Coarse Grain Carry Architecture for FPGA

Hyuk-Jun Lee and Michael Flynn

Technical Report : CSL-TR-99-780

February 1999

This research has been supported by the Dept. of the Army under contract DABT63-96-C-0106

Coarse Grain Carry Architecture for FPGA

by

Hyuk-Jun Lee and Michael Flynn

Technical Report : CSL-TR-99-780

February 1999

Computer Systems Laboratory

Stanford University

Gates Building 3A, Room 230

Stanford, California 94305

Email: hyukjunl@umunhum.stanford.edu

Web: http://umunhum.stanford.edu

Abstract

In this report we investigated several methods to improve the performance of FPGA for general purpose computing. In the early stage of this research we identified the fine grain size of current FPGA as the major performance bottleneck. To increase the grain size, we introduced coarse grain carry architecture that can increase the granularity of arithmetic operations including addition and multiplication. We used throughput density as a cost/performance metric to justify the benefit of the new architecture. We could achieve roughly up to 5 times larger throughput density for selected applications. Along with that we also introduced a dual-rail carry structure to improve the performance of a carry chain, which usually set the cycle time of a FPGA design. A carry select adder built from the dual-rail carry structure reduces the carry chain delay by a factor of two.

Key Words and Phrases: FPGA, grain, carry, carry save, carry select,throughput

Contents

List of Figures

List of Tables

1 Introduction

For the past decades the use of FPGA has been limited to prototyping ASIC. With recent advances in architecture and CAD tools its use has been extended to the areas ranging from high performance computing[2] to mobile communications. The reasons for the growing popularity have three folds. First, given performance requirements FPGAs are a cost effective solution compared with DSPs or microprocessors. Recent study[1] shows that FPGAs outperform Microprocessors and DSPs by one to two orders in terms of throughput density, table 1. The difference mainly arises from the fact that the degree of parallelism that can be exploited is greater in FPGAs than in DSPs. While superscalar, VLIW, and MMX processors exploit parallelism in a one-dimensional fashion, limited by branches, hazards, exceptions, and data compaction/expansion, FPGAs use the parallelism in a two-dimensional fashion by mapping the entire pipelined algorithm on a two-dimensional array. Secondly, the FPGA is an energy-efficient solution in mobile computing and communications. It is because FPGAs can control the factors determining energy consumption more easily than others. For instance, programmability allows a FPGA to minimize energy consumption through activating only the necessary part of a chip. In addition, voltage scaling at the cost of using more area(computing elements) can significantly reduce the energy consumption. Not as much important as the first two reasons, FPGAs show good performance in some areas such as CORDIC[3] and Distributed Arithmetic[11]. It is because embedded memory and programmable interconnect make it possible to choose an optimal number system and algorithm.

As semiconductor technology approaches a deep sub-micron regime, the FPGA faces several challenges. The increase in programming complexity is one of the issues. Technology scaling into deep sub-micron enables the number of computing elements fabricated into a single FPGA chip to grow from a hundred CLBs(Configuration Logic Blocks) to eight thousand CLBs. It allows a single chip to contain not only a few computing elements but the entire algorithm for a particular application. Tremendous amounts of work have been done in developing mapping algorithms for various types of circuits. However, applications requiring high performance still favor manual placement. Recent efforts[9],[10] to generate modules for hierarchical design make it feasible to build a large high-performance system.

Another issue raised by technology scaling is the performance of interconnect. Interconnect in FPGA is built from wires and pass transistors. And the wires are not scaled well as technology scales, and the delay remains relatively unchanged. The performance of a FPGA system, such as cycle time, heavily depends on interconnect performance, and recently some researchers proposed a scheme to dedicate more area to already dominant interconnect that currently takes $80\% \sim 90\%$ of total area[4].

In this research we identified the interconnect delay as a major factor limiting the performance. However, we also noticed that the fine grain size is much crucial factor that causes the large performance gab between custom and FPGA design, table 1. Measurements have shown that directly changing the grain size of current FPGA architecture can improve the performance more effectively without improving the interconnect and reduce the effect of interconnect on cycle time.

As a method to increase the grain size, we implemented coarse grain carry architecture.

Dedicated carry architecture in FPGA is a basis for many arithmetic operations. Coarse grain carry architecture can map more arithmetic operations on a single CLB and increase the throughput density up to 5 times. Along with the grain size change, we designed a dual-rail carry structure to improve the carry chain delay. A carry select adder built from the structure reduces the delay up to 2.4 times, which set the upper bound for cycle time reduction.

	unit	Custom	FPGA	DSP
Multiplier(16 bit)(0.6um)	$\frac{mpy}{\lambda^2 \times sec}$	9.6	0.097	0.057
FIR filter(8 bit)(0.6um)	$\frac{TAPs}{\lambda^2 \times sec}$	3.5	1.9	0.057
IIR filter(10 bit)(0.6,0.9um)	$\frac{TAPs}{\lambda^2 \times sec}$	5.0(0.9um)	0.093	0.01
DES Key Search	$\frac{Keys}{\lambda^2 \times sec}$	0.028(1.5um)	0.00086(0.3um)	0.000023(0.3um)
DNA Sequence Matching	$\frac{CU}{\lambda^2 \times sec}$	1.9(2.0um)	0.070(0.6um)	0.0032(0.75um)

Table 1: Comparing computing machines(metric:throughput density)

The organization of this report is as follows. In section 2 we will review the architecture of a SRAM based FPGA. In section 3 performance bottleneck will be analyzed from a system perspective and its implication will be discussed. In section 4 we discuss methodological issues including metrics, benchmark selection, and a simulation method. In section 5 and 6 we will present the detailed implementation of new carry architectures and the simulation results.

2 Background

2.1 Architecture of SRAM based FPGA

A SRAM based FPGA consists of Configuration logic blocks(CLB) and routing architecture. Figure 1 and 2 show one CLB and its associated routing architecture of Xilinx XC4000E. Each CLB and its associated routing is laid out in roughly $1.2M\lambda^2$ area(XC4000). Each CLB consists of two 4-input lookup tables, one 3-input lookup tables, two flip-flops, and two tri-state buffers. Later series of XC4000 include dedicated carry chains shown in figure 3 to speed up addition. In this research we used the architecture of Xilinx XC4000 as a baseline FPGA and performed our simulation.

2.1.1 configuration logic block

An N-input lookup table(LUT), referred as logic function in figure 1, consists of 2^N SRAM cells. The 4-input lookup table can implement any 4-input Boolean functions. Two of these with one 3-input lookup table can implement any 5-input Boolean functions. Rose[7] showed that the 4-input lookup table gives the smallest area assuming each size of lookup tables implements an identical function. This is because for the LUT with less than 4 inputs inputs and ouputs to the SRAM dominate the area and for the LUT with larger than 4 inputs the

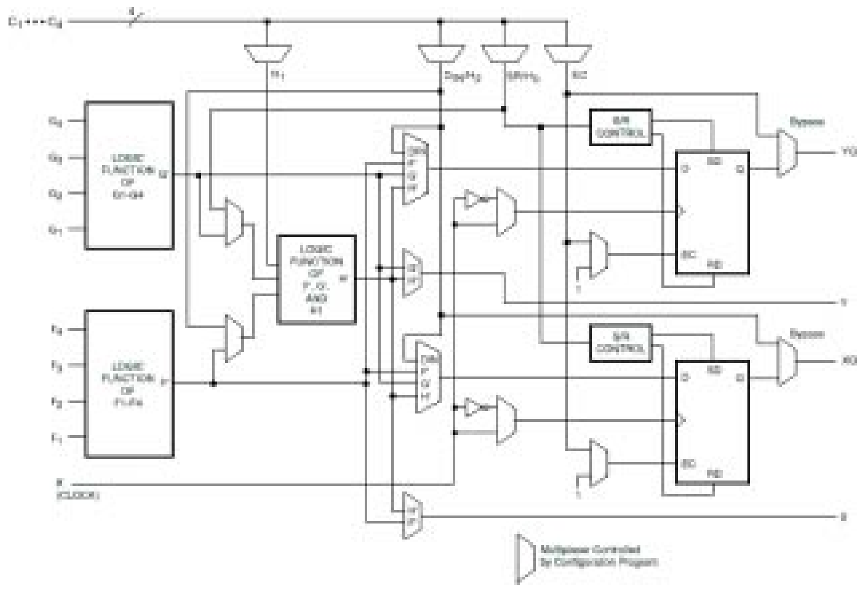


Figure 1: Configuration Logic Block(XC4000E)

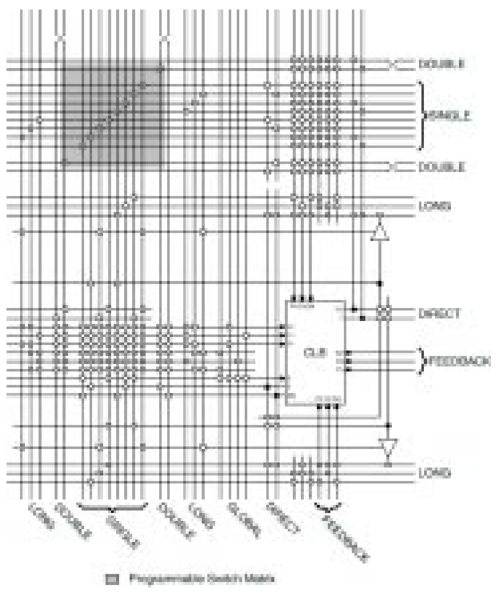


Figure 2: Interconnect(XC4000E)

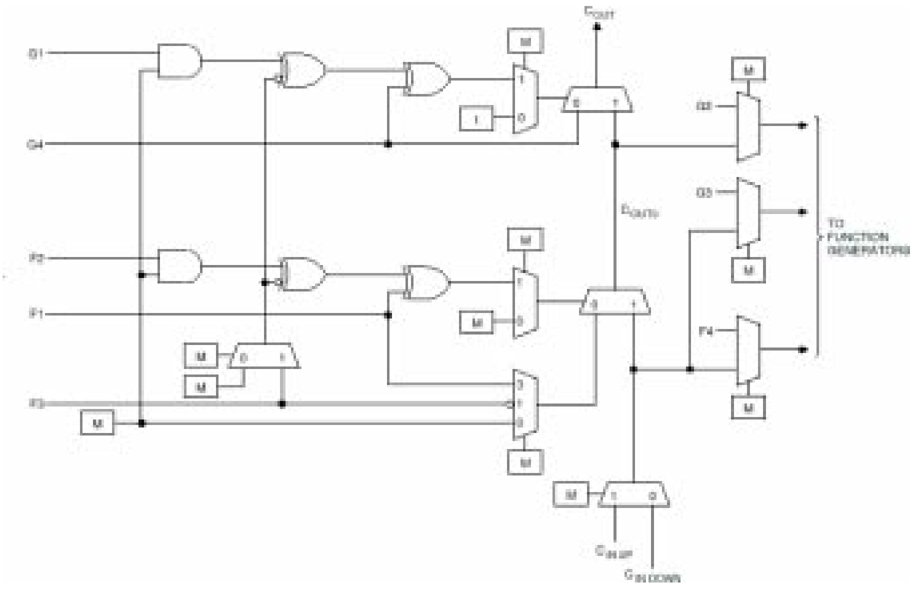


Figure 3: Dedicated carry chain(XC4000E)

SRAM size grows exponentially. From a delay perspective, a 5-input lookup table showed sufficiently good average delay of a critical path over various circuit implementations[6]. The lookup table size of XC4000 architecture is based on these results. The outputs of lookup tables in a CLB drive interconnect either directly or through flip-flops.

