High-Performance Decimal Floating-Point Units

PhD Dissertation

Álvaro Vázquez Álvarez

DEPARTAMENTO DE ELECTRÓNICA E COMPUTACIÓN
UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
DEPARTAMENTO DE ELECTRÓNICA E COMPUTACIÓN

PhD. Dissertation

High-Performance Decimal Floating-Point Units

Álvaro Vázquez Álvarez
Santiago de Compostela, January 2009

Prof. **Elisardo Antelo Suárez**, at the Department of Electronics and Computer Engineering of the University of Santiago de Compostela,

Hereby CERTIFIES:
That the research work entitled "**High-performance decimal floating-point units**", has been developed by **Álvaro Vázquez Álvarez** under my supervision in the Department of Electronics and Computer Engineering of the University of Santiago de Compostela, and qualifies him eligible for the PhD degree in Physics (Computer Science).

Santiago de Compostela, November 2008

Dr. **Elisardo Antelo Suárez**, Profesor Titular de Arquitectura e Tecnoloxía de Computadores da Universidade de Santiago de Compostela,

CERTIFICA:
Que a memoria titulada "**High-performance decimal floating-point units**", foi realizada por D. **Álvaro Vázquez Álvarez** baixo a miña dirección no Departamento de Electrónica e Computación da Universidade de Santiago de Compostela e constitúe a Tese que presenta para optar ao grado de Doutor en Ciencias Físicas (Electrónica).

Santiago de Compostela, Novembro de 2008

Asinado: **Elisardo Antelo Suárez**,
Director da tese.

Asinado: **Javier Díaz Bruguera**,
Director do Departamento de
Electrónica e Computación.

Asinado: **Álvaro Vázquez Álvarez**,
Doutorando.

To my family
Á miña familia

Acknowledgements

It has been a long way to see this thesis successfully concluded, at least longer than what I imagined it. Perhaps the moment to thank and acknowledge everyone's contributions is the most eagerly awaited. This thesis could not have been possible without the support of several people and organizations whose contributions I am very grateful.

First of all, I want to express my sincere gratitude to my thesis advisor, Elisardo Antelo. Specially, I would like to emphasize the invaluable support he offered to me all these years. His ideas and contributions have a major influence on this thesis.

I would like to thank all people in the Departamento de Electrónica e Computación for the material and personal help they gave me to carry out this thesis, and for providing a friendly place to work. In particular, I would like to mention to Prof. Javier D. Bruguera and the other staff of the Computer Architecture Group. Many thanks to Paula, David, Pichel, Marcos, Juanjo, Óscar, Roberto and my other workmates for their friendship and help.

I am very grateful to IBM Germany for their financial support though a one-year research contract. I would like to thank Ralf Fischer, lead of hardware development, and Peter Roth and Stefan Wald, team managers at IBM Deutchland Entwicklung in Böblingen. I would like to extend my gratitude to the FPU design team, in special to Silvia Müller and Michael Kröner, for their help and the warm welcome I received during my stay in Böblingen. I would also like to thank Eric Schwarz from IBM for his support.

Many thanks to Paolo Montuschi from Politecnico di Torino for his collaboration in several parts of this research.

Finally, I want to thank the institutions that have financially supported this research through trip grants for attending to different conferences: Universidade de Santiago de Compostela, Ministerio de Ciencia y Tecnología (Ministry of Science and Technology) of Spain under contracts TIN2004-07797-C02 and TIN2007-67537-C03, and Xunta de Galicia under contract PGIDT03-TIC10502PR.

"*Before computers,*
I used my 10 fingers,
but now . . .
what am I supposed to do
with the other 9 ?"

Anonymous

Contents

List of Figures

List of Tables

Preface

The decimal arithmetic units implemented in early computers have been replaced in today's processors by binary fixed and floating-point units. Binary arithmetic is suitable for scientific applications due to its mathematical properties and performance advantage, since, in electronic computers (based on two-state transistors), binary data can be stored more efficiently and processed faster than decimal data.

However, the decimal format is preferred for many other non-scientific applications. Thus, current financial, e-commerce and user-oriented applications make an intensive use of integer and fractional numbers represented in decimal radix. General-purpose microprocessors only provide hardware support for binary-to-from-decimal conversions while, until very recently (mid 2007), hardware support for decimal arithmetic in mainframe microprocessors was limited to speedup some basic decimal integer arithmetic operations.

Therefore, microprocessors had to manage decimal numbers using, mainly, their binary fixed and floating-point arithmetic units, introducing costly performance penalties due to conversion operations. Thus, decimal data is converted to a binary representation before being processed using binary arithmetic and the results are converted back to a decimal representation. For intensive decimal data workloads, these conversion operations contribute significatively to the total processing time.

More importantly, the binary floating-point units may generate accuracy errors when processing the decimal data directly, since many decimal fractions cannot be represented exactly in binary (for example, 1/10 = 0.1 has not an exact binary representation). In many financial and monetary applications, the errors introduced by decimal to binary conversions are not tolerated and must be corrected since they may violate the legal requirements of accuracy.

Software implementations of decimal floating-point arithmetic fulfill the accuracy and precision requirements, but are at least an order of magnitude slower than hardware units and cannot satisfy the increasing workloads and performance demands of future financial and commercial applications. By other hand, the continuous size scaling of transistors permits the fabrication of integrated circuits with a complexity of billions of transistors at a moderate cost. This makes an attractive opportunity for the microprocessor manufactures to provide a dedicated DFU (decimal floating-point unit) in their new high-end microprocessors.

This interest has been supported by the efforts in defining a standard for decimal floating-point arithmetic. Specifically, the revision of the IEEE 754-1985 and IEEE 854-1987 floating-point standards (IEEE 754-2008), incorporates specifications for decimal arithmetic. It can be implemented in hardware, software or in a combination of both. In this way, the first IEEE 754-2008 compliant DFUs are already in the market. The dual core IBM Power6 processor, released in June 2007, and the quad core IBM z10 mainframe microprocessor, released

in March 2008, implement a DFU in each core. Other manufactures, such as Intel, have opted for software implementations, but they are planning to incorporate some hardware to assist the binary arithmetic unit in IEEE 754-2008 decimal-floating point operations. More processors are expected to gradually incorporate hardware support for decimal floating-point arithmetic. In addition, it is arising a significant academic work on high-performance decimal hardware design.

Decimal floating-point arithmetic is again a hot topic of research. Also, more efficient architectures for decimal fixed-point units are necessary to satisfy the increasing demands of high-performance decimal processing. So, welcome back decimal arithmetic!

In this context, this Ph.D. thesis presents the research and design of new algorithms and high-performance architectures for DFX (decimal fixed-point) and DFP (decimal floating-point) arithmetic units. The most part of the results are already published in [144, 145, 148, 149, 150, 147]. The main contributions of this work to the field of decimal computer arithmetic are the following:

- A new algorithm for 10's complement carry-propagate addition or subtraction [144, 145] of two integer operands represented in BCD (binary coded decimal), and the extension for sign-magnitude BCD addition/subtraction. The proposed high-performance architectures can be implemented using any fast binary carry prefix tree topology, resulting in more efficient designs of DFX adders.

- A new algorithm to combine IEEE 754-2008 significand BCD addition or subtraction with decimal rounding that only requires a single full word carry propagation. This leads to more efficient implementations of high performance decimal DFP adders and DFP multipliers.

- A new method for sum error checking applied to (10's complement and sign-magnitude) BCD addition or subtraction.

- A novel method for fast multioperand decimal carry-save addition or subtraction [148, 150] using unconventional (non BCD) decimal codings.

- Different efficient designs of decimal carry-save adder trees for any number of input operands based on the previous method.

- Two new parallel DFX multipliers [148, 150] implemented using different signed digit recodings of the multiplier and the proposed decimal carry-save adder trees.

- A combined parallel binary/decimal fixed-point multiplier that presents a reduced latency for binary operations.

- A new design of a IEEE 754r DFP multiplier combining some of the proposed DFX parallel multipliers with the proposed BCD adder with rounding.

- A proposal for the design of a DFP fused multiply-add (FMA).

- A new digit-recurrence algorithm and architecture for radix-10 division [149] that adapts conventional techniques developed to speed-up binary radix-2^k division with novel features to improve radix-10 division.

The structure of this thesis is as follows. Chapter 1 presents an overview of decimal computer arithmetic, including a review of the use of decimal arithmetic in the history of computation, a discussion of the current needs of high-performance implementations of decimal arithmetic, and future trends in the field.

Chapter 2 summarizes the decimal specification of the IEEE 754-2008 standard and discusses some important issues about compliant IEEE 754-2008 DFU design. We finally present the preferred set of decimal arithmetic operations to implement based on estimations of real demands and the benefits and costs of accelerating them in hardware.

These implementations are detailed from Chapter 3 to Chapter 8. The structure of these Chapters is very similar: first, we provide a background of the previous and most representative methods and architectures proposed to implement in hardware each arithmetic operation, detailing their strengths and weaknesses. Next, we introduce and discuss our proposals for the selected operation, and finally we present the area and delay evaluation figures of the proposed high-performance implementations and a comparative study between the different alternatives. Appendix A is included to provide a better understanding of the evaluation method used to estimate the area and delay of the different circuit topologies at the gate level.

Chapters 3 and 4 deal with decimal integer and fixed-point addition of two operands. Specifically, Chapter 3 is devoted to 10's complement BCD carry-propagate addition or subtraction while Chapter 4 is devoted to sign-magnitude BCD carry-propagate addition or subtraction.

Chapter 5 is focused on DFP addition, specifically on significand BCD addition and IEEE 754-2008 decimal rounding.

In Chapter 6 we include the carry-free propagation methods for multioperand decimal addition, used to speed-up other operations such as decimal multiplication and division.

Chapter 7 covers both DFX and DFP multiplication. It also contains a proposal for an implementation of a decimal FMA (fused multiply-add) unit.

Chapter 8 deals with decimal division, in particular with subtractive methods based on the radix-10 digit-recurrence iteration. An overview of DFP division is first presented to discuss next the implementation of DFX division.

Finally, we present the main conclusions and future work.

Chapter 1

Decimal Computer Arithmetic: An Overview

Decimal is the most natural arithmetic system for humans. People represent and exchange numerical information in base (or radix) 10 and perform calculations applying the rules of decimal arithmetic they are taught at school. However, in today's computers, data is processed mainly in the binary system, using some fixed-point or floating-point representation in radix 2. Although most of the early computers used decimal arithmetic [60, 61], binary has practically replaced decimal in current computers because it is faster an more efficient [17].

Until very recently, only decimal fixed-point or integer BCD arithmetic units were implemented in some high-end processors oriented to commercial servers [19, 20]. High performance floating-point computations were carried out in binary, appropriate for scientific computations. The use of DFP (decimal floating-point) hardware in real applications has been limited to hand-held calculators [66, 135] or early computers [18, 78]. But now, due to the high demands of numerical processing for new financial, commercial and Internet-based applications, efficient implementations of DFP arithmetic are required to address the limitations of binary hardware or DFX (decimal fixed-point) formats [35].

In this Chapter we present the precedents, facts and future trends related to this recent and increasing interest in providing efficient high-performance implementations of DFP arithmetic. Section 1.1 outlines the evolution of the ALUs (arithmetic logic units) from the early computers to the modern microprocessors available nowadays. Section 1.2 discuss the characteristics of the decimal processing demands for the new financial and business applications. It also describes the main drawbacks of the binary floating-point arithmetic and integer and fixed-point arithmetic to deal with decimal data. Section 1.3 covers the recent efforts towards the definition of efficient specifications and a standard for DFP arithmetic. Next, we analyze whether the new software and hardware implementations of DFP arithmetic satisfy the accuracy and performance demands, introducing several research studies leaded from both industry and academy. Finally, Section 1.4 presents a vision of the evolution and future trends in this field. A quite complete bibliography on decimal arithmetic can be found in Mike Cowlishaw's general decimal arithmetic website [33].

1.1 The evolution of ALUs: decimal vs. binary

The evolution of the ALUs goes in parallel to the history of computers [24]. The decimal number system was used in mechanical computers to help with the manual calculations of commerce and science. Two well-known examples are the specific-purpose Difference Engine and the general-purpose Analytical Engine of Charles Babbage (1792-1871), which was projected to have a differentiate ALU that used columns of 10-toothed wheels and other mechanical components to perform decimal calculations [60]. In 1914, Torres y Quevedo, a Spanish scientist and engineer, wrote a paper describing how electromechanical technology (based on relays) could be applied to produce a complete Analytical Engine [112].

Separate decimal and binary ALUs. From 1945 to 1960.

The electronic era of computers began with the vacuum tubes, that substantially reduced the delays of the mechanical and electromechanical computers. Many of the early electronic computers, such as the ENIAC [61], were decimal. Their arithmetic units operated directly with numbers and/or addresses represented in decimal. BCD (binary coded decimal) was the most frequent encoding of decimal digits although many other codes were considered [163]. The ALUs of these computers could only perform simple arithmetic and logic operations (usually add and compare) in circuits called accumulators [24]. In the ENIAC, fully operative in 1946, the decimal arithmetic circuits were mixed with the program circuitry, which had to be reconfigured (rewired) for a new task. Based on the notes of Eckert and Mauchly, the designers of the ENIAC, and the previous works of Alan Turing, John von Neumann formalized in 1945 the stored-program architecture (the EDVAC computer [151]), on which the programs were internally stored in a memory (both data and instructions) and separated from the ALU and the other components (input/output system and control logic) [129].

This work contributed widely to popularize the general-purpose electronic digital computers. The first generation of commercial models appeared around 1950. Examples of decimal computers are the UNIVAC and the IBM 650. The arithmetic processor of the UNIVAC consisted of four general-purpose accumulators and could perform about 465 multiplications per second [24]. It used four binary digits to code each decimal digit. Each word was 45 bits long, representing 11 decimal digits plus a sign.

During all the 1950s both decimal and binary arithmetic were implemented in computers [113]. The advantage of binary arithmetic over decimal implementations was already considered by Burks, Goldstine, and von Neumann in their reports from the Princeton Institute for Advanced Study (IAS) [17]. They proposed a pure binary architecture (for both addressing and data processing), the IAS computer, and concluded that it was simpler to implement than a decimal design using two-state digital electronic devices (vacuum tubes, in this case). Since it required a reduced number of components (less memory and arithmetic circuits), it was therefore more reliable. They also pointed out that binary arithmetic was optimal for scientific computations because of their mathematical properties.

However, other authors [13, 117] suggested later that the combination of binary addressing with decimal data arithmetic was more powerful. They considered that, in many applications in which few arithmetic steps are taken on huge data workloads, format conversions can contribute significatively to the processing time when data are represented in a different format from that used in the arithmetic unit. This implied that computers needed at least two

arithmetic units, one for binary addressing and the other for decimal data processing.

Therefore, the development of two separate lines of computers persisted into the end of the 1950s. One type of computers was focused on scientific and engineering applications, involving long and complex calculations with numbers in a wide range of magnitudes. The ALUs implemented in these models were primarily binary floating-point units, such as the one in the Z4 (derived from Konrad Zuse's first programmable floating-point units), and the one in the IBM 704. The first computers to operate with decimal floating point arithmetic hardware were the Bell Laboratories Model V and the Harvard Mark II, both of which were relay calculators designed in 1944.

Floating-point arithmetic was proposed independently by Torres y Quevedo (1914), Zuse (1936) and Stibitz (1939) [85]. It allowed their users to manage with the overall scale of a computation in hardware, otherwise having to program the complex scaling operations required by fixed-point formats. However, floating-point units require much more circuitry than fixed-point ALUs and only binary implementations were finally considered [24].

The other type of computers was intended for business applications, which involved processing large amounts of data. However, the commercial calculations of that period did not need to handle numbers of many digits and large ranges. So the business-oriented computers used a shorter word length than scientific computers and decimal fixed-point arithmetic usually with two digits to the right of the decimal point.

Merging of decimal and binary ALUs. From 1960 to 1970.

The arrival of the solid-state semiconductor transistor and the integrated circuit contributed to the large scale production of computers, greatly reducing the size and cost of computers. In this way, more complex arithmetic and logic were introduced in the second generation of electronic computers from the 1960s.

At the beginning of the 1960s, the tendency was to provide only binary arithmetic units. Binary was much easier to implement than other number systems in the Boolean logic of electronic devices, and more reliable after the introduction of the transistor in computers. A survey of computer systems carried out in the USA by 1961 [161] reported that the most preferred system was binary, "131 utilize a straight binary system internally, whereas 53 utilize the decimal system (primarily BCD, binary coded decimal)...".

However, several models, such as the IBM 1401, the IBM 1620 and the IBM 7070, implemented decimal arithmetic units for data processing, the majority integer and fixed-point. Manufacturers felt that commercial customers did not need floating point, so DFP in hardware was very rare [35]. Some exceptions are the normalized variable-precision DFP feature implemented in the IBM 1620 computer [78] and the Burroughs B5500 computer [18], that could use an integer or a fixed-point coefficient of 21 or 22 precision digits.

Other early representatives of this generation are the IBM 7094 and the UNIVAC 1100 and 2200 series targeted to high-performance applications of science and engineering. The introduction of the IBM 7094 led origin to mainframe computers. It could perform between 50,000 and 100,000 binary floating-point operations per second using 36-bit words [24]. This word size could handle a precision of 10 decimal digits, which was adequate for most business applications, so customers used floating-point hardware for business computations as well.

Moreover, many customers used scientific computers for commercial applications, as well as business-oriented computers was often installed in university centers, where professors and students develop floating-point software for it [24].

Thus, IBM finally combined both lines (scientific and commercial) with the introduction of the System/360 line of computers in 1964 [2]. The IBM System/360 mainframes implemented a hexadecimal floating-point unit [3][1], but decimal computations were carried out using a BCD integer arithmetic unit (mainly a BCD adder).

The transistor also helped create a new type of computers, the minicomputers, that provided computing facilities at reduced cost than the mainframes. The early minicomputers appeared around 1960, and end users could interact directly with the computer. To reduce costs, the minicomputers used a short word length (12 bits in the PDP-8). Thus, at first, they could not compete with mainframes in decimal arithmetic (a 12-bit word only supports 3 decimal digits) or floating-point computations, but for many other applications the minicomputer was the better performance/cost solution.

With the massive production of integrated circuits (chips) in the second half of the 1960s, minicomputers were fabricated at lower cost and could offer support for wider word lengths and some high-performance characteristics available in mainframe computers. Thus, minicomputers became also competitive for some commercial and scientific uses. To minimize the threat of minicomputers, mainframes also replaced the circuits of discrete components by the integrated circuit at the end of the decade. This allowed the use of high-performance techniques to speed up ALUs.

Dedicated binary floating-point units. From 1970 to 1985.

In the 1970s, binary was very popular with very few computers supporting decimal. The new parallel techniques proposed to speed up arithmetic operations in hardware [7, 40, 86, 94, 114, 153] were implemented in many cases in the high-performance binary arithmetic units of mainframes, represented by the IBM S/370 line of computers [21]. Only in a few cases they were applied to decimal arithmetic and limited to improve the BCD integer units of mainframe computers [118]. In mainframes, the different integer ALUs and binary floating-point units [142] were implemented in separate chips from the control logic unit.

In spite of the architectural and technological advances of minicomputers, they could not replace the mainframes in the business data-processing field. On the contrary, they generated a new demand for low cost computation, leading to the personal computers and being finally replaced by these. In addition, the programmable pocket calculators [66], introduced in the mid-1970s by Hewlett-Packard and Texas Instruments at the price of a current laptop, were presented as personal computers and also created a huge demand among engineers and professional and financial people. However, these devices were not general-purpose computers but specialized numerical processors that could compute, apart from the basic arithmetic operations, logarithms and trigonometric functions using decimal floating-point arithmetic to 10 decimal digits of precision.

The microprocessor incorporated a general-purpose stored-program architecture into a single integrated circuit, making possible the rising of personal computers at the end of the

[1]This unit is of a similar hardware complexity as an equivalent binary floating-point unit, since the hexadecimal radix is an integer multiple of 2.

1970s. The early microprocessors, marketed for personal computers, only integrated binary fixed-point or integer ALUs [75]. Some of them, such as those based on the Intel x86 and on the Motorola 68x architectures, provided instructions for 8-bit word BCD addition and subtraction [76, 104]. Nevertheless, these arithmetic operations were actually performed in the binary integer ALU with the support of 8-bit BCD from/to binary conversion instructions [70]. Moreover, these instructions were not extended to formats wider than 8 bits, since this decimal support would be incorporated later into the binary floating-point units.

Because of their complexity, the binary floating-point units were, at first, not integrated into the microprocessors, but in specialized chips called floating-point accelerators or coprocessors [65, 69, 110, 134]. These floating-point coprocessors also contributed to the growth of a new type of specialized high-end computers intended for scientific intensive floating-point calculations, the supercomputers, while the mainframes were focused mainly on business applications.

Standardized binary floating-point units. From 1985 to 2007.

The fast spread of floating-point units led to many proprietary floating-point formats and different rounding behaviors for the arithmetic operations [59]. Therefore it was very necessary to standardize a floating-point system to provide reliable and portable results to users. Thus, in 1985 a binary floating-point arithmetic standard was released (IEEE 754-1985 [72]) and is now implemented in almost all microprocessors.

When the scale of integration made it possible, the binary floating-point coprocessors were incorporated into the microprocessors [55]. Compliant IEEE 754 floating-point units and increasing performance capabilities were implemented in microprocessors for embedded systems [6, 107], laptop and desktop computers [11, 58], workstations and servers [105, 28] and supercomputers [87, 152]. Because of the widespread acceptance of IEEE 754-1985, IBM decided finally, at the end of the 1990s, to provide compliant IEEE 754-1985 floating-point units in its microprocessors for the S/390 mainframes [1, 123], and in the following generation of 64-bit microprocessors for the IBM z/Series mainframes [57, 123].

Although the IEEE 754-1985 standard was soon adopted by practically all the microprocessor developers, the attempts to popularize another radices different than binary at the end of the 1980s were not very successful. The IEEE standard for radix-independent floating-point arithmetic (IEEE 854-1987 [73]) lacked of some features such as an efficient binary encoding to represent the decimal numbers, and the manufactures did not perceive in the market a real demand of decimal floating-point processing. Moreover, the fabrication technology was not mature enough to integrate a dedicated decimal floating-point unit into a general-purpose microprocessor.

But at the beginning of the 2000s, these factors had changed and evolved. The new financial and business demands of a global market [52], the recent developments of reliable and efficient decimal specifications and encodings [39] and the technological improvements that allowed the integration of multiple processors on a chip (multicore), have created an interest in the microprocessor industry to provide support for decimal floating-point arithmetic [31, 35]. In addition, an incipient research was producing significant advances in this field [121]. Because of this interest, the recently approved revision of the IEEE standard for floating-point arithmetic (IEEE 754-2008 [74]) includes specifications for decimal floating-point arithmetic.

However, prior to 2007, few computing systems provided decimal facilities in hardware and none of these implementations was for decimal floating-point. For instance, the early IBM z/Series mainframes only included decimal integer arithmetic units [19, 20], consisting basically in a BCD adder with some hardware support to accelerate other decimal integer operations. Microprocessors based on the Intel x86 architecture provide a 18-digit packed BCD format to simplify decimal integer arithmetic operations, but these are carried out as binary integer computations in the binary floating-point unit, which requires costly format conversions [70]. A very similar limited support can be found in other architectures including Motorola 68x, IBM PowerPC and HP PA-RISC.

The early implementations of compliant IEEE 754-2008 DFUs (decimal floating point units) in commercial microprocessors arrived in the first semester of 2007. The dualcore IBM Power6 server microprocessor includes a IEEE 754-2008 DFU in each core [45, 122]. Also, similar IEEE 754-2008 DFUs (one per core) are included in the 64-bit quad core microprocessor of the z/Series z10 mainframe [160], launched in March 2008. Previously, a compliant firmware implementation of IEEE 754-2008 DFP, accelerated using the available BCD hardware, was provided for the IBM z9 mainframe microprocessor [44].

Finally, we expect that DFUs follow the same roadmap of binary floating-point units, from high-end computers to low-end processors, and that they become as popular as binary units are nowadays.

1.2 The new financial and business demands

Financial and commercial applications for risk management, banking, accounting, tax calculation, currency conversion, insurance, marketing, retail sales, among many other business areas, make an intensive use of numerical data processing. In addition, the recent movement of many business processes to the Web (e-commerce, e-banking,...), the rapid expansion and development of the new emergent economies and the global economic and financial markets have triggered the demand of computational power.

Commercial software applications [71, 96] are typically executed in high-performance mainframes [2] [88, 160], which offer a specialized support such as:

- Reliability. They provide features to detect and correct system faults.
- Availability. They are available for long periods of time even when parts of the system are malfunctioning.
- Serviceability. The maintenance and repair tasks are carried out affecting as little as possible the normal operation of the system.
- Parallel processing to run a diversity of tasks using multithreading [88] with multiple processors and cores [160].
- Dedicated service processors for cryptographic support, I/O handling, monitoring, memory handling,...

[2] We use this term only for servers oriented to commercial applications such as databases, Web services, ERP (Enterprise Resource Planning) systems, financial tools, ...

- Numerical processing capabilities optimized to perform simple computations involving large amounts of data workloads. These numerical data are mainly decimal fractions (rational numbers whose denominator is a power of ten) and integer numbers in a wide range of values [138].

This last issue is of special importance for the design of the processor arithmetic. The calculations used in commercial and financial applications follow the human rules and conventions of decimal arithmetic, that may differ from the conventional arithmetic used for scientific calculations.

For instance, in scientific applications, numerically identical values such as 32.60 and 32.6 are treated in the same way. The binary floating-point arithmetic implemented in computers is normalized and does not distinguish numbers of equal value. In commercial and other human-oriented applications, the trailing fractional zero in 32.60 may represent extra information that should be preserved [35], such as the unit of measurement (centimeter), of currency (cent)...

For this reason, the decimal numbers used in financial applications are usually encoded as integer coefficients scaled by a power of ten. For example, the value 764.50 is represented as an integer coefficient 76450 with a scale of 2 (or an exponent -2), that is, as $76450 \cdot 10^{-2}$. This integer scaled encoding is redundant since more than one coefficient may represent the same value: both coefficients 009 (with an exponent 2) and 090 (with an exponent 1) represent the value 900. Although a normalized fixed-point (non redundant) coefficient could also be used, this is more adequate for scientific calculations.

The scale or exponent can be fixed or floating. Some applications work with a fixed scale (fixed decimal point) which is preserved for all the calculations. Other applications, such as currency conversions, need a floating-point type. For instance, the Euro exchange rates are given to 6 precision digits [52], so the decimal point position is floating (1 Euro = 166.386 Pesetas = 6.55957 Francs).

Consequently, in addition to compute numerical values, the implementations of decimal arithmetic should be able to preserve the full precision of the numbers when required, including the trailing fractional zeroes. Early computers used exact decimal arithmetic, so they had to provide word length enough to support the full precision and range that required the financial calculations. However, 11 or 12 digits were sufficient for the applications of that time [24].

Current applications use, typically, 25 to 32 precision digits in order to represent a good range of values exactly. But, since applications are now more complex, having to deal with a wider range of values, implementations of exact arithmetic have become inefficient for many situations [35]. For instance, an exact decimal multiplication requires the double of the precision digits of the largest input value. In this way, a sequence of multiplications would exceed soon any precision available in hardware. Therefore, the use of rounding is necessary in financial and commercial applications in two different situations:

- To approximate to the exact result in many complex computations (rounding imposed by precision).
- To reduce the exact result to a lower precision demanded by the application (rounding

imposed by legal requirements).

The most common decimal rounding modes used in financial calculations are *round-half-up* (round to nearest, ties away from zero) and *round-down* (rounded towards zero or truncation), though *round-half-even* (round to nearest, ties to even) is sometimes applied to cancel out rounding errors on average.

Numerical processing is carried out by computers using different arithmetic systems:

1. Binary floating-point arithmetic. Numbers are represented as

$$FX = (-1)^{s_X} \ (c_X.f_X) \ \cdot 2^{E_X} \tag{1.1}$$

   where s_X is the sign, c_X and f_X are respectively the integer and fractional parts of the coefficient $X = c_X.f_X$ and E_X is the integer exponent ($c_X = 1$ for a normalized representation).

2. Integer (binary or BCD) arithmetic. Numbers represented as $X = (-1)^{s_X} \ c_X$

3. Fixed-point (binary or decimal) arithmetic. Numbers represented as $FX = (-1)^{s_X} \ (c_X.f_X)$ ($c_X = 1$ in case of a normalized binary representation and $c_X \in \{1, \ldots, 9\}$ in case of a normalized decimal representation).

4. Decimal floating-point arithmetic. Numbers are represented as

$$FX = (-1)^{s_X} \ (c_X.f_X) \ \cdot 10^{E_X} \tag{1.2}$$

   where $f_X = 0$ in case of an integer coefficient, and $c_X \in \{1, \ldots, 9\}$ in case of a normalized fractional coefficient.

The straightforward method to perform a decimal calculation would be to use a **binary floating-point** unit directly. However, though integer numbers have exact conversions, most decimal fractions can only be approximated by binary floating-point numbers [27, 56]. For example, decimal values such as 0.1 (= 10^{-1}) would require an infinite sum of binary fractions (infinite precision) for an exact binary representation. Moreover, decimal to/from binary conversions [27, 56, 130] are implemented in software routines with high computational cost.

Conversion errors could lead to inaccurate results when a decimal calculation is carried out with binary floating-point arithmetic. Fig. 1.1 details an example of a 5% tax applied over a phone call of cost 0.70 euros, rounded to the nearest cent (ties to even) [35]. Using double precision binary floating-point for the calculation, the result is one cent less than the expected (tax calculations are regulated by law). These systematic one-cent errors add up, so, for a mobile phone company with millions of calls a day, the annual loses could represent over a million euros [35].

Notice that these errors are due to the lack of accuracy of binary floating-point, and not to rounding. These type of decimal exactly rounded results, although inexact, are imposed by legal and financial requirements and are the expected results of a decimal computation. Therefore, binary floating-point cannot be used for financial calculations or for any application based on decimal human-oriented arithmetic, since it cannot neither meet legal requirements nor provide decimal exactly rounded results.

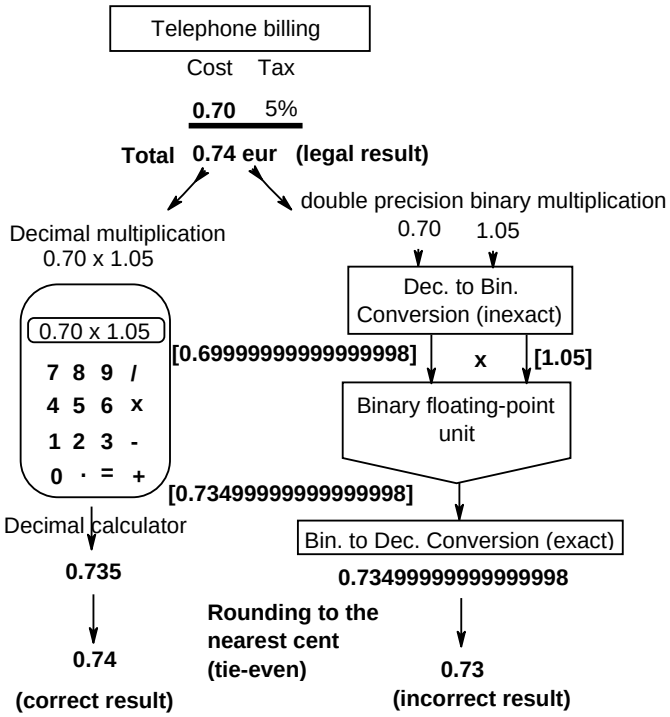


Figure 1.1. *Example of a decimal tax calculation using binary floating-point.*

Both **fixed-point and integer arithmetic units** can be used to satisfy the accuracy requirements of decimal applications. However, the range of numbers they can hold is limited by the format precision[3], being necessary, for many current applications, to work with scaled integer data (a type of floating-point). Thus, scaling operations must be applied using software solutions and libraries [38, 42, 103] or by manual programming to keep the coefficients in the available range during the calculations. This manual tracking of scales is error-prone and difficult to manage, specially when both large and small values are involved in a calculation [35].

Another limitation is that fixed-point and integer arithmetic units do not implement rounding. Rounding operations are characteristic of floating-point computations and when no rounding occurs the operation can be reduced to an integer or fixed-point computation. However, as it was previously mentioned, rounding is demanded by many current financial and commercial applications, so it must be explicitly applied in software when using integer or fixed-point arithmetic.

The integer coefficients are represented in binary form or in a decimal (mainly BCD) representation. The advantage of BCD over binary is that it simplifies rounding, scaling and conversions, although the BCD arithmetic units are slightly complex and slower than binary units. But even with the speedup support of BCD integer and fixed-point units [19, 20], the simulation of scaled arithmetic has a significant overhead with respect to floating-point hardware implementations.

Therefore, the use of a decimal floating-point arithmetic unit could be a solution to:

[3]Maximum number of digits available to represent the integer (preferred) or fixed-point coefficient.

- Overcome the inaccuracy of binary floating-point arithmetic.
- Extend the limitations of fixed-point and integer hardware for scaling and rounding.

In addition, to be widely used, it should be conformed to modern standards for floating-point arithmetic. However, the radix independent floating-point system defined in IEEE 854-1987 was primary designed for scientific and engineering uses and seems to not satisfy the commercial needs [79]. For instance, this decimal floating-point arithmetic should also meet other requirements of commercial and financial applications more oriented for fixed-point computations such as

- To provide the exact result in simple calculations (rounding imposed by precision must be avoided in these cases).
- To provide rounding of the exact result to a variable lower precision when it is demanded (rounding defined by the application).
- To preserve the scales in exact computations (normalization loses scale information).

which seem to be, apparently, in contradiction with a floating-point type. Nevertheless, decimal floating-point types using a non-normalized, integer coefficient can satisfy the previous requirements [39]. Furthermore, exact computations could be carried out directly in hardware, supporting enough precision digits to represent this integer coefficient in the most part of practical situations, applying scaling by software only to exceptional cases.

So, before providing reliable decimal floating-point units for commercial use, it was necessary to develop a consistent specification of a decimal floating-point arithmetic [37], efficient when implemented in both software and hardware and compatible with the established rules of human-oriented decimal arithmetic used in finances. Overall, it was necessary to incorporate this specification in a standard, the IEEE 754-2008, that could be widely accepted as the IEEE 754-1985 was.

1.3 Decimal floating-point: Specifications, standard and implementations

Apart from its use in early computers, proposed designs of DFP arithmetic units prior to 2000 [12, 29, 115] were mainly intended for scientific and engineering uses, since they assumed fractional coefficients and did not meet the special requirements of scaled decimal arithmetic in commercial applications. Thus, in real applications, the use of those scientific-oriented implementations was limited to low-performance applications as hand-held calculators [66, 135], since, for a given investment, binary floating-point units provide better performance and accuracy.

On the other hand, the implementations of scaled integer decimal arithmetic [42] (in effect a floating-point system), of common use in databases and other software for commercial applications prior to 2000, did not incorporate support for the floating-point types defined by the IEEE 854-1987 standard, so they could not make use of future hardware facilities for DFP arithmetic to improve these computations.

Some initial attempts to provide a more general data type for decimal arithmetic were the Microsoft Decimal class, and the Rexx DFP [99], but a more reliable and general specification for DFP was already required. Therefore, at the beginning of the 2000s, IBM introduced a more general specification for decimal floating-point [39, 37] also suitable for commercial uses. This specification provides the necessary decimal support for commercial applications, including exact unrounded decimal arithmetic and integer arithmetic, and is also compatible with the DFP arithmetic defined in IEEE 854-1987, allowing efficient implementations in both hardware and software.

To avoid the different proprietary formats, IBM suggested in 2002 to incorporate its proposal for decimal arithmetic into the ongoing revision of the IEEE 754-1985. Thus, these decimal specifications were merged into the draft of the IEEE 754-2008 and the IEEE 854-1987 was, in fact, incorporated to the revision. The main addition in IEEE 754-2008 with respect to IEEE 854-1987 is the inclusion of format specifications for the new DFP data types, used for integer, fixed-point, and floating-point decimal arithmetic.

The IEEE 754-2008 [74] defines three interchange decimal formats (of 32, 64 and 128 bits) with two different encodings for decimal floating-point numbers (represented as a sign, a coefficient and an exponent). The exponent in encoded as an unsigned binary integer. The coefficient can be encoded as a binary or a decimal unsigned integer. To allow an efficient packing of decimal digits (BCD requires 17% more storage capacity than binary) IBM proposed a decimal encoding using DPD (Densely Packed Decimal) [34] to pack 3 decimal digits into 10 bits (storage efficiency of 97.6%). A limitation of this encoding is that arithmetic operations must be performed in other less compact decimal codings such as BCD, but the DPD encoding allows fast and low cost conversions to/from BCD in hardware. To improve software implementations of IEEE 754-2008, Intel proposed an alternate binary encoding, the BID (Binary Integer Decimal) encoding [133], since the DPD to BCD conversions imposed some performance penalties in software. In both cases, the number of digits encoded in the integer coefficient are the same for the three format widths (7 digits for Decimal32, 16 digits for Decimal64 and 34 digits for Decimal128). Format encoding and the decimal arithmetic specifications incorporated in the IEEE 754-2008 are more detailed in Chapter 2.

The IEEE 754-2008 DFP data types are now replacing the old decimal formats in many commercial software products [64] and software libraries for compilers. These include Sun BigDecimal for Java 5 [103], Intel IEEE 754-2008 BID library [31, 30] and IBM decNumber library for ANSI C and C++ [36]. These DFP software packages are currently adequate for some applications, but their performance may not suffice for the increasing demands of multinational corporations and global e-business.

For instance, initial studies from IBM [35, 49] report that some applications spend 50% to 90% of their time in decimal processing because software implementations are typically 100 to 1000 times slower than binary floating-point hardware. They estimate a performance improvement of using DFP hardware for commercial applications from 2× to 10×. Other study [119] compares the performance of a software DFP library (IBM decNumber) running a benchmark (simulating a telephone company) with estimations of latency from available DFP hardware designs [50, 136, 154]. They conclude that hardware implementations of decimal floating-point arithmetic operations are one to two orders of magnitude faster than software implementations. However, a performance analysis reported from Intel [10], shows that, re-

lated to the performance, the overhead of using software decimal implementations in some commercial Java applications is low (less than 4%) and, at least from the point of view of these workloads, there are insufficient performance benefits to use DFP hardware. Moreover, Intel also claims that the speedup of DFP hardware with respect to the Intel BID decimal floating-point library [31, 30] is not as dramatic as reported by IBM using the decNumber library, and that hardware implementations are only interesting if applications spend a large percentage of their time in DFP computations.

These reports indicate that there is a lack of representative workloads to take objective design decisions. A recent work [158] provides a benchmark suite covering a more diverse and broad set of financial and commercial applications, including banking, currency conversion, risk management, tax preparation and telephone billing. Using a similar comparative technique as that reported in [119], they obtain that more than 75% of the execution time when using software libraries (IBM decNumber) is spent in DFP processing, and the use of reference DFP hardware [97, 106, 157, 156] results in speedups ranging from 1.5 to about 30.

From these performance studies, IBM has considered an interesting option to incorporate DFP hardware in their high-performance microprocessors for mainframes and high-end servers. The first compliant IEEE 754-2008 DFU implemented in the IBM Power6 [45, 122] and z10 [160] microprocessors is area efficient, at the cost of a reduction in performance.

On the other hand, Sun and Intel rely on their DFP software libraries and they have not yet announced the incorporation of some hardware acceleration for DFP in their high-end microprocessors.

In addition to the industry efforts, there has been a significant academic research stimulated by the perspective of a new IEEE 754-2008 standard. An important part of the published work is related to the design of high-performance DFX and DFP arithmetic units. Practically all the DFP designs use the IEEE 754-2008 decimal (DPD) encoding, since leads to much faster hardware implementations. For instance, there have been proposals for high-performance integer decimal BCD adders [144] and DFP adders [136, 157], DFX iterative [50, 82] and combinational [91, 148] multipliers, DFP multipliers [97, 67], digit recurrence radix-10 dividers [92, 106, 149] and decimal dividers and a square-root unit based on multiplicative methods [154, 155, 156]. But the design of an efficient multioperand carry free adder [41, 81, 91, 149] has been the most recurrent topic due to its key role to speedup some basic operations as multiplication and division. These designs and other representative examples are described and analyzed in Chapters 3 to 8.

Implementations based on the IEEE 754-2008 binary (BID) encoding [139, 140] propose the reuse of binary hardware as a potential advantage. However, the complexity of performing decimal rounding and alignment over binary coefficients makes BID hardware a tradeoff solution between software implementations and high-performance DPD hardware.

1.4 Current and future trends

In summary, we can conclude that, in effect, there is a generalized interest in providing support for DFP arithmetic by means of software or hardware. This demand obeys primarily

to a need of performance boost for commercial and financial applications which must meet strict accuracy requirements in decimal calculations. The DFP arithmetic defined in the new IEEE 754-2008 standard not only satisfies the rules of decimal arithmetic for finances, but is also adequate for a more general use, as it conforms to the IEEE 854-1987. Thus, IEEE 754-2008 DFP is intended for a broad audience, which includes scientific, engineering, commercial and financial users.

The different preferences of microprocessors manufacturers about a software or a hardware implementation of DFP are probably imposed by the profile of their customers, which have different computational demands. For instance, Sun Niagara [88], Intel Itanium [101] and AMD Opteron [80] microprocessor families for commercial and Web servers are mainly oriented to a broad range of diverse medium scale and distributed computing applications. Currently, they only provide software DFP support, since they have not detected the sufficient performance demand of decimal processing to justify the cost of a dedicated DFU. Otherwise, if market demands more performance, their first option could be to add some hardware support in the binary floating-point unit to speedup the performance of a software DFP (BID) library.

IBM preference for dedicated DFUs is due to its privileged position in the market of mainframes for large scale enterprise computing. The rapid growth of the e-business transactions processed by mainframe software has required major investments in performance. In this way, they have detected an urgent necessity to speedup decimal processing. To meet the requirements of exact decimal computations for commercial and financial uses, the first DFU design [45] supports 34 digits of precision (Decimal128 DFP format). Because of this wide word length (144-bit datapath), the performance of commercial DFUs must be sacrificed to meet the quite exigent area constraints.

By other hand, the current research in this field is focused on providing decimal hardware with the maximum performance by exploiting parallelism and adapting high-performance techniques from binary, aiming to reduce the performance gap with respect to binary floating-point units. Other hot topic of research is the design of efficient pipelined implementations of arithmetic units for future commercial DFUs, that should lead to the build of a decimal fused-multiply-adder [121].

But the future of DFP hardware depends on many factors [79] and is very hard to predict:

- First, DFP hardware should build up enough volume in commercial applications. If software DFP implementations (with some hardware help) became fast enough for commercial applications, probably the performance benefits of dedicated DFUs will be eclipsed by their higher costs, and the investments in chip area will be derived to other uses. Since the preliminary analysis of performance demands do not apport a definitive conclusion in one or other direction, the market will ultimately determine the success or downfall of these early DFUs.

- By other hand, if customers really feel that these processors satisfy their real demands, they will be willing to pay for the extra cost of more performance. Thus, the next generations of DFUs will gradually reduce most of the performance gap with respect to binary hardware. In this sense, the current research is contributing to the necessary advances to minimize this performance gap.

- With the adequate volume of production and the continuous reduction of the scale of integration, high-performance DFP units will be cheap enough for their use in personal computers. Thus, when the performance of DFP units is close to the performance of the binary units, the human preference for decimal representations could favor the use of DFP also for scientific and engineering applications.
- And then, maybe decimal will finally replace binary in all but a few applications which require the superior numerical properties of binary.

Chapter 2

Decimal Floating-Point Arithmetic Units

One of the main drawbacks of decimal arithmetic is that it is less efficient than binary for hardware implementation. Future microprocessors will support a DFP format only if it requires relatively little hardware (memory storage and arithmetic logic) with much better performance than binary for decimal data processing. To be competitive with binary hardware, the efficiency of existing decimal hardware must be improved by means of new algorithms and better architectures.

For this purpose, in this Chapter we consider the improvement of a set of decimal arithmetic operations that maximizes the performance benefits with respect to the costs. This Chapter also covers several issues concerning the design of hardware architectures for DFP arithmetic. First, Section 2.1 summarizes the decimal specifications (both arithmetic and formats) included in the IEEE 754-2008 standard [74], which currently constitutes the reference for DFP implementations. Section 2.2 discusses several factors that influence the efficient implementation of a IEEE 754-2008 compliant DFU. Finally, in Section 2.3 we present a preferred set of decimal arithmetic operations that require improved performance by hardware acceleration.

2.1 IEEE 754-2008 standard for floating-point

The IEEE 754-2008 is the revision to the IEEE 754-1985 standard for binary floating-point arithmetic [72] and the IEEE 854-1987 standard for radix independent floating-point arithmetic [73]. It specifies formats, methods and exception condition handling for binary and decimal floating-point arithmetic in computers. The IEEE 754-2008 standard can be implemented in software, in hardware, or in any combination of both.

Apart from the incorporation of decimal specifications, the main additions with respect to the IEEE 754-1985 are two new 16-bit and a 128-bit binary types, new operations as fused multiply add, recommended correctly rounded elementary functions and a significant clarification in terminology. We only summarize the specifications for decimal floating-point.

Decimal formats and encodings.

One of key points of IEEE 754-1985 is that it includes an explicit representation for the binary formats. On the other hand, the IEEE 854-1987 does not specify any representation

Parameter	DFP format		
	Decimal32 storage	Decimal64 basic	Decimal128 basic
p	7	16	34
e_{max}	+96	+384	+6144
e_{min}	-95	-383	-6143

Table 2.1. *Parameters defining basic and storage format DFP numbers.*

for decimal data, which makes difficult to share decimal numerical data among different machines. Therefore, both decimal formats and encodings (for the decimal interchange formats) are now an integral part of the IEEE 754-2008 standard.

