ARITHMETIC ARRAYS FOR RECONFIGURABLE FABRICS

by

Saket A. Jamkar

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2004

APPROVED

By

Advisor Signature:__________________________

Advisor Title: _____________________________

Date:______________________________________

Abstract

Commercial fine-grained FPGA architectures are known to be 50 times slower than custom circuits for structured logic such as ALUs and multipliers [Ebeling1996]. Many architectures have tried to incorporate coarse-grained structures on reconfigurable platforms to speed up arithmetic operations, because generally speed and area improve with increasing granularity for these operations.

This thesis proposes a novel building block for arithmetic structures for reconfigurable computing. The building block is capable of performing many arithmetic operations including addition/subtraction of 16 or 32-bit operands, multiplication of 8-bit operands, comparison and logical operations such as AND, OR, Exclusive-OR operations on 16-bit operands. Multiple blocks can be used together to perform these computations on larger word-sizes. Synthesis results show that the area of the reconfigurable arithmetic block is 17,579 square microns in a 0.11 micron standard cell technology. The delay for 8x8-bit multiplication is 2.48ns and 32-bit addition is 2.01ns. We compare 110nm standard cell post-synthesis results for the reconfigurable arithmetic block to results of synthesis using Look-Up Tables on Xilinx Spartan-3 FPGAs implemented in 90nm technology. The FPGA designs were found to be roughly 10 times slower than the standard cell designs, which imply an approximate 20x speedup for a full custom layout of our block.

Acknowledgements

I would like to acknowledge and thank my advisor Katherine Compton for providing me with continual guidance, support and valuable insight in completing this thesis. I also have to thank her for her effective management style which ensured that steady progress was made every week. I couldn’t have asked for a better advisor. I am certain all her current students also appreciate her kindness and the genuine interest she takes in all her students’ welfare.

I would also like to thank my parents for their enduring love, for their excellent genes and for the incredible examples they set that inspires me to emulate them. It was only because of the value for principles that they instilled in me that I have achieved everything that I have. I would like to thank my friends, especially Nivedita, for their patient ears and excellent advice in all the bad times and all the celebrations in all the good times.

I am also indebted to the Department of Electrical and Computer Engineering and the University of Wisconsin-Madison for fostering an environment in which I could excel. The quality of the program made it an honor to have studied here. I am also grateful to the University for funding my graduate work.

I’d also like to thank Professor Volkan Kursun for his help with schemes for reducing static leakage power during stand-by operation modes. I also appreciate Kyle Rupnow’s help with Design Compiler and other tool related issues, and for being a fun lab-mate to work with.

Table of Contents

Table of Figures

1 Introduction

Most computing applications need some arithmetic. In applications which do not require a lot of arithmetic operations, the look-up table (LUT) elements, found in commercial fine-grained FPGA architectures are appropriate. A k-input LUT element is a small memory which can be configured to perform any logical function with k-inputs. Therefore the LUT approach results in one of the most flexible architectures. However in arithmetic-intensive applications, the LUT approach tends to have a significant speed handicap. The speed of arithmetic implemented in LUTs is 50 times slower than that in ASIC designs [Ebeling1996]. Heterogeneous architectures, consisting of LUTs and some arithmetic units, can help make these reconfigurable computers more competitive with ASIC technology for arithmetic. A number of commercial FPGA architectures have moved in this direction, including Xilinx's Virtex-II [Xilinx2001], Virtex-IV[Xilinx2004] and Altera's Stratix [Altera2004] and Altera's Stratix-II [Altera2005]. However, these architectures are still primarily LUT-based.

Our goal is to take this specialization a step further, and design new high-speed reconfigurable fabric primarily composed of arithmetic-optimized structures. In particular, we wish to create a reconfigurable arithmetic fabric able to perform the operations commonly used in digital signal processing and other math-intensive algorithms more efficiently than a traditional LUT-based FPGA, yet flexible enough to allow for both re-use among applications and support for different word sizes.

In keeping with the principle of reconfigurable computing, we would like to incorporate some degree of flexibility in the arithmetic structures, so that they can serve different purposes. Such flexibility will help us minimize the wastage of resources that can occur in coarse-grained heterogeneous architectures. We would also like to provide some scalability, i.e. the ability to treat larger operands, by combining many building blocks together. The target application domain is that of Digital Signal Processing (DSP). Hence care was taken to ensure that most operations required in DSP are handled by the reconfigurable arithmetic block. An attempt has been made to utilize the advances in Computer Arithmetic to achieve good speed/area tradeoffs in our design.

Chapter 2 introduces the area of reconfigurable computing. Coarse-grained architectures are such as RaPiD, CHESS and MATRIX are discussed. This chapter also discusses commercial heterogeneous such as Xilinx's Virtex FPGA and Altera's Stratix FPGA. Chapter 3 gives a summary of computer arithmetic. It explains various adder and multiplier designs. It also summarizes the results of synthesis of many of these schemes as implemented in Verilog and a standard-cell tool flow.

Chapter 4 discusses the approach used to design the reconfigurable arithmetic block. It discusses the methodology and also the features of the arithmetic block. Extensions to the block, such as structures to help in pipelining, sign-extension and cascaded subtraction, are also discussed.

Chapter 5 presents areas of future work, including techniques to reduce static power dissipation, routing structure design, and the inclusion of some amount of general-purpose LUT-based logic to make the design more robust.

2 Reconfigurable Computing

Reconfigurable Computing refers to systems incorporating some form of hardware programmability – customizing how the hardware is used using a number of physical control points. These control points can then be changed periodically in order to execute different applications using the same hardware [Compton2002]. It is an actively researched area with a large amount of work being done to make this computing platform accessible to a wider spectrum of users. It is also a very promising technology in its ability to speed up a host of applications.

2.1 Introduction

There are two popular technologies for implementing computation. First is the Application Specific Integrated Circuits (ASICs), which are marked by their high performance. The high initial costs for designing and fabricating an ASIC make it such an expensive proposition that only products with large volumes can be implemented in this way. These devices are custom designed for a specific application. The drawback of ASICs is inflexibility: ASICs can only implement the function or functions for which they were designed. Microprocessors are the other popular means for computing. Microprocessors implement numerous small operations called instructions. The desired functionality can then be achieved by writing software programs composed from this set of instructions. However, a drawback of this technique is the time taken for reading each instruction from memory and decoding its meaning is overhead for the execution of every instruction.

Reconfigurable computing is more flexible than ASICs but has better performance than microprocessors for compute-intensive applications. Reconfigurable computing devices consist of hardware-programmable computational elements connected by programmable routing resources. The programmable computational elements vary from reconfigurable device to reconfigurable device. They can be fine-grained, such as a 3-input Look-Up Table (LUT), or coarse-grained, such as 32-bit ALUs. Most commonly, the programmable routing is two-dimensional, and consists of connection blocks and switch blocks derived usually from tri-state elements or multiplexers. Tri-state buffers or transistors controlled by programming bits allow two wires to optionally be connected depending on the implemented circuit. Multiplexers allow the choice from a number of possible inputs. Next we examine coarse-grained reconfigurable architectures.

2.2 Review of Prior Art

Because traditional FPGAs must decompose all arithmetic functions into small LUT-sized pieces, they can be much slower than custom arithmetic circuits. Even simple circuits such as ALUs and multipliers can be up to 50 times slower in an FPGA than full custom layouts [Stone1996]. Apart from the fine-grained aspect of the LUTs, the performance is also degraded simply because the LUTs are very general-purpose, whereas custom circuitry can be optimized for the targeted computation. Therefore, some researchers and commercial FPGA vendors have examined the use of custom arithmetic circuitry in reconfigurable fabrics.

2.2.1 RaPiD

RaPiD (Reconfigurable Pipelined Datapath) is a coarse-grained style of reconfigurable architecture that is one dimensional in nature. Totem [Compton2002] is a tool used to automatically generate these RaPiD-style architectures customized for a particular set of applications. A cell from the manually-designed RaPiD-I architecture, a well-known implementation of this style, is shown in Figure 2-1. The cell consists of datapath registers, ALUs and RAMs that operate on 16-bit operands. A multiplier is also present in this architecture which produces a 32-bit product, separately output as the 16-bit high result and 16-bit low result. The horizontal lines in the figure represent 16-bit routing tracks, while the vertical lines symbolize the 16-bit multiplexers and demultiplexers connecting each unit to each of the tracks.

Due to its coarse-grained nature, RaPiD-I is area- and delay-efficient when performing operations with 16-bit operands. However, if one needs to work with operands of smaller bit-widths, there is an unavoidable wastage of resources. Set architectural bit-widths are also problematic when operands larger than word-width of 16 have to be processed.

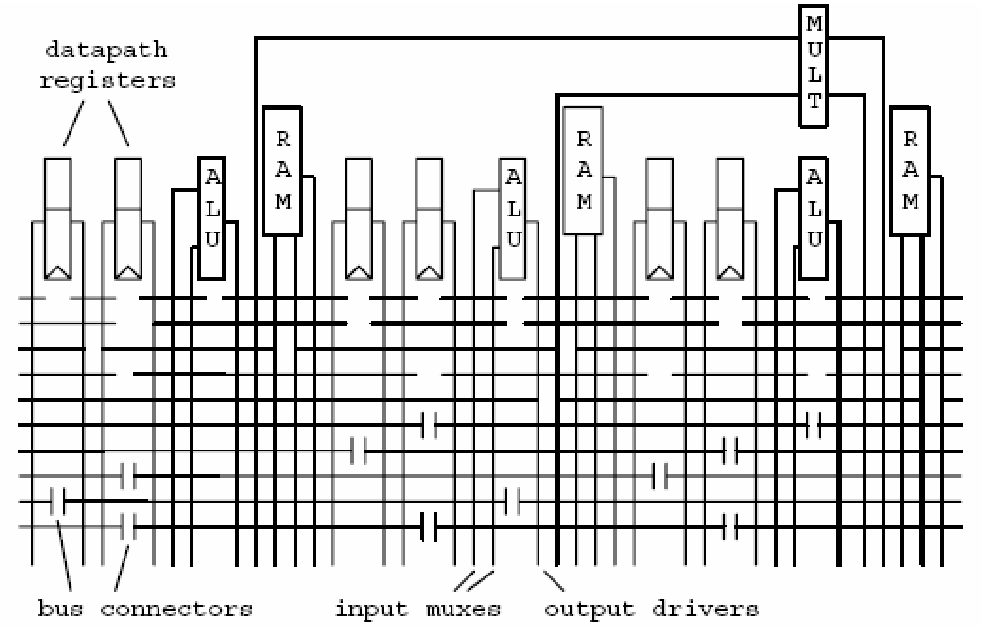


Figure 2-1: One cell from the RaPiD-I architecture with 16-bit ALUs, RAMs, datapath registers and buses, and a 16x16 multiplier [Ebeling1996]

Although the single-dimensional nature of RaPiD can make it easier for the place and route tool to operate, it also results in reduced flexibility. One can imagine situations where more routing resources are required and could have been satisfied if the routing architecture was two-dimensional. For instance if one needs to connect two diagonally-placed blocks in a two dimensional architecture there are two possible paths, one traveling horizontally then vertically, and one traveling vertically then horizontally. In RaPiD however, there is only one possible channel to connect ells. Once that channel is full at a given horizontal offset, no more signals may pass through that location.

2.2.2 CHESS

CHESS is a two-dimensional coarse-grained architecture, which consists of a grid of 4-bit ALUs interspersed with flexible routing resources. The area occupied by the routing resources is roughly equal to that occupied by ALU blocks. While this may be considered an improvement when compared to the area occupied by routing on fine-grained commercial FPGAs, it could also limit the routing flexibility of the architecture overall.

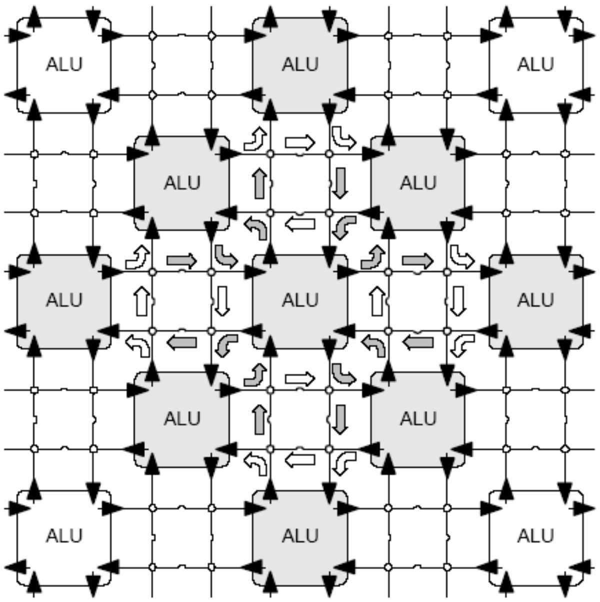


Figure 2-2: CHESS Layout and nearest neighbor routing [Marshall1999]

The 4-bit granularity of this architecture allows for 8, 12, 16-bit or larger operations by combining ALUs together. However, adders above the native granularity would propagate the carry using a ripple carry scheme, which is much slower than a carry look-ahead scheme. Thus the 4-bit granularity implies that CHESS is likely slower than RaPiD for 16-bit arithmetic, but most likely performs better than commercial FPGA architectures with fine-grained LUT-based structures. As this architecture does not have dedicated multiplier logic, multiple ALUs have to be used together for multiplication. For non-constant multiplies, this will almost always be slower than what could be achieved with a custom multiplier unit.

2.2.3 MATRIX

MATRIX is a coarse-grain reconfigurable computing architecture that supports configurable instruction distribution. Specifically, it is able to operate in various computing styles, such as SIMD, VLIW, Systolic and MSIMD. MATRIX is formed of an array of Basic Functional Units (BFUs) arranged in a 2D array surrounded by a configurable routing network. The BFU structure is shown in Figure 2-3. The BFU in MATRIX consists of an 8-bit ALU, an 8x8-bit multiply unit, a 256 x 8-bit memory and reduction control logic including a 20 x 8 NOR plane. The routing network is hierarchical supporting 3 levels of interconnect.

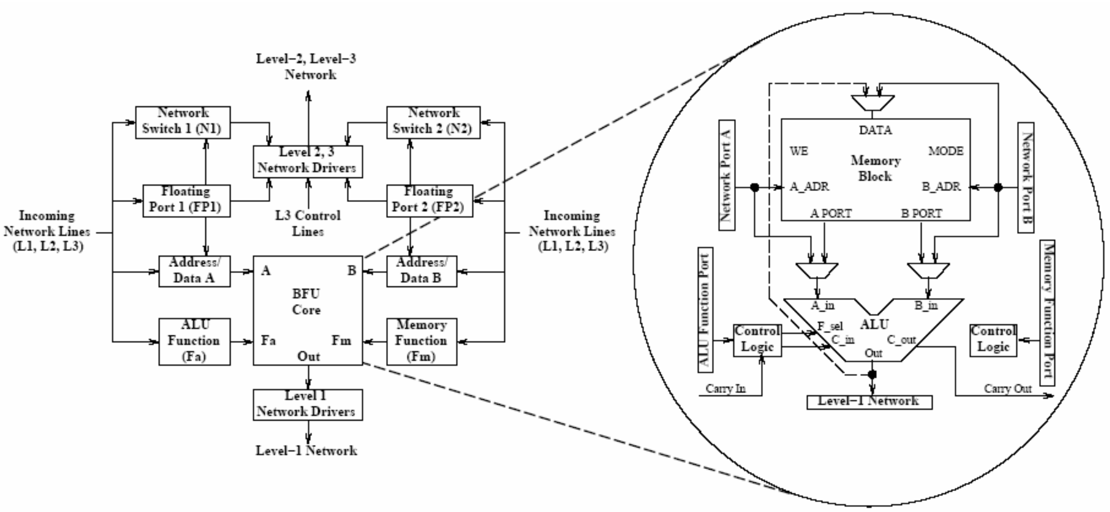


Figure 2-3: MATRIX Basic Functional Unit [Mirsky96]

The BFU supports 8-bit operands for ALU operations and 2-cycle multiplication. A Verilog “*”operator is used to implement multiplication which might be good for standard cell implementations with a multiplier cell, but a specific multiplier implementation would need to be chosen for custom circuit design and layout, which would be required for maximum performance and minimum area.. The internal structure of the ALU is not discussed [Mirsky96].