2.1.2 routing architecture

Routing architecture in FPGA is used to establish connections between inputs and outputs of CLBs. It consists of segmented wires of different length and programmable switch matrices. The segmented wires are called single, double, and quad as they span one, two, and four CLBs. Each wire is connected to another through a programmable switch matrix(PSM) shown in Figure 2. 'Long' wires span the half-width or the entire width of a chip without going through a PSM, which are desirable for signal with a large fanout. Performance and cost of routing architecture will be discussed in detail later in section 3.

2.1.3 dedicated carry chain

To improve the performance of implementations utilizing a carry operation, XC4000 series facilitate dedicated carry chains. A carry chain is a basis for building adders, subtracters, multipliers, dividers, and counters. The programmable carry chains span the entire width of a chip in columnwise direction. The direct interconnect for carry propagation between CLBs doesn't suffer from the large capacitance of other types of wires and provides small delay comparable to that of a custom adder. Two of four inputs to lookup tables serve as two inputs to a carry function. The carry output is connected to the input of a lookup table through a multiplexor.

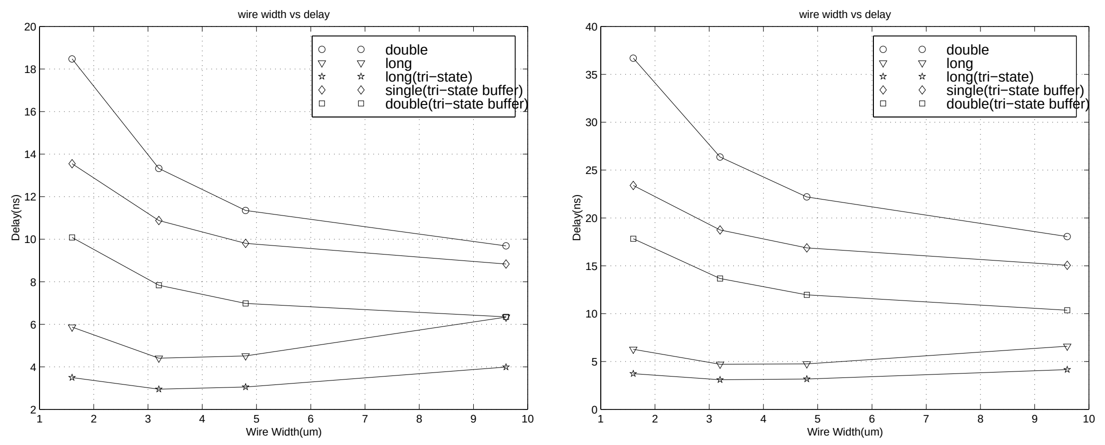


Figure 4: Interconnect delay vs. Interconnect width (a)8-bit(FO=8)(b)16-bit(FO=16)

3 System perspective

3.1 Locating the performance bottleneck and its implication

3.1.1 interconnect and grain size

In FPGA, interconnects are segmented to provide minimum delay over various lengths of interconnects and to ensure routability. Xilinx XC4000 chips are equipped with single, double, quad, and long lines. These interconnects suffer from substantially large capacitive loads. Two major sources are the diffusion capacitance of pass transistors connecting segmented lines to the input and output of CLBs and programmable switch matrices. For instance, a single line is connected to 10 $\sim$ 14 pass transistors and two programmable switching matrices per segment. The size of transistors exceeds that of transistors used in a logic portion by an order in magnitude[13].

Simulation, figure 4, explains why large transistors are used for the interconnects. (a) and (b) are the interconnect delays spanning 8 and 16 CLBs-fanout of 8 and 16-with respect to the width of transistors used in interconnect. This is a common configuration found in many applications where 8-bit or 16-bit data are selected or manipulated by a single source of signal. To have reasonable performance, quite large size transistors are necessary. As a result, interconnect including wires, programmable switch, I/O to the CLB takes 80 $\sim$ 90% of the area[13]. The large capacitance also accounts for the power consumption related to the interconnects, which is 90 % of the total power consumption[13].

From a delay perspective the effect of interconnect on the cycle time is substantial. Figure 5 shows a cycle time, for various applications, broken into a CLB and interconnect portion. Interconnect delay accounts for 50 $\sim$ 60 % of the cycle time. The large interconnect delay comes from FPGA's inherent programmable structure. As technology scales and more inputs are added to a CLB, interconnect delay gets worse with respect to the delay through CLBs.

In general there are two ways to reduce the effect of interconnect on throughput. One

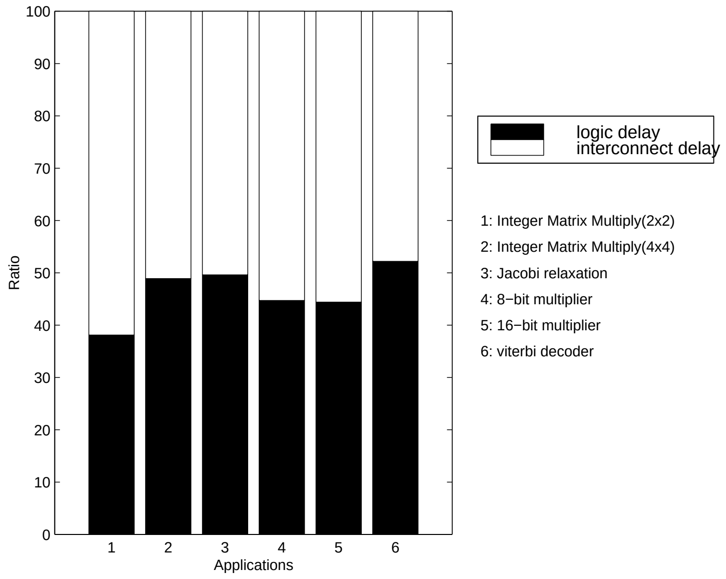


Figure 5: Interconnect delay vs. Logic delay

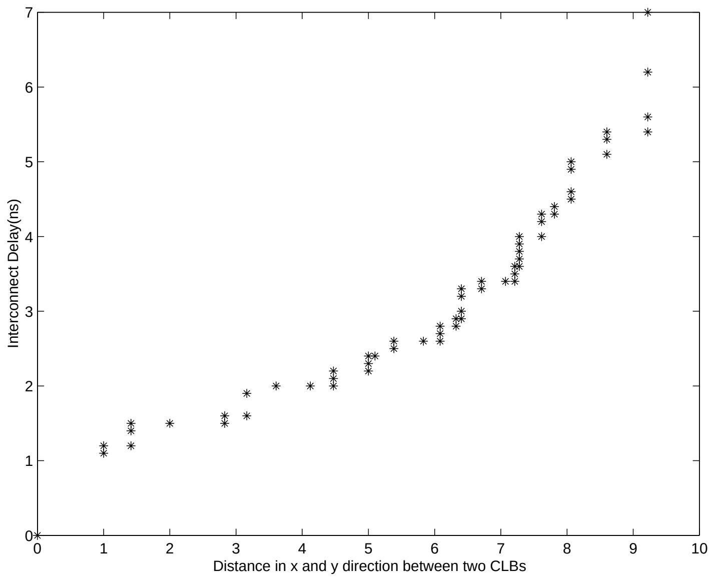


Figure 6: Interconnect delay between two CLBs

is a device-level or circuit-level solution that physically improves the delay of interconnect. The other is an architectural solution such as increasing the grain size. As far as throughput is concerned, the latter is more promising because throughput is increased not only from reduction in interconnect delay, which will be explained, but also from performing more computation during cycle time.

When we define the grain size as the functional capacity of a CLB, the impact of increasing the grain on cycle time can be estimated from the measurement shown in figure 6. It basically shows the interconnect delay between two points that are separated horizontally by x and vertically by y CLBs. The horizontal axis represents the radial distance between two points expressed as $\sqrt{x^2 + y^2}$. The key observation is that the delay between two points increases roughly linearly with respect to the radial distance of two points. It implies that twice the functional capacity of a CLB could reduce the interconnect delay potentially to its half since logic in-between can be presumably placed on the half number of CLBs.

In FPGA, the CLB grain size can be changed in two ways. The first method is to increase the SRAM lookup table size. The drawback of this scheme is that the lookup table size grows exponentially as the number of input increases. Collapsing two 4-input lookup tables would need 16 times the area of one 4-input lookup table. Another problem is that the increased lookup table is likely to be wasted for functions with a few inputs. The second method to increase grain size is to increase the grain size of carry architecture because it is the key element in arithmetic operations including addition, subtraction, multiplication, and division. Carry architecture in FPGA is implemented in custom logic with direct interconnect. It takes a small portion of logic area in the CLB, and thus the effect on area is small. The grain size and performance of current carry architecture will be discussed in detail in the next section.

3.1.2 carry architecture

Carry propagation and carry saving are two methods to deal with carries in addition. In carry propagation addition(CPA), the base cell(a full adder) generates a sum and a carry, and the carry generated in the current bit position is used as a carry input for the next higher bit position. While the carry propagation suffers from considerable propagation delay, carry save addition(CSA) resolves this problem by keeping the result of addition in a redundant form: both sum and carry. Carry save additions are often used for two reasons. First, it can reduce cycle time when we need iterative addition such as in division by avoiding carry propagation. Second, it can reduce the critical path delay in adder trees such as in a multiplier.

	CPA	CSA(per stage)
custom	first carry generation+0.2N+final sum(ns)	0.5 (ns)
FPGA	first carry generation+0.4N+final sum(ns)	$4 \sim 5$ (ns)

Table 2: CPA and CSA performance

Both carry propagation and carry saving adder performance are crucial in FPGA because they determine the performance for a large set of applications. Table 2 shows the performance of two adders implemented in custom logic and FPGA(CMOS $0.35\mu m$). N in the CPA column represents the input bit width.

While dedicated carry chain makes the CPA in FPGA comparable to the custom design, The CSA in FPGA performs about ten times worse than the custom CSA. Throughput difference is not necessarily ten times because the FPGA design is more deeply pipelined than the custom design. However, this difference cannot be hidden in the unpipelinable system.