The **formats** defined by the standard are classified as follows:

- **Interchange formats**, which have defined encodings. They are available for storage and for data interchange among platforms. The interchange formats are grouped in:
  - **basic formats**, are the interchange formats available for arithmetic. The standard defines two basic DFP formats of lengths 64 (decimal64) and 128 bits (decimal128).
  - **storage formats**, are narrow interchange formats not required for arithmetic. The standard defines one decimal storage floating-point format of 32 bits length (decimal32).
  - **extended precision formats**, are used to extend a supported basic format by providing wider precision and range.
- **Non-interchange formats**, which are extended precision formats without defined encodings. These formats are not required by the standard, but they can be used for arithmetic. For data interchange, they need to be converted into a interchange format of a suitable extended precision.

A conforming implementation must provide support for at least one basic format (decimal64 and/or decimal128). Each format is characterized by the number of significant digits, p (or precision), and the minimum and maximum exponents e_{max} and $e_{min} = 1 - e_{max}$. The values of the parameters defining the basic and storage interchange DFP formats are shown in Table 2.1.

Signed zero and non-zero DFP numbers are represented in a given format by a sign, an exponent and a not normalized significand or coefficient. Normalization is not required for DFP numbers. It does offer an advantage in binary floating-point, since it decreases the length of the coefficient by one bit (a hidden bit), but not in DFP. Moreover, normalization is incompatible with scaled-integer decimal arithmetic. The standard also includes special representations for two infinities, $+\infty$ and $-\infty$ and two NaNs, qNaN (quiet) and sNaN (signaling). The non-zero DFP numbers with magnitude less than $10^{e_{min}}$ are called subnormal.

A **DFP representation** can be given in a scientific form or in a financial form. In a scientific notation, the DFP number represented by $\{s, e, M\}$ has a value

$$(-1)^s \times M \times 10^e \tag{2.1}$$

where $s \in \{0, 1\}$ is the sign, $e_{min} \leq e \leq e_{max}$ is the integer exponent and $M < 10$ is an unsigned fractional coefficient, not normalized, of the form $M_0.M_1M_2M_3M_4 \ldots M_{p-1}$, $M_i \in \{0, 1, \ldots, 9\}$.

In the financial notation, the not normalized coefficient is interpreted as an integer C. A unsigned integer coefficient allows to integrate the exact integer and the scaled decimal arithmetic systems discussed in Chapter 1 into the standard using a single DFP data type. In this case, the finite DFP numbers represented by $\{s, q, C\}$ have a value of

$$(-1)^s \times C \times 10^q \tag{2.2}$$

where the exponent q (or quantum) is an integer $e_{min} \leq q + p - 1 \leq e_{max}$, and the coefficient C$< 10^p$ is represented by a string of decimal digits of the form $C_0C_1C_2C_3C_4 \ldots C_{p-1}$. The quantum is the same concept as the scale, since it indicates the magnitude of the unit of measurement, such as (10^{-3}) millimeters or (10^{-2}) cents . To preserve the scale (when possible), each operation is defined to have a preferred quantum.

Both forms of representing a finite DFP number (scientific and financial) are equivalent, since $e = q + p - 1$ and $M = C \times 10^{1-p}$. In the subsequent, we use the financial notation to represent the DFP data[4].

Since the coefficient is not normalized, DFP numbers might have multiple representations. The set of different representations of a DFP number is called a cohort. For instance, if C is a multiple of 10 and $q < e_{max}$, then $\{s, q, C\}$ and $\{s, q + 1, C/10\}$ are two representations for the same DFP number and are members of the same cohort.

An **encoding** maps a representation of a floating-point number into a bit string. The standard specifies the layouts for the decimal interchange formats, and allows the integer coefficient to be encoded either in a decimal compressed form (DPD, Densely packed decimal) or in a pure binary form (BID, Binary Integer Decimal). The DPD encoding [34] packs each 3 decimal digits into 10 bits, providing more compression than BCD (1000 combinations out of 1024, or more than 97.6% compression ratio). This allows to encode the same number of digits in a fixed length DFP format as using BID encoding [133], but providing faster conversions to/from BCD (about three simple gate delays in hardware).

A DFP number (a finite DFP number $\{s, q, C\}$, an infinity or a NaN) is encoded in k bits using the following three fields, detailed in Fig. 2.1:

- **A 1-bit sign field**, encoding the sign of the coefficient.
- **A w+5-bit combo field**, comprising the w+2 bit binary biased exponent E = q + *bias* and the 4 most significant bits of the p-digit coefficient. The value of the 2 most significant bits of the exponent cannot be 3.
- **A 10t-bit trailing coefficient field**, encoding p-1= 3×t trailing digits of the p-digit integer coefficient using DPD, or binary integer values from 0 through 2^{10t-1} using BID.

For instance, Table 2.2 shows the values of the format encoding parameters corresponding to the decimal basic and storage formats of Table 2.1. The total number of coefficient digits encoded using DPD or BID is similar in both cases, and equal to p = 3×t +1.

[4]We use quantum to name the exponent and coefficient for the integer significand.

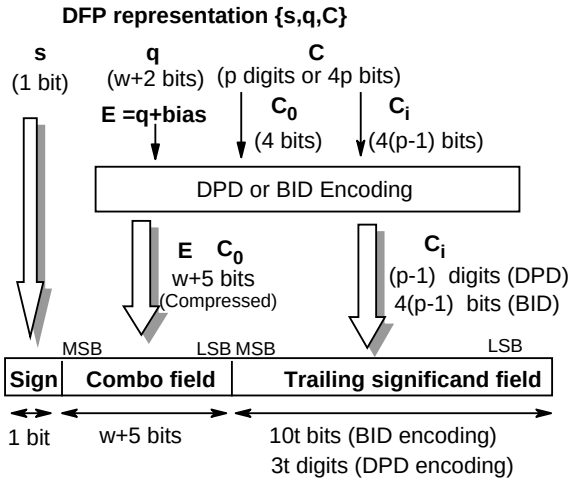


Figure 2.1. *DFP interchange format encodings.*

Parameter	DFP format		
	Decimal32	Decimal64	Decimal128
	storage	basic	basic
k	32	64	128
t	2	5	11
w+5	11	13	17
bias	101	398	6176

Table 2.2. *Encoding parameters for basic and storage formats.*

The specific encoding of the combo field depends on whether the integer coefficient uses the DPD or the BID encoding. If the coefficient uses the DPD encoding, then the 2 MSBs (most significant bits) of the biased exponent and the MSD (most significant digit) of the coefficient are compressed in the first 5 bits of the combo field. The remaining w bits of the combo field contain the trailing exponent bits.

If the coefficient uses the BID encoding, the exponent and the 4 MSBs of the coefficient are determined by the value of the 2 first bits of the combo field:

- If these bits are 00, 01 or 10, the w+2 first bits of the combo field contains the biased exponent E. The leading 4 bits of the BID coefficient are formed concatenating a leading 0 to the 3 LSBs (least significant bits) of the combo field.
- If these bits are 11, then the biased exponent E is encoded from bits 3 to w+4. The 4 MSBs of the coefficient are obtained concatenating '100' and the bit at position w+5 of the combo field.

Decimal arithmetic operations.

The operations required by the standard must be implemented for all the supported formats, in software, in hardware, or in a combined hardware and software solution. Among the decimal operations required are the following:

- Basic arithmetic operations: addition, subtraction, multiplication, division, square-root and fused multiply addition. Correct rounding is required to provide the exactly rounded result for inexact computations.
- Two new decimal-specific operations: samequantum and quantize. The quantize operation is used to scale the numerical value of a DFP representation to a given quantum. It is intended for exact computations, so invalid or inexact exceptions must be signaled. Samequantum compares the quantum of the representation of two DFP numbers.
- Comparisons. These operations compare the numerical values of the decimal operands, and therefore do not distinguish between redundant representations of the same number.
- Different types of conversions:
  - Between integer and floating-point formats.
  - Between different floating-point formats. Conversions between decimal and binary floating point must be correctly rounded.
  - Between floating-point data in internal formats and external string representations.

In addition, the standard recommends to provide correctly rounded implementations of elementary functions, such as exponentials, trigonometric functions and logarithms.

Since a DFP number might have multiple representations (the number's cohort), decimal arithmetic involves not only computing the numerical result but also selecting the proper representation of the number. Thus, each operation is defined to have a preferred quantum (exponent). For all the operations (except for quantize), if a result is inexact, then the cohort member with the least possible quantum is selected to get the longest possible coefficient, using the maximum precision digits available.

If a result is exact, the cohort member with its exponent equal to or closest to the preferred quantum is chosen. The preferred quantum depends on the operation. For decimal addition and subtraction, the preferred quantum is the minimum quantum of the operands, that is q_r = min(q_x, q_y). The preferred quantum for exact decimal multiplication is q_r = q_x+q_y, q_r = q_x-q_y for decimal division, and q_r = floor(q_x/2) for decimal square-root. For exact decimal fused multiply-add computations (R=X$\times$Y+Z), the preferred quantum is q_r = min(q_x+q_y,q_z).

This simplifies the scale preservation in exact computations. For instance, the addition of 10.5 euro (= 105×10^{-1}) and 3.50 euro (= 350×10^{-2}) results in 1400×10^{-2}, and not 14×10^{-1}, preserving the units of measurement (cents) given by the minimum quantum.

Decimal Rounding.

Floating-point numbers can only represent exactly a finite subset of the real numbers. Thus, inexact results must be converted to a close representable DFP number in a given finite precision format. Operations defined by the IEEE 754-2008 must provide correctly (exactly) rounded results. This is equivalent to compute an intermediate result correct to infinite precision (exact result), and then select one of the two possible closest DFP numbers, according to the rounding direction (rounding mode). The standard specifies the following five rounding modes:

- Two rounding modes to the nearest DFP number. These modes deliver the DFP number nearest to the infinitely precise result, differing in case of two DFP numbers equally near:
  - **roundTiesToEven**, round to nearest even, the result is the one with an even least significant digit.
  - **roundTiesToAway**, round to nearest ties away from zero, the result is the one with larger magnitude.
- Three directed rounding modes:
  - **roundTowardPositive**, round towards positive infinity, delivers the closest DFP number greater than the exact result.
  - **roundTowardNegative**, round towards negative infinity, delivers the closest DFP number lower than the exact result.
  - **roundTowardZero**, truncate, delivers the closest DFP number lower in magnitude than the exact result.

For DFP formats, the default rounding mode for results is defined by program, but it is recommended to be roundTiesToEven. The standard also requires to specify separate rounding modes for binary and decimal, so results are rounded according to the corresponding rounding mode of their radix.

Other three rounding modes, not defined by the standard, are often used in DFP implementations:

- **roundTiesToTowardZero**, round to nearest ties towards zero, delivers the nearest DFP with the lower magnitude in case of two DFP numbers equally near.
- **roundAwayZero**, round away from zero, delivers the closest DFP number greater than the exact result if this is positive, or the closest DFP number lower than the exact result if this is negative.
- **roundToVariablePrecision**, round to prepare for a shorter variable precision rounding, delivers a p-digit truncated version of the exact result incremented in one ulp (unit in the last place) when the LSD (least significant digit) is 0 or 5. This mode is used for further rounding to less precision digits.

Exception handling.

When the result of an operation is not the expected floating-point number an exception must be signaled and handled. The default nonstop exception handling uses a status flag to signal each exception and continues execution, delivering a default result. The IEEE 754-2008 standard defines 5 types of exceptions, listed in order of decreasing importance:

- **Invalid operation.** The result of an operation is not defined, e.g., in computations with NaN operands, multiplications of zero by ∞, subtraction of infinities, square-root of negative operands. In this case, the default result is a qNaN that provides some diagnostic information.

- **Division by zero.** The divisor of a divide operation is zero and the dividend is a finite non-zero number. The default result is a signed ∞.
- **Overflow.** The result of an operation exceeds in magnitude the largest finite number representable in the destination format. The default result, determined by the rounding mode and the sign of the result, is either the largest finite number representable or a signed ∞. In addition, an inexact exception is signaled.
- **Underflow.** The result of a DFP operation in magnitude is below $10^{e_{min}}$. This is detected before rounding examining the precision digits and the exponent range. The default result is a p-digit rounded result which can be zero, a subnormal number (with a magnitude $< 10^{e_{min}}$) , or $\pm 10^{e_{min}}$. If the rounded result is not the exact result an inexact exception is signaled.
- **Inexact.** The correctly rounded result of an operation differs from the infinite precision result. The default result is the rounded or the overflowed result.

Optionally, alternative methods for exception handling may be defined by a programming language standard. These mechanisms include traps and other models such as try/catch.

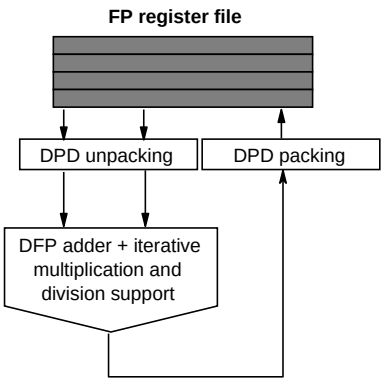
2.2 Decimal floating-point unit design

The proposed IEEE 754-2008 DFP and BFP (binary floating-point) arithmetic systems can be implemented in separated units on a microprocessor, or can share some hardware, in which case there are some possible variations:

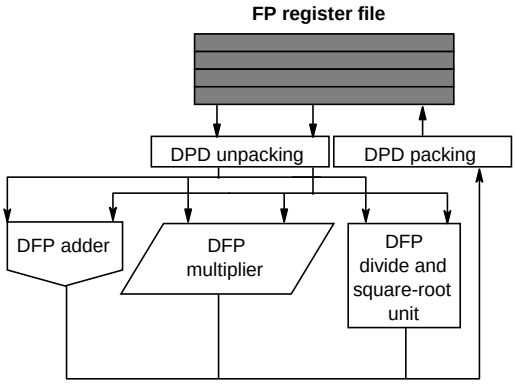
- Implement a BFU (binary floating-point unit) and use a software library for DFP (BID encoding) with some hardware support.
- Implement a mixed binary/decimal (DPD encoding) FPU.
- Implement a DFU and use a software library for BFP with some hardware support.

The use of a single BFU for both floating-point arithmetic systems may be an appropriate solution for reduced hardware cost implementations, since compliant IEEE 754-1985 BFUs are already incorporated in every microprocessor and their adaptation to IEEE 754-2008 is more immediate. In this case, a binary BID encoded coefficient is preferred for DFP implementations. Decimal BCD integers are not efficiently supported in binary units, and the required hardware for operations with binary coefficients can be reused for BFP and DFP. However, rounding and operand shifting by multiples of 10 are more difficult with binary integers. Rounding to a decimal point requires many operations in a radix-2 format, such as leading-ones detection, table lookup, reciprocal multiplication, and trailing zeros detection. This makes the binary BID units [139, 140] significatively slower and, in principle, less adequate for high-performance DFP applications than the decimal DPD units.

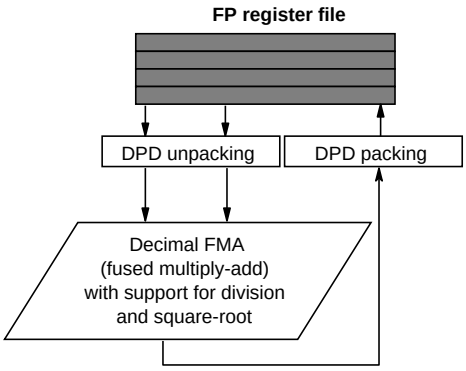
A mixed binary/decimal DPD FPU might reduce the penalty overhead of BID implementations, while also sharing a significant part of the hardware. The BFP and DFP formats are very similar so the register file and the input multiplexors, which expand the data into sign,



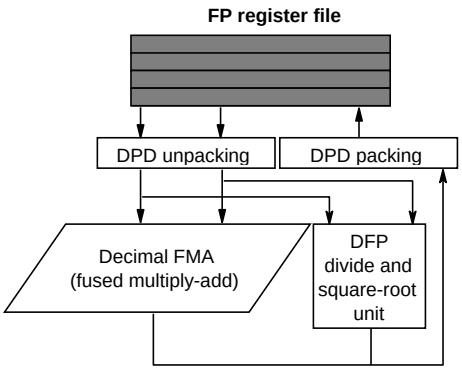
(a) DFP adder with support for other operations.



(b) Separate adder, multiplier and div/sqrt unit.



(c) Combined FMA/div/sqrt unit.



(d) Separate FMA and div/sqrt units.

Figure 2.2. *Configurations for the architecture of the DFU.*

exponent and coefficient, could be shared. The sign and exponent dataflow could also reuse the most part of the hardware. The main problem to overcome is to provide area efficient implementations for the combined BCD/binary coefficient datapath with a very reduced performance overhead with respect to the standalone binary coefficient datapath. Thus, rounding and operand shifting by multiples of the corresponding radix could be done in the same way.

The use of a DFU for general-purpose computations, including some hardware to assist the BFP computations, is limited by the low performance of current implementations, even though all BFP operands have an exact DFP representation. For general-purpose high-performance applications, currently, it may be best to implement dedicated DFUs and BFUs. For a performance/cost tradeoff design a mixed binary/decimal DPD FPU might be an attractive option.

Another issue is the set of operations to implement in hardware. For instance, the Power6 DFU [45, 122] only implements the basic arithmetic operations, add, subtract, multiply and divide and the remaining must be implemented in software. Moreover, the decimal multiplication and division operations use iterative algorithms integrated in the DFP adder datapath. The block diagram of the architecture corresponding to this DFU is shown in Fig. 2.2(a).

The implementation of the different decimal arithmetic units is covered from Chapters 3 to 8. Thought not implemented in the Power6 DFU, the decimal square root operation could

be included in the division recurrence. A commonality in all DPD units is a hardware block for DPD to BCD packing and unpacking, which includes also multiplexors for sign, exponent and coefficient separation. Since DPD encoding cannot be used to perform arithmetic operations, it is necessary to convert the DPD operands to a suitable decimal encoding for arithmetic, primarily BCD. The hardware for encoding 3 BCD digits into 10 bits requires approximately 33 NAND2 gates, and decoding back to BCD about 54 NAND2 gates, with three simple gate delays in both directions. This is quite acceptable for high-performance implementations, moreover when compared with the several cycles of delay of the binary from/to BCD conversions, which depends linearly on the number of digits.

The integration of iterative multiplication in the DFP adder datapath reduces the throughput of the decimal addition/subtraction operation. Another three configurations for high performance DFUs are shown in Figs. 2.2(b), 2.2(c) and 2.2(d). The architecture of Fig. 2.2(b) uses separate units for DFP addition, multiplication and division/square root. The other option is to build a decimal FMA (fused multiply-adder). The advantage of a FMA is that a compiler can make use of the Horners rule to transform a set of equations into a series of multiply-adds, delivering multiply-add operations with a similar throughput than separate units for addition and multiplication. The configuration of Fig. 2.2(c) integrates the division/square-root operations into the FMA using multiplicative based methods, while Fig. 2.2(d) includes a division/square-root unit separated from the FMA.

The research efforts presented in this PhD. thesis are aimed to obtain efficient implementations of both high-performance decimal DPD units for floating-point computation and combined binary/BCD units for coefficient computation. The preferred set of operations to implement in hardware is determined in the next Section.

2.3 Decimal arithmetic operations for hardware acceleration

A set of preferred decimal arithmetic operations has been selected in base to the estimated benefits and costs of a hardware implementation. For this estimation, diverse factors have been taken into account, such as the performance of existing decimal implementations, the relative frequency of the operations in commercial programs, and the contribution to the speedup of the basic arithmetic operations defined in the IEEE 754-2008.

For instance, decimal multiplication is a frequent operation used in finances, e.g. for tax calculation and currency conversion. Current hardware implementations of decimal multiplication are mainly serial and present low performance and throughput. On the other hand, the BFUs of current microprocessors incorporate a binary parallel multiplier, which presents considerably more performance than a serial multiplier. Therefore, an efficient parallel implementation of decimal multiplication could reduce significatively the performance gap between DFUs and BFUs. Although a 34-digit fully combinational parallel implementation is now beyond the scope of commercial DFUs due to area and power constraints, a parallel architecture that could be easily pipelined is interesting to scale the performance for different area and power constraints. Also, a decimal pipelined FMA could be an interesting architecture for future commercial DFUs [120, 121].

In this way, we consider that the preferred decimal operations, covered in Chapters 3 to

8, which need improved hardware implementations are the following:

- Mixed BCD/binary (two-operand) addition/subtraction (Chapters 3 and 4). Previous proposals rely on certain carry tree topologies to improve performance, imposing more area constrains. A more flexible architecture should lead to designs with better area and performance tradeoffs.
- DFP (two-operand) addition/subtraction (Chapter 5). Current DFP adders use two word length carry propagations for BCD coefficient addition and decimal rounding. To improve the performance of DFP adders, a combined BCD adder with rounding should perform this operation in a single carry propagation time delay with a little constant overhead, as in binary.
- Decimal carry-free multioperand addition/subtraction (Chapter 6). This operation is the base to speed up parallel multiplication and radix-10 division based on subtractive methods. Previous methods use either trees of radix-10 carry-save adders, slower than binary CSA (carry-save adders), or binary CSAs directly over BCD operands, requiring time-consuming corrections of invalid BCD digit representations. In either case, a binary CSA multioperand tree presents much better performance and area figures than the equivalent decimal tree, and this significant gap should be reduced.
- Fixed-point/integer decimal and mixed binary/decimal multiplication (Chapter 7).
- DFP multiplication (Chapter 7).
- Decimal fused multiply-addition (Chapter 7).
- Fixed-point decimal division (Chapter 8). A decimal divider adequate for a high performance DFU should use a separate datapath from multiplication and addition and should present a reduced area implementation able to generate at least a radix-10 quotient digit per cycle. A normalized fixed-point unit could support both DFP and integer division adding a few extra hardware for operand alignment and sign and exponent calculation. This unit should be easily extended to compute decimal square-roots.

Chapter 3

10's Complement BCD Addition

The addition of two decimal integer operands (usually represented in BCD) is a basic operation in decimal fixed and floating-point computations such as non-redundant coefficient addition, assimilation of a decimal redundant operand and rounding. This Chapter deals with the methods and architectures for 10's complement carry-propagate addition/subtraction of BCD operands. Sign-magnitude BCD addition/subtraction is considered in Chapter 4. By other hand, carry-free decimal multioperand integer addition/subtraction is covered in Chapter 6.

We introduce the architectures of several high-performance 10's complement BCD and mixed 2's complement binary/10's complement BCD adders. These adders are based on a carry-propagate algorithm for decimal addition [144] that shows a lower dependency on the carry-tree topology than previous methods. This algorithm gives more flexibility to the designer to choose the adder architecture and the area/latency trade-offs and also allows for efficient implementations of mixed binary/decimal addition.

We provide implementations using different representative high-performance prefix tree and hybrid carry-select/prefix tree structures [86, 89, 100]. We also present a simpler reformulation of Ling addition as a prefix computation [147] and the subsequent implementation for the proposed architectures.

Another important issue is that users of financial and e-commerce services demand a high degree of reliability. On the other hand, soft errors[5] are becoming more significative due to the higher densities and reduced sizes of transistors on a chip [102, 105]. In this context, we introduce a scheme for BCD sum error checking that avoids the replication of arithmetic units without sacrificing significant area or performance.

To compare the potential performance and/or cost advantages of our proposals with respect to other different representative algorithms and architectures, we have developed a rough area and delay evaluation model for CMOS circuits based on logical effort [131]. We provide area and delay estimations for both proposed and other academical and patented implementations, most of them currently in use in industrial designs.

The Chapter is organized as follows. Section 3.1 outlines some representative carry-propagate methods to compute 10's complement addition/subtraction in hardware. In Section 3.2 we introduce a method to improve decimal carry-propagate computations for 10's com-

[5]Temporary circuit failures caused by high-energy transient particles that can lead to incorrect results.

Dec. digit	BCD digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Invalid	1010
Invalid	1011
Invalid	1100
Invalid	1101
Invalid	1110
Invalid	1111

Table 3.1. *Conventional BCD coding.*

plement BCD addition. In Section 3.3 we provide several implementations for the proposed 10's complement BCD adder and the mixed 2's complement/10's complement adder using different representative prefix carry tree topologies. The proposed scheme for BCD sum error checking is discussed in Section 3.4. The area and delay evaluation results and comparisons are shown in Section 3.5. Finally, the conclusions are summarized in Section 3.6.

3.1 Previous work on BCD addition/subtraction

We only consider carry-propagate methods for the addition/subtraction of two decimal signed integer operands, $S = X \pm Y$. Although redundant addition has a lower delay independent of the number of operand digits, it is best suited for multioperand addition, since the conversion to a non-redundant representation presents a similar delay than a carry propagate addition. Also, we assume non-redundant decimal operands coded in BCD. The i^{th} decimal digit $X_i \in [0, 9]$ of an operand X is represented in BCD as [6]

$$X_i = \sum_{j=0}^{3} x_{i,j} \cdot 2^i \tag{3.1}$$

where $x_{i,j} \in \{0, 1\}$ is the j^{th} bit of the weighted 4-bit vector ($x_{i,3}$,$x_{i,2}$,$x_{i,1}$,$x_{i,0}$), that is, the BCD digit i. Note that only 10 out of the 16 possible combinations of 4 bits represent a valid BCD digit, as shown in Table 3.1.

Thus, a p-BCD digit unsigned integer operand X is represented as

$$X = \sum_{j=0}^{p-1} X_j \cdot 10^j = \sum_{j=0}^{p-1} \Big(\sum_{k=0}^{3} x_{j,k} \cdot 2^k \Big) \cdot 10^j \tag{3.2}$$

[6] We use capital letters to represent decimal digits, and small letters for single bits.

To represent a signed BCD number $Y \equiv (y_p, Y_{p-1}, Y_{p-2}, \ldots, Y_0)$ (an additional bit y_p is required for sign) we consider several possibilities:

- **True and complement systems.** The most commonly used true and complement systems are 9's complement and 10's complement:
  - **9's complement.** The signed BCD number Y in 9's complement has a value equal to
$$Y = -y_p \cdot (10^p - 1) + \sum_{i=0}^{p-1} Y_i \cdot 10^i \tag{3.3}$$
where
$$y_p = \begin{cases} 0 & \texttt{If } (Y \geq 0) \\ 1 & \texttt{Else} \end{cases} \tag{3.4}$$
is the sign bit and $-(10^p - 1) \leq Y \leq 10^p - 1$.
The value $-Y$ is obtained by bit complementing each BCD digit plus 6 (modulo 16) and inverting the sign bit (9's complement), that is
$$-Y = -\overline{y_p}\ (10^p - 1) + \sum_{i=0}^{p-1} \neg Y_i \cdot 10^i \tag{3.5}$$
where
$$\neg Y_i = 9 - Y_i = 15 - (Y_i + 6) = \overline{Y_i + 6} \tag{3.6}$$
  - **10's complement.** Y is equal to
$$Y = -y_p \cdot 10^p + \sum_{i=0}^{p-1} Y_i \cdot 10^i \tag{3.7}$$
where $-10^p \leq Y \leq 10^p - 1$. The value $-Y$ is obtained from its 9's complement plus one:
$$-Y = -\overline{y_p} \cdot 10^p + \sum_{i=0}^{p-1} \neg Y_i \cdot 10^i + 1 \tag{3.8}$$
- **Sign-magnitude system.** The BCD digit vector $Y \equiv (y_p, Y_{p-1}, Y_{p-2}, \ldots, Y_0)$ has an arithmetic value equal to $Y = (-1)^{y_p}\ |Y|$, where $|Y| = \sum_{i=0}^{p-1} Y_i\ 10^i$. The value $-Y$ is obtained by a simple inversion of the sign bit y_p.

The best choice for an efficient implementation of signed decimal integer and fixed-point addition/subtraction is 10's complement, since subtraction is simply performed by adding the 10's complement of the subtrahend [117]. Although sign-magnitude subtraction is more complex, it is required for IEEE 754-2008 decimal floating-point computations, so it is discussed in Chapter 4. In the remaining Section we analyze several representative carry-propagate methods for 10's complement addition/subtraction.

3.1.1 Basic 10's complement algorithm

The carry propagate algorithm for 10's complement addition/subtraction [113, 117] is described in Fig. 3.1, where $C_i \in \{0, 1\}$ is the input decimal carry[7] to decimal position i. If an

[7]Although decimal carries can be also represented as single bits, we consider them to be decimal digits.

[Algorithm: 10's complement Addition/Subtraction(S=X±Y)]

Inputs: $X := -x_p \cdot 10^p + \sum_{i=0}^{p-1} X_i \cdot 10^i$, $Y := -y_p \cdot 10^p + \sum_{i=0}^{p-1} Y_i \cdot 10^i$

$$C_0 := \begin{cases} 1 & \texttt{If(op == sub)} \\ 0 & \texttt{Else} \end{cases}$$

For (i:=0;i<p;i++){

$$Y_i^* = \begin{cases} \neg Y_i & \texttt{If(op == sub)} \\ Y_i & \texttt{Else} \end{cases}$$

$$C_{i+1} = \lfloor (X_i + Y_i^* + C_i)/10 \rfloor$$

$$S_i = \mathrm{mod}_{10}(X_i + Y_i^* + C_i)$$

}

$$s_p = \begin{cases} 1 & \texttt{If (op == add) and } (x_p == \overline{y_p} == 1 \texttt{ or } (x_p != y_p) \texttt{ and } C_p == 0) \\ 1 & \texttt{Else If (op == sub) and } (x_p == y_p == 1 \texttt{ or } (x_p == y_p) \texttt{ and } C_p == 0) \\ 0 & \texttt{Else} \end{cases}$$

Figure 3.1. *10's complement Addition/Subtraction Algorithm.*

overflow does not occur, then the logical implementation of the sign bit is straightforward[8]:

$$s_p = x_p \oplus y_p \oplus sub \oplus C_p \tag{3.9}$$

where sub indicates the selected operation ($sub = 1$ for subtraction). An overflow occurs when the following expression is evaluated as true:

$$overflow \Leftrightarrow x_p \ (y_p \oplus sub) \ \overline{C_p} \vee \overline{x_p} \ \overline{y_p \oplus sub} \ C_p \tag{3.10}$$

In this case, the sign bit s_p is given by:

$$s_p = x_p \ (y_p \oplus sub) \vee (x_p \oplus y_p \oplus sub) \ \overline{C_p} \tag{3.11}$$

In the rest of the Chapter, we only deal with the addition of the trailing p-digits. The main limitation of this algorithm is, obviously, the carry propagation dependency. Moreover, in binary systems, radix-10 operations, such as module 10 addition and integer division by 10, are not implemented as efficiently as radix-2^n ($n > 0$) operations. For instance, BCD addition is more complicated than binary addition, since six of the sixteen possible 4-bit combinations are unused ('1010' to '1111'), and they must be skipped. Thus, binary carry-propagate algorithms cannot be used directly to implement decimal addition. There are mainly two strategies to implement 10's complement addition:

- *Direct Decimal Addition* [25, 50, 118]. The decimal carries C_i are obtained by a direct implementation of a decimal carry propagate recurrence which indicates the conditions for the generation and propagation of decimal carries for each digit.
- *Speculative Decimal Addition* [14, 19, 20, 63, 117]. The bit-vector representation of decimal operands is modified to allow the computation of decimal addition as an hexadecimal

[8] We represent the logical OR by $\vee$, the logical AND by an empty space between signals and the logical XOR by $\oplus$.

[Algorithm: Direct Decimal Addition/Subtraction (S=X±Y)]

Inputs: $X := \sum_{i=0}^{p-1} X_i \cdot 10^i$, $Y := \sum_{i=0}^{p-1} Y_i \cdot 10^i$

$$C_0 := \begin{cases} 1 & \texttt{If(op == sub)} \\ 0 & \texttt{Else} \end{cases}$$

For (i:=0;j<p;i++){

$$Y_i^* = \begin{cases} \overline{Y_i + 6} & \texttt{If(op == sub)} \\ Y_i & \texttt{Else} \end{cases}$$

$$G_i = \begin{cases} 1 & \texttt{If}(X_i + Y_i^* > 9) \\ 0 & \texttt{Else} \end{cases} \qquad A_i = \begin{cases} 1 & \texttt{If}(X_i + Y_i^* \geq 9) \\ 0 & \texttt{Else} \end{cases}$$

$$C_{i+1} = \lfloor (X_i + Y_i^* + C_i)/10 \rfloor = G_i \vee A_i \; C_i$$

$$S_i = \mathrm{mod}_{10}(X_i + Y_i^* + C_i) = \mathrm{mod}_{16}\left(X_i + Y_i^* + C_i + 6 \cdot C_{i+1}\right)$$

}

Figure 3.2. *Direct Decimal Algorithm.*

(radix-16) addition and a decimal correction. Since the bit-vectors of the hexadecimal and binary representations are identical, this permits the use of the same binary addition techniques.

These methods can be implemented in a serial (linear delay) or parallel (logarithmic delay) configuration. We only consider parallel implementations for high-performance applications. Moreover, the resulting architectures need to support wide word lengths (up to 144 bits or 34 BCD digits).

3.1.2 Direct decimal addition

The direct decimal addition method implements 10's complement addition using the carry recurrence $C_{i+1} = G_i \vee A_i \; C_i$, as is described in Fig. 3.2. The decimal carry generate G_i and decimal carry alive A_i functions, defined in Fig. 3.2, state the conditions for the generation and propagation of a decimal carry to the next decimal position $i+1$. Their logical expressions only depend on the bit-vector representation of X_i and Y_i^*. We use the decimal carry alive function instead of decimal carry propagate, defined as $P_i = 1 \;\; \texttt{If}(X_i + Y_i^* = 9)$, due to its simpler logical implementation. For BCD, G_i and A_i are given by

$$\begin{aligned} G_i &= G_i^U \vee A_i^U \; x_{i,0} \; y_{i,0}^* \\ A_i &= A_i^U \; (x_{i,0} \vee y_{i,0}^*) \end{aligned} \tag{3.12}$$

where the upper decimal carry-generate (G_i^U) and carry-alive (A_i^U) functions are obtained from the three most significant bits of each input digit as

$$\begin{aligned} G_i^U &= x_{i,3} \; (y_{i,3}^* \vee y_{i,2}^* \vee y_{i,1}^*) \vee y_{i,3}^* \; (x_{i,2} \vee x_{i,1}) \vee x_{i,2} \; y_{i,2}^* \; (x_{i,1} \vee y_{i,1}^*) \\ A_i^U &= x_{i,3} \vee y_{i,3}^* \vee x_{i,2} \; y_{i,2}^* \vee (x_{i,2} \vee y_{i,2}^*) \; x_{i,1} \; y_{i,1}^* \end{aligned} \tag{3.13}$$

A_i and G_i can be expressed in terms of the binary carry-generate ($g_{i,j} = x_{i,j}\ y^*_{i,j}$), and binary carry-alive signals ($a_{i,j} = x_{i,j} \vee y^*_{i,j}$), as $G_i = G^U_i \vee A^U_i\ g_{i,0}$, and $A_i = A^U_i\ a_{i,0}$, where

$$\begin{aligned} G^U_i &= g_{i,3} \vee g_{i,2}\ a_{i,1} \vee a_{i,3}\ (a_{i,2} \vee a_{i,1}) \\ A^U_i &= a_{i,3} \vee g_{i,2} \vee a_{i,2}\ g_{i,1} \end{aligned} \tag{3.14}$$

The direct decimal carry propagate recurrence $C_{i+1} = G_i \vee A_i\ C_i$ can be evaluated using conventional high-performance parallel carry evaluation techniques such as prefix tree [84, 86, 89]. We consider the carry look-ahead [162] and Ling carry [95] recurrences as particular cases of prefix carry tree computations. In Section 3.3 we present a unified framework for parallel carry computation [147], which reformulates most of the carry-propagate adders as prefix tree adders.

By other hand, the BCD sum digits S_i are given by

$$S_i = \mathrm{mod}_{16}\big(X_i + Y^*_i + C_i + 6 \cdot C_{i+1}\big) \tag{3.15}$$

That is, the BCD (modulo 10) sum is equivalent to a hexadecimal (modulo 16) sum corrected by a +6 factor when a decimal carry-out is produced ($X_i + Y^*_i + C_i \geq 10$). The expression (3.15) can be evaluated either directly from X_i, Y_i and the computed carries using combinational logic, as it was originally proposed in [118], or using two-conditional 4-bit presums with a carry-select output, as proposed in [91, 144]. This leads to two distinct architectures for 10's complement adders based on direct decimal addition.

In the first case, the BCD sum digits are obtained by splitting the direct decimal digit recurrence $C_{i+1} = G_i \vee A_i\ C_i$ into two recurrences as

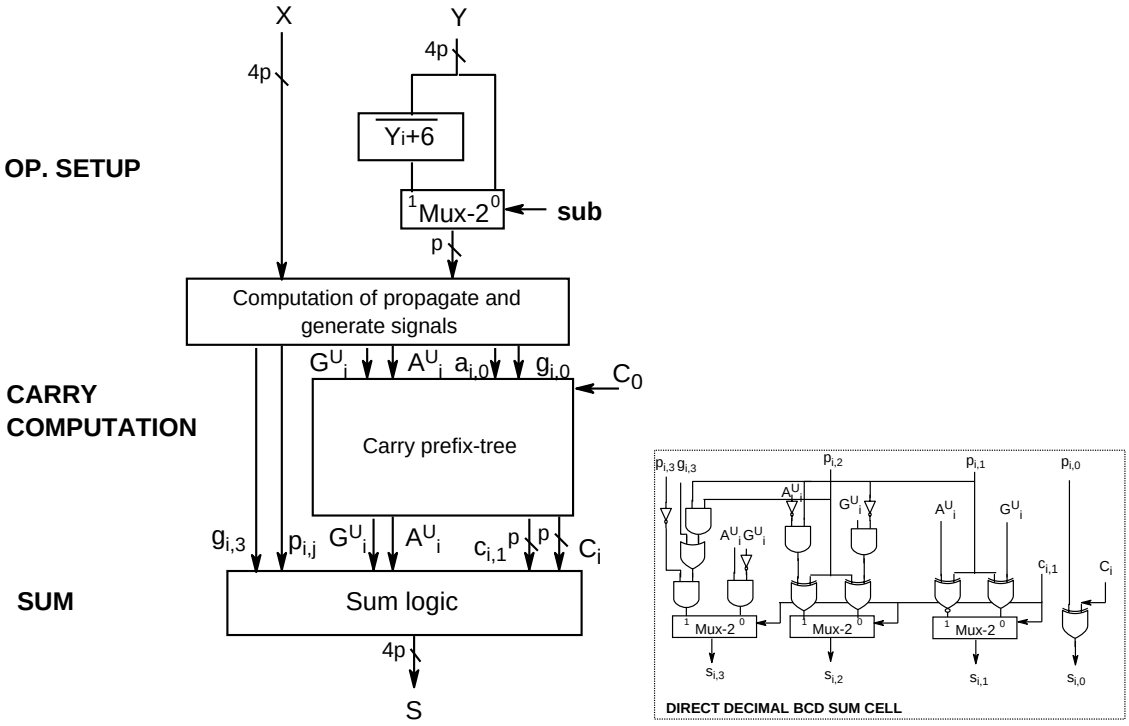
$$\begin{aligned} c_{i,1} &= g_{i,0} \vee a_{i,0}\ C_i \\ C_{i+1} &= G^U_i \vee A^U_i\ c_{i,1} \end{aligned} \tag{3.16}$$

and introducing the expression for C_{i+1} in (3.15). After expanding and reorganizing terms, we obtain the following logical expressions for the bits of S_i:

$$S_i = \begin{cases} s_{i,3} = (g_{i,3} \vee p_{i,2}\ p_{i,1})\ \overline{p_{i,3}}\ c_{i,1} \vee A^U_i\ \overline{G^U_i}\ \overline{c_{i,1}} \\ s_{i,2} = \overline{(p_{i,2} \oplus p_{i,1}\ \overline{A^U_i})}\ c_{i,1} \vee (p_{i,2} \oplus \overline{p_{i,1}}\ G^U_i)\ \overline{c_{i,1}} \\ s_{i,1} = p_{i,1} \oplus \overline{A^U_i}\ c_{i,1} \vee (p_{i,1} \oplus G^U_i)\ \overline{c_{i,1}} \\ s_{i,0} = p_{i,0} \oplus C_i \end{cases} \tag{3.17}$$

where the terms $p_{i,j} = x_{i,j} \oplus y^*_{i,j}$ are the binary carry propagates. BCD carry-lookahead adders using similar equations [50, 118] are implemented in the functional units of the IBM G4, G5 and G6 S/390 microprocessors [25]. Fig. 3.3(a) shows a prefix tree 10's complement BCD adder based on this algorithm. We distinguish the following stages: operand setup, carry evaluation and sum. The 9's complement of Y is evaluated in the operand setup stage as $\neg Y = \sum_{i=0}^{p-1} \overline{Y_i + 6} \cdot 10^i$ in a digitwise form (no carry is propagated between digits). This operation (the +6 digitwise increment and the bit inversion) only requires a simple logic stage, that is

$$\overline{Y_i + 6} = \begin{cases} \overline{y_{i,3} \vee y_{i,2} \vee y_{i,1}} \\ y_{i,2} \oplus y_{i,1} \\ y_{i,1} \\ \overline{y_{i,0}} \end{cases}$$



(a) Block diagram of the architecture. (b) BCD (4-bit) direct decimal sum cell.

Figure 3.3. *Direct decimal adder with direct BCD sum.*

(3.18)

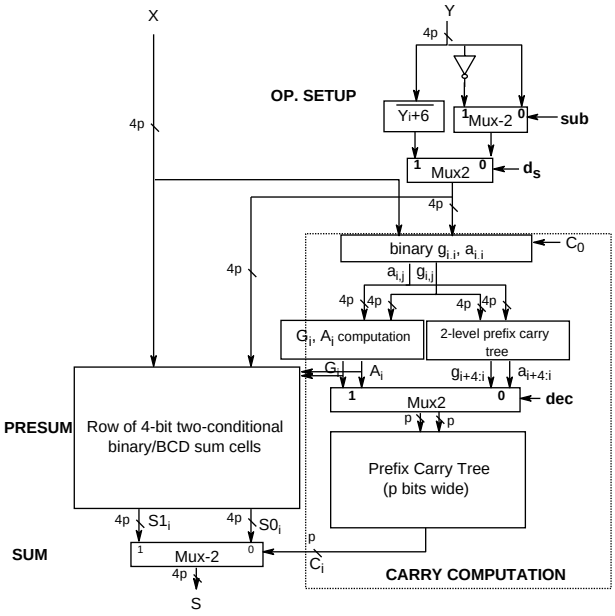
The 10's complement of Y is formed selecting the 9's complement of Y ($\neg Y$) through a row of 2:1 multiplexes controlled by the operation signal sub ($sub = 1$ for subtraction) and setting the carry input C_0 to 1 (we use $C_0 = sub$).

The carry computation is performed in two steps. First, the carry-generate functions G_i^U, $g_{i,0}$ and carry-alive functions A_i^U, $a_{i,0}$ are evaluated as shown in Equation (3.13). In addition, the binary carry-propagate $p_{i,j}$ and the binary carry-generates $g_{i,3}$ are also computed for further use in the sum stage. Next, the decimal carries C_i and the binary carries $c_{i,1}$ (carries produced from bit one of each digit), are computed in a prefix tree of 2p-bit wide[9] that implements the recurrences (3.16). After the carry computation, the sum digits are obtained using a row of the BCD sum cells of Fig. 3.3(b), which implement equation (3.17).

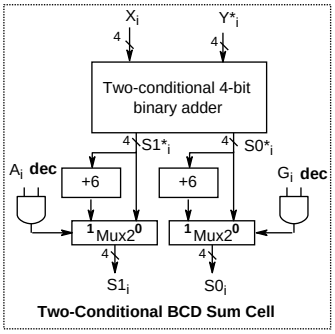
The second type of direct decimal architectures [91, 144] uses a hybrid adder configuration [159], which combines a prefix carry tree and a carry-select sum stage. In this case, the appropriate sum digit is selected from the corresponding decimal carry C_i as $S_i = S1_i\ C_i \vee S0_i\ \overline{C_i}$, where the pre-sum digits $S1_i$, $S0_i$ are computed as

$$S1_i = \mathrm{mod}_{10}(X_i + Y_i^* + 1) = \begin{cases} \mathrm{mod}_{16}\big((X_i + Y_i^* + 6) + 1\big) & \texttt{If}(A_i == 1) \\ \mathrm{mod}_{16}\big((X_i + Y_i^*) + 1\big) & \texttt{Else} \end{cases}$$

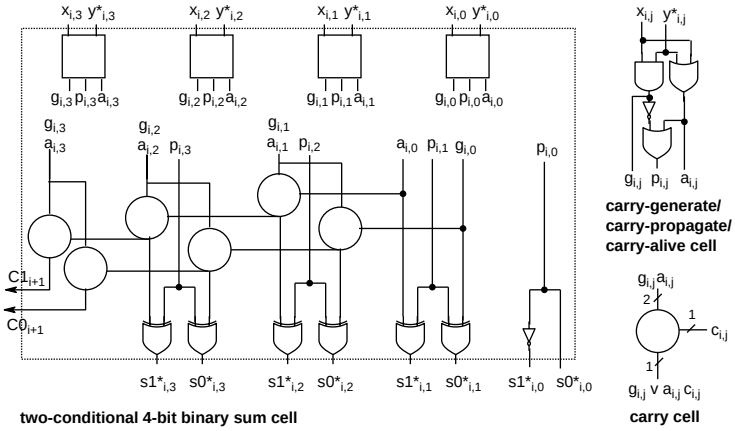
[9] A detailed description of parallel prefix carry computation is presented in Section 3.3



(a) Block diagram of the architecture.



(b) Two-conditional sum cell.



(c) Two-conditional 4-bit binary adder.

Figure 3.4. *Mixed binary/direct decimal adder using a hybrid configuration.*

$$S0_i = \mathrm{mod}_{10}\big(X_i + Y_i^*\big) = \begin{cases} \mathrm{mod}_{16}\big(X_i + Y_i^* + 6\big) & \texttt{If}(G_i == 1) \\ \mathrm{mod}_{16}\big(X_i + Y_i^*\big) & \texttt{Else} \end{cases} \tag{3.19}$$

using a row of two-conditional 4-bit adders (modulo16 adders) and additional hardware for the +6 BCD digit corrections.

