Diss. ETH No. 12480

Binary Adder Architectures for Cell-Based VLSI and their Synthesis

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of technical sciences

presented by
RETO ZIMMERMANN
Dipl. Informatik-Ing. ETH
born 17. 9. 1966
citizen of Vechigen BE

accepted on the recommendation of
Prof. Dr. W. Fichtner, examiner
Prof. Dr. L. Thiele, co-examiner

1997

Acknowledgments

I would like to thank my advisor, Prof. Wolfgang Fichtner, for his overall support and for his confidence in me and my work. I would also like to thank Prof. Lothar Thiele for reading and co-examining the thesis.

I am greatly indebted in Hubert Kaeslin and Norbert Felber for their encouragement and support during the work as well as for proofreading and commenting on my thesis. I also want to express my gratitude to all colleagues at the Integrated Systems Laboratory who contributed to the perfect working environment. In particular, I want to thank the secretaries for keeping the administration, Hanspeter Mathys and Hansjörg Gisler the installations, Christoph Wicki and Adam Feigin the computers, and Andreas Wieland the VLSI design tools running.

I want to thank Hanspeter Kunz and Patrick Müller for the valuable contributions during their student projects. Also, I am grateful to Rajiv Gupta, Duncan Fisher, and all other people who supported me during my internship at Rockwell Semiconductor Systems in Newport Beach, CA.

I acknowledge the financial support of MicroSwiss, a Microelectronics Program of the Swiss Government.

Finally my special thanks go to my parents for their support during my education and for their understanding and tolerance during the last couple of years.

Contents

Abstract

The addition of two binary numbers is the fundamental and most often used arithmetic operation on microprocessors, digital signal processors (DSP), and data-processing application-specific integrated circuits (ASIC). Therefore, binary adders are crucial building blocks in very large-scale integrated (VLSI) circuits. Their efficient implementation is not trivial because a costly carry-propagation operation involving all operand bits has to be performed.

Many different circuit architectures for binary addition have been proposed over the last decades, covering a wide range of performance characteristics. Also, their realization at the transistor level for full-custom circuit implementations has been addressed intensively. However, the suitability of adder architectures for cell-based design and hardware synthesis — both prerequisites for the ever increasing productivity in ASIC design — was hardly investigated.

Based on the various speed-up schemes for binary addition, a comprehensive overview and a qualitative evaluation of the different existing adder architectures are given in this thesis. In addition, a new multilevel carry-increment adder architecture is proposed. It is found that the ripple-carry, the carry-lookahead, and the proposed carry-increment adders show the best overall performance characteristics for cell-based design.

These three adder architectures, which together cover the entire range of possible area vs. delay trade-offs, are comprised in the more general prefix adder architecture reported in the literature. It is shown that this universal and flexible prefix adder structure also allows the realization of various customized adders and of adders fulfilling arbitrary timing and area constraints.

A non-heuristic algorithm for the synthesis and optimization of prefix adders is proposed. It allows the runtime-efficient generation of area-optimal adders for given timing constraints.

Zusammenfassung

Die Addition zweier binärer Zahlen ist die grundlegende und am meisten verwendete arithmetische Operation in Mikroprozessoren, digitalen Signalprozessoren (DSP) und datenverarbeitenden anwendungsspezifischen integrierten Schaltungen (ASIC). Deshalb stellen binäre Addierer kritische Komponenten in hochintegrierten Schaltungen (VLSI) dar. Deren effiziente Realisierung ist nicht trivial, da eine teure *carry-propagation* Operation ausgeführt werden muss.

Eine Vielzahl verschiedener Schaltungsarchitekturen für die binäre Addition wurden in den letzten Jahrzehnten vorgeschlagen, welche sehr unterschiedliche Eigenschaften aufweisen. Zudem wurde deren Schaltungsrealisierung auf Transistorniveau bereits eingehend behandelt. Andererseits wurde die Eignung von Addiererarchitekturen für zellbasierte Entwicklungstechniken und für die automatische Schaltungssynthese — beides Grundvoraussetzungen für die hohe Produktivitätssteigerung in der ASIC Entwicklung — bisher kaum untersucht.

Basierend auf den mannigfaltigen Beschleunigungstechniken für die binäre Addition wird in dieser Arbeit eine umfassende Übersicht und ein qualitativer Vergleich der verschiedenen existierenden Addiererarchitekturen gegeben. Zudem wird eine neue *multilevel carry-increment* Addiererarchitektur vorgeschlagen. Es wird gezeigt, dass der *ripple-carry*, der *carry-lookahead* und der vorgeschlagene *carry-increment* Addierer die besten Eigenschaften für die zellbasierte Schaltungsentwicklung aufweisen.

Diese drei Addiererarchitekturen, welche zusammen den gesamten Bereich möglicher Kompromisse zwischen Schaltungsfläche und Verzögerungszeit abdecken, sind in der allgemeineren *Prefix*-Addiererarchitektur enthalten, die in der Literatur beschrieben ist. Es wird gezeigt, dass diese universelle und flexible Prefix-Addiererstruktur die Realisierung von verschiedensten spezial-

isierten Addierern mit beliebigen Zeit- und Flächenanforderungen ermöglicht.

Ein nicht-heuristischer Algorithmus für die Synthese und die Zeitoptimierung von Prefix-Addierern wird vorgeschlagen. Dieser erlaubt die recheneffiziente Generierung flächenoptimaler Addierer unter gegebenen Anforderungen and die Verzögerungszeit.

1

Introduction

1.1 Motivation

The core of every microprocessor, digital signal processor (DSP), and data-processing application-specific integrated circuit (ASIC) is its data path. It is often the crucial circuit component if die area, power dissipation, and especially operation speed are of concern. At the heart of data-path and addressing units in turn are arithmetic units, such as comparators, adders, and multipliers. Finally, the basic operation found in most arithmetic components is the binary addition. Besides of the simple addition of two numbers, adders are also used in more complex operations like multiplication and division. But also simpler operations like incrementation and magnitude comparison base on binary addition.

Therefore, binary addition is the most important arithmetic operation. It is also a very critical one if implemented in hardware because it involves an expensive carry-propagation step, the evaluation time of which is dependent on the operand word length. The efficient implementation of the addition operation in an integrated circuit is a key problem in VLSI design.

Productivity in ASIC design is constantly improved by the use of cell-based design techniques — such as standard cells, gate arrays, and field-programmable gate arrays (FPGA) — and by low- and high-level hardware synthesis. This asks for adder architectures which result in efficient cell-based circuit realizations which can easily be synthesized. Furthermore, they should provide enough flexibility in order to accommodate custom timing and area constraints as well as to allow the implementation of customized adders.

1.2 Related Work

Much work has been done and many publications have been written on circuit architectures for binary addition. Different well-known adder architectures are widely used and can be found in any book on computer arithmetic [Kor93, Cav84, Spa81, Hwa79, Zim97]. Many adder circuit implementations at the transistor level are reported in the literature which use a variety of different adder architectures and combinations thereof [D$^+$92, G$^+$94, M$^+$94, OV95, O$^+$95, M$^+$91].

On the other hand, a systematic overview of the basic addition speed-up techniques with their underlying concepts and relationships can hardly be found. This, however, is a prerequisite for optimal adder implementations and versatile synthesis algorithms. Furthermore, optimality of adder architectures for cell-based designs was not investigated intensively and comprehensive performance comparisons were carried out only marginally [Tya93].

Most work so far has focused on the standard two-operand addition. The efficient realization of customized adders — such as adders with flag generation, non-uniform signal arrival times [Okl94], fast carry-in processing, modulo [ENK94] and dual-size adders — were not considered widely.

Finally, the synthesis of adder circuits was addressed only marginally up to now. This is because the generation of fixed adder architectures is rather straightforward and because no efficient synthesis algorithms for flexible adder architectures were known. Exceptions are some publications on the computation of optimal block sizes e.g. for carry-skip adders [Tur89] and on heuristic algorithms for the optimization of parallel-prefix adders [Fis90, GBB94].

1.3 Goals of this Work

As a consequence, the following goals have been formulated for this work:

- Establish an overview of the basic addition speed-up schemes, their characteristics, and their relationships.
- Derive all possible adder architectures from the above speed-up schemes and compare them qualitatively and quantitatively with focus on cell-based circuit implementation, suitability for synthesis, and realization of customized adders.
- Try to unify the different adder architectures as much as possible in order to come up with more generic adder structures. The ideal solution would be a flexible adder architecture covering the entire range of possible area-delay trade-offs with minor structural changes.
- Elaborate efficient and versatile synthesis algorithms for the best performing adder architectures found in the above comparisons. The ideal solution would consist of one universal algorithm for a generic adder architecture, which takes automatically into account arbitrary timing and area constraints.
- Incorporate the realization and generation of customized adders into the above adder architectures and synthesis algorithms.
- Address other important VLSI aspects — such as circuit verification, layout topologies, and pipelining — for the chosen adder architectures.

1.4 Structure of the Thesis

As a starting point, the basic conditions and their implications are summarized in Chapter 2. It is substantiated why cell-based combinational carry-propagate adders and their synthesis are important in VLSI design and thus worthwhile to be covered by this thesis.

Chapter 3 introduces the basic addition principles and structures. This includes 1-bit and multi-operand adders as well as the formulation of carry-propagation as a prefix problem and its basic speed-up principles.

The different existing adder architectures are described in Chapter 4. In addition, a new carry-increment adder architecture is introduced. Qualitative and quantitative comparisons are carried out and documented on the basis of a unit-gate model and of standard-cell implementations. It is shown that the best-performing adders are all prefix adders.

The implementation of special adders using the prefix adder architecture is treated in Chapter 5.

In Chapter 6, synthesis algorithms are given for the best-performing adder architectures. Also, an efficient non-heuristic algorithm is presented for the synthesis and optimization of arbitrary prefix graphs used in parallel-prefix adders. An algorithm for the verification of prefix graphs is also elaborated.

Various important VLSI aspects relating to the design of adders are summarized in Chapter 7. These include verification, transistor-level design, and layout of adder circuits, library aspects for cell-based adders, pipelining of adders, and the realization of adder circuits on FPGAs.

Finally, the main results of the thesis are summarized and conclusions are drawn in Chapter 8.

2

Basic Conditions and Implications

This chapter formulates the motivation and goals as well as the basic conditions for the work presented in this thesis by answering the following questions: Why is the efficient implementation of combinational carry-propagate adders important? What will be the key layout design technologies in the future, and why do cell-based design techniques — such as standard cells — get more and more importance? How does submicron VLSI challenge the design of efficient combinational cell-based circuits? What is the current status of high- and low-level hardware synthesis with respect to arithmetic operations and adders in particular? Why is hardware synthesis — including the synthesis of efficient arithmetic units — becoming a key issue in VLSI design? How can area, delay, and power measures of combinational circuits be estimated early in the design cycle? How can the performance and complexity of adder circuits be modeled by taking into account architectural, circuit, layout, and technology aspects?

Although some of the following aspects can be stated for VLSI design in general, the emphasis will be on the design of arithmetic circuits.

2.1 Arithmetic Operations and Units

The tasks of a VLSI chip — whether as application-specific integrated circuit (ASIC) or as general-purpose microprocessor — are the processing of data and the control of internal or external system components. This is typically done by algorithms which base on logic and arithmetic operations on data items.

2.1.1 Applications

Applications of arithmetic operations in integrated circuits are manifold. Microprocessors and digital signal processors (DSPs) typically contain adders and multipliers in their data path, forming dedicated integer and/or floating-point units and multiply-accumulate (MAC) structures. Special circuit units for fast division and square-root operations are sometimes included as well. Adders, incrementers/decrementers, and comparators are arithmetic units often used for address calculation and flag generation purposes in controllers.

Application-specific ICs use arithmetic units for the same purposes. Depending on their application, they even may require dedicated circuit components for special arithmetic operators, such as for finite field arithmetic used in cryptography, error correction coding, and signal processing.

2.1.2 Basic arithmetic operations

The arithmetic operations that can be computed in electronic equipment are (ordered by increasing complexity, see Fig. 2.1) [Zim97]:

- shift / extension operations
- equality and magnitude comparison
- incrementation / decrementation
- complementation (negation)
- addition / subtraction
- multiplication
- division
- square root
- exponentiation
- logarithmic functions
- trigonometric and inverse trigonometric functions

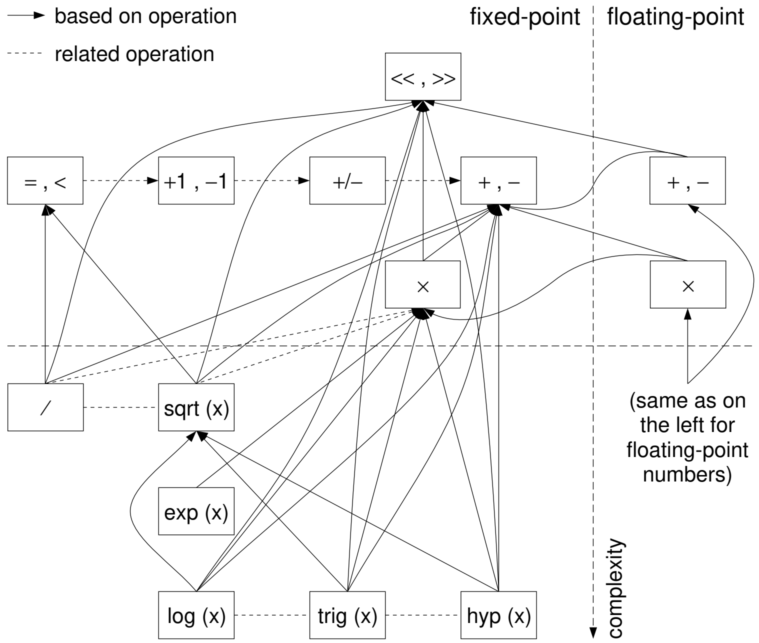


Figure 2.1: *Dependencies of arithmetic operations.*

- hyperbolic functions

For trigonometric and logarithmic functions as well as exponentiation, various iterative algorithms exist which make use of simpler arithmetic operations. Multiplication, division and square root extraction can be performed using serial or parallel methods. In both methods, the computation is reduced to a sequence of conditional additions/subtractions and shift operations. Existing speed-up techniques try to reduce the number of required addition/subtraction operations and to improve their speed. Subtraction corresponds to the addition of a negated operand.

The addition of two n-bit numbers itself can be regarded as an elementary operation. In fact, decomposition into a series of increments and shifts is possible but of no relevance. The algorithm for complementation (negation) of a number depends on the chosen number representation, but is usually accomplished by bit inversion and incrementation. Incrementation and decrementation are simplified additions with one input operand being constantly 1 or -1. Equality and magnitude comparison operations can also be regarded as simplified additions, where only some the respective addition flags, but no sum bits are used as outputs. Finally, shift by a constant number of bits and extension operations, as used in some of the above more complex arithmetic functions, can be accomplished by appropriate wiring and thus require no additional hardware.

This short overview shows that the addition is the key arithmetic operation, which most other operations are based on. Its implementation in hardware is therefore crucial for the efficient realization of almost every arithmetic unit in VLSI. This is in terms of circuit size, computation delay, and power consumption.

2.1.3 Number representation schemes

The representation of numbers and the hardware implementation of arithmetic units are strongly dependent on each other. On one hand, each number representation requires dedicated computation algorithms. On the other hand, efficient circuit realizations may ask for adequate number representations.

Only fixed-point number representations are considered in this thesis. This is justified since arithmetic operations on floating-point numbers are accomplished by applying various fixed-point operations on mantissa and exponent. Moreover, fixed-point numbers are reduced to integers herein, since every integer can be considered as a fraction multiplied by a constant factor.

Binary number systems

The radix-2 or binary number system is the most widely used number representation, which is due to its implementation efficiency and simplicity in digital circuit design. An n-bit number is represented as $A = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$, where $a_i \in \{0, 1\}$. The following representations for unsigned and signed fixed-point numbers are used:

Unsigned numbers are used for the representation of positive integers (i.e., natural numbers).

Value: $A = \sum_{i=0}^{n-1} a_i \cdot 2^i$,

Range: $[0, 2^n - 1]$.

Two's complement is the standard representation of signed numbers.

Value: $A = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i$,

Range: $[-2^{n-1}, 2^{n-1} - 1]$,

Complement: $-A = 2^n - A = \overline{A} + 1$,
where $\overline{A} = (\overline{a}_{n-1}, \overline{a}_{n-2}, \ldots, \overline{a}_1, \overline{a}_0)$,

Sign: a_{n-1},

Properties: asymmetric range (i.e., 2^{n-1} negative numbers, $(2^{n-1} - 1)$ positive numbers), compatible with unsigned numbers in most arithmetic operations.

One's complement is a similar representation as the two's complement.

Value: $A = -a_{n-1} \cdot (2^{n-1} + 1) + \sum_{i=0}^{n-2} a_i \cdot 2^i$,

Range: $[-(2^{n-1} - 1), 2^{n-1} - 1]$,

Complement: $-A = 2^n - A - 1 = \overline{A}$,

Sign: a_{n-1},

Properties: double representation of zero, symmetric range, modulo $(2^n - 1)$ number system.

Sign magnitude is an alternative representation of signed numbers. Here, the bits $a_{n-2}, a_{n-3}, \ldots, a_0$ are the true magnitude.

Value: $A = -a_{n-1} \cdot \sum_{i=0}^{n-2} a_i \cdot 2^i$,

Range: $[-(2^{n-1} - 1), 2^{n-1} - 1]$,

Complement: $-A = (\overline{a}_{n-1}, a_{n-2}, \ldots, a_1, a_0)$,

Sign: a_{n-1},

Properties: double representation of zero, symmetric range.

Due to their advantages and wide-spread use, the unsigned and two's complement signed number representations will be considered throughout the thesis.

Redundant number systems

Some redundant number systems[1] exist, which e.g. allow for speeding-up arithmetic operations [Kor93].

Carry-save is the redundant representation of the result when adding up three numbers without carry propagation (i.e., the individual carry bits are *saved* for later carry propagation). A carry-save number consists of two numbers, one containing all carry bits and the other all sum bits.

Delayed-carry or *half-adder form* [LJ96] is the corresponding representation when adding up only two numbers.

Signed-digit is a redundant number system, which makes use of the digit set $\{-1, 0, 1\}$.

The carry-save number representation plays an important role in multi-operand adders (see Sec. 3.4). Otherwise, redundant number systems are of no concern in carry-propagate adders, since they are used precisely to *avoid* carry propagation.

Residue number systems

Residue number system (RNS) do not use a fixed radix for all digits, but are constructed from a set of different residues, so that each digit has a different radix [Kor93]. Arithmetic operations in RNS can be computed on each digit independently and in parallel. The resulting speed-up is considerable, but conversion from and to conventional number systems is very expensive. The individual operations performed on each single digit are done using normal or modular integer arithmetic, and again mainly additions. The investigations on efficient integer addition in this thesis thus also become important for RNS systems.

[1] In redundant number systems, the number of representable digits is larger than the radix, thus allowing for multiple representations of the same number.

2.1.4 Sequential and combinational circuits

Many arithmetic operations can be realized as *combinational* or *sequential* circuits. Bit-serial or pipelined adders are examples for sequential adder circuits. However, since adder architectures deal with speeding up carry-propagation logic, only combinational adder implementations are covered in this thesis.

2.1.5 Synchronous and self-timed circuits

The realization of a synchronous circuit can be done in a *synchronous* or a *self-timed* asynchronous fashion, which also influences the implementation of the combinational circuits. In particular, self-timed combinational circuits have to provide completion signals, which are not trivial to generate. As a matter of fact, synchronous circuit techniques are standard in the VLSI design community.

However, adders are very appealing for self-timed realization since they have a short average carry-propagation length (i.e., $O(\log n)$) [GO96]. Because the simplest adder architecture — namely the ripple-carry adder — takes most advantage of self-timed implementation, a further study of adder architectures for self-timed circuit realization makes no sense.

2.1.6 Carry-propagate and carry-save adders

Addition is a prefix problem (see Sec. 3.5), which means that each result bit is dependent on all input bits of equal or lower magnitude. Propagation of a carry signal from each bit position to all higher bit positions is necessary. *Carry-propagate adders* perform this operation immediately. The required carry propagation from the least to the most significant bit results in a considerable circuit delay, which is a function of the word length of the input operands.

The most efficient way to speed-up addition is to avoid carry propagation, thus saving the carries for later processing. This allows the addition of two or more numbers in a very short time, but yields results in a redundant (carry-save) number representation.

Carry-save adders — as the most commonly used redundant arithmetic adders — play an important role in the efficient implementation of multi-operand addition circuits. They are very fast due to the absence of any carry-propagation paths, their structure is very simple, but the potential for further optimization is minimal. The same holds for signed-digit adders, which use a slightly different redundant number representation. The addition results, however, usually have to be converted into an irredundant integer representation in order to be processed further. This operation is done using a carry-propagate adder.

2.1.7 Implications

As we have seen so far, the combinational, binary carry-propagate adder is one of the most often used and most crucial building block in digital VLSI design. Various well-known methods exist for speeding-up carry-propagation in adders, offering very different performance characteristics, advantages, and disadvantages. Some lack of understanding of the basic concepts and relationships often lead to suboptimal adder implementations. One goal of this thesis is the systematic investigation and performance comparison of all existing adder architectures as well as their optimization with respect to cell-based design technologies.

2.2 Circuit and Layout Design Techniques

IC fabrication technologies can be classified into *full-custom*, *semi-custom*, and *programmable* ICs, as summarized in Table 2.1 (taken from [Kae97]). Further distinctions are made with respect to *circuit design techniques* and *layout design techniques*, which are strongly related.

2.2.1 Layout-based design techniques

In *layout-based* design techniques, dedicated full-custom layout is drawn manually for circuits designed at the *transistor-level*. The initial design effort is very high, but maximum circuit performance and layout efficiency is achieved. *Full-custom cells* are entirely designed by hand for dedicated high-performance units, e.g., arithmetic units. The tiled-layout technique can be used to simplify, automate, and parameterize the layout task. For reuse purposes, the circuits

Table 2.1: *IC classification scheme based on fabrication depth and design level.*

<table>
<tr><td>Fabrication depth</td><td>Programming only</td><td>Semi-custom fabrication</td><td colspan="2">Full-custom fabrication</td></tr>
<tr><td>Design level</td><td colspan="3">Cell-based, as obtained from schematic entry and/or synthesis</td><td>Hand layout</td></tr>
<tr><td>Type of integrated circuit</td><td>Programmable IC (PLD, FPGA, CPLD, etc.)</td><td>Gate-array or sea-of-gates IC</td><td>Standard cell IC (possibly also with macrocells and megacells)</td><td>Full-custom IC</td></tr>
</table>

and layouts are often collected in libraries together with automatic generators. *Mega-cells* are full-custom cells for universal functions which need no parameterization, e.g., microprocessor cores and peripherals. *Macro-cells* are used for large circuit components with regular structure and need for word-length parameterization, e.g., multipliers, ROMs, and RAMs. *Data paths* are usually realized in a bit-sliced layout style, which allows parameterization of word length (first dimension) and concatenation of arbitrary data-path elements (second dimension) for logic, arithmetic, and storage functions. Since adders are too small to be implemented as macro-cells, they are usually realized as data-path elements.

2.2.2 Cell-based design techniques

At a higher level of abstraction, arbitrary circuits can be composed from elementary logic gates and storage elements contained in a library of pre-designed cells. The layout is automatically composed from corresponding layout cells using dedicated layout strategies, depending on the used IC technology. *Cell-based* design techniques are used in standard-cell, gate-array, sea-of-gates, and field-programmable gate-array (FPGA) technologies. The design of logic circuits does not differ considerably among the different cell-based IC technologies. Circuits are obtained from either schematic entry, behavioral synthesis, or circuit generators (i.e., structural synthesis). Due to the required generic properties of the cells, more conventional logic styles have to be used for their circuit implementation.

The advantages of cell-based design techniques lie in their universal usage, automated synthesis and layout generation for arbitrary circuits, portability between tools and libraries, high design productivity, high reliability, and high flexibility in floorplanning. This comes at the price of lower circuit performance with respect to speed and area. Cell-based design techniques are mainly used for the implementation of random logic (e.g., controllers) and custom circuits for which no appropriate library components are available and custom implementation would be too costly. Cell-based design techniques are widely used in the ASIC design community.

Standard cells

Standard cells represent the highest performance cell-based technology. The layout of the cells is full-custom, which mandates for full-custom fabrication of the wavers. This in turn enables the combination of standard cells with custom-layout components on the same die. For layout generation, the standard cells are placed in rows and connected through intermediate routing channels. With the increasing number of routing layers and over-the-cell routing capabilities in modern process technologies, the layout density of standard cells gets close to the density obtained from full-custom layout. The remaining drawback is the restricted use of high-performance (transistor-level) circuit techniques.

Gate-arrays and sea-of-gates

On gate-arrays and sea-of-gates, preprocessed wafers with unconnected circuit elements are used. Thus, only metalization used for the interconnect is customized, resulting in lower production costs and faster turnaround times. Circuit performance and layout flexibility is lower than for standard cells, which in particular decreases implementation efficiency of regular structures such as macro-cells.

FPGAs

Field-programmable gate-arrays (FPGA) are electrically programmable generic ICs. They are organized as an array of logic blocks and routing channels, and the configuration is stored in a static memory or programmed e.g. using anti-fuses. Again, a library of logic cells and macros allows flexible and efficient design of arbitrary circuits. Turnaround times are very fast making FPGAs the ideal solution for rapid prototyping. On the other hand, low circuit performance, limited circuit complexity, and high die costs severely limit their area of application.

2.2.3 Implications

In the field of high-performance IC design, where layout-based and transistor-level design techniques are applied, much research effort has been invested in the realization of efficient adder circuits, and many different implementations have been proposed.

Efficient adder implementations for cell-based design, however, have hardly been addressed so far. Here, the issues to be investigated are the technology mapping, cell library properties, routing, synthesis, and portability aspects. The widespread use of cell-based design techniques justifies a closer inspection of the efficient circuit implementation of addition and related arithmetic operations.

2.3 Submicron VLSI Design

With evolving process technologies, feature sizes of 0.5μm, 0.35μm, and less become standard. These submicron technologies offer smaller and faster circuit structures at lower supply voltages, resulting in considerably faster and more complex ICs with a lower power dissipation per gate. Changing physical characteristics, however, strongly influence circuit design. Increasing gate densities and clocking frequencies lead to higher power densities, making low power an important issue in order to be able to dissipate the high energy of large chips.

2.3.1 Multilevel metal routing

As processes with three and more metalization levels become available, routing densities increase massively. Over-the-cell routing eliminates the drawback of area-consuming routing channels in cell-based technologies, yielding layout densities comparable to custom-layout. This also results in a larger amount of local interconnects (circuit locality), higher layout flexibility, and more efficient automated routers. Especially standard-cell technologies benefit from these advantages, providing both high design productivity as well as good circuit and layout performance.

2.3.2 Interconnect delay

The delay of interconnections becomes dominant over switching delays in submicron VLSI. This is because RC delays increase (higher wire resistances at roughly constant capacitances) and wire lengths typically scale with chip size but not with feature size. Therefore, circuit connectivity, locality, and fan-out are becoming important performance optimization criteria.

2.3.3 Implications

Cell-based design techniques take advantage from emerging submicron VLSI technologies, partly approaching densities and performance of full-custom techniques. Interconnect aspects have to be accounted for, also with respect to the optimality of circuit architectures.

2.4 Automated Circuit Synthesis and Optimization

Circuit synthesis denotes the automated generation of logic networks from behavioral descriptions at an arbitrary level. Synthesis is becoming a key issue in VLSI design for many reasons. Increasing circuit complexities, shorter development times, as well as efficient and flexible usage of cell and component libraries can only be handled with the aid of powerful design automation tools. Arithmetic synthesis addresses the efficient mapping of arithmetic functions onto existing arithmetic components and logic gates.

2.4.1 High-level synthesis

High-level synthesis, or behavioral/architectural synthesis, allows the translation of algorithmic or behavioral descriptions of high abstraction level (e.g., by way of data dependency graphs) down to RTL (register-transfer level)

representation, which can be processed further by low-level synthesis tools. The involved architectural synthesis, including resource allocation, resource binding, and scheduling tasks, is far from being trivial and is currently researched intensively. High-level arithmetic synthesis makes use of arithmetic transformations in order to optimize hardware usage under given performance criteria. Thereby, arithmetic library components are regarded as the resources for implementing the basic arithmetic operations.

2.4.2 Low-level synthesis

Low-level synthesis, or logic synthesis, translates an RTL specification into a generic logic network. For random logic, synthesis is achieved by establishing the logic equations for all outputs and implementing them in a logic network.

2.4.3 Data-path synthesis

Efficient arithmetic circuits contain very specific structures of large logic depth and high factorization degree. Their direct synthesis from logic equations is not feasible. Therefore, parameterized netlist generators using dedicated algorithms are used instead. Most synthesis tools include generators for the basic arithmetic functions, such as comparators, incrementers, adders, and multipliers. For other important operations (e.g., squaring, division) and specialized functions (e.g., addition with flag generation, multiplication without final addition) usually no generators are provided and thus synthesis of efficient circuitry is not available. Also, the performance of the commonly used circuit architectures varies considerably, which often leads to suboptimal cell-based circuit implementations.

2.4.4 Optimization of combinational circuits

The optimization of combinational circuits connotes the automated minimization of a logic netlist with respect to area, delay, and power dissipation measures of the resulting circuit, and the technology mapping (i.e., mapping of the logic network onto the set of logic cells provided by the used technology/library). The applied algorithms are very powerful for optimization of random logic by performing steps like flattening, logic minimization, timing-driven factorization, and technology mapping. However, the potential for optimization is rather limited for networks with large logic depth and high factorization degree, especially arithmetic circuits. There, only local logic minimization is possible, leaving the global circuit architecture basically unchanged. Thus, the realization of well-performing arithmetic circuits relies more on efficient data-path synthesis than on simple logic optimization.

2.4.5 Hardware description languages

Hardware description languages allow the specification of hardware at different levels of abstraction, serving as entry points to hardware synthesis. VHDL, as one of the most widely used and most powerful languages, enables the description of circuits at the behavioral and structural level. In particular, parameterized netlist generators can be written in structural VHDL.

Synthesis of arithmetic units is initiated by using the standard arithmetic operator symbols in the VHDL code, for which the corresponding built-in netlist generators are called by the synthesis tool. Basically, the advantages of VHDL over schematic entry lie in the possibility of behavioral hardware description, the parameterizability of circuits, and portability of code thanks to language standardization.

2.4.6 Implications

Due to their manyfold occurrences and flexible usage, arithmetic units form an integral part in automated hardware synthesis for high-productivity VLSI design. The used circuit architectures must be highly flexible and easily parameterizable and must result in simple netlist generators and efficient circuit implementations. Thus, this thesis also focuses on algorithms for the synthesis of adder circuits and investigates the suitability of various adder architectures with respect to netlist synthesis and optimization.

2.5 Circuit Complexity and Performance Modeling

One important aspect in design automation is the complexity and performance estimation of a circuit early in the design cycle, i.e., prior to the time-consuming logic synthesis and physical layout phases. At a higher design level, this is

achieved by using characterization information of the high-level components to be used and by complexity estimation of the interconnect. At gate level, however, estimation is more difficult and less accurate because circuit size and performance strongly depend on the gate-level synthesis results and on the physical cell arrangement and routing.

For a rough preliminary characterization of adder architectures, we are interested in simple complexity and performance models for gate-level circuits. Given a circuit specified by logic formulae or a generic netlist (i.e., a netlist built from basic logic gates), we need estimations of the expected area, speed, and power dissipation for a compiled cell-based circuit as a function of the operand word length.

2.5.1 Area modeling

Silicon area on a VLSI chip is taken up by the active circuit elements and their interconnections. In cell-based design techniques, the following criteria for area modeling can be formulated:

- Total *circuit complexity* (GE_{total}) can be measured by the number of *gate equivalents* ($1\ GE \equiv 1$ 2-input NAND-gate $\equiv 4$ MOSFETs).
- *Circuit area* ($A_{circuit}$) is occupied by logic cells and inter-cell wiring. In technologies with three and more metal layers, over-the-cell routing capabilities allow the overlap of cell and wiring areas, as opposed to 2-metal technologies. This means that most of the cell area can also be used for wiring, resulting in very low routing area factors. ($A_{circuit} = A_{cells} + A_{wiring}$)
- Total *cell area* (A_{cells}) is roughly proportional to the number of transistors or gate equivalents (GE_{total}) contained in a circuit. This number is influenced by technology mapping, but not by physical layout. Thus, cell area can be roughly estimated from a generic circuit description (e.g. logic equations or netlist with simple gates) and can be precisely determined from a synthesized netlist. ($A_{cells} \propto GE_{total}$)
- *Wiring area* (A_{wiring}) is proportional to the total wire length. The exact wire lengths, however, are not known prior to physical layout. ($A_{wiring} \propto L_{total}$)

- Total *wire length* (L_{total}) can be estimated from the number of nodes and the average wire length of a node [Feu82, KP89] or, more accurate, from the sum of cell fan-out and the average wire length of cell-to-cell connections (i.e. accounts for the longer wire length of nodes with higher fan-out). The wire lengths also depend on circuit size, circuit connectivity (i.e., locality of connections), and layout topology, which are not known prior to circuit partitioning and physical layout [RK92]. ($L_{total} \propto FO_{total}$)

- Cell *fan-out* (FO) is the number of cell inputs a cell output is driving. *Fan-in* is the number of inputs to a cell [WE93], which for many combinational gates is proportional to the size of the cell. Since the sum of cell fan-out (FO_{total}) of a circuit is equivalent to the sum of cell fan-in, it is also proportional to circuit size. ($FO_{total} \propto GE_{total}$)

- Therefore, in a first approximation, cell area as well as wiring area are proportional to the number of gate equivalents. More accurate area estimations before performing actual technology mapping and circuit partitioning are hardly possible. For circuit comparison purposes, the proportionality factor is of no concern. ($A_{circuit} \propto GE_{total} \propto FO_{total}$)

Our area estimation model we are interested in must be simple to compute while being as accurate as possible, and it should anticipate from logic equations or generic netlists (i.e. netlists composed of simple logic gates) alone. By considering the above observations, possible candidates are:

Unit-gate area model This is the simplest and most abstract circuit area model, which is often used in the literature [Tya93]. A unit gate is a basic, monotonic 2-input gate (or logic operation, if logic equations are concerned), such as AND, OR, NAND, and NOR. Basic, non-monotonic 2-input gates like XOR and XNOR are counted as two unit gates, reflecting their higher circuit complexities. Complex gates as well as multi-input basic gates are built from 2-input basic gates and their gate count equals the sum of gate counts of the composing cells.

Fan-in area model In the fan-in model, the size of 2- and multi-input basic cells is measured by counting the number of inputs (i.e., fan-in). Complex cells are again composed of basic cells with their fan-in numbers summed up, while the XOR/XNOR-gates are treated individually. The obtained numbers basically differ from the unit-gate numbers only by an offset of 1 (e.g., the AND-gate counts as one unit gate but has a fan-in of two).

Other area models The two previous models do not account for transistor-level optimization possibilities in complex gates, e.g., in multiplexers and full-adders. More accurate area models need individual gate count numbers for such complex gates. However, some degree of abstraction is sacrificed and application on arbitrary logic equations is not possible anymore. The same holds true for models which take wiring aspects into consideration. One example of a more accurate area model is the gate-equivalents model (GE) mentioned above, which bases on gate transistor counts and therefore is only applicable after synthesis and technology mapping.

Inverters and buffers are not accounted for in the above area models, which makes sense for pre-synthesis circuit descriptions. Note that the biggest differences in buffering costs are found between low fan-out and high fan-out circuits. With respect to area occupation however, these effects are partly compensated because high fan-out circuits need additional buffering while low fan-out circuits usually have more wiring.

Investigations showed that the unit-gate model approach for the area estimation of complex gates, such as multiplexers and full-adders, does not introduce more inaccuracies than e.g. the neglection of circuit connectivity for wiring area estimation. With the XOR/XNOR being treated separately, the unit-gate model yields acceptable accuracy at the given abstraction level. Also, it perfectly reflects the structure of logic equations by modeling the basic logic operators individually and by regarding complex logic functions as composed from basic ones. Investigations showed comparable performance for the fan-in and the unit-gate models due to their similarity. After all, the unit-gate model is very commonly used in the literature. Therefore, it is used in this work for area estimations and comparisons from logic circuit specifications. Comparison results of placed and routed standard-cell solutions will follow in Section 4.2.

2.5.2 Delay modeling

Propagation delay in a circuit is determined by the cell and interconnection delays on the critical path (i.e. longest signal propagation path in a combinational circuit). As opposed to area estimation, not average and total numbers are of interest, but individual cell and node values are relevant for path delays. Critical path evaluation is done by static timing analysis which involves graph-based search algorithms. Of course, timings are also dependent on temperature, voltage, and process parameters which, however, are not of concern for our comparison purposes.

- *Maximum delay* (t_{crit_path}) of a circuit is equal to the sum of cell inertial delays, cell output ramp delays, and wire delays on the critical path. ($t_{crit\_path} = \sum_{\in crit\_path}((t_{cell} + t_{ramp}) + \sum_{\in crit\_path} t_{wire})$
- *Cell delay* (t_{cell}) depends on the transistor-level circuit implementation and the complexity of a cell. All simple gates have comparable delays. Complex gates usually contain tree-like circuit and transistor arrangements, resulting in logarithmic delay-to-area dependencies. ($t_{cell} \propto \log{(A_{cell})}$)
- *Ramp delay* (t_{ramp}) is the time it takes for a cell output to drive the attached capacitive load, which is made up of interconnect and cell input loads. The ramp delay depends linearly on the capacitive load attached, which in turn depends linearly on the fan-out of the cell. ($t_{ramp} \propto FO_{cell}$)
- *Wire delay* or *interconnection delay* (t_{wire}) is the RC-delay of a wire, which depends on the wire length. RC-delays, however, are negligible compared to cell and ramp delays for small circuits such as the adders investigated in this work. ($t_{wire} = 0$).
- Thus, a rough delay estimation is possible by considering sizes and, with a smaller weighting factor, fan-out of the cells on the critical path. ($t_{crit\_path} \propto \sum_{\in crit\_path}(\log{(A_{cell})} + kFO_{cell})$)

Possible delay estimation models are:

Unit-gate delay model The unit-gate delay model is similar to the unit-gate area model. Again, the basic 2-input gates (AND, OR, NAND, NOR) count as one gate delay with the exception of the XOR/XNOR-gates which count as two gate delays [Tya93]. Complex cells are composed of basic cells using the fastest possible arrangement (i.e., tree structures wherever possible) with the total gate delay determined accordingly.

Fan-in delay model As for area modeling, fan-in numbers can be taken instead of unit-gate numbers. Again, no advantages over the unit-gate model are observed.

Fan-out delay model The fan-out delay model bases on the unit-gate model but incorporates fan-out numbers, thus accounting for gate fan-out numbers and interconnection delays [WT90]. Individual fan-out numbers can be obtained from a generic circuit description. A proportionality factor has to be determined for appropriate weighting of fan-out with respect to unit-gate delay numbers.

Other delay models Various delay models exist at other abstraction levels. At the transistor level, transistors can be modeled to contribute one unit delay each (τ-model [CSTO91]). At a higher level, complex gates like full-adders and multiplexers can again be modeled separately for higher accuracy [Kan91, CSTO91].

The impact of large fan-out on circuit delay is higher than on area requirements. This is because high fan-out nodes lead to long wires and high capacitive loads and require additional buffering, resulting in larger delays. Therefore, the fan-out delay model is more accurate than the unit-gate model. However, due to the much simpler calculation of the unit-gate delay model and its widespread use, as well as for compatibility reasons with the chosen unit-gate area model, this model will be used for the circuit comparisons in this work.

As already mentioned, delay calculation for a circuit requires static timing analysis, which corresponds to the search for the longest path in a weighted directed acyclic graph. In our case, false path[2] detection [MB89] is not of importance since false paths do not occur in adder circuits with one exception, which will be discussed later.

2.5.3 Power measures and modeling

An increasingly important performance parameter for VLSI circuits is power dissipation. *Peak power* is a problem with respect to circuit reliability (e.g. voltage drop on power buses, ground bounce) which, however, can be dealt with by careful design. On the other hand, *average power* dissipation is

[2] A false path is a signal path in a combinational circuit which cannot be sensitized.

becoming a crucial design constraint in many modern applications, such as high-performance microprocessors and portable applications, due to heat removal problems and power budget limitations.

The following principles hold for average power dissipation in synchronous CMOS circuits [ZF97]:

- *Total power* (P_{total}) in CMOS circuits is dominated by the dynamic switching of circuit elements (i.e., charging and discharging of capacitances), whereas dynamic short-circuit (or overlap) currents and static leakage are of less importance. Thus, power dissipation can be assumed proportional to the total capacitance to be switched, the square of the supply voltage, the clock frequency, and the switching activity α in a circuit [CB95]. ($P_{total} = \frac{1}{2} \cdot C_{total} \cdot V_{dd}^2 \cdot f_{clk} \cdot \alpha$)
- *Total capacitance* (C_{total}) in a CMOS circuit is the sum of the capacitances from transistor gates, sources, and drains and from wiring. Thus, total capacitance is proportional to the number of transistors and the amount of wiring, both of which are roughly proportional to circuit size. ($C_{total} \propto GE_{total}$)
- *Supply voltage* (V_{dd}) and *clock frequency* (f_{clk}) can be regarded as constant within a circuit and therefore are not relevant in our circuit comparisons. ($V_{dd}, f_{clk} =$ const.)
- The *switching activity factor* (α) gives a measure for the number of transient nodes per clock cycle and depends on input patterns and circuit characteristics. In many cases, input patterns to data paths and arithmetic units are assumed to be random, which results in a constant average transition activity of 50% on all inputs (i.e., each input toggles each second clock cycle). Signal propagation through several levels of combinational logic may decrease or increase transition activities, depending on the circuit structure. Such effects, however, are of minor relevance in adder circuits and will be discussed later in the thesis. ($\alpha =$ const.)
- Therefore, for arithmetic units having constant input switching activities, power dissipation is approximately proportional to circuit size. ($P_{total} \propto GE_{total}$)

If average power dissipation of a circuit can be regarded as proportional to its size, the presented area models can also be used for power estimation.

Thus, the **unit-gate model** is chosen for the power comparisons of generic circuit descriptions.

2.5.4 Combined circuit performance measures

Depending on the constraints imposed by the design specifications, the performance of combinational circuits is measured by means of either circuit size, propagation delay, or power dissipation, or by a combination of those. Frequently used combined performance measures are the *area-time* or *area-delay* product (AT-product) and the *power-time* or *power-delay* product (PT-product). The PT-product can also be regarded as the amount of energy used per computation. The unit-gate models presented above for area, delay, and power estimation can also be used for AT- and PT-product comparisons.

Additionally, circuits and circuit architectures can be characterized with respect to *simplicity* (for implementation and understanding) and *regularity* (for synthesis and layout) of structure.

2.5.5 Implications

Influences on the performance of cell-based circuits are manifold thus making accurate modeling a difficult task. At the level of generic netlists or specifications by logic equations, however, accurate performance estimation is not possible due to the lack of detailed circuit and layout information. There, the simplified unit-gate model fits well and will be used in the following text for abstract comparisons and classifications of adder circuit architectures.

2.6 Summary

Arithmetic units belong to the basic and most crucial building blocks in many integrated circuits, and their performance depends on the efficient hardware implementation of the underlying arithmetic operations. Changing physical properties of submicron VLSI require circuit architectures and styles to be reconsidered. Advances in computer-aided design as well as the ever growing design productivity demands tend to prefer cell-based design techniques and hardware synthesis, also for arithmetic components. Complexity and performance modeling allows architecture and circuit evaluations and decisions early in the design cycle. In this thesis, these aspects are covered for binary carry-propagate addition and related arithmetic operations.

3

Basic Addition Principles and Structures

This chapter introduces the basic principles and circuit structures used for the addition of single bits and of two or multiple binary numbers. Binary carry-propagate addition is formulated as a prefix problem, and the fundamental algorithms and speed-up techniques for the efficient solution of this problem are described.

Figure 3.1 gives an overview of the basic adder structures and their relationships. The individual components will be described in detail in this and the following chapter.

3.1 1-Bit Adders, (m,k)-Counters

As the basic combinational addition structure, a *1-bit adder* computes the sum of m input bits of the same magnitude (i.e., 1-bit numbers). It is also called *(m,k)-counter* (Fig. 3.2) because it counts the number of 1's at the m inputs $(a_{m-1}, a_{m-2}, \ldots, a_0)$ and outputs a k-bit sum $(s_{k-1}, s_{k-2}, \ldots, s_0)$, where $k = \lceil \log(m+1) \rceil$.

Arithmetic equation:

$$\sum_{j=0}^{k-1} 2^j s_j = \sum_{i=0}^{m-1} a_i \tag{3.1}$$

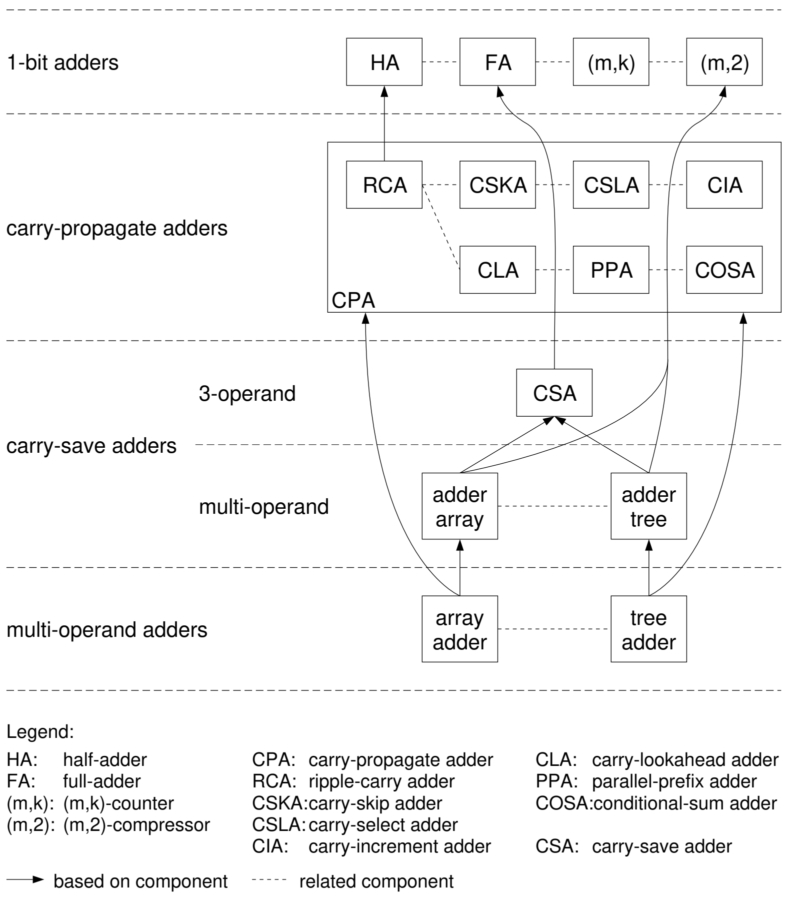


Figure 3.1: *Overview of adder structures.*

3.1.1 Half-Adder, (2,2)-Counter

The *half-adder* (HA) is a (2,2)-counter. The more significant sum bit is called *carry-out* (c_{out}) because it carries an overflow to the next higher bit position. Figure 3.3 depicts the logic symbol and two circuit implementations of a half-adder. The corresponding arithmetic and logic equations are given below, together with the area (A) and time (T) complexity measures under the

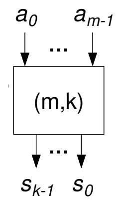


Figure 3.2: *(m,k)-counter symbol.*

unit-gate models described in Section 2.5.

Arithmetic equations:

$$2c_{out} + s = a + b \tag{3.2}$$
$$s = (a + b) \bmod 2$$
$$c_{out} = (a + b) \operatorname{div} 2 = \tfrac{1}{2}(a + b - s) \tag{3.3}$$

Logic equations:

$$s = a \oplus b \tag{3.4}$$
$$c_{out} = ab \tag{3.5}$$

Complexity:

$$T_{\mathrm{HA}}\,(a, b \rightarrow c_{out}) = 1 \qquad A_{\mathrm{HA}} = 3$$
$$T_{\mathrm{HA}}\,(a, b \rightarrow s) = 2$$

3.1.2 Full-Adder, (3,2)-Counter

The *full-adder* (FA) is a (3,2)-counter. The third input bit is called *carry-in* (c_{in}) because it often receives a carry signal from a lower bit position. Important internal signals of the full-adder are the *generate* (g) and *propagate* (p) signals. The generate signal indicates whether a carry signal — 0 or 1 — is generated within the full-adder. The propagate signal indicates whether a carry at the input is propagated unchanged through the full-adder to the carry-out.

