MULTI-BIT CARRY CHAINS FOR HIGH-PERFORMANCE RECONFIGURABLE FABRICS

Michael T. Frederick and Arun K. Somani

Dependable Computing and Network Laboratory
Iowa State University
Ames, IA 50011 USA
email: {freds,arun}@iastate.edu

ABSTRACT

Ripple-carry architectures are the norm in today's reconfigurable fabrics. They are simple, require minimal routing, and are easily formed across arbitrary cells in a fabric. However, their computation delay grows linearly with operand width. Many different fabric carry-chains have been presented in literature offering non-linear delays, but generally require a significant investment in routing and processing area. Carry-skip chains are well-known in arithmetic logic design, and although they too possess a linear delay, their performance is 2x or more faster than simple ripple-carry schemes. They require an expanded carry chain and minimal extra logic, but offer impressive speed-ups for arithmetic.

This paper presents a reconfigurable cell that supports carry-skip arithmetic using a multi-bit carry chain achieving $2 \cdot k \cdot b + \frac{n}{b}$ performance, where b is the block size and k is an architecture constant. The cell is specialized for arithmetic and Boolean operations with reduced configuration memory. Additional resources are provided to reuse the multi-bit carry chain for 3-source operand arithmetic to explore how multi-bit chains can be reused.

1. INTRODUCTION

Simple 1-bit ripple-carry chains logic limit the performance of reconfigurable fabrics. Using a minimal amount of extra logic and routing, cells optimized for arithmetic functions can be designed to efficiently implement 3-operand, carry-skip, and carry-save structures. This paper presents a reconfigurable cell that features a multi-bit carry chain for 2-operand carry-skip and 3-operand ripple-carry arithmetic. Fig. 1 shows high-level operating modes of the cell, including carry-save arithmetic (CSA) and Boolean logic.

FPGAs made by Xilinx and Altera incorporate ripple-carry schemes into all of their device families, including their flagship chips, the Virtex 4 [1] and Stratix II [2]. The width of each of these carry chains is typically 1-bit between

The research reported in this paper is partially supported by NSF grants numbers 0311061, 0306007, and the Jerry R. Junkins Endowment at Iowa State University.

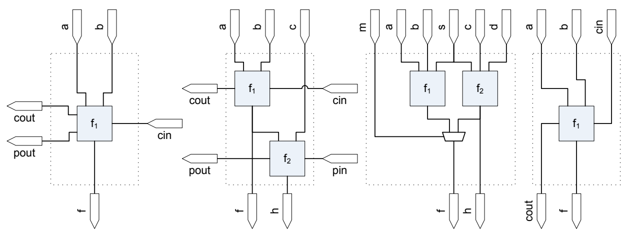

Fig. 1. (a) Carry-skip (b) 3-operand (c) Boolean (d) CSA

basic cells, with the exception of the Stratix II. Ripple-carry schemes are very straightforward to incorporate, exhibit linear delay with respect to datawidth, require minimal area, and are easily broken at any point along the chain.

Automatic cell-based ASIC design techniques are important in circuit design as they facilitate reuse and substantially reduce design time. Analysis of the trade-offs between area, performance, and testability for cell-based ASICs reveals how carry-chains can be readily employed in FPGAs, which are cellular in nature. Parallel prefix adders encompass a wide range of the most area/delay efficient architectures, including ripple-carry, carry-increment, and carry-lookahead adders [3]. The results indicate that the Brent-Kung, carry increment, and Sklansky parallel chains provide the best performance/cost trade-off. However, each requires the array to be viewed differently at different points within, requiring chains to be formed at a relatively few definite locations, reducing the flexibility of the array. As datapath width increases, the number of locations where chains can be formed within the array further decreases.

Advanced carry schemes specifically for FPGAs have been explored in literature, particularly in [4]. These include carry strategies such as Brent-Kung, carry-select, variable block, and variations of standard ripple-carry chains. The results indicate that advanced carry schemes are a very attractive solution for reconfigurable fabrics where arithmetic performance is paramount. Due to their FPGA-specificity and simple delay model analysis, the carry chains in [4] will be used as a point of comparison.