2.2.4 Xilinx Virtex-II and IV

The Virtex-II [Xilinx2001] series is fabricated in a 0.15 micron 8 metal layer process technology. The largest device contains 168 18 x 18-bit multipliers supporting 18-bit signed or 17-bit unsigned multiplication, and up to 3Mb of block RAM. Columns of these multiplier and RAM units are flanked by columns of logic cells. The Virtex-II Pro series is similar to the Virtex-II, but also contains IBM PowerPC processors on-chip and is fabricated in a 0.13 micron process.

Virtex-IV [Xilinx2004] is one of the latest series of Xilinx FPGAs. The series introduces a new development model from Xilinx called “Application Specific Modular Blocks” (ASMBL).

The purpose of this model is to deliver devices with the best mix of logic, memory, multipliers and clock management optimized for particular application domains. For instance, the DSP version of Virtex-IV has more multipliers than other versions because DSP is a multiplication-intensive application domain.

The DSP48 slice from Virtex-IV (Figure 2-4) supports many arithmetic functions with a 18x18-bit 2's complement multiplier followed by a 48-bit adder/subtractor with register elements in between. As the result of the multiplier is 36 bits wide, the 48-bit adder can perform many intermediate additions without overflow problems. There is also a facility to sign extend most operands.

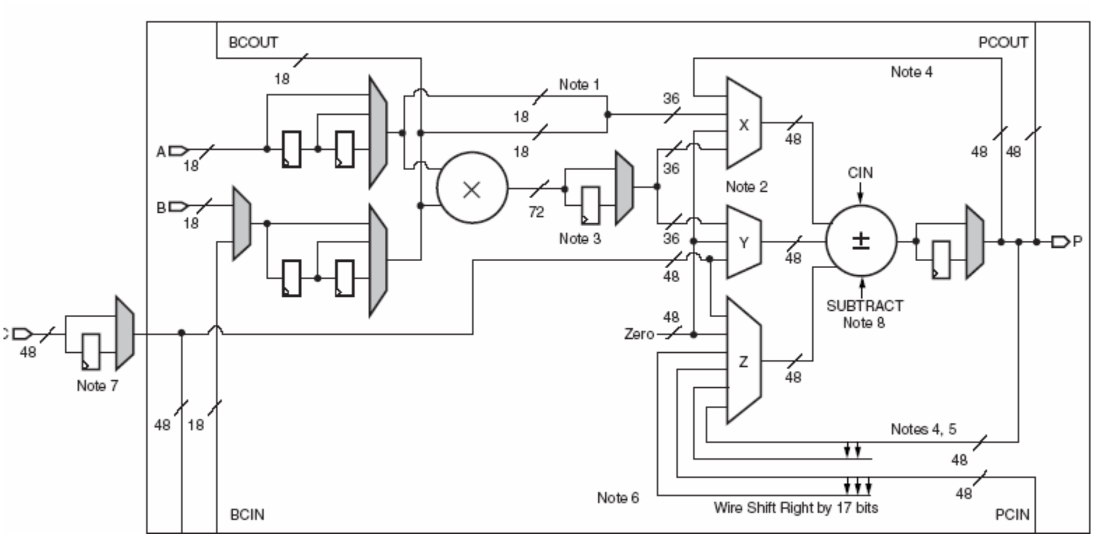


Figure 2-4: DSP48 Slice from Virtex–IV containing 18-bit multiplier followed by 48-bit adder/subtractor [Xilinx2004]

2.2.5 Altera Stratix and Stratix-II

The Stratix and Stratix-II series are heterogeneous FPGAs available from Altera. These devices contain a two-dimensional row- and column-based architecture to implement custom logic. A series of column and row interconnects of varying length and speed provide signal interconnects between LUT elements, memory block structures, and DSP blocks that implement multiply or multiply and add functions.

Stratix is a series of FPGAs built in a 0.13 micron technology. A Logic Element (LE), containing a 4-input LUT is the basic building block in this series. The LE can also be configured in the adder/subtractor mode to implement 1-bit of a carry-select adder. The largest of the Stratix devices also contains 88 18x18 multipliers for fast multiplication. [Altera2004]

Stratix-II is among the latest FPGA devices and is fabricated in a 90nm technology. The Stratix-II devices contain a new kind of logic block, called Adaptive Logic Module (ALM), which is like a 7-input LUT which can be flexibly repartitioned into a number of configurations. For example, it can compute a function of 5 inputs and a function of 3 inputs, or two different functions of 4 inputs. The number of DSP blocks to perform multiplications has also been increased: the largest member of Stratix-II family contains 384 18x18-bit multipliers.

The structure of the DSP Block in the Stratix-II series is the same as the Stratix series, as shown in Figure 2-66. Each DSP block can be configured to support up to eight 9x9-bit multipliers, four 18x18-bit multipliers, or one 36x36-bit multiplier. Registers are provided for retiming the design to reduce the clock period. Adders are provided to perform multiply and accumulate or multiply and add functions.

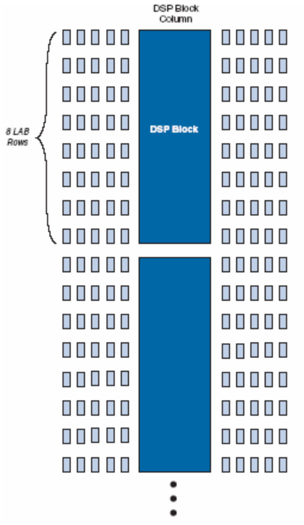


Figure 2-5: The DSP Blocks are placed in columns surrounded by many rows of LUT elements [Altera2005]

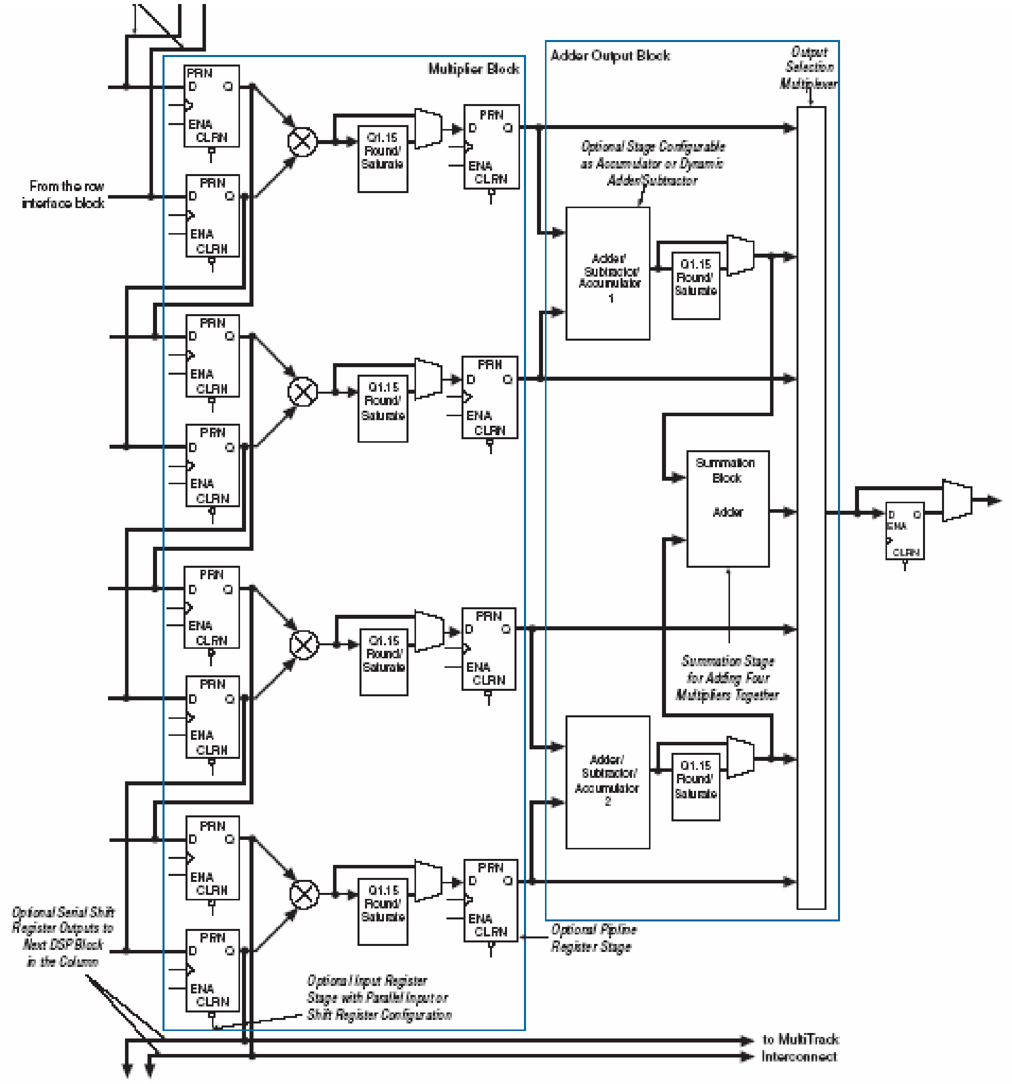


Figure 2-6: DSP Block diagram for 18 x 18 bit multiplier configuration. 4 such multipliers can be seen feeding into adders. Registers are present throughout the block [Altera2005]

3 Computer Arithmetic

Arithmetic operations, such as addition, subtraction, and multiplication, are required for nearly all applications. DSP applications in particular require a large number of these types of operations. A great deal of research has been done to increase the speed or reduce the area and/or power of these structures. This section discusses different adder and multiplier implementations, along with the pros and cons of each.

This section also presents area/delay comparisons of various types of adders and multipliers. These structures were all modeled in a Register Transfer Level (RTL) style of coding, primarily using continuous assign statements with simple bit operations such as &, |, and ^. In places where case statements were used, such as for multiplexers, the cases were completely specified to avoid the inference of latches. The designs were simulated with Mentor Graphics ModelSim. Verification of the designs was then done by an intelligent choice of test vectors. The vectors were chosen so as to exercise all the functional blocks constituting each adder/multiplier completely. In large designs the subcomponents were first tested, followed by the interconnections between components. Thus although the verification was not exhaustive, a high level of coverage was achieved. These designs were then compiled and synthesized using Synopsys Design Compiler for the gflxp 0.11 micron standard cell library from LSI Logic, and the area and delay measurements were gathered.

3.1 Adders

Adders are a basic component of microprocessors and nearly any arithmetic circuit, and are frequently on the critical path. Fast adders speed up the addition calculation through a rearrangement of the adder equations or through some intelligent observations about the addition process. While adder speed is essential, adder area is also important, especially for arithmetic circuits that may require many adders. The next few sections discuss adder design.

3.1.1 Adder Basics

A multi-bit addition operation can be decomposed into half adder and full adder structures, with fast adders containing some additional circuitry. Half adders and full adders compute the well-known logic functions given as follows:

Half adder: $Sum = A \oplus B$, $Cout = AB$

Full adder: $Sum = A \oplus B \oplus C = A\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}\overline{B}C + ABC$

$Cout = AB + BC + CA$

3.1.2 Ripple Carry Adder

This is the simplest adder circuit. An N-bit ripple carry adder consists of N full adders with the carry signal propagating from one full-adder stage to the next from LSB to MSB. A 4-bit ripple carry adder structures is shown in Figure 3-1.

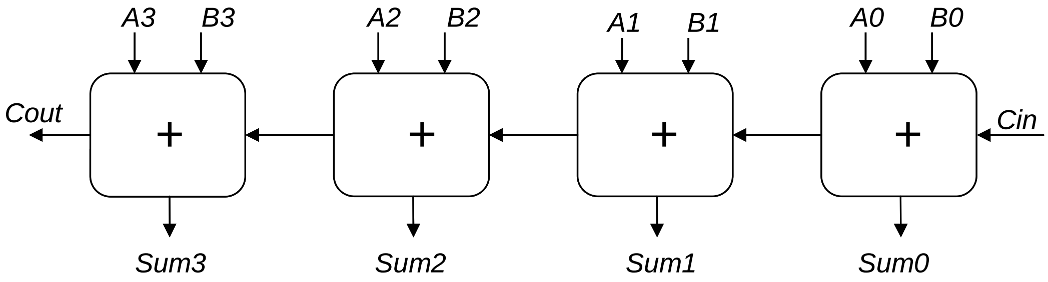


Figure 3-1: 4-bit Ripple carry adder

The ripple-carry adder is a good baseline design for comparison with other adders. It has many advantages which include low power [CNagendra1994], low area and a simple layout. The drawback of the ripple carry adder, though, is its slow speed. The delay of the adder is linearly dependent on the bit-width (N) of the adder. The critical path of the ripple carry adder consists of the carry chain from the first full adder to the last. Therefore, during circuit-level design, the carry signal is frequently assigned to the transistor closest to the gate output for the carry computation.

Circuits are optimized to produce fast carries because it constitutes a large fraction of the critical path. The delay of a ripple carry adder is given by the following equation: [Rabaey2003]

$$T(CriticalPath - RCA) = (N-1) \times T(Carry) + T(Sum)$$

3.1.3 Carry-Select Adder

The carry-select adder employs an intelligent technique to reduce the carry propagation delay. As the carry signal takes on a value of either a 1 or a 0, if we calculate the sum for both these cases in advance, we can reduce the carry propagation chain to just the selection of the correct outputs at each stage using multiplexers. The critical path will now just consist of multiplexers at the output of each bit, given by the following equation: [Rabaey2003]

$$T(CriticalPath - CSA) = (N-1) \times T(MuxSelect - Out) + T(Sum)$$

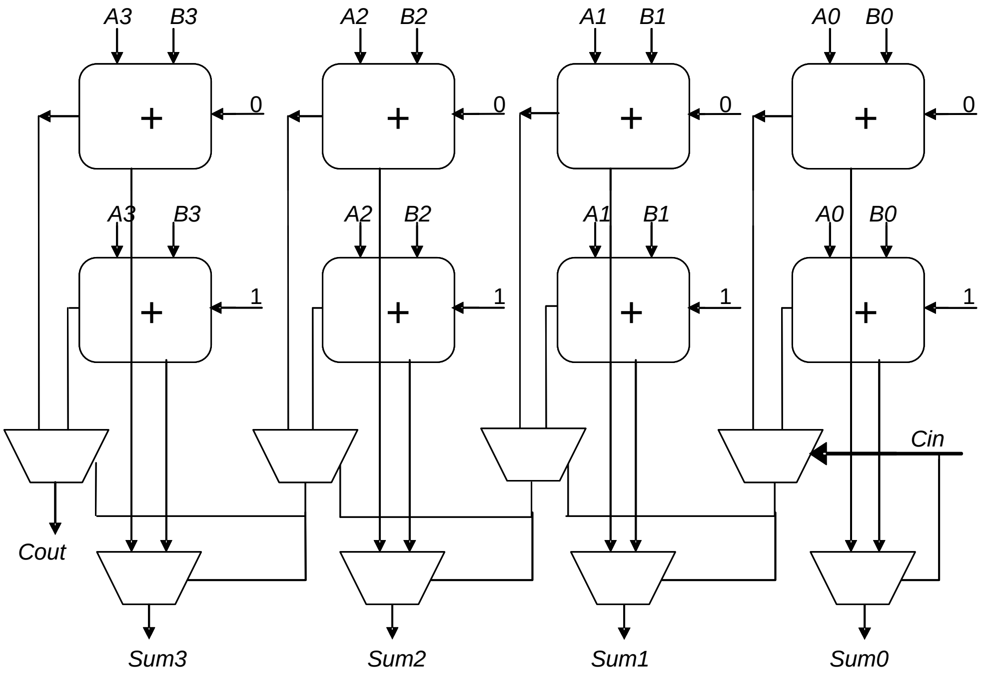


Figure 3-2: 4-bit carry-select adder

The speed of the carry-select adder is achieved at the cost of doubling the area, because we now require two adders per bit: one adder to calculate the sum with a carry-in of 0, and another adder to calculate the sum with a carry-in of 1. In addition, we need a multiplexer for every bit to choose the result based on the actual carry value. As a consequence of this duplication of logic, this design also consumes more power. Figure 3-2 shows the structure of the carry-select adder.