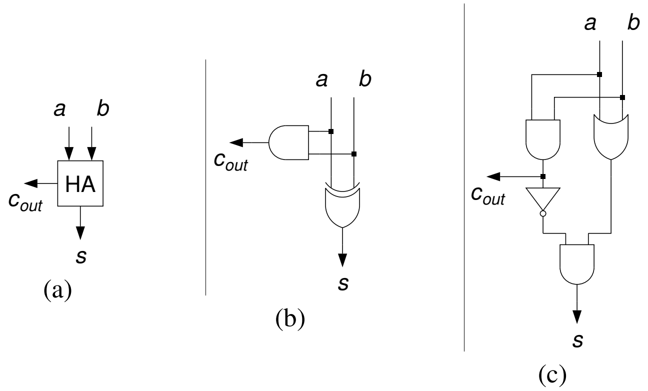


Figure 3.3: *(a) Logic symbol, and (b, c) schematics of a half-adder.*

Alternatively, two intermediate carry signals c^0 and c^1 can be calculated, one for $c_{in} = 0$ and one for $c_{in} = 1$. Thus, the carry-out can be expressed by the (g, p) or the (c^0, c^1) signal pairs and the carry-in signal and be realized using an AND-OR or a multiplexer structure. Note that for the computation of c_{out} using the AND-OR structure, the propagate signal can also be formulated as $p = a + b$. The propagate signal for the sum bit calculation, however, must be implemented as $p = a \oplus b$.

Arithmetic equations:

$$2c_{out} + s = a + b + c_{in} \tag{3.6}$$

$$\begin{aligned} s &= (a + b + c_{in}) \bmod 2 \\ c_{out} &= (a + b + c_{in}) \operatorname{div} 2 = \tfrac{1}{2}(a + b + c_{in} - s) \end{aligned} \tag{3.7}$$

Logic equations:

$$g = ab \tag{3.8}$$

$$p = a \oplus b \tag{3.9}$$

$$\begin{aligned} c^0 &= ab \\ c^1 &= a + b \end{aligned} \tag{3.10}$$

$$\begin{aligned} s &= a \oplus b \oplus c_{in} \\ &= p \oplus c_{in} \end{aligned} \tag{3.11}$$

$$\begin{aligned} c_{out} &= ab + ac_{in} + bc_{in} \\ &= ab + (a + b)c_{in} = ab + (a \oplus b)c_{in} \\ &= g + pc_{in} \\ &= \overline{p}g + pc_{in} = \overline{p}a + pc_{in} \\ &= \overline{c}_{in}c^0 + c_{in}c^1 \end{aligned} \tag{3.12}$$

Complexity:

$$\begin{aligned} T_{\text{FA}}\,(a, b \to c_{out}) &= 4\ (2) \\ T_{\text{FA}}\,(a, b \to s) &= 4 \\ T_{\text{FA}}\,(c_{in} \to c_{out}) &= 2 \\ T_{\text{FA}}\,(c_{in} \to s) &= 2\ (4) \end{aligned} \qquad A_{\text{FA}} = 7\ (9)$$

A full-adder can basically be constructed using half-adders, 2-input gates, multiplexers, or complex gates (Figs. 3.4b–f). The solutions (b) and (d) (and to some extent also (e)) make use of the generate g and propagate p signals (*generate-propagate scheme*). Circuit (f) bases on generating both possible carry-out signals c^0 and c^1 and selecting the correct one by the carry-in c_{in} (*carry-select scheme*). Solution (c) generates s by a 3-input XOR and c_{out} by a *majority gate* directly. This complex-gate solution has a faster carry generation but is larger, as becomes clear from the complexity numbers given in parenthesis. Because the majority gate can be implemented very efficiently at the transistor level, it is given a gate count of 5 and a gate delay of only 2. The multiplexer counts 3 gates and 2 gate delays.

3.1.3 (m,k)-Counters

Larger counters can be constructed from smaller ones, i.e., basically from full-adders. Due to the associativity of the addition operator, the m input bits can be added in any order, thereby allowing for faster tree arrangements of the full-adders (see Fig. 3.5).

Complexity:

$$T_{(\text{m,k})} = O(m) \qquad A_{(\text{m,k})} = O(\log m)$$

An *(m,2)-compressor* is a 1-bit adder with a different sum representation. It is used for the realization of multi-operand adders and will be discussed in Section 3.4.

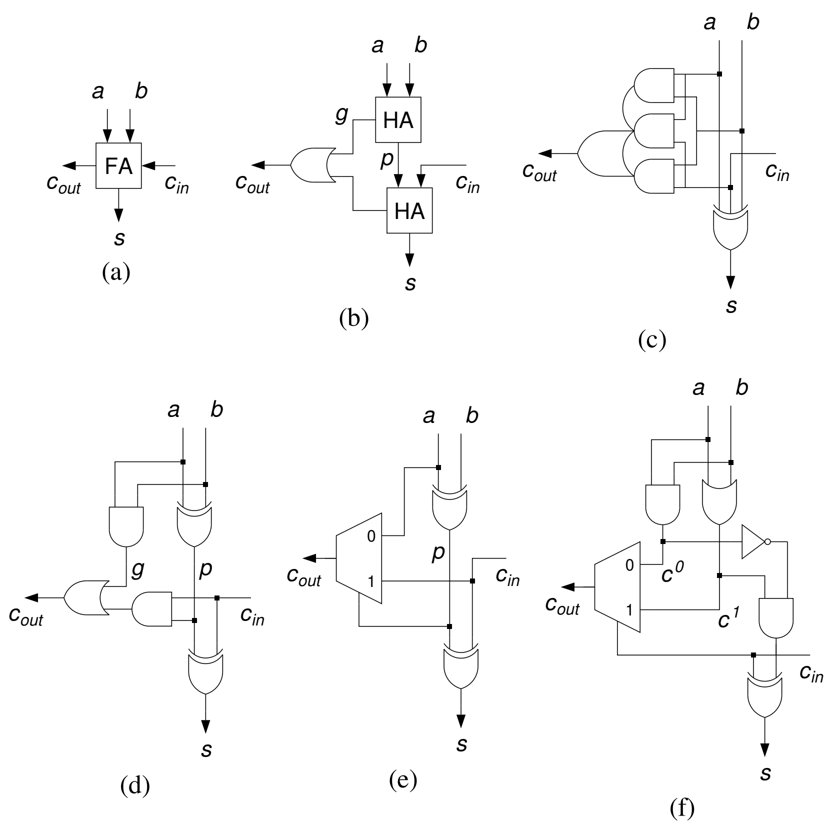


Figure 3.4: *(a) Logic symbol, and (b, c, d, e, f) schematics of a full-adder.*

3.2 Carry-Propagate Adders (CPA)

A *carry-propagate adder* (CPA) adds two n-bit operands $A = (a_{n-1}, a_{n-2}, \ldots, a_0)$ and $B = (b_{n-1}, b_{n-2}, \ldots, b_0)$ and an optional carry-in c_{in} by performing carry-propagation. The result is an irredundant $(n+1)$-bit number consisting of the n-bit sum $S = (s_{n-1}, s_{n-2}, \ldots, s_0)$ and a carry-out c_{out}.

Equation 3.16 describes the logic for bit-sequential addition of two n-bit numbers. It can be implemented as a combinational circuit using n full-adders connected in series (Fig. 3.6) and is called *ripple-carry adder* (RCA).

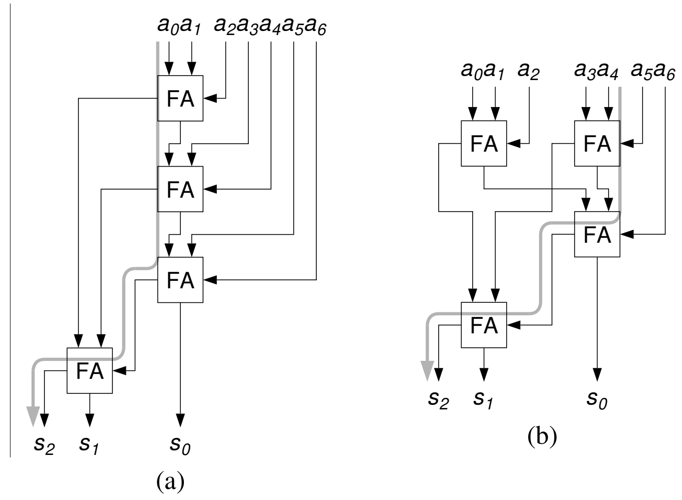


Figure 3.5: *(7,3)-counter with (a) linear and (b) tree structure.*

Arithmetic equations:

$$2^n c_{out} + S = A + B + c_{in} \tag{3.13}$$

$$\begin{aligned} 2^n c_{out} + \sum_{i=0}^{n-1} 2^i s_i &= \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i + c_{in} \\ &= \sum_{i=0}^{n-1} 2^i (a_i + b_i) + c_{in} \end{aligned} \tag{3.14}$$

$$\begin{aligned} 2c_{i+1} + s_i &= a_i + b_i + c_i \;; \quad i = 0, 1, \ldots, n-1 \\ & \text{where } c_0 = c_{in} \text{ and } c_{out} = c_n \end{aligned} \tag{3.15}$$

Logic equations:

$$\begin{aligned} g_i &= a_i \, b_i \\ p_i &= a_i \oplus b_i \\ s_i &= p_i \oplus c_i \\ c_{i+1} &= g_i + p_i \, c_i \;; \quad i = 0, 1, \ldots, n-1 \\ & \text{where } c_0 = c_{in} \text{ and } c_{out} = c_n \end{aligned} \tag{3.16}$$

Complexity:

$$
\begin{aligned}
T_{\text{CPA}}\,(a, b \rightarrow c_{out}, s) &= 2n + 2 \\
T_{\text{CPA}}\,(c_{in} \rightarrow c_{out}, s) &= 2n
\end{aligned}
\qquad A_{\text{CPA}} = 7n
$$

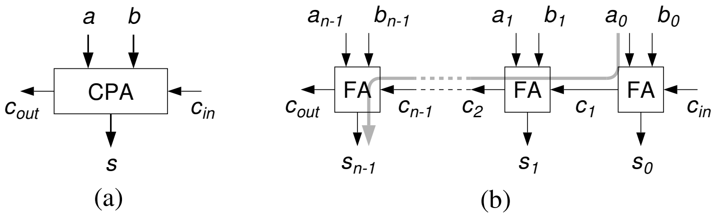


Figure 3.6: *(a) Symbol and (b) ripple-carry implementation of carry-propagate adder (CPA).*

Note that the computation time of this adder grows linearly with the operand word length n due to the serial carry-propagation.

3.3 Carry-Save Adders (CSA)

The *carry-save adder* (CSA) avoids carry propagation by treating the intermediate carries as outputs instead of advancing them to the next higher bit position, thus saving the carries for later propagation. The sum is a (redundant) n-digit carry-save number, consisting of the two binary numbers S (sum bits) and C (carry bits). A Carry-save adder accepts three binary input operands or, alternatively, one binary and one carry-save operand. It is realized by a linear arrangement of full-adders (Fig. 3.7) and has a constant delay (i.e., independent of n).

Arithmetic equations:

$$2C + S = A_0 + A_1 + A_2 \tag{3.17}$$

$$\sum_{i=1}^{n} 2^i c_i + \sum_{i=0}^{n-1} 2^i s_i = \sum_{j=0}^{2} \sum_{i=0}^{n-1} 2^i a_{j,i} \tag{3.18}$$

$$2c_{i+1} + s_i = \sum_{j=0}^{2} a_{j,i} \;\; ; \quad i = 0, 1, \ldots, n-1 \tag{3.19}$$

Complexity:

$$
\begin{aligned}
T_{\text{CSA}}\,(a_0, a_1 \rightarrow c, s) &= 4 \\
T_{\text{CSA}}\,(a_2 \rightarrow c, s) &= 2
\end{aligned}
\qquad A_{\text{CSA}} = 7n
$$

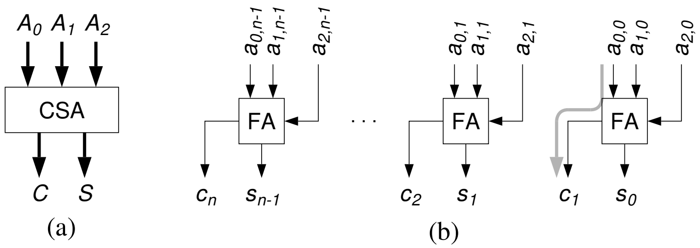


Figure 3.7: *(a) Symbol and (b) schematic of carry-save adder (CSA).*

3.4 Multi-Operand Adders

Multi-operand adders are used for the summation of m n-bit operands $A_0, \ldots, A_{m-1}$ $(m > 2)$ yielding a result S in irredundant number representation with $(n + \lceil \log m \rceil)$ bits.

Arithmetic equation:

$$S = \sum_{j=0}^{m-1} A_j \tag{3.20}$$

3.4.1 Array Adders

An m-operand adder can be realized either by serial concatenation of $(m-1)$ carry-propagate adders (i.e., ripple-carry adders, Fig. 3.8) or by $(m-2)$ carry-save adders followed by a final carry-propagate adder (Fig. 3.9). The two resulting *adder arrays* are very similar with respect to their logic structure, hardware requirements, as well as the length of the critical path. The major difference is the unequal bit arrival time at the last carry-propagate adder. While in the carry-save adder array (CSA array), bit arrival times are balanced, higher bits arrive later than lower bits in the carry-propagate adder array

(CPA array) which, however, is exactly how the final adder "expects" them. This holds true if ripple-carry adders are used for carry-propagate addition throughout.

Speeding up the operation of the CPA array is not efficient because each ripple-carry adder has to be replaced by some faster adder structure. On the other hand, the balanced bit arrival profile of the CSA array allows for massive speed-up by just replacing the final RCA by a fast parallel carry-propagate adder. Thus, fast *array adders*[3] are constructed from a CSA array with a subsequent fast CPA (Fig. 3.10).

Complexity:

$$
\begin{aligned}
T_{\text{ARRAY}} &= (m-2)T_{\text{CSA}} + T_{\text{CPA}} \\
A_{\text{ARRAY}} &= (m-2)A_{\text{CSA}} + A_{\text{CPA}}
\end{aligned}
$$

3.4.2 (m,2)-Compressors

A single bit-slice of the carry-save array from Figure 3.9 is a 1-bit adder called *(m,2)-compressor*. It compresses m input bits down to two sum bits (c, s) by forwarding $(m-3)$ intermediate carries to the next higher bit position (Fig. 3.11).

Arithmetic equation:

$$2(c + \sum_{l=0}^{m-4} c_{out}^{l}) + s \quad = \quad \sum_{i=0}^{m-1} a_i + \sum_{l=0}^{m-4} c_{in}^{l} \tag{3.21}$$

No horizontal carry-propagation occurs within a compressor circuit, i.e., c_{in}^{l} only influences $c_{out}^{k>l}$. An (m,2)-compressor can be built from $(m-2)$ full-adders or from smaller compressors. Note that the full-adder can also be regarded as a (3,2)-compressor. Again, cells can be arranged in tree structures for speed-up.

Complexity:

$$T_{(\text{m},2)} = O(m) \qquad A_{(\text{m},2)} = O(\log m)$$

[3]Note the difference between *adder array* (i.e., CSA made up from an array of adder cells) and *array adder* (i.e., multi-operand adder using CSA array and final CPA).

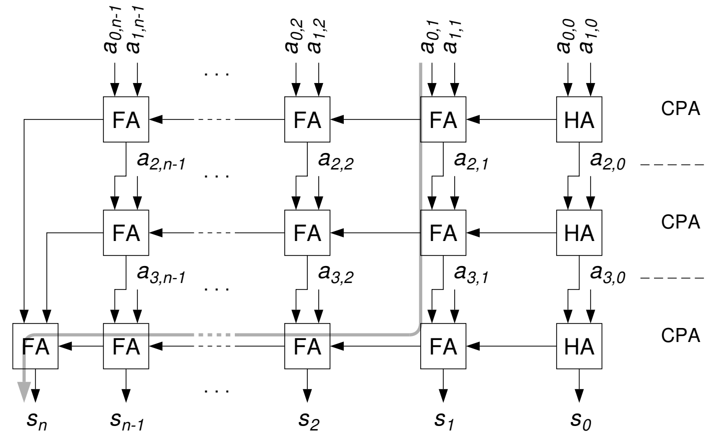


Figure 3.8: *Four-operand carry-propagate adder array.*

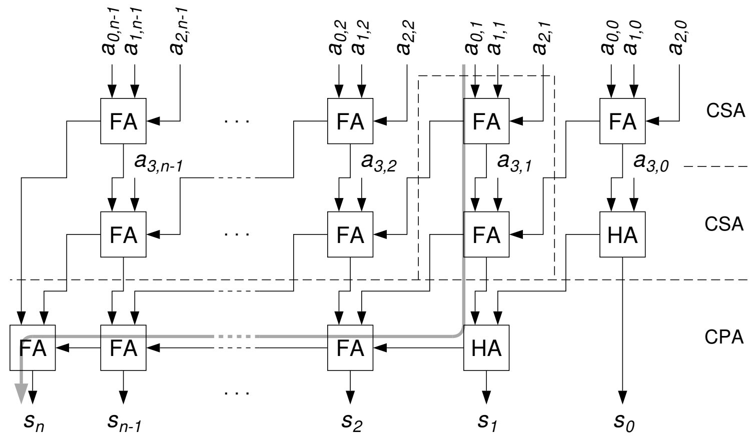


Figure 3.9: *Four-operand carry-save adder array with final carry-propagate adder.*

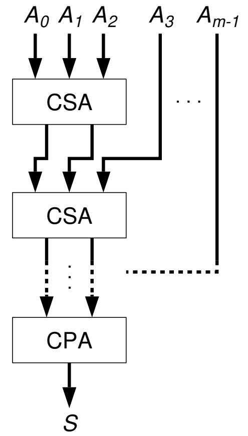


Figure 3.10: *Typical array adder structure for multi-operand addition.*

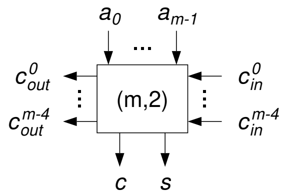


Figure 3.11: *(m,2)-compressor symbol.*

(4,2)-compressor

The *(4,2)-compressor* allows for some circuit optimizations by rearranging the EXORs of the two full-adders (Fig. 3.12). This enables the construction of more shallow and more regular tree structures.

Arithmetic equation:

$$2(c + c_{out}) + s \quad = \quad \sum_{i=0}^{3} a_i + c_{in} \tag{3.22}$$

Complexity:

$$\begin{aligned} T_{(4,2)}\ (a_i \rightarrow c, s) &= 6 \\ T_{(4,2)}\ (a_i \rightarrow c_{out}) &= 4 \\ T_{(4,2)}\ (c_{in} \rightarrow c, s) &= 2 \end{aligned} \qquad A_{(4,2)} = 14$$

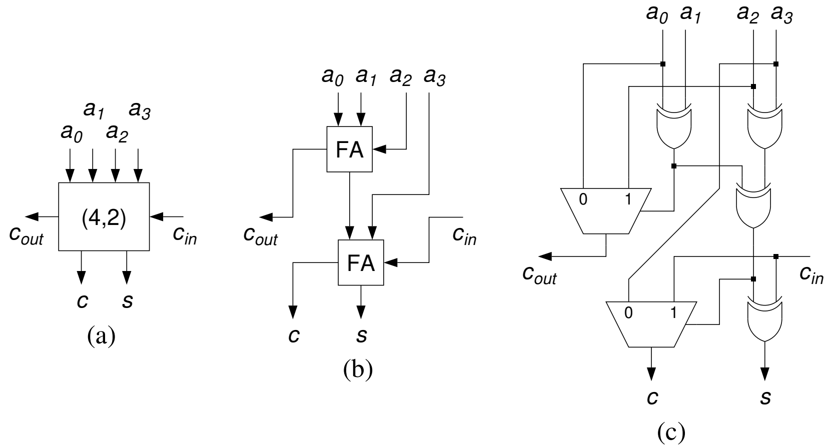


Figure 3.12: *(a) Logic symbol and (b, c) schematics of a (4,2)-compressor.*

3.4.3 Tree Adders

Adder trees (or Wallace trees) are carry-save adders composed of tree-structured compressor circuits. *Tree adders* are multi-operand adders consisting of a CSA tree and a final CPA. By using a fast final CPA, they provide the fastest multi-operand adder circuits. Figure 3.13 shows a 4-operand adder using (4,2)-compressors.

Complexity:

$$\begin{aligned} T_{\text{TREE}} &= T_{(\text{m},2)} + T_{\text{CPA}} \\ A_{\text{TREE}} &= nA_{(\text{m},2)} + A_{\text{CPA}} \end{aligned}$$

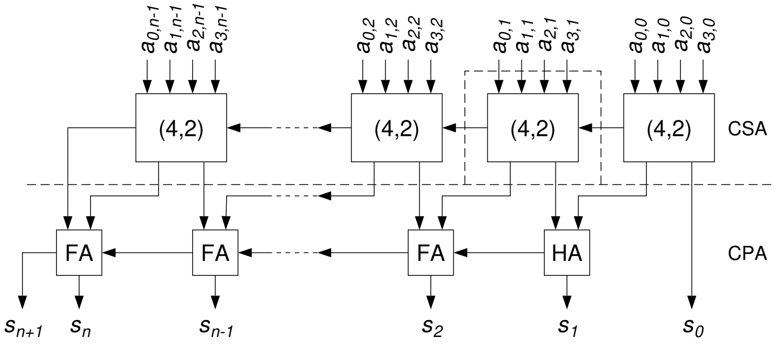


Figure 3.13: *4-operand adder using (4,2)-compressors.*

3.4.4 Remarks

Some general remarks on multi-operand adders can be formulated at this point:

- Array adders have a highly regular structure which is of advantage for both netlist and layout generators.
- An m-operand adder accommodates $(m-1)$ carry inputs.
- The number of full-adders does only depend on the number of operands and bits to be added, but not on the adder structure. However, the number of half-adders as well as the amount and complexity of interconnect wiring depends on the chosen adder configuration (i.e., array or tree).
- Accumulators are sequential multi-operand adders. They also can be sped up using the carry-save technique.

3.5 Prefix Algorithms

The addition of two binary numbers can be formulated as a prefix problem. The corresponding parallel-prefix algorithms can be used for speeding up binary addition and for illustrating and understanding various addition principles.

This section introduces a mathematical and visual formalism for prefix problems and algorithms.

3.5.1 Prefix problems

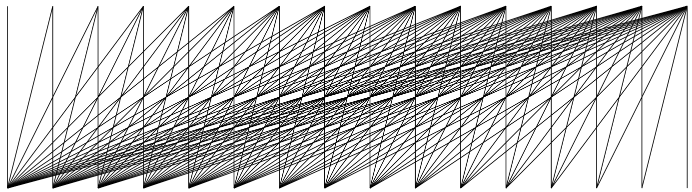
In a prefix problem, n outputs $(y_{n-1}, y_{n-2}, \ldots, y_0)$ are computed from n inputs $(x_{n-1}, x_{n-2}, \ldots, x_0)$ using an arbitrary associative binary operator $\bullet$ as follows:

$$
\begin{aligned}
y_0 &= x_0 \\
y_1 &= x_1 \bullet x_0 \\
y_2 &= x_2 \bullet x_1 \bullet x_0 \\
\vdots \quad & \quad \vdots \\
y_{n-1} &= x_{n-1} \bullet x_{n-2} \bullet \cdots \bullet x_1 \bullet x_0
\end{aligned} \tag{3.23}
$$

The problem can also be formulated recursively:

$$
\begin{aligned}
y_0 &= x_0 \\
y_i &= x_i \bullet y_{i-1} \; ; \quad i = 1, 2, \ldots, n-1
\end{aligned} \tag{3.24}
$$

In other words, in a prefix problem every output depends on all inputs of equal or lower magnitude, and every input influences all outputs of equal or higher magnitude.



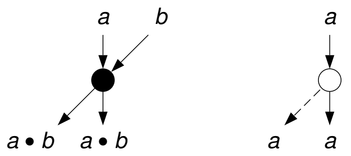
Due to the associativity of the prefix-operator $\bullet$, the individual operations can be carried out in any order. In particular, sequences of operations can be grouped in order to solve the prefix problem partially and in parallel for groups (i.e., sequences) of input bits $(x_i, x_{i-1}, \ldots, x_k)$, resulting in the group variables $Y_{i:k}$. At higher levels, sequences of group variables can again be evaluated, yielding m levels of intermediate group variables, where the group variable $Y^l_{i:k}$ denotes the prefix result of bits $(x_i, x_{i-1}, \ldots, x_k)$ at level l. The group variables of the last level m must cover all bits from i to 0 ($Y^m_{i:0}$) and therefore represent the results of the prefix problem.

$$\begin{aligned} Y^0_{i:i} &= x_i \\ Y^l_{i:k} &= Y^{l-1}_{i:j+1} \bullet Y^{l-1}_{j:k} \ , \ k \le j \le i \ ; \quad l = 1, 2, \ldots, m \\ y_i &= Y^m_{i:0} \ ; \quad i = 0, 1, \ldots, n-1 \end{aligned} \tag{3.25}$$

Note, that for $j = i$ the group variable $Y^{l-1}_{i:k}$ is unchanged (i.e., $Y^l_{i:k} = Y^{l-1}_{i:k}$). Since prefix problems describe a combinational input-to-output relationship, they can be solved by logic networks, which will be the major focus in the following text.

Various serial and parallel algorithms exist for solving prefix problems, depending on the bit grouping properties in Equation 3.25. They result in very different size and delay performance measures when mapped onto a logic network. The major prefix algorithms are now described and visualized by 16-bit examples using a graph representation (see also [LF80, Fic83, LA94]).

In the graphs, the *black nodes* $\bullet$ depict nodes performing the binary associative operation $\bullet$ on its two inputs ($j < i$ in Eq. 3.25), while the *white nodes* $\circ$ represent feed-through nodes with no logic ($j = i$ in Eq. 3.25; in hardware: cells are empty or contain buffers).



Each of the n columns corresponds to one bit position. Black nodes working in parallel are arranged in the same row, and black nodes connected in series are placed in consecutive rows. Thus, the number of rows m corresponds to the maximum number of binary operations to be evaluated in series. The outputs of row l are the group variables $Y^l_{i:k}$. The spacing between rows reflects the amount of interconnect (i.e., number of required wire tracks) between consecutive rows. At the same time, the graphs represent possible hardware topologies if realized in tiled layout.

The following complexity measures are given for each prefix algorithm with respect to logic circuit implementation:

computation time $T_\bullet$*:* black nodes on the critical path or number of rows (levels), $T_\bullet = m$

black node area $A_\bullet$*:* total number of black nodes, important for cell-based designs where the empty white nodes are not of concern

black and white node area $A_{\bullet+\circ}$*:* total number of black and white nodes, which are usually incorporated for regularity reasons in custom layout designs

area-time product $A_\bullet T_\bullet$

interconnect area A_{tracks}*:* total number of horizontal wire tracks used for interconnecting the given hardware topology

maximum fan-out FO_{max}*:* fan-out number of the node with the highest fan-out

The formulae containing an equal sign ("=") are exact for all word length being a power of 2 (i.e., $n = 2^m$), approximations otherwise.

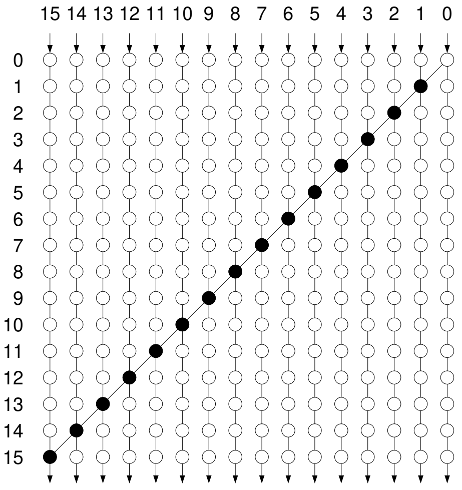
Three categories of prefix algorithms can be distinguished: the serial-prefix, the group-prefix, and the tree-prefix algorithms. In the literature, the tree-prefix algorithms are commonly referred to as parallel-prefix algorithms. The introduction of the new group-prefix algorithms in this thesis, however, makes new naming conventions necessary. Since both algorithms, group-prefix and tree-prefix, include some parallelism for calculation speed-up, they form the category of *parallel-prefix algorithms*.

3.5.2 Serial-prefix algorithm

Equation 3.24 represents a serial algorithm for solving the prefix problem (Fig. 3.14). The *serial-prefix algorithm* needs a minimal number of binary operations $\bullet$ ($O(n)$) but is inherently slow ($O(n)$). Obviously, the $n - 1$ black nodes can be arranged in a single row for hardware implementation, thus eliminating all white nodes (i.e., $A_{\bullet+\circ} = A_\bullet$, $A_{tracks} = 1$).

3.5.3 Tree-prefix algorithms

Unoptimized tree-prefix algorithm According to Equation 3.23 all outputs can be computed separately and in parallel. By arranging the operations



$$
\begin{aligned}
T_\bullet &= n-1 \\
A_\bullet &= n-1 \\
A_{\bullet+\circ} &= n^2 \\
A_\bullet T_\bullet &\approx n^2 \\
A_{tracks} &= 1 \\
FO_{max} &= 2
\end{aligned}
$$

Figure 3.14: *Serial-prefix algorithm.*

• in a tree structure, the computation time for each output can be reduced to $O(\log n)$ (Fig. 3.15). However, the overall number of operations • to be evaluated and with that the hardware costs grow with $(O(n^2))$ if individual evaluation trees are used for each output.

As a trade-off, the individual output evaluation trees can be merged (i.e., common subexpressions be shared) to a certain degree according to different *tree-prefix algorithms*, reducing the area complexity to $O(n \log n)$ or even $O(n)$. Examples are the following algorithms well known from the literature.

Sklansky tree-prefix algorithm Simple overlaying of all output evaluation trees from the unoptimized prefix algorithm leads to the tree-prefix algorithm proposed by Sklansky [Skl60] (Fig. 3.16). Intermediate signals are computed by a minimal tree structure and distributed in parallel to all higher bit positions which require the signal. This leads to a high fan-out of some black nodes ($O(n)$, *unbounded fan-out*), but results in the smallest possible number of node delays (i.e., minimal depth), a small number of signals and very few wiring tracks ($O(\log n)$).

Brent-Kung tree-prefix algorithm A similar structure with quite different characteristics was presented by Brent and Kung [BK82] (Fig. 3.17).

There, the parallel distribution of intermediate signals from the Sklansky algorithm is replaced by a tree-like and partially serial signal propagation. This almost doubles the number of node delays but reduces the number of black nodes to $O(n)$ and limits the fan-out to $\log n$ or even to 3, if the maximum fan-out on single rows is regarded (makes sense if white nodes are allowed to contain buffers). Therefore, this prefix structure is regarded to have bounded fan-out.

Kogge-Stone tree-prefix algorithm The algorithm proposed by Kogge and Stone [KS73] has minimal depth (like Sklansky) as well as bounded fan-out (i.e., maximum fan-out is 2) at the cost of a massively increased number of black nodes and interconnections (Fig. 3.18). This is achieved by using a large number of independent tree structures in parallel.

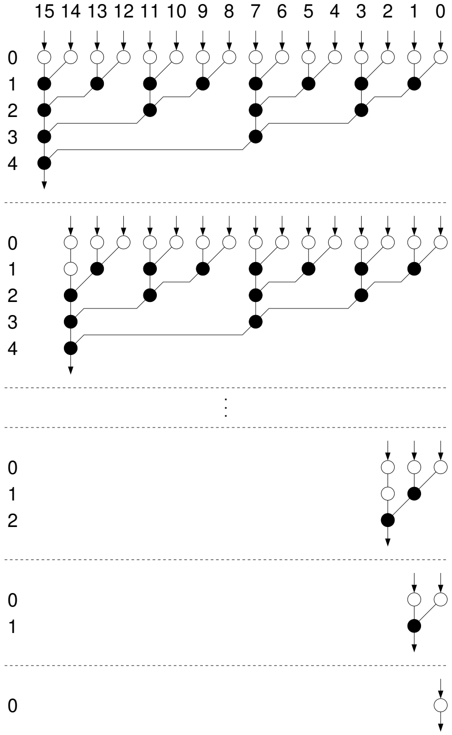
Han-Carlson tree-prefix algorithm Han and Carlson proposed an algorithm which combines the advantages of the Brent-Kung and the Kogge-Stone algorithms by mixing them [HC87]. The first and last k levels are of the Brent-Kung type while the Kogge-Stone graph is used in the middle (typically $k = 1$, Fig. 3.19). The number of parallel trees and thus the number of black nodes and interconnections is reduced at the cost of a slightly longer critical path, compared to the Kogge-Stone algorithm.

The Sklansky prefix algorithm requires additional buffering due to its unbounded fan-out. The Sklansky and Kogge-Stone algorithms are the fastest ones. Depending on the amount of speed degradation caused by high fan-out numbers (Sklansky) as well as large circuit and wiring complexity (Kogge-Stone), their performance measures may differ to a certain degree. The Brent-Kung and Han-Carlson prefix algorithms offer somewhat slower, but more area-efficient solutions.

3.5.4 Group-prefix algorithms

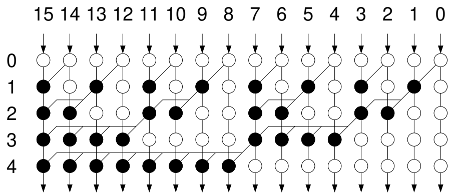
Tree structures typically divide operands into fix-sized (and in most cases minimal) bit groups and apply a maximum number of levels for prefix evaluation. Another approach uses processing of variable-sized bit groups in a fixed number of levels (e.g., one or two levels). The resulting *group-prefix algorithms* again open a wide range of different prefix evaluation strategies.

Fixed-group, 1-level group-prefix algorithms The input operand is divided into m fixed-size bit groups. The prefix result of each group is evaluated



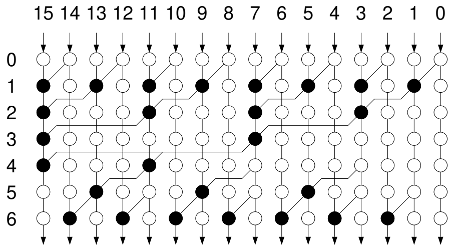
$$
\begin{aligned}
T_\bullet &= \log n \\
A_\bullet &= \tfrac{1}{2}n^2 - \tfrac{1}{2}n \\
A_\bullet T_\bullet &\approx \tfrac{1}{2}n^2 \log n \\
A_{tracks} &= \log n \\
FO_{max} &= \begin{cases} n & \text{at inputs} \\ 1 & \text{internally} \end{cases}
\end{aligned}
$$

Figure 3.15: *Tree-prefix algorithm: unoptimized.*



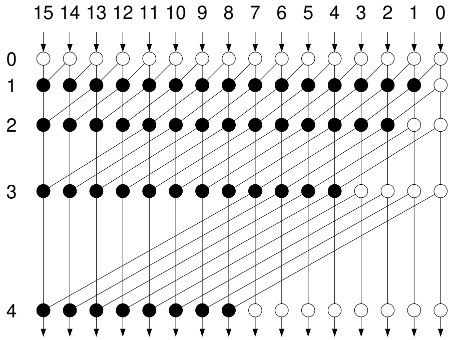
$$
\begin{aligned}
T_\bullet &= \log n \\
A_\bullet &= \tfrac{1}{2}n \log n \\
A_{\bullet+\circ} &= n \log n \\
A_\bullet T_\bullet &\approx \tfrac{1}{2}n \log^2 n \\
A_{tracks} &= \log n \\
FO_{max} &= \tfrac{1}{2}n
\end{aligned}
$$

Figure 3.16: *Tree-prefix algorithm: Sklansky.*



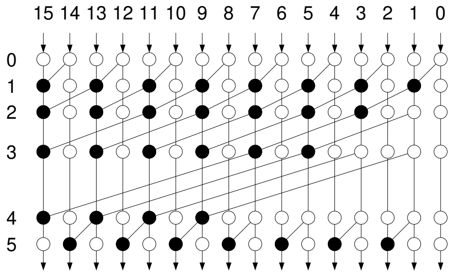
$$
\begin{aligned}
T_\bullet &= 2\log n - 2 \\
A_\bullet &= 2n - \log n - 2 \\
A_{\bullet+\circ} &= 2n \log n - 2n \\
A_\bullet T_\bullet &\approx 4n \log n \\
A_{tracks} &= 2\log n - 1 \\
FO_{max} &= \log n
\end{aligned}
$$

Figure 3.17: *Tree-prefix algorithm: Brent-Kung.*



$$
\begin{aligned}
T_\bullet &= \log n \\
A_\bullet &= n \log n - n + 1 \\
A_{\bullet+\circ} &= n \log n \\
A_\bullet T_\bullet &\approx n \log^2 n \\
A_{tracks} &= n - 1 \\
FO_{max} &= 2
\end{aligned}
$$

Figure 3.18: *Tree-prefix algorithm: Kogge-Stone.*



$$
\begin{aligned}
T_\bullet &= \log n + 1 \\
A_\bullet &= \tfrac{1}{2}n \log n \\
A_{\bullet+\circ} &= n \log n + n \\
A_\bullet T_\bullet &\approx \tfrac{1}{2}n \log^2 n \\
A_{tracks} &= \tfrac{1}{2}n + 1 \\
FO_{max} &= 2
\end{aligned}
$$

Figure 3.19: *Tree-prefix algorithm: Han-Carlson.*

according to the serial-prefix scheme, which is done for all groups in parallel. The result of a group is propagated to all bits of the next higher group in parallel. The final prefix result is in turn computed from the group results again using the serial-prefix scheme. Thus, prefix calculation for individual bit groups is done in parallel at exactly one level. Figures 3.20–3.22 give 16-bit examples for the 1-level group-prefix algorithms with two, four, and m (resp. eight in the graph) bit groups. It can be seen that the number of black nodes in a column never exceeds two, which results in efficient layout topology (i.e., small $A_{\bullet+\circ}$) and low wiring complexity (i.e., small A_{tracks}). The depth of the graph depends on the group sizes, with some intermediate group size for the optimal solution.

Fixed-group, 2-level group-prefix algorithms In the example of Figure3.23 a second level of parallel prefix evaluation is included. Here, many combinations of group sizes at the two levels are possible. The higher parallelism results in larger area but smaller delay complexity.

Fixed-group, multilevel group-prefix algorithms The number of levels for parallel prefix computation can be increased further up to a maximum of $\log n$ levels. Note that by adding a third parallel prefix level to the structure of Figure 3.23, we obtain a $2\times2\times2$ groups, 3-level group-prefix algorithm, which is equivalent to Sklansky's tree-prefix algorithm from Figure 3.16. Thus, Sklansky tree-prefix algorithms and maximum-level group-prefix algorithms are identical.

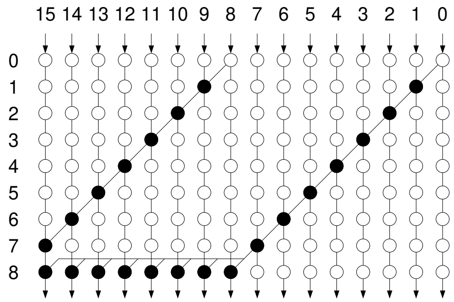
Variable-group, 1-level group-prefix algorithms As can be seen in Figure 3.21 fixed group sizes lead to unnecessary idle times (i.e., white nodes on evaluation paths) at higher bit groups. Their evaluation is completed long before the results from the lower bit groups are obtained for final prefix computation. This can be avoided by using *variable* group sizes. Optimal group sizes are obtained if each group counts one more bit than the preceeding group. Figure 3.24 gives a 16-bit example with group sizes 1, 2, 3, 4, and 5.

Variable-group, 2- and multilevel group-prefix algorithms Again, additional parallel prefix levels can be applied for further delay reduction. The 2-level structure depicted in Figure 3.25 is equivalent to Sklansky's tree-prefix structure (Fig. 3.16) except for the highest bit. This suggests that variable-group, maximum-level group-prefix algorithms also result in the same prefix structure as Sklansky's algorithm. Note that

the 2-level version from Figure 3.25 shows massively increased maximum fan-out for increased adder sizes. This can be avoided by placing some of the black nodes further down in the graph. The resulting optimized structure (Fig. 3.26) has a latency increased by one for some adder sizes but has a much smaller maximum fan-out and counts less black nodes. This structure now resembles the tree-prefix structure of Brent and Kung (Fig. 3.17). Thus, variable-group, maximum-level, optimized group-prefix algorithms are equivalent to the Brent-Kung prefix algorithm.

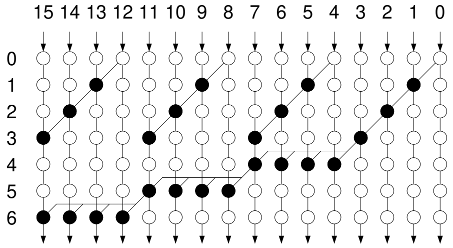
An important property of the group-prefix structures is that the number of $\bullet$-operators per bit position is limited by the number of levels (i.e., max. $\bullet$-operators / bit = $m+1$) and thus is independent of the adder word length. With that, the $\bullet$-operators are more evenly distributed over all bit positions than in the more irregular tree-prefix structures.

A close relation between group-prefix and tree-prefix algorithms, which together form the class of *parallel-prefix* algorithms, can be observed. By applying the maximum number of prefix levels to group-prefix structures, tree-prefix schemes are again obtained. Since distinguishing between group- and tree-prefix schemes is not necessary in the following text, they are both referred to as parallel-prefix schemes.



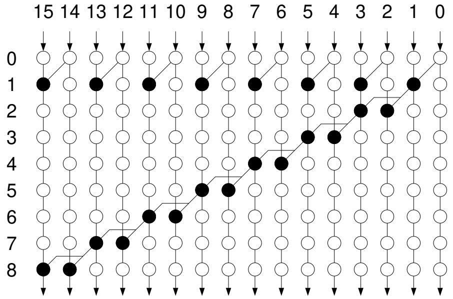
$$
\begin{aligned}
T_\bullet &= \tfrac{1}{2}n \\
A_\bullet &= \tfrac{3}{2}n - 2 \\
A_{\bullet+\circ} &= 2n \\
A_\bullet T_\bullet &\approx \tfrac{3}{4}n^2 \\
A_{tracks} &= 2 \\
FO_{max} &= \tfrac{1}{2}n
\end{aligned}
$$

Figure 3.20: *Group-prefix algorithm: 2 groups, 1-level parallel.*



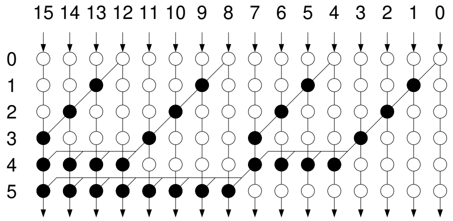
$$\begin{aligned} T_\bullet &= \tfrac{1}{4}n + 2 \\ A_\bullet &= \tfrac{7}{4}n - 4 \\ A_{\bullet+\circ} &= 2n \\ A_\bullet T_\bullet &\approx \tfrac{7}{16}n^2 \\ A_{tracks} &= 2 \\ FO_{max} &= \tfrac{1}{4}n \end{aligned}$$

Figure 3.21: *Group-prefix algorithm: 4 groups, 1-level parallel.*



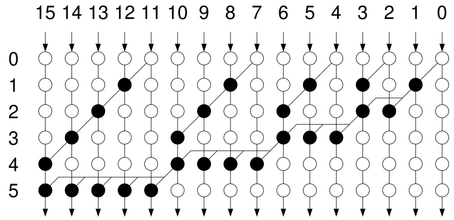
$$\begin{aligned} T_\bullet &= \tfrac{1}{m}n + m - 2 \\ A_\bullet &= 2n - \tfrac{1}{m}n - m \\ A_{\bullet+\circ} &= 2n \\ A_\bullet T_\bullet &\approx \tfrac{2}{m}n^2 \\ A_{tracks} &= 2 \\ FO_{max} &= \tfrac{1}{m}n \end{aligned}$$

Figure 3.22: *Group-prefix algorithm: m (8) groups, 1-level parallel.*



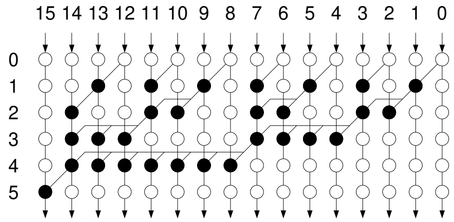
$$\begin{aligned} T_\bullet &= \tfrac{1}{4}n + 1 \\ A_\bullet &= 2n - 4 \\ A_{\bullet+\circ} &= 3n \\ A_\bullet T_\bullet &\approx \tfrac{1}{2}n^2 \\ A_{tracks} &= 3 \\ FO_{max} &= \tfrac{1}{2}n \end{aligned}$$

Figure 3.23: *Group-prefix algorithm: 2×2 groups, 2-level parallel.*



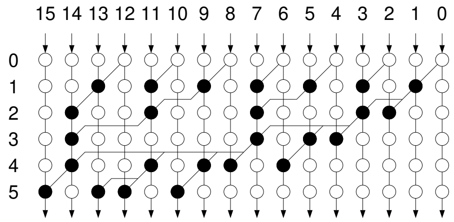
$$\begin{aligned} T_\bullet &\approx \sqrt{2n} \\ A_\bullet &\approx 2n - \sqrt{2n} - 2 \\ A_{\bullet+\circ} &= 2n \\ A_\bullet T_\bullet &\approx 2n\sqrt{2n} \\ A_{tracks} &= 2 \\ FO_{max} &\approx \sqrt{2n} \end{aligned}$$

Figure 3.24: *Group-prefix algorithm: variable groups, 1-level parallel.*



$$\begin{aligned} T_\bullet &\approx \sqrt[3]{6n} \\ A_\bullet &\approx 3n \\ A_{\bullet+\circ} &= 3n \\ A_\bullet T_\bullet &\approx 3n\sqrt[3]{6n} \\ A_{tracks} &= 3 \\ FO_{max} &\approx (6n)^{2/3} \end{aligned}$$

Figure 3.25: *Group-prefix algorithm: variable groups, 2-level parallel.*



$$\begin{aligned} T_\bullet &\approx \sqrt[3]{6n} + 1 \\ A_\bullet &\approx 2n \\ A_{\bullet+\circ} &= 2n \\ A_\bullet T_\bullet &\approx 2n\sqrt[3]{6n} \\ A_{tracks} &= 3 \\ FO_{max} &\approx \sqrt[3]{6n} \end{aligned}$$

Figure 3.26: *Group-prefix algorithm: variable groups, 2-level parallel, optimized.*

3.5.5 Binary addition as a prefix problem

Binary carry-propagate addition can be formulated as a prefix problem using the generate-propagate scheme or the carry-select scheme described in the introduction of the full-adder (Sec. 3.1). The semantics of the prefix operator $\bullet$ and the prefix variables $Y^l_{i:k}$ is defined accordingly in the following.

