The Report Committee for Vignesh Naganathan
Certifies that this is the approved version of the following report:

A Comparative Analysis of Parallel Prefix Adders in 32nm and 45nm Static CMOS Technology

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Earl Swartzlander

Lizy John

A Comparative Analysis of Parallel Prefix Adders in 32nm and 45nm Static CMOS Technology

by

Vignesh Naganathan, B.Tech.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2015

Dedication

I would like to dedicate the report to my parents Naganathan Venkataraman and Radha Balasubramanian. Without their love and unconditional support, I could not have completed this degree away from my home country. I would also like to dedicate it to my fiancé, Anusha Balan, for her support and encouragement.

Acknowledgements

I would like to thank my supervisor, Dr. Earl Swartzlander, for his inspirational support and guidance on helping me to work towards a Masters degree. His strong technical support and initial encouraging words guided me towards the completion of the report for this degree. I thank Dr. Lizy John for her contributions as well.

I would also like to acknowledge the support and assistance given by Vignesh MG and Balavinayagam Swaminathan for contributing ideas and feedback.

Abstract

A Comparative Analysis of Parallel Prefix Adders in 32nm and 45nm static CMOS Technology

Vignesh Naganathan, M.S.E

The University of Texas at Austin, 2015

Supervisor: Earl Swartzlander

Binary adders form a major part in various arithmetic logical operation units including multipliers, dividers and digital signal processors. Parallel prefix adders represent a set of efficient structures for binary addition, greatly suited for VLSI implementation due to their regularity and speed. This report is focused on the comparative analysis of 5 major types of parallel prefix adder frameworks namely Kooge-Stone, Knowles adders, Brent-Kung, Han-Carlson and Ladner-Fischer adders implemented in Synopsys's SAED 32nm static CMOS technology operating at 1.05V for 8-bit, 16-bit and 32-bit input vectors based on power, performance and area (PPA) metrics. The process technology is modeled with 9 metal tracks. Power, performance and area metrics based on circuit simulations are used for comparison. The metrics are compared across SAED 32nm and FreePDK 45nm technology to quantify the impact of technology on architecture.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

Addition is the most common arithmetic operation and binary adders are used widely in almost all the arithmetic operations in the modern digital systems. Multipliers, dividers, arithmetic logic units (ALU), digital system processors (DSP) among others extensively use binary addition operations. Most often, the performance of the digital systems depends critically on the performance of binary adders. In addition, power efficiency is very critical in portable electronic systems like smartphones and tablets with limited battery power. Hence, binary adders are required to be faster, smaller and extremely power efficient.

The critical path in a binary adder is the carry out computation path from the inputs. The propagation delay in the carry chain limits the performance of the binary adders. In conventional ripple carry adders, as the width of the input vectors increases, the length of the carry chain increases. This carry-propagation problem can be efficiently addressed by parallel prefix computation. Parallel prefix adders (PPA) are variations of the well-known carry look ahead (CLA) adders. They are the fastest and most efficient computation structure for binary addition in VLSI digital systems because of the regularity and parallel execution nature. The parallel prefix adders attain logarithmic time complexity and the propagation delay is directly proportional to the number of levels in carry propagation logic.

This report focuses on the performance of the different parallel prefix adders implemented in SAED 32nm and FreePDK 45nm static CMOS technology node. The

parallel prefix adders analyzed in the report are Kogge-Stone adder, Brent-Kung adder, Han Carlson adder, Ladner Fischer adder and Knowles family of adders. The power, performance and area (PPA) metrics are used to perform the comparative analysis of these parallel prefix adders. Based on the simulation studies, Kogge-Stone adder is the fastest adder while it burns the highest power due to its extensive parallel prefix operations while Brent-Kung adder is the smallest adder and burns the least power due to its prefix tree structure.

1.1 Literature Review

There are many ways of approaching the process of binary addition providing different insight, resulting in different implementations. The simplest adder architecture is ripple carry adder [1] where every block computes a 1-bit sum and provides the resulting output and the carry bit for the next 1-bit adder. The worst-case delay is linearly proportional to the width of the input operands. Some other adders like bit-serial adders and Manchester Carry chain adders also have at least linear time complexity with respect to word width of inputs.

Weinberger and Smith's carry-lookahead adders (CLA) [2] are commonly used structure for logarithmic time addition. CLA employ multiple levels of Manchester carry chains of generate, propagate and kill signals to solve carry-propagation problem. The carry-skip adder provides a compromise between ripple carry adder and a CLA adder. It splits input vectors into multiple bit groups or blocks and then computes "group propagate" signals for each block of inputs to establish bypass or skip paths around that blocks to speed up the carry propagation.

Sklansky's conditional-sum adder [3] realizes addition with modules by performing computation of conditional sums and carries that result from the assumption of all the possible distributions of carries for column groups. Bedrij's carry-select adder [4] approaches the carry-propagation problem by independently generating multiple-radix carries and using these carries to select between simultaneously generated sums. Since input operands are added twice to produce two sums with carry assumed in one addition and carry not assumed in another addition, it is not efficient in terms of cell count and area. Several variants of high-speed adders include Nadler's pyramid adder [5], Ling adder [6] and spanning tree carry-lookahead adder [7].

The prefix formulation is an excellent approach to compute carry propagation network as adders implemented based on parallel prefix operators can be implemented compact and regular in VLSI. The associativity and idempotency properties of prefix operators give extreme flexibility in formulating various variants in prefix addition algorithm and their implementation in VLSI. The prefix computation approach and the existing variants of prefix tree algorithms are briefly explained in the coming section.

1.2 Prefix computation approach

For n-bit addition, where n is a power of 2, a minimum-depth prefix adder comprises 3+log2n inverting gate stages in CMOS technology. The first stage of a prefix adder computes carry generate (g), propagate (p) and kill (k) terms for each bit according to the relations:

$$g_i = a_i \cdot b_i$$

$$k_i = !(a_i + b_i)$$

$$p_i = a_i \oplus b_i$$

A carry is generated if both the input addend bits of the particular stage are ones and an input carry to a particular stage is propagated to the next stage if one of the addend bits is one. Thus, based on the generate and propagate signals, the carry bits of each stage are derived by:

$$c_{i+1} = g_i + p_i c_i$$

The final stage computes sum bits as:

$$s_i = p_i \oplus c_i$$

Thus summing up the above steps, in general carry lookahead adders have a 3-step structure:

- Pre-computation stage of generate g_i, propagate p_i signals for each bit position
- Computation of carry C_i for each bit position
- Post-computation stage of combining C_i, and P_i to generate sum S_i for each bit position

The first and last stages are fast because they involve simple operations on signals local to each bit position. The intermediate stages are used to compute carry propagation network using prefix operation •in a parallel prefix adder. The prefix operation •

$$(g_x,p_x) \bullet (g_y,p_y) = (g_x + p_x.g_y, p_x.p_y)$$

The carry into any bit position can be computed using a chain of prefix operations:

$$(C_{i+1},P_i) = (g_i,p_i)\bullet(g_{i-1},p_{i-1})\bullet(g_{i-2},p_{i-2})\bullet \ \ldots\bullet(g_0,p_0)$$

The problem of carry determination can be formulated as follows [8]:

Given

(g0,p0), (g1,p1)…(gi-1,pi-1),(gi,pi)

Find

(g0…0,p0…0),(g0…1,p0…1)…(g0…i-1,p0…1-1),(g0…i,p0…i)

Since the prefix operator is associative and idempotent, these operations can be performed in greater parallelism, allowing parallel prefix adders to be much faster circuits than any of the other adder implementations.

$$(g_{h\ldots j},p_{h\ldots j})\bullet (g_{j\ldots k},p_{j\ldots k}) = (g_{h\ldots i},p_{h\ldots i})\bullet (g_{i\ldots k},p_{i\ldots k}) \text{ where } h > i \geq j > k$$

$$(g_{h\ldots j},p_{h\ldots j})\bullet (g_{i\ldots k},p_{i\ldots k}) = (g_{h\ldots k},p_{h\ldots k})$$

Thus, the carry problem is converted to a parallel prefix operations and there are various prefix computation schemes to find all the carries. This multilevel-lookahead idea is to compute small group of intermediate prefixes and then find large group prefixes, until all the carry bits are computed.

CHAPTER 2

PARALLEL PREFIX ADDERS

Adders in which the computation of carries is based on the above-defined prefix equations are called “prefix adders.” When multiple sub-terms are computed in parallel by exploiting associative and/or idempotency properties, then the prefix adders are called “parallel prefix adders.” These structures are very commonly used in high performance adders because the delay is logarithmically proportional to the input operand width. Based on the variations of the prefix tree structure, there are several types of parallel prefix adders. Trade-offs involved in different prefix tree structures include

- radix/valency
- area of the adder
- logic tree depth
- fan-out of the nodes at each stage
- the overall wiring network

In this report, 5 major parallel prefix adders, namely Ladner-Fischer, Kogge-Stone, Brent-Kung, Han-Carlson, and Knowles family of adders are analyzed from a VLSI designer point of view. Additional insight on how the architecture and minimum feature size of SAED 32nm and FreePDK 45nm impact circuit technology.

2.1 Taxonomy

For any *n*-bit prefix trees with fixed radix of 2 (the number of inputs to the logic gates is always 2 for the PPA discussed in the report), design trade-off is made among logic levels, fan-out and wire tracks. Hence the taxonomy, as proposed by D. Harris[9], uses the (*l*,*f*,*t*) with each variable representing logic levels, fan-out and wire tracks, respectively. *l*,*f* and *t* are integers between 0 and $log_2(n)$ *-1*.

- Logic levels: $log_2(n) + l$
- Fan-out: $2^f + 1$
- Wire tracks: 2^t

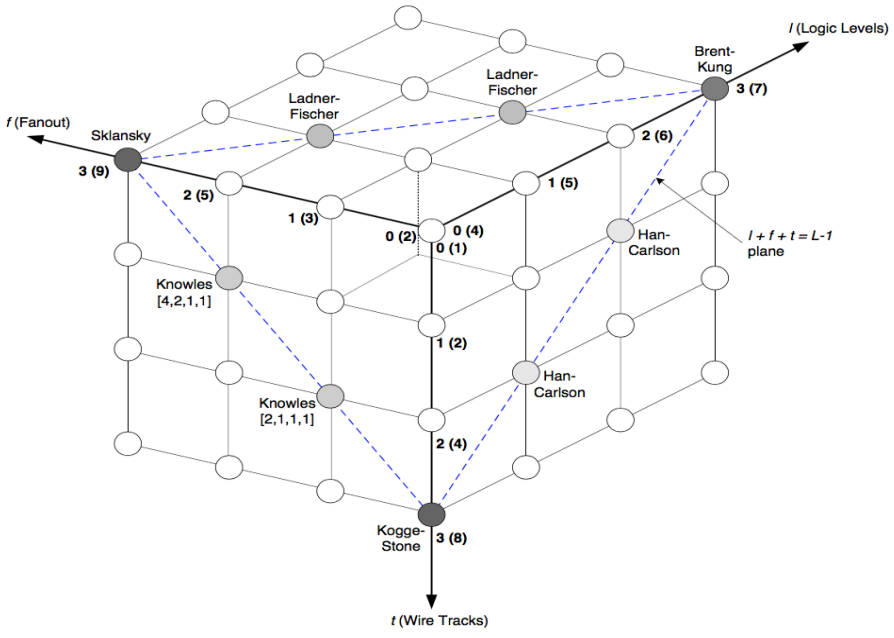

Figure 1. Taxonomy of 16-bit prefix tree from D. Harris

Based on the taxonomy, Kogge-Stone prefix tree ($0,0,log_2(n)-1$) has the least logic levels and fan-out. However, the wire track is $2^{log2(n)-1}$, which is the maximum among the prefix structures. This results in a dense gate structure compromising on area and power for performance gain. Brent-Kung prefix tree ($log_2(n)-1,0,0$) has the least fan-out and wiring track. However it requires the most logic levels among the prefix trees. Therefore Brent-Kung compromises the speed significantly for smallest area and least power dissipation. Ladner-Fischer prefix tree ($log_2(n)-2,1,0$) employs smaller number of logic levels and wiring tracks while the fan-out increases with wider inputs towards the later stages of prefix computation graph. This results in significant loss in performance in CMOS implementation due to increased capacitive output load on last stage drivers for wider words. Knowles family ($0,f,t$) has the least logic levels while fan-out and wire tracks depend on the specific prefix structure. This family has a high dense gate structure. Han-Carlson prefix tree ($1, 0, \log_2(n)-2$) reduces the gate density by introducing one extra logic level than minimum. However the number of wire tracks is logarithmically proportional to input width.