3.1.4 Carry Look-Ahead Adder

The carry look-ahead adder (CLA) is theoretically one of the fastest methods for addition. Weinberger and Smith invented the CLA in 1958 [Weinberger58]. The CLA uses intermediate information to determine in advance if there will be a carry out of a given bit position. Figure 3-3 shows the truth table for a full adder, including this extra carry information. For the delete condition, there will be no carry out of the bit position. For the propagate condition, there will only be a carry out if there is a carry in. For the generate/propagate condition, there will always be a carry out at that position.

A	***B***	***C***	***Sum***	***Cout***	***Condition***
0	*0*	*0*	*0*	*0*	*Delete*
0	*0*	*1*	*1*	*0*	*Delete*
0	*1*	*0*	*1*	*0*	*Propagate*
0	*1*	*1*	*0*	*1*	*Propagate*
1	*0*	*0*	*1*	*0*	*Propagate*
1	*0*	*1*	*0*	*1*	*Propagate*
1	*1*	*0*	*0*	*1*	*Generate/Propagate*
1	*1*	*1*	*1*	*1*	*Generate/Propagate*

Figure 3-3: Generate and propagate information for a CLA

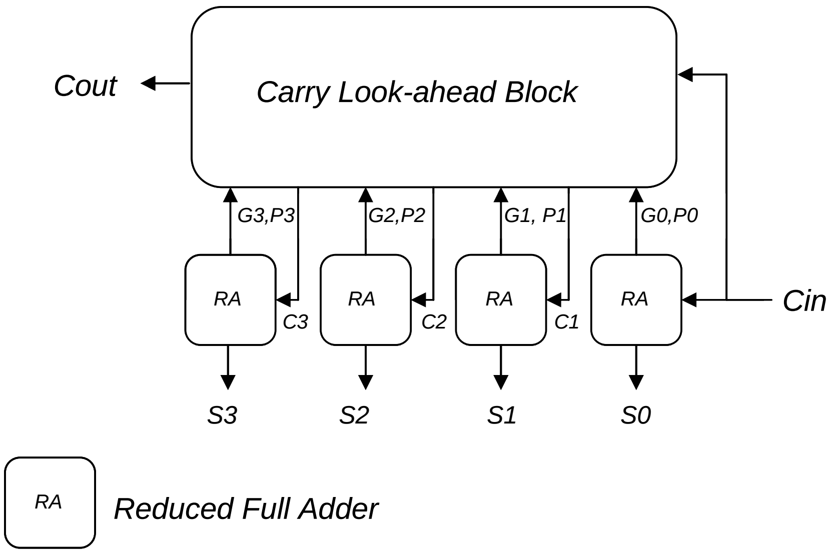


Figure 3-4: 4-bit Carry look-ahead Adder: Reduced adders are used, as they are no longer required to compute the carry-out

Figure 3-4 shows the block diagram for a 4-bit section of a CLA. The CLA block at the top of the diagram is a set of circuitry that creates generate and propagate signals for a group of full adders, as well as the carry-out from that group. The following equations compute each position's generate and propagate signals:

$$\text{Generate:} \qquad Gi = Ai \bullet Bi$$

$$\text{Propagate:} \qquad Pi = Ai + Bi$$

Some books instead define the propagate signal as the exclusive-OR of the A and B signals, but this does not change the result of addition. We chose the above definition because the implementation of the OR operation is more efficient than that of exclusive-OR in most technologies. The equations for the carries in a CLA are given by:

$$C1 = G0 + P0 \bullet Cin$$

$$C2 = G1 + G0 \bullet P1 + P \bullet \cdot P2 \bullet Cin$$

$$C3 = G2 + G1 \bullet P2 + G0 \bullet P1 \bullet P2 + P0 \bullet P1 \bullet P2 \bullet Cin$$

As we attempt to compute the carries further and further in advance, larger gates are required. For example, computing C3 requires the use of a 4 input AND gate and a 4 input OR gate. Hence, usually the size of the look-ahead logic is limited to 3 carries. AND Gates with 5/6 inputs would be needed for the next 2 carry signals, which makes their implementation in CMOS very slow due to the stacked transistors in the pull-up or pull-down paths.

As the carry calculation is performed by the carry look-ahead block, the one-bit adder equations for a CLA are the reduced full-adder equations because carry calculation is no longer needed. The reduced full adder performs the operation given by equation below:

$$Sum = A \oplus B \oplus C = A\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}\overline{B}C + ABC$$

A 16-bit CLA extends the concepts of the 4-bit CLA by creating a block generate and block propagate signal for each blocks of 4 bits (0:3, 4:7, 8:11, 12:15) and applying the look-ahead equations on these signals, as shown in Figure 3-4. The equations computed by the Block Generate and the Block Propagate Signals are:

$$Block\ Generate = G(0{:}3) = G3 + G2 \bullet P3 + G1 \bullet P2 \bullet P3 + G0 \bullet P1 \bullet P2 \bullet P3$$

$$Block\ Propagate = P(0{:}3) = P0 \bullet P1 \bullet P2 \bullet P3$$

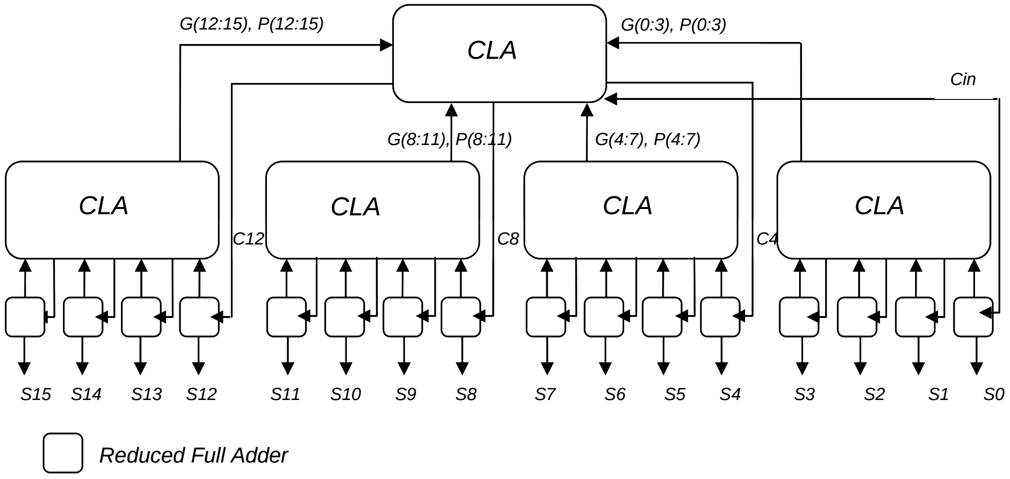


Figure 3-5: 16-bit carry look-ahead adder

3.1.5 Prefix Adders

Among the various binary adder architectures, prefix adders are particularly attractive because they have the minimum possible logic depth. These adders are also termed logarithmic adders because their critical path is O(log(N)). Examples of prefix adders are Kogge-Stone (Figure 3-6), Brent-Kung (Figure 3-7), and Han-Carlson (Figure 3-8).

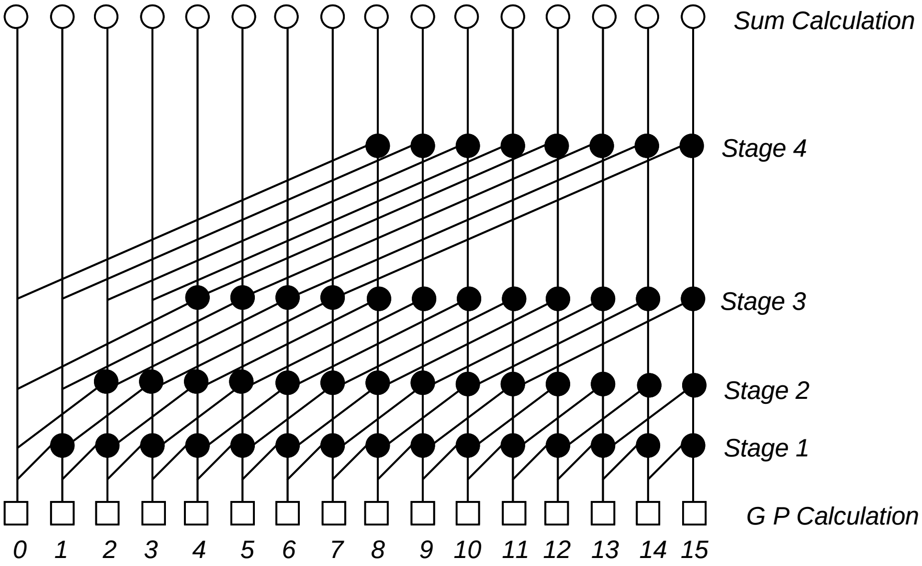


Figure 3-6: 16-bit radix-2 Kogge-Stone Adder [KoggeStone1973]

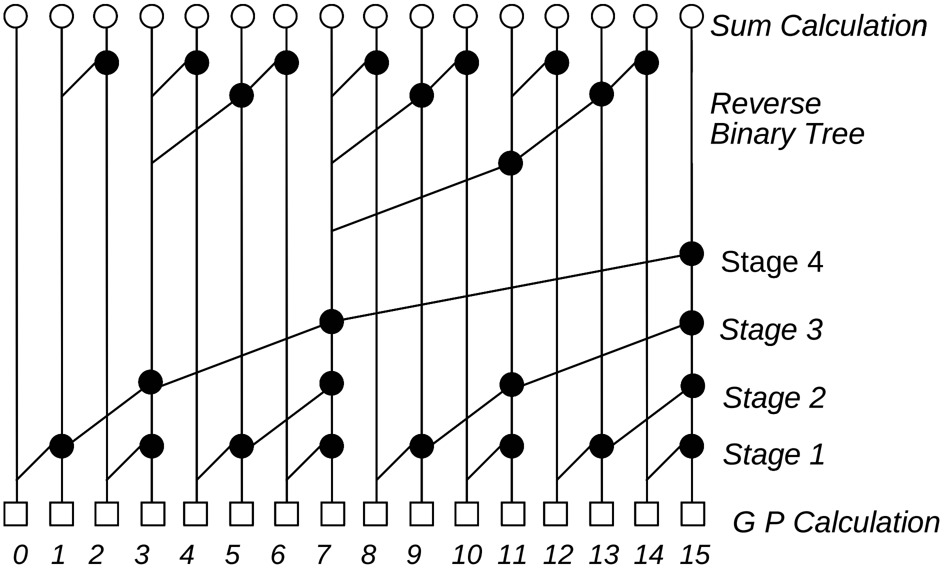


Figure 3-7 16-bit radix-2 Brent-Kung adder [BrentKung1982]

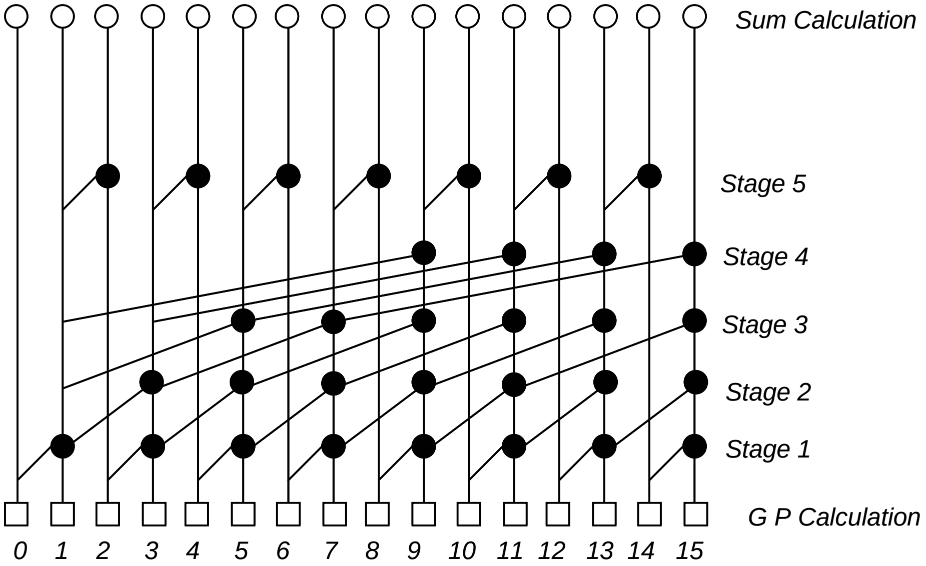


Figure 3-8: 16-bit radix-2 Han-Carlson adder [HanCarlson1987]

The addition of two binary numbers can be formulated as a *prefix problem*. In a prefix problem, outputs {y(n-1), y(n-2)… y(0)} are computed from n inputs {(x(n-1), x(n-2)… x(0))} using an arbitrary associative operator (*) as follows:

$y(0) = x(0)$

$y(1) = x(1) * x(0)$

$y(n-1) = x(n-1) * x(n-2) * \ldots x(1) * x(0)$

The problem can be formulated recursively as

$y(0) = x(0)$

$y(i) = x(i) * y(i-1) \qquad \text{where } i = 1, 2\ldots, n-1$

In other words in a prefix problem, every output depends on all inputs of equal and lower magnitude, and every input influences all outputs of equal or higher magnitude. Due to the associativity of the prefix-operator, the individual operations can be carried out in any order. This is a fundamental property which explains why there are various tree structures for addition.

Let us define the * operator to be the carry-merge operator which combines generate and propagate signals using the following equations:

$$(Gout, Pout) = (G2, P2) * (G1,P1)= (G2 + P2\bullet G1, P2\bullet P1$$

Therefore the addition operation can be performed using the following steps:

Calculate the Generate and Propagate signals for each bit. Perform the carry-merge operation from the equation above at each junction in the carry merge tree for the desired logarithmic adder, Brent-Kung, Kogge-Stone, or Han-Carlson. The underlying carry-merge operation is the same for all logarithmic adders but the connections in the tree are different. After $\log_2(N)$ stages, where N is the bit-width of the adder, the carry signals will be fully generated. As with the CLA, a reduced full-adder is used to find the sum using the carry, generate and propagate signals.

