

Type	*4bit*	*16bit*	*64bit*	*256bit*
Carry-Ripple	8	32	128	512
Carry-Completion	8	12	16	20
Carry-Lookahead	5	10	15	20
Carry-Select	7	7	7	7
Pyramid	7	11	15	19
Carry-Feedback (D flip-flop)	12	22	32	42
Carry-Feedback (SRAM)	8	14	20	26

Table 6 displays the theoretical gate count for each of the six adder designs at various bit widths. Some items of note: as expected, the carry-ripple adder has the smallest number of gates. Also as expected, the carry-select adder has the largest number, becoming unfeasible in the implementation chosen for large adders, showing the price paid for a constant addition time irrespective of size. The pyramid adder increases in size more rapidly than the remaining adders, all of which have sizes on the same rough order of magnitude.

Table 6

Theoretical Gate Count of the Six Adders (block sizes of 4 are used)

Type	*4bit*	*16bit*	*64bit*	*256bit*
Carry-Ripple	24	96	384	1536
Carry-Completion	37	145	577	2305
Carry-Lookahead	30	134	492	1779
Carry-Select	30	467	2031599	5.7185E+20
Pyramid	25	209	1537	9985
Carry-Feedback (D flip-flop)	61	217	841	3337
Carry-Feedback (SRAM)	41	137	521	2057

Table 7 displays the theoretical addition times and gate counts of the six adder designs for an arbitrary size of N bits. Some items of note: when the adders are compared side by side, the value of the carry-lookahead becomes apparent. The block size 'M' can be chosen to optimally balance the fan-in/fan-out issues of the technology used, as well as being convenient for the adder width. The size does not become prohibitive as to prevent its use, making it obvious why this adder is commonly chosen for microprocessor designs (Koren, 1993).

Table 7

Theoretical Addition Time and Gate Count of the Six Adders

Type	*Addition Time*	*Gate Count*
Carry-Ripple	$2Nt$	$6N$
Carry-Completion	$(2\log_2 N + 4)t$	$9N + 1$
Carry-Lookahead	$(5\log_M N)t$	$N/M(4M + M(M+1)/2 - 1) + (\log_M N(\log_M N - 1)/2)(5M + M(M+1)/2 - 1) + (M + 1)$
Carry-Select	$7t$	$(2^{N/M}-1)(4M + (M+1)(M+2)/2 - 1) + (2^{N/M} + N/M - 3)$
Pyramid	$(3+2\log_2 N)t$	$(2N+2) + N(\log_2 N(\log_2 N+1)/2) + N(1-2^{-\log_2 N})$
Carry-Feedback (D flip-flop)	$(5\log_2 N + 2)t$	$13N + 9$
Carry-Feedback (SRAM)	$(3\log_2 N + 2)t$	$8N + 9$

Table 8 displays the performance characteristics of the theoretical size and speed at various bit widths. The table clearly shows the rapid drop in performance in the carry-select adder caused by its exponential growth in size. Likewise, the linear decrease in speed shows the rapid drop in performance in the carry-ripple adder. Using the theoretical calculations, the carry-lookahead adder stands out as the clear victor in performance.

Table 8

Theoretical Performance of the Six Adders (lower is better)

Type	*4bit*	*16bit*	*64bit*	*256bit*
Carry-Ripple	192	3072	49152	786432
Carry-Completion	296	1740	9232	46100
Carry-Lookahead	150	1340	7380	35580
Carry-Select	210	3269	14221193	4.003E+21
Pyramid	175	2299	23055	189715
Carry-Feedback (D flip-flop)	732	4774	26912	140154
Carry-Feedback (SRAM)	328	1918	10420	53482

CHAPTER 4

DESIGN

Design Considerations

Since six binary adder designs of various sizes were being compared, a method to reduce the amount of overhead involved in the implementation was needed. Two objectives of reducing the overhead were desirable. The first was to use modern software tools and methodologies to provide for fast and easy design and implementation. The second was to use modern hardware technology to reduce or eliminate the need for custom chip design or lengthy breadboard wiring.

A single solution was found to address both these issues. A common design practice that is radically changing the way in which digital circuits are developed is to use an HDL (Hardware Description Language) on a programmable logic chip. An HDL is a language that is used to describe a digital circuit design in text. VHDL (VLSI HDL) and Verilog are commonly used HDLs. VHDL and Verilog use different syntaxes, but accomplish the same purpose: they force a specific grammar on a digital circuit design in such a way that the chip manufacturer can convert that design into an output format necessary to program the chip (Skahill, 1996).

A programmable logic chip is a highly configurable chip that can be programmed to represent different digital circuit designs. The chips are configurable by using flash or other technologies to allow gate configurations to be programmed in. The two flavors of programmable logic chips are CPLDs (Complex Programmable Logic Device) and FPGAs (Field Programmable Gate Array).

CPLDs are typically conventional AND/OR logic with latched outputs arranged in a complex fashion to allow extensive programmability, often using flash to store the configuration (making the configuration non-volatile). FPGAs are typically digital multiplexed lookup tables combined with a smaller amount of AND/OR logic that often allows for extreme programmability and optimization at the sacrifice of limited control on the implementation equations. FPGAs often use a volatile configuration storage method meaning they must be configured on each power cycle. Modern versions of both programmable logic chip types often include SRAM for storage usages.

Several programmable chip manufacturers were available, but Cypress was the one chosen for this project (Skahill, 1996). Their development tool, called "Warp", claims to provide all the features necessary to provide accurate simulations and implementations of the various designs. Warp (Version 6.2) provided an important feature, the ability to turn off all optimization. Optimization has the potential to taint the design comparisons by altering the implementation equations to match a logic layout suitable for the chip. Cypress produces a line of CPLDs that were ideal for this project. FPGAs could not be used because their implementation methods would not provide an exact match to the desired logic equations, a requirement for adequate comparison of the adder designs. Xilinx was also investigated, but their VHDL design tool did not allow disabling optimization, and as such, simulations produced addition times completely mismatching theory.

Although either Verilog or VHDL could have been used for development, VHDL was selected. Each design was implemented in its own VHDL file, with the bit width a variable. The designs were compiled into a library that could then be used by any VHDL file. A single "master file" was created for the project which allowed a conditional compile to select the current design, bit width, and any specific implementation constraints.

The specific chip selected was a Cypress C37128P84-100JC, a CPLD with roughly 64 pins available for input and output. This chip allows a maximum clocked frequency of 100MHz and contains 450 macrocells. However, only the carry-feedback design was clocked, meaning the other designs could execute as fast as the logic gates would run internally on the chip.

Because of the high integration among the design, simulation, and implementation of the design, all of the stages of development were performed simultaneously. This resulted in some suboptimal implementation constraints. However, these constraints were applied to all designs, affecting them all equally. One of these constraints was fixing the pin locations. This took away Warp's ability to assign the inputs and output to the pins that were most optimal. Further investigation showed that for all cases but one, the affects of this were negligible. Nevertheless, the important discovery was finding an adequate placement of the clock signal. Moving the clock from one pin to another nearly doubled the speed of the carry-feedback design. The elimination of path length delays caused by the poor choice of clock placement is likely the cause for such a dramatic improvement.

To ensure the highest performance, optimization from Warp was attempted on each adder to determine if the optimizer could lay out the implementation in a better method while retaining the implementation method. The result of this found that on all adders but the carry-feedback, optimization resulted in the tool modifying the adder equations, which invalidated the results for that design. Therefore, the optimizer was disabled for all designs except the carry-feedback. The resulting change provided roughly a 45% increase in speed for the carry-feedback design, a worthy optimization. An examination of the report file produced an explanation for this dramatic improvement. An intermediate signal had been introduced in the un-optimized implementation for latching the results, which added several gate delays to each iteration of the feedback loop. The optimization removed this intermediate signal and sped up the entire loop.

All of the adders except the carry-feedback incorporate a clock to activate the input signals and start the addition. This provided a signal for synchronization at the tradeoff of a constant delay added to each adder. The carry-feedback adder provides an additional "load" signal which is used to initiate addition. However, its design still requires the use of a clock for timing the sequences. With the improvements previously mentioned, the clock signal was 100MHz, the chip's maximum speed. However, this does not necessarily mean that 100MHz was the carry-feedback's maximum speed. Chip manufacturers are required to be conservative on the speed ratings for their chips, and the underlying silicon may have been capable of running much more quickly.