The Stratix II [2] uses a 2-bit carry chain in its Adaptive Logic Module (ALM). The first carry chain is driven by LUT logic, while a dedicated full adder drives the second carry chain. Both carry chains are ripple-carry, and allow the ALM to implement cascaded arithmetic operations, where the result of the first operation is input to the second. This allows the ALM to execute back-to-back arithmetic functions without requiring the result of the first operation to enter general routing, an expensive proposition in terms of latency and monopolization of routing.

The goal of this work is to present a reconfigurable cell using a carry-skip chain for high-performance arithmetic. Additionally, reuse of the multi-bit carry chain will be explored in the context of 3-input operand arithmetic, along the lines of those realized by the Stratix II.

2. DELAY MODEL

Using a simple CMOS gate-delay model similar to that in [4], simple gates (NAND, NOR, NOT) of 1 to 3 inputs are assumed to have a 1-gate delay. Complex 2 and 3 input gates (AND, OR) are assumed to have a 2-gate delay. Each 2:1 multiplexer uses transmission gates with a 1-gate delay for both the transmission gate and selection input inverter. Once the selection input is available, there is a 1 gate delay before the transmission gate can pass a value, an action that has a latency of 1 gate. If the selection input reaches the multiplexer before the other inputs, the multiplexer has delay of 1 gate after the input is available. XOR gates are implemented as 2 gate delay "tiny" XORs.

The effects of LUTs, as long as they are not on the critical path, are ignored because they are the same for all chains. If their delay must be considered, a delay of $lg(2^n)+1$ gates is imposed, where n is the number of inputs to the LUT. For example, a 2-LUT has a delay of 3 gates. The LUTs presented within are all 2-LUTs unless otherwise specified. Additionally, when the effects of general routing must be considered, a 2 gate delay is incurred. As a bare minimum, a signal must leave a source cell and, through a pass transistor, propagate on a signal line and enter the destination cell through an additional pass transistor.

3. ARITHMETIC OPERATIONS

Add/subtract, multiply, and word-width Boolean arithmetic operations are supported by the architecture. Carry-skip accelerate basic ripple-carry chains but rely on them as a basis for computation. Efficient implementation of the basic function and carry computation is the first problem that will be explored. Fig. 2 depicts the standard simple cell.

The subtract, multiply, and adder symmetry strategies in [5] are used to implement function computation performed by $f(x, y)$, a 2-LUT. The AND gate providing input x is used for multiplication, while the XOR providing y enables dynamic addition/subtraction. The XOR gate at the output of the 2-LUT performs output inversion as dictated by the carry into the cell, c_{i-1} or the signal c. The ability to choose between c_{i-1} and c is useful in chain initialization, carry-save arithmetic, and word-width Boolean operations.

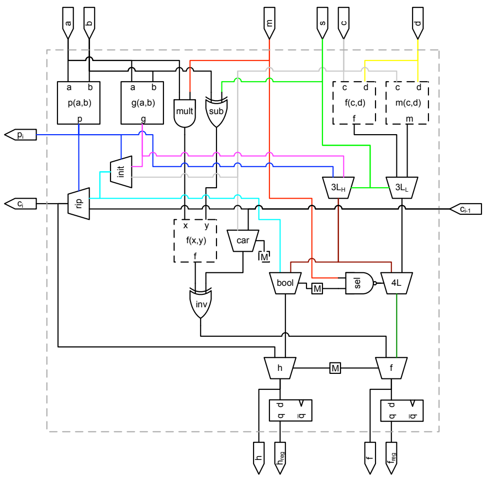

Fig. 2. Simple cell.

Word-width Boolean operations are (N)AND, (N)OR, or X(N)OR where the result is the gate operation performed on two inputs, a and b, with c_{i-1} having no effect on the local result. The c input can also be used to invert the output of *f(x,y)* and provide a mechanism for dynamically realizing a Boolean gate or its inverse, e.g. AND and NAND. The generated carry indicates something about all the bits in the operation. One option is to detect if an operation has been performed on operands with identical bit values. This is useful in determining if the application of a mask results in a value different from the input. Unary reduction is another possible application for carry generation in word-width Boolean functions. A simple Boolean gate is applied to each of the bits of the operand(s) with a 1 bit output, an example of which is parity generation.