Generate-propagate scheme

Because the prefix problem of binary carry-propagate addition computes the generation and propagation of carry signals, the intermediate prefix variables can have three different values – i.e., generate a carry 0 (or kill a carry 1), generate a carry 1, propagate the carry-in — and must be coded by two bits. Different codings are possible, but usually a *group generate* $G^l_{i:k}$ and a *group propagate* $P^l_{i:k}$ signal is used forming the *generate/propagate* signal pair $Y^l_{i:k} = (G^l_{i:k},\ P^l_{i:k})$ at level l. The initial prefix signal pairs $(G^0_{i:i},\ P^0_{i:i})$ corresponding to the bit generate g_i and bit propagate p_i signals have to be computed from the addition input operands in a preprocessing step (Eq. (3.27)), also denoted by the operator $\square$. According to Eq. 3.16, the prefix signal pairs of level l are then calculated from the signals of level $l-1$ by an arbitrary prefix algorithm using the binary operation

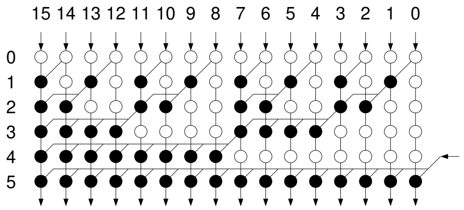
$$\begin{aligned}(G^l_{i:k},\ P^l_{i:k}) &= (G^{l-1}_{i:j},\ P^{l-1}_{i:j}) \bullet (G^{l-1}_{j:k},\ P^{l-1}_{j:k}) \\ &= (G^{l-1}_{i:j} + P^{l-1}_{i:j} G^{l-1}_{j:k},\ P^{l-1}_{i:j} P^{l-1}_{j:k}) \qquad (3.26)\end{aligned}$$

The generate/propagate signals from the last prefix stage $(G^m_{i:0}, P^m_{i:0})$ are used to compute carry signals c_i. The sum bits s_i are finally obtained from a postprocessing step (Eq. (3.29)), represented by the operator $\diamond$.

Combining Equations 3.16 and 3.26 yields the following generate-propagate-based (Fig. 3.4d) addition prefix problem formalism:

$$\begin{aligned}g_i &= a_i b_i \\ p_i &= a_i \oplus b_i \ ; \quad i = 0, 1, \ldots, n-1 \qquad (3.27)\end{aligned}$$

$$\begin{aligned}(G^0_{i:i},\ P^0_{i:i}) &= (g_i,\ p_i) \\ (G^l_{i:k},\ P^l_{i:k}) &= (G^{l-1}_{i:j} + P^{l-1}_{i:j} G^{l-1}_{j:k},\ P^{l-1}_{i:j} P^{l-1}_{j:k})\ , \\ & \quad k \le j \le i \ ; \quad l = 1, 2, \ldots, m\end{aligned}$$



$$\begin{aligned}T_\bullet &= \log n + 1 \\ A_\bullet &= \tfrac{1}{2} n \log n + n \\ A_{\bullet + \circ} &= n \log n + n \\ A_\bullet T_\bullet &\approx \tfrac{1}{2} n \log^2 n \\ A_{tracks} &= \log n + 1 \\ FO_{max} &= n\end{aligned}$$

Figure 3.27: *Sklansky parallel-prefix algorithm with fast carry processing.*

$$c_{i+1} = G^m_{i:0} + P^m_{i:0} c_{in} \ ; \quad i = 0, 1, \ldots, n-1 \qquad (3.28)$$

$$\begin{aligned}s_i &= p_i \oplus c_i \ ; \quad i = 0, 1, \ldots, n-1 \\ & \text{where } c_0 = c_{in} \\ c_{out} &= c_n \qquad (3.29)\end{aligned}$$

Note that an additional level of $\bullet$ operators is added to the prefix graph for accommodating the input carry c_{in}. This comes at the cost of some hardware overhead but allows fast processing of the carry-in. As an example, Figure 3.27 shows the Sklansky parallel-prefix graph with the additional input carry processing level. As an alternative, the carry-in can be incorporated at bit position 0 using a special 3-input $\square$-operator (Eq. (3.30)), leaving the original parallel-prefix graph unchanged (i.e., no additional $\bullet$-level is required, see Eq. (3.31)). This solution comes with negligible carry processing logic but has comparable signal delays on the carry-in and the summand inputs.

$$\begin{aligned}g_0 &= a_0 b_0 + a_0 c_{in} + b_0 c_{in} \\ g_i &= a_i b_i \ ; \quad i = 1, 2, \ldots, n-1 \\ &\ \vdots \qquad (3.30)\end{aligned}$$

$$\begin{aligned}&\ \vdots \\ c_{i+1} &= G^m_{i:0} \ ; \quad i = 0, 1, \ldots, n-1 \qquad (3.31)\end{aligned}$$

In the graph representation of the prefix addition algorithm, an extra row has to be attached for the preprocessing operator $\square$ as well as for the postprocessing operator $\diamond$.

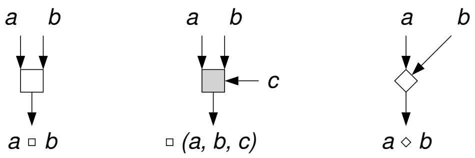


Figure 3.28 shows the graph of a general prefix addition algorithm, where any prefix structure can be used for the central carry-propagation unit. Note that the bit propagate signals p_i have to be routed through the prefix structure because they are reused in the final step for sum bit calculation. Also notice the left-shift of the carry signals c_i by one bit position before the final stage for magnitude adjustment. Two possibilities exist for processing of the carry-in: a slow one (Fig. 3.28b) and a fast one (Fig. 3.28a) which requires one more prefix level. Note that the propagate signals $P^m_{i:0}$ computed in the last prefix level are no longer required, if Eqs. (3.30) and (3.31) are implemented. Therefore, the AND-gate of the bottommost •-operator of each bit position for computing $P^m_{i:0}$ can be saved.

Carry-select scheme

An alternative formulation of the addition prefix problem is based on the carry-select scheme (see Fig. 3.4f). Here, the prefix variable $Y^l_{i:k}$ is encoded by the two possible carry signals $C0^l_{i:k}$ (assuming $c_{in} = 0$) and $C1^l_{i:k}$ (assuming $c_{in} = 1$).

$$
\begin{aligned}
c^0_i &= a_i b_i \\
c^1_i &= a_i + b_i \\
p_i &= a_i \oplus b_i \;; \quad i = 0, 1, \ldots, n-1
\end{aligned}
\tag{3.32}
$$

$$
\begin{aligned}
(C0^0_{i:i},\, C1^0_{i:i}) &= (c^0_i,\, c^1_i) \\
(C0^l_{i:k},\, C1^l_{i:k}) &= (C0^{l-1}_{i:j+1}\overline{C0}^{l-1}_{j:k} + C1^{l-1}_{i:j+1}C0^{l-1}_{j:k}, \\
&\quad\; C0^{l-1}_{i:j+1}\overline{C1}^{l-1}_{j:k} + C1^{l-1}_{i:j+1}C1^{l-1}_{j:k})\;, \\
&\quad\; k \le j \le i \;; \quad l = 1, 2, \ldots, m \\
c_{i+1} &= C0^m_{i:0}\,\overline{c}_{in} + C1^m_{i:0}\,c_{in} \;; \quad i = 0, 1, \ldots, n-1
\end{aligned}
\tag{3.33}
$$

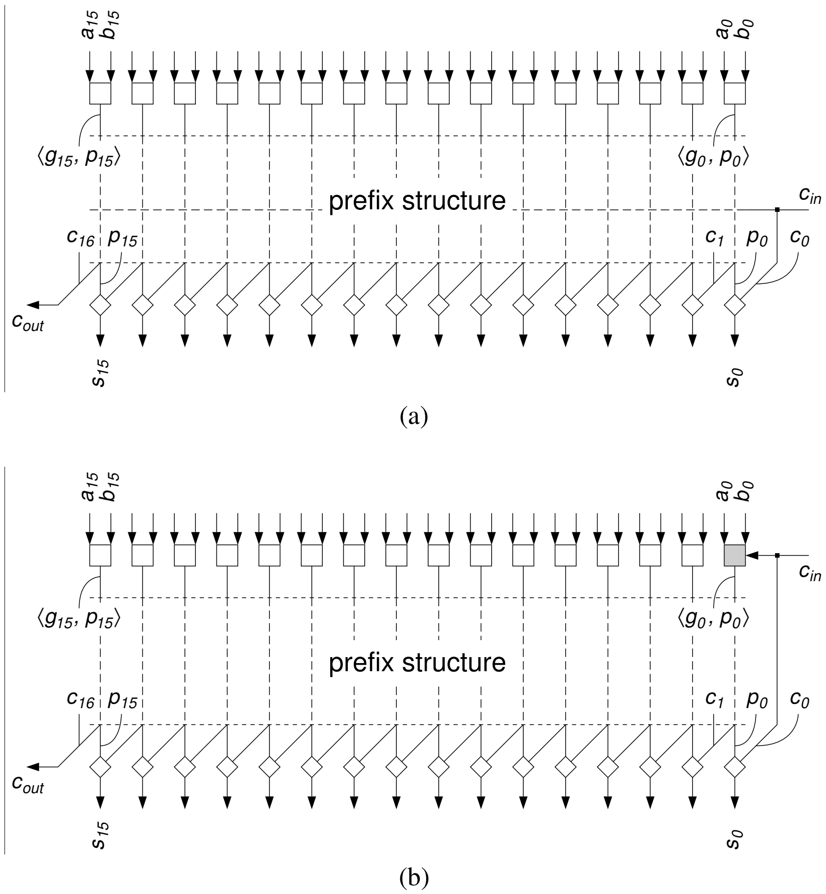


Figure 3.28: *General prefix addition algorithm with (a) fast and (b) slow input carry processing.*

$$
\begin{aligned}
s_i &= p_i \oplus c_i \;; \quad i = 0, 1, \ldots, n-1 \\
&\quad \text{where } c_0 = c_{in} \\
c_{out} &= c_n
\end{aligned}
\tag{3.34}
$$

Basically, the generate-propagate and carry-select schemes are equivalent, and the same prefix algorithms can be used. The carry-select scheme, however, plays only a minor role in cell-based design technologies because its black nodes are composed of two multiplexers instead of the more efficient AND-OR/AND combination used in the generate-propagate scheme.

3.6 Basic Addition Speed-Up Techniques

Carry-propagate adders using the simple ripple-carry algorithm are far too slow for most applications. Several addition speed-up techniques exist, which reduce the computation time by introducing some degree of parallelism at the expense of additional hardware. The underlying principles are summarized in this section.

A carry-propagate adder (CPA) calculates the sum of two input operands while a *partial* CPA adds up only a portion of the operand bits, denoted by $a_{i:k}$ and $b_{i:k}$ (Fig. 3.29).

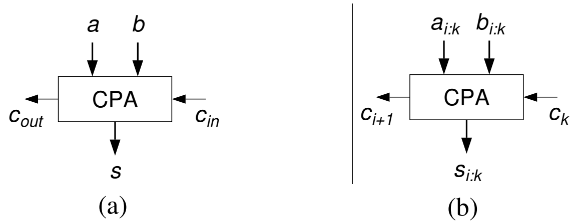


Figure 3.29: *Symbols for (a) CPA and (b) partial CPA.*

First of all, we have to distinguish between the four main input-to-output signal propagation paths in a CPA (Fig. 3.30). Depending on the application, various combinations of signal path timing requirements can arise:

Critical paths: None In low-speed applications, all signal paths in a CPA are non-critical.

Critical paths: All In applications where signal arrival times at all inputs are equal and all outputs of the CPA are expected to be ready at the same time (e.g., when the CPA is the only combinational block between two registers or when the surrounding logic has balanced signal delays), all

1) $T_{\text{CPA}}\ (a, b \rightarrow s)$
2) $T_{\text{CPA}}\ (a, b \rightarrow c_{out})$
3) $T_{\text{CPA}}\ (c_{in} \rightarrow s)$
4) $T_{\text{CPA}}\ (c_{in} \rightarrow c_{out})$

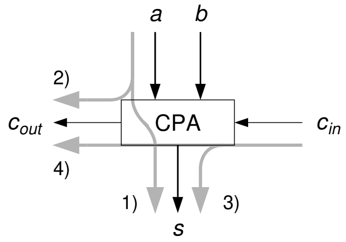


Figure 3.30: *Main signal paths in a CPA.*

signal paths within the CPA are subject to the same timing constraints and thus are equally critical.

Critical paths: 2) + 4) Various applications ask for a *fast carry-out* because this signal controls some subsequent logic network, like e.g. the carry flag in ALUs.

Critical paths: 3) + 4) Other applications require fast input carry propagation due to a *late carry-in* signal provided to the CPA. Some of the addition speed-up techniques introduced in this chapter will rely on that fast carry-in processing property.

Critical paths: 4) Finally, fast carry-in to carry-out propagation is sometimes required. Partial CPAs with late carry-in and fast carry-out properties can again be used for speeding up larger CPAs.

Critical paths: Individual bits In the above cases all bits of the operand and sum vectors were assumed to have equal arrival times. In some applications, however, individual bits arrive at different times, resulting in substantially differing critical paths and more complex timing requirements (e.g., final adder of multipliers). Adders with non-equal input signal arrival profiles will be treated in Section 5.4.

The basic schemes for constructing and speeding up carry-propagate adders can be divided into bit-level and block-level schemes.

3.6.1 Bit-Level or Direct CPA Schemes

Adders using *direct* CPA schemes implement the logic equations of binary addition at the *bit-level* as they are (Eqs. 3.27–3.29). Accordingly, they are built from bit-slices containing the operators $\square$, $\bullet$, and $\diamond$ where some prefix algorithm is used for carry propagation. These adders form the elementary addition structures found in all adder architectures.

Ripple-carry or serial-prefix scheme

The *ripple-carry* addition scheme uses the *serial-prefix* algorithm for carry propagation (Fig. 3.31a).

Properties:

- Minimal combinational adder structure, minimal hardware costs ($O(n)$).
- Slowest adder structure ($O(n)$).
- Used as basic partial CPA in other adder structures.

Carry-lookahead or parallel-prefix scheme

A parallel-prefix algorithm can be used for faster carry propagation (Fig. 3.31b). It results in the *parallel-prefix* or *carry-lookahead* addition scheme, since all carries are precomputed (i.e., "looked ahead") for final calculation of the sum bits.

In the traditional carry-lookahead adders [Kor93], the carries of 4-bit groups are computed in parallel according to the following equations:

$$\begin{aligned}
c_{i+1} &= g_i + c_i p_i \\
c_{i+2} &= g_{i+1} + g_i p_{i+1} + c_i p_i p_{i+1} \\
c_{i+3} &= g_{i+2} + g_{i+1} p_{i+2} + g_i p_{i+1} p_{i+2} + c_i p_i p_{i+1} p_{i+2} \\
c_{i+4} &= g_{i+3} + g_{i+2} p_{i+3} + g_{i+1} p_{i+2} p_{i+3} \\
&\quad + g_i p_{i+1} p_{i+2} p_{i+3} + c_i p_i p_{i+1} p_{i+2} p_{i+3}
\end{aligned} \tag{3.35}$$

Several of these 4-bit structures can be arranged linearly or hierarchically in order to realize carry-lookahead structures for larger word lengths. This carry-lookahead structure is basically one variant of the parallel-prefix scheme.

Properties:

- Increased hardware costs ($O(n \log n)$).
- Speed-up on all signal paths ($O(\log n)$).
- Trade-off between speed-up and hardware overhead exists by using different prefix algorithms.

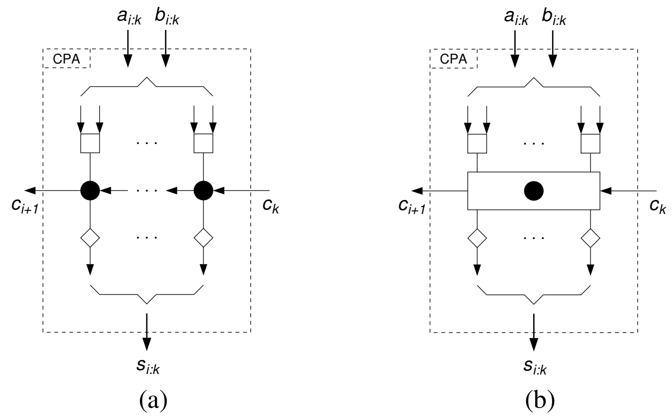


Figure 3.31: *(a) Ripple-carry and (b) carry-lookahead addition schemes at the bit-level.*

3.6.2 Block-Level or Compound CPA Schemes

The other class of addition schemes bases on speeding up carry propagation of existing partial CPAs and combining several of them to form faster and larger adders. Therefore, these adders are *compounded* from one or more CPAs and some additional logic. They work at the *block-level* because bits are always processed in groups (or blocks) by the contained CPAs. A distinction between *concatenation* and *speed-up* schemes can be made: the former is used to build larger adders from smaller ones (i.e., concatenation of several bit groups), while the latter speeds up the processing of a fixed group of bits.

Ripple-carry scheme

The ripple-carry scheme at the block-level is the basic concatenation scheme for constructing larger CPAs from arbitrary smaller CPAs. This is done by concatenating CPAs in series so that a carry ripples through the sequence of partial CPAs (Fig. 3.32a).

Properties:

- Concatenation of CPAs.

Carry-skip scheme

Carry computation for a single bit position, $c_{i+1} = \overline{p_i} g_i + p c_i$ (Eq. (3.12)), can be reformulated for a whole CPA (i.e., group of bits),

$$c_{i+1} \quad = \quad \overline{P_{i:k}} c'_{i+1} + P_{i:k} c_k \tag{3.36}$$

where $P_{i:k}$ denotes the group propagate of the CPA and acts as select signal in this multiplexer structure. c' is the carry-out of the partial CPA (see Fig. 3.32b). Two cases can now be distinguished:

$P_{i:k} = 0$ **:** The carry c'_{i+1} is generated within the CPA and selected by the multiplexer as carry-out c_{i+1}. The carry-in c_k does *not* propagate through the CPA to the carry-out c_{i+1}.

$P_{i:k} = 1$ **:** The carry-in c_k propagates through the CPA to c'_{i+1} but is not selected by the multiplexer. It *skips* the CPA and is directly selected as carry-out c_{i+1} instead. Thus, the combinational path from the carry-in to the carry-out through the CPA is never activated.

In other words, the slow carry-chain path from the carry-in to the carry-out through the CPA is broken by either the CPA itself or the multiplexer. The resulting *carry-skip* addition block therefore is a CPA with small and constant $T_{\text{CPA}}\ (c_k \rightarrow c_{i+1})$, i.e., it can be used for speeding up carry propagation. It is composed from an arbitrary CPA with group propagate output and a 2-to-1 multiplexer (Fig. 3.32b).

In the literature, an OR-gate is often used instead of the multiplexer (e.g., [Kor93]). This, however, speeds up only 0-to-1 transitions on the carry path;

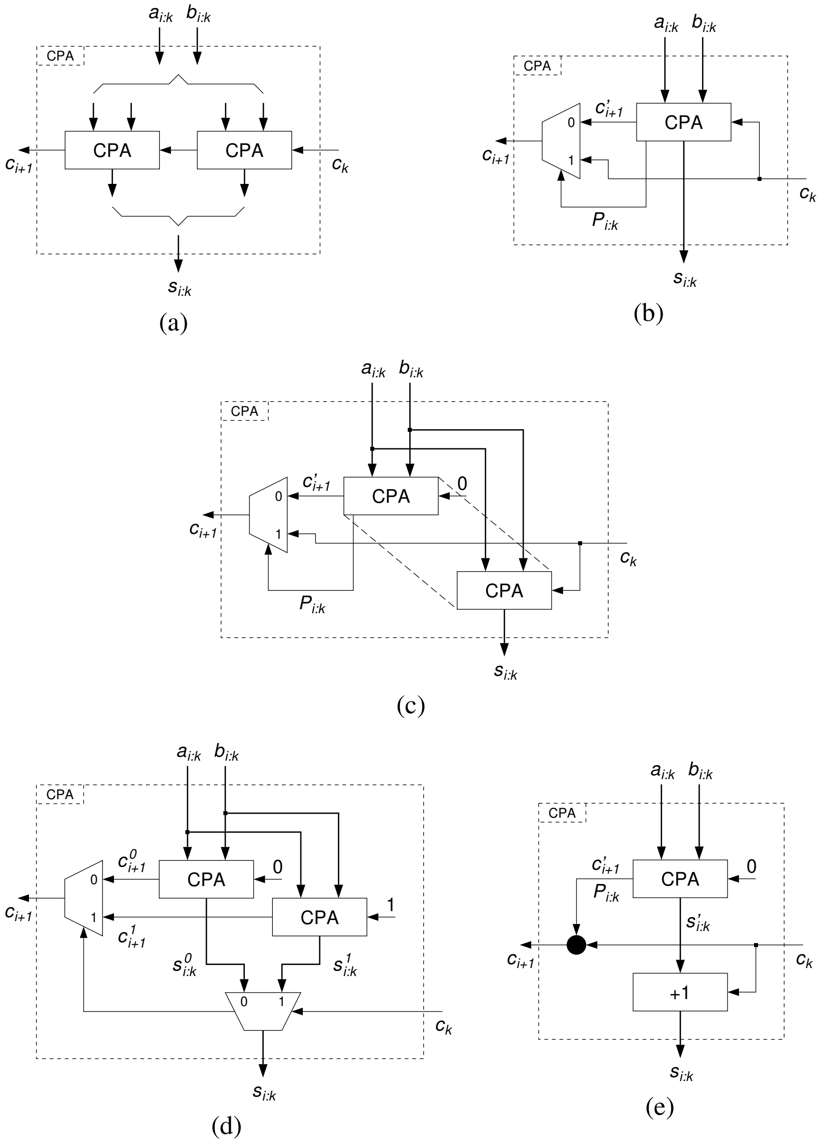


Figure 3.32: *(a) Ripple-carry, (b) redundant and (c) irredundant carry-skip, (d) carry-select, and (e) carry-increment addition schemes at the block-level.*

for 1-to-0 transitions the adder works as ripple-carry adder. Applications are limited to implementations using precharging (e.g., dynamic logic).

Note that the multiplexer in this circuit is logically redundant, i.e., the signals c'_{i+1} and c_{i+1} are logically equivalent and differ only in signal delays. The carry-in c_k has a reconvergent fan-out. This inherent logic redundancy results in a false longest path (i.e., combinational signal path that is never sensitized) which leads from the carry-in through the CPA to the carry-out. This poses a problem in automatic logic optimization and static timing analysis. Due to computation complexity of these tools, the logic state of a circuit and thus path sensitization usually is not considered [C$^+$94, MB89]. Also, testability is concerned, since a non-working skip mechanism can not be logically detected (redundant faults). An additional detection capability is therefore required. These faults are also called delay faults, because they only affect circuit delay, but not logic behavior). Redundancy removal techniques exist which base on duplication of the carry-chain in the CPA: one carry-chain computes the carry-out c'_{i+1} without a carry-in, while the other takes the carry-in for calculation of the sum bits [KMS91, SBSV94]. Figure 3.32c shows the basic principle where the reconvergent fan-out of c_k is eliminated. Note that not the entire CPA but only the carry-propagation chain has to be duplicated (i.e., the logic of the two CPAs can be merged to a certain degree) which, however, still signifies a considerable amount of additional logic compared to the redundant carry-skip scheme.

Properties:

- Constant signal delay $T_{\text{CPA}}\ (c_k \rightarrow c_{i+1})$.
- Inherent logic redundancy.
- Small hardware overhead: group propagate logic and single multiplexer.
- Medium hardware overhead for irredundant version: double carry-chain.

Carry-select scheme

The basic problem faced in speeding up carry propagation is the fast processing of a late carry input. Since this carry-in can have only two values (0 and 1), the two possible addition results ($s^0_{i:k}$, c^0_{i+1} resp. $s^1_{i:k}$, c^1_{i+1}) can be precomputed and selected afterwards by the late carry-in c_k using small and constant time:

$$s_{i:k} = \overline{c_k} s^0_{i:k} + c_k s^1_{i:k} \tag{3.37}$$

$$c_{i+1} = \overline{c_k} c^0_{i+1} + c_k c^1_{i+1} \tag{3.38}$$

The resulting *carry-select* addition scheme requires two CPAs — one with $c_k = 0$ and the other with $c_k = 1$ — and a 2-to-1 multiplexer for each sum bit and the carry-out (Fig. 3.32d).

Properties:

- Constant signal delays $T_{\text{CPA}}\ (c_k \rightarrow c_{i+1})$ and $T_{\text{CPA}}\ (c_k \rightarrow s_{i:k})$.
- High hardware overhead: double CPA and multiplexers.

Carry-increment scheme

In the *carry-increment* addition scheme only the result with carry-in 0 is precomputed ($s'_{i:k}$) and incremented by 1 afterwards, if $c_k = 1$. The carry-out c_{i+1} is calculated from the CPA's carry-out c'_{i+1} and group propagate $P_{i:k}$ using the $\bullet$-operator of binary addition (Fig. 3.32e):

$$s_{i:k} = s'_{i:k} + c_k \tag{3.39}$$

$$\begin{aligned} c_{i+1} &= G_{i:k} + P_{i:k} c_k \\ &= c'_{i+1} + P_{i:k} c_k \end{aligned} \tag{3.40}$$

where $c'_{i+1} = G_{i:k}$ since the carry-in to the CPA is 0. The required incrementer circuit provides constant-time carry propagation and is much cheaper than the additional CPA and selection circuitry used in the carry-select scheme. Also, the logic of the CPA and the incrementer can be merged to some extent (see Sec. 4.1.5).

Properties:

- Constant signal delays $T_{\text{CPA}}\ (c_k \rightarrow c_{i+1})$ and $T_{\text{CPA}}\ (c_k \rightarrow s_{i:k})$.
- Medium hardware overhead: incrementer, group propagate logic, and $\bullet$-operator of Eq. 3.26.

3.6.3 Composition of Schemes

The direct and compound addition schemes presented above can now be composed arbitrarily in order to realize larger and faster adders. Note that each scheme results in a generic CPA which again can be used in compound addition schemes, allowing for linear and hierarchical compositions.

Table 3.1 gives an overview of the basic addition speed-up schemes and their characteristics. The block-level ripple-carry scheme is the natural (and only) way to compose larger adders from partial CPAs by propagating the carry from the lower to the upper bit group (concatenation scheme). All compound speed-up schemes (skip, select, and increment) only provide propagation speed-ups on signal paths starting at the carry input. They can be used either for adder applications with late carry-in requirements or, by appropriate combination, for realization of fast CPAs. The carry-lookahead scheme is the only addition scheme which provides a speed-up on all signal path without relying on the composition of different schemes (i.e., direct speed-up scheme).

Table 3.1: *Speed-up characteristics of addition schemes.*

speed-up paths	ripple	skip	select	increment	look-ahead
$T_{\text{CPA}}\ (c_{in} \rightarrow c_{out})$		$\surd$	$\surd$	$\surd$	$\surd$
$T_{\text{CPA}}\ (c_{in} \rightarrow s_i)$			$\surd$	$\surd$	$\surd$
$T_{\text{CPA}}\ (a_i, b_i \rightarrow c_{out})$					$\surd$
$T_{\text{CPA}}\ (a_i, b_i \rightarrow s_i)$					$\surd$

Linear compositions

CPAs can be arranged *linearly* by repeated application of the concatenation scheme. Put differently, input operands can be divided into bit groups which are processed by serially concatenated partial CPAs. The ripple-carry nature of the concatenation scheme leads to late carry-in signals at high order CPA, which can be compensated by making use of the fast carry processing properties of the compound speed-up schemes. This is why linear arrangements of compound addition schemes, which by themselves only speed-up propagation of the carry-in, allow the construction of adders with speed-up on all signal paths.

As an example, Figure 3.33a shows the adder structure resulting from composition of the carry-increment and the concatenation scheme. Note the speed-up on the critical carry path by the fast carry processing of the second carry-increment CPA.

Hierarchical compositions

Hierarchical compositions are possible by repeated application of concatenation *and* speed-up schemes. The resulting structures make use of arbitrary speed-up schemes at arbitrary hierarchy levels in order to achieve further speed improvement. Figure 3.33b depicts an adder structure resulting from application of the carry-increment, the concatenation, and again the carry-increment scheme. Note that in the hierarchical version the input carry is processed faster than in the linear one.

Pure and mixed composition

Pure compositions are linear or hierarchical compositions which make use of only one speed-up scheme. *Mixed* compositions try to take advantage of different speed-up schemes by combining them. Some compromise with respect to area and speed can be achieved by mixing slow and area-efficient schemes with fast but area-intensive ones.

Circuit simplifications

Each speed-up scheme requires some additional circuitry, which often can be combined with the existing CPA logic. Especially hierarchical compositions allow for massive circuit simplifications in many cases.

Group sizes

Depending on the position of individual bit groups within an adder, partial CPAs may have different arrival times of carry-in and carry-out signals. This leads to varying computation times for the individual partial CPAs and thus to different group sizes. In compound speed-up schemes, groups at higher bit positions are typically made larger in order to take full advantage of the late carry-in signals. Optimal group sizes are determined by equalizing all signal paths or, in other words, by maximizing all groups with the restriction of a given overall adder delay.

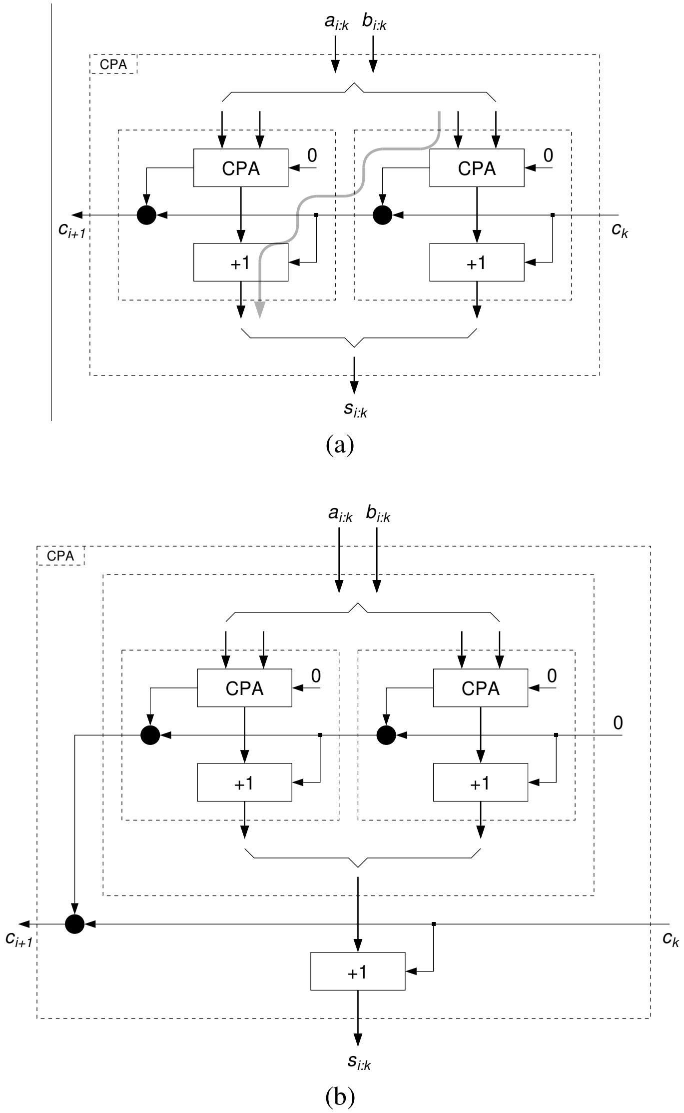


Figure 3.33: *(a) Linear and (b) hierarchical composition of carry-increment schemes.*

4

Adder Architectures

This chapter discusses — based on the structures and schemes introduced in the previous chapter — the various circuit architectures that exist for binary addition. Their complexities and performance are compared with focus on cell-based design techniques.

4.1 Anthology of Adder Architectures

The efficient implementation of adder circuits does not only rely on optimal composition of speed-up schemes but also includes potential circuit simplifications and optimizations. This leads us to the various adder architectures described in this chapter.

The circuit structure of every architecture will be given by the set of logic equations for the composing bit slices. Maximum adder and group sizes for a given adder delay are summarized in a table. Finally, exact time and area complexities are given for each architecture based on the unit-gate model.

4.1.1 Ripple-Carry Adder (RCA)

The *ripple-carry adder* (RCA) has already been introduced as the basic and simplest carry-propagate adder in the previous chapter. It is composed of a series of full-adders (**fa**), where the initial full-adder (**ifa**) may use a majority-gate for fast carry computation. The corresponding logic equations, adder sizes, and complexity measures are given below. The table for the adder sizes gives the maximum number of bits n that can be computed within the given delay T.

Logic equations:

ifa	$c_1 = a_0b_0 + a_0c_0 + b_0c_0$ $s_0 = a_0 \oplus b_0 \oplus c_0$	$A = 9$
fa	$g_i = a_ib_i$ $p_i = a_i \oplus b_i$ $c_{i+1} = g_i + p_ic_i$ $s_i = p_i \oplus c_i$	$A = 7$

Adder sizes vs. time:

T	4	4	8	16	32	64	128	256	512
n	1	2	4	8	16	32	64	128	256

Complexity:

$$T_{\text{RCA}} = 2n$$
$$A_{\text{RCA}} = 7n + 2$$

4.1.2 Carry-Skip Adder (CSKA)

Composition of the concatenation scheme and the carry-skip scheme yields the *carry-skip adder* (CSKA).

1-level redundant carry-skip adder (CSKA-1L)

The *1-level* carry-skip adder (CSKA-1L) is composed of a series of skipping groups (or blocks) and an initial full-adder (**ifa**) at the LSB (see Fig. 3.32b). Each skipping group consists of a series of full-adders (**bfa**) with additional group propagate signal generation (P_i), an initial full-adder (**bifa**) at the group LSB, and a final carry-generator (**bcg**) at the group MSB. c_{pb} and c_{tb} denote the carry-out of the previous and the current (i.e., "this") block, respectively.

Highest speed is achieved by sizing the bit groups individually. Because the skipping scheme only speeds up $T_{\text{CPA}}\,(c_k \rightarrow c_i)$ but not $T_{\text{CPA}}\,(c_k \rightarrow s_i)$, carry generation starts and carry redistribution ends in slow ripple-carry blocks. Therefore, groups at the lower and upper end are smaller while groups in the middle can be made larger. Since the delay through a full-adder equals the delay of a multiplexer under the unit-gate model asumption, neighboring groups differ in size by one bit. k is the size of the largest group.

Logic equations:

ifa	$c_{tb} = a_0b_0 + a_0c_0 + b_0c_0$ $s_0 = a_0 \oplus b_0 \oplus c_0$	$A = 9$
bifa	$p_i = a_i \oplus b_i$ $P_i = p_i$ $c_{i+1} = a_ib_i + a_ic_{pb} + b_ic_{pb}$ $s_i = p_i \oplus c_{pb}$	$A = 9$
bfa	$g_i = a_ib_i$ $p_i = a_i \oplus b_i$ $P_i = p_iP_{i-1}$ $c_{i+1} = g_i + p_ic_i$ $s_i = p_i \oplus c_i$	$A = 8$
bcg	$c_{tb} = \overline{P_i}c_{i+1} + P_ic_{pb}$	$A = 3$

Adder and group sizes vs. time:

T	4	4	8	10	12	14	16	18	20	22	24	26	28	30	32	$\cdots$	46
k			2	2	3	3	4	4	5	5	6	6	7	7	8	$\cdots$	11
n	1	2	5	7	10	13	17	21	26	31	37	43	50	57	65	$\cdots$	133

Complexity:

$$T_{\text{CSKA-1L}} = 4k$$
$$A_{\text{CSKA-1L}} = 8n + 6k - 6$$

$$k = \lceil \sqrt{n-1} \rceil$$

1-level irredundant carry-skip adder (CSKA-1L')

The inherent logic redundancy of the carry-skip adder can be removed. This leads to the *1-level irredundant carry-skip adder* (CSKA-1L') [KMS91, SBSV94]. The basic bit-slice counts two unit-gates more than the conventional carry-skip adder.

Logic equations:

ifa	$c_{tb} = a_0b_0 + a_0c_0 + b_0c_0$ $s_0 = a_0 \oplus b_0 \oplus c_0$	$A = 9$
bifa	$p_i = a_i \oplus b_i$ $P_i = p_i$ $c_{i+1} = a_ib_i + a_ic_{pb} + b_ic_{pb}$ $c'_{i+1} = a_ib_i$ $s_i = p_i \oplus c_{pb}$	$A = 9$
bfa	$g_i = a_ib_i$ $p_i = a_i \oplus b_i$ $P_i = p_iP_{i-1}$ $c_{i+1} = g_i + p_ic_i$ $c'_{i+1} = g_i + p_ic'_i$ $s_i = p_i \oplus c_i$	$A = 8$
bcg	$c_{tb} = \overline{P}_ic'_{i+1} + P_ic_{pb}$	$A = 3$

Adder and group sizes vs. time:

T	4	4	8	10	12	14	16	18	20	22	24	26	28	30	32	$\cdots$	46
k			2	2	3	3	4	4	5	5	6	6	7	7	8	$\cdots$	11
n	1	2	5	7	10	13	17	21	26	31	37	43	50	57	65	$\cdots$	133

Complexity:

$$\begin{aligned} T_{\mathrm{CSKA-1L}} &= 4k \\ A_{\mathrm{CSKA-1L}} &= 10n - 8k - 6 \end{aligned}$$

$$k = \lceil \sqrt{n-1} \rceil$$

2-level carry-skip adder (CSKA-2L)

Hierarchical application of the carry-skip scheme results in multilevel carry-skip adders. The *2-level* carry-skip adder (CSKA-2L) contains a series of second-level blocks which are composed from the initial full-adder (**bifa**^{2}), the final carry-generator (**bcg**^{2}) and an intermediate series of first-level blocks (**bifa**^{1} + **bfa**^{1} + **bcg**^{1}). Each level has its own carry and group propagate signal (c^l_{tb}, P^l_{tb}).

Optimal block sizes become highly irregular for multilevel carry-skip adders and cannot be expressed by exact formulae. This problem was intensively addressed in the literature [Hob95, Kan93, CSTO91, Tur89, GHM87].

Logic equations:

ifa	$c^2_{tb} = a_0b_0 + a_0c_0 + b_0c_0$ $s_0 = a_0 \oplus b_0 \oplus c_0$	$A = 9$
bifa^2	$p_i = a_i \oplus b_i$ $P^2_{tb} = p_i$ $c^1_{tb} = a_ib_i + a_ic^2_{pb} + b_ic^2_{pb}$ $s_i = p_i \oplus c^2_{pb}$	$A = 9$
bifa^1	$p_i = a_i \oplus b_i$ $P^1_i = p_i$ $c_{i+1} = a_ib_i + a_ic^1_{pb} + b_ic^1_{pb}$ $s_i = p_i \oplus c^1_{pb}$	$A = 9$
bfa^1	$g_i = a_ib_i$ $p_i = a_i \oplus b_i$ $P^1_i = p_iP^1_{i-1}$ $c_{i+1} = g_i + p_ic_i$ $s_i = p_i \oplus c_i$	$A = 8$
bcg^1	$P^2_{tb} = P^1_iP^2_{pb}$ $c^1_{tb} = \overline{P}^1_ic_{i+1} + P^1_ic^1_{pb}$	$A = 4$
bcg^2	$c^2_{tb} = \overline{P}^2_{tb}c^1_{tb} + P^2_{tb}c^2_{pb}$	$A = 3$

Adder sizes vs. time:

T	10	12	14	16	18	20	24	28	30	32
n	7	11	15	22	29	41	67	103	121	152

Complexity:

$$
\begin{aligned}
T_{\text{CSKA}-2\text{L}} &= O(n^{\frac{1}{3}}) \\
A_{\text{CSKA}-2\text{L}} &= 8n + O(n^{\frac{1}{3}})
\end{aligned}
$$

4.1.3 Carry-Select Adder (CSLA)

A *carry-select adder* (CSLA) is the composition of the concatenation and the selection scheme. Each bit position includes the generation of two sum (s_i^0, s_i^1) and carry bits (c_i^0, c_i^1) and the selection multiplexers for the correct sum bit. The correct carry bit is selected at the end of a block (**bcg**).

Because the signal paths $T_{\text{CPA}}\ (c_k \rightarrow c_i)$ as well as $T_{\text{CPA}}\ (c_k \rightarrow s_i)$ are sped up by the selection scheme, groups can be made larger towards the MSB.

Logic equations:

ifa	$c_{tb} = a_0 b_0 + a_0 c_0 + b_0 c_0$ $s_0 = a_0 \oplus b_0 \oplus c_0$	$A = 9$
biha	$g_i = a_i b_i$ $p_i = a_i \oplus b_i$ $c_{i+1}^0 = g_i$ $c_{i+1}^1 = g_i + p_i$ $s_i^0 = p_i$ $s_i^1 = \overline{p}_i$ $s_i = \overline{c}_{pb} s_i^0 + c_{pb} s_i^1$	$A = 7$
bfa	$g_i = a_i b_i$ $p_i = a_i \oplus b_i$ $c_{i+1}^0 = g_i + p_i c_i^0$ $c_{i+1}^1 = g_i + p_i c_i^1$ $s_i^0 = p_i \oplus c_i^0$ $s_i^1 = p_i \oplus c_i^1$ $s_i = \overline{c}_{pb} s_i^0 + c_{pb} s_i^1$	$A = 14$
bcg	$c_{tb} = c_{i+1}^0 + c_{pb} c_{i+1}^1$	$A = 2$

Adder and group sizes vs. time:

T	4	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34
k		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
n	1	2	4	7	11	16	22	29	37	46	56	67	79	92	106	121	139

Complexity:

$$
\begin{aligned}
T_{\text{CSLA}} &= 2k + 2 \\
A_{\text{CSLA}} &= 14n - 5k - 5
\end{aligned}
\qquad k = \left\lceil \tfrac{1}{2}\sqrt{8n-7} - \tfrac{1}{2} \right\rceil
$$

4.1.4 Conditional-Sum Adder (COSA)

Also for the carry-select adder, multiple application of the selection scheme is possible. Because the selection scheme rests on duplication of the partial CPA, the hardware overhead in multilevel structures becomes prohibitively large due to the repeated CPA duplication. However, since both results (i.e., carry-in 0 and 1) are available at each level, only the multiplexers (rather than the entire CPA) have to be duplicated in order to get an additional selection level (Fig. 4.1).

A carry-select adder with a maximum number of levels ($= \log n$) and using the above simplification scheme is called *conditional-sum adder* (COSA). Group sizes start with one bit at the lowest level and are doubled at each additional level. Figure 4.2 depicts the circuit structure of a conditional-sum adder.

The logic formulae are organized in levels instead of bit groups. In the first level (**csg**), both possible carry and sum bits are generated for each bit position ($c_i^{0,0}$, $c_i^{1,0}$, $s_i^{0,0}$, $s_i^{1,0}$). The following levels select new carry and sum bit pairs ($c_i^{0,l}$, $c_i^{1,l}$, $s_i^{0,l}$, $s_i^{1,l}$) for increasingly larger bit groups (**ssl**, **csl**). The last level performs final carry and sum bit selection (**fssl**, **fcsl**).

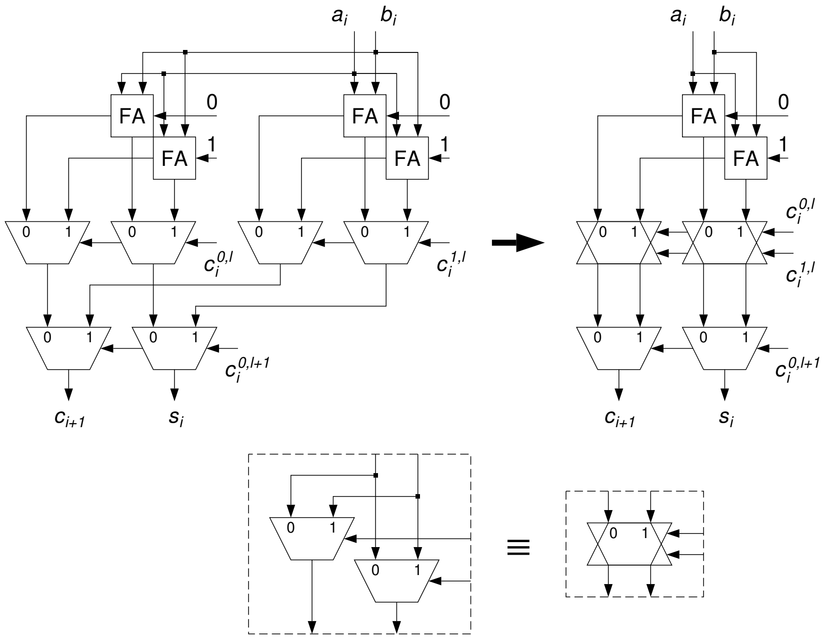


Figure 4.1: *Multilevel carry-select simplifications.*

Logic equations:

csg	$c_{i+1}^{0,0} = a_i b_i$ $c_{i+1}^{1,0} = a_i + b_i$ $s_i^{0,0} = a_i \oplus b_i$ $s_i^{1,0} = \overline{a_i \oplus b_i}$	$A = 6$
ssl	$s_i^{0,l+1} = s_i^{0,l}\overline{c}_j^{0,l} + s_i^{1,l}c_j^{0,l}$ $s_i^{1,l+1} = s_i^{0,l}\overline{c}_j^{1,l} + s_i^{1,l}c_j^{1,l}$	$A = 6$
csl	$c_i^{0,l+1} = c_i^{0,l} + c_i^{1,l}c_j^{0,l}$ $c_i^{1,l+1} = c_i^{0,l} + c_i^{1,l}c_j^{1,l}$	$A = 4$
fssl	$s_i = s_i^{0,m}\overline{c}_0 + s_i^{1,m}c_0$	$A = 3$
fcsl	$c_{i+1}^{l} = c_i^{0,l} + c_i^{1,l}c_0$	$A = 2$

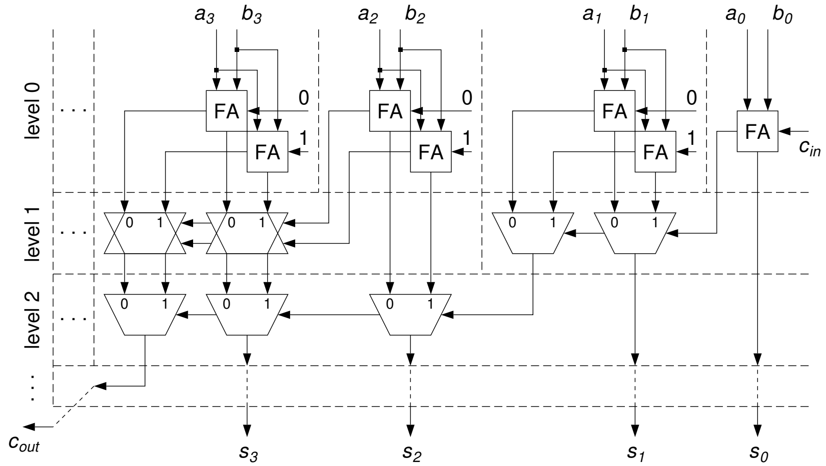


Figure 4.2: *Conditional-sum adder structure.*

Adder sizes and number of levels vs. time:

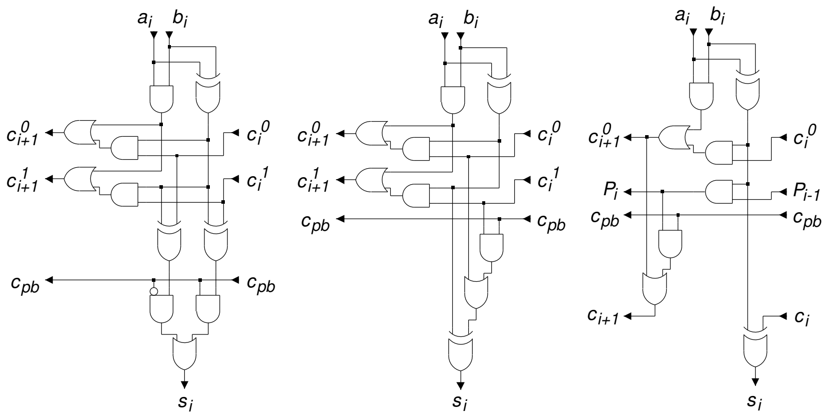
T	4	4	6	8	10	12	14	16	18
m		1	2	3	4	5	6	7	8
n	1	2	4	8	16	32	64	128	256

Complexity:

$$T_{\text{COSA}} = 2\log n + 2$$
$$A_{\text{COSA}} = 3n\log n + 7n - 2\log n - 7$$

4.1.5 Carry-Increment Adder (CIA)

The *carry-increment adder* (CIA) results from combining the concatenation scheme with the incrementation scheme. Repeated application of the incrementation scheme yields multilevel carry-increment adders. However, simple attachment of an incrementer circuit to the partial CPA does not result in a very efficient circuit structure. Although an incrementer can provide constant delay from the carry-in to the outputs, carry-propagation has to be performed for



(a) : $A = 14$ (b) : $A = 11$ (c) : $A = 10$

Figure 4.3: *(a), (b) Carry-select and (c) carry-increment adder cells.*

all input operand bits, which can be done only with $O(\log n)$. Therefore, the adder delays $T_{\text{CPA}}\ (a, b \rightarrow s)$ and $T_{\text{CPA}}\ (a, b \rightarrow c_{out})$ are increased massively due to double carry-propagation.