CHAPTER 5

SIMULATION

Design Simulation

Each design was compiled into a file suitable for simulation with the integrated ActiveSim tool. During this process, it was determined that the ActiveSim tool is a logic simulator, not a transistor simulator, meaning that gate delays are treated as exact, and unstable electrical conditions are not always detected. This was discovered when the initial carry-feedback design (which was not clocked) was implemented and worked in simulation but not when implemented. The simulator incorrectly showed that the carry-feedback design would work correctly in all scenarios, when during implementation it was proven that only a very few select cases would work correctly. Figure 6 shows a screen capture of the simulator showing a case that appears to work. Figure 7 shows the same case failing when implemented. A simple redesign of the carry-feedback adder using latches corrected the problem at the cost of a slight decrease in speed and increase in size. Figure 8 shows the corrected simulation.

Several of the designs had multiple configurations available at a specific bit width. Given a choice, speed was weighted as more important than size, except when the size prohibited the design from fitting on the chip. These choices occurred on the carry-lookahead and carry-select adders where the group size is configurable.

For the carry-complete and carry-feedback adders, the input signals for testing were ones that produced $\log_2 N$ carry iterations (an average case): 0xB + 0x3 + 1 for 4bit, 0xBB + 0xFF + 1 for 8bit, and 0xBBBB + 0x7777 + 1 for 16bit. These numbers were chosen arbitrarily from a large sample of available number combinations. For the remaining designs, the worst case was chosen: 0xF + 0x0 + 1 for 4bit, 0xFF + 0x00 + 1 for 8bit, and 0xFFFF + 0x0000 + 1 for 16bit. These numbers were chosen as being the typical "worst case" addition, always causing N carries to ripple.

A limitation of the tool was found when the carry-complete design did not function properly. A glitch on the C_0^1 signal caused the DONE signal to assert prematurely. Cypress is currently investigating this to determine the cause. A rough estimation of performance was still attainable based upon the understanding of the design. At this point, Cypress has still not addressed this issue. Figure 9 shows the failure in simulation.

Table 9 shows the simulated addition times for each of the six adder designs at various bit widths. As expected, most of the designs performed similarly; the addition times increasing logarithmically proportional to the bit width. One exception to this of course was the carry-ripple design, whose time increases linearly in proportion to the bit width. The other exception was the carry-select design, whose addition time remains fixed at the cost of exponentially increasing size.

Table 9

Simulated Addition Times of the Six Adders

Type	*4bit*	*8bit*	*16bit*
Carry-Ripple	50ns	88ns	164ns
Carry-Completion	31ns	40.5ns	50ns
Carry-Lookahead	40.5ns	40.5ns[1]	59.5ns[2]
Carry-Select	50ns	50ns[3]	50ns[4*]
Pyramid	40.5ns	50ns	59.5ns
Carry-Feedback	31ns	41ns	51ns

[1]A group size of 8bits was used

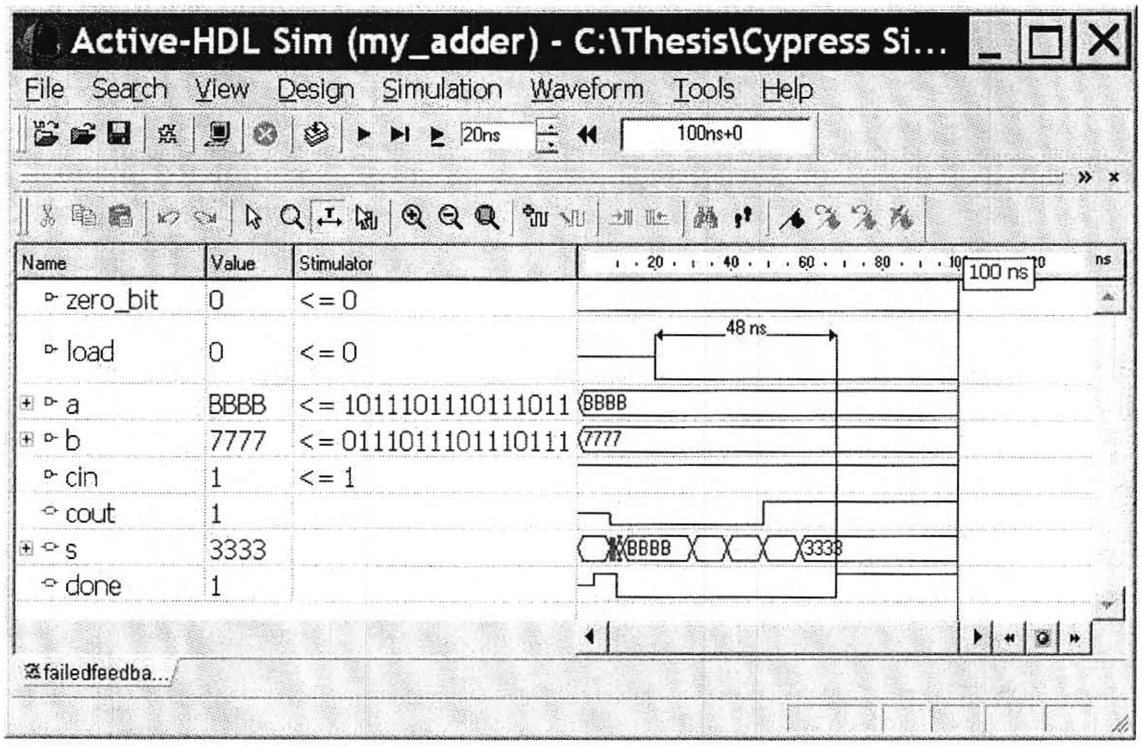
[2]A group size of 4bits was used

[3]A group size of 4bits was used

[4]A group size of 4bits was used

[*]Estimated because the design would not fit

As noted in the diagram, it was determined that a 16bit carry-select adder of any grouping size was too large to fit onto the Cypress chip. The macrocells required were very near the maximum allowed, so an individual experiment was performed. This validated that as a standalone design where Warp assigns the pins, the design would indeed fit on the chip, completely consuming its resources. The results from those simulations were used to fill in that entry in the table.


Active-HDL Sim (my_adder) - C:\Thesis\Cypress Si...
File Search View Design Simulation Waveform Tools Help
20ns
100ns+0
Name
Value
Stimulator
20
40
60
80
100 ns
ns
48 ns
BBBB
7777
BBBB
3333
failedfeedba...

Name	Value	Stimulator
zero_bit	0	<= 0
load	0	<= 0
a	BBBB	<= 1011101110111011
b	7777	<= 0111011101110111
cin	1	<= 1
cout	1	
s	3333	
done	1	

Figure 6. Simulator Output of Flawed Carry-Feedback Adder.

The outputs of the simulator illustrate a successful implementation, showing the output correctly iterate to the sum.