2.2 Prefix graph convention

The general convention used in the prefix graph shown in the coming sections is group generate/propagate signals are the only signals used in the purple colored circular dots, which represent prefix operators. The group generate/propagate signals are based on the single bit generate/propagate signals computed in the pre-computation stage. They are represented by white colored circular dots. Solid lines show the lateral connectivity

required between nodes at each stage while the dashed lines show the implicit vertical connections between the nodes in the same column.

CHAPTER 3

PARALLEL PREFIX ADDER SCHEMES

3.1 Kogge-Stone adders

Kogge-Stone adder [10] is one of the widely used prefix tree structure for high performance adders. It employs fewest logic levels with maximum fan-out limited to 2 in all logic levels for all width Kogge-Stone prefix trees. It is one of the members of the Knowles family of adders with the special case of the maximum branch fan-out at each level limited to 1. For example, 16-bit Kogge-Stone adder as shown in the figure below can be expressed as Knowles [1,1,1,1] where the numbers in the brackets represent the maximum branch fan-out at each logic level.

Kogge-Stone adders achieve very high performance by extensive parallelism of prefix operator execution employing both associativity and idempotency. Idempotency property limits the lateral logical fanout at each node to unity but dramatically increases the number of lateral wires at each level. The increased wire tracks result in highly dense gate structure. Hence wire capacitance is high even though logical fanout is minimized. This results in requirement of buffering to drive higher wire capacitance. Due to extreme parallel execution, the total power dissipated by Kogge-Stone is highest among the prefix adders.

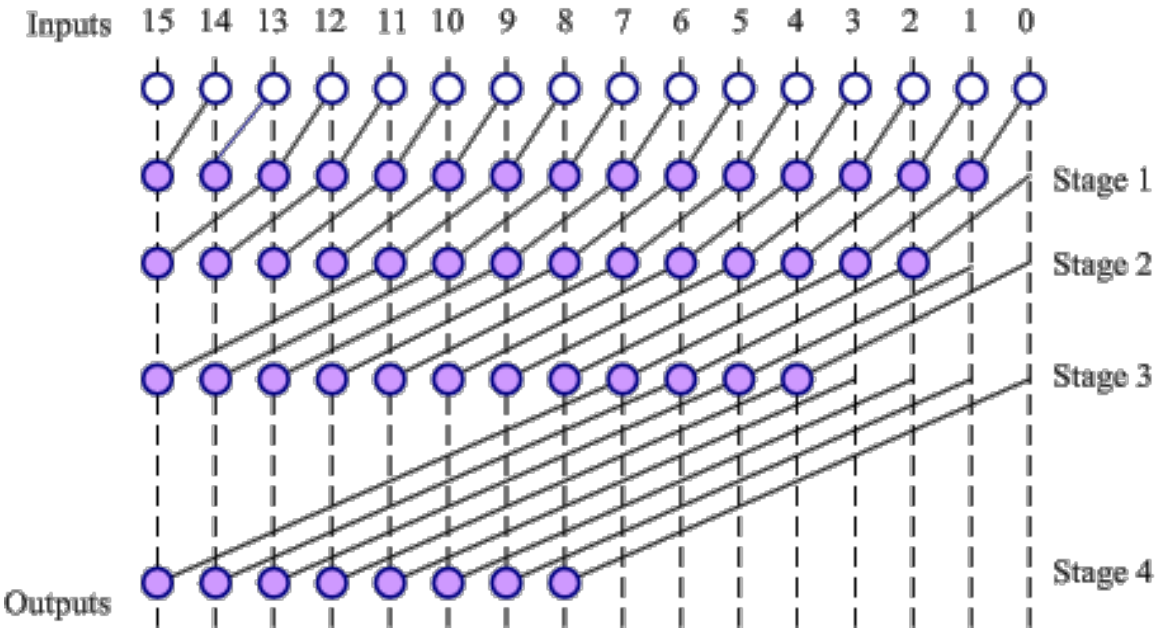

Figure 2: 16-bit Kogge-Stone prefix tree

The area of Kogge-Stone is also very high. For a technology independent comparison of area, we can use the prefix cell count as an estimate. However the actual area should include the pre-computation and post-computation logic circuits along with buffering. For *n*-bit Kogge Stone adder, total number of prefix operators can be calculated as $n\log_2 n - n + 1$. When n=16, the number of prefix operators is 49.

3.2 Knowles Family of adders

Knowles [11] proposed a family of adders with flexible architectures of prefix tree computation structures. As mentioned earlier, Knowles prefix adders are distinguished by the maximum branch fan-out at each logic level. Knowles [4,2,2,1] prefix structure for 16-bit adder is shown below. 16-bit Kogge-Stone adder is nothing but Knowles [1,1,1,1] prefix structure. In fact, there are 14 different topologies for 16-bit

adders such as [4,2,1,1], [4,4,2,1], [8,2,1,1] and [8,2,2,1] with Kogge-Stone [1,1,1,1] and Ladner-Fischer [8,4,2,1] as limiting cases in terms of branch fan-out in each logic stage. Knowles family of adders also includes Hybrid Knowles prefix adders, which allow different fan-out in the same logic level.

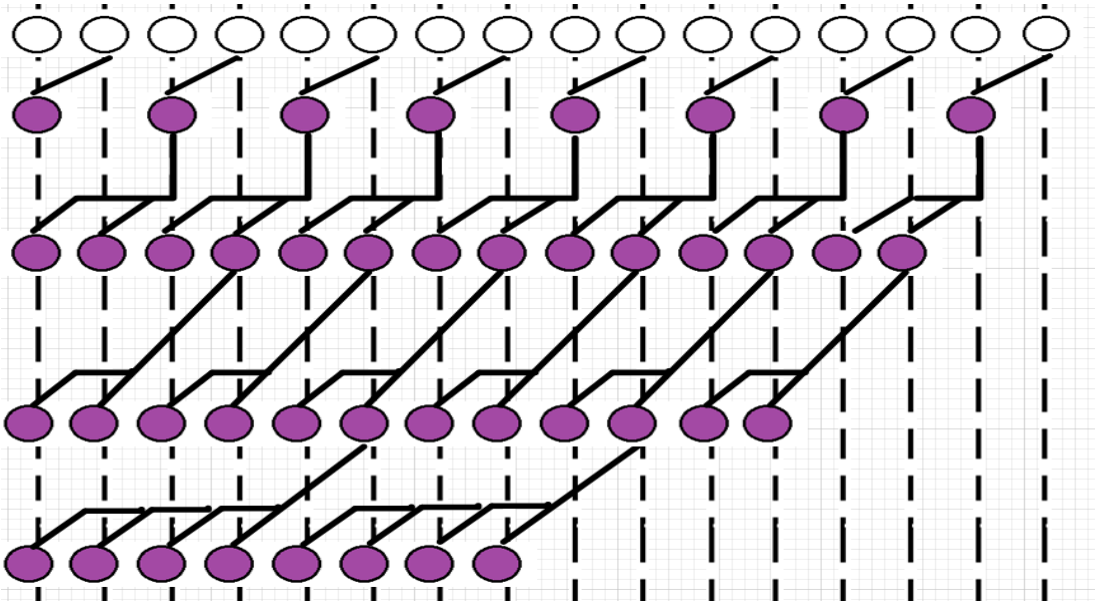

Figure 3: Knowles 16-bit prefix [4,2,2,1] structure

The Knowles prefix tree family can be built based on Kogge-Stone prefix trees and can be gradually moved on to more complex prefix structures like Brent-Kung or Han-Carlson tree. It allows us to limit the lateral wire tracks at each logic level and the branch fan-out at each logic level based on the technology requirements. Also it allows reuse of the blocks of smaller adders in the prefix tree and to combine the best of different prefix adder qualities based on the constraints.

The area of Knowles adders depends on the prefix tree implementation. Knowles

[2,1,1,1] contains the same number of logic levels as Kogge-Stone [1,1,1,1]. Kogge-Stone 8-bit adder can be reused for the first three logic levels while the logic level 4 is modified to have a fan-out of 3 instead of 2. Thus, the Knowles [2,1,1,1] has same number of prefix operator cells as Kogge-Stone 16-bit adder. However the lateral wire-tracks is reduced in the final logic level. The cell count is estimated as $nlog_2n - n + 1$, exactly same as Kogge-Stone 16-bit adder.

3.3 Brent-Kung adders

Brent-Kung prefix tree [12] is a well-known structure, which has the least fan-out and lateral wire tracks among the popular prefix adders. However it is a complex structure because it has the most logic levels. A 16-bit Brent-Kung prefix tree structure is shown below.

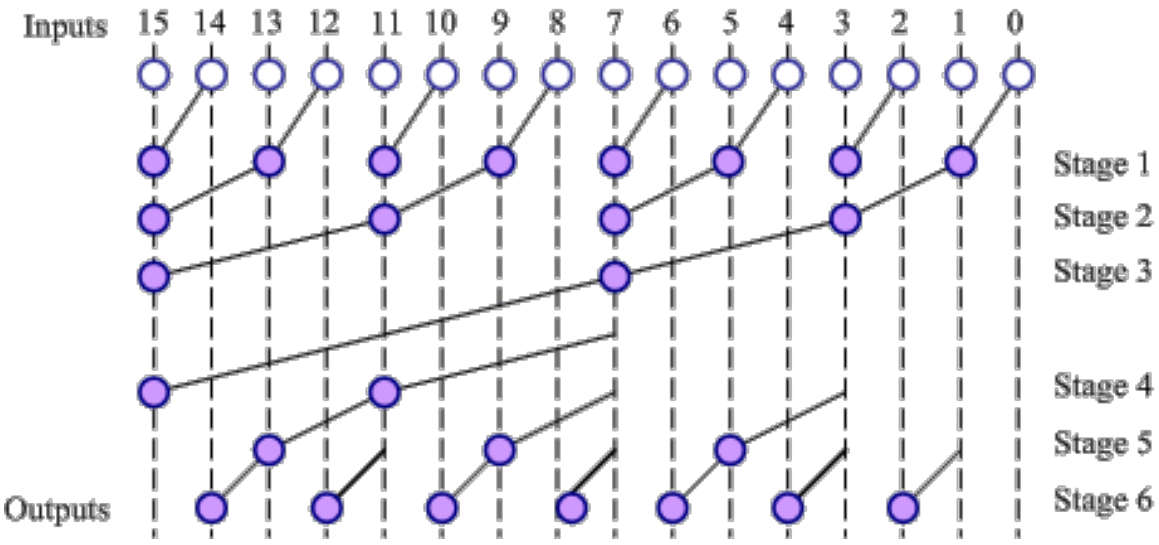

Figure 4: 16-bit Brent Kung prefix tree

Brent-Kung adder restricts the lateral fan-out of each node to unity, as in Kogge-Stone adder but without the explosion of wires. The capacitive load is still high due to the wide span of wires. For instance, in a 16-bit adder shown above, the structure starts with prefix operators every 2 bits. The input span is 1 bit and the output span is 2 bits. At logic level 2, the distance between each operator is 4 bits while it is 8 bits in logic level 3. At logic level 4, the only prefix operation is at the MSB with input span of 8 bits and the output span of 16 bits. At logic levels 5 through 7, the input bit spans are decremented and they are 4, 2 and 1 bit respectively. The critical path for a 16-bit adder is from bit 0 in the pre-computation stage to bit 14 in logic stage 6. Hence, even with buffering, the Brent-Kung adders are among the slowest prefix adders.