The Brent-Kung adder has the lowest area and the slowest speed of all the logarithmic adders. The Kogge-Stone adder is the fastest in the family of logarithmic adders, but is the largest in area due to reduced fan-outs on internal nodes. The Han-Carlson adder (HCA) combines the Brent-Kung and Kogge-Stone carry merge trees to achieve a balance both with respect to speed and area.

3.1.6 Adder Comparison

The RCA, CLA, and HCA adder designs were created in Verilog and synthesized in a LSI Logic gflxp 0.11 micron standard cell library using Synopsys Design Compiler in order to compare areas and speeds. For many of these designs, several different bit-widths were examined.

Table 1: The area and timing results for all synthesized adders. Timing in ns and Area in square microns.

Adders		
Design	**Timing**	**Area**
Ripple Carry Adders		
RCA4	0.71	262.34
RCA8	1.33	524.62
RCA16	2.59	1049.14
RCA32	5.1	2098.2
Carry Lookahead Adders		
CLA4	0.49	428.56
CLA8	0.88	857.02
CLA16	1.09	1792.34
CLA32	1.48	3584.58
Han Carlson Adder		
HCA16	1.33	1758

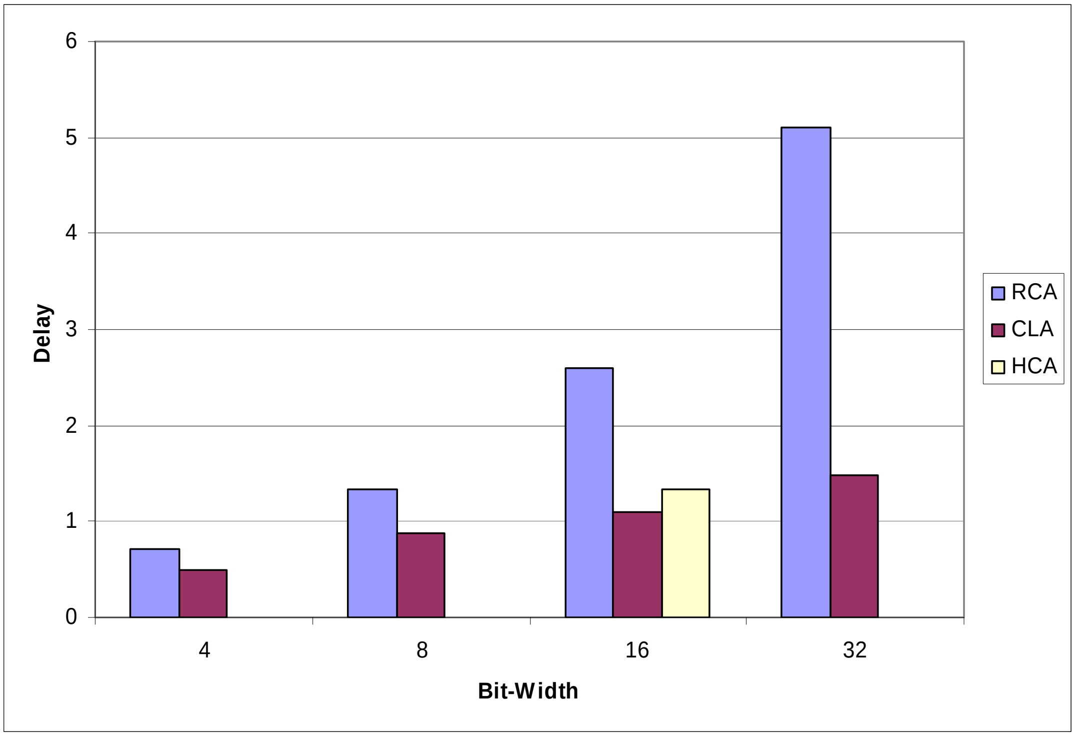


Figure 3-9: Adder delay (ns) vs. bit-width

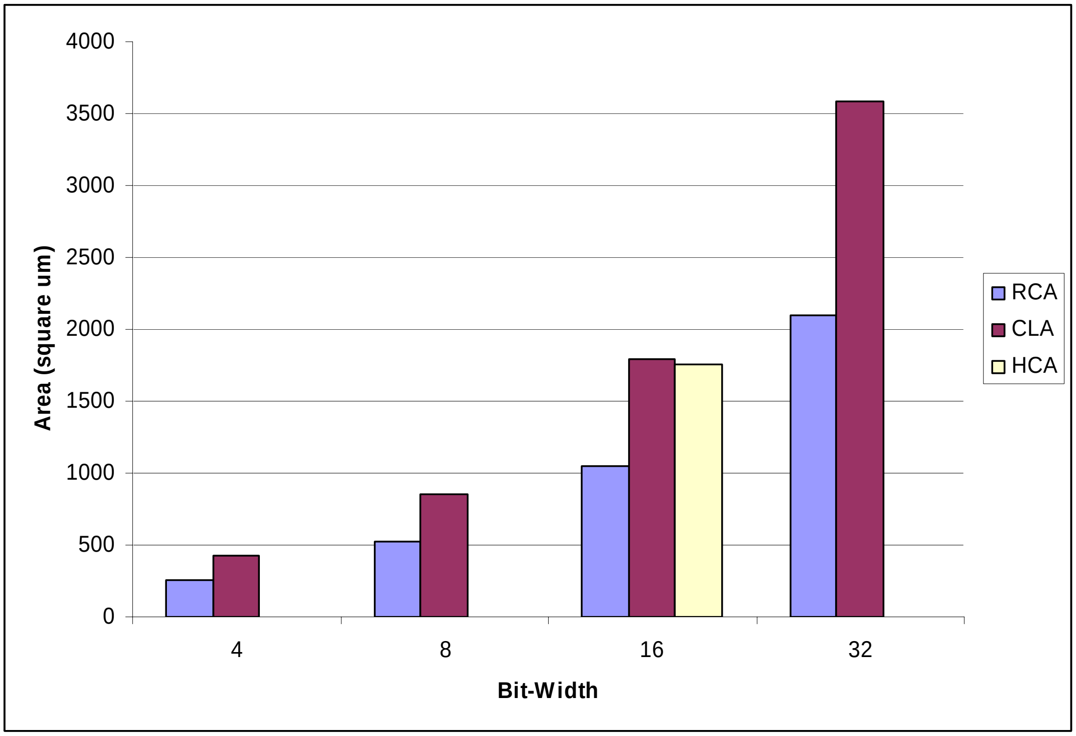


Figure 3-10: Adder area (square micrometer) vs. bit-width (bits)

As shown by our results, the CLA is the fastest scheme for implementing addition, where as the RCA is the smallest. [HanCarlson1986] describes the Han-Carlson Adder as the fastest area-efficient adder, as it is slightly smaller than a CLA, but significantly faster than an RCA. The HCA is used in the Intel Pentium 4 processor, as it is considered to be the “best” implementation considering all metrics i.e. area, speed and power, but this is primarily true only for a full-custom layout, as some wire length savings over a CLA are possible. However, for all practical purposes for our study, the CLA is only slightly larger than the HCA, and is measurably faster. Since our goal is to target speed in order to combat one of the criticisms of reconfigurable logic, we chose the faster CLA adder structure, and feel that the small area penalty can be neglected.

The last line of Table 1 shows the speed and area of the adder automatically synthesized by Synopsys when the Verilog “+” operator is used. It appears that the actual design of this automatically-created adder is an RCA given the correlation of the area and delay values. However, the standard cell library being used contains a special full adder standard cell that

Synopsys only uses when the adder structure is unspecified, as for the “+” operator.

This special cell is what we feel accounts for the small area and delay improvements over our RCA implementation.

3.2 Multipliers

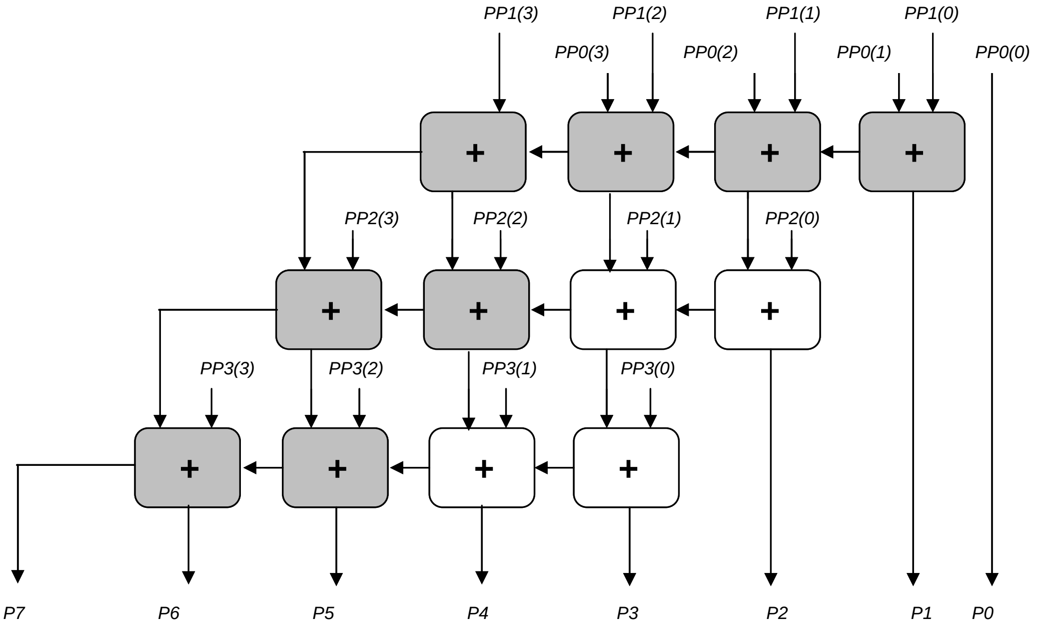
Multiplication is an operation that occurs frequently in digital signal processing and many other applications. However, multipliers occupy a much larger area and incur much longer delays than adders. Therefore it is imperative that special techniques be used to speed up the calculation of the product while maintaining a reasonable area.

The product is the result of multiplying the multiplicand to the multiplier. The multiplication operation is performed in two main steps. First is the partial product formation, which consists of AND-ing each bit of the multiplier with the multiplicand. Each successive partial product belongs one place to the left of the previous partial product. The second step is partial product accumulation, where the partial products are combined to form the result.

3.2.1 Array Multiplier

This is one of the simplest techniques for implementing multiplication. The idea is to add all the N partial products sequentially using N-1 adders. If we are multiplying N bit values then in effect we will need N-1 N-bit adders or N*(N-1) single adder cells. The structure of the array multiplier is shown in Figure 3-11: This figure shows that there are many identical critical paths traveling from the top right to the lower left. The blocks shaded in grey show one such critical path. The delay of the array multiplier is given by [Rabaey2003]:

$$T(critical) = \{(N-1) + (N-2)\}\, T(Carry) + (N-1)\, T(Sum) + T(And)$$



PPx(y) – Partial Product
x – Row Number
y – Bit Number

P - Product

Figure 3-11: 4 x 4 array multiplier (simplified)

Array multipliers are very slow as their critical path is very long. If we assume that the time taken to produce a sum is *klog(N)*, then the time for multiplication with an array multiplier would usually be many multiples of *klog(N)* for current practical values of N. The advantage of the array multiplier is that because of its regular structure it is easy to design and layout. The area requirement is also very low, but by routing the signals a little differently and adding another adder as we do in a carry-save multiplier, the critical path can be reduced substantially.

3.2.2 Carry-Save Multiplier

An optimization to the array multiplier uses the following observation to reduce the critical path of the multiplication. The carry signal being propagated from a certain bit position to the next, need not be propagated horizontally to the next stage but could also be sent diagonally left to the next adder. We observe that as long as each carry-bit maintains its bit position, it doesn't matter how it is eventually added up. Therefore in the carry-save multiplier, all adders connect their carries to an adder to the adjacent adder in a lower row located on the left hand side diagonally. This operation is continued for all carry signals until finally a carry-merge adder adds up the sum and the carries from the previous adder stage to obtain the correct output. The critical path for the carry-save multiplier is shown in the figure below. The critical path delay for the carry-save multiplier is [Rabaey2003]:

$$T(Critical\ path) = 2(N-1)T(Carry) + T(Sum) + T(And)$$

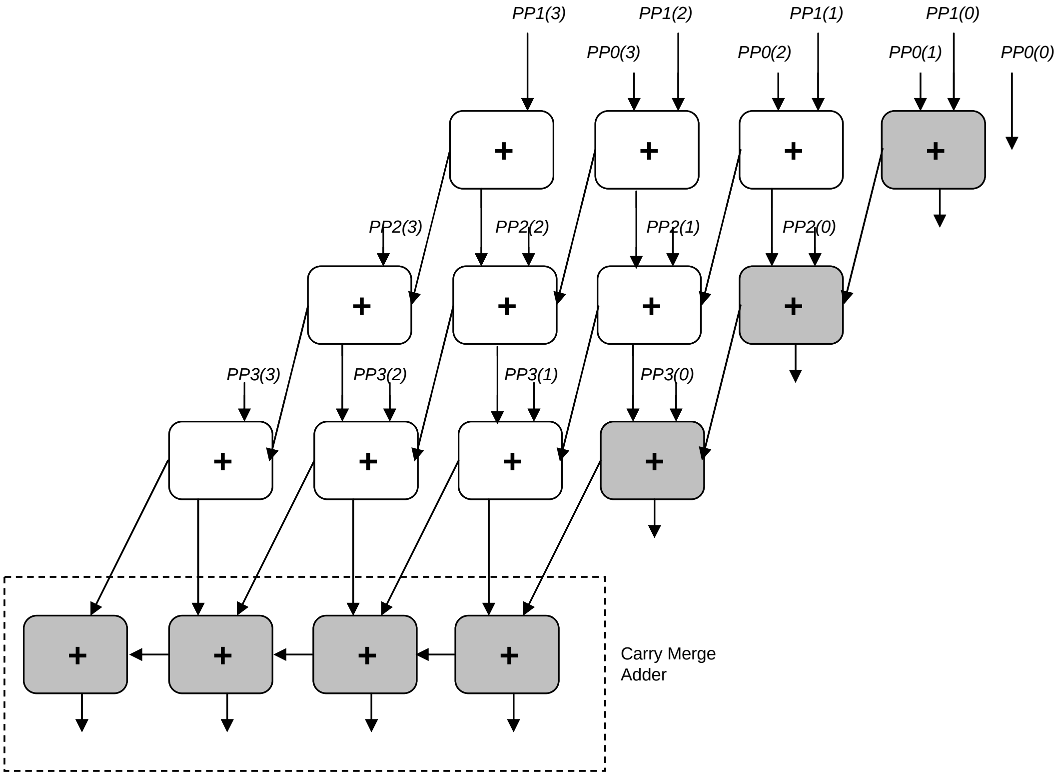


Figure 3-12: 4 x 4 carry-save multiplier (simplified)

Due to the carry-save multipliers gain in speed over the array multiplier and its regular structure, this multiplier scheme is preferred in many applications.

3.2.3 Wallace Tree Multiplier

C. S. Wallace propounded a fast technique to perform multiplication in 1964 [Wallace1964]. Although the amount of hardware required to perform this style of multiplication is large, the delay is near optimal. The delay is proportional to log(N) for column compression multipliers.