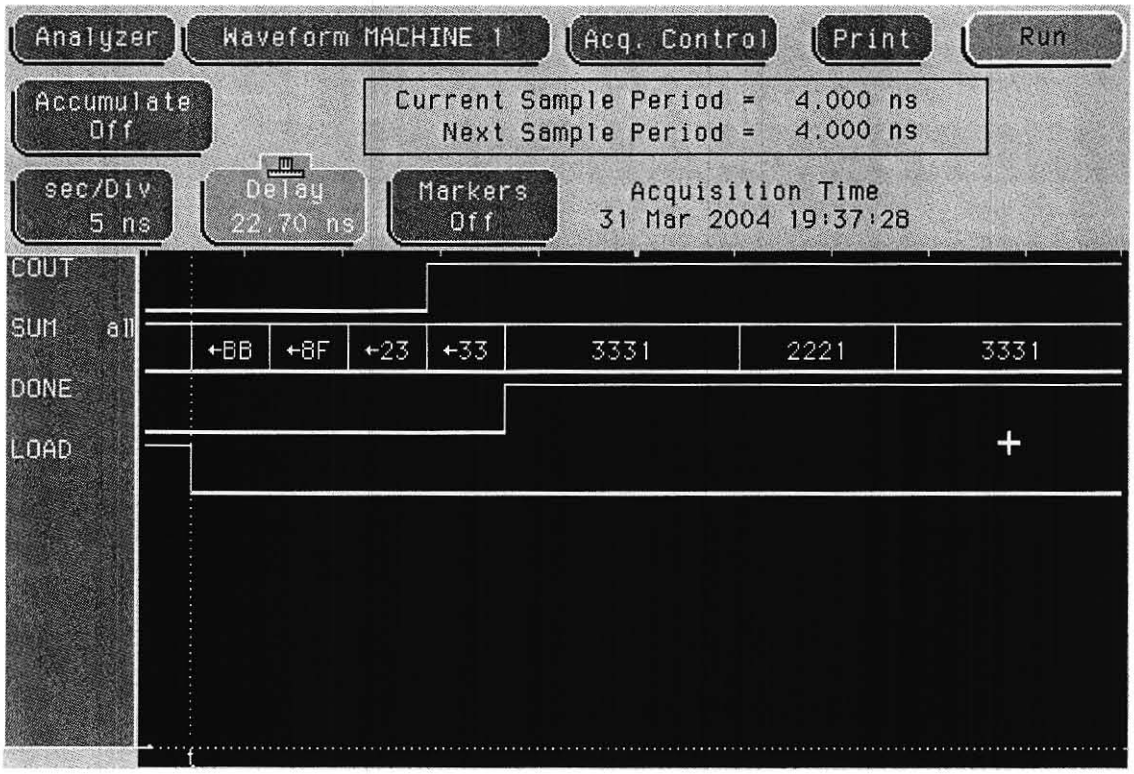
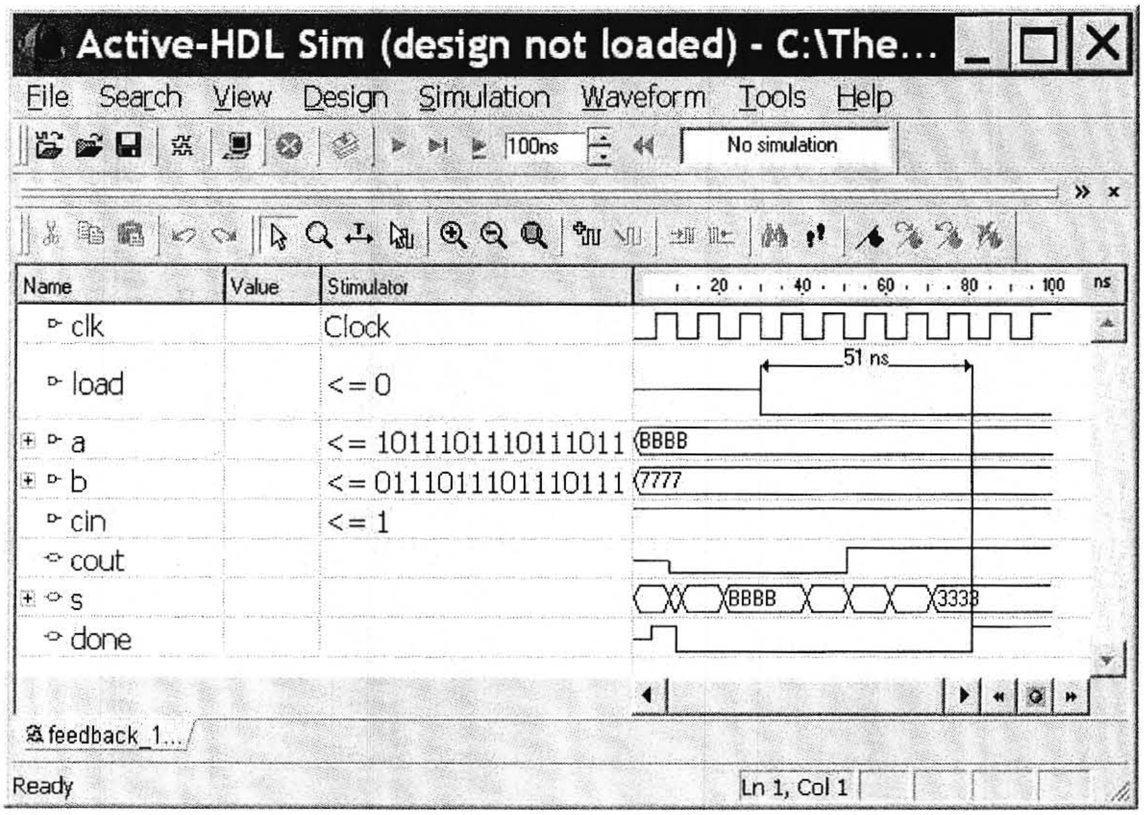


Figure 7. Logic Analyzer Screen Capture of Flawed Carry-Feedback Adder.

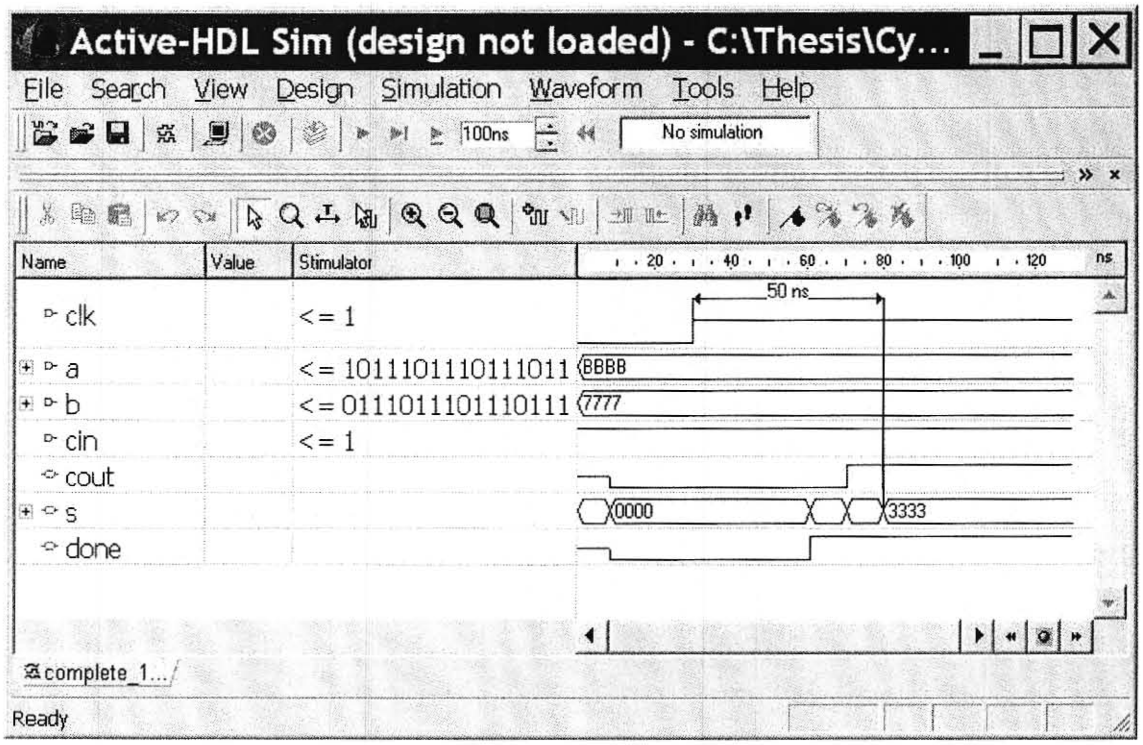
The simulated scenario from Figure 6 is applied in the implementation, and fails because of race conditions inside the initial flawed carry-feedback adder.



Name	Value	Stimulator
clk		Clock
load		<= 0
a		<= 1011101110111011
b		<= 0111011101110111
cin		<= 1
cout		
s		
done		

Figure 8. Simulator Output of Corrected Carry-Feedback Adder.

The corrections are applied to the flawed carry-feedback adder and simulated.


Active-HDL Sim (design not loaded) - C:\Thesis\Cy...
File Search View Design Simulation Waveform Tools Help
100ns
No simulation
Name
Value
Stimulator
20 40 60 80 100 120 ns
50 ns
BBBB
7777
0000
3333
complete_1...
Ready

Name	Value	Stimulator
clk		<= 1
a		<= 1011101110111011
b		<= 0111011101110111
cin		<= 1
cout		
s		
done		

Figure 9. Simulator Output of Flawed Carry-Complete.

The DONE signal incorrectly asserts before addition completes.

CHAPTER 6

TESTING AND VERIFICATION

Implementation of the Design on Programmable Logic

To validate the simulation results, the designs were programmed onto a test board containing the Cypress chip used for simulation. The test board was placed on a breadboard, and the signals wired to a bus, which was connected to an HP 1660A 250MHz logic analyzer. Because of the low sample frequency of the logic analyzer, the maximum accuracy attainable for any measure was +/- 4ns. An HP 8656B Signal Generator connected to a bias tee was used to generate all clock signals for the project. For all the unclocked designs, a low frequency of 1 MHz was used to trigger the addition. For the carry-feedback design, a full 100 MHz clock with 60/40 duty cycle was generated.

Errors similar to those seen during simulation occurred on the implementation of the carry-complete adder, as seen in Figure 10. The output of the corrected carry-feedback adder is shown in Figure 11. The screen capture from the logic analyzer almost exactly matches that from the simulation, affirming Cypress' statement of the simulator being the most exact on clocked designs.