In terms of prefix operator count, for a *n*-bit adder, the total number of prefix operators is $2(n-1) - \log_2 n$. For n = 16, it is 26.

3.4 Ladner-Fischer adders

Ladner-Fischer prefix tree structure [13] is shown below. Sklansky's conditional-sum adder can be included in this family of prefix structure. It exploits the associativity property of prefix operators extensively, but not the idempotency property, while constructing a binary tree of prefix operators. It is in some sense a basis for the other prefix tree structures.

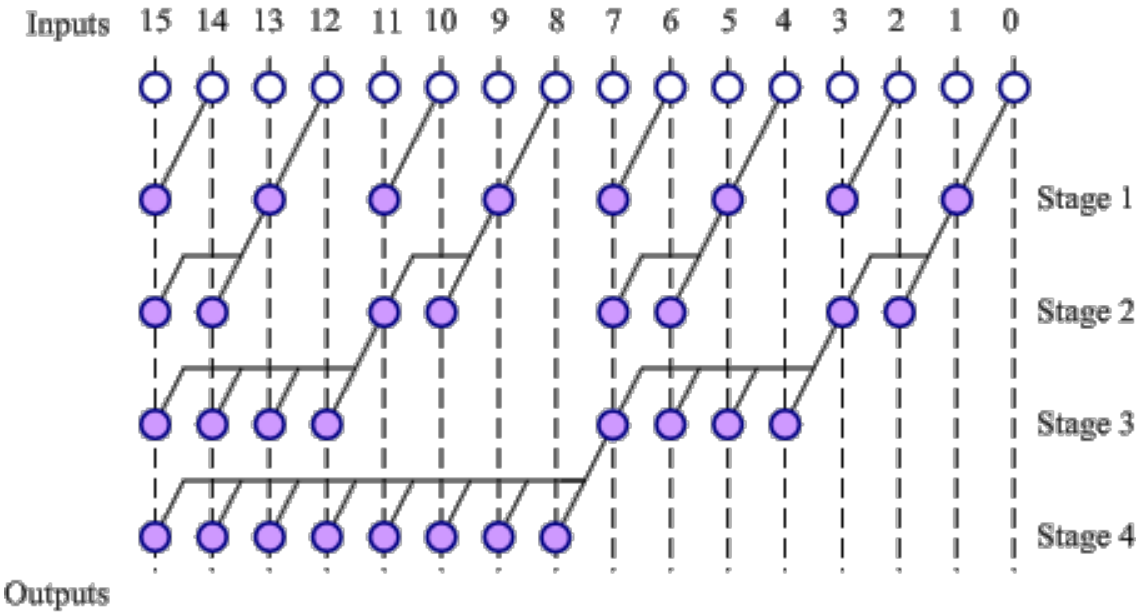

Figure 5: 16-bit Ladner-Fischer prefix tree

This structure has minimum logic depth but has large fan-out requirement up to *n/2*. The longest lateral fanning wires run from a node to *n/2* other nodes. Capacitive fan-out loads are very large for later levels in the graph for wider input operands. In VLSI CMOS implementations, it involves increasing the drive strengths of the buffers and inverters to support larger loads. This increases the area, limits the performance by increasing the delay and burns more power due to larger drive cells.

The number of logic levels of $log_2 n$ is always the minimum in this scheme for an *n*-bit adder. Each logic level has *n/2* cells. Therefore, the total number of prefix operators is $n log_2 n$

3.5 Han-Carlson adders

The idea of Han-Carlson prefix tree [14] is very similar to Kogge-Stone adder in terms of maximum fan-out of 2 at each logic stage. However Han-Carlson scheme uses fewer cells and lateral wire tracks than Kogge-Stone adder by adding one extra logic level. A 16-bit Han-Carlson prefix tree is shown below.

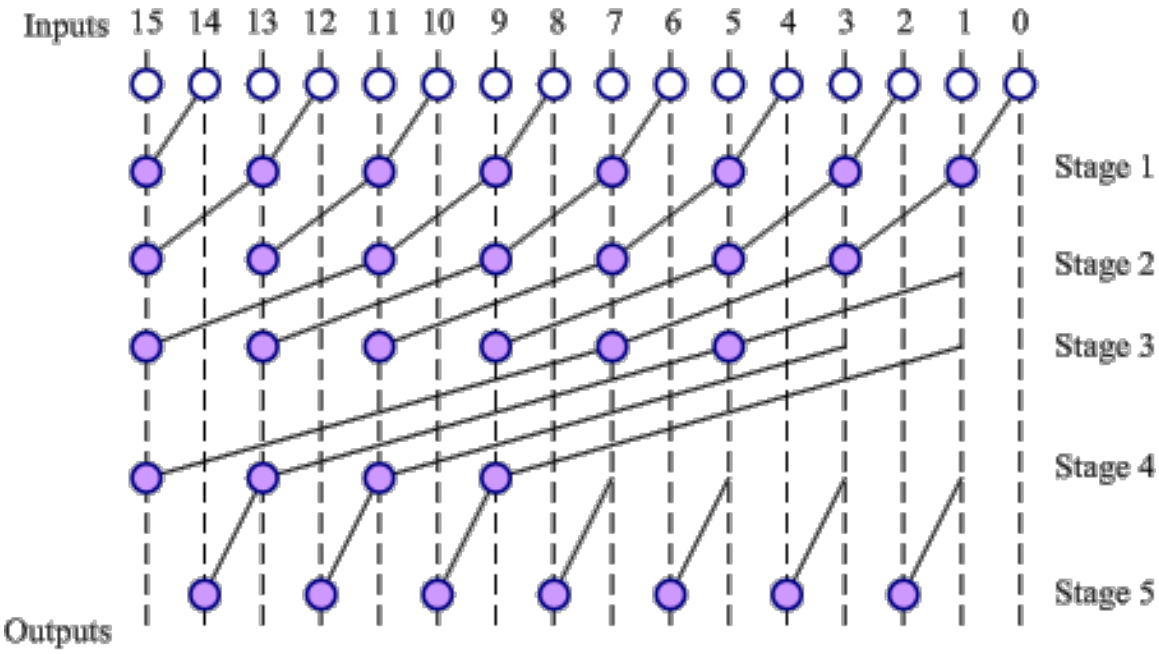

Figure 6: 16-bit Han-Carlson prefix tree

Han-Carlson prefix tree performs prefix operations every other bit in each logic level. The extra last logic stage performs prefix operations for the missing carries. This approach accomplishes minimum branch fan-out at each logic level without using more wiring track resources. The critical path is from bit 8 in the pre-computation stage to carry bit 10 in the final stage.

As mentioned above, the number of logic levels is log2n + 1 for any n-bit adder. It can be observed that the total number of prefix operators in this scheme is $(n/2)log_2n$. For n = 16, 32 prefix operators are used.

3.6 Summary

By tabulating the logic levels, fanout, prefix operator count and lateral wiring tracks for the above discussed adders, a clear idea of the trade-offs involved in the design is provided.

Table 1: Algorithmic comparison of prefix tree structure

Prefix structure	*Logic Levels*	*Prefix count*	*Fan-out*	*Wire Tracks*
Kogge-Stone	log_2n	$nlog_2n - n + 1$	2	$n/2$
Knowles[2,1,1,1]	log_2n	$nlog_2n - n + 1$	3	$n/4$
Brent-Kung	$2log_2n-1$	$2(n-1) - log_2n$	2	1
Ladner-Fischer	log_2n	$(n/2)log_2n$	$n/2+1$	1
Han-Carlson	log_2n+1	$(n/2)log_2n$	2	$n/4$

Thus in this chapter, the construction analysis of parallel prefix adders is done in detail. The trade-offs involved in designing a prefix adder scheme is tabulated for comparison.

CHAPTER 4

SIMULATION METHODOLOGY

To understand the impact of circuits and CMOS technology on the prefix tree structure, simulation data on power, performance and area from placed and routed circuits are needed. To obtain the simulation results for the report, the following methodology is followed.

The 8-bit, 16-bit and 32-bit versions of each of the above 5 parallel prefix adders were designed in structural Verilog model. The individual bit generate and propagate signals are generated using an AND gate and a XOR gate respectively. A prefix operator is designed using an AND gate and an AOI cell. The multiple carries are computed using different prefix tree structures with proper buffering necessary, according to the implementation. The post-computation stage is designed using XOR gates to generate the sum bits.

To understand the impact of technology, this report uses Synopsys 32/28nm Generic Library and FreePDK 45nm [15] design kit. The 32nm Digital Standard Cell library [16] consists of 350 cells for different drive strengths to optimize the IC design. The library is designed for 1.05V operation with a process technology of 1P9M. Since University of Texas at Austin is a member of Synopsys University Program, it is possible to perform System-on-Chip (SoC) implementations and statistical circuit analysis through commercial front-end and back-end tools without violating the intellectual property (IP). FreePDK is an open source, variation aware Process Design Kit (PDK), provided by North Carolina State University.

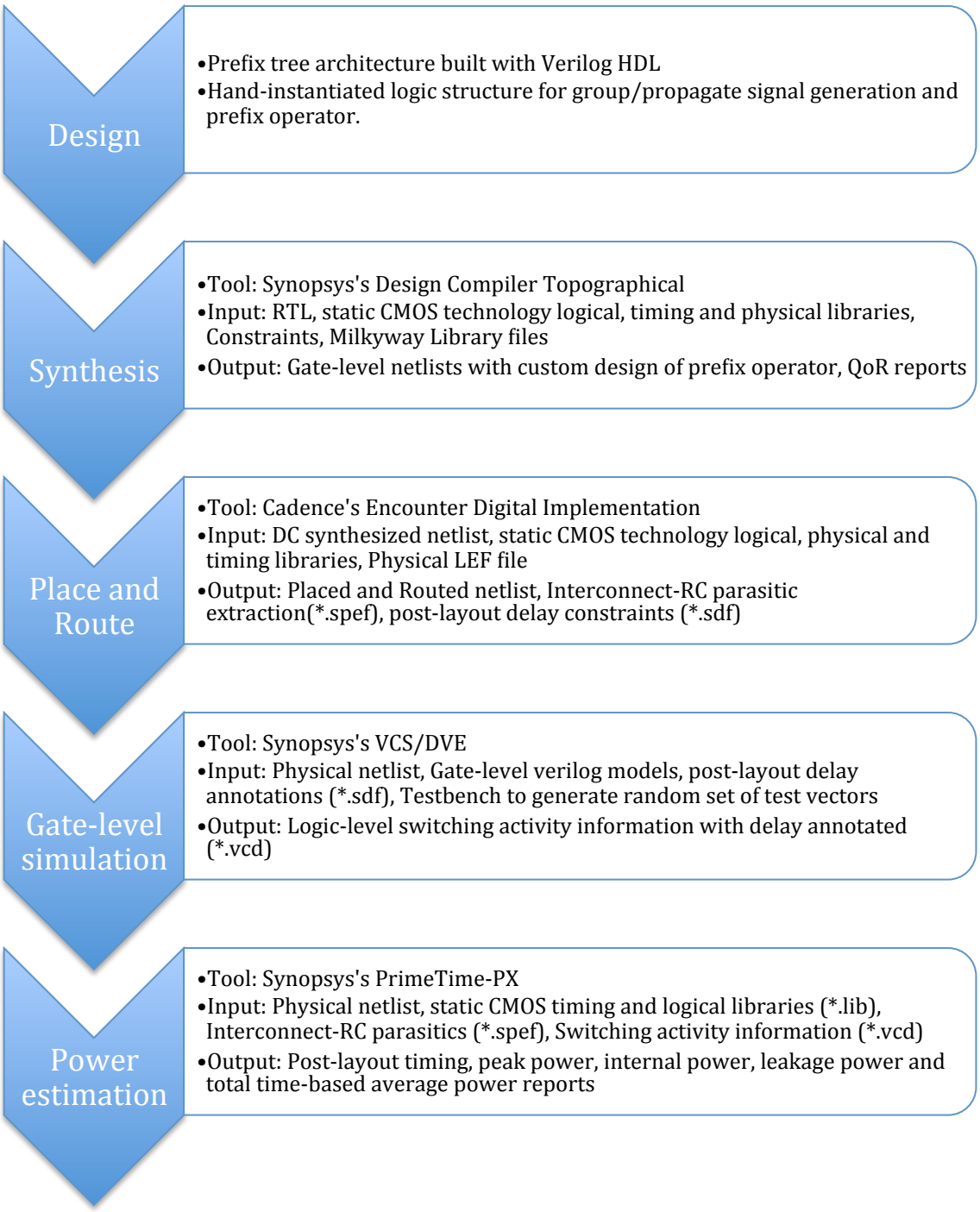

Figure 7: Simulation methodology

The high level description is then synthesized using Synopsys's SAED 32nm static CMOS technology using Design Compiler Topographical. It is a physical-aware synthesis where interconnect RC modeling is based on coarse placement performed by Design Compiler.