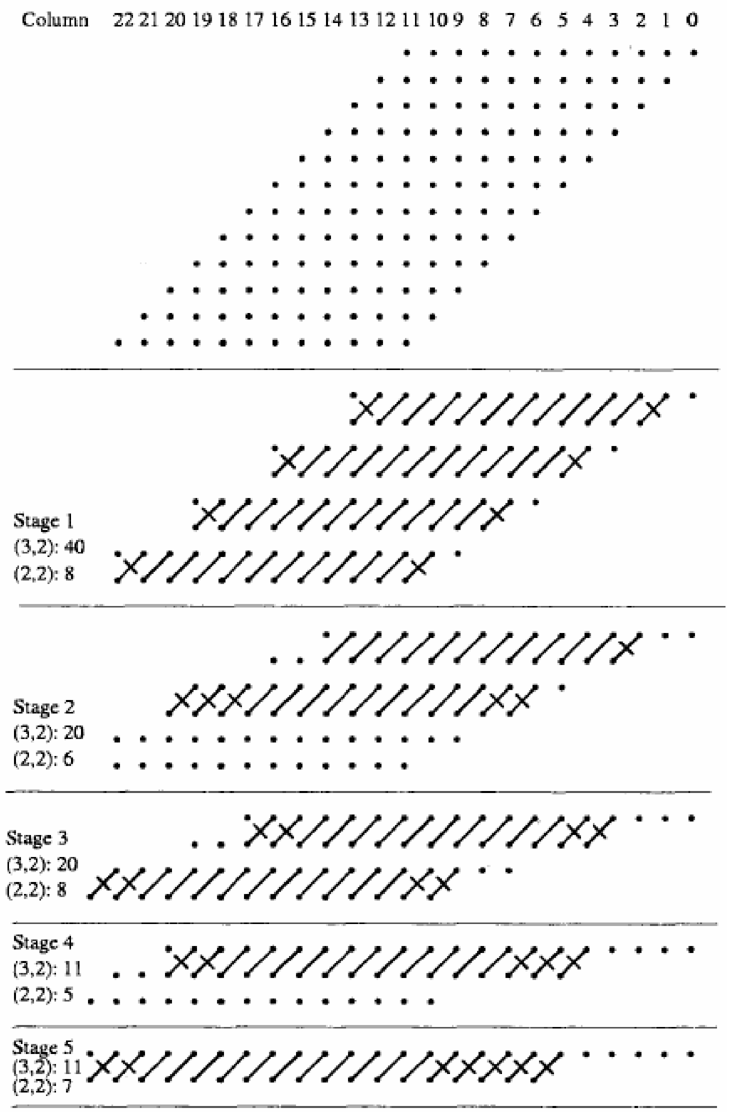


Figure 3-13: Dot diagram of a 12 x 12 Wallace tree multiplier showing column compression from one stage to next. Straight lines represents full adders or (3:2) compressors and the crossed line represent half adders or (2:2) compressors [Bickerstaff2001]

The Wallace tree multiplier belongs to a family of multipliers called column compression multipliers. The underlying principle in this family of multipliers is to achieve partial product accumulation by successively reducing the number of bits of information in each column using full adders or half adders. The full adder is known as a (3:2) compressor because of its ability to add 3 bits from a single column of the partial product matrix and output 2 bits, 1 bit in the same column and 1 bit in the next column of the output matrix. The half adder is known as a (2:2) compressor because of its ability to take 2 bits from a single column of the partial product matrix and output 2 bits, 1 bit in the same column and 1 bit in the next column of the output matrix.

The Wallace tree consists of numerous levels of such column compression structures until finally, only two full-width operands remain. These two operands can then be added using regular 2N-bit adders to obtain the product result. Figure 3-13 shows a 12x12 Wallace tree Multiplier that produces a 24-bit output. What differentiates the Wallace tree multiplier from other column compression multipliers is that in the Wallace tree every possible bit in every column is covered by the (3:2) or (2:2) compressors repetitively until finally the partial product matrix has a depth of only 2. Thus the Wallace tree multiplier uses as much hardware as possible to compress the partial product matrix as quickly as possible into the final product.

3.2.4 Dadda Multiplier

Dadda multipliers were first described by L. Dadda in 1965. [Dadda1965] They also belong to the column compression family of multipliers like the Wallace tree multiplier. Figure 3-14 shows a dot diagram of an 8x8 Dadda multiplier.

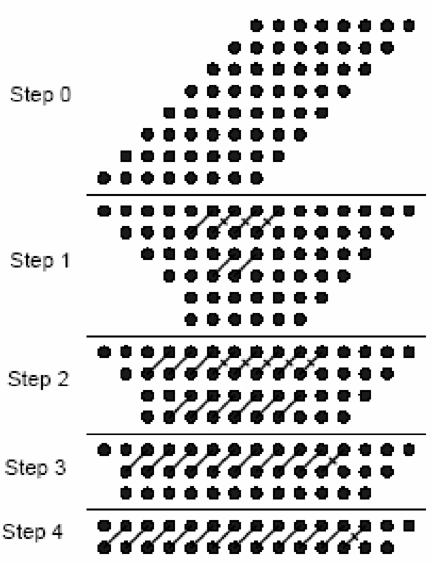


Figure 3-14: Dot diagram of the Dadda multiplier [Dadda1965]

The Dadda multiplier compresses columns differently from the Wallace-tree multiplier. Dadda observed that the maximum compression factor possible was 3/2, which is the compression factor of the full adder. The conclusion he drew was that instead of covering the whole partial product matrix with half and full adders, one only need cover enough to obtain the maximum compression of 2/3 of the current depth at each step in partial product accumulation. Thus fewer columns are compressed in the first few steps of the column compression tree, and more columns in the later levels of the multiplier.

The partial product depths in the Dadda tree follow a sequence 28, 19, 13, 9, 6, 4, 3 and 2. At each stage just enough compressors are used to obtain depth equal to the next number from the sequence above. Table 2 gives the number of stages (S) in the Dadda compression tree based on the operand bit-width of bits in multiplier N. At the last stage, a normal multi-bit adder is used to combine the remaining two compressed partial products into the final product output. This adder can be implemented using any of the addition techniques presented earlier.

Table 2: Number of stages in a Dadda Multiplier based on operand bit-width

Bitwidth of Multiplier (N)	# Stages (S)
2	0
3	1
4	2
5 to 6	3
7 to 9	4
10 to 13	5
14 to 19	6
20 to 28	7
29 to 42	8
43 to 63	9
64 to 94	10

The result of this compression scheme for the partial product matrix is that the delay of the structure is nearly equal to that of the Wallace tree multiplier, but with a smaller area. In fact for our simulations in the LSI Logic 0.11 micron technology, the Dadda multiplier is faster than the Wallace tree multiplier of the same bit-width, possibly due to the lower fan-outs of intermediate signals. The smaller area of the Dadda multiplier is from a reduced number and size of compressors. [Bickerstaff2001] mentions that for an 8x8 multiplier the Dadda tree requires 35 full adders and 7 half adders, whereas the Wallace tree requires 40 full adders and 15 half adders. Thus in our implementations, the Dadda tree outperforms the Wallace scheme in both area and timing. The Dadda multiplier is also faster and smaller than all other multiplier schemes that we examined of the same bit-width.

3.2.5 Comparison of Multipliers

Several multiplier structures were coded in Verilog, and synthesized in LSI Logic's gflxp 0.11 micron standard cell library using Synopsys Design Compiler. The area and timing results are given in Table 3. Two versions of the Dadda multiplier were examined, one using an RCA in the final stage, and one using a CLA in the final stage.

Table 3: The synthesis results for multiplier structures. Timing in ns and Area in square microns

Multipliers		
Design	**Timing**	**Area**
Array Multiplier 4	1.65	1012
Array Multiplier 8	3.95	4488
Array Multiplier 16	8.81	21236
Carry-Save Multiplier 4	1.56	1290
Carry-Save Multiplier 8	3.24	5052
Dadda4 using Ripple Carry Adder	1.12	878
Dadda8 using Ripple Carry Adder	2.7	4358
Dadda4 using Ripple Carry Adder	1.1	1430
Dadda8 using Carry Look-ahead Adder	2.17	5102
WallaceTree4	1.3	948
WallaceTree8	2.8	4970
8 bit Multiplier Default Synopsys Implementation	1.79	3512

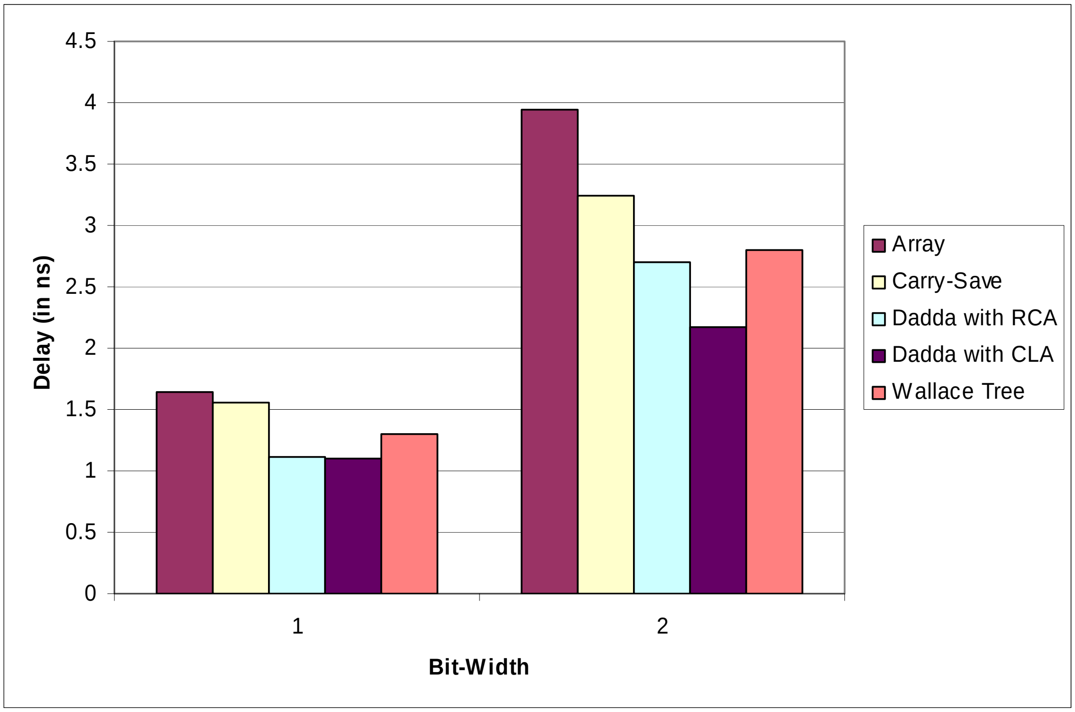


Figure 3-15: Multiplier delay vs. bit-width

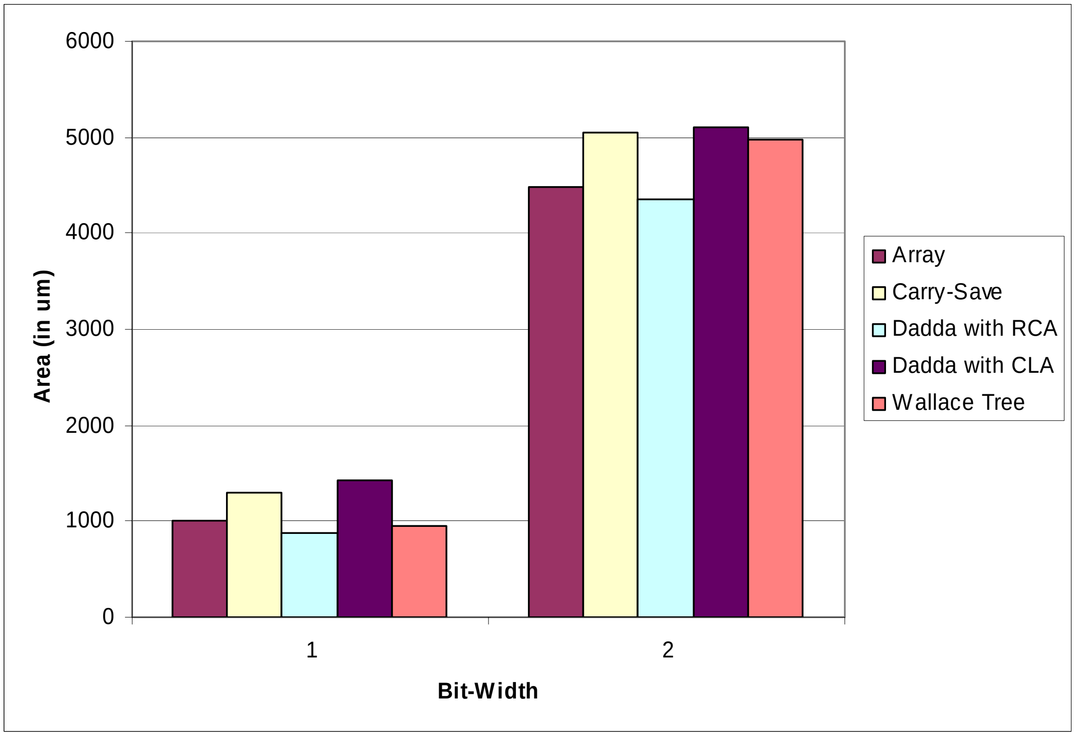


Figure 3-16: Multiplier area (square micrometer) vs. bit-width

The Dadda multiplier is the fastest design style tested, regardless of the structure of the final adder (RCA vs. CLA). However, the Dadda with a CLA is in fact faster than the Dadda with an RCA. The smallest multiplier by area is the Dadda multiplier with ripple carry adder. If one allows Synopsys Design Compiler to choose the multiplier implementation on its own we get the performance from the last row of the table above. Again, a special standard cell is used, with a structure not transparent to the designer.

4 Reconfigurable Arithmetic Block

This chapter details the methodology used to design our reconfigurable arithmetic block. The steps in the process are explained and the motivations behind some of the engineering decisions are highlighted. Some of the key questions when designing a reconfigurable arithmetic block are as follows: What operations should be supported? What schemes should be used for implementing each of those operations? What should be the bit-widths for the chosen operations? How can the flexibility of the arithmetic block be improved? What sort of a routing architecture should be provided? We discuss these issues in this chapter.

4.1 Supported Arithmetic Operations

As we are targeting Digital Signal Processing (DSP) applications, addition, subtraction and multiplication obviously need to be supported. Logical operations such as ANDing, ORing, EX-ORing were chosen as they might be needed in some DSP applications and in cryptography. Comparison operations such as “==” and “<” are also supported...

In order to choose the method used to perform arithmetic calculations, first a survey of computer arithmetic techniques was performed, as summarized in Chapter 3. Candidate techniques for addition and multiplication were chosen based on the survey. For addition we compared ripple carry adder, carry look-ahead adder, and the Han-Carlson adder. For multiplication, we compared array multipliers, carry-save multipliers, Wallace tree multipliers and Dadda multipliers. These comparisons were made on an area and delay basis for a variety of bit-widths.

As mentioned in the previous chapter, the fastest multiplier tested is the Dadda multiplier with a carry-look-ahead (CLA) in the final stage. It is also the most area-efficient scheme when used with a ripple carry adder (RCA). Because we wished to emphasize speed, we chose the Dadda with CLA implementation. The CLA was also chosen as the base adder type given its speed.