However, the logic of the adder and the incrementer can be combined so that only a single carry has to be propagated. This circuit structure was first presented by Tyagi [Tya93]. A comprehensive description of its derivation from a carry-select adder structure is given in this section [ZK].

Multilevel carry-increment structures allow for even further optimizations, resulting in one of the most efficient gate-level adder architectures. As will become clear soon, the carry-increment structures correspond to the group-prefix algorithms using the generate-propagate scheme presented in Section 3.5.

1-level carry-increment adder (CIA-1L)

In his reduced-area scheme for carry-select adders [Tya93], Tyagi shows how the two ripple-chains for both possible block-carry-in values in a typical carry-select adder can be replaced by one ripple-chain and some additional increment logic. Let us start with the logic equations of a carry-select adder bit-slice (Fig. 4.3a) with a gate count of 14:

Table 4.1: *Signal descriptions.*

signal	description
a_i, b_i, s_i	ith primary adder input/output bit
c_{in}, c_{out}, c_i	carry-in, carry-out, ith carry bit
$c_i^0, c_i^1, s_i^0, s_i^1$	ith carry/sum bits for block-carry-in = 0/1
g_i, p_i	ith generate/propagate bit
$c_{pb}, c_{pb}^1, c_{pb}^2$	carry-out of previous (level-1/2) block
$c_{tb}, c_{tb}^1, c_{tb}^2$	carry-out of this (level-1/2) block
P_i, P_i^1, P_i^2	(level-1/2) block propagate up to ith bit
$P_{pb}, P_{pb}^1, P_{pb}^2$	propagate of previous (level-1/2) block
$P_{tb}, P_{tb}^1, P_{tb}^2$	propagate of this (level-1/2) block

$$\begin{aligned} g_i &= a_i\, b_i \ , \ p_i = a_i \oplus b_i \\ c_{i+1}^0 &= g_i + p_i\, c_i^0 \ , \ c_{i+1}^1 = g_i + p_i\, c_i^1 \\ s_i^0 &= p_i \oplus c_i^0 \ , \ s_i^1 = p_i \oplus c_i^1 \ , \ s_i = \overline{c_{pb}}\, s_i^0 + c_{pb}\, s_i^1 \end{aligned} \tag{4.1}$$

where $c_i^0(c_i^1)$ denotes the carry at the ith bit position with block-carry-in 0(1) and c_{pb} is the carry output of the previous block. Table 4.1 gives a summary of all signal names used and their meanings.

In a first step, the order of the XOR and multiplexer operation for the sum bit computation can be reversed, resulting in $c_{i+1} = \overline{c_{pb}}\, c_i^0 + c_{pb}\, c_i^1$ and $s_i = p_i \oplus c_i$. Since $c_i^1 = c_i^0 + c_i^1$ holds for the two carries (i.e., $c_i^0 = 1 \Rightarrow c_i^1 = 1$), the first equation can be reduced to $c_i = c_i^0 + c_{pb}\, c_i^1$. Thus, the simplified carry-select adder bit-slice (Fig. 4.3b) counts 11 gates and computes

$$\begin{aligned} g_i &= a_i\, b_i \ , \ p_i = a_i \oplus b_i \\ c_{i+1}^0 &= g_i + p_i\, c_i^0 \ , \ c_{i+1}^1 = g_i + p_i\, c_i^1 \\ c_i &= c_i^0 + c_{pb}\, c_i^1 \ , \ s_i = p_i \oplus c_i \end{aligned} \tag{4.2}$$

The following transformations show that $c_{i+1}^1 = g_i + p_i\, c_i^1$ can be reduced to $c_{i+1}^{1'} = p_i\, c_i^1$:

$$\begin{aligned} c_{i+1} &= c_{i+1}^0 + c_{pb}\, c_{i+1}^1 \\ &= g_i + p_i\, c_i^0 + c_{pb}\,(g_i + p_i\, c_i^1) \\ &= g_i + p_i\, c_i^0 + c_{pb}\, g_i + c_{pb}\, p_i\, c_i^1 \\ &= g_i + p_i\, c_i^0 + c_{pb}\, p_i\, c_i^1 \\ &= c_{i+1}^0 + c_{pb}\, c_{i+1}^{1'} \end{aligned} \tag{4.3}$$

Here, $c_{i+1}^{1'}$ becomes a block propagate for bit positions k through i and is renamed P_i (k is the first bit of the block and $P_{k-1} = 1$). Also, the principle of sum bit selection has changed to an incrementer structure, where each sum bit is toggled depending on the carry of the previous bit position, the carry of the previous block, and the actual block propagate. Therefore this adder type is referred to as *1-level carry-increment* adder. Its basic full-adder bit-slice counts 10 gates and contains the following logic (Fig. 4.3c):

$$\begin{aligned} & g_i = a_i\, b_i\ ,\ p_i = a_i \oplus b_i\ ,\ P_i = p_i\, P_{i-1} \\ & c_{i+1}^0 = g_i + p_i\, c_i^0\ ,\ c_{i+1} = c_{i+1}^0 + P_i\, c_{pb}\ ,\ s_i = p_i \oplus c_i \end{aligned} \tag{4.4}$$

The carry-out c_{i+1} of the last slice in a block is the block-carry-out c_{tb}. The AND-OR gates determining c_{i+1} have now been rearranged (i.e., moved into the previous slice) in order to get a more regular block structure (see Fig. 4.4). Note that the delay from c_{pb} to s_i increased while the delay from c_i^0 to s_i decreased compared with the original carry-select cell (Fig. 4.3a), which, however, has no effect on block sizes or overall addition speed.

The entire adder structure is depicted in Figure 4.4 and implements the variable-group, 1-level group-prefix algorithm from Figure 3.24. As demonstrated, only three logically different slices are required, and their arrangement is linear and straightforward. The unit-gate delay model used yields the optimal block and maximum adder sizes given below.

As an example, an adder with 24 gate delays can have a maximum of 67 bits with block sizes of 2, 3, ..., 11. Note that each block counts one more bit than its predecessor (same as in Tyagi [Tya93]), and that each additional block adds two gate delays to the adder. The carry computation in the first slice can be sped up using a fast majority gate (i.e., $c_0 = a_0\, b_0 + a_0\, c_{in} + b_0\, c_{in}$) which is counted here as two gate delays.

Note also that the block-carry c_{pb} is only connected to one gate instead of two gates in the carry-select adder cell (Fig. 4.3). Since this is the only

signal with unbounded fan-out within the carry-increment adder, the maximum fan-out is cut in half compared to a carry-select adder.

Logic equations:

ifa	$c_{tb} = a_0 b_0 + a_0 c_0 + b_0 c_0$	$A = 9$
	$s_0 = a_0 \oplus b_0 \oplus c_0$	
biha	$g_i = a_i b_i$	$A = 7$
	$p_i = a_i \oplus b_i$	
	$P_i = p_i$	
	$c_{i+1}^0 = g_i$	
	$c_{i+1} = c_{i+1}^0 + P_i c_{pb}$	
	$s_i = p_i \oplus c_{pb}$	
bfa	$g_i = a_i b_i$	$A = 10$
	$p_i = a_i \oplus b_i$	
	$P_i = p_i P_{i-1}$	
	$c_{i+1}^0 = g_i + p_i c_i^0$	
	$c_{i+1} = c_{i+1}^0 + P_i c_{pb}$	
	$s_i = p_i \oplus c_{i+1}$	
bcg	$c_{tb} = c_{i+1}$	$A = 0$

Adder and group sizes vs. time:

T	4	4	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36
k		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n	1	4	7	11	16	22	29	37	46	56	67	79	92	106	121	137	154

Complexity:

$$\begin{aligned} T_{\text{CIA-1L}} &= 2k + 2 \\ A_{\text{CIA-1L}} &= 10n - k + 2 \end{aligned}$$

$$k = \lceil \frac{1}{2}\sqrt{8n-7} - \frac{1}{2} \rceil < \lceil \sqrt{2n} \rceil$$

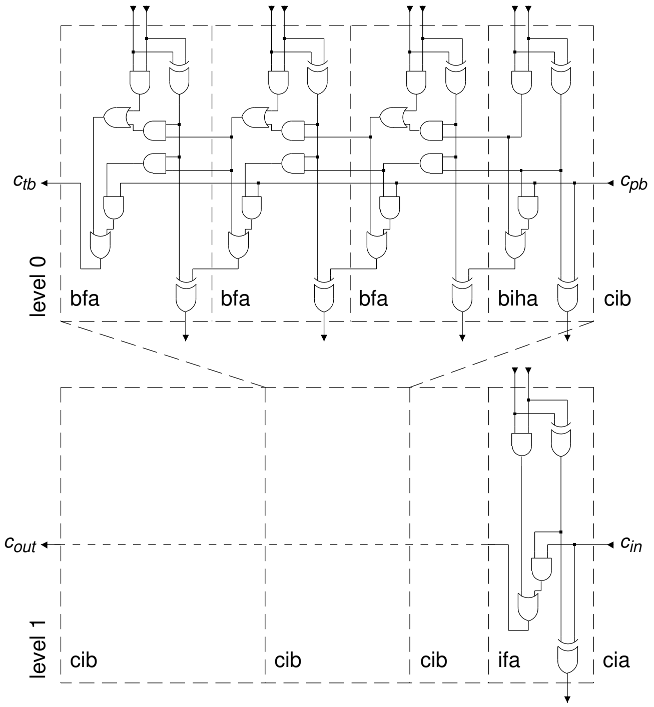


Figure 4.4: *1-level carry-increment adder structure.*

2-level carry-increment adder (CIA-2L)

For further speed improvement, Tyagi proposes a select-prefix adder scheme where the ripple-carry blocks of his carry-select adder are replaced by parallel-prefix blocks. The resulting delay reduction is considerable, but at the expense of a massively increased area occupation.

The basic idea of the new adder scheme to be introduced in this thesis is to exchange the ripple-carry blocks of the carry-increment adder by a second level of carry-increment adders. This section shows how the two levels of increment logic can be merged in a structure which makes use of the same basic full-adder cell as the 1-level adder. The resulting 2-level carry-increment adder will have nearly the same size as the 1-level version. Yet, the size of the

largest ripple-carry block and thus the delay grows only with $O(n^{1/3})$ instead of $O(n^{1/2})$, which substantially increases speed for medium and large word lengths.

Let us now derive the structure and the bit-slice logic of the 2-level carry-increment adder (see Fig. 4.6). A first-level increment block (**cib1**) consists of full-adder slices with ripple-carry propagation, whereas the second-level increment blocks (**cib2**) are composed of first-level blocks. Finally, the whole adder is composed of several second-level increment blocks. Each second-level block gets c^2_{pb} as block-carry-in and advances its carry-out c^2_{tb} to the next block. The inputs to a first-level block are c^1_{pb} and c^2_{pb} as block-carry-ins of levels 1 and 2, and P^2_{pb} as propagate signal from all previous first-level blocks within the same second-level block.

By adding the second-level increment logic to the formulae of the 1-level carry-increment bit-slice, we obtain:

$$\begin{aligned} g_i &= a_i\, b_i \;,\; p_i = a_i \oplus b_i \\ P^1_i &= p_i\, P^1_{i-1} \;,\; P^2_i = p_i\, P^2_{i-1} \\ c^0_{i+1} &= g_i + p_i\, c^0_i \;,\; c_{i+1} = c^0_{i+1} + P^1_i\, c^1_{pb} + P^2_i\, c^2_{pb} \\ s_i &= p_i \oplus c_i \end{aligned} \tag{4.5}$$

Additionally, each first-level block has to compute its propagate and carry-out signal,

$$P^2_{tb} = P^1_{tb}\, P^2_{pb} \;,\; c^1_{tb} = c^0_{tb} + P^1_{tb}\, c^1_{pb} \tag{4.6}$$

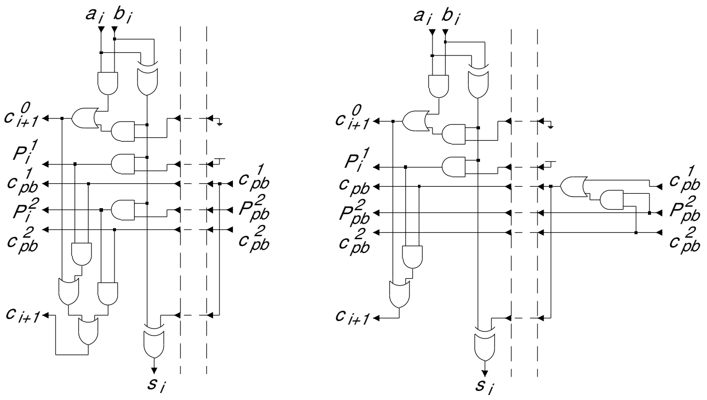
and each second-level block its carry-out,

$$c^2_{tb} = c^1_{tb} + P^2_{tb}\, c^2_{pb} \tag{4.7}$$

P^1_{tb} and c^0_{tb} denote P^1_i and c^0_i of the last first-level block slice, whereas P^2_{tb} is used as P^2_{i-1} of the first slice in the next first-level block.

However, this adder slice has an increased gate count of 13 (Fig. 4.5a). Since P^2_i can be expressed by $P^2_i = P^1_i\, P^2_{pb}$, the following transformations are possible:

$$\begin{aligned} c_{i+1} &= c^0_{i+1} + P^1_i\, c^1_{pb} + P^2_i\, c^2_{pb} \\ &= c^0_{i+1} + P^1_i\, c^1_{pb} + P^1_i\, P^2_{pb}\, c^2_{pb} \\ &= c^0_{i+1} + P^1_i\, (c^1_{pb} + P^2_{pb}\, c^2_{pb}) \\ &= c^0_{i+1} + P^1_i\, c_{pb} \end{aligned} \tag{4.8}$$



(a) (b)

Figure 4.5: *2-level increment adder cell with (a) 13 and (b) 10 gates.*

where $c_{pb} = c^1_{pb} + P^2_{pb}\, c^2_{pb}$ is constant within each first-level increment block and can be precomputed once (Fig. 4.5b right). Thus, the simplified full-adder bit-slice has again the same structure as the 1-level adder slice with a gate count of 10 (Fig. 4.5b left part):

$$\begin{aligned}
&g_i = a_i\, b_i \;,\; p_i = a_i \oplus b_i \;,\; P^1_i = p_i\, P^1_{i-1} \\
&c^0_{i+1} = g_i + p_i\, c^0_i \;,\; c_{i+1} = c^0_{i+1} + P^1_i\, c_{pb} \;,\; s_i = p_i \oplus c_i
\end{aligned} \tag{4.9}$$

Furthermore, every first-level block has to compute

$$\begin{aligned}
&P^2_{tb} = P^1_{pb}\, P^2_{pb} \\
&c^1_{tb} = c^0_{tb} + P^1_{tb}\, c^1_{pb} \;,\; c_{tb} = c^1_{tb} + P^2_{tb}\, c^2_{pb}
\end{aligned} \tag{4.10}$$

once, while the block-carry-out c^2_{tb} of a second-level block corresponds to the carry-out c_{tb} of its last first-level block.

The resulting 2-level carry-increment adder structure is depicted in Figure 4.6. It is interesting to note that it exactly implements the variable-group, 2-level, optimized group-prefix algorithm of Figure 3.26. By omitting the simplification of Figure 4.5 an adder structure equivalent to the variable-group, 2-level group-prefix algorithm of Figure 3.25 is obtained. As can be seen, all the gates needed can be arranged such that every bit-slice contains the same 10 gates, with the exception of some smaller slices. However, some slices differ in routing, and one additional small slice is required for the final carry-out generation, thus resulting in 6 logically different slices. The linear arrangement of the slices is again straightforward and thus perfectly suited for tiled-layout and automated layout generation as well as for standard cells.

The block size computation for the 2-level carry-increment adder is still quite simple and can be expressed by exact formulae. With respect to block sizes, note again that each first-level block counts one more bit than its predecessor, and that each second-level block counts one more first-level block than its predecessor. Thus an increase of the overall delay by two gates allows the adder to be expanded by an additional (larger) second-level block.

As was demonstrated, the 2-level carry-increment adder consists of the same basic cell as the 1-level version and has only slightly larger cells at the beginning of each increment block. Thus the massive speed improvement by the second increment level comes at negligible additional area costs.

Adder and group sizes vs. time:

T	4	6	10	12	14	16	18	20	22	24	26	28
k_1			2	3	4	5	6	7	8	9	10	11
k_2		2	4	7	11	16	22	29	37	46	56	67
n	1	3	9	16	27	43	65	94	131	177	233	300

Complexity:

$$\begin{aligned}
T_{\mathrm{CIA-2L}} &= 2k_1 + 6 \\
A_{\mathrm{CIA-2L}} &= 10n - k_1 + 1
\end{aligned}$$

$$k_1 = \left\lceil l^{1/3} - \frac{5}{3}\frac{1}{l^{1/3}} - 1 \right\rceil < \left\lceil \sqrt[3]{6(n-1)} \right\rceil ,$$

$$l = \frac{1}{9}\sqrt{729n^2 - 2916n + 3291} + 3n - 6$$

Multilevel carry-increment adders

Carry-increment adders with more than two increment levels are built by applying the same scheme as for the 2-level adder repeatedly. The example of a 3-level carry-increment adder shows that the gate count increase remains

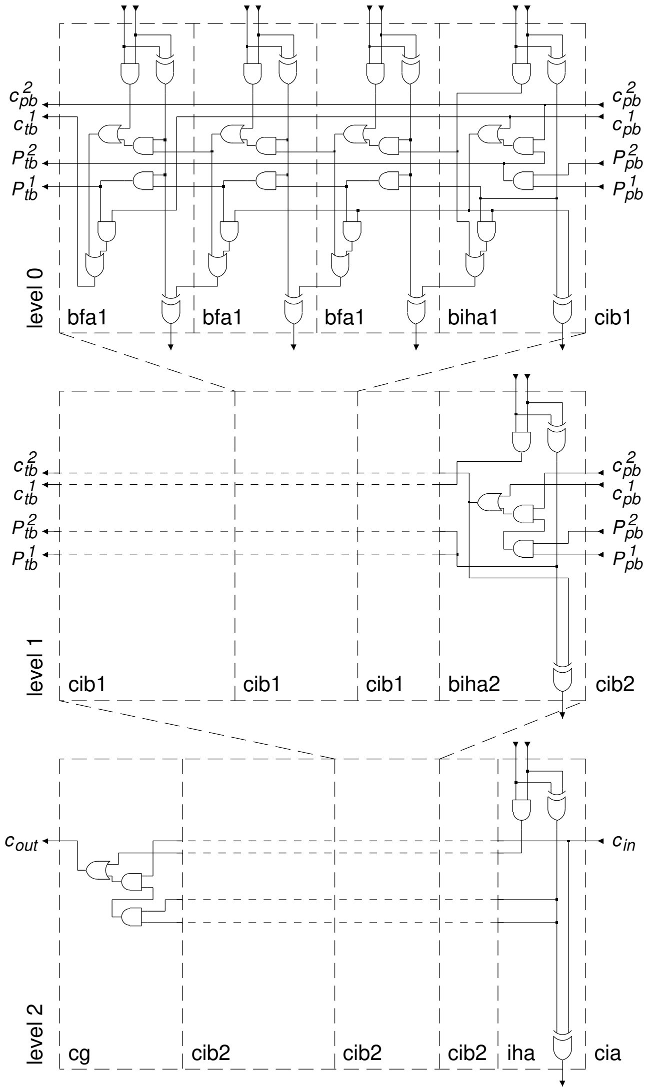


Figure 4.6: *2-level carry-increment adder structure.*

small, whereas a gate-delay reduction is achieved only for adder sizes larger than 64 bits. This holds true for a larger number of levels as well. Also, the circuit structure becomes more complex, and the upper limit of 10 gates is exceeded for some bit-slices. Therefore, the 2-level carry-increment adder seems to be the best compromise between high area-time performance and low circuit complexity for adder sizes of up to 128 bits.

Carry-increment and parallel-prefix structures

At this point, it can be observed that the carry-increment adders again have the same basic adder structure as the parallel-prefix or carry-lookahead adders in that they consist of a preprocessing, carry-propagation, and a postprocessing stage. A closer look even reveals the carry-propagation structure of an m-level carry-increment adder to be equivalent to the m-level group-prefix algorithms with variable groups described in Section 3.5. Thus, the carry-increment adders belong to the family of parallel-prefix adders.

4.1.6 Parallel-Prefix / Carry-Lookahead Adders (PPA / CLA)

Parallel-prefix adders (PPA) are adders using the direct parallel-prefix scheme for fast carry computation. They are also called *carry-lookahead adders* (CLA). As mentioned in Section 3.5, different parallel-prefix algorithms exist, resulting in a variety of adders with different performances. They all have the initial generate and propagate signal generation (**igpg**, **gpg**) and the final sum bit generation (**sg**) and differ only in the arrangement of the intermediate m carry generation levels (**cg**).

Usually, binary or 2-bit architectures are used, i.e., the prefix operator processes two bits or, in other words, block sizes of two bits are used in the first level.

Logic equations:

igpg	$g_0^0 = a_0b_0 + a_0c_0 + b_0c_0$ $p_0^0 = a_0 \oplus b_0$	$A = 7$
gpg	$g_i^0 = a_ib_i$ $p_i^0 = a_i \oplus b_i$	$A = 3$
cg	$g_i^{l+1} = g_i^l + p_i^l g_j^l$ $p_i^{l+1} = p_i^l p_j^l$	$A = 3$
	$c_{i+1} = g_i^m$	
sg	$s_i = p_i^0 \oplus c_i$	$A = 2$

Sklansky parallel-prefix algorithm (PPA-SK)

Adder sizes and number of levels:

T	4	6	8	10	12	14	16	18	20
m		1	2	3	4	5	6	7	8
n	1	2	4	8	16	32	64	128	256

Complexity:

$$\begin{aligned} T_{\text{PPA-SK}} &= 2 \log n + 4 \\ A_{\text{PPA-SK}} &= \frac{3}{2} n \log n + 4n + 5 \end{aligned}$$

Brent-Kung parallel-prefix algorithm (PPA-BK)

Adder sizes and number of levels:

T	4	6	8	12	16	20	24	28	32
m		1	2	4	6	8	10	12	14
n	1	2	4	8	16	32	64	128	256

Complexity:

$$\begin{aligned} T_{\text{PPA-BK}} &= 4 \log n \\ A_{\text{PPA-BK}} &= 10n - 3 \log n - 1 \end{aligned}$$

Kogge-Stone parallel-prefix algorithm (PPA-KS)

Adder sizes and number of levels:

T	4	6	8	10	12	14	16	18	20
m		1	2	3	4	5	6	7	8
n	1	2	4	8	16	32	64	128	256

Complexity:

$$\begin{aligned} T_{\text{PPA-KS}} &= 2 \log n + 4 \\ A_{\text{PPA-KS}} &= 3n \log n + n + 8 \end{aligned}$$

Multi-bit parallel-prefix adders

The prefix operator for binary addition can be adapted so that it processes several bits at a time (i.e., block sizes larger than two). The corresponding logic becomes more complex, but the resulting prefix algorithm counts less levels.

The standard *carry-lookahead adder* (CLA) described in the literature (e.g., [Kor93]) is actually a 4-bit Brent-Kung parallel-prefix adder. Here, two phases for carry-propagation can be distinguished: in the first phase (**cg1**) the carry bits for every fourth bit position are computed. The second phase (**cg2**) then calculates all the remaining carries from the carries of phase one.

Logic equations:

igpg	$g_0^0 = a_0b_0 + a_0c_0 + b_0c_0$ $p_0^0 = a_0 \oplus b_0$	$A = 7$
gpg	$g_i^0 = a_ib_i$ $p_i^0 = a_i \oplus b_i$	$A = 3$
cg1	$g_{i3}^{l+1} = g_{i3}^l + p_{i3}^l g_{i2}^l + p_{i3}^l p_{i2}^l g_{i1}^l + p_{i3}^l p_{i2}^l p_{i1}^l g_{i0}^l$ $p_{i3}^{l+1} = p_{i3}^l p_{i2}^l p_{i1}^l p_{i0}^l$	$A = 12$
	$c_{i+1} = g_i^{m/2}$	
cg2	$g_{i2}^{l+1} = g_{i2}^l + p_{i2}^l g_{i1}^l + p_{i2}^l p_{i1}^l g_{i0}^l + p_{i2}^l p_{i1}^l p_{i0}^l c_{i0}$ $g_{i1}^{l+1} = g_{i1}^l + p_{i1}^l g_{i0}^l + p_{i1}^l p_{i0}^l c_{i0}$ $g_{i0}^{l+1} = g_{i0}^l + p_{i0}^l c_{i0}$	$A = 16$
	$c_{i+1} = g_i^m$	
sg	$s_i = p_i^0 \oplus g_{i-1}^l$	$A = 2$

Adder sizes and number of levels vs. time:

T	4	6	8	12	16	20	24	28	32
m		1	1	2	3	4	5	6	7
n	1	2	4	8	16	32	64	128	256

Complexity:

$$\begin{aligned} T_{\text{CLA}} &= 4 \log n \\ A_{\text{CLA}} &= 14n - 20 \end{aligned}$$

4.1.7 Hybrid Adder Architectures

The adder architectures presented up to now were clean architectures, i.e., no mixing of different speed-up schemes was done. However, the generic nature of most speed-up schemes allows for arbitrary combination of those. Since every scheme comes with some different kind of additional circuitry, mixing them up results in relatively high circuit overhead.

Hybrid adder architectures, which are mainly used in full-custom implementations of ALUs and multipliers [D+92, G+94, M+94, OV95, O+95, M+91], were marginally considered in this study. The advantages of these architectures seem to lie in the efficient implementation of specific sub-blocks using advanced circuit techniques, such as pass-transistor logic or dynamic logic (e.g., Manchester-chain adders [Kor93]), which are not compatible with cell-based technologies. Unit-gate model based investigations on various hybrid adder architectures from the literature have not shown any performance advantages of such architectures. Put differently, all addition speed-up techniques seem to reveal their full potential when consistently applied to as large blocks as possible instead of mixing them up.

The most often used hybrid adder architecture uses carry-lookahead blocks with one final carry-select stage [SP92]. Under the unit-gate delay model, speed is exactly the same as for a pure carry-lookahead adder. The gate count, however, is increased drastically due to the multiplexer stage, which is expensive in cell-based technologies.

4.2 Complexity and Performance Comparisons

This section summarizes the results obtained from comparing the adder architectures presented. Comparisons include the unit-gate models for area and delay as well as placed and routed standard-cell implementations.

4.2.1 Adder Architectures Compared

All main adder architectures were compared for word lengths of 8, 16, 32, 64, and 128 bits with carry input and output. The realization of the ripple-carry adder (RCA[4]) is straightforward. The 1/2-level carry-skip adders (CSKA-1L/-2L), the 1-level carry-select adders (CSLA-1L), and the 1/2/3-level carry-increment adders (CIA-1L/-2L/-3L) were implemented using variable block sizes. The optimal block sizes were determined by minimizing the overall circuit delay and equalizing all parallel signal paths under the given unit-gate delay model. Minimization was achieved by constructing adders with maximum block sizes and numbers of bits for some given delays and cutting them down to the required adder sizes [Tur89]. The block sizes for the carry-select adders are the same as for the carry-increment adders which are given in the tables of the previous sections. Irredundant carry-skip adders were not

[4] All adder acronyms are summarized in Table 4.3 with a short architecture description.

implemented because no efficient circuit solutions can be expected. The same holds true for multilevel carry-select adders.

Three types of parallel-prefix architectures were chosen: the unbounded fan-out structure used by Sklansky [Skl60, LA94] (PPA-SK) and the bounded fan-out prefix structures by Brent and Kung [BK82] (PPA-BK) and by Kogge and Stone [KS73] (PPA-KS). The conditional-sum adders (COSA) use the unbounded fan-out prefix structure by Sklansky. Finally, a carry-lookahead adder (CLA) with 4-bit blocks was chosen as a representative of the ordinary carry-lookahead scheme [Hwa79]. As already mentioned, this scheme corresponds to PPA-BK with a blocking factor of four rather than two. For adder sizes not being a power of four, CLA uses 2-bit blocks in the first level.

Other adder architectures were also studied but not included in the comparisons here because they do not provide better performance for cell-based design techniques than the above architectures they are derived from. They include various parallel-prefix [WT90, HC87, KOIH92], conditional-sum [LA94], and carry-skip [GHM87, CSTO91] adders as well as some hybrid architectures [D$^+$92, G$^+$94, M$^+$94, OV95, O$^+$95, M$^+$91] which partly are not suited for standard-cell implementation due to special circuit techniques.

4.2.2 Comparisons Based on Unit-Gate Area and Delay Models

With respect to asymptotic time and area complexity, binary adder architectures can be divided into four primary classes. Table 4.2 lists these classes with their complexity measures and their associated adder schemes. n denotes the operand word length, while l corresponds to the number of levels in multi-level architectures. The first class consists of the ripple-carry adder. The second class contains the compound adder schemes – i.e., carry-skip, carry-select and carry-increment — with fixed number of levels and variable block sizes. Carry-lookahead and some parallel-prefix adders form the third class while some other parallel-prefix adders and the conditional-sum adder belong to the fourth adder class.

Table 4.3 lists the unit-gate count, unit-gate delay, and gate-count$\times$gate-delay product (gc$\times$gd-product) complexities for all investigated adder architectures as a function of the word length n (all adders with carry-in). Because some of the exact formulae are quite complex, only their asymptotic behavior is given by the highest exponent term. Note that PPA-BK and CLA belong to

Table 4.2: *Classification of adder architectures.*

area	delay	AT-product	adder schemes
$O(n)$	$O(n)$	$O(n^2)$	ripple-carry
$O(n)$	$O(n^{\frac{1}{l+1}})$	$O(n^{\frac{l+2}{l+1}})$	carry-skip, carry-select, carry-increment
$O(n)$	$O(\log n)$	$O(n \log n)$	carry-lookahead, parallel-prefix
$O(n \log n)$	$O(\log n)$	$O(n \log^2 n)$	parallel-prefix, conditional-sum

the third class with linear gate-count complexity only if circuit size is of concern (e.g., in cell-based designs). Custom layout solutions, however, usually require $O(n \log n)$ area because of regularity reasons (i.e., n bits $\times$ $O(\log n)$ prefix levels).

The exact unit-gate count and unit-gate delay numbers for all adder architectures and sizes are given in Tables 4.4 and 4.5. Table 4.6 gives their gate-count$\times$gate-delay products relative to the reference 2-level carry-increment adder.

4.2.3 Comparison Based on Standard-Cell Implementations

After schematic entry, automatic timing and area minimization was performed on all circuits in order to optimize performance under standard-cell library specifications. Circuit size and static timing (i.e., critical path) information was extracted from the layout after place and route by taking into account the actual wiring contributions as well as ramp delays at the primary inputs and outputs based on typical external driving strength and capacitive load values (fan-out = 1). A "prop-ramp" delay model was used which accounts for cell propagation and output ramp delays (as a function of the attached load), but not for input signal slopes. Finally, the average power consumption was calculated based on standard-cell power as well as extracted node capacitance and transition information obtained from gate-level simulation with a set of

Table 4.3: *Asymptotic adder complexities (unit-gate model).*

adder type	gate count	gate delays	gc×gd-product	architecture description
RCA	$7\,n$	$2\,n$	$14\,n^2$	ripple-carry
CSKA-1L	$8\,n$	$4\,n^{1/2}$	$32\,n^{3/2}$	1-level carry-skip
CSKA-1L'	$10\,n$	$4\,n^{1/2}$	$40\,n^{3/2}$	irredundant 1-level carry-skip
CSKA-2L	$8\,n$	$x\,n^{1/3}$ *	$x\,n^{4/3}$ *	2-level carry-skip
CSLA-1L	$14\,n$	$2.8\,n^{1/2}$	$39\,n^{3/2}$	1-level carry-select
CIA-1L	$10\,n$	$2.8\,n^{1/2}$	$28\,n^{3/2}$	1-level carry-increment
CIA-2L	$10\,n$	$3.6\,n^{1/3}$	$36\,n^{4/3}$	2-level carry-increment
CIA-3L	$10\,n$	$4.4\,n^{1/4}$	$44\,n^{5/4}$	3-level carry-increment
CLA	$14\,n$	$4\,\log n$	$56\,n\,\log n$	"standard" carry-lookahead
PPA-BK	$10\,n$	$4\,\log n$	$40\,n\,\log n$	parallel-prefix (Brent-Kung)
PPA-SK	$3/2\,n\,\log n$	$2\,\log n$	$3\,n\,\log^2 n$	parallel-prefix (Sklansky)
PPA-KS	$3\,n\,\log n$	$2\,\log n$	$6\,n\,\log^2 n$	parallel-prefix (Kogge-Stone)
COSA	$3\,n\,\log n$	$2\,\log n$	$6\,n\,\log^2 n$	conditional-sum (Sklansky)

* The exact factors for CSKA-2L have not been computed due to the highly irregular optimal block sizes.

Table 4.4: *Gate count.*

adder	word length [bits]				
type	8	16	32	64	128
RCA	**58**	**114**	**226**	**450**	**898**
CSKA-1L	76	146	286	554	1090
CSKA-2L	71	158	323	633	1248
CSLA-1L	87	194	403	836	1707
CIA-1L	78	157	314	631	1266
CIA-2L	79	158	316	635	1273
CIA-3L	80	159	324	639	1280
CLA	92	204	428	876	1772
PPA-SK	73	165	373	837	1861
PPA-BK	70	147	304	621	1258
PPA-KS	88	216	520	1224	2824
COSA	115	289	687	1581	3563

Table 4.5: *Gate delay.*

adder	word length [bits]				
type	8	16	32	64	128
RCA	16	32	64	128	256
CSKA-1L	12	16	24	32	48
CSKA-2L	12	16	20	24	32
CSLA-1L	10	12	18	24	34
CIA-1L	10	12	18	24	34
CIA-2L	10	12	16	18	22
CIA-3L	10	12	16	18	20
CLA	12	16	20	24	28
PPA-SK	10	12	14	16	18
PPA-BK	12	16	20	24	28
PPA-KS	10	12	14	16	18
COSA	**8**	**10**	**12**	**14**	**16**

Table 4.6: *Gate-count×gate-delay product (normalized).*

adder	word length [bits]				
type	8	16	32	64	128
RCA	1.17	1.92	2.86	5.04	8.21
CSKA-1L	1.15	1.23	1.36	1.55	1.87
CSKA-2L	1.08	1.33	1.28	1.33	1.43
CSLA-1L	1.10	1.23	1.43	1.76	2.07
CIA-1L	0.99	**0.99**	1.12	1.32	1.54
CIA-2L	1.00	1.00	**1.00**	**1.00**	1.00
CIA-3L	1.01	1.01	1.03	1.01	**0.91**
CLA	1.40	1.72	1.69	1.84	1.77
PPA-SK	**0.92**	1.04	1.03	1.17	1.20
PPA-BK	1.06	1.24	1.20	1.30	1.26
PPA-KS	1.11	1.37	1.44	1.71	1.82
COSA	1.16	1.52	1.63	1.94	2.04

1000 random input patterns [Naj94, CJ93]. All examinations were done using the Passport 0.6μm 3V three-metal CMOS high-density standard-cell library and the design tools by COMPASS Design Automation.

The cell primitives used by the circuit optimizer include multi-input AND-/NAND-/OR-/NOR-gates, various AOI-/OAI-gates, two-input XOR-/XNOR-gates and two-input multiplexers. Since the usage of full-adder cells for the ripple-carry and carry-select adders showed better area, but worse speed and AT-/PT-product performances, no such cells were used for the comparisons.

Tables 4.7–4.9 list the area, delay, and relative area-delay (AT) product measures for the standard-cell implementations after placement and routing. Area is given in $1000 \times \lambda^2$ ($1\lambda = 0.3\mu$m). The corresponding average power dissipation and relative power-delay (PT) product numbers are given in Tables 4.10 and 4.11 with the proposed CIA-2L acting as reference. Note that the delays are given for typical-case PTV conditions (typical process, 25^o C, 3.3 V). Worst-case conditions are assumed for power estimation (fast transistors, 0^o C, 3.6 V).

Figures 4.7 and 4.8 give a graphical representation of the comparison results for the standard-cell implementations. Area vs delay resp. power vs delay measures are drawn on a logarithmic scale visualizing the area-delay and power-delay trade-offs for some important adder architectures.

Table 4.7: *Post-layout area* ($1000 \times \lambda^2$).

adder type	word length [bits] 8	16	32	64	128
RCA	**238**	**457**	**821**	**1734**	**3798**
CSKA-1L	298	518	885	1932	4468
CSKA-2L	297	512	924	2196	4402
CSLA-1L	339	612	1322	2965	6381
CIA-1L	299	584	1119	2477	5189
CIA-2L	289	574	1094	2426	5353
CLA	324	649	1267	2816	6543
PPA-SK	266	580	1276	2979	7918
PPA-BK	270	549	1051	2316	5170
PPA-KS	408	1027	2292	5080	13616
COSA	419	924	1789	4399	10614

Table 4.8: *Post-layout delay* (ns).

adder type	word length [bits] 8	16	32	64	128
RCA	4.6	8.2	15.8	30.4	61.8
CSKA-1L	4.2	5.7	9.0	11.9	15.9
CSKA-2L	4.2	5.7	8.1	10.2	13.3
CSLA-1L	**3.3**	4.8	6.1	8.6	12.8
CIA-1L	3.6	4.7	6.1	8.0	11.2
CIA-2L	3.8	4.7	5.7	6.8	8.5
CLA	3.9	4.7	5.8	6.7	8.2
PPA-SK	3.5	**4.2**	5.2	**6.0**	**8.1**
PPA-BK	4.1	5.4	6.2	7.8	9.3
PPA-KS	3.4	**4.2**	5.3	6.9	9.3
COSA	3.4	4.5	**5.1**	6.4	9.2

Table 4.9: *Post-layout AT-product (normalized).*

adder type	word length [bits] 8	16	32	64	128
RCA	0.99	1.39	2.09	3.21	5.15
CSKA-1L	1.14	1.10	1.28	1.40	1.56
CSKA-2L	1.14	1.08	1.21	1.36	1.28
CSLA-1L	1.03	1.08	1.30	1.55	1.79
CIA-1L	0.97	1.01	1.10	1.20	1.28
CIA-2L	1.00	1.00	**1.00**	**1.00**	**1.00**
CLA	1.14	1.14	1.19	1.14	1.17
PPA-SK	**0.85**	**0.90**	1.07	1.09	1.40
PPA-BK	1.00	1.09	1.04	1.09	1.05
PPA-KS	1.28	1.59	1.94	2.14	2.79
COSA	1.28	1.52	1.48	1.71	2.14

Table 4.10: *Post-layout power (μW/MHz).*

adder type	word length [bits] 8	16	32	64	128
RCA	**24**	52	95	**194**	**387**
CSKA-1L	29	**48**	**90**	195	402
CSKA-2L	29	50	99	210	**387**
CSLA-1L	36	70	163	395	818
CIA-1L	32	64	116	257	494
CIA-2L	28	60	124	267	558
CLA	34	66	138	294	640
PPA-SK	27	60	134	305	704
PPA-BK	29	60	117	237	498
PPA-KS	40	102	232	498	1246
COSA	41	101	208	521	1276

Table 4.11: *Post-layout PT-product (normalized).*

adder type	word length [bits] 8	16	32	64	128
RCA	1.02	1.52	2.13	3.26	5.04
CSKA-1L	1.14	0.98	1.14	1.28	1.35
CSKA-2L	1.14	1.00	1.14	1.18	1.08
CSLA-1L	1.12	1.19	1.42	1.88	2.20
CIA-1L	1.06	1.06	**1.00**	1.13	1.17
CIA-2L	1.00	1.00	**1.00**	**1.00**	1.00
CLA	1.21	1.11	1.15	1.08	1.10
PPA-SK	**0.87**	**0.88**	**1.00**	1.02	1.19
PPA-BK	1.11	1.14	1.02	1.01	**0.97**
PPA-KS	1.29	1.52	1.73	1.91	2.45
COSA	1.28	1.59	1.52	1.84	2.47

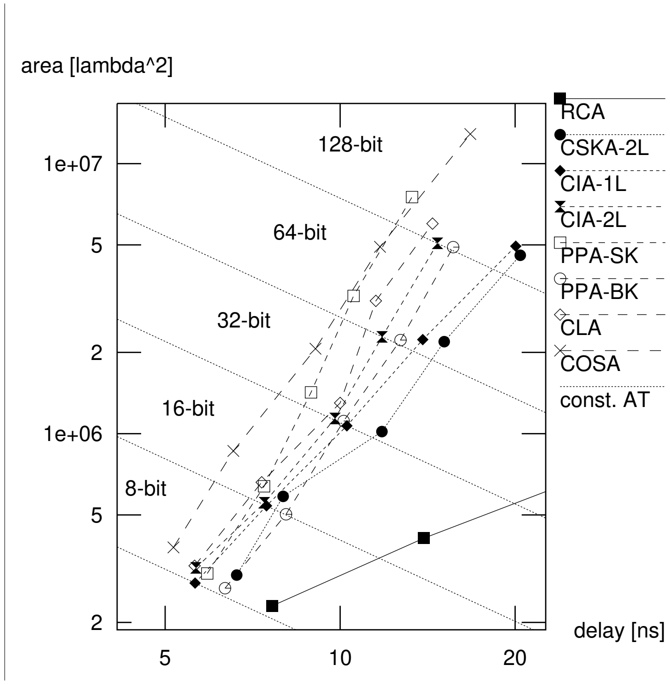


Figure 4.7: *Area vs delay (logarithmic scale).*

4.2.4 Results and Discussion

Unit-gate model

The results from the unit-gate model comparisons are of minor importance due to the inaccuracy of the model. However, the results are quite interesting and still allow the observation of some general tendencies.

From the circuit area point of view, the ripple-carry adder (RCA) and the carry-skip adders (CSKA) are the most efficient ones, followed by the carry-increment adders (CIA) which require only little additional logic. Note that the multilevel carry-increment adders have a negligible area increase compared to the one-level version. The Brent-Kung parallel-prefix adder (PPA-BK) shows

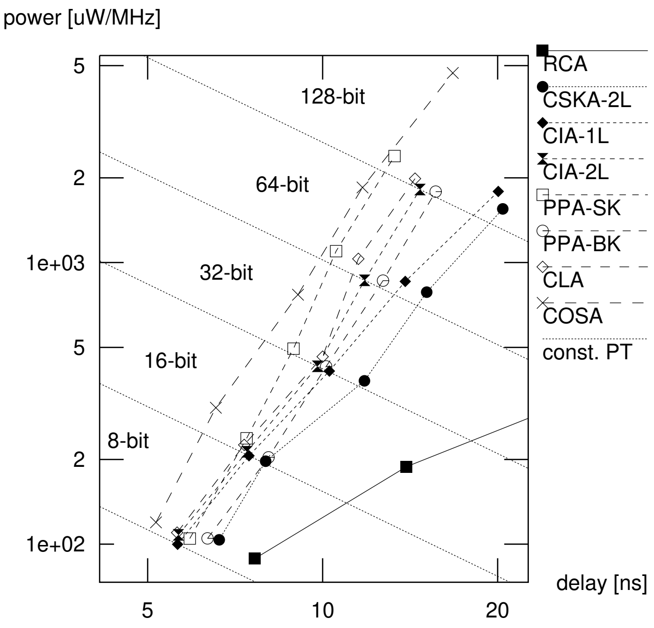


Figure 4.8: *Power vs delay (logarithmic scale).*

roughly the same area complexity as the carry-increment adders, while all other architectures from the $\log n$ time-complexity class have considerably higher area requirements. In particular, the Kogge-Stone parallel-prefix adder (PPA-KS) and the conditional-sum adder (COSA) result in very large logic networks.

The opposite holds true if circuit delay is considered. The conditional-sum adder (COSA) is the fastest one for every word length. It is faster by at least two gate delays than all other adders with $\log n$ time complexity because it works without the final sum-bit generation level built from XORs. The parallel-prefix adders PPA-SK and PPA-KS are the second fastest circuits, while the multilevel carry-increment adders come very close to their speed. All remaining architectures are considerably slower, with the ripple-carry adder being far the slowest one.

The gate-count×gate-delay product (or area-delay product) gives a good measure for the area *and* time efficiency of logic networks. Here, the good area and delay characteristics of the proposed carry-increment adders result in the lowest AT-product values of all adder architectures and for all word lengths larger than 8 bits.

Standard-cell implementation

More reliable results are obtained from the standard-cell implementation comparisons. These are now discussed in detail.

Compared to the ripple-carry adder (RCA) which has the smallest area and longest delay, the carry-skip adders (CSKA) are much faster with a moderate area increase. The carry-increment adders (CIA) achieve further speed improvement at small additional area costs. 2-level implementations of both carry-skip and carry-increment adders (CSKA-2L, CIA-2L) are considerably faster but only slightly larger than their 1-level counterparts (CSKA-1L, CIA-1L). Because the carry-increment adder is an optimization of the carry-select adder (CSLA), it outperforms the latter in all respects.

The various parallel-prefix addition schemes open a wide range of solutions with different area and time performances. The unbounded fan-out parallel-prefix scheme (PPA-SK) represents the fastest adder architecture for large word lengths at the price of quite large area requirements. The bounded fan-out structure by Brent and Kung (PPA-BK) is more area efficient but has a longer computation time. The carry-lookahead adder (CLA) being a 4-bit block version of PPA-BK is considerably faster but also larger than the latter. Note that the 8-, 32-, and 128-bit versions of CLA have better area performance because their first lookahead level consist of 2-bit blocks (as in PPA-BK). Finally, the conditional-sum adder (COSA) as well as the bounded fan-out parallel-prefix adder by Kogge and Stone (PPA-KS) are very fast for small and medium word lengths but suffer from very large circuit sizes and routing overheads with respect to speed for high word lengths. Their very high area costs also result in bad area-delay product values. The advantage of bounded fan-out of PPA-KS is partly undone by the large capacitive load of long wires, which degrades circuit speed and overall performance significantly.

Regarding the area-delay product, the two carry-increment and the PPA-BK architectures perform best for all adder sizes with the proposed CIA-2L presenting the lowest AT-product for large word lengths. The least area-time efficient structures are RCA, PPA-KS, COSA, and CSLA.

A high correlation can be observed between area and power requirements, which is clearly documented by the similarity of the two graphs in Figures 4.7 and 4.8. This fact is not surprising, however, because dynamic power dissipation mainly originates from charging node capacitances and thus is linearly dependent on the number of (toggling) circuit nodes and on wire lengths. Therefore, all area-efficient structures like RCA, CSKA, CIA, and PPA-BK are also the most power-efficient ones. Regarding the power-delay product, the carry-increment adders perform very well for all adder sizes with the 2-level version (CIA-2L) having the lowest PT-product for large word lengths. On the other hand, RCA, CSLA, PPA-KS, and COSA show poor power-delay performance. Note that structures with heavily loaded nodes (PPA-KS, COSA) present a significantly higher power/area ratio.

The power dissipated in glitching transitions (i.e., transitions that are followed by an inverse transition before settling to a steady state) is of special interest and was investigated, too. Generally, sources of glitches are gates with an output transition caused by a first input change which is undone by a second, delayed transition on a different input. This potentially occurs in every circuit with unequal signal path delays and multiple, uncorrelated input signals. Sources of glitches within adder circuits are the XOR/multiplexer gates used for sum bit generation and the carry propagation circuits, which are subject to race conditions under certain stimulations. The average glitching power contributions range from 10% for 8-bit up to 20% for 128-bit adders, whereas the amount of glitching transitions at the primary outputs can be as high as 50% of all transitions in the worst case. Partly significant variations between different adder architectures can be observed regarding glitching power. Furthermore, the potential for power savings by suppressing glitching transitions (e.g., balancing of path delays by buffer insertion to avoid race conditions) is very limited due to the large signal delay differences and the introduced additional buffer node activity.

The comparison results obtained allow the conclusion that RCA and CSKA are the choices for small area and moderate speed requirements, whereas CIA-2L and PPA-SK are the best performing candidates for high-speed demands. It is interesting to observe that the area and speed numbers of CIA-2L lie in-between the two parallel-prefix algorithms PPA-BK and PPA-SK. This is not surprising because it corresponds to a parallel-prefix adder with a prefix structure similar to the ones of PPA-BK and PPA-SK. Thus, the proposed 2-level carry-increment scheme proves to be a high-performing adder architecture which strongly conserves low area and short delay properties also for large adder sizes and under consideration of actual wiring contributions.