3.2 Ideas

From the comparison in previous section, we can draw two points. First, increasing the grain size of carry architecture to support a CSA would increase the throughput. The optimal grain size and the necessary functionality can be determined from mapping various applications on the new architecture. Several functional complexity levels varying from addition to addition/subtraction to conditional addition/subtraction can be implemented and compared at the cost/performance basis. Secondly, high performance carry chain such as carry select, carry lookahead or Brenk-Kung would further increase the performance by decreasing the cycle time. Following section will discuss the methodological issues to carry out the comparison.

4 Methodology

4.1 Metrics

The metric to compare different FPGA architectures varies according to the set of benchmarks chosen for the comparison. Two metrics that are particularly important, and will be used, are throughput density and energy-delay product. Throughput density is proposed by DeHon[1] to compare FPGA with custom circuits or DSP for computation-intensive and latency-insensitive applications. Energy-delay product is a metric to compare the energy demand for applications used in mobile communication and computing. To formulate the equations for these metrics, we need to begin with how to compute or measure area and delay.

4.1.1 area

The area of FPGA design can be determined by multiplying the number of CLBs used in the design by the CLB area(including associated routing) in FPGA. We define two ratios to determine the area reduction rate between two architectures. Area Reduction(AR) is the ratio between the numbers of CLBs used in the design for two architectures. Normalized Area Reduction(NAR) is the AR normalized to the CLB area difference. The distinction is made to emphasize the difference between the ratio with consideration of area penalty in new architecture and the ratio without it.

$$\begin{aligned} Area(\lambda^2) &= Area\ for\ a\ single\ CLB(\lambda^2) \times number\ of\ CLBs\ in\ the\ design \quad (1) \\ AR &= \frac{Number\ of\ CLBs\ for\ Arch.\ 1}{Number\ of\ CLBs\ for\ Arch.\ 2} \quad (2) \\ NAR &= \frac{Number\ of\ CLBs\ for\ Arch.\ 1 \times CLB\ area\ for\ Arch.\ 1}{Number\ of\ CLBs\ for\ Arch.\ 2 \times CLB\ area\ for\ Arch.\ 2} \quad (3) \end{aligned}$$

4.1.2 delay(cycle time)

The delay in FPGA is usually referred to a maximum delay between two stages of flip-flops, which also set the cycle time of a pipelined system. The delay can only be measured after performing logic placement and interconnect routing because of heuristic nature of place and route algorithms. The ratio between two cycle times is defined as cycle time reduction(CTR).

$$CTR = \frac{Cycle\ time\ for\ Arch.\ 1}{Cycle\ time\ for\ Arch.\ 2} \quad (4)$$

4.1.3 throughput and throughput density

The throughput of a FPGA design is a reciprocal of a cycle time. Throughput density is the throughput normalized to the used area. It has a unit of $\frac{1}{ns \times \lambda^2}$. This metric is easy to calculate and provides a fair cost/performance comparison for applications with large throughput requirement and sufficient parallelism.

The ratio of two throughput densities is defined as multiplication of three factors.

$$\begin{aligned} Throughput\ density\ gain(TDG) &= Normalizedarea\ reduction\ factor(NAR) \\ &\times cycle\ reduction\ factor(CR) \\ &\times cycle\ time\ reduction\ factor(CTR) \quad (5) \end{aligned}$$

where

$$CR = \begin{cases} \frac{cycles\ taken\ for\ current\ arch}{cycles\ taken\ for\ new\ arch} & ,if\ not\ fully\ pipelined \\ 1 & ,if\ fully\ pipelined \end{cases}$$

4.1.4 power-delay product

The power-delay product is often referred as energy per operation. In general,

$$\begin{aligned} Power &= Capacitance \times SupplyVoltage^2 \times Frequency \quad (6) \\ Power \times Delay &= \frac{Energy}{operation} \quad (7) \\ &= Capacitance \times SupplyVoltage^2 \quad (8) \end{aligned}$$

This metric gives the energy that is required to execute a single operation. However, it fails to deliver information on the performance such as the throughput. Energy-Delay product resolves this problem.

4.1.5 energy-delay product

Normalizing energy to throughput provides a metric that measures the energy efficiency of mobile computing and communication systems under identical throughput requirements.

In CMOS design, the first order equation can be written as

$$\begin{aligned} Energy \times Delay &= \frac{Energy}{Throughput} && (9)\\ &= Power \times Delay^2 && (10)\\ &= Capacitance(C) \times Supply\ Voltage(V)^2 \\ &\quad \times Frequency(F) \times Delay(D)^2 && (11)\\ &= CV^2D && (12)\end{aligned}$$

$$Delay = \frac{CV}{k(V-V_{th})^a} where\ a\ is\ 1\ to\ 2\ and\ k\ is\ a\ constant \qquad (13)$$

In FPGA, the first order approximation of equation 11 becomes

$$Energy \times Delay = scalingfactor \times (NumberofCLBs) \times V^2 \times Delay \qquad (14)$$

Scaling factor reflects the averaged switching activity times the averaged capacitance of a CLB.

An Energy-Delay product ratio between two FPGA designs is

$$\frac{ED_1}{ED_2} = \frac{Number\ of\ CLB_1 \times V_1^2 \times \frac{V_1}{(V_1-V_{th})^a}}{Number\ of\ CLB_2 \times V_2^2 \times \frac{V_2}{(V_2-V_{th})^a}} \qquad (15)$$

assuming scaling factors, equation 14, are cancelled out, and C and k, equation 13, are cancelled. From the fact that more than 90% of total capacitance are associated with the routing architecture, above assumption is valid if the routing architecture for the new FPGA architecture doesn't much differ from the original.

The importance of this metric in this research is its relation to the throughput density gain. For throughput-intensive and highly-paralellizable applications throughput is often proportional to the used area. The design flexibility in FPGA allows application-specific optimization for a particular metric at the cost of other metrics. Trade-offs between energy consumption and throughput are the example.

In a FPGA system throughput density gain from architectural improvement can be used to reduce the energy consumption through voltage scaling while aggregate throughput

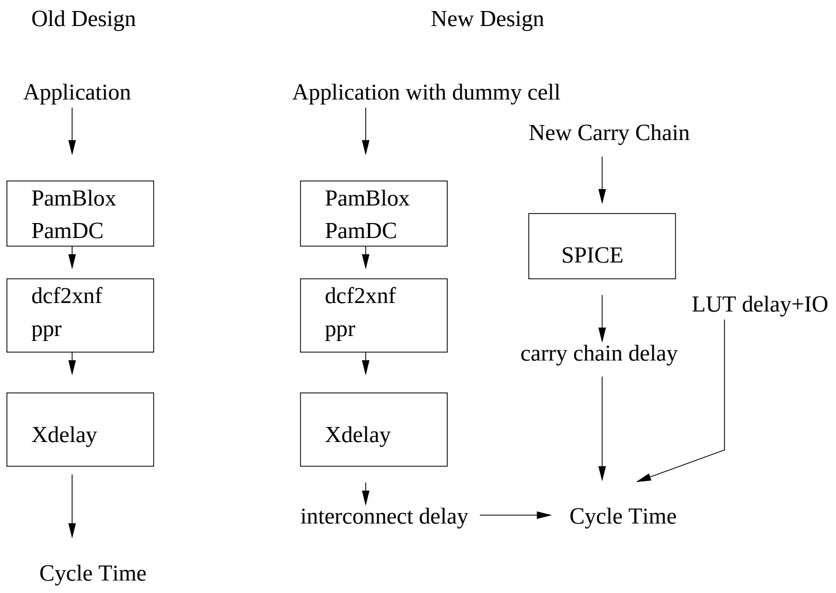


Figure 7: Simulation flow

meets the requirement. The equation 15 turns out to be handy to estimate how much energy saving we can achieve from throughput density gain. For $0.35\mu m$ process with $a = 1.25, V_{th} = 0.5V, V_1 = 3.3V.$, four times increase in throughput density can lead to energy saving by an order in magnitude. We will keep this fact in mind throughout this report to estimate the energy saving for the new carry architecture.

4.2 Benchmark selection

In this research, we chose benchmarks from three different categories: arithmetic units, a DSP library from Xilinx, and a subset of RAW benchmark suite from MIT. Arithmetic units include adders and multipliers. The DSP library is composed of FIR, IIR filters, correlators, FFT, and Viterbi decoder. The original RAW benchmark consists of Binary Heap, Bubble Sort, DES Encryption, FFT, Semiring Graph Problems, Integer Matrix Multiply, Jacobi, Life, Merge Sort, N Queens. The subset of RAW benchmarks includes Binary Heap, Integer Matrix Multiply, and Jacobi. Life was not considered because it doesn't use any adder structure and merge sort was also dropped because its structure is similar to Binary heap. DCT(discrete cosine transform over integer numbers) was considered instead of FFT over complex numbers. N Queens, Semiring Graph Problems, and DES still remain to be studied.

4.3 Simulation method

Of three factors constituting throughput density gain(TDG), area reduction factor and cycle reduction factor can be computed using a formula derived from careful design and verification. Cycle time reduction factor, however, only can be measured through the simulation whose procedure is shown in figure 7 .

The target FPGA in this research is Xilinx XC4000XL series($0.35\mu m$ process) with speed grade of -1. Cycle time for the current architecture can be measured using Xdelay from Xilinx after the benchmarks are placed and routed on the target FPGA. Measuring the cycle time for the new architecture requires additional steps. Cycle time in FPGA design is broken into two portions: interconnect delay and logic delay. Logic delay is again broken into carry chain delay, and lookup table and I/O delay. Measuring interconnect delay requires the place and route of actual design due to the non-deterministic nature of a heuristic algorithm. For the measuement we could use our current place and route tools because we didn't alter the interconnect architecture.

To measure the three delay components, three different methods, figure 7, were used. First, to measure new carry chain delay, SPICE model was developed for a $0.35\mu m$ process and the new carry circuit is simulated to generate delay for different bit width. Secondly, the lookup table and I/O delay were taken from the switching characteristic table of XC4000XL($0.35\mu m$) series [14]. Finally, interconnect delay was measured using a dummy CLB. The dummy CLBs were placed where the new CLBs are needed such that in the new architecture, we can replace the dummy CLB with new CLB without changing input and output connectivity. Thus, measuring the interconnect delay with dummy CLBs is essentially same as with new CLBs. The cycle time of new architecture was determined by combining three delays. Xdelay was used to measure the cycle time of the design for new architecture by measuring the worst case delay in pipeline stages.

The functionality of the new CLB for new architecture was verified by mapping a new CLB on several CLBs on the FPGA of current architecture.