A survey of DSP applications was then performed to figure out the bit-widths required for each operation. Discrete Cosine Transform (DCT), Finite Impulse Response (FIR) filtering, Fast Fourier Transform (FFT), Pulse Code Modulation (PCM), JPEG Image Compression, 2D Image Filtering, MPEG Video Coding, MPEG-3 Audio, encryption, and AAC were some of the applications that were examined. It was determined that the bit-width used for many of the above operations is 8 bits. For instance, with video and image processing applications, each of the color streams (Red Green Blue (RGBA) Alpha) is 8 bits wide. The intermediate operations that are performed on these streams might need a slightly larger precision. For JPEG and other applications using DCT, some internal information is stored using 12 bits. Still, the majority of the data observed was 8 bits wide. We therefore concluded that 8 bits should be our minimum supported bit-width, and that our structure should be able to chain computational structures together to provide support for larger bit-widths.

4.2 Scaling Bit-widths

Scaling the bit-width of addition operations is a trivial problem. The carry signals need to be propagated between two adder blocks to achieve larger addition. For 2’s complement subtraction the result is slightly less straightforward. In this case, the carry input to the least-significant block needs to be high, but for adder units higher in the chain, the carry signal needs to be propagated from the previous block. Hence the facility to choose between various carry input alternatives (low, high, carry-in signal) is required and provided in our blocks.

Scaling of multiplication is more involved than addition or subtraction. For instance, if we were to perform a 2NxN-bit multiplication using NxN-bit multipliers the structure in the following figure is required. Assume that we are trying to multiply two numbers A (2N-bit) and B (N-bit). We decompose A and B in terms of N-bit digits, where B has a single N-bit digit, and A has two N-bit digits. Let $\mathbf{A_{LS}}$ denote the N least significant bits of A (lower half), $\mathbf{A_{MS}}$ denote the N most significant bits of A (upper half). Let us look at the multiplication of two such numbers on paper. We observe that there are two partial products formed $\mathbf{A_{LS}B}$ and $\mathbf{A_{MS}B}$. $\mathbf{A_{MS}B}$ belongs N-bits to the left of $\mathbf{A_{LS}B}$ in this N-bits-per-digit multiplication.

$$\begin{array}{r}
A_{MS} \quad A_{LS} \\
\times \qquad\qquad B \\
\hline
A_{LS}B \\
+\, A_{MS}B \quad 0 \\
\hline
\text{Product}
\end{array}$$

Figure 4-1: A 2NxN multiplier can be built from two NxN multipliers by examining the multiplication process in terms of N-bit digits.

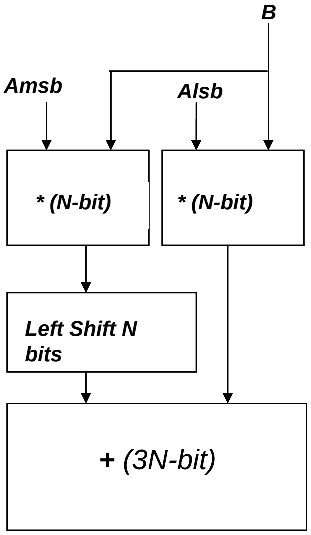


Figure 4-2: 2NxN-bit multiplication using two N x N-bit multipliers

Now, assume that we are trying to multiply two 2N-bit numbers A and B. Divide both A and B into two N-bit digits each, $\mathbf{A_{LS}}$, $\mathbf{A_{MS}}$, $\mathbf{B_{LS}}$ and $\mathbf{B_{MS}}$. Similar observations lead to the following method for implementing 2Nx2N-bit multiplication using NxN-bit multipliers. This time we need to effectively shift the $\mathbf{A_{MS}B_{MS}}$ partial product by 2N bits to the left, and both the $\mathbf{A_{MS}B_{LS}}$ and the $\mathbf{A_{LS}B_{MS}}$ partial products by N bits to the left. The resulting structure is shown in Figure 4-3. One other observation can be made at this point. The bit-widths of the adders increase progressively as more shifting is performed. For instance the scaling in Figure 4-2 requires a 3N-bit adder, while the scaling in Figure 4-3 requires 2 3N-bit adders and 1 4N-bit adder. The delay of the overall circuit is also affected by the scheme used for addition.

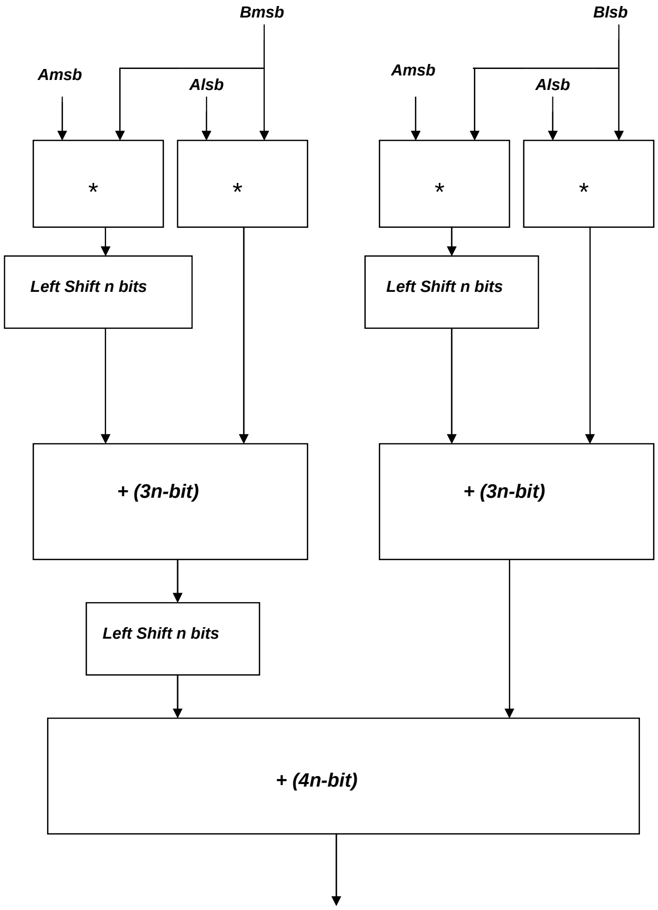


Figure 4-3: 2N x 2N-bit multiplier using N x N-bit multiplier

4.3 Block 1

Block 1 was our first attempt at designing a reconfigurable arithmetic block. It contains custom multiplier and ALU circuitry optimized for 8x8 multiplies or 16-bit additions. We use the adder and multiplier designs chosen: Dadda and CLA. We noted that the Dadda multiplier contains an adder in its last stage to add the two final values emerging from the partial product compression tree. Since we would like our block to perform either a multiplication or one or more additions, we decided to pull adder structures such as this out of the Dadda multiplier. The multiplier is no longer a “true” Dadda multiplier, but the hope is to retain the majority of the benefits of this style while making the hardware used for the final steps also useable as stand-alone adders.

To form Block 1, let us change the Dadda column compression tree and remove

its final step so that three values emerge instead of two. The sum of these three values is the product. We perform this calculation by using two carry look-ahead adders (CLA). The reduced 8x8 Dadda tree, coupled with the two 16-bit CLAs performs the function of an 8x8-bit multiplier.

We also add multiplexers M1, M2, M3 and M4 at the inputs of the adders so that two operands can be supplied to each of the CLAs independent from the Dadda tree. Another multiplexer is added at the second CLA's carry input to select between the carry out signal from the first adder or an external carry input Cin2. This allows the two 16-bit adders to either be used separately or to easily be cascaded to form a 32-bit addition. However, these adders are only available when not performing a multiplication with this block. Block 1 is intended to represent the smallest possible block given our previously-discussed design choices. The tradeoff for this area, however, is a potential need for many such blocks. For example, a multiply-accumulate operation cannot be contained within a single block. The multiplication must be performed in one block, and the addition in another.

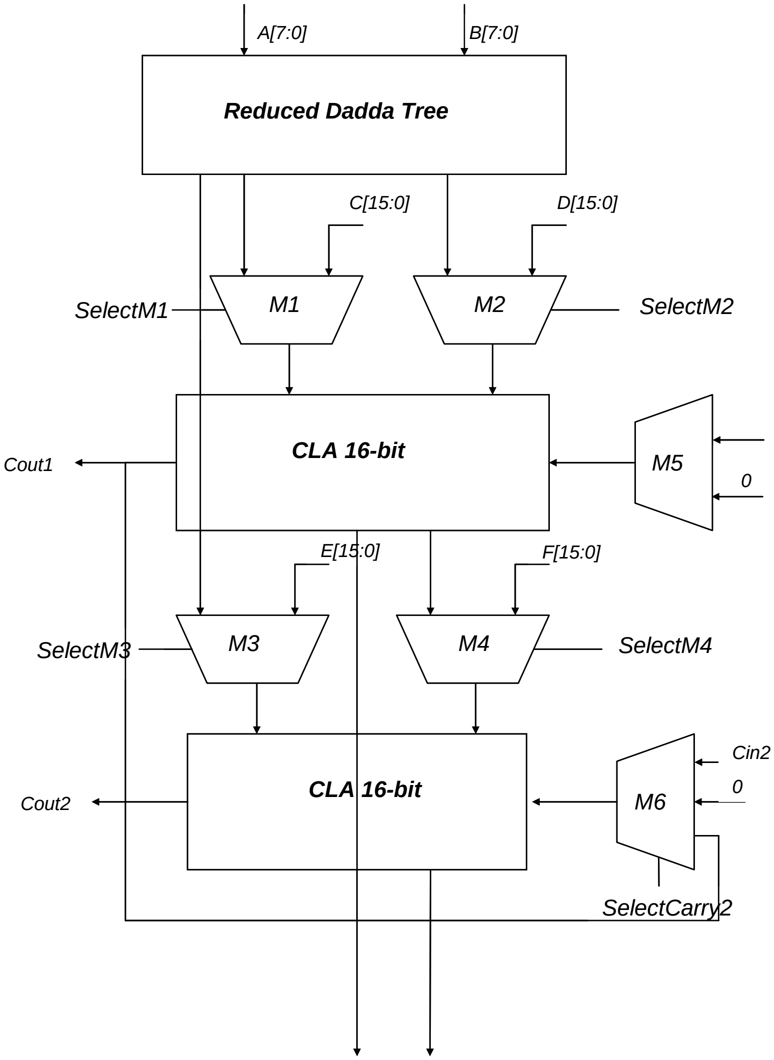


Figure 4-4: Block 1 can be used as either an 8x8 bit multiplier, two 16-bit adders, or one 32-bit adder.

4.4 Block 2

In Block 2 the complete Dadda partial product compression tree is included in the design. The addition of the final two partial products is performed using a CLA that is alternately available for independent addition or subtraction. Multiplexers M1 and M2 allow us to select the inputs to the CLA from either the outputs of the Dadda tree or external block inputs. Because there are fewer multiplexers on the multiplication path, we expect this design to be faster than Block 1 for an 8x8 multiplication, and only one 2:1 multiplexer delay slower than a standard 8x8

Dadda multiplier. We add another CLA that is not used in the Dadda multiplication, and is therefore available for independent calculations, or to aid in multiply-accumulate operations.

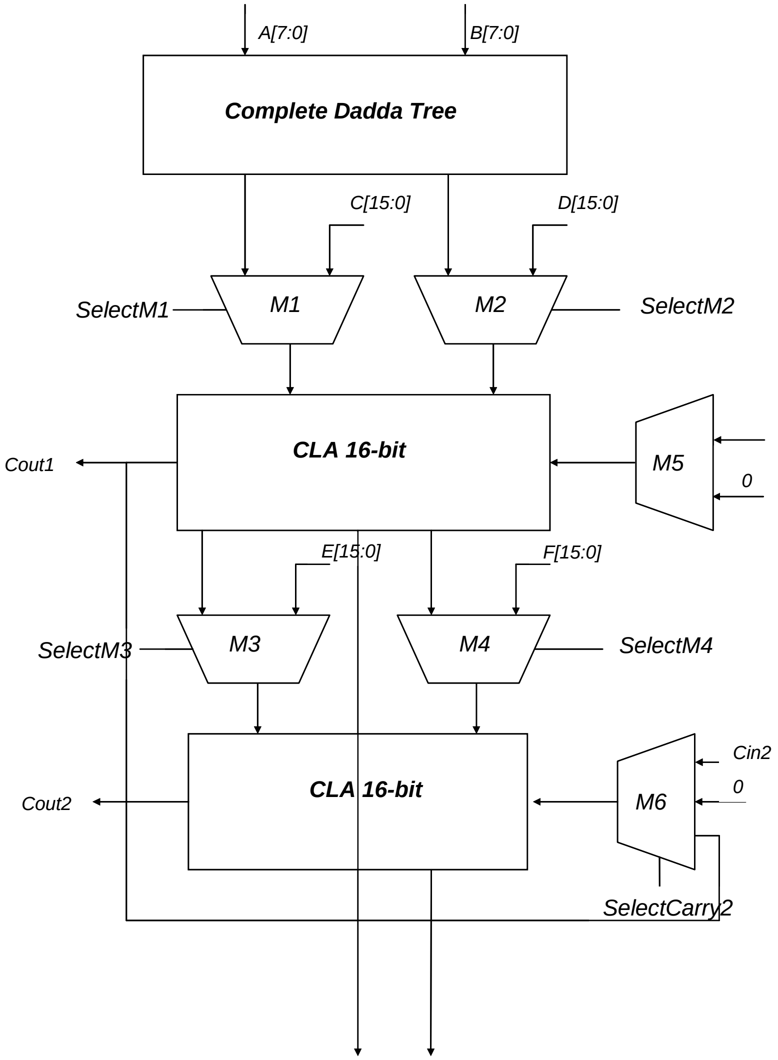


Figure 4-5: Block 2 contains the complete Dadda compression tree and two 16-bit CLAs. Note that the multiplication now completes in the top adder.

4.5 Synthesis Results for Block 1 and Block 2

Both blocks were implemented in Verilog and synthesized using Design Compiler for gflxp 0.11 micron standard cell library. The results are given as follows:

Table 4: Synthesis Delay Results for Blocks 1 and 2

Delay in ns	ASIC	Block1	Overhead Block1	Block2	Overhead Block2
8 x 8 Multiplier	2.17	2.74	1.262672811	2.23	1.02764977
Top 16-bit Adder	1.09	1.21	1.110091743	1.26	1.155963303
Bottom 16-bit Adder	1.09	1.21	1.110091743	1.21	1.110091743
32-bit Adder	1.48	1.73	1.168918919	1.73	1.168918919

Table 5 Synthesis Area Results for Blocks 1 and 2

Area	Block1	Block2	Dadda 8x8	CLA16	CLA32
Total Logic Block Area	8551	9477.4	5102	1792	3584.58
Dadda Specific Area	3521	4447			
Multiplexer Area	1446	1446			

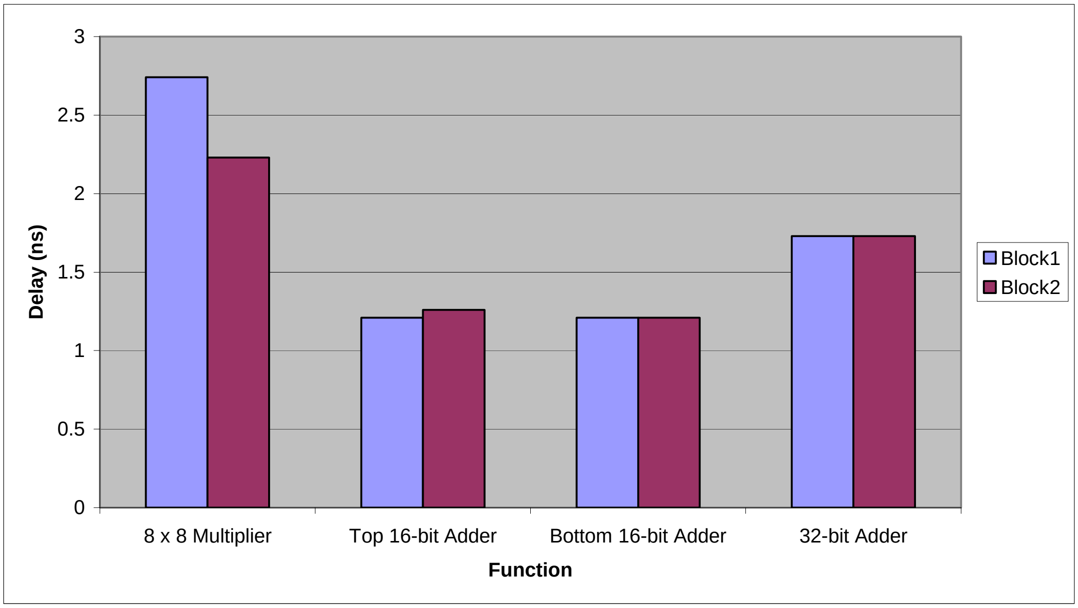


Figure 4-6: Delay results for both blocks. Block 2 is faster for multiplication.