Moreover, this hybrid configuration is more suitable to implement mixed binary/direct decimal adders. Fig. 3.4(a) shows the architecture of a mixed binary/direct decimal adder with the following stages: operand setup, pre-sum, carry evaluation and sum. In the operand setup stage Y_i^* is evaluated as

$$Y_i^* = \begin{cases} \overline{Y_i + 6} & \texttt{If(op == sub \& dec)} \\ \overline{Y_i} & \texttt{Else If(op == sub \& not(dec))} \\ Y_i & \texttt{Else} \end{cases} \tag{3.20}$$

Control signal d_s selects $\overline{Y+6}$ for decimal subtraction. The control signal dec is enabled for decimal operations. Another control signal, sub is used to select the 2's complement of Y in case of binary subtractions. To exploit some hardware sharing in mixed binary/direct decimal implementation, G_i^U and A_i^U are evaluated according to Equation (3.14). Next, G_i and A_i are evaluated as $G_i = G_i^U \vee A_i^U\ g_{i,0}$ and $A_i = A_i^U\ a_{i,0}$. In parallel with the evaluation of the $G_i's$ and $A_i's$, a two-level prefix tree computes, for binary operations, the block carry-generate ($g_{i+4:i}$) and block carry-alive ($a_{i+4:i}$) of each group of 4 bits as

$$(g_{i+4:i}, a_{i+4:i}) = \prod_{j=0}^{3} (g_{i,j}, a_{i,j}) \tag{3.21}$$

where the product uses the prefix operator $\bullet$ defined as (see Section 3.3 for more detail):

$$(g_{i,j}, a_{i,j}) \bullet (g_{i,j-1}, a_{i,j-1}) = (g_{i,j} \vee a_{i,j}\ g_{i,j-1}, a_{i,l}\ a_{i,j-1}) \tag{3.22}$$

In this way, the remaining levels of the prefix carry tree can be shared for decimal and binary operations.

In the pre-sum stage, $S0_i$ and $S1_i$ are computed for the two possible values of C_i using a row of the two-conditional mixed binary/BCD sum cells of Fig. 3.4(b). Each two-conditional mixed sum cell consists of the two-conditional 4-bit adder shown in Fig. 3.4(c) and two conditional +6 digit increment blocks. The decimal carry generate and carry alive signals computed in the carry stage are used to obtain the conditional digit carry outputs, $C1_{i+1} = A_i$ and $C0_{i+1} = G_i$, since $C_{i+1} = G_i \vee A_i\ C_i$. The conditional carry outputs determines when the conditional 4-bit binary pre-sums $S1_i^*$ and $S0_i^*$ need to be biased by 6, producing the BCD sum digits $S1_i$ and $S0_i$ according to Equations (3.19). Finally, in the sum stage, the decimal carries C_i computed in the parallel prefix tree, select the appropriate binary/BCD sum digits $S0_i$ or $S1_i$ using the control signals $A_i\ dec$ and $G_i\ dec$. For binary operations, we select $S1_i = S1_i^*$ and $S0_i = S0_i^*$ with $dec = 0$.

3.1.3 Speculative decimal addition

The advantage of decimal speculative methods is their simple implementation in a binary carry-propagate adder since direct decimal addition requires dedicated combinational logic to produce generate and propagate (or alive) signals and sum digits. Because of this, decimal speculative methods are also best suited for implementing combined 2's complement binary/10's complement BCD adders [144]. However, unlike direct decimal addition, they may require a sum digit correction after carry evaluation. To avoid an overhead delay in the critical path due to this decimal correction, several high-performance implementations use an hybrid sparse prefix tree/carry-select topology [14, 63].

The algorithm for speculative addition is shown in Fig. 3.5. When the sum of two BCD digits is higher than 9, the corresponding carry-out C_{i+1} must be set to one and the invalid 4-bit sum vector corrected. To accomplish this, the speculative decimal addition method unconditionally increments by 6 each decimal digit position, performs a carry-propagate binary addition S^* and then correct the speculative sum digit S_i^* (subtracting 6) if the corresponding carry-out C_{i+1} is zero.

[Algorithm: Speculative Decimal Addition/Subtraction (S=X±Y)]

Inputs: $X := \sum_{i=0}^{p-1} X_i \cdot 10^i$, $Y := \sum_{i=0}^{p-1} Y_i \cdot 10^i$

$$C_0 := \begin{cases} 1 & \texttt{If(op == sub)} \\ 0 & \texttt{Else} \end{cases}$$

For (i:=0;i<p;i++){

$$Y_i^* = \begin{cases} \overline{Y_i + 6} & \texttt{If(op == sub)} \\ Y_i & \texttt{Else} \end{cases}$$

1. 4-bit binary carry propagate addition:

$$c_{i+1,0} = \lfloor (X_i + Y_i^* + 6 + C_i)/16 \rfloor$$
$$C_{i+1} = \lfloor (X_i + Y_i^* + C_i)/10 \rfloor = c_{i+1,0}$$

$$S_i^* = \mathrm{mod}_{16}(X_i + Y_i^* + 6 + c_{i,0})$$

2. Decimal correction:

$$S_i = \mathrm{mod}_{10}(X_i + Y_i^* + C_i) = \begin{cases} \mathrm{mod}_{16}(S_i^* - 6) & \texttt{If } C_{i+1} == 0 \\ S_i^* & \texttt{Else} \end{cases}$$

}

Figure 3.5. *Decimal Speculative Algorithm.*

Since the decimal carries are generated when $X_i + Y_i^* + C_i > 9$, they have the same value as the corresponding binary carries in the same position ($c_{i+1,0}$), generated when $X_i + Y_i^* + 6 + C_i > 15$ (hexadecimal carries). Thus, the addition of 6 in each digit position allows to use any high-speed binary carry propagation technique for the evaluation of C_{i+1}, for instance, a prefix carry tree.

For the evaluation of S_i there are two possibilities:

- Using a full binary prefix carry tree to obtain all the binary carries. The speculative sum digits S_i^* are obtained from the XOR operation over the bit vectors of the +6 biased input operands and the binary carries. A post-correction stage, placed after carry evaluation, is necessary to obtain the appropriate BCD sum digits S_i by correcting the wrong speculative sum digits S_i^* when C_{i+1} is zero.

- Using an hybrid prefix carry tree/carry-select adder. For an efficient implementation, an appropriate carry tree topology is a quaternary prefix tree (QT) [100], that is, a sparse prefix carry tree that generates only one carry (decimal carry) for every 4 bits. In parallel with the carry computation, the conditional BCD sum digits $S1_i$ and $S0_i$ are evaluated in the pre-sum stage for each possible value of the decimal carry input ($S1_i$ for $C_i == 1$ and $S0_i$ for $C_i == 0$). In the final sum stage, the decimal carries C_i computed in the quaternary tree select the appropriate sum digits as $S_i = S1_i \; C_i \vee S0_i \; \overline{C_i}$.

The initial $+6$ digit additions are performed digitwise using a single level of combinational

logic as

$$X_i + 6 = \begin{cases} x_{i,3} \vee x_{i,2} \vee x_{i,1} \\ \overline{x_{i,2} \oplus x_{i,1}} \\ \overline{x_{i,1}} \\ x_{i,0} \end{cases} \tag{3.23}$$

where $X_i + 6 \in [6, 15]$ are represented in BCD excess-6.

Instead of biasing X, the $+6$ digitwise additions can also be implemented as a $+6$ digit biased operand Y^*, that is,

$$Y_i^* + 6 = \begin{cases} \overline{Y_i} & \texttt{If(op == sub)} \\ Y_i + 6 & \texttt{Else} \end{cases} \tag{3.24}$$

or alternatively, biasing X for decimal addition and Y^* for decimal subtraction. Another proposal [136], computes the $+6$ digit additions representing the input digits in BCD excess-3, as $X_i + 3 \in [3, 12]$ and

$$Y_i^* + 3 = \begin{cases} \overline{Y_i + 3} & \texttt{If(op == sub)} \\ Y_i + 3 & \texttt{Else} \end{cases} \tag{3.25}$$

Nevertheless, the resultant implementations are similar.

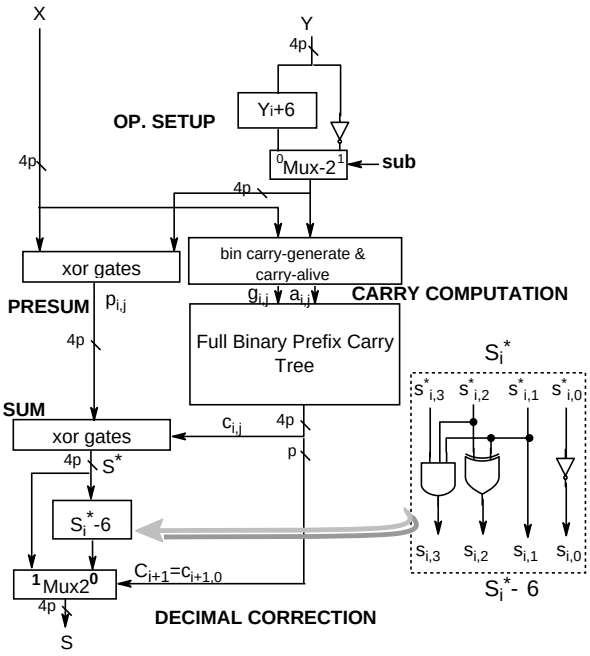
Fig. 3.6(a) shows a block diagram of a 10's complement BCD prefix tree adder using decimal speculative addition with a decimal post-correction scheme [117]. The architecture is structured in operand setup, binary carry computation, pre-sum (a level of xor gates), sum (another level of xor gates) and post-correction stages. The prefix carry tree computes the full $4p$ binary carries. The decimal carry-outs C_{i+1} correspond to the binary caries $c_{i+1,0}$. Post-correction takes place after the sum using a row of multiplexes to select between $S_i = S_i^*$ when $C_{i+1} = 1$ or $S_i = S_i^* - 6$ when C_{i+1} is zero. The +6 digitwise subtractions (equivalent to +10 (modulo 16) additions) are implemented as

$$S_i^* - 6 = \begin{cases} s_{i,3}^* \; s_{i,2}^* \; s_{i,1}^* \\ s_{i,2}^* \oplus s_{i,1}^* \\ \overline{s_{i,1}^*} \\ s_{i,0}^* \end{cases} \tag{3.26}$$

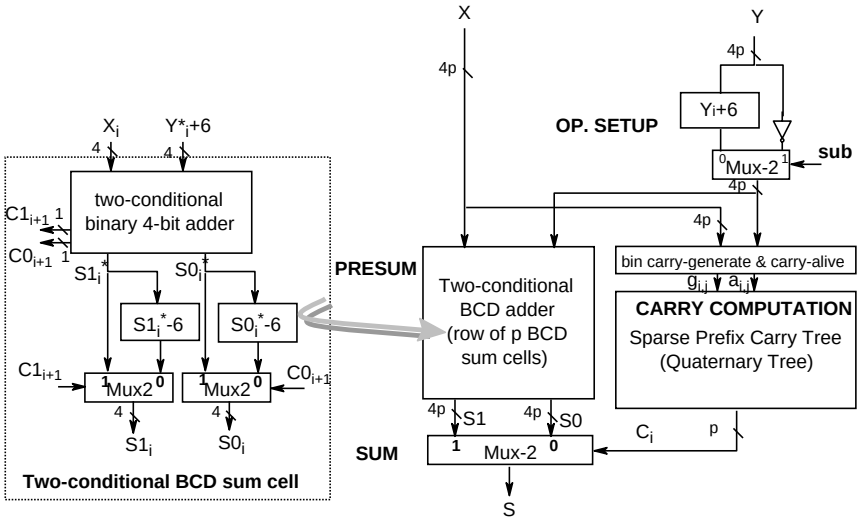
The critical path delay is composed of the delay of the operand setup, the binary carry computation, the sum and the post-correction stages.

To speed up speculative decimal addition, the evaluation and correction of BCD sum digits may be performed in parallel to the carry computation using an hybrid prefix tree/carry-select adder. BCD and mixed binary/BCD hybrid adders are implemented in the fixed-point units of the IBM z900 and z990 microprocessors [19, 20] (described in more detail in [14, 63]).

Fig. 3.6(b) shows the general block diagram of the BCD architecture and the layout of a two-conditional BCD sum cell. The prefix tree is implemented as a sparse quaternary tree which computes only the decimal carries C_i. In this case, the decimal correction is included in the pre-sum stage out of the critical path (carry path). It is performed subtracting +6 to



(a) Full binary prefix carry tree.

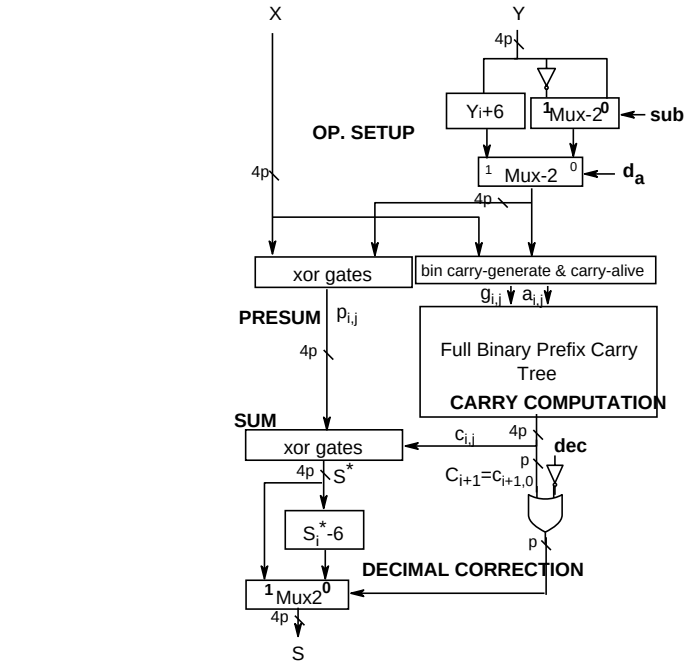


(b) Quaternary prefix carry tree.

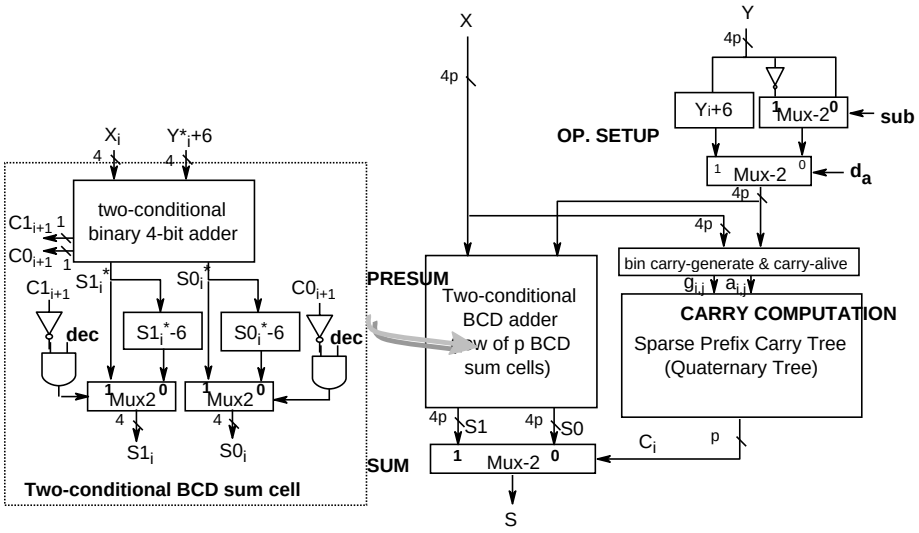
Figure 3.6. *10's complement BCD speculative adders.*

each conditional speculative sum digit $S0_i^*$ and $S1_i^*$ as indicated by the algorithm, that is, $S0_i = S0_i^* - 6$ when $C0_{i+1}$ is zero and $S1_i = S1_i^* - 6$ when $C1_{i+1}$ is zero. The two-conditional 4-bit binary cells are the same as that shown in Fig. 3.4(c).

Fig. 3.7 shows the corresponding two architectures for the mixed binary/BCD implementations of decimal speculative addition. Binary and speculative decimal addition use the same carry network, sharing the same carry path. For BCD subtraction, the simplification $\overline{Y_i + 6} + 6 = \overline{Y_i}$ (with $\overline{Y_i} \in [6, 15]$) is taken into account. Control signal dec is enabled for decimal operations while d_a is only enabled for decimal addition.



(a) Full binary prefix carry tree.



(b) Quaternary prefix carry tree.

Figure 3.7. *Mixed binary/BCD speculative adders.*

3.2 Proposed method: conditional speculative decimal addition

The direct decimal addition method provides high-performance implementations of 10's complement BCD adders at the expense of limiting the carry evaluation topology. This reduces the choices to optimize the area/power-delay tradeoffs of resultant designs with respect to binary adders. On the other hand, the speculative decimal addition method can be implemented using any binary carry evaluation topology. However, its performance is highly dependent on the adder topology due to the decimal post-correction of the binary sum, requiring the use of

[Algorithm: Conditional speculative 10's complement BCD Add/Sub]

Inputs: $X := \sum_{i=0}^{p-1} X_i \cdot 10^i := \sum_{j=0}^{p-1} (X_i^U \ 2 + x_{i,0}) \cdot 10^i$

$Y := \sum_{i=0}^{p-1} Y_i \cdot 10^i := \sum_{j=0}^{p-1} (Y_i^U \ 2 + y_{i,0}) \cdot 10^i$

$$C_0 := \begin{cases} 1 & \texttt{If(op == sub)} \\ 0 & \texttt{Else} \end{cases}$$

For (i:=0;i<p;i++){

$$Y_i^* := (Y_i^*)^U \ 2 + y_{i,0} = \begin{cases} \overline{Y_i + 6} & \texttt{If(op == sub)} \\ Y_i & \texttt{Else} \end{cases}$$

1. Conditional speculation:

$$A_i^U = \begin{cases} 1 & \texttt{If } X_i^U + (Y_i^*)^U \geq 8 \\ 0 & \texttt{Else} \end{cases}$$

$$Z_i^U = X_i^U + (Y_i^*)^U + 6 \cdot A_i^U$$

2. Binary carry propagation:

$$c_{i+1,0} = \lfloor (Z_i^U + x_{i,0} + y_{i,0}^* + c_{i,0})/16 \rfloor$$

$$C_{i+1} = \lfloor (X_i + Y_i^* + C_i)/10 \rfloor = c_{i+1,0}$$

3. Decimal digit addition:

$$s_{i,0} = \mathrm{mod}_2(x_{i,0} + y_{i,0}^* + c_{i,0})$$

$$S_i^U = \begin{cases} 8 & \texttt{If}\Big(Z_i^U == 14 \ \texttt{AND} \ c_{i,1} == 0\Big) \\ \mathrm{mod}_{16}(Z_i^U + 2 \cdot c_{i,1}) & \texttt{Else} \end{cases}$$

$$S_i = \mathrm{mod}_{10}(X_i + Y_i^* + C_i) = S_i^U + s_{i,0}$$

}

Figure 3.8. *Proposed Conditional Speculative Algorithm.*

hybrid prefix tree/carry-select configurations for low-latency implementations.

As in the speculative method, we propose to implement 10's complement BCD addition using any 2's complement binary adder, but removing the decimal correction placed after the binary carry evaluation. This allows for an efficient implementation of a mixed binary/BCD adder using any existing binary prefix tree adder and a few additional hardware [144]. Moreover, our scheme presents a lower dependency on the carry tree topology than previous representative methods for BCD addition, [118, 117], giving the designer more flexibility to choose the adder architecture and area/latency trade-offs.

Our proposal is to use a simple condition for speculation (digit addition of +6) that reduces the cases for which the speculative sum digits S_i^* (obtained from a binary addition) do not correspond with the BCD sum digits S_i. This results in a simpler decimal correction scheme that can be placed out of the critical path (carry propagation path) independently of the binary adder topology. The proposed algorithm is shown in Fig. 3.8. We structure it into three different stages: conditional speculation, binary carry propagation and decimal digit addition.

The control signal A_i^U is used as a condition to speculate when a decimal carry-out C_{i+1} is produced, that is, we state that C_{i+1} is equal to A_i^U. Then, when $A_i^U == 1$ is verified, we speculate that a decimal carry-out is produced and the digit position i is incremented in 6 units to compute the decimal sum digit as a binary sum, as described in Section 3.1.3 for the speculative algorithm. An uncorrect sum digit is produced when the digit position i has been

incremented by +6 ($A_i^U == 1$) and a decimal carry-out has not been produced ($C_{i+1} == 0$).

The condition for speculation A_i^U is obtained as follows. The decimal alive function A_i defined as

$$A_i = \begin{cases} 1 & \texttt{If } X_i + Y_i^* \geq 9 \\ 0 & \texttt{Else} \end{cases} \tag{3.27}$$

indicates a necessary (but not sufficient) condition for the generation or propagation of a decimal carry from position i to $i+1$. This means that a carry-out C_{i+1} is produced only if A_i is one, and in that case the digit position is incremented by 6. Therefore, we could use A_i for speculation and then perform a binary addition to obtain the decimal carries.

However, the computation of A_i, performed in parallel with the $+6$ digitwise increment, may contribute to the critical path delay. To minimize this delay, we derive a simpler condition for speculation from expression (3.27).

For a decimal digit X_i, we call X_i^U (upper part of X_i) the 3 left-most significant bits of the BCD digit. Moreover, we have that

$$\texttt{If } X_i + Y_i^* \geq 9 \Rightarrow X_i^U + (Y_i^*)^U \geq 8 \tag{3.28}$$

This simplified condition results in two implementations with different area-delay trade offs:

- We can use A_i^U defined by[10]

$$A_i^U = \begin{cases} 1 & \texttt{If } X_i^U + (Y_i^*)^U \geq 8 \\ 0 & \texttt{Else} \end{cases} \tag{3.29}$$

  as the condition for speculation. We add $+6$ to the digit position i if A_i^U is true, that is,

$$Z_i^U = X_i^U + (Y_i^*)^U + 6 \cdot A_i^U, \quad Z_i^U \in \{0, 2, 4, 6, 14, 16, 18, 20, 24\} \tag{3.30}$$

  The conditional +6 digit additions are performed biasing digits X_i or Y_i^* by 6 when A_i^U is one.

- To obtain a faster condition for speculation, we can check it separately for addition and subtraction. For decimal addition, since $Y^* = Y$, the condition to check is $X_i^U + Y_i^U \geq 8$. We define a control signal A_i^{U+} as

$$A_i^{U+} = \begin{cases} 1 & \texttt{If } X_i^U + Y_i^U \geq 8 \\ 0 & \texttt{Else} \end{cases} \tag{3.31}$$

  Similarly, the resultant condition for subtraction is $X_i^U + (\overline{Y_i + 6})^U \geq 8$. Since $\overline{Y_i + 6} = 15 - (Y_i + 6)$, the condition is expressed as $X_i^U + (15 - Y_i)^U - 6 \geq 8$, resulting in $X_i^U + (\overline{Y_i}^U) \geq 14$. So we define the conditional speculation control signal for subtraction A_i^{U-} as

$$A_i^{U-} = \begin{cases} 1 & \texttt{If } X_i^U + \overline{Y_i^U} \geq 14 \\ 0 & \texttt{Else} \end{cases} \tag{3.32}$$

[10] This definition corresponds with the upper direct decimal carry-alive A^U given by equation (3.13).

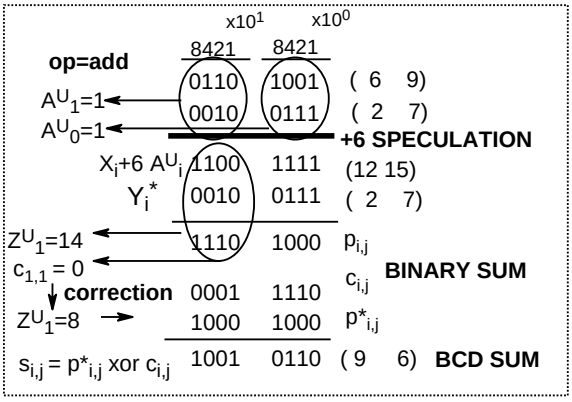


Figure 3.9. *Example: Conditional Speculative BCD Addition.*

Therefore, the value of Z_i^U (represented as two 3-bit vectors) is determined in terms of A_i^{U+} and A_i^{U-} as follows:

$$Z_i^U = \begin{cases} X_i^U + \overline{Y_i^U + 6 \cdot \overline{A_i^{U-}}} & \texttt{If}(op == sub) \\ X_i^U + 6 \cdot A_i^{U+} + Y_i^U & \texttt{Else} \end{cases} \tag{3.33}$$

We compute the 10's complement BCD addition $S = X \pm Y$ as a modified binary sum. An example of a two-digit addition is shown in Fig. 3.9. The value of the BCD sum digit S_i correspond with the binary sum $\text{mod}_{16}(Z_i^U + x_{i,0} + y_{i,0}^* + C_i)$ when C_{i+1} is equal to A_i^U. In other case, the +6 digit bias added in excess must be removed.

According to the direct decimal carry-propagate recurrence ($C_{i+1} = G_i^U \vee A_i^U \ c_{i,1}$), the carry-out C_{i+1} differs from A_i^U only when $G_i^U == 0$, $c_{i,1} == 0$ and $A_i^U == 1$, since in this case $C_{i+1} = G_i^U = 0$. From the definition of G_i^U and A_i^U (equations (3.13) and (3.27)), we get that this occurs for $X_i^U + (Y_i^*)^U == 8$ and therefore

$$Z_i^U = X_i^U + (Y_i^*)^U + 6 \cdot A_i^U = 8 + 6 = 14 \tag{3.34}$$

To obtain the decimal carries C_i we compute the binary carry-propagate recurrence

$$c_{i+1,0} = \lfloor (Z_i^U + x_{i,0} + y_{i,0}^* + c_{i,0})/16 \rfloor \tag{3.35}$$

It can be evaluated indistinctly in a full binary prefix tree (obtaining all the binary $c_{i,j}$) or in a sparse prefix tree, computing only $c_{i,0}$ (quaternary carry tree). Decimal carries correspond with binary carries at hexadecimal positions, that is $C_i = c_{i,0}$.

Therefore, $S = X \pm Y = \sum_{i=0}^{p-1} S_i \ 10^i$ is computed as the binary sum $S^* = \sum_{i=0}^{p-1} (S_i^U + s_{i,0}) \ 16^i$, with

$$\begin{aligned} S_i^U &= \begin{cases} 8 (= 14 - 6) & \texttt{If}\big(Z_i^U == 14 \ \texttt{AND} \ c_{i,1} == 0\big) \\ \text{mod}_{16}(Z_i^U + c_{i,1} \ 2) & \texttt{Else} \end{cases} \\ s_{i,0} &= \text{mod}_2(x_{i,0} + y_{i,0}^* + c_{i,0}) \end{aligned} \tag{3.36}$$

and $c_{i,1} = \lfloor (x_{i,0} + y_{i,0}^* + c_{i,0})/2 \rfloor$.

In this way, the bit vector representations of S and S^* are similar. Moreover, S^* can be computed in any conventional binary adder with a slight correction that does not affect to its critical path. This decimal correction consists basically on the detection and replacement of $Z_i^U == 14$ (binary '111-') by 8 (binary '100-') when $c_{i,1} == 0$ as shown in Fig. 3.9.

The condition $Z_i^U == 14$ can be detected examining the binary carry-propagate functions $p_{i,j}$, obtained by xoring the bits of Z_i^U, since $Z_i^U == 14 \Leftrightarrow p_{i,3}\ p_{i,2}\ p_{i,1} == 1$. To replace the value ('111-') by ('100-') when $c_{i,1}$ is zero, we define the modified binary carry-propagate functions $p^*_{i,j}$ as

$$p^*_{i,j} = \begin{cases} 0 & \texttt{If}\big(p_{i,3}\ p_{i,2}\ p_{i,1}\ \overline{c_{i,1}} == 1\ \texttt{AND}\ j \in \{1,2\}\big) \\ p_{i,j} & \texttt{Else} \end{cases} \tag{3.37}$$

and use them to compute the BCD sum bits as $s_{i,j} = p^*_{i,j} \oplus c_{i,j}$. Depending on the carry tree topology (binary or quaternary), the decimal sum digits S_i are obtained from $p_{i,j}$ and $c_{i,j}$ (or C_i) as follows:

- For a **full binary prefix carry tree**, sum bits are computed directly as $s_{i,j} = p^*_{i,j} \oplus c_{i,j}$. Expressing the $p^*_{i,j}$'s as a function of the $p_{i,j}$'s (equation (3.37)) and replacing them in the previous expression, we have that the BCD sum digits S_i are given by:

$$S_i = \begin{cases} s_{i,3} = & p_{i,3} \oplus c_{i,3} \\ s_{i,2} = & \overline{p_{i,3}\ p_{i,2}\ p_{i,1}}(p_{i,2} \oplus c_{i,2}) = \overline{p_{i,2}}\ c_{i,2} \vee \overline{p_{i,3}}\ p_{i,2}\ \overline{c_{i,2}} \\ s_{i,1} = & \overline{p_{i,3}\ p_{i,2}\ p_{i,1}}(p_{i,1} \oplus c_{i,1}) = \overline{p_{i,1}}\ c_{i,1} \vee \overline{p_{i,3}\ p_{i,2}}\ p_{i,1}\ \overline{c_{i,1}} \\ s_{i,0} = & p_{i,0} \oplus c_{i,0} \end{cases} \tag{3.38}$$

  Note that $p_{i,3}\ p_{i,2}\ p_{i,1}\ \overline{c_{i,1}} = 1$ implies $c_{i,2} = g_{i,2} \vee p_{i,1}\ c_{i,1} = 0$.

- For a **quaternary prefix carry tree** configuration, two modified conditional 4-bit binary sums, $S1_i$ with input carry one ($C1_i = 1$) and $S0_i$ input carry zero ($C0_i = 1$), are computed for each digit in parallel to the carry computation. After introducing expression (3.37) in the 4-bit binary carry-propagate sum equations for each condition, the following expression for $S1_i$ and $S0_i$ are obtained:

$$S1_i = \begin{cases} s1_{i,3} = & p_{i,3} \oplus c1_{i,3} \\ s1_{i,2} = & \overline{p_{i,2}}\ c1_{i,2} \vee \overline{p_{i,3}}\ p_{i,2}\ \overline{c1_{i,2}} \\ s1_{i,1} = & \overline{p_{i,1}}\ c1_{i,1} \vee \overline{p_{i,3}\ p_{i,2}}\ p_{i,1}\ \overline{c1_{i,1}} \\ s1_{i,0} = & \overline{p_{i,0}} \end{cases}$$

$$S0_i = \begin{cases} s0_{i,3} = & p_{i,3} \oplus c0_{i,3} \\ s0_{i,2} = & \overline{p_{i,2}}\ c0_{i,2} \vee \overline{p_{i,3}}\ p_{i,2}\ \overline{c0_{i,2}} \\ s0_{i,1} = & \overline{p_{i,1}}\ c0_{i,1} \vee \overline{p_{i,3}\ p_{i,2}}\ p_{i,1}\ \overline{c0_{i,1}} \\ s0_{i,0} = & p_{i,0} \end{cases} \tag{3.39}$$

  where the binary carries for each condition are computed in a carry-ripple form as:

$$\begin{aligned} c0_{i,1} &= g_{i,0} \\ c0_{i,2} &= g_{i,1} \vee a_{i,1}\ c0_{i,0} \\ c0_{i,3} &= g_{i,2} \vee a_{i,2}\ c0_{i,1} \\ c1_{i,1} &= a_{i,0} \\ c1_{i,2} &= g_{i,1} \vee a_{i,1}\ c1_{i,0} \\ c1_{i,3} &= g_{i,2} \vee a_{i,2}\ c1_{i,1} \end{aligned} \tag{3.40}$$

Each sum digit is selected from the conditional sum digits depending on the value of the correspondent decimal carry input C_i as $S_i = S1_i\ C_i \vee S0_i\ \overline{C_i}$.

Therefore, the decimal correction of the binary sum does not contribute to the critical path delay of the adder, independently of its carry tree topology. In addition, any 2's complement adder can be used to compute the conditional speculative BCD sum introducing only minor modification in the binary sum cells. In the next Section we present several 10's complement BCD and mixed binary/BCD adders implementing the proposed method.

3.3 Proposed architectures

There exists many topologies to implement carry-propagate addition, namely carry ripple, carry skip, carry-select [9], conditional sum [128], CLA (carry look-ahead) [162], prefix tree (or parallel prefix) [84, 86, 89] and Ling adders [95] among others.

Current high-speed (logarithmic-time) adders use variations of prefix tree schemes because they leads to efficient implementations in VLSI. They have simple cells and regular structures providing high flexibility to implement adders in a wide range of design trade-offs. Also, the prefix formulation describes, in a very flexible and simple way, different carry-propagate addition schemes, including CLA addition [84] and Ling addition [147].

Thus, we have expressed the binary carry-propagate recurrence in terms of the following three prefix tree schemes, which cover a wide range of state of the art designs:

- Full binary prefix tree adders (Kogge-Stone [86], Han-Carlson, Ladner-Fisher [89], Brent-Kung, Knowles adders [84],...).
- Hybrid adders [100], combining a sparse prefix carry tree and a carry-select output stage.
- Ling adders reformulated in terms of a prefix tree computation [147]. This includes both full binary Ling and hybrid Ling prefix carry tree/carry-select schemes.

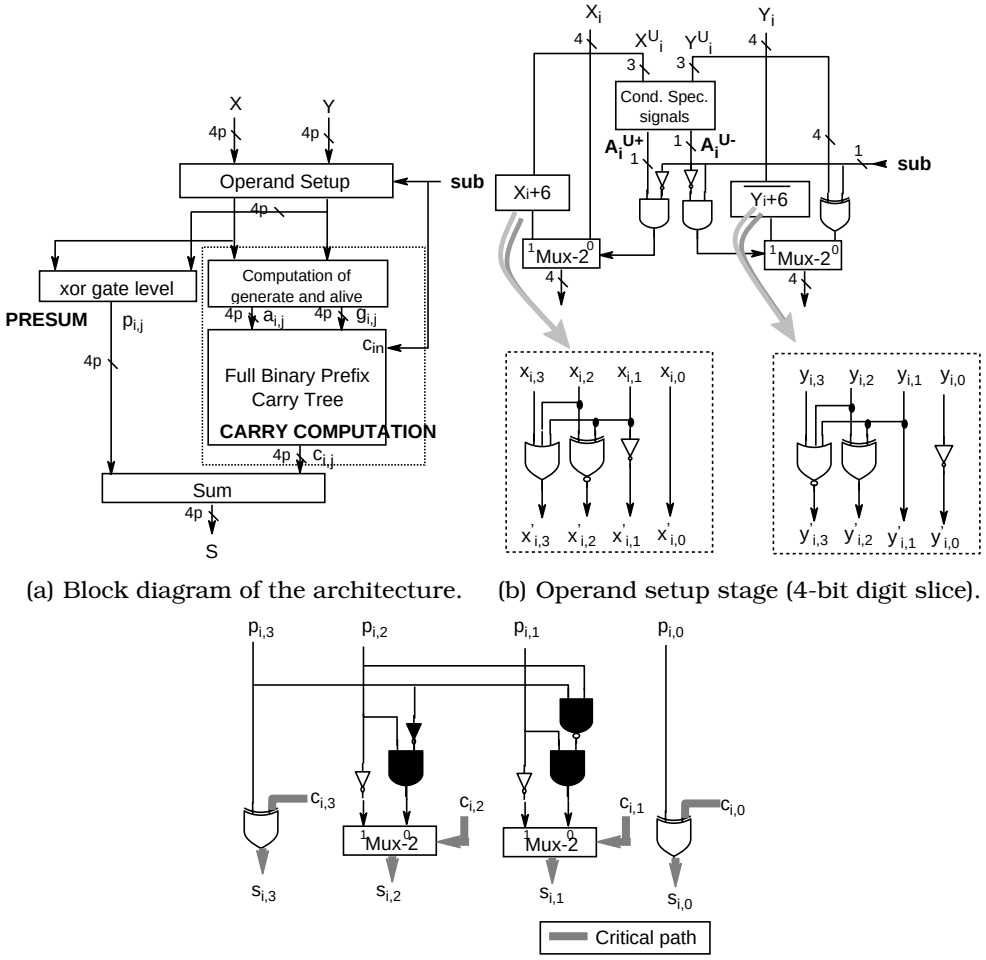
Then, we implemented the conditional speculative decimal addition algorithm using these adder topologies, resulting in three different architectures for both 10's complement and mixed 2's complement/10's complement addition. We detail next the resulting architectures.

3.3.1 Binary prefix tree architectures

Fig. 3.10(a) shows the proposed full binary prefix carry tree architecture for 10's complement BCD addition. It consists of operand setup, pre-sum, carry computation and sum stages. Fig. 3.10(b) details a digit (4-bit) slice of the operand setup stage [11]. It performs a conditional speculation simultaneously with the 10's complement of operand Y required for subtraction (binary control signal sub is defined true for subtraction).

For a low latency implementation we opt for computing the conditions for speculation for addition (A_i^{U+}) and subtraction (A_i^{U-}) separately. Thus, conditional speculation is performed

[11] We include in this stage all the operations previous to carry computation.



(a) Block diagram of the architecture. (b) Operand setup stage (4-bit digit slice).

(c) Digit (4-bit) sum cell (see (3.38)).

Figure 3.10. *10's complement BCD adder using a binary prefix carry tree.*

selecting the digit $X_i + 6$ when the control signal $\overline{sub}$ A_i^{U+} is true or the digit X_i if the corresponding control signal is false. By other hand, $\overline{Y_i + 6}$ is selected when control signal sub $\overline{A_i^{U-}}$ is true. In other case, the selected digit is $\overline{Y_i}$ $sub \vee Y_i$ $\overline{sub}$. Signals A_i^{U+}, A_i^{U-} are implemented in terms of the bits of X_i^U and Y_i^U as:

$$\begin{aligned} A_i^{U+} &= x_{i,3} \vee y_{i,3} \vee (x_{i,2}\ y_{i,2}) \vee (x_{i,2} \vee y_{i,2})\ x_{i,1}\ y_{i,1}) \\ A_i^{U-} &= x_{i,3} \vee \overline{y_{i,3}}(x_{i,2}\ \overline{y_{i,2}} \vee (x_{i,2} \vee \overline{y_{i,2}})(x_{i,1} \vee \overline{y_{i,1}})) \end{aligned} \tag{3.41}$$

while digitwise operations X_i+6 and $\overline{Y_i + 6}$ are implemented as in (3.23) and (3.18) respectively.

The implementation of a 4-bit sum cell is shown in Fig. 3.10(c). It is basically a conventional 4-bit binary sum cell, consisting of 4 XOR/MUX-2 gates, and three additional simple gates, shown in black. BCD sum bits $s_{i,j}$ are obtained from the binary carry-propagate signals $p_{i,j}$ and the binary carries $c_{i,j}$ as it is indicated by expression (3.38). The binary carry-propagates $p_{i,j}$ are computed in the pre-sum stage. Note that the black gates, used for decimal correction of uncorrect values ('111-'), are not in the critical path (the carry path, highlighted in gray in Fig. 3.10(c)), because the carry-in dependency is at the very last stage (XOR or

MUX-2 level) as in standard binary addition.

The computation of the binary carries $c_{i,j}$ is performed in a full binary prefix tree. For a clear description of algorithms when dealing with binary variables, we represent the double index (i, j) with a single index k as $(i, j) \rightarrow k = 4 \cdot i + j$. Thus, the conventional binary carry recurrence is represented as $c_{k+1} = g_k \vee a_k \; c_k$, where $g_i = x_i \; y_i^*$ is the binary carry-generate signal and $a_i = x_i + y_i^*$ is the binary carry-alive signal. Note that we assume $g_{-1} = c_0 = c_{in}$ and $a_{-1} = 0$. This recurrence is described as a prefix computation as follows,

$$(c_{k+1}, a_{k:-1}) = (g_{k:0}, a_{k:0}) \bullet (c_{in}, 0) = \prod_{q=-1}^{k} (g_q, a_q) \tag{3.42}$$

where $a_{k:-1} = a_{k:0} \; a_{-1} = 0$, and the product $\prod$ uses the prefix operator $\bullet$ defined in (3.22).

This operator is associative and idempotent, so that highly parallel tree-like structures can be used for the computation of the carries. For two arbitrary positions r and l the block carry-generate signal ($g_{r:l}$) and block carry-alive signal ($a_{r:l}$) are

$$(g_{r:l}, a_{r:l}) = \prod_{k=r}^{l} (g_k, a_k) \tag{3.43}$$

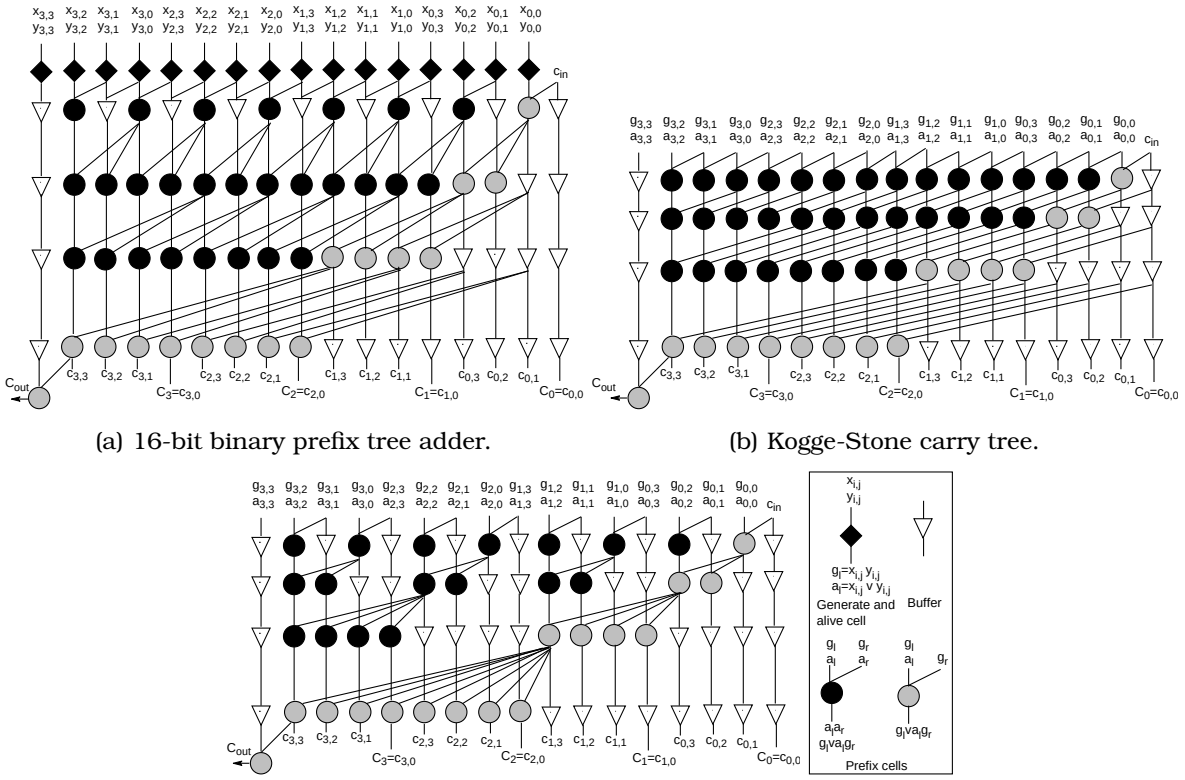
The block carry-generate and block carry-alive signals allow the computation of the output carry from a block in terms of its input carry, that is, $c_{r+1} = g_{r:l} \vee a_{r:l} \; c_l$.

These terms can be grouped in many different ways, leading to prefix tree adders with different area-delay tradeoffs. The minimum logic depth n-bit prefix adder requires $\log_2 n$ stages of prefix cells to compute the carries needed for the sum bits (or $\log_2(n + 1)$ stages to compute the carry output C_{out}).

For instance, Fig. 3.11(a) shows a graph representation of a 16-bit binary prefix tree adder with minimum logic depth. It includes the computation of the binary carry-generate and carry-alive signals. This configuration ([2221] Knowles adder [84]) presents a trade-off between area and delay, by means of limiting the fanout and exploiting some idempotency of the prefix nodes (black dots). The last level of the prefix tree computes the block generate and alive signals $(g_{k:-1}, a_{k:-1}) = (c_{i+1,0}, 0)$. These prefix nodes (grey dots) are simpler, since the logic for the block alive signals $a_{k:-1}$ is not necessary ($a_{k:-1} = 0$).

Two minimum logic prefix trees in opposite points of the area-delay space design are the Kogge-Stone [86] (Fig. 3.11(b)) and the Ladner-Fisher [89] (Fig. 3.11(c)) schemes. The Kogge-Stone exploits idempotency in order to present the minimum possible fanout (output load capacitance) at each prefix node, but at the expense of the cost in area (maximum number of logic cells and high wiring). The Ladner-Fisher [89] scheme reduces the hardware complexity by means of some high fanout nodes, incrementing the delay.

The mixed 2's complement/10's complement architecture is shown in Fig. 3.12. Binary addition and subtraction can be integrated in the 10's complement BCD adder datapath introducing minor changes. Thus, the mixed architecture presents the same latency and practically the same area as the 10's complement BCD adder of Fig. 3.10. For a better integration of binary operations in the 10's complement BCD adder, we make use of an extended set of control signals. Decimal mode is indicated by signal $dec = 1$, while decimal addition is selected with $d_a = 1$. The binary mode correspond to values of $dec = d_a = d_s = 0$ while $sub = 1$ indicates a binary or a decimal subtraction.



(a) 16-bit binary prefix tree adder.

(b) Kogge-Stone carry tree.

(c) Ladner-Fisher carry tree.

(d) Notation.