5 Implementation

5.1 Current architecture-ripple carry adder

The dedicated carry logic in XC4000, we used as a baseline architecture, adopts a ripple carry architecture and can compute the carries in O(N) time where N is the input width. Its diagram is shwon in figure 3. Assuming all the inputs to the adder are available at the same time, the delay of a carry chain is summation of the delay for the first carry bit generation and the delay through a series of multiplexors up to the last bit. The first carry generation gives constant delay, and a series of multiplexors gives delay proportional to the input size. For this reason, it is critical to optimize the multiplexor chain. As a base architecture, two SPICE models were developed, shown in figure 8. One uses pass transistors, figure 8(a) , to minimize the area and the other uses tri-state buffers, figure 8(b) , to put reasonable buffering. In diagram (a) and (b) carry-in is connected to carry-out through either two pass transistors, (a), or two tri-state buffers,(b). We took into account all the capacitive loads in the critical path Different transistor size was simulated to find a optimal size. The performance of the former was significantly degraded by series of pass transistors, and the latter with a transistor size of 8λ gave about 0.5 ns through the carry chain that is close to the performance of current Xilinx carry chain($0.35\mu m$). Hence, we used the latter as a base architecture for this research.

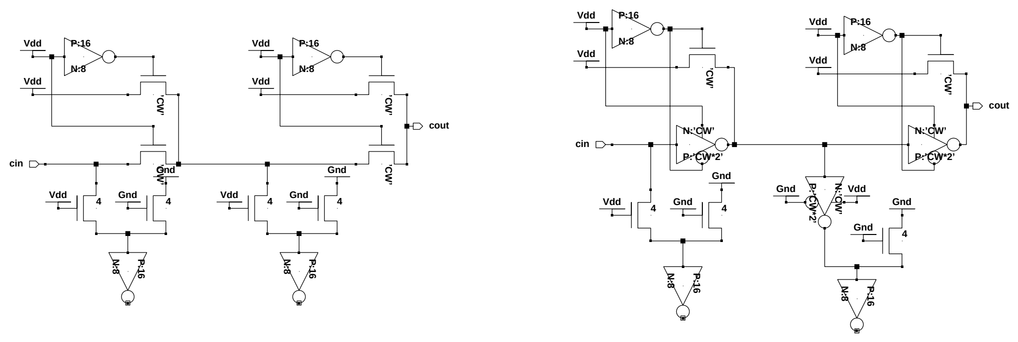


Figure 8: Spice model for carry chain (a) pass transistor (b) tri-state buffer

5.2 High performance carry adder

Carry lookahead, carry select, and Brent-Kung adders are widely used adders to break the linear dependency in the ripple carry adder's delay. Hauck et. al applied these techniques to Mux based carry chain(Altera) and reported that the Brent-Kung adder reduces the delay upto 3.8 times with 8% increase in area for a 32-bit addition[12]. Their implementation, however, has a limitation that the adder has a fixed starting and ending point, which doesn't reflect the real case in which the starting and ending point of a FPGA carry chain are decided dynamically for a particular implementation. The flexibility introduces more gates and capacitance in the critical path. In that sense their measurement gave an upper bound.

In this research we implemented a dual-rail carry structure that can be used as a carry select adder. Figure 9 shows a dual-rail carry chain built into a single CLB. In implementing a carry select adder, each cell is programmed as either the last cell or the non-last cell. In the last cell mode the carry select output is enabled, and the carry-out of the current CLB is chosen based on the previous carry select signal. In the non-last cell mode, the carry select output is disabled and two carry paths are connected to the next CLB. The carry select adder implemented using this structure is different from the usual ASIC design in a sense that the dual-rail carry structure only provides one level of carry select. Hence, a N-bit carry select adder is made of a N-bit adder broken into M $\frac{N}{M}$-bit ripple adders, and the critical path is the summation of the delay through the first $\frac{N}{M}$-bit ripple adder and M-1 carry select logics. We tried to achieve high performance through two optimizations.

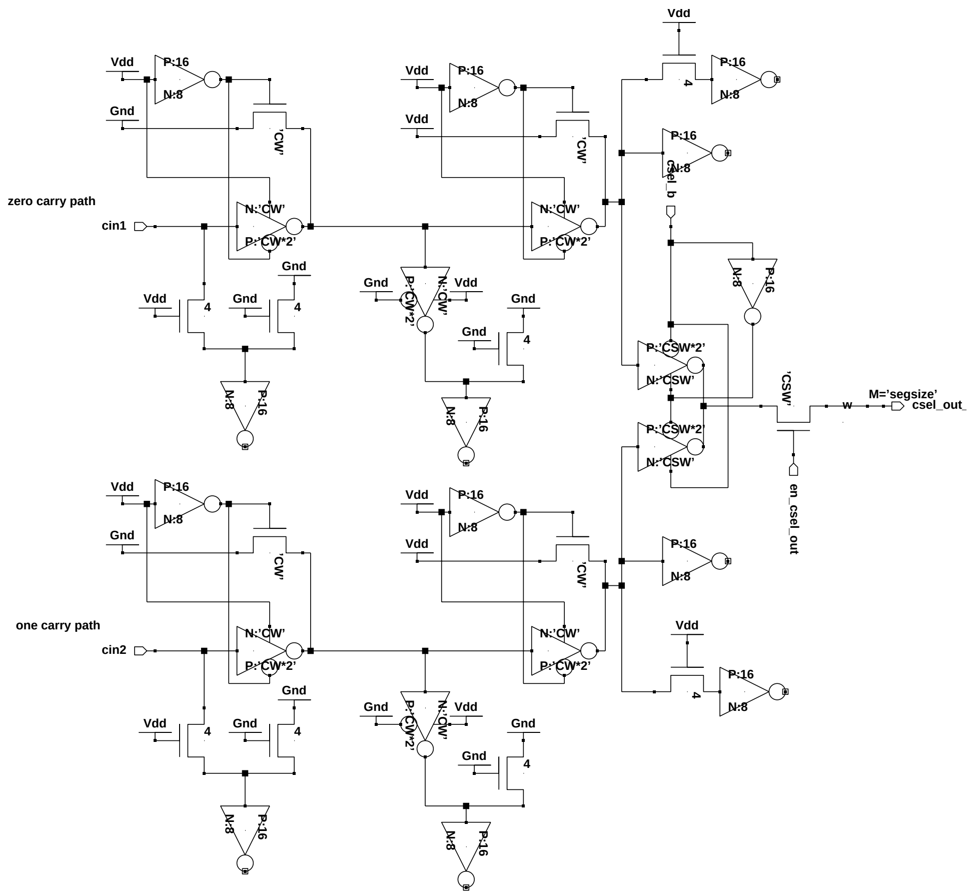


Figure 9: Dual-rail carry chain for a carry select adder

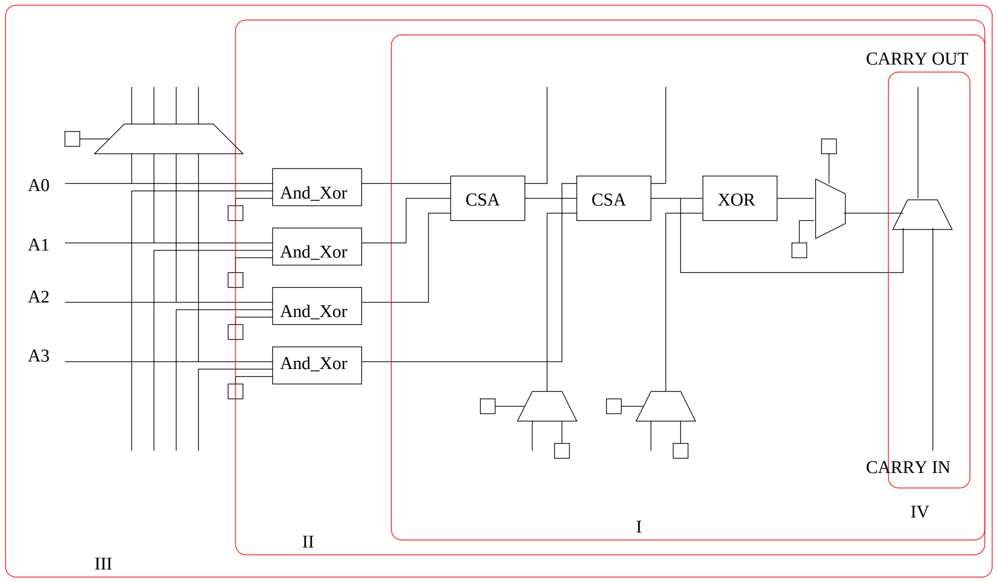


Figure 10: Coarse grain carry architecture

The first is reducing the number of gates in the critical path by alternating polarity of carry selection signal. The second optimization was achieved through the study of trade-offs between the bypass segment size-size of the smaller ripple adders-and the transistor size in carry select path. The detailed result will be presented in section 6.4.1. Multi-level carry select adders were also implemented but didn't perform better than the single-level.

5.3 Coarse grain carry adder

A ripple carry adder only uses two out of four inputs to the LUT. That is, we can increase the grain size from two-bit addition to four-bit addition without modifying the i/o interface of current architecture. The four-bit addition requires a (4,2) counter. The proposed coarse grain carry architecure implements a (4,2) counter and connects the two outputs(external) to the original two inputs of carry architecture.

We built three different versions to determine how many new local(direct) wires are needed and which inputs should be provided as a dynamic inputs by mapping various applications on the new architecture. Three version will be introduced according to the incremental complexity in each design.

5.3.1 adder only

The simplest design was a carry structure only supporting addition. To compute $A_0 + A_1 + A_2 + A_3$, what we really need is a (4,2) counter that can add 7 bits of same weight and generate three carries and one sum bits. Each (4,2) counter takes 3 carry bits from previous bit and 4 inputs: A_0, A_1, A_2, and A_3. As shown in figure 10 I, The first (3,2) counter takes A0,A1,and A2 and the second (3,2) counter takes the sum of the first, the carry from the first (3,2) counter in the cell below, and A3. The delay of added (3,2) counters only appears once in the first bit carry generation and doesn't affect the delay of the carry chain. In this design, we introduce two new direct interconnects for internal carries. To initialize the first carry-in for a (3,2) counter and a xor, two extra multiplexors are used.

5.3.2 adder/subtracter

To compute $\pm A_0 \pm A_1 \pm A_2 \pm A_3$, we need to provide two things: inverting inputs, A_0, A_1, A_2, A_3 and providing correct carry-ins. An extra xor gate for each input can invert the inputs. There are only three paths for carries. This limits the number of subtractions to three per slice. That is, up to three subtractions, $A_0 - A_1 - A_2 - A_3$, are possible by inverting three inputs and setting three carry-ins to ones. Newly added portion is enclosed by a box marked as II.

The necessity of switching between adder and subtracter dynamically can be justified from benchmark analysis. The initial analysis was that none of the benchmarks requires dynamic switching. Thus, we only provided static programmability to choose addition or subtraction.