Several measurements of each addition time were taken, and the average value was selected to try and filter out the logic analyzer error. Table 10 displays the average actual addition times for each of the six adder designs at various bit widths:

Table 10

Implemented Addition Times of the Six Adders

Type	*4bit*	*8bit*	*16bit*
Carry-Ripple	16ns	32ns	56ns
Carry-Completion	12ns	16ns	20ns
Carry-Lookahead	12ns	16ns	32ns
Carry-Select	12ns	12ns	12ns*
Pyramid	12ns	16ns	20ns
Carry-Feedback	24ns	28ns	40ns

*Estimated because the design would not fit

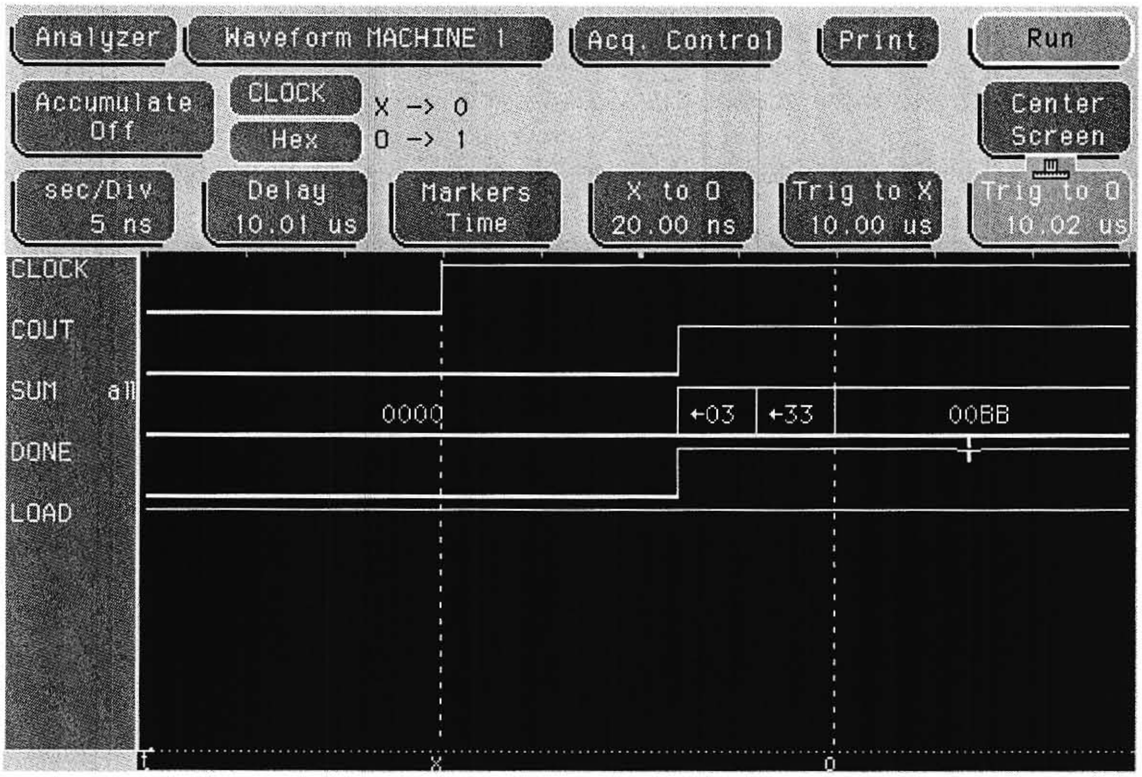


Figure 10. Logic Analyzer Screen Capture of Flawed Carry-Complete Adder.

The DONE signal incorrectly asserts before addition is complete.

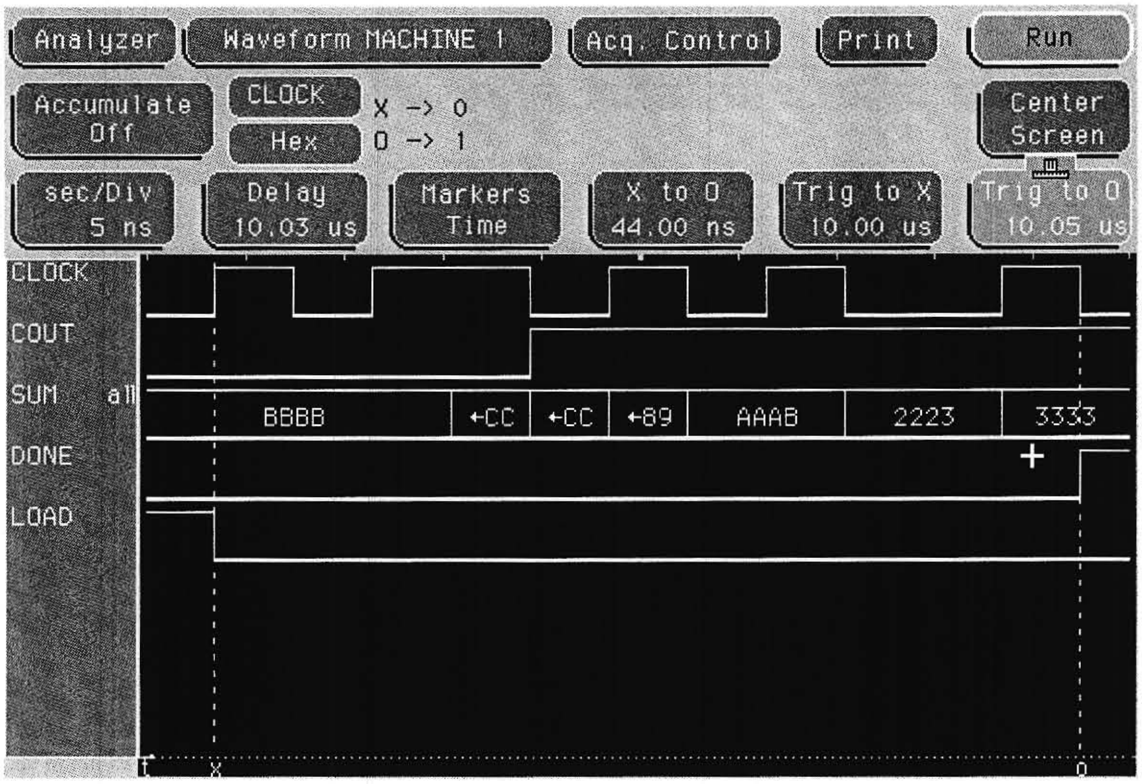


Figure 11. Logic Analyzer Screen Capture of Corrected Carry-Feedback Adder. Shown is an average case addition for a 16 bit addition where 4 rounds of carries ripple.

The first observation that can be made is that the actual addition times for all of the unclocked designs are less than half what was determined through simulation. Cypress was contacted to investigate this phenomenon, and informed that a possible cause could be the inaccuracy of the logic analyzer. However, it was pointed out that for some of the designs, the time variance between simulated and actual approaches 100ns. Cypress indicated that the ActiveSim simulator software only performs logic simulations, not true transistor level simulations, and therefore can produce very inaccurate times for completely asynchronous circuits. Cypress' support team also indicated that clocked designs, such as the carry-feedback, would produce simulations times very close to actual implementation. For comparison purposes, however, since the times are compared to each other only in their own context (simulated-to-simulated or actual-to-actual), agreement between the simulator and actual data is not a requirement.

The second metric that was investigated was adder size. An exact measurement of gate count from the Warp compiler for each design was not attainable. However, an exact count of the macrocells used was attained, and provides a useful metric for comparing the size of the adder designs at various bit widths. Inaccuracies in using macrocell counts come from the design of CPLDs. Macrocells have a fixed logic layout, and each design will use a different fraction of a macrocell. Some designs will utilize the logic inside a macrocell more favorably than others. Since only rough numbers for comparative purposes were needed, the use of a macrocell as a measurement should suffice.

It should be noted that to provide a constant pin interface to the logic analyzer, the widths of all input and output vectors were fixed at 16. As such, the smaller vector widths suffer a macrocell penalty for the additional logic to support the unused signals. However, this number is constant and applies to all adders in the same way. Table 11 displays the actual macrocell usage for each of the six adder designs at various bit widths:

Table 11

Implemented Macrocell Usage of the Six Adders

Type	*4bit*	*8bit*	*16bit*
Carry-Ripple	65	106	169
Carry-Completion	95	169	292
Carry-Lookahead	82	149	296
Carry-Select	92	158	450*
Pyramid	76	138	267
Carry-Feedback	69	101	161

*Estimated because the design would not fit

As expected, the size of the adders increased similarly; the macrocell usage increased exponentially proportional to the bit width. The exceptions to this were the carry-ripple and carry-feedback designs, whose sizes increased linearly in proportion to the bit width. However, further investigation showed this comparison to be unfair. Each macrocell includes a flip-flop, a necessary piece of the carry-feedback design, but unimportant to the other designs. Therefore, the size of the carry-feedback design is incorrectly seen as lower than it actually is. This should be noted in all comparisons between it and the other designs.