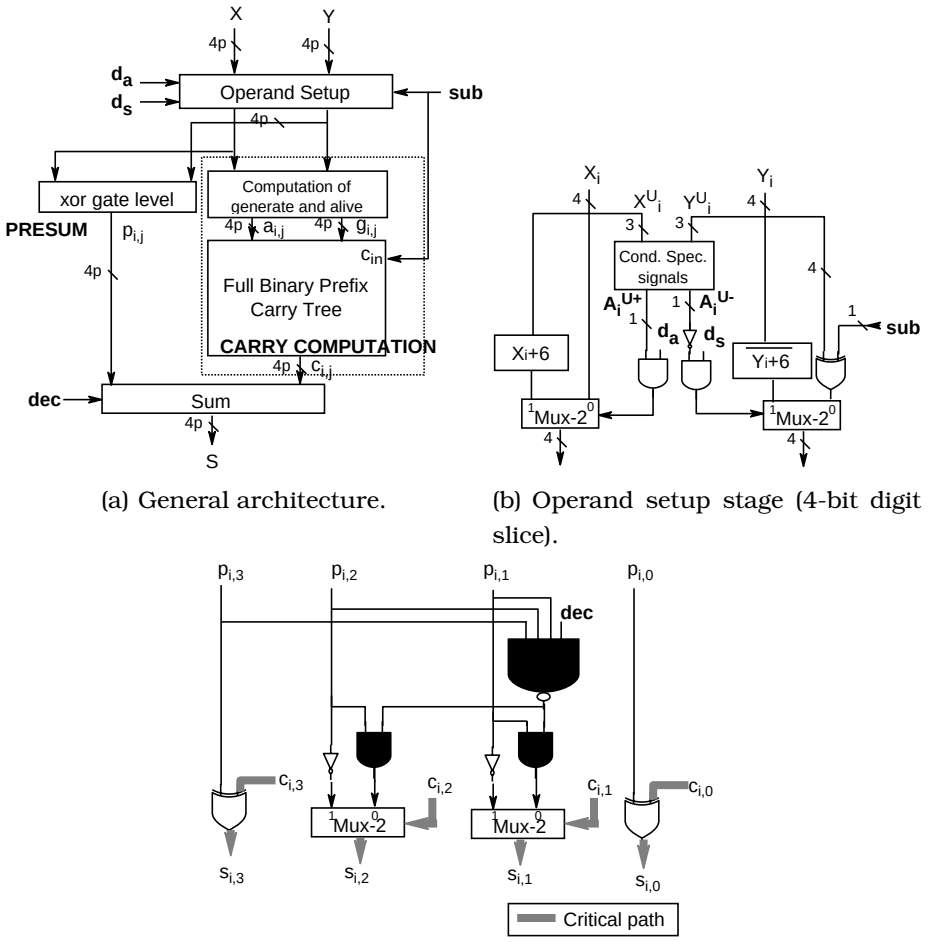
Figure 3.11. *Graph representations of binary prefix tree adders.*

The implementation of the operand setup stage is detailed in Fig. 3.12(b). The only difference with respect to the same stage in Fig. 3.10(b) is the inclusion of control signals d_a and d_s to select operands X and Y ($sub = 0$) or $\overline{Y}$ ($sub = 1$) in case of binary operations ($d_a = d_s = 0$).

Fig. 3.12(c) shows the mixed 4-bit sum cell. The black gates replace the bit string ('111-') of the binary carry propagates $p_{i,j}$ by string ('100-') when $c_{i,1} = 0$ and only for the decimal mode ($dec = 1$). If dec is disabled, then the 4-bit sum cell is equivalent to a conventional 4-bit binary sum cell composed of 4 xor gates.

3.3.2 Hybrid prefix tree/carry-select architectures

Hybrid sparse prefix tree/carry-select topologies minimize the power dissipation of full binary prefix tree adders by reducing the wiring complexity, the logic density and the transistor sizes of the carry tree [100]. Moreover, they are also appropriate for low-latency implementations. On the contrary, they may require a little more area to compute two-conditional k-bit presums in parallel with carry computation. However, since the evaluation of presum digits is not usually in the critical path, design constraints can be relaxed, which leads to designs with interesting area and delay tradeoffs. For decimal BCD addition, a suitable bit length for each two-conditional presum is $k = 4$.



(a) General architecture.

(b) Operand setup stage (4-bit digit slice).

(c) Digit (4-bit) sum cell.

Figure 3.12. *Mixed binary/decimal adder using a binary parallel prefix carry tree.*

The conditional speculative decimal addition method can be easily implemented in a quaternary prefix tree adder. The block diagram of the proposed hybrid prefix tree/carry-select BCD adder is shown in Fig. 3.13. It consists of the following stages: operand setup, carry computation, conditional presum and sum. The conditional presum stage computes the BCD digit presums $S1_i$ and $S0_i$ given by expression (3.39). Fig. 3.14 shows the implementation of a two-conditional BCD (4-bit) sum cell. It consists of a 4-bit two-conditional binary adder and 3 additional simple gates (shown in black). The two-conditional binary adder computes simultaneously the 4-bit pre-sums $S1_i^*$ (assuming $C_i = 1$), and $S0_i^*$ (assuming $C_i = 0$) and two 4-bit binary carry recurrences with $c1_{i,0} = 1$ (so $c1_{i,1} = a_{i,0}$) and $c0_{i,0} = 0$ (so $c0_{i,1} = g_{i,0}$). The black gates replace the 4-bit binary values ('111-') by the correct BCD digits ('100-'). For the other cases, $S1_i = S1_i^*$ and $S0_i = S0_i^*$. The appropriate sum digits $S1_i$ or $S0_i$ are then selected by the corresponding decimal carries C_i in the sum stage using a level of MUX-2 gates.

For the operand setup stage, we propose the three implementations of Fig. 3.15, which present different area and delay tradeoffs:

- Fig. 3.15(a) shows a digit slice of the operand setup stage when the decimal carries C_i

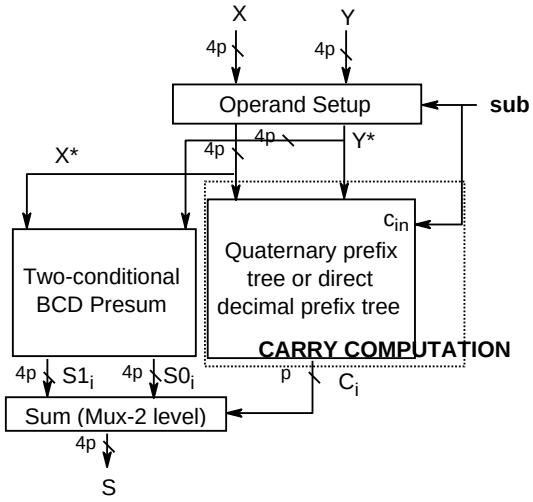


Figure 3.13. *Block diagram of the 10's complement BCD hybrid adder.*

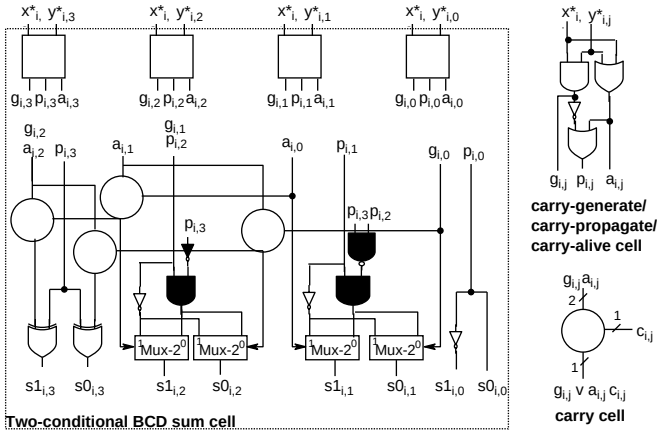


Figure 3.14. *Proposed two-conditional BCD (4-bit) sum cell.*

are computed using the conditional speculative carry recurrence as

$$C_{i+1} = \lfloor (X_i + Y_i^* + 6 \cdot (A_i^{U+} \ \overline{sub} \vee A_i^{U-} \ sub) + C_i)/16 \rfloor \tag{3.44}$$

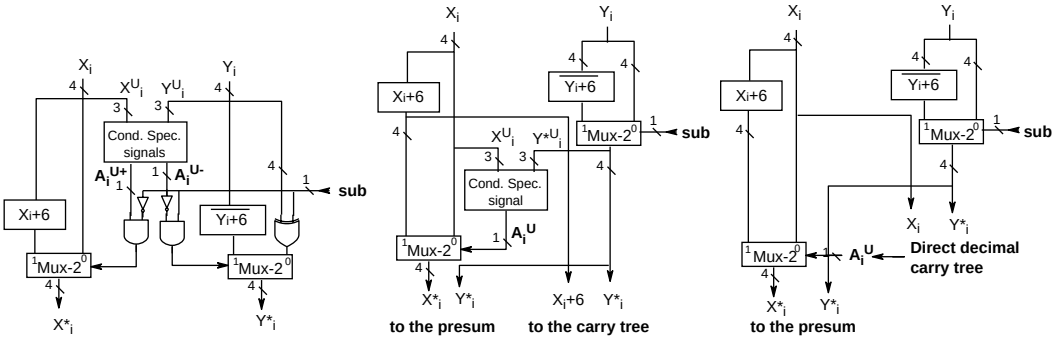
This configuration is similar to the one described in Fig. 3.10 for the full binary prefix tree adder. It requires to evaluate A_i^{U+} and A_i^{U-} before the carry computation. For carry computation, an appropriate topology is the quaternary prefix tree (QT), that is, an sparse prefix tree which generates 1 in 4 carries.

For the quaternary configuration, the decimal carries C_i are obtained as

$$(C_i, 0) = (g_{i:-1}, a_{i:-1}) = \prod_{k=-1}^{i} (g_{k+3:k}, a_{k+3:k}) \tag{3.45}$$

where

$$(g_{k+3:k}, a_{k+3:k}) = (g_{k,3}, a_{k,3}) \bullet (g_{k,2}, a_{k,2}) \bullet (g_{k,1}, a_{k,1}) \bullet (g_{k,0}, a_{k,0}) \tag{3.46}$$



(a) Basic configuration. (b) Low-latency config. (c) Area-optimized config.

Figure 3.15. *Proposed implementations for the operand setup stage (1-digit slices).*

with $g_{-1} = C_{in}$ and $a_{-1} = 0$. Therefore, to obtain all the decimal carries, only the block carry generate and block carry alive $(g_{k+3:k}, a_{k+3:k})$ are necessary.

The block diagram of a 16-bit (two BCD-digit) quaternary prefix tree is shown in Fig. 3.16. The reduction in the number of nodes and wires is significant with respect to

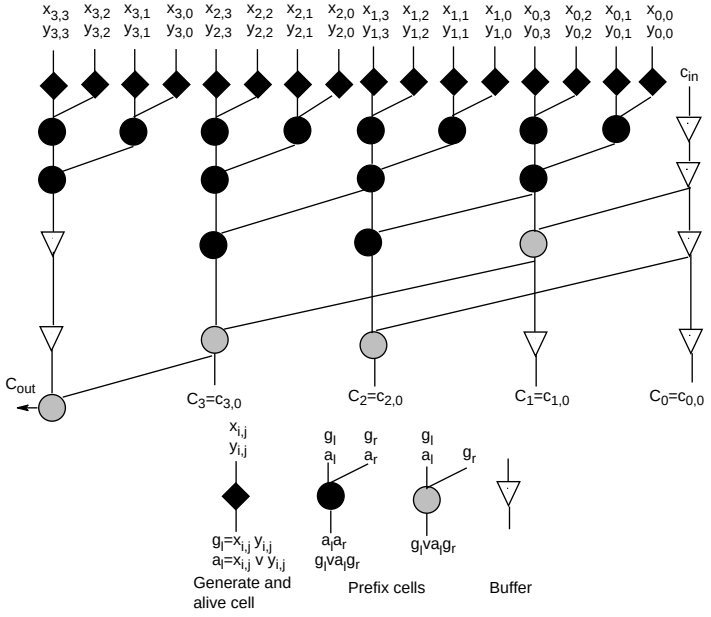


Figure 3.16. *Quaternary prefix carry tree.*

the 16-bit full binary prefix trees of Fig. 3.11. There are multiple choices for the prefix tree topology. The prefix carry tree of Fig. 3.16 obtains the block generate and block alive signals $(g_{k+3:k}, a_{k+3:k})$ for each digit using the first two levels of prefix nodes. Then, it computes the decimal carries using a Kogge-Stone scheme of $\log_2 p$ levels for $p = 4n$ decimal digits.

- Fig. 3.15(b) shows the implementation (one digit) of the operand setup stage corresponding to the carry recurrence $C_{i+1} = \lfloor (X_i + Y_i^* + 6 + C_i)/16 \rfloor$. This carry recurrence is evaluated

as before in a quaternary prefix tree, where digits $X_i + 6$ and $Y_i^* = Y_i \ \overline{sub} \vee \overline{Y_i + 6} \ sub$ are computed as inputs for the carry recurrence.

Thus, the condition for speculation is only required to obtain the inputs to the presum stage as $X_i^* = X_i + 6 \cdot A_i^U$ and $Y_i^* = Y_i \ \overline{sub} \vee \overline{Y_i + 6} \ sub$. The main difference with respect to the scheme of Fig. 3.15(a) is that the evaluation of the condition for speculation is now out of the critical path. This allows to reduce the hardware complexity of the operand setup by implementing signal A_i^U (instead of two separate conditions A_i^{U+} and A_i^{U-}) in terms of the bits of X_i^U and $(Y_i^*)^U$ as given by expression (3.13).

- Fig. 3.15(c) shows the configuration of the operand setup stage required to compute the decimal carries C_i using the direct decimal carry recurrence. The inputs to the prefix carry tree are X_i and $Y_i^* = Y_i \ \overline{sub} \vee \overline{Y_i + 6} \ sub$. Instead of a quaternary prefix tree, a direct decimal prefix tree evaluates

$$C_{i+1} = G_i \vee A_i \ C_i = G_i^U \vee A_i^U \ (g_{i,0} \vee a_{i,0} \ C_i) \tag{3.47}$$

where the G_i^U and A_i^U are the upper decimal carry generate and carry alive signals given by expression (3.13). An example of an 8-bit (two digit) direct decimal prefix tree is shown in Fig. 3.17. It performs the following operation:

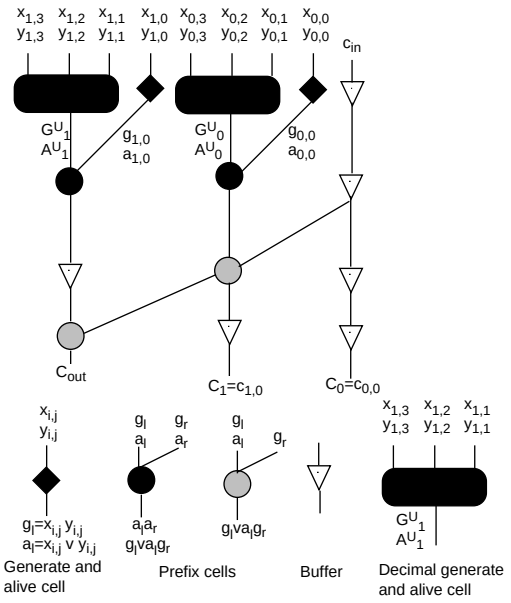
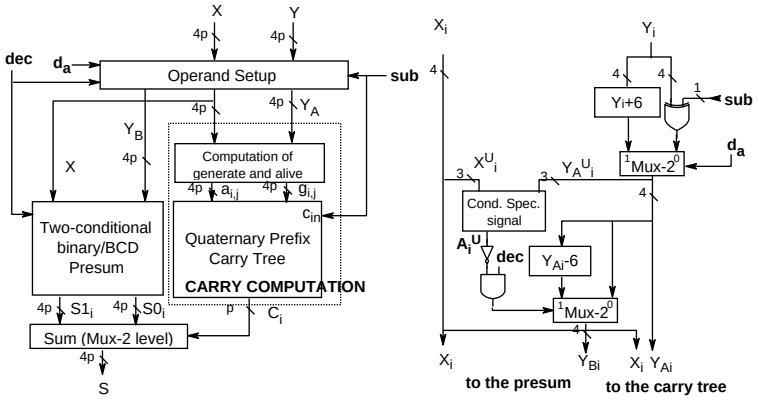


Figure 3.17. *Direct decimal prefix carry tree.*

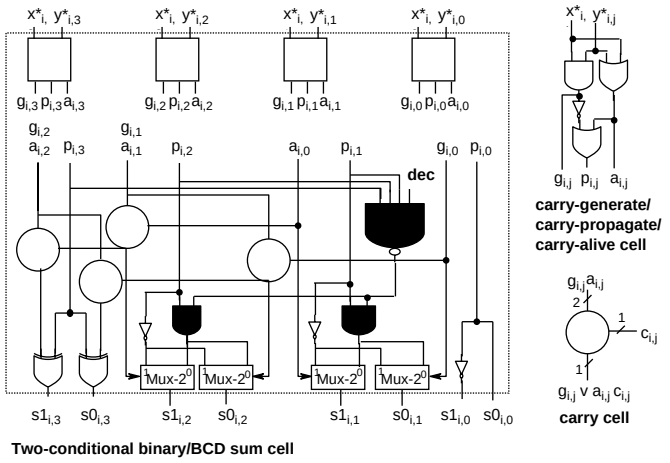
$$(C_i, 0) = \prod_{k=-1}^{i} (G_k, A_k) = \prod_{q=-1}^{i} (G_k^U, A_k^U) \bullet (g_{k,0}, a_{k,0}) \tag{3.48}$$

so that it requires a prefix tree level less than a quaternary tree to compute the same number of carries. But, on the other hand, G_i^U and A_i^U are more costly to compute than their binary counterparts.

An advantage of this scheme is that it reduces, even more, the hardware complexity of the operand setup stage scheme of Fig. 3.15(b). The signals A_i^U, computed in the first



(a) General block diagram. (b) Operand setup stage (one digit).



(c) Two-conditional binary/BCD (4-bit) sum cell.

Figure 3.18. *Mixed binary/BCD quaternary prefix tree/carry-select adder.*

level of the direct decimal prefix tree, are reused to obtain the input digits $X_i^* = X_i + 6 \cdot A_i^U$ required for the presum stage.

In Fig. 3.18 we show the implementation of the mixed binary/decimal architecture. The key issue is to incorporate the binary add/sub operations into a BCD hybrid adder without introducing a latency overhead. For a good area-delay tradeoff, the preferred scheme combines the operand setup stage of Fig. 3.18(b), a quaternary prefix tree and a presum stage implementing the two-conditional binary/BCD 4-bit sum cell of Fig. 3.18(c). Input operand X is passed through the setup stage to the carry prefix tree and the presum stage without modifications. Operand Y is modified to provide support for decimal addition ($dec == 1$, $d_a == 1$), decimal subtraction ($dec == 1$, $sub == 1$), binary addition and binary subtraction ($sub == 1$). Thus, in the operand setup stage we compute

$$\begin{aligned} Y_i^A &= (Y_i + 6)\ d_a \vee (Y_i \oplus sub)\ \overline{d_a} \\ Y_i^B &= Y_i^A\ \overline{\overline{A_i^U}\ dec} \vee (Y_i^A - 6)\ \overline{A_i^U}\ dec \end{aligned} \tag{3.49}$$

where the XOR operator $\oplus$ acts on each bit of Y_i. Digits Y_i^A are passed to the quaternary

carry prefix tree to compute the recurrence $C_{i+1} = \lfloor (X_i + Y_i^A + C_i)/16 \rfloor$. Note that these digits are obtained faster than Y_i^B and do not depend on A_i^U. The condition for speculation A_i^U is obtained from the 3 most significant bits of X_i and Y_i^A as follows:

$$A_i^U = y_{i,3}^A (x_{i,3} \vee x_{i,2}\ x_{i,1}) \vee (x_{i,3} \vee y_{i,3}^A)\ (y_{i,2}^A\ y_{i,1}^A) \tag{3.50}$$

Digits Y_i^B are passed to the two-conditional presum stage. The mixed binary/BCD sum cell of Fig. 3.18(c) performs the computations $S1_i = mod_{16}(X_i + Y_i^B + 1)$ and $S0_i = mod_{16}(X_i + Y_i^B + 0)$. The black gates perform the BCD digit correction only for decimal operations ($dec == 1$). For binary operations ($dec = d_a == 0$) we have that $Y_i^A = Y_i^B = (Y_i \oplus sub)$ and the sum cell is equivalent to a conventional two-conditional 4-bit binary adder.

3.3.3 Ling prefix tree architectures

In 1981 Ling [95] proposed a new carry recurrence to reduce the logical depth for carry computation in carry look-ahead structures. Not up to very recently, different research works [43, 62, 168, 169] have proposed the formulation of the Ling recurrence as a prefix computation to obtain high performance parallel adder implementations.

In [147] we presented a reformulation of Ling addition, particularly suitable to implement in any existing prefix adder topology. We obtain the Ling scheme directly from a standard prefix formulation of the carry computation, and show that any prefix adder can be transformed into a Ling adder with minor modifications and with the corresponding speed improvement. In this Section we extend this formulation of Ling addition to include also decimal addition. In fact, we have applied it to the 10's complement BCD and mixed binary/BCD prefix tree adders presented in Section 3.3.1 and Section 3.3.2.

Ling used an alternative carry h_k instead of c_k. The recurrence for the Ling carries is

$$h_{k+1} = g_k \vee a_{k-1}\ h_k \tag{3.51}$$

The relation between the two kind of carries is

$$c_{k+1} = a_k\ h_{k+1} \tag{3.52}$$

obtained from $c_{k+1} = g_k \vee a_k\ c_k$ and (3.51) where we have used the property $g_k\ a_k = g_k$, which is the basis for the Ling approach.

The prefix operator can be used to obtain the Ling carries h_k, resulting in fast parallel tree-like implementations. The Ling carry recurrence can be formulated as a simple optimization of the prefix computation. Using the following identity

$$(g_k, a_k) = (0 \vee g_k\ a_k, a_k\ 1) = (0, a_k) \bullet (g_k, 1) \tag{3.53}$$

the expression of the carry prefix computation (3.42) is transformed into

$$(c_{k+1}, a_{k:-1}) = \prod_{q=-1}^{k} (0, a_q) \bullet (g_q, 1) = (0, a_k) \bullet \prod_{q=-1}^{k} (g_q, a_{q-1}) \tag{3.54}$$

By other hand, since $c_{k+1} = a_k\ h_{k+1}$ this results in

$$(c_{k+1}, a_{k:-1}) = (a_k h_{k+1}, a_{k:-1}) = (0, a_k) \bullet (h_{k+1}, a_{k-1:-1}) \tag{3.55}$$

Therefore, examining both right sides of (3.54) and (3.55) we conclude that

$$(h_{k+1}, 0) = \prod_{q=-1}^{k} (g_q, a_{q-1}) \tag{3.56}$$

with $a_{-2} = 0$. The evaluation of the prefix operation over two consecutive terms of (3.56) results in

$$(g_q, a_{q-1}) \bullet (g_{q-1}, a_{q-2}) = (g_q \vee g_{q-1}, a_{q-1}a_{q-2}) \tag{3.57}$$

where we have used again $a_{q-1}g_{q-1} = g_{q-1}$. We define a simplified prefix operator $\circ$ as

$$(g_l, a_l) \circ (g_r, a_r) = (g_l \vee g_r, a_l a_r) \tag{3.58}$$

so the evaluation of the Ling carries is simplified as follows:

$$(h_{k+1}, a_{k-1:-1}) = \begin{cases} \texttt{For k odd (even number of terms):} \\ \prod_{q=-1}^{\lfloor k/2 \rfloor} (g_{2q}, a_{2q-1}) \circ (g_{2q-1}, a_{2q-2}) \\ \texttt{For k even (odd number of terms) pairing from the left:} \\ \left(\prod_{q=0}^{\lfloor k/2 \rfloor} (g_{2q}, a_{2q-1}) \circ (g_{2q-1}, a_{2q-2}) \right) \bullet (c_{in}, 0) \\ \texttt{For k even (odd number of terms) pairing from the right:} \\ (g_k, a_{k-1}) \bullet \left(\prod_{q=-1}^{\lfloor k/2 \rfloor - 1} (g_{2q+1}, a_{2q}) \circ (g_{2q}, a_{2q-1}) \right) \end{cases} \tag{3.59}$$

The binary sum bits s_k are obtained as

$$s_k = p_k \oplus c_k = p_k \oplus (h_k \; a_{k-1}) = p_k \; \overline{h_k} \vee (p_k \oplus a_{k-1}) \; h_k \tag{3.60}$$

Although the computation of the sum bits seems to be more complex, a carry-select structure requires only one 2:1 multiplexer after the computation of h_{k+1}, which is of similar delay as the xor used in the conventional approach. The additional xor gate required to compute $p_k \oplus a_{k-1}$ is out of the critical path (the carry path).

BCD prefix adders based on conditional speculative decimal addition may also use Ling carries to compute the BCD sum digits. To obtain the expression of BCD sum digits S_i as a function of the Ling carries h_i, we replace c_k by $h_k \; a_{k-1}$ in equation (3.38). In this way, we get the following expression (subscript k is changed by double subscript notation (i, j) as before):

$$S_i = \begin{cases} s_{i,3} = & p_{i,3}\overline{h_{i,3}} \vee (p_{i,3} \oplus a_{i,2}) h_{i,3} \\ s_{i,2} = & \overline{p_{i,3} \; p_{i,2} \; p_{i,1}} \; (p_{i,2}\overline{h_{i,2}} \vee \overline{p_{i,3}} \; (p_{i,2} \oplus a_{i,1}) \; h_{i,2}) \\ s_{i,1} = & \overline{p_{i,3} \; p_{i,2}} \; (p_{i,1}\overline{h_{i,1}} \vee \; (p_{i,1} \oplus a_{i,0}) \; h_{i,1}) \\ s_{i,0} = & p_{i,0}\overline{h_{i,0}} \vee (p_{i,0} \oplus a_{i-1,3}) h_{i,0} \end{cases} \tag{3.61}$$

A BCD sum cell implementing equations (3.61) is also of similar delay than a conventional binary xor sum cell.

Therefore, we conclude that it is only necessary to introduce the following changes in any of the proposed binary/BCD prefix adders to get the corresponding Ling prefix tree adder:

1. The prefix operations are performed over pairs (g_j, a_{j-1}) instead of (g_j, a_j).
2. The prefix operations at the first level of the tree are simplified since the $\circ$ operator is used.

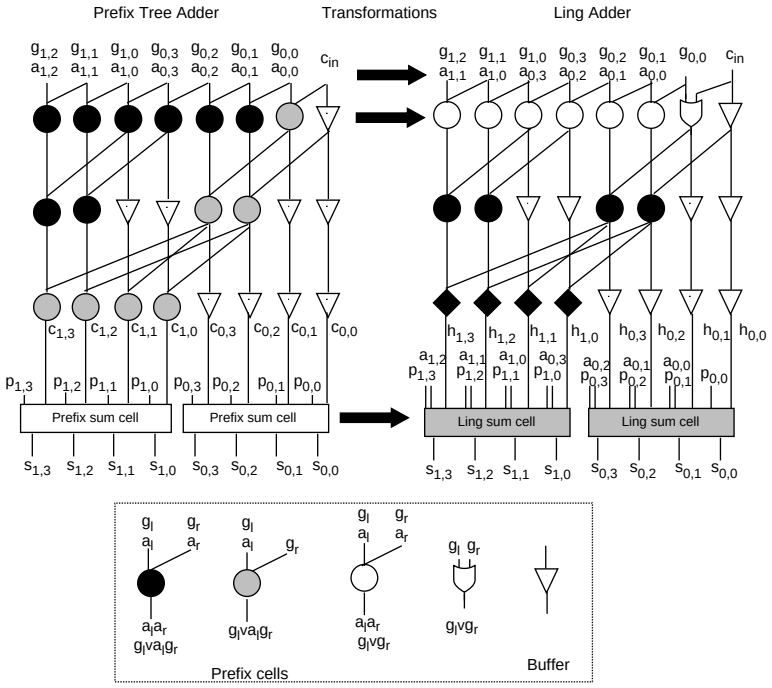
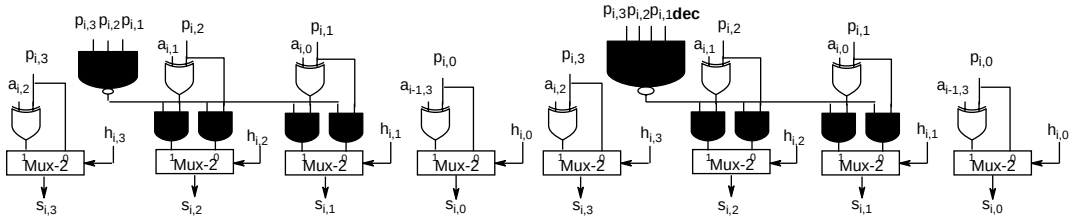


Figure 3.19. *Transformation of a prefix adder into a Ling adder.*



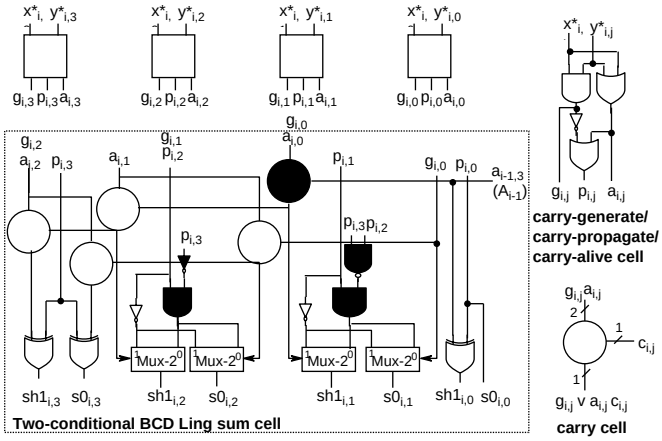
(a) BCD Ling sum cell. (b) 4-bit Binary/BCD Ling sum cell.

Figure 3.20. *Implementation of Ling digit sum cells.*

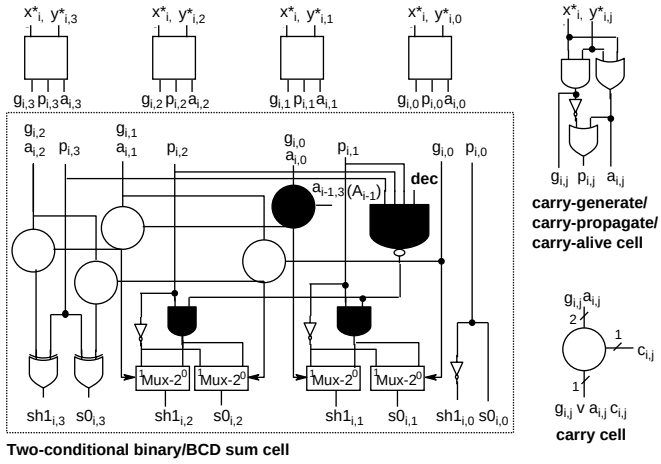
3. The sum digits are expressed as a function of the Ling carries h_k instead of the conventional carries c_k. Thus, for a binary adder, the sum bits are obtained as indicated by (3.60). For a 10's complement BCD adder based on the conditional speculative decimal addition method, the BCD sum digits are obtained as indicated by (3.61).

In Fig. 3.19 we show an example of the transformation process for a prefix tree adder into a Ling prefix tree adder. The prefix operators $\bullet$ at the first level of the tree are replaced by $\circ$ operators. The transformation process is completed replacing the prefix sum cell of Fig. 3.10(c) by the BCD Ling sum cell of Fig. 3.20(a). Black-filled gates represent the additional hardware with respect to a binary Ling sum cell which implements (3.60). The equivalent binary/BCD Ling sum cell is shown in Fig. 3.20(b). Control signal dec is activated only for decimal operations.

For the hybrid prefix tree/carry-select architectures of Section 3.3.2, the transformation sequence is straightforward:



(a) BCD Ling sum cell.



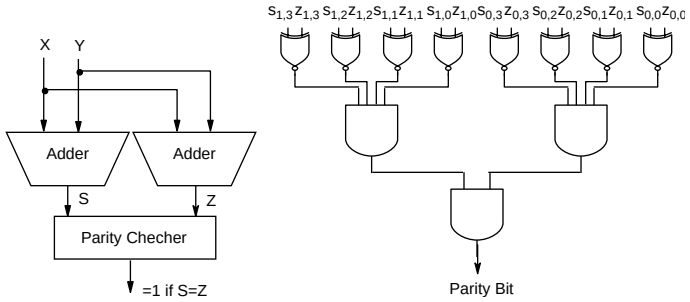
(b) 4-bit binary/BCD Ling sum cell.

Figure 3.21. *Implementation of two-conditional Ling digit sum cells.*

- For the quaternary sparse prefix tree of Fig. 3.16, the Ling carries at decimal positions H_i are computed introducing the same transformations as in Fig. 3.19, since $C_i = c_{i,0} = a_{i-1,3}\ h_{i,0} = a_{i-1,3}\ H_i$.
- For the direct decimal prefix tree of Fig. 3.17 it is also possible to define a Ling direct decimal carry recurrence as $H_{i+1} = G_i \vee A_{i-1}\ H_i$. G_i and A_{i-1} are the decimal carry generate at position i and the decimal carry alive at position $i-1$. Using the relation $G_i\ A_i = A_i$ and $C_{i+1} = G_i \vee A_i\ C_i$, we have that $C_i = A_{i-1}\ H_i$. Therefore the same transformation process of Fig. 3.19 is also valid for the direct decimal prefix tree using G_i and A_{i-1} instead of the equivalent binary functions g_k and a_{k-1}.
- The two-conditional BCD sum cell of Fig. 3.14 is replaced by the two-conditional Ling BCD sum cell of Fig. 3.21(a).

This digit addition is computed using the decimal Ling carries H_i as

$$S_i = H_i\ a_{i-1,3} S1_i \vee \overline{H_i\ a_{i-1,3}} S0_i$$



(a) General layout. (b) Parity checker.

Figure 3.22. *Sum error checking using unit replication.*

$$\begin{aligned} &= \left(S1_i \; a_{i-1,3} \vee S0_i \overline{a_{i-1,3}}\right) H_i \vee S0_i \overline{H_i} \\ &= SH1_i \; H_i \vee S0_i \overline{H_i} \end{aligned} \tag{3.62}$$

while $C_i = c_{i,0} = h_{i,0} \; a_{i-1,3} = H_i \; a_{i-1,3}$. For the direct decimal prefix tree the term $a_{i-1,3}$ corresponds to A_{i-1}. $S1_i$ and $S0_i$ are the two-conditional BCD digit sums given by (3.39). The term $SH1_i = S1_i a_{i-1,3} \vee S0_i \overline{a_{i-1,3}}$ is efficiently computed using $a_{i-1,3}$ as a carry input instead of the logical one in the corresponding two-conditional adder (this is performed by the black prefix cell in Fig. 3.21). Therefore, we do not expect any significant increase in hardware complexity of an hybrid sparse prefix tree/carry-select adder due to the Ling transformation.

$S1_i$ and $S0_i$ can also represent two-conditional 4-bit binary sums. For this case, the mixed binary/BCD two-conditional Ling sum cell is shown in Fig. 3.21(b).

3.4 Sum error detection

Users of financial and e-commerce services demand a high degree of reliability. By other hand, VLSI circuits are increasingly becoming very sensitive to noise and transient particles due to the higher densities and reduced sizes of transistors on a chip. This fact may cause frequent transient upsets in a circuit producing a failure (soft error) that can lead to an incorrect result. Therefore, high-end microprocessors support RAS (reliability-accessibility-serviceability) features for soft error detection and recovery [4, 98, 102, 105].

Adders are specially sensitive to soft errors because of their intensive use and the high clock frequencies of operation. To protect the arithmetic units of high-end microprocessors against soft errors, they are replicated and the output of both units compared using parity checking [102]. Fig. 3.22(a) shows a conventional layout based on unit replication to detect a soft error in an addition. It consists on two adders that evaluate the same operation in parallel and a parity checker (see Fig. 3.22(b)) to compare the output of the adders.

Due to the high cost in area of this solution, the fixed and floating-point units incorporate some other mechanisms for error detection such as parity prediction [105], residue checking [4, 98] or use dual-rail logic circuit technologies with parity checking [105] (by XOR-ing the true and the complement outputs). However, the parity prediction algorithms for the arithmetic functions are expensive both in terms of area and speed while residue checking is only

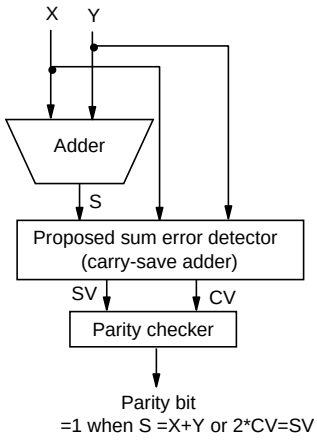


Figure 3.23. *Proposed scheme for sum error checking.*

efficient for multiplication and division.

Another approach, proposed in [166], is the use of a simple lazy unit for sum error detection. It consists of a fast carry-save adder and a parity checker, which has a negligible impact on performance in "out-of-order" microprocessors (the instruction's results are speculative for one or more cycles). Input operands X and Y and the sum result S are the inputs of a modified carry-save adder, which performs the following bit-level operations:

$$\begin{aligned} sv_k &= x_k \oplus y_k \oplus s_k = p_k \oplus p_k \oplus c_k = c_k \\ cv_k &= x_k\ y_k \vee (x_k \vee y_k)\ sv_k = x_k\ y_k \vee (x_k \vee y_k)\ c_k = c_{k+1} \end{aligned} \tag{3.63}$$

Each output sv_k is compared with cv_{k-1} to check if

$$err = \sum_{k=0}^{n-1} sv_k \oplus cv_{k-1} = 0 \tag{3.64}$$

where $\sum_{k=0}^{n-1}$ stands for the logical OR operation. A logical one means an error in some bit of the sum.

In a similar way, we consider a simple algorithm to check possible errors in BCD and combined binary/BCD addition/subtraction by using only a fast carry-save adder and a parity checker. Fig. 3.23 shows the diagram of the proposed method.

The main difference with respect to the previous proposal for binary addition/subtraction is that we compare with the complement of the sum $\overline{S}$, and that we use the conventional binary carry-save addition equations as

$$\begin{aligned} sv_k &= x_k \oplus y_k \oplus \overline{s_k} = p_k \oplus p_k \oplus \overline{c_k} = \overline{c_k} \\ cv_k &= x_k\ y_k \vee (x_k \vee y_k)\ \overline{s_k} = x_k\ y_k \vee (x_k \vee y_k)\ c_k = c_{k+1} \end{aligned} \tag{3.65}$$

Therefore, to detect an error in the sum we have to check if

$$\prod_{k=0}^{n-1} sv_k \oplus cv_{k-1} = \prod_{k=0}^{n-1} (\overline{c_k} \oplus c_k) \tag{3.66}$$

is zero, where $\sum_{k=0}^{n-1}$ stands for the AND operation.

The advantage of this second approach is that it can be formulated in terms of word-length operands X, Y and S, obtaining also a simple condition for BCD 10's complement and sign-magnitude BCD addition/subtraction. Next, we describe the application of this method to 2's complement binary, 10's complement BCD and mixed binary/BCD adders.

3.4.1 2's complement binary addition

To check the correctness of a 2's complement addition we use the following condition:

$$X + Y = S \Rightarrow X + Y + \overline{S} + 1 = 0 \Rightarrow X + Y + \overline{S} = -1 \tag{3.67}$$

This was used by other authors [32] to check conditions of the type $A + B = K$.

For $n + 1$ precision bits (including a sign bit), the two's complement of 1 is an array of $n + 1$ ones. In this way, the condition (3.67) is transformed into:

$$X + Y + \overline{S} = -2^n + \sum_{k=0}^{n-1} 2^k \tag{3.68}$$

In the case of 2's complement subtraction we have

$$X - Y = S \Rightarrow X + \overline{Y} + \overline{S} = -2 \Rightarrow X + \overline{Y} + \overline{S} = -2^n + \sum_{k=1}^{n-1} 2^k \tag{3.69}$$

Conditions (3.68) and (3.69) are formulated in a single expression as

$$X + Y^* + \overline{S} = -2^n + \sum_{k=1}^{n-1} 2^k + \overline{sub} \tag{3.70}$$

where sub points out a subtraction and $Y^* = Y$ for addition or $Y^* = \overline{Y}$ for subtraction. Using a binary carry-save addition to compute $X + Y^* + \overline{S} = 2 \cdot C + SV$, condition (3.70) is only verified if all the XORed carry and sum bits at each position are one, that is

$$X + Y^* + \overline{S} = SV + 2 \cdot C = -2^n + \sum_{k=1}^{n-1} 2^k + \overline{sub} \Leftrightarrow \prod_{k=0}^{n} (sv_k \oplus c_{k-1}) == 1 \tag{3.71}$$

where $c_{-1} = sub$ and $\prod$ is a product of logical AND gates. Fig. 3.24 shows the proposed sum error detector for an 8-bit 2's complement binary addition/subtraction. Before 2's complement subtraction, operand Y is inverted and a 'hot one' is used as a carry input. Thus, a 2's complement binary addition/substraction can be checked using a level of 3:2 CSA (carry-save adder or full adder) and a parity checker which performs the XOR of each pair of bits at each position and a subsequent logical AND over all the XOR's output bits.

3.4.2 10's complement BCD addition

A similar verification scheme can be applied for 10's complement addition and subtraction. In this case, for $p + 1$-digit precision we have the following condition,

$$X + Y = S \Rightarrow X + Y - s_p \cdot 10^p + \sum_{i=0}^{p-1} (9 - S_i) \cdot 10^i = -1 \Rightarrow$$

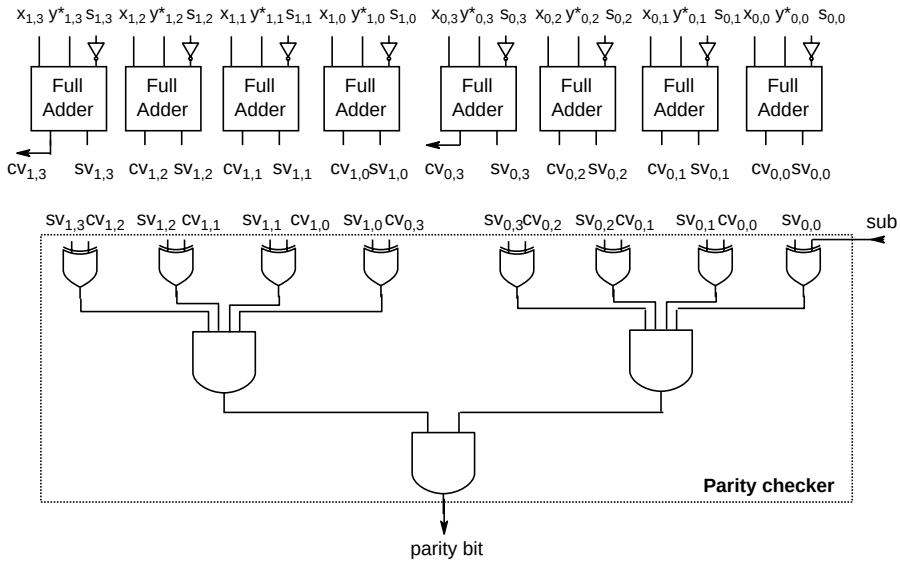


Figure 3.24. *Proposed architecture for the detection of 2's complement sum errors (8-bits).*

$$X + Y + \overline{S} - \sum_{i=0}^{p-1} 6 \cdot 10^i = -1 \tag{3.72}$$

where we have used $9 - S_i = \overline{S_i} - 6$ and $s_p \in \{0, 1\}$. The 10's complement of 1 is represented as an array of p 9's and a leading sign bit one. Introducing this value in the previous expression we obtain

$$\begin{aligned} X + Y + \overline{S} &= -10^p + \sum_{i=0}^{p-1} (9 + 6) \cdot 10^i \Rightarrow \\ X + Y + \overline{S} &= -2^0 \cdot 10^p + \sum_{i=0}^{p-1} \sum_{j=0}^{3} 2^j \cdot 10^i \end{aligned} \tag{3.73}$$

The condition of correctness for 10's complement substraction is given by

$$\begin{aligned} X + \neg Y + \overline{S} &= -10^p + \sum_{i=1}^{p-1} 15 \cdot 10^i + 14 \Rightarrow \\ X + \neg Y + \overline{S} &= -2^0 \cdot 10^p + \sum_{i=1}^{p-1} \sum_{j=0}^{3} 2^j \cdot 10^i + \sum_{j=1}^{3} 2^j \end{aligned} \tag{3.74}$$

where $\neg Y = \sum_{i=0}^{p-1} \overline{Y_i + 6}\ 10^i$ is the 9's complement of Y.

Expressions (3.73) and (3.73) are reformulated in a single condition as

$$X + Y^* + \overline{S} = -2^0 \cdot 10^p + \sum_{i=1}^{p-1} \sum_{j=0}^{3} 2^j \cdot 10^i + \sum_{j=1}^{3} 2^j + \overline{sub} \tag{3.75}$$

where

$$Y_i^* = \begin{cases} Y_i & \texttt{If}(sub == 0) \\ \neg Y_i = \overline{Y_i + 6} & \texttt{Else} \end{cases} \tag{3.76}$$

Thus, since $n = 4p$, an error in the 10's complement BCD addition $S = X \pm Y$ can be detected comparing the result of $X + Y^* + \overline{S}$ with a $n + 1$-bit row of ones, as in 2's complement binary addition.

Condition (3.75) is stated for a digit $0 < i < p$ as

$$X_i + Y_i^* + \overline{S_i} + C_i^* = 10 \cdot C_{i+1}^* + 15 \tag{3.77}$$

where C_i^* and C_{i+1}^* are the decimal input and output carries of $X + Y^* + \overline{S}$ at position i. Introducing in (3.77) the following relations,

$$\begin{aligned} S_i &= mod_{10}(X_i + Y_i^* + C_i) \\ \overline{S_i} &= 15 - S_i \\ X_i + Y_i^* + C_i &= 10 \cdot C_{i+1} + S_i \end{aligned} \tag{3.78}$$

we have that the decimal carries C_i^* are equal to the decimal carries C_i of $S = X + Y^*$. Therefore, condition (3.75) is equivalent to

$$X_i + Y_i^* + \overline{S_i} + C_i = 10 \cdot C_{i+1} + 15 \tag{3.79}$$

for $0 < i < p$ and

$$X_0 + Y_0^* + \overline{S_0} = 10 \cdot C_1 + 14 + \overline{sub} \tag{3.80}$$

for $i = 0$.