The resulting gate-level synthesized netlists are physically placed and routed using Cadence's Encounter Digital Implementation (EDI). The resulting layout is used as the basis for the circuit simulation. The interconnect RC parasitic extraction is done based on EDI's routing and exported in SPEF format.

Each physically placed and routed adder implementation is simulated with 1250 pseudo-random input vectors through a testbench, which annotates the cell timing delay and interconnect delay. The back-annotation of post-layout timing delay SDF and parasitic SPEF, is done to ensure accurate modeling of the switching behavior for the randomly generated inputs. The toggle count per net of each adder is calculated using logic simulation VCD. Synopsys's VCS/DVE circuit simulator is used to generate the switching activity information of each adder.

Finally, the total power dissipation is estimated from the detailed time-based average mode power simulation in Synopsys's PrimeTime-PX. The logic switching activity VCD information dumped by the previous gate simulation is used to estimate the dynamic switching power and internal power. To ensure better correlation with post-layout implementation, the extracted RC parasitic SPEF constraints are applied during power estimation. Leakage power is also included in the total power estimation.

The same methodology is followed using FreePDK 45nm technology to analyze the impact of circuit technology on architecture.

CHAPTER 5

CIRCUIT SIMULATION RESULTS

Based on the above-described methodology, the following parallel prefix adders were simulated: Brent-Kung, Kogge-Stone, Knowles family of adders, Han-Carlson and Ladner-Fischer in SAED 32nm technology and FreePDK 45nm technology.

5.1 SAED 32nm technology

The performance is typically limited by the worst-case delay of the critical path in an adder implementation. The critical path delay in worst-case corner operating at 1.05V for the parallel prefix adders is tabulated below. As can be seen, Kogge-Stone adder is the fastest while Brent-Kung adder is the slowest among the adders. The best performing Knowles adder structure in terms of power, performance and area is picked for each input vector width and presented in this table.

Table 2: Critical path delay of prefix adders in SAED 32nm technology

Adders	Critical path delay (ns)		
	8-bit	16-bit	32-bit
Brent-Kung	0.37	0.49	0.59
Kogge-Stone	0.30	0.38	0.45
Han-Carlson	0.34	0.42	0.48
Ladner-Fischer	0.32	0.42	0.51
Knowles Family 8-bit: [2,1,1] 16-bit: [4,4,2,1] 32-bit: [16,2,2,2,1]	0.31	0.39	0.48

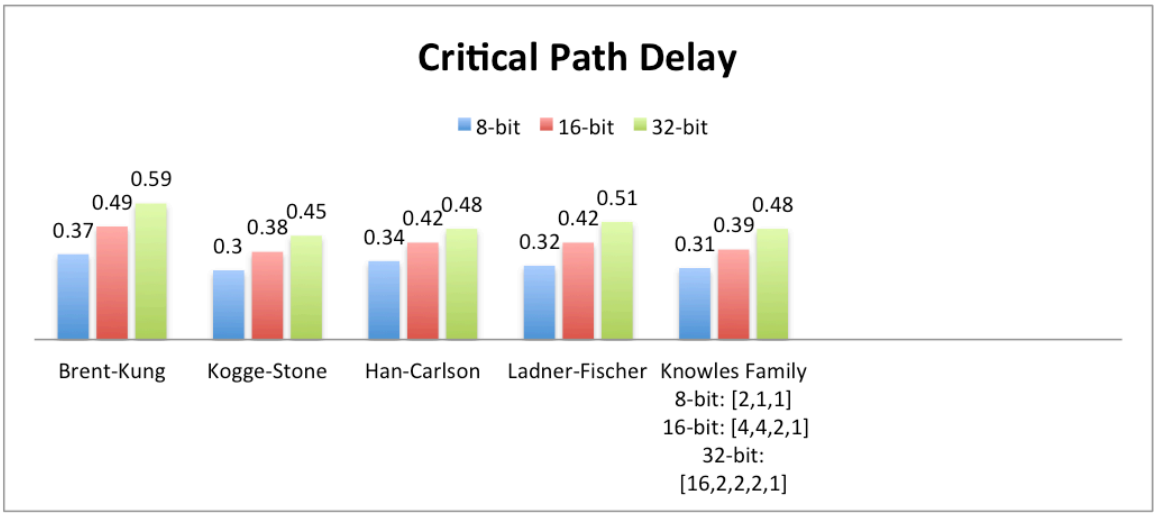

Figure 8: Critical path delay of prefix adders in 32nm technology

The area comparison is shown in the Table.

Table 3: Total cell area of prefix adders in SAED 32nm technology

Adders	Area		
	8-bit	16-bit	32-bit
Brent-Kung	127.83	277.27	584.79
Kogge-Stone	155.28	382.49	910.09
Han-Carlson	132.41	304.72	685.93
Ladner-Fischer	132.41	303.19	680.09
Knowles Family 8-bit: [2,1,1] 16-bit: [4,4,2,1] 32-bit: [16,2,2,2,1]	157.06	321.24	772.60

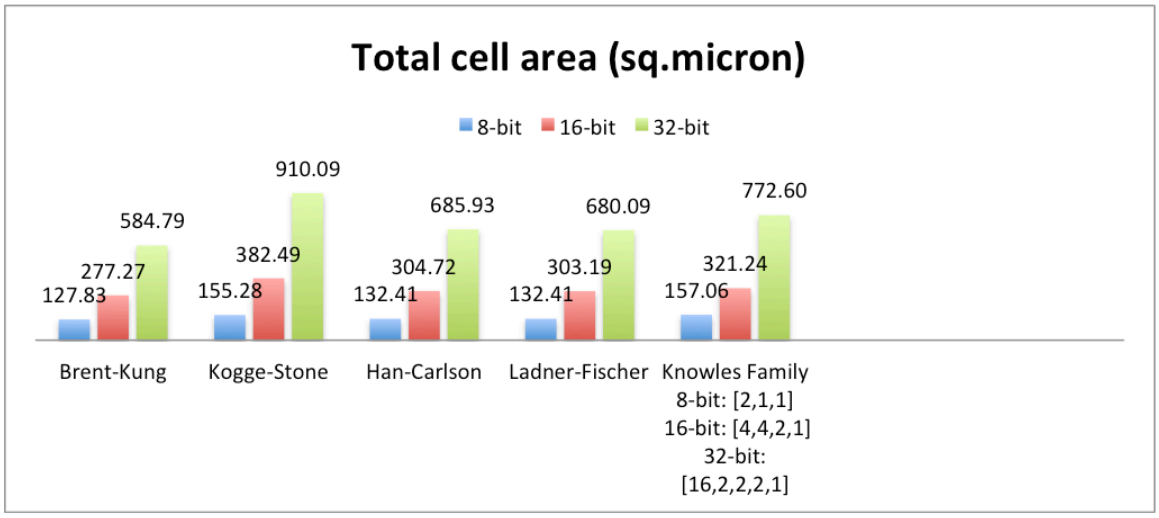

Figure 9: Total cell area of synthesized prefix adders in SAED 32nm technology

The total power dissipation of different adders is tabulated here. As can be seen, Kogge-Stone adders burn the highest amount of power as the input word width increases. It is due to extensive application of parallel prefix operators for high performance. Brent-Kung adders burn the least amount of power. Knowles adders achieve comparable performance to the Kogge-Stone adders without exploding in area and burning high power.

Table 4: Total power dissipation of prefix adders in SAED 32nm technology

Adders	Total power (W)		
	8-bit	16-bit	32-bit
Brent-Kung	6.12E-05	1.11E-04	2.31E-04
Kogge-Stone	5.91E-05	1.36E-04	3.20E-04
Han-Carlson	6.33E-05	1.44E-04	2.59E-04
Ladner-Fischer	6.32E-05	1.44E-04	2.67E-04
Knowles Family 8-bit: [2,1,1] 16-bit: [4,4,2,1] 32-bit: [16,2,2,2,1]	5.76E-05	1.25E-04	2.89E-04

The detailed comparison of Knowles family of adders is presented in Table below. The numbers in the bracket represent the maximum branch fan-out at each logic stage for the corresponding prefix tree structure. Power, performance and area of the implementation in 32nm static CMOS technology are shown here.

Table 5: Power, performance and area metrics of Knowles family of adders

Input width	Knowles structure	Critical path delay (ns)	Area (sq μ)	Internal Power (W)	Leakage Power (W)	Total Power (W)
	[2,1,1]	0.31	157.06	1.01E-06	5.66E-05	5.76E-05
	[2,2,1]	0.33	141.56	1.01E-06	5.43E-05	5.53E-05
8-bit	[4,1,1]	0.32	141.56	1.01E-06	5.43E-05	5.53E-05
	[2,1,1,1]	0.42	384.27	2.03E-06	1.36E-04	1.38E-04
	[2,2,1,1]	0.42	384.27	2.03E-06	1.36E-04	1.38E-04
	[2,2,2,1]	0.39	350.46	2.03E-06	1.29E-04	1.31E-04
	[4,1,1,1]	0.43	382.49	2.03E-06	1.37E-04	1.40E-04
	[4,2,1,1]	0.44	382.49	2.03E-06	1.37E-04	1.40E-04
	[4,2,2,1]	0.44	380.71	2.03E-06	1.39E-04	1.41E-04
	[4,4,1,1]	0.42	381.22	2.03E-06	1.40E-04	1.42E-04
	[4,4,2,1]	0.39	321.24	2.03E-06	1.23E-04	1.25E-04
	[8,1,1,1]	0.39	348.69	4.06E-06	1.75E-04	1.79E-04
	[8,2,1,1]	0.40	350.46	4.06E-06	1.75E-04	1.79E-04
	[8,2,2,1]	0.39	321.24	4.06E-06	1.68E-04	1.72E-04
16-bit	[8,4,1,1]	0.39	321.24	2.03E-06	1.23E-04	1.25E-04
	[16,2,2,2,1]	0.48	772.60	3.93E-06	2.85E-04	2.89E-04
	[16,4,2,2,1]	0.50	834.86	3.93E-06	3.04E-04	3.08E-04
	[2,2,2,1,1]	0.49	843.25	3.93E-06	2.96E-04	3.00E-04
32-bit	[4,4,2,2,1]	0.49	836.90	3.93E-06	3.03E-04	3.07E-04

The trade-off between circuit speed and area in 32nm CMOS technology is very tight. The area increases drastically to achieve superior speed targets and the chip fabrication cost is very high when area increases even by a few percent in modern CMOS

technology. As the table for Knowles adders shows, the speedup in adder network is achieved by introduction of parallel, logically redundant prefix operators. Similar trend is observed in prefix tree structure evolution from Ladner-Fischer adder to Kogge-Stone adder by exploiting idempotency property of prefix operators. However this results in the explosion of lateral wire track count and results in significant power and area cost.