Carry generation strategies such as the adder inversion property [5] and carry multiplexers, as used in Xilinx products [1], were explored to ascertain the most effective technique also allowing the result of the propagate condition to be output to the block logic. Dedicated propagate and generate LUTs were determined to be the most flexible and efficient design choice for the skip architecture. While the propagate and generate conditions can vary between arithmetic operations, one equation governing c_i, Eqn. 1, is used for all functions as long as the propagate and generate conditions are mutually exclusive. This is exemplified by the propagate condition for a standard adder. If $p = a + b$ and $g = a \cdot b$, the equation will fail, however if $p = a \oplus b$ the expression is valid. This can be implemented as a multiplexer,

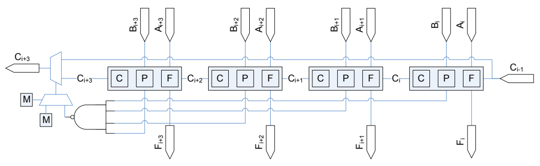

Fig. 3. A 4-cell carry-skip block.

whose selection signal is the propagate condition and inputs are generate and c_{i-1}. For most cases 1-gate delay carry generation is achieved once c_{i-1} becomes available.

$$c_i = \overline{p} \cdot g + p \cdot c_{i-1} \tag{1}$$

The basic operation of the carry logic adheres to Eqn. 1, with additional logic to initialize the chain with c, and, in the case of the complex cell, to select the block carry-in, c_{i-b}. The first block's final cell or block logic only accepts the carry in from the preceding cell because the entire chain must await the completion of the first block. Further influencing the design is that a priority has been placed on the architecture being able to realize 1-gate delay carry-skip logic and 1 or 2-gate delay ripple carry logic.

Carry-skip arithmetic is characterized by its ability to allow a carry bit to skip groups of cells known as blocks. If all of the cells within a block assert their propagate conditions, the carry into the block bypasses the block and becomes the input to the next block. If any cell within a block doesn't propagate the carry, the chain is effectively partitioned, and carry result i depends only on the operand inputs at cell i, not cells $i-1, ..., 0$. This allows each partition in the chain to compute its results in parallel. Blocks are introduced to define groups of cells that can take advantage of this phenomena. Fig. 3 depicts a carry-skip block where $b = 4$.

One of the drawbacks of the carry-skip strategy is the existence of false logic paths. These occur because the carry out of a block does not necessarily depend on the carry into the block. Such paths present a problem primarily in the timing of the circuit. The authors of [6] present a detailed description of the redundancy and false path problems within carry-skip chains and provide a means to optimizing the redundancies that lead to incorrect estimation of chain timing.

Computation delay in a carry-skip architecture is described by Eqn. 2. The delay represents the generation of a carry in the first cell of the chain that traverses the ripple-carry chain to the end of its block. As all other blocks perform ripple computation in parallel, the delay through the carry-skip chain is the carry generation through the first block, the block carry selection multiplexer in the skip logic of each of $\frac{n}{b}$ blocks, and the ripple-carry delay through the last block. Each of the $\frac{n}{b}$ skip multiplexers contributes 1 gate delay, as the inversion of the selection bit has already happend when the carry reaches the multiplexer. Eqn. 2 shows the delay of a carry-skip architecture. In general, $T_{first} = T_{last} = T_{ripple} = k \cdot b + c$, where k is a recurring cost, c is a chain initialization cost, and b is the length of a block. The optimal block size is found by taking the derivative of Eqn. 2 with respect to b (Eqn. 3), setting it to 0, and solving for b [7].

$$T_{skip} = T_{first} + \frac{n}{b} + T_{last} = 2 \cdot k + c + \frac{n}{b} \tag{2}$$

$$\frac{dT_{dly}}{db} = 2 \cdot k - \frac{n}{b^2} = 0 \longrightarrow b = \sqrt{\frac{n}{2 \cdot k}} \tag{3}$$

Carry generation in the skip chain has been implemented in two different ways, a simple cell structure with block logic available only among predefined groups of b adjacent cells, and a complex structure that incorporates block logic at each cell and allows blocks of up to b cells to be formed among any set of adjacent cells.