Table 12 displays the calculated performance from the six implemented adders. An important note before discussing the number in the table is the small vector widths used in implementation prevents a thorough analysis of performance. Nonetheless, an important observation can be made. The limitations mentioned previously, namely the routing delay variations on programmable logic and the inaccurate gate count from macrocell usage make it difficult to determine an accurate comparison of theoretical and implemented performance. These problems would be fixed by a more powerful VHDL compiler that allowed complete control over optimization that generated a report file that gave more detailed information of macrocell usage.

Table 12

Implemented Performance of the Six Adders (lower is better)

Type	*4bit*	*8bit*	*16bit*
Carry-Ripple	1040	3392	9464
Carry-Completion	1140	2704	5840
Carry-Lookahead	984	2384	9472
Carry-Select	1104	1896	5400
Pyramid	912	2208	5340
Carry-Feedback	1656	2828	6440

CHAPTER 7

ANALYSIS

Analysis of Theoretical, Simulated, and Tested Data

Analysis of the data produced during all stages of evaluation reveal some useful numbers for comparison of the carry-feedback adder to the other adders. From the theoretical data (using a 64-bit vector width as the comparative point), the carry-feedback adder is nearly six times as fast as the carry-ripple adder while only being 33% larger. The carry-feedback adder is 10% smaller than the carry-complete adder for a 25% increase in speed. The carry-lookahead adder stands out as the clear victor, its size being 5% less than the carry-feedback adder and its speed being 25% greater. It is important to note that both size metrics ignore fan-in, an important factor for each of them. The carry-feedback has a single N-bit wide gate, and no others wider than three inputs. On the other hand, the carry-lookahead with a block-size of just 4bits has numerous gates with five or more inputs and many more above three.

The data is most clear from Table 8, the theoretical performance. By sorting from low to high, a quick picture of the ranking of each adder in comparison to the others is seen. The carry-lookahead is first, second is the carry-completion, followed closely by the carry-feedback (SRAM). A large gap follows, and next are the pyramid and carry-feedback (D flip-flop) adders followed by the carry-ripple in a distance sixth place. In last place is the carry-select adder, which in the selected implementation becomes too large to be practically feasible.

Investigating the tested data reveals the penalties of using flip-flops to latch data in the carry-feedback adder. This coupled with the chip's maximum speed bottlenecking the carry-feedback's iteration time unfairly decreased its speed compared to the other adders able to run at the full speed of the silicon. The tested data (using a 16-bit vector width as the comparative point) shows the carry-feedback adder to be just under 30% faster than the carry-ripple while being slightly smaller, a phenomenon that can be attributed to the enabling of optimization for the carry-feedback adder. The wider gates required for many of the designs becomes apparent in the disagreement between macrocell usage and theoretical size. In implementation, the carry-lookahead adder is the second largest adder, where in theory it was the second smallest. This is not surprising given the knowledge of the surface area cost of wider gates.

CHAPTER 8

CONCLUSIONS

Conclusions Drawn From the Design, Simulation, Testing and Verification

Complex logic chips, such as system-on-chip (SoC) designs, are finding the benefits of being asynchronous. Asynchronous designs promise the possibility of lower power and easier design (Cole, 2003). Proper designs to take advantage of well-designed asynchronous chips can even offer increased speed over their synchronous counterparts, although this is currently an area of debate (Donovan, 2003). The complexity tradeoff comes from eliminating the globally distributed chip clock, and replacing it with individual units that must negotiate to each other through handshake signals (Cravotta, 2004).

The innovative approach to addition executed in the carry-feedback design, coupled with its features, reasonable gate count, and high performance in the average addition implies it is a valid new design that should be further investigated. Despite its requirement for a clock, the design's ability to run on a clock independent of its counterparts keeps the design asynchronous in a "complete chip" picture (Cole, 2003). Additionally, the carry-feedback design latches the sum automatically when the addition is complete. Since the Arithmetic Logic Unit (ALU) of most microprocessors must latch the sum, the additional size from adding the clock to fix the flaw adds little to the complete chip picture. The market for such a design has a potential to be very large as binary addition is a fundamental piece of every microprocessor (Patterson & Hennessy, 1994).

The technology available through programmable logic is invaluable for rapid prototyping and comparison of logic designs, allowing time to be devoted to the more important aspects of development, design and analysis. Further improvements in the programmable logic arena will continue to increase productivity, allowing larger and larger designs to be incorporated more quickly and easily. Since the research was started, new chip features are already available, such as the addition of SRAM. Each feature increases the programmability and use of these chips for rapid prototyping of designs. Also, newer CPLDs provide additional I/O lines, increasing the vector widths that could be tested to realms more in line with modern microprocessors.

While the performance hit of adding a clock to the carry-feedback design is minimal when using SRAM, it is the author's belief that further research could once again lead to a design requiring no clock. Gate delay balancing is a common task performed by microprocessor designers, and proper balancing of the feedback signals could completely eliminate the need for the clock. Since each gate delay in the loop is multiplied by the number of iterations, eliminating even one gate delay results in tremendous speed improvements, although the impact to size is minimal. The nature of the design automatically provides latching of the sum data making the SRAM redundant.

CHAPTER 9

FUTURE WORK

Further Research Opportunities and Areas of Interest

This research opened several opportunities for future research. For the carry-feedback adder, a research opportunity is finding new technologies that allowed simulation and implementation using SRAM to validate the theory. Additionally, a valuable effort is further investigation into the design to determine if the latches could be eliminated completely, perhaps by gate delay balancing or implementation of the algorithm in a different way.

For the research in general, using more powerful software and hardware tools that allow more control over the routing as well as more information into the sub-macrocell usage would provide far more accurate implementation results. Direct implementation onto custom silicon obviously would provide the highest level of validation of theory with implementation.

Lastly, some interesting future research is applying the innovative concepts that led to the development of the carry-feedback to other parts of the microprocessor. If done properly, a microprocessor could be designed that is completely asynchronous.

REFERENCES

Omondi, A. (1994). *Computer Arithmetic Systems*. Cambridge, Great Britain: The University Press.

Koren, I. (1993). *Computer Arithmetic Algorithms*. Englewood Cliffs, New Jersey: Prentice Hall

Wakerly, J. (2000). *Digital Design Principles and Practices*. Englewood Cliffs, New Jersey: Prentice Hall

Patterson, D. & Hennessy J. (1994). *Computer Organization & Design: The Hardware / Software Interface*. San Francisco, California: Morgan Kaufmann Publishers, Inc.

Skahill, K. (1996). *VHDL for Programmable Logic*. Menlo Park, California: Addison-Wesley Publishing Company, Inc.

Donovan, J. (2003). 'Clockless' Chip Risk May Pay Off. *EETimes* 1281, 41

Cole, B. (2003). Clocked or Clockless? Time to Readjust the 'Timing' Rules. *EETimes* 1273, 57

Cravotta, R. (2004). Squeeze Play: Wring the Power Out of Your Designs. *EDN* 49(4), 36-46.