To perform this comparison using decimal carry-save arithmetic we propose to use a conventional binary carry-save addition over the three BCD operands obtaining a sum word SV and a carry word CV as

$$\begin{aligned} sv_{i,j} &= x_{i,j} \oplus y_{i,j}^* \oplus \overline{s_{i,j}} \\ cv_{i,j} &= (x_{i,j}\ y_{i,j}^*) \vee (x_{i,j} \vee y_{i,j}^*)\ \overline{s_{i,j}} \end{aligned} \tag{3.81}$$

Condition (3.75) is expressed in terms of SV and CV as

$$X + Y^* + \overline{S} = SV + L1_{shift}[H] = -10^p + \sum_{i=1}^{p-1} 15\ 10^i + 14 + \overline{sub} \tag{3.82}$$

where $L1_{shift}$ denotes a 1-bit binary wired left shift and

$$HV_i = \begin{cases} CV_i + 3, & CV_i \in [5, 12] \quad \texttt{If}(C_{i+1} == 1) \\ CV_i \in [0, 7] & \texttt{Else} \end{cases} \tag{3.83}$$

The term HV_i is introduced to correct the 4-bit vector representation of CV_i when a decimal carry-out C_{i+1} is generated at position i, as stated in (3.79). That is, C_{i+1} is 1 when the sum $X_i + Y_i^* + \overline{S_i} + C_i = SV_i + (L1_{shift}[HV])_i$ is equal to $10 \cdot hv_{i,3} + 15 = 25$. Note that $C_{i+1} = hv_{i,3} = (L1_{shift}[HV])_{i+1,0}$. In this case, a +3 is added (digitwise) to CV_i to get the correct BCD representation after the left shifting [12].

[12] This is equivalent to shift CV_i 1-bit to the left and then add +6.

The boolean expressions for $hv_{i,j}$ are given by

$$\begin{aligned} hv_{i,3} &= C_{i+1} \\ hv_{i,2} &= cv_3 \ (cv_{i,2} \vee cv_{i,1} \vee cv_{i,0}) \ C_{i+1} \vee cv_{i,2} \ \overline{C_{i+1}} \\ hv_{i,2} &= \overline{cv_{i,1} \oplus cv_{i,0}} \ C_{i+1} \vee cv_{i,1} \ \overline{C_{i+1}} \\ hv_{i,0} &= \overline{cv_{i,0}} \ C_{i+1} \vee cv_{i,0} \ \overline{C_{i+1}} \end{aligned} \tag{3.84}$$

Therefore, in terms of SV and HV, condition (3.80) is verified when

$$\prod_{k=0}^{n} (sv_k \oplus hv_{k-1}) = 1 \tag{3.85}$$

where $\prod$ stands for the logical AND operation, $n = 4p$, $k = 4i + j$ and $hv_{-1} = sub$.

Fig. 3.25 shows the proposed error checking architecture for 10's complement addition/subtraction (2-digit slice). The architecture consists of a binary 3:2 CSA with inputs X,

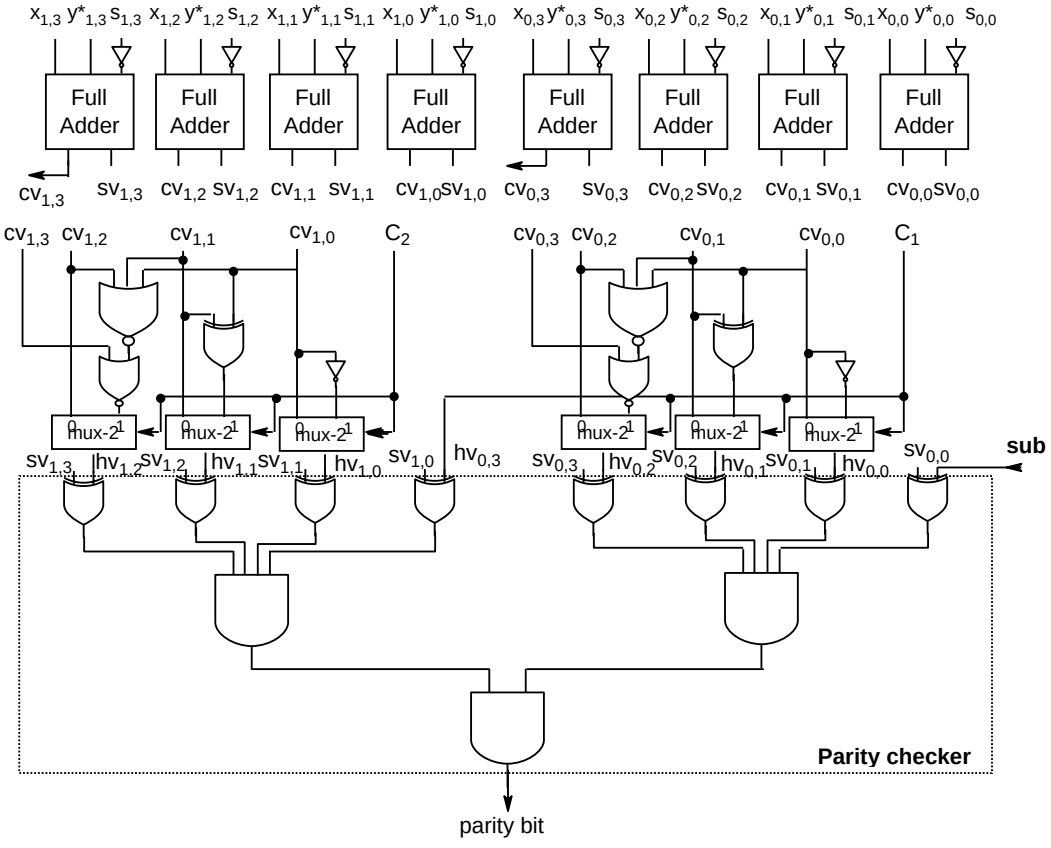


Figure 3.25. *Proposed architecture to check BCD addition/subtraction errors (2 digits).*

Y^* and $\overline{S}$, a block to compute HV from the carry bit vector CV, which implements (3.84) and a parity checker. Control signal sub indicates a subtraction.

3.4.3 Mixed binary/BCD addition

The extension of the sum checker architecture to support the mixed 2'complement binary/10's complement BCD addition is straightforward. Fig. 3.26 shows an scheme for the proposed error detector.

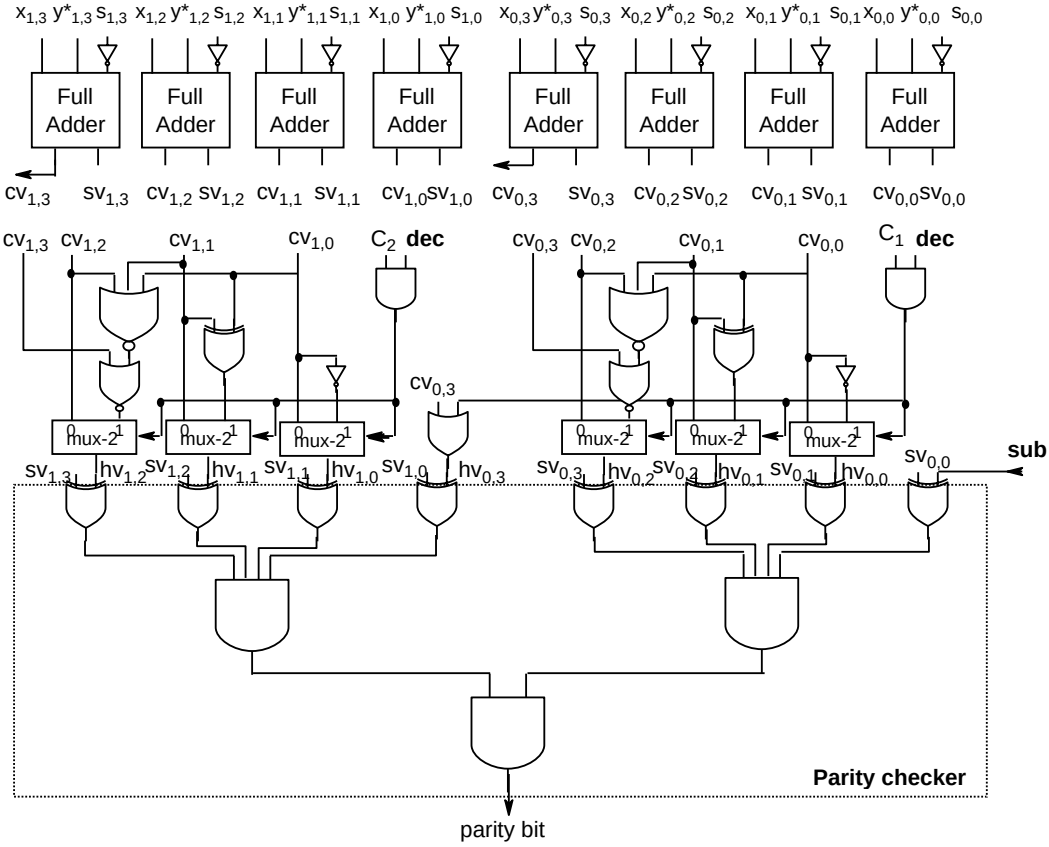


Figure 3.26. *Proposed architecture to detect decimal and binary sum errors (8-bits)*

The only significant change with respect to the decimal architecture of Fig. 3.25 affects to equations (3.84) for $hv_{i,j}$. A control signal dec is used to indicate a decimal operation, so the equations of $hv_{i,j}$ for the mixed binary/10's complement BCD checker are given by

$$\begin{aligned} hv_{i,3} &= cv_3 \vee C_{i+1}\ dec \\ hv_{i,2} &= cv_3\ (cv_{i,2} \vee cv_{i,1} \vee cv_{i,0})\ (C_{i+1}\ dec) \vee cv_{i,2}\ \overline{C_{i+1}\ dec} \\ hv_{i,2} &= \overline{cv_{i,1} \oplus cv_{i,0}}\ (C_{i+1}\ dec) \vee cv_{i,1}\ \overline{C_{i+1}\ dec} \\ hv_{i,0} &= \overline{cv_{i,0}}\ (C_{i+1}\ dec) \vee cv_{i,0}\ \overline{C_{i+1}\ dec} \end{aligned} \tag{3.86}$$

The other minor difference affect to the input Y^*. For compatibility of both binary and BCD operations, Y^* is defined as

$$Y_i^* = \begin{cases} Y_i & \texttt{If}(sub == 0) \\ \neg Y_i = \overline{Y_i + 6} & \texttt{Else If}(sub == 1\ \texttt{AND}\ dec == 1) \\ \overline{Y_i} & \texttt{Else} \end{cases} \tag{3.87}$$

3.5 Evaluation results and comparison

The area and delay figures used in this work for evaluation and comparisons have been estimated from gate level descriptions of the architectures. For this purpose, we have developed a rough evaluation model for CMOS logic circuits to obtain area and delay estimations just performing simple hand calculations. This method allows us to explore the advantages and costs of the different algorithms and architectures in a faster and more flexible way than using time-consuming synthesis tools, limited also by the target technology. The area and delay evaluation model is detailed in Appendix A.

The delay model is based on the logical effort method [131]. It has been widely used to evaluate and improve critical timings in many designs [54, 126]. We introduce some simplifications to allow for faster hand calculations. We consider input and output loads, but gate sizing optimizations and the effect of interconnections are beyond the scope of this model. Instead, we assume gates with the drive strength of the minimum sized (1x) inverter using buffers for high loads. This assumption is supported by the results presented a in recent paper [111], that deals with the evaluation of CMOS adders. In order to provide fair comparison results, we consider that the output load (C_{out}) of each architecture is equal to its input load (C_{in}). The delay is given in FO4 units (delay of a 1x inverter with a fan-out of four 1x inverters). The area or hardware complexity of a design is given as the number of equivalent minimum size two-input NAND gates (NAND2 units).

We do not expect this rough model to give absolute area and delay figures, due to the high wiring complexity of current deep sub-micron technologies [68]. However we consider that it is good enough for making design decisions at gate level and that it provides quite accurate area and delay ratios to compare different designs. Moreover, it is expected that two architectures with similar structures scale in the same way when implemented in the same technology.

To extract fair conclusions from comparison results, we must be aware of that the comparative advantages are not due to a better circuit logic implementation, but a faster algorithm or a more advanced architectural design. Therefore, for each algorithm we provide the best possible architecture, and for each architecture, several implementations with representative circuit topologies.

In this way, we have estimated the area and delay of the different binary, decimal BCD and mixed binary/BCD adders discussed in this Chapter, using several representative prefix tree topologies: Kogge-Stone (**K-S**) [86], Ladner-Fischer (**L-F**) [89], a quaternary-tree (**Q-T**) [100] and the Ling scheme [95]. We provide area-delay figures for the operand lengths of most common use in decimal arithmetic: 64-bit or 16-BCD digit coefficients (Decimal64 format) and 136-bit or 34-BCD digit coefficients (Decimal128 format). We have also evaluated the area and delay of the sum error check architectures of Section 3.4. Next, we present the results of this evaluation and a comparative study among the different proposals.

3.5.1 Evaluation results

The area and delay estimations are presented in three separate groups:

Prefix tree topology	Total		Stages			
	Delay (# FO4)	Area (Nand2)	Setup* Delay/Area	Pre-sum Delay/Area	Carry* Delay/Area	Sum* Delay/Area
16-BCD digit adders						
Prefix K-S	**16.9**	**2400**	6.3/880	2.0/240	7.6/1000	3.0/280
Prefix L-F	**18.5**	**2030**	6.3/880	2.0/240	9.2/630	3.0/280
Prefix Q-T	**15.9**	**2220**	4.0/760	5.6/820	8.4/370	3.5/230
Ling K-S	**16.6**	**2590**	6.3/880	2.0/240	7.3/940	3.0/530
Ling L-F	**18.2**	**2250**	6.3/880	2.0/240	8.9/600	3.0/530
Ling Q-T	**15.6**	**2260**	4.0/760	5.6/930	8.1/340	3.5/230
34-BCD digit adders						
Prefix K-S	**19.3**	**5630**	6.3/1880	2.0/510	10.0/2650	3.0/590
Prefix L-F	**20.7**	**4800**	6.3/1880	2.0/510	11.4/1820	3.0/590
Prefix Q-T	**18.1**	**4920**	4.0/1610	5.6/1760	10.6/1050	3.5/500
Ling K-S	**19.0**	**6020**	6.3/1880	2.07510	9.7/2520	3.0/1150
Ling L-F	**20.4**	**5260**	6.3/1880	2.0/510	11.1/1750	3.0/1150
Ling Q-T	**17.8**	**4950**	4.0/1610	5.6/1950	10.3/890	3.5/500

* Stages in the critical path.

Table 3.2. *Delay and area figures for 10's complement adders.*

- 10's complement adders.
- Mixed 2's complement/10's complement adders.
- Architectures for sum error checking.

We have considered the following stages for the adders: i) operand setup (hardware to prepare operands for binary addition/subtraction), ii) pre-sum (logic to compute the two-conditional pre-sums or the binary carry-propagate signals) iii) carry computation (including the evaluation of the carry-generate and carry-alive signals and the carry tree logic), iv) sum (determination of final sum digits depending on carries).

Table 3.2 presents the evaluation results for the 10's complement architectures of Section 3.3.

We have analyzed six different configurations for both 16 and 34 BCD digits. Below each column, we present the delay and area estimations for each adder or the corresponding stage in number of FO4 and NAND2 units. For 16-digit and 34-digit operand lengths, the pre-sum stage (constant delay) does not contribute to the total delay. The critical path delay is the sum of the delays of the operand setup stage, carry computation (logarithmic delay) and sum stage.

The adders based on the Kogge-Stone (**Prefix K-S**, **Ling K-S**) and Ladner-Fischer (**Prefix L-F**, **Ling L-F**) carry tree topologies implement the architecture of Fig. 3.10. The critical path delay of the setup stage (Fig. 3.10(b)) is given by the computation of the selection signal A_i^{U-} plus the delay required to distribute this signal to three 2:1 multiplexes.

The **Prefix K-S** adder uses the high-density logic prefix tree topology of Fig. 3.11(b) while the **Prefix L-F** adder implements the reduced logic prefix tree of Fig. 3.11(c). The equivalent **Ling K-S** and **Ling L-F** adders were obtained applying the transformation method of Section 3.3.3 to the corresponding **K-S** and **L-F** prefix topologies.

In the case of the quaternary tree configurations (**Prefix Q-T** and **Ling Q-T**), the general architecture is shown in Fig. 3.13 and the carry tree in Fig. 3.16. The preferred configuration for the operand setup stage is the one shown in Fig. 3.15(b). The critical path goes through the $\overline{Y_i + 6}$ block and a level of 2:1 muxes. The stage delay is reduced from the 6.3 FO4 of the configurations of Fig. 3.10(b) or Fig. 3.15(a) to 4 FO4. This leads to the fastest **Q-T** adder with a moderate area overhead with respect to the configuration of Fig. 3.15(c).

The pre-sum stage of the **Prefix Q-T** and **Ling Q-T** adders implements the BCD two-conditional adders of Fig. 3.14 and Fig. 3.20(b) respectively.

The carry computation includes the generation of carry-generate and carry-alive signals and the computation of the carries in the prefix tree. The binary carry-generate and carry-alive signals are evaluated in a single level of NAND2–NOR2 CMOS gates. The nodes of the prefix tree are implemented in CMOS logic alternating logical levels of AOI–NAND2 and OAI–NOR2 gates. The number of levels of the minimum logical-depth prefix trees for 16 and 34 digits are 6 and 8 respectively. The graph representations of the prefix trees show that the critical path goes from the top-right corner to the bottom-left corner, which corresponds with the largest wire delay. We have neglected the impact of wire delay, which could favor, at first glance, the K-S topology with respect to the L-F and Q-T topologies. However, the fanout of the last level of the Q-T is higher than the K-S one (minimum fanout at each level), and thus, the quaternary carry tree is slightly slower than the K-S carry tree.

The critical path delay of the sum stage is equivalent to the delay of an XOR gate (assuming an output load equal to the input capacitance of each adder).

From the area and delay estimations of Table 3.2, we conclude that the better configuration for a 10's complement BCD adder is the quaternary tree. It is the fastest alternative (more than 1 FO4 faster) and presents the better area and delay trade-offs. Moreover, the Ling scheme applied to the **Prefix Q-T** adder (**Ling Q-T** adder) leads to a slightly faster adder (about 3 % faster) with a similar area.

The area and delay evaluation results for the mixed 2's complement/10's complement architectures are presented in Table 3.3. Fig. 3.12 shows the mixed binary/BCD architecture for the full binary prefix tree topologies (**K-S** and **L-F**). For the **Q-T** topologies, the architecture is shown in Fig. 3.18. Comparing the estimated area and delay figures with those of Table Table 3.2, it can be seen that, extending the 10's complement BCD adders to mixed binary/BCD adders, only affects to the area figures. Thus, the area overhead ranges from 3% in the case of **Prefix K-S** and **Prefix L-F** adders to less than 10% in the case of the **Ling Q-T** adder.

In Table 3.4 we present the estimated area and delay figures for the different sum checkers of Section 3.4: 2's complement (Fig. 3.24), 10's complement (Fig. 3.25) and mixed 2's complement/10's complement (Fig. 3.26). The contribution of the carry-save adder and the parity checker blocks to the overall area and delay is also detailed. The proposed sum checkers cost approximately half the area and have less latency than the fastest adder of each type. Since the sum checkers are separated from the adders, these units have no impact in the

Prefix tree topology	Total		Stages			
	Delay (# FO4)	Area (Nand2)	Setup* Delay/Area	Pre-sum Delay/Area	Carry* Delay/Area	Sum* Delay/Area
64-bit/16-BCD digit adders						
Prefix K-S	**17.1**	**2470**	6.3/880	2.0/240	7.6/1000	3.2/350
Prefix L-F	**18.7**	**2100**	6.3/880	2.0/240	9.2/630	3.2/350
Prefix Q-T	**16.1**	**2460**	4.0/990	5.6/870	8.4/370	3.7/230
Ling K-S	**16.8**	**2650**	6.3/880	2.0/240	7.3/940	3.2/590
Ling L-F	**18.4**	**2310**	6.3/880	2.0/240	8.9/600	3.2/590
Ling Q-T	**15.8**	**2520**	4.0/990	5.6/960	8.1/340	3.7/230
136-bit/34-BCD digit adders						
Prefix K-S	**19.5**	**5730**	6.3/1880	2.0/510	10.0/2650	3.2/690
Prefix L-F	**20.9**	**4900**	6.3/1880	2.0/510	11.4/1820	3.2/690
Prefix Q-T	**18.3**	**5420**	4.0/2110	5.6/1760	10.6/1050	3.7/500
Ling K-S	**19.2**	**6120**	6.3/1880	2.0/510	9.7/2520	3.2/1250
Ling L-F	**20.6**	**5360**	6.3/1880	2.0/510	11.1/1750	3.2/1250
Ling Q-T	**18.0**	**5500**	4.0/2110	5.6/2000	10.3/890	3.7/500

* Stages in the critical path.

Table 3.3. *Delay and area figures for 2's complement/10's complement adders.*

Architecture	Total		Stages			
			Carry-save adder		Parity checker	
	Delay (# FO4)	Area (Nand2)	Delay (# FO4)	Area (Nand2)	Delay (# FO4)	Area (Nand2)
16-BCD digit architectures						
2's complement	9.0	1000	3.4	700	5.6	300
10's complement	12.6	1260	7.0	960	5.6	300
Mixed bin/dec	12.6	1300	7.0	1000	5.6	300
34-BCD digit architectures						
2's complement	9.6	2160	3.4	1480	6.2	680
10's complement	13.2	2720	7.0	2040	6.2	680
Mixed bin/dec	13.2	2790	7.0	2110	6.2	680

Table 3.4. *Evaluation results for sum error checkers.*

processor cycle time, usually determined by the adder latency. Moreover, the sum checking takes at most one cycle after the sum execution. Thus, in some microprocessors[13], it could be performed in the time interval between the sum completion (obtention of result) and the instruction commit, removing error detection from the critical path. In this case, this sum checking would have no effect on the microprocessor performance.

[13]For instance, in superscalar "out-of-order" microprocessors, such as the Intel Pentium4 or the IBM Power5.

Prefix tree topology	Binary Adders		Decimal addition method			
			Proposed		Speculative/†Direct	
	Delay	Area	Delay #FO4/ratio	Area Nand2/ratio	Delay #FO4/ratio	Area Nand2/ratio
64-bit/16-BCD digit adders						
Prefix K-S	12.7	1720	**16.9/1.35**	**2400/1.40**	18.7/1.45	2200/1.30
Prefix L-F	14.3	1350	**18.5/1.30**	**2030/1.50**	20.3/1.40	1750/1.30
Prefix Q-T	13.9	1640	**15.9/1.15**	**2220/1.35**	15.9/1.15	2370/1.45
†Prefix Q-T					16.6/1.20	2050/1.25
Ling K-S	12.4	1880	**16.6/1.35**	**2590/1.40**	18.4/1.50	2400/1.30
Ling L-F	14.0	1540	**18.2/1.30**	**2250/1.45**	20.0/1.40	1950/1.25
Ling Q-T	13.6	1670	**15.6/1.15**	**2260/1.35**	15.6/1.15	2450/1.50
†Ling Q-T					16.3/1.20	2150/1.30
136-bit/34-BCD digit adders						
Prefix K-S	15.1	4180	**19.3/1.30**	**5630/1.35**	21.1/1.40	5150/1.25
Prefix L-F	16.5	3350	**20.7/1.25**	**4800/1.45**	22.5/1.35	4300/1.30
Prefix Q-T	16.1	3690	**18.1/1.10**	**4920/1.35**	18.1/1.10	5250/1.40
†Prefix Q-T					18.9/1.20	4570/1.25
Ling K-S	14.8	4500	**19.0/1.30**	**6020/1.35**	20.8/1.40	5520/1.20
Ling L-F	16.2	3740	**20.4/1.25**	**5260/1.40**	22.2/1.40	4750/1.25
Ling Q-T	15.8	3730	**17.8/1.15**	**4950/1.30**	17.8/1.15	5280/1.40
†Ling Q-T					18.5/1.20	4800/1.30

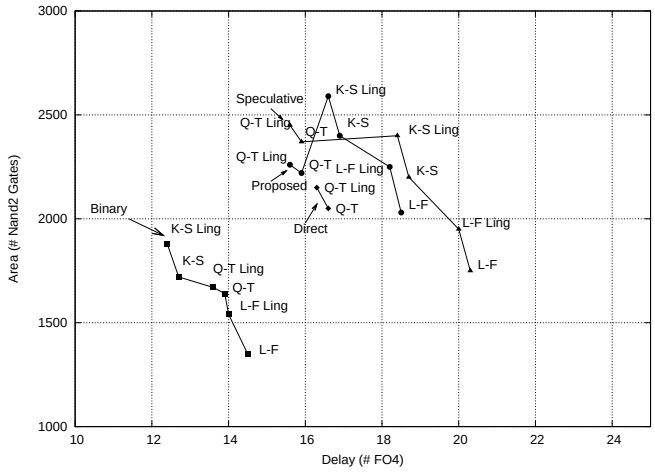
Table 3.5. *Delay and area figures for 10's complement BCD adders.*

3.5.2 Comparison

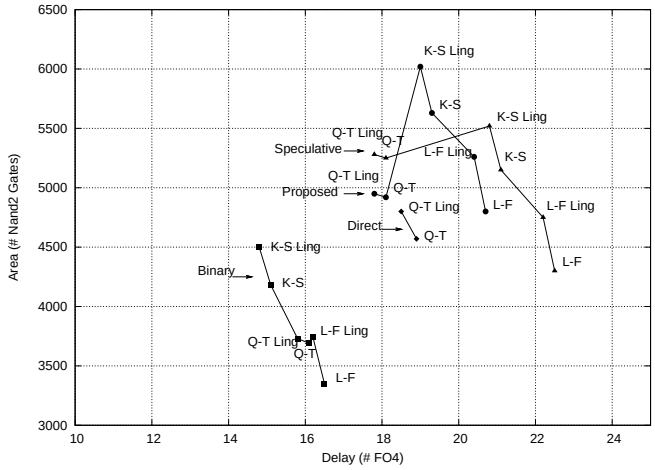
In this Section we present the results of comparative study among the different decimal addition methods analyzed in this Chapter. Table 3.5 presents the area and delay figures for 16-digit and 34-digit 10's complement BCD adders. We have implemented each method using several full binary (Kogge-Stone and Ladner-Fischer), quaternary and Ling prefix tree topologies. As a reference we also show the results for the corresponding binary adders. Beside each absolute value (area in NAND2 or delay in # FO4) we present a ratio value as the quotient of the area or delay for the decimal adder over the corresponding magnitude for the equivalent binary adder topology. This ratio represents how many times is more costly or slower the decimal adder with respect to a binary (2's complement) adder with a similar prefix or Ling topology (for instance, 1.30 means 30% more area or 30% slower).

The two rightmost columns may contain the area-delay figures for the speculative or for the direct decimal adders. We mark with a † symbol only the rows containing the area-delay figures for the direct decimal architectures. Note that the direct decimal addition method is not implementable in a full binary prefix tree topology, since only the decimal carries and not the binary carries are defined.

To extract easily conclusions from the comparison, we represent this area-delay values graphically in Fig. 3.27(a) for 16 digits and in Fig. 3.27(b) for 34 digits. We also provide the area-delay curve of binary adders as a reference.



(a) 16-BCD digits.



(b) 34-BCD digits.

Figure 3.27. *Area/delay space of 10's complement BCD adders.*

Speculative adders with a full binary prefix or Ling carry tree (**Prefix K-S**, **Prefix L-F**, **Ling K-S** and **Ling L-F**) need an additional stage after carry computation for decimal post-correction (see Fig. 3.6(a)). Thus, thought the operand setup stage is slightly faster than the corresponding stage in the proposed full binary prefix architectures, the overall latency is about 10% higher (for example, 18.7 FO4 vs. 16.9 FO4 for the proposed 16-digit **Prefix K-S** adder). However, the hardware required to evaluate the signals for conditional speculation (Fig. 3.10(b)) makes the proposed architecture to be at least 10% more complex in area (for example, 2200 NAND2 vs. 2400 NAND2 for the 16-digit **Prefix K-S** adder). The Ling scheme applied to these adders leads from 2% to 3% faster adders but a cost of about 5% increment in area.

For the **Q-T** based schemes, the speculative (Fig. 3.6(b)) and conditional speculative (proposed) architectures present a similar latency, since the BCD sum digits are computed speculatively out of the critical path, in the pre-sum stage. Now, the area is dramatically increased in the speculative **Prefix Q-T** and **Ling Q-T** adders since the hardware for the

decimal correction is required in each one of the two-conditional pre-sum paths. By other hand, in the proposed **Q-T** architecture (Fig. 3.13), the evaluation of the conditional signals for speculation is out of the critical path, so the hardware is further simplified (Fig. 3.13(b)). This results in a reduction in area ranging from 5% to 15% for the proposed **Q-T** adders with respect to the speculative ones. Furthermore, the **Ling Q-T** adders have an additional 3% speedup with respect to the **Prefix Q-T** adders with practically no increment in area. This makes very interesting the use of the Ling scheme for the **Q-T** architectures.

The direct decimal addition method can be efficiently implemented using the architecture of Fig. 3.3. The resulting adder is slightly slower than the speculative and the proposed **Q-T** adders, but presents better area ratio (more than 5%), since the sum cell is optimized for BCD addition.

From our comparison, we conclude that for low latency the **Ling Q-T** based schemes are the best choice. In this case our proposal requires slightly less hardware than the decimal speculative adder and is slightly faster than the direct decimal adder. For low hardware cost, the speculative adder with a **Prefix L-F** topology for carry generation fits the requirement (requires 0.80 times the hardware complexity of the proposed **Prefix Q-T** adder). For the case of trading-off hardware complexity and latency, the proposed or the direct decimal **Ling Q-T** adders are a good alternative. Note that the **K-S** based schemes are not the best choice in any case, although they are close in terms of the estimated hardware complexity and latency. However, it is expected that real implementations with aggressive technologies and circuit techniques result in a significant degradation of its figures of merit due to its high routing complexity and the high amount of logic involved in critical paths.

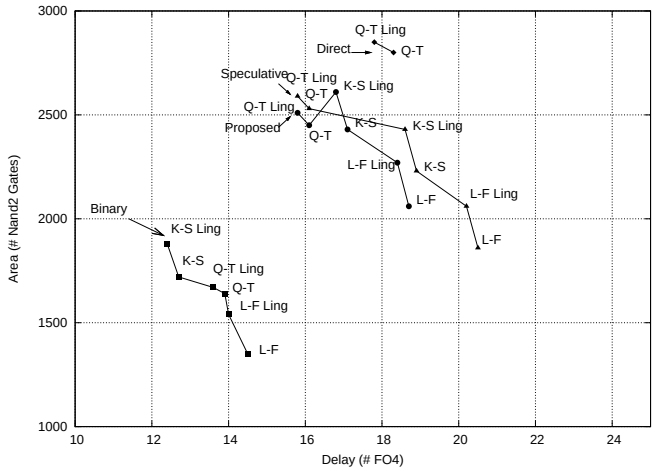
Note that the fastest **Q-T** based decimal schemes are only 1.15 slower than the correspondent **Q-T** binary adder, although the hardware complexity increases by a factor of more than 1.30.

The area and delay estimations for the combined binary/decimal adders are shown in Table 3.6 and in Fig. 3.28(a) for 16 digits and Fig. 3.28(b) for 34 digits. Binary addition/subtraction can be incorporated into BCD speculative and decimal conditional speculative (proposed) adders with no significant increment in latency. In addition, practically all the hardware is shared by binary and BCD operations. Thus, in full binary prefix topologies (**K-S** and **L-F**) the area overhead is not significant (some additional gates or buffering for control signals). For **Q-T** topology the area overhead for combined binary/BCD adders is around 5%.

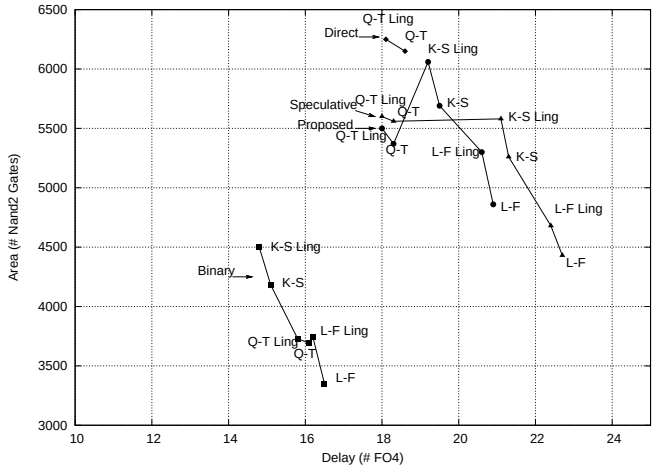
The mixed binary/direct decimal adders (Fig. 3.4) have no apparent advantage in comparison with the speculative and conditional speculative with **Q-T** topology (25% more hardware than our proposals and 10% more latency). The rows which contain the area-delay figures for the direct decimal architectures are signaled with a † symbol.

An additional level of multiplexes is introduced in the the critical path to perform both the direct decimal recurrence and the binary recurrence sharing the most part of the prefix carry tree. Thus, the use of direct decimal based adders is only competitive for decimal specific applications.

For low latency or trading-off area-delay implementations a good choice is the proposed **Ling Q-T** adder, which is only 1.15 slower than the corresponding binary implementation (but requires 40% more area). For low hardware cost a better option is the speculative **Prefix L-F**,



(a) 16-BCD digits.



(b) 34-BCD digits.

Figure 3.28. *Area/delay space of mixed binary/BCD adders.*

which requires about 30% less area than the proposed **Ling Q-T** adder.

In conclusion, incorporating the binary add/sub operation in an existing BCD adder based on the speculative or conditional speculative methods is inexpensive. Thus, it is common that commercial decimal ALUs implement both binary integer and BCD addition or subtraction operations [19, 20].

Finally, we also compare the two alternatives for sum error checking: using replication of the primary adder and a parity checker (Fig. 3.22) against the use of the proposed sum error checker (Fig. 3.23). Table 3.7 shows the area and delay estimations for the two configurations and a primary adder of each type (2's complement, 10's complement, mixed 2's complement/10's complement) as a reference. The area-delay figures of the primary adders correspond with the proposed **Prefix Q-T** adders.

Observe that the replicated adder+parity checker scheme requires about 50% more area than the proposed sum checker. Though the replication scheme is in some cases two times

Prefix tree topology	Binary Adders		Decimal addition method			
			Proposed		Speculative/†Direct	
	Delay	Area	Delay #FO4/ratio	Area Nand2/ratio	Delay #FO4/ratio	Area Nand2/ratio
64-bit/16-BCD digit adders						
Prefix K-S	12.7	1720	**17.1/1.35**	**2430/1.40**	18.9/1.50	2230/1.30
Prefix L-F	14.3	1350	**18.7/1.30**	**2060/1.50**	20.5/1.45	1860/1.40
Prefix Q-T	13.9	1700	**16.1/1.20**	**2450/1.40**	16.1/1.20	2530/1.50
†Prefix Q-T					18.3/1.30	2800/1.65
Ling K-S	12.4	1880	**16.8/1.35**	**2610/1.40**	18.6/1.50	2430/1.30
Ling L-F	14.0	1540	**18.4/1.30**	**2270/1.50**	20.2/1.45	2050/1.30
Ling Q-T	13.6	1740	**15.8/1.15**	**2510/1.45**	15.8/1.15	2590/1.50
†Ling Q-T					17.8/1.30	2850/1.65
136-bit/34-BCD digit adders						
Prefix K-S	15.1	4180	**19.5/1.30**	**5690/1.35**	21.3/1.40	5260/1.25
Prefix L-F	16.5	3350	**20.9/1.25**	**4860/1.45**	22.7/1.40	4430/1.30
Prefix Q-T	16.1	3800	**18.3/1.15**	**5370/1.40**	18.3/1.15	5560/1.45
†Prefix Q-T					20.6/1.30	6150/1.60
Ling K-S	14.8	4500	**19.2/1.30**	**6060/1.35**	21.0/1.40	5580/1.25
Ling L-F	16.2	3740	**20.6/1.25**	**5300/1.40**	22.4/1.40	4680/1.25
Ling Q-T	15.8	3860	**18.0/1.15**	**5500/1.40**	18.1/1.15	5600/1.45
†Ling Q-T					20.1/1.30	6250/1.60

Table 3.6. *Delay and area figures for mixed binary/BCD adders.*

	Adder	Type of sum checker			
		Proposed		Adder+Parity checker	
	Delay/Area (#FO4/Nand2)	Delay (t_{fo4}/ratio)	Area (Nand2/ratio)	Delay (t_{fo4}/ratio)	Area (Nand2/ratio)
16-BCD digit architectures					
Binary	13.9/1640	9.0/0.65	1000/0.60	5.6/0.40	1940/1.20
Decimal	15.9/2220	12.6/0.80	1260/0.55	5.6/0.35	2520/1.15
Mixed	16.1/2450	12.6/0.80	1300/0.55	5.6/0.35	2750/1.10
34-BCD digit architectures					
Binary	16.1/3800	9.6/0.60	2160/0.60	6.2/0.40	4480/1.20
Decimal	18.1/4920	13.2/0.75	2720/0.55	6.2/0.35	5600/1.15
Mixed	18.3/5370	13.2/0.70	2790/0.50	6.2/0.35	6050/1.15

Table 3.7. *Comparison results for sum checkers.*

faster than the proposed sum checker, it would require to increment the cycle time[14] about 35% in case of performing addition and error detection in the same cycle, which is not realistic. Thus, the parity checking should be computed one cycle latter than the sum execution, as in the case of the sum checker. Therefore, there is no advantage in using unit replication and

[14]We assume a cycle time equal to the adder latency.

parity checking instead of the sum checker in real applications.

3.6 Conclusions

In this Chapter we have presented a new high-performance decimal addition algorithm that allows the computation of 10's complement and binary addition and subtraction operations. The proposed algorithm computes the decimal carries using the conventional binary carry propagate recurrence. For this purpose, each BCD digit of one input operand is conditionally incremented +6 units. Unlike other methods based on an initial +6 BCD digit speculation, this condition also simplifies the evaluation of the BCD sum from the resultant binary sum, avoiding the use of decimal post-correction schemes that increments the critical path.

Thus, any 2's complement adder can be used to compute 10's complement additions and subtractions with a little delay overhead due to the initial +6 conditional BCD digit speculations. This leads to very efficient combined 2's complement/10's complement adders. We have implemented the proposed method using representative high-performance prefix tree adders with different area and delay trade-offs. We have applied the Ling scheme to prefix tree adders, resulting in faster adders but only requiring a few modifications in the prefix tree adder.

The rough evaluations performed show that the proposed algorithm leads to adders with interesting area-delay figures. For example, we have shown that a very interesting adder topology for low latency applications with good area and delay trading-off is a Ling quaternary tree adder (Ling Q-T). The proposed Ling Q-T architecture present better area than other representative previous proposals.

Moreover, the proposed algorithm might be of industrial interest since it is very competitive in comparison with commercial and patent-protected ones [14, 19, 20, 25, 63]. Moreover, it opens new alternatives for exploring optimizations with aggressive circuit level techniques.

Finally, we have proposed a unit for sum error detection for 10's complement and mixed 2's complement/10's complement. It extends a previous architecture for binary sum detection. These units can replace the schemes of soft error protection based on parity checking and replication of arithmetic units used in commercial processors, reducing the hardware complexity without a performance penalty.

Chapter 4

Sign-Magnitude BCD Addition

Sign-magnitude arithmetic simplifies some key operations such as multiplication and division. Thus, IEEE DFP formats use a sign-magnitude representation for signed BCD coefficients instead of the 10's complement or 9's complement formats. However, sign-magnitude BCD addition/subtraction is more complex to implement than 10's complement BCD addition, since the result, when negative, requires an additional conversion from 10's complement to a sign-magnitude form.

In this Chapter we present a new sign-magnitude BCD adder, which is based on an extension of the conditional speculative method introduced in Chapter 3. Section 4.1 contains an overview of the basic concepts of sign-magnitude BCD addition. Section 4.2 describes two recent proposals of sign-magnitude BCD adders [136, 157].

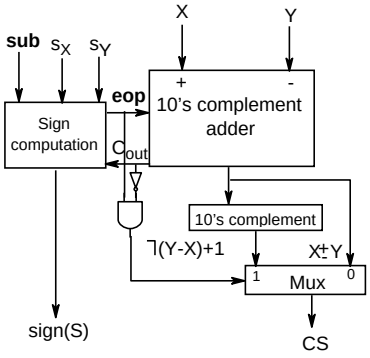
Our method to improve sign-magnitude BCD addition/subtraction is detailed in Section 4.3. In Section 4.4 we propose several architectures of a sign-magnitude BCD adder using different high-performance prefix tree and hybrid carry-select/prefix tree adders [86, 89, 100]. We also present the resulting implementations using the Ling prefix tree topologies [147]. In Section 4.5 we discuss a new unit to detect soft errors in sign-magnitude BCD adders. Section 4.6 presents the area and delay evaluation results and a comparison among different representative proposals. Finally, Section 4.7 contains the conclusions.

4.1 Basic principles

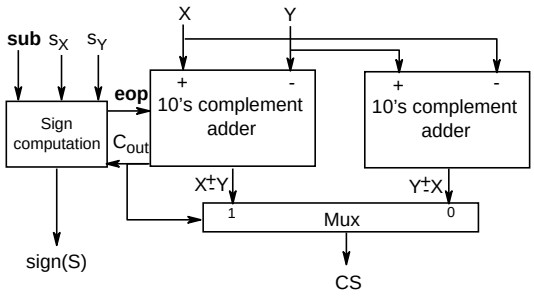
A sign-magnitude BCD adder computes both the magnitude and the sign of two operands $FX = (-1)^{s_X}\ X$ and $FY = (-1)^{s_Y}\ Y$ separately, as

$$FS = X + (-1)^{sub}\ Y = (-1)^{s_X}\left(X + (-1)^{s_Y + sub - s_X}\ Y\right) = (-1)^{sign(S)}\ CS \tag{4.1}$$

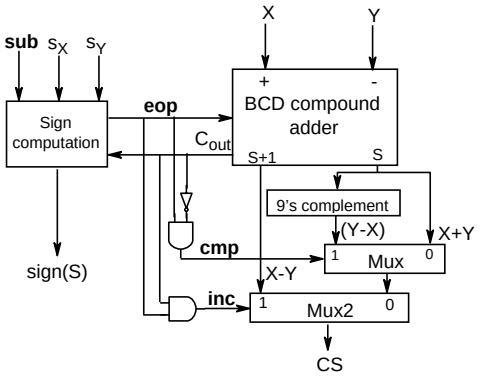
where sub is the operation specified by the instruction ($sub = 0$ for addition $sub = 1$ for subtraction). The magnitude addition CS consists on the computation of $CS = |X + (-1)^{eop}\ Y|$, where eop is the effective operation ($eop = s_X \oplus s_Y \oplus sub$). It is performed as an addition $X + Y$ if $eop = 0$ and as a magnitude subtraction $|X - Y|$ if $eop = 1$. The sign is computed as



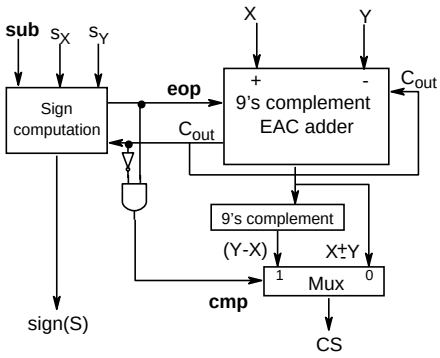
(a) Using a 10's complement BCD adder.



(b) Using a dual 10's complement BCD addition.



(c) Using a BCD compound adder.



(d) Using an 9's complement EAC adder.

Figure 4.1. *Implementation of sign-magnitude BCD addition.*

$sign(S) = s_X\ \overline{eop} \vee sign(X-Y)\ eop$, where $sign(X-Y)$ is defined for effective subtractions as

$$sign(X-Y) = \begin{cases} 0 & \texttt{If}(X \geq Y) \\ 1 & \texttt{Else} \end{cases} \tag{4.2}$$

Fig. 4.1 shows four approaches[15] to compute CS and $sign(X-Y)$ using different configurations of 10's complement and 9's complement adders. Fig. 4.1(a) uses a single 10's complement BCD adder to compute $X \pm Y$. The carry-out of the adder determines $sign(X-Y)$ in case of effective subtraction ($sign(X-Y) = \overline{C_{out}}$). Thus, if $C_{out} == 0$ and $eop = 1$, then $Y > X$, and the output of the adder needs to be 10's complemented to obtain the absolute magnitude result $CS = |X-Y|$. This implies an additional carry propagation from the LSD (least significant digit).

To avoid the additional carry-propagation, Fig. 4.1(b) ("dual adder" approach) uses a pair of 10's complement BCD adders to compute two speculative results $X \pm Y$ and $Y \pm X$. For the left adder we have that $sign(X-Y) = \overline{C_{out}}$, so it can be used to select the appropriate result. For instance, if $C_{out} = 0$ and $eop = 1$, then $X < Y$ and $CS = Y - X$. This configuration leads to low-latency implementations but with high hardware complexity. Due to area constraints, the IBM Power6 DFPU [45] uses a single 10's complement BCD hybrid carry-select/prefix tree

[15] Extrapolated from well-known techniques used to implement binary sign-magnitude adders [142].

adder based on speculative decimal addition (see Section 3.1.3) to compute both $X \pm Y$ and $Y \pm X$ sequentially in case of effective subtractions.

A more efficient alternative for low latency consists on implementing a compound adder that computes both sum and sum+1 simultaneously. Compound adders [15, 116, 167] are widely used in binary floating-point units to implement significand addition and to merge it with rounding [16, 54]. The advantage of these implementations is that a "late increment" or a "late complement" can be incorporated into a 1's complement carry-propagate adder in an additional small constant time [53].

Thus, 1's complement carry-select and conditional adders (and hybrid implementations) can be easily modified to accommodate the sum and sum+1 operations [116, 167]. A more efficient VLSI implementation of low-latency binary compound adder is the flagged prefix adder [15]. It can perform two related sums ($X + Y$ and $X + Y + 1$) or subtractions ($X - Y = X + \overline{Y} + 1$ and $Y - X = \overline{X + \overline{Y}}$) with a slightly increment of hardware complexity and delay respect the corresponding 1's complement adders.