In addition to the investigations described so far, the same adder circuits were also optimized and compared using a 0.8μm standard-cell library from VLSI Technology Inc. and the tools by Compass as well as a 0.5μm standard-cell library and tools by Synopsys Inc. which, however, allowed only predictive capacitance information. The results highly match the ones given above and show again the best performance figures for the CIA-2L, PPA-BK, and PPA-SK architectures. Performance degradations of high-area architectures like PPA-KS and COSA tend to even higher values for large word lengths.

4.2.5 More General Observations

It can be observed that the unit-gate delay and unit-gate count measures are quite inaccurate when compared to the numbers obtained from actual layout realizations, especially for the area intensive adder architectures. This is because the unit-gate model used disregards basic aspects such as fan-out and wiring contributions. The fan-in model mentioned in Section 2.5 has not shown better results. In order to get more reliable pre-route information on circuit complexity and speed, the model has to be refined by incorporating fan-out and interconnection aspects. This becomes even more important for deep submicron technologies, where RC delays from wiring become dominant over gate delays. On the other hand, the unit-gate models are good enough for indicating some general tendencies and for allowing rough architecture classifications with respect to circuit area and delay complexity.

Another obvious observation is that area-intensive structures (like PPA-KS, COSA) suffer from considerable speed degradations caused by long wires and interconnection delays, whereas circuits with smaller area demands preserve their predicted performance during the layout phase much more. This fact is nicely documented by the 1-level carry-select and carry-increment adders. Having exactly the same blocking scheme and thus the same critical paths and gate-delay numbers, the area-intensive CSLA becomes slower for increasing adder sizes compared to CIA. In other words, efficient speed-up is not always achieved by using exhaustive parallelization and hardware duplication techniques. The conclusion is that architectures resulting in compact circuits will profit more in area, delay, and power respects when process feature sizes shrink.

Note that all adder architectures can also be classified regarding their fan-out properties. Bounded fan-out structures are found in RCA, CSKA, PPA-BK, and CLA, whereas CSLA, CIA, PPA-SK, PPA-KS, and COSA have unbounded fan-out. Unbounded fan-out circuits are usually faster due to a higher parallelism but also larger which, together with the higher fan-out loads, slows down computation again. Both classes of fan-out schemes contain area and time efficient adder structures.

The unit-gate model based examinations demonstrate that CIA-2L is slower than PPA-SK by only two gate delays with the exception of the 128-bit adder. It can be shown that this holds for all multilevel carry-increment adders except for the one with the maximum ($\log n$) number of levels, which actually is equivalent to PPA-SK. Thus, all carry-increment adders with an intermediate number of levels offer no speed advantage over the 2-level implementation but have higher area costs. Therefore, the two extremes with two (CIA-2L) and $\log n$ (PPA-SK) increment levels represent the best performing multilevel carry-increment adder schemes.

Further investigations on 4-bit block versions of different parallel-prefix adder architectures have not shown any advantages over their 2-bit block counterparts, whereas solutions with block sizes of eight bits have turned out to become considerably larger and slower.

As already mentioned before, hybrid adder architectures have not shown performance advantages neither under the unit-gate model nor in standard-cell implementations.

Full-custom implementations and layout generators ask for adder architectures with highly regular circuits, like e.g. CSKA, CIA, and PPA. Because the layout size of fast parallel-prefix and conditional-sum adders ($O(n \log n)$) grows in both dimensions with the word length ($1st$ dimension: number of bits, $2nd$ dimension: number of levels), the 2-level carry-increment adder is the fastest adder structure with linear layout arrangement and area demands ($O(n)$).

AT- and PT-product minimization are, of course, not the only optimization criteria for adder circuits. However, AT- and PT-product measures help finding the most efficient solution from a set of possible circuit candidates.

The presented results of standard-cell adder realizations can by no means be applied to transistor-level design techniques, which open many more circuit alternatives and leave room for further AT- and PT-product optimizations: dedicated logic styles and circuit techniques – such as pass-gate/pass-transistor logic or dynamic logic — potentially increase circuit efficiency of multiplexer structures and linear carry-chains. As a result, custom conditional-sum or hybrid adder implementations, for instance, are documented to be highly competitive [D$^+$92, G$^+$94, M$^+$94, OV95, O$^+$95, M$^+$91].

4.2.6 Comparison Diagrams

The most interesting properties and comparison results are documented in the diagrams of Figures 4.9–4.14. The diagrams rely on results from the post-layout solutions, with exceptions mentioned in the diagram title. Numbers are normalized to one bit (i.e. divided by the word length) in order to allow comparisons between adders of different word lengths.

Figure 4.9 demonstrates the high congruence between post-layout circuit area and the area models based on gate counts, gate equivalents, and cell area. Despite of the simplicity of the gate count estimation model, its accuracy is quite good except for PPA-KS (where routing is underestimated) and COSA (area for MUX overestimated). The gate equivalents model gives only slightly more accurate results. Finally, cell area correlates very well with the final circuit area since the routing overhead is almost constant for different adder architectures and grows slightly with increasing adder sizes. The high area / cell area ratios (i.e., routing factors) of the 128-bit versions of some low-area adders (RCA, CSKA) are determined by the large number of connectors at the standard-cell block borders and not by internal routing congestion.

Figure 4.10 compares final circuit delay with unit-gate and cell delay. Again, the unit-gate model can be used for a rough but simple delay estimation, while the cell delay matches the final circuit delay quite well. Exceptions are again PPA-KS and COSA where the interconnect delays — originating from high wiring and circuit complexity — are underestimated.

Figures 4.11 and 4.12 give some comparisons related to power dissipation. The percentage of glitching power does vary considerably between different adder architectures and sizes. In particular, CSLA-1L shows far the highest amount of glitching power. Together with COSA, it has the highest power consumption / circuit area ratio. Otherwise, the power dissipation correlates quite well with the final circuit area. This substantiates the suggestion that area-efficient adder architectures are also power efficient. The wiring power to cell power ratio does not differ substantially between architectures and word lengths. Power estimation through toggle count — which can be determined before place-and-route — is quite accurate with the exception of PPA-KS, where again routing overhead is underestimated. On the other hand, power estimation from the total wiring capacitance does not give such accurate results. Not surprisingly, the product of average toggle count and wiring capacitance is a very good measure for wiring power and thus also for total power.

Figure 4.13 illustrates properties related to wiring complexity. Wiring capacitance, which highly correlates with total wire length and circuit area, is much higher for PPA-KS and COSA than for all other architectures. The maximum number of pins per net reflects nicely the fan-out properties of the circuits. Automatic circuit optimization attenuates these numbers to some degree. Constant or bounded fan-out architectures are RCA, CSKA, CLA, PPA-BK, and PPA-KS. Unbounded fan-out architectures are CIA with relatively low, CSLA with medium, and PPA-SK and COSA with very high maximum fan-out values.

Figure 4.14 finally contains some other numbers of interest. Number of cells and number of nodes correlate perfectly among each other as well as with circuit area. The average capacitance of a wire is quite constant and shows only larger values for the area-inefficient architectures PPA-KS and COSA. The average toggle count per node has interestingly high values for CSLA and relatively low values for PPA-KS.

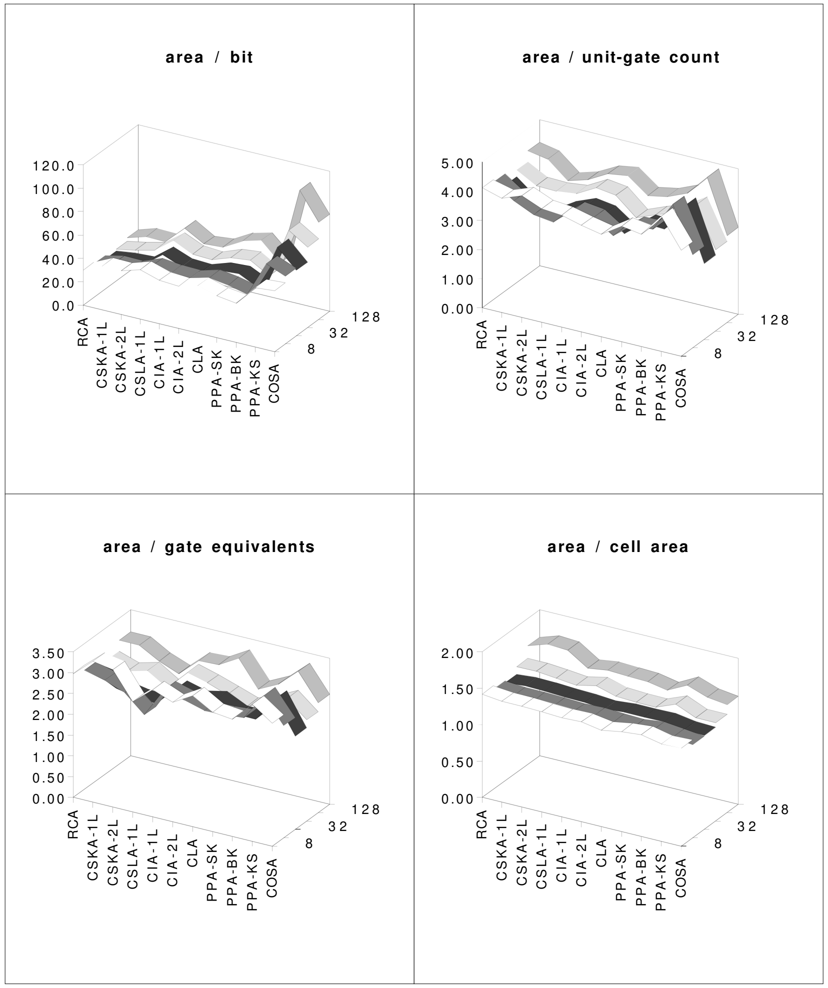


Figure 4.9: *Area-related comparisons.*

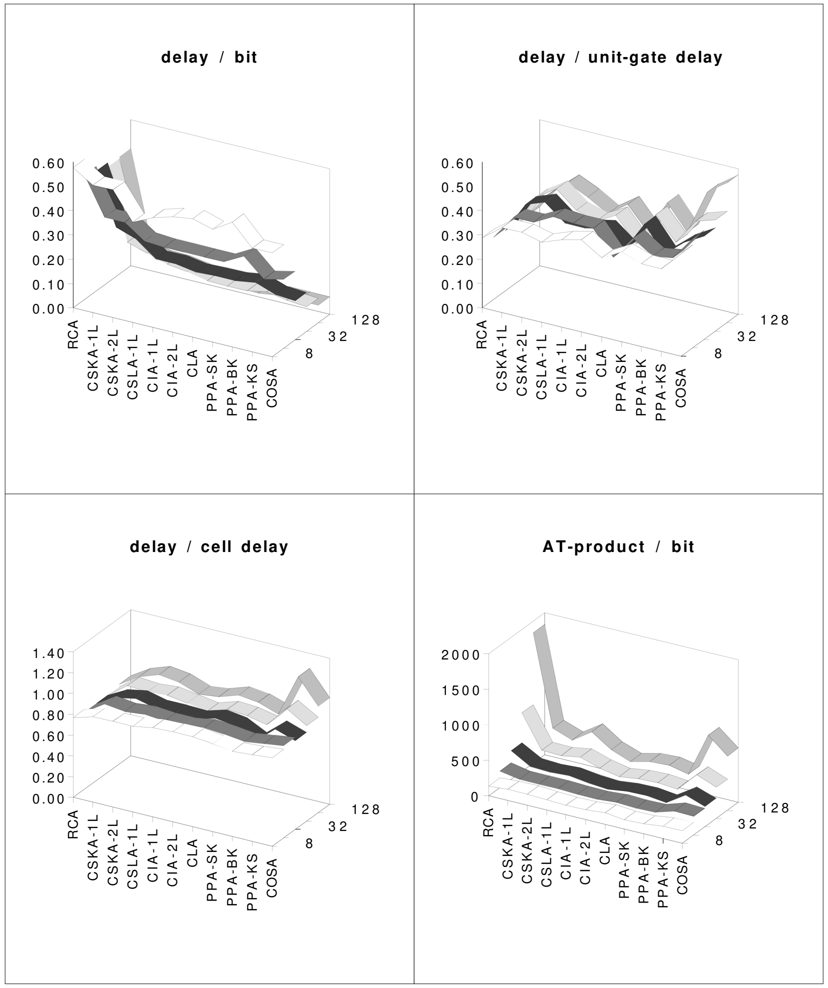


Figure 4.10: *Delay-related comparisons.*

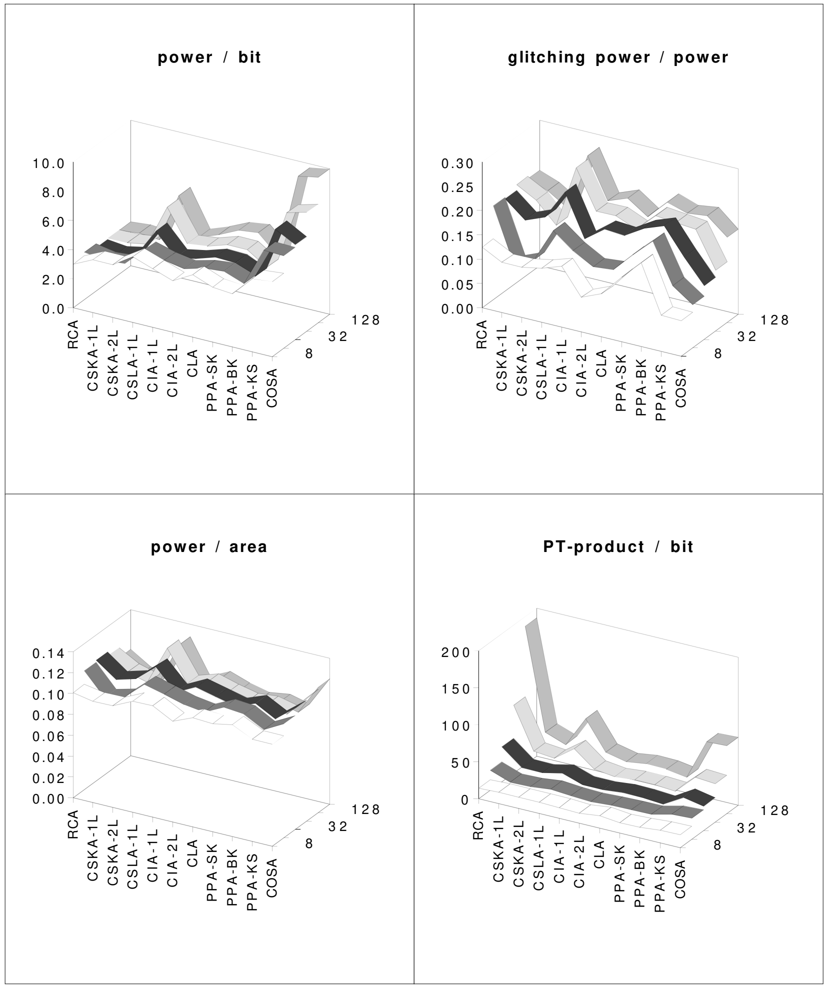


Figure 4.11: *Power-related comparisons.*

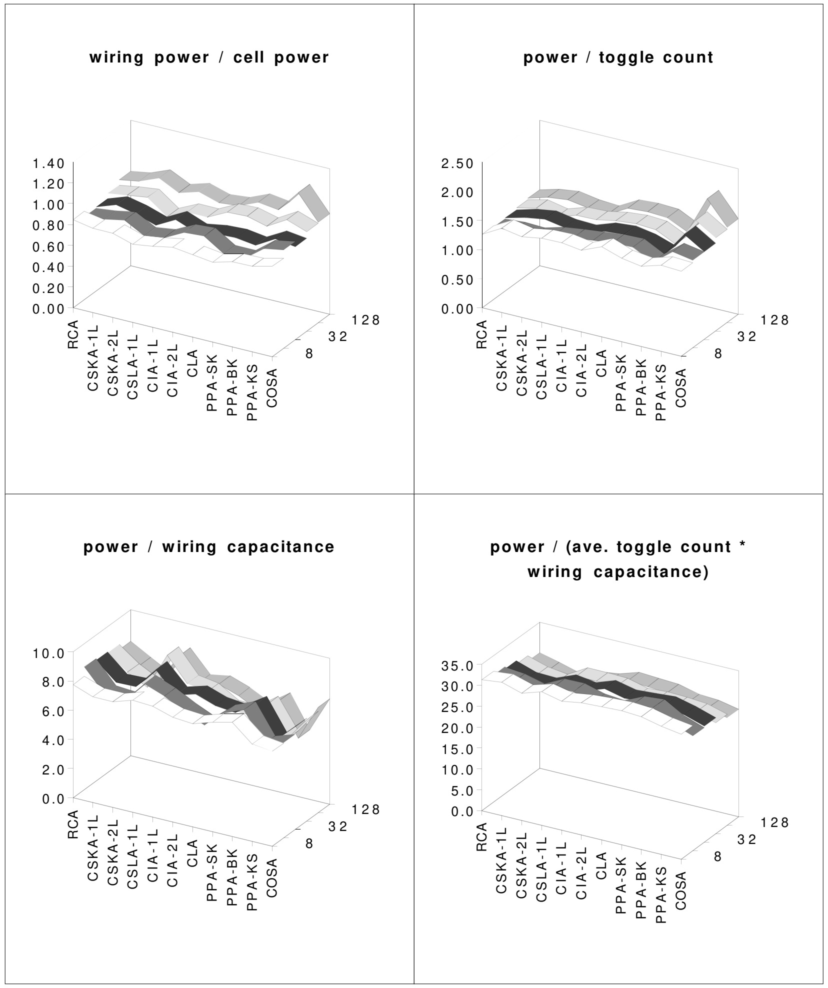


Figure 4.12: *Power-related comparisons (cont.).*

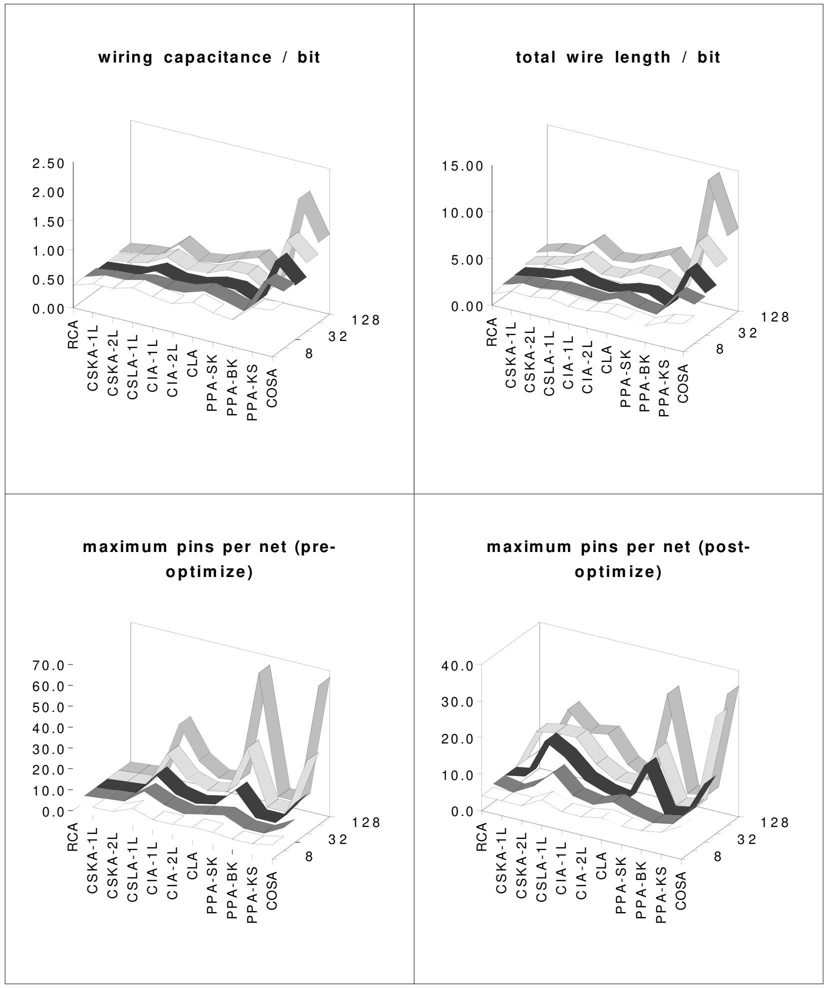


Figure 4.13: *Circuit-related comparisons.*

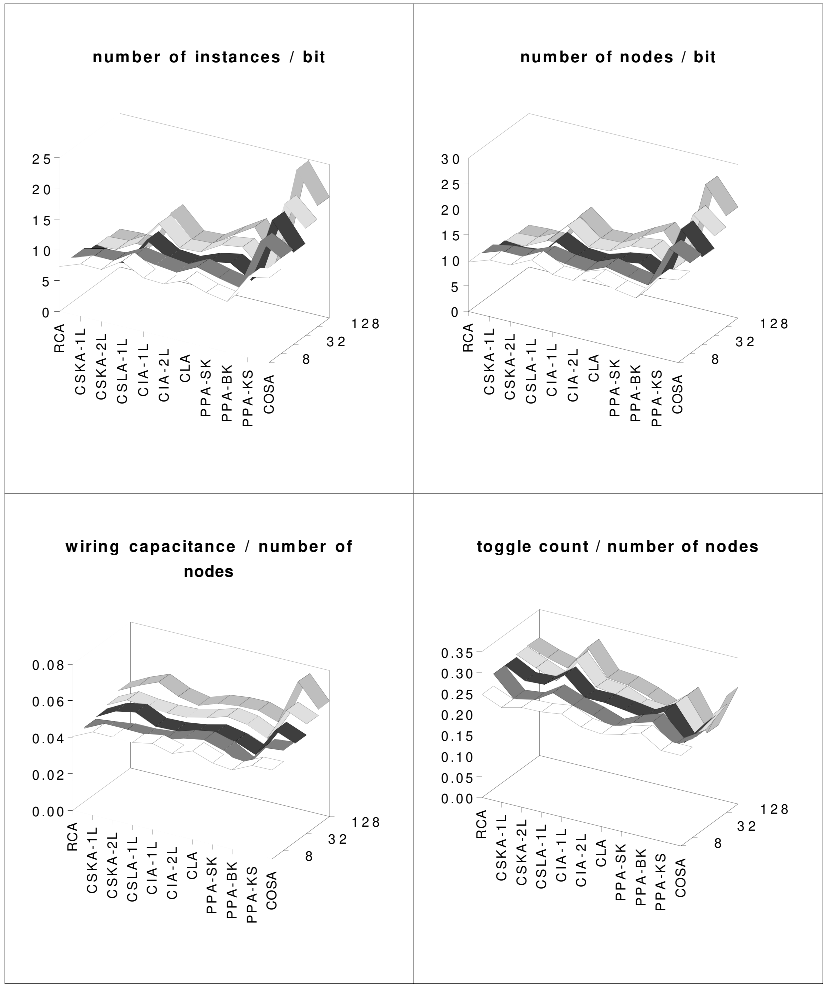


Figure 4.14: *Circuit-related comparisons (cont.).*

4.3 Summary: Optimal Adder Architectures

This section summarizes the results obtained from the adder architecture comparisons in Table 4.12 and gives some recommendations for adder design with focus on cell-based design techniques. The important design criteria for cell-based adders are circuit performance (area and delay), regularity (synthesis), and applicability of automated circuit optimization.

Table 4.12: *Optimality of adder architectures.*

adder architecture	performance [1]	regularity	autom. optim.	prefix scheme	requirements area	requirements speed
RCA	a tttttt	highest	√	√	lowest	lowest
CSKA-1L	aa ttttt	medium			low	low
CSKA-2L	aa ttttt	low	√		—	—
CSLA-1L	aaaa tttt	high	√		—	—
CIA-1L	aaa tttt	high	√	√	medium	medium
CIA-2L	aaa tt	high	√	√	medium	high
CLA	aaaa tt	medium	√	(√)[2]	—	—
PPA-SK	aaaa t	medium	√	√	high	highest
PPA-BK	aaa ttt	medium	√	√	medium	medium
PPA-KS	aaaaaa t	medium	√	√	—	—
COSA	aaaaa t	low	√		—	—

[1] the number of a's/t's gives a qualitative measure for the area/delay requirements
[2] 4-bit prefix scheme

As can be seen, the ripple-carry, carry-increment, and parallel-prefix/carry-lookahead adders cover the entire range from lowest to highest performance with, however, some gaps in-between. They all belong to the family of prefix adders, which actually contains the smallest (RCA), the fastest (PPA-SK), and some efficient medium-performance (PPA-BK, CIA) adder architectures. The new 2-level carry-increment adder proposed in this thesis proves to be one of the best performing adder architectures for medium speed requirements.

5

Special Adders

As was demonstrated in the previous chapters, the parallel-prefix scheme for binary addition is very universal and the most efficient adder architectures are based on it. Furthermore, this scheme presents some additional properties which can be used for the implementation of special (or customized) adders and related units.

5.1 Adders with Flag Generation

The basic addition flags are the carry flag C, the 2's complement overflow flag V, the zero flag Z, and the negative flag N.

Carry flag C The carry flag corresponds to the carry-out signal of binary addition:

$$C = c_{out} = c_n \tag{5.1}$$

Most adder architectures provide the carry-out without any additional logic. In parallel-prefix adders all carries are computed before final sum bit generation. Thus, the carry-out is available two gate delays before the sum. In some parallel-prefix structures (e.g., Brent-Kung), c_n is even available some gate delays before most other carries.

2's complement overflow flag V Addition overflow of unsigned numbers is detected by the carry flag. Overflow of 2's complement signed numbers is detected by the overflow flag V using one of the following formulae:

$$\begin{aligned} V &= c_n \oplus c_{n-1} && (5.2) \\ &= a_n b_n \overline{s}_n + \overline{a}_n \overline{b}_n s_n && (5.3) \end{aligned}$$

Since parallel-prefix adders compute all carries, Equation (5.2) provides an efficient and fast overflow flag computation (i.e., one additional XOR, same delay as sum bits).

Zero flag Z The zero flag indicates whether an addition or subtraction result is zero or not. Obviously, the flag can be determined using the equation

$$Z = \overline{s_{n-1} + s_{n-2} + \cdots + s_0} \tag{5.4}$$

This solution, however, is slow because calculation has to wait for the final sum and uses an n-input NOR-gate. For faster solutions two cases are to be distinguished. If a subtraction is carried out (i.e., $c_{in} = 1$) the result is zero if both operands are equal ($A = B$). Since subtrahend B is in 2's complement form, the zero flag can be expressed as

$$Z = (a_{n-1} \oplus b_{n-1})(a_{n-2} \oplus b_{n-2}) \cdots (a_0 \oplus b_0) \tag{5.5}$$

which exactly corresponds to the propagate signal $P_{n-1:0}$ for the entire adder. Theoretically, this propagate signal is available in parallel-prefix adders (Eq. 3.28). The calculation, which has been omitted in the presented implementations because it is not used for normal addition, requires only ($\log n$) additional AND-gates. The critical path through an XOR and an AND tree makes the zero-flag calculation even faster than carry calculation.

In the second case, where addition is also allowed (i.e., $c_{in} = 0$), fast zero flag generation is more expensive. It is shown in the literature [CL92] that zero flag calculation is possible without carry-propagation. It bases on the following formulae:

$$\begin{aligned} z_0 &= ((a_0 \oplus b_0) \odot c_{in}) \\ z_i &= ((a_i \oplus b_i) \odot (a_{i-1} + b_{i-1})) \\ Z &= z_{n-1} z_{n-2} \cdots z_0 && (5.6) \end{aligned}$$

Here, only the XOR- and OR-gates can be used from the parallel-prefix adder logic. The remaining XNOR-gates ($\odot$) and the AND tree are to be realized separately.

Negative flag N The negative flag is used for 2's complement signed numbers and corresponds to the MSB of the sum:

$$N = s_{n-1} \tag{5.7}$$

5.2 Adders for Late Input Carry

As already described in Section 3.5 and depicted in Figure 3.28, two universal prefix adder structures exist with different carry-processing properties. In the first solution (Fig. 3.28a) the carry is fed into an additional prefix level resulting in fast input-carry propagation at the cost of additional logic. The resulting adder allows for a *late input carry* signal. The amount of delay reduction and hardware increase depends on the chosen parallel-prefix structure. The fastest input-carry processing is achieved by attaching one row of $\bullet$ operators to the end of the prefix stage containing an arbitrary prefix algorithm (Fig. 5.1). The overall delay of the adder is increased by two gate delays, while the delay from the carry-in to the outputs is constant ($T_{\text{CPA}}\ (c_{in} \rightarrow c_{out}) = 2$, $T_{\text{CPA}}\ (c_{in} \rightarrow s) = 4$). Note, however, that the fan-out of the carry-in grows linearly with the word length and thus adds some delay in real circuit implementations.

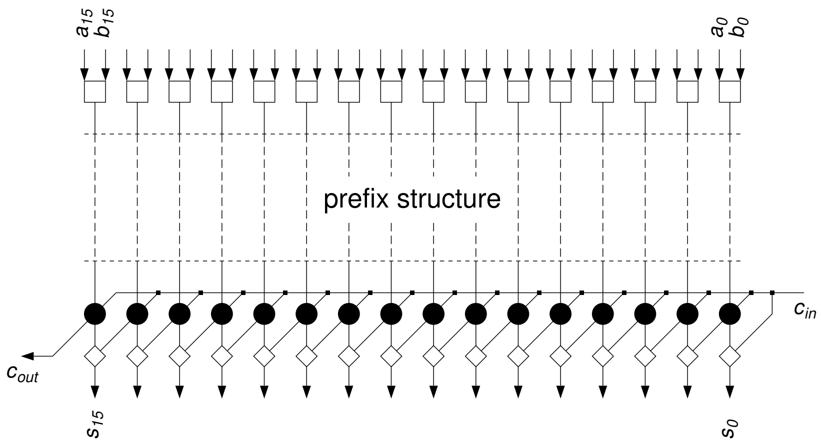
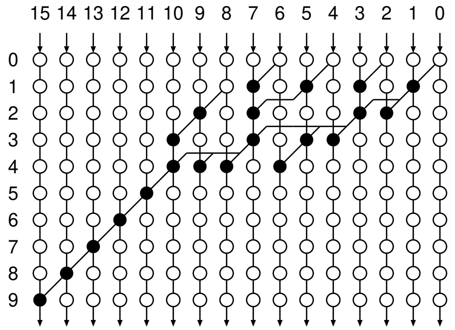


Figure 5.1: *Parallel-prefix structure with fast carry processing.*

5.3 Adders with Relaxed Timing Constraints

As we have seen so far, the serial-prefix (or ripple-carry) adder is the slowest but smallest one, while the parallel-prefix adders are faster but considerably larger. If the timing constraints lie somewhere between the delay of the serial-prefix and of a parallel-prefix adder, these two adder structures can be mixed: for the lower bits a parallel-prefix structure can be realized, while a serial-prefix structure is used for the upper bits (Fig. 5.2). The resulting circuit represents a compromise between the two extremes with respect to delay and area. Such mixed prefix structures are investigated in more detail in Chapter 6.



$$T_\bullet = n - 1 - k$$
$$A_\bullet = n - 1 + k$$
$$0 \le k \le n - 2\log n + 2$$

Figure 5.2: *Mixed serial/parallel-prefix algorithm.*

5.4 Adders with Non-Equal Bit Arrival Times

All adder architectures described so far expect all input bits to arrive simultaneously (i.e., equal bit *arrival times*) and deliver all output bits at the same moment of time (i.e., equal bit *required times*). Under this assumption, the fastest adder circuits are obtained by introducing as much parallelism as possible and thus equalizing all signal paths. Depending on the surrounding logic, however, individual input operand bits may arrive and output bits be required at different times, resulting in unequal signal path lengths through the adder. A fast adder circuit has to compensate for this by trading off different signal path delays. As will be demonstrated here, prefix structures are perfectly suited for matching arbitrary signal path profiles due to their generality and flexibility.

Prefix graphs for the most common signal arrival profiles are given in

Figures 5.3–5.6. The graphs are optimized by hand with respect to gate delays and, in second priority, gate counts. Fan-out as well as gate-delay/gate-count trade-offs, which may result in smaller AT-product values, were not considered.

In Figure 5.3a the input bits arrive in a staggered fashion from LSB to MSB, i.e., each bit arrives later than its right neighbor by one •-operator delay. A normal serial-prefix (or ripple-carry) adder perfectly fits this case. If bit arrival differences are smaller, a prefix structure similar to the one of Figure 3.22 can be used. In Figure 5.3b the entire higher half word arrives later. Here, a serial-prefix algorithm is used for the lower half word while calculation is sped up in the higher half word by a parallel-prefix structure.

The opposite bit arrival profiles are assumed in Figure 5.4. The solution for the staggered bit arrival times towards the LSB of Figure 5.4a is fast but very expensive. Allowing only one more •-operator latency, a much more economic structure is obtained (Fig. 5.4b), which most likely results in a faster circuit due to its smaller size and smaller fan-out numbers. Figure 5.5a depicts the case where all the lower half word bits are late. A fast parallel-prefix structure is used for the lower half word while a serial-prefix structure suffices for carry calculation in the upper half word.

In Figure 5.5b the input bits in the middle are assumed to arrive latest. This situation occurs typically in the final addition of multiplication, where a Wallace tree is used for summing up the partial products [Okl94, SO96]. The adder can be divided into three sections. In the first section higher bits arrive later than lower bits. Therefore a simple serial-prefix scheme can be used. The second section contains bit positions with roughly equal signal arrival times. A fast parallel-prefix structure is used here. In the third section higher bits arrive again earlier. Basically, the parallel-prefix structure from the middle section is extended into the upper section and optimized by taking advantage of the earlier MSBs. This structure optimization considerably decreases circuit area and delay compared to a structure optimized for equal bit arrival times.

Finally, Figure 5.6 shows the graphs for staggered output bit required times towards the LSB and the MSB, respectively. Fast processing of the high order bits basically requires a fast parallel-prefix structure which, however, can take almost no advantage of the slow LSBs for area optimization (Fig. 5.6a). On the other hand, fast processing of the low order bits is for free by using the serial-prefix scheme (Fig. 5.6b).

The given prefix graphs are just simple examples. Optimal prefix graphs have to be constructed individually from case to case, depending on the exact signal arrival profiles. The automatic generation of optimal prefix graphs under arbitrary timing constraints will be discussed in Chapter 6.

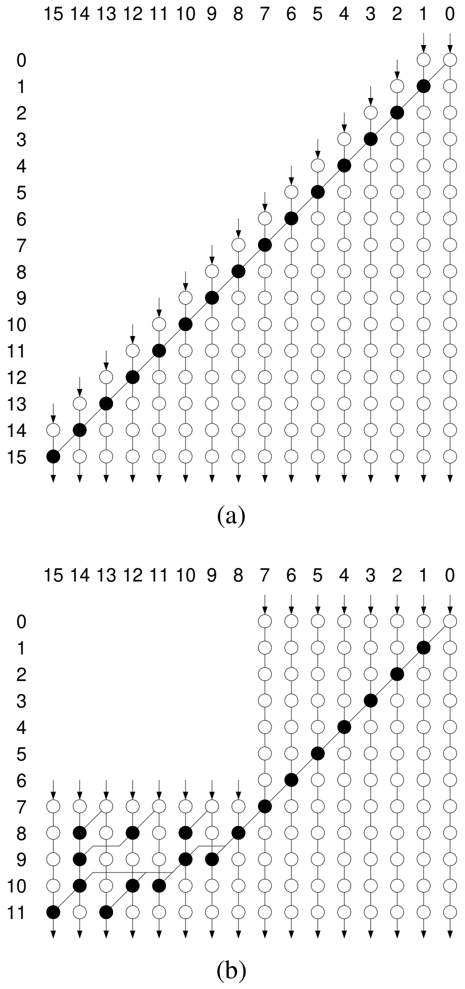


Figure 5.3: *Prefix graphs for adders with late input MSB arrival times.*

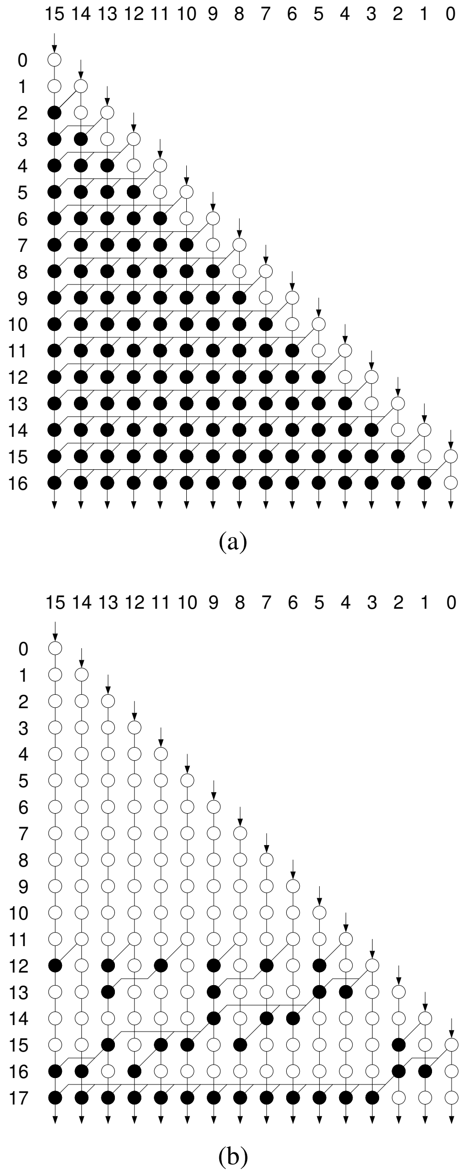


Figure 5.4: *Prefix graphs for adders with late input LSB arrival times.*

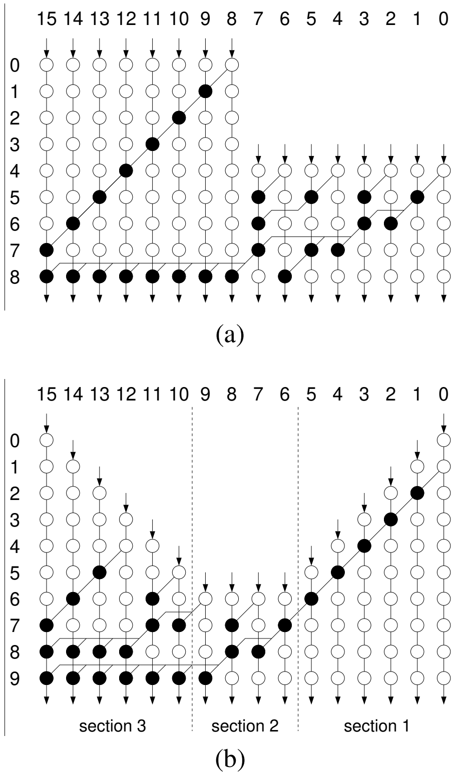


Figure 5.5: *Prefix graphs for adders with (a) late input LSB and (b) late intermediate input bit arrival times.*

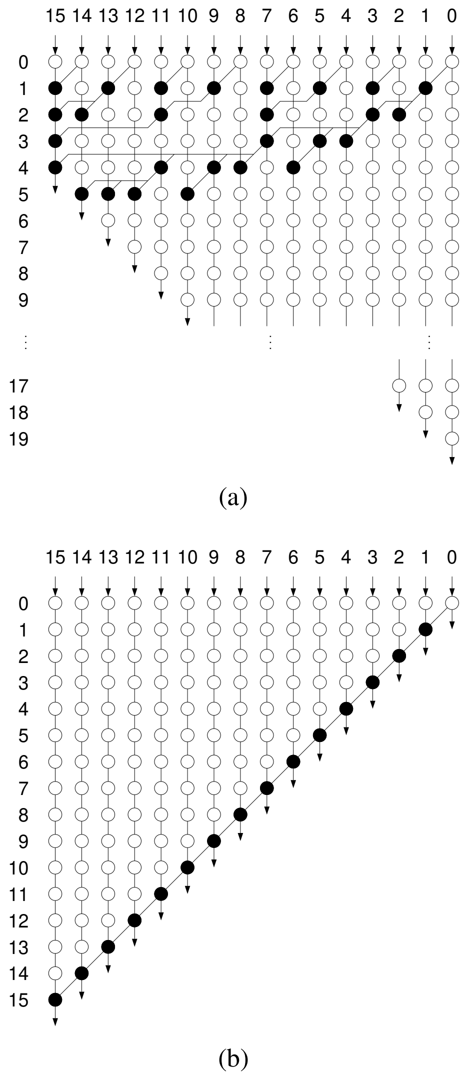


Figure 5.6: *Prefix graphs for adders with early output (a) MSB and (b) LSB required times.*

5.5 Modulo Adders

In *end-around carry adders* the carry-in depends on the carry-out, i.e., the carry-out c_{out} is fed through some logic back to the carry-in c_{in} (Fig. 5.7). In particular, this is used for addition modulo $(2^n + 1)$ and $(2^n - 1)$ (or 1's complement addition). Such *modulo adders* are used in residue number systems (RNS) [Kor93], cryptography [ZCB+94, Cur93], and error detection and correction codes [Kor93]. Because the carry-in signal is used for the modulo addition itself, no additional carry input is provided in such adders. The basic algorithm for modulo $(2^n + 1)$ and $(2^n - 1)$ addition rely on decrementation resp. incrementation of the addition result depending on the carry-out. Since prefix algorithms actually rely on incrementer structures, considering parallel-prefix schemes for this kind of adders is very promising.

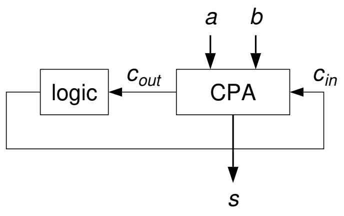


Figure 5.7: *General adder structure with end-around carry.*

Basically, the carry-out of an end-around carry adder is functionally independent of the carry-in. Thus, it is possible to build an end-around carry adder with no signal path from c_{in} to c_{out}. However, if the end-around carry technique is applied to a normal adder containing a signal path from c_{in} to c_{out}, a combinational loop is created which may lead to oscillations in some special cases. This can be avoided by inserting appropriate logic into the carry-feedback path or by breaking up the c_{in} to c_{out} signal path within the adder, which can be achieved by different approaches.

In order to obtain fast end-around carry adders both conditions of fast carry-out generation and fast carry-in processing have to be met. This implies a third condition which is that no combinational path exists between c_{in} and c_{out}. The parallel-prefix structure with fast carry processing introduced in Section 3.5 fulfills all these requirements. A fast end-around carry adder can be built using the prefix structure depicted in Figure 5.8. Here, the last prefix

stage is used as an incrementer which is controlled by the carry-out of the previous prefix stages.

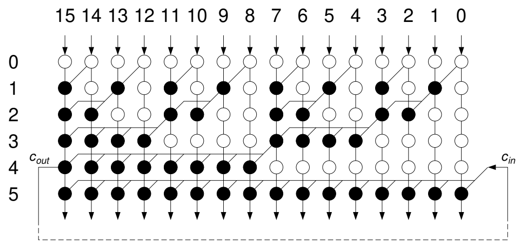


Figure 5.8: *Prefix graph with fast end-around carry.*

5.5.1 Addition Modulo $(2^n - 1)$

Addition *modulo* $(2^n - 1)$ or *one's complement* addition can be formulated by the following equation:

$$A + B \pmod{2^n - 1} = \begin{cases} A + B - (2^n - 1) = A + B + 1 \pmod{2^n} \\ \qquad\qquad \text{if} \quad A + B \geq 2^n - 1 \\ A + B \quad \text{otherwise} \end{cases} \tag{5.8}$$

However, the condition $A + B \geq 2^n - 1$ is not trivial to compute. Equation 5.8 can be rewritten using the condition $A + B \geq 2^n$ which is equivalent to $c_{out} = 1$:

$$A + B \pmod{2^n - 1} = \begin{cases} A + B - (2^n - 1) = A + B + 1 \pmod{2^n} \\ \qquad\qquad \text{if} \quad A + B \geq 2^n \\ A + B \quad \text{otherwise} \end{cases} \tag{5.9}$$

Now, the carry-out c_{out} from the addition $(A + B)$ can be used to determine whether incrementation has to be performed or, even simpler, c_{out} can be added to the sum of $(A + B)$. This equation, however, results in a double-representation of zero (i.e., $0 = 00 \cdots 0 = 11 \cdots 1$). The prefix adder structure is given in Figure 5.9.

If a single-representation of zero is required, equation 5.8 has to be realized. The condition $A + B \geq 2^n - 1$ is fulfilled if either $A + B \geq 2^n$ or $A + B = 2^n -$

$1 = 11 \cdots 1$ which corresponds to the propagate signal $P_{n-1:0}$ of a parallel-prefix adder. Thus, an adder modulo $(2^n - 1)$ with single-representation of zero can also easily be implemented using a parallel-prefix structure (Fig. 5.10).

Another approach for fast modulo $(2^n - 1)$ addition bases on modification of the traditional carry-lookahead adder scheme [ENK94]. There, the logic formula for the carry-out c_{out} is re-substituted as carry-in c_{in} in the logic equations for the sum bits. As a consequence, each sum bit does not only depend on input bits of equal or lower binary weight but is a function of all input bits. Thus, the coding logic per bit position is doubled on the average, which results in a considerable hardware overhead.

5.5.2 Addition Modulo $(2^n + 1)$

Addition *modulo* $(2^n + 1)$ is of more specialized interest. One application is its use in the modulo $(2^n + 1)$ multiplier of the IDEA cryptography algorithm [LM90, ZCB+94]. Here, the *diminished-one* number system is used where a number A is represented by $A' = A - 1$ and the value 0 is not used. Normal addition in this number system looks as follows:

$$\begin{aligned} A + B &= S \\ (A' + 1) + (B' + 1) &= (S' + 1) \\ A' + B' + 1 &= S' \end{aligned} \tag{5.10}$$

Modulo $(2^n + 1)$ addition can now be formulated as

$$A' + B' + 1 \pmod{2^n + 1} = \begin{cases} A' + B' + 1 - (2^n + 1) & \\ \qquad = A' + B' \pmod{2^n} & \text{if } A' + B' + 1 \geq 2^n \\ A' + B' + 1 & \text{otherwise} \end{cases} \tag{5.11}$$

Thus, the sum $(A' + B')$ is incremented if $A' + B' + 1 < 2^n$ i.e., $c_{out} = 0$. This results in the same parallel-prefix adder structure as for modulo $(2^n - 1)$ addition except for the inverter in the carry feedback path (Fig. 5.11).

As was demonstrated, highly efficient end-around carry adders can be implemented using the parallel-prefix adder structure with fast carry-processing. Since the algorithms for modulo $(2^n - 1)$ and modulo $(2^n + 1)$ addition base on conditional final incrementation, they fit perfectly into the carry-increment and the (more general) prefix adder schemes.