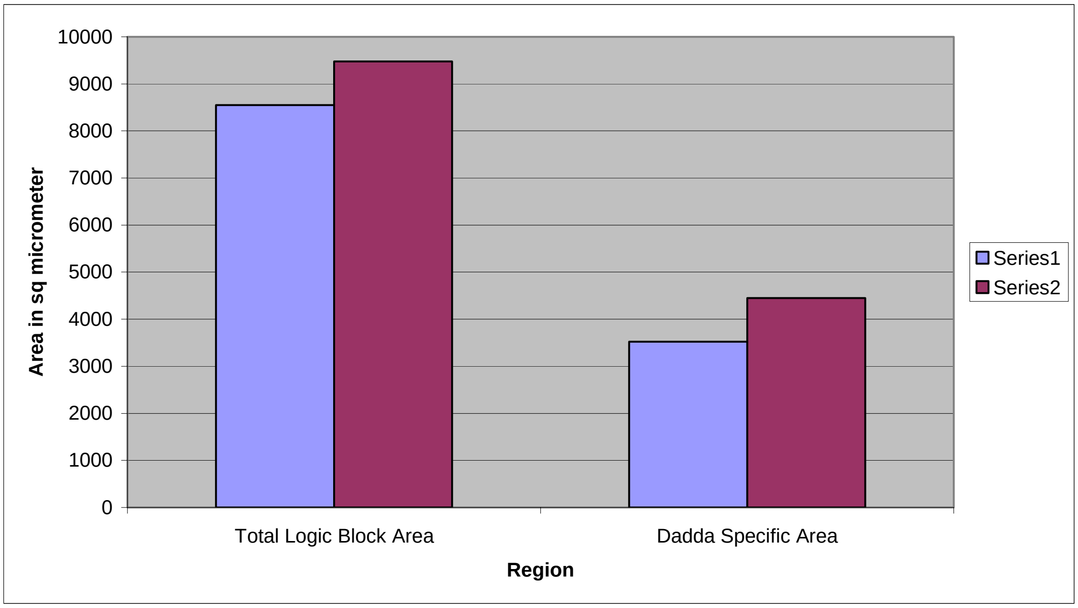


Figure 4-7: Area results for Block 1 and Block2. Block 1 is smaller.

One might expect the 32-bit CLA on both Block 1 and Block 2 to require two times the delay of the 16-bit CLA as we are simply propagating the carry from one 16-bit CLA to the next. However, during simulation it was observed that the carry-out signal from a CLA is produced much faster than the sum value. Hence the addition operation in the second CLA can begin early and the result of the 32-bit addition is obtained faster. The top 16-bit CLA is slower than the bottom 16-bit CLA for Block 2 because of the higher fan-out on the sum node of the top 16-bit CLA. It can be seen that in Block 2 the top CLA is driving 3 buses, whereas the top CLA in Block 1 is driving only one bus.

The results, as expected, show that Block 1 has a smaller area, though only by a slight margin. The speed of Block 2 for multiplication is 17% faster than Block 1. The increase of the Dadda Specific Area in Block 2 in Figure 4-7 is simply because only one addition was pulled out of the multiplier instead of two as in Block 1. The actual area devoted to multiplication is nearly identical, and the small increase in Block 2 area is mainly from the additional 16-bit CLA. We choose the gain in speed for multiplication despite the very small increase in area, and use Block 2 as the basis for the next design iteration.

4.6 Block 3

As stated previously, Block 2 was used as the basis for the design of Block 3. Block 3, pictured in Figure 4-8, is intended to support more of the most commonly-used operations than the previous block versions. For example, in order to support logical and comparative operations, we convert the stand-alone CLA of Block 2 to a full 16-bit ALU for Block 3. We can also still perform a 32-bit addition by cascading the CLA and the ALU together.

Unlike Block 2, Block 3 supports subtraction. We add a multiplexer to allow us to select between the inverted and un-inverted value of one of the operands based on the "Subtract" select signal. When Subtract = "1", the inverted value of B is given to the CLA and when Subtract = "0" the un-inverted value of B is given to the CLA. Secondly, the carry-in multiplexer has another input, a constant 1, as the carry input of the CLA should be 1 if it is the least significant subtractor. If the CLA is not the least significant subtractor then the carry from the previous subtractor should be used as the carry-in. The ability to sign-extend 8-bit values into 16-bit values and 24-bit values into 32-bit values has also been incorporated and can be selected by choosing the appropriate inputs on the multiplexers.

Registers were added next to allow for pipelining to achieve higher clock frequencies. The registered or non-registered values can be selected using the appropriate multiplexers. Hence flexibility for circuit pipelining has been provided. The registers are represented by squares in Figure 4-8.

Using the scaling scheme from Chapter 4, we can perform a 16x16 multiplication efficiently in Block 3. We notice that a 24-bit adder is needed in the first stage for the two 8 x 8-bit multiplier. At the same time, in both Block 2 cells the second 16-bit CLA is lying unused. Due to the facility for sign extension provided, we can add/subtract the 24-bit numbers in the two idle CLAs, by propagating the carry from the first CLA to the next and getting the first 16 bits of the result from the CLA in the block on the right and the next 8 bits from the CLA in the block on the left.

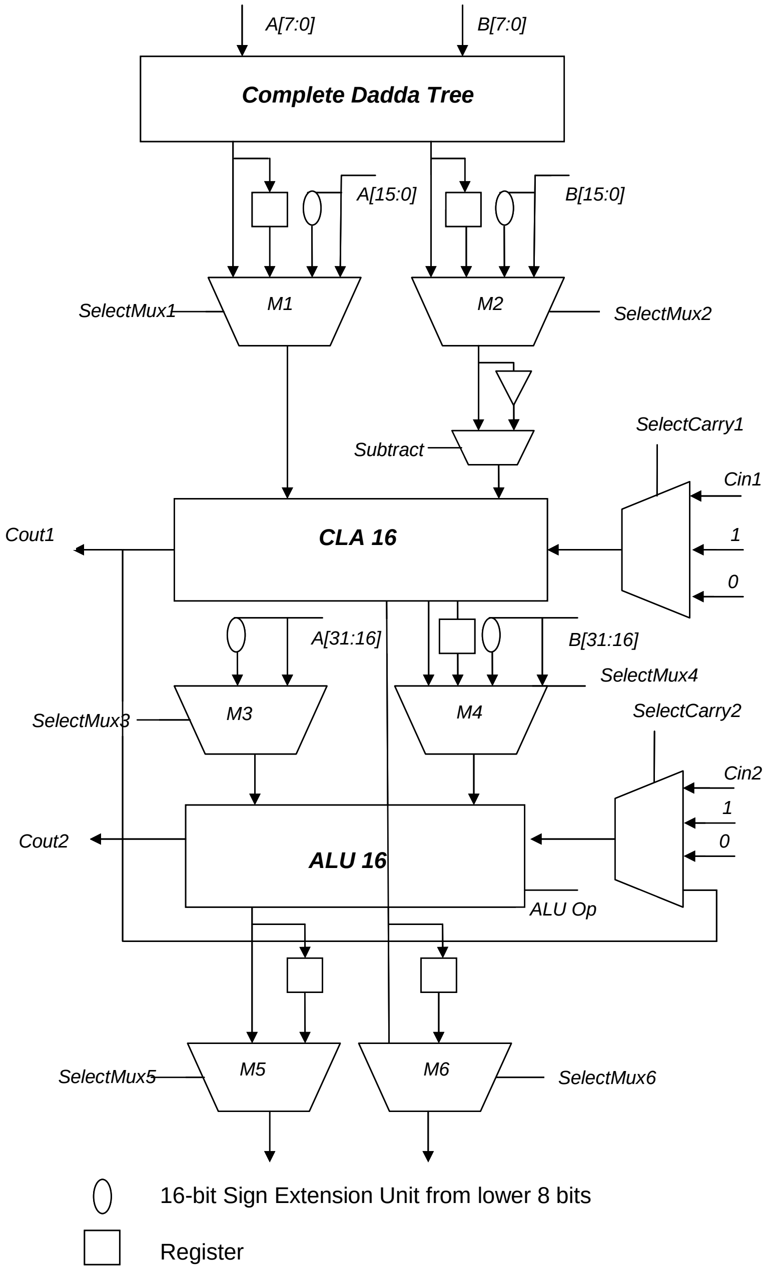


Figure 4-8: Block 3 is based on the design of Block 2, but contains additional flexibility, particularly in the form of multiplexers.

4.7 Synthesis Results for Block 3

After adding the features mentioned in the previous chapter the synthesis results are as follows for operations in Block 3:

Table 6: Timing results for all blocks in ns

		Delay			Overheads		
	ASIC	Block1	Block2	Block3	Block1	Block2	Block3
8 x 8 Multiplier	2.17	2.74	2.23	2.51	1.15668	1.262673	1.02765
Top 16-bit Adder	1.09	1.21	1.26	1.49	1.36697	1.110092	1.155963
Bottom 16-bit Adder	1.09	1.21	1.21	1.75	1.6055	1.110092	1.110092
32-bit Adder	1.48	1.73	1.73	2.75	1.85811	1.168919	1.168919

Table 7: Area results for all blocks in square microns

Area	Block1	Block2	Block3	Dadda 8x8	CLA16	CLA32
Total Logic Block Area	8551	9477.4	17,579	5102	1792	3584.58
Dadda Specific Area	3521	4447	4447			
Multiplexer Area	1446	1446	2391			

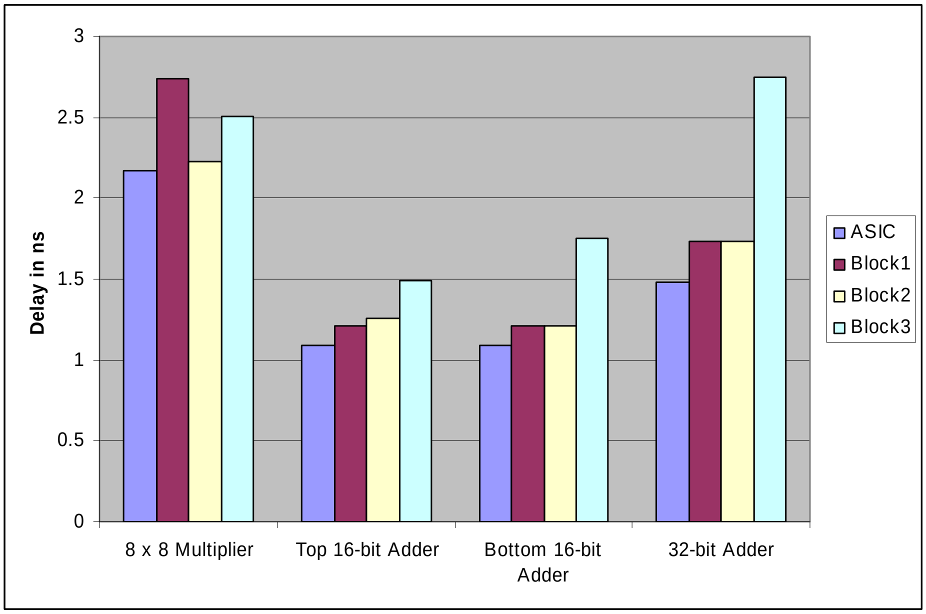


Figure 4-9: Delay Results for all Blocks

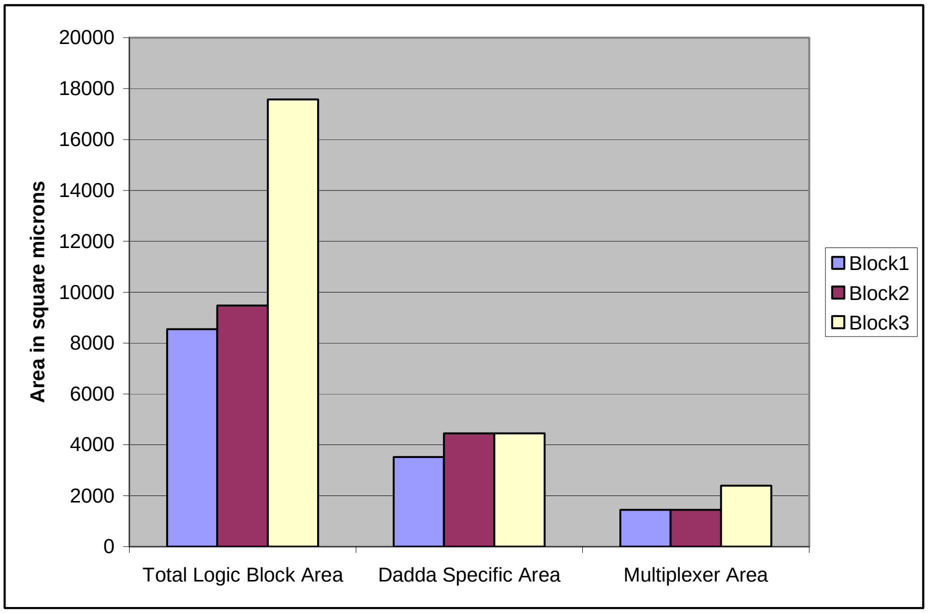


Figure 4-10: Area results in square microns

Block 3 is slower than other blocks for most operations, though is still faster than Block 1 for 8x8 multiplication. The addition operations are slower due to the presence of more/larger multiplexers in the top 16-bit CLA and also likely in great part due to the bottom CLA being converted to an ALU. The majority of the area increase of Block 3 over Block 2 is due to the ALU and the registers and multiplexers needed for optional registering. However, Block 3 can now perform many more practical functions and support pipelining, which is critical for high-performance DSP applications.

4.8 Synthesis Results for LUT-Based FPGA Implementations

We compared our Block 3 speed results with implementations in the Xilinx Spartan-3 device in order to measure the effectiveness of custom structures over LUTs. Because the Spartan-3 is fabricated in a 0.09 micron technology, we have scaled the Spartan-3 results by $(.11)^2/(.09)^2$. We chose the fastest speed grade (-6) for Spartan-3 devices. However, since our results are from standard cell implementations, and we expect our arithmetic array to eventually be implemented in a full-custom layout, we have scaled our results by a factor of 2, a commonly-accepted difference between standard cell and manual implementations [Stone1996].