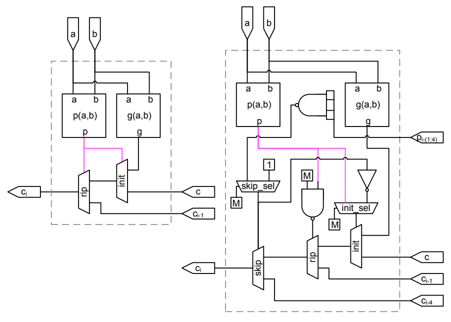

Fig. 4. (a) Simple and (b) complex carry logic.

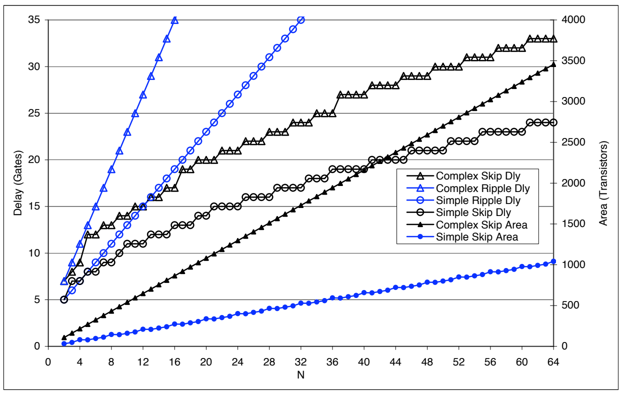

Fig. 5. Simple and Complex Area/Delay Trade-offs.

The complex cell allows carry-skip blocks to be created out of b adjacent cells located anywhere. It requires each cell to have a carry selection multiplexer (*rip*), a block propagation signal (p_{blk}), and a carry-skip selection multiplexer (*skip_sel*). The skip-MUX chooses between c_{i-b} and c_i.

The block propagation signal is the NAND of the b propagation signals, and controls the skip-MUX. The skip_sel-MUX chooses between the block propagation signal and a programmable selection bit. When set to 0, it allows the intermediate cells of the skip chain to operate in ripple-carry mode. If set to 1, it signifies the end of a block and allows the block propagate signal to control carry selection. With the collusion of other cells that may not have a vested interest in a carry skip computation, a chain can be constructed of variable block widths up to b.

The complex cell offers a more flexible array by allowing chains to be formed among any group of up to b adjacent cells. However, as depicted in Fig. 5, the complex cell requires a 3x the area, and incurs a delay roughly twice that of the simple cell. These results indicate that the complex cell is most likely not a cost efficient design choice relative to the simple implementation. Therefore, for the remainder of this paper, only the simple cell will be considered.

4. BOOLEAN AND MULTIPLEXING OPERATIONS

An effective fabric must support Boolean and multiplexing operations. The defacto standard for Boolean operations in reconfigurable fabrics is the 4-LUT, requiring 16 bits, while a 4:1 1-bit multiplexer requires 6 inputs. These two requirements drive the design of the Boolean logic. The most efficient way to realize both is to create two 3-LUTs using the three existing 2-LUTs and reuse as many carry-chain configuration bits to form the last 2-LUT. Since only one bit of the standard carry-chain can be reused, three Boolean-specific bits have been introduced. Shared bits are shown as dashed lines in Figs. 2 and 6.

Two 3-LUTs are the most efficient design choice as they allow 1 shared (selection bit s) and 4 unique inputs of the 4:1 multiplexers to drive each LUT, and another input (selection bit m) to multiplex the LUT outputs. Additionally a 2-bit 3-LUT and 2-bit 2:1 multiplexer can be formed in this configuration. The remaining shared input also provides output selection through the NAND gate. When the programming bit is set to 1, the selection signal m controls the multiplexer output, and when set to 0 the multiplexer selects the output of one of the $3L_L-$LUT for 2-bit Boolean/multiplexing.

5. THREE-OPERAND ARITHMETIC

Three-input operand arithmetic has the goal of reusing the multi-bit carry-skip chain without interfering or introducing any delay in the chain to harm carry-skip performance. Thus, each cell can only drive the carry and propagate signals that it sources, but can read any propagate signal passing by. For three-operand arithmetic, this requires that only the propagate signal from the preceding cell is read.