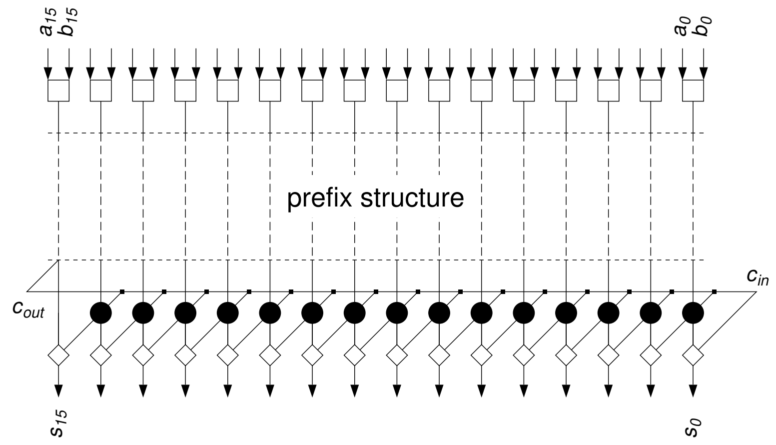


Figure 5.9: *Parallel-prefix adder modulo* $(2^n - 1)$ *with double-representation of zero.*

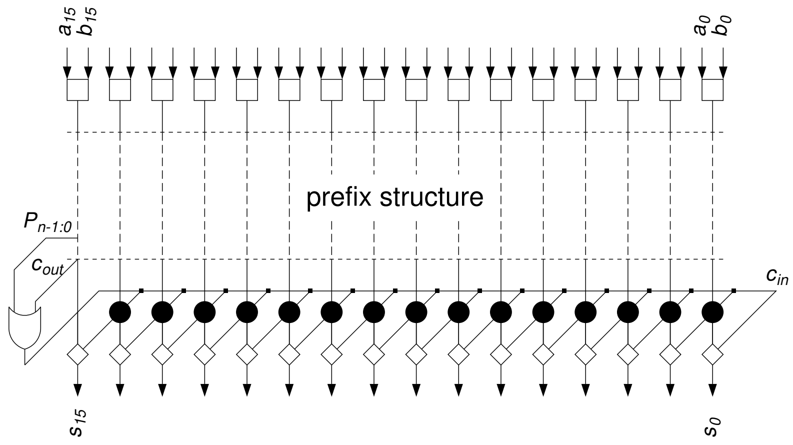


Figure 5.10: *Parallel-prefix adder modulo* $(2^n - 1)$ *with single-representation of zero.*

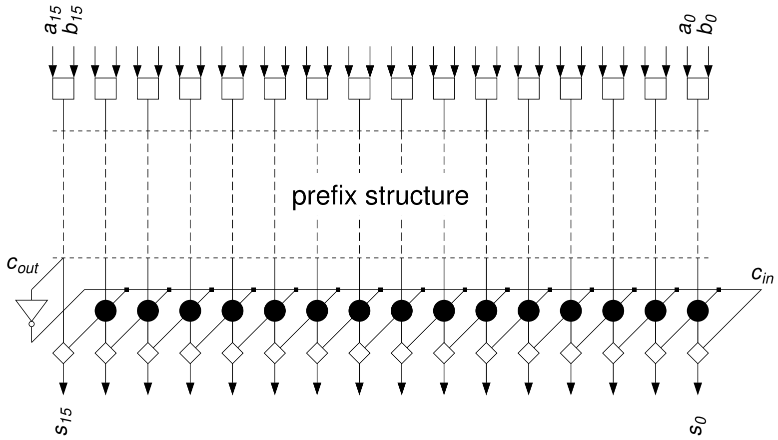


Figure 5.11: *Parallel-prefix adder modulo* $(2^n + 1)$ *using the diminished-one number system.*

5.6 Dual-Size Adders

In some applications an adder must perform additions for different word lengths depending on the operation mode (e.g. multi-media instructions in modern processors). In the simpler case an n-bit adder is used for one k-bit addition ($k \leq n$) at a time. A correct k-bit addition is performed by connecting the operands to the lower k bits ($a_{k-1:0}$, $b_{k-1:0}$, $s_{k-1:0}$) and the carry input to the carry-in (c_{in}) of the n-bit adder. The carry output can be obtained in two different ways:

1. Two constant operands yielding the sum $s_{n-1:k} = 11 \cdots 1$ are applied to the upper $n-k$ bits (e.g., $a_{n-1:k} = 00 \cdots 0$, $b_{n-1:k} = 11 \cdots 1$). A carry at position k will propagate through the $n-k$ upper bits and appear at the adder's carry-out c_{out}. This technique works with any adder architecture.

2. If an adder architecture is used which generates the carries for all bit positions (e.g., parallel-prefix adders), the appropriate carry-out of a k-bit addition (c_k) can be obtained directly.

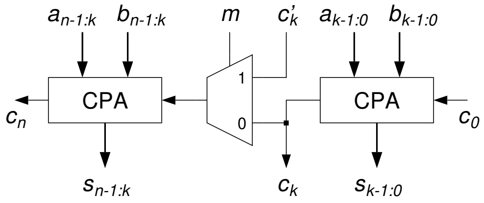


Figure 5.12: *Dual-size adder composed of two CPAs.*

In a more complex case an n-bit adder may be used for an n-bit addition in one mode and for two smaller additions (e.g., a k-bit and a $n-k$-bit addition) in the other mode. In other words, the adder needs selectively be partitioned into two independent adders of smaller size (*partitioned* or *dual-size adder*). For partitioning, the adder is cut into two parts between bits $k-1$ and k. The carry c_k corresponds to the carry-out of the lower adder, while a multiplexer is used to switch from c_k to a second carry-in c'_k for the upper adder.

Figure 5.12 depicts a dual-size adder composed of two CPAs. The logic equations are:

$$m = 0 \quad : \quad (c_n, s_{n-1:0}) = a_{n-1:0} + b_{n-1:0} + c_0 \tag{5.12}$$

$$m = 1 \quad : \quad \begin{array}{l} (c_n, s_{n-1:k}) = a_{n-1:k} + b_{n-1:k} + c'_k, \\ (c_k, s_{k-1:0}) = a_{k-1:0} + b_{k-1:0} + c_0 \end{array} \tag{5.13}$$

In order to achieve fast addition in the full-length addition mode ($m = 1$), two fast CPAs need to be chosen. Additionally, the upper adder has to provide fast input carry processing for fast addition in the single-addition mode ($m = 0$). However, depending on the adder sizes, this approach may result in only suboptimal solutions.

Again, the flexibility and simplicity of the parallel-prefix addition technique can be used to implement optimal dual-size adders: a normal n-bit parallel-prefix adder is cut into two parts at bit k. This approach allows the optimization of the n-bit addition, which typically is the critical operation. Because the prefix graph is subdivided at an arbitrary position, there may be several intermediate generate and propagate signal pairs crossing the cutting line (i.e., all $(G_{i:j}, P_{i:j})$ with $i < k$ that are used at bit positions $\geq k$). For correct operation in the full-length addition mode, the following aspects are to be considered:

1. The carry signal c_k is taken as carry-out of the lower adder.
2. No carries from the lower prefix graph partition must propagate into the upper one. This is achieved by treating the generate and propagate signals at the partition boundary appropriately.
3. The carry-in c'_k of the upper adder must be fed into the upper prefix graph partition at the appropriate location(s) so that it propagates to all bit positions $\geq k$.

Points 2 and 3 require additional logic which may be located on critical signal paths. Therefore, the goal is to reduce the number of inserted logic gates to a minimum in order to minimize area and delay overhead. Different solutions exist:

1. All generate signals $G_{i:j}$ crossing the cutting line are exchanged by the upper carry-in c'_k using multiplexers. The propagate signals $P_{i:j}$ crossing the cutting line can be left unchanged. Note that insertion of the same carry-in c'_k at different intermediate carry locations is allowed since the final carry of each bit position is the OR-concatenation of all intermediate carries. In this algorithm a multiplexer is inserted into each generate signal path which crosses the cutting line. The maximum number of multiplexers to be inserted grows with $O(\log n)$ for the Brent-Kung and the Sklansky prefix algorithms.
2. Only the generate signals originating from bit position $k-1$ ($G_{k-1:j}$) are exchanged by the upper carry-in c'_k. This satisfies point 3 because a prefix graph propagates the carry generated at position $k-1$ to any higher bit position only through the generate signals $G_{k-1:j}$. Additionally, the corresponding propagate signals $P_{k-1:j}$ must be forced to zero using an AND-gate. This prevents the remaining generate signals ($G_{i:j}$, $i < k-1$) from propagating into the upper prefix graph partition (point 2). This solution requires a multiplexer and an AND-gate for each generate/propagate signal pair originating from bit position $k-1$ but leaves all other signal pairs unchanged. In most cases no gates are inserted into the most critical path (i.e., the one with the highest fan-out numbers) which results in dual-size adders with negligible delay penalty. In the Sklansky prefix algorithm, only one generate/propagate signal pair exists per bit position that is connected to higher bit positions. It lies on the most critical path only if the adder is subdivided at bit positions being a power of two (i.e., $k = 2^x$). This case, however, can be avoided if necessary by cutting off the LSB from the prefix graph and thus shifting the entire graph to the right by one bit position. Thus, a Sklansky parallel-prefix adder can always be converted into a dual-size adder without lengthening the critical path.

Figures 5.13 and 5.14 show the mandatory multiplexer locations for dual-size Sklansky and Brent-Kung parallel-prefix adders for different values of k. Each multiplexer symbol actually represents a multiplexer for the generate signal and an AND-gate for the propagate signal, according to the following formulae:

$$\begin{aligned} G'_{k-1:j} &= G_{k-1:j}\overline{m} + c'_k m \\ P'_{k-1:j} &= P_{k-1:j}\overline{m} \end{aligned} \tag{5.14}$$

As can be seen, an additional multiplexer is used at the bottom of the adder graph for selection of the final carry.

As a conclusion, parallel-prefix adders can be partitioned and converted into dual-size adder circuits very effectively by only minor and simple modifications. The additional hardware costs as well as the speed degradation thereby are very small while the circuit regularity and simplicity is preserved.

5.7 Related Arithmetic Operations

Several arithmetic operations — such as subtraction, incrementation, and comparison — base on binary addition [Zim97]. For their efficient implementation, the presented adder architectures can be used and adapted accordingly. In particular, the parallel-prefix scheme proves to be universal and flexible enough to provide efficient circuit solutions for these operations, too.

5.7.1 2's Complement Subtractors

A *2's complement subtractor* relies on 2's complementation of the input operand B and subsequent addition, according to the following formula:

$$\begin{aligned} A - B &= A + (-B) \\ &= A + \overline{B} + 1 \end{aligned} \tag{5.15}$$

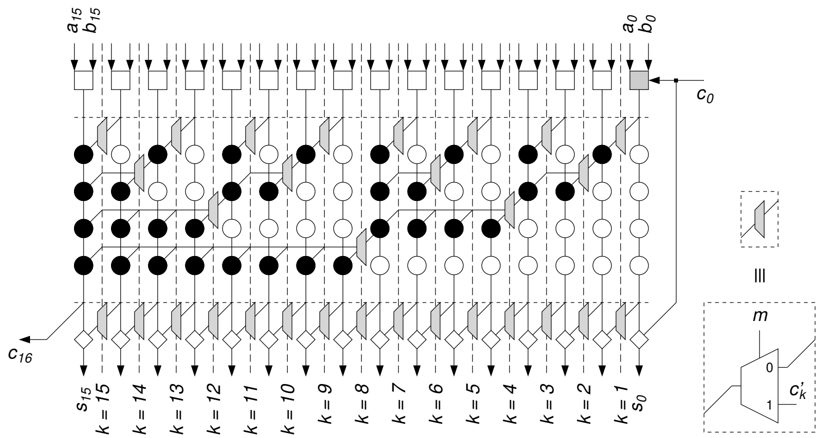


Figure 5.13: *Sklansky parallel-prefix dual-size adder with cutting line and required multiplexers for each value of k.*

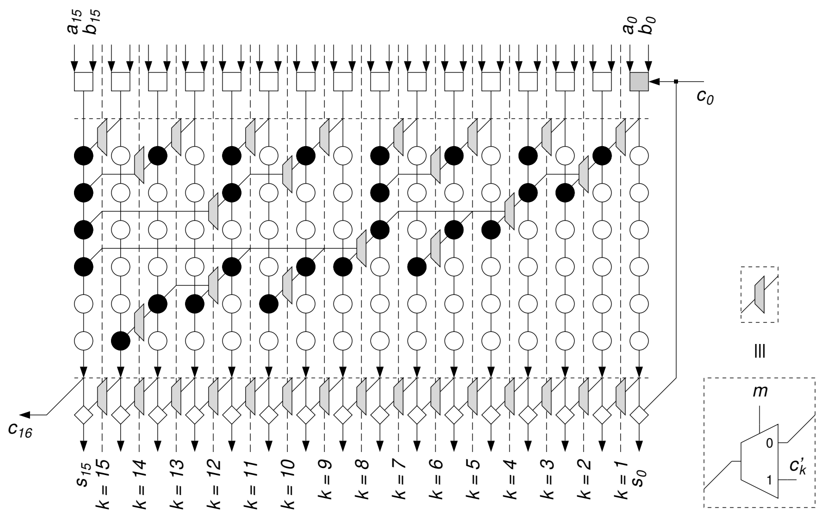


Figure 5.14: *Brent-Kung parallel-prefix dual-size adder with cutting line and required multiplexers for each value of k.*

Therefore, an arbitrary adder circuit can be taken with the input bits b_i inverted and the input carry set to 1.

A *2's complement adder/subtractor* performs either addition or subtraction as a function of the input signal sub:

$$\begin{aligned} A \pm B &= A + (-1)^{sub}\, B \\ &= A + (B \oplus sub) + sub \end{aligned} \tag{5.16}$$

The input operand B has to be conditionally inverted, which requires an XOR-gate at the input of each bit position. This increases the overall gate count by $2n$ and the gate delay by 2. There is no way to optimize size or delay any further, i.e., the XORs cannot be merged with the adder circuitry for optimization.

5.7.2 Incrementers / Decrementers

Incrementers and *decrementers* add or subtract one single bit c_{in} to/from an n-bit number ($A \pm c_{in}$). They can be regarded as adders with one input operand set to 0 ($B = 0$). Taking an efficient adder (subtractor) architecture and removing the redundancies originating from the constant inputs yields an efficient incrementer (decrementer) circuit. Due to the simplified carry propagation (i.e., $C_{i:k} = C_{i:j}C_{j:k}$), carry-chains and prefix trees consist of AND-gates only. This makes parallel-prefix structures even more efficient compared to other speed-up structures. Also, the resulting circuits are considerably smaller and faster than comparable adder circuits. Any prefix principles and structures discussed for adders work on incrementer circuits as well.

5.7.3 Comparators

Equality and *magnitude comparison* can be performed through subtraction by using the appropriate adder flags. Equality (EQ-flag) of two numbers A and B is indicated by the zero flag Z when computing $A - B$. As mentioned earlier, the Z flag corresponds to the propagate signal $P_{n-1:0}$ of the whole adder and is available for free in any parallel-prefix adder. The greater-equal (GE) flag corresponds to the carry-out c_{out} of the subtraction $A - B$. It is for free in any binary adder. All other flags (NE, LT, GT, LE) can be obtained from the EQ- and GE-flags by simple logic operations.

Since only two adder flags are used when comparing two numbers, the logic computing the (unused) sum bits can be omitted in an optimized comparator. The resulting circuit is not a prefix structure anymore (i.e., no intermediate signals are computed) but it can be implemented using a single binary tree. Therefore, a delay of $O(\log n)$ can be achieved with area $O(n)$ (instead of $O(n \log n)$). Again, a massive reduction in circuit delay and size is possible if compared to an entire adder.

6

Adder Synthesis

6.1 Introduction

Hardware synthesis can be addressed at different levels of hierarchy, as depicted in Figure 6.1. High-level or architectural synthesis deals with the mapping of some behavioral and abstract system or algorithm specification down to a block-level or register-transfer-level (RTL) circuit description by performing resource allocation, scheduling, and resource binding. Special circuit blocks — such as data paths, memories, and finite-state machines (FSM) — are synthesized at an intermediate level using dedicated algorithms and structure generators. Low-level or logic synthesis translates the structural description and logic equations of combinational blocks into a generic logic network. Finally, logic optimization and technology mapping is performed for efficient realization of the circuit on a target cell library and process technology.

The synthesis of data paths involves some high-level arithmetic optimizations — such as arithmetic transformations and allocation of standard arithmetic blocks — as well as low-level synthesis of circuit structures for the individual blocks. As mentioned in Section 2.4, dedicated structure generators are required for that purpose rather than standard logic synthesis algorithms. Generators for standard arithmetic operations, such as comparison, addition, and multiplication, are typically included in state-of-the-art synthesis tools.

Stand-alone netlist generators can be implemented for custom circuit struc-

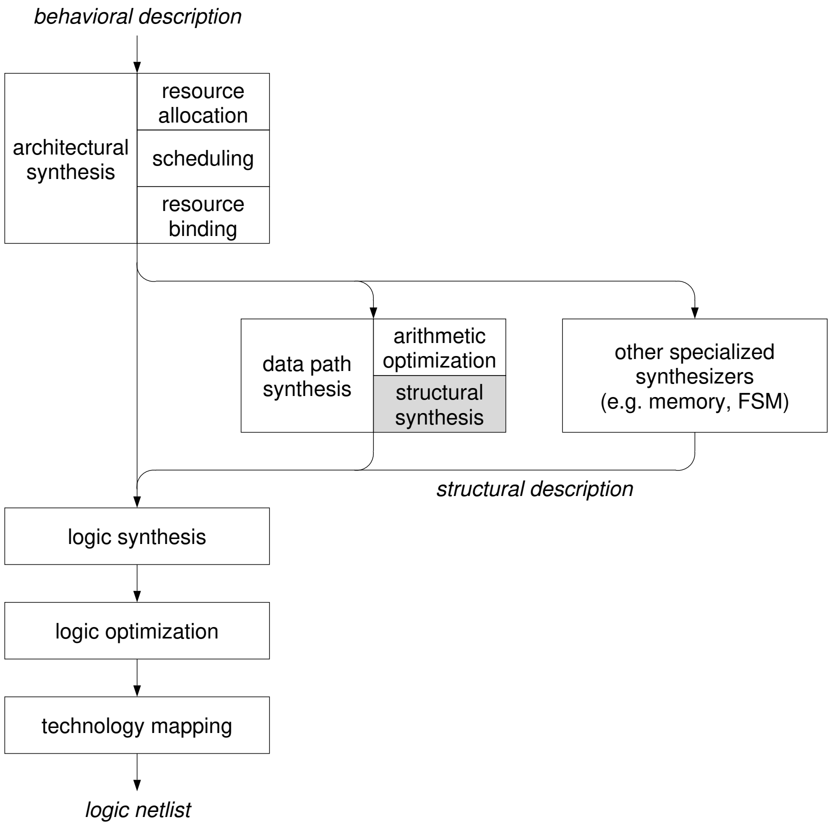


Figure 6.1: *Overview of hardware synthesis procedure.*

tures and special arithmetic blocks. They produce generic netlists, e.g., in form of structural VHDL code, which can be incorporated into a larger circuit through instantiation and synthesis. Such a netlist generator can be realized as a stand-alone software program or by way of a parameterized structural VHDL description.

This chapter deals with the synthesis of efficient adder structures for cell-based designs. That is, a design flow is assumed where synthesis generates generic netlists while standard software tools are used for technology mapping

and gate-level circuit optimization. Different synthesis algorithms are given for the generation of dedicated and highly flexible adder circuits.

6.2 Prefix Graphs and Adder Synthesis

It was shown in the previous chapters that the family of parallel-prefix adders provides the best adder architectures and the highest flexibility for custom adders. Their universal description by simple prefix graphs makes them also suitable for synthesis. It will be shown that there exists a simple graph transformation scheme which allows the automatic generation of arbitrary and highly optimized prefix graphs.

Therefore, this chapter focuses on the optimization and synthesis of prefix graphs, as formulated in the prefix problem equations (Eq. 3.25). The generation of prefix adders from a given prefix graph is then straightforward according to Equations 3.27–3.29 or Equations 3.32–3.34.

6.3 Synthesis of Fixed Parallel-Prefix Structures

The various prefix adder architectures described in Chapter 4, such as the ripple-carry, the carry-increment, and the carry-lookahead adders, all base on fixed prefix structures. Each of these prefix structures can be generated by a dedicated algorithm [KZ96]. These algorithms for the synthesis of fixed prefix structures are given in this section.

6.3.1 General Synthesis Algorithm

A general algorithm for the generation of prefix graphs bases on the prefix problem formalism of Eq. 3.25. Two nested loops are used in order to process the m prefix levels and the n bit positions.

Algorithm: General prefix graph

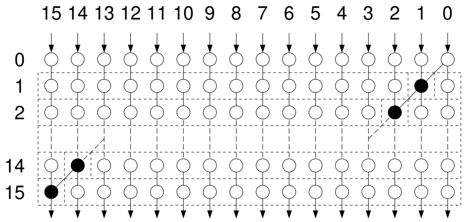
for ($i = 0$ to $n-1$) $Y_i^0 = x_i$;
for ($l = 1$ to m) {
  for ($i = 0$ to $n-1$) {
    if (white node) $Y_i^l = Y_i^{l-1}$;
    if (black node) $Y_i^l = Y_i^{l-1} \bullet Y_j^{l-1}$; /* $0 \leq j < i$ */
  }
}
for ($i = 0$ to $n-1$) $y_i = Y_i^m$;

Note that the group variables Y_i^l are now written with a simple index i representing the significant bit position rather than an index range $i:k$ of the bit group they are representing (i.e., $Y_{i:k}^l$ was used in Eq. 3.25). For programming purposes, the prefix variables Y_i^l can be described as a two-dimensional array of signals (e.g., $Y(l, i)$) with dimensions m (number of prefix levels) and n (number of bits). The algorithms are given in simple pseudo code. Only simple condition and index calculations are used so that the code can easily be implemented in parameterized structural VHDL and synthesized by state-of-the-art synthesis tools [KZ96].

6.3.2 Serial-Prefix Graph

The synthesis of a serial-prefix graph is straightforward since it consists of a linear chain of $\bullet$-operators. Two algorithms are given here. The first algorithm bases on the general algorithm introduced previously and generates $m = n - 1$ prefix levels. Each level is composed of three building blocks, as depicted in the prefix graph below: a lower section of white nodes, one black node in-between, and an upper section of white nodes.

Prefix graph:

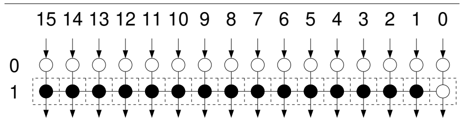


Algorithm: Serial-prefix graph

for ($i = 0$ to $n-1$) $Y_i^0 = x_i$;
for ($l = 1$ to $n-1$) {
  for ($i = 0$ to $l-1$) $Y_i^l = Y_i^{l-1}$;
  $Y_l^l = Y_l^{l-1} \bullet Y_{l-1}^{l-1}$;
  for ($i = l+1$ to $n-1$) $Y_i^l = Y_i^{l-1}$;
}
for ($i = 0$ to $n-1$) $y_i = Y_i^{n-1}$;

The second algorithm is much simpler and bases on the fact that the graph can be reduced to one prefix level because each column consists of only one $\bullet$-operator. Here, neighboring black nodes are connected horizontally. This algorithm implements Equation 3.24 directly.

Reduced prefix graph:



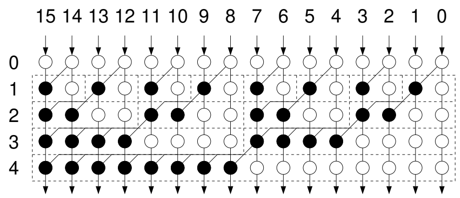
Algorithm: Serial-prefix graph (optimized)

$y_0 = x_0$;
for ($i = 1$ to $n-1$) $y_i = x_i \bullet y_{i-1}$;

6.3.3 Sklansky Parallel-Prefix Graph

The minimal-depth parallel-prefix structure by Sklansky (structure depth $m = \log n$) can be generated using a quite simple and regular algorithm. For that purpose, each prefix level l is divided into 2^{m-l} building blocks of size 2^l. Each building block is composed of a lower half of white nodes and an upper half of black nodes. This can be implemented by three nested loops as shown in the algorithm given below. The if-statements in the innermost loop are necessary for adder word lengths that are not a power of two ($n < 2^m$) in order to avoid the generation of logic for bits $> n-1$.

Prefix graph:



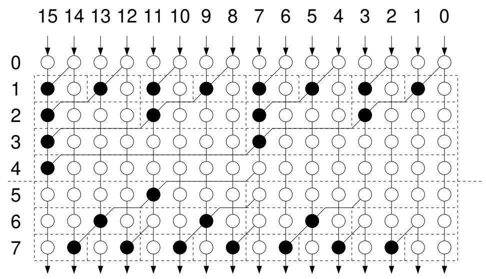
Algorithm: Sklansky parallel-prefix graph

$m = \lceil \log n \rceil;$
for ($i = 0$ to $n-1$) $Y_i^0 = x_i;$
for ($l = 1$ to m) {
  for ($k = 0$ to $2^{m-l} - 1$) {
    for ($i = 0$ to $2^{l-1} - 1$) {
      if ($k2^l + i < n$) $Y^l_{k2^l+i} = Y^{l-1}_{k2^l+i};$
      if ($k2^l + 2^{l-1} + i < n$)
        $Y^l_{k2^l+2^{l-1}+i} = Y^{l-1}_{k2^l+2^{l-1}+i} \bullet Y^{l-1}_{k2^l+2^{l-1}-1};$
    }
  }
}
for ($i = 0$ to $n-1$) $y_i = Y_i^m;$

6.3.4 Brent-Kung Parallel-Prefix Graph

The algorithm for the Brent-Kung parallel-prefix structure is more complex since two tree structures are to be generated: one for carry collection and the other for carry redistribution (see prefix graph below). The upper part of the prefix graph has similar building blocks as the Sklansky algorithm with, however, only one black node in each. The lower part has two building blocks on each level: one on the right with no black nodes followed by one or more blocks with one black node each. For simplicity, the algorithm is given for word lengths equal to a power of two only ($n = 2^m$). It can easily be adapted for arbitrary word lengths by adding if-statements at the appropriate places (as in the Sklansky algorithm).

Prefix graph:



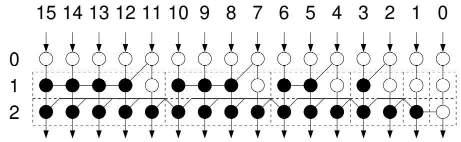
Algorithm: Brent-Kung parallel-prefix graph

$$
\begin{array}{l}
m = \lceil \log n \rceil; \\
\textbf{for}\ (i = 0 \text{ to } n-1)\ Y_i^0 = x_i; \\
\textbf{for}\ (l = 1 \text{ to } m)\ \{ \\
\quad \textbf{for}\ (k = 0 \text{ to } 2^{m-l}-1)\ \{ \\
\quad\quad \textbf{for}\ (i = 0 \text{ to } 2^l-2)\ Y^l_{k2^l+i} = Y^{l-1}_{k2^l+i}; \\
\quad\quad Y^l_{k2^l+2^l-1} = Y^{l-1}_{k2^l+2^l-1} \bullet Y^{l-1}_{k2^l+2^{l-1}-1}; \\
\quad \} \\
\} \\
\textbf{for}\ (l = m+1 \text{ to } 2m-1)\ \{ \\
\quad \textbf{for}\ (i = 0 \text{ to } 2^{2m-l}-1)\ Y^l_i = Y^{l-1}_i; \\
\quad \textbf{for}\ (k = 1 \text{ to } 2^{l-m}-1)\ \{ \\
\quad\quad \textbf{for}\ (i = 0 \text{ to } 2^{2m-l-1}-2)\ Y^l_{k2^{2m-l}+i} = Y^{l-1}_{k2^{2m-l}+i}; \\
\quad\quad Y^l_{k2^{2m-l}+2^{2m-l-1}-1} = Y^{l-1}_{k2^{2m-l}+2^{2m-l-1}-1} \bullet Y^{l-1}_{k2^{2m-l}-1}; \\
\quad\quad \textbf{for}\ (i = 2^{2m-l-1} \text{ to } 2^{2m-l}-1)\ Y^l_{k2^{2m-l}+i} = Y^{l-1}_{k2^{2m-l}+i}; \\
\quad \} \\
\} \\
\textbf{for}\ (i = 0 \text{ to } n-1)\ y_i = Y_i^{2m-1};
\end{array}
$$

6.3.5 1-Level Carry-Increment Parallel-Prefix Graph

Similarly to the serial-prefix graph, the 1-level carry-increment prefix graph of Figure 3.24 can be reduced to two prefix levels (see prefix graph below) with horizontal connections between adjacent nodes. The algorithm is quite simple, despite the more complex group size properties. The square root evaluation for the upper limit of the loop variable k must not be accurate since the generation of logic is omitted anyway for indices higher than $n-1$. Therefore, the value can be approximated by a simpler expression for which $\lceil \sqrt{2n} \rceil$ must be a lower bound.

Reduced prefix graph:



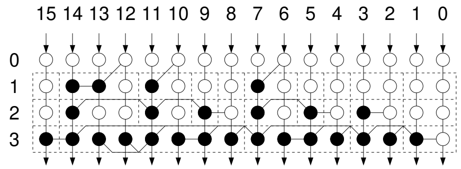
Algorithm: 1-level carry-increment parallel-prefix graph

$$
\begin{array}{l}
\textbf{for}\ (i = 0 \text{ to } n-1)\ Y_i^0 = x_i; \\
Y_0^2 = Y_0^1 = Y_0^0; \\
\textbf{for}\ (k = 0 \text{ to } \lceil \sqrt{2n} \rceil)\ \{ \\
\quad j = k(k-1)/2 + 1; \\
\quad \textbf{if}\ (k > 0)\ Y_j^1 = Y_j^0; \\
\quad \textbf{for}\ (i = 1 \text{ to } \min(k,\ n-j)-1)\ Y^1_{j+i} = Y^0_{j+i} \bullet Y^1_{j+i-1}; \\
\quad \textbf{for}\ (i = 0 \text{ to } \min(k,\ n-j)-1)\ Y^2_{j+i} = Y^1_{j+i} \bullet Y^2_{j-1}; \\
\} \\
\textbf{for}\ (i = 0 \text{ to } n-1)\ y_i = Y_i^2;
\end{array}
$$

6.3.6 2-Level Carry-Increment Parallel-Prefix Graph

The prefix graph below shows how the 2-level carry-increment parallel-prefix graph of Figure 3.26 can be reduced to three prefix levels. Again, the graph can be generated by a similar, but more complex algorithm as used for the 1-level version. Since the implementation details are rather tricky, the algorithm details are not given here. This is justified by the fact that the universal prefix graph synthesis algorithm presented in the next section is able to generate this prefix structure as well.

Prefix graph:



6.4 Synthesis of Flexible Parallel-Prefix Structures

Each synthesis algorithm presented in the previous section generates a dedicated parallel-prefix structure. Thus, a variety of algorithms is required for the generation of some few prefix structures.

This section describes a universal and flexible algorithm for the optimization and synthesis of prefix structures which is based on local prefix graph transformations [Zim96]. This efficient non-heuristic algorithm allows the synthesis of all of the above prefix structures and many more. It generates size-optimal parallel-prefix structures under arbitrary depth constraints and thereby also accommodates also non-uniform input signal arrival and output signal required times.

6.4.1 Introduction

The synthesis of adder circuits with different performance characteristics is standard in today's ASIC design packages. However, only limited flexibility is usually provided to the user for customization to a particular situation. The most common circuit constraints arise from dedicated timing requirements, which may include arbitrary input and output signal arrival profiles, e.g., as found in the final adder of multipliers [Okl94]. The task of meeting all timing constraints while minimizing circuit size is usually left to the logic optimization step which starts from an adder circuit designed for uniform signal arrival times. Taking advantage of individual signal arrival times is therefore very limited and computation intensive. If, however, timing specifications can be taken into account earlier during adder synthesis, more efficient circuits as well as considerably smaller logic optimization efforts will result. The task of adder synthesis is therefore to generate an adder circuit with minimal hardware which meets all timing constraints. This, however, asks for an adder architecture which has a simple, regular structure and results in well-performing circuits, and which provides a wide range of area-delay trade-offs as well as enough flexibility for accommodating non-uniform signal arrival profiles.

6.4.2 Parallel-Prefix Adders Revisited

All of the above adder requirements are met by the family of parallel-prefix adders comprising the ripple-carry, carry-increment, and the carry-lookahead adders, as outlined earlier in this thesis. Let us now shortly summarize these adder architectures from a parallel-prefix structure point of view. Thereby, we rely on the prefix addition formalism and structure introduced in Section 3.5.

Due to the associativity of the prefix operator $\bullet$, a sequence of operations can be evaluated in any order. Serial evaluation from the LSB to the MSB has the advantage that all intermediate prefix outputs are generated as well. The resulting *serial-prefix* structure does with the minimal number of $n-1$ black nodes but has maximal evaluation depth of $n-1$ (Fig. 6.2). It corresponds to ripple-carry addition. Parallel application of operators by arranging them in tree structures allows a reduction of the evaluation depth down to $\log n$. In the resulting *parallel-prefix* structures, however, additional black nodes are required for implementing evaluation trees for all prefix outputs. Therefore, structure depth (i.e., number of black nodes on the critical path, circuit delay) — ranging from $n-1$ down to $\log n$, depending on the degree of parallelism — can be traded off versus structure size (i.e., total number of black nodes, circuit area). Furthermore, the various parallel-prefix structures differ in terms of wiring complexity and fan-out.

Adders based on these parallel-prefix structures are called *parallel-prefix adders* and are basically carry-lookahead adders with different lookahead schemes. The fastest but largest adder uses the parallel-prefix structure introduced by Sklansky [Skl60] (Fig. 6.3(c)). The prefix structure proposed by Brent and Kung [BK82] offers a trade-off having almost twice the depth but much fewer black nodes (Fig. 6.3(d)). The linear size-to-depth trade-off described by Snir [Sni86] allows for mixed serial/parallel-prefix structures of any depth between $2\log n-3$ and $n-1$, thus filling the gap between the serial-prefix and the Brent-Kung parallel-prefix structure. The carry-increment parallel-prefix structures exploit parallelism by hierarchical levels of serial evaluation chains rather than tree structures (Figs. 6.3(a) and (b)). This results in prefix structures with a fixed maximum number of black nodes per bit position ($\#^{max}_{\bullet/b}$) as a function of the number of applied increment levels (i.e., $\#^{max}_{\bullet/b}-1$ levels). They are also called *bounded*-$\#^{max}_{\bullet/b}$ prefix structures in this section. Note that, depending on the number of increment levels, this carry-increment prefix structure lies somewhere between the serial-prefix ($\#^{max}_{\bullet/b}=1$) and the Sklansky parallel-prefix structure ($\#^{max}_{\bullet/b}=\log n$).

$D_\bullet$	$\#_\bullet$
31	31

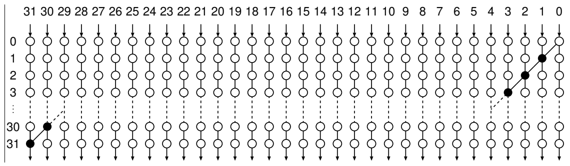


Figure 6.2: *Ripple-carry serial-prefix structure.*

(a)

$D_\bullet$	$\#_\bullet$
8	54

(b)

$D_\bullet$	$\#_\bullet$
6	68

(c)

$D_\bullet$	$\#_\bullet$
5	80

(d)

$D_\bullet$	$\#_\bullet$
8	57

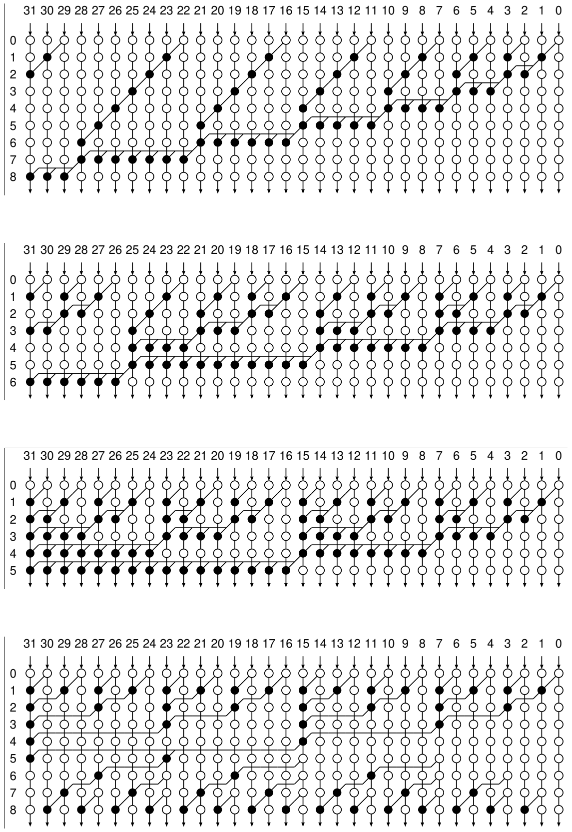


Figure 6.3: *(a) 1-level carry-increment, (b) 2-level carry-increment, (c) Sklansky, and (d) Brent-Kung parallel-prefix structure.*

All these prefix structures have growing maximum fan-out numbers (i.e., out-degree of black nodes) if parallelism is increased. This has a negative effect on speed in real circuit implementations. A fundamentally different prefix tree structure proposed by Kogge and Stone [KS73] has all fan-out bounded by 2, at the minimum structure depth of $\log n$. However, the massively higher circuit and wiring complexity (i.e., more black nodes and edges) undoes the advantages of bounded fan-out in most cases. A mixture of the Kogge-Stone and Brent-Kung prefix structures proposed by Han and Carlson [HC87] corrects this problem to some degree. Also, these two bounded fan-out parallel-prefix structures are not compatible with the other structures and the synthesis algorithm presented in this section, and thus were not considered any further for adder synthesis.

Table 6.1 summarizes some characteristics of the serial-prefix and the most common parallel-prefix structures with respect to:

$D_\bullet$: maximum depth, number of black nodes on the critical path,

$\#_\bullet$: size, total number of black nodes,

$\#^{max}_{\bullet/b}$: maximum number of black nodes per bit position,

$\#_{tracks}$: wiring complexity, horizontal tracks in the graph,

$FO^{max}_\bullet$: maximum fan-out,

synthesis : compatibility with the presented optimization algorithm, and

$A\,/\,T$: area and delay performance.

The area/delay performance figures are obtained from a very rough classification based on the standard-cell comparisons reported in Section 4.2. A similar performance characterization of parallel-prefix adders can be found in [TVG95].

6.4.3 Optimization and Synthesis of Prefix Structures

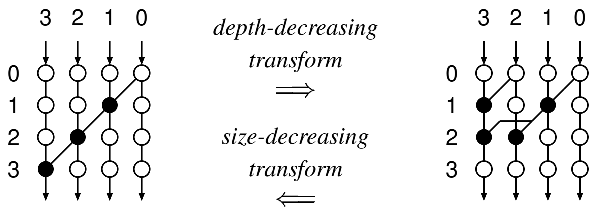
Prefix Transformation

The optimization of prefix structures bases on a simple local equivalence transformation (i.e., factorization) of the prefix graph [Fis90], called *prefix transformation* in this context.

Table 6.1: *Characteristics of common prefix structures.*

prefix structure	$D_\bullet$	$\#_\bullet$	$\#^{max}_{\bullet/b}$	$\#_{tracks}$	$FO^{max}_\bullet$	synthesis (this work)	perform. A	perform. T
serial	$n-1$	$n-1$	1	1	2	yes	$++$	$--$
1-level carry-incr. par.	$\sim\sqrt{2n}$	$\sim 2n-\sqrt{2n}-2$	2	2	$\sim\sqrt{2n}$	yes	$+$	$-$
2-level carry-incr. par.	$\sim\sqrt[3]{6n}$	$\sim 3n - \ldots$	3	3	$\sim(6n)^{2/3}$	yes	$-$	$+$
Sklansky parallel	$\log n$	$\frac{1}{2}n\log n$	$\log n$	$\log n$	$\frac{1}{2}n+1$	yes	$-$	$++$
Brent-Kung parallel	$2\log n-2$	$2n-\log n-2$	$\log n$	$2\log n-1$	$\log n+1$	yes	$+$	$-$
Kogge-Stone parallel	$\log n$	$n\log n-n+1$	$\log n$	$n-1$	2	no	$--$	$++$
Han-Carlson parallel	$\log n+1$	$\frac{1}{2}n\log n$	$\log n$	$\frac{1}{2}n+1$	3	no	$-$	$+$
Snir variable ser./par.	$n-1-k$ *	$n-1+k$ *				yes	variable	

* range of size-depth trade-off parameter $k : 0 \le k \le n - 2\log n + 2$



By using this basic transformation, a serial structure of three black nodes with $D_\bullet = 3$ and $\#_\bullet = 3$ is transformed into a parallel tree structure with $D_\bullet = 2$ and $\#_\bullet = 4$ (see Fig. above). Thus, the depth is reduced while the size is increased by one $\bullet$-operator. The transformation can be applied in both directions in order to minimize structure depth (i.e., depth-decreasing transform) or structure size (i.e., size-decreasing transform), respectively.

This local transformation can be applied repeatedly to larger prefix graphs resulting in an overall minimization of structure depth or size or both. A transformation is possible under the following conditions, where (i, l) denotes the node in the i-th column and l-th row of the graph:

$\Longrightarrow$: nodes $(3, 1)$ and $(3, 2)$ are white,

$\Longleftarrow$: node $(3, 3)$ is white and
nodes $(3, 1)$ and $(3, 2)$ have no successors $(i, 2)$ or $(i, 3)$ with $i > 3$.

It is important to note that the selection and sequence of local transformations is crucial for the quality of the final global optimization result. Different heuristic and non-heuristic algorithms exist for solving this problem.

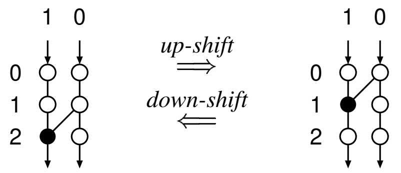
Heuristic Optimization Algorithms

Heuristic algorithms based on local transformations are widely used for delay and area optimization of logic networks [SWBSV88, Mic94]. Fishburn applied this technique to the timing optimization of prefix circuits and of adders in particular [Fis90], and similar work was done by Guyot [GBB94]. The basic transformation described above is used. However, more complex transforms are derived and stored in a library. An area-minimized logic network together with the timing constraints expressed as input and output signal arrival times are given. Then, repeated local transformations are applied to subcircuits until the timing requirements are met. These subcircuits are selected heuristically, that is, all possible transforms on the most critical path are evaluated by consulting the library, and the simplest one with the best benefit/cost ratio is then carried out.

On one hand, the advantage of such heuristic methods lies in their generality, which enables the optimization of arbitrary logic networks and graphs. On the other hand, the computation effort — which includes static timing analysis, search for possible transformations, and the benefit/cost function evaluation — is very high and can be lessened only to some degree by relying on comprehensive libraries of precomputed transformations. Also, general heuristics are hard to find and only suboptimal in most cases. In the case of parallel-prefix binary addition, very specific heuristics are required in order to obtain perfect prefix trees and the globally optimal adder circuits reported by Fishburn.

Non-Heuristic Optimization Algorithm

In the heuristic optimization algorithms, only depth-decreasing transformations are applied which are necessary to meet the timing specifications and therefore are selected heuristically. In the new approach proposed in this thesis, all possible depth-decreasing transformations (prefix graph *compression*) are perform first, resulting in the fastest existing prefix structure. In a second step, size-decreasing transformations are applied wherever possible in order to minimize structure size while remaining in the permitted depth range (depth-controlled prefix graph *expansion*). It can be shown that the resulting prefix structures are optimal in most cases and near-optimal otherwise if the transformations are applied in a simple linear sequence, thus requiring no heuristics at all. Only a trivial up- and down-shift operation of black nodes is used in addition to the basic prefix transformation described above.



The conditions for the shift operations are:

$\Longrightarrow$: nodes $(1,1)$ and $(0,1)$ are white,

$\Longleftarrow$: node $(1,2)$ is white and node $(1,1)$ has no successor $(i,2)$ with $i > 1$.

Timing constraints are taken into account by setting appropriate top and bottom margins for each column.

Step 1) *Prefix graph compression*: Compressing a prefix graph means decreasing its depth at the cost of increased size, resulting in a faster circuit implementation. Prefix graph compression is achieved by shifting up the black nodes in each column as far as possible using depth-decreasing transform and up-shift operations. The recursive function *COMPRESS_COLUMN* (i,l) shifts up a black node (i,l) by one position by applying an up-shift or depth-decreasing transform, if possible. It is called recursively for node $(i, l-1)$ starting at node (i,m), thus working on an entire column from bottom to top. The return value is true if node (i,l) is white (i.e., if a black node (i,l) can be shifted further up), false otherwise. It is used to decide whether a transformation at node (i,l) is possible. The procedure *COMPRESS_GRAPH* () compresses the entire prefix graph by calling the column compressing function for each bit position in a linear sequence from the LSB to the MSB. It can easily be seen that the right-to-left bottom-up graph traversal scheme used always generates prefix graphs of minimal depth, which in the case of uniform signal arrival times corresponds to the Sklansky prefix structure. The pseudo code for prefix graph compression is given below.

This simple compression algorithm assumes to start from a serial-prefix graph (i.e., only one black node exists per column initially). The algorithm can easily be extended by distinguishing an additional case in order to work on arbitrary prefix graphs. However, in order to get a perfect minimum-depth graph, it must start from serial-prefix graph.

Step 2) *Prefix graph expansion*: Expanding a prefix graph basically means reducing its size at the cost of an increased depth. The prefix graph obtained after compression has minimal depth on all outputs at maximum graph size. If depth specifications are still not met, no solution exists. If, however, graph depth is smaller than required, the columns of the graph can be expanded again in order to minimize graph size. At the same time, fan-out numbers on the critical nets are reduced thus making circuit implementations faster again by some small amount. The process of graph expansion is exactly the opposite of graph compression. In other words, graph expansion undoes all unnecessary steps from graph compression. This makes sense since the necessity of a depth-decreasing step in column i is not a priori known during graph compression because it affects columns $j > i$ which are processed

Algorithm: Prefix graph compression

```
COMPRESS_GRAPH () {
    for (i = 0 to n − 1)
        COMPRESS_COLUMN (i, m);
}

boolean COMPRESS_COLUMN (i, l) {
                                        /* return value = (node (i, l) is white) */
    if (node (i, l) is at top of column i) return false;
    else if (node (i, l) is white) {
        COMPRESS_COLUMN (i, l − 1);
        return true;
    } else if (black node (i, l) has white predecessor (j, l − 1)) {
        if (predecessor (j, l − 1) is at top of column j) return false;
        else {
            shift up black node (i, l) to position (i, l − 1);
            COMPRESS_COLUMN (i, l − 1);
            return true;
        }
    } else {            /* black node (i, l) has black predecessor (j, l − 1) */
        shift up black node (i, l) to position (i, l − 1);
        if (COMPRESS_COLUMN (i, l − 1)) {
                        /* node (k, l − 2) is predecessor of node (j, l − 1) */
            insert black node (i, l − 1) with predecessor (k, l − 2);
            return true;
        } else {
            shift back black node (i, l − 1) down to position (i, l);
            return false;
        }
    }
}
```

later. Thus, prefix graph expansion performs down-shift and size-decreasing transform operations in a left-to-right top-down graph traversal order wherever possible (*EXPAND_GRAPH* (i, l) and *EXPAND_COLUMN* ()). The pseudo code is therefore very similar to the code for graph compression (see below).

This expansion algorithm assumes to work on a minimum-depth prefix graph obtained from the above compression step. Again, it can easily be

Algorithm: Prefix graph expansion

```
EXPAND_GRAPH () {
    for (i = n − 1 to 0)
        EXPAND_COLUMN (i, 1);
}

boolean EXPAND_COLUMN (i, l) {
                                        /* return value = (node (i, l) is white) */
    if (node (i, l) is at bottom of column i) return false;
    else if (node (i, l) is white) {
        EXPAND_COLUMN (i, l + 1);
        return true;
    } else if (black node (i, l) has at least one successor) {
        EXPAND_COLUMN (i, l + 1);
        return false;
    } else if (node (i, l + 1) is white) {
            shift down black node (i, l) to position (i, l + 1);
            EXPAND_COLUMN (i, l + 1);
            return true;
    } else {            /* black node (i, l) from depth-decreasing transform */
                        /* node (k, l) is predecessor of node (i, l + 1) */
        remove black node (i, l + 1) with predecessor (k, l);
        shift down black node (i, l) to position (i, l + 1);
        if (EXPAND_COLUMN (i, l + 1)) return true;
        else {
            shift back black node (i, l + 1) up to position (i, l);
            re-insert black node (i, l + 1) with predecessor (j, l + 1);
            return false;
        }
    }
}
```

adapted in order to process arbitrary prefix graphs. Under relaxed timing constraints, it will convert any parallel-prefix structure into a serial-prefix one.