5.3.3 conditional adder/subtracter

To mask out four inputs, A_0, A_1, A_2, and A_3, selectively, we need an AND gate for each input and four selection signals. The four selection signals run vertically to provide common selection signals to CLBs in the same column.

Provided that four signal are encoded, only two signals are sufficient, which reduces the area for the wires between CLBs. However, this scheme suffers from its overhead because we can use minimum size wires for selection signals unlike other general purpose interconnect, and encoding requires a 4-to-2 encoder and a 2-to-4 decoder per 16-bit LUT(half CLB) that take relatively large area.

6 Result

6.1 AR and CR

Operation complexity	Expressions	comments
4-input Add	$A_0 + A_1 + A_2 + A_3 (A_0 + A_1 \pm A_3)$	
4-input Add/Sub	$\pm A_0 \pm A_1 \pm A_2 \pm A_3$	
Conditional 4-input Add/Sub	$\pm x_0 \times A_0 \pm x_1 \times A_1 \pm x_2 \times A_2 \pm x_3 \times A_3$	x_i is a Boolean

Table 3: Three different functional complexities for coarse grain carry architecture

Area and cycle reduction factor were computed using formulae derived from mapping and verification. The benchmarks are categorized into three groups according to the functional complexity required. The different complexity levels are shown in table 3. Each

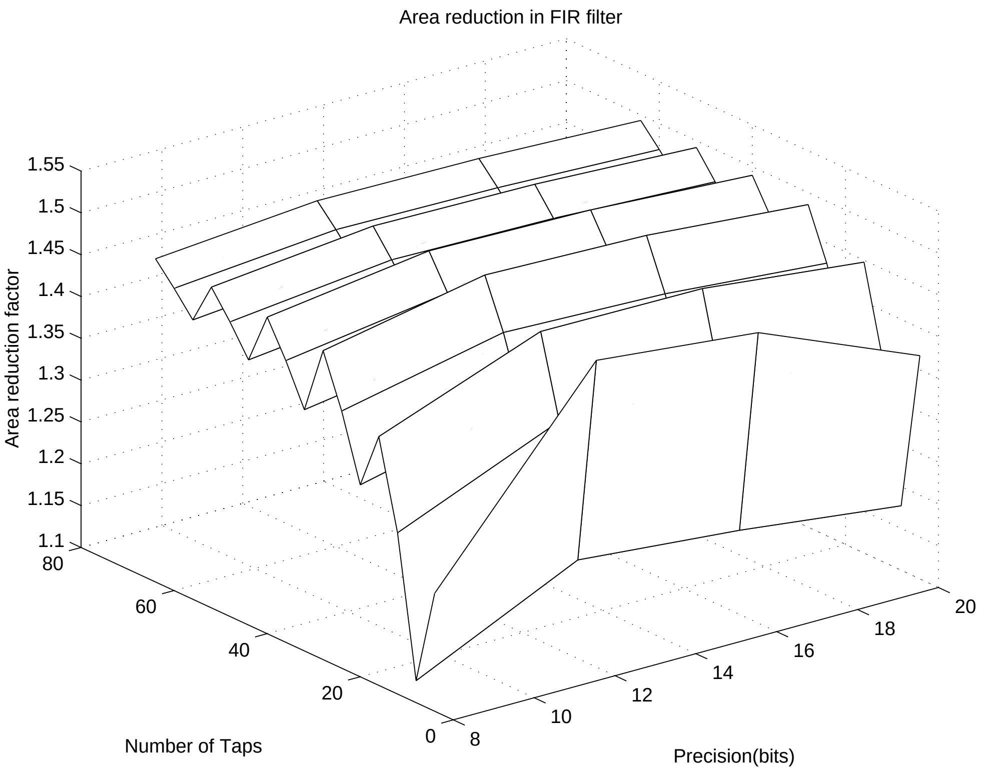


Figure 11: Area Reduction for M-tap N-bit FIR filter

group includes the benchmarks that achieve area and cycle reduction from each incremental complexity.

6.1.1 group I: 4-input addition

Benchmarks using tree adder structures take advantage of this improvement. FIR filter from DSP library, integer matrix multiplication and Jacobi relaxation from RAW benchmark suite, a multiplier fall into this category.

Operation	Number of CLBs
2-input adder	$\sum_{i=1}^{\lceil \log_2 m \rceil} \lceil \frac{m}{2^i} \rceil \times \lceil \frac{n+i}{2} \rceil$
4-input adder	$\sum_{i=1}^{\lceil \log_4 m \rceil} \lceil \frac{m}{4^i} \rceil \times \lceil \frac{n+2i}{2} \rceil$

Table 4: Number of CLBs required for adding m n-bit numbers

Table 4 shows the number of CLBs required for 2-input adders and 4-input adders in case where m n-bit numbers are added using tree topology. The area reduction factor(AR) for the tree ranges from 2.25 to 3, which is in many cases the upper bound for the AR of the entire application. For instance, AR for the FIR filter for different TAP size and precision, shown in figure 11, ranges from 1.13 to 1.51. This is because FIR filter uses distributed arithmetic in which the area for precomputed coefficient table dominates the total area. More information on distributed arithmetic can be found in [11].

Operation	Number of CLBs
2-input adder	$\frac{nm}{2} + \frac{mn(n+2)}{8} + n\lceil \frac{m}{4} - 1 \rceil \lceil \frac{\log_2 m+n}{2} \rceil + \sum_{i=1}^{\lceil \log_2 n \rceil} \lceil \frac{n}{2^i} \rceil \lceil \frac{n+log_2 m+2^i}{2} \rceil$
4-input adder	$\frac{nm}{2} + \frac{mn(n+2)}{8} + n\lceil \frac{m-4}{12} - 1 \rceil \lceil \frac{\log_2 m+n}{2} \rceil + \sum_{i=1}^{\lceil \log_4 n \rceil} \lceil \frac{n}{4^i} \rceil \lceil \frac{n+log_2 m+4^i}{2} \rceil$

Table 5: Number of CLBs required for a M-tap N-bit FIR

Table 5 shows the number of CLBs required to implement a M-tap N-bit precision FIR filter with 2-input and 4-input adders. Since the FIR filter is fully pipelined, CR is 1.

As for multipliers and integer matrix multiplication, the area reduction only from 4-input addition is also limited due to the dominant area used for booth multiplexors. However, the area reduction for these applications get significantly increases using conditional addition and subtraction. This will be discussed later.

Operation	Number of CLBs
2-input adder	$4n + (n^2 + 2n - 1) + 2n^2$
4-input adder	$4n + 2n^2$

Table 6: Number of CLBs used for Jacobi Relaxation

Jacobi relaxation from RAW benchmarks suite also gets improved using 4-input adders. Jacobi relaxation is an iterative algorithm to find a solution to differential equations of the form $\nabla^2 A + B = 0$ given a set of boundary conditions. Each step of this algorithm replace the node's content with the average of the values of its four neighbor nodes. Figure 13 visualizes an example with small input size[9]. In each iteration of the algorithm, the content of a node is added with that of a node in upper right side. This requires 7 adders to add two numbers grouped together. Then, additional 4 adders add two of these 7 results to update four Y nodes. In general, for $n \times n$ nodes, we need $2n^2 + 2n - 1$ adders using the original 2-input adder. With 4-input adders, we only need n^2 adders. Table 6 compares the CLB numbers used to implement the entire Jacobi relaxation for 2-input and 4-input adders. For large n, AR approaches $\frac{3}{2}$. CR is 1 because initialization can be ignored.

Although we didn't include in our benchmarks, a median filter, found in multimedia applications, is basically same as Jacobi relaxation and potentially gets same benefit from this new architecture.

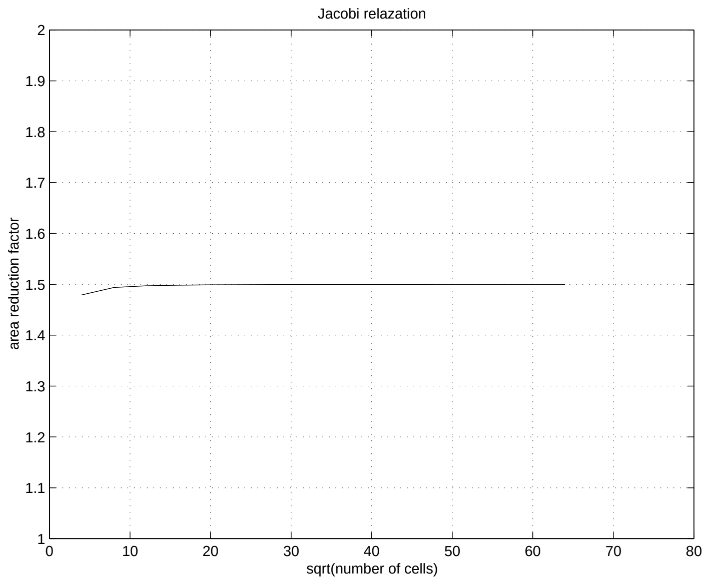


Figure 12: Area Reduction for Jacobi Relaxation

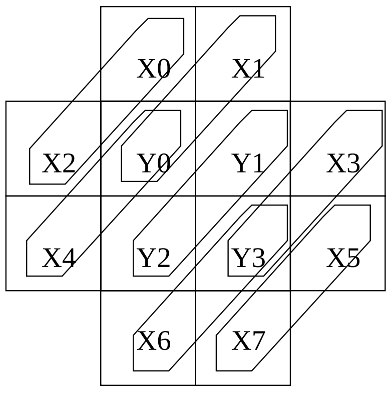


Figure 13: Jacobi Relaxation(2x2)

6.1.2 group II: 4-input addition and subtraction

The benchmarks that utilize this additional functionality are Discrete Cosine Transform(DCT) and Viterbi decoder.

The DCT is a common application used in image processing. Figure 14 (a) shows a fast DCT algorithm proposed by Arai, Agui, Nakajima[17]. It contains only 5 multiplications and 8 scalings. Often scaling is absorbed in the stages following DCT, and thus it is ignored for analysis.

Figure 14 (b) and (c) represent the 8-bit precision 8-point DCT implementations for 2-input adder and 4-input adder/subtracter separately. An 8-bit constant coefficient multiplier takes two cycles, which is shown as two squares in series. Two stages of shuffle network implemented with 2-input adders can be compressed to one using 4-input adder/subtracters. In addition, final adders for a1 and a5 multiplications can be shared using 4-input adders. As a result, the number of CLBs for (b) and (c) are 382 and 270. AR = 1.4. In this case, CR = 1 since the entire circuit is fully pipelined.