APPENDIX A: CODE LISTINGS

```
-- File    : my_adder.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file is the main function file for the adder implementations.  To
--           vary the implementation or vector width, the 'adder_type' option is
--           changed.  Adjusting the vector width does not adjust the actual width
--           of the output, only the width of the adder generated.

-- libraries
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- include the definition file of adder types
use work.adder_package.all;

-- adder declaration
entity my_adder is
    -- configurable options
    generic
    (
        width : integer := 16;
        size : integer := 8;
        -- type 0 is ripple
        -- type 1 is lookahead
        -- type 2 is select
        -- type 3 is pyramid
        -- type 4 is complete
        -- type 5 is feedback
        -- type 6 is clocked feedback
        adder_type : integer := 0
    );
    -- port declaration
    port
    (
        a : in std_logic_vector(15 downto 0);  -- addend 1
        b : in std_logic_vector(15 downto 0);  -- addend 2
        cin : in std_logic;                    -- carry input
        cout : inout std_logic;                -- carry output
        s : out std_logic_vector(15 downto 0); -- sum
        clk : in std_logic;                    -- multipurpose clock
        done : out std_logic;                  -- completion indicator
        zero_bit : in std_logic;               -- dummy signal to prevent optimization
        load : in std_logic                    -- loads the signals
    );
    -- pin assignments
    attribute pin_numbers of my_adder : entity is
        "a(0):3 " &
        "a(1):4 " &
        "a(2):5 " &
        "a(3):6 " &
        "a(4):7 " &
        "a(5):8 " &
        "a(6):9 " &
        "a(7):10 " &
        "a(8):12 " &
        "a(9):13 " &
        "a(10):15 " &
        "a(11):16 " &
        "a(12):17 " &
        "a(13):18 " &
        "a(14):19 " &
        "a(15):20 " &
        "b(0):61 " &
        "b(1):66 " &
        "b(2):67 " &
        "b(3):68 " &
        "b(4):69 " &
        "b(5):70 " &
        "b(6):71 " &
        "b(7):73 " &
        "b(8):75 " &
        "b(9):76 " &
        "b(10):77 " &
        "b(11):78 " &
        "b(12):79 " &
        "b(13):80 " &
        "b(14):81 " &
        "b(15):82 " &
        "s(0):31 " &
        "s(1):33 " &
        "s(2):34 " &
        "s(3):36 " &
        "s(4):37 " &
        "s(5):38 " &
        "s(6):39 " &
        "s(7):40 " &
        "s(8):45 " &
        "s(9):46 " &
        "s(10):47 " &
        "s(11):48 " &
        "s(12):49 " &
        "s(13):50 " &
        "s(14):52 " &
        "s(15):54 " &
        "clk:23 " &
        "cin:24 " &
        "done:30 " &
        "zero_bit:60 " &
        "load:59 " &
        "cout:55 ";
end my_adder;

architecture Behavioral of my_adder is

    signal x : std_logic_vector((width - 1) downto 0);
    signal y : std_logic_vector((width - 1) downto 0);
    signal c : std_logic;

begin

    process (clk, a, b, cin)
    begin
        -- for all adders but carry-feedback, activate addends
        -- only when clock is high
        if (adder_type < 5) then
            for i in 0 to (width - 1) loop
                x(i) <= a(i) and clk;
                y(i) <= b(i) and clk;
            end loop;
            c <= cin and clk;
        end if;
        -- set unused sum outputs to '0'
        for i in width to 15 loop
            s(i) <= '0';
        end loop;
    end process;
    -- carry-ripple definition
    ripple : if (adder_type = 0) generate
        u1 : carry_ripple generic map(width) port map(x, y, c, cout, s((width-1) downto
0));
        u2 : done_signal port map(done);
    end generate ripple;
    -- carry-lookahead definition
    lookahead : if (adder_type = 1) generate
        u1 : carry_lookahead generic map(width, size) port map(x, y, c, cout, s((width-1)
downto 0));
        u2 : done_signal port map(done);
    end generate lookahead;
    -- carry-select definition
    cselect : if (adder_type = 2) generate
        u1 : carry_select generic map(width, size) port map(x, y, c, cout, s((width-1)
downto 0));
        u2 : done_signal port map(done);
    end generate cselect;
    -- pyramid definition
    pyramid : if (adder_type = 3) generate
        u1 : carry_pyramid generic map(width) port map(x, y, c, cout, s((width-1) downto
0));
        u2 : done_signal port map(done);
    end generate pyramid;
    -- carry-complete definition
    complete : if (adder_type = 4) generate
        u1 : carry_complete generic map(width) port map(x, y, c, clk, cout, s((width-1)
downto 0), done);
    end generate complete;
    -- flawed unclocked carry-feedback definnition
    feedback : if (adder_type = 5) generate
        u1 : carry_feedback generic map(width) port map(a((width - 1) downto 0), b((width
- 1) downto 0), cin, load, cout, s((width-1) downto 0), done, zero_bit);
    end generate feedback;
    -- correct clocked carry-feedback definition
    feedback2 : if (adder_type = 6) generate
        u1 : carry_feedback2 generic map(width) port map(a((width - 1) downto 0),
b((width - 1) downto 0), cin, load, cout, s((width-1) downto 0), done, clk);
    end generate feedback2;
end Behavioral;
```

```
-- File    : adder_package.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This is the file that defines the inputs and outputs of
--           all the adders.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

package adder_package is
    component carry_ripple
        generic
        (
            width : integer
        );
        port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin  : in std_logic;
            cout : out std_logic;
            s    : out std_logic_vector((width - 1) downto 0)
        );
    end component;
    component carry_complete
        generic
        (
            width : integer
        );
        Port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin  : in std_logic;
            load : in std_logic;
            cout : out std_logic;
            s    : out std_logic_vector((width - 1) downto 0);
            done : out std_logic
        );
    end component;
    component carry_feedback
        generic
        (
            width : integer
        );
        port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin : in std_logic;
            load : in std_logic;
            cout : out std_logic;
            s    : out std_logic_vector((width - 1) downto 0);
            done : out std_logic;
            zero_bit : in std_logic
        );
    end component;
    component carry_feedback2
        generic
        (
            width : integer
        );
        port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin : in std_logic;
            load : in std_logic;
            cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0);
        done : out std_logic;
        clk : in std_logic
    );
end component;
component carry_lookahead
    generic
    (
        width : integer;
        block_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select1
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select2
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select3
generic
(
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select4
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select5
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select6
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select7
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select8
    generic
```

```
        (
            width : integer;
            minimum_size : integer
        );
        port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin  : in std_logic;
            cout : out std_logic;
            s    : out std_logic_vector((width - 1) downto 0)
        );
    end component;
    component carry_select9
        generic
        (
            width : integer;
            minimum_size : integer
        );
        port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin  : in std_logic;
            cout : out std_logic;
            s    : out std_logic_vector((width - 1) downto 0)
    );
    end component;
    component carry_select10
        generic
        (
            width : integer;
            minimum_size : integer
        );
        port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin  : in std_logic;
            cout : out std_logic;
            s    : out std_logic_vector((width - 1) downto 0)
        );
    end component;
    component carry_select11
        generic
        (
            width : integer;
            minimum_size : integer
        );
        port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin  : in std_logic;
            cout : out std_logic;
            s    : out std_logic_vector((width - 1) downto 0)
        );
    end component;
    component carry_select12
        generic
        (
            width : integer;
            minimum_size : integer
        );
        port
        (
            a : in std_logic_vector((width - 1) downto 0);
            b : in std_logic_vector((width - 1) downto 0);
            cin  : in std_logic;
            cout : out std_logic;
            s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select13
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_select14
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component carry_pyramid
    generic
    (
        width : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0)
    );
end component;
component pg_carry
    generic
    (
        bit_number : integer
    );
    port
    (
        p : in std_logic_vector(bit_number downto 0);
        g : in std_logic_vector(bit_number downto 0);
        cin : in std_logic;
        cout : inout std_logic
    );
end component;
component superblock
    generic
    (
        width : integer;
        current_width : integer;
        block_size : integer
    );
    port
    (
        p : in std_logic_vector((current_width*block_size - 1) downto 0);
        g : in std_logic_vector((current_width*block_size - 1) downto 0);
        c : inout std_logic_vector(width downto 0)
    );
end component;
component superblock2
    generic
    (
        width : integer;
        current_width : integer;
        block_size : integer
    );
    port
    (
        p : in std_logic_vector((current_width*block_size - 1) downto 0);
        g : in std_logic_vector((current_width*block_size - 1) downto 0);
        c : inout std_logic_vector(width downto 0)
    );
end component;
component superblock3
    generic
    (
        width : integer;
        current_width : integer;
        block_size : integer
    );
    port
    (
        p : in std_logic_vector((current_width*block_size - 1) downto 0);
        g : in std_logic_vector((current_width*block_size - 1) downto 0);
        c : inout std_logic_vector(width downto 0)
    );
end component;
component pyramid_block
    generic