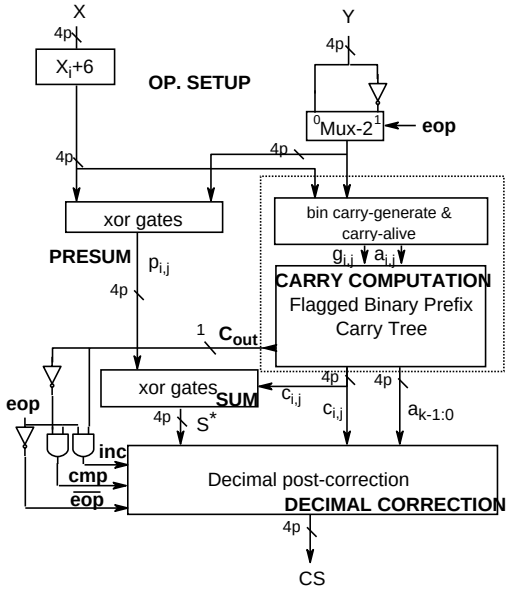
This scheme can be easily extended to implement a BCD compound adder using any 9's complement adder, as shown in Fig. 4.1(c). A 9's complement adder computes $S = X + Y$ (for $eop = 0$) or $S = X + (\neg Y)$ (for $eop = 1$) and the decimal output carry C_{out}. In case of an effective subtraction, we have that $X - Y = X + (\neg Y) + 1 = S + 1$. On the other hand, $Y - X$ is obtained as the 9's complement of S, since $Y - X = -(X - Y) = \neg(X - Y) + 1 = \neg(S + 1) + 1 = \neg S$). Therefore, the magnitude result $CS = |X - Y|$ is implemented selecting $S + 1$ for $C_{out} = 1$ ($X \geq Y$) and $\neg S$ for $C_{out} == 0$ ($Y > X$). The control signal $inc = eop\ C_{out}$ indicates a $S + 1$ operation. For effective addition, $CS = X + Y = S$.

A more aggressive implementation of compound addition is the EAC (End Around Carry) adder described in [120]. An advantage of this EAC adder is that it mitigates the delay due to the high fanout of the selection signal inc by introducing the "end around carry" C_{out} into the carry propagate recurrence. Thus, for effective subtractions, the magnitude $CS = |X - Y|$ is obtained just complementing the EAC sum output when C_{out} is zero ($Y > X$) (the +1 ulp increment when C_{out} is one ($X \geq Y$) is included into the carry propagation). For addition, the carry input must be set to 0. The propagation of the EAC is known to be limited to the operands length [8], so the EAC tree has the same logical depth as the equivalent 2's complement adder carry tree. A general diagram of sign-magnitude BCD addition using a 9's complement EAC adder is shown in Fig. 4.1(d). However, up to date, no implementation of a 9's complement EAC adder has been proposed.

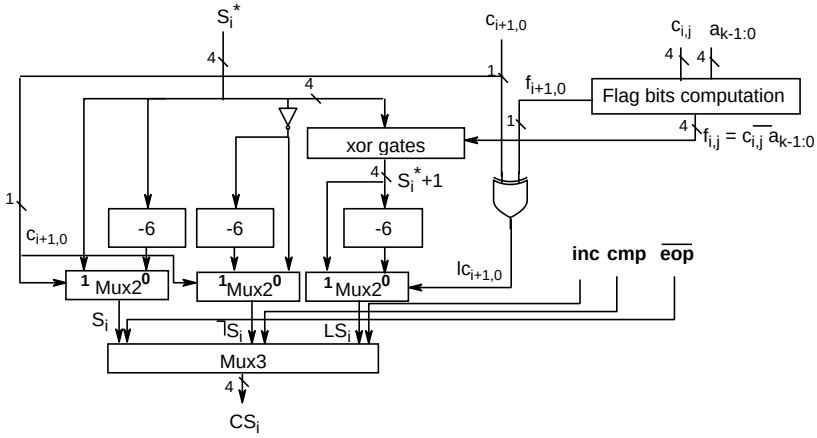
In the next Section we describe two recent proposals [136, 157] that implement a BCD flagged prefix adder based on the decimal speculative addition method (see Section 3.1.3). In Section 4.3 we extend our conditional speculative method to support sign-magnitude BCD addition, while in Section 4.4 we show the resulting prefix tree implementations.

4.2 Sign-magnitude BCD speculative adder

The binary flagged prefix adder [15, 16] has been recently extended by Thompson, Karra and Schulte [136], and by Wang and Schulte [157], to support BCD compound addition. Fig. 4.2 shows the architecture of this 9's complement BCD flagged prefix tree adder. It implements



(a) General Architecture.



(b) Decimal post-correction (4-bit slice).

Figure 4.2. *9's complement prefix tree adder [136, 157] using speculative decimal addition.*

the speculative decimal addition method (Section 3.1.3) in four stages: operand setup, binary carry propagation (implemented in a full binary Kogge-Stone flagged prefix tree), sum and decimal post-correction.

The operand setup stage is similar to that of the 10's complement adder of Fig. 3.6(a). The Y operand is 9's complemented for decimal subtraction. Every BCD digit of one input operand is unconditionally biased by 6 to compute the decimal carries using a binary carry propagation. The BCD sum is estimated from the binary sum of the input biased BCD operands.

A carry propagation due to a +1 ulp (unit in the last place) increment of the sum is determined by the trailing chain of 9's of the BCD sum. The trailing chain of 9's of the BCD sum corresponds to a trailing chain of 15's ('1111') of the binary sum. This trailing chain of

15's is detected using the flagged bits signals f_k ($k = 4i + j$), computed in the binary flagged carry prefix tree along with the binary carries c_k.

A flagged bit f_{k+1} indicates if the possible carry generated due to incrementing the sum one ulp is propagated from lsb (least significant bit) to position $k+1$. By definition, it is equal to the block carry propagate signal $p_{k:-1}$ given by

$$f_k = p_{k-1:-1} = \prod_{q=-1}^{k-1} p_q = p_{k-1} \ldots p_1 \; p_0 \; p_{-1} = \overline{c_k} \; a_{k-1:-1} \tag{4.3}$$

where $c_k = g_{k-1:-1}$, $\prod$ means here the logic AND operation. That is, a carry generated due a +1 ulp increment of sum is propagated to bit k only if $p_{k-1:-1}$ is one. The carry alive group $a_{k:-1}$ can be computed in parallel to the carries c_{k+1} as

$$(c_{k+1}, a_{k:-1}) = (g_{k:0}, a_{k:0}) \bullet (0, 1) = (g_{k:0}, a_{k:0}) = \prod_{q=0}^{k} (g_q, a_q) \tag{4.4}$$

where we assume $g_{-1} = c_0 = c_{in} = 0$ and $a_{-1} = 1$. The difference of this scheme with respect to a conventional prefix tree is that now the terms $a_{k:-1} = a_{k:0} \; a_{-1}$ are not necessarily zero, since $a_{-1} = 1$. These terms can be computed in any prefix tree where the prefix cells of its last level are full logic (the flagged prefix carry tree [15, 16]). The flag bits are computed using an additional level of simple gates as $f_k = \overline{c_k} \; a_{k-1:-1}$.

Next, the binary sum bits are computed as $s^*_{i,j} = p_{i,j} \oplus c_{i,j}$. In the post-correction stage, shown in Fig. 4.2(b), the BCD sum is obtained correcting the binary sum S^* using the flag bits $f_{i,j}$ as follows:

- For effective decimal addition ($eop = 0$), BCD sum digits are obtained subtracting -6 (digitwise) from each 4-bit binary sum S^*_i, if the decimal carry-out C_{i+1} is zero, as described for 10's complement BCD speculative addition (Section 3.1.3).

- For effective subtraction operations ($eop = 1$), decimal digit correction is performed as:

  - If $X \geq Y$, then ($sign(X - Y)$) is positive ($C_{out} = 1$) and the result is given by $LS = S + 1 = X + (\neg Y) + 1$. In this case, the bits of the binary sum $LS^* = S^* + 1$ are computed inverting the bits of S^* when the corresponding flag bit is one, that is, $ls^*_{i,j} = s^*_{i,j} \oplus f_{i,j}$. The BCD digits of $LS = S + 1$ are obtained subtracting 6 (digitwise) from each 4-bit binary sum vector LS^*_i at position i when the corresponding signal $c_{i+1,0} \oplus f_{i+1,0}$ is zero.

  - If $Y > X$, then ($sign(X - Y)$) is negative ($C_{out} = 0$) and the result is given by $\neg(X + (\neg Y))$ (9's complement of sum). The 9's complement of $X + (\neg Y)$ is obtained by bit inverting the binary sum S^* and then subtracting 6 from each 4-bit digit $\overline{S^*_i}$ whose corresponding decimal carry output C_{i+1} is one.

The different cases, i.e., effective decimal addition $X + Y$, effective subtraction for $X \geq Y$ and effective subtraction for $Y > X$, are selected by control signals $\overline{eop}$, $inc = eop \; C_{out}$ and $cmp = eop \; \overline{C_{out}}$ using the final 3 to 1 multiplexer.

4.3 Proposed method for sign-magnitude addition

We have extended the conditional decimal speculative algorithm presented in Section 3.2 to support sign-magnitude BCD addition. An advantage of a BCD adder based on conditional decimal speculation is that the BCD sum digits are obtained from a minor modification of the binary sum. This results in a simpler and faster scheme for sign-magnitude addition/subtraction than the speculative BCD compound adders of Section 4.2.

Fig. 4.3 shows the proposed algorithm for sign-magnitude addition of BCD operands X and Y. We opt for the BCD compound addition scheme of Fig. 4.1(c). A future objective is to extend the conditional speculative method to implement the 9's complement EAC addition scheme of Fig. 4.1(d).

[Algorithm: Conditional Speculative BCD Compound Addition]

Inputs: $X := \sum_{i=0}^{p-1} X_i \cdot 10^i := \sum_{j=0}^{p-1} (X_i^U \ 2 + x_{i,0}) \cdot 10^i$
$Y := \sum_{i=0}^{p-1} Y_i \cdot 10^i := \sum_{j=0}^{p-1} (Y_i^U \ 2 + y_{i,0}) \cdot 10^i$
$C_0 := 0 \ LC_0 := 1$

For (i:=0;i<p;i++){

$$Y_i^* := (Y_i^*)^U \ 2 + y_{i,0} = \begin{cases} \neg Y_i = \overline{Y_i + 6} & \texttt{If(eop == 1)} \\ Y_i & \texttt{Else} \end{cases}$$

1. Conditional speculation:

$$A_i^U = \begin{cases} 1 & \texttt{If } X_i^U + (Y_i^*)^U \geq 8 \\ 0 & \texttt{Else} \end{cases}$$

$$Z_i^U = X_i^U + (Y_i^*)^U + 6 \cdot A_i^U$$

2. Binary carry-propagate computation:

$$C_{i+1} = c_{i+1,0} = \lfloor (Z_i^U + x_{i,0} + y_{i,0}^* + c_{i,0})/16 \rfloor$$
$$LC_{i+1} = lc_{i+1,0} = \lfloor (Z_i^U + x_{i,0} + y_{i,0}^* + lc_{i,0})/16 \rfloor$$

}

3. Decimal digit addition:

$C_{out} := C_p$

For (i:=0;i<p;i++){

$$s_{i,0} = \text{mod}_2(x_{i,0} + y_{i,0}^* + c_{i,0})$$

$$S_i^U = \begin{cases} 8 & \texttt{If}\Big(Z_i^U == 14 \ \texttt{AND} \ c_{i,1} == 0\Big) \\ \text{mod}_{16}(Z_i^U + 2 \cdot c_{i,1}) & \texttt{Else} \end{cases}$$

$$ls_{i,0} = \text{mod}_2(cx_{i,0} + cy_{i,0}^* + lc_{i,0})$$

$$LS_i^U = \begin{cases} 8 & \texttt{If}\Big(Z_i^U == 14 \ \texttt{AND} \ lc_{i,1} == 0\Big) \\ \text{mod}_{16}(\overline{Z_i^U + 2 \cdot lc_{i,1}}) & \texttt{Else} \end{cases}$$

$$CS_i = \begin{cases} \neg S_i = \overline{S_i^U + 6 + s_{i,0}} & \texttt{Else If(eop == 1 AND } C_{out} == 0\texttt{)} \\ LS_i = LS_i^U + ls_{i,0} & \texttt{Else If(eop == 1 AND } C_{out} == 1\texttt{)} \\ S_i = S_i^U + s_{i,0} & \texttt{Else} \end{cases}$$

}

Figure 4.3. *Proposed method for sign-magnitude BCD addition/subtraction.*

The computation is divided in 3 stages: operand setup (includes 9's complement of operand Y and the +6 conditional digit speculation), binary carry propagate computation and BCD digit addition. The operation of the setup stage is similar to that described in Section 3.2 for 10s complement addition: an input digit (either X_i or $Y_i^* = \overline{Y_i + 6}$ $eop \vee Y_i$ $\overline{eop}$) is biased

by 6 when the condition A_i^U is verified. The effective operation is indicated by eop ($eop = 1$ for subtraction).

The BCD compound addition method evaluates both $S = X + Y^*$ and $LS = S + 1 = X + Y^* + 1$. The main difference with respect to the 10's complement algorithm is the need to compute a dual binary carry propagation. In addition to the chain of binary carries c_k of S^*, we evaluate the binary carries of LS^*, or binary late carries lc_k, assuming carry inputs $c_{in} = g_{-1} = 0$ (for the primary carry chain) and $lc_{in} = a_{-1} = 1$ (for late carry chain). Actually, a late carry lc_k is produced due to a carry c_k or a trailing chain of $k-1$ ones in S^*, which indicate that an increment of one ulp in S is propagated up to position k. Hence, the late binary carries lc_k may be computed from the binary carries c_k in an additional single gate delay time using the block binary alive signals $a_{k-1:-1} = a_{k-1:0}\ a_{-1} = a_{k-1} \dots\ a_0$ of S as

$$lc_k = c_k \oplus p_{k-1:-1} = c_k \oplus \overline{c_k}\ a_{k-1:-1} = c_k \vee a_{k-1:0} \tag{4.5}$$

Decimal carries C_i and late decimal carries LC_i correspond to the binary carries $c_{i,0}$ and $lc_{i,0} = c_{i,0} \vee a_{4i-1:-1}$ at hexadecimal positions (1 each 4). Therefore, we can use either a binary or a quaternary flagged prefix tree [15, 16] to evaluate signals $c_{i,0}$ and $a_{4i-1:0}$.

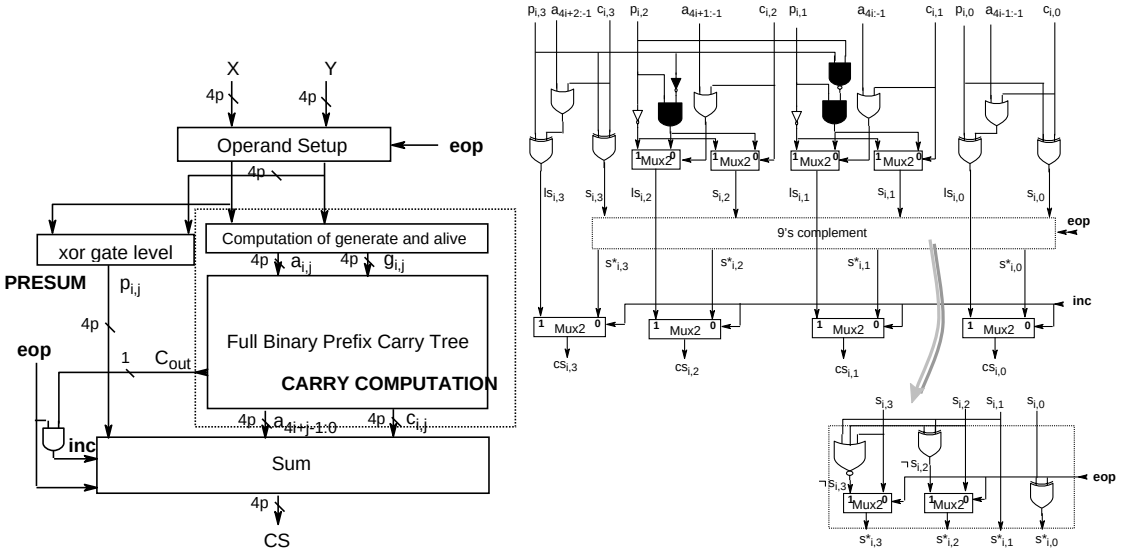
The digits of the BCD magnitude result CS are obtained as follows:

- For effective addition ($eop = 0$), the BCD sum $S = X + Y$ is computed as detailed in Section 3.2 for the 10's complement algorithm. Therefore, $CS_i = S_i$, where the logical expressions for S_i are given by (3.38) for the full binary prefix adders and by equation (3.39) for the hybrid prefix/carry-select architectures.
- In case of effective subtraction $X - Y$ ($eop = 1$), the decimal carry output $C_{out} = C_p = c_{p,0}$ determines the sign of $X - Y$:
  - If $C_{out} = 1$, then $X \geq Y$, and the magnitude result is given by $LS = X + (\neg Y) + 1$. That is, $CS_i = LS_i$, where the logical expressions for the sum digits LS_i are obtained introducing the late carries $lc_{i,j} = c_{i,j} \vee a_{4i+j-1:0}$ in (3.38) or (3.39) instead of the conventional carries $c_{i,j}$.
  - If $C_{out} = 0$, then $X < Y$ and the magnitude result is computed as $\neg S = \neg(X + \neg Y)$. Magnitude digits CS_i are then given by the 9's complement of S_i, that is, $CS_i = 9 - S_i = \overline{S_i + 6}$.

The gate level implementations of these sum cells is detailed in the next Section for the full binary prefix and the hybrid prefix/carry-select topologies. Additionally, we introduce the Ling scheme into the flagged prefix tree adders to speedup the evaluation of the carries.

4.4 Architectures for the sign-magnitude adder

This Section presents several architectures for the proposed sign-magnitude BCD adder. The main component of each architecture is a binary flagged prefix adder modified to support the conditional speculative algorithm of Section 4.3. A key point is that any binary prefix adder can be used by introducing some simple transformations. We show the architectures for the full binary prefix tree, the hybrid prefix/carry-select and the Ling prefix topologies.



(a) Block diagram. (b) BCD magnitude sum cell (4-bit digit slice).

Figure 4.4. *Proposed sign-magnitude BCD prefix tree adder.*

4.4.1 Binary prefix tree architecture

Fig. 4.4(a) shows the block diagram of the proposed sign-magnitude BCD adder for the full binary prefix carry tree topology. The different blocks implement the computational stages of the algorithm of Fig. 4.3. The operation setup, decimal correction and prefix carry tree blocks are similar as those described for the proposed 10's complement BCD prefix adders of Section 3.3. The only difference is that, in a flagged prefix tree, the prefix nodes of the last level implement full logic [16] and the carry input C_{in} is always 0.

Thus, the differences with respect to the 10's complement adder are inside the sum block that computes the BCD magnitude result. A signal inc is used to select the appropriate BCD magnitude digits of the result. It is computed from the effective operation control bit eop and the carry output of the prefix carry tree as $inc = eop\ C_{out}$. Fig. 4.4(b) shows a gate level implementation of this sum block for one digit. The computation of the resultant BCD magnitude digit is separated in two logical paths for the BCD data and one additional path for the selection signal inc:

- The first path ($inc = 0$) computes the BCD sum bits S_i according to the conditional decimal speculative addition method as

$$S_i = \begin{cases} s_{i,3} = & p_{i,3} \oplus c_{i,3} \\ s_{i,2} = & \overline{p_{i,2}}\ c_{i,2} \vee \overline{p_{i,3}}\ p_{i,2}\ \overline{c_{i,2}} \\ s_{i,1} = & \overline{p_{i,1}}\ c_{i,1} \vee \overline{p_{i,3}}\ \overline{p_{i,2}}\ p_{i,1}\ \overline{c_{i,1}} \\ s_{i,0} = & p_{i,0} \oplus c_{i,0} \end{cases} \tag{4.6}$$

  The black gates in Fig. 4.4(b) performs the replacement of values $S_i = \{14, 15\}$ by BCD digits $S_i = \{8, 9\}$ as it was detailed in Section 3.2.

[16] In a conventional carry prefix tree, the final alive groups are not computed.

Moreover, to reduce the hardware complexity, the 9's complement of S_i, given by

$$\overline{S_i + 6} = \begin{cases} \overline{s_{i,3} \vee s_{i,2} \vee s_{i,1}} \\ s_{i,2} \oplus s_{i,1} \\ s_{i,1} \\ \overline{s_{i,0}} \end{cases} \tag{4.7}$$

is computed along with S_i as $S_i^* = S_i \ \overline{eop} \vee \overline{S_i + 6} \ eop$. The boolean equations for this path can be simplified as

$$S_i^* = \begin{cases} s_{i,3}^* = & s_{i,3}\overline{eop} \vee \overline{s_{i,3} \vee s_{i,2} \vee s_{i,1}} \ eop \\ s_{i,2}^* = & s_{i,2}\overline{eop} \vee (s_{i,2} \oplus s_{i,1}) eop \\ s_{i,1}^* = & s_{i,1} \\ s_{i,0}^* = & s_{i,0} \oplus eop \end{cases} \tag{4.8}$$

In this way, $S_i^* = S_i$ is computed for effective addition and $S_i^* \neg S_i = \overline{S_i + 6}$ for effective subtraction.

- The second path ($inc = 1$) implements the operation $LS = S + 1$. The following expression for LS_i was obtained replacing $c_{i,j}$ in (4.6) by $lc_{i,j} = c_{i,j} \vee a_{4i+j-1:-1}$:

$$LS_i = \begin{cases} ls_{i,3} = & p_{i,3} \oplus c_{i,0} \vee a_{4i+2:-1} \\ ls_{i,2} = & \overline{p_{i,2}} \ (c_{i,2} \vee a_{4i+1:-1}) \vee \overline{p_{i,3}} \ p_{i,2} \ \overline{c_{i,2} \vee a_{4i+1:-1}} \\ ls_{i,1} = & \overline{p_{i,1}} \ (c_{i,1} \vee a_{4i:-1}) \vee \overline{p_{i,3} \ p_{i,2}} \ p_{i,1} \ \overline{c_{i,1} \vee a_{4i:-1}} \\ ls_{i,0} = & p_{i,0} \oplus c_{i,0} \vee a_{4i-1:-1} \end{cases} \tag{4.9}$$

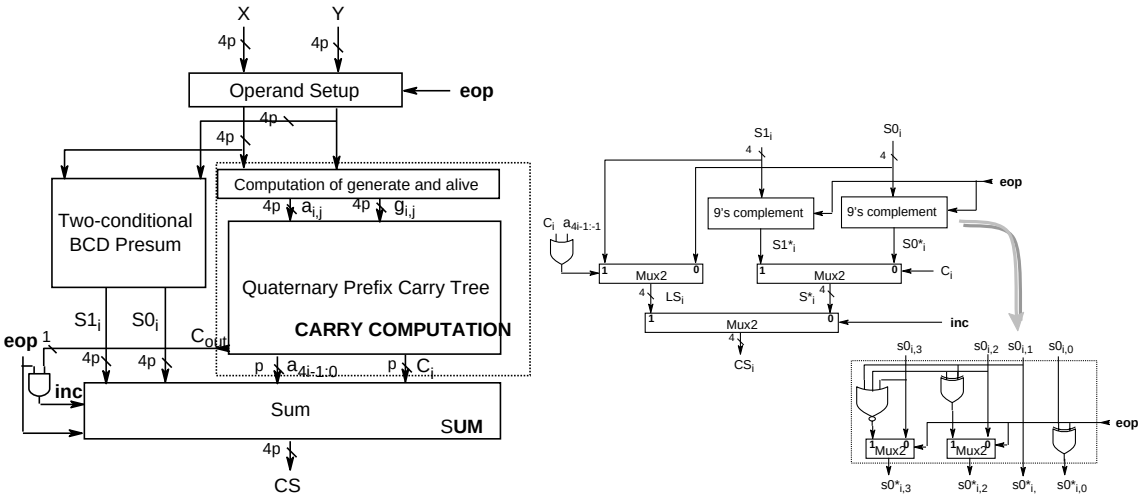
The black gates are also used to replace values $LS_i = \{14, 15\}$ by BCD digits $LS_i = \{8, 9\}$. Both binary carries $c_{i,j}$ and alive groups $a_{4i+j-1:-1}$ are computed in the full binary prefix tree. Note that this path is selected only in case of effective subtraction.

- The signal inc controls the final level of 2:1 multiplexes, selecting between LS_i (path $inc = 1$) or $S_i^* = \neg S_i$ in case of effective subtraction. For effective addition ($eop = 0$), inc is zero and $S_i^* = S_i$ is always selected. Therefore, the BCD magnitude digits of the result are selected according to

$$CS_i = LS_i \ inc \vee S_i^* \ \overline{inc} \tag{4.10}$$

Due to the huge load of the 2:1 multiplexes driven by inc, it is necessary to put a chain of buffers to distribute the signal inc. Thus, the actual latency of the sum block depends not only on the delay of paths S_i and LS_i but also on the delay of signal inc. This configuration optimizes the latency of the whole sum block for large word lengths (16 BCD digits and above) since it balances the delay of the logic in the BCD sum datapath with the delay of signal inc.

With respect to the implementation of Fig. 4.2 (based on speculative decimal addition), our proposal uses a faster and simpler scheme to obtain the BCD magnitude sum from the signals computed in the binary tree adder. This advantage comes not only from a different reorganization of the final logic, but from the fact that the correction of the binary sum does not depend on each decimal carry output.



(a) General diagram. (b) Sum block (4-bit digit slice).

Figure 4.5. *Proposed sign-magnitude BCD quaternary-tree adder.*

4.4.2 Hybrid prefix/carry-select architecture

The diagram of the proposed hybrid quaternary prefix tree/carry-select architecture is shown in Fig.4.5(a). Basically, this sign-magnitude BCD adder differs from the 10's complement adder of Fig. 3.13 in the implementation of the sum stage. The gate level implementations for the other stages corresponds to the blocks presented for the 10's complement adder[17]: the operand setup (Fig. 3.15), the two-conditional presum stage (Fig. 3.14) and the prefix carry tree (Figs. 3.16 and 3.17).

The gate level implementation of the sum block (one digit slice) is shown in Fig. 4.5(b). This sum block performs the 9's complement of presums $S1_i$, $S0_i$ (out of the critical path). Then, the true or complement form is selected depending on the effective operation eop. In this way, the two-conditional digits $S1_i^*$, $S0_i^*$ are computed as

$$\begin{aligned} S1_i^* &= S1_i \ \overline{eop} \vee \overline{S1_i + 6} \ eop \\ S0_i^* &= S0_i \ \overline{eop} \vee \overline{S0_i + 6} \ eop \end{aligned} \tag{4.11}$$

The BCD digits S_i^* and LS_i, which correspond to sum and sum+1 respectively, are selected in parallel from the two-conditional presum pairs ($S1_i^*$, $S0_i^*$) and ($S1_i$, $S0_i$) depending on the decimal carry C_i and group alive signals $a_{4i-1:-1}$ as

$$\begin{aligned} S_i^* &= S1_i^* \ C_i \vee \ \overline{C_i} \\ LS_i &= S1_i \ (C_i \vee a_{4i-1:-1}) \vee S0_i \ \overline{C_i \vee a_{4i-1:-1}} \end{aligned} \tag{4.12}$$

The final level of 2:1 muxes is required to obtain the BCD magnitude digits of the result depending on the signal $inc = eop \ C_{out}$ as

$$CS_i = LS_i \ inc \vee S_i^* \ \overline{inc} \tag{4.13}$$

[17] Except that, for the sign-magnitude adder, the prefix nodes of the last level of the carry tree are full logic.

The additional delay of this BCD sign-magnitude adder with respect to the equivalent hybrid prefix tree/carry-select 10's complement adder (Fig. 3.13) is equal to the maximum of the delay of a 2:1 mux or the propagation delay of the buffered inc signal with a load of $4p$ 2:1 muxes.

4.4.3 Ling prefix tree architectures

The Ling scheme can be incorporated into a BCD sign-magnitude prefix tree adders to speedup the carry propagate evaluation using the same 3 transformations presented in Section 3.3.3, that is:

1. The prefix operations are performed over pairs (g_j, a_{j-1}) instead of (g_j, a_j).
2. The prefix operations at the first level of the tree are simplified.
3. The BCD magnitude digits are obtained as a function of the Ling carries $h_{i,j}$ instead of conventional carries $c_{i,j}$.

The two first transformations are straightforward while the transformation of the BCD magnitude-sum cell requires more detail. We separate the cases for the full binary prefix tree and the hybrid prefix tree/carry-select adders:

- For a **binary prefix carry tree** configuration, the expression for the BCD magnitude digits is stated as

$$cs_{i,j} = ls_{i,j}\ inc \vee s^*_{i,j}\ \overline{inc} \tag{4.14}$$

where signals $s^*_{i,j}$ are defined in (4.8) in terms of signals $s_{i,j}$ and eop. The signal inc is computed in terms of the Ling carry output h_{out} as

$$inc = (eop\ a_{p-1,3})\ h_{out} \tag{4.15}$$

The implementation of the Ling sum block (for one digit) is shown in Fig. 4.6(a). The difference with respect to the prefix sum cell of Fig. 4.4(b) is limited to the evaluation of $ls_{i,j}$ and $s_{i,j}$. The expression of signals $ls_{i,j}$ and $s_{i,j}$ as a function of the Ling carries $h_{i,j}$ is obtained introducing the following expressions

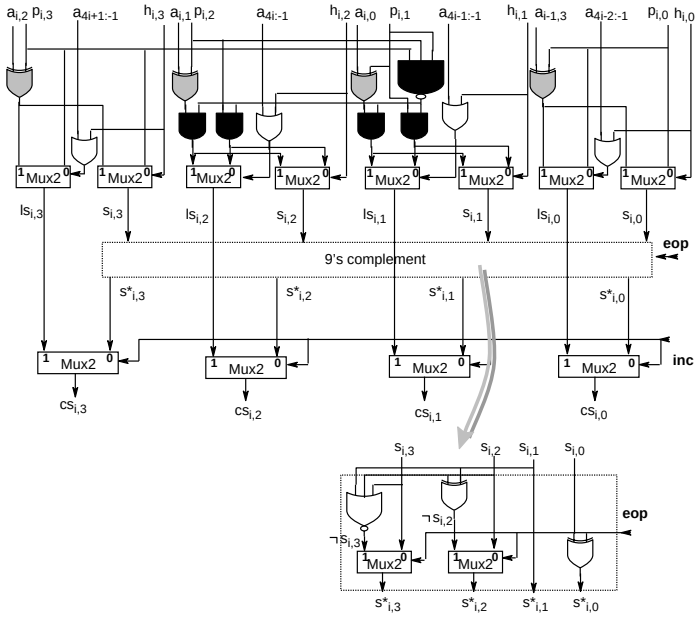
$$\begin{aligned} c_{i,j} &= a_{k-1}\ h_{i,j} \\ lc_{i,j} &= c_{i,j} \vee a_{k-1:-1} = a_{k-1}(h_{i,j} \vee a_{k-2:-1}) \end{aligned} \tag{4.16}$$

with $k = 4i + j$ in (4.6) and (4.9), that is

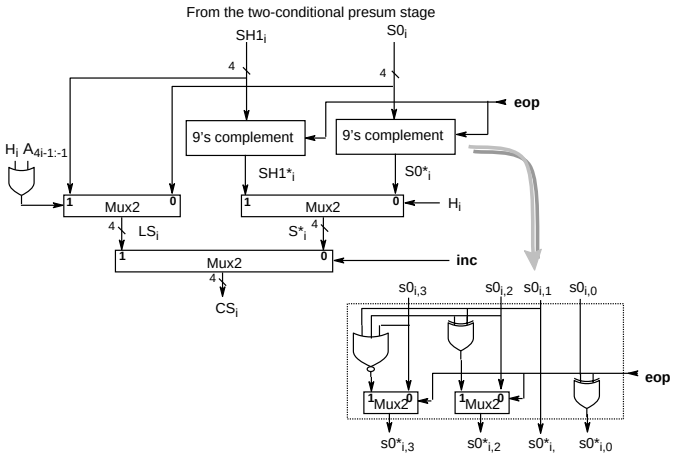
$$S_i = \begin{cases} s_{i,3} = & p_{i,3}\overline{h_{i,3}} \vee (p_{i,3} \oplus a_{i,2})h_{i,3} \\ s_{i,2} = & \overline{p_{i,3}\ p_{i,2}\ p_{i,1}}\ (p_{i,2}\overline{h_{i,2}} \vee \overline{p_{i,3}}\ (p_{i,2} \oplus a_{i,1})\ h_{i,2}) \\ s_{i,1} = & \overline{p_{i,3}\ p_{i,2}}\ (p_{i,1}\overline{h_{i,1}} \vee\ (p_{i,1} \oplus a_{i,0})\ h_{i,1}) \\ s_{i,0} = & p_{i,0}\overline{h_{i,0}} \vee (p_{i,0} \oplus a_{i-1,3})h_{i,0} \end{cases} \tag{4.17}$$

and

$$LS_i = \begin{cases} ls_{i,3} = & p_{i,3}\overline{h_{i,3} \vee a_{4i+1:-1}} \vee (p_{i,3} \oplus a_{i,2})(h_{i,3} \vee a_{4i+1:-1}) \\ ls_{i,2} = & \overline{p_{i,3}\ p_{i,2}\ p_{i,1}}\ (p_{i,2}\overline{h_{i,2} \vee a_{4i:-1}} \vee \overline{p_{i,3}}\ (p_{i,2} \oplus a_{i,1})\ (h_{i,2} \vee a_{4i:-1})) \\ ls_{i,1} = & \overline{p_{i,3}\ p_{i,2}}\ (p_{i,1}\overline{h_{i,1} \vee a_{4i-1:-1}} \vee\ (p_{i,1} \oplus a_{i,0})\ (h_{i,1} \vee a_{4i-1:-1})) \\ ls_{i,0} = & p_{i,0}\overline{h_{i,0} \vee a_{4i-2:-1}} \vee (p_{i,0} \oplus a_{i-1,3})(h_{i,0} \vee a_{4i-2:-1}) \end{cases} \tag{4.18}$$



(a) Full binary tree adder.



(b) Quaternary tree adder.

Figure 4.6. *Digit sum block of BCD sign-magnitude Ling prefix adders.*

The black gates detect the wrong values $\{14, 15\}$ for both S_i and LS_i replacing them by the correct BCD result digits $\{8, 9\}$. The grey striped gates represent the additional hardware with respect to the prefix sum cell of Fig.4.4(b). Observe that the critical path (carry paths or inc path) of Figs. 4.4(b) and 4.6(a) is similar. Thus, the computation of the BCD magnitude result CS is faster using the Ling carries $h_{i,j}$ and the group alive $a_{k-2:-1}$ instead of $c_{i,j}$ and $a_{k-1:-1}$, as it is stated by equations (4.17) and (4.18).

- For the **quaternary prefix carry tree** configuration, the BCD magnitude digits of the result, CS_i, are computed using the sum cell of Fig. 4.6(b) and a row of the two-conditional 4-bit BCD adders of Fig. 3.21(a) in three steps: First, computation of two-conditional presums $SH1_i$ (with carry input $a_{i-1,3}$) and $S0_i$ (with carry input 0), using the two-conditional

4-bit BCD adder of Fig. 3.21(a), where

$$SH1_i = S1_i \; a_{i-1,3} \vee S0_i \overline{a_{i-1,3}} \tag{4.19}$$

Next, the sum digits S_i^* and LS_i are computed from the presums $SH1_i$ and $S0_i$, the decimal Ling carries $H_i = h_{i,0}$ and the group alive signals $A_{i-1:-1} = a_{4i-1:-1}$ as

$$\begin{aligned} S_i &= SH1_i \; H_i \vee S0_i \; \overline{H_i} \\ S_i^* &= \overline{S_i + 6} \; eop \vee \overline{S_i} \; \overline{eop} \\ LS_i &= SH1_i \; (H_i \vee A_{i-1:-1}) \vee S0_i \overline{H_i \vee A_{i-1:-1}} \end{aligned} \tag{4.20}$$

and finally, the CS_i digits are evaluated from LS_i and S_i^* as

$$CS_i = LS_i \; inc \vee S_i^* \; \overline{inc} \tag{4.21}$$

where $inc = (eop \; a_{p-1,3}) \; h_{out}$.

Therefore, this Ling adder is of similar complexity as the quaternary prefix tree BCD adder of Fig. 4.5 while a gain in latency is obtained due to a slightly faster computation of Ling carries H_i with respect to the conventional carries C_i.

4.5 Sum error detection

We propose a single sum checker to protect all the sum paths of the sign-magnitude BCD adder. This sum checker is based on the method for error detection in a 10's complement BCD addition/subtraction presented in Chapter 3. Depending on the effective operation eop and on $inc = eop \; C_{out}$, we have to check these three sum paths:

$$CS = \begin{cases} X + Y & \texttt{If}(eop == 0) \\ \neg(X + \neg Y) & \texttt{Else If}(eop == 1 \texttt{ AND } inc == 0) \\ X + \neg Y + 1 & \texttt{Else} \end{cases} \tag{4.22}$$

For the first path ($eop = 0$), the condition to check for each digit is (see Section 3.4.2 for more detail):

$$X_i + Y_i + \overline{CS_i} + C_i = 10 \cdot C_{i+1} + 15 \tag{4.23}$$

where the bit vector of 15_{10} is $(1111)_2$ and the C_i's corresponds with the decimal carries of $X + Y$.

For the second path ($eop = 1$, $C_{out} = 0$), the sum computed is given by $CS = \neg(X + \neg Y)$, so the condition to detect an error is obtained as

$$CS - \neg(X + \neg Y) = 0 \Rightarrow CS + (X + \neg Y) = -1 \tag{4.24}$$

Replacing in (4.24) the values $\neg Y$ and -1 by

$$\begin{aligned} \neg Y &= \overline{Y} - \sum_{i=0}^{p-1} 6 \; 10^i \\ -1 &= -10^p + \sum_{i=0}^{p-1} 9 \; 10^i \end{aligned} \tag{4.25}$$

we get the following condition for CS

$$CS + X + \overline{Y} = -10^p + \sum_{i=0}^{p-1} 15\ 10^i \tag{4.26}$$

or equivalently,

$$CS_i + X_i + \overline{Y_i} + C_i = 10 \cdot C_{i+1} + 15 \tag{4.27}$$

for each digit, where the C_i's are the decimal carries of $X + \neg Y$.

For the third path ($eop = 1$, $C_{out} = 1$), the sum is computed as $CS = X + \neg Y + 1$, and the condition for correct result is stated as

$$X + \neg Y + \neg CS + 2 = 0 \Rightarrow X + \neg Y + \overline{CS} = -10^p + \sum_{i=1}^{p-1} 15\ 10^i + 14 \tag{4.28}$$

To obtain the condition for each digit, we need to know the carry input to each digit position. The decimal carries of $X + \neg Y + \overline{CS}$ coincide with the decimal carries of $CS = X + \neg Y + 1$, that is, the late decimal carries $LC_i = C_i \vee a_{4i-1:-1}$, where $a_{4i-1:-1} = \prod_{k=-1}^{4i-1} a_k$ are the group alive signals at decimal positions. Therefore, we have the following condition for CS_i:

$$X_i + \neg Y_i + \overline{CS_i} + LC_i = 10 \cdot LC_{i+1} + 15 \tag{4.29}$$

Conditions (4.23), (4.27) and (4.29) for the BCD magnitude digit CS_i can be summarized in a single expression as

$$X_i + Y_i^* + CS_i^* + (C_i \vee a_{4i-1:-1}\ inc) = 10 \cdot (C_{i+1} \vee a_{4i+3:-1}\ inc) + 15 \tag{4.30}$$

for $0 < i < p$ and

$$X_0 + CY_0^* + CS_0^* = 10 \cdot (C_1 \vee a_{3:-1}\ inc) + 14 + \overline{inc} \tag{4.31}$$

for $i = 0$, where

$$Y_i^* = \begin{cases} Y_i & \texttt{If}(eop == 0) \\ \neg Y_i = \overline{Y_i + 6} & \texttt{Else If}(eop == 1\ \texttt{AND}\ inc == 1) \\ \overline{Y_i} & \texttt{Else} \end{cases}$$
$$CS_i^* = \overline{CS_i}\ (\overline{eop} \vee inc) \vee CS_i\ \overline{\overline{eop} \vee inc} \tag{4.32}$$

The resulting architecture is shown in Fig. 4.7. As for the previous designs, the sum checker uses a binary parity checker to compare the outputs of a modified binary 3:2 CSA with inputs CX, CY^*, CS^*. Two control signals, the effective operation eop and the increment condition $inc = eop\ C_{out}$ are used as selection signals for the different paths. Each decimal carry-out $LC_{i+1} = C_{i+1} \vee a_{4(i+1)-1:-1}\ inc$ indicates whether it is necessary to modify the carry digit CV_i to obtain the corrected carry HV_i, as stated by

$$\begin{aligned} hv_{i,3} &= LC_{i+1} \\ hv_{i,2} &= cv_3\ (cv_{i,2} \vee cv_{i,1} \vee cv_{i,0})\ LC_{i+1} \vee cv_{i,2}\ \overline{LC_{i+1}} \\ hv_{i,2} &= \overline{cv_{i,1} \oplus cv_{i,0}}\ LC_{i+1} \vee cv_{i,1}\ \overline{LC_{i+1}} \\ hv_{i,0} &= \overline{cv_{i,0}}\ LC_{i+1} \vee cv_{i,0}\ \overline{LC_{i+1}} \end{aligned} \tag{4.33}$$

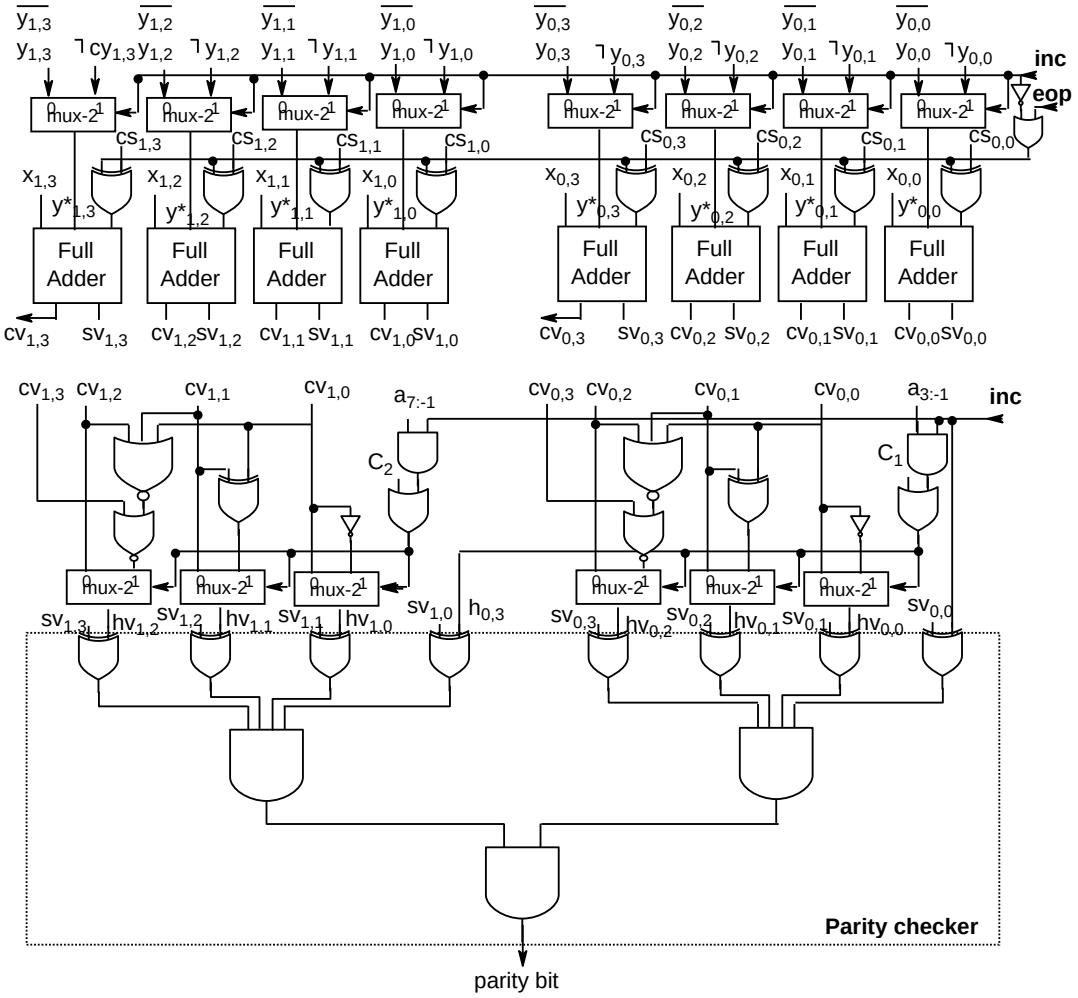


Figure 4.7. *Proposed checker for BCD sign-magnitude addition/subtraction errors (2 digits).*

4.6 Evaluation results and comparison

We have use our evaluation model for CMOS circuits (Appendix A) to estimate the area and delay figures of the different architectures discussed in this Chapter. First, we show the evaluation results of the proposed sign-magnitude BCD adder for different prefix tree topologies: Kogge-Stone (**Prefix K-S**) [86], Ladner-Fischer (**Prefix L-F**) [89], hybrid carry-select/quaternary-tree (**Prefix Q-T**) [100], and the corresponding Ling prefix tree adders (**Ling K-S**, **Ling L-F**, **Ling Q-T**). The word lengths of the BCD coefficients considered are 16 and 34 digits (64-bit and 134-bit respectively) with a 1-bit sign. We have also evaluated the area and delay of the sum error check architecture of Section 4.5. We finally present a comparative study with the different proposals of sign-magnitude BCD adders [136, 157].