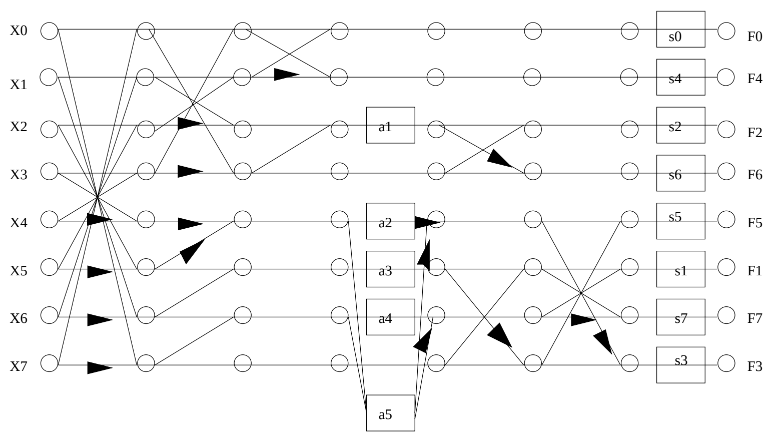
The Viterbi algorithm is a method commonly used for decoding bit streams encoded by convolution coders. The details of a particular decoder algorithm depends on the encoder. We implemented a decoder that can decode a 4-state optimal rate 1/2 convolution code. Viterbi decoding can be broken into two major operations, metric update and traceback. In metric update, two things are done: the accumulated state metric is calculated for each state and the optimal incoming path associated with each state is determined. Traceback uses this information to derive an optimal path through the trellis.

The major area reduction is achieved in metric update process. The state metric update requires add-compare-select(ACS) units that takes 60 % of the total area. A single ACS unit consists of two adders, one comparator and one multiplexor. Two adders add two state metrics(Λ) and path metrics(Γ), and the results are compared by the comparator. The sign of comparator's result set the multiplexor. With 4-input additions/subtractions, two adders and a comparator performing $(\Lambda_{00} + \Gamma_{00}) - (\Lambda_{01} + \Gamma_{01})$ can be mapped on a single adder/subtracter. The area reduction for the Viterbi decoder is 1.7. CR is 1 because the algorithm is fully pipelined.

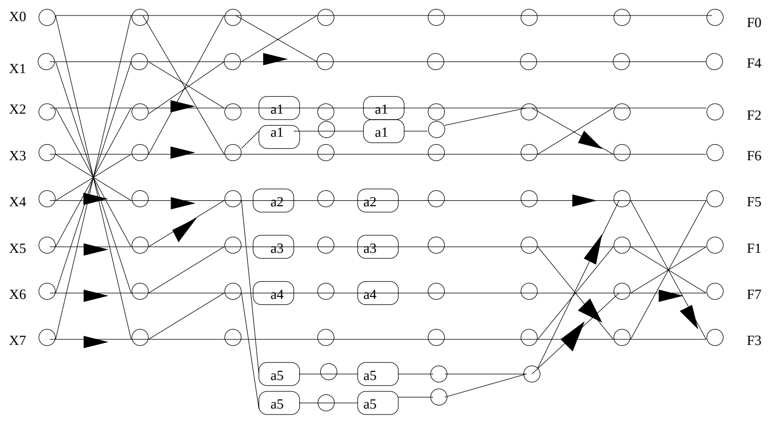
6.1.3 group III: conditional 4-input additions/subtractions

The conditional addition/subtraction integrates control into arithmetic operation. Multipliers, integer matrix multiplication, and binary heap are the applications that can be significantly improved from the added functionality.

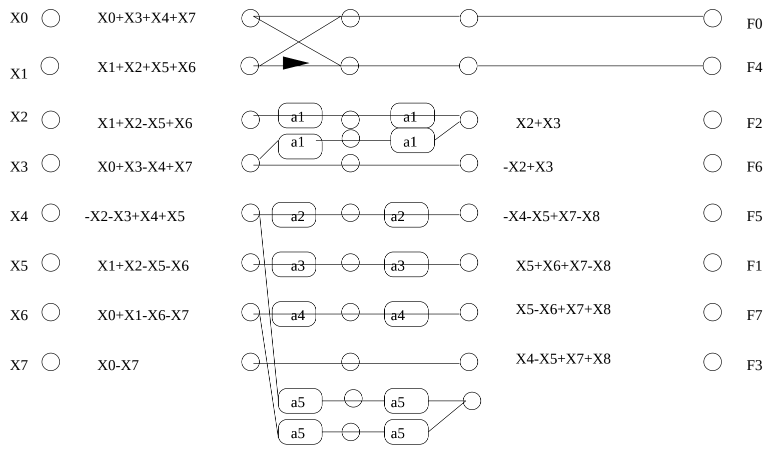
FPGA multipliers can be grouped into two according to the characteristics of operands. If one operand is a constant, it is called a constant coefficient multiplier. This is first proposed by Xilinx[11] and takes smaller area than a two-variable multiplier by using SRAM as a lookup table of precomputed multiples of a multiplicand. If both operands are variables, we need a full multiplier composed of partial product generation and addition.



(a) original algorithm



(b) mapped on 2-input adder



(c) mapped on 4-input adder/subtracter

Figure 14: Fast DCT algorithm by Arai, Agui, Nakajima

Two-variable multiplication requires the process called partial product generation. In microprocessor, Booth encoding is generally used to reduce the number of partial products. The drawback of Booth encoding is to generate the negative multiples of a multiplicand(M). The negation can be simply obtained in ASIC by inverting bits. In FPGA, however, this inversion is generally expensive. Chapman[18] shows that non-Booth requires the same area and latency to generate the partial product by making use of XC4000 carry architecture. The non-Booth encoder generates $0, M, 2M, 3M$ instead of $0, \pm M, \pm 2M$ using its carry chain.

The 4-input conditional adder/subtracter can implement the non-Booth encoder that can produce the multiples of a multiplicand(M) from 0 to 15M. The equation 16 can be rewritten by replacing A_i with $A_0 \times 2^i$,

$$\begin{aligned} & \pm x_0 \times A_0 \pm x_1 \times A_1 \pm x_2 \times A_2 \pm x_3 \times A_3 & (16) \\ &= x_0 \times A_0 \times 2^i + x_1 \times A_0 \times 2^{i+1} + x_2 \times A_0 \times 2^{i+2} + x_3 \times A_0 \times 2^{i+3} & \\ &= A_0 \times 2^i \times (x_0 + 2 \times x_1 + 4 \times x_2 + 8 \times x_3) & (17) \end{aligned}$$

$A_0 \times 2^i, A_0 \times 2^{i+1}, A_0 \times 2^{i+2}$, and $A_0 \times 2^{i+3}$ can be achieved by simply connecting properly shifted A_0 to the A_1, A_2, A_3. The combinations of x_0, x_1, x_2, x_3 from 0000 to 1111 generate multiples ofA_0 from 0 to $15 \times A_0$.

For an N-bit multiplicand, a column of $\frac{N}{2} + 2$ CLBs can generate 0, M, 2M, 3M, 4M, 5M, ..., 14M, 15M. Each column takes 4 multiplier bits. This is about four times area reduction compared to the original partial product generation in a 2- input adder since a column of N+1 CLBs are required to generate 0,M,2M,3M, and each column only consumes 2 multiplier bits. In addition to that, the number of partial products has been halved. That is, the number of adders required to sum up the partial products is halved as well.

Operation	Number of CLBs
2-input adder	$\overbrace{n + \lceil \frac{n}{2} \rceil \times (n+2)}^{partial\ product\ generation} + \overbrace{\sum_{i=1}^{\lceil \log_2 n \rceil - 2} \{\lceil \frac{n}{2^{i+1}} \rceil \times (\frac{n + 2^{i+1}}{2})\} + n}^{adder\ tree}$
cond. 4-input adder	$\overbrace{n + \lceil \frac{n}{4} \rceil \times \frac{(n+4)}{2}}^{partial\ product\ generation} + \overbrace{\sum_{i=1}^{\lceil \log_4 n \rceil - 2} \{\lceil \frac{n}{4^{i+1}} \rceil \times (\frac{n + 4^{i+1}}{2})\} + n}^{adder\ tree}$

Table 7: Number of CLBs used for parallel pipelined multipliers

The table 7 shows the number of CLBs to implement a n-bit pipelined parallel multiplier.

Figure 15 shows two CLB numbers and AR for input bit width from 8 to 64. The area reduction factor ranges from 2.6 and 4.2. The mean and median of area reduction factor are 3.78 and 3.92. CR is 1 for a fully pipelined parallel multiplier.

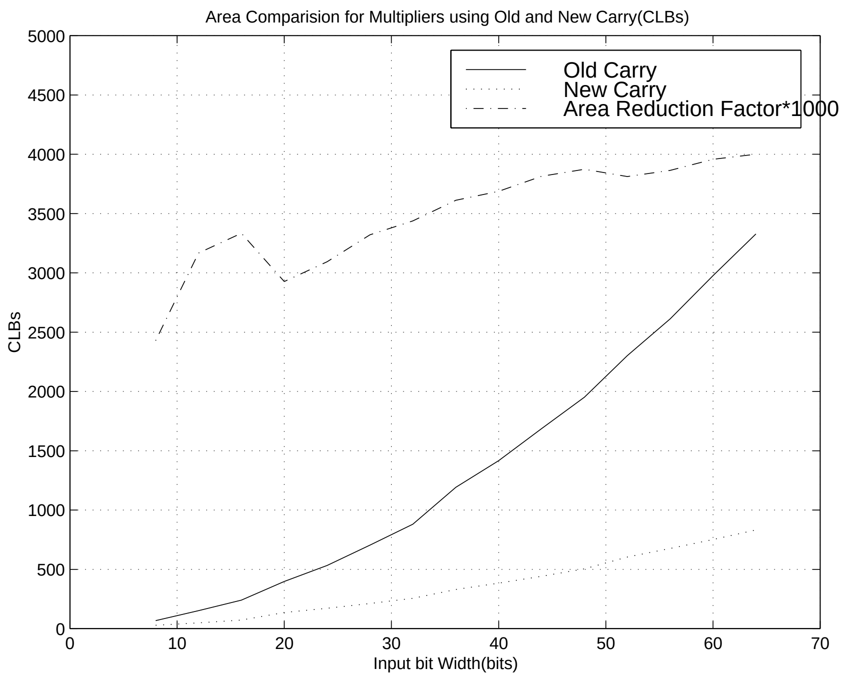


Figure 15: Area reduction for a multiplier

Integer matrix multiplication performs a multiplication of two $m \times m$ matrices in $O(m \log m)$ time with m^2 multipliers. M trees of $\log m$ adders are required to sum m^2 multiplier results. Based on this hardware requirement, table 8 shows the number of CLBs for integer matrix multiplication over n-bit numbers.