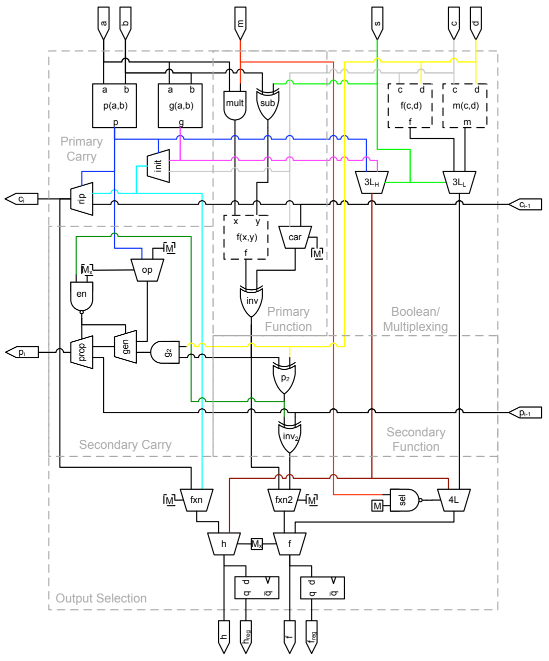

Fig. 6. The enhanced cell.

Depending on the secondary chain implementation, operations can include any combination of two arithmetic operations described in Section 3 if LUTs are used (not pictured), or an arbitrary primary arithmetic operation and a secondary addition/subtraction as pictured in Fig. 6. The performance of 3-operand arithmetic depends on the ripple carry latency through the longest carry chain. The primary carry chain produces the longest propagation delay because the block logic still resides on the carry chain, even when operating in ripple-carry mode. The latency of 3-operand arithmetic is the 3 gate delay through the primary LUTs, the ripple carry delay through the primary chain, and the lag between when the primary function result is known to when the secondary function result can be computed, yielding $3 + n + 1 + \frac{n}{b} + 5 = n + \frac{n}{b} + 9$.

The secondary arithmetic logic can be implemented in the same way as the primary arithmetic logic or as a dedicated full adder like the Stratix II ALM [2]. Here the function computation is the standard XOR of the inputs and the carry, however, the carry is produced using a multiplexer to preserve a 1-gate delay secondary ripple chain. This is important because p_i must fulfill its function in carry-skip arithmetic, but also perform ripple-carry as quickly as the primary chain to be a feasible solution.

An alternative means of carry and function implementation is to use the same LUT structure as the primary carry chain. The advantage of an LUT implementation is that it is inherently more flexible and can realize the same scope of functions as the primary arithmetic logic. The drawback is that it requires roughly twice the area and configuration space. To justify such an expenditure, the performance of 3-operand arithmetic must be weighed against back-to-back carry skip (Skip/Skip) operations and a carry-save operation terminated by a carry skip (CSA/Skip).

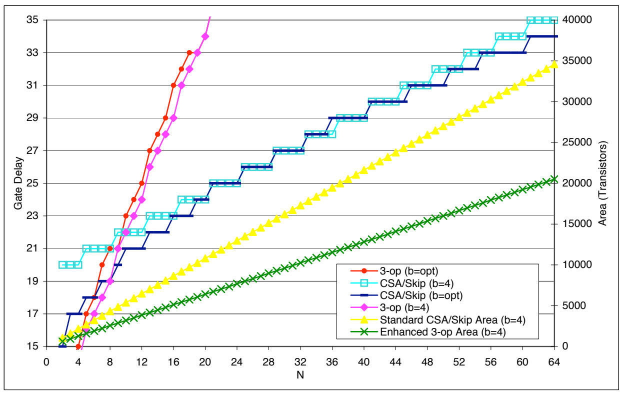

Fig. 7. Three-operand performance comparison.

If the Skip/Skip performed, the second operation consumes the result of the first and the worst case delay depends on the first function bit becoming available and allowing the second skip-chain to begin. This requires the result of the first operation to pass through general routing to reach the second, which is assumed to be a 2 gate delay. The first bit is ready after a delay of 7 gates, undergoes a 2 gate delay in general routing, and a 3 gate delay through the first 2-LUT before entering the skip-carry chain, and incurs the carry-skip propagation delay, making the overall delay $7+2+3+2\cdot b+\frac{n}{b}+1 = 2\cdot b+\frac{n}{b}+13$.