    (
        width : integer;
        block_size : integer
    );
    port
    (
        sin : in std_logic_vector((width - 1) downto 0);
        cin : in std_logic_vector((width - 1) downto 0);
        s : out std_logic_vector((width - 1) downto 0);
        cout : out std_logic
    );
end component;
component pyramid_block2
    generic
    (
        width : integer;
        block_size : integer
    );
    port
    (
        sin : in std_logic_vector((width - 1) downto 0);
        cin : in std_logic_vector((width - 1) downto 0);
        s : out std_logic_vector((width - 1) downto 0);
        cout : out std_logic
    );
end component;
component pyramid_block3
    generic
    (
        width : integer;
        block_size : integer
    );
    port
    (
        sin : in std_logic_vector((width - 1) downto 0);
        cin : in std_logic_vector((width - 1) downto 0);
        s : out std_logic_vector((width - 1) downto 0);
        cout : out std_logic
    );
    end component;
    component pyramid_block4
        generic
        (
            width : integer;
            block_size : integer
        );
        port
        (
            sin : in std_logic_vector((width - 1) downto 0);
            cin : in std_logic_vector((width - 1) downto 0);
            s : out std_logic_vector((width - 1) downto 0);
            cout : out std_logic
        );
    end component;
    component pyramid_block5
        generic
        (
            width : integer;
            block_size : integer
        );
        port
        (
            sin : in std_logic_vector((width - 1) downto 0);
            cin : in std_logic_vector((width - 1) downto 0);
            s : out std_logic_vector((width - 1) downto 0);
            cout : out std_logic
        );
    end component;
    component pyramid_block6
        generic
        (
            width : integer;
            block_size : integer
        );
        port
        (
            sin : in std_logic_vector((width - 1) downto 0);
            cin : in std_logic_vector((width - 1) downto 0);
            s : out std_logic_vector((width - 1) downto 0);
            cout : out std_logic
        );
    end component;
    component done_signal
        port
        (
            done : out std_logic
        );
    end component;
end adder_package;
```

```
-- File    : carry_ripple.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file defines the carry-ripple adder.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity carry_ripple is
   generic
   (
      width : integer := 16
   );
   port
   (
      a : in std_logic_vector((width - 1) downto 0);
      b : in std_logic_vector((width - 1) downto 0);
      cin : in std_logic;
      cout : out std_logic;
      s : out std_logic_vector((width - 1) downto 0)
   );
end carry_ripple;

architecture Behavioral of carry_ripple is

   signal c : std_logic_vector(width downto 0);

begin

   c(0) <= cin;
   process (a, b, cin, c)
   begin
      for i in 0 to (width - 1) loop
         s(i) <= a(i) xor b(i) xor c(i);
         c(i+1) <= (a(i) and b(i)) or (a(i) and c(i)) or (b(i) and c(i));
      end loop;
   end process;
   cout <= c(width);

end Behavioral;
```

```
-- File    : carry_lookahead.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file defines the carry-lookahead adder.  It includes
--           the following files: pg_carry and superblock.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.adder_package.all;

entity carry_lookahead is
    generic
    (
        width : integer;
        block_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin : in std_logic;
        cout : out std_logic;
        s : out std_logic_vector((width - 1) downto 0)
    );
end carry_lookahead;

architecture Behavioral of carry_lookahead is

    signal c : std_logic_vector(width downto 0);
    signal p : std_logic_vector((width - 1) downto 0);
    signal g : std_logic_vector((width - 1) downto 0);

begin

    c(0) <= cin;
    process (a, b, cin, c)
    begin
        for i in 0 to (width-1) loop
            s(i) <= a(i) xor b(i) xor c(i);
            p(i) <= a(i) or b(i);
            g(i) <= a(i) and b(i);
        end loop;
    end process;
    cout <= c(width);

    F : for i in 0 to (width-1) generate
        C : if (((i+1) mod block_size) /= 0) or (width = block_size)) generate
            carry : pg_carry  generic map (i mod block_size)
                                    port map(p(i downto (i-(i mod block_size))),
                                                g(i downto (i-(i mod block_size))),
                                                c((i/block_size)*block_size),
                                                c(i+1)
                                                );
        end generate C;
    end generate F;
    PG : if (width /= block_size) generate
        block_carry : superblock generic map (width, width/block_size, block_size)
                                     port map (p, g, c);
    end generate PG;

end Behavioral;
```

```
-- File    : pg_carry.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file is used to calculate the pg carries for the
--           carry-lookahead adder.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pg_carry is
   generic
   (
      bit_number : integer := 0
   );
   port
   (
      p : in std_logic_vector(bit_number downto 0);
      g : in std_logic_vector(bit_number downto 0);
      cin : in std_logic;
      cout : inout std_logic
   );
end pg_carry;

architecture Behavioral of pg_carry is

begin

    process (p, g, cin)

        variable and_level, or_level : std_logic;

    begin
        or_level := g(bit_number);
        for j in bit_number downto 0 loop
            and_level := '1';
            for k in bit_number downto j loop
                and_level := and_level and p(k);
            end loop;
            if (j = 0) then
                and_level := and_level and cin;
            else
                and_level := and_level and g(j - 1);
            end if;
            or_level := or_level or and_level;
        end loop;
        cout <= or_level;
    end process;

end Behavioral;
```

```
-- File    : superblock.vhd, superblock2.vhd, superblock3.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : These files are identical.  Three copies of the same file
--           had to be used because the Warp compiler could not handle
--           recursive generations of an object.  This file combines the
--           pg carries into a superblock.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.adder_package.all;

entity superblock is
    generic
    (
        width : integer;
        current_width : integer;
        block_size : integer
    );
    port
    (
        p : in std_logic_vector((current_width*block_size - 1) downto 0);
        g : in std_logic_vector((current_width*block_size - 1) downto 0);
        c : inout std_logic_vector(width downto 0)
    );
end superblock;

architecture Behavioral of superblock is

    signal bp : std_logic_vector((current_width - 1) downto 0);
    signal bg : std_logic_vector((current_width - 1) downto 0);

begin

    process (p, g, bp)

        variable and_level : std_logic;

    begin
        for i in 0 to (current_width-1) loop
            and_level := '1';
            for j in 0 to (block_size-1) loop
                and_level := and_level and p(i*block_size+j);
            end loop;
            bp(i) <= and_level;
        end loop;
    end process;

    F : for i in 0 to (current_width-1) generate
        BG : pg_carry generic map(block_size-1)
                          port map (p(((i+1)*block_size-1) downto (i*block_size)),
                                    g(((i+1)*block_size-1) downto (i*block_size)),
                                    '0',
                                    bg(i)
                                    );
        C : if ((((i+1) mod block_size) /= 0) or (current_width = block_size)) generate
            carry : pg_carry generic map(i mod block_size)
                                port map(bp(i downto (i-(i mod block_size))),
                                         bg(i downto (i-(i mod block_size))),
                                         c(0),
                                         c((i+1)*(width/current_width))
                                         );
        end generate C;
    end generate F;
    PG : if (current_width /= block_size) generate
        block_carry : superblock2 generic map (width, current_width/block_size,
block_size)
                                    port map (bp, bg, c);
    end generate PG;

end Behavioral;
```

```
-- File    : carry_select.vhd, carry_select1.vhd through carry_select14.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : These files are identical copies used to define the carry_select
--           adder.  Multiple copies had to be created to overcome a deficiency
--           in the Warp compiler that disallowed recursive generations of an
--           object.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.adder_package.all;

entity carry_select is
    generic
    (
        width : integer;
        minimum_size : integer
    );
    port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin : in std_logic;
        cout : out std_logic;
        s : out std_logic_vector((width - 1) downto 0)
    );
end carry_select;

architecture Behavioral of carry_select is

    signal upper_select : std_logic;
    signal c_1 : std_logic;
    signal c_0 : std_logic;
    signal left_1 : std_logic_vector((width - minimum_size - 1) downto 0);
    signal left_0 : std_logic_vector((width - minimum_size - 1) downto 0);
    signal right  : std_logic_vector((minimum_size - 1) downto 0);

begin

    -- generate bits minimum_size downto 0 with carry_lookahead
    R : carry_lookahead generic map(minimum_size, minimum_size)
                        port map(a((minimum_size - 1) downto 0),
                                 b((minimum_size - 1) downto 0),
                                 cin,
                                 upper_select,
                                 right);
    -- If more bits left and not on final stage
    LeftBig: if ((width - minimum_size) > minimum_size) generate
        -- generate bits width downto minimum_size with carry_select with carry_in of 0
        L0 : carry_select1 generic map(width - minimum_size, minimum_size)
                             port map(a((width-1) downto minimum_size),
                                      b((width-1) downto minimum_size),
                                     '0',
                                     c_0,
                                     left_0);
        -- generate bits width downto minimum_size with carry_select with carry_in of 1
        L1 : carry_select2 generic map(width - minimum_size, minimum_size)
                             port map(a((width-1) downto minimum_size),
                                      b((width-1) downto minimum_size),