Table 8: Comparison of Block 3 and Spartan-3 implementations. Scaling factors of 0.5 for standard cell to custom layout change and 1.493 for technology scaling are used.

	Block3	Block3 Scaled	Spartan 3	Spartan Technology Scaled	SpeedUp
Multiply 8x8	2.51	1.255	20.749	30.9948562	24.6971
Add 16-bit	1.49	0.745	15.537	23.2091706	31.15325
Add 32-bit	2.75	1.375	18.975	28.344855	20.61444

Therefore, considering only the logic delay of Block 3, our implementations of the arithmetic structures are expected to be about 20-30 times faster than LUT-based implementations. Because these are operations that are completely contained within a single Block 3, no external routing delays (apart from to external I/Os) would apply. We have not yet finalized the routing structure for Block 3, which would affect results for arithmetic operations requiring more than one block. This is an area we will be investigating in the future.

4.9 Application Placement

In order to have a better comparison of the speeds between various technologies, we implemented an 8-point 1D DCT and the FFT Butterfly operations in various technologies, as shown in Table-9. Scaled delay values with assumptions similar to the previous section are given in Table-10. Finally speedup values calculated are shown in Table-11.

Table 9: Non-scaled speed (ns) and area (sq um) results for 8 point DCT and FFT-butterfly

Speed (ns)			Unscaled	
	Technology		DCT	FFT
Spartan 3	0.09 micron	LUT-based	33.417	30.942
Block 3	0.11 micron		8.24	5.49
Semi-custom ASIC	0.11 micron		3.82	3.14
Virtex-II	0.15 micron	Multipliers and	25.834	20.188

Table 10: Scaled speed values for 8 point DCT and FFT-Butterfly

	Technology	DCT	FFT
Spartan 3	0.09 micron	49.891581	46.19641
Block 3	0.11 micron	4.12	2.745
Semi-custom ASIC	0.11 micron	1.91	1.57
Virtex-II	0.15 micron	13.89671867	10.8596

Table 11: Speedup factors of Block 3 over other implementations

	Technology	**DCT**	**FFT**
Spartan 3	0.09 micron	12.1096068	16.82929
Block 3	0.11 micron	1	1
Semi-custom ASIC	0.11 micron	0.463592233	0.571949
Virtex-II	0.15 micron	3.372990291	3.956139

The standard cell implementation of both the 8-point DCT and the FFT butterfly are faster than all other implementations as expected. Standard cells as seen to be almost two times faster than Block 3 implementations. LUT-based implementations on Spartan-3 devices are the slowest for both applications by very large factors to the order of 15. Virtex-II implementations with multipliers and LUTs are about 3 times slower than Block 3. These results are presented with the caveat that a routing structure will create some additional delay in the Block 3 implementations.

5 Future Work

Although we feel that our Block 3 design is well-suited for arithmetic computations, there are a number of different directions we plan to examine in future work. These directions include the design of the reconfigurable fabric beyond the logic structure, improvements to the block itself, and examining implementation options for our logic-level designs.

5.1 Routing

Block 3 contains a great deal of logic, but we have not yet defined the routing structure that connects these blocks together. We expect to include some nearest-neighbor style routing to facilitate fast operations on larger bit-widths, but we also need a more general routing network. This work focuses on coarse-grained structures, so we expect the routing architecture to be primarily coarse-grained as well. The smallest data size we deal with in our block is 8 bits. Therefore, it is a natural extension to focus on a routing structure that routes signals in groups of 8 bits.

The choice of an 8-bit routing granularity was also motivated by examining the multiplication process for greater than 8x8 multiplies. To be able to perform 16 x 16-bit multiplication with the scaling technique mentioned earlier, we require a 24-bit adder, a 32-bit adder and an 8-bit shifter in addition to the 8x8-bit multipliers. Block 3 can achieve all these functions mentioned above except for an 8-bit shift. However, we expect to be able to implement this shift using the routing fabric to route groups of 8 bits to either the lower or upper half of the 16-bit registers, with 8 bits of 0 supplied to the other half though additional multiplexer inputs.

Each external input of Block 3 will be fed by an additional multiplexer in the connection block, such as the one shown in Figure 5-1. The connection block is the part of the routing fabric that implements the ability to connect a block input to the routing fabric outside the block. In some cases we may also need the ability to sign-extend certain input values, which has been provided in the block’s internal multiplexers. The specifics about the number of wires, the degree of connection (Figure 5-1 shows a fully-populated connection block), the hierarchy of wiring and such other details will be covered in the future.

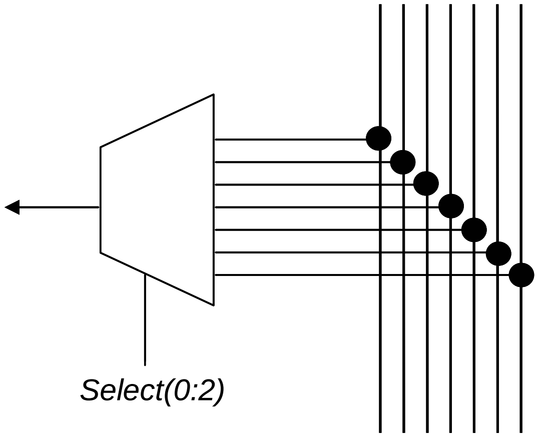


Figure 5-1: Each logic block input can connect to the routing fabric outside the logic block through a multiplexer in the connection block. All wires are 8 bits wide except for Select line

5.2 Logic Block Improvements

Support for signed multiplication is another issue that needs to be tackled in the future, although it will be relatively easy to implement. In signed multiplication, the absolute value of the two operands A and B are multiplied, and the sign of the product is determined based on the original signs of the inputs using an exclusive-OR operation. We may also require additional circuitry for larger multiplications, and will also need to be able to find the absolute value of numbers greater than will fit in a single block. It is likely this will therefore require additional routing structures for 1 bit "control" values, as well as some smaller bit-level logic support for operations such as the required exclusive-OR of two 1-bit values.

Some amount of general-purpose LUT-based logic may be required for efficient implementation of some DSP computations, which could lead either to a more complex block structure, or separate LUT-based blocks in a heterogeneous fabric.

Finally, we have not yet specified which of the control signals are dynamic, and which are programming bits in SRAM. Signed multiplication is one of the operations that will guide this decision, as Block 3 can implement an absolute value function if LUT elements can dynamically control some of the multiplexer select lines such as Subtract and Carry In.

5.3 Leakage Power Dissipation

One of the common complaints about FPGAs is that they dissipate a lot of power compared to ASIC implementations. We would like to incorporate features for minimizing stand-by leakage power dissipation in this new reconfigurable arithmetic block, such as when a given block isn't being used by the current circuit. Two schemes were examined for the same. The first from [Mutoh1995] is called Multi-Threshold CMOS (MTCMOS) and essentially adds a sleep transistor next to the supply rail connection to the circuits shown in Figure 5-2. As the sleep transistor is a high-Vt device while the rest of the circuit contains low-Vt devices, the leakage power in the whole circuit is reduced. In another paper [Mutoh1996], a DSP processor designed in this technology was found to consume 1/10th the power of a conventional circuit designed in the same technology but without power management. This technique can be applied to static or dynamic CMOS circuits but has the drawback that the speed of the logic circuitry overall goes down because of additional transistors in charging and discharging paths.

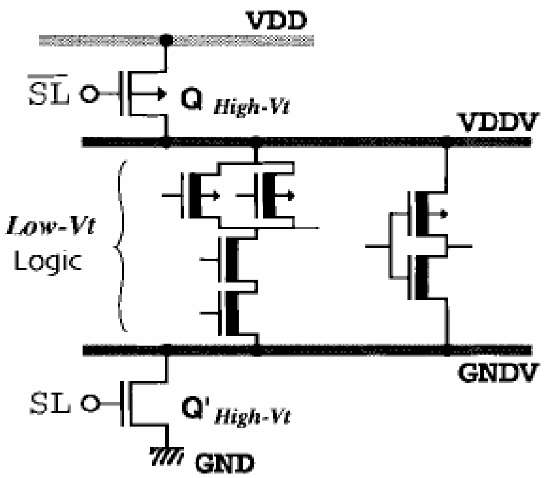


Figure 5-2: Multi-Threshold CMOS circuit with sleep transistors made from high-Vt transistors located next to VDD and GND lines. The Virtual VDD (VDDDV) and Virtual GND (GNDV) wires are also shown. [Mutoh1995]

A second scheme [Kursun2004] that can be applied to domino logic circuits is shown in Figure 5-3. The advantage of this scheme is that only one sleep transistor needs to be added in the first stage of the pipeline to pull all the subsequent circuits into sleep mode. When the sleep transistor has a high input value the output node will be pulled high. Thus the next stage's intermediate node will be pulled low and its output again pulled high due to the inverter, because in domino logic there is no foot transistor for the following stages. This same phenomenon propagates through all the domino circuits connected to each other. As we begin to implement our design in a full-custom layout, this is the technique that we expect to use.

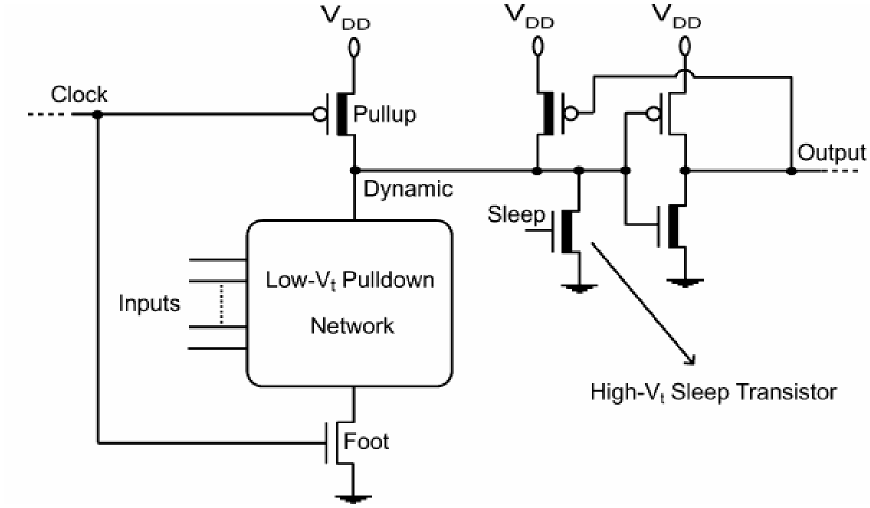


Figure 5-3: High-Vt Sleep Transistor mode to reduce leakage power dissipation in dual-Vt dynamic circuits. [Kursun2004]

6 Conclusions

We examined three different arithmetic logic blocks, and compared these designs based on area and performance. We found that the added functionality in Block 3 for optional registering, sign extension, ALU operations comes at the cost of more than doubling the area over Block 2, but provides necessary functionality for today's high-performance applications. Also due to the additional multiplexers present in Block 3, all arithmetic operations are slower than other implementations of the same operations in Blocks 1 and 2. Block 3 performs 20-30 times better than LUT-based implementations of the arithmetic operations. Thus the arithmetic arrays will likely enhance the performance of many DSP applications, as noted by the speedup factors for FFT and DCT over commercial LUT-based reconfigurable devices. Further investigations with logic block improvements, routing fabric design, and custom design with schemes for minimizing leakage power dissipation will help us realize the full potential of this new arithmetic logic block.

Bibliography

[Altera2004] Altera, *Stratix Device Handbook*, Altera, San Jose, CA. September 2004.

[Altera2005] Altera, *Stratix II Device Handbook*, Altera, San Jose, CA. January 2005.

[Bickerstaff2001] K.C. Bickerstaff, E.E. Swartzlander, Jr., and M.J. Schulte, "Analysis of Column Compression Multipliers", *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, IEEE Computer Society Press, pp. 33-39, June 2001.

[BrentKung1982] R.P. Brent and H.T. Kung, “A Regular Layout for Parallel Adders”, *IEEE Transactions on Computers*, Vol. C31, No. 3, p. 260-264, March 1982.

[CNagendra1994] C. Nagendra, et al, “Power-Delay Characteristics of CMOS Adders”, *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, Vol. 2 No. 3, September 1994.

[Compton2002] K. Compton, A. Sharma, S. Phillips, S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", *International Conference on Field Programmable Logic and Applications*, 2002.

[Compton2001] K. Compton, S. Hauck, "Totem: Custom Reconfigurable Array Generation", *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.

[Dadda1965] L. Dadda, “Some Schemes for Parallel Multipliers”, *Alta Frequenza*, Vol.34, p.349-356, March 1965.

[Dehon1999] André DeHon. “Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)”. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 125-134, February 1999.

[HanCarlson1987] T Han, DA Carlson, “Fast Area Efficient VLSI Adders”, *Proceedings of the 8th Symposium on Computer Arithmetic*, pp. 49-56, 1987.

[KoggeStone1973] P.M. Kogge and H.S. Stone, “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations”, *IEEE Transactions on Computers*, Vol C-22, No. 8, pp. 786-793, 1973.

[Kursun2004] V. Kursun, E. Friedman, "Sleep Switch Dual Threshold Domino Logic with reduced stand-by leakage current", *IEEE Transactions on VLSI systems,* Vol.12 No. 5, May 2004.

[Marshall1999] A. Marshall, J. Vuillemin T. Stansfield, I. Kostarnov and B. L. Hutchings, "A Reconfigurable Arithmetic Array for Multimedia Applications", *Proceedings of the 1999 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey (CA), 135--143, Feb. 1999.

[Mutoh1995] S Mutoh et al, "A 1-V high-speed MTCMOS circuit scheme for power-down application circuits", *IEEE Journal of Solid-State Circuits*, Vol. 32, No. 6, June 1995

[Mutoh1996] S. Mutoh et al**,** "A 1-V multithreshold-voltage CMOS digital signal processor for mobile phone application**",** *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 11, Nov. 1996

[SKMathew2001] Sanu K Mathew et al, "Sub-500 ps 64b ALUs in 0.18 um SOI/Bulk CMOS: Design and Scaling Trends", *IEEE Journal of Solid-State Circuits*, Vol 36, No 11, pp. 1636-1646, 2001.

[Stone1996] G. O. Stone, A Comparison of ASIC Implementation Alternatives, M.S. Thesis, Northwestern University, Dept. of ECE, October, 1996.

[Rabaey2003] Jan M Rabaey, Anantha Chandrakasan, Borivoje Nikolic, "Digital Integrated Circuits", Second Edition, 2003.

[VOklobdžija2001] V Oklobdžija, "High Speed VLSI Arithmetic Units: Adders and Multipliers" in *Design of High-Performance Microprocessor Circuits*, A. Chandrakasan, W. Bowhill, F. Fox, IEEE, 2001.

[Wallace1964] C.S. Wallace, "A Suggestion for a Fast Multiplier", *IEE Transactions on Electronic Computers*, EC-13, p.14-17, 1964.

[Weinberger1956] Weinberger, "A One-Microsecond Adder using One-Megacycle Circuitry", *IRE Transactions on Electronic Computers*, 1956.

[Xilinx2001] Xilinx, Inc.. *Virtex-II 1.5V Field Programmable Gate Arrays: Advance Product Specification.* Xilinx, Inc., San Jose, CA, 2001.

[Xilinx2004] Xilinx, Inc., *Virtex-IV 1.5V Field Programmable Gate Arrays: Advance Product Specification.* Xilinx, Inc., San Jose, CA. 2004.