Carry-save arithmetic can also be used to evaluate 3-operand instructions, provided the same associative operator is used for both operations. The first stage of the computation is to reduce three operands to two values. Once the CSA stage completes, the two values would then become the input to cells configured to perform carry-skip arithmetic. The end result is a 3-operand computation that requires a delay of 7 gates for carry-save operation evaluation, 2 gates for general routing, 3 gates for the skip LUTs to evaluate, and the standard delay for a carry-skip operation yielding the expression $7+2+3+2\cdot b+\frac{n}{b}+1 = 2\cdot b+\frac{n}{b}+13$.

An advantage of the enhanced cell is that it uses the same configuration space than the standard cell. The bits necessary to configure the secondary chain can be reused in Boolean logic, and therefore don't increase the bits per cell. Conversely, the CSA/Skip and Skip/Skip implementations require 2x the cells, and, consequently, double the bits. Inter-cell communication also requires bits to configure general routing and consumes valuable routing resources.

Fig. 7 shows delay for 3-operand ripple-carry, Skip/Skip, and CSA/Skip operations. Because the delay expressions for Skip/Skip and CSA/Skip operations are the same, only one has been plotted. The area is expressed in total transistors and reflects the CSA/Skip and Skip/Skip operations being performed using the standard cell and 2 rows of the array. The area effect of general routing has been neglected, but its delay has been included as a 2 gate penalty. The 3-operand operation uses the enhanced cell, but occupies one row of the array.

The results indicate that the enhanced cell is faster at performing 3-operand arithmetic for datapath widths of less than 10 bits. Additionally, as it occupies only n cells, it requires about 60% of the area of the CSA/Skip, Skip/Skip implmentations which requre $2\cdot n$. For small datawidth applications, enhanced cells are a better design choice from the viewpoint of both delay and area impact. For increasing datawidths, the standard cell would be the best choice as area increases at a much slower rate than the delay penalties associated the ripple-carry nature of 3-operand arithmetic. If area and configuration space are bigger design considerations than performance, the enhanced cell is a better choice.

6. PERFORMANCE COMPARISON

To get an idea of how the carry-skip chain compares with other high performance carry chains for FPGAs, the work in [4] has been used as a point of comparison. Delay characteristics of various carry-chains are shown in Fig. 8 and indicate that the standard carry-skip chain with simple cells performs better than, or comparably to, more advanced carry chains such as Brent-Kung for datawidths of 16 bits and less.

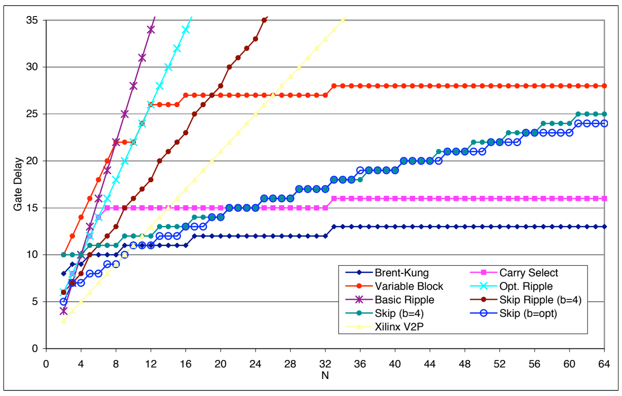

Fig. 8. Simple cell performance vs. other carry-chains.

As datawidth increases, the advanced chains perform increasingly better than the skip chain. The performance gap at 64-bit data is less than 2x between the skip chain and the best advanced chain, the Brent-Kung. The area estimations are transistor counts of just the carry chains of the respective strategies. For the case of the simple skip-cell, this includes the logic necessary to implement the extra p-LUT, but not the configuration bits, as they are reused from Boolean

Table 1. Bitstream Lengths

Fabric	Bytes	32-bit Mem. Latency
DSP-FPGA [5]	34	10
Standard	72	18
Enhanced	72	18
CHESS[a] [8]	100	26
V2P[b] [1]	128	32
Stratix II[c] [2]	176	44
Garp [9]	196	48
Chimaera [10]	208	52
PRISC [11]	256	64

[a]CHESS is a 64-bit datapath, this represents it as 32-bit.
[b]Estimated value based on 16 slices.
[c]Estimated value based on 16 ALMs.

logic. When compared to the area requirements of the other chains, the carry-skip cell mimics the optimized ripple and outperforms the other advanced carry schemes.