Operation	Number of CLBs
2-input adder	$m \log_2 m \times 2n + m^2 \times$#CLB for a n-bit multiplier
cond. 4-input adder	$m \log_4 m \times 2n + m^2 \times$#CLB for a n-bit multiplier

Table 8: Number of CLBs used for integer matrix multiplication

Unlike other applications, CR affects the throughput density. This is because the matrix multiplication requires an initialization step that prohibits fully pipelined implementation. To compute CR, total number of cycles to complete the entire matrix multiplication have to be determined. Current Xilinx XLA and XV series can hold 576(24x24) to 8464(92x92) CLBs with pin numbers of 192 to 448. Since the area for an n-bit parallel multiplier is relatively large, the throughput is not pin-bandwidth limited but area-limited. Under the assumption that in each cycle one row or column of matrix variables are loaded,

$$CR = \frac{1 + m + (1 + \log_2(m \times n))}{1 + m + (1 + \log_4(m \times n))} \tag{18}$$

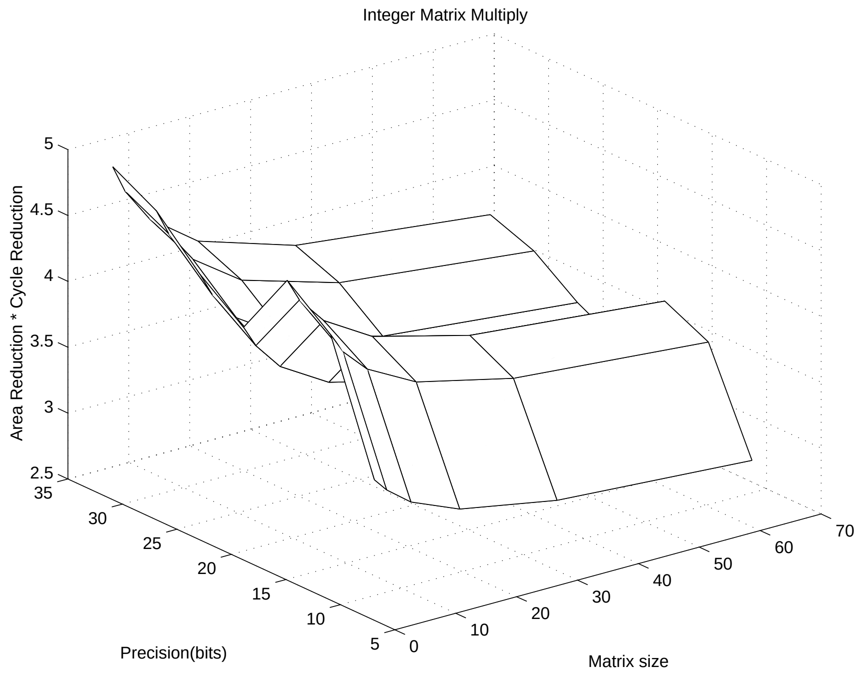


Figure 16: $AR \times CR$ for integer matrix multiplication

The first term is for loading Matrix A into a single or multiple chips. The second term represents Matrix B shift. The third term is the latency of a multiplier and an adder tree.

Figure 16 shows $AR \times CR$ with respect to m and n. It ranges from 2.8 to 4.9 with median = 4.0.

Binary heap implements the heapifying operation to convert a binary tree into a binary heap. A binary heap is a binary tree that satisfies the condition that the value at each node is greater than the values of children nodes. The key operation of this algorithm is three-way comparison. The three-way comparison in each node requires two comparators and one multiplexor to find the largest number. This can be implemented with a single comparator through time-multiplexing. Initially, a left and right child are compared. Then, according to the outcome of the comparison, the larger number and parent node's value are compared in the next phase. The outcome of two comparisons determines the swap patterns.

Each node in the tree can serve as a parent or a child. This requires a 4-to-1 multiplexor per node to execute the swapping in steady state or data load in initialization. Assuming input data size is n, each node takes $3(\frac{n}{2}+1)$ CLBs for the latched 4-to-1 multiplexor. This can be reduced to $\frac{n}{2}+1$ using conditional adders/subtracters as a multiplexor.

Thus, total area per node is reduced from $3(\frac{n}{2}+1)+\frac{3n}{2}$ to $\frac{n}{2}+\frac{n}{2}$. Roughly three times area reduction can be achieved. CR ~ 1 assuming heapifying time is dominant compared to initialization.

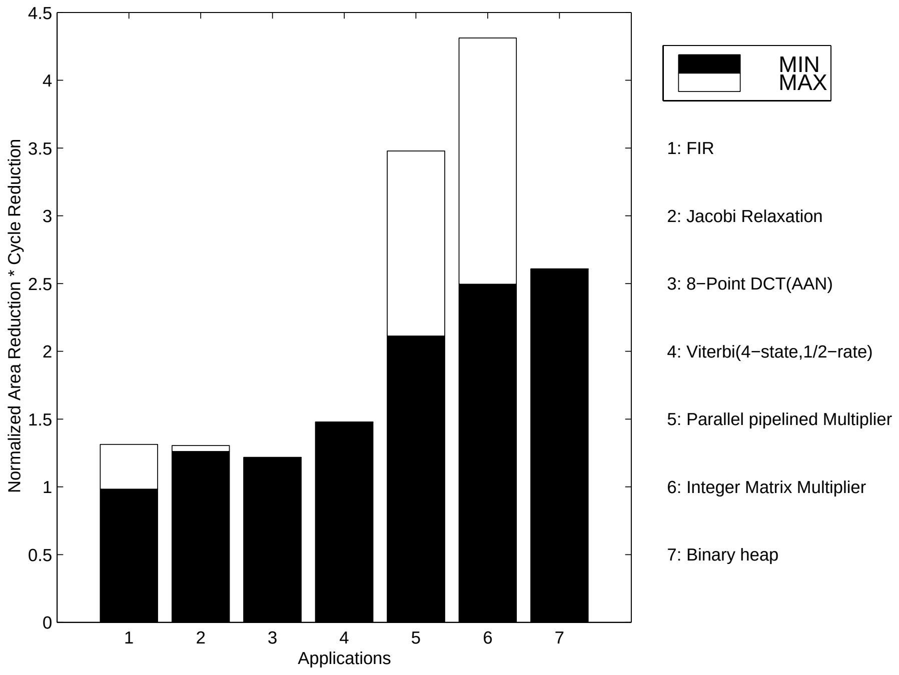


Figure 17: Normalized Area Reduction factor × Cycle Reduction factor

6.2 NAR × CR

Figure 17 shows the summary of normalized area reduction and cycle reduction factors for selected benchmarks. The values include the additional area cost for more complex carry architecture. Approximately 10 ∼ 15 % additional area was used due to the added complexity.

6.3 Throughput density gain measurement

Normalized area reduction factor, cycle reduction factor, and cycle time reduction factor are measured from placing and routing various applications with small problem size, figure 18. Area reduction factors range from 1.0 to 3.5, roughly 3 for applications requiring multiplications. AR for FIR filter is small compared to other applications. An explanation for this is that the throughput density for FIR filter implemented in FPGA is already comparable to that for custom design as shown in table 1. Cycle reduction factors range from 1.4 to 1.6 except for Jacobi relaxation which has 2.2 because two adders in the critical path is reduced to one adder. Throughput density gain ranges from 2 to 5 for most of applications except for FIR filter. Applications utilizing multipliers get the most benefit from the coarse grain carry architectures.

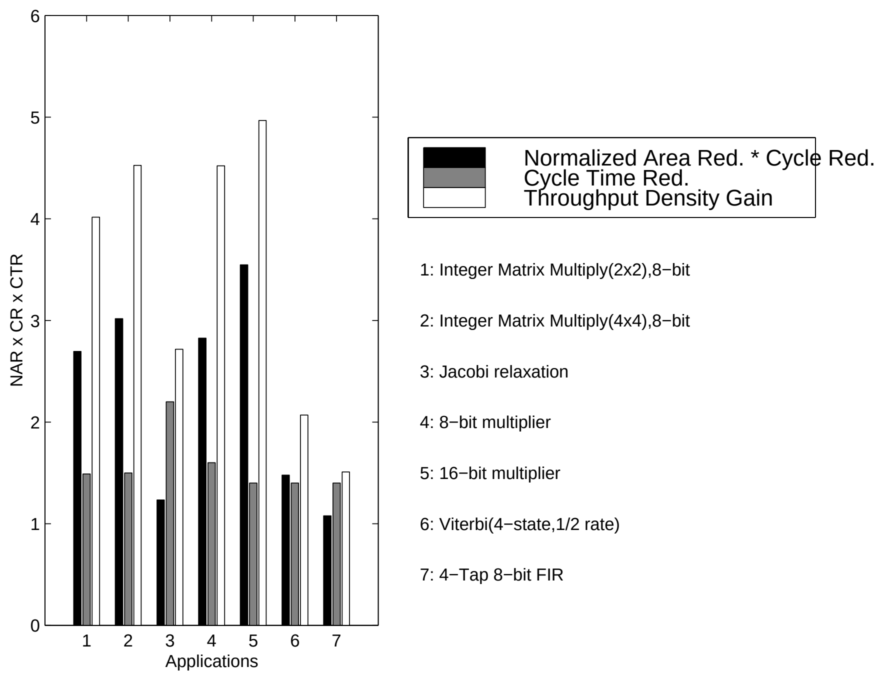


Figure 18: Throughput density gain

6.4 Results from SPICE simulation

6.4.1 high performance carry chain

The performance of carry select adders was determined by two factors: the transistor size in the carry select chain and the bypass segment size. Two factors are not mutually independent because the segment size affects the capacitive load that each transistor drives. SPICE simulation is done to measure the optimal size of the transistor and segment. Figure 19 shows three plots with bypass segment size of 4,8 and 12 bits. Each plot contains the carry delay for 8,16,32, and 64-bit carry select adders with four different transistor sizes.

In general, the delay with respect to the transistor size has a knee around $3.2\mu m$, which is 16λ in a $0.35\mu m$ process. For the segment size of 4 bits, the delay decreases as the transistor size increases for all input widths. For the segment size of 8 and 12-bits, the delay for 8 and 16-bit adders increases as the transistor size increases. This is because carry select adders with small input width don't get improved much from the large segment size, and larger transistors in the carry select path hurt the performance with larger capacitive loading.