Prefix tree topology	Total		Stages			
	Delay (# FO4)	Area (Nand2)	Setup* Delay/Area	Pre-sum Delay/Area	Carry* Delay/Area	Sum* Delay/Area
16-BCD digit adders						
Prefix K-S	**20.2**	**3300**	6.3/880	2.0/240	7.6/1060	6.3/1120
Prefix L-F	**21.8**	**2930**	6.3/880	2.0/240	9.2/690	6.3/1120
Prefix Q-T	**19.2**	**3200**	4.0/760	5.6/820	8.4/430	6.8/1070
Ling K-S	**19.9**	**3490**	6.3/880	2.0/240	7.3/1000	6.3/1370
Ling L-F	**21.5**	**3150**	6.3/880	2.0/240	8.9/660	6.3/1370
Ling Q-T	**18.9**	**3240**	4.0/760	5.6/930	8.1/400	6.8/1070
34-BCD digit adders						
Prefix K-S	**23.3**	**7530**	6.3/1880	2.0/510	10.0/2780	7.0/2360
Prefix L-F	**24.7**	**6700**	6.3/1880	2.0/510	11.4/1950	7.0/2360
Prefix Q-T	**22.1**	**7010**	4.0/1610	5.6/1760	10.6/1050	7.5/2460
Ling K-S	**23.0**	**7960**	6.3/1880	2.0/510	9.7/2650	7.0/2920
Ling L-F	**24.4**	**7140**	6.3/1880	2.0/510	11.1/1830	7.0/2920
Ling Q-T	**21.8**	**7040**	4.0/1610	5.6/1950	10.3/1020	7.5/2460

* Stages in the critical path.

Table 4.1. *Delay and area figures for sign-magnitude BCD adders.*

4.6.1 Evaluation results

Table 4.1 presents the estimated area and delay figures for the sign-magnitude BCD architectures of Section 4.4. The area (in NAND2) and delay (in # FO4) figures included in the first two columns are for the whole architectures. The remaining columns include the area and delay for each stage: operand setup, pre-sum, carry computation and sum.

The architecture for the full binary based trees (**Prefix K-S**, **Prefix L-F**, **Ling K-S**, **Ling L-F**) is shown in Fig. 4.4(a), while the blocks used in the sum stage are outlined in Fig. 4.4(b) for the conventional prefix scheme, and in Fig. 4.6(a) for the Ling scheme. The architecture diagram for the quaternary tree adders is shown in Fig. 4.5(a). Fig. 4.5(b) and Fig. 4.6(b) detail the sum blocks for the **Prefix Q-T** and the **Ling Q-T** sign-magnitude BCD adders.

Like as in the 10's complement and mixed binary/BCD architectures, the quaternary tree topologies are faster and present better area-delay trade-offs. The area and delay overheads with respect to the 10's complement architectures come from the sum stage. The critical path delay in the sum stage is increased because of the huge chain of buffers inserted to distribute the signal $inc = eop\ C_{out}$ to the final level of 2:1 muxes (3.3 FO4 delay for 64 bits and 4 FO4 delay for 136 bits). The increase in area for each adder topology is around 30%.

Table 4.2 shows the estimated area (in NAND2 units) and delay (in # FO4) figures for the sum checker of Fig. 4.7). We also provide the delay/area ratio figures with respect to the proposed 16-digit and 34-digit sign-magnitude BCD **Ling Q-T** adders. The delay/area figures for the reference adder are included the second and fourth data rows of Table 4.2. For comparison, in the third and sixth data rows we have included the delay/area figures for the error detection scheme that uses unit replication and parity checking (Fig. 3.22).

Architecture	Delay (#FO4/ratio)	Area (Nand2/ratio)
16-BCD digit architectures		
Proposed checker	14.6/0.75	1730/0.55
Sign-magnitude BCD adder	19.2/1.00	3200/1.00
2^{nd} Adder + Parity checker	5.6/0.30	3500/1.10
34-BCD digit architectures		
Proposed checker	15.2/0.70	3710/0.55
Sign-magnitude BCD adder	22.1/1.00	7010/1.00
2^{nd} Adder + Parity checker	6.2/0.30	7690/1.10

Table 4.2. *Delay-area figures for sign-magnitude BCD sum checkers.*

The proposed sum checker requires half the area of the replication scheme. It has from 0.75 to 0.70 times the latency of the sign-magnitude BCD adder. Thus, each sum error detection can be completed in one cycle without modifying the add/sub instruction performance, taking advantage of the time gap between the end of the sum execution and the instruction completion. Moreover, although the replication scheme is more than 2.5 times faster than the proposed sum checker, placing it with sum execution, it would require to increment the cycle time about 30%.

4.6.2 Comparison

Table 4.3 presents the area and delay figures of the comparative study among the sign-magnitude BCD adders analyzed in this Chapter. In addition, these results are represented graphically in Fig. 4.8(a), for 16 digits, and in Fig. 4.8(b), for 34 digits.

We include in the comparison the BCD adders based on the proposed method (Section 4.3) and those based on the speculative method [136, 157] (Fig. 4.2). These adders use a full binary flagged prefix K-S tree [15]. We also provide the area-delay curve of the sign-magnitude binary adders as a reference. The ratio values of Table 4.3 were obtained as a quotient of the delay or area of the BCD adder over the corresponding magnitude of the binary adder with an equivalent topology.

The BCD adder proposed in [157] also incorporates hardware for decimal floating-point addition and rounding, thought this was removed from the evaluation. To complete the comparison we have extended the original K-S architecture proposed in [136], obtaining area and delay estimations for the other topologies: prefix L-F and Q-T and Ling L-F and Q-T. However, we have not included in the comparison the direct decimal method, since there is no known implementation of a sign-magnitude BCD adder based on that method.

Basically, the sign-magnitude BCD architectures differ from the 10's complement architectures in the sum stage, which computes the BCD magnitude result depending on the effective operation, the decimal carry output and the signals computed in the flagged prefix tree. The main conclusion we extract from Table 4.3 is that the sum stage proposed in [136, 157] (Fig. 4.2(b)) is significantly more complex than our proposed sum stage (Fig.

Prefix tree topology	Binary Adders		Decimal addition method			
			Proposed		Speculative	
	Delay #FO4	Area Nand2	Delay #FO4/ratio	Area Nand2/ratio	Delay #FO4/ratio	Area Nand2/ratio
64-bit/16-BCD digit adders						
Prefix K-S	16.0	2350	**20.2/1.25**	**3300/1.40**	22.3/1.40	3750/1.60
Prefix L-F	17.6	1980	**21.8/1.25**	**2930/1.50**	23.9/1.40	3250/1.65
Prefix Q-T	17.2	2330	**19.2/1.10**	**3200/1.35**	19.2/1.10	3850/1.65
Ling K-S	15.7	2510	**19.9/1.25**	**3490/1.40**	22.0/1.40	3950/1.55
Ling L-F	17.3	2170	**21.5/1.25**	**3150/1.45**	23.6/1.35	3450/1.60
Ling Q-T	16.9	2370	**18.9/1.10**	**3240/1.35**	18.9/1.10	3900/1.65
136-bit/34-BCD digit adders						
Prefix K-S	19.1	5520	**23.3/1.20**	**7530/1.35**	24.7/1.25	8150/1.50
Prefix L-F	20.5	4690	**24.7/1.20**	**6700/1.40**	26.1/1.25	7300/1.55
Prefix Q-T	20.1	5140	**22.1/1.10**	**7010/1.35**	22.1/1.10	8250/1.60
Ling K-S	18.8	5840	**23.0/1.20**	**7960/1.35**	24.4/1.30	8520/1.45
Ling L-F	20.2	5080	**24.4/1.20**	**7140/1.35**	25.8/1.30	7750/1.55
Ling Q-T	19.8	5200	**21.8/1.10**	**7040/1.35**	21.8/1.10	8350/1.60

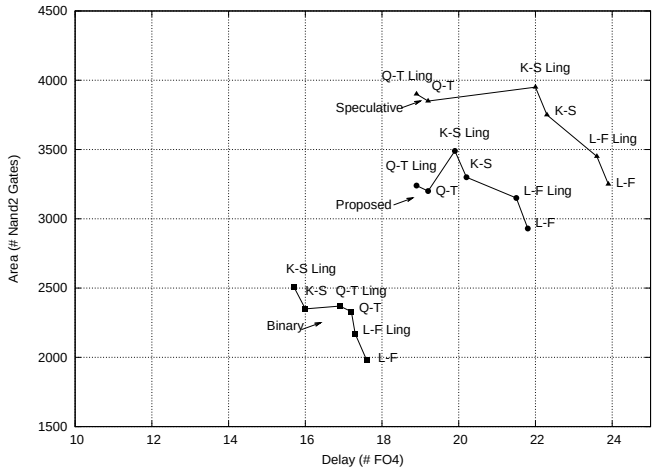
Table 4.3. *Delay and area figures for sign-magnitude BCD adders.*

4.4(b)). For this reason, the full binary prefix tree implementations (**K-S** and **L-F**) based on the speculative method require at least 15% more area and are between 15% (16-digit) and 5% (34-digit) slower than our proposals for sign-magnitude BCD addition.

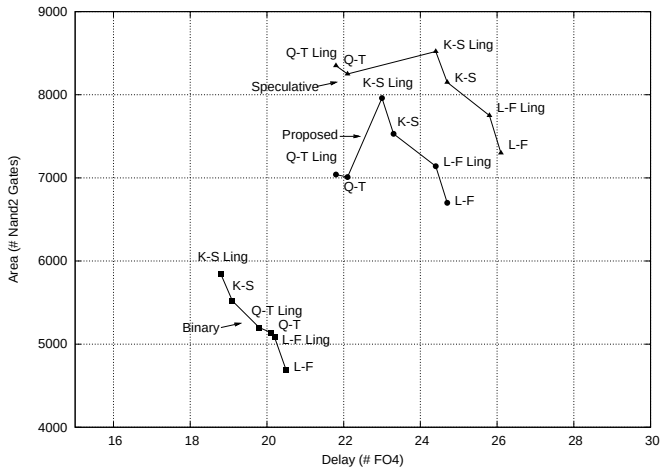
In the case of **Q-T** architectures both proposals have the same latency, but our implementations require between 25% and 30% less area depending on the number of digits and if they use the Ling scheme. Therefore, for low latency and trading-off area-delay applications, the preferred sign-magnitude BCD adder is the proposed **Ling Q-T** implementation, while for low hardware cost a better choice is the proposed **Prefix L-F** implementation (it uses 20% less area).

With respect to the binary sign-magnitude adders, the proposed **Ling Q-T** is only 10% slower but cost 35% more area than the corresponding binary **Ling Q-T** implementation. With respect to the proposed 10's complement BCD **Ling Q-T** adders (for both 16-digit and 34-digit), the proposed 16-digit and 34-digit sign-magnitude BCD **Ling Q-T** adders are 20% slower and require 40% more area.

Thus, due to their complexity in terms of area and latency, the use of sign-magnitude BCD adders is limited to certain applications. One of the most important uses is in a floating-point unit for magnitude addition/subtraction of integer coefficients. To get an efficient implementation of a decimal floating-point adder, the decimal rounding of the magnitude BCD result should be performed in additional little constant time. As we show in the next Chapter, decimal rounding may be incorporated to the proposed sign-magnitude BCD adders with a slightly penalty in latency.



(a) 16-BCD digits.



(b) 34-BCD digits.

Figure 4.8. *Area/delay space of BCD adders.*

4.7 Conclusions

We have presented a new method and a high-performance architecture to compute sign-magnitude BCD addition and subtraction. This operation is used in many applications, such as in decimal floating-point addition to add/subtract the BCD coefficients. The proposed algorithm is an extension of the method presented in Chapter 3 for 10's complement BCD addition. These methods use a conditional +6 digit increment of input BCD operands to allow the computation of the BCD addition as a conventional binary carry-propagate addition. In this way, sign-magnitude BCD addition/subtraction can be implemented using any sign-magnitude binary adder.

In case of subtraction, the carry-out of the adder determines the sign of the result, so the sum result must be complemented or incremented +1 ulp to obtain the correct BCD magnitude result. The previous most representative proposals [136, 157], require a complex post-correction of the binary sum to obtain the BCD magnitude result. In addition to the

carry-out, this correction also depends on the decimal carries and a set of flag bits computed in the carry tree. Our sign-magnitude BCD adder obtains the BCD magnitude result from the binary sum bits independently of the decimal carries or the flag bits, which results in a simpler and faster scheme.

We provide different high-performance implementations of the proposed architecture using several prefix tree and Ling adders. Using an area-delay model for CMOS gates, we have evaluated the different sign-magnitude BCD adders. Independently of the adder topology, the proposed architecture presents better area and delay than the other representative proposals. For low latency applications, a very interesting design with good area and delay trading-off is the Ling quaternary-tree adder (**Ling Q-T**).

Finally, we have proposed a unit to detect error in sign-magnitude BCD additions and subtractions. This unit presents roughly half the area of other solutions used in commercial microprocessors (such as unit replication), while it does not affect to the performance of the processor.

Chapter 5

Decimal Floating-Point Addition

In this Chapter we present a IEEE 754-2008 compliant high-performance decimal floating-point (DFP) adder. Specifically, we concentrate on significand (or sign-magnitude) BCD addition and decimal rounding due to their significative contribution to the total delay of DFP addition [136]. Furthermore, we improve the latency and area of two previous high-performance implementations:

- Thompson, Karra and Schulte [136] perform decimal rounding after significand BCD addition, which requires an additional word length carry propagation. The significand BCD addition consists of a pre-correction of the BCD input operands, a binary compound addition and a decimal post-correction.
- Wang and Schulte [157] combine part of the BCD addition and rounding by overlapping the evaluation of some signals required for rounding and the decimal post-correction. Besides other components, their implementation uses two parallel trees of gates (delay proportional to the logarithm of the operands size) to obtain a set of flag bits for rounding.

The key component of our proposal is a new BCD compound adder that performs sign-magnitude BCD addition and *S+1* and *S+2* operations. This allows to merge IEEE 754-2008 decimal rounding with significand BCD addition in little constant time delay. We incorporate IEEE 754-2008 decimal rounding into the enhanced sign-magnitude BCD adder using two schemes: one based on a direct implementation of the IEEE 754-2008 rounding modes and an injection based rounding technique adapted from binary [54]. We have performed an area and delay analysis of our designs and a comparison with current proposals using our area-delay model for static CMOS logic (see Appendix A). We show that the resultant DFP adder has less latency and reduces significatively the area of the significand computation and rounding with respect to two previous representative proposals [136, 157]. Moreover, a Decimal64 implementation (16 precision digits) presents a moderate fixed overhead (roughly 5.7 FO4) with respect to a high-performance IEEE 754-1985 double precision binary adder with rounding [125].

The Chapter is organized as follows. Section 5.1 contains some important issues about IEEE 754-2008 DFP addition and describes three recent implementations of IEEE 754-2008 compliant DFP adders [45, 136, 157]. It also outlines the most representative methods to implement significand BCD addition and decimal rounding. In Section 5.2 we introduce a

method which allows to incorporate decimal rounding into significand BCD addition in little constant delay time. In Section 5.3 we present the proposed architecture of the sign-magnitude BCD adder with decimal rounding using two different rounding schemes. Evaluation results are shown in Section 5.4. We also provide a comparison among the different schemes for significand addition and rounding implemented in the DFP adders analyzed. Finally, the conclusions are summarized in Section 5.5.

5.1 Previous work on DFP addition

This Section provides an overview of IEEE 754-2008 decimal floating-point addition and describes some proposals of IEEE 754-2008 compliant decimal adders [45, 136, 157]. Next, we describe the different implementations of significand BCD addition and decimal rounding.

5.1.1 IEEE 754-2008 compliant DFP adders

Decimal arithmetic operations are computed as if they first produced a result correct to infinite precision and then rounded to the destination format precision according to the rounding mode. The standard defines a preferred exponent to select a representation of an operation result among all the members of its cohort (see Chapter 2). For decimal addition and subtraction ($FR = FX \pm FY$), the preferred exponent E_R is the least possible for inexact results (to allow for minimum loss of precision) and $min(E_X, E_Y)$ for exact results.

A DFP addition/subtraction is carried out as follows. Operands FX and FY, stored in DPD (densely packed decimal) format, are unpacked as

$$\begin{aligned} FX &= (-1)^{s_X} \cdot X \cdot 10^{E_X} \\ FY &= (-1)^{s_Y} \cdot Y \cdot 10^{E_Y} \end{aligned} \tag{5.1}$$

where s_X,s_Y are the sign bits, E_X, E_Y are the biased binary integer exponents and X, Y are the p-digit BCD coefficients (or significands).

After unpacking the DPD operands, the exponents are compared and one of these cases occurs [45]:

1. **The exponents are equal.** The BCD coefficients X and Y are added and the carry out from the adder is examined. If there is a carry out, the exponent is incremented and the resultant coefficient is shifted one digit right and rounded (inexact result).
2. **The exponent difference $E_D = |E_X - E_Y|$ is not zero**. The operands are swapped if $E_X < E_Y$ (requires computation of sign($E_X - E_Y$)) and the number of leading zeroes (lz_U) in the operand with the larger exponent is examined. Two cases can occur:
   (a) $E_D \leq lz_U$**.** The coefficient of the operand with the larger exponent is shifted left E_D digits and then added to the other unshifted coefficient. The exponent of the result (preferred exponent) is the smallest exponent of the two operands. Similar to the previous case, the coefficient result is shifted one digit right if there is a carry out from the adder. In this case, a rounding is performed and the preferred exponent incremented (inexact result).

(b) $E_D > lz_U$. Both operands are shifted. The operand with the larger exponent is shifted left lz_U digits and the operand with the smaller exponent $E_D - lz_U$ digits right. Both shifts can be replaced by single shift right of E_D digits if the adder is 2p digits wide. The aligned coefficients are then added and rounded. No normalization is necessary. At most, a shift by one digit right or left, depending on the effective operation (subtraction requires a guard digit) and rounding.

Design efforts in current IEEE 754-2008 DFP commercial adders are focused on providing support for the widest format in a reduced area. Therefore, performance is sacrificed by the reuse of hardware in different parts of the DFP computation. For instance, the adder implemented in the DFP units of the IBM Power6 [45] and z10 [160] microprocessors is a 144-bit (36 BCD digit) adder with a 13 FO4 cycle time that completes a Decimal128 floating-point addition in 19 cycles for the worst case. Shift operations are performed sequentially in a two-cycle pipelined rotator (performs both right and left shifts).

Implementations from academic research aim for high-performance by exploiting parallelism without caring about area constraints [136, 157]. Fig. 5.1 shows the block diagram of the DFP adder proposed in [136]. The architecture was implemented only for the IEEE 754-2008 Decimal64 format but can be easily extended for the Decimal128 format. DPD input operands FX and FY are first unpacked and their binary biased exponents, E_X and E_Y, compared using a binary comparator of few bits (10 bits for the Decimal64 format). This unit computes $sign(E_X - E_Y)$ from the carry output of $E_X - E_Y$ to determine the operand with the higher exponent, that is $E_U = max(E_X, E_Y)$. Operands are exchanged if $E_Y > E_X$.

Then, before being added, BCD coefficients U (higher exponent E_U) and L (lower exponent E_L) are aligned, so both operands have the same exponent. This implementation includes an extra digit buffer to the left of operand U. Thus, U can be shifted up to lz_U+1 decimal positions to the left without a loss of precision as explained before (case 2(a)). The leading zeroes of U, lz_U, are detected using a leading zero detector (LZD). In parallel, when necessary, L is shifted to the right to complete the alignment process (case 2(b)). The left (sla) and right shift amounts (sra) are determined as

$$\begin{aligned} sla &= min(E_U - E_L, lz_U + 1) \\ sra &= min\big(max(E_U - E_L - (lz_U + 1), 0), p + 3\big) \end{aligned} \tag{5.2}$$

using binary comparators. The alignment is performed using two barrel shifters, which shift by multiples of 4 bits.

The result of the addition/subtraction of the aligned BCD coefficients (significand BCD addition) is a BCD magnitude $CS = |U + (-1)^{eop}L|$ and a sign bit $sign(S)$. The effective operation, eop, is determined by $eop = s_U \oplus s_L \oplus sub$, where s_U,s_L are the signs of input operands after operand swapping and sub indicates the operation specified by the instruction ($sub = 0$ for addition, $sub = 1$ for subtraction). In order to get correctly rounded results, the addition/subtraction is carried out with three extra precision digits (rounding digit RD, guard digit GD, and sticky bit st). Two extra buffers are placed to the right of operand L for the round digit and the sticky bit. The extra buffer placed to the left of operand U accounts for the guard digit. We detail this implementation in the next Section.

The $p + 4$-digit BCD coefficient CS (including the carry-out of the adder) needs to be

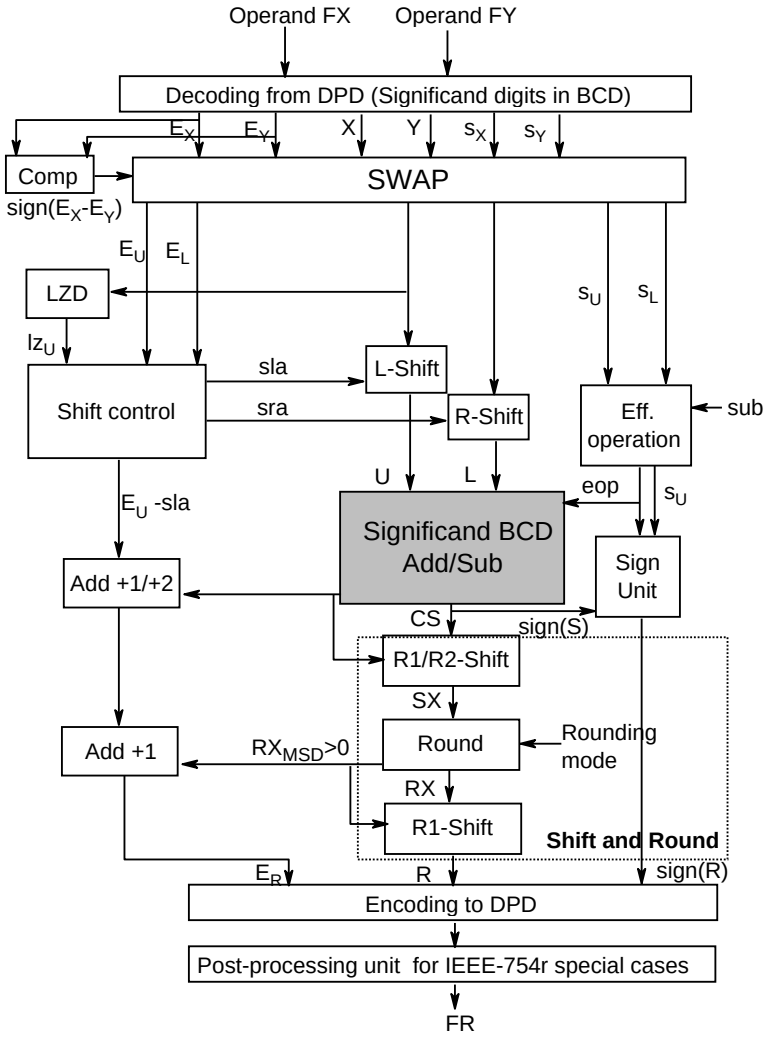


Figure 5.1. *Block diagram of the DFP adder proposed in [136].*

shifted at most two decimal position to the right, but normalization is not required. Rounding is performed after significand addition by selecting $RX = SX$ (coefficient CS shifted and truncated to p-digit precision digits) or $RX = SX + 1$, depending on the rounding digit, the sticky bit and the rounding mode. The sign of the result is computed as

$$sign(R) = \overline{eop}\ s_U \vee eop\ sign(S) \tag{5.3}$$

An additional right shift of one digit may be necessary to obtain the final p-digit rounded result R in case of a carry-out from rounding. The post-processing unit handles special input operands in IEEE 754-2008 such as infinities, NaNs and other exceptions.

Fig. 5.2 shows an improved implementation proposed in [157]. It combines significand BCD addition with rounding using an enhanced sign-magnitude BCD adder to reduce the delay. The scheme for significand BCD addition and decimal rounding is discussed in the next Section.

Other minor modifications are introduced to speedup the computation of the previous

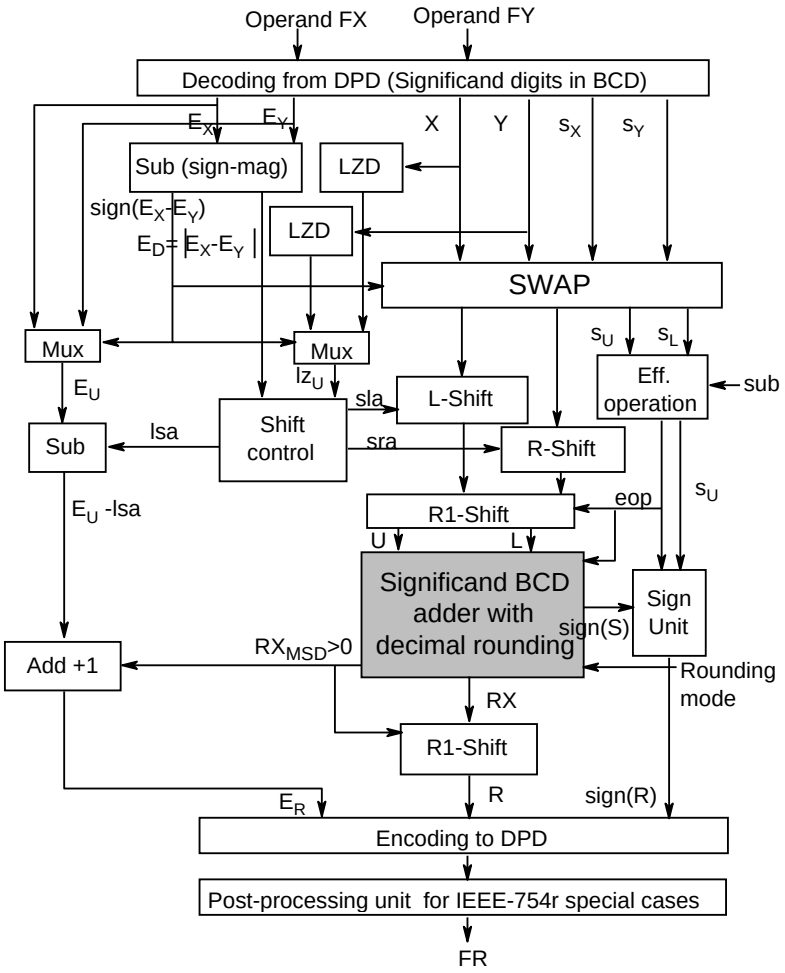


Figure 5.2. *Block diagram of the DFP adder proposed in [157].*

DFP adder. For instance, it uses a 10-bit binary sign-magnitude adder to compute both the exponent difference ($E_D = |E_X - E_Y|$) and $sign(E_X - E_Y)$. In addition, two LZDs compute the number of leading zeroes of X and Y (lz_X, lz_Y), in order to determine speculatively the number of leading zeroes lz_U as

$$lz_U = \begin{cases} lz_X & \texttt{If } E_X \geq E_Y \\ lz_Y & \texttt{Else} \end{cases} \tag{5.4}$$

Thus, the shift amounts for operand alignment are obtained faster as

$$\begin{aligned} sla &= min(E_D, lz_U) \\ sra &= min\big(max(E_D - lz_U, 0), p+3\big) \end{aligned} \tag{5.5}$$

In case of effective addition operations ($eop = 0$), U and L are shifted one extra digit to the right before addition to simplify rounding operations.

By other hand, the biased exponent of the result E_R is computed as

$$E_R = \begin{cases} E_U - sla & \texttt{If } RX_{MSD} = 0 \\ E_U - sla + 1 & \texttt{Else} \end{cases} \tag{5.6}$$

where RX_{MSD} is the MSD of RX (the $p+1$-digit rounded result before the final 1-digit right shift). Finally, to obtain R, the operand RX is shifted one digit to the right if RX_{MSD} is not zero and the guard digit discarded.

5.1.2 Significand BCD addition and rounding

In low-performance implementations [45, 160], significand BCD addition is evaluated using a single 10's complement BCD adder, which computes both $U-L$ (or $U+L$) and $L-U$ sums sequentially. The magnitude result CS is then obtained by selecting the positive sum. The sign of the result is negative ($sign(S)=1$) only when $L-U$ is positive and in case of effective subtraction. By other hand, high-performance DFP adders [136, 157] include a sign-magnitude BCD adder, which compute $sign(S)$ and the BCD coefficient CS directly. However, while the sign-magnitude BCD adder may be implemented using a single compound adder (S, $LS=S+1$), rounding requires an additional +1 ulp increment of the BCD magnitude CS.

Fig. 5.3 shows the implementation for significand addition and rounding proposed in [136], which uses the sign-magnitude BCD adder described in Section 4.2 (see Fig. 4.2 for a detailed implementation of the decimal post-correction unit) and a separate decimal rounding unit. This sign-magnitude BCD adder uses a binary (compound) flagged prefix adder to compute the BCD sums $S=U+L$ ($S=U+\neg L$ for $eop=1$) and $LS=S+1$. Operands U and L are aligned as shown in Fig. 5.3. The sticky bit L_{st} is obtained as the logical OR of the bits of L placed to the right of the round digit L_{RD} (up to $4(p-2)$ bits). These bits are discarded and replaced by[18] L_{st}. The result CS is a $p+4$-digit BCD coefficient obtained as $CS=S$ if $eop=0$, $CS=LS$ if $eop=1$ and $U \geq L$ ($C_{out}=1$), or $CS=\neg S$ if $eop=1$ and $U<L$ ($C_{out}=0$). In case of effective addition ($eop=0$), the carry-out C_{out} is the MSD (most significant digit) of CS. The sign $sign(S)$ of the sign-magnitude BCD addition is computed as $sign(S)=\overline{C_{out}}\ eop$. The sign of the result $sign(R)$ is given by equation (5.3).

Once the addition is complete, CS may need to be shifted two positions to the right if $CS_{MSD}=C_{out}\ \overline{eop}$ is not zero or one position to the right if CS_{MSD-1} is not zero.

The decimal rounding unit of Fig. 5.3 is placed after the significand BCD addition and the R1/R2 shifter. Thus, it receives the BCD p-digit truncated and shifted coefficient SX, the round digit (SX_{RD}) and the sticky bit (SX_{st}). It uses a straightforward implementation of IEEE 754-2008 rounding. This involves detecting a condition for each rounding mode in order to increment or not SX by one ulp. Each increment condition (inc) is determined from the rounding digit, the sticky bit, the sign of the result $sign(R)$ and SX_{lsb}, the least significant bit of SX, as shown in Table 5.1. In addition to the five IEEE 754-2008 decimal rounding modes, it also implements two additional rounding modes, round to nearest down and away from zero.

The +1 ulp increment can generate a new carry propagation from the LSD (least significant digit) up to the MSD of SX. This carry propagation is determined examining the number of consecutive 9's starting from the LSD of SX. A prefix tree of AND gates detects this chain

[18] Actually, the sticky bit is expanded to a 4-bit BCD digit which depends on its value and the effective operation eop.

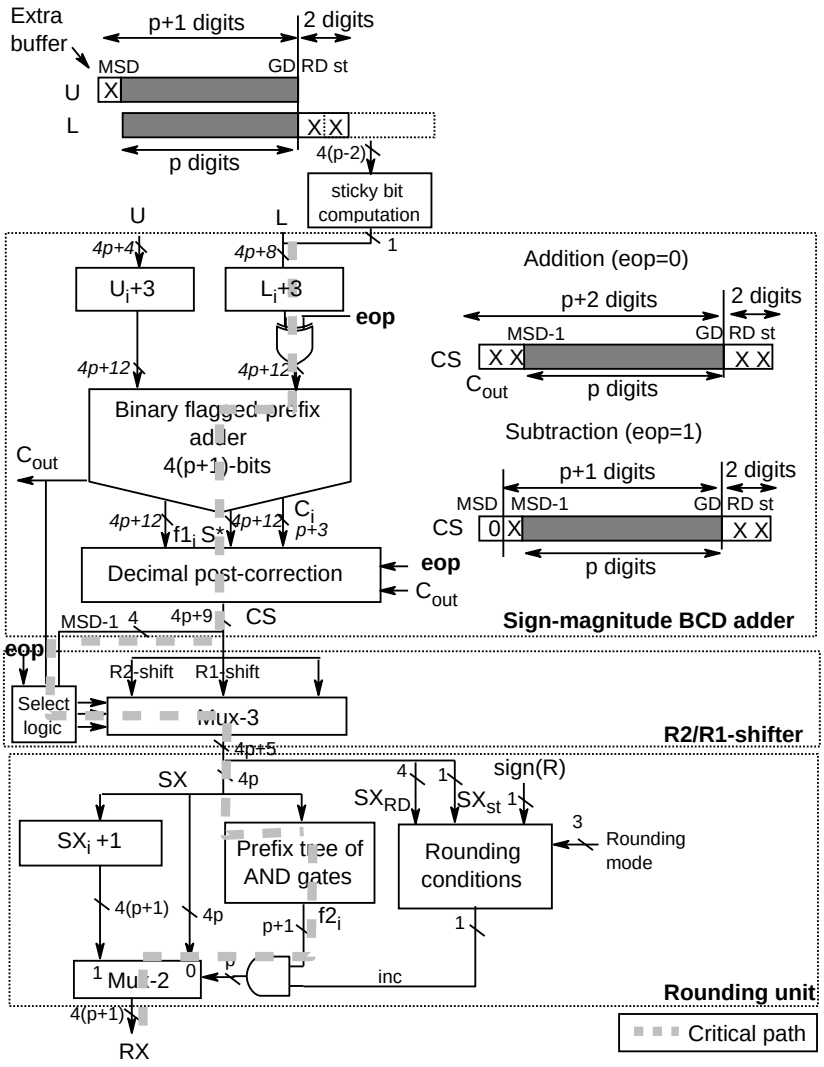


Figure 5.3. *Significand BCD addition and rounding unit [136].*

by computing the following flag bits

$$f2_i = \prod_{k=0}^{i-1} (sx_{k,3}\ sx_{k,0}) \tag{5.7}$$

where $sx_{k,3}$ and $sx_{k,0}$ are the msb and the lsb of each digit SX_k, and $\prod$ represents the product of logical AND operators. When the increment condition for the selected rounding mode is true ($inc = 1$), rounding is performed adding one unit to each BCD digit if the corresponding $f2_i$ is one, that is,

$$RX_i = mod_{10}\big(SX_i + f2_i\ inc\big) \tag{5.8}$$

The additional right shift after rounding only occurs when all the digits of SX are nines and $f2_p\ inc = 1$, so RX is a one followed by p zeroes. Therefore, this shift can be implemented as $R_{MSD} = RX_{MSD} \vee f2_p\ inc$ and $R_i = RX_i$ for $i = 0, \ldots, MSD - 1$.

The implementation proposed in [157] reduces the latency of the scheme of Fig. 5.3

Rounding mode	Increment condition (inc)
To nearest even	$SX_{RD} > 5 \vee (SX_{RD} = 5\ (SX_{lsb} = 1 \vee SX_{st} = 1))$
To nearest up	$SX_{RD} \geq 5$
To nearest down	$SX_{RD} > 5 \vee (SX_{RD} = 5\ S_{st} = 1)$
Toward $+\infty$	$sign(R) = 0\ (SX_{RD} > 0 \vee (SX_{RD} = 0\ SX_{st} = 1))$
Toward $-\infty$	$sign(R) = 1\ (SX_{RD} > 0 \vee (SX_{RD} = 0\ SX_{st} = 1))$
Toward zero	0
Away from zero	$SX_{RD} > 0\ \vee (SX_{RD} = 0\ SX_{st} = 1)$

Table 5.1. *Rounding modes implemented in [136].*

Rounding mode	$sign(R^*)$	inj (RD,st)	inj_{cor} (GD,RD)
To nearest even		(5,0)	(4,5)
To nearest up		(5,0)	(4,5)
To nearest down		(4,9)	(4,5)
Toward $+\infty$	0	(9,9)	(9,0)
	1	(0,0)	(0,0)
Toward $-\infty$	0	(0,0)	(0,0)
	1	(9,9)	(9,0)
Toward zero		(0,0)	(0,0)
Away from zero		(9,9)	(9,0)

Table 5.2. *Injection values for the rounding modes implemented in [157].*

combining part of significand BCD addition with decimal rounding. Fig. 5.4 shows the block diagram of the architecture. Although the implementation was originally proposed for the Decimal64 format (16 digits), we also consider an implementation for Decimal128 (34 digits). It uses the sign-magnitude BCD adder of Fig. 3.23 and a decimal variation of the rounding by injection algorithm [54]. This method is based on adding an injection value (inj) to the input operands that reduces all the rounding modes to round towards zero (truncation by the round digit position). However, when the truncated result has more significant digits than the format precision p, it is necessary to inject a correction amount inj_{cor} to obtain the correctly rounded result.

The layout of the $p+3$-digit coefficients U and L after operand alignment is shown in Fig. 5.4. The 3 extra positions to the right of L are for the guard digit(L_{GD}), the round digit (L_{RD}) and the sticky bit (L_{st}). Note that for effective addition ($eop = 0$) the MSD of U and L is zero, so the carry-out of the addition is the MSD digit of the $p+3$-digit magnitude result $CS = |U + (-1)^{eop}L + inj|$. Therefore, only a 1-digit right shift of RX ($p+1$ digit result after rounding) is required when RX_{MSD} is not zero.

The decimal injection values inj are inserted into the RD and st positions of U. The injection values inj and inj_{cor} for IEEE 754-2008 decimal rounding modes are shown in Table 5.2. The value

$$sign(R^*) = \overline{eop}\ s_U \tag{5.9}$$

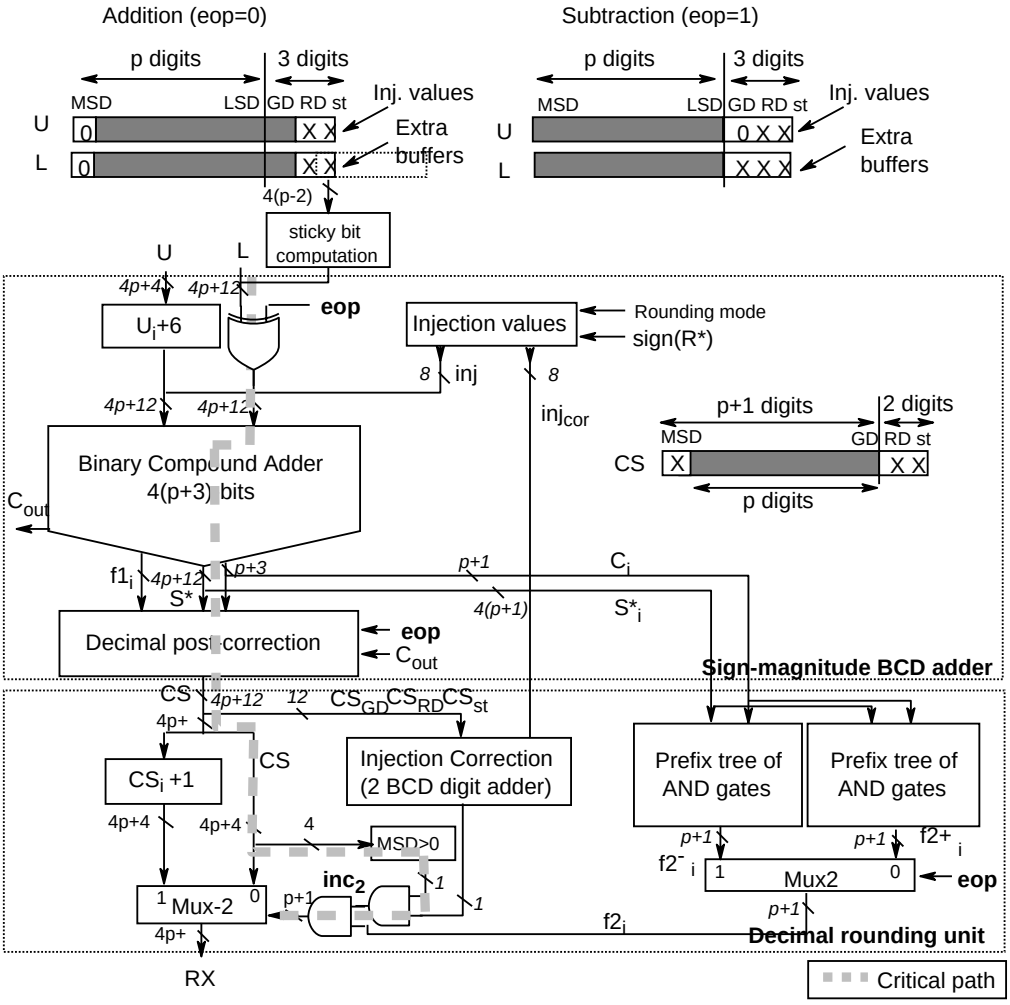


Figure 5.4. *Significand addition and rounding unit [157].*

is the expected sign of the result $sign(R)$ assuming that $U - L$ is always positive. When $U - L$ is negative, there is no need to perform rounding since R is guaranteed to be at most a p-digit operand.

When the MSD of CS is not zero, the corresponding injection correction value inj_{cor}, shown in Table 5.2, is used to compute the correctly rounded result R. The inj_{cor} value is added to CS_{GD} and CS_{RD} using a 2-digit BCD adder. The increment signal inc is true if a carry-out is generated to the LSD position of CS.

This correction can generate a decimal carry propagation from the LSD of CS, determined by the trailing chain of 9's of S. To reduce the critical path delay of the unit, the trailing 9's detection (logarithmic delay, depends on the number of precision digits) is performed examining the uncorrected binary sum digits S_i^* and the decimal carries C_i, obtained before the BCD correction in the sign-magnitude BCD adder (see Fig. 3.23 in Section 3.4.2). A pair

of prefix trees (one for addition and one for subtraction) compute the flags $f2_i^+$ and $f2_i^-$ as

$$f2_i^+ = \prod_{k=0}^{i-1}((S_k^* == 15) \vee (S_k^* == 9)\ C_{k+1})$$
$$f2_i^- = \prod_{k=0}^{i-1}(S_k^* == 15) \tag{5.10}$$

The digits of the $p+1$-digit correctly rounded result RX are selected from $CS_i + 1$ (digitwise increment) or CS_i, depending on these flags ($f2_i^+$ for $eop = 0$, $f2_i^-$ for $eop = 1$) and the signal inc.

In the next Section we present the proposed method for significand BCD addition and rounding. The resulting architecture presents the following advantages with respect to the two previous implementations [136, 157]:

1. Decimal rounding does not require any additional carry propagation for trailing 9's detection. The signals required for rounding the significand BCD result, CS, are computed in an enhanced compound BCD adder.

2. The modifications introduced into the compound BCD adder to support decimal rounding only increments its critical path delay a little constant amount, independently of the number of input digits.

3. The BCD sums are obtained from a binary compound adder using a simple and fast decimal post-correction.

5.2 Proposed method for combined BCD significand addition and rounding

This method is an extension of the conditional speculative algorithm for sign-magnitude BCD addition proposed in Section 3.5. We use a single compound BCD adder to perform both sign-magnitude BCD addition and +1/+2 ulp increments of the significand for IEEE 754-2008 decimal rounding.

Input BCD significands X and Y are swapped and aligned as described in [157] (Fig. 5.2). The layout of operands U and L before significand addition is shown in 5.5. As a difference with respect to Fig. 5.2, operand alignment is performed in a single step, replacing the left and right shift amounts defined in expression (5.5) by

$$lsa_U = min(E_D + eop, lz_U + eop)$$
$$rsa_L = min\big(max(E_D - lz_U + \overline{eop}, \overline{eop}), p+3\big) \tag{5.11}$$

while expression (5.6) is still valid to compute the biased exponent E_R.

A digit buffer is placed to the left of U ($p+1$-digit operand) to allow for the extra 1-digit left shift in case of effective subtractions. Operand L includes two extra digits to the right (GD, RD) and the sticky bit st, computed as the logical OR of the bits of L placed to the right of RD. We describe the proposed method independently of the implementation of rounding. Hence,

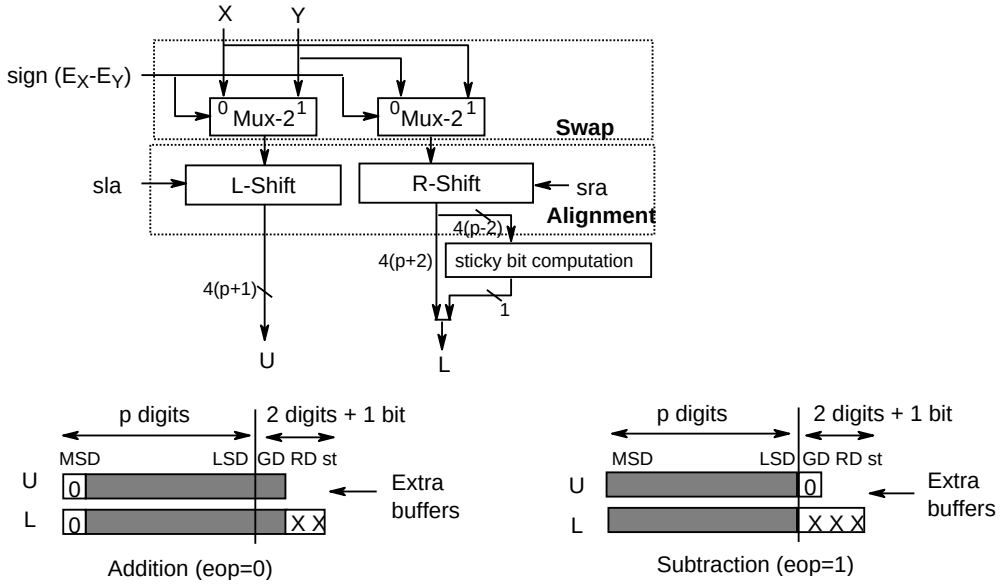


Figure 5.5. *Alignment and layout of input operands.*

the layout of Fig. 5.5 does not include the extra buffers for rounding by injection. We provide the implementations for two different rounding methods (direct rounding and rounding by injection) in Section 5.3.

A diagram with the proposed method to compute significand BCD addition with IEEE 754-2008 decimal rounding is detailed in Fig. 5.6. The result is the $p+1$-digit BCD rounded coefficient RX. Each computation consists of five stages: pre-correction stage, modified binary addition, rounding decision, increment decision and selection.

BCD operands U and L are split in two parts, a p-digit most significant one (U^D, L^D) and a least significant one, which includes the digits/bits placed at the guard (U_{GD}, L_{GD}), round (L_{RD}) and sticky (L_{st}) positions. The least significant part is used by the rounding decision stage. The rounding decision stage may produce at most a +2 ulp increment ($inc_1 + inc_2$) of the BCD sum $S^D = U^D + L^D$ (or $S^D = U^D + \neg L^D$), which would require the computation of S^D ($\neg S^D$), $S^D + 1$ and $S^D + 2$.