5.2 FreePDK 45nm technology

The critical path delay in 45nm technology for the adders operating at 1.8V is presented here. Even though Kogge-Stone adder is the fastest adder scheme, the slowest adder implementation is not always Brent-Kung adder. For wider inputs, Ladner-Fischer adder becomes the slowest due to the high fan-out capacitive load seen by the later stages. The technology plays an important role in how the capacitive load and wiring delay affect the performance of adders in wider inputs. In general, performance speedup of 33-45% is achieved in scaling the technology from 45nm to 32nm CMOS technology.

Table 6: Critical path delay of prefix adders in FreePDK 45nm technology

Adders	Critical path delay		
	8-bit	16-bit	32-bit
Brent-Kung	0.54	0.77	0.94
Kogge-Stone	0.40	0.53	0.72
Han-Carlson	0.45	0.58	0.72
Ladner-Fischer	0.46	0.62	1.03
Knowles Family 8-bit: [2,1,1] 16-bit: [4,4,2,1] 32-bit: [16,2,2,2,1]	0.42	0.57	0.77

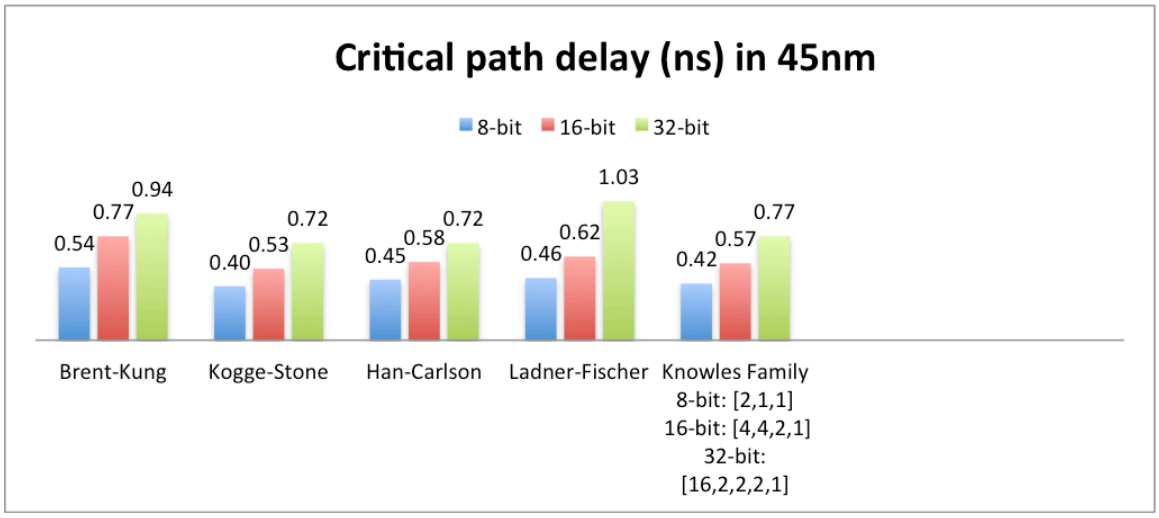

Figure 10: Critical path delay of prefix adders in 45nm technology

The area in 45nm technology is shown in Table 7. As can be seen, similar trend of Kogge-Stone being the largest adder structure especially for wider inputs is observed here. On average, the adder structures grow by 50-60% in area when compared to 32nm technology.

Table 7: Total cell area in FreePDK 45nm technology

Adders	Total cell area (sq.micron)		
	8-bit	16-bit	32-bit
Brent-Kung	187.25	414.86	884.63
Kogge-Stone	240.75	619.95	1521.00
Han-Carlson	196.17	468.36	1084.08
Ladner-Fischer	196.17	468.36	1084.08
Knowles Family 8-bit: [2,1,1] 16-bit: [4,4,2,1] 32-bit: [16,2,2,2,1]	229.31	481.86	1212.98

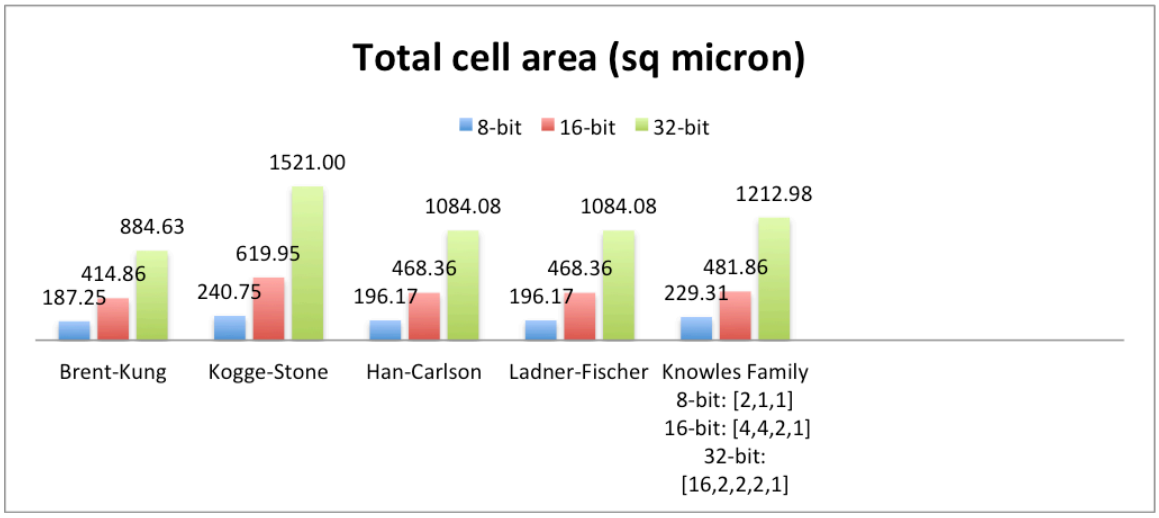

Figure 12: Total cell area of synthesized prefix adders in 45nm technology

The total power dissipation in 45nm technology is shown in Table 8. The 45nm FreePDK technology involved a thick oxide gate with 10 metal layers. The leakage power is the primary component in the total power dissipation. Also, the operating voltage is 1.8V at 45nm technology while it was 1.05V in 32nm technology. That also explains why the power dissipation is very high at 45nm technology when compared against 32nm technology.

Table 8: Total power dissipation of prefix adders in FreePDK 45nm technology

Adders	Total power (W)		
	8-bit	16-bit	32-bit
Brent-Kung	9.17E-05	1.66E-04	3.70E-04
Kogge-Stone	8.86E-05	2.17E-04	5.43E-04
Han-Carlson	9.49E-05	2.26E-04	4.27E-04
Ladner-Fischer	9.47E-05	2.26E-04	4.41E-04
Knowles Family 8-bit: [2,1,1] 16-bit: [4,4,2,1] 32-bit: [16,2,2,2,1]	8.64E-05	1.92E-04	4.71E-04

CHAPTER 6

CONCLUSION

In this Master's report, the comparative analysis of parallel prefix adders in terms of power, performance and area is done in SAED 32nm and FreePDK 45nm static CMOS technologies. It is clearly shown that Kogge-Stone adder structure is the fastest parallel prefix adder scheme at the cost of highest area and power dissipation. Knowles adders achieve comparable performance to Kogge-Stone adders by balancing the maximum branch fan-out across logic stages and limiting the lateral wire track count. Interconnect delay is the dominant component than gate delays in modern technology. Also, the routing resources are very expensive in modern technology.

Brent-Kung adder is the smallest and least power hungry parallel prefix adder. However because of its increased logic levels for wider inputs, it is also the slowest adder. Hence it is not practical in CMOS implementations with very high performance targets. Ladner-Fischer adder has a very high fan-out capacitive load in the critical path, especially for wider inputs. Hence it is not suitable for high speed applications for longer words without proper buffering.

Thus, there is no perfect adder for a particular technology due to the trade-off involved in speed, area and wire tracks. In order to be power efficient, operating voltages and multi-threshold voltage based cells can be used for power savings in adder structure.

APPENDIX

Generate/Propagate logic

```
module BITPG ( Gi, Pi, Ai, Bi );
output  Gi, Pi;

input  Ai, Bi;
AND2X1_RVT U_gen_and (.A1(Ai),.A2(Bi),.Y(Gi));
XOR2X1_RVT U_prop_xor (.A1(Ai),.A2(Bi),.Y(Pi));
//assign Gi = Ai & Bi;
//assign Pi = Ai ^ Bi;

endmodule

module GROUPGP ( G2, P2, G0, G1, P0, P1 );
output  G2, P2;

input  G0, G1, P0, P1;
AO21X1_RVT U_prefix_gen (.A1(G0),.A2(P1),.A3(G1),.Y(G2));
AND2X1_RVT U_prefix_prop (.A1(P1),.A2(P0),.Y(P2));
//assign G2 = G1 | (G0 & P1);
//assign P2 = P1 & P0;

endmodule
```

Kogge-Stone 16-bit adder

```
module kogge_stone16 ( A, B, Sum, Cout);
parameter WIDTH = 16;
input [WIDTH:1] A, B;
output [WIDTH:1] Sum;
output Cout;

wire [WIDTH:1] g, p;
wire [WIDTH:1] inter_g [4:1];
wire [WIDTH:1] inter_p [4:1];

genvar j,k,l,m,n;
generate
for (j=1; j<= WIDTH; j = j+1)
begin : bit_pg
    BITPG pg (.Gi(g[j]), .Pi(p[j]), .Ai(A[j]), .Bi(B[j]) );
end
endgenerate

assign inter_g[1][1] = g[1];
assign inter_p[1][1] = p[1];
```

```
generate
for (k=1; k<=(WIDTH-1); k = k+1)
begin : group_prefix_stage_1
GROUPGP prefix (.G2(inter_g[1][k+1]), .P2(inter_p[1][k+1]),
.G0(g[k]), .G1(g[k+1]), .P0(p[k]), .P1(p[k+1]));
end
endgenerate

assign inter_g[2][1] = g[1];
assign inter_p[2][1] = p[1];

assign inter_g[2][2] = inter_g[1][2];
assign inter_p[2][2] = inter_p[1][2];

generate
for (l=1; l<=(WIDTH-2); l = l+1)
begin : group_prefix_stage_2
GROUPGP prefix (.G2(inter_g[2][l+2]), .P2(inter_p[2][l+2]),
.G0(inter_g[1][l]), .G1(inter_g[1][l+2]), .P0(inter_p[1][l]),
.P1(inter_p[1][l+2]));
end
endgenerate

assign inter_g[3][1] = g[1];
assign inter_p[3][1] = p[1];

assign inter_g[3][2] = inter_g[1][2];
assign inter_p[3][2] = inter_p[1][2];

assign inter_g[3][3] = inter_g[2][3];
assign inter_p[3][3] = inter_p[2][3];

assign inter_g[3][4] = inter_g[2][4];
assign inter_p[3][4] = inter_p[2][4];

generate
for (m=1; m<=(WIDTH-4); m = m+1)
begin : group_prefix_stage_3
GROUPGP prefix (.G2(inter_g[3][m+4]), .P2(inter_p[3][m+4]),
.G0(inter_g[2][m]), .G1(inter_g[2][m+4]), .P0(inter_p[2][m]),
.P1(inter_p[2][m+4]));
end
endgenerate

assign inter_g[4][1] = g[1];
assign inter_p[4][1] = p[1];

assign inter_g[4][2] = inter_g[1][2];
assign inter_p[4][2] = inter_p[1][2];
```

```
assign inter_g[4][3] = inter_g[2][3];
assign inter_p[4][3] = inter_p[2][3];

assign inter_g[4][4] = inter_g[2][4];
assign inter_p[4][4] = inter_p[2][4];

assign inter_g[4][5] = inter_g[3][5];
assign inter_p[4][5] = inter_p[3][5];
assign inter_g[4][6] = inter_g[3][6];
assign inter_p[4][6] = inter_p[3][6];
assign inter_g[4][7] = inter_g[3][7];
assign inter_p[4][7] = inter_p[3][7];
assign inter_g[4][8] = inter_g[3][8];
assign inter_p[4][8] = inter_p[3][8];

generate
for (n=1; n<=(WIDTH-8); n = n+1)
begin : group_prefix_stage_4
GROUPGP prefix (.G2(inter_g[4][n+8]), .P2(inter_p[4][n+8]),
.G0(inter_g[3][n]), .G1(inter_g[3][n+8]), .P0(inter_p[3][n]),
.P1(inter_p[3][n+8]));
end
endgenerate

//post-processing
assign Sum[1] = p[1];
assign Sum[2] = p[2] ^ g[1];
assign Sum[3] = p[3] ^ inter_g[1][2];
assign Sum[4] = p[4] ^ inter_g[2][3];
assign Sum[5] = p[5] ^ inter_g[2][4];
assign Sum[6] = p[6] ^ inter_g[3][5];
assign Sum[7] = p[7] ^ inter_g[3][6];
assign Sum[8] = p[8] ^ inter_g[3][7];
assign Sum[9] = p[9] ^ inter_g[3][8];
assign Sum[10] = p[10] ^ inter_g[4][9];
assign Sum[11] = p[11] ^ inter_g[4][10];
assign Sum[12] = p[12] ^ inter_g[4][11];
assign Sum[13] = p[13] ^ inter_g[4][12];
assign Sum[14] = p[14] ^ inter_g[4][13];
assign Sum[15] = p[15] ^ inter_g[4][14];
assign Sum[16] = p[16] ^ inter_g[4][15];

assign Cout = inter_g[4][16];

endmodule
```