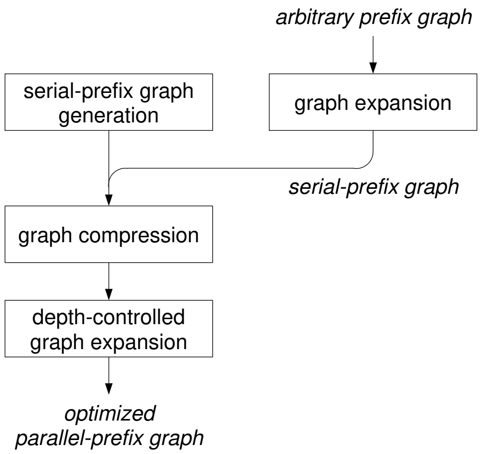


Figure 6.4: *Prefix graph synthesis.*

Synthesis of Parallel-Prefix Graphs

The synthesis of size-optimal parallel-prefix graphs — and with that of parallel-prefix adders — under given depth constraints is now trivial. A serial-prefix structure is first generated which then undergoes a graph compression step and a depth-controlled graph expansion step, as illustrated in Figure 6.4. For a more intuitive graph representation, a final up-shift step can be added which shifts up all black nodes as far as possible without performing any transformation, thus leaving the graph structure unchanged (used in Figs. 6.5–6.11).

Carry-increment (i.e., bounded-$\#^{max}_{\bullet/b}$) prefix structures are obtained by limiting the number of black nodes per column ($\#^{max}_{\bullet/b}$) through an additional case distinction in the graph compression algorithm. Also, a simple adaption of the graph expansion algorithm allows the generation of size-constrained prefix structures (i.e., the total number of black nodes is limited), resulting in the synthesis of area-constrained adder circuits.

6.4.4 Experimental Results and Discussion

The described synthesis algorithm was implemented as a C-program and tested for a wide range of word lengths and depth constraints. The runtime efficiency of the program is very high thanks to the simple graph traversal algorithms, resulting in computation times below 1s for prefix graphs of up to several hundred bits (Sun SPARCstation-10).

Uniform Signal Arrival Profiles

Figures 6.8(a)–(e) depict the synthesized parallel-prefix structures of depth five to eight and 12 for uniform signal arrival times. Structure depth ($D_\bullet$) and size ($\#_\bullet$) are indicated for each graph. The numbers in parenthesis correspond to structure depth and size after the compression but before the expansion step.

The structures (a) and (d) are size-optimized versions of the Sklansky and Brent-Kung prefix graphs.

For depths in the range of $(2 \log n - 3) \leq D_\bullet \leq (n-1)$ a linear trade-off exists between structure depth and size [Sni86]. This is expressed by the lower bound $(D_\bullet + \#_\bullet) \geq (2n-2)$ which is achieved by the synthesized structures, i.e., the algorithm generates size-optimal solutions within this range of structure depths. This linear trade-off exists because the prefix structures are divided into an upper serial-prefix region (with one black node per bit) and a lower Brent-Kung parallel-prefix region (with two black nodes per bit on the average). Changing the structure depth by some value therefore simply moves the border between the two regions (and with that the number of black nodes) by the same amount (Figs. 6.8(c)–(e)). In other words, one depth-decreasing transform suffices for an overall graph depth reduction by one.

In the depth range $\log n \leq D_\bullet < (2 \log n - 3)$, however, decreasing structure depth requires shortening of more than one critical path, resulting in an exponential size-depth trade-off (Figs. 6.8(a)–(c)). Put differently, an increasing number of depth-decreasing transforms has to be applied for an overall graph depth reduction by one, as depth gets closer to $\log n$. Most synthesized structures in this range are only near-optimal (except for the structure with minimum depth of $\log n$). A strictly size-optimal solution is obtained by a bounded-$\#^{max}_{\bullet/b}$ prefix structure with a specific $\#^{max}_{\bullet/b}$ value (compare Figs. 6.5 and 6.8(b)).

Non-Uniform Signal Arrival Profiles

Various non-uniform signal arrival profiles were applied, such as late upper/lower half-words, late single bits, and increasing/decreasing profiles on inputs, and vice versa for the outputs. For most profiles, size-optimal or near-optimal structures were generated using the basic algorithm with unbounded $\#^{max}_{\bullet/b}$. As an example, Figures 6.7(a) and (b) show how a single bit which is late by four black node delays can be accommodated at any bit position in a prefix structure with depth $D_\bullet = \log n + 1$. The structure of Figure 6.6 has a fast MSB output (corresponds to the carry-out in a prefix adder) and is equivalent to the Brent-Kung prefix algorithm. Figures 6.9(a)–(d) depict the synthesized prefix graphs for late input and early output upper and lower half-words.

Input signal profiles with steep "negative slopes" (i.e., bit i arrives earlier than bit $i-1$ by one $\bullet$-operator delay for each i) are the only exceptions for which inefficient solutions with many black nodes in some columns are generated. This, however, can be avoided by using bounded-$\#^{max}_{\bullet/b}$ prefix structures. It can be observed that by bounding the number of black nodes per column by $\log n$ ($\#^{max}_{\bullet/b} = \log n$), size-optimal structures are obtained. This is demonstrated in Figure 6.10 with a typical input signal profile found in the final adder of a multiplier, originating from an unbalanced Wallace tree adder. This example shows the efficient combination of serial and parallel substructures generated, which smoothly adapts to the given signal profiles. In Figure 6.11, the same signal profile with less steep slopes is used.

Discussion

As mentioned above, cases exist where size-optimal solutions are obtained only by using bounded-$\#^{max}_{\bullet/b}$ parallel-prefix structures. However, near-optimal structures are generated throughout by setting $\#^{max}_{\bullet/b} = \log n$. Note that this bound normally does not come into effect since most structures (e.g., all structures with uniform signal arrival profiles) have $\#^{max}_{\bullet/b} \leq \log n$ by default.

The synthesis algorithm presented works for any word length n. Because it works on entire prefix graphs, it can be used for structural synthesis but not for the optimization of existing logic networks. For the latter, the corresponding prefix graph has first to be extracted which, however, resembles the procedure of subcircuit optimization in the heuristic methods.

Fan-out significantly influence circuit performance. The total sum of fan-out in an arbitrary prefix structure is primarily determined by its degree of parallelism and thus by its depth. In the prefix structures used in this work, the accumulated fan-out on the critical path, which determines the circuit delay, is barely influenced by the synthesis algorithm. This is why fan-out is not considered during synthesis. Appropriate buffering and fan-out decoupling of uncritical from critical signal nets is left to the logic optimization and technology mapping step which is always performed after logic synthesis.

Validation of the results on silicon bases on the standard-cell implementations described in Section 4.2, where the prefix adders used in this work showed the best performance measures of all adder architectures. As far as technology mapping is concerned, the synthesized prefix structures can be mapped very efficiently onto typical standard-cell libraries, since the basic logic functions (such as AND-OR, AND, and XOR) exist as cells in any library. Most libraries also include optimized full-adder cells, which can be used for the efficient realization of serial-prefix structures (see also Sec. 7.4).

6.4.5 Parallel-Prefix Schedules with Resource Constraints

Parallel-prefix computation not only plays an important role in adder and circuit design, but also in digital signal processing, graph optimizations, computational geometry, and loop parallelization containing loop-carried dependencies. Here, we have to distinguish between problems with simple prefix operations, where all of them are typically performed in parallel (e.g., combinational circuits), and applications with complex prefix operations, where one single or only a few parallel operations are executed sequentially in m time steps in order to perform the entire computation (corresponds to the m levels in a prefix graph). Since in many such applications the amount of resources — such as functional units or processors — is fixed and independent of the problem size, schemes or *schedules* are required which perform a prefix computation in minimal time under certain resource constraints [WNS96]. In particular, a prefix problem of size n has to be computed on p processors with minimal time steps. This problem can be translated into the prefix graph domain, where a prefix graph of width n and minimal depth is to be found having a maximum number of p black nodes per row. The similarities between this prefix graph optimization problem and the optimization problems discussed in this chapter so far, but also the fact that these problems can be solved by the same algorithm, are the reasons why it is mentioned at this point. On the other hand, this resource-constrained prefix problem has no significance in adder design, since for combinational circuits only constraining the total number of black nodes, but not the number of black nodes per prefix level, makes sense.

In [WNS96], several algorithms are presented which yield time-optimal schedules for prefix computation problems. Basically, two problem sizes are distinguished:

$n > p(p+1)/2$: Time-optimal *harmonic schedules* are generated using a simple algorithm. The harmonic schedule for $n = p(p+1)/2$ is equivalent to the 1-level carry-increment parallel-prefix structure generated by our synthesis algorithm with $\#^{max}_{\bullet/b} = 2$. A harmonic schedule for larger n simply repeats this former schedule for higher bits, which in our algorithm can be achieved using an additional $\#^{max}_{\bullet/l} = 2$ parameter (i.e., maximum number of black nodes per prefix level). An example of a synthesized harmonic schedule is given in Figure 6.12.

$n \le p(p+1)/2$: A general scheme for generation of strict time-optimal schedules (also for $n > p(p+1)/2$) is described. The algorithm proposed is quite complex, and these schedules cannot be generated by our synthesis algorithm. However, the above harmonic schedules yield near-optimal schedules, which in the worst case are deeper by only two levels and in the typical case by one level.

Thus, the universal parallel-prefix synthesis algorithm proposed in this work also generates harmonic schedules used for resource-constrained parallel-prefix computations. These schedules are time-optimal for $n > p(p+1)/2$ and near-optimal for $n \le p(p+1)/2$. However, the direct synthesis algorithm presented in [WNS96] for harmonic schedules is simpler and more efficient (analogously to the algorithms for fixed parallel-prefix structures of Sec. 6.3).

$D_\bullet$	$\#_\bullet$
6	56
(6)	(68)

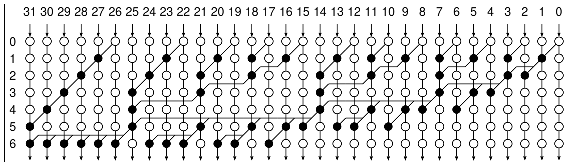


Figure 6.5: *Synthesized minimum-depth bounded-$\#^{\max}_{\bullet/b}$ prefix structure ($\#^{\max}_{\bullet/b} = 3$).*

$D_\bullet$	$\#_\bullet$
8	57
(8)	(80)

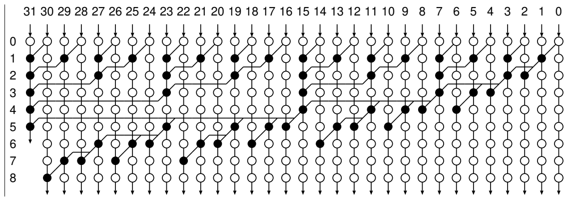


Figure 6.6: *Synthesized minimum-depth prefix structure for the MSB output early by 3 $\bullet$-delays.*

$D_\bullet$	$\#_\bullet$
6	78
(6)	(86)

(a)

$D_\bullet$	$\#_\bullet$
6	68
(6)	(77)

(b)

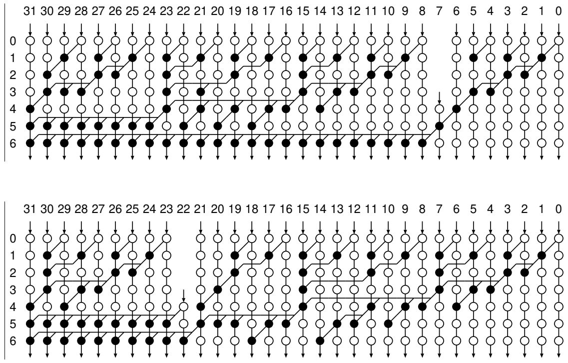


Figure 6.7: *Synthesized minimum-depth prefix structures (a), (b) for a single input bit late by 4 $\bullet$-delays.*

$D_\bullet$	$\#_\bullet$
5	74
(5)	(80)

(a)

$D_\bullet$	$\#_\bullet$
6	59
(5)	(80)

(b)

$D_\bullet$	$\#_\bullet$
7	55
(5)	(80)

(c)

$D_\bullet$	$\#_\bullet$
8	54
(5)	(80)

(d)

$D_\bullet$	$\#_\bullet$
12	50
(5)	(80)

(e)

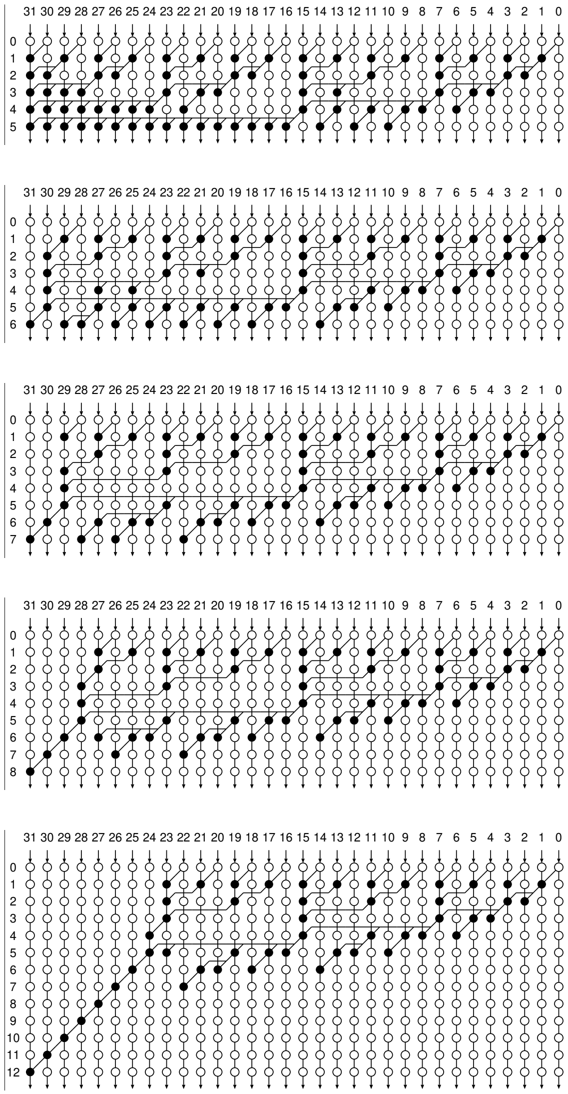


Figure 6.8: *Synthesized prefix structures (a)–(e) of depths 5–8 and 12.*

$D_\bullet$	$\#_\bullet$
13	50
(13)	(65)

(a)

$D_\bullet$	$\#_\bullet$
13	61
(13)	(80)

(b)

$D_\bullet$	$\#_\bullet$
13	73
(13)	(80)

(c)

$D_\bullet$	$\#_\bullet$
12	55
(12)	(80)

(d)

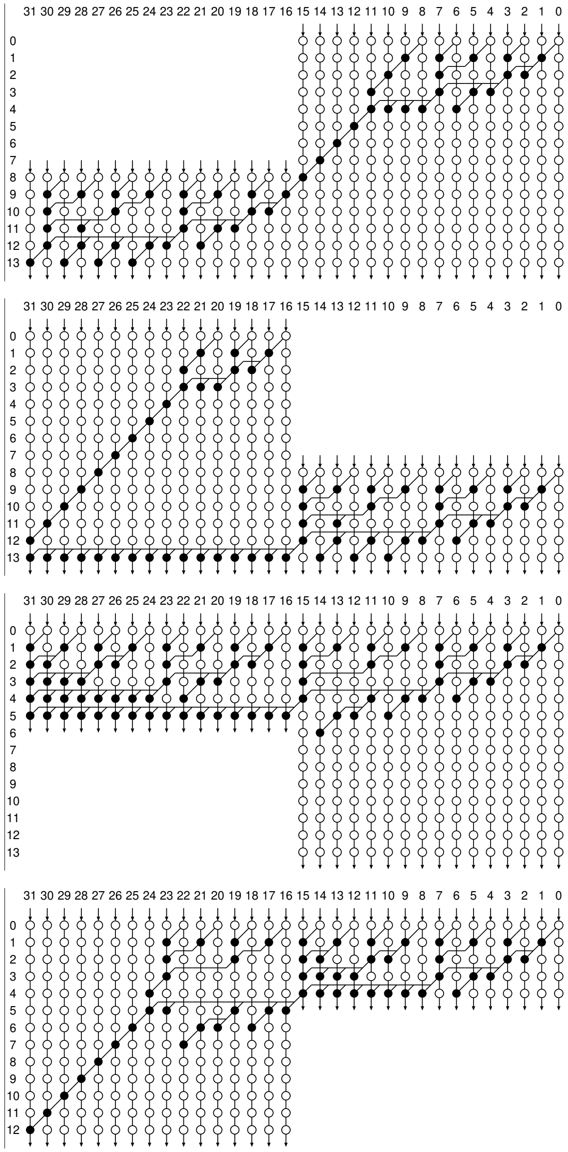
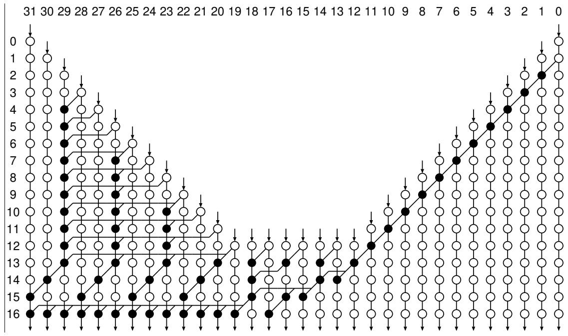


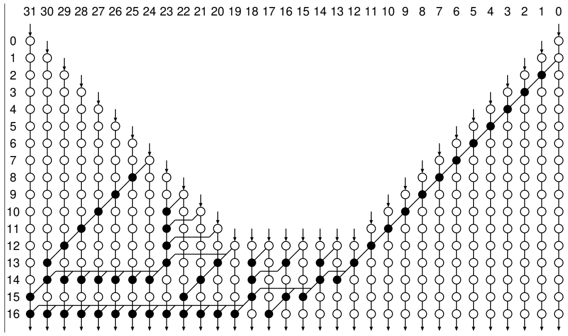
Figure 6.9: *Synthesized minimum-depth prefix structures for (a) late input upper word, (b) late input lower word, (c) early output upper word, and (d) early output lower word by 8 $\bullet$-delays.*

$D_\bullet$	$\#_\bullet$
16	65
(16)	(114)



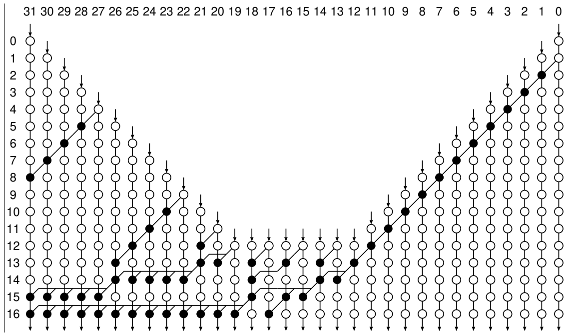
(a)

$D_\bullet$	$\#_\bullet$
16	56
(16)	(68)



(b)

$D_\bullet$	$\#_\bullet$
16	56
(16)	(57)



(c)

Figure 6.10: *Synthesized minimum-depth prefix structures with (a) no $\#^{\max}_{\bullet/b}$ bound, (b) $\#^{\max}_{\bullet/b} = 5$ $(= \log n)$ bound, and (c) $\#^{\max}_{\bullet/b} = 3$ bound for the typical input signal arrival profile in the final adder of a multiplier (steep slopes).*

$D_\bullet$	$\#_\bullet$
10	63
(10)	(71)

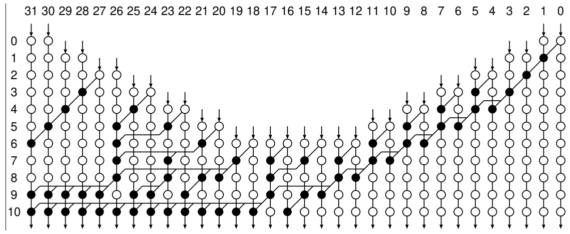


Figure 6.11: *Synthesized minimum-depth prefix structure with $\#^{\max}_{\bullet/b} = 5$ $(= \log n)$ bound for typical input signal arrival profile in the final adder of a multiplier (flat slopes).*

$D_\bullet$	$\#_\bullet$
11	51

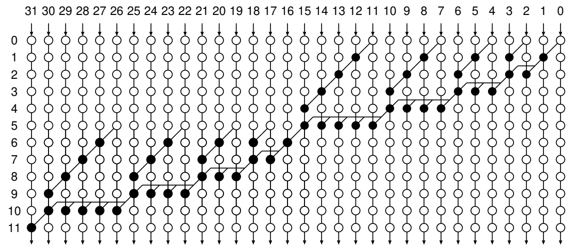


Figure 6.12: *Synthesized minimum-depth prefix structure with $\#^{\max}_{\bullet/b} = 2$ and $\#^{\max}_{\bullet/l} = 5$ bound (resource-constrained harmonic schedule for $n = 32$ and $p = 5$).*

6.5 Validity and Verification of Prefix Graphs

Obviously, not all directed acyclic graphs are valid prefix graphs. The validity of a prefix graph can be checked by some simple properties. In addition, valid prefix graphs exist which are redundant but can be converted into irredundant ones. Although the prefix graphs generated by the above synthesis algorithms are valid (i.e., correct-by-construction), this section gives the corresponding theoretical background and an algorithm for the verification of arbitrary prefix graphs.

6.5.1 Properties of the Prefix Operator

Associativity of the Prefix Operator

The addition prefix operator ($\bullet$) is *associative*:

$$Y^l_{i:j_1+1} \bullet (Y^l_{j_1:j_2+1} \bullet Y^l_{j_2:k}) \quad = \quad (Y^l_{i:j_1+1} \bullet Y^l_{j_1:j_2+1}) \bullet Y^l_{j_2:k} \qquad (6.1)$$

Proof: (note that for addition $Y^l_{i:k} \equiv (G^l_{i:k},\ P^l_{i:k})$)

$$\begin{aligned}
&(G^l_{i:j_1+1},\ P^l_{i:j_1+1}) \bullet ((G^l_{j_1:j_2+1},\ P^l_{j_1:j_2+1}) \bullet (G^l_{j_2:k},\ P^l_{j_2:k})) \\
&= (G^l_{i:j_1+1},\ P^l_{i:j_1+1}) \bullet (G^l_{j_1:j_2+1} + P^l_{j_1:j_2+1} G^l_{j_2:k},\ P^l_{j_1:j_2+1} P^l_{j_2:k}) \\
&= (G^l_{i:j_1+1} + P^l_{i:j_1+1}(G^l_{j_1:j_2+1} + P^l_{j_1:j_2+1} G^l_{j_2:k}),\ P^l_{i:j_1+1}(P^l_{j_1:j_2+1} P^l_{j_2:k})) \\
&= ((G^l_{i:j_1+1} + P^l_{i:j_1+1} G^l_{j_1:j_2+1}) + P^l_{i:j_1+1} P^l_{j_1:j_2+1} G^l_{j_2:k}, \\
&\qquad\qquad (P^l_{i:j_1+1} P^l_{j_1:j_2+1}) P^l_{j_2:k}) \\
&= (G^l_{i:j_1+1} + P^l_{i:j_1+1} G^l_{j_1:j_2+1},\ P^l_{i:j_1+1} P^l_{j_1:j_2+1}) \bullet (G^l_{j_2:k},\ P^l_{j_2:k}) \\
&= ((G^l_{i:j_1+1},\ P^l_{i:j_1+1}) \bullet (G^l_{j_1:j_2+1},\ P^l_{j_1:j_2+1})) \bullet (G^l_{j_2:k},\ P^l_{j_2:k}))
\end{aligned}$$

□

The associativity property of the prefix operator allows the evaluation of the prefix operators in any order. It is used in the prefix transformation applied for prefix graph optimization (Sec. 6.4).

Idempotence of the Prefix Operator

The addition prefix operator ($\bullet$) is *idempotent*:

$$Y^l_{i:k} \bullet Y^l_{i:k} \quad = \quad Y^l_{i:k} \qquad (6.2)$$

Proof:

$$\begin{aligned}
(G^l_{i:k},\ P^l_{i:k}) \bullet (G^l_{i:k},\ P^l_{i:k}) &= (G^l_{i:k} + \cancel{P^l_{i:k} G^l_{i:k}},\ P^l_{i:k} \cancel{P^l_{i:k}}) \\
&= (G^l_{i:k},\ P^l_{i:k})
\end{aligned}$$

□

The idempotence property of the prefix operator allows for insertion or removal of redundancy in prefix graphs. Note that the $\bullet$-operator in Eq. 6.2 is redundant and can be eliminated.

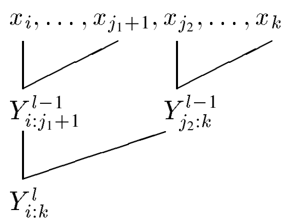
6.5.2 Generalized Prefix Problem

For validation of prefix graphs, it is important to understand the validity of group variables. A valid group variable $Y^l_{i:k}$ is the prefix result of all inputs $x_i, \ldots, x_k$. Let us rewrite Eq. 3.25 in a more generalized form, namely with the index j replaced by j_1 and j_2, as follows:

$$\begin{aligned}
Y^0_{i:i} &= x_i \\
Y^l_{i:k} &= Y^{l-1}_{i:j_1+1} \bullet Y^{l-1}_{j_2:k}\ ,\ k \le j_1, j_2 \le i\ ;\ l = 1, 2, \ldots, m \\
y_i &= Y^m_{i:0}\ ;\quad i = 0, 1, \ldots, n-1
\end{aligned} \qquad (6.3)$$

Three cases can now be distinguished:

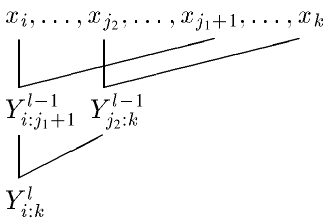
$j_1 = j_2$: Eq. 3.25 and Eq. 6.3 are equivalent. The bit groups represented by the group variables $Y^{l-1}_{i:j_1+1}$ and $Y^{l-1}_{j_2:k}$ are *adjacent*. $Y^l_{i:k}$ is a valid group variable.



$j_1 < j_2$: The bit groups represented by the group variables $Y^{l-1}_{i:j_1+1}$ and $Y^{l-1}_{j_2:k}$ are *overlapping* (i.e., bits $j_1+1, \ldots, j_2$ are covered twice by $Y^{l-2}_{j_2:j_1+1}$). $Y^l_{i:k}$ is a valid group variable, because the $\bullet$-operator is idempotent, i.e.:

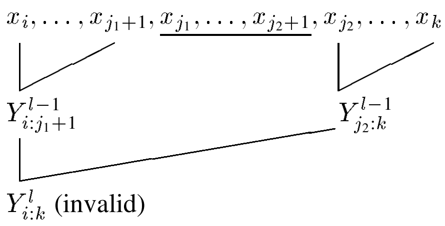
$$
\begin{aligned}
\underline{Y^l_{i:k}} &= Y^{l-1}_{i:j_1+1} \bullet Y^{l-1}_{j_2:k} \\
&= (Y^{l-2}_{i:j_2+1} \bullet \cancel{Y^{l-2}_{j_2:j_1+1}}) \bullet (Y^{l-2}_{j_2:j_1+1} \bullet Y^{l-2}_{j_1:k}) \\
&= Y^{l-2}_{i:j_2+1} \bullet (Y^{l-2}_{j_2:j_1+1} \bullet Y^{l-2}_{j_1:k}) \\
&= \underline{Y^{l-1}_{i:j_2+1} \bullet Y^{l-1}_{j_2:k}}
\end{aligned}
$$

Note, that a redundant $\bullet$-operation is performed.



$j_1 > j_2$: The bit groups represented by $Y^{l-1}_{i:j_1+1}$ and $Y^{l-1}_{j_2:k}$ are *not adjacent* (i.e., bits $j_1, \ldots, j_2+1$, or variable $Y^{l-2}_{j_1:j_2+1}$, are not covered). $Y^l_{i:k}$ is *not* a valid group variable, since

$$
Y^l_{i:k} = Y^{l-1}_{i:j_1+1} \bullet Y^{l-1}_{j_2:k} \neq Y^{l-1}_{i:j_1+1} \bullet Y^{l-2}_{j_1:j_2+1} \bullet Y^{l-1}_{j_2:k}
$$



6.5.3 Transformations of Prefix Graphs

From the above prefix graph and group variable properties, all elementary prefix graph transformations can now be summarized. They are depicted in Figure 6.13–6.17, with the indices ($i > j > k$ resp. $i > j_1 > j_2 > k$) to the right of each node denoting the index range of the corresponding group variable ($i{:}j_1$ actually denotes $Y^l_{i:j_1+1}$). The basic prefix transform operation used in Section 6.4 for prefix graph optimization bases on the associativity property of the prefix operator (Fig. 6.13). It works in both directions while validity and irredundancy of the graph is preserved. The shift operation of a single black node, which is the other transformation used in the presented optimization algorithm, also retains the validity and irredundancy properties of a graph (Fig. 6.14). The idempotence property of the prefix operator allows to merge two black nodes (Fig. 6.15). This is the only prefix graph transformation which removes redundancy from or inserts redundancy (by duplication of one black node) to a prefix graph, respectively, while validity is again preserved.

Swapping of two black nodes in the same column only results in a valid prefix graph if the graph contains some specific redundancy (Fig. 6.16a). This transformation applied to an irredundant prefix graph results in an invalid group variable (Fig. 6.16b). The same holds true for the relative shift of two black nodes depicted in Figure 6.17. I.e., valid prefix graphs are only obtained if they are redundant (Fig. 6.17a), since otherwise one group variable covers a too small range of bits and thus is invalid (Fig. 6.17b).

It can be shown that any valid redundant prefix graph can be converted into an irredundant one using the above graph transformations.

6.5.4 Validity of Prefix Graphs

The validity of a prefix graph can now be defined in several ways. A prefix graph is *valid* if and only if:

a) it computes its outputs according to Eq. 3.23,

b) it is functionally equivalent to the corresponding serial-prefix graph,

c) there exists a sequence of legal graph transformations that it can be derived from the corresponding serial-prefix graph,

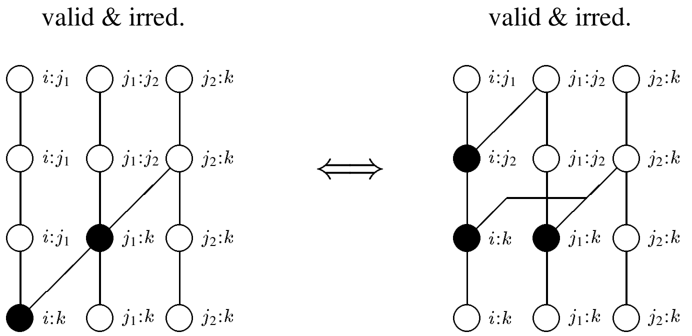


Figure 6.13: *Prefix transformation using associativity property.*

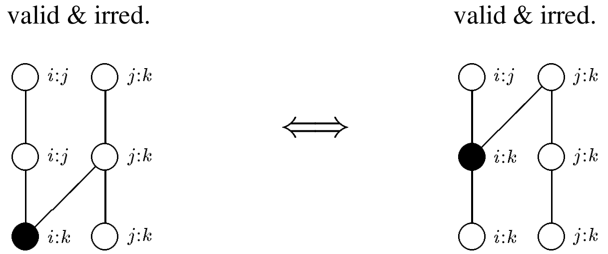


Figure 6.14: *Shift of single black node.*

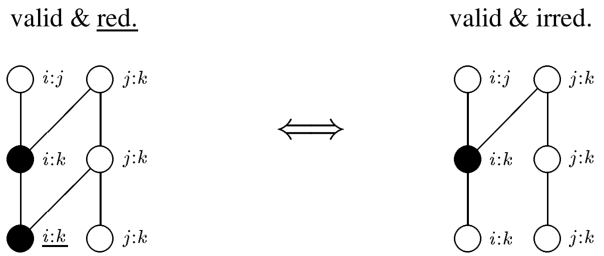


Figure 6.15: *Merging of two black nodes using idempotence (redundancy removal/insertion).*

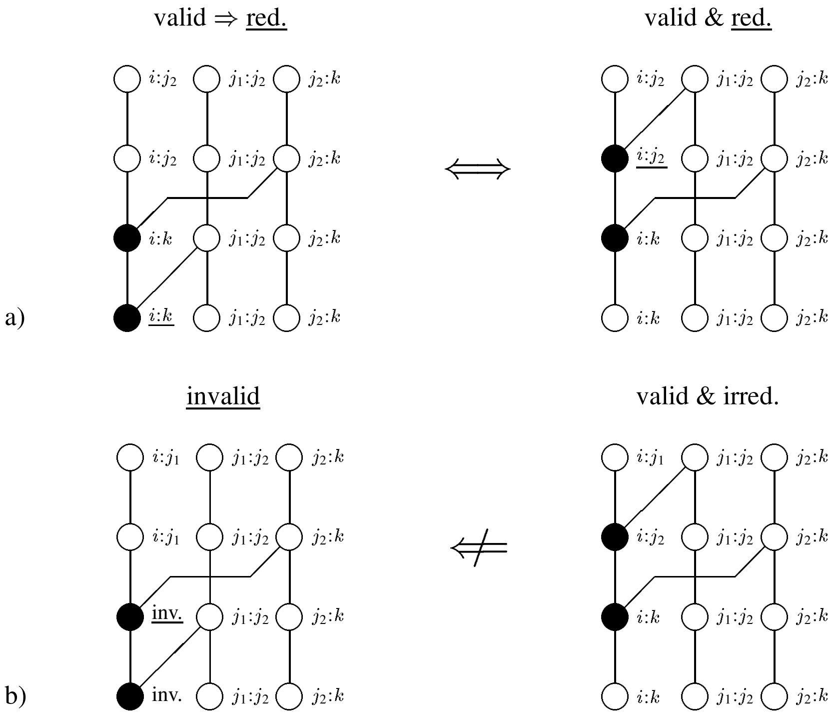


Figure 6.16: *(a) Legal (with redundancy) and (b) illegal (without redundancy) swapping of two black nodes in same column.*

d) it computes the group variables $Y^l_{i:k}$ according to Eq. 6.3 with $j_1 \leq j_2$, or

e) *at least one* path to group variable $Y^l_{i:k}$ (output $y_i = Y^m_{i:0}$) exists from each input $x_k, \ldots, x_i$ (i.e., group variable $Y^l_{i:k}$ is the (only) sink node of a directed acyclic graph with the inputs $x_k, \ldots, x_i$ as source nodes and an in-degree of two on all nodes).

6.5.5 Irredundancy of Prefix Graphs

A prefix graph is *valid* and *irredundant* if and only if:

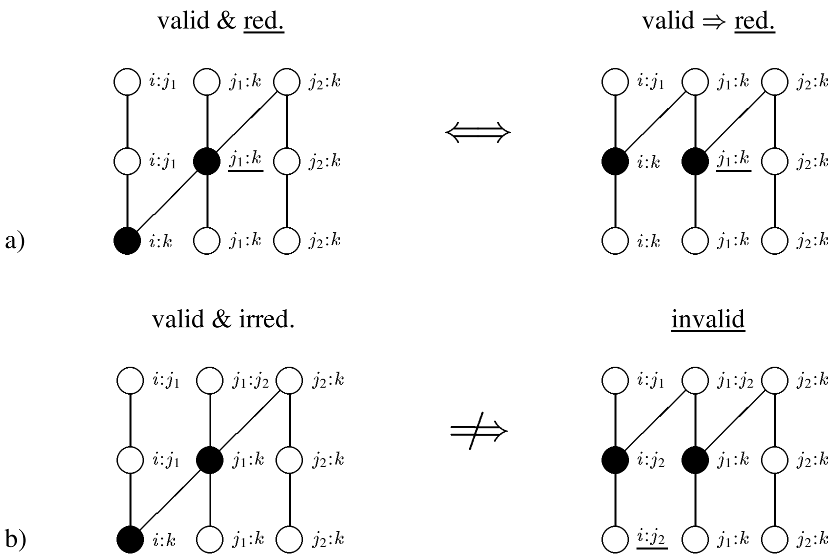


Figure 6.17: *(a) Legal (with redundancy) and (b) illegal (without redundancy) relative shift of two black nodes.*

a) it can be derived from the corresponding serial-prefix graph using the irredundancy preserving graph transformations of Figures 6.13 and 6.14 (corresponds to what the proposed prefix graph synthesis algorithm does),

b) it computes the group variables $Y^l_{i:k}$ according to Eq. 6.3 with $j_1 = j_2$, or

c) *exactly one* path to group variable $Y^l_{i:k}$ exists from each input $x_k, \ldots, x_i$ (i.e., group variable $Y^l_{i:k}$ is the root of a binary in-tree[5] with the inputs $x_k, \ldots, x_i$ as leaves, see Fig. 3.15).

Consequently, a prefix graph is *valid* and *redundant* if it computes at least one group variable $Y^l_{i:k}$ with $j_1 < j_2$.

[5] An *in-tree* is a rooted tree with reverse edge direction, i.e., edges lead from the leaves to the root.

Basically, redundant prefix graphs are of no importance since they offer no advantages while their irredundant counterparts are more efficient (e.g., smaller size).

6.5.6 Verification of Prefix Graphs

The prefix graphs synthesized by the presented algorithm can be regarded as *correct-by-construction* since only validity and irredundancy preserving graph transformations are applied. Thus, no verification is required. For graphs from another source, a verification procedure may be desirable.

From the above graph-theoretical conditions for valid and irredundant prefix graphs, a simple verification algorithm can be formulated. Its pseudo code is given below.

6.6 Summary

The regularity and implementation efficiency of the most common prefix structures allows the realization of relatively simple adder synthesis algorithms. Such netlist generators for fixed adder architectures can also be described in parameterized structural VHDL and thus be incorporated easily into hardware specification and synthesis.

The generality and flexibility of prefix structures proves to be perfectly suited for accommodating arbitrary depth constraints at minimum structure size, thereby allowing for an efficient implementation of custom binary adders. The universal algorithm described for optimization and synthesis of prefix structures is simple and fast, and it requires no heuristics and knowledge about arithmetic at all. It generates prefix structures that are optimal or near-optimal with respect to size under given depth constraints. It also works under other constraints, such as size and resource constraints.

Another approach for the generation of new adders using evolutionary algorithms (EA) was considered but not followed any further due to severe implementation problems [CL94].

Algorithm: Prefix graph verification

```
VERIFY_GRAPH () {
    valid = true;
    irredundant = true;
    for (l = 1 to m) {
        for (i = 1 to n - 1) {
            if (node (i, l) is black) {
                unmark all inputs;
                if (node (i, l) is at output) k = 0; else k = i;
                TRAVERSE_TREE (i, l);
                for (j = k to i) {
                    if (input j is not marked ) {
                        valid = false;
                        return;
                    }
                    else if (input j is marked more than once)
                        irredundant = false;
                }
            }
        }
    }
}

TRAVERSE_TREE (i, l) {
    k = min (i, k);
    if (node (i, l) at top of column i) {
        mark input i;
        return;
    }
    TRAVERSE_TREE (i, l - 1);
    if (node (i, l) is black and node (j, l - 1) its predecessor)
        TRAVERSE_TREE (j, l - 1);
}
```

7

VLSI Aspects of Adders

Some special aspects, which are related to VLSI design as well as to the implementation of parallel-prefix adders, are finally summarized in this chapter. In particular, circuit verification, transistor-level design, layout topologies, cell library requirements, pipelining, and FPGA implementations of prefix adders are investigated. This chapter impressively demonstrates the versatility of the parallel-prefix adder structure and its suitability for manifold applications.

7.1 Verification of Parallel-Prefix Adders

The importance of circuit verification is a matter of fact in VLSI design [CK94]. Functional circuit verification is used to validate the logic correctness of a circuit and its faultless fabrication. It is performed by simulating or testing a circuit with appropriate input stimuli against expected output responses. This section gives a simple and general test bench for the verification of all parallel-prefix adder circuits synthesized by the algorithms introduced in Chapter 6. Since prefix adders do not contain any logic redundancy (as opposed e.g. to the carry-skip adder), they are completely testable.

7.1.1 Verification Goals

Functional circuit verification by way of simulation and hardware testing must address the following two goals:

Logic verification: Circuits obtained from synthesis are usually regarded as logically correct-by-construction. Logical circuit verification through simulation, however, is still sometimes desired. The most crucial faults to be uncovered by a corresponding test bench are *gate faults* (false logic gates, e.g., an AND instead of a NAND) and *connection faults* (false circuit connections). False logic gates can be detected by applying all possible input combinations to each gate while making its output observable. Such a test is feasible for most circuits but is typically more expensive than a test for node faults only (node faults are explained below). On the other hand, a complete test for correct circuit connections is not feasible since this would require the simulation of all signal combinations on all possible circuit node pairs, which grows exponentially with the circuit size. However, a very high percentage of connection faults is usually covered by gate and node fault tests (see fabrication verification below).

Fabrication verification: A fabricated circuit may be affected by a variety of manufacturing defects. These defects are typically modeled by simplified *node fault* models, such as the *stuck-0/1* and the *open-0/1* models. A test covering these faults basically applies low and high levels to each circuit node while making the signals observable at the primary outputs.

A reliable circuit verification test bench for library components — such as the described parallel-prefix adders — should cover 100% of all possible faults under the above fault models.

7.1.2 Verification Test Bench

In addition, the following requirements for a verification test bench can be stated:

1. The test vector set should be kept as small as possible in order to save tester resources and reducing simulation and testing time.

2. The test vector set should be highly regular so that it can be generated algorithmically as a function of the word length or by a simple on-chip circuitry for efficient self test.

The test bench in Table 7.1 was developed for the verification of parallel-prefix adders. It was obtained by examining all faults for the various possible circuit structures, deriving the respective test vectors to cover them, and summarizing the vectors in simple and regular test vector sets with highly repetitive patterns. The test bench fulfills all of the above requirements. The number of test vectors is $4n + 4$, where n is the operand word length. The input patterns can be generated using a shift register and some multiplexers. The output responses can be compared with a pattern generated by another shift register and few multiplexers or by a signature checker. The test vector set guarantees 100% fault coverage under above (logic and fabrication) fault models with the exception of some hard-to-detect connection faults[6]. This holds true for all prefix adders — both AOI- and multiplexer-based — that are generated by the synthesis algorithms presented in Chapter 6. The test vector set was verified by fault simulations carried out using the System HILO software by Veda Design Automation Ltd.

7.2 Transistor-Level Design of Adders

So far, the design of adder circuits using cell-based techniques was addressed. When going down to the transistor level, new possibilities for circuit design are showing up. On one hand, various logic styles with varying performance characteristics exist for the implementation of logic gates. On the other hand, special circuit solutions exist at the transistor level for some arithmetic functions, such as the carry-chain or Manchester-chain circuit for the carry propagation in adders. Also, buffering and transistor sizing can be addressed at the lowest level.

A large variety of custom adder implementations exists and has been reported in the literature. A detailed discussion of transistor-level adder circuits is beyond the scope of this thesis, the main focus of which is on cell-based (or gate-level) design techniques. However, some conclusions from the cell-based investigations as well as the proposed adder architectures apply to the

[6]Some connection faults are very hard to detect and would required individual additional test vectors each.

Table 7.1: *Test bench for parallel-prefix adders.*

# vect.	A	B	c_{in}	S	c_{out}	coverage of
4	$000\cdots000$	$111\cdots111$	0	$111\cdots111$	0	special single faults
	$111\cdots111$	$000\cdots000$	0	$111\cdots111$	0	
	$000\cdots000$	$111\cdots111$	1	$000\cdots000$	1	
	$111\cdots111$	$000\cdots000$	1	$000\cdots000$	1	
n	$000\cdots000$	$111\cdots110$	1	$111\cdots111$	0	+ all stuck-0/1 faults for AOI-based adders
	$000\cdots000$	$111\cdots101$	1	$111\cdots110$	0	
	$000\cdots000$	$111\cdots011$	1	$111\cdots100$	0	
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
	$000\cdots000$	$110\cdots111$	1	$111\cdots000$	0	
	$000\cdots000$	$101\cdots111$	1	$110\cdots000$	0	
	$000\cdots000$	$011\cdots111$	1	$100\cdots000$	0	
n	$111\cdots111$	$000\cdots001$	0	$000\cdots000$	1	
	$111\cdots111$	$000\cdots010$	0	$000\cdots001$	1	
	$111\cdots111$	$000\cdots100$	0	$000\cdots011$	1	
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
	$111\cdots111$	$001\cdots000$	0	$000\cdots111$	1	
	$111\cdots111$	$010\cdots000$	0	$001\cdots111$	1	
	$111\cdots111$	$100\cdots000$	0	$011\cdots111$	1	
n	$000\cdots001$	$000\cdots001$	1	$000\cdots011$	0	+ all open-0/1 faults for AOI-based adders
	$000\cdots010$	$000\cdots010$	1	$000\cdots101$	0	
	$000\cdots100$	$000\cdots100$	1	$000\cdots001$	0	
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
	$001\cdots000$	$001\cdots000$	1	$010\cdots001$	0	
	$010\cdots000$	$010\cdots000$	1	$100\cdots001$	0	
	$100\cdots000$	$100\cdots000$	1	$000\cdots001$	1	
n	$111\cdots110$	$111\cdots110$	0	$111\cdots100$	1	+ remaining open-0/1 faults for mux-based adders
	$111\cdots101$	$111\cdots101$	0	$111\cdots010$	1	
	$111\cdots011$	$111\cdots011$	0	$111\cdots110$	1	
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
	$110\cdots111$	$110\cdots111$	0	$101\cdots110$	1	
	$101\cdots111$	$101\cdots111$	0	$011\cdots110$	1	
	$011\cdots111$	$011\cdots111$	0	$111\cdots110$	0	

transistor level as well. These aspects are shortly summarized in this section.

7.2.1 Differences between Gate- and Transistor-Level Design

In cell-based designs, circuits are composed of logic gates from a library. The set of available logic functions as well as the choice of gate drive strengths are limited. During circuit design and optimization, an optimal combination of the available logic gates and buffers is to be found for given circuit performance requirements.

At the transistor level, the designer has full flexibility with respect to the implementation of logic functions and to performance tuning. In particular, the following possibilities exist when compared to gate-level design:

- Gates for any arbitrary logic function can be realized
- Transistor sizing allows fine-tuning of gate performance (i.e., area, delay, power dissipation)
- Individual signal buffering allows fine-tuning of circuit performance
- Special circuit techniques and logic styles can be applied for
  - the implementation of special logic functions
  - the improvement of circuit performance
- Full layout flexibility yields higher circuit quality (especially area, but also delay and power dissipation)

Performance measures

For gate-level as well as for transistor-level circuits, performance comparisons by measuring or simulating actual circuit implementations make only sense if all circuits are realized under the same conditions and in the same technology. A direct comparison of performance numbers taken from different publications is not very reliable.

Analogously to the unit-gate model in cell-based design (Chap. 4), the *unit-transistor delay* τ [WE85] can be used for speed comparison of transistor-level circuits. In this model, the number of all transistors connected in series on the critical signal path is determined. The model accounts for the number of signal inversion levels and the number of series transistors per inversion level. It does not account for transistor sizes and wiring capacitances. Under the assumption that similar circuit architectures have similar critical signal paths with similar transistor sizes and wire lengths, the qualitative comparison of adder architectures using this delay model is adequate.

A more accurate qualitative comparison is possible for circuit size and power dissipation using the *transistor-count* model (analogous to the unit-gate model for cell-based circuits), where the total number of transistors of a circuit is determined.

7.2.2 Logic Styles

Different logic styles exist for the transistor-level circuit implementation of logic gates [Rab96]. The proper choice of logic style considerably influences the performance of a circuit.

A major distinction is made between *static* and *dynamic* circuit techniques. Dynamic logic styles allow a reduction of transistors and capacitance on the critical path by discharging pre-charged nodes through single transistor networks. The better speed comes at the cost of higher power consumption due to higher transition activities and larger clock loads. Also, dynamic logic styles are not directly compatible with cell-based design techniques and are not considered any further at this place.

Static logic styles can be divided into *complementary CMOS* and *pass-transistor logic*. While complementary CMOS is commonly used e.g. in standard cells, various pass-transistor logic styles — such as *complementary pass-transistor logic* (CPL) — were proposed for low-power applications. However, recent investigations showed complementary CMOS to be superior to pass-transistor logic if low power, low voltage, small power-delay products, and circuit area, but also circuit robustness are of concern [ZG96, ZF97].

7.2.3 Transistor-Level Arithmetic Circuits

Some special transistor-level circuits exist for adder-related logic functions.

Carry chain or Manchester chain: The Manchester chain is a transistor-level carry-propagation circuit, which computes a series of carry signals in a ripple-carry fashion using generate (Eq. (3.8)), propagate (Eq. (3.9)), and kill ($k = \neg(a + b)$) signals. It computes the carry with only three transistors per bit position (Fig. 7.1), compared to two gates in the cell-based version of Eq. (3.12). Note that the generation of the kill signal k requires some additional logic and that the length of a Manchester chain must be limited due to the number of transistors in series (i.e., typically 4 bits). The Manchester chain allows the area-efficient implementation of short ripple-carry chains and is typically used for the calculation of intermediate non-critical carry signals [LA95, M+94].

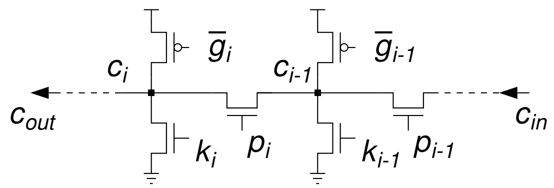


Figure 7.1: *Transistor-level carry-chain circuit.*

Pass-transistor/pass-gate multiplexer: Carry-select, conditional-sum, and also one variant of parallel-prefix adders consist of multilevel multiplexer structures. Such series multiplexers can efficiently be implemented using pass-transistor or pass-gate (transmission-gate) circuits. Hence, multiplexer-based adder architectures, which showed inferior performance in cell-based design due to inefficient multiplexer gates, yield better circuits at the transistor-level.

Full-adder: Special transistor-level circuits exist for full-adders, which differ from the typical implementations using simple gates (Fig. 3.4). One of the most efficient solutions in complementary CMOS logic style is depicted in Figure 7.2. Other solutions exist in pass-transistor logic styles, which are discussed in [ZF97].

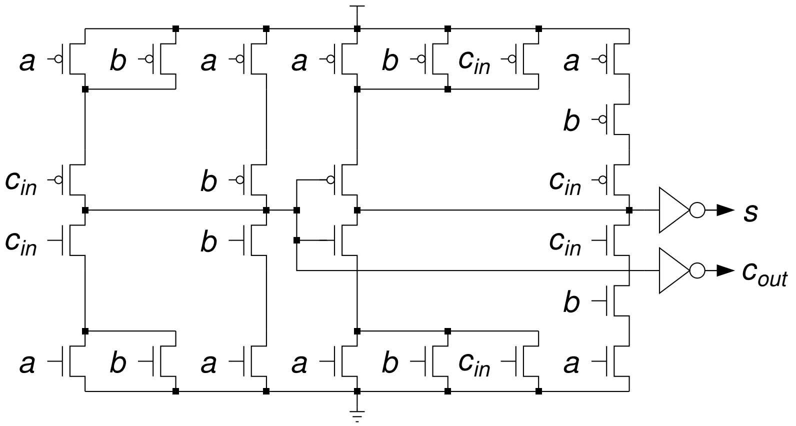


Figure 7.2: *Transistor-level full-adder circuit.*

7.2.4 Existing Custom Adder Circuits

Many different transistor-level adder implementations have been reported in the literature. Basically, all of them rely on the adder architectures described in Chapter 4. However, combinations of different speed-up techniques are often used which, at the transistor level, yield performance advantages by applying dedicated transistor-level arithmetic circuits (as described above) and circuit techniques.

Many custom adder implementations use a carry-lookahead adder architecture for the computation of some carries and a Manchester chain for the remaining intermediate carries [O+95]. Alternatively, short ripple-carry adders are used for the calculation of intermediate carries and pairs of sum bits, which are then selected by a subsequent carry-select stage [Lo97, M+96, DB95, G+94, LS92]. Conditional-sum architectures were used in [BDM95, AKY94], carry-skip in [Hob95], and parallel-prefix architectures in [KOIH92, S+94]. A combination of Manchester chain, carry-select, and conditional-sum adder was realized in [D+92]. Some adder architectures were compared in [NIO96].

Solutions with long series transistor chains (e.g., Manchester chain, series of pass-transistor multiplexers) are difficult to compare without simulating or even measuring actual circuit implementations. All other solutions can be

compared qualitatively using the transistor-delay model described above.

7.2.5 Proposed Custom Adder Circuit

The goal of custom adders is usually highest possible performance, i.e., circuit speed. Therefore, the fastest cell-based adder architecture from the presented comparisons, the Sklansky parallel-prefix adder, was investigated with respect to a transistor-level implementation. Its parallel-prefix stage consisting of •-operators (1 AND-OR-gate + 1 AND-gate, see Eq. 3.28) can be realized very efficiently in complementary CMOS logic style using and-or-invert/or-and-invert (AOI/OAI) and NAND/NOR gate combinations. The few nodes with high fan-out can be decoupled from the critical signal path by inserting one level of buffers into the prefix structure, as depicted in Figure 7.3.

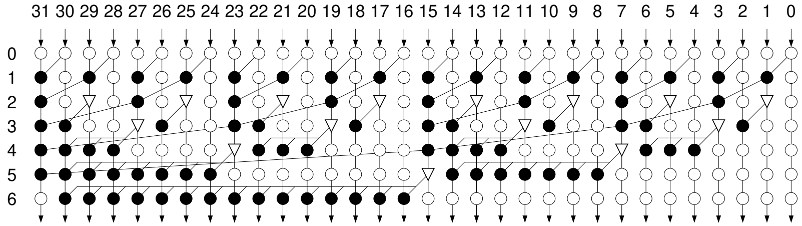


Figure 7.3: *Buffered Sklansky parallel-prefix structure.*

The resulting *buffered Sklansky parallel-prefix adder* circuit has minimum number of transistor delays and minimum node capacitances on the critical path. If compared qualitatively (i.e., transistor-delays, transistor-counts), this adder performs as well as the best custom adders reported in the literature. A 32-bit version has been implemented at the transistor level in a 0.5 μm process. The 1 607 transistors circuit has been simulated at worst-case conditions (2.8 V, 110° C, @ 100 MHz): worst-case delay is 4.14 ns at an average power dissipation of 7.5 mW. Quantitative adder comparisons are not made at this point since no other custom implementations were realized as part of this work.

7.3 Layout of Custom Adders

In custom adder implementations, layout regularity and topology are of major importance for circuit area and performance efficiency. That is, an array-like, parameterizable arrangement of a few simple layout cells with only few interconnections and a small amount of unused area slots would be ideal.

All the presented — and especially the Sklansky — parallel-prefix structures are highly regular. Their graph representations can directly be mapped into a layout topology, resulting in an array of black and white layout cells with only very few wires routed through them. The same holds true for the buffered Sklansky prefix structure.

Note that half of the nodes in the Sklansky prefix structure are white, thus containing no logic. Since they occupy the same area as the white nodes for regularity reasons, half the area is wasted. As can be seen in Figure 7.4, a 2^k-bit wide prefix structure can be divided into two $2^{(k-1)}$-bit parts which have an antisymmetric structure (i.e., they are symmetric with respect to the drawn axis, if the black and white nodes are exchanged). Therefore, the left part of the structure can be mirrored and overlaid over the right part, filling out all white node locations with black nodes. The resulting folded structure is a $(n/2) \times \log n$ array of identical black nodes with still modest and regular wiring, and thus is highly area-efficient and layout-friendly. The same structure folding can be applied to the buffered Sklansky prefix structure (Fig. 7.5). Note that for the folded structures, the high order bits are in reverse order and interleaved with the low order bits. Whereas this is no problem for a macro-cell implementation (i.e., the bit order in surrounding routing channels is of minor importance), it is less convenient for data-path (i.e., bus-oriented) applications.

Other prefix structures with high layout efficiency and regularity are the serial-prefix and the 1- and 2-level carry-increment parallel-prefix structures (Figs. 7.6–7.8). They are well suited for data-path implementations, since they are composed of regular and small bit-slices and bits are in ascending order (see also Section 6.3). Note that also the 2-level carry-increment prefix structure can be arranged in a way that each column counts no more than two black nodes.

For these layout topologies, the unit-gate area model used in Chapter 4 allows very accurate area estimations, since the amount of unused circuit area

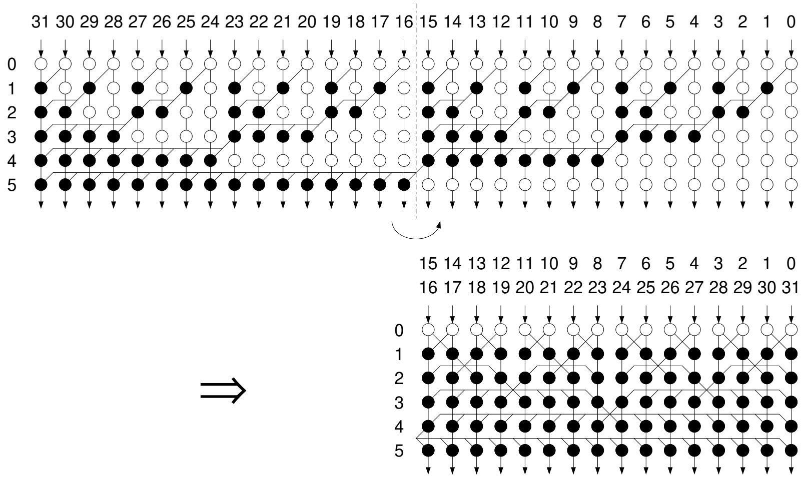


Figure 7.4: *Folded Sklansky parallel-prefix structure.*

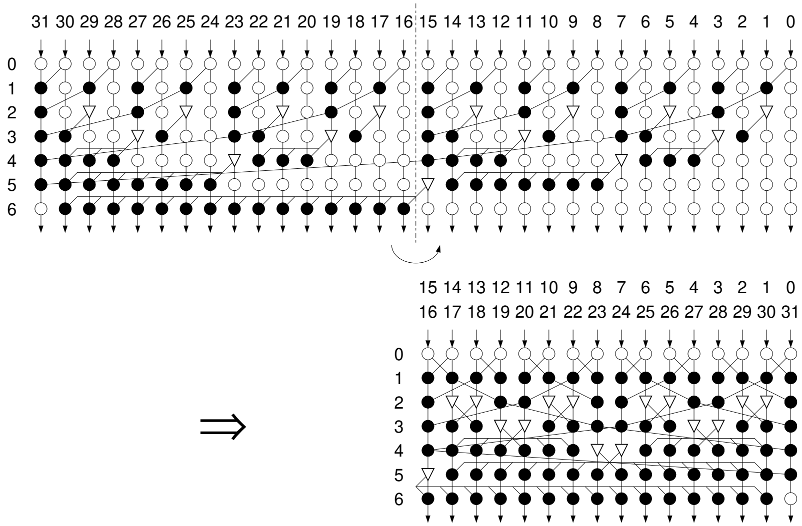


Figure 7.5: *Folded buffered Sklansky parallel-prefix structure.*

and wiring is negligible.

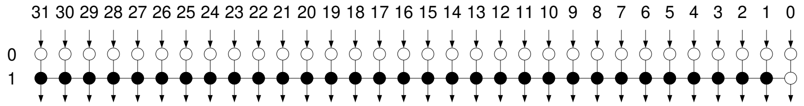


Figure 7.6: *Serial-prefix structure.*

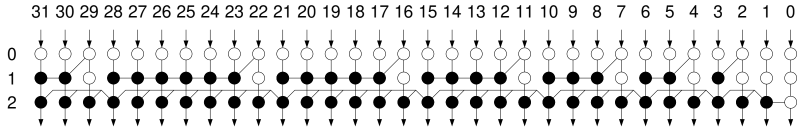


Figure 7.7: *Compacted 1-level carry-increment parallel-prefix structure.*

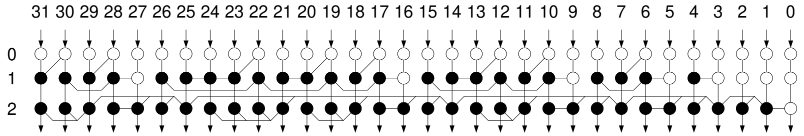


Figure 7.8: *Compacted 2-level carry-increment parallel-prefix structure.*

7.4 Library Cells for Cell-Based Adders

So far, we have addressed the implementation of parallel-prefix adders using either standard-cell libraries or by doing custom design. But what about cell-based design with custom cells? What cells should a standard-cell library contain in order to achieve highest adder circuit performance?

7.4.1 Simple Cells

As we have seen in Chapter 3, the gate-level specification of a parallel-prefix adder basically makes use of XOR, AND-OR, and AND gates (Eqs. 3.27–3.29). During automatic circuit optimization, series of AND-OR resp. AND gates are typically replaced by a series of alternating AOI and OAI resp. NAND and NOR gates. I.e., faster inverting gates are used so that all output signals of an odd prefix stage are inverted. Additionally, buffers and inverters are used for appropriate signal buffering and fan-out decoupling. All these cells are typically provided in any standard-cell library.

7.4.2 Complex Cells

Two complex cells, which are also included in most cell libraries, can be used for a more efficient implementation of prefix adders.

Majority gate: A majority gate implements Eq. 3.30 directly as one inverting gate. It can be used for the fast computation of the first generate signal in a carry chain (if a carry-in is present), since its delays $a_0 \rightarrow g_0$ and $b_0 \rightarrow g_0$ are shorter than in a typical full-adder structure (Fig. 3.4).

Full-adder: A single full-adder cell from the library is typically more efficient with respect to area and delay than an implementation using simpler gates. This is due to the efficient transistor-level circuits used in full-adder cells. However, entire full-adders are only used in ripple-carry (or serial-prefix) but not in parallel-prefix adders, where functionality is split into preprocessing, parallel-prefix computation, and post-processing. Of course, full-adders can be used in the serial-prefix part of a mixed serial/parallel-prefix adder presented in Sections 5.3 and 6.4.

No other special cells are required for the efficient realization of serial- or parallel-prefix adders. Thus, prefix adders are highly compatible with existing standard-cell libraries.

Synthesis and technology mapping

In order to take full advantage of the mixed serial/parallel-prefix adders synthesized under relaxed timing constraints by the algorithm presented in Section 6.4, full-adder cells should be used during technology mapping for the serial-prefix part. This can be accomplished easily if the adder synthesis algorithm is incorporated into an entire synthesis package, where circuit synthesis and technology mapping typically work hand in hand. However, if the synthesis of the prefix adder structure is performed outside the circuit synthesis tool (e.g., by generating intermediate structural VHDL code), a tool-specific approach must be taken (e.g., by instantiating full-adder cells explicitly, or by letting a ripple-carry adder be synthesized for the corresponding range of bits).

7.5 Pipelining of Adders

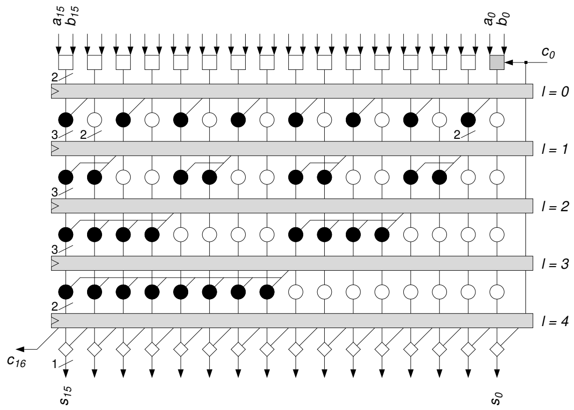
In this work, the fastest adder architectures were evaluated for combinational circuit implementations. However, if throughput requirements are not met by the fastest combinational realization, *pipelining* can be applied in order to increase throughput at the cost of increased latency. With respect to pipelining of adder circuits, the following aspects must be considered:

- High *regularity* of an adder structure simplifies the insertion of pipeline registers.
- Basically, every adder (i.e., every combinational circuit) can be made faster (i.e., to run in circuits clocked at higher frequencies) using pipelining. However, the kind of adder architecture used for pipelining strongly influences the amount of storage elements required and the circuit's latency (i.e., length of the pipeline).
- The number of *internal signals*, which have to be stored in a pipeline register, determine the register size. An adder architecture should have as few internal signals as possible in order to minimize the number of pipeline storage elements per stage.
- Faster adders require less *pipeline stages* for a requested throughput. An adder architecture should be as fast as possible (i.e., minimal logic depth) in order to minimize the number of pipeline registers. Note that on the other hand, fast adders tend to have more internal signals (due to higher parallelism), which again increases register size.
- If some *latency constraints* — and with that the maximum number of pipeline stages — are given, then a fast adder architecture may be required in order to fulfill cycle time requirements.

Pipelining of prefix adders

Pipelining of prefix adders is very *straightforward*, since they are composed of stages (pre-processing stage, several prefix levels, and post-processing stage) with a logic depth of only two unit gates each (i.e., one XOR resp. AOI-gate). Therefore, $m+1$ locations exist for placing pipeline registers, where m is the number of prefix levels. This enables the realization of pipeline stages of arbitrary size and allows for *fine-grained pipelining*. Fine-grained pipelined Sklansky parallel-prefix and serial-prefix adders are depicted in Figures 7.9 and 7.10, respectively, while Figure 7.11 shows a medium-grained pipelined Brent-Kung parallel-prefix adder. Qualitative performance measures are given for cycle time ($T_\bullet$), black node ($A_\bullet$) and flip-flop area (A_{FF}), latency (L), and maximum fan-out (FO_{max}). As can be seen, selecting a good adder structure as starting point pays off.

The number of internal signals in prefix adders is rather high (i.e., up to three signals per bit position in the prefix-computation stage), resulting in large pipeline registers. It can be reduced to some degree – especially in the serial-prefix adder — by shifting up the post-processing operators $\diamond$ as far as possible. The resulting flip-flop area numbers are given in parenthesis.



$$\begin{aligned} T_\bullet &= 1 \\ A_\bullet &= 32 \\ A_{FF} &= 188 \\ &\quad (173) \\ L &= 5 \\ FO_{max} &= \log n \end{aligned}$$

Figure 7.9: *Fine-grained pipelined Sklansky parallel-prefix adder.*

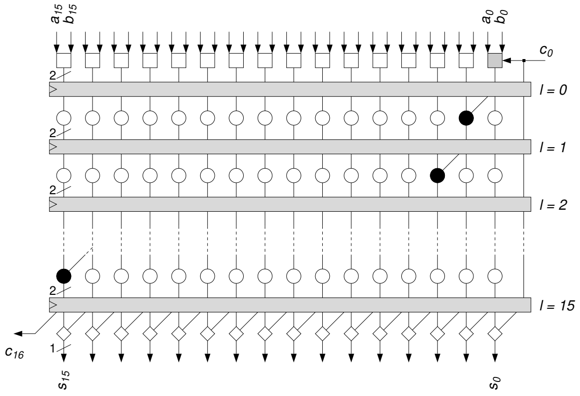
Pipelining of other adders

With respect to the number of internal signals and thus the size of the pipeline registers, other adder architectures do not yield better results. The lower regularity of some of them makes the insertion of pipeline registers considerably more difficult.

Another approach was proposed in [DP96], where small carry-lookahead adders are inserted between pipeline registers in order to increase throughput and decrease latency of a medium-grain pipelined ripple-carry adder (corresponds to the structure of Fig. 7.12). This solution, however, is not competitive if compared to pipelined parallel-prefix adders, because its global prefix structure is not optimal (compare Figs. 7.11 and 7.12, where cycle time and maximum fan-out are equal but size and latency of the latter much larger). This especially holds true for larger word lengths, where the number of pipeline registers grows logarithmically for the parallel-prefix adder, but linearly for the adder architecture of [DP96].

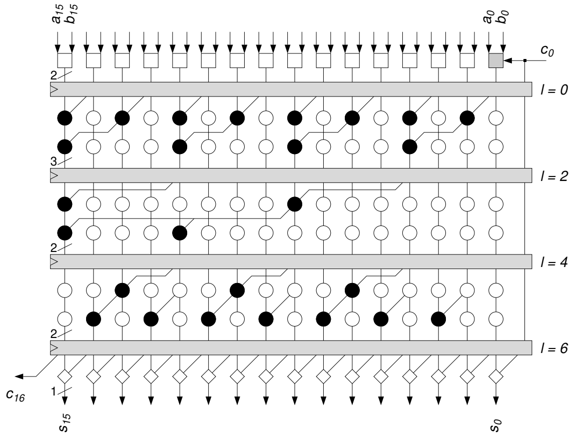
7.6 Adders on FPGAs

Cell-based design techniques are also used for the realization of circuits on *field programmable gate arrays* (FPGAs). Here, the set of available gates, or logic functions respectively, varies considerably between different FPGA architec-



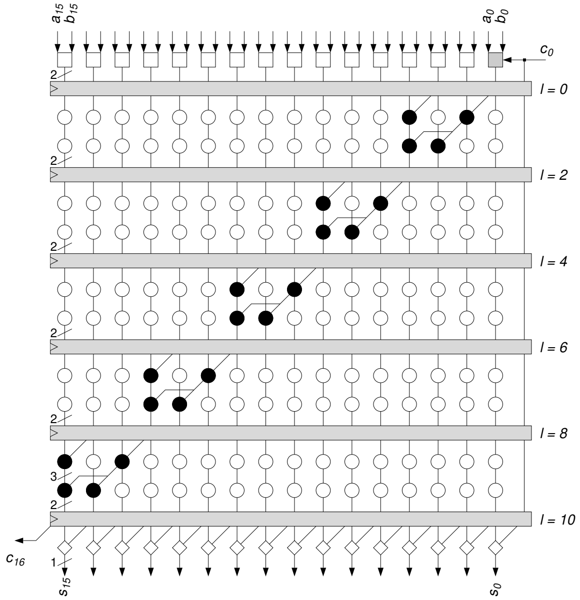
$$
\begin{aligned}
T_\bullet &= 1 \\
A_\bullet &= 16 \\
A_{FF} &= 528 \\
&\quad (400) \\
L &= 16 \\
FO_{max} &= 1
\end{aligned}
$$

Figure 7.10: *Fine-grained pipelined serial-prefix adder.*



$$
\begin{aligned}
T_\bullet &= 2 \\
A_\bullet &= 26 \\
A_{FF} &= 141 \\
&\quad (127) \\
L &= 4 \\
FO_{max} &= 2
\end{aligned}
$$

Figure 7.11: *Medium-grained pipelined Brent-Kung parallel-prefix adder.*



$$
\begin{aligned}
T_\bullet &= 2 \\
A_\bullet &= 20 \\
A_{FF} &= 198 \\
&\quad (158) \\
L &= 6 \\
FO_{max} &= 2
\end{aligned}
$$

Figure 7.12: *Medium-grained pipelined serial-prefix adder with parallel-prefix stages.*

tures and granularities. In particular, fine-grained FPGAs are comparable to other cell-based technologies, such as standard cells. Some investigations on the realization of adders on fine-grained FPGAs have been carried out and are summarized here.

7.6.1 Coarse-Grained FPGAs

Coarse-grained FPGAs — such as the members of the Xilinx XC4000 and Altera FLEX8000 families — are composed of logic blocks with about four or more inputs. Their output(s) are computed using look-up tables (LUT), which allow quite complex logic functions per single logic block. The adder circuits presented cannot be mapped directly onto such complex logic blocks. On the contrary, adder architectures and circuits have to be adapted in order to take full advantage of the corresponding logic block resources. This also implies the usage of macros for the implementation of library components (such as adders) rather than the more universal approach using gate-level synthesis/optimization and technology mapping. Furthermore, the inherent large logic depth of the complex logic blocks disallows for the implementation of fast carry chains. Therefore, most coarse-grained FPGAs include an extra fast-carry logic. This fast-carry logic makes ripple-carry adders — made accessible through vendor-specific soft- or hard-macros — the best choice for all but very large word lengths.

Due to these incompatibilities between the prefix adder architectures and the coarse-grained FPGA structures and design techniques, no further investigations were done in this direction.

7.6.2 Fine-Grained FPGAs

Fine-grained FPGAs — such as the members of the Xilinx XC6200 and Atmel AT6000 families — typically allow the realization of an arbitrary 2-input gate or a 2-input multiplexer per logic cell. Since this logic cell complexity is comparable to the complexity of standard cells, standard gate-level circuits and synthesis techniques can be used. At this granularity, the presented adder architectures again exploit their full potential, and the absence of dedicated fast-carry logic makes their application mandatory for efficient circuit implementations.

Differences between fine-grained FPGAs and standard cells

The basic differences between fine-grained FPGAs and custom cell-based technologies (such as standard cells) are:

- In standard-cell technologies, *AND / OR gates* perform better (area and speed) than *AND-OR / OR-AND gates*, which in turn perform better than *multiplexers*. On fine-grained FPGAs, AND / OR gates and multiplexers have the same performance, since both are implemented by one logic cell. Thus, AND-OR / OR-AND gates require two logic cells, which makes them much less efficient than multiplexers. Put differently, multiplexers are the only two-level logic functions which can be realized in one logic cell. While AND-OR / OR-AND gates are preferred over multiplexers in standard-cell technologies, the opposite holds true for FPGAs. As worked out in Section 3.5, the prefix circuit of an adder can be realized using AND-OR gates (Eqs. 3.27–3.29) or multiplexers (Eqs. 3.32–3.34). Therefore, on FPGAs the multiplexer-based prefix adder structure is the better choice.

- As opposed to standard-cell technologies, where *routing resources* are almost unlimited (i.e., routing channels can be made as wide as required), routing resources on FPGAs are very limited. The amount of wiring compared to the number of logic cells as well as the proper placement of the logic cells are crucial for the routability of a circuit. Also, the routing of placed cells using the limited wiring resources is very difficult, and software algorithms for automated routing are still a challenge. For the efficient implementation of library components, circuit architectures have to be chosen which provide a good balance between wiring and logic complexity.

- Finally, circuit *regularity* of library components is of major importance on FPGAs with respect to layout generation, layout efficiency, and routability. *Bit-sliced layout* techniques are helpful for combining several components to form entire data paths, thus limiting the amount of area-intensive inter-block routing.

Optimal adder architectures for fine-grained FPGAs

As mentioned above, adders for fine-grained FPGAs should be highly regular, have low wiring requirements, and allow for a bit-sliced layout implementation. We can conclude from the adder architecture comparisons of Section 4.2 that the ripple-carry adder (low speed), the carry-skip adder (medium speed) and the 1-level carry-increment adder (high speed) perfectly fulfill the above requirements. The 2-level carry-increment adder has more complex wiring and thus is more difficult to implement. All other adder architectures — such as parallel-prefix, carry-lookahead, carry-select, and conditional-sum adders — lack efficient realizations on fine-grained FPGAs due to low regularity and high routing complexity.

Note that the unit-gate model used in the qualitative adder architecture comparisons of Chapter 4 yield very accurate performance estimations for circuits implemented on fine-grained FPGAs. This is because each logic cell exactly implements one simple 2-input gate — with the exception of the multiplexer — and has a roughly constant delay. However, wiring (including the costly routing switches) must be considered as well as it contributes a considerable amount to the overall circuit delay (more than e.g. for standard cells).

Adder implementations on a Xilinx XC6216

For validation of the above qualitative evaluation, a ripple-carry and a 1-level carry-increment adder were realized on a Xilinx XC6216 FPGA [Xil97, Mül97]. The basic XC6216 logic cell implements any arbitrary 2-input gate or a 2-input multiplexer plus one flip-flop. It has one input and one output connector to the neighbor cell on each side. A hierarchical routing scheme allows the efficient realization of long distance connections. The XC6216 contains $64 \times 64 = 4096$ logic cells.

A bit-slice layout strategy was used for the circuit realizations with a pitch of two logic cells per bit.

Ripple-carry adder (RCA): The ripple-carry adder consists of a series of full-adders (Fig. 7.13). Each full-adder requires three logic cells but occupies $2 \times 2 = 4$ logic cells for regularity reasons. The total size of a ripple-carry adder is therefore $4n$ logic cells.

1-level carry-increment adder (CIA-1L): As depicted in Figure 7.14, the 1-level carry-increment adder is made up of three different slices (A, B, and C), each of them occupying six logic cells (only two more than the full-adder). Long wires (i.e., wires ranging over 4 logic cells) are used for propagating the block carries. Slice B and C are equivalent except for the carry-out, which in slice C is the block carry-out connected to a long wire. The size of a 1-level carry-increment adder is $6n$ logic cells.

Circuit sizes and delays (from static timing verification) for both adder implementations are given in Table 7.2 (taken from [Mül97]). Although the 1-level carry-increment adder does not implement the fastest existing architecture for standard cells, its proposed FPGA implementation can hardly be beaten since the small speed advantage of other adder architectures is undone by their less efficient layout and routing. To be more specific, other adder architectures — such as the parallel-prefix and carry-select adders — need to propagate three intermediate signals per bit position, which can only be accomplished by increasing the pitch from two to three logic cells per bit (i.e., 50% more area) or by using also the longer wires of the secondary interconnection level (i.e., much larger pitch in the second dimension).

Table 7.2: *Comparison of adder implementations on a XC6216.*

# bits	area (# logic cells)		delay (ns)	
	RCA	CIA-1L	RCA	CIA-1L
4	16	24	16.2	19.4
8	32	48	29.1	25.7
16	64	96	54.9	34.1
32	128	192	106.5	44.7

These adder implementations demonstrate the importance of proper cell placement for routability. In the case of the carry-increment adder, only two cell placements within a bit-slice were found which were routable at all. More complex circuits and adder architectures are only routable if more empty cells are inserted and the pitch per bit is increased. Note that only one signal can be exchanged between two neighboring cells in each direction, i.e., only two signals per bit position can propagate in parallel through a circuit with a bit-slice pitch of two logic cells.

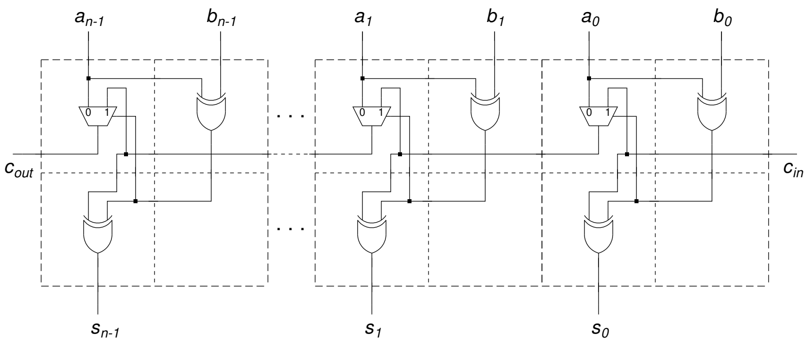


Figure 7.13: *Ripple-carry adder on a XC6216.*

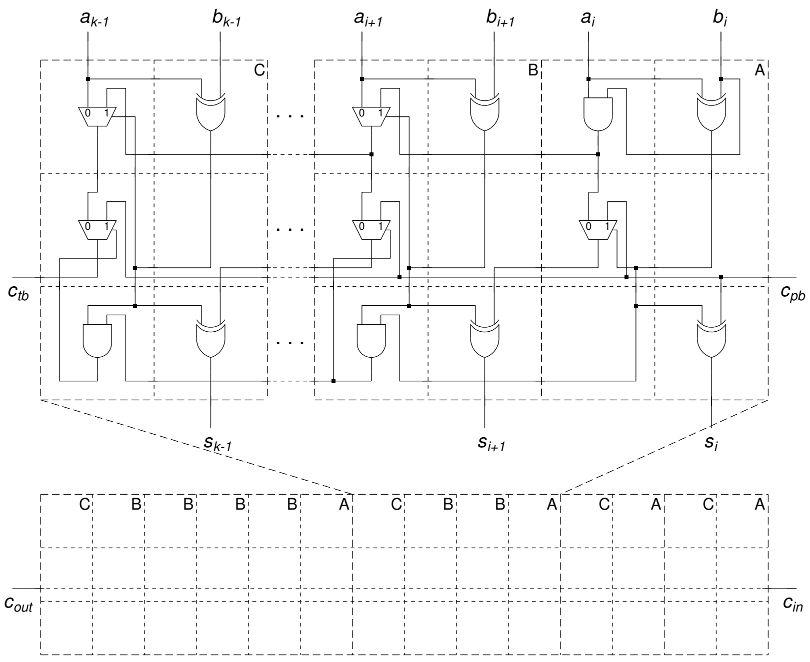


Figure 7.14: *1-level carry-increment adder on a XC6216.*

8

Conclusions

Binary adder architectures for cell-based design and their synthesis have been investigated in this thesis. The research items and results of this work can be summarized as follows:

- The basic addition principles and speed-up schemes for the carry-propagate addition of two binary numbers have been assembled.

  ⇒ A comprehensive and consistent overview of the existing adder architectures is given.

  ⇒ A new multilevel carry-increment adder architecture with excellent performance characteristics has been developed and proposed.

- Qualitative and quantitative comparisons of adder architectures for cell-based design have been carried out.

  ⇒ All adder architectures are characterized with respect to circuit speed, area and power requirements, and suitability for cell-based design and synthesis.

  ⇒ The ripple-carry, carry-increment, and the carry-lookahead adders show the best characteristics in all respects and fill the entire range of possible area-delay trade-offs.

- The parallel-prefix scheme reported in the literature was found to represent *the* universal adder architecture.

  ⇒ It provides a universal and consistent description of all well-performing adder architectures — i.e., ripple-carry, carry-increment, and carry-lookahead adders — and summarizes them in the class of prefix adders.

  ⇒ Its flexibility allows the efficient and simple realization of various special adders.

  ⇒ A simple local prefix transformation enables the optimization of prefix adders for speed or area or both.

  ⇒ Prefix circuits can be generated by simple algorithms and thus be described in parameterized structural VHDL.

  ⇒ Prefix adders allow for simple circuit verification, efficient transistor-level design and layout topologies, and simple pipelining.

- A fast non-heuristic optimization and synthesis algorithm has been developed for prefix graphs.

  ⇒ A universal algorithm exists for the synthesis of all prefix adders.

  ⇒ The runtime-efficient synthesis of area-optimal adders for the entire range of area-delay trade-offs and for arbitrary timing constraints is possible.

In addition, some important observations and experiences have been made:

- Smaller circuits typically also provide a speed advantage compared to larger ones, even if logic depth is the same. This is due to the smaller interconnect delays of shorter wires, which becomes an even more important performance parameter in deep-submicron VLSI. Also, smaller circuits are more power-efficient.

- Adder architectures are a striking example illustrating the possibility for trading off area versus delay in circuit design.

- Collecting and comparing different solutions to a problem at the conceptual level gives a better understanding and more abstract view of the underlying principles. On this basis, more reliable characterization and performance comparison of existing solutions is possible and new solutions can be found.

- Universal structure representations, such as the parallel-prefix scheme for binary addition, often provide a consistent description of efficient and flexible circuit structures and allow their synthesis by simple algorithms. Graph representations in particular are very effective to that respect.

Finally, the following outlook and topics for future work can be formulated:

- The qualitative results presented in this thesis are expected to be valid also in future deep-submicron VLSI technologies. While such technologies have a high impact on large circuits with long interconnects, the rather small cell-based adder circuits treated here are qualitatively affected only marginally by the expected technology parameter changes.

- With the increasing number of metalization levels in modern process technologies, cell-based designs almost become as layout-efficient as custom designs. This, together with the productivity increase, will make cell-based circuit design and synthesis more and more important in the future.

- The comparison between cell-based and custom adder implementations in a modern submicron technology (e.g., 0.35μm, four-metal) would be a valuable extension of this thesis.

- The desirable continuation of this work would be the integration of the proposed adder synthesis algorithm into a commercial hardware synthesis tool.

Bibliography

[AKY94] I. S. Abu-Khater and R. H. Yan. A 1-V low-power high-performance 32-bit conditional sum adder. In *Proc. 1994 IEEE Symp. Low Power Electron.*, pages 66–67, San Diego, October 1994.

[BDM95] B. Becker, R. Drechsler, and P. Molitor. On the generation of area-time optimal testable adders. *IEEE Trans. Computer-Aided Design*, 14(9):1049–1066, September 1995.

[BK82] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Comput.*, 31(3):260–264, March 1982.

[C$^+$94] S. W. Cheng et al. The role of long and short paths in circuit performance optimization. *IEEE Trans. Computer-Aided Design*, 13(7):857–864, July 1994.

[Cav84] J. J. F. Cavanagh. *Digital Computer Arithmetic: Design and Implementation*. McGraw-Hill, 1984.

[CB95] A. P. Chandrakasan and R. W. Brodersen. *Low Power Digital CMOS Design*. Kluwer, Norwell, MA, 1995.

[CJ93] T. K. Callaway and E. E. Swartzlander Jr. Estimating the power consumption of CMOS adders. In *Proc. 11th Computer Arithmetic Symp.*, pages 210–219, Windsor, Ontario, June 1993.

[CK94] Z. Chen and I. Koren. A yield study of VLSI adders. In *Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 239–245, 1994.

[CL92] J. Cortadella and J. M. Llaberia. Evaluation of A + B = K conditions without carry propagation. *IEEE Trans. Comput.*, 41(11):1484–1488, November 1992.

[CL94] M. Capula and L. Luiselli. Reproductive strategies in alpine adders, vipera berus. *Acta Oecologica*, 15(2):207–214, 1994.

[CSTO91] P. K. Chan, M. D. F. Schlag, C. D. Thomborson, and V. G. Oklobdzija. Delay optimization of carry-skip adders and block carry-lookahead adders. In *Proc. 10th Computer Arithmetic Symp.*, pages 154–164, Grenoble, June 1991.

[Cur93] A. Curiger. *VLSI Architectures for Computations in Finite Rings and Fields*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, 1993.

[D+92] D. W. Dobberpuhl et al. A 200-MHz 64-b dual-issue CMOS microprocessor. *IEEE J. Solid-State Circuits*, 27(11):1555–1564, November 1992.

[DB95] J. M. Dobson and G. M. Blair. Fast two's complement VLSI adder design. *Electronics Letters*, 31(20):1721–1722, September 1995.

[DP96] L. Dadda and V. Piuri. Pipelined adders. *IEEE Trans. Comput.*, 45(3):348–356, March 1996.

[ENK94] C. Efstathiou, D. Nikolos, and J. Kalamatianos. Area-time efficient modulo $2^n - 1$ adder design. *IEEE Trans. Circuits and Syst.*, 41(7):463–467, July 1994.

[Feu82] M. Feuer. Connectivity of random logic. *IEEE Trans. Comput.*, C-31(1):29–33, January 1982.

[Fic83] F. E. Fich. New bounds for parallel prefix circuits. In *Proc. 15th ACM Symp. Theory Comput.*, pages 100–109, April 1983.

[Fis90] 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 *Proc. 27th Design Automation Conf.*, pages 361–364, 1990.

[G+94] G. Gerosa et al. A 2.2 W, 80 MHz superscalar RISC microprocessor. *IEEE J. Solid-State Circuits*, 29(12):1440–1454, December 1994.

[GBB94] A. Guyot, M. Belrhiti, and G. Bosco. Adders synthesis. In *IFIP Workshop on Logic and Architecture Synthesis*, pages 280–286, Grenoble, December 1994.

[GHM87] A. Guyot, B. Hochet, and J. M. Muller. A way to build efficient carry-skip adders. *IEEE Trans. Comput.*, 36(10), October 1987.

[GO96] A. De Gloria and M. Olivieri. Statistical carry lookahead adders. *IEEE Trans. Comput.*, 45(3):340–347, March 1996.

[HC87] T. Han and D. A. Carlson. Fast area-efficient VLSI adders. In *Proc. 8th Computer Arithmetic Symp.*, pages 49–56, Como, May 1987.

[Hob95] R. F. Hobson. Optimal skip-block considerations for regenerative carry-skip adders. *IEEE J. Solid-State Circuits*, 30(9):1020–1024, September 1995.

[Hwa79] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.

[Kae97] H. Kaeslin. *VLSI I: Architectures of Very Large Scale Integration Circuits*. Lecture notes, Integrated Systems Laboratoty, ETH Zürich, 1997.

[Kan91] V. Kantabutra. Designing optimum carry-skip adders. In *Proc. 10th Computer Arithmetic Symp.*, pages 146–153, Grenoble, June 1991.

[Kan93] V. Kantabutra. Designing optimum one-level carry-skip adders. *IEEE Trans. Comput.*, 42(6):759–764, June 1993.

[KMS91] K. Keutzer, S. Malik, and A. Saldanha. Is redundancy necessary to reduce delay? *IEEE Trans. Computer-Aided Design*, 10(4):427–435, April 1991.

[KOIH92] T. P. Kelliher, R. M. Owens, M. J. Irwin, and T.-T. Hwang. ELM – a fast addition algorithm discovered by a program. *IEEE Trans. Comput.*, 41(9):1181–1184, September 1992.

[Kor93] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[KP89] F. J. Kurdahi and A. C. Parker. Optimizing power using transformations. *IEEE Trans. Computer-Aided Design*, 8(1):81–92, January 1989.

[KS73] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.*, 22(8):783–791, August 1973.

[KZ96] H. Kunz and R. Zimmermann. High-performance adder circuit generators in parameterized structural VHDL. Technical Report No. 96/7, Integrated Systems Laboratory, ETH Zürich, August 1996.

[LA94] H. Lindkvist and P. Andersson. Techniques for fast CMOS-based conditional sum adders. In *Proc. IEEE Int. Conf. Comput. Design: VLSI in Computers and Processors*, pages 626–635, Cambridge, USA, October 1994.

[LA95] H. Lindkvist and P. Andersson. Dynamic CMOS circuit techniques for delay and power reduction in parallel adders. In *Proc. 16th Conf. Advanced Research in VLSI*, pages 121–130, Chapel Hill, March 1995.

[LF80] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

[LJ96] D. R. Lutz and D. N. Jayasimha. Programmable modulo-k counters. *IEEE Trans. Circuits and Syst.*, 43(11):939–941, November 1996.

[LM90] X. Lai and J. L. Massey. A proposal for a new block encryption standard. In *Advances in Cryptology – EUROCRYPT'90*, pages 389–404, Berlin, Germany: Springer-Verlag, 1990.

[Lo97] J.-C. Lo. A fast binary adder with conditional carry generation. *IEEE Trans. Comput.*, 46(2):248–253, February 1997.

[LS92] T. Lynch and E. E. Swartzlander. A spanning tree carry lookahead adder. *IEEE Trans. Comput.*, 41(8):931–939, August 1992.

[M^+91] J. Mori et al. A 10-ns 54×54-b parallel structured full array multiplier with 0.5-μm CMOS technology. *IEEE J. Solid-State Circuits*, 26(4):600–606, April 1991.

[M^+94] M. Matsui et al. A 200 MHz 13 mm^2 2-D DCT macrocell using sense-amplifying pipeline flip-flop scheme. *IEEE J. Solid-State Circuits*, 29(12):1482–1490, December 1994.

[M^+96] H. Morinaka et al. A 2.6-ns 64-b fast and small CMOS adder. *IEICE Trans. Electron.*, E79-C(4):530–537, April 1996.

[MB89] P. C. McGeer and R. K. Brayton. Efficient algorithms for computing the longest viable path in a combinatorial network. In *Proc. 29th Design Automation Conf.*, pages 561–567, June 1989.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[Mül97] P. Müller. Arithmetische Einheiten auf FPGAs. Student thesis, Institut für Integrierte Systeme, ETH Zürich, February 1997.

[Naj94] F. N. Najm. A survey of power estimation techniques in VLSI circuits. *IEEE Trans. VLSI Syst.*, 2(4):446–455, December 1994.

[NIO96] C. Nagendra, M. J. Irwin, and R. M. Owens. Area-time-power tradeoffs in parallel adders. *IEEE Trans. Signal Processing*, 43(10):689–702, October 1996.

[O^+95] N. Ohkubo et al. A 4.4 ns CMOS 54×54-b multiplier using pass-transistor multiplexer. *IEEE J. Solid-State Circuits*, 30(3):251–257, March 1995.

[Okl94] V. G. Oklobdzija. Design and analysis of fast carry-propagate adder under non-equal input signal arrival profile. In *Proc. 28th Asilomar Conf. Signals, Systems, and Computers*, pages 1398–1401, November 1994.

[OV95] V. G. Oklobdzija and D. Villeger. Improving multiplier design by using improved column compression tree and optimized final adder in CMOS technology. *IEEE Trans. VLSI Syst.*, 3(2):292–301, June 1995.

[Rab96] J. M. Rabaey. *Digital Integrated Circuits*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[RK92] C. Ramachandran and F. J. Kurdahi. Combined topological and functionality based delay estimations using a layout-driven approach for high level applications. In *Proc. European Design Automation Conf.*, pages 72–78, 1992.

[S^+94] K. Suzuki et al. A 500 MHz, 32 bit, 0.4 μm CMOS RISC processor. *IEEE J. Solid-State Circuits*, 29(12):1464–1473, December 1994.

[SBSV94] A. Saldanha, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Circuit structure relations to redundancy and delay. *IEEE Trans. Computer-Aided Design*, 13(7):875–883, July 1994.

[Skl60] J. Sklansky. Conditional sum addition logic. *IRE Trans. Electron. Comput.*, EC-9(6):226–231, June 1960.

[Sni86] M. Snir. Depth-size trade-offs for parallel prefix computation. *J. Algorithms*, 7:185–201, 1986.

[SO96] P. F. Stelling and V. G. Oklobdzija. Design strategies for optimal hybrid final adders in a parallel multiplier. *J. VLSI Signal Processing Systems*, 14(3):321–331, December 1996.

[SP92] H. R. Srinivas and K. K. Parhi. A fast VLSI adder architecture. *IEEE J. Solid-State Circuits*, 27(5):761–767, May 1992.

[Spa81] O. Spaniol. *Computer Arithmetic*. John Wiley & Sons, 1981.

[SWBSV88] K. J. Singh, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proc. IEEE Conf. Computer-Aided Design*, pages 282–285, 1988.

[Tur89] S. Turrini. Optimal group distribution in carry-skip adders. In *Proc. 9th Computer Arithmetic Symp.*, pages 96–103, Santa Monica, CA, September 1989.

[TVG95] V. Tchoumatchenko, T. Vassileva, and A. Guyot. Timing modeling for adders optimization. In *Proc. PATMOS'95*, pages 93–105, Oldenburg, Germany, October 1995.

[Tya93] A. Tyagi. A reduced-area scheme for carry-select adders. *IEEE Trans. Comput.*, 42(10):1162–1170, October 1993.

[WE85] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, Reading, MA, 1985.

[WE93] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, Reading, MA, 1993.

[WNS96] H. Wang, A. Nicolau, and J-Y. S. Siu. The strict time lower bound and optimal schedules for parallel prefix with resource constraints. *IEEE Trans. Comput.*, 45(11):1257–1271, November 1996.

[WT90] B. W. Y. Wei and C. D. Thompson. Area-time optimal adder design. *IEEE Trans. Comput.*, 39(5):666–675, May 1990.

[Xil97] Xilinx Inc. *XC6200 Field Programmable Gate Arrays*, 1997.

[ZCB+94] R. Zimmermann, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber, and W. Fichtner. A 177 Mb/s VLSI implementation of the international data encryption algorithm. *IEEE J. Solid-State Circuits*, 29(3):303–307, March 1994.

[ZF97] R. Zimmermann and W. Fichtner. Low-power logic styles: CMOS versus pass-transistor logic. *IEEE J. Solid-State Circuits*, 32(7):1079–1090, July 1997.

[ZG96] R. Zimmermann and R. Gupta. Low-power logic styles : CMOS vs CPL. In *Proc. 22nd European Solid-State Circuits Conf.*, pages 112–115, Neuchâtel, Switzerland, September 1996.

[Zim96] R. Zimmermann. Non-heuristic optimization and synthesis of parallel-prefix adders. In *Proc. Int. Workshop on Logic and Architecture Synthesis*, pages 123–132, Grenoble, France, December 1996.

[Zim97] R. Zimmermann. *Computer Arithmetic: Principles, Architectures, and VLSI Design*. Lecture notes, Integrated Systems Laboratoty, ETH Zürich, 1997.

[ZK] R. Zimmermann and H. Kaeslin. Cell-based multilevel carry-increment adders with minimal AT- and PT-products. submitted to *IEEE Trans. VLSI Syst.*

Curriculum Vitae

I was born in Thusis, Switzerland, on September 17, 1966. After finishing high school at the *Kantonsschule Chur GR* (Matura Typus C) in 1986, I enrolled in Computer Science at the Swiss Federal Institute of Technology *ETH Zürich*. I received the Diploma (M.Sc.) degree in Computer Science (*Dipl. Informatik-Ing. ETH*) in 1991. In May 1991 I joined the Integrated Systems Laboratory (IIS) of ETH, where I worked as a research and teaching assistant in the field of design and verification of digital VLSI circuits. I was involved in the implementation of VLSI components for cryptographic and spread-spectrum systems and in the design and synthesis of arithmetic units for cell-based VLSI. My research interests include digital VLSI design and synthesis, high-speed and low-power circuit techniques, computer-aided design, computer arithmetic, cryptography, and artificial intelligence.