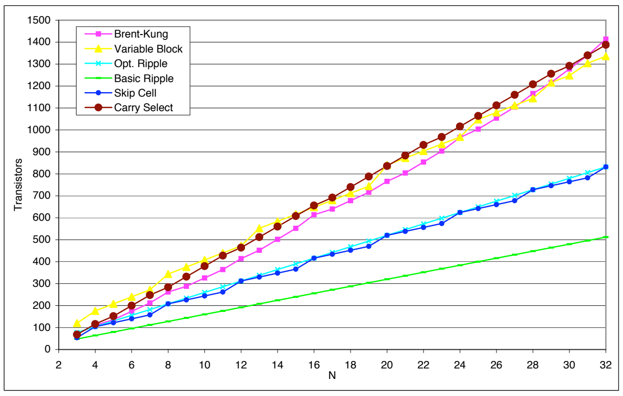

Fig. 9. Area of the skip cell carry-chain relative to others.

The basic cell architecture's configuration needs compare well with existing fabrics. Configuration sharing is used to reduce memory requirements at the expense of increased processing layer area. This reduces overall array area, SRAM exposure to SEUs, and improving the temporal characteristics of the fabric. The enhanced cell requires the same configuration space as the standard cell, making it attractive from a dynamically reconfigurable standpoint.

7. CONCLUSION

In this paper a cell architecture for carry-skip arithmetic has been presented. It strikes a good balance between delay and area when compared with more advanced carry-chains. While skip chains are not a new concept for adder design, the presented cell architecture facilitates such arithmetic in FPGAs while also being useful for standard Boolean and multiplexing operations. Because of the skip chain's extremely "flat" topology, regularity, and ability to be created and broken anywhere, it is an attractive solution for high-performance regularly-structured reconfigurable fabrics.

A novel reuse of the carry-skip multi-bit carry chain has also been presented. By introducing extra logic, the multi-bit carry-skip chain can be reused for 3-input operand ripple-carry arithmetic operations. These operations consume less area than sequential carry-skip or carry-save/carry-skip operations, and perform comparably or better for datawidths of less than 16 bits and can make use of synthesis tools already developed for the Stratix II 2-bit ripple-carry chain.

Future work includes identifying other ways to reuse multi-bit carry chains for reconfigurable fabrics. Additionally, the carry-skip block logic can be enhanced to realize wide Boolean and multiplexing, distributed RAM, and fault tolerant operations. Multi-bit carry chains have the potential to increase the flexibility and efficiency of general reconfigurable fabrics by accelerating arithmetic computation and providing resources for advanced or wide operations.

8. REFERENCES

[1] Xilinx, *Virtex II Pro User Guide*. http://www.xilinx.com.

[2] Altera, *Altera Stratix II User Guide*. http://www.altera.com.

[3] R. Zimmermann, "Non-heuristic optimization and synthesis of parallel-prefix adders," in *Proceedings of the Int'l Workshop on Logic and Architecture Synthesis*, December 1996.

[4] S. Hauck, M. M. Hosler, and T. W. Fry, "High performance carry chains for fpgas," *IEEE Transactions on VLSI Systems*, vol. 8, no. 2, April 2000.

[5] K. Leijten-Nowak and J. L. van Meerbergen, "An fpga architecture with enhanced datapath functionality," in *Proceedings of the 11th Int'l Symposium on FPGAs*, 2003, pp. 195–204.

[6] K. Keutzer, S. Malik, and A. Saldanha, "Is redundancy necessary to reduce delay?" *IEEE Transactions On Computer-Aided Design*, vol. 10, no. 4, pp. 421–435, April 1991.

[7] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Design*. Morgan Kauffman, 2000.

[8] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proceedings of the 7th Int'l symposium on Field programmable gate arrays*, 1999, pp. 135–143.

[9] J. R. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997, pp. 12–21.

[10] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The chimaera reconfigurable functional unit," *IEEE Transactions On VLSI Systems*, vol. 12, no. 2, pp. 1063–8210, February 2004.

[11] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th Int'l symposium on Microarchitecture*, 1994, pp. 172–180.