The effect of different segment size can be seen by comparing the delay for adders of same transistor size in three different plots. The 4-bit segment adders outperform others for input widths up to 32 bits, which means that the fine segment size is effective unless we implement larger than 32-bits adders. The segment size of 12 bits only shows the best delay for a 64-bit adder. Figure 20 shows the delay of carry select adders(transistor size of 24λ) and a ripple carry adder. For a 8-bit adder the carry select adder with 4-bit segment has the

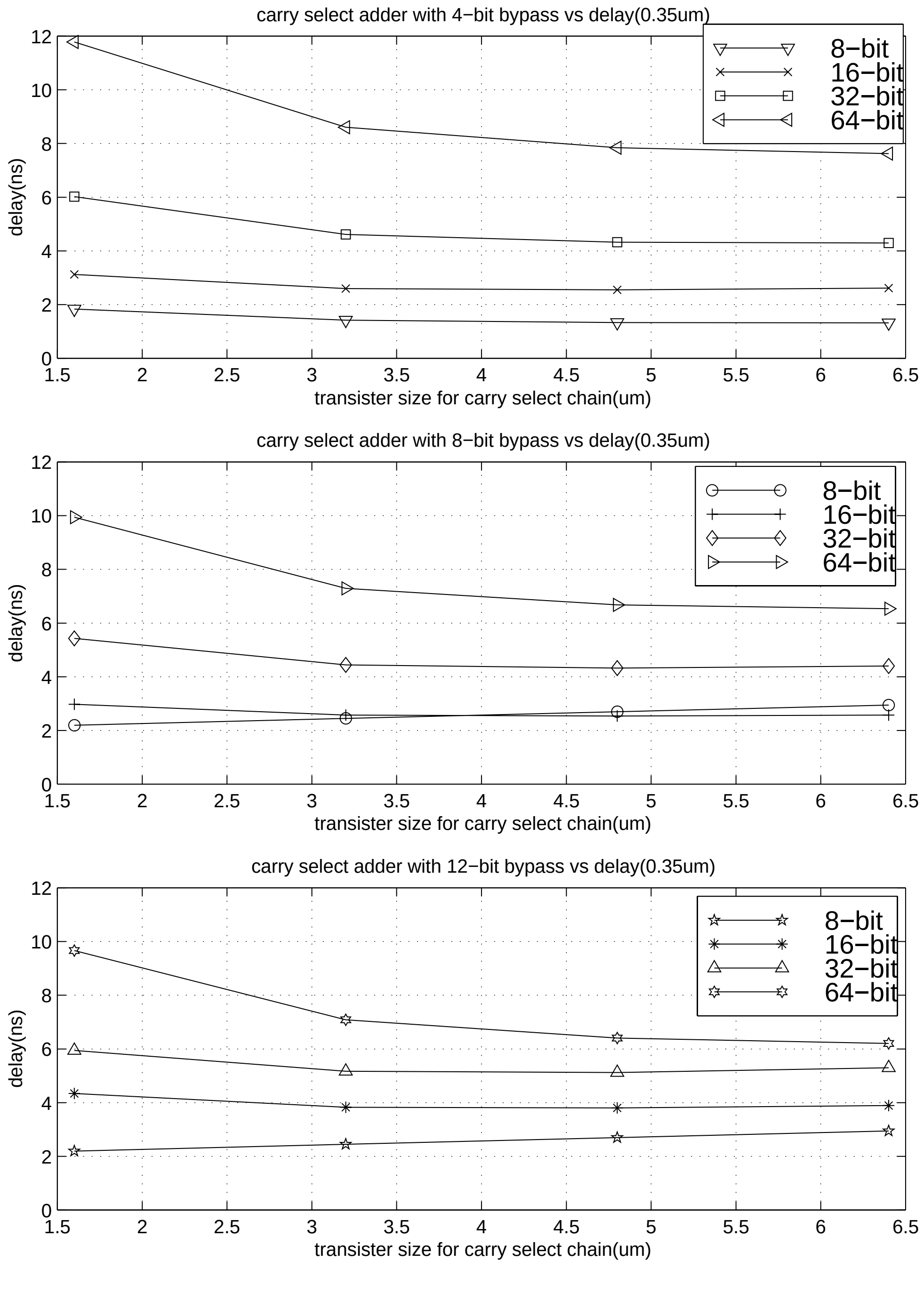


Figure 19: Transistor size in Carry Select Path vs. Delay(Carry Select Adder)

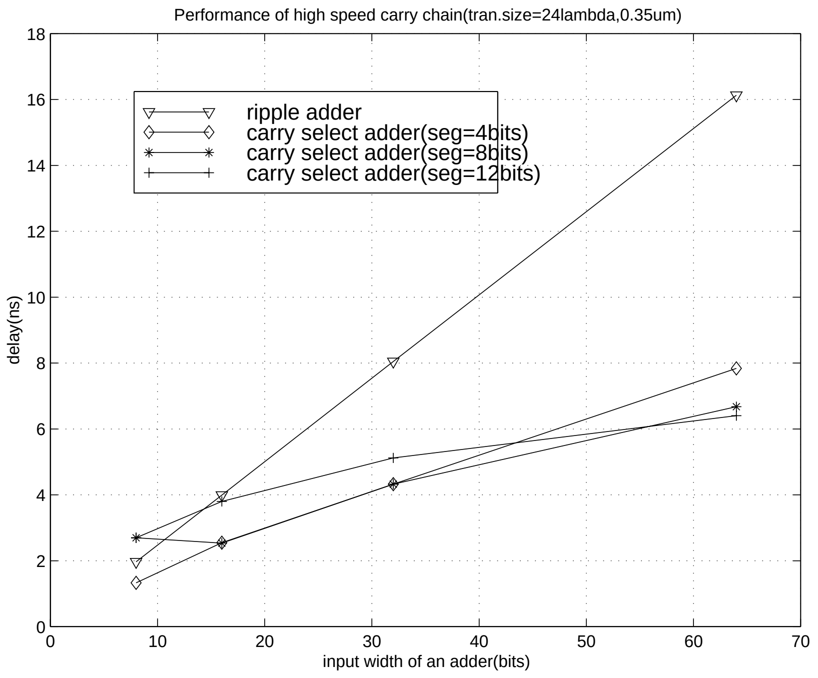


Figure 20: Performance of high speed carry chain

smallest delay, and adders with 8-bit and 12-bit segment perform worse than a ripple adder due to increased capacitance. For 16-bit and 32-bit addition, the adders with 4-bit and 8-bit segment show the best performance. For all cases 4-bit segment adders outperform a ripple carry adder roughly by a factor of two. This improvement set the upper bound for the cycle time reduction.

We also implemented variable segment size such as 4 and 8-bit segments or 8 and 16-bit segments. Each CLB in this implementation should be capable of providing one carry to two different segments and selecting one carry from two segments. Additional capacitance in these designs gave only marginal gain or showed worse performance over single segment design.

6.4.2 coarse grain carry architecture

The effect of adding (3,2) counters on the critical path delay in current carry architecture appears only in the first carry bit generation. The delay for the first carry generation increases to 1.74 ns from 0.72 ns.

7 Future Work

In this work, we limit the grain size such that addition of up to four numbers can be done in a single adder, which is roughly doubling the throughput of the current architecture. This limitation was set by simulation tools to measure the accurate the interconnect delay.

Larger grain size for addition would increase the throughput density gain more up to a certain point. Our future work involves the search for the point where increasing grain size no longer improves throughput density.

Another area we will focus on in the future is to measure the factor of energy saving through voltage scaling with a throughput constraint. The energy saving can be obtained at the cost of area, the reduction of which is promised by coarse grain architecture.

8 Conclusion

This report presented coarse grain carry architecture to increase the throughput density. The most gain comes from the area reduction or more functional capacity for the same area. Applications using multiplication get the most benefit and their normalized area reduction factors were measured from 2 to 4.5. The reasons for the reduction have two folds. First, a conditional adder/subtracter significantly reduces the area for multiplexors that are used for partial product generation and reduces the number of partial products. This accounts for roughly four times reduction. Second, area reduction rate for a tree of adders by increasing grain size grows larger than linearly. Theoretically, it grows exponentially but it is beset by the fact that each adder now needs larger bit width to contain the bigger result. Cycle time reduction is another factor that plays an important role in throughput density gain. Cycle time reduction comes from two reasons. First, shorter interconnect length from higher logic compression ratio gives shorter delay. Secondly, the number of CLBs in the critical path get smaller due to coarser grain size. These account for roughly $1.5 \sim 2$ times throughput density gain. In addition to the aforementioned reasons, conditional addition/subtraction provides the integration of control and ALU operation that can be utilized by adds followed by compares.

In addition to coarse grain carry architecture we implemented a dual-rail carry structure that can reduces the carry chain delay. Trade-offs between different transistor size and bypass segment size have been studied. Up to 2.4 reduction in carry chain delay is achieved using a carry select adder with bypass segment size of 8 bits.

References

[1] A. DeHon, "Comparing Computing Machines",*In Configurable Computing: Technology and Applications, Proceedings of SPIE 3526*, p. 124, November 1998.

[2] J.M. Arnold, et al., "The Splash 2 processor and applications",*IEEE International Conference on Computer Design*, Oct. 1993.

[3] Ray Andraka, "A survey of CORDIC algorithms for FPGA based computers",*International Symposium on Field Programmable Gate Arrays* , pp. 191-200, Feb. 1998.

[4] A. Takahara, et al., "More wires and fewer LUTs: A design methodology for FPGAs",*International Symposium on Field Programmable Gate Arrays* , pp. 12-19, Feb. 1998.

[5] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays",*Proceedings of the IEEE*, Vol. 81, No.7, pp. 1013-1029, July. 1993.

[6] S.Singh et al., "The Effect of Logic Block Architecture on FPGA Performance", *IEEE J. Solid-State Circuits*, Vol. 27, No.3, pp. 281-287, March 1992.

[7] S. Brown, "FPGA Architectural Research: A Survey",*IEEE Design & Test of Computers*, pp. 9-15, winter 1996.

[8] J. Babb and et al., "The RAW Benchmark Suite: Computation Structure for General Purpose Computing", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 134-43 , April 1997.

[9] O.Mencer, M.Morf, and M.Flynn, "PAM-Blox:High Performance FPGA Design for Adaptive Computing",*IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.167-174, April 1998.

[10] M. Chu, et al., "Object Oriented Circuit-Generators in Java",*IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.158-166, April 1998.

[11] G. Goslin, "A Guide to Using Field Programmable Gate Arrays(FPGAs) for Application -Specific Digital Signal Processing Performance", *Xilinx Corporation*, 1995.

[12] S. Hauck, "High-Performance Carry Chains for FPGAs",*International Symposium on Field Programmable Gate Arrays*, pp. 223-233, Feb. 1998.

[13] E. Kusse, "Analysis and Circuit Design for Low Power Programming Logic Modules", *Master Thesis, University of California, Berkeley*, 1998.

[14] Xilinx Corporation, "XC4000 Field Programmable Gate Arrays:Programmable Logic Databook", 1996.

[15] Xilinx Corporation, "Application Note #13, Using the Dedicated Carry Logic in XC4000E", June 1997.

[16] Xilinx Corporation, "Application Brief #14, A Simple Method of Estimating Power in XC40000 XL/EX/E FPGAs", June 1997.

[17] Y. Arai, et al., "A fast DCT-SQ scheme for images",*Transactions of the Institute of Electronics, Information and Communication Engineers*, vol.E71, no.11, pp. 1095-1097 Nov. 1988.

[18] *http://www.xilinx.com/xcell/best/xl14_28.pdf*