Knowles [4,4,2,1] 16-bit adder

```
module knowles_adder16_4421 ( A, B, Sum, Cout);
```

```
parameter WIDTH = 16;
input [WIDTH:1] A, B;
output [WIDTH:1] Sum;
output Cout;

wire [WIDTH:1] g, p;
wire [WIDTH:1] inter_g [4:1];
wire [WIDTH:1] inter_p [4:1];

genvar j,k,l,m,n;
generate
for (j=1; j<= WIDTH; j = j+1)
begin : bit_pg
    BITPG pg (.Gi(g[j]), .Pi(p[j]), .Ai(A[j]), .Bi(B[j]) );
end
endgenerate

assign inter_g[1][1] = g[1];
assign inter_p[1][1] = p[1];
assign inter_g[1][3] = g[3];
assign inter_p[1][3] = p[3];
assign inter_g[1][5] = g[5];
assign inter_p[1][5] = p[5];
assign inter_g[1][7] = g[7];
assign inter_p[1][7] = p[7];
assign inter_g[1][9] = g[9];
assign inter_p[1][9] = p[9];
assign inter_g[1][11] = g[11];
assign inter_p[1][11] = p[11];
assign inter_g[1][13] = g[13];
assign inter_p[1][13] = p[13];
assign inter_g[1][15] = g[15];
assign inter_p[1][15] = p[15];

generate
for (k=1; k<=(WIDTH-1); k = k+2)
begin : group_prefix_stage_1
GROUPGP prefix (.G2(inter_g[1][k+1]), .P2(inter_p[1][k+1]),
.G0(g[k]), .G1(g[k+1]), .P0(p[k]), .P1(p[k+1]));
end
endgenerate

assign inter_g[2][1] = g[1];
assign inter_p[2][1] = p[1];

assign inter_g[2][2] = inter_g[1][2];
assign inter_p[2][2] = inter_p[1][2];
assign inter_g[2][5] = inter_g[1][5];
assign inter_p[2][5] = inter_p[1][5];
assign inter_g[2][6] = inter_g[1][6];
assign inter_p[2][6] = inter_p[1][6];
```

```
assign inter_g[2][9] = inter_g[1][9];
assign inter_p[2][9] = inter_p[1][9];
assign inter_g[2][10] = inter_g[1][10];
assign inter_p[2][10] = inter_p[1][10];
assign inter_g[2][13] = inter_g[1][13];
assign inter_p[2][13] = inter_p[1][13];
assign inter_g[2][14] = inter_g[1][14];
assign inter_p[2][14] = inter_p[1][14];

generate
for (l=2; l<=(WIDTH-2); l = l+4)
begin : group_prefix_stage_2
GROUPGP prefix1 (.G2(inter_g[2][l+1]), .P2(inter_p[2][l+1]),
.G0(inter_g[1][l]), .G1(inter_g[1][l+1]), .P0(inter_p[1][l]),
.P1(inter_p[1][l+1]));
GROUPGP prefix2 (.G2(inter_g[2][l+2]), .P2(inter_p[2][l+2]),
.G0(inter_g[1][l]), .G1(inter_g[1][l+2]), .P0(inter_p[1][l]),
.P1(inter_p[1][l+2]));
end
endgenerate

assign inter_g[3][1] = g[1];
assign inter_p[3][1] = p[1];

assign inter_g[3][2] = inter_g[1][2];
assign inter_p[3][2] = inter_p[1][2];

assign inter_g[3][3] = inter_g[2][3];
assign inter_p[3][3] = inter_p[2][3];

assign inter_g[3][4] = inter_g[2][4];
assign inter_p[3][4] = inter_p[2][4];

generate
for (m=4; m<=(WIDTH-4); m = m+4)
begin : group_prefix_stage_3
GROUPGP prefix1 (.G2(inter_g[3][m+1]), .P2(inter_p[3][m+1]),
.G0(inter_g[2][m]), .G1(inter_g[2][m+1]), .P0(inter_p[2][m]),
.P1(inter_p[2][m+1]));
GROUPGP prefix2 (.G2(inter_g[3][m+2]), .P2(inter_p[3][m+2]),
.G0(inter_g[2][m]), .G1(inter_g[2][m+2]), .P0(inter_p[2][m]),
.P1(inter_p[2][m+2]));
GROUPGP prefix3 (.G2(inter_g[3][m+3]), .P2(inter_p[3][m+3]),
.G0(inter_g[2][m]), .G1(inter_g[2][m+3]), .P0(inter_p[2][m]),
.P1(inter_p[2][m+3]));
GROUPGP prefix4 (.G2(inter_g[3][m+4]), .P2(inter_p[3][m+4]),
.G0(inter_g[2][m]), .G1(inter_g[2][m+4]), .P0(inter_p[2][m]),
.P1(inter_p[2][m+4]));
end
endgenerate
```

```
assign inter_g[4][1] = g[1];
assign inter_p[4][1] = p[1];

assign inter_g[4][2] = inter_g[1][2];
assign inter_p[4][2] = inter_p[1][2];

assign inter_g[4][3] = inter_g[2][3];
assign inter_p[4][3] = inter_p[2][3];

assign inter_g[4][4] = inter_g[2][4];
assign inter_p[4][4] = inter_p[2][4];

assign inter_g[4][5] = inter_g[3][5];
assign inter_p[4][5] = inter_p[3][5];
assign inter_g[4][6] = inter_g[3][6];
assign inter_p[4][6] = inter_p[3][6];
assign inter_g[4][7] = inter_g[3][7];
assign inter_p[4][7] = inter_p[3][7];
assign inter_g[4][8] = inter_g[3][8];
assign inter_p[4][8] = inter_p[3][8];

generate
for (n=4; n<=(WIDTH-8); n = n+4)
begin : group_prefix_stage_4
GROUPGP prefix1 (.G2(inter_g[4][n+5]), .P2(inter_p[4][n+5]),
.G0(inter_g[3][n]), .G1(inter_g[3][n+5]), .P0(inter_p[3][n]),
.P1(inter_p[3][n+5]));
GROUPGP prefix2 (.G2(inter_g[4][n+6]), .P2(inter_p[4][n+6]),
.G0(inter_g[3][n]), .G1(inter_g[3][n+6]), .P0(inter_p[3][n]),
.P1(inter_p[3][n+6]));
GROUPGP prefix3 (.G2(inter_g[4][n+7]), .P2(inter_p[4][n+7]),
.G0(inter_g[3][n]), .G1(inter_g[3][n+7]), .P0(inter_p[3][n]),
.P1(inter_p[3][n+7]));
GROUPGP prefix4 (.G2(inter_g[4][n+8]), .P2(inter_p[4][n+8]),
.G0(inter_g[3][n]), .G1(inter_g[3][n+8]), .P0(inter_p[3][n]),
.P1(inter_p[3][n+8]));
end
endgenerate

//post-processing
assign Sum[1] = p[1];
assign Sum[2] = p[2] ^ g[1];
assign Sum[3] = p[3] ^ inter_g[1][2];
assign Sum[4] = p[4] ^ inter_g[2][3];
assign Sum[5] = p[5] ^ inter_g[2][4];
assign Sum[6] = p[6] ^ inter_g[3][5];
assign Sum[7] = p[7] ^ inter_g[3][6];
assign Sum[8] = p[8] ^ inter_g[3][7];
assign Sum[9] = p[9] ^ inter_g[3][8];
assign Sum[10] = p[10] ^ inter_g[4][9];
assign Sum[11] = p[11] ^ inter_g[4][10];
```

```
assign Sum[12] = p[12] ^ inter_g[4][11];
assign Sum[13] = p[13] ^ inter_g[4][12];
assign Sum[14] = p[14] ^ inter_g[4][13];
assign Sum[15] = p[15] ^ inter_g[4][14];
assign Sum[16] = p[16] ^ inter_g[4][15];

assign Cout = inter_g[4][16];

endmodule
```

Brent-Kung 16-bit adder

```
module brent_kung16 ( A, B, Sum, Cout);
parameter WIDTH = 16;

input [WIDTH:1] A, B;
output [WIDTH:1] Sum;
output Cout;

wire [WIDTH:1] g, p;
wire [WIDTH:1] inter_g [6:1];
wire [WIDTH:1] inter_p [6:1];

genvar j,k,l,m1,m2,n;
generate
for (j=1; j<= WIDTH; j = j+1)
begin : bit_pg
    BITPG pg (.Gi(g[j]), .Pi(p[j]), .Ai(A[j]), .Bi(B[j]) );
end
endgenerate

generate
for (k=1; k<=(WIDTH-1); k = k+2)
begin : group_prefix_stage_1
assign inter_g[1][k] = g[k];
assign inter_p[1][k] = p[k];
GROUPGP prefix (.G2(inter_g[1][k+1]), .P2(inter_p[1][k+1]),
.G0(g[k]), .G1(g[k+1]), .P0(p[k]), .P1(p[k+1]));
end
endgenerate

generate
for (l=1; l<=(WIDTH-1); l = l+4)
begin : group_prefix_stage_2
assign inter_g[2][l] = inter_g[1][l];
assign inter_p[2][l] = inter_p[1][l];
assign inter_g[2][l+1] = inter_g[1][l+1];
assign inter_p[2][l+1] = inter_p[1][l+1];
assign inter_g[2][l+2] = inter_g[1][l+2];
assign inter_p[2][l+2] = inter_p[1][l+2];
```

```
GROUPGP prefix (.G2(inter_g[2][l+3]), .P2(inter_p[2][l+3]),
.G0(inter_g[1][l+1]), .G1(inter_g[1][l+3]), .P0(inter_p[1][l+1]),
.P1(inter_p[1][l+3]));
end
endgenerate

GROUPGP prefix_stage3_8 (.G2(inter_g[3][8]), .P2(inter_p[3][8]),
.G0(inter_g[2][4]), .G1(inter_g[2][8]), .P0(inter_p[2][4]),
.P1(inter_p[2][8]));
GROUPGP prefix_stage3_16 (.G2(inter_g[3][16]), .P2(inter_p[3][16]),
.G0(inter_g[2][12]), .G1(inter_g[2][16]), .P0(inter_p[2][12]),
.P1(inter_p[2][16]));
generate
for (m1=1; m1<= 7; m1 = m1 + 1)
begin : group_prefix_stage_3_first_half
assign inter_g[3][m1] = inter_g[2][m1];
assign inter_p[3][m1] = inter_p[2][m1];
end
endgenerate

generate
for (m2=9; m2<= 15; m2 = m2 + 1)
begin : group_prefix_stage_3_second_half
assign inter_g[3][m2] = inter_g[2][m2];
assign inter_p[3][m2] = inter_p[2][m2];
end
endgenerate

assign inter_g[4][1] = inter_g[3][1];
assign inter_p[4][1] = inter_p[3][1];
assign inter_g[4][2] = inter_g[3][2];
assign inter_p[4][2] = inter_p[3][2];
assign inter_g[4][3] = inter_g[3][3];
assign inter_p[4][3] = inter_p[3][3];
assign inter_g[4][4] = inter_g[3][4];
assign inter_p[4][4] = inter_p[3][4];
assign inter_g[4][5] = inter_g[3][5];
assign inter_p[4][5] = inter_p[3][5];
assign inter_g[4][6] = inter_g[3][6];
assign inter_p[4][6] = inter_p[3][6];
assign inter_g[4][7] = inter_g[3][7];
assign inter_p[4][7] = inter_p[3][7];
assign inter_g[4][8] = inter_g[3][8];
assign inter_p[4][8] = inter_p[3][8];

assign inter_g[4][9] = inter_g[3][9];
assign inter_p[4][9] = inter_p[3][9];

assign inter_g[4][10] = inter_g[3][10];
assign inter_p[4][10] = inter_p[3][10];
```

```
assign inter_g[4][11] = inter_g[3][11];
assign inter_p[4][11] = inter_p[3][11];

GROUPGP prefix_stage4_12 (.G2(inter_g[4][12]), .P2(inter_p[4][12]),
.G0(inter_g[3][8]), .G1(inter_g[3][12]), .P0(inter_p[3][8]),
.P1(inter_p[3][12]));

assign inter_g[4][13] = inter_g[3][13];
assign inter_p[4][13] = inter_p[3][13];

assign inter_g[4][14] = inter_g[3][14];
assign inter_p[4][14] = inter_p[3][14];

assign inter_g[4][15] = inter_g[3][15];
assign inter_p[4][15] = inter_p[3][15];

GROUPGP prefix_stage4_16 (.G2(inter_g[4][16]), .P2(inter_p[4][16]),
.G0(inter_g[3][8]), .G1(inter_g[3][16]), .P0(inter_p[3][8]),
.P1(inter_p[3][16]));

assign inter_g[5][1] = inter_g[4][1];
assign inter_p[5][1] = inter_p[4][1];
assign inter_g[5][2] = inter_g[4][2];
assign inter_p[5][2] = inter_p[4][2];
assign inter_g[5][3] = inter_g[4][3];
assign inter_p[5][3] = inter_p[4][3];
assign inter_g[5][4] = inter_g[4][4];
assign inter_p[5][4] = inter_p[4][4];
assign inter_g[5][5] = inter_g[4][5];
assign inter_p[5][5] = inter_p[4][5];

GROUPGP prefix_stage_5_6 (.G2(inter_g[5][6]), .P2(inter_p[5][6]),
.G0(inter_g[4][4]), .G1(inter_g[4][6]), .P0(inter_p[4][4]),
.P1(inter_p[4][6]));
assign inter_g[5][7] = inter_g[4][7];
assign inter_p[5][7] = inter_p[4][7];
assign inter_g[5][8] = inter_g[4][8];
assign inter_p[5][8] = inter_p[4][8];
assign inter_g[5][9] = inter_g[4][9];
assign inter_p[5][9] = inter_p[4][9];

GROUPGP prefix_stage_5_10 (.G2(inter_g[5][10]), .P2(inter_p[5][10]),
.G0(inter_g[4][8]), .G1(inter_g[4][10]), .P0(inter_p[4][8]),
.P1(inter_p[4][10]));
assign inter_g[5][11] = inter_g[4][11];
assign inter_p[5][11] = inter_p[4][11];
assign inter_g[5][12] = inter_g[4][12];
assign inter_p[5][12] = inter_p[4][12];
assign inter_g[5][13] = inter_g[4][13];
assign inter_p[5][13] = inter_p[4][13];
```

```
GROUPGP prefix_stage_5_14 (.G2(inter_g[5][14]), .P2(inter_p[5][14]),
.G0(inter_g[4][12]), .G1(inter_g[4][14]), .P0(inter_p[4][12]),
.P1(inter_p[4][14]));
assign inter_g[5][15] = inter_g[4][15];
assign inter_p[5][15] = inter_p[4][15];
assign inter_g[5][16] = inter_g[4][16];
assign inter_p[5][16] = inter_p[4][16];

assign inter_g[6][1] = inter_g[5][1];
assign inter_p[6][1] = inter_p[5][1];
assign inter_g[6][16] = inter_g[5][16];
assign inter_p[6][16] = inter_p[5][16];
generate
for (n=2; n<=(WIDTH-1);n=n+2)
begin: group_prefix_stage_6
assign inter_g[6][n] = inter_g[5][n];
assign inter_p[6][n] = inter_p[5][n];
GROUPGP prefix (.G2(inter_g[6][n+1]), .P2(inter_p[6][n+1]),
.G0(inter_g[5][n]), .G1(inter_g[5][n+1]), .P0(inter_p[5][n]),
.P1(inter_p[5][n+1]));
end
endgenerate

//post-processing
assign Sum[1] = p[1];
assign Sum[2] = p[2] ^ g[1];
assign Sum[3] = p[3] ^ inter_g[6][2];
assign Sum[4] = p[4] ^ inter_g[6][3];
assign Sum[5] = p[5] ^ inter_g[6][4];
assign Sum[6] = p[6] ^ inter_g[6][5];
assign Sum[7] = p[7] ^ inter_g[6][6];
assign Sum[8] = p[8] ^ inter_g[6][7];
assign Sum[9] = p[9] ^ inter_g[6][8];
assign Sum[10] = p[10] ^ inter_g[6][9];
assign Sum[11] = p[11] ^ inter_g[6][10];
assign Sum[12] = p[12] ^ inter_g[6][11];
assign Sum[13] = p[13] ^ inter_g[6][12];
assign Sum[14] = p[14] ^ inter_g[6][13];
assign Sum[15] = p[15] ^ inter_g[6][14];
assign Sum[16] = p[16] ^ inter_g[6][15];
assign Cout = inter_g[6][16];

endmodule
```

Ladner-Fischer 16-bit adder

```
module ladner_fischer16(A,B,C,cout);
input [16:1] A;
input [16:1] B;
output [16:1] C;
```

```
output  cout;

wire [16:1] Gi,Pi;
wire [16:1]  St1g,St1p;
wire [16:1]  St2g,St2p;
wire [16:1]  St3g,St3p;
wire [16:1]  St4g,St4p;

// Prefix Computation
genvar i;
generate
for ( i = 1; i <= 16; i = i+1 )
begin : bit_gen
BITPG bitpg (.Gi(Gi[i]), .Pi(Pi[i]), .Ai( A[i]),.Bi(B[i]) );
end
endgenerate
// Stage 1
genvar j;
generate
for ( j = 1; j < 16; j = j+2 )
begin
GROUPGP groupgp1 (.G0(Gi[j]), .P0(Pi[j]), .G1(Gi[j+1]),
.P1(Pi[j+1]), .G2(St1g[j+1]), .P2(St1p[j+1]) );
end
endgenerate
// Stage 2
genvar k;
generate
for ( k = 1; k < 16; k = k+4 )
begin
GROUPGP groupgp1 ( .G0(St1g[k+1]), .P0(St1p[k+1]), .G1(Gi[k+2]),
.P1(Pi[k+2]),     .G2(St2g[k+2]), .P2(St2p[k+2]) );
GROUPGP groupgp2 ( .G0(St1g[k+1]), .P0(St1p[k+1]), .G1(St1g[k+3]),
.P1(St1p[k+3]), .G2(St2g[k+3]), .P2(St2p[k+3]) );
end
endgenerate

// Stage 3
genvar l;
generate
for ( l = 1; l < 16; l = l+8 )
begin
GROUPGP groupgp1 ( .G0(St2g[l+3]), .P0(St2p[l+3]), .G1(Gi[l+4]),
.P1(Pi[l+4]),   .G2(St3g[l+4]), .P2(St3p[l+4]));
GROUPGP groupgp2 ( .G0(St2g[l+3]), .P0(St2p[l+3]), .G1(St1g[l+5]),
.P1(St1p[l+5]), .G2(St3g[l+5]), .P2(St3p[l+5]));
GROUPGP groupgp3 ( .G0(St2g[l+3]), .P0(St2p[l+3]), .G1(St2g[l+6]),
.P1(St2p[l+6]), .G2(St3g[l+6]), .P2(St3p[l+6]));
GROUPGP groupgp4 ( .G0(St2g[l+3]), .P0(St2p[l+3]), .G1(St2g[l+7]),
.P1(St2p[l+7]), .G2(St3g[l+7]), .P2(St3p[l+7]));
end
```

```
endgenerate



genvar m;
generate
for ( m = 1; m < 16; m = m+16 )
begin
GROUPGP groupgp1 ( .G0(St3g[m+7]), .P0(St3p[m+7]), .G1(Gi[m+8]),
.P1(Pi[m+8]),     .G2(St4g[m+8]), .P2(St4p[m+8]));
GROUPGP groupgp2 ( .G0(St3g[m+7]), .P0(St3p[m+7]), .G1(St1g[m+9]),
.P1(St1p[m+9]),   .G2(St4g[m+9]), .P2(St4p[m+9]));
GROUPGP groupgp3 ( .G0(St3g[m+7]), .P0(St3p[m+7]), .G1(St2g[m+10]),
.P1(St2p[m+10]), .G2(St4g[m+10]), .P2(St4p[m+10]));
GROUPGP groupgp4 ( .G0(St3g[m+7]), .P0(St3p[m+7]), .G1(St2g[m+11]),
.P1(St2p[m+11]), .G2(St4g[m+11]), .P2(St4p[m+11]));
GROUPGP groupgp5 ( .G0(St3g[m+7]), .P0(St3p[m+7]), .G1(St3g[m+12]),
.P1(St3p[m+12]), .G2(St4g[m+12]), .P2(St4p[m+12]));
GROUPGP groupgp6 ( .G0(St3g[m+7]), .P0(St3p[m+7]), .G1(St3g[m+13]),
.P1(St3p[m+13]), .G2(St4g[m+13]), .P2(St4p[m+13]));
GROUPGP groupgp7 ( .G0(St3g[m+7]), .P0(St3p[m+7]), .G1(St3g[m+14]),
.P1(St3p[m+14]), .G2(St4g[m+14]), .P2(St4p[m+14]));
GROUPGP groupgp8 ( .G0(St3g[m+7]), .P0(St3p[m+7]), .G1(St3g[m+15]),
.P1(St3p[m+15]), .G2(St4g[m+15]), .P2(St4p[m+15]));

end
endgenerate



assign C[1] = Pi[1];
assign C[2] = Pi[2]^Gi[1];
assign C[3] = Pi[3]^St1g[2];
assign C[4] = Pi[4]^St2g[3];

assign C[5] = Pi[5]^St2g[4];
assign C[6] = Pi[6]^St3g[5];
assign C[7] = Pi[7]^St3g[6];
assign C[8] = Pi[8]^St3g[7];
assign C[9] = Pi[9]^St3g[8];
assign C[10] = Pi[10]^St4g[9];
assign C[11] = Pi[11]^St4g[10];
assign C[12] = Pi[12]^St4g[11];
assign C[13] = Pi[13]^St4g[12];
assign C[14] = Pi[14]^St4g[13];
assign C[15] = Pi[15]^St4g[14];
assign C[16] = Pi[16]^St4g[15];

assign cout = St4g[16];
```

```
endmodule
```

Han-Carlson 16-bit adder

```
module han_carlson32(A,B,C,cout);
input [32:1] A;
input [32:1] B;
output [32:1] C;
output  cout;

wire [32:1] Gi,Pi;
wire [32:1]  St1g,St1p;
wire [32:1]  St2g,St2p;
wire [32:1]  St3g,St3p;
wire [32:1]  St4g,St4p;
wire [32:1]  St5g,St5p;

// Prefix Computation
genvar i;
generate
for ( i = 1; i <= 32; i = i+1 )
begin : bit_gen
BITPG bitpg (.Gi(Gi[i]), .Pi(Pi[i]), .Ai( A[i]),.Bi(B[i]) );
end
endgenerate
// Stage 1
genvar j;
generate
for ( j = 1; j < 32; j = j+2 )
begin
GROUPGP groupgp1 (.G0(Gi[j]), .P0(Pi[j]), .G1(Gi[j+1]),
.P1(Pi[j+1]), .G2(St1g[j+1]), .P2(St1p[j+1]) );
end
endgenerate
// Stage 2
genvar k;
generate
for ( k = 2; k < 32; k = k+2 )
begin
GROUPGP groupgp1 ( .G0(St1g[k]), .P0(St1p[k]), .G1(St1g[k+2]),
.P1(St1p[k+2]),     .G2(St2g[k+2]), .P2(St2p[k+2]) );
end
endgenerate
// Stage 3
genvar l;
assign St2g[2] = St1g[2];
assign St2p[2] = St1p[2];
generate
for ( l = 4; l < 32; l = l+2 )
```

```
begin
GROUPGP groupgp1 ( .G0(St2g[l-2]), .P0(St2p[l-2]), .G1(St2g[l+2]),
.P1(St2p[l+2]),     .G2(St3g[l+2]), .P2(St3p[l+2]) );
end
endgenerate

// Stage 4
genvar m;
assign St3g[2] = St2g[2];
assign St3p[2] = St2p[2];
assign St3g[4] = St2g[4];
assign St3p[4] = St2p[4];
generate
for ( m = 8; m < 32; m = m+2 )
begin
GROUPGP groupgp1 ( .G0(St3g[m-6]), .P0(St3p[m-6]), .G1(St3g[m+2]),
.P1(St3p[m+2]),     .G2(St4g[m+2]), .P2(St4p[m+2]) );
end
endgenerate

// Stage 5
genvar n;
assign St4g[2] = St3g[2];
assign St4p[2] = St3p[2];
assign St4g[4] = St3g[4];
assign St4p[4] = St3p[4];
assign St4g[6] = St3g[6];
assign St4p[6] = St3p[6];
assign St4g[8] = St3g[8];
assign St4p[8] = St3p[8];
generate
for ( n = 16; n < 32; n = n+2 )
begin
GROUPGP groupgp1 ( .G0(St4g[n-14]), .P0(St4p[n-14]), .G1(St4g[n+2]),
.P1(St4p[n+2]),     .G2(St5g[n+2]), .P2(St5p[n+2]) );
end
endgenerate


// Stage 6 Not comfortable to use generate
//genvar m;
//generate
//for ( m = 2; m < 32; m = m+2 )
//begin
//GROUPGP groupgp1 ( .G0(St1g[m]), .P0(St1p[m]), .G1(Gi[m+1]),
.P1(Pi[m+1]),     .G2(St2g[m+1]), .P2(St2p[m+1]) );
//end
//endgenerate
GROUPGP groupgp1 ( .G0(St1g[2]), .P0(St2p[2]), .G1(Gi[3]),
.P1(Pi[3]),     .G2(St2g[3]), .P2(St2p[3]) );
```

```
GROUPGP groupgp2 ( .G0(St2g[4]), .P0(St2p[4]), .G1(Gi[5]),
.P1(Pi[5]),     .G2(St2g[5]), .P2(St2p[5]) );
GROUPGP groupgp3 ( .G0(St3g[6]), .P0(St3p[6]), .G1(Gi[7]),
.P1(Pi[7]),     .G2(St2g[7]), .P2(St2p[7]) );
GROUPGP groupgp4 ( .G0(St3g[8]), .P0(St3p[8]), .G1(Gi[9]),
.P1(Pi[9]),     .G2(St2g[9]), .P2(St2p[9]) );
GROUPGP groupgp5 ( .G0(St4g[10]), .P0(St4p[10]), .G1(Gi[11]),
.P1(Pi[11]),     .G2(St2g[11]), .P2(St2p[11]) );
GROUPGP groupgp6 ( .G0(St4g[12]), .P0(St4p[12]), .G1(Gi[13]),
.P1(Pi[13]),     .G2(St2g[13]), .P2(St2p[13]) );
GROUPGP groupgp7 ( .G0(St4g[14]), .P0(St4p[14]), .G1(Gi[15]),
.P1(Pi[15]),     .G2(St2g[15]), .P2(St2p[15]) );
GROUPGP groupgp8 ( .G0(St4g[16]), .P0(St4p[16]), .G1(Gi[17]),
.P1(Pi[17]),     .G2(St2g[17]), .P2(St2p[17]) );
GROUPGP groupgp9 ( .G0(St5g[18]), .P0(St5p[18]), .G1(Gi[19]),
.P1(Pi[19]),     .G2(St2g[19]), .P2(St2p[19]) );
GROUPGP groupgp10 ( .G0(St5g[20]), .P0(St5p[20]), .G1(Gi[21]),
.P1(Pi[21]),     .G2(St2g[21]), .P2(St2p[21]) );
GROUPGP groupgp11 ( .G0(St5g[22]), .P0(St5p[22]), .G1(Gi[23]),
.P1(Pi[23]),     .G2(St2g[23]), .P2(St2p[23]) );
GROUPGP groupgp12 ( .G0(St5g[24]), .P0(St5p[24]), .G1(Gi[25]),
.P1(Pi[25]),     .G2(St2g[25]), .P2(St2p[25]) );
GROUPGP groupgp13 ( .G0(St5g[26]), .P0(St5p[26]), .G1(Gi[27]),
.P1(Pi[27]),     .G2(St2g[27]), .P2(St2p[27]) );
GROUPGP groupgp14 ( .G0(St5g[28]), .P0(St5p[28]), .G1(Gi[29]),
.P1(Pi[29]),     .G2(St2g[29]), .P2(St2p[29]) );
GROUPGP groupgp15 ( .G0(St5g[30]), .P0(St5p[30]), .G1(Gi[31]),
.P1(Pi[31]),     .G2(St2g[31]), .P2(St2p[31]) );

assign C[1] = Pi[1];
assign C[2] = Pi[2]^Gi[1];
assign C[4] = Pi[4]^St2g[3];
assign C[6] = Pi[6]^St2g[5];
assign C[8] = Pi[8]^St2g[7];
assign C[10] = Pi[10]^St2g[9];
assign C[12] = Pi[12]^St2g[11];
assign C[14] = Pi[14]^St2g[13];
assign C[16] = Pi[16]^St2g[15];
assign C[18] = Pi[18]^St2g[17];
assign C[20] = Pi[20]^St2g[19];
assign C[22] = Pi[22]^St2g[21];
assign C[24] = Pi[24]^St2g[23];
assign C[26] = Pi[26]^St2g[25];
assign C[28] = Pi[28]^St2g[27];
assign C[30] = Pi[30]^St2g[29];
assign C[32] = Pi[32]^St2g[31];

assign C[3] = Pi[3]^St1g[2];
assign C[5] = Pi[5]^St2g[4];
assign C[7] = Pi[7]^St3g[6];
assign C[9] = Pi[9]^St3g[8];
```

```
assign C[11] = Pi[11]^St4g[10];
assign C[13] = Pi[13]^St4g[12];
assign C[15] = Pi[15]^St4g[14];
assign C[17] = Pi[17]^St4g[16];
assign C[19] = Pi[19]^St5g[18];
assign C[21] = Pi[21]^St5g[20];
assign C[23] = Pi[23]^St5g[22];
assign C[25] = Pi[25]^St5g[24];
assign C[27] = Pi[27]^St5g[26];
assign C[29] = Pi[29]^St5g[28];
assign C[31] = Pi[31]^St5g[30];

assign cout = St5g[32];

endmodule
```

BIBLIOGRAPHY

[1] R. K. Richards, *Arithmetic Operations in Digital Computers*, Princeton, N.J.: D. Van Nostrand Co., 1955.

[2] A. Weinberger and J. Smith, "A logic for high-speed addition," National Bureau of Standards, no. 591, pp. 3–12, 1958.

[3] J. Sklansky, "Conditional-sum addition logic," *IRE Transactions on Electronic Computers*, vol. EC-9, pp. 226–231, 1960.

[4] O. J. Bedrij, "Carry-select adder," *IRE Transactions on Electronic Computers*, vol. EC-11, pp. 340–346, 1962.

[5] M. Nadler, "A High-Speed Electronic Arithmetic Unit for Automatic Computing Machines," *Alta Technica* (Prague), vol. 6, pp. 464-478, 1956.

[6] H. Ling, "High speed binary adder," *IBM Journal of Research and Development*, vol. 25, pp. 156–166, 1981.

[7] Thomas Lynch and Earl E. Swartzlander, Jr., "A Spanning Tree Carry Lookahead Adder," *IEEE Transactions on Computers*, vol. 41, 1992, pp. 931-939.

[8] B. Parhami, *Computer Arithmetic, Algorithm and Hardware Design*, New York: Oxford University Press, pp. 91-119, 2000.

[9] D. Harris, "A taxonomy of parallel prefix networks," in *Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, Nov. 2003, pp. 2213–2217.

[10] P. Kogge and H. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence relations," *IEEE Transactions on Computers*, vol. C-22, pp. 786–793, 1973.

[11] S. Knowles, “A family of adders,” *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, June 2001, pp. 277–281

[12] R. P. Brent and H. T. Kung, “A regular layout for parallel adders,” *IEEE Transactions on Computers*, vol. C-31, pp. 260–264, 1982.

[13] R. Ladner and M. Fischer, “Parallel prefix Computation,” *Journal of the ACM*, vol. 27, pp. 831–838, 1980

[14] T. Han and D. Carlson, “Fast area-efficient VLS Adders,” *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, Sept. 1987, pp. 49–56

[15] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, “FreePDK: An open-source variation-aware design kit,” *Proceedings of IEEE International Microelectronic System Education*, 2007, pp. 173–174

[16] Synopsys 32/28nm Interoperable PDK for Teaching AMS/Custom Design http://www.synopsys.com/Community/UniversityProgram/Pages/32-28nm-ipdk.aspx

VITA

Vignesh Naganathan graduated from National Institute of Technology, Tiruchirapalli, India as Bachelor of Technology in Electronics and Communication Engineering in 2010. He pursued Masters of Science in Electrical Engineering with focus on Integrated Circuits and Systems at The University of Texas at Austin.

After completing the Bachelor's degree, Vignesh Naganathan worked as ASIC Design Engineer at NVidia Corporation, India from 2010 to 2013. He is working at Apple Inc., as CPU Implementation Engineer since January 2015. He can be reached at vignesh.radnag@gmail.com. This report was typed by the author.