                                     '1',
                                     c_1,
                                     left_1);
    end generate LeftBig;

    -- if more bits left and ARE on final stage
    LeftSmall: if (((width - minimum_size) <= minimum_size) and ((width - minimum_size) >
0)) generate
```

```
        -- generate bits width downto minimum_size with carry_lookahead with carry_in of
0
        L2 : carry_lookahead generic map(width - minimum_size, minimum_size)
                             port map(a((width-1) downto minimum_size),
                                       b((width-1) downto minimum_size),
                                      '0',
                                      c_0,
                                      left_0);
        -- generate bits width downto minimum_size with carry_lookahead with carry_in of
1
        L3 : carry_lookahead generic map(width - minimum_size, minimum_size)
                             port map(a((width-1) downto minimum_size),
                                       b((width-1) downto minimum_size),
                                      '1',
                                      c_1,
                                      left_1);
    end generate LeftSmall;

    -- select which carry_select based upon carry_out of the carry_lookahead
    process (a, b, cin, upper_select, left_0, left_1, right, c_0, c_1)
    begin
        if (width > minimum_size) then
            if (upper_select = '0') then
                s <= left_0 & right;
                cout <= c_0;
            else
                s <= left_1 & right;
                cout <= c_1;
            end if;
        end if;
    end process;

end Behavioral;
```

```
-- File    : carry_pyramid.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file defines the pyramid adder.  It includes the following
--           file: pyramid_block.vhd.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.adder_package.all;

entity carry_pyramid is
   generic
   (
      width : integer
   );
   Port
   (
      a : in std_logic_vector((width - 1) downto 0);
      b : in std_logic_vector((width - 1) downto 0);
      cin : in std_logic;
      cout : out std_logic;
      s : out std_logic_vector((width - 1) downto 0)
   );
end carry_pyramid;

architecture Behavioral of carry_pyramid is

    signal s0 : std_logic_vector((width - 1) downto 0);
    signal c0 : std_logic_vector((width - 1) downto 0);

begin

    process (a, b, s0, cin)
    begin
        s0(0) <= a(0) xor b(0) xor cin;
        c0(0) <= (a(0) and b(0)) or (a(0) and cin) or (b(0) and cin);
        for i in 1 to (width - 1) loop
            s0(i) <= a(i) xor b(i);
            c0(i) <= a(i) and b(i);
        end loop;
    end process;

    -- Generate the next level or be done
    PYR: if (width > 2) generate
        PB : pyramid_block generic map (width, 1) port map (s0, c0, s, cout);
    end generate PYR;

end Behavioral;
```

```
-- File    : pyramid_block.vhd, pyramid_block2.vhd through pyramid_block6.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file defines a single level of the pyramid adder.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.adder_package.all;

entity pyramid_block is
    generic
    (
        width : integer;
        block_size : integer
    );
    Port
    (
        sin : in std_logic_vector((width - 1) downto 0);
        cin : in std_logic_vector((width - 1) downto 0);
        s : out std_logic_vector((width - 1) downto 0);
        cout : out std_logic
    );
end pyramid_block;

architecture Behavioral of pyramid_block is

    signal snext : std_logic_vector((width - 1) downto 0);
    signal cnext : std_logic_vector((width - 1) downto 0);

begin

    process (sin, cin)

        variable and_level : std_logic;
        variable temp : integer;

    begin
        if (block_size = width) then
            for i in 0 to (width -1) loop
                s(i) <= sin(i);
            end loop;
            cout <= cin(width - 1);
        else
            for i in 0 to (width - 1) loop
                if (((i+1) mod (block_size*2)) = 0) then
                    temp := (i / block_size) * block_size;
                    and_level := cin(temp - 1);
                    for j in temp to i loop
                        and_level := and_level and sin(j);
                    end loop;
                    cnext(i) <= cin(i) or and_level;
                end if;
                if (((i / block_size) mod 2) = 0) then
                    snext(i) <= sin(i);
                else
                    temp := (i / block_size) * block_size;
                    and_level := cin(temp - 1);
                    if (i > temp) then
                        for j in temp to (i-1) loop
                            and_level := and_level and sin(j);
                        end loop;
                    end if;
                    snext(i) <= sin(i) xor and_level;
                end if;
            end loop;
        end if;
    end process;
    -- Generate the next level or be done
    PYR: if (width >= (block_size * 2)) generate
        PB: pyramid_block2 generic map (width, block_size*2) port map (snext, cnext, s,
cout);
    end generate PYR;

end Behavioral;
```

```
-- File    : carry_complete.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file defines the carry_complete adder.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity carry_complete is
    generic
    (
        width : integer := 16
    );
    Port
    (
        a : in std_logic_vector((width - 1) downto 0);
        b : in std_logic_vector((width - 1) downto 0);
        cin  : in std_logic;
        load : in std_logic;
        cout : out std_logic;
        s    : out std_logic_vector((width - 1) downto 0);
        done : out std_logic
    );
end carry_complete;

architecture Behavioral of carry_complete is

    signal c1 : std_logic_vector(width downto 0);
    signal c0 : std_logic_vector(width downto 0);
    signal ps : std_logic_vector((width - 1) downto 0);
    signal pd : std_logic_vector((width - 1) downto 0);

begin

    process (a, b, cin, load, c1, c0, ps, pd)

        variable tdone : std_logic;

    begin
        c0(0) <= not(cin) and load;
        c1(0) <= cin and load;
        tdone := '1';
        for i in 0 to (width - 1) loop
            ps(i) <= a(i) xor b(i);
            pd(i) <= c0(i+1) or c1(i+1);
--          Apparently, these formulas from Omondi are bad?
--            c0(i+1) <= ((not(a(i)) and not(b(i))) or (ps(i) and not(c0(i)))) and load;
--            c1(i+1) <= ((a(i) and b(i)) or (ps(i) and c1(i))) and load;
            c0(i+1) <= ((not(a(i)) and not(b(i))) or ((not(a(i)) or not(b(i))) and
c0(i))) and load;
            c1(i+1) <= ((a(i) and b(i)) or ((a(i) or b(i)) and c1(i))) and load;
            s(i) <= ps(i) xor c1(i);
            tdone := tdone and pd(i);
        end loop;
        done <= tdone;
        cout <= c1(width);
    end process;

end Behavioral;
```

```
-- File    : carry_feedback.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file defines the flawed, unclocked version of the carry-
--           feedback adder.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity carry_feedback is
   generic
   (
      width : integer := 16
   );
   port
   (
      a : in std_logic_vector((width - 1) downto 0);
      b : in std_logic_vector((width - 1) downto 0);
      cin  : in std_logic;
      load : in std_logic;
      cout : out std_logic;
      s    : out std_logic_vector((width - 1) downto 0);
      done : out std_logic;
      zero_bit : in std_logic
   );
end carry_feedback;

architecture Behavioral of carry_feedback is

   signal x : std_logic_vector((width - 1) downto 0);
   signal y : std_logic_vector((width - 1) downto 0);
   signal xin : std_logic;
   signal yin : std_logic;
   signal yout : std_logic;

begin

   process(a, b, cin, load, xin, yin, x, y, yout, zero_bit)

      variable d_check : std_logic;

   begin
      yin <= ((zero_bit and not(load)) or
              (cin and load));
      xin <= ((xin and not(yin) and not(load)) or
              (not(xin) and yin and not(load)) or
              (cin and load));
      y(0) <= (xin and yin and not(load)) or
              (b(0) and load);
      x(0) <= (x(0) and not(y(0)) and not(load)) or
              (not(x(0)) and y(0) and not(load)) or
              (a(0) and load);
      for i in 1 to (width - 1) loop
         -- to be clockess, the following two equations must not race
         y(i) <= (x(i-1) and y(i-1) and not(load)) or (b(i) and load);
         x(i) <= (x(i) and not(y(i)) and not(load)) or
                 (not(x(i)) and y(i) and not(load)) or
                 (a(i) and load);
      end loop;
      yout <= (x(width - 1) and y(width - 1)) or
              (yout and not(load));
      s <= x;
      cout <= yout;
      d_check := yin;
      for i in 0 to (width - 1) loop
         d_check := d_check or y(i);
      end loop;
      done <= not(d_check) and not(load);
   end process;
end Behavioral;
```

```
-- File    : carry_feedback2.vhd
-- Author  : James Haas
-- Date    : July, 2003
-- Purpose : This file defines the clocked carry_feedback design.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity carry_feedback2 is
   generic
   (
      width : integer := 16
   );
   port
   (
      a : in std_logic_vector((width - 1) downto 0);
      b : in std_logic_vector((width - 1) downto 0);
      cin  : in std_logic;
      load : in std_logic;
      cout : out std_logic;
      s    : out std_logic_vector((width - 1) downto 0);
      done : out std_logic;
      clk : in std_logic
   );
end carry_feedback2;

architecture Behavioral of carry_feedback2 is

   signal x : std_logic_vector(width downto 0);
   signal y : std_logic_vector(width downto 0);
   signal carry : std_logic_vector(width downto 0);
   signal sum : std_logic_vector(width downto 0);

begin

   process (clk, load, a, b, cin, cout, s, done, carry, sum, x, y)

      variable d_check : std_logic;

   begin
      if clk'event and clk = '1' then
         if (load = '1') then
            sum <= a & cin;
            carry <= b & cin;
         else
            sum <= x;
            carry <= y;
         end if;
      end if;
      x <= sum xor carry;
      y <= (sum((width-1) downto 0) and carry((width-1) downto 0)) & '0';
      d_check := not(load);
      for i in 0 to width loop
         d_check := d_check and not(carry(i));
      end loop;
      done <= d_check;
   end process;
   cout <= (x(width) and y(width)) or (cout and not(load));
   s <= sum(width downto 1);

end Behavioral;
```

CURRICULUM VITAE

NAME:	James Arthur Haas
ADDRESS:	6320 Horizon Way Charlestown, IN 47111
DOB:	Jeffersonville, IN – June 2, 1971
EDUCATION & TRAINING:	B.S. Electrical Engineering Purdue University 1989-94

AWARDS:

PROFESSIONAL SOCIETIES:

PUBLICATIONS:

NATIONAL MEETING PRESENTATIONS:

REFEREED JOURNALS:

BOOKS AND SYMPOSIA:

INVITED PRESENTATIONS: