

THESIS
1996
L991
DISS
COP.1

UNIVERSITY OF TEXAS AT AUSTIN - GEN LIBS



3002830422

0 5917 3002830422

THE GENERAL LIBRARIES
THE UNIVERSITY
OF TEXAS
AT AUSTIN



PRESENTED BY
THE AUTHOR

THE UNIVERSITY OF TEXAS AT AUSTIN
THE GENERAL LIBRARIES
PERRY-CASTAÑEDA LIBRARY
LIMITED CIRCULATION

DATE/TIME DUE	DATE/TIME RETURNED
PCL RESERVES DEC 20 1996 LIBRARY USE ONLY	PCL / RES DEC 20 1996
PCL RESERVES MAR 14 1997 LIBRARY USE ONLY	PCL / RES MAR 14 1997

BINARY ADDERS

by

THOMAS WALKER LYNCH, B.S.E.E.

THESIS

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

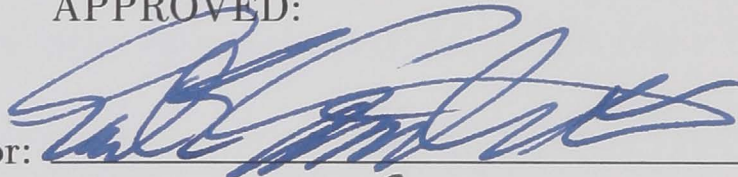
May, 1996

BINARY ADDERS

To students of computer arithmetic, may they find this useful.

APPROVED:

Supervisor:



7/9 Lagon

THIS IS AN ORIGINAL MANUSCRIPT
IT MAY NOT BE COPIED WITHOUT
THE AUTHOR'S PERMISSION

To students of computer arithmetic, may they find this useful.

Acknowledgments

It occurred to me to thank “God, country and family” for support. However, my thankfulness for Life, Liberty, and Home, transcends that which can be acknowledged through the presentation of an engineering thesis.

I think it is proper to acknowledge the teachers and fellow engineers who have read this paper and discussed its contents. I am thankful that my advisor has created a forum where such work is possible, and that my employer has taken up the development of a product for which this material is relevant. I hope that this thesis will, in a small way, represent my gratitude for these things which made this writing possible.

THOMAS WALKER LYNCH

The University of Texas at Austin

May, 1996

ABSTRACT

BINARY ADDERS

by

THOMAS WALKER LYNCH, M.S.E.

THE UNIVERSITY OF TEXAS AT AUSTIN, 1996

SUPERVISOR: Dr. Earl E. Swartzlander, Jr.

This thesis focuses on the logical design of binary adders. It covers topics extending from cardinal numbers to carry skip optimization. The conventional adder designs are described in detail, including: carry completion, ripple carry, carry select, carry skip, conditional sum, and carry lookahead. We show that the method of parallel prefix analysis can be used to unify the conventional adder designs under one parameterized model. The parallel prefix model also produces other useful configurations, and can be used with carry operator variations that are associative. Parallel prefix adder parameters include group sizes, tree shape, and device sizes. We also introduce a general algorithm for group size optimization. Code for this algorithm is available on the World

Wide Web¹. Finally, the thesis shows the derivation for some carry operator variations including those originally given by Majerski and Ling.

¹<http://devil.ece.utexas.edu/~lynch>

Table of Contents

Acknowledgments	iv
ABSTRACT	v
List of Figures	x
List of Tables	xii
Chapter 1. Introduction	1
1.1 Aspects of Addition	1
1.2 The Role of Binary Adders in Microprocessors	3
1.3 Survey	5
1.3.1 Conceptual	6
1.3.2 Strategies and Logic	7
1.3.3 Structured Technology Mapping	17
Chapter 2. From Cardinal Numbers to Mechanical Adder	21
2.1 The Representation of Cardinal Numbers and Unary Addition .	21
2.2 Higher Number Systems	23
2.3 Ripple Carry Addition in the Arabic System	25
Chapter 3. Conventional Addition Algorithms	27
3.1 Serial	27
3.2 Ripple Carry	29
3.3 Carry Select	32

3.4	Ripple Carry Select	34
3.5	Conditional Sum	36
3.6	Propagate Generate Class of Adders	38
3.6.1	Propagate Generate Ripple Carry	41
3.6.2	Carry Skip	45
3.6.3	Carry Lookahead	48
Chapter 4. Gate Delay Models for the Conventional Adders		50
4.1	Ripple Carry	52
4.2	Carry Skip	63
Chapter 5. <i>cso</i> Operator Based Adders		71
5.1	One Level Structure - Ripple Carry	71
5.2	Two Level Structure - Carry Select	73
5.3	N Level Structures	79
Chapter 6. <i>fco</i> Operator Based Adders		84
6.1	One Pass <i>fco</i> Trees	84
6.2	Folded <i>fco</i> Trees – Conventional Adders	88
Chapter 7. Optimization		94
7.1	Definition of the Carry Skip Optimization Problem	94
7.2	Weak Monotonicity in the Delay to Sum Function	96
7.3	Weak Reciprocal Relationship between Partitions	97
7.4	Optimum Carry Skip Algorithm	97
7.5	Module Parameters	100
7.6	Optimization Program Assumptions	103
7.7	Optimization Program Examples	105

Chapter 8. Alternative Carries	108
8.1 Relationship between Carry, Propagate, and Generate	108
8.2 Majerski's nor Gate Ripple Carry Adder	113
8.3 Ling's Adder	116
8.4 Other Adders	116
Chapter 9. Performance	119
9.1 Worst Case Path Lengths	119
Chapter 10. Conclusion	123
Bibliography	125
Vita	136

List of Figures

1.1	Carry Lookahead Adder	12
1.2	Carry Select Adder	15
2.1	Adding with Odometers	26
3.1	Serial Addition	28
3.2	Ripple Carry Addition	31
3.3	Carry Select Addition	33
3.4	A Ripple Carry Select Addition	35
3.5	Recursive Carry Select Addition – Conditional Sum	37
3.6	A Propagate Generate Version of Ripple Carry Addition	42
3.7	A Transfer Generate Version of Ripple Carry Addition	44
3.8	Carry Skip Addition	48
4.1	Generate and Propagate Logic Blocks	53
4.2	XOR and XNOR Blocks	53
4.3	Comparison between OR-NAND, AND-NOR and Simple Gates	54
4.4	4 Bit Ripple Carry	56
4.5	3 Bit xor Gates	57
4.6	Net List of Two Series Inverters	58

4.7	Three Full Adder Variations	62
4.8	Carry Skip Section	64
4.9	A Three Section Carry Skip Adder	64
5.1	Ripple Carry Adder	72
5.2	Carry Select Adder	75
5.3	Ripple Carry Select Adder	77
5.4	Conditional Sum Adder	80
5.5	Simplified Conditional Sum Adder	82
6.1	Brent & Kung Adder	87
6.2	Kogge & Stone Adder	87
6.3	Han & Carlson Adder	88
6.4	Carry Skip Adder, Timing Correct	90
6.5	Carry Skip Adder, Logic Correct	90
6.6	Carry Skip Adder, Layout Correct	92
6.7	Carry Skip Adder, Variable Block Sizes	92
6.8	Manchester Carry Skip Adder - Real Evaluation Time	93
8.1	Carry Lookahead with Partition	109
8.2	Transforming Standard Ripple to Majerski Style Ripple	115
8.3	nand per Stage Ripple Carry Adder	118

List of Tables

9.1	Evaluation Time for One Level Skip Adders	121
9.2	Evaluation Time for Four Level Skip Adders	122
9.3	Worst Case Path Lengths Through Various Adders	122

Chapter 1

Introduction

1.1 Aspects of Addition

Since 1960 there have been over 700 papers written with something about addition in them. In an effort to narrow the topic we note that these papers often focus on one of the following aspects:

- characteristics of the physical substrate,
- model for a logic element or fundamental operation,
- number system,
- logical configuration of the adder,
- efficient mapping of the configuration to the substrate.

For instance, substrates have included things as unusual as organic molecules in solution and super cold alloys. Practical substrates have included various things such as cams and gears, air passages and valves, relays, tubes, silicon, Gallium Arsenide, etc.

The logic model is the bridge between symbolic logic design and the physical substrate. Some common logic models include *RTL*, resistor transistor

logic; *DTL*, diode transistor logic; *TTL*, transistor transistor logic; *ECL*, emitter coupled logic; *NMOS*, N-type metal oxide semiconductor; and *CMOS*, complementary metal oxide semiconductor. These can be broken down further into static, dynamic domino, complementary voltage switch logic (CVSL), pseudo NMOS, etc.

Many number systems have been used, the most common being the base two Arabic system, which we just call *binary*. Some of the more esoteric include the signed digit, residue, and logarithmic number systems. Negative numbers in the Arabic system have typically been handled with variations called two's complement, one's complement, and sign magnitude.

The conventional logic configurations for adders are carry completion, ripple carry, carry skip, carry select, conditional sum, and carry lookahead.

The problem of mapping adder configurations to the substrate (i.e, the problem of optimally using the logic model) has been embodied in a carry skip optimization algorithm. According to carry skip optimization, the configuration is fixed to a multilevel carry tree made of variable length modules. The optimizer then finds the best number of levels, number of blocks, and sizes for the blocks. An improved approach to optimization also sizes the buffers which drive critical speed carry signals.

This thesis narrows the field of study to those logical configurations which would not be unusual to find in CMOS implementations. It further limits the scope to binary adders which would be implemented in either static or dynamic domino gates.

1.2 The Role of Binary Adders in Microprocessors

The need to identify optimum adder designs for a modern microprocessor initiated this study. Adders often appear in the integer execution unit, and sometimes in the address generation path. If a floating-point unit is present they appear in the significand adder, at the base of multiplier array, and in the divider. Smaller adders appear in the exponent manipulation circuitry for multiply and divide. Incrementers and comparators are also forms of adders, and they appear in various places. Hence, the identification of an appropriate adder generator is a high leverage tool for creating an efficient design.

Adder design requirements vary. For example in some cases it is desirable for the execution unit adder to be very flexible, leaving speed and area as secondary constraints. The Intel 80486 execution unit adder was designed to naturally produce carries on 8, 16, or 32 bit boundaries since these are the native data types [1] for that architecture. In the high end Alpha [2] the large word width, fine grain pipe, and high clock speed, conspire to make speed the primary requirement.

Outside of the execution units of the integer and floating-point cores there are few adders in a RISC processor. Indeed one of the advantages of RISC architecture is the removal of adders from the critical paths such as address generation. However, not all processors are RISC designs. The x86 architecture requires a four operand adder for address generation¹.

Often instruction mixes show addition to be the most common arithmetic

¹segment + base + index + displacement

instruction. However, there are many stages involved in the execution of any instruction, so the significance of the evaluation time of the adder circuit, which dominates only the execution stage, is not always obvious[3, 4]. In the remainder of this section we will describe when adder circuit evaluation time is important.

In all but the least sophisticated processors, instructions are pipelined through fetch, decode, and execution stages. In modern processors there is also fine grain parallelism in the form of multiple execution units which run in parallel. Hence, the focus of the processor is to keep the execute units busy doing useful work, and to the extent this is possible, the speed of the execution units is important. Factors/features that effect keeping the execute units busy include:

- an efficient operating system that allocates sufficient resources
- instruction level parallelism and locality in the code
- a large enough decode window to take advantage of the parallelism
- large enough caches to take advantage of locality
- successful branch prediction to decrease the penalty of branches
- low penalty on misprediction of branches
- register renaming to remove false dependencies
- out of order issue and execution so that stalled instructions do not stop the pipe.
- result forwarding to reduce latency in dependent sequences

- an instruction set (architecture) that facilitates the above goals

These items place a large burden on the operating system, the compiler, the memory system, and the decoder. When this burden is met - performance may be strongly determined by adder latency. That is to say, when instructions can be provided at a maximum rate with maximum parallelism, execution speed is determined solely by serially dependent instruction steps for which add instructions are common [5, 6]. Most machines can approach this ideal only in specialized applications.

1.3 Survey

Perhaps it is reasonable to say that adders have gone through three major stages of development. Initially philosophers/mathematicians grappled with the problem of conceptualizing how an abstract operation such as addition could be performed via an incarnate machine. Then, once addition machinery was common place, the focus moved on to a sort of competition among ad hoc procedures for generating faster sums. Most recently progress has been in the area of optimization of free variables, such as block widths and driver sizes, on an otherwise determined structure.

The remainder of the sections in this introduction, and subsequent chapters in this thesis, follow the evolutionary steps hypothesized in the previous paragraph. The discussion in chapter 2 develops from the practical use of cardinal numbers to the realization of a mechanical ripple carry adder. Chapters 3 and 4 cover strategies for configuring logic. Chapters 5 and 6 show how generalized carry hierarchies can be created and parameterized. Chapter 7 dis-

cusses optimizing the parameters in a constrained design in order to maximize performance. Chapter 8 covers a side issue of how the unconventional carry recursions that some authors have proposed fit into the general *fco* scheme (*fco* is introduced in chapter 3 and developed in chapter 6).

1.3.1 Conceptual

Developing a history of adders could be thesis in itself; here we endeavor only to give some perspective. Randell places the oldest known mechanical adder with Hero of Alexandria [7]. During the renaissance some astronomers built machines to model events in the heavens - no doubt some of these embodied addition operations. In the 17th century Blaise Pascal built a calculating machine, not much later so did Gottfried Leibniz. In the late 19th century Charles Babbage made the first modern computer architecture, his drawings showed mechanisms which performed carry skip addition [7, 8, 9].

Some of these machines used rotating wheels organized in a fashion similar to mechanical odometers. Such machines perform operations in direct analogy to the structure of the Arabic number representation, a concept described in more detail in chapter 2.

Calculating machines prospered along with the industrial revolution; perhaps culminating with the ‘Mark’ series of machines built by Howard Aiken and associates. ‘Mark I’ which was delivered in 1944 occupied a small room and had a main axle speed of about 200 r.p.m. [9]. Two years later Ekert and Machley had a completely electronic machine based on digital pulse trains, ENIAC [10]. ENIAC’s internal architecture was based on circular shift registers made from

vacuum tube circuits in analogy to indexed mechanical gears [11]. ENIAC had an axle speed equivalent of about 45000 r.p.m.

1.3.2 Strategies and Logic

In the late 1930s Claude Shannon combined a propositional calculus (which traces its roots to Leibniz), with the binary valued algebra of George Boole, and the electrical characteristics of electronic relays, and produce the method of logic design as we know it today [12]. Shannon's work allowed the question of what is the most appropriate logic configuration for an adder to be asked. Shannon gave the first logic design for a ripple carry adder, and thus supplied the conceptual bridge leading from mechanical to logic based adders.

In Germany, Konrad Zuse and his associates independently developed a propositional calculus into a method for logic design. They implemented a programmable relay based computer in the mid 1940s [13] at about the same time Howard Aiken's work. In the United States the work of Shannon was applied to a programmable machine by Burks, Goldstine, and Von Neumann [14, 15, 16] in the vacuum tube based EDSAC.

Shortly after this change in paradigm many of the conventional adder circuits were introduced into the literature. Faster or smaller adders were celebrated. We will call these the *conventional* logic designs:

- ripple carry
- carry completion
- carry skip

- carry select
- conditional sum
- carry lookahead

All of these, except possibly carry completion, are discussed in common texts on the subject [17, 18, 19, 20, 21].

Carry Completion

It was noted by Von Neumann [16], and later refined by Briley, Glass, and Varshavskii [22, 23, 24, 25], that given uniformly distributed random operands, the *average* length of the longest carry propagation chain is proportional to the log of the width of the adder. Hence, there is a performance advantage in detecting the event of a carry propagation completing, and then using the sum immediately. A quintessential carry completion circuit is that of Gilchrist, Pomerene, and Wong [26].

Perhaps there are two reasons that the carry completion scheme is uncommon today. One is that asynchronous architectures never became common. The other reason, which may also have contributed to the demise of asynchronous machines, is that other adder circuits also reach log time performance.

The fastest theoretical addition time reported is that of the conditional sum adder with carry completion logic. This is reported to add in an average of $\Omega \log \log N$ time [27] where N is the word size. Ω is used here to indicate that this is reasonable approximation of the addition time - *bounded from below*. If this is a tight bound, then this adder is faster than $O(\log N)$.

Carry Skip

In [28] Lehman and Burla summarized the purpose of carry skip addition:

In their contribution to the IEE Discussion Meeting on New Digital Computer Techniques, Morgan and Jarvis [29] describe a binary skip circuit. The technique is based on the detection and by-passing of those stages of a parallel binary adder in which, during a given addition ($X+Y$), there exists the condition for carry-propagation. That is, the carry signal is enabled to by-pass those stages of carry circuits for which $x_i \neq y_i$. An alternative criterion which does not differentiate between propagated and generated carries, but which is more efficient in the skip circuit is $x_i \vee y_i$.

Lehman and Burla found the optimum number of equal size groups for a carry skip adder, and suggested that variable block sizes would be helpful. They suggested that succeeding groups should increase in width, until near the middle of the adder, and then the groups should decrease. Indeed they also suggested that multiple levels of skip would be advantageous. This paper set the stage for much of the work to come on the subject.

Majerski [30] published the formulas necessary for accomplishing the variable block size optimization. He also extended the work and gave formulas for two level adders. As significant as this work is to the evolution of adder design, it fails to help in the modern engineering of optimum adders. Majerski assumed in one of his adders that the bitwise *exclusive-or* of the operands was available upon input. He also used wired OR logic. Wire delays and nonrestoring logic were not considered. Nor were a derivations given for the formulas.

Oklobdzija and Barnes [31] published a proven optimum scheme for determining one level and two level carry skip adder group lengths. Previous

optimization schemes were heuristic. They assumed that the time necessary for skipping a block was constant; which because of significant metal delays in MOS processes, was not ideal. Guyot, Hochet, and Muller [32] improved this result by allowing the rate of increase and decrease in group size to be set from a ratio between the ripple speed and the skip speed.

The evolution of carry skip adders, more so than work on other adders types, has pushed the mapping of technology back onto logical structure. The optimization engine drives this mapping. Oklobdzija published another paper [33] showing that a well designed carry skip adder was faster than other common types implemented in the same technology, including carry lookahead and hybrid parallel prefix adders. Included in this paper was a description of the linear increase in gate delay with increasing fan-in (with constant fan-out), and a linear increase in gate delay within increasing fan-out (with constant fan-in). However, this paper did not give an algorithm for integrating the technology parameters into the adder design beyond those already known.

Turrini published a heuristic search algorithm capable of producing carry skip adders with arbitrary numbers of levels[34]. Although this approach was potentially less encumbered by specific process data, the presentation did not include the integration of more process information, nor were the adders provably optimum. Also, the algorithm appears to require a large amount of execution time.

Chan and Shlag published some solid work [35] which successfully integrated second order CMOS technology parameters into one level carry skip

optimization. They analyzed RC models of the Manchester carry chain² and reasoned that ripple delay is proportional to the square of the length of the block, that skip delay is linearly related to the length of the block, and that there is a constant restoring delay between stages. They then assembled carry life time equations, based on a carry being generated in the first block, and then skipping, and being absorbed by the last block. They minimized this path, while making sure no other paths were longer. Also, the material presented in this paper does assist in the design of multiple levels of skip.

Carry Lookahead

For the sake of maintaining consistent nomenclature, we use the term *Carry Lookahead* exclusively to describe the adders of the form shown in figure 1.1. In this circuit each carry is calculated independently from propagate and generate signals (propagate and generate are described in chapter 3). The more significant the carry, the larger the fan-in of the carry lookahead logic. Early on engineers realized that the carry lookahead adder was limited to only four or five bits [37], so they began to build tree structures from carry lookahead modules.

Swartzlander [18] cites Weinberger and Smith's paper [38] as the seminal work on carry lookahead *trees*. Weinberger showed that carry lookahead modules can be connected together in shallow trees to form a composite adder. Weinberger's adders produce all of the carries in one pass through the tree.

²The Manchester carry chain was originally described by Kilburn, Edwards, and Aspinall [36].

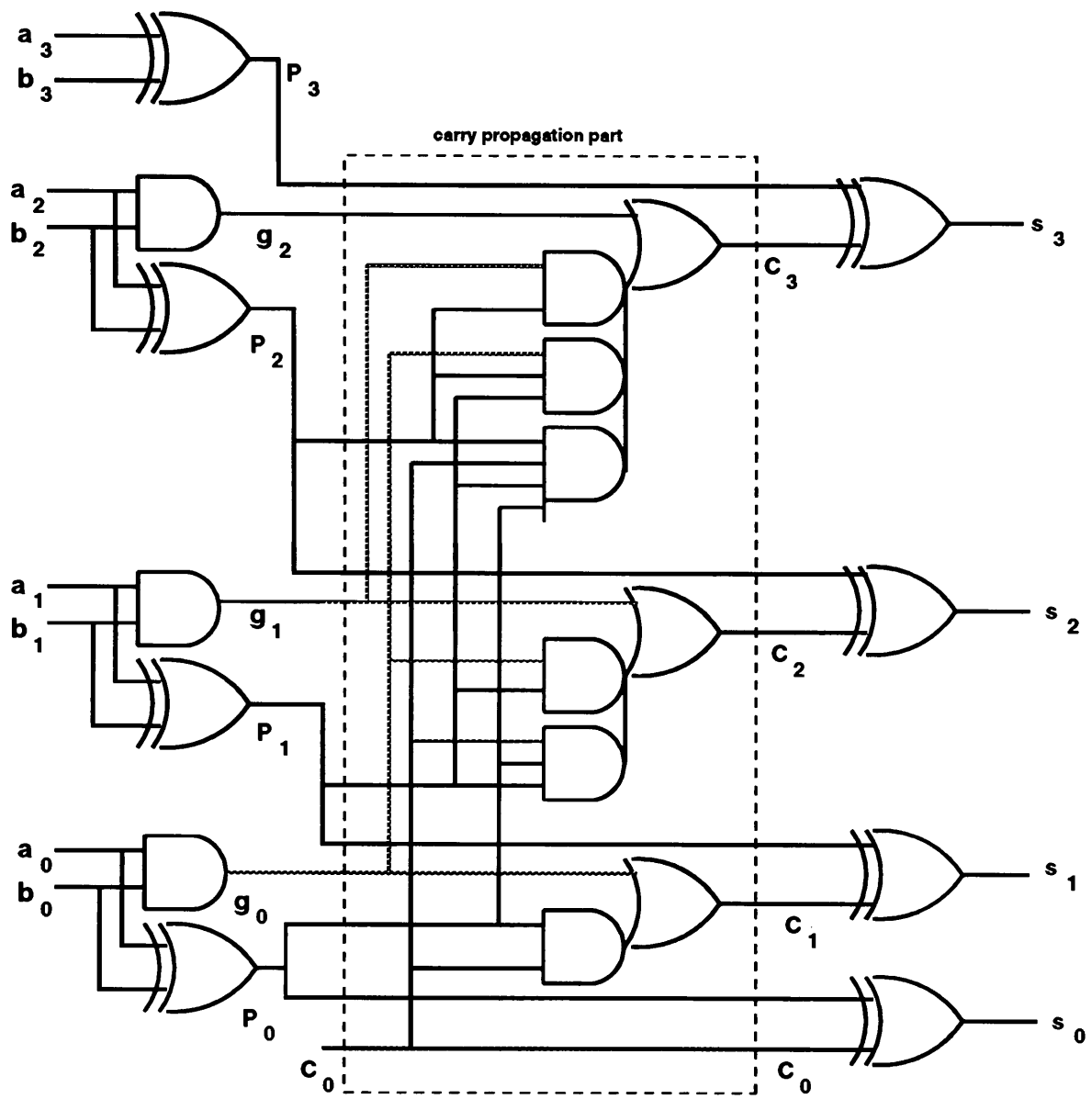


Figure 1.1: Carry Lookahead Adder

MacSoreley [39], presented what is the most commonly recognized *carry lookahead adder*. This structure cleverly conserves devices by using two passes. First the tree is used for calculating carries at indices of powers of the module size. Second, these carries are used to initialize the module inputs in the tree so that re-evaluation produces carries instead of group generate signals. The two steps typically flow together without any clock boundaries.

Davis [40] describes the carry lookahead adder used in ILLIAC IV.

In 1971 Texas Instruments introduced two TTL chips for building MacSoreley style carry lookahead tree adders, the 74181 and the 74182. The 74181 contains the base adder, while the 74182 is a carry lookahead module. It is probably the availability of these chips which made this style of adder commonplace.

In [41] Weinberger showed a PLA based carry lookahead adder with some interesting variations on the carry lookahead equation. Four years later he published yet another ripple carry module based carry lookahead adder. In this adder the ripple carry module was implemented with a pass gate version of the Manchester carry chain [42].

Bechade and Hoffman [43] published a static NMOS based module, along with a comparison between carry lookahead and ripple carry adder performance in their NMOS implementations. Rhyne [37] published an often cited work on the fan-in and fan-out limitations of carry lookahead modules, but certainly by this time the problem was already well understood. Crawley [44] published a representative paper on pipelining a carry lookahead adder. Hwang [45] published a reconfigurable carry lookahead adder which may have been the

basis of [1].

Fagin [46] published an interesting paper with a summary of fan-in/fan-out restrictions. He tied into the parallel prefix work (surveyed in section 1.3.3), and he pointed out that with constant time communication delay it is possible to obtain the theoretical log time performance described by Winograd [47] and Ofman[22].

Lee, Park, and Kyung [48] published a clever paper with the corrected version of the CMOS static carry lookahead module that was erroneously portrayed in Weste and Eshraghian's book [19]³. They also showed how the module can be modified to return prefix functions. Normally prefix functions are only directly obtained with Manchester carry chains.

Carry Select

For the sake of maintaining consistent nomenclature, we use the term *Carry Select Adder* exclusively to describe the circuit of the form shown in figure 1.2. This figure shows the sum output from two adder modules, one with a carry-in of one, the other with a carry-in zero, going into a multiplexer. The select line of the multiplexer chooses between the two sums based on the actual carry-in. Unless otherwise stated, the adder modules are ripple carry adders.

Bedrij introduced the “carry select adder” in [49]. This paper shows a circuit where carry select modules are used at the base of a carry lookahead

³first edition only, page 323

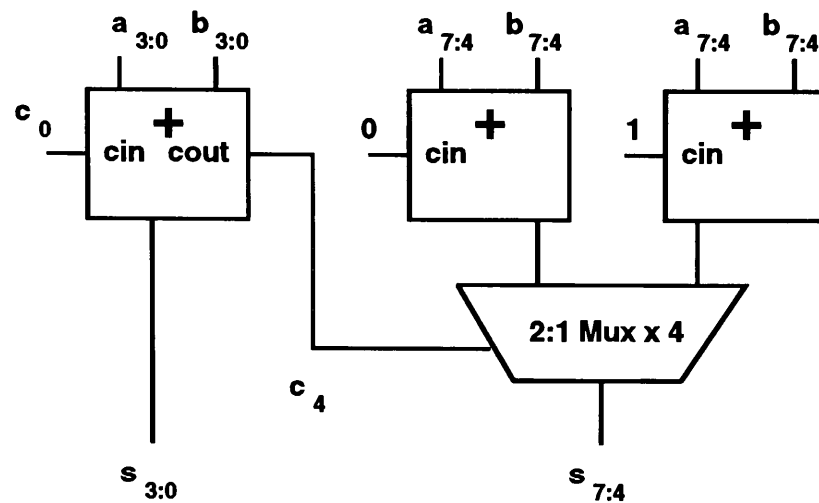


Figure 1.2: Carry Select Adder

tree instead of the expected carry lookahead style modules. In this adder sparsely separated carries are assembled going up the tree, these carries are then folded back into the tree, and upon arriving at the bottom they form carries on every module boundary. The module boundary carries are then used select the appropriate sum.

To the extent that higher fan-out is not a problem, this configuration is faster than what would have been achieved with a MacSoreley type carry lookahead adder tree, because of the last step. In Bedrij's adder there is only one multiplexer delay after all the carries are generated, instead of a carry lookahead module delay followed by an exclusive-OR. Variations which would sacrifice some speed, would also reduce the number of devices.

Lynch and Swartzlander improved upon this approach by taking advantage of parallel prefix methods and the idempotency of the carry combining operator to obtain the select carries on small groups in one pass through the tree [50, 51]. Kantabutra [52] improved upon Lynch and Swartzlander's adder [51] configuration by using variable sized modules. This approach is related to

the block carry lookahead optimizations. Even more recently, Nigaglioni and Swartzlander have developed further improvements [53].

Bedrij's adder is often not what is referred to by the term *carry select adder*. His adder might more aptly be described as a hybrid carry lookahead carry select. Commonly the term carry select adder refers to carry select modules hooked up in series, as in [20]. In the series scheme the width of carry select modules may be adjusted to improve the speed as is done for carry skip adders.

As noted, a carry select module multiplexes between two module sums which were created as a result of different constant carry inputs. If these modules are in turn implemented with smaller carry select adders (as explained in chapter 5), the result is the conditional sum adder. Sklansky introduced the conditional sum adder in [54].

The conditional sum adder has logarithmic performance like the carry lookahead adder, but it does not fold carries back into the tree, and therefore potentially has fewer levels of logic from input to output. This adder requires many more devices than either Bedrij's carry select adder, or Weinberger or MacSoreley's carry lookahead tree adders. The extra devices are probably so cumbersome that in practice the adder will be slower than the other configurations.

A pipelined version of the conditional sum adder was presented by Hallin in 1972 [55]. A carry lookahead style circuit which propagated conditional propagate and generate signals (i.e. one set for carry-in of zero, and one for carry-in of 1) was presented by Tyagi in 1990 [56]. There is little advantage in this configuration since the usual group propagate and generate signals can

already be locally combined to produce the same signals, as shown in chapter 8.

1.3.3 Structured Technology Mapping

Both carry skip adders and parallel prefix adders were the first to be defined in terms of free parameters which could be optimized to fit technology constraints. As we show in chapter 6, the parallel prefix method is a general approach which can encompass carry skip addition. Commonly optimized parameters have included block widths, tree configuration, and driver sizes. Block width optimization was surveyed with carry skip adders, and will be discussed in more detail in chapter 7. Tree configurations for binary carry lookahead nodes, and driver sizing for them are surveyed in this section.

After studying the arithmetic implementation problems on ILLIAC IV, Kogge and Stone [57] observed that carry propagation has the form of a *parallel prefix calculation*. The ripple carry formula for calculating c_n is

$$c_n = pg_0 \ fco \ pg_1 \ fco \ \dots \ fco \ pg_{n-3} \ fco \ pg_{n-2} \ fco \ pg_{n-1} \quad (1.1)$$

The longest proper prefix of this sequence of operations,

$$c_{n-1} = pg_0 \ fco \ pg_1 \ fco \ \dots \ fco \ pg_{n-3} \ fco \ pg_{n-2} \quad (1.2)$$

calculates c_{n-1} , etc. Accordingly, the problem of fast carry propagation, is equivalent to the problem of calculating all the prefixes to equation 1.1 as quickly as possible. Kogge and Stone identified the carry operator, which we call *fco*, and showed how to take advantage of its associativity to create log time adders.

Unger [58] showed many variations on folded⁴ carry lookahead networks with varying performance. This work was not tied directly into Kogge's work, but does indirectly demonstrate the flexibility of the carry operator method.

Ladner and Fisher [59] used the parallel prefix notation to show direct tradeoffs between the number of carry combining operations used and the number of carry operations along the critical speed path.

Brent and Kung [60] showed a parallel prefix adder which uses a tree followed by an *inverse tree* to limit the fan-out of each node to just two other devices. Furthermore they showed how the tree can be efficiently layed out on a grid, where each node either contains a carry operator or a buffer.

In [61] Montoye introduced CMOS technology parameters instead of speaking of just fan-in and fan-out. He considered drive strengths and capacitances and related them to performance and cost. However, he considered only a limited form of parallel prefix adder.

Ngai and Irwin [62] showed that parallel prefix adders can be efficiently layed out in a square instead of a long strip.

Chen and Wei [63, 64, 65, 66] reiterated a parallel prefix adder with some driver sizes left as free parameters for optimization. They also discussed recursive methods for generating Brent and Kung's and Ladner and Fisher's parallel prefix adders.

Han and Carlson [67] reviewed the more salient parallel prefix adder configurations and discussed a method for building adders with performance and

⁴MacSoreley style carry lookahead tree

complexity between that suggested by Kogge and Stone, and that of Brent and Kung. They continued the study of fixed fan-in and fan-out adders in [68].

Sugla and Carlson [69] pointed out that the area/time tradeoffs in parallel prefix addition on constant fan-in/fan-out circuits is peculiar in that the area rises disproportionately as evaluation time is reduced. Lee and Oklobdzija [70, 71] later suggested this was due to not assigning optimum variable carry lookahead module lengths.

Fishburn discussed the algorithm used by a program to reduce the depth in Ladner and Fisher's parallel prefix adders [72]. In the same vein Hsu and Bair discuss a compiler for generating fast adders [73].

In [51] Lynch and Swartzlander presented a parallel prefix adder based on four bit Manchester carry chain modules. By taking advantage of the idempotency property of the carry operator the tree was able to produce carries on all 8 bit boundaries. Also the use of 4 bit modules in the tree lead to more ideal device input to output load ratios and thus faster evaluation than that of a 2 bit module based tree.

As we discuss in chapters 5 and 6, the parallel prefix method is general enough to encapsulate all of the conventional adder designs. For example, when ripple carry modules are used in a folded tree the adder is called a carry skip adder. If these ripple modules are built out of Manchester carry chains, and all of the modules are the same length, the adder is commonly known as a Manchester carry lookahead adder. If the modules are of various lengths, and the tree has $N + 1$ levels, then the adder is called an N level carry skip adder. Finally, when carry lookahead modules are used, but the modules are varied in

size, the adder is conventionally called a block carry lookahead adder.

Chapter 2

From Cardinal Numbers to Mechanical Adder

2.1 The Representation of Cardinal Numbers and Unary Addition

A flock of four sheep and a grove of four trees are related to each other in a way in which neither is related to a pile of three stones or a grove of seven trees. Although the words for numbers have been used to state this truism on the printed page, the relationship to which we refer underlies the concept of cardinal number. *S.C.Kleene* [74].

Herders have counted animals going out to graze by marking on sticks, or by putting rocks in a pile. This fundamental arithmetic is tied to an abstract system by recognizing that such a rock used for counting *represents a counting symbol*. The individuality of the rocks is not significant for purposes of counting, so there is only one symbol in the abstract system. Hence such a system is called *unary*. Each pile of stones thusly created *represents* the abstraction we call a *number*. Like the represented counting symbols, the represented numbers have no important distinguishing characteristics – beyond the fact that they embody the placed stones. It follows that piles have no internal organization.

The term *instantiation* refers to the act of choosing a random stone from outside the system, and then placing it in the pile as a counting object. Instantiation of a symbol provides a step away from direct manipulation of bulky physical objects towards the manipulation of less bulky physical objects which

may be as simple as ink on paper. That is to say, instantiation is the mechanism that facilitates abstraction. Instantiated objects originate from a disorganized pool of objects and are then placed into an environment organized into piles. Hence, instantiation requires a quanta to be placed into the system, and it reduces the entropy of the environment system combination.

In this system the operation of combining piles is *addition*. The movement of an object from one location to another of equivalent potential energy requires no net input of energy, so combining the rock piles does not appear to require that more energy is placed into the system. However, from an entropy point of view, combining piles disorganizes the system, since there is less knowledge about the location of the stones after the operation is performed. Hence, it is not possible to start with a combined pile and then “reverse add” into two piles without employing some knowledge about the original piles. This classical “requires no energy” versus the thermodynamic “requires energy” is the same essential contradiction captured by the Maxwell’s daemon problem.

The answer may be found in an analogous quantum mechanical problem. Here, the piles are like containing boxes, and the stones are like indistinguishable particles with unknown locations in the boxes. Unary addition is the process of coercing the contents of two boxes into one box. This problem is analyzed using quantum mechanics in [75], and it is demonstrated that limiting the bounding volume of an otherwise free quantum particle is analogous to compressing an ideal gas, independent of how the particle’s boundaries of travel are reduced (daemon or no). Hence, the quantum mechanical part of this mostly classical system of stones requires some consumption of energy.

This reasoning is applicable to electronic computing also. In an electronic computer electron clouds are moved across busses to be deposited in registers etc. To the extent that operations are not reversible, the computation will require the equivalent of compressing the electron cloud.

2.2 Higher Number Systems

In the system we have described, a represented number is a physical structure which can be created by *counting* the objects in a *set*. A number is an abstraction of the represented number. According to our example, a herd of animals forms a set, so one can speak of the *number* of animals out grazing, or equivalently, the *cardinality* of the herd. *Adding* the representations for the cardinality of individual sets produces a representation for the cardinality of the union of those sets. Hence, addition is useful in that it provides a short cut around having to form a union and counting the resultant elements.

Definition 1 (addition) *Addition is an operation which produces a representation for the cardinality of the union of sets from only the representations for the cardinality of the individual sets.*

Unary manipulation becomes unwieldy when the magnitude of numbers extends into the 100s. The inclusion of more counting symbols leads to a more powerful system. For example, symbols for groups of units reduces the number of necessary instantiations - in other words the amount of energy placed into the calculation. The Roman numeral system uses this approach by including the symbols **V**, **X**, **L**, **C** to stand for five, ten, fifty, and one hundred **I**s, where **I** is the unit symbol.

When performing operations such as subtracting **II** from **V**, the higher valued symbol is converted to unit equivalence **IIII**, and the unary rule for subtraction is applied to produce the result **III**. Alternatively, all the values for the addition and subtraction of interesting operands can be memorized (listed in a table). Table lookup is almost always used in multiplication.

The Roman representation, though usable for accounting into the tens of thousands, does not lend itself to scientific computation. The Arabic system contains an important structural adjunct not well developed in the Roman system: a count is recorded not as a single lumped ‘pile’ of instantiated symbols, but as an organized array of instantiated symbols occupying ‘digit positions’. The unit equivalence of an instantiated symbol is determined by its value, and by its ‘digit position’. This system has two advantages, it further reduces the amount of writing to the point that representing scientific numbers becomes feasible, and it allows counting to progress in an elegant domino fashion.

The mechanical odometer is a physical definition of the Arabic number system. For the benefit of those not familiar with the odometer, and don’t have one available to examine, we describe it here: An odometer is a series of wheels with the ten symbols, 0 through 9, printed on their circumference of each wheel. A window in the front of the odometer shows which symbol has been instantiated in each digit. The odometer may be incremented repeatedly by rotating the wheel at the far right a unit at a time until all of the symbols have been used once, then bumping the wheel to the left one place for a carry. After all of the symbols in the second wheel have been used, a third wheel is bumped up one unit, etc.

2.3 Ripple Carry Addition in the Arabic System

Two numbers coded on separate odometers may be added by coupling the odometer axles via meshing gears, and then turning the axle so that one of the odometers is turned back to zero; the other odometer will then read the sum, figure 2.1. This is the rotational equivalent to combining the stone piles. However, this method is slow. If the shafts were turned by hand at 10 r.p.m., adding two five digit numbers of all 9s would require about 17 hours.

One way to make an adder faster is to use ripple carry addition. Accordingly, the digit wheels are made into gears. The procedure is to first contact the unit gears and roll the units wheel on the first odometer back to zero - thus causing the units wheel on the second odometer to increase. The second odometer's unit wheel may roll past 9 causing a carry to ripple up the odometer in the usual way. Then, the 10s gears are placed into contact and the procedure and the axle is again rotated, perhaps causing a carry. This is repeated until all of the digits on the first odometer have been, one by one, rolled back to zero. Adding five digit numbers of 9s in this fashion requires five rotations, one for each pair of wheels. At 10 r.p.m. this is 30 seconds. The Mark I, which ran at 200 r.p.m., would, if it used ripple carry addition, add these numbers in 1.5 seconds. ENIAC, running at 100khz, or 600,000 r.p.m. , could add such numbers, using ripple carry addition, in 0.5 milliseconds.

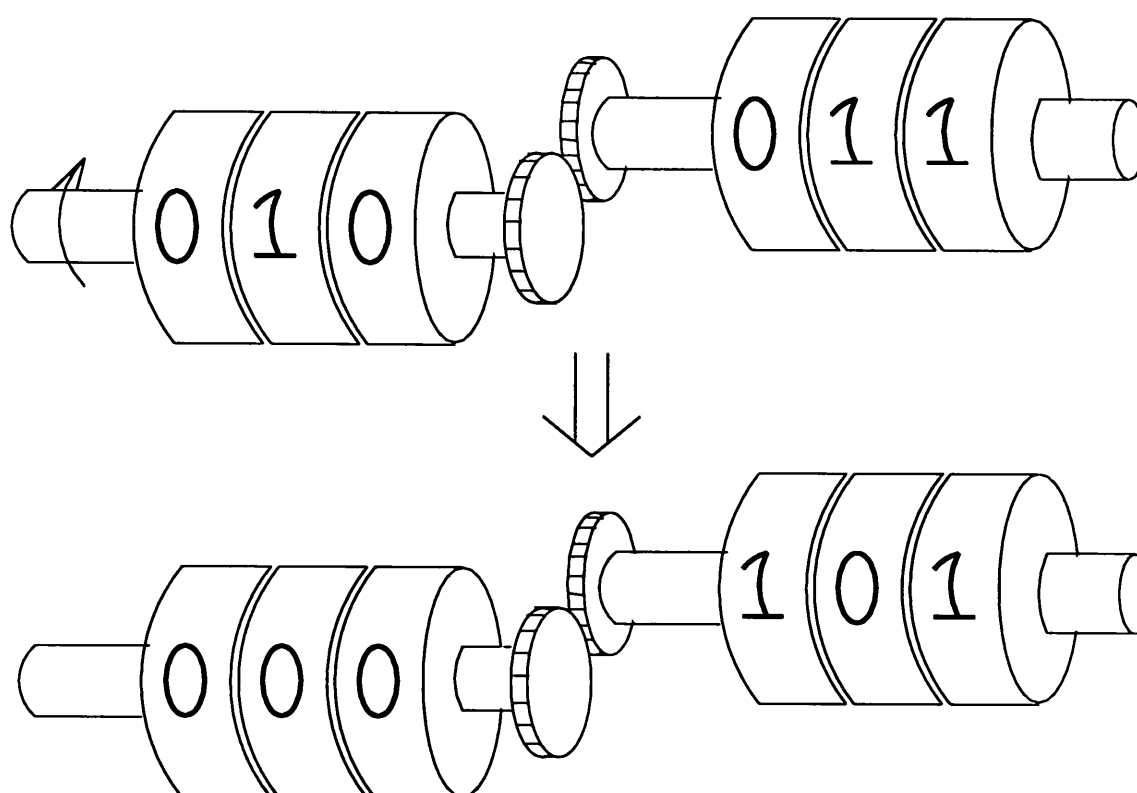


Figure 2.1: Adding with Odometers

Chapter 3

Conventional Addition Algorithms

This chapter looks at the conventional methods for adding in binary as though they are algorithms that can be followed step by step. The algorithms are specified such that each serial step contains a maximum number of parallel substeps. A central question will be the growth in the number of serial steps for increasing operand lengths, as this sets the performance limits for circuits. A circuit which implements an algorithm can do no better than perform all the given parallel substeps simultaneously while going forward one serial step after another.

3.1 Serial

Following the steps in the algorithm for serial addition generates the sum one bit at a time, starting from the least significant bit. This is the familiar hand method but done using the binary number system. This is also the *fast* algorithm for mechanical addition described in the previous chapter. This algorithm was described using Boolean algebra by Shannon [12]:

1. initialization
 - $s_0 = a_0 \vee b_0 \vee c_0$
 - $c_1 = \text{majority}(a_0, b_0, c_0)$
- 2, ..., N . calculate for bit $i = 1, 2, 3, \dots N - 1$
 - $s_i = a_i \vee b_i \vee c_i$
 - $c_{i+1} = \text{majority}(a_i, b_i, c_i)$
- $N+1$. set the last sum bit
 - $s_N = c_N$

Here, the input operands are of equal length, N , bits. Hence, $N - 1$ is the index of the most significant bit. The input operands are A and B, where a_i is the i th bit of A, etc. Each sum bit is s_i , while the carry-out is c_{i+1} .

Figure 3.1 shows the result of performing these three steps with two three bit operands. The operands are 3 (011) added to 2 (010) to yield 5 (101). The operands are in large bold face. The carry values are shown underneath the dotted line. The dotted line is used to signify that the carries are generated separately from the original operands, but that they are still added in as though they form a third operand. The solid line separates the sum from the carries. Addition proceeds from the right to the left.

The usual implementation of serial addition contains a parallel component since both sums and carries are generated simultaneously in each column.

$$\begin{array}{rcccc}
 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \\
 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \\
 \hline
 & & \text{2b} & \text{1b} & \\
 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \text{carries} \\
 \hline
 & \text{3} & \text{2a} & \text{1a} & \\
 & \mathbf{1} & \mathbf{0} & \mathbf{1} & \text{sums}
 \end{array}$$

Figure 3.1: Serial Addition

However, the algorithm is serial in the sense that the columns are processed sequentially starting from the least significant bit. The serial steps in figure 3.1 are numbered consecutively, while letters are used to distinguish among the steps that may occur in parallel or in any order. For example, step 1 shows that the carry into the second column (carry 1b) may be generated before, after, or at the same time as the sum in column one (sum 1a). The lettered cases will be performed simultaneously in maximally parallel implementations. In the worst case, there are $N + 1$ steps to perform, so this algorithm is order N . Although, $N = 3$ in 3.1 there 3 and not 4 steps because there is no carry-out.

$$O(\text{serial}(\text{any operands})) = N \quad (3.1)$$

3.2 Ripple Carry

Ripple carry addition is similar to serial addition. The principal difference is that ripple carry addition is asynchronous while serial addition is clocked. In ripple carry addition all of the domino style carry chains start simultaneously, as shown for the example case in figure 3.2. Because of the overlapping calculations, signals in the ripple carry adder can transition multiple times¹. The vacillation is denoted in the example with crossed out bits.

In the following algorithm, each δ_i denotes the time delay function through cell i . The ϵ denotes a small timing margin. This algorithm describes a continuous update cellular automaton. It is a precise description, but not very

¹At most one carry will pass through any bit.

intuitive, so we are lead to develop a more comprehensible replacement.

The ripple carry algorithm:

1. Stop after $\sum_0^N \delta_i + \epsilon$ time.
 - Continuously add the two least significant bits to produce a sum bit of the same significance, and a carry output bit of one greater significance. Post the results after time δ_0 .
 - Continuously, for bits $i = 1, 2, \dots, N - 1$, add the two operand bits and the carry bit of significance i , to produce a sum bit of the same significance, and a carry bit of one greater significance. Post results changes δ_i time after the input operand transition that caused the change.
 - Continuously set the last sum bit $s_N = c_N$. Post a completed result after $\delta_N + \epsilon$ time.

Although the ripple carry algorithm refers to continuous updates, the updates only propagate when they cause transitions. This is analogous to the moving object effects in cellular automata; however we want to use a simpler model than a cellular automata. Some signal propagations are not important because they are overwritten, or because they do not lead to an output. These unimportant paths are ‘transients’. We will focus on the non-transient paths in the remainder of this chapter.

Following the mostly the non-transient information path through the ripple

carry adder produces the following algorithm:

- Simultaneously add the three bits in each column without propagating carries. Sums which would later be rewritten can be skipped.
- Locate the beginning of all the carry strings. The carry strings start at the rightmost carry generation. Simultaneously, for each carry string, until reaching the end of the longest carry string:
 1. Add the bits and propagate the carry.
 2. Move to the left one bit.

Figure 3.2 shows the salient events for an example add: the generation of two carries, the carries propagating across the adder, and then later one path running over the other in the upper four bits. More specifically, when assuming that the adder starts with all zero carries, the first step in the algorithm produces the sums in columns 0 and 3, and the carries for columns 1 and 4. The sum calculations for the other columns are later rewritten. The second step produces the results for column 1 and 4, etc. Carry propagation from step 3 is used to recalculate the carry into column 3. This makes the previous column 3 calculation invalid, so it must be redone, which causes a new carry to be propagated into column 4. Re-doing the column 4 calculation produces a different

5	4	3	2	1	0 columns
0	1	1	1	1	1
0	1	0	1	1	0
<hr/>					
^{2d} 1	^{1d 4b} 0 1	^{3b} 0 1	^{2b} 1	^{1b} 0	0 carries
³ 1	^{2c 5a} 0 1	^{1c 4a} 1 0	^{3a} 1	^{2a} 0	^{1a} 1 sums

Figure 3.2: Ripple Carry Addition

sum bit, but does not produce a different carry bit, so the algorithm finishes after 5 steps. Note that the correct most significant sum bit was produced in 3 steps, yet the addition was not finished for two more steps because of the changes in lower significant bits.

3.3 Carry Select

This method is a *divide and conquer* strategy which hedges bets on the outcome of intermediate carries. The length of ripple propagate chains must be reduced somehow in order to speed up addition. If we assume at any particular point in the adder that the carry would always be zero, then the adder could be broken at that point into separate parallel processes, and the sum could be completed sooner. However, the assumption would be wrong 50% of the time.

We can hedge bets on the outcome of such a carry calculation by starting two upper half adds, each with a different assumption about the value of the particular carry-out. Figure 3.3 shows an example of adding 01011111 to 01010110. The upper four bits of the add are duplicated, the top version has the carry input bit set, while the bottom version does not. In the first serial step six items are calculated simultaneously. These include the first column sum bits in each of the three additions, and the first column carry-out bits. The three additions continue on independently as in ripple carry addition until the carry emanating from the least significant four bits causes the selection of the correct upper four bits, 1011 in the last step. This carry select addition requires only five serially dependent steps to add 8 bit operands. Ripple carry addition of the same length operands would require 8 serially dependent steps.

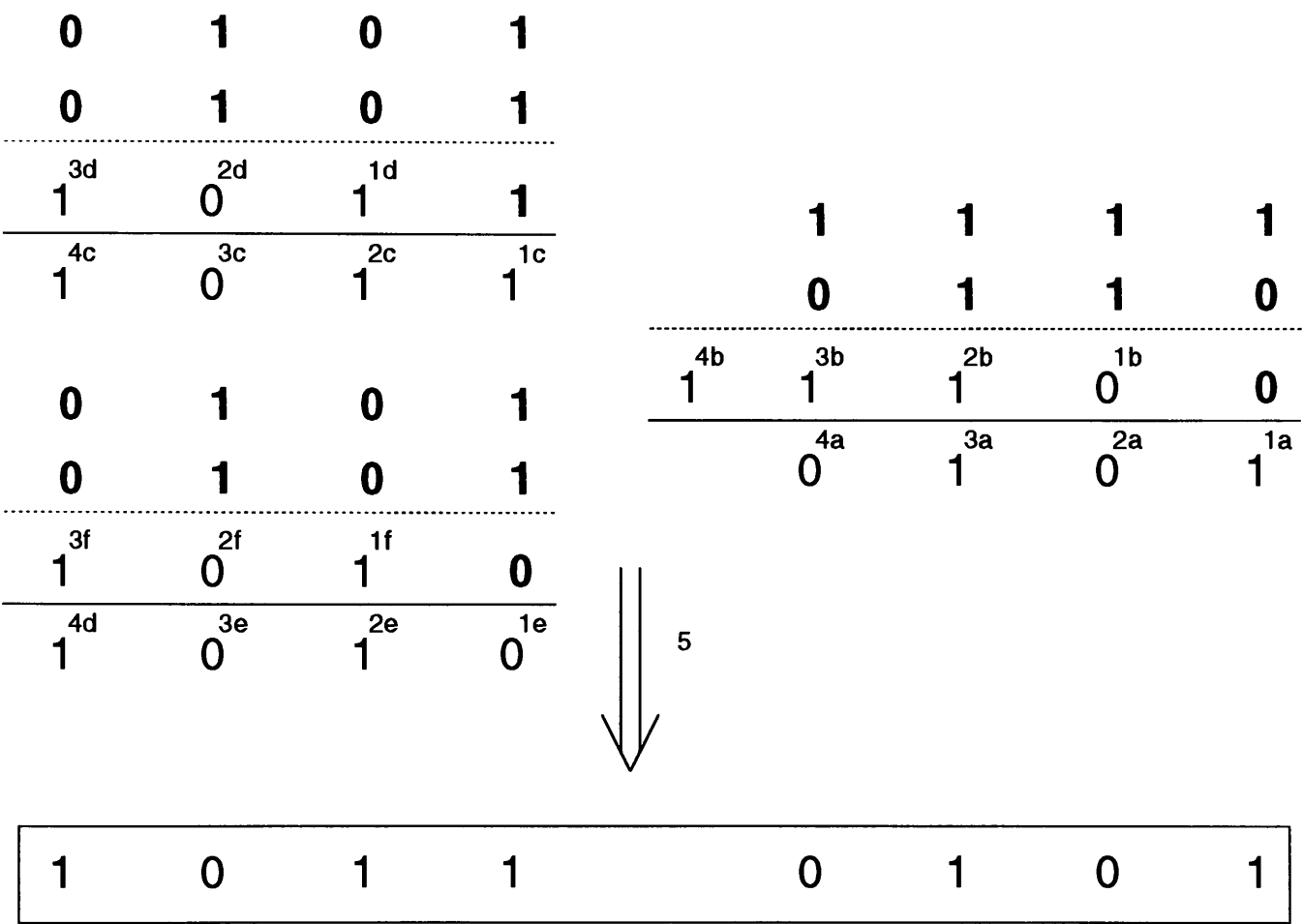


Figure 3.3: Carry Select Addition

The carry select algorithm is then:

1. perform three ripple carry adds simultaneously:
 - (a) One add with the lower half bits.
 - (b) Two adds with the upper half bits.
 - i. One with the carry-in set to one.
 - ii. One with the carry-in set to zero.
2. Pick the correct upper half.

Although the number of series steps for performing this type of addition is approximately half the number for ripple carry addition, constant factors do not change the order of growth, and the performance is still linear:

$$O(\text{Carry Select}(\text{worst_operands})) = N \quad (3.2)$$

3.4 Ripple Carry Select

Another conventional algorithm for addition is made by breaking ripple carry addition into a larger number of pieces than the two pieces of the carry select method. For example, the 8 bit addition from the previous example can be broken into four 2 bit pieces, as shown in figure 3.4. Each of the 2 bit pieces contains duplicated 2 bit adders: one copy adds with an assumed carry-in of one, and the other with an assumed carry-in of zero. The correct pieces are selected in ripple fashion: the bottom 2 bit add selects among the correct second section add, and the correct second section add then provides the correct carry for the third section, which provides the correct carry for the fourth section. This adder still has order N performance.

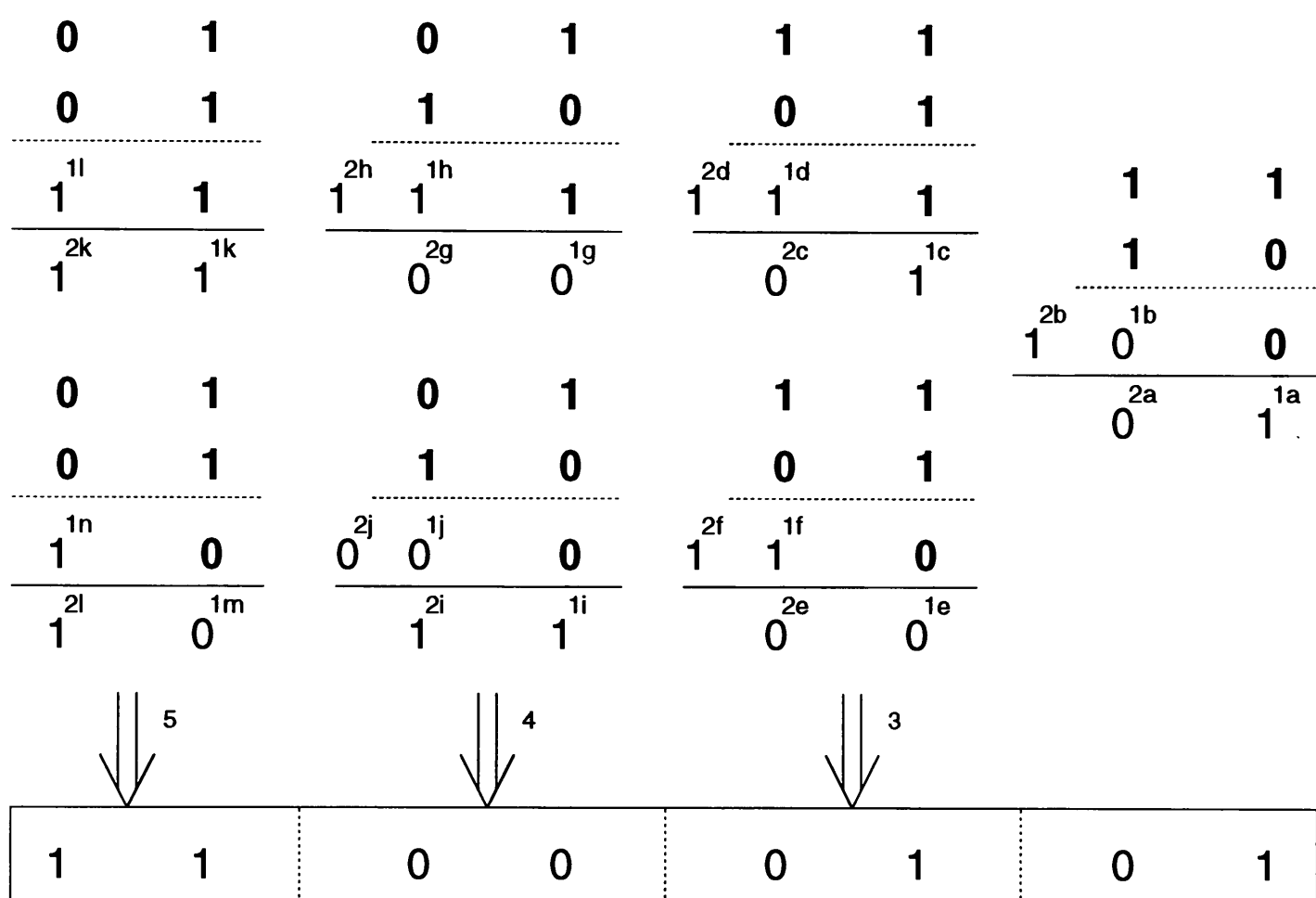


Figure 3.4: A Ripple Carry Select Addition

3.5 Conditional Sum

The previous section showed how the carry select algorithm can be applied in a series to produce a ripple carry select hybrid add to reduce the number of serially dependent steps. However this reduction did not change the order of the number of serially dependent steps (measured against operand width). This section explains how the carry select algorithm can be applied recursively to produce a tree configuration with a logarithmic number of serially dependent steps.

Figure 3.5 shows an example of conditional sum addition. This example and the one from the previous section differ on the third step. In this new adder, the first section carry still selects the correct sum and carry from the second section, but now the third section also picks among the fourth section sums. There are four possible combinations of results from the combined third and fourth sections: the top sum of the fourth section concatenated to the top sum of the third section, the top of the fourth section concatenated to the bottom of the third section, the bottom of the fourth section concatenated to the top of the third section, and the two bottoms concatenated together. However, after the second step in the algorithm the carry outputs from the two third section adds are known, so it is possible to narrow the four possible upper half sums to two possibilities. These two possibilities are called conditional sums; they are written down as part of the third step. On the fourth step the carry from the bottom two sections, which is shown at the upper left corner of the box, is used to select among the two possible upper half adds to form the completed sum.

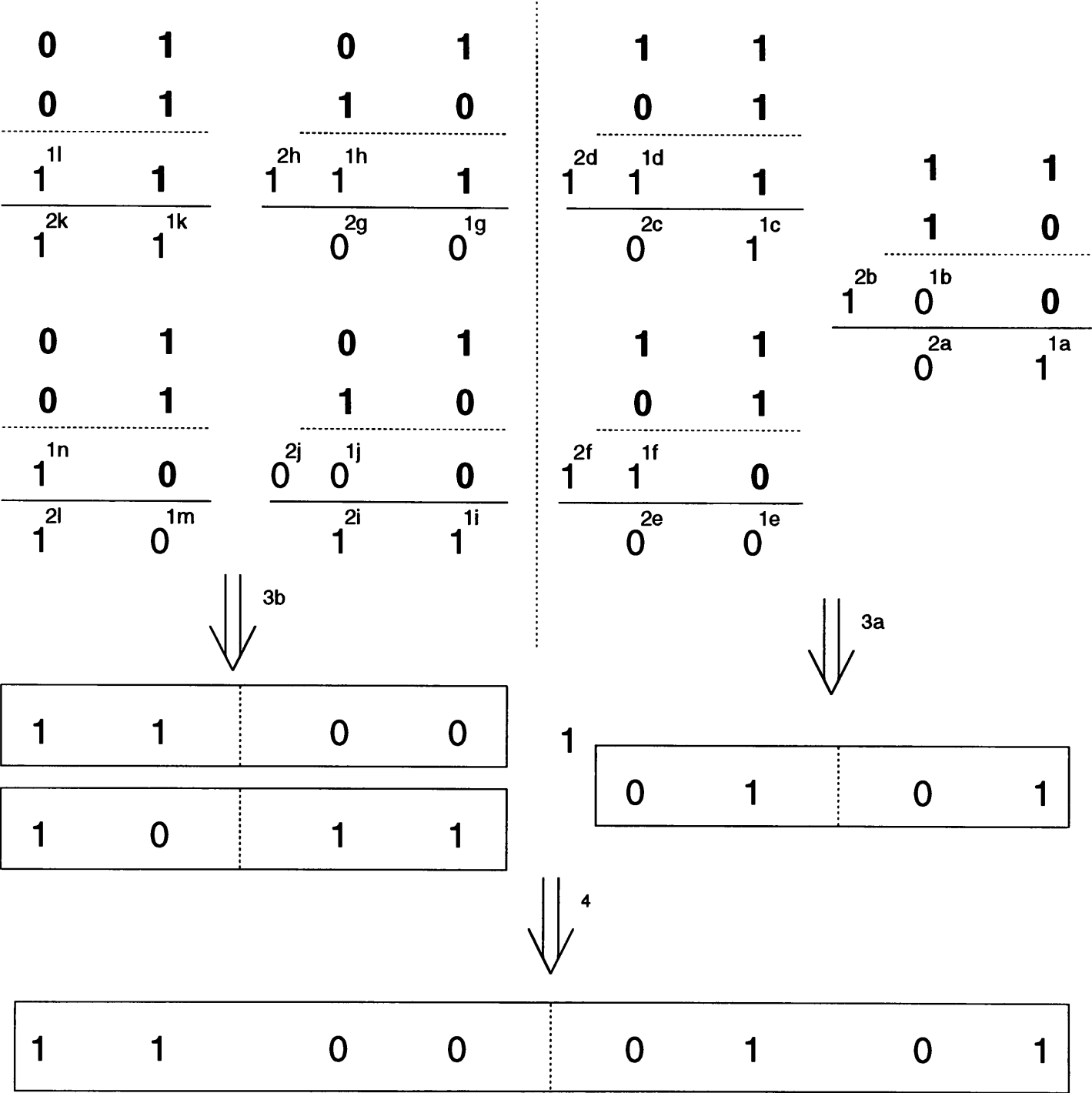


Figure 3.5: Recursive Carry Select Addition – Conditional Sum

Conventionally the application of carry select is repeated until only one bit adds are left, instead of the two bit adds as we showed in the example. The conditional sum algorithm is:

1. Break the add in half resulting in three adds half the size and a multiplexer.
2. perform the addition for each of the three adders in one of two ways.
 - (a) if the add is larger than one bit, recurse by going back to step one.
 - (b) if the add is one bit wide, then use a half adder to get the sum and carry.
3. finish the sum for the three subadder group by performing the multiplex operation.

There are two serial steps per level of recursion in this algorithm. At each level of recursion the problem is broken in half, so the order of performance is:

$$O(\text{Conditional Sum}(\text{worst_operands})) = \log N \quad (3.3)$$

3.6 Propagate Generate Class of Adders

The method of propagate and generate partitions an addition into three serially dependent steps:

1. Calculate two Boolean vectors called propagate (p) and generate (g) from the input operands by bitwise *exclusive-or*-ing for propagate, and bitwise *and*-ing for generate.
2. Calculate the carries (c) from the propagate and generate vectors.

3. Given the propagate vector and the carry vector calculate the sum vector (s).

The logic equations for propagate and generate add are:

For all i between 0 and $N - 1$ inclusive:

$$g_i = a_i b_i \quad (3.4)$$

$$p_i = a_i \vee b_i \quad (3.5)$$

$$c_{i+1} = \text{fco}(< g_i, p_i >, c_i) \quad (3.6)$$

$$(a + b)_i = c_i \vee p_i \quad (3.7)$$

Where $(a + b)_i = s_i$.

The concept embedded in these equations is the basis for a large number of fast and efficient adders. In the first equation, the operands are transformed into propagate and generate vectors by component-wise *exclusive-or*-ing and *and*-ing. In the third equation, the fco function produces a carry from propagate generate signals and a carry input. The fourth equation shows how the carry together with propagate can be used to produce a sum bit. The carry operation is defined as:

$$\text{fco}(< g_i, p_i >, c_i) \equiv c_i p_i \vee g_i \leftrightarrow c_{i+1} \quad (3.8)$$

The first and third steps in the algorithm are constant time operations. Only the carry evaluation step has operand dependent timing. From this point of view, the carry select and conditional sum algorithms contain unnecessary steps

for partitioning the calculation of sums. It is not the sum calculations which need to be accelerated, it is the carry calculations.

Bit propagate can also be defined with the *or* of the operands instead of *exclusive-or*. The *exclusive-or* version of propagate is false when a carry is generated. An *or* version would be true when a carry is generated. Either way a carry appears at the output of the column, so the behavior of the adder is the same. The next set of equations are based on the assumption that a carry is propagated when one is generated. In order to keep the two types of propagates distinct we will call this *or* version ‘transfer’, t .

For all i between 0 and $N - 1$ inclusive:

$$g_i = a_i b_i \quad (3.9)$$

$$t_i = a_i \vee b_i \quad (3.10)$$

$$c_{i+1} = \text{fco}(< g_i, t_i >, c_i) \quad (3.11)$$

$$(a + b)_i = c_i \vee a_i \vee b_i \quad (3.12)$$

$$\text{fco}(< g_i, t_i >, c_i) \equiv c_i t_i \vee g_i \leftrightarrow c_{i+1} \quad (3.13)$$

3.6.1 Propagate Generate Ripple Carry

1. Form the propagate vector by component-wise *exclusive-or*-ing the input operands, and form the generate vector by component-wise *and*-ing.
2. Starting at the right most one in the generate vector perform the fco function by:
 - (a) Move to the next column and set the carry to one
 - (b) If there is a one in the propagate vector for this column, go back to the previous step and repeat.

Continue scanning to the left looking for set generate bits, if one is found repeat this step.

3. Copy the propagate vector into the sum row, *exclusive-or* all bits which have a carry above them.

Figure 3.6 shows a propagate generate style ripple carry adder. In the first step the propagate and generate vectors are formed. In the second step, a one is placed in the carry vector for column 2. Since the propagation bit is set, another carry is placed in column 3, again the propagation bit above this carry is set, so a carry is placed in column 4. This time the propagation bit in column 4 is not set, so the carry is not propagated into column 5. Scanning to the left from column 4 there are no more generate bits set, so the algorithm proceeds to the final step where the sum is formed by copying the propagate vector into the sum column with the bits 2,3, and 4 flipped.

The propagate signal is set in a bit column when a carry into that column causes a carry output. The generate signal is true for a column when a carry output leaves the column independent of a carry in. There is a third possibility, and that is the case where no carry leaves a particular column independent of the carry-in. This is the case of ‘carry kill’. Carry kill is identical to ‘not

5	4	3	2	1	0	columns
1	0	1	1	1	1	
0	0	0	0	1	0	operands
^{1l} 0	^{1k} 0	^{1j} 0	¹ⁱ 0	^{1h} 1	^{1g} 0	generates
^{1f} 1	^{1e} 0	^{1d} 1	^{1c} 1	^{1b} 0	^{1a} 1	propagates
<hr/>						
	⁴ 1	³ 1	² 1			carries
^{5f} 1	^{5e} 1	^{5d} 0	^{5c} 0	^{5b} 0	^{5a} 1	sums

Figure 3.6: A Propagate Generate Version of Ripple Carry Addition

transfer', or $\bar{p} \wedge \bar{g}$. The propagate, generate, and kill bits are only functions of the two operand bits with the same significance (and not of the carries), so they can be calculated in constant time.

The carry strings will always start one column to the left of a generate and continue leftwards until a column with a carry kill condition is found. This observation can be used to write the carry strings down directly by inspection. Hence, propagate generate ripple carry addition is a useful method for adding long binary numbers by hand. Note that the performance of this method of addition is still linear.

$$c_{i+1} = \text{fco}(< g_i, p_i >, c_i) \quad (3.14)$$

This implies that the fco function is to perform:

$$c_{i+1} = g_i \vee p_i c_i \quad (3.15)$$

The same applies for ripple carry addition based on transfer:

$$c_{i+1} = g_i \vee t_i c_i \quad (3.16)$$

An interesting property of the transfer equation is that:

$$(t_i = 0) \rightarrow (g_i = 0) \quad (3.17)$$

A transfer bit equal to zero implies that the corresponding generate bit is zero. This follows since transfer is the *or* of the input operand bits, and generate is the *and* of the same bits. This implication does not hold for the propagate formulation. From this implication we can deduce:

$$c_{i+1} = g_i \overline{c_i} \vee t_i c_i \quad (3.18)$$

This is the 2:1 multiplexor equation. Hence transfer-generate ripple carry addition can optionally be performed using multiplexers instead of the usual **and-or** gate combination:

1. Form the propagate vector by component-wise *exclusive-or*-ing the input operands, and form the generate vector by component-wise *and*-ing.
2. Starting at the right most one in the generate vector perform the fco function by:
 - (a) Move to the next column and set the carry to one.
 - (b) Copy the transfer bit as the carry into the next column. Move to the next column, if the carry is set, go back to the previous step and repeat.

Continue scanning to the left looking for set generate bits, if one is found repeat this step.

3. *exclusive-or* the operand bits and the carries in each column to form the sum bits.

Figure 3.7 shows the previous example using transfers in place of propagates.

5	4	3	2	1	0	columns
1	0	1	1	1	1	
0	0	0	0	1	0	operands
^{1l} 0	^{1k} 0	^{1j} 0	¹ⁱ 0	^{1h} 1	^{1g} 0	generates
^{1f} 1	^{1e} 0	^{1d} 1	^{1c} 1	^{1b} 1	^{1a} 1	transfers
<hr/>						
	⁴ 1	³ 1	² 1			carries
^{5f} 1	^{5e} 1	^{5d} 0	^{5c} 0	^{5b} 0	^{5a} 1	sums

Figure 3.7: A Transfer Generate Version of Ripple Carry Addition

The complete equation for the carry leaving the i th bit of a ripple carry add can be found by applying equation 3.16 once per bit position. If the first column of the ripple carry add is j , and the last column is i , then:

$$c_{i+1} = g_i \vee t_i c_i \quad (3.19)$$

$$c_i = g_{i-1} \vee t_{i-1} c_{i-1} \quad (3.20)$$

...

$$c_{j+1} = g_j \vee t_j c_j \quad (3.21)$$

By recursively substituting for the carry we obtain:

$$c_{i+1} = g_i \vee t_i g_{i-1} \vee t_i t_{i-1} g_{i-2} \vee \cdots \vee t_i t_{i-1} \cdots t_j c_j \quad (3.22)$$

Since equation 3.16 has the same form as equation 3.15, this last result applies to the propagate form also:

$$c_{i+1} = g_i \vee p_i g_{i-1} \vee p_i p_{i-1} g_{i-2} \vee \cdots \vee p_i p_{i-1} \cdots p_j c_j \quad (3.23)$$

For example, the carry-out of a 4 bit wide ripple carry addition extending from bit position 0 to bit position 3 is described by:

$$c_4 = g_3 \vee t_3 g_2 \vee t_3 t_2 g_1 \vee t_3 t_2 t_1 g_0 \vee t_3 t_2 t_1 t_0 c_0 \quad (3.24)$$

3.6.2 Carry Skip

Since only the carry logic needs to be duplicated for performance, repeating the calculation of sum bits in the redundant blocks of a carry select adder is not efficient. A carry skip adder can be obtained by moving the summation logic out of the parallel part of the algorithm, and then calculating sums serially after the carries are obtained. After the sum logic is deleted from the ripple carry select, each of the blocks only produces two conditional carries, except the bottom block, which produces an actual carry.

The carry-out from the bottom block selects from the two conditional carry-outs from the second block, and these select among the conditional carries from the third block, etc. The carry select adder uses multiplexers, so the correct signal to use is *transfer*, as shown in equation (3.18). The multiplexers are gone from the sum path in the carry skip adder. Also, the carry propagation optionally can be done with an **and** / **or** gate combination instead of multiplexer. In this later case, *propagate* could be used in place of *transfer*.

Two carry calculations are still required per section: one while assuming a carry-in of one, and one while assuming a carry-in of zero. However, the carry

calculations are easier to perform than complete additions. Since the calculation of carries for the ripple carry adder example in the previous section did not use the sum values, that method can be used to find the carry conditioned on a zero carry input. The carry output for the case of a carry input of one is found simply by *and*-ing all of the carry transfers for the block. Hence, the equations for the conditional carries are:

$$c_{i+1}^0 = g_i \vee t_i g_{i-1} \vee t_i t_{i-1} g_{i-2} \vee \cdots \vee t_i t_{i-1} \cdots t_{j+1} g_j \quad (3.25)$$

$$c_{i+1}^1 = t_i t_{i-1} \cdots t_{j+1} t_j \quad (3.26)$$

Here the superscripts 0 and 1 stand for the case of a carry-in of zero and of one respectively.

A carry conditioned on a carry-in of zero is the same as group generate, while carry conditioned on a carry-in of one is the same as group transfer.

$$\langle g_{i:j}, t_{i:j} \rangle = \langle c_{i+1}^0, c_{i+1}^1 \rangle \quad (3.27)$$

The algorithm for carry skip addition follows that for ripple carry select,

except the formation of the sums is left until the end:

1. for each block perform both steps simultaneously:
 - (a) Form the zero conditional carry by (group generate) calculating the carry-out according to the ripple carry algorithm while holding the carry-in at zero.
 - (b) Form the one conditional carry by (group transfer) conjunct-ing (*and-ing*) all of the transfer bits together.
2. Take the carry-out from the bottom block, and select the correct carry-out for the next block, continue until the top block has a correct carry-in.
3. Perform ripple carry additions in each block to form the final sums.

Figure 3.8 shows an 8 bit carry skip add composed of four two bit blocks. The first two rows show the operand bits. The third two rows show the conditional carries derived from the ripple carry algorithm. The next row shows the actual block carry outputs. The 0 conditional carry output of the bottom block is an actual carry so this value is just copied from an higher row. The other block outputs are selected in series. The next to last row shows the intermediate carries for the blocks, which are produced in ripple carry fashion. The bottom row shows the sums which are found by column-wise *exclusive-or-ing* the two operand bits and the carry bit.

The level of performance of this algorithm is linear since the carries ripple serially across N/n blocks, where n is the constant number of bits per block.²

²We use the variable N for adder widths. Small adders can be arrayed to make larger adders; hence an adder of width n_i is a member of such an array. Conventionally block widths are signified with k instead of n .

0	1	0	1	1	1	1	1	A bits
0	1	1	0	0	1	1	0	B bits
^{1h} 0	^{1g} 1	^{1f} 0	^{1e} 0	^{1d} 0	^{1c} 1	^{1b} 1	^{1a} 0	generates
¹ⁱ 0	^{1j} 1	^{1k} 1	^{1l} 1	^{1m} 1	¹ⁿ 1	^{1o} 1	^{1p} 1	transfers
		^{2c} 0	^{3a} 1		^{2b} 1	conditional carries 0		
		^{2e} 1	^{2d} 1		conditional carries 1			
		^{5a} 1	^{4a} 1		^{2a} 1	actual block carry outs		
^{7a} 1	^{6a} 1	^{6b} 1	^{5b} 1	^{4b} 1	^{3b} 1	^{2f} 0	^{1l} 0	carries within block
⁸ 1	^{7c} 1	^{7b} 0	^{6c} 0	^{5c} 0	^{4c} 1	^{3c} 0	^{1l} 1	

Figure 3.8: Carry Skip Addition

$$O(\text{carry_skip}(\text{worst_operands})) = N \quad (3.28)$$

3.6.3 Carry Lookahead

Carry Lookahead addition is performed by following these steps:

1. Calculate bit generates and propagates/transfers.
2. Evaluate each carry equation in parallel.
3. Use the carries and propagates to calculate the sums.

Carry lookahead is based on the fact that Boolean logic equations can always be evaluated in two stages of logic.

$$c_1 = g_0 + p_0 c_0 \quad (3.29)$$

$$c_2 = g_1 + p_1 g_0 + p_{1:0} c_0 \quad (3.30)$$

$$c_3 = g_2 + p_2 g_1 + p_{2:1} g_0 + p_{2:0} c_0 \quad (3.31)$$

$$\dots \quad (3.32)$$

$$O(\text{carry_lookahead}(\text{worst_operands})) = C \quad (3.33)$$

Where C is a constant.

Such calculations based on two level **and-or** evaluation with unlimited fan-in/fan-out ignore several real effects. This isn't the issue however, as all adder models which show less than linear order evaluation time are ignoring real effects³. The issue is whether the model appropriately predicts dominate device effects for the size of problem of interest. In the case of the carry lookahead adder, this is only the case for N of a few bits.

³the very propagation of information over a distance is at best linear time.

Chapter 4

Gate Delay Models for the Conventional Adders

...during my study of the adder unit I got the idea of solving virtually all statements - today we speak of data or information - with yes/no values. We realized that this principle could be applied to all computing machine components, especially to the control device, and led to switching algebra with the aid of propositional calculus. – *K. Zuse.*

In this chapter we show logical gate implementations of algorithms discussed in the previous chapter. After obtaining a net list for an algorithm, we derive the worst case path lengths through the net lists¹.

A gate level net list is often an ASCII file which contains a list of all the gate instances with their input and output pins labeled, along with a list of the named connections (nets) between the pins. Various attributes of the circuit can be attached to the nodes, pins, and nets, so the net list typically holds physical circuit information, such as device sizes, connection lengths, and even parasitics. The connection information in the net list can be interpreted formally as a graph. Hence, the net list of an adder is the nexus of its logical topology, gate implementation, and physical parameters. Apparently the net list is the appropriate place to start our study of adder implementations.

¹By gate level we mean that all of the instantiated objects in the net list are macros for the common gates such as **nand** and **nor**. Such macros would be expanded out to the transistor implementation for the gates when needed.

According to the graph interpretation, a gate delay count is simply the length of a path through the net list.² Because net lists are computer based representations, functions for describing net list properties such as path lengths can be implemented as computer programs. The functions in this chapter have a syntax akin to a procedure call and accept as input or produce as output objects such as net lists, nets, pins, and numbers.

In general, the longest path through a net list can not be assumed to be the slowest to evaluate since evaluation time is also a function of the device physics and geometries involved, but in the case of the conventional adder circuits, it is a reasonable heuristic. It was apparent from the previous chapter that the rate limiting sequence of steps is the propagation of the carries. Propagate, generate, and sum formation were simple constant time operations. In general carry path lengths are not directly proportional to actual evaluation times, since gate delays are not all the same.

The approach of looking at path lengths for determining evaluation time can be improved upon by using device sizes and loads along with a simple timing model. This refinement is necessary when optimizing multiple carry paths. To obtain reasonably accurate estimates of the worst case evaluation time, the slowest among the paths recognized by such an improved timing model³ should be extracted from layout with parasitics, and simulated with SPICE. The worst result obtained from SPICE is typically an acceptable worst case

²In actuality, gate delay counts are expressions of path lengths through the dual graph of the net list, since counts are of the nodes crossed instead of arcs traversed.

³Of course there must exist a combination of inputs which activate the paths.

evaluation time estimate.

Note, there are five distinct usages of *and* in this thesis (the other logical operators are analogous). There is the grammatical version in text, ‘and’. The propositional logic operator, ‘ \wedge ’. The word for referring to the operation, ‘*and*’. The word for referring to the gate which performs this operation ‘and’. Finally, the name of the macro function called out in the net list, ‘AND’. All of these forms carry unique information. We have endeavored to use them consistently; however there were some ambiguous situations. For example, when a schematic was used to show how a net list was implemented.

4.1 Ripple Carry

Figure 4.1 shows three variations on the logic for creating transfer and generate, or propagate and generate signals. **xor** gates cause two delays in the worst case, according to the implementations shown in Figure 4.2. The variations in Figure 4.1 are all based on:

$$a \nabla b = (a \vee b) \overline{(ab)} \quad (4.1)$$

Figure 4.3 shows the implementation of two and three input NOR and NAND macros along with the three input OR-NAND, and AND-NOR macros. In general, gates with higher fan-in are slower. The delay is roughly related to the number of series transistors between the output and the power supply rails; this measure is known as the “stack height”.

Hence, the performance of the OR-NAND and AND-NOR is comparable

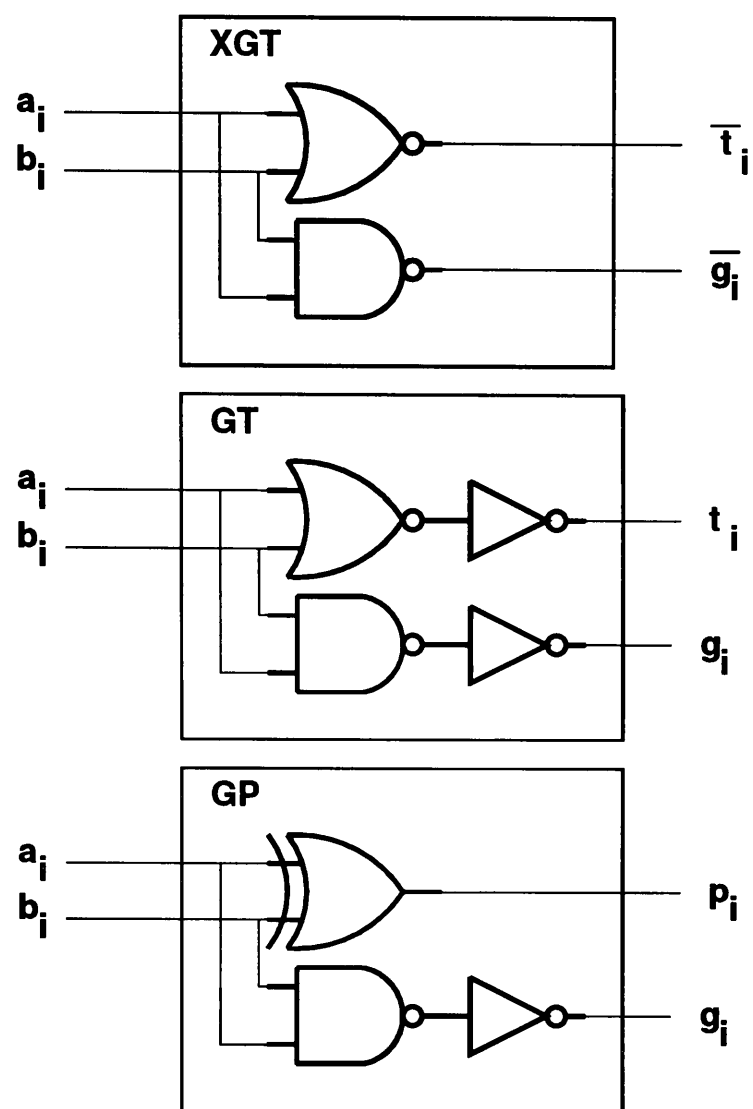


Figure 4.1: Generate and Propagate Logic Blocks

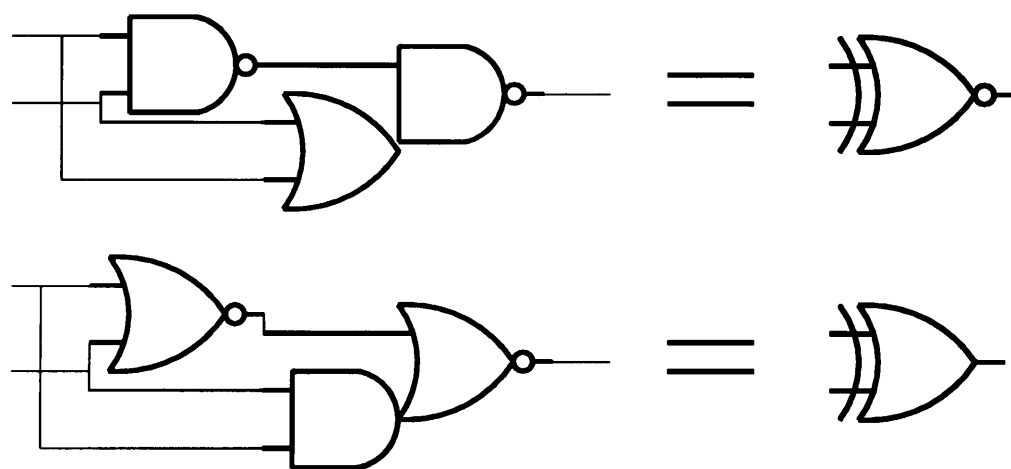


Figure 4.2: XOR and XNOR Blocks

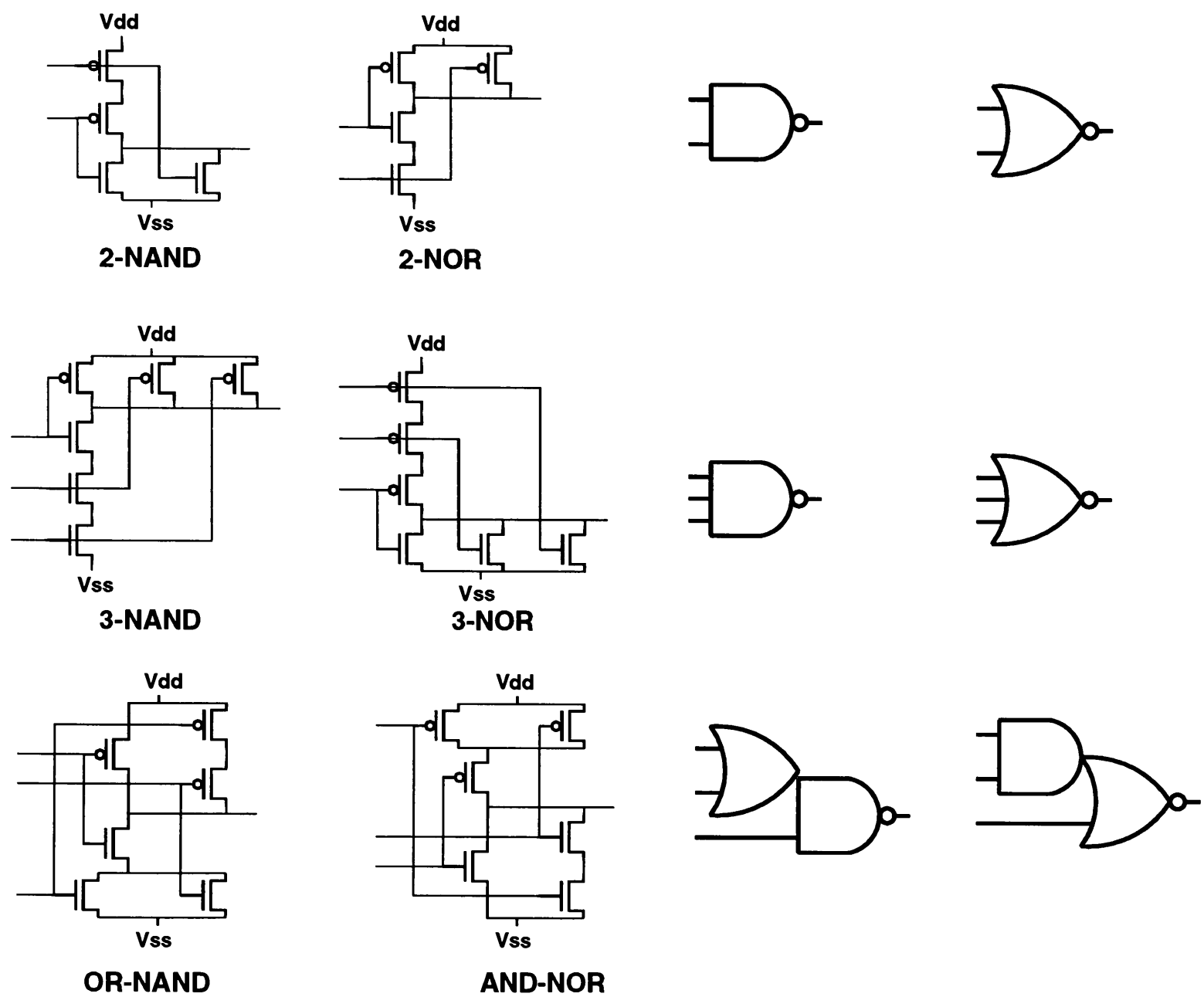


Figure 4.3: Comparison between OR-NAND, AND-NOR and Simple Gates

to a 2-NOR for falling edges, and comparable to a 2-NAND for rising edges, except that they will have more internal capacitance. In [76] a simplification of the AND-NOR macro is shown to switch faster. The macro shown turns out to be equivalent to using the Ling recursion (see chapter 8). Also, this observation on stack heights suggests that the three input **nor** based recursion used by Majerski [77] might be disadvantageous when implemented in CMOS.

In this chapter, the AND-NOR and OR-NAND macros will be counted as single gate delays.

Figure 4.4 shows a 4 bit ripple carry adder. (A clearly portrayed example implementation of such a ripple carry adder is in given in [78].) The blocks on the left transform the a and b inputs into t and g inputs. The block labeled ‘ $R(4)$ ’ then propagates the carries. The carries are shown leaving at the right. The **xor** logic is not shown.

Figure 4.5 shows variations of three input **xor** gates implemented efficiently by using the t and g signals instead of the operand bits. These variations were created using DeMorgan’s law and the following equation:

$$a \vee b = t\bar{g} \quad (4.2)$$

‘ R ’ is a special function which returns the net list of a ripple carry path. A net list contains a list of the blocks, pins, and nets used in a circuit. Blocks contain logical functions, pins are the terminals on the blocks, and nets are the connections between the pins. Figure 4.6 shows the net list of two series inverters with input A , an unnamed intermediate node, and output C :

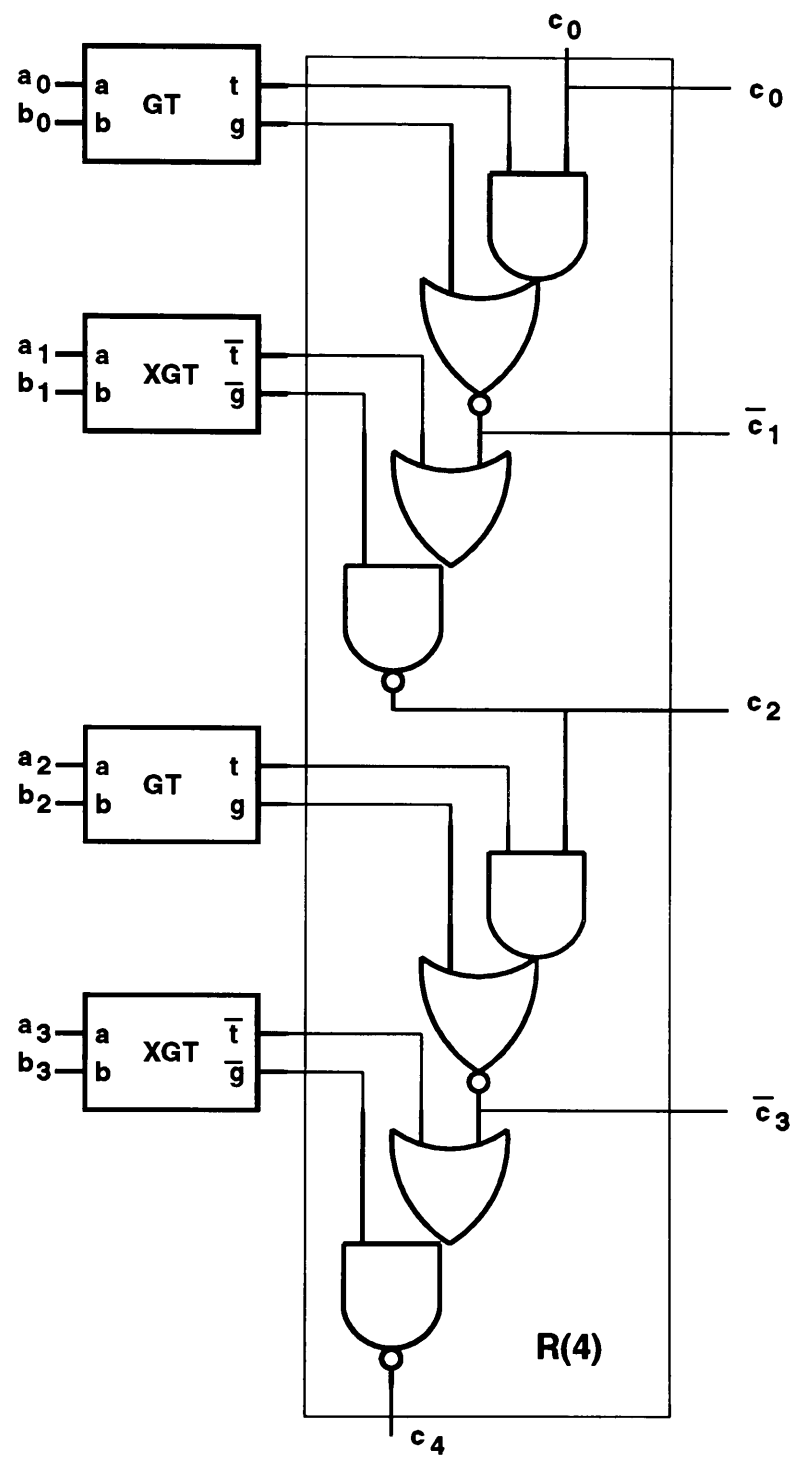


Figure 4.4: 4 Bit Ripple Carry

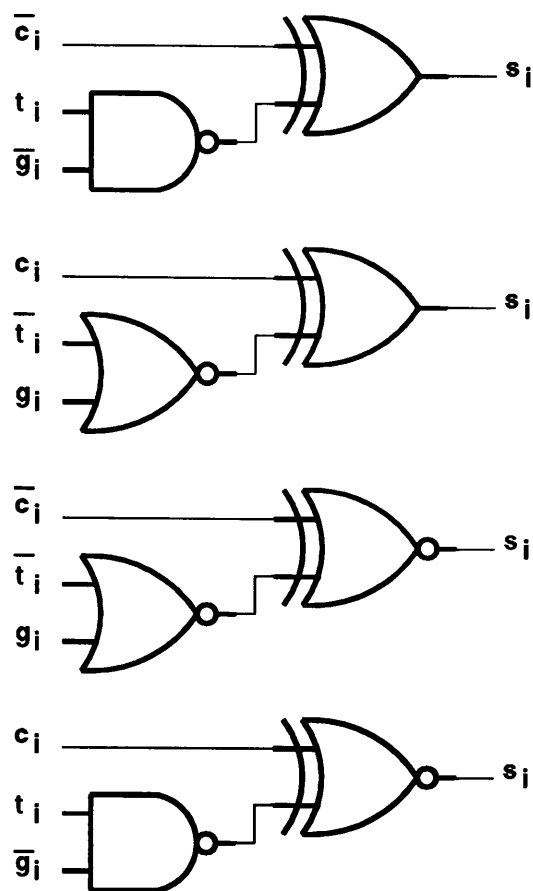


Figure 4.5: 3 Bit xor Gates

Functions can be used to manipulate or to return information about net lists. For example, the ‘D’ function returns the path length of the longest path between nodes.⁴ When applied to the ‘InvInv’ example above it yields:

$$D(\text{InvInv}, A, C) = 2 \quad (4.3)$$

This equation says that starting from the A pin of the InvInv block and traversing over the longest path to the C pin, one passes through two gates. It is easy to imagine how such a gate counting program could be written by using conventional graph traversal algorithms, and in fact popular CAD programs

⁴Sometimes it returns the longest ‘interesting’ path. Defining ‘interesting’ is one of the bothersome problems in designing such tools. Certainly the ability to turn the path on makes the path more interesting.

1. block declarations:

(a) inverter

i. pins:

A. I

B. O

ii. function: $O = \text{not } I$

2. blocks:

(a) $\text{inv1} = \text{inverter}$

(b) $\text{inv2} = \text{inverter}$

3. nets:

(a) $\text{net1} = (A, \text{inv1.I})$

(b) $\text{net2} = (\text{inv1.O}, \text{inv2.I})$

(c) $\text{net3} = (\text{inv2.I}, C)$

Figure 4.6: Net List of Two Series Inverters

typically have such a facility. Often a list of the nodes in the path is also returned so that the path can be inspected and simulated in more detail.

The three input **xor** is made from a **nand** or **nor** gate and a two input **xor** as shown in Figure 4.5.

$$D(\text{XOR3}, C, S) = 2 \quad (4.4)$$

$$D(\text{XOR3}, T, S) = 3 \quad (4.5)$$

The net list generating function ' R ' accepts one operand: the adder width. This is why the ripple carry block in Figure 4.4 is labeled $R(4)$. The resulting net list always contains one carry-in pin, N t input pins, N g input pins, N sum output pins, and one carry-out pin. R , GT , XGT , and GP are all net list generating functions. Net list generating functions are also called 'macros'. Note the R function is conceptually different from the fco function discussed earlier. The fco function is a Boolean function which operates on ordered pairs of Boolean values; the implementation of the fco function may vary between adder designs. In contrast, the R function returns a net list for a specific implementation of the fco functions; namely the ripple carry implementation under consideration.

$$R(2)(c_0, G_{1:0}, T_{1:0}) \rightarrow C_{2:0} \quad (4.6)$$

Equation 4.6 states that a 2 bit ripple carry adder is created, and that the signals $c_0, G_{1:0}, T_{1:0}$ are provided as input, and after the circuit evaluates, the output is comprised of the signals $C_{2:0}$. Here the capital letters are used in the classical way to indicate vectors. Whereas $g_{1:0}$ represents one signal which

contains summary information over the bits zero and one, $G_{1:0}$ is a vector of two signals, $\langle g_0, g_1 \rangle$.

The delay from carry input to carry output of a ripple carry adder is:

$$D(R(N), c_0, c_N) = N \quad (4.7)$$

When the critical delay path starts at the operand input instead of the carry input, an extra two gate delays are required to go through the GT block.

$$D(\text{ripple}, a_0, c_N) = D(R(N), p_0, c_N) + D(\text{ripple}, a_0, p_0) = N + 2 \quad (4.8)$$

It may be important to know the time delay (path length) from introduction of the carry-in, or the operands, to the emergence of the most significant sum bit. The carry used in creating the last sum bit appears one gate delay before the carry-out:

$$D(R(N), c_0, c_{N-1}) = N - 1 \quad (4.9)$$

After this carry is setup, the isolating inverter and the XOR macro take up three more gate delays, for a total time of $N + 2$. If the path starts at a_0 instead of c_0 an additional two delays would be incurred:

$$\begin{aligned} D(\text{ripple}, c_0, s_N) &= D(R(N), c_0, c_{N-1}) + D(\text{ripple}, c_{N-1}, s_{N-1}) \\ &= N + 2 \end{aligned} \quad (4.10)$$

$$\begin{aligned} D(\text{ripple}, a_0, s_N) &= D(\text{ripple}, a_0, p_0) \\ &+ D(R(N), p_0, c_{N-1}) \\ &+ D(\text{ripple}, c_{N-1}, s_{N-1}) \\ &= N + 4 \end{aligned} \quad (4.11)$$

All of the propagate generate style adders have the same propagate-generate and **xor** summation logic implementations in common, therefore each adder is characterized only by its carry path. In the case of the ripple carry adder:

$$D(R(N), c_0, c_N) = N \quad (4.12)$$

$$D(R(N), c_0, c_{N-1}) = N - 1 \quad (4.13)$$

$$D(R(N), g_0, c_N) = N \quad (4.14)$$

$$D(R(N), g_0, c_{N-1}) = N - 1 \quad (4.15)$$

Application of our 'A' (area) operator to a net list returns the total number of gates in the net list. For the ripple carry adder:

$$A(\text{ripple}(N)) = \frac{N}{2}A(\text{GT}) + \frac{N}{2}A(\text{XGT}) + A(R(N)) + NA(\text{XOR}) \quad (4.16)$$

The bit cells which use the XGT block require another inverter in the XOR macro to give the correct logic sense. (The GT blocks already have extra inverters.) Hence, the average gate count for the 3 bit XOR macro is 3.5 gates. The GT block requires four gates per bit position, the XGT block requires two, and the carry logic requires one. Hence the total area of the ripple carry adder is $7.5N$; however, the carry-in signal to the block fans out to three separate gates inside the block.

The bit cells in the ripple carry adder are full adders. Each one accepts two operand bits and a carry-in, then produces one sum bit and a carry-out (i.e., each corresponds to a single column in addition algorithm). Combining the input OR with the gates inside the XOR eliminates another gate, but causes a violation of the three input fan-in limit. Also, there should be an inverter

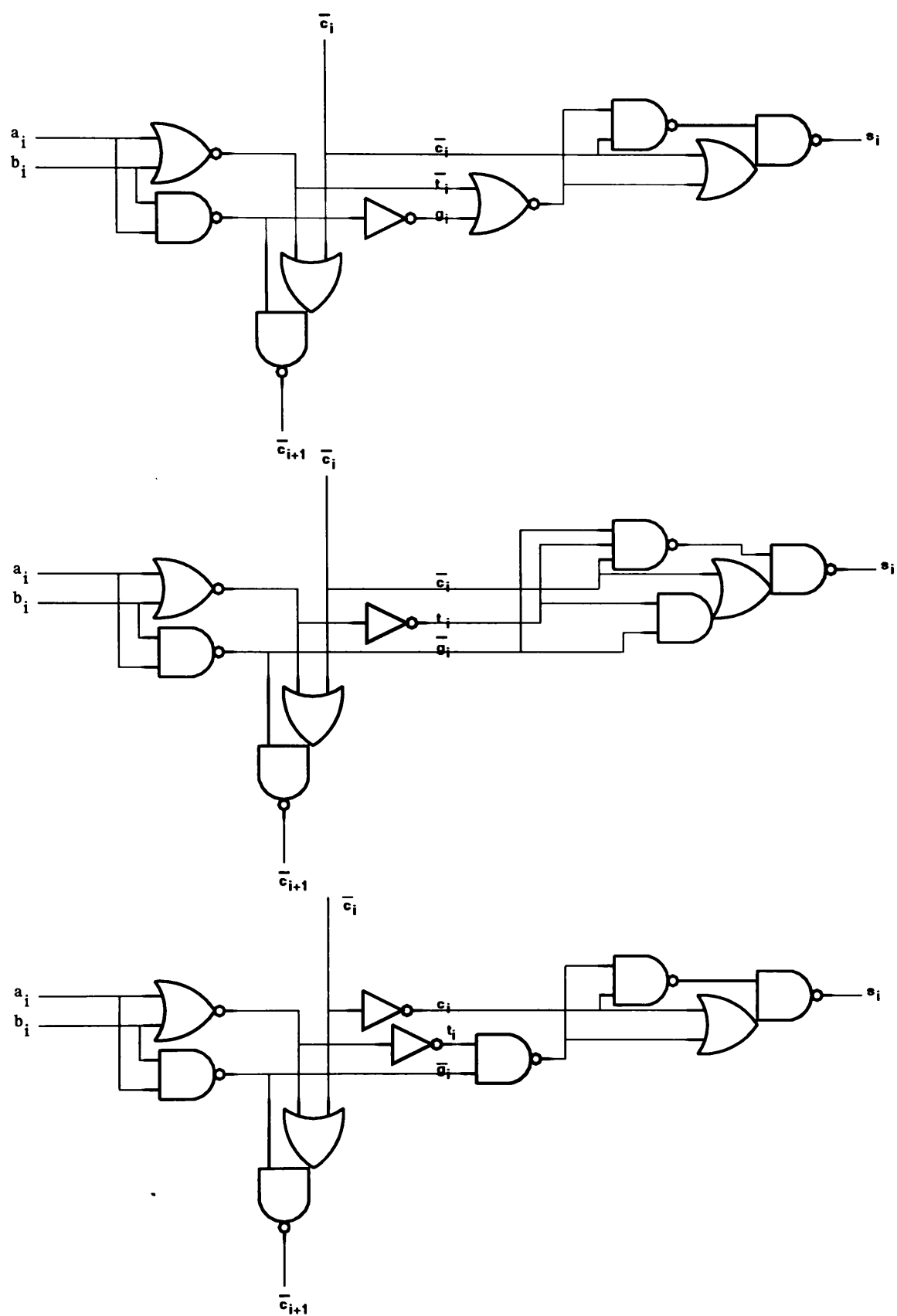


Figure 4.7: Three Full Adder Variations

between the carry path and the `xor` logic in order to isolate the load caused by feeding two gates inside the XOR macro. This isolated carry variation of the full adder will be used for analysis in this section. The complete full adders are shown in Figure 4.7. The total area is:

$$A(\text{ripple}(N)) = 8.5N \quad (4.17)$$

4.2 Carry Skip

Figure 4.8 shows a carry skip module from which all carry skip adders are made. The string of `nand` and `nor` gates which run through the block calculates the group transfer signal. This signal is always faster than the group generate signal, so it will be ignored. Group generate is derived, as explained earlier, by setting the initial carry into the ripple carry block to zero. (Hence this adder must be initialized with the carries set to zero.) The ripple carry block gets used twice, first to calculate group generate from a carry-in of zero, and then later after the carry input becomes valid, to calculate the block's carries.

The $K(n)$ macro is used to create a net list for a ripple carry adder and the string of gates needed for calculating group transfer. The carry pins on all K blocks are labeled from c_0 to c_{n-1} (for example if n is 3: c_0, c_1, c_2). When multiple K blocks are used there is a signal name ambiguity on the top level net list. To avoid this ambiguity we rename the top level nets. The names become effective at the block boundaries at points called "pins". It is conventional to also adapt a hierarchical naming convention, where the signal in a particular block is distinguished by prepending the block name onto the

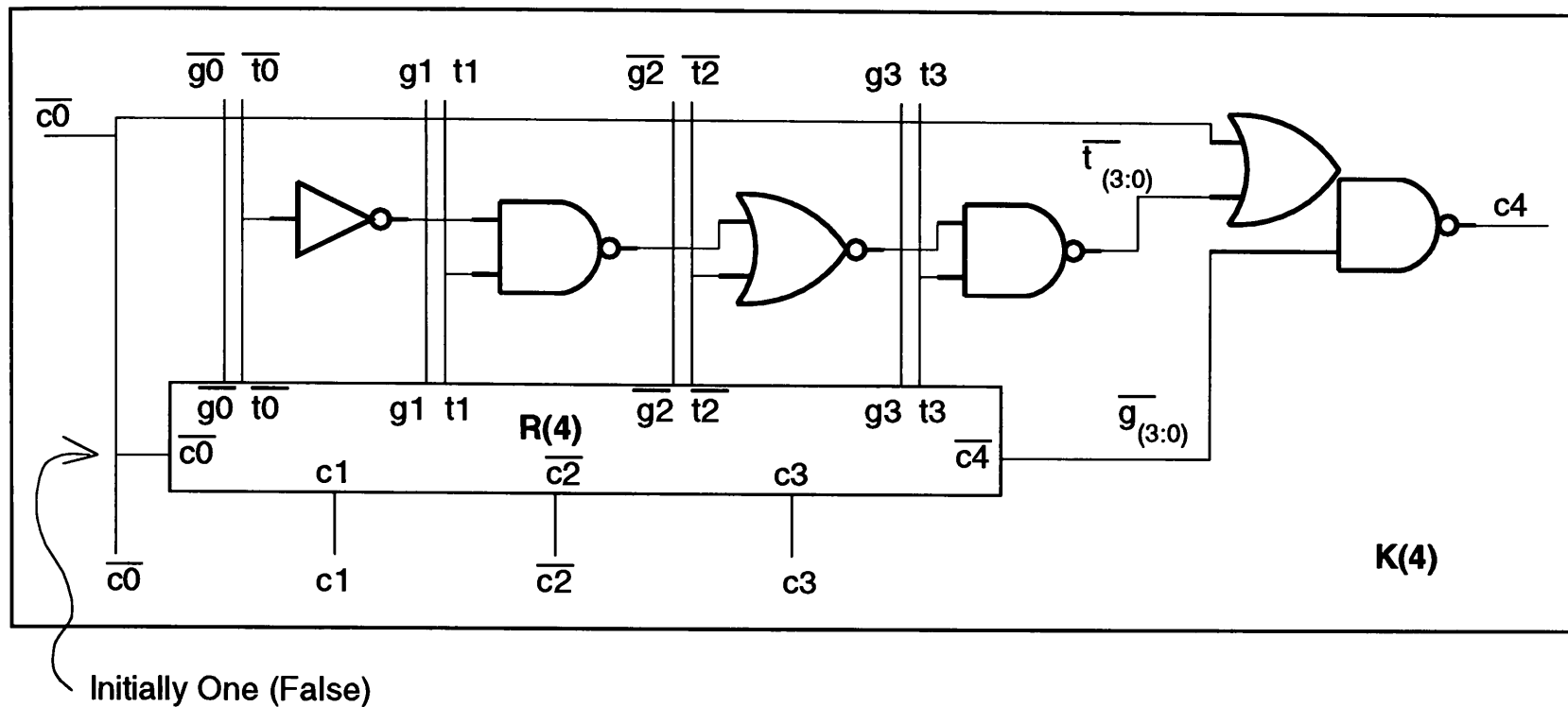


Figure 4.8: Carry Skip Section

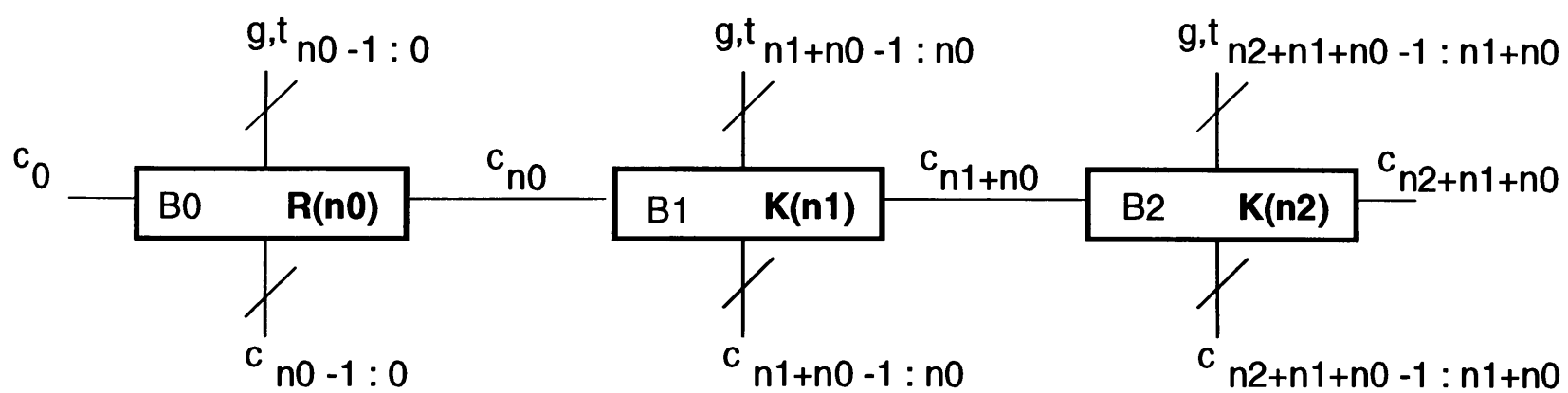


Figure 4.9: A Three Section Carry Skip Adder

signal name; hence, c_2 of macro B1 would be called B1. c_2 . We will sometimes rely on context to disambiguate the names. Usually context eliminates any ambiguity.

$$D(K(N), c_0, c_N) = 1 \quad (4.18)$$

$$D(K(N), c_0, c_{N-1}) = N - 1 \quad (4.19)$$

$$D(K(N), g_0, c_N) = N + 1 \quad (4.20)$$

$$D(K(N), g_0, c_{N-1}) = N - 1 \quad (4.21)$$

The node count of the carry skip adder carry chain is only slightly larger than that of the ripple carry adder chain:

$$A(K(N)) = 9.5N + 1 \quad (4.22)$$

The longest path length through the carry skip block is potentially much shorter than the path from carry-in to carry-out through a ripple carry block. However, the carry skip block has a slightly longer path from the least significant $\langle g, t \rangle$ input to carry output. Hence, this adder will only be faster when skipping groups makes up for the extra gate overhead accumulated by going from generate/transfer to carry-out. The maximum path length through a one block wide carry skip adder is the same as through a ripple carry adder, since the bottom block in a skip adder is a ripple carry (the bottom block cannot be skipped). A two block carry skip adder is again the same as ripple carry adder, except the carry comes out one gate later. A three block wide carry skip adder (figure 4.9) is potentially faster because carries can skip the middle

section. Usually more than three blocks will be necessary to have significant speedup.

All one level carry skip adders are uniquely defined by their block length parameters, n_0, n_1, n_2, \dots ; therefore, all of the following equations will be stated in terms of these parameters. Later, we will find that it is necessary to find an optimum set of parameters to produce the fastest carry skip adder. This is the subject of chapter 7.

The latest carry will always appear at the end of one of the skip blocks. This follows because the carry chain inside the carry skip block is a ripple carry chain, $R(n)$, so it evaluates serially with the most significant carry appearing last in the slowest case addition, as discussed in the algorithm chapter (chapter 2). Since we don't know in advance which block will produce the latest among the slow end carries, we will refer to this latest carry indirectly as c_{last} . The value of *last* for the three block carry skip adder will be one of $n_0 - 1$, $n_1 + n_0 - 1$, or $n_2 + n_1 + n_0 - 1$ (for a 12 bit wide adder composed of 4 bit wide blocks, *last* is either 3, 7, or 11). c_{last} is used to compose a sum bit on the end of one the blocks. Hence, we are considering the case where the longest path begins with the least significant operand bit and ends at the sum bit calculated from c_{last} .

There are two paths leading to the end carry of each K block (the end carry is c_{n-1} , c_n is the carry-out). One path starts from the least significant operand bit, g_0 of the K block, and the other starts from the carry-in of the particular K block. Signal names in the D calls are given relative to the macro passed in as the first operand.

$$D(\text{skip3}, a_0, c_{\text{last}}) = \text{Max}(d_0, d_1, d_2) \quad (4.23)$$

$$d_0 = D(GT) + D(R(n_0), g_0, c_{n_0-1})$$

$$e_0 = D(GT) + D(R(n_0), g_0, c_{n_0})$$

$$d_1 = \text{Max}(e_0 + D(K(n_1), c_0, c_{n_1-1}), D(GT) + D(K(n_1), g_0, c_{n_1-1}))$$

$$e_1 = \text{Max}(e_0 + D(K(n_1), c_0, c_{n_1}), D(GT) + D(K(n_1), g_0, c_{n_1}))$$

$$d_2 = \text{Max}(e_1 + D(K(n_2), c_0, c_{n_2-1}), D(GT) + D(K(n_2), g_0, c_{n_2-1}))$$

The three variables d_0, d_1, d_2 represent the path lengths to the end carry on each block. d_0 is the worst case path length to the least significant block's (i.e. block B0's) end carry, c_{n_0-1} . e_0 is the path length to the least significant block's carry-out, c_{n_0} . There are two ways to make the block B1's end carry, d_1 , and the maximum of the two path lengths is the one with the longest path. This is true in general for skipped blocks. One path goes through the skip logic, while the other goes through the group generate logic. Since we are not considering the delay to carry-out from the adder, there is no e_2 equation.

Since the middle block equation is the same for any skipped block, the carry skip equation for any number of blocks can be written down in general with three equations, where the second equation is iterated for $N - 2$ times. Among the following equations, the first is for the least significant block, it is special because it cannot be skipped. The third is special because it does not produce a carry output. We introduced M which is equal to $N - 1$ to make the subscripts legible.

$$\text{where } 0 \leq j \leq M : D(\text{skip3}, a_0, c_{\text{last}}) = \text{Max}(d_j) \quad (4.24)$$

For the bottom block:

$$d_0 = D(GT) + D(R(n_0), g_0, c_{n_0-1}) \quad (4.25)$$

$$e_0 = D(GT) + D(R(n_0), g_0, c_{n_0}) \quad (4.26)$$

Where $1 \leq i \leq (M - 1), :$

$$d_i = \text{Max}(e_{i-1} + D(K(n_i), c_0, c_{n_i-1}), D(GT) + D(K(n_i), g_0, c_{n_i-1})) \quad (4.27)$$

$$e_i = \text{Max}(e_{i-1} + D(K(n_i), c_0, c_{n_i}), D(GT) + D(K(n_i), g_0, c_{n_i})) \quad (4.28)$$

For the most significant block:

$$d_M = \text{Max}(e_{M-1} + D(K(n_M), c_0, c_{n_M-1}), D(GT) + D(K(n_M), g_0, c_{n_M-1})) \quad (4.29)$$

As noted earlier, the two operands in the Max function come from the two sources for a carry: a generate from the skip block, or a propagate over the skip block. In the case of the path to end carry of the skip block, the carry-in and the carry generate path coincide. Hence the carry-in to end carry will always be later than the operand bit end carry. The above equations are simplified to:

For the bottom block:

$$d_0 = D(GT) + D(R(n_0), g_0, c_{n_0-1}) \quad (4.30)$$

$$e_0 = D(GT) + D(R(n_0), g_0, c_{n_0}) \quad (4.31)$$

Where $1 \leq i \leq (M - 1)$:

$$d_i = e_{i-1} + D(K(n_i), c_0, c_{n_i-1}) \quad (4.32)$$

$$e_i = \text{Max}(e_{i-1} + D(K(n_i), c_0, c_{n_i}), D(GT) + D(K(n_i), g_0, c_{n_i})) \quad (4.33)$$

For the most significant block:

$$d_M = e_{M-1} + D(K(n_M), c_0, c_{n_M-1}) \quad (4.34)$$

These equations were derived without any restrictions on device behavior or connectivity - they are only topological description of paths which are traversed. Hence, many device and connectivity models can be placed into them. We will continue from this point to use the gate delay relationships listed in equations 4.18 through 4.21 for the skip blocks, and equations 4.12 through 4.15 for the ripple block. The skip path equations then reduce to the following gate delay equations:

$$\text{where } 0 \leq j \leq M : D(\text{skip3}, a_0, c_{\text{last}}) = \text{Max}(d_j) \quad (4.35)$$

For the bottom block:

$$d_0 = n_0 \quad (4.36)$$

$$e_0 = n_0 + 1 \quad (4.37)$$

Where $0 < i < M$, and M is the number of blocks minus 1.

$$d_i = e_{i-1} + n_i - 1 \quad (4.38)$$

$$e_i = \text{Max}(e_{i-1} + 1, n_i + 2) \quad (4.39)$$

For the most significant block:

$$d_M = e_{M-1} + n_M - 1 \quad (4.40)$$

With the exceptions of the conditional sum adder, and some of the surveyed adders, we have not yet discussed hierarchical adder structures. For example, it is possible to implement carry skip adders, where instead of using $R(n_i)$ inside the $K(n_i)$ module, another carry skip is used. For large adders carry skip is faster than ripple carry, so doing this will result in a faster module. The topic of carry select nesting is discussed in Chapter 5, while the topic of carry operation nesting is discussed in chapter 6. Chapter 9 gives relative performance of the adders.

Chapter 5

cso Operator Based Adders

This is the first of two chapters on unifying adder designs. The basic idea is that the different adder configurations can be created by changing the associativity of an associative carry operator. Hence, a generic optimized adder generator would accept a timing constraint and then adjust the associativities to be as linear as possible while still making speed. Faster adders tend towards trees.

This chapter explores the implications of the associativity of the carry select operator. The carry select operator multiplexes both sums and carries. The next chapter expands on the potentially more powerful principle of multiplexing only the carries.

5.1 One Level Structure - Ripple Carry

In order to illustrate the principles of this chapter, it will be necessary to use a diagram capable of packing more bits into a figure, so we have adopted the dot diagrams found in many articles on the parallel prefix problem. Figure 5.1 shows a ripple carry adder. The darkened nodes are *fco* operations while the *Xed* nodes are *xor* gates.

Generate 7 and the associated *fco* operation are omitted since they are not needed for calculating the 8th sum bit. The nodes in these graphs represent

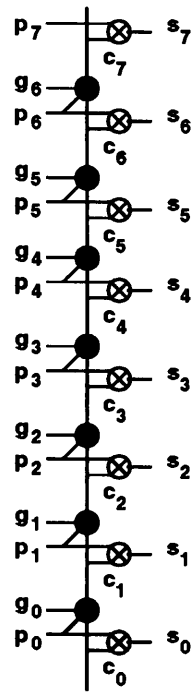


Figure 5.1: Ripple Carry Adder

fco delays, and not gate delays. Also the paths start at propagate generate and lead to carry.

$$D(\text{ripple}(N)) = N \quad (5.1)$$

The total number of nodes is also measured in *fco* counts. The logic for producing propagate, generate, and the sum is not included.

$$A(\text{ripple}(N)) = N \quad (5.2)$$

The metric *columns* gives the number of vertical *fco* tracks required to layout the adder. Because some adders have a peculiar shape which leave behind holes in the layout which are difficult to utilize, the columns count may be more indicative of required VLSI real estate than the number of nodes in the graph.

$$columns(ripple(N)) = 1 \quad (5.3)$$

5.2 Two Level Structure - Carry Select

An adder with a shorter D can be created by partitioning the ripple carry adder into two smaller adders. It follows from the definition of weighted binary code

$$a_{n:m} = \sum_{i=m}^n 2^i a_i \quad (5.4)$$

(where n and m are a range in the components of the bit vector used to represent the number a) that numbers can be broken into the sum of separate digit groups:

$$\forall_j | m \leq j \leq n : a_{n:m} = a_{n:j} + a_{j-1:m} \quad (5.5)$$

From this it follows that the sum of two bit vectors can be decomposed into the sum of two smaller sums:

$$a_{n:m} + b_{n:m} = (a_{n:j} + b_{n:j}) + (a_{j-1:m} + b_{j-1:m}) \quad (5.6)$$

Which is little more than saying the bottom bits and top bits can be added separately and the resulting two sums then added together. A simplification is possible here, the final sum can almost be formed with the computationally cheap operation of concatenating the two short sums:

$$a_{n:m} + b_{n:m} \neq (a_{n:j} + b_{n:j}) \circ (a_{j-1:m} + b_{j-1:m}) \quad (5.7)$$

However, this fails when the least significant sum carries. We can imagine another operator which also has computational advantages over that of general addition which still takes the carry case into account:

$$a_{n:m} + b_{n:m} = (a_{n:j} + b_{n:j}) \text{ } cso_j \text{ } (a_{j-1:m} + b_{j-1:m}) \quad (5.8)$$

This *carry select operator*, *cso*, chooses between two possible concatenations, one of just the two short sums, and one of the most significant short sum + 1, and the least significant short sum - 2^j , as shown in the example adder in figure 5.2. In this adder, the trapezoids are used to symbolize 2:1 multiplexers. One can see in the graph that the ripple carry adder from figure 5.1 has been broken in half, and the upper half has been duplicated, once with a carry in of zero, and once with a carry in of one. The carry in of one performs the increment. This splitting action on the graph corresponds to the *cso* operation in equation 5.8.

Compared to addition the *cso* operator is rather ugly as it is nonlinear, not commutative, requires a third operand to indicate where to locate the carry (which we will omit from the equations for brevity), and the subscripts on the adjoining operands to be added must be aligned (as j and $j - 1$ above). However, a most important quality is preserved, *cso* is associative:

$$((a_{n:j} + b_{n:j}) \text{ } cso \text{ } (a_{j-1:k} + b_{j-1:k})) \text{ } cso \text{ } (a_{k-1:m} + b_{k-1:m})$$

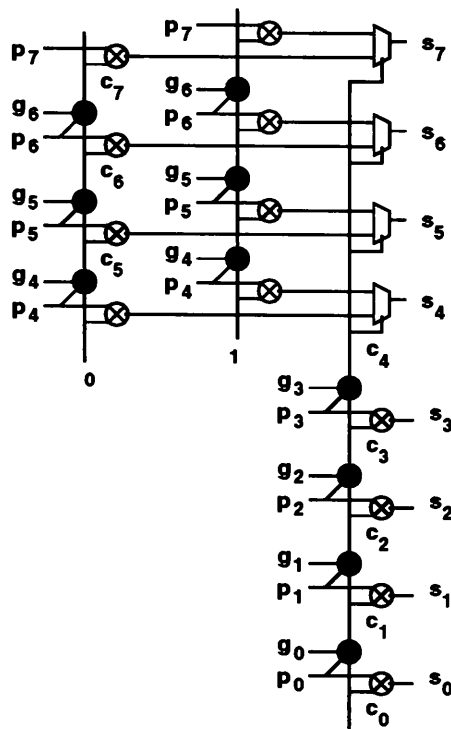


Figure 5.2: Carry Select Adder

$$= (a_{n:j} + b_{n:j}) \text{ } cso \text{ } ((a_{j-1:k} + b_{j-1:k}) \text{ } cso \text{ } (a_{k-1:m} + b_{k-1:m})) \quad (5.9)$$

Just as was done for ripple carry addition, we define a function which returns the graph of a carry select adder, *select*. The operands to this function are two adders which must be combined with the *cso* operator. For example, the adder on the left hand side of equation 5.9 is described by:

$$\text{select}(\text{select}(\text{ripple}(n - j + 1), \text{ripple}(j - k)), \text{ripple}(k - m)) \quad (5.10)$$

The adder on the right hand side is:

$$\text{select}(\text{ripple}(n - j + 1)), \text{select}(\text{ripple}(j - k), \text{ripple}(k - m)) \quad (5.11)$$

If we count the final selector as one processor delay the delay caused by using *cso* to evenly split an adder is:

$$D(\text{select}(\text{ripple}(n_1), \text{ripple}(n_0))) = \max(n_0, n_1) + 1 \quad (5.12)$$

It follows that to optimally split a ripple carry adder in two with a *cso* operator, n_0 must equal n_1 , and that the total delay is ¹ is nearly cut in half from that of ripple carry:

$$N/2 + 1 \quad (5.13)$$

The area resulting from applying one carry select operator to a ripple carry adder is:

$$A(\text{select}(\text{ripple}(n_1), \text{ripple}(n_0))) = n_0 + 3n_1 \quad (5.14)$$

The n_0 comes from the lower ripple carry adder, one of the $3n_1$ comes from the upper adder with carry-in equal to zero, another from the upper adder with carry-in equal to one, and the third from the n_1 selector processors required at one per bit.

$$\text{columns}(\text{select}(\text{ripple}(n_1), \text{ripple}(n_0))) = 2 \quad (5.15)$$

The *cso* operator can be applied many times in series as shown in the adder in figure 5.3. This adder still only requires two columns. By carefully selecting the block sizes it is possible to produce a polynomial time adder. This

¹for an odd value of N , add one more delay

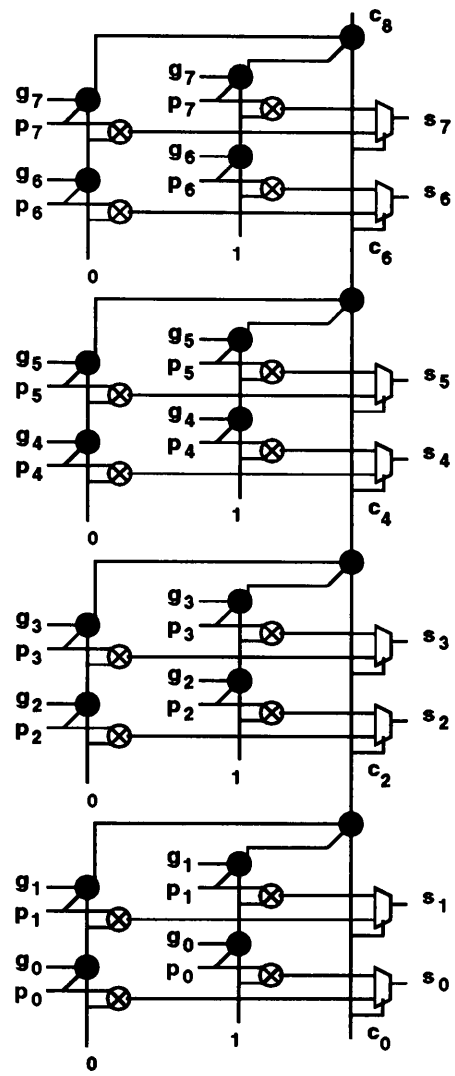


Figure 5.3: Ripple Carry Select Adder

is done by setting each block to the maximum size it can be without making the carry select wait for the sum data to arrive. If the first block is set to one bit wide, then the next block can be two bits wide without creating a speed path, since the carry will be delayed by one *fco* operation in the first block. The carry in the second block is again delayed by one as it travels through the second *fco* operation etc. Each block can be one larger than the previous block, until running off the end of the adder. For such an adder the sum of the bits in the groups must be equal to number of bits in the adder. The last group may be cut short, hence:

$$\min g \text{ such that } \sum_i^g n_i \geq N \quad (5.16)$$

If we say that M is the smallest value that fits the pattern of the Σ (i.e. 1, 3, 6, 10, 15, ... bits), and is greater than or equal to N , then

$$M = \sum_i^g n_i = g(g+1)/2 \quad (5.17)$$

We solve for g using the quadratic formula. Without considering the problems caused by the large fan out of some of the multiplexer select lines, there is one delay element per group, so:

$$D(\text{optimum ripple carry select}) = \frac{\sqrt{1+8M}-1}{2} \quad (5.18)$$

5.3 N Level Structures

It is well known that the shortest delay in evaluating an expression made from associative operators is when the operators are grouped in a tree. Accordingly, in the following adder all possible pairs are first calculated - because this is the most that can be done in parallel with a binary operator such as *cso*. Then these results from these pairings are paired again, since, once again, this is the most that can be done in parallel, etc. The equation for a maximum speed 8 bit carry select based adder is:

$$(((a_7ob_7)o(a_6ob_6))o((a_5ob_5)o(a_4ob_4)))o(((a_3ob_3)o(a_2ob_2))o((a_1ob_1)o(a_0ob_0))) \quad (5.19)$$

Here *o* is a short form of *cso*.

The corresponding graph is generated by:

$$s(s(s(s(r, r), s(r, r)), s(s(r, r), s(r, r))), s(s(s(r, r), s(r, r)), s(s(r, r), s(r, r)))) \quad (5.20)$$

Where *s* is used for *select* and *r* is used for *ripple*(1). The adder is shown in figure 5.4. Whenever the *cso* operator is used to recursively break down an adder to one bit ripple blocks in this way, the adder is called a *Conditional Sum Adder*.

The following equations are the conditional sum adder parameters, when *N* is a power of 2.

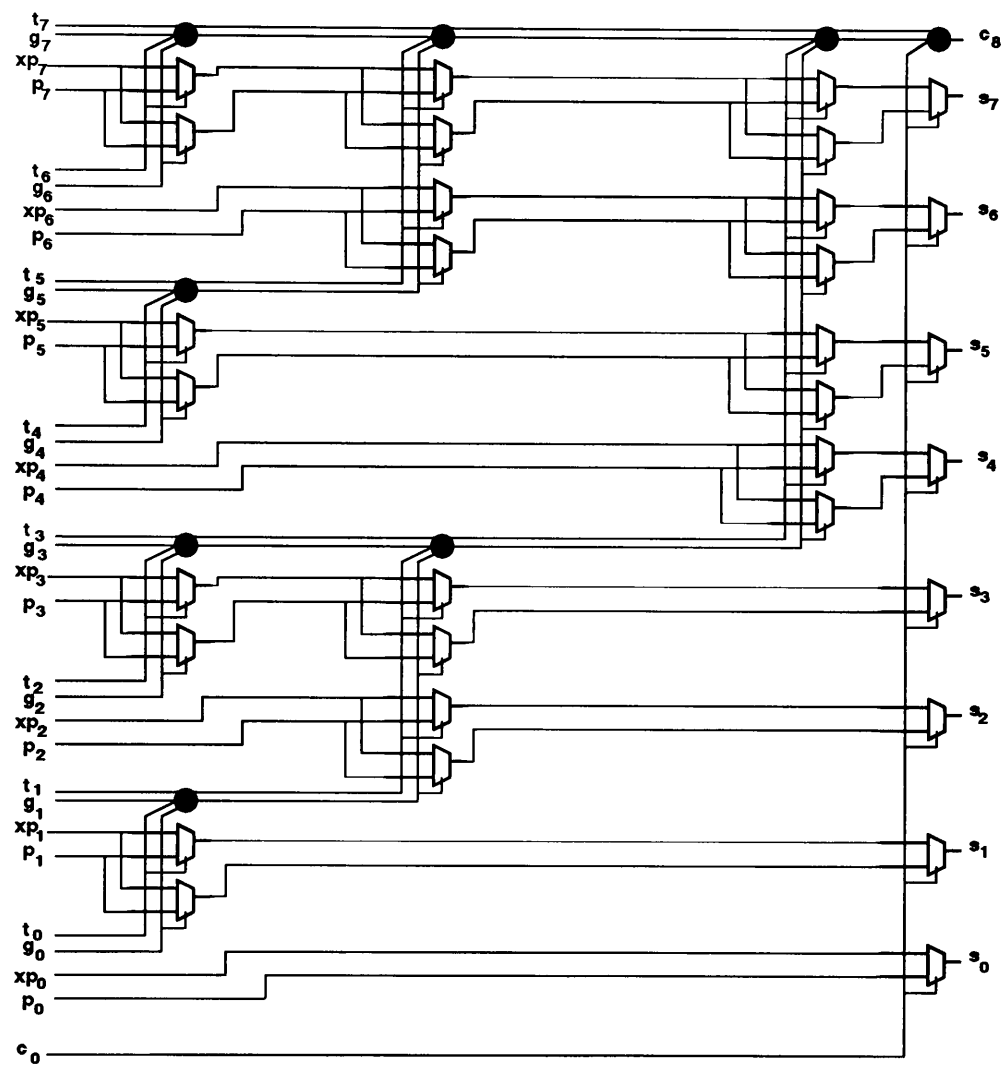


Figure 5.4: Conditional Sum Adder

$$D(\textit{conditional_sum}(N)) = \log_2 N + 1 \quad (5.21)$$

$$A(\textit{conditional_sum}(N)) = N(\log_2 N + 1) + N \quad (5.22)$$

$$\textit{columns}(\textit{conditional_sum}(N)) = \log_2 N + 1 \quad (5.23)$$

The delay number is equal to the levels in the tree, plus one level for the final select based on carry in. The $\log_2 N + 1$ term in the area equation is the number of levels in the adder. Each level has N 2:1 multiplexers, so the log term is multiplied by N . The N summed in at the end is the number of *fco* operations required for propagating the carry.

The conditional sum adder for 7 bits (or 8 bits without a carry input) is quite a bit simpler, as is apparent from figure 5.5.

These are the parameters for the simplified adder. Here N is a power of two and is the number of bits in the adder without a carry in:

$$D(\textit{conditional_sum}(N)) = \log_2 N \quad (5.24)$$

$$\textit{columns}(\textit{conditional_sum}(N)) = \log_2 N \quad (5.25)$$

$$A(\textit{conditional_sum}(N)) = \left(\frac{3}{4}N - 1\right)\log_2 N + \frac{1}{4}N \quad (5.26)$$

In summary this section showed how a property of the binary number representation lead to an associative operator which could be used to build adders which have anywhere from linear to logarithmic growth in the number of circuit elements in the longest delay path, when measured as a function of

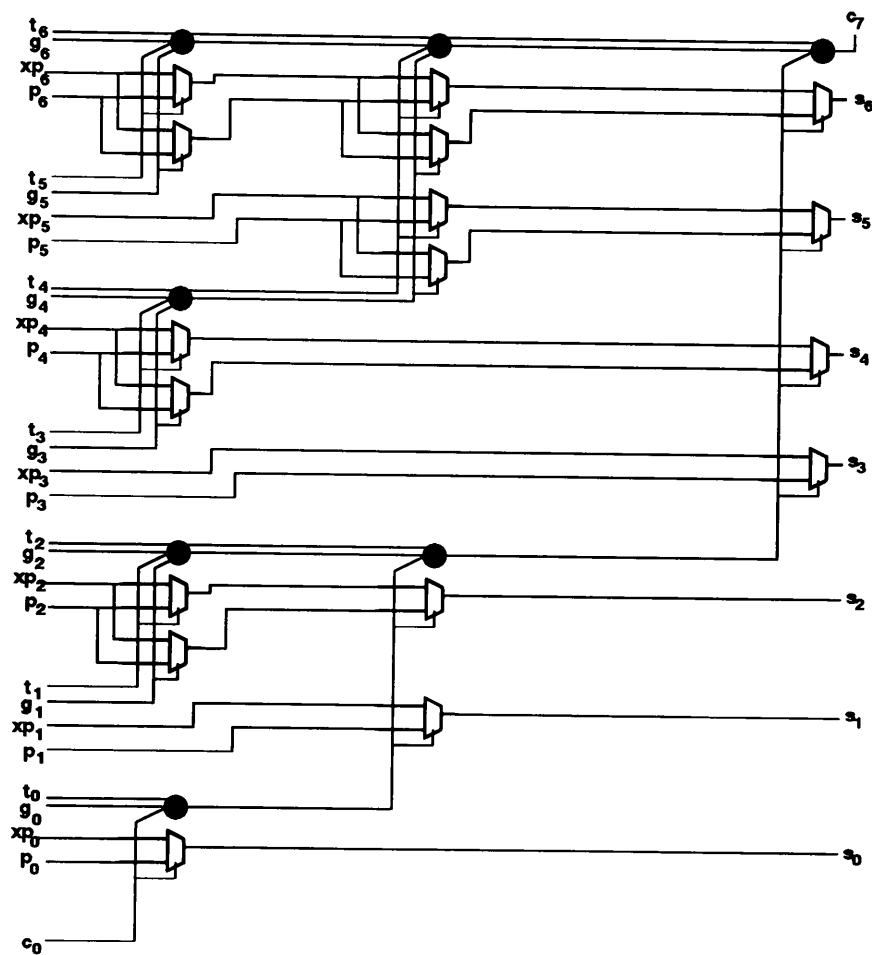


Figure 5.5: Simplified Conditional Sum Adder

the operand length. These same adders had an order of growth between linear and $N \log N$ total number of nodes in the circuit graph.

Chapter 6

fco Operator Based Adders

This chapter completes the unification of adder designs by demonstrating that changing the associativity on the carry multiplexing operator yields the remaining conventional designs, and most of the recent designs.

6.1 One Pass *fco* Trees

As described earlier, addition operands can be normalized by applying these operations:

$$g_i = a_i \wedge b_i \tag{6.1}$$

$$p_i = a_i \vee b_i \tag{6.2}$$

$$\tag{6.3}$$

The g_i and p_i signals contain all the information needed from the operands, and they give direct logical information on how to propagate the carry:

$$c_{i+1} = g_i \vee p_i c_i \tag{6.4}$$

The propagate and generate signals are collected using an operator, *fco*. This operator combines propagate and generate information about smaller

groups into propagate generate information for a larger group:

$$p_{n:m} = p_{n:j} \wedge p_{j-1:m} \quad (6.5)$$

$$g_{n:m} = g_{n:j} \vee p_{n:j} g_{j-1:m} \quad (6.6)$$

The colon notation used in the subscripts indicates that the *group* propagate and generate signals are being used, as was described in chapter 3. For example, $p_{n:m}$ is a single signal which is true when a carry into bit m would propagate through the block, and appear as a carry-out from bit n . Similarly $g_{j-1,m}$ is a single signal which would be true if a carry could be generated in the block of bits between bit $j - 1$ and bit m , inclusive. It follows that:

$$c_{n+1} = g_{n:m} \vee p_{n:m} c_m \quad (6.7)$$

When the generate signal is asserted, the circuit outputs a carry. The converse may not be true. That is, if the generate signal is not asserted there may still be a carry due to propagation.

Note, equations 6.4, 6.5, and 6.7 all have the same form.

Just as for the *cso* operator, the *fco* operator is associative. Because it is associative, it can be applied in series to make ripple structures:

$$c_{n+1} = (\dots ((c_0 \text{ fco } (p_1, g_1)) \text{ fco } (p_2, g_2)) \dots \text{ fco } (p_n, g_n)) \quad (6.8)$$

Or nested to produce an order log time sum:

$$c_{n+1} = (\dots ((c_0 \text{ fco } (p_1, g_1)) \text{ fco } ((p_2, g_2) \text{ fco } (p_3, g_3))) \dots) \quad (6.9)$$

To make all of the sum bits, all of the carries must be known. In equation 6.8 each prefix of the equation is an equation of the same form as the whole equation. Hence it is obvious that all of the carries are generated. However it is not obvious that all the carries can be gotten directly from implementing equation 6.9. In fact, they cannot. Supplemental logic must be put in place to calculate the remaining carries.

Figure 6.1 from Brent and Kung [60] conveys both electrical and layout information. Each white cell in the graph is an inverting buffer, and each black cell is an appropriate polarity *fco* function. The graph also shows a limited amount of timing information as there is one *fco* delay per column in the graph. The adder shown requires

$$\text{delay}(BK(N)) = 2 \log_2(N) - 1 \quad (6.10)$$

fco delays for creating the carries ¹. Producing the propagate and generate signals requires another delay element, as does producing the sums from the carries.

The Kogge and Stone adder removes the multiplying factor of 2 and the “−1” in equation 6.10. Their adder is shown in figure 6.2. Han and Carlson [68, 67] observed that Kogge and Stone’s adder required a great number of metal

¹Brent and Kung’s adder can be collapsed by one column when non-inverting logic cells are used. This action increases the fanout of the middle right tree cell (in row c_7).

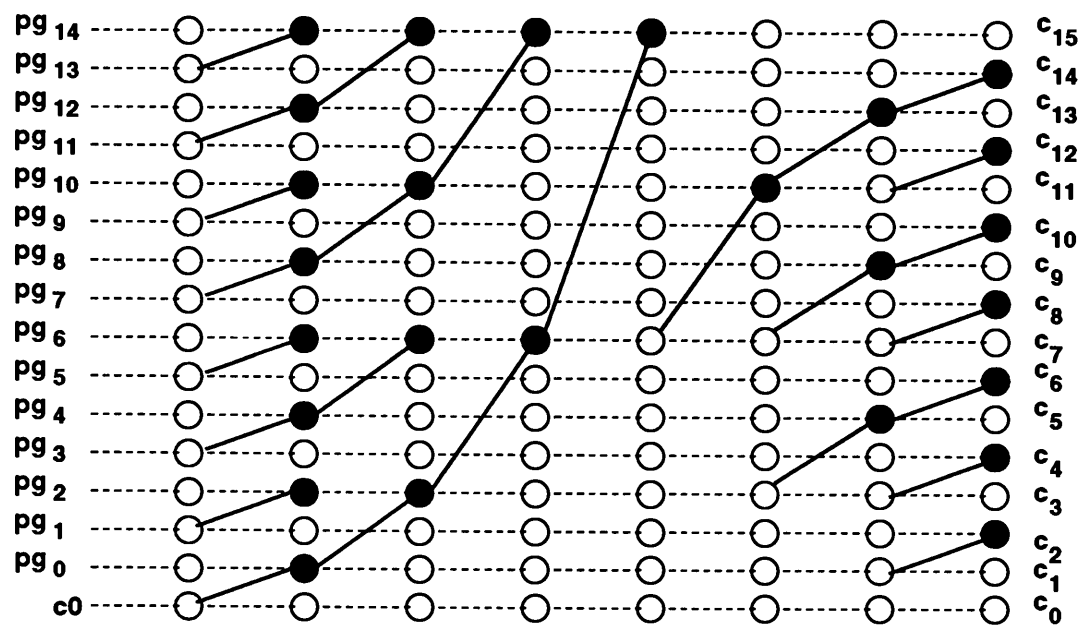


Figure 6.1: Brent & Kung Adder

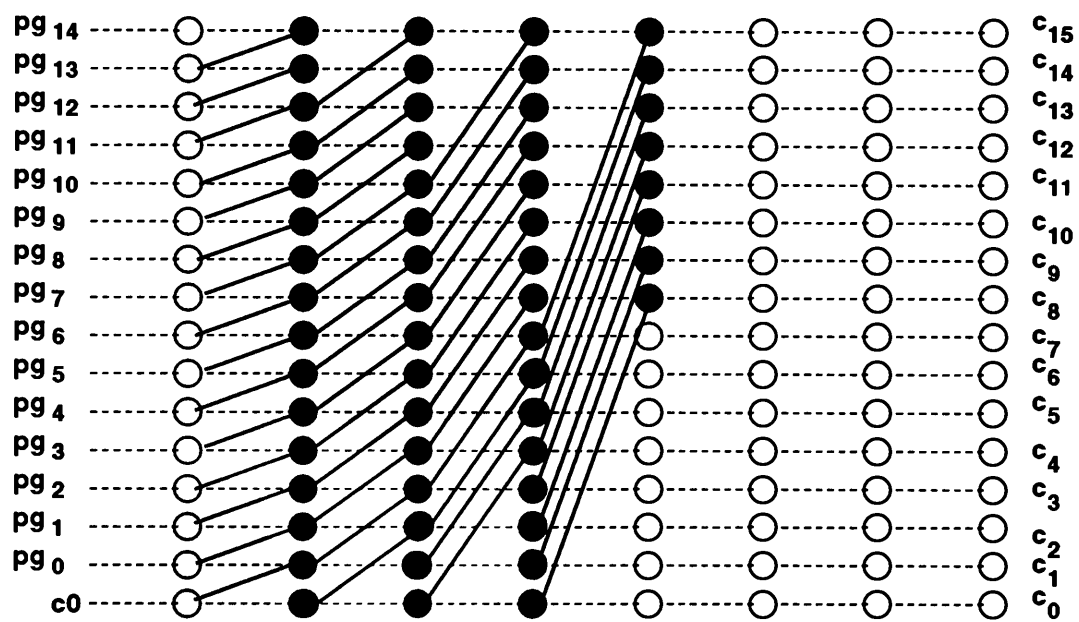


Figure 6.2: Kogge & Stone Adder

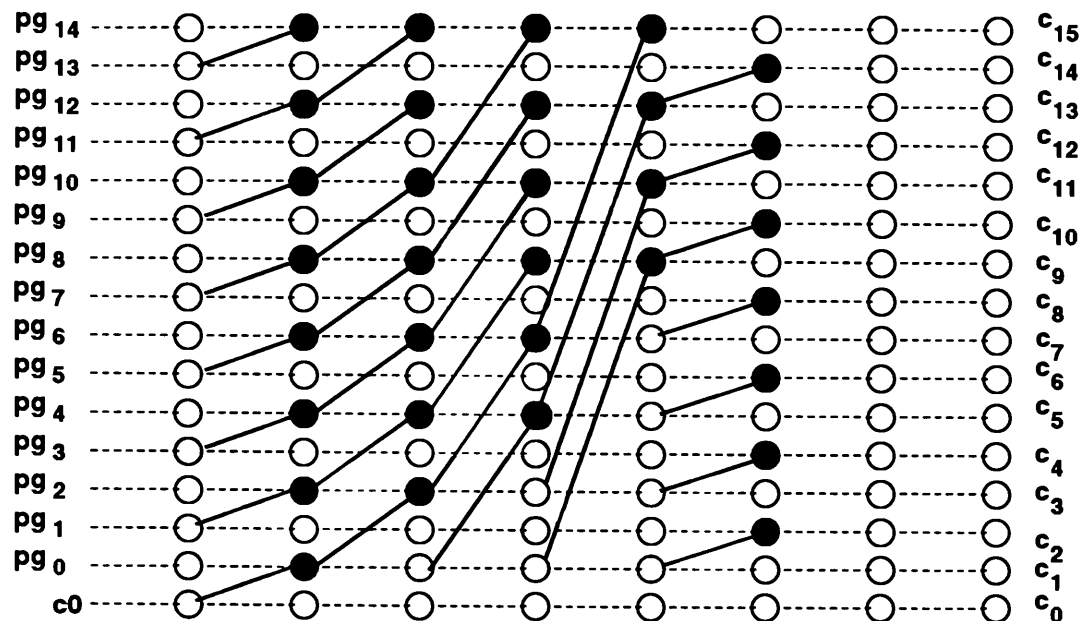


Figure 6.3: Han & Carlson Adder

tracks for group propagate and generate signals, so they created an efficient hybrid between the Brent and Kung's adder and the Kogge and Stone adder. This adder is shown in figure 6.3. As Han and Carlson discuss, it is possible to make adders with anywhere from linear to logarithmic time performance.

Becker and Kolla [79] suggested implementing the *fco* operation with multiplexers. As we explained in an earlier in chapter 3, this is an equivalent *fco* operation. Becker and Kolla also stated that they were seeking an adder which had a higher base in the logarithm.

6.2 Folded *fco* Trees – Conventional Adders

In this section we introduce a parallel prefix graph operation called *folding*. Armed with this extra tool it becomes possible to write parallel prefix graphs for all of the adders. We demonstrated the principle with carry skip. There have been many hints that such a thing was possible. In their paper on carry

skip, Lehman and Burla refer to a carry lookahead adder as an extreme type of carry skip adder. We showed in chapter 2 (Algorithms) that conditional carries are the same as group propagate and transfer signals, hence collapsing carry select and carry skip. It is also apparent from this thesis that the carry path from the conditional sum adder is the same as the tree part of Brent and Kung's parallel prefix adder. Lee and Oklobdzija even showed how carry skip optimization could be applied to carry lookahead adders.

The ripple carry adder which forms the building blocks of other kinds of adders has the simplest parallel prefix form as it is a just linear sequence of *fco* operators.

A 16 bit carry skip adder is shown in figure 6.4. In this graph, the columns represent time, and not layout. This carry skip adder with equal size blocks requires only one more *fco* delay over Brent and Kung's adder; however, Brent and Kung's adder has better load balancing.

The carry skip adder has the unusual property of using the block logic twice. The diagonal string of *fco* operations going off the right side are not actually implemented. Instead the diagonal strands shown on the left are reused as shown in Figure 6.5. We call this reuse *folding*. The tree part and inverse tree part of the prefix graph must be symmetric for folding to be possible.

Because the latter diagram loses the property of having the columns mark the number of *fco* delays we will prefer the first diagram, with notes marking when logic is reused.

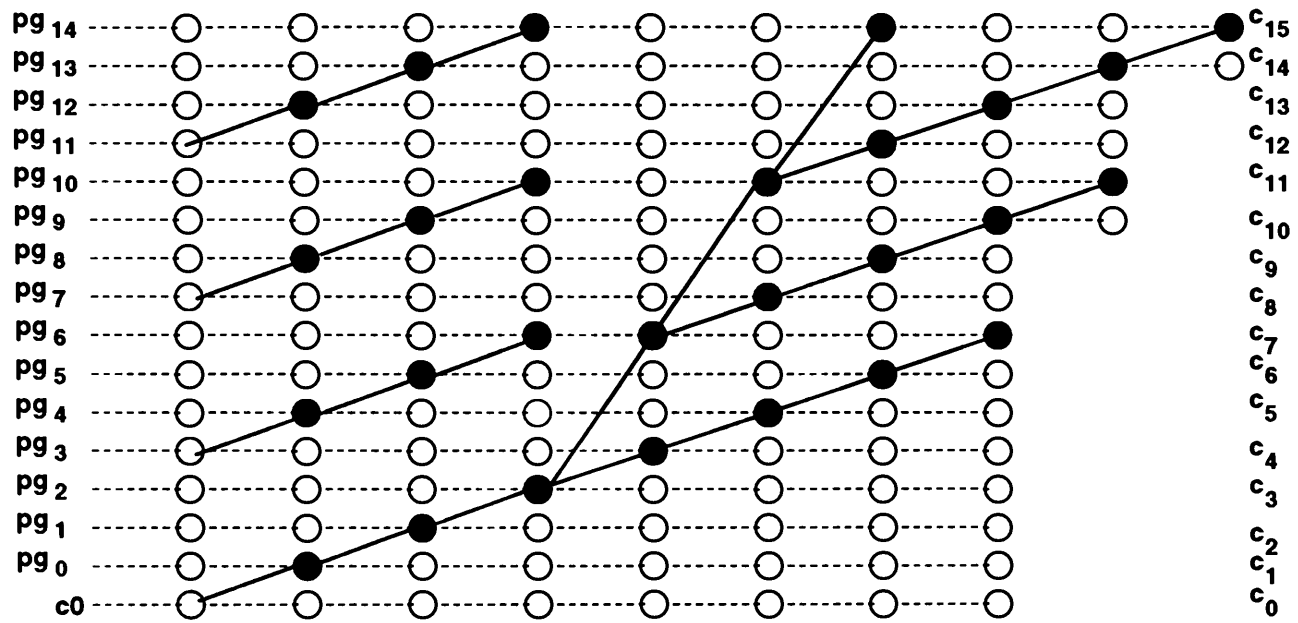


Figure 6.4: Carry Skip Adder, Timing Correct

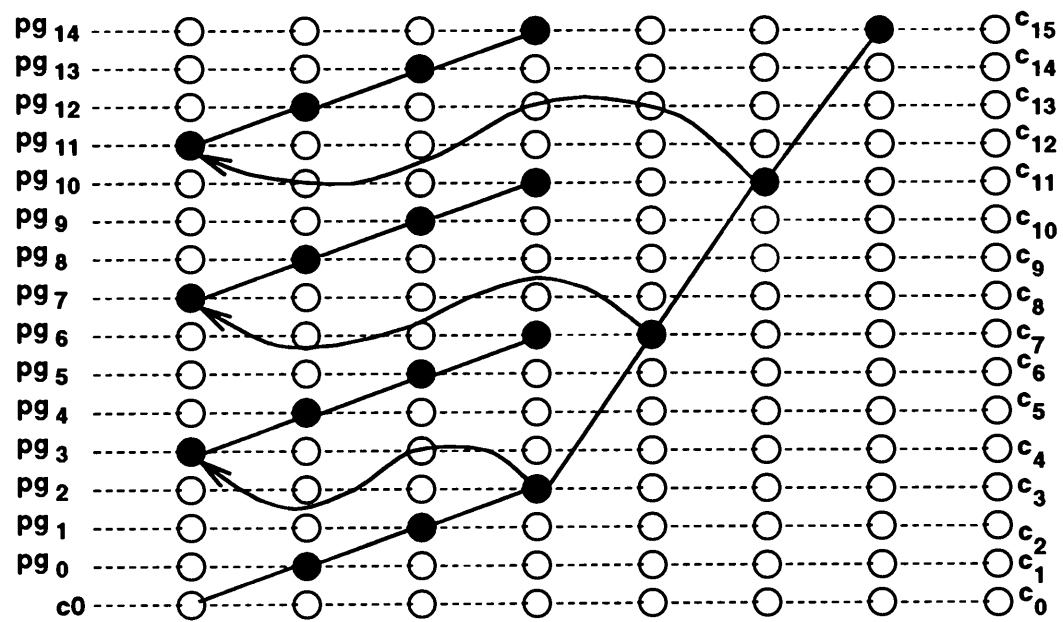


Figure 6.5: Carry Skip Adder, Logic Correct

Neither of figures 6.4 and 6.5 corresponds to probable layout. This is shown in Figure 6.6 where the cells have been packed to the left.

This example carry skip adder has equal block sizes. A faster adder can be created by varying the block sizes. Figure 6.7 shows the time diagram view of an optimum one level carry skip adder. There is one fewer *fco* delay in the speed path here than in the worst case path through Brent and Kung's adder. Note, we have simplified figure 6.7 by not darkening the circles where the *fco* logic is. It is apparent from context which cells contain *fco* logic. The shared logic chains are marked with identical letters (A,B,C, or D).

Many adders are made from a hybrid of different technologies. CMOS Manchester carry chains are nonrestoring dynamic circuits which are combined together with restoring inverter stages. Hence, it may be useful to generalize the prefix graphs by placing time on the x axis and N (bits) on the y axis. Figure 6.8 shows a square law based generalized prefix graph for the carry skip adder.

For figure 6.8 we assumed that evaluation time of the Manchester carry chain blocks is x^2 , while the skip time is $3x$, where x is the "width" of the MCC block. Thus, the skipped carry is seen traveling accross the adder in linear time, while the carry generates are parabolic. For example, at time 6ns we see that block **a** has just finished evaluating for the second time, block **c** is evaluting on its first pass, and a worst case carry would be between the third and fourth bits.

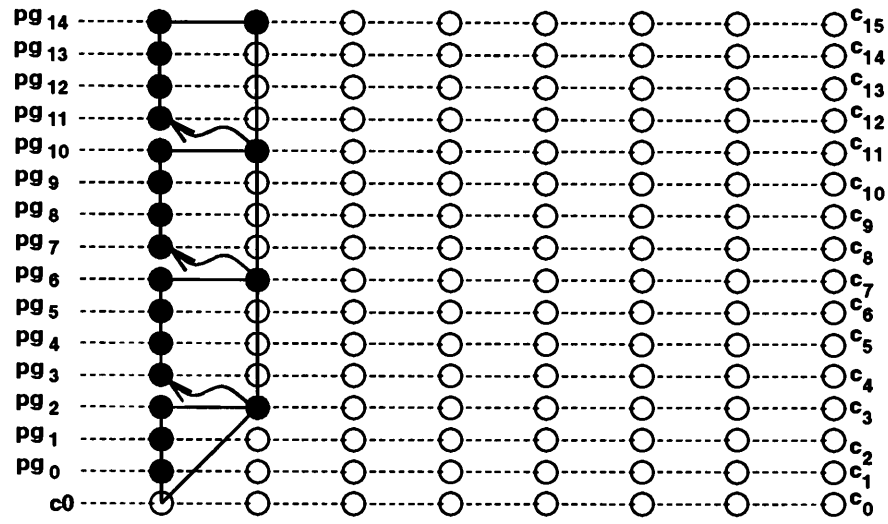


Figure 6.6: Carry Skip Adder, Layout Correct

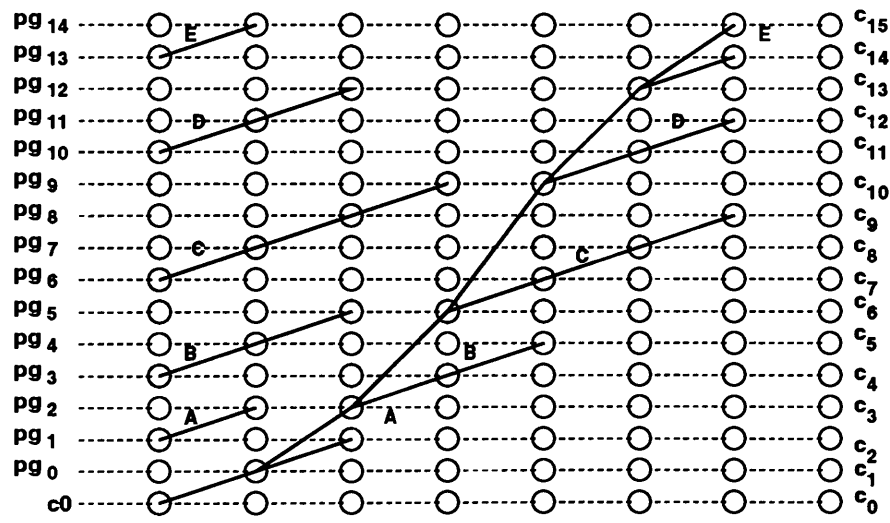


Figure 6.7: Carry Skip Adder, Variable Block Sizes

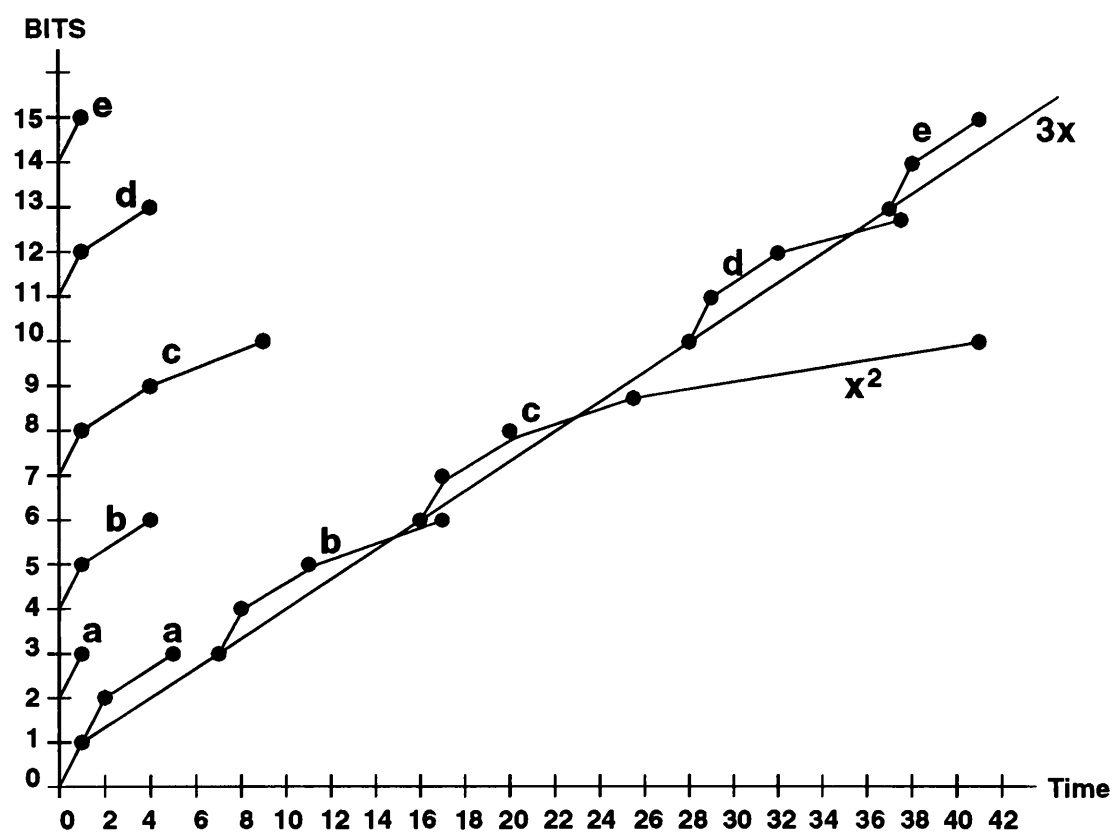


Figure 6.8: Manchester Carry Skip Adder - Real Evaluation Time

Chapter 7

Optimization

7.1 Definition of the Carry Skip Optimization Problem

Given a net list generator which produces adders implementations from a set of parameters:

$$A(N, P) \tag{7.1}$$

where N is the length of the adder to be generated, and P is the list of parameters which guide the generation of the net list. For example, $p_i \in P$ may give the device size placed on the i th transistor. Another example of parameters is the block lengths in a carry skip adder. In previous chapters we often made the parameters implicit, hence we said $K(N)$ instead of $K(N, n_0, n_1, n, \dots)$.

Optimization is performed relative to ranges of values for some of the parameters, while other parameters remain fixed or set by implicit rules. This defines the domain, X , over which optimization is applied. Optimization problems also require that some function is minimized or maximized. This function is the *cost function*, $C(X)$.

In this chapter the goal is to minimize the worst case sum time through a carry skip adder by adjusting the block lengths. Hence, we will drive the optimization from the point of view of block widths; it is implicit that the

devices in the blocks will have optimum sizing for the choices made. Other parameters will be set via fixed functions defined over the same domain as that for the cost function. Now the optimization problem can be written as:

$$\{ \min (d(A, \{c_0, a, b\}, s_{\text{latest}})), X, p_0(X), p_1(X), \dots \} \quad (7.2)$$

where

$$X = \{n_0, n_1, n_2, \dots\} \quad (7.3)$$

$$n_i \in [0, N] \quad (7.4)$$

n_i are the block lengths, and the p_i functions are additional parameters.

Little d is a function which returns the actual time through the slowest path. This differs from D we used in the previous chapters, since D measured the distance through the circuit graph of the net list. More specifically, d returns a value in seconds, and D returns a value in node count. The gate delay model makes the supposition that

$$d = kD \quad (7.5)$$

which often leads to interesting results. The useful property of this mapping is that the longest path and slowest path are the same, so searches over the circuit graph which discover the longest length path also provide the slowest path.

Other models are more accurate. However, more accurate models in general do not guarantee that the longest path and the slowest path are the same.

In the following sections we develop a method for carry skip optimization based on d , not D , and we then show successful results from a partial implementation of our method.

7.2 Weak Monotonicity in the Delay to Sum Function

Theorem 1 *An optimum adder's delay function, $d(A(N), \{c_0, a, b\}, s_{latest})$, is always weakly monotonic.*

By substituting in the definition of weakly monotonic we obtain the equivalent statement:

Given two optimum adders, say $A_1(n_1)$ and $A_2(n_2)$, where n_1 is the width of adder A_1 and n_2 is the width of adder A_2 . If $n_2 > n_1$, then the worst case sum time from operands or carry-in of A_2 is not faster than that for A_1 .

That is to say a wider optimum adder is never faster than a narrower optimum adder.

The proof is based on showing that exceptions to the rule are absurd: Suppose an adder $K_2(n_2)$ was proposed to be both faster and wider than the optimum adder $K_1(n_1)$. It would follow that $K_1(n_1)$ cannot be an optimal adder, since a third faster adder can be created, $K_3(n_1)$, where K_3 is identical to K_2 except that the upper bits are ignored.

7.3 Weak Reciprocal Relationship between Partitions

A carry skip adder can always be partitioned into two adders by drawing the partition between two top level modules such that only the carry lines cross it. The two adders on each side of such a partition have an interesting reciprocal relationship with both size and timing. Given that the width of the first partition is n_0 and the width of the second partition is n_1 :

$$N = n_0 + n_1 \quad (7.6)$$

Also if the width of the first partition is increased, the speed of the first partition either remains the same, or becomes slower (theorem 1). Because of equation 7.6, the second partition becomes smaller, and therefore becomes faster or remains the same speed.

7.4 Optimum Carry Skip Algorithm

We start by making the first partition the first module, and the second partition the rest of the adder. Accordingly, the carry-out from the module is the carry into the rest of the adder. We will call the part of the adder on the left of the partition the *prefix adder*, and the adder on the right the *suffix adder*.

The first module is set to be an optimal 1 bit adder, so its implementation is known. The exact implementation of the suffix adder is not known, but according to the design scheme under consideration, it's general form is known. For example, for one level carry skip adders it is known the adder is made from

a series of ripple carry adder modules.

Adder evaluation starts at time zero. Given the arrival time for carry input, and the operands, the carry-out of the first module will be found to occur at some absolute time called $d(c_1)$.

The second step is to perturb the first module by expanding it to a total length of two bits. In the case of optimizing a multilevel carry skip adder, this step may cause recursion into the design of the optimum module. The carry-out from the module now occurs at $d(c_2)$.

As a response to increasing the size of the prefix adder, that adder may have become slower. Also, the suffix adder receives its carry input at $d(c_2)$, instead of at time $d(c_1)$, but the remaining adder is one bit smaller. According to theorem 1, $d(c_2)$ is greater than or equal to $d(c_1)$. Hence, the size decrease and the belated carry are complementary effects.

These effects of increasing the module size can be quantified. If the prefix adder containing the module does not grow to contain the critical speed path, and the time increase caused by the belated carry into the suffix adder does not outweigh the time decrease due to having a smaller suffix adder; then the module can safely be made larger.

These effects can be partially quantified without knowledge of the implementation of the modules. The first partial finite difference in function f relative to the variable x is defined as:

$$\Delta(1, f, x) = f(x + 1) - f(x) \quad (7.7)$$

Suppose that the partition is located just after bit p , and just before bit $p + 1$. Then the module growth perturbation on the suffix adder is:

$$\Delta(1, D(S_{\text{latest}}, c_i), i)|_p - \Delta(1, D(c_i, 0), i)|_p \quad (7.8)$$

The left hand term is the difference between producing c_i and c_{i+1} evaluated at the partition. The right side is the difference in time from carry-in to the latest sum, also evaluated at the partition point. The right hand side lumps together the carry and size effects into one relation on the time the latest sum arrives. When this quantity is less than zero, growth in the module is a good thing.

Because of the weak monotonicity guaranteed by theorem 1, we can build a finite difference descent program (i.e. finite difference version of gradient descent) which is always directed by the rule: *make the blocks bigger until equation 7.8 becomes negative*. We have written the finite difference descent program based on various circuit models, which we have used to generate values for the examples in this paper. Such programs have an advantage over solving the equations when working with contemporary CMOS models since the device equations cannot be solved directly.

Such a program, based on growing modules and reducing a global sum time, and subject to the constraints discussed in the next two sections, is available at <http://devil.ece.utexas.edu/~lynch>. This program uses the following algorithm:

1. grow the module just to the left of the partition as much as possible

without violating a simplified form of equation 7.8, and without violating the global sum time, S .

2. move the partition over to the next 1 bit module.
3. repeat the previous two steps until all N bits are used up.

7.5 Module Parameters

The current optimization program uses *module parameters* to characterize the timing of left side of the partition modules. In retrospect, this approach should have been used only in the internal implementation of the module carry time calculation. Only three functions need to be defined in order to build an optimum carry skip adder.

- $generate(n, e)$
- $skip(n, c, e)$
- $sum(n, c, e)$

These are independent of how the modules are implemented. Ripple carry sections are traditional for carry skip adders. Carry lookahead modules can be used, as in [71]. Or, for a multilevel structure, the modules may also be made from carry skip adders.

$generate(n, e)$ is the worst case time required for the module to produce an independent carry. n is the width of the module in bits. e is an environment function which for simple models is not needed, but more sophisticated modules need to know about output loads. We did not extend our program to adders

where drivers would iteratively be resized, so e was not needed. The result of this function is considered to be an absolute time, so if the generate path starts at a time other than zero, that information also needs to be passed to the function.

$skip(n, c, e)$ is the worst case time for a skipped carry to appear from the module. c is the carry input arrival time. The value of the function is again an absolute time. The skip logic is considered to be part of the module.

$sum(n, c, e)$ is the worst case time for the generation of the sum given the module width and the carry arrival time.

As an example, consider the functions implemented for optimizing the gate based carry skip adder from chapter 4, the generate time is:

$$generate(n) = 1 + (n - 1)2 + 1 \quad (7.9)$$

The first 1 is for the AND gate which produces the bit generate in the least significant bit, then generate is propagated through $(n - 1)$ fco delays, and ORed into the skip logic.

The skip delay for the chapter 4 gate model is:

$$skip(n, c) = max(n, c) + 2 \quad (7.10)$$

Here the maximum is taken between the group propagate signal arriving at the skip logic (1 delay per bit according to this implementation), or the carry arriving. Since times are absolute, this may be a race for the least significant blocks.

The sum bit comes from either the propagate/generate logic, or from the carry:

$$sum(n, c) = max(6, generate(n) + c + 3) \quad (7.11)$$

In our C++ program, models for modules must be inherited from the following:

```
class adder;
class model{
public:
    model(){levels=1;};
    int levels;
    model *skip1_model;

    virtual double    propagate(int i, int n, adder *a);
    virtual double    set_skip(int i, int n, adder *a);
    virtual double    generate(int i, int n, adder *a);
    virtual double    skip(int i, int n, double c_i, adder *a);
    virtual double    sum(int i, int n, double c_i, adder *a);
    virtual void      info();
    virtual double    initial_guess(int i, int n, adder *a);
    virtual double    resolution();

protected:
};
```

Set skip is the constant restoring stage delay; it is added into the skip of a multilevel carry skip adder. For one level adders it can be built into the other functions. info() prints a message out about the model. sum() is the time to produce a sum. Typically this the is maximum of the path coming from

the operands and that path coming from the carry input. `initial_guess()` and `resolution()` are used to control setting of the global sum constraint, S .

Modules are classified as either:

- carry dominated
- sum dominated
- end dominated

depending on the test that failed to allow further block growth. The module is *carry dominated* if it was the carry test given in equation 7.8. The module is *sum dominated* if it was the global sum time constraint that would be violated. And, the module is *end dominated* if the width constraint, N , would be violated.

7.6 Optimization Program Assumptions

Although the optimization method we have presented is completely general, the program we have developed uses some simplifying assumptions. Even with these assumptions, the program is capable of producing multilevel carry skip adders which take into account real number gate delay times, metal loading, fan-out, and fan-in etc. It is only a question of implementing the correct module parameter functions. The results from this program are satisfactory, and show a proof of principle. However, there is a lot of room for improvement should we take up development of a second version.

The carry-out time from a module is calculated by the optimizing program to be the maximum of the skipped carry time and the generated carry time.

The program assumes that carry skip per bit is faster than carry ripple per bit. This assumption is fundamental to carry skip addition. Roughly, if this were not true, then skip adders would be slower than ripple adders. However, it may be that skipping is slower in special cases. In versions of our program skipping one bit blocks is not allowed so as to avoid the extra overhead logic needed to change a ripple carry block into a skip block. In another case, running a skip line a long distances without buffering may slow it past the point that it is faster than rippling.

Based on certain assumption, the skip optimization program can, without further knowledge of the remaining part adder's behavior, make a block larger if the carry is delayed by skipping and not by generating. Hence the program increases the size of the module until the generate path and the skip path produce a carry-out at the same time. If the block was made larger, then the carry would be delayed by one ripple delay. This would effect all the skip blocks 'upstream', and though the remaining adder is one bit smaller, only one of the modules in the remaining adder will benefit and actually be one bit smaller. As a consequence of the late carry, the other modules will probably produce a later sum. Hence, under our assumptions, it can not be beneficial to increase the block size beyond this point.

The equations presented by Majerski [30] can be derived by solving for equal path lengths between generate and carry skip, without violating a global sum time.

7.7 Optimization Program Examples

The following shows the transcript from running the optimizing program with no arguments. Doing this just invokes the usage message:

```
bash$ skip
usage: skip <width> <levels> <adder> [-cin <time>]
      or: skip <width> <levels> table <directory> [-cin <time>]

cin_time defaults to 0
adder is one of:  fundamental - thesis, chapter three skip adder
                  Majerski_S1 - S1 adder from Majerski's paper
                  Chan_Schlag_3 - section 3 adder example from Chan and Schlag
```

The program is called *skip*. The first parameter is the width of the desired adder, the second parameter is the number of levels, and the third parameter is the adder type. The adder type is either specified as one built into the program, or it is supplied in the form of an ASCII table of parameters. The tables must be placed in a directory, and that directory name is specified after the key word *table*. Optionally a carry input time may be given. The verbose option gives more information about the adder.

This produces a 256 bit 1 level carry skip adder based on the chapter 4 model, *K*:

```
bash$ skip 256 1 fundamental
width:256      cin: 0.000e+00 sum: 6.400e+01 cout: 6.300e+01
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 15 14 13 12
 11 10 9 8 7 6 5 4 3 2 1)
```

This optimization program has been used to reproduce the results from Majerski[30] and Chan and Schlag [80].

Majerski's adders can be produced by a simple command:

```
bash$ skip_test 41 1 Majerski_S1
width:41      cin: 0.000e+00 sum: 1.200e+01 cout: 1.100e+01
(1 2 3 4 5 6 6 5 4 3 2)
```

Which is identical to the entry in Majerski's figure 3. Multilevel optimization can also be done by specifying more than one level on the command line.

Chan and Schlag suggested using the equations:

$$R(x) = x^2 \quad (7.12)$$

$$S(x) = 3x + 2 \quad (7.13)$$

$$\delta = 1 \quad (7.14)$$

where $R(x) + \delta$ is the generate delay, $S(x) + 1$ is the skip delay, and $R(x)$ is equal to the delay from carry-in to the carry used to make the most latest sum bit - which we will go ahead and assume is the most significant sum bit in the module. Hence the delay through an M group adder is:

$$(R(n_0) + 1) + \sum_{i=1}^{M-1} (S(n_i) + 1) + R(n_M) \quad (7.15)$$

According to their model, there is no carry input, and skipping up to 3 bits is slower than rippling. This breaks two of the assumptions. However, we can force our program to follow this model by setting our carry-in to -4. This forces the output of the first module to appear when it should, had the first module been a ripple carry adder instead of a carry skip adder. Also, since only the first module is in danger of breaking the skip assumption this problem

is also avoided. Hence, we duplicate the result in their example 1, and show a 24 bit adder with delay of 81:

```
bash$ skip_test 24 1 Chan_Schlag_3 -cin -4
width:24      cin: -4.000e+00 sum: 8.100e+01 cout: 8.600e+01
(2 4 6 6 4 2)
```

Chan and Schag's example 2 shows a 64 bit adder with a maximum delay of 216, (2,4,5,7,8,10,8,7,6,4,2). Our optimizer, using the same model and counting delay the same way, did a little better and found a 64 bit adder with delay of 215:

```
bash$ skip_test 64 1 Chan_Schlag_3 -cin -4
width:64      cin: -4.000e+00 sum: 2.150e+02 cout: 2.180e+02
(2 4 6 8 10 10 9 7 5 3)
```

Also, our optimizer appears to be faster, it calculated the previous result in about 60 milliseconds while using 24% of the cpu on a 486 lap top.

```
bash$ time skip_test 64 1 Chan_Schlag_3 -cin -4
width:64      cin: -4.000e+00 sum: 2.150e+02 cout: 2.180e+02
(2 4 6 8 10 10 9 7 5 3)
```

```
0.03user 0.03system 0:00.25elapsed 24%CPU
```

Chapter 8

Alternative Carries

8.1 Relationship between Carry, Propagate, and Generate

Most adders can be drawn as a sequence of constrained bit modules. Such modules can always be placed such that the operands come from the left, and the sums leave at the right, as in figure 8.1. Also, the modules may talk to one another, hence there may be inputs other than just the operands bits. We will call these *gamma* signals.¹ When the adder is drawn as in figure 8.1, these gamma signals will always cross the horizontal line that separates the bit modules.

Because the carry input to the lsb of a module is *exclusive-or* -ed with propagate to make the sum, the value of the carry is always significant, there are no don't care states. The carry-in/propagate partitioning of the sum function is interesting in that propagate is a function only of the input operands to the bit cell, while the carry-in is *not* a function of these bits. It is only a function of the lower significance bits. It follows that carry-in must be derivable from the *gamma* signals described above.

In the case shown in figure 8.1 the input signals crossing the partition line

¹it is possible for a lower significance operand bit to be passed as a gamma signal

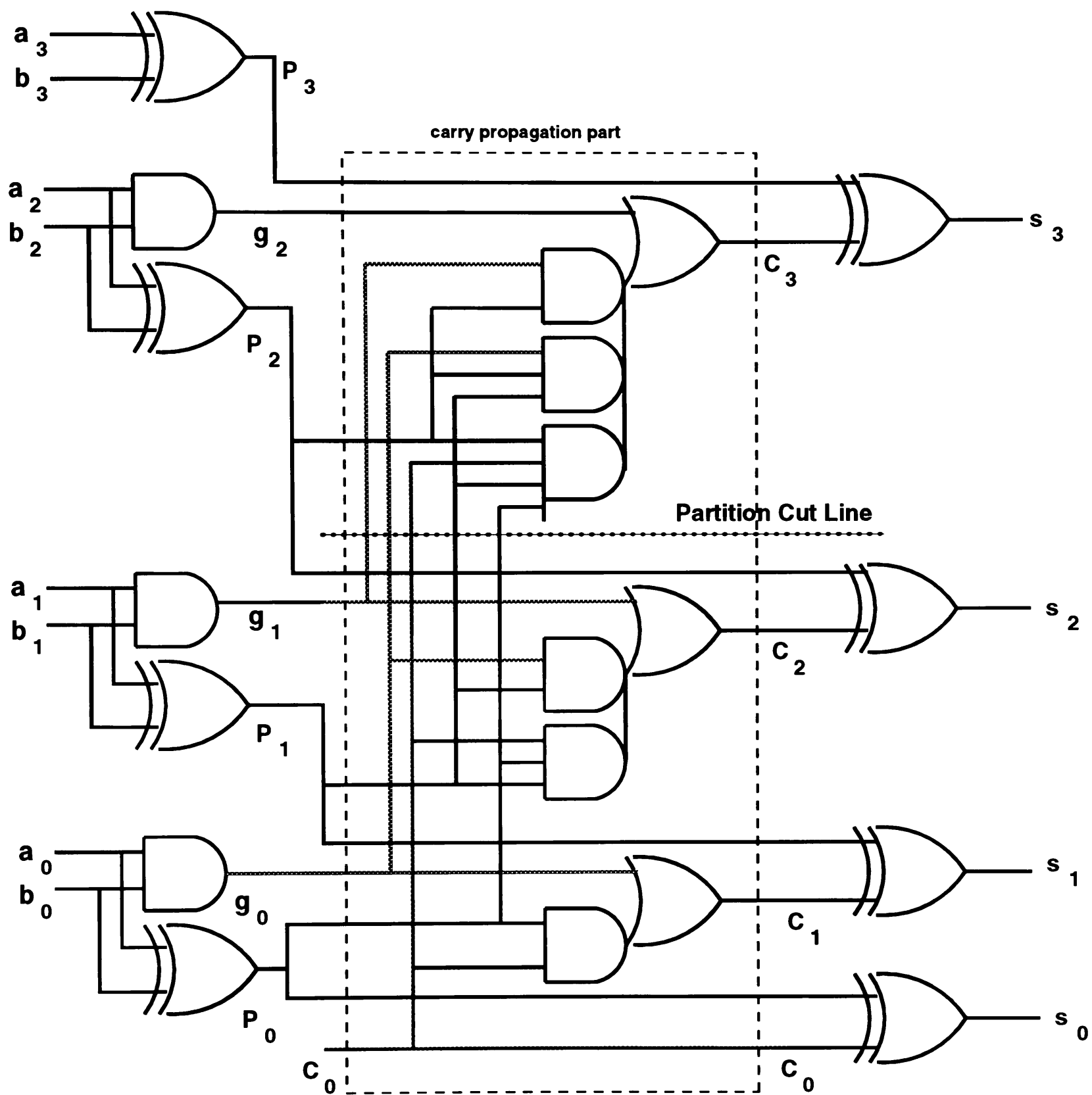


Figure 8.1: Carry Lookahead with Partition

are, g_1, g_0, c_0, p_1, p_0 . The carry-in is c_2 and this can be expressed as:

$$c_2 = g_1 \vee g_0 p_1 \vee c_0 p_1 p_0 \quad (8.1)$$

The following is a list of relationships between propagate, generate, and transfer will be useful in the ensuing discussion.

$$p_i g_i = 0 \quad (8.2)$$

$$c_{i+1} \bar{t}_i = 0 \quad (8.3)$$

$$g_i = c_{i+1} \bar{p}_i \quad (8.4)$$

$$g_i = t_i \bar{p}_i \quad (8.5)$$

$$g_i = g_i t_i \quad (8.6)$$

$$k_i = \bar{t}_i \quad (8.7)$$

$$k_i = \bar{g}_1 \bar{p}_1 \quad (8.8)$$

$$p_i = \bar{g}_1 \bar{k}_i \quad (8.9)$$

where,

$$t_i = a_i \vee b_i \quad (8.10)$$

$$p_i = a_i \nabla b_i \quad (8.11)$$

$$g_i = a_i b_i \quad (8.12)$$

$$k_i = \bar{a}_i \bar{b}_i \quad (8.13)$$

k_i is *carry kill*. Carries do not propagate past a bit with a carry kill. All of these identities fall out of Boolean algebra manipulations on the operand bits.

Based on these identities there are several variations on the conventional carry propagate formula:

$$c_{i+1} = g_i \vee p_i c_i \quad (8.14)$$

$$c_{i+1} = g_i \vee t_i c_i \quad (8.15)$$

$$c_{i+1} = t_i (g_i \vee c_i) \quad (8.16)$$

$$c_{i+1} = g_i \overline{c_i} \vee t_i c_i \quad (8.17)$$

$$c_{i+1} = g_i \vee p_i c_i \quad (8.18)$$

Each of the carry equations 8.14 - 8.14 is maximally local in that all information needed on the previous bits is coalesced into the carry signal, and all information from the current bit is encapsulated in either $\langle p_i, g_i \rangle$ or $\langle t_i, g_i \rangle$.

The completion of the carry can also be procrastinated; so that partial information, as in the case of the carry lookahead adder, can be propagated between bits (i.e. the locality can be reduced further). This leads to a class of adders which includes the conventional carry lookahead adder, and both Ling's and Majerski's variations which are discussed below.

Equation 8.14 is the standard propagate generate equation. In Manchester carry chains, carries are prevented from moving backwards down the carry chain by the exclusive nature of p_i and g_i .

Equation 8.14 is a variation which allows the case of generated carries to overlap with the case of transferred carries. This is also the carry select adder form since generate is equivalent to the zero conditioned carry c_i^0 and the transfer is equivalent to the conditional one conditioned carry c_i^1 . As explained

in chapter 3, a logically equivalent way of writing equation 8.14 is equation 8.14, since $t_i = \text{zero}$ implies that g_i is zero. In general then:

$$g_i = c_i^0 \quad (8.19)$$

$$t_i = c_i^1 \quad (8.20)$$

$$p_i = c_i^1 \overline{c_i^0} \quad (8.21)$$

Equation 8.14 allows the AND and OR functions to be reversed. Since $g_i t_i = g_i$, multiplying through by t_i changes the equation 8.14 to that of 8.14.

Equation 8.14 is the half adder equation. Obaidat and Irshid [81] used a half adder tree to build a interesting, but expensive, ripple carry adder which has one two input gate delay per bit. The use of *exclusive-or* in this equation is logically the same as using *or*, because the two sum terms can never both be true.

In place of propagating a carry signal, one can propagate not carry:

$$\overline{c_{i+1}} = \overline{t_i} \vee \overline{g_i} \overline{c_i} \quad (8.22)$$

$$\overline{c_{i+1}} = k_i \vee \overline{g_i} \overline{c_i} \quad (8.23)$$

$$\overline{c_{i+1}} = k_i \vee p_i \overline{c_i} \quad (8.24)$$

The third equation is derived from the second, based on the fact that $\overline{g_i}$ includes the case of carry kill, which is redundant. Hence, by identity 8.2 $\overline{g_i}$ can be replaced with p_i .

An adder circuit can take advantage of any of the above properties, and others, to gain maximum advantage in the technology of implementation. As described in chapters 6 and 7, these carry combining equations can be used to produce adders of many different configurations ranging from ripple carry to variable block sized carry lookahead trees.

8.2 Majerski's **nor** Gate Ripple Carry Adder

Majerski [77] showed how procrastinating the *or*-ing together of the $p_i c_i$ with g_i , and using equation 8.22 can be used to create a single **nor** gate per stage ripple carry adder. This is advantageous in technologies where wired **or** is an essentially free function.

This is true in CMOS where parallel N-channel pull downs are faster than pulling through a series of transistors; however, to make a sequence of **nor** gates probably requires that alternating stages use a series of transistors in the pull down path.

1. procrastinate combining $p_i c_i$ and g_i
2. use c and \bar{c} in alternating stages

Instead of propagating c_i , Majerski's adder propagates either

$$\langle g_i, \gamma_{i+1}^0 \rangle \quad (8.25)$$

or,

$$\langle g_i, \gamma_{i+1}^1 \rangle \quad (8.26)$$

where

$$\gamma_{i+1}^0 = p_i c_i \quad (8.27)$$

$$\gamma_{i+1}^1 = p_i \overline{c_i} \quad (8.28)$$

The carry can be recovered by:

$$c_{i+1} = g_i \vee \gamma_{i+1}^0 \quad (8.29)$$

$$\overline{c_{i+1}} = k_i \vee \gamma_{i+1}^1 \quad (8.30)$$

Equation 8.27 and 8.27 can be turned into a recursive formula by substituting equations 8.29 and 8.29 for c_i and $\overline{c_i}$ respectively:

$$\begin{aligned} \gamma_{i+1}^0 &= p_i c_i && \text{equation 8.27} \\ \gamma_{i+1}^0 &= p_i (k_{i-1} \vee \gamma_i^1) && \text{substitute 8.29} \\ \gamma_{i+1}^0 &= \overline{p_i} \vee k_{i-1} \vee \gamma_i^1 && \text{DeMorgan's Theorem} \end{aligned} \quad (8.31)$$

$$\begin{aligned} \gamma_{i+1}^1 &= p_i \overline{c_i} && \text{equation 8.27} \\ \gamma_{i+1}^1 &= p_i (g_{i-1} \vee \gamma_i^0) && \text{substitute 8.29} \\ \gamma_{i+1}^1 &= \overline{p_i} \vee g_{i-1} \vee \gamma_i^0 && \text{DeMorgan's Theorem} \end{aligned} \quad (8.32)$$

These manipulations are shown graphically in figure 8.2.

These manipulations reduce the carry path to one three input **nor** gate per stage, as shown in equations 8.31 and 8.32. Majerski's paper [77] shows that at the expense of one more input, the device count for the bit propagate can be reduced; it also employs a clever scheme for deriving the sum from the carries with few devices. Also the fan-in can be reduced to two inputs by combining the bit local terms separate from the carry.

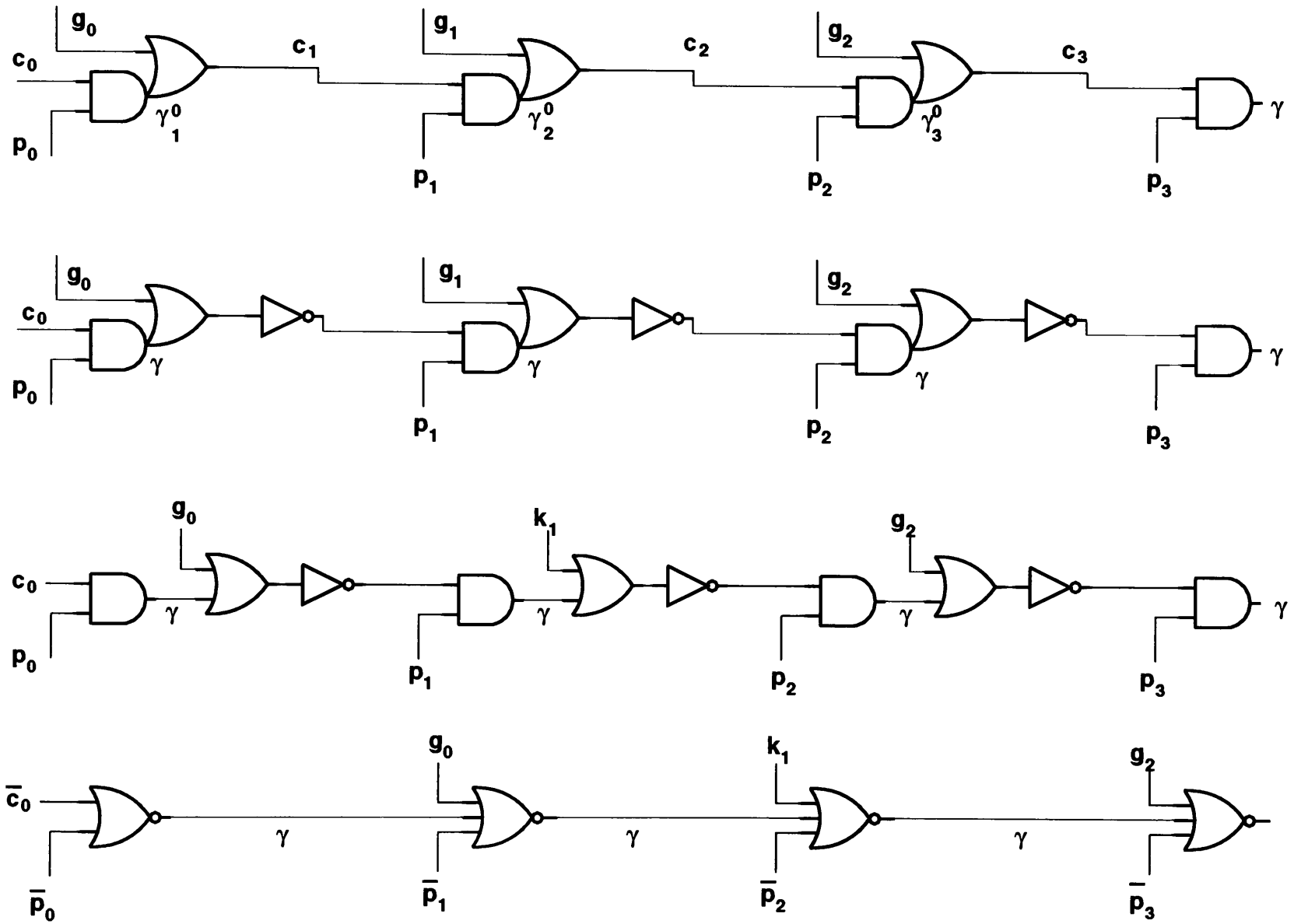


Figure 8.2: Transforming Standard Ripple to Majerski Style Ripple

Although it is possible to apply Majerski's recursion in a tree because it is associative (see chapters 7 and 8) a complication is that the group p, k , and g signals are still required. The adder can be simplified by using t in place of p in the carry path, then k is no longer needed as it is just \bar{t} , but the requirement for group g remains.

8.3 Ling's Adder

Ling suggested using the fco variant given in equation 8.14, and then procrastinating the final **and**. In Ling's paper the subscripts go in the opposite direction, and do not follow the same initial value for carry. In the form consistent with the rest of the thesis we find:

$$H_{i+1} = (g_i \vee c_i) \quad (8.33)$$

and then the pieces to make the carry must be propagated between bits:

$$< t_i, H_{i+1} > \quad (8.34)$$

The carry can be recovered by assembling the pieces:

$$c_{i+1} = t_i H_{i+1} \quad (8.35)$$

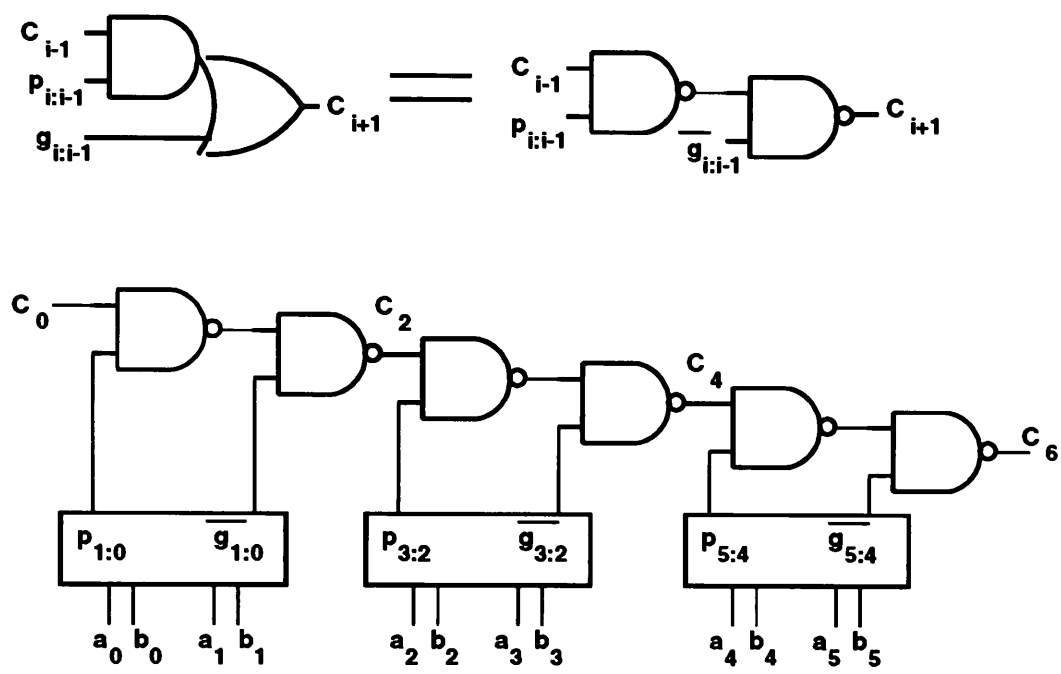
8.4 Other Adders

Reed, et al. [82] presented an adder nearly identical to Majerski's, where the gamma signals were inverted to produce a circuit based on **nand** instead of **nor**.

Vassiliadis [83, 84, 85] also explored Ling's adder and some variations.

One can imagine procrastinating any one or more of the final operations in the carry operation from any of the carry equation variations, and then propagating the partial information in a multitude of signals. When one step is procrastinated in such a way that only signals in bits i and $i + 1$ are needed, the signals display *distance one locality*. Doran [86] reported on a study of all possible ways to send pieces of partial carry information while using two signals and distance one locality.

As a point of reference it is important to note that the conventional propagate generate approach can be used to obtain ripple carry adders of all **nor**, all **nand**, or carry lookahead adders with reduced fan-in. Figure 8.3 shows how the *fco* operation can be used to create a ripple carry adder which has one **nand** gate delay per bit in the carry path. Accordingly, the group propagate and generate signals are created for adjacent 2 bit blocks, then the carry is propagated across the blocks. Since each *fco* operation requires an **and/or** function, and there is one *fco* operation per 2 bit group, there is a net of one **nand** gate delay per bit. This approach is equivalent to building a carry skip adder which has 2 bit groups. A carry skip adder with optimum block sizes would be a further improvement.

Figure 8.3: **nand** per Stage Ripple Carry Adder

Chapter 9

Performance

This chapter gives worst case path lengths for various adders as a method for gauging relative performance. This summarizes the path length discussion which was started in chapter 4, continued in chapter 6 where hierarchical structures were added, and finalized in chapter 7 where the necessary work on carry skip optimization was done. Although path length measures give a first order approximation to speed, finding actual evaluation times requires sample layout and spice simulations.

9.1 Worst Case Path Lengths

The following equations summarize the path length information for various adders discussed in this thesis. The B&K, K&S, and L&S adders are those of Brent and Kung, Kogge and Stone, and Lynch and Swartzlander, respectively.

$$D(\text{ripple}(N), a_0, s_{\text{last}}) = N + 2 \quad (9.1)$$

$$D(\text{CLA}(N), a_0, s_{\text{last}}) = 4\lceil \log_4 N \rceil + 2 \quad (9.2)$$

$$D(\text{ConditionalSum}(N), a_0, s_{\text{last}}) = \lceil \log_2 N \rceil + 2 \quad (9.3)$$

$$D(\text{B\&K}(N), a_0, s_{\text{last}}) = 2\lceil \log_2 N \rceil + 3 \quad (9.4)$$

$$D(\text{K\&S}(N), a_0, s_{\text{last}}) = \lceil \log_2 N \rceil + 4 \quad (9.5)$$

$$D(\text{L\&S}(N), a_0, s_{\text{last}}) = 2\lceil \log_4 N \rceil + 2 \quad (9.6)$$

The ripple carry path length equation was given earlier. The delay through the CLA is derived in [39], and is also apparent from material in chapter 6:

$$D(\text{CLA}(N), pg_0, s_{\text{last}}) = 2\lceil \log_{gs} N \rceil - 1)D(\text{Module}(gs)) + D((XOR)) \quad (9.7)$$

If we allow a fan-in of six gates, two gate delays are required for a signal to pass through the 4 bit module, one more gate delay to make the bitwise propagate and generate, and 3 gate delays to make the sum from carry-out. An optimized circuit may be able to hide the large fan-in.

The conditional sum adder algorithm was described in detail in the chapters 3, and was revisited in chapter 5. It became apparent in chapter 6 that the carry delay through the adder is the same as for a binary prefix tree. The sum delay is just one later, provided that a very large multiplexor select is one delay.

For comparison we chose to show one and four level carry skip adders. The carry skip times come from our optimization program using a variation on the $K(N)$ model, i.e., short ripple carry blocks are sometimes faster than a carry skip module, so we allowed for ripple carry blocks. These are signified in the output as a series of width one blocks.

The results for the one level carry skip adder given in Table 9.1 were generated by looping on the following command, where \$w is replaced with the width of the adder:

Table 9.1: Evaluation Time for One Level Skip Adders

width	time to sum	distribution
3	6	(1 1 1)
5	7	(1 1 2 1)
7	8	(1 1 2 2 1)
10	9	(1 1 2 3 2 1)
13	10	(1 1 2 3 3 2 1)
17	11	(1 1 2 3 4 3 2 1)
21	12	(1 1 2 3 4 4 3 2 1)
26	13	(1 1 2 3 4 5 4 3 2 1)
31	14	(1 1 2 3 4 5 5 4 3 2 1)
37	15	(1 1 2 3 4 5 6 5 4 3 2 1)
43	16	(1 1 2 3 4 5 6 6 5 4 3 2 1)
50	17	(1 1 2 3 4 5 6 7 6 5 4 3 2 1)
57	18	(1 1 2 3 4 5 6 7 7 6 5 4 3 2 1)
65	19	(1 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1)

skip \$w 1 fundamental

The results for a 4 level carry skip adder optimization (given in Table 9.2) were generated by looping on the following command, where \$w is replaced with the width of the adder:

skip \$w 4 fundamental

The delays through the Kogge & Stone adder, and the Brent & Kung adder were given in chapter 6. Here we have added an appropriate number of gate delays for propagate generate formation and for sum formation. The Lynch and Swartzlander adder is based on dynamic Manchester carry chains which have large fan-in. Since these chains have the same functionality as the CLA block, we assigned them two gate delays.

Table 9.2: Evaluation Time for Four Level Skip Adders

width	time to sum	distribution
3	6	((1)(1 1))
6	7	((1)(1 1)((1 1)(1)))
10	8	((1)((1)(1))(1 2 1)(2 1))
15	9	((1)((1)(1 1))(1 1 2)((1 2 1)(2 1)))
22	10	((1)((1)(1 1))((1 1)(1 1 1))(1 1 2 1)(3 2)(2 1))
32	11	((1)((1)((1)(1 1)))((1)(1 1)(1 1 1))(1 2 3 2)(2 3 2)(3 2 1))

The worst case path lengths are compared in Table 9.3.

Experience has shown that the devices in any of the log time adders can be sized to evaluate quickly. Although in the limit, one of the adders will have superior performance, the question really is which adder is the smallest while still evaluating fast enough. Here, the carry skip adder has an advantage. This is not surprising since the number of levels, group size, and driver size parameters can be set to imitate any of the folded adders.

Table 9.3: Worst Case Path Lengths Through Various Adders

Bits	Cond	L&S	K&S	skip-4	CLA	B&K	skip-1
4	4	4	6	7	6	7	7
8	5		7	8		9	9
16	6	6	8	10	10	11	11
32	7		9	11		13	15
64	8	8	10	13	14	15	19

Chapter 10

Conclusion

This thesis has dealt with fundamental concepts of addition and optimization.

We presented a number of interesting results. In chapter 2 we showed how mechanical addition follows from the rules for manipulating cardinal numbers. In this chapter we also gave a foundation for studying the fundamental physics of addition by showing the connection between addition and a quantum ‘particle in a box’ problem.

In chapter 3 we gave maximally parallel algorithms for conventional adder designs. In chapter 4 we formalized the idea of counting gate delays by introducing operators on a net lists. In chapter 9 we used this information for a first order approximation of relative performance of the adders.

In chapters 5 and 6 we unified the conventional adder designs by identifying an associative adder partitioning operator, and an associative carry operator. The carry operator was the part of the adder partitioning that handled the carry signals. In chapter 6 we demonstrated how all the propagate generate adder designs can be built by varying the associativity of the carry operator. This chapter finalized the argument that adder designs can be reduced to a generalized structure and a set of optimization parameters which determine the specific structure.

In chapter 7 we showed a solution to the optimization problem. The code for the carry skip optimization algorithm we developed is freely available on the world wide web (<http://devil.ece.utexas.edu/lynch>).

In chapter 8 we discussed variations of the carry operator used in chapter 7. We discovered that the conventional carry can always be recovered from signals which cross a partition line which divides an adder into two parts. Since the equation for this carry can always be written in the recognized form as a function of the local bit propagate and generate and the global previous carry, there is fundamentally one carry operator. This operator can work with information in redundant form, or the logic for it can be partitioned in different ways, resulting in variations such as those of Ling and Majerski.

An area of further work is expanding the optimization program to networks which are not folded (folding is described in chapter 6). The simplifying assumptions should also be removed. Finally, more work needs to be done with actual technology data; although the program supports technology tables, the potential here was not been fully explored. Such an improved program could be the basis of a useful optimum adder generator.

Bibliography

- [1] J. Miller, B. Roberts, and P. Madland, “High performance circuits for the i486 processor,” in *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 188–192, IEEE, 1989.
- [2] B. Ryan, “Alpha rides high,” *Byte*, vol. 19, pp. 197–199, Oct 1994.
- [3] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. Shen, “Instruction level profiling and evaluation of the IBM RS/6000,” *Computer Architecture News*, pp. 180–189, May 1991.
- [4] M. Obaidat, H. Khalid, and K. Sadiq, “A methodology for evaluating the performance of CISC computer systems under single and two-level cache environments,” *Microprocessing and Microprogramming*, pp. 411–426, July 1994.
- [5] L. K. John, V. Reddy, P. T. Hulina, and L. D. Craor, “Program balance and its impact on high performance RISC architectures,” in *Proceedings of 1st IEEE Symposium on High Performance Computer Architecture*, pp. 370–379, IEEE Computer Society Press, 1995.
- [6] S. Vajapeyam and W. C. Hsu, “Toward effective scalar hardware for highly vectorizable applications,” *Journal of Parallel and Distributed Computing*, vol. 19, pp. 147–162, Nov 1993.

- [7] B. Randell, ed., *The Origins of Digital Computers: Selected Papers*. Springer-Verlag, 1973.
- [8] H. W. Buxton, *Memoir of the Life and Labours of the Late Charles Babbage Esq., F.R.S.*, vol. 13 of *Charles Babbage Institute reprint series for the history of computing*. The MIT Press and Tomash Publishers, 1988.
- [9] M. V. Wilkes, *Automatic Digital Computers*. New York: Wiley, 1956.
- [10] S. Rosen, "Electronic computers: A historical survey," *Computing Surveys*, vol. 1, pp. 7–35, Mar 1969.
- [11] M. V. Wilkes, "The ENIAC - high-speed electronic calculating machine," *Electronic Engineering*, vol. 230, pp. 104–108, Apr 1947.
- [12] N. Sloane and A. D. Wyner, eds., *Claude Elwood Shannon Collected Papers*. IEEE Press, 1993.
- [13] K. Zuse, *The Computer - My life*. Springer-Verlag, 1991.
- [14] A. Burks, H. Goldstine, and J. V. Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," in *Report Prepared for U.S. Army Ordinance Department*, U.S. Government Report, 1946.
- [15] H. Goldstine and J. V. Neumann, "Planning and coding of problems for an electronic computing instrument," in *Report Prepared for U.S. Army Ordinance Department*, U.S. Government Report, 1947.
- [16] A. H. Taub, *John Von Neumann, Collected Works*. New York: Pergamon Press, 1961-1963.
- [17] I. Koren, *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

- [18] E. E. Swartzlander, Jr., ed., *Computer Arithmetic*, vol. 1. IEEE Computer Society Press, 1990.
- [19] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Publishing Company, 1985.
- [20] S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*. Holt, Rinehart and Winston, 1982.
- [21] K. Hwang, *Computer Arithmetic Principles, Architecture, and Design*. John Wiley and Sons, 1979.
- [22] U. Ofman, "On the algorithmic complexity of discrete functions," *Dokl. Akad. Nauk USSR*, vol. 145, no. 1, pp. 48–51, 1962.
- [23] B. E. Briley, "Some new results on average worst case carry," *IEEE Transactions on Computers*, vol. C-22, pp. 459–463, May 1973.
- [24] O. N. Garcia, H. Glass, and S. C. Haimes, "An approximation and empirical study of the distribution of adder inputs and maximum carry length propagation," in *4th Symposium on Computer Arithmetic*, pp. 97–103, IEEE, Oct 1978.
- [25] V. I. Varshavskii, L. Y. Rozenblyum, and N. A. Starodubtsev, "Mean generation time for carry signals in aperiodic counter and adder circuits," *Automatic Control and Computer Sciences*, vol. 9, pp. 75–77, Jul 1975.
- [26] B. Gilchrist, J. H. Pomerene, and S. Y. Wong, "Fast carry logic for digital computers," *IRE Transactions on Electronic Computers*, pp. 133–136, Apr 1955.

- [27] A. Martin, "Asynchronous datapaths and the design of an asynchronous adder," *Formal Methods in System Design*, vol. 1, pp. 117–137, Jul 1992.
- [28] M. Lehman and N. Burla, "Skip techniques for high-speed carry-propagation in binary arithmetic units," *IRE Transactions on Electronic Computers*, pp. 691–698, Dec 1961.
- [29] C. P. Morgan and D. B. Jarvis, "Transistor logic using current switching and routing techniques and its application to a fast-carry propagation adder," *Proceedings of the IEE*, vol. 106 pt. B, pp. 467–468, Sep 1959.
- [30] S. Majerski, "On determination of optimal distributions of carry skips in adders," *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 45–58, Feb 1967.
- [31] V. G. Oklobdzija and E. R. Barnes, "Some optimal schemes for ALU implementation in VLSI technology," in *Proceedings - 7th Symposium on Computer Arithmetic*, pp. 2–8, IEEE, Jun 1985.
- [32] A. Guyot, B. Hochet, and J. Muller, "A way to build efficient carry-skip adders," *IEEE Transactions on Computers*, vol. C-36, pp. 1144–1152, Oct 1987.
- [33] V. G. Oklobdzija, "Simple and efficient CMOS circuit for fast VLSI adder realization," in *Proceedings - IEEE International Symposium on Circuits and Systems*, pp. 235–238, IEEE, Jun 1988.
- [34] S. Turrini, "Optimal group distribution in carry-skip adders," in *Proceedings - 9th Symposium on Computer Arithmetic*, pp. 96–103, IEEE, Sep 1989.

- [35] P. K. Chan and M. D. F. Schlag, "Analysis and design of CMOS Manchester adders with variable carry-skip.," in *Proceedings - 9th Symposium on Computer Arithmetic*, pp. 86–95, IEEE, Sep 1989.
- [36] T. Kilburn, D. B. G. Edwards, and D. Aspinall, "Parallel addition in digital computers: A new fast "carry" circuit," *IEE Proceedings*, vol. 106 pt. B, pp. 464–466, 1959.
- [37] T. Rhyne, "Limitations on carry lookahead networks," *IEEE Transactions on Computers*, vol. C-33, pp. 373–374, Apr 1984.
- [38] A. Weinberger and J. L. Smith, "A logic for high-speed addition," *National Bureau of Standards Circular 591*, pp. 3–12, 1958.
- [39] O. MacSoreley, "High-speed arithmetic in a binary computer," *IRE Proceedings*, vol. 49, pp. 67–91, 1961.
- [40] R. L. DAVIS, "ILLIAC IV processing element," *IEEE Transactions on Computers*, vol. C-18, pp. 800–816, Sept 1969.
- [41] A. Weinberger, "High-speed programmable logic array adders," *IBM Journal of Research and Development*, vol. 23, pp. 163–178, Mar 1979.
- [42] A. Weinberger, "Carry look-ahead address using static pass transistors," *IBM Technical Disclosure Bulletin*, vol. 26, pp. 1621–1623, Aug 1983.
- [43] R. Bechade and W. K. Hoffman, "Generalized 2 bit slice ALU," in *Proceedings - IEEE International Conference on Circuits and Computers*, pp. 1094–1098, IEEE, Oct 1980.

- [44] D. G. Crawley and G. A. J. Amaratunga, "Pipelined carry look-ahead adder," *Electronics Letters*, vol. 22, pp. 661–662, Jun 1986.
- [45] I. S. Hwang and P. S. Magarschack, "High-speed dynamically reconfigurable 32-bit CMOS adder," in *Proceedings of the IEEE 1988 Custom Integrated Circuits Conference*, pp. 17.5/1–6, IEEE, 1988.
- [46] B. S. Fagin, "Fast addition of large integers," *IEEE Transactions on Computers*, vol. 41, pp. 1069–1077, Sep 1992.
- [47] S. Winograd, "On the time required to perform addition," *J. ACM*, vol. 12, pp. 277–285, Apr 1965.
- [48] Y. Lee, I. Park, and C. Kyung, "Design of compact static CMOS carry look-ahead adder using recursive output property," *Electronics Letters*, vol. 29, pp. 794–796, Apr 1993.
- [49] O. J. Bedrij, "Carry-select adder," *IRE Transactions on Electronic Computers*, pp. 340–346, Jun 1962.
- [50] T. Lynch and E. E. Swartzlander, Jr., "The redundant cell adder," in *Proceedings - 10th IEEE Symposium on Computer Arithmetic*, pp. 165–170, IEEE, Jun 1991.
- [51] T. Lynch and E. E. Swartzlander, Jr., "A spanning tree carry lookahead adder," *IEEE Transactions on Computers*, vol. 41, pp. 931–939, Aug 1992.
- [52] V. Kantabutra, "A recursive carry-lookahead/carry-select hybrid adder," *IEEE Transactions on Computers*, vol. 42, pp. 1495–1499, Dec 1993.

- [53] E. E. Swartzlander and R. H. Nigaglioni, "Variable spanning tree adder," in *29th Asilomar Conference on Signals, Systems and Computers*, 1995.
- [54] J. Sklansky, "Conditional sum addition logic," *IRE Transactions on Electronic Computers*, vol. EC-9, pp. 226–231, 1960.
- [55] T. G. Hallin and M. J. Flynn, "Pipelining of arithmetic functions," *IEEE Transactions on Computers*, vol. C-21, pp. 880–886, Aug 1972.
- [56] A. Tyagi, "A reduced area scheme for carry-select adders," in *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pp. 255–258, IEEE, Sep 1990.
- [57] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, pp. 786–793, Aug 1973.
- [58] S. H. Unger, "Tree realizations of iterative circuits," *IEEE Transactions on Computers*, vol. C-26, pp. 365–383, Apr 1977.
- [59] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the Association for Computing Machinery*, vol. 27, pp. 831–838, Oct 1980.
- [60] R. P. Brent and H. T. Kung, "Regular layout for parallel adders," *IEEE Transactions on Computers*, vol. C-31, pp. 260–264, Mar 1982.
- [61] R. Montoye, "Area-time efficient addition in charge based technology," in *ACM IEEE Eighteenth Design Automation Conference Proceedings*, pp. 862–872, Jun 1981.

- [62] T. F. Ngai and M. J. Irwin, "Regular, area-time efficient carry-lookahead adders," in *Proceedings - 7th Symposium on Computer Arithmetic*, pp. 9–15, IEEE, Jun 1985.
- [63] Y. F. Chen and B. W. Y. Wei, "QAC: a CMOS implementation of the 32-bit Q adder," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pp. 49–52, IEEE, Oct 1985.
- [64] B. W. Y. Wei, C. D. Thompson, and Y. F. Chen, "Time-optimal design of a CMOS adder," in *19th Asilomar Conference on Circuits, Systems & Computers*, pp. 186–191, IEEE, Nov 1986.
- [65] B. W. Y. Wei, "A high-speed multiplier using a variable-block carry lookahead adder," in *24th Asilomar Conference on Signals, Systems and Computers*, pp. 907–912, Nov 1990.
- [66] B. W. Y. Wei and C. D. Thompson, "Area-time optimal adder design," *IEEE Transactions on Computers*, vol. 39, pp. 666–675, May 1990.
- [67] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," in *Proceedings - 8th Symposium on Computer Arithmetic*, pp. 49–56, IEEE, Oct 1987.
- [68] T. Han, D. A. Carlson, and S. P. Levitan, "VLSI design of high-speed, low-area addition circuitry," in *Proceedings - 1987 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pp. 418–422, IEEE, Oct 1987.

- [69] V. Sugla and D. A. Carlson, "Extreme area-time tradeoffs in VLSI," *IEEE Transactions on Computers*, vol. C-39, pp. 251–257, Feb 1990.
- [70] B. D. Lee and V. G. Oklobdzija, "Improved CLA scheme with optimized delay," *Journal of VLSI Signal Processing*, vol. 3, pp. 265–274, Oct 1991.
- [71] B. D. Lee and V. G. Oklobdzija, "Optimization and speed improvement analysis of carry-lookahead adder structure," in *24th Asilomar Conference on Signals, Systems and Computers*, pp. 918–922, Nov 1990.
- [72] J. P. Fishburn, "A depth-decreasing heuristic for combinational logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between," in *Proceedings - 27th ACM/IEEE Design Automation Conference*, pp. 361–364, IEEE, Jun 1990.
- [73] J. Hsu and O. Bair, "A compiler for optimal adder design," in *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pp. 25.6/1–4, IEEE, May 1992.
- [74] S. C. Kleene, *Introduction to MetaMathematics*. New York: Elsevier Science Publishing Company, Inc., 1952–1991.
- [75] T. Lynch, "The energy content of knowledge," *Workshop on Physics and Computation*, pp. 78–82, Oct 1992.
- [76] J. Lotz, B. Miller, E. Delano, J. Lam, M. Forsyth, and T. Hotchkiss, "A CMOS RISC CPU designed for sustained high performance on large applications," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1190–1198, Oct 1990.

- [77] S. Majerski, "High-speed computer arithmetic," Tech. Rep. 631, Institute of Computer Science, Polish Academy of Science, Jun 1978.
- [78] C. W. Weller, "High-speed carry circuit for binary adders," *IEEE Transactions on Computers*, vol. C-18, pp. 728–732, Aug 1969.
- [79] B. Becker and R. Kolla, "A regular layout for parallel adders," in *Proceedings of the 5th Annual Symposium on Theoretical Aspects of Computer Science*, pp. 18–28, Springer Verlag, Feb 1988.
- [80] P. K. Chan and M. D. F. Schlag, "Analysis and design of CMOS Manchester adders with variable carry-skip," *IEEE Transactions on Computers*, vol. C-39, pp. 983–992, Aug 1990.
- [81] M. S. Obaidat and M. I. Irshid, "Fast multi-step addition algorithm," *International Journal of Electronics*, vol. 70, pp. 839–849, May 1991.
- [82] I. S. Reed, B. Sharma, M. T. Shih, J. Bailey, and T. K. Truong, "VLSI implementation of GSC architecture with a new ripple carry adder," in *IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pp. 520–523, IEEE, Oct 1988.
- [83] S. Vassiliadis, "Comparison between adders with new defined carries and traditional schemes for addition," *International Journal of Electronics*, vol. 64, pp. 617–626, Apr 1988.
- [84] S. Vassiliadis, D. S. Lemon, and M. Putrino, "S/370 sign-magnitude floating-point adder," *IEEE Journal of Solid-State Circuits*, vol. 24, pp. 1062–1070, Aug 1989.

- [85] S. Vassiliadis, "Recursive equations for hardwired binary adders," *International Journal of Electronics*, vol. 67, pp. 201–213, Aug 1989.
- [86] R. W. Doran, "Variants of an improved carry look-ahead adder," *IEEE Transactions on Computers*, vol. C-37, pp. 1110–1113, Sep 1988.

This digitized document does not include the vita page from the original.