We use a single compound BCD adder (computes S^D and $S^D + 1$) to perform both significand BCD addition and decimal rounding. To support the additional $S^D + 2$ operation, we perform a carry-save addition before the compound adder. This technique has been widely used in binary floating-point adders [8, 16, 125]. Moreover, to simplify the increment logic, we split the evaluation of S^D in a most significant part S^H and a lsb (least significant bit) S^D_{lsb}, such that $S^D = S^H + S^D_{lsb}$ (note that the lsb of S^H is zero). In this way, an increment of +0,+1 or +2 ulp in S^D is equivalent to an increment inc of +0 or +2 ulp in S^H.

Thus, operands U^D and L^D are first processed (in the pre-correction stage) using a carry-save adder which produces two p-digit operands $Ops = Ops^H + S^D_{lsb}$ and Opd. Both p-digit operands Ops^H and Opd have the lsb set to zero. S^D_{lsb} is delivered together with inc_1 and inc_2 to the increment decision stage to compute a control signal inc and the lsb R_{lsb}. Furthermore, the pre-correction stage uses a scheme based on conditional decimal speculative addition (see

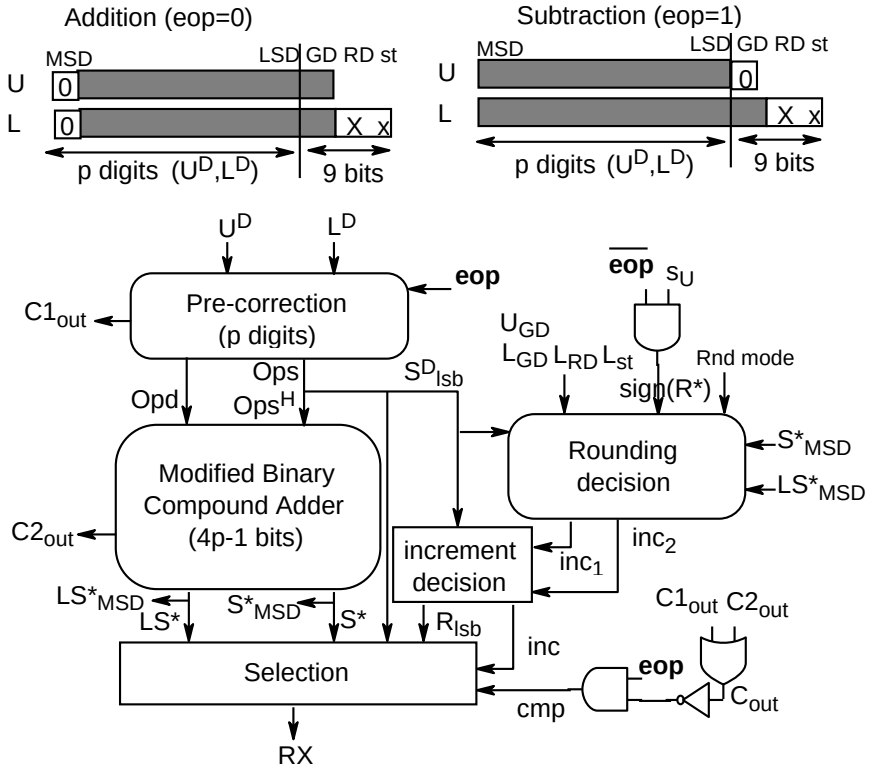


Figure 5.6. *Proposed method for significand BCD addition with rounding.*

Section 4.3) which allows to obtain the BCD sums S^H and $LS^H = S^H + 2$ from a $(4p-1)$-bit binary compound addition of Ops^H and Opd (shifted 1-bit to the right since their lsb's are zero).

Finally, the $4p-1$ most significant bits of RX are selected from the binary sums S^* and $LS^* = S^* + 1$ depending on the values of $cmp = eop\ \overline{C_{out}}$ and inc, as detailed in Section 4.4. The p-digit rounded result R is obtained from the p least significant digits of RX after performing a 1-digit right shift and removing the digit RX_{GD} if RX_{MSD} is not zero.

The whole computation is carried out as follows:

1. **Pre-correction stage.** To allow the use of binary operations, p-digit BCD operands U^D and L^D are processed in the pre-correction stage as detailed in Fig. 5.7.

   The carry-save addition of U^D and L^D (or $\neg L^D$) produces a $4p$-bit sum operand $Ops = Ops^H + Ops_{lsb}$ and a $4p$-bit carry operand Opc, such that $S^H = Ops^H + 2\ Opc$ and $S^D_{lsb} = Ops_{lsb}$. To compute this operation using a binary carry-save adder, each decimal digit position is first incremented +6 units. Hence, instead of multiplying Opc by 2, we only shift Opc 1-bit to the left ($Opb = L1_{shift}[Opc]$), as in a binary carry-save addition. The resultant 4-bit vector Opb_i represents the BCD i-digit of $2\ Opc$ if $Opb_{i+1,0}$ (bit 2^0 of Opb_i) is one. In other case, Opb_i is the BCD excess-6 representation of $(2\ Opc)_i$.

   Due to the initial +6 addition, the decimal carries of S^H corresponds with the binary carries of $Ops^H + L1_{shift}[Opc]$ at hexadecimal positions (1 out of 4). After the binary carry-save addition, a value $U_i + L_i + 6 = 24$ (or $U_i + \overline{L_i} = 24$) at position i, results in a value 8

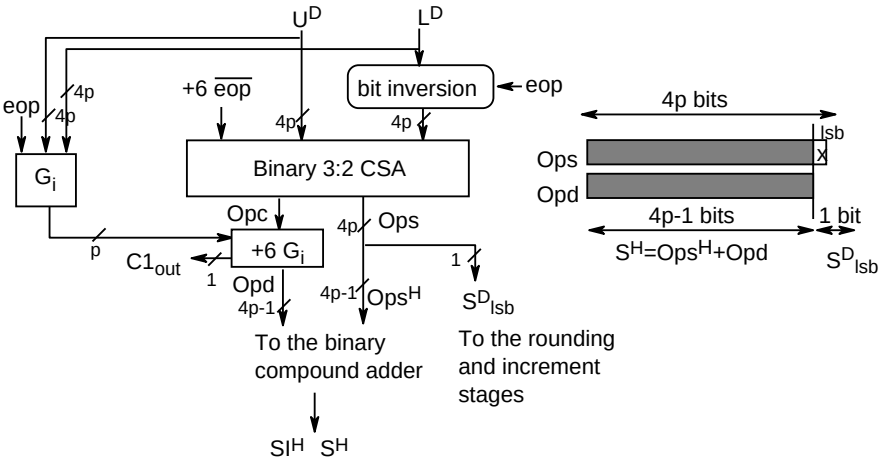


Figure 5.7. *Pre-correction stage.*

or 9, since a carry output with a value of 16 is passed to the next significant digit, and a carry 0 or 1 comes from the previous significant digit. An increment of +2 ulp in S^H could produce a new decimal carry propagation. Thus, to obtain the decimal carries (late carries) of the incremented sum $LS^H = S^H + 2$ by means of the binary addition of Ops^H and $L1_{shift}[Opc]$, we would need to add an additional +6 to each position i if $U_i + L_i = 18$.

Moreover, to avoid a complex post-correction, we also add this additional +6 under certain conditions, so the resultant binary sum vectors S^* and $LS^* = S^* + 1$ represents the BCD sums S^H and LS^H but coded in BCD excess-6 (except in a single case, corrected as shown below).

Therefore, to obtain the BCD sums S^H and LS^H using a binary compound addition, we propose to replace the operand $2\ Opc$ by Opd, obtained by incrementing conditionally +6 units each digit of $L1_{shift}[Opc]$. We use the decimal generate function

$$G_i = G_i^{+}\overline{eop} \vee G_i^{-}\ eop \tag{5.12}$$

as the condition for the additional +6 digit increment of $L1_{shift}[Opc]$, where G_i^+ and G_i^- are defined as

$$G_i^{+} = \begin{cases} 1 & \texttt{If } U_i + L_i \geq 10 \\ 0 & \texttt{Else} \end{cases}$$
$$G_i^{-} = \begin{cases} 1 & \texttt{If } U_i + \overline{L_i} \geq 16 \\ 0 & \texttt{Else} \end{cases} \tag{5.13}$$

Hence, a +6 is added to the digit i of $L1_{shift}[Opc]$ if G_i is true, or equivalently,

$$Opd = L1_{shift}[\sum_{i=0}^{p-1}(Opc_i + 3\ G_i)\ 10^i] \tag{5.14}$$

Since the digits of Opc_i are in the range $[0, 10]$ (we assume $6 \leq U_i + L_i + 6 \leq 24$), the sum $Opc_i + 3\ G_i$ may be represented in 4 bits. However, the subsequent 1-bit left shift generates a decimal carry-out $C1_{i+1} = Opd_{i+1,0}$ from decimal position i. Note that another

decimal carry-out $C2_{i+1}$ is produced at decimal position i due to the subsequent carry-propagate addition of Ops and Opd, but they cannot have simultaneously a value '1', so

$$C_{i+1} = C1_{i+1} \vee C2_{i+1} \tag{5.15}$$

The carry-out $C_{out} = C1_{out} \vee C2_{out}$ (see Fig. 5.7) is used to determine the sign of the result in case of effective subtraction, so if $cmp = eop\ \overline{C_{out}} = 1$, then $U - L < 0$.

2. **Modified binary carry-propagate compound addition.** In this way, we evaluate the BCD sum S^H by means of a modified $(4p-1)$-bit binary carry-propagate addition $S^* = R1_{shift}[Ops^H] + R1_{shift}[Opd]$, where $R1_{shift}$ is a fixed 1-bit right shift. The binary carries of S^* are computed as $c_{k+1} = g_k \vee a_k\ c_k$, where g_k and a_k are the conventional binary carry-generate and carry-alive functions.

   We define a digit S_i^* as

$$S_i^* = \sum_{j=0}^{3} s_{i,j}^*\ 2^j = \sum_{k=4i}^{4i+3} s_{k-1}^*\ 2^{k-4i} \tag{5.16}$$

   where s_k^* ($k = 4i + j = 0, \ldots, 4p-2$) are the bits of the modified binary sum and $s_{-1}^* = 0$. Depending on the value of the decimal carry output at position i (C_{i+1}) and the increment of the input digits (+6 /+12), S_i^* is related with the BCD sum digit S_i^H as follows:

   - If C_{i+1} is zero (so[19] $G_i = 0$), a +6 has been added in excess to position i. Thus, S_i^* represents the value S_i^H in BCD excess-6 ($S_i^* = S_i^H + 6$).
   - If $C_{i+1} = 1$ and $G_i = 0$, the +6 initially added compensates for the +6 digit correction (since $C_{i+1} = 1$), so S_i^* is equal to the BCD sum digit S_i^H. However, to avoid a post-correction that would depend on C_{i+1}, S_i^* should be equal to $S_i^H + 6$. This case only occurs when $U_i + L_i = 9$ (that is, the decimal carry-alive $A_i = 1$) and $C_i = 1$, so $S_i^H = 0$ ('0000') and $S_i^* = 6$ ('0110'). This case is detected by checking the following boolean expression

$$p_{i,3}\ p_{i,2}\ p_{i,1}\ c_{i,1} = 1 \tag{5.17}$$

     where the $p_{i,j} = p_{k-1}$ ($k = 4i + j$), are the binary xor propagate functions and $c_{i,1} = c_{k-1}$. Replacing the conventional binary sum equation $s_k^* = p_k \oplus c_k$ by

$$\begin{aligned} s_{i,3}^* &= p_{i,3} \oplus c_{i,3} \\ s_{i,2}^* &= p_{i,2}\ \overline{c_{i,2}} \vee (\overline{p_{i,2}} \vee p_{i,3}\ p_{i,1})\ c_{i,2} \\ s_{i,1}^* &= p_{i,1}\ \overline{c_{i,1}} \vee (\overline{p_{i,1}} \vee p_{i,3}\ p_{i,2})\ c_{i,1} \\ s_{i,0}^* &= p_{i,0} \oplus c_{i,0} \end{aligned} \tag{5.18}$$

     then, the value $S_i^H = 0$ ('0000') is transformed into its BCD excess-6 representation $S_i^* = 6$ ('0110') when (5.17) is true. This modification does not affect to the critical path (the carry path).
   - If $C_{i+1} = 1$ and $G_i = 1$, the sum digit is incremented in +12 units but only +6 units are required for digit correction (since $C_{i+1} = 1$), so the resulting S_i^* is the BCD excess-6 representation of the BCD sum digit S_i^H.

[19] G_i is related to C_{i+1} as $C_{i+1} = G_i \vee A_i\ C_i$, where A_i is the decimal carry-alive.

By other hand, the length of the decimal carry propagation due to a late +1/+2 ulp increment of the BCD sum S^H is determined by a trailing chain of k 9's or $(k-1)$ 9's and an 8. Since $S_i^* = S_i^H + 6$ and $s_{0,0}^* = 0$, then the previous chains corresponds to

$$\sum_{i=k}^{p-1} S_i^* 10^i + 15\ 10^{k-1} + \ldots + 15\ 10^1 + 14 \tag{5.19}$$

that is, to a trailing chain of bits '1' in the $(4p-1)$-bit binary sum S^*.

As we have shown in Section 4.2, a carry due a +1 ulp increment of S^* is propagated from the lsb to bit k if the binary carry-alive group $a_{k-1:0}$, computed in a prefix tree as

$$a_{k-1:0} = a_{k-1}\ a_{k-2} \ldots\ a_0 \tag{5.20}$$

is true. The late binary carries lc_k, given by

$$lc_k = c_k \vee a_{k-1:0} \tag{5.21}$$

correspond to the binary carries of $LS^* = S^* + 1$. We define the digits LS_i^* from the sum bits of LS^* as

$$LS_i^* = \sum_{j=0}^{3} ls_{i,j}^*\ 2^j = \sum_{k=4i}^{4i+3} ls_{k-1}^*\ 2^{k-4i} \tag{5.22}$$

with $ls_{-1}^* = 0$ and $k = 4i + j$.

Hence, the expressions for the digits LS_i^* are obtained replacing the carries $c_{i,j}$ by the late carries $lc_{i,j}$ in equation (5.18), that is

$$\begin{aligned}
ls_{i,3}^* &= p_{i,3} \oplus lc_{i,3} \\
ls_{i,2}^* &= p_{i,2}\ \overline{lc_{i,2}} \vee (\overline{p_{i,2}} \vee p_{i,3}\ p_{i,1})\ lc_{i,2} \\
ls_{i,1}^* &= p_{i,1}\ \overline{lc_{i,1}} \vee (\overline{p_{i,1}} \vee p_{i,3}\ p_{i,2})\ lc_{i,1} \\
ls_{i,0}^* &= p_{i,0} \oplus lc_{i,0}
\end{aligned} \tag{5.23}$$

The digits LS_i^* represents the digits of the BCD sum $LS^H = S^H + 2$ but coded in BCD excess-6.

3. **Rounding decision stage.** In the rounding decision step, we determine the value of signals inc_1, inc_2 and R_{GD}, the guard digit of the magnitude result. Each one of the bit signals inc_1 and inc_2 represent an increment of +1 unit at the lsb position of $S = U^H + (-1)^{eop}\ L^H$.

   Apart from the rounding mode, the rounding decision depends on the effective operation eop, the guard digits U_{GD}, L_{GD}, the round digit L_{RD}, the sticky bit L_{st}, the MSD and the carry-out C_{out} (always zero for effective addition) of S^* and the MSD of LS^*. Some rounding modes also require the lsb of Op_s (S_{lsb}^D), or the expected sign of the result,

$$sign(R^*) = \overline{eop}\ sign(U) \tag{5.24}$$

   It is obtained considering only the cases where rounding is necessary, and it is equal to sign S_{inj} of expression (5.9). The rounding logic has to manage the following five cases:

- **Case 1:** Addition ($eop = 0$) and $S^*_{MSD} = 6$ ($S^H_{MSD} = 0$). The addition of the guard digits $S_{GD} = U_{GD} + L_{GD}$ produces a carry-out inc_1. The second carry-out inc_2 is obtained, depending on the rules of the rounding mode, from the digit in the guard and round positions (S_{GD}, L_{RD}), the sticky bit Lst, the lsb bit ($S^D_{lsb} \oplus inc_1$), and $sign(R^*)$. An exception can occur when $LS^*_{MSD} = 7$ ($S^H_{MSD} = 1$), which corresponds to $S^H = 0999\ldots98$ and $LS^H = 1000\ldots00$. In this case if $S^D_{lsb} + inc_1 = 2$, then the digit in the round position is S_{GD}, not L_{RD}, and the rounding conditions to obtain inc_2 are similar as those in **Case 2**.
- **Case 2:** Addition ($eop = 0$) and $S^*_{MSD} > 6$ ($S^H_{MSD} > 0$). The addition of the guard bits $S_{GD} = U_{GD} + L_{GD}$ also produces a carry-out inc_1. A second carry-out inc_2 is produced from rounding. But now, S_{GD} is the digit in the round position and a new sticky bit is computed as $L_{RD} \vee L_{st}$.
- **Case 3:** Subtraction ($eop = 1$) and $C_{out} = 0$. This case corresponds with $U - L < 0$ ($E_U \geq E_L$). In this case, only operand U is shifted to the left if the leading zeroes of U $l_z \geq E_U - E_L > 0$ or $E_U = E_L$. Therefore, no rounding is required if $cmp = eop\ \overline{C_{out}}$ is one.
- **Case 4:** Subtraction ($eop = 1$), $C_{out} = 1$ and $S^*_{MSD} = 6$ ($S^H_{MSD} = 0$). This corresponds with $U - L = U + \neg L + 1 \geq 0$. A +1 ulp must be added to $\neg L$ to form the 10's complement. This unit is incorporated into the sticky bit calculation. If $L_{st} = 0$, then the digits of $\neg L$ placed to the right of $\neg L_{RD}$ are all 9's. In this case a carry-out '1' is propagated to $\neg L_{RD}$ and the sticky bit of $\neg L$ is unchanged ($\neg L_{st} = L_{st} = 0$). If $L_{st} = 1$, then, at least one of the digits of $\neg L + 1$ placed to the right of L_{RD} is different from zero, so $\neg L_{st} = L_{st} = 1$ and no carry-out is propagated to $\neg L_{RD}$. Therefore, the sticky bit is equal to L_{st} and a bit $\overline{L_{st}}$ is added to $\neg L_{RD}$, which generates a carry-out $inc_1 = 1$ if $\neg L_{GD} = 9$.

  A second carry-out inc_2 is produced applying the rounding rules, with $S_{RD} = \neg L_{RD} + \overline{L_{st}}$ (the rounding digit) and L_{st} (the sticky bit). Note that, if $LS^*_{MSD} = 7$ ($LS^H_{MSD} = 1$), then $inc_1 = 1$ and $S_{GD} = S_{RD} = L_{st} = 0$, so we can use the rounding rules of case **Case 4** or **Case 5** indifferently.
- **Case 5:** Subtraction ($eop = 1$), $C_{out} = 1$ and $S^*_{MSD} > 6$ ($S^H_{MSD} > 0$). The sticky bit L_{st}, the guard digit S_{GD}, the round digit S_{RD} and inc_1 are evaluated as in the previous case. However, the digit in the rounding position is now S_{GD} and the sticky bit is $S_{RD} \vee Lst$.

The implementation of the rounding decision stage is detailed in Section 5.3.1, for direct decimal rounding, and in Section 5.3.2 for decimal rounding by injection.

4. **Increment decision stage.** S^D_{lsb} is added to bits inc_1 and inc_2, producing a carry bit inc and a sum bit RX_{lsb}. Signal inc represents an increment of +2 ulp of S^H, and it is used as a control signal to select between S^H or $S^H + 2$.

5. **Selection stage.** The block diagram of this stage is shown in Fig. 5.8. The $p + 1$-digit rounded significand result RX is formed as follows:

   - If $cmp = 1$ then $U - L < 0$ (no rounding required), and the magnitude result R is given by the 9's complement of $S = S^H + S^D_{lsb}$. Since the 9's complement of the BCD sum

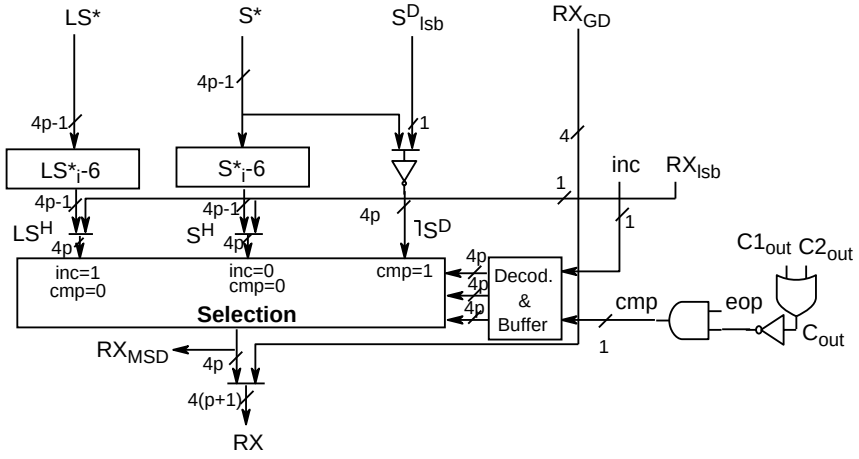


Figure 5.8. *Proposed selection stage.*

digits $S_i^H = S_i^* - 6$ is given by

$$\neg S_i^H = \overline{(S_i^H + 6)} = \overline{S_i^* - 6 + 6} = \overline{S_i^*} \tag{5.25}$$

then RX is obtained by bit inverting S^* and S^D_{lsb} and concatenating them with $RX_{GD} = 0$ (the guard digit computed in the rounding decision stage).

- If $inc = 1$ and $cmp = 0$ then RX is obtained concatenating LS^H, RX_{lsb} and RX_{GD}, where RX_{lsb} was computed in the increment decision stage. The BCD sum LS^H is obtained from the digits LS_i^* after a +6 digit subtraction, that is

$$LS = \sum_{i=0}^{p-1} (LS_i^* - 6)\ 10^i \tag{5.26}$$

- If $inc = 0$ and $cmp = 0$, then $RX = (S^H, RX_{lsb}, RX_{GD})$, where S^H is obtained from the S_i^* digits as

$$S^H = \sum_{i=0}^{p-1} (S_i^* - 6)\ 10^i \tag{5.27}$$

The p-digit coefficient result R is obtained from the p least significant digits of RX after performing a 1-digit right shift and removing the digit RX_{GD} if RX_{MSD} is not zero, as described in Section 5.1.1.

5.3 Architecture of the significand BCD adder with rounding

In this Section we detail the architecture of the significand BCD adder with IEEE 754-2008 decimal rounding. First, we describe the general block diagram depicted in Fig. 5.9. The implementation of the rounding logic is detailed in Section 5.3.1 for the direct scheme and in Section 5.3.2 for the injection scheme. The proposed architecture accepts a $(p + 1)$-digit BCD operand U, a $(p+2)$-digit BCD operand L and a sticky bit L_{st}, pre-aligned as described in Section 5.2. Actually, the input digits U_i and L_i are allowed to be in the range $[0, 15]$ (overloaded

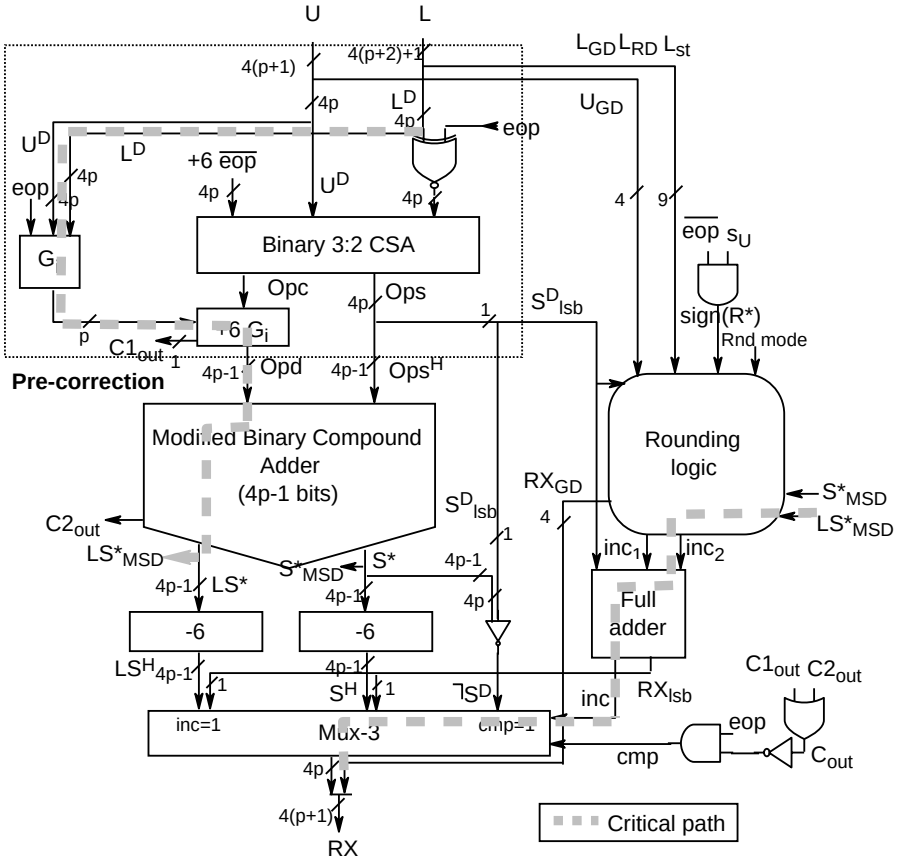


Figure 5.9. *Proposed sign-magnitude BCD adder with rounding.*

BCD) under the following constraints:

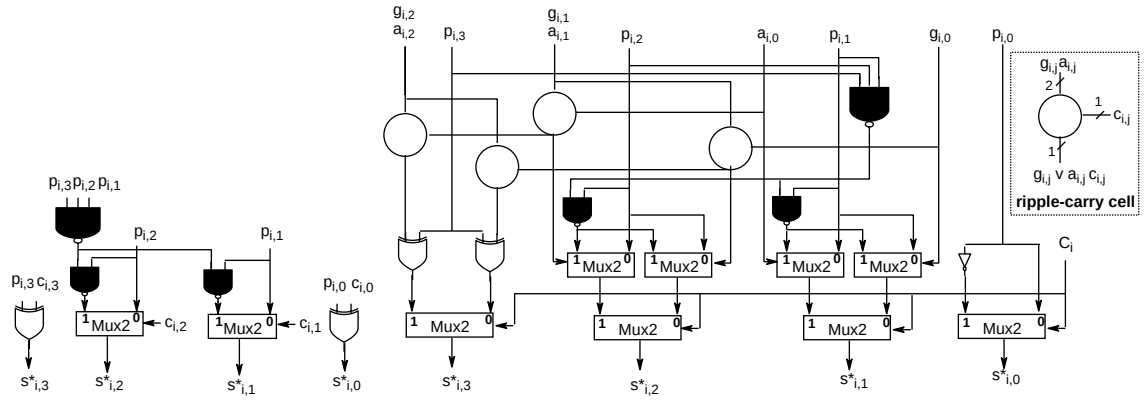
$$\begin{aligned} 0 \leq U_i + L_i \leq 18 \qquad & \texttt{If(eop=0)} \\ 0 \leq U_i + \neg L_i \leq 18 \qquad & \texttt{Else} \end{aligned} \tag{5.28}$$

U^D and L^D (the p most significant digits of U and L) are first processed in the pre-correction stage. For effective addition ($eop = 0$), we perform the digit additions $U_i + L_i + 6$ in a $4p$-bit binary 3:2 CSA, obtaining the $4p$-bit sum and carry operands Ops and Opc. The +6 digits are coded in BCD as $(0, \overline{eop}, \overline{eop}, 0)$ and are connected to an input of each 4-bit block of the binary 3:2 CSA. The $4p$-bit vectors U^D and L^D are introduced into the other two inputs.

For effective subtraction ($eop = 1$), the +6 digit additions are performed together with the 9's complement of L by inverting the bits of L_i, that is, $U_i + \neg L_i + 6 = U_i + \overline{L_i}$.

Next, the $4p$-bit vectors U^D and $\overline{L^D}$ are introduced in the binary 3:2 CSA. The evaluation of $\overline{L^D}\ eop \vee L^D\ \overline{eop}$ is performed in a level of $4p$ XOR gates, which is connected to the fast input of the 3:2 binary CSA.

The operand Ops is split in the p-digit operand Ops^H ($ops_{0,0} = 0$) and in the bit S^D_{lsb}. The operand Opd is obtained in the $+6\ G_i$ block from the digits of Opc and the decimal carry generates G_i as stated in (5.14). This block implements the following logical equations for



(a) For full binary prefix tree adders.

(b) For hybrid carry-select/prefix tree adders.

Figure 5.10. *Modified binary 4-bit sum cells.*

each digit i:

$$
\begin{aligned}
Opd_{i,3} &= Opc_{i,2} \oplus (Opc_{i,1} \vee Opc_{i,0})\ G_i \\
Opd_{i,2} &= Opc_{i,1} \oplus \overline{Opc_{i,0}}\ G_i \\
Opd_{i,1} &= Opc_{i,0} \oplus G_i \\
Opd_{i,0} &= Opc_{i-1,3} \vee Opc_{i-1,2}(Opc_{i-1,1} \vee Opc_{i-1,0})\ G_{i-1}
\end{aligned}
\tag{5.29}
$$

The decimal carry generates G_i are evaluated in parallel with Opc to reduce the critical path delay. The critical path is indicated as a discontinuous thick line in Fig. 5.9. The block G_i implements equations (5.12) and (5.13) as

$$
\begin{aligned}
G_i^+ &= a_{i,3}^+(g_{i,3}^+ \vee a_{i,2}^+) \vee (a_{i,3}^+ \vee g_{i,2}^+)(a_{i,1}^+ \vee g_{i,0}^+) \\
G_i^- &= a_{i,3}^-\big(g_{i,3}^- \vee g_{i,2}^- \vee a_{i,2}^-\ a_{i,1}^-\ (g_{i,1}^- \vee g_{i,0}^-)\big) \\
G_i &= G_i^+\ \overline{eop} \vee G_i^-\ eop
\end{aligned}
\tag{5.30}
$$

with $g_{i,j}^+ = u_{i,j}\ l_{i,j}$, $a_{i,j}^+ = u_{i,j} \vee l_{i,j}$, $g_{i,j}^- = u_{i,j}\ \overline{l_{i,j}}$ and $a_{i,j}^- = u_{i,j} \vee \overline{l_{i,j}}$.

Despite the logic depth of both parallel paths is practically similar (4 logic levels vs. 2 XOR gate levels), the computation of the G_i is slightly slower than the computation of Opc and Ops due to the higher loads. A quantitative delay analysis is presented in Section 5.4.1.

The operands $R1_{shift}[Ops^H]$ and $R1_{shift}[Opd]$ are introduced in the $(4p-1)$-bit binary compound adder obtaining S^* and $LS^* = S^* + 1$. The adder implements a prefix carry tree of $\log_2(4p-1)$ levels. The sum cells are modified as shown in Fig. 5.10(a). This 4-bit sum cell evaluates expression (5.18) from the binary carries $c_{i,j}$ and carry-propagates $p_{i,j}$. The black gates represent additional hardware with respect to a conventional 4-bit XOR sum cell. These gates, placed out of the critical path (the carry path), replace the values $S_i^* = 0$ ('0000') by $S_i^* = 6$ ('0110'). Equivalent 4-bit sum cells are required to evaluate the bits of the incremented sum LS^*, as shown in (5.23). In this case, a level of OR gates computes the binary late carries as $lc_{i,j} = c_{i,j} \vee a_{k-1:-1}$ ($k = 4i + j$) from the binary carries and the carry-alive groups obtained previously in the prefix tree.

An hybrid binary carry-select/sparse prefix-tree adder can be used instead of a full binary prefix tree adder to reduce the power consumption [100]. In this case, a row of the 4-bit carry-select sum cells of Fig. 5.10(b) is used to obtain the presum digits $S1_i^*$ (assuming $C_i = 1$) and $S0_i^*$ (assuming $C_i = 0$). A quaternary (sparse) prefix tree computes in parallel the carries C_i (1 each 4 bits). The digits of the sum S^* are then selected as $S_i^* = S1_i^*\ C_i \vee S0_i^*\ \overline{C_i}$. Only the binary presum $S1^*$ requires the digit correction performed by the black gates (replacement of '0000' by '0110').

The decimal rounding unit computes the increment signals inc_1 and inc_2 and the guard digit of the result RX_{GD}. The most part of this evaluation is overlapped with the binary sum. However, as described in Section 5.2, the evaluation of inc_2 and RX_{GD} depends on the MSDs of the binary sums S^* and LS^*. The implementation of the rounding unit is detailed in the next Sections (Section 5.3.1 for the direct scheme and Section 5.3.2 for injection scheme).

The signal inc and the lsb of the rounded result RX_{lsb} are computed adding inc_1, inc_2 and S_{lsb}^D in a full adder as

$$\begin{aligned} RX_{lsb} &= (S_{lsb} \oplus inc_1) \oplus inc_2 \\ inc &= S_{lsb}^D\ (S_{lsb}^D \oplus inc_1) \vee inc_2 \overline{(S_{lsb}^D \oplus inc_1)} \end{aligned} \tag{5.31}$$

This full adder contributes with an XOR gate delay to the total delay of the critical path (inc_2 signal).

The $4p-1$ most significant bits of RX are selected from the BCD sums S^H, $\neg S^H$ and LS^H using a level of 3:1 muxes, as explained in Section 5.2. Since for a fast implementation of rounding we have assumed that, for subtraction, the sign of $(U-L)$ is positive ($C_{out} = 1$), then, when $cmp = eop\ \overline{C_{out}}$ is true, the value of inc does not care. Therefore the decoded selection signals for the 3:1 muxes are cmp, $(inc\ \overline{cmp})$ and $(\overline{inc}\ \overline{cmp})$, and must be buffered due to the high load.

By other hand, to obtain the BCD sums S^H and LS^H, a +6 is subtracted digitwise from digits S_i^* and LS_i^*. The blocks labeled as '-6' in Fig. 5.9 implement this digit subtraction as

$$\begin{aligned} s_{i,3}^H &= s_{i,3}^*\ s_{i,2}^*\ s_{i,1}^* \\ s_{i,2}^H &= s_{i,2}^* \oplus s_{i,1}^* \\ s_{i,1}^H &= \overline{s_{i,1}^*} \\ s_{i,0}^H &= s_{i,0}^* \end{aligned} \tag{5.32}$$

The BCD sum $\neg S^H$ is obtained inverting the bits of S^*. The lsb of RX is obtained as $\overline{S_{lsb}^D} cmp \vee RX_{lsb}\overline{cmp}$ in a 2:1 mux. Finally, the guard digit RX_{GD} is concatenated to the right of the lsb to form the rounded $p+1$-digit significand RX.

5.3.1 Direct implementation of decimal rounding

We propose two different implementations for the rounding unit. The first implementation is shown in Fig. 5.11. It uses combinational logic to implement directly a rounding condition for each decimal rounding mode (Rnd mode). Moreover, different conditions must be applied when the MSD of the significand result (before rounding) is zero or not zero.

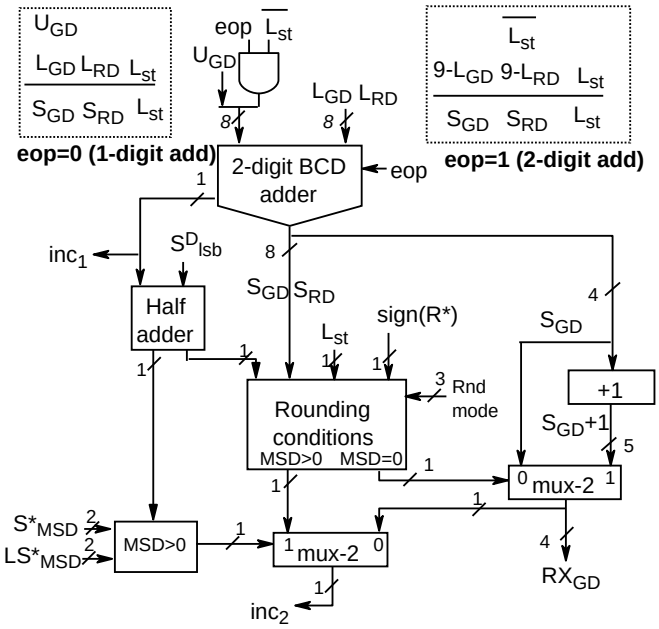


Figure 5.11. *Diagram of direct decimal rounding.*

To simplify this logic, the guard and round digits of U and L are previously assimilated in a 2-digit BCD adder, producing two sum digits S_{GD}, S_{RD} and a decimal carry-out inc_1. This bit represents a +1 ulp increment of $S = U^H + L^H$. The layout of the input operands differs if the effective operation is addition ($eop = 0$, top left corner of Fig. 5.11) or subtraction ($eop = 1$, top right corner of Fig. 5.11). For $eop = 0$, inc_1 and S_{GD} are the result of the 1-digit BCD addition $U_{GD} + LGD$, while $S_{RD} = L_{RD}$. For $eop = 1$, the addition of $\neg(L_{GD}\ 10 + L_{RD})$ and the bit $\overline{L_{st}}$ produces two BCD sum digits S_{GD}, S_{RD}, while the decimal carry-out inc_1 is 1 only if $L_{GD} = L_{RD} = L_{st} = 0$. Therefore, the logic of the 2-digit BCD adder can be simplified in order to obtain inc_1 faster.

In addition to the five IEEE 754-2008 decimal rounding modes (see Section 2.1), we implement two additional rounding modes: round to nearest down and away from zero. The conditions for each decimal rounding mode are summarized in Table 5.3, where $sign(R^*)$ is the expected sign of the rounded result, given by expression (5.24) and $s_{GD,0}$ is the lsb of S_{GD}. Depending on the MSD of the unrounded result, the roundoff may produce an increment of +1 unit in S_{LSD} or S_{GD}. The first case ($MSD > 0$) corresponds with a +1 ulp increment of S (inc_2). In the second case ($MSD = 0$), inc_2 is true if the corresponding rounding condition is verified and $S_{GD} = 9$. To obtain RX_{GD} and inc_2 faster, we compute $S_{GD}+1$ (module 10) in parallel with the rounding conditions, and then S_{GD} or $S_{GD}+1$ is selected from the corresponding rounding condition for $MSD = 0$.

The condition $MSD > 0$ occurs when $S^H_{MSD} > 0$ or $S^H = 099\ldots98$ and $S^D_{lsb} + inc_1 = 2$. This is determined from S^*_{MSD} and LS^*_{MSD} (coded in BCD excess-6) as

$$MSD > 0 \Longleftrightarrow (S^*_{MSD} > 6) \vee (LS^*_{MSD} > 6)\ (inc_1\ S^D_{lsb}) \tag{5.33}$$

The comparisons with 6 are implemented examining if the msb or the lsb of S^*_{MSD} (or LS^*_{MSD}) are '1'. The critical path goes from LS^*_{MSD} to inc_2 (4 gate delays).

Rnd mode	$MSD > 0$	Rounding condition
To nearest even	0	$S_{RD} > 5 \vee (S_{RD} = 5\ (s_{GD,0} = 1 \vee L_{st} = 1))$
	1	$S_{GD} > 5 \vee (S_{GD} = 5\ ((S^D_{lsb} \oplus inc_1) = 1 \vee S_{RD} > 0 \vee L_{st} = 1))$
To nearest up	0	$S_{RD} \geq 5$
	1	$S_{GD} \geq 5$
To nearest down	0	$S_{RD} > 5 \vee (S_{RD} = 5\ st = 1)$
	1	$S_{GD} > 5 \vee (S_{GD} = 5\ (S_{RD} > 0 \vee L_{st} = 1))$
Toward $+\infty$	0	$sign(R^*) = 0\ (S_{RD} > 0 \vee (S_{RD} = 0\ L_{st} = 1))$
	1	$sign(R^*) = 0\ (S_{GD} > 0 \vee (S_{GD} = 0\ (S_{RD} > 0 \vee L_{st} = 1)))$
Toward $-\infty$	0	$sign(R^*) = 1\ (S_{RD} > 0 \vee (S_{RD} = 0\ L_{st} = 1))$
	1	$sign(R^*) = 1\ (S_{GD} > 0 \vee (S_{GD} = 0\ (S_{RD} > 0 \vee L_{st} = 1)))$
Toward zero	$\{0, 1\}$	0
Away from zero	0	$S_{RD} > 0\ \vee (S_{RD} = 0\ L_{st} = 1)$
	1	$S_{GD} > 0\ \vee (S_{GD} = 0\ (S_{RD} > 0 \vee L_{st} = 1))$

Table 5.3. *Conditions for the decimal rounding modes implemented.*

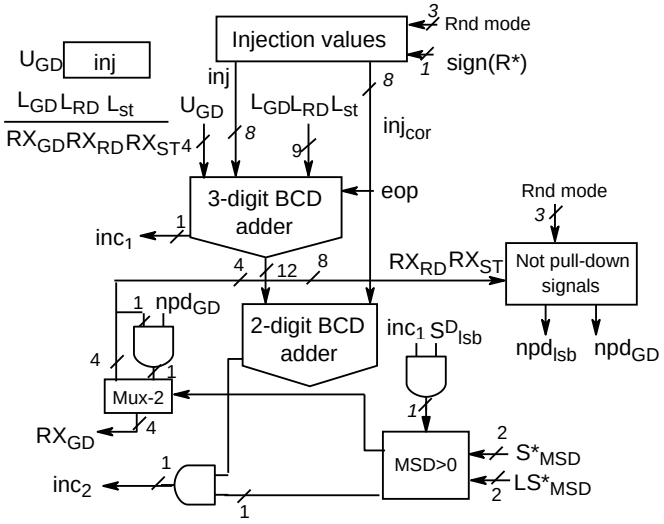


Figure 5.12. *Diagram of decimal rounding by injection.*

5.3.2 Decimal rounding by injection

The rounding unit of Fig. 5.12 reduces all the decimal rounding modes to truncation (rounding towards zero) by injecting an 2-digit value (inj) in the sticky and round digit positions. This allows the reuse of the same logic for the different rounding modes. The inj values injected for each decimal rounding mode (see Table 5.2) are obtained using a table look-up. Moreover, the expected sign of the rounded result $sign(R^*) = \overline{eop}\ sign(U)$ is also required to determine the injection for the rounding modes toward $\pm\infty$.

The injection is added to the guard and round digits of U and L and to the sticky bit L_{st} using a 3-digit BCD adder. This adder produces a 3-digit BCD sum operand (RX_{GD}, RX_{RD},RX_{ST}) and a carry-out inc_1 to the lsb of S. The layout of the input operands is shown in

the top left of Fig. 5.12. For effective subtraction ($eop = 1$), the operand (L_{RD}, L_{GD}, Lst) must be complemented.

However, as we have explained in Section 5.1.2, when the MSD of the addition $U+L+inj$ is not zero ($MSD > 0$), an 2-digit injection correction value inj_{cor} (see Table 5.2) must be added to (RX_{GD}, RX_{RD}) to compute the correctly rounded result RX. This second BCD adder produces a carry-out which represents an increment of an additional +1 ulp of S (inc_2) for $MSD > 0$. The signal $MSD > 0$ is implemented as stated in (5.33). Moreover, in the case of rounding to nearest even and $MSD = 0$, the lsb of RX_{GD} needs to be pulled down (R goes from odd to even) if $RX_{RD} = 5$ and $RX_{ST} = 0$ (signal $npd_{GD} = 0$), as shown in Fig. 5.12. If $MSD > 0$ and the rounding mode is nearest even, then the bit $S^D_{lsb} \oplus inc_1$ is pulled down when $RX_{GD} = 5$ and RX_{RD} and RX_{ST} are zeroes. This is done in the increment stage placing an additional AND gate inside the full adder, that is, $(S^D_{lsb} \oplus inc_1)\ npd_{lsb}$.

The critical path goes from LS^*_{MSD} to inc_2. Thus, it is one gate delay less than in the previous implementation (3 gate delays).

5.4 Evaluation results and comparison

In this Section we analyze the area and the delay of our DFP adder using the evaluation model for CMOS technology of Appendix A. In particular, we focus on the proposed significand BCD adder with decimal rounding for the IEEE-754 Decimal64 (16 precision digits) and Decimal128 (34 precision digits) formats. We compare this results with the two previous high-performance designs described in Section 5.1 [136, 157] and with a binary implementation for IEEE 754-1985 double precision (53-bits) [125].

5.4.1 Evaluation results

The general block diagram of our architecture is depicted in Fig. 5.9. The critical path is indicated as a discontinuous thick line. In terms of logic levels (XOR gate=2 logic levels) the critical path goes through 4+2 levels of the pre-correction stage, $4+\log_2(4p)$ levels of the binary compound adder, 3 levels of the rounding logic, 1 level in the increment stage, and though a chain of buffers (with a load of $4p$ muxes) plus 2 levels of the selection stage. The total number of levels (excluding the buffering) is 22 for Decimal64 ($p = 16$) and 24 for ($p = 34$).

In Table 5.4 we present the delay (in # FO4) and area (in # NAND2 gates) estimated for the IEEE-754 Decimal64 ($p = 16$) and Decimal128 ($p = 34$) implementations of the architecture proposed in Fig. 5.9.

The delay figures correspond with the critical path delay of each stage. The area and critical path delay of the rounding unit corresponds with the injection scheme of Fig. 5.12. For the binary compound adder, the figures correspond with a 63-bit Kogge-Stone prefix tree adder. Note that an important contribution to the delay of the selection stage is due to the buffering chain. In addition, most of the hardware cost comes from the pre-correction stage. For instance, the implementation of G_i costs about 33 NAND2 gates per digit, while the cost of the block $+6\ G_i$ is about 18 NAND2 gates per digit. However, this simplifies significatively the logic required to merge rounding with the significand addition and the decimal post-correction

Stage	Delay (# FO4)	Area (Nand2)
IEEE 754-2008 Decimal64		
Pre-correction	6.5	1010
63-bit binary compound adder	10.5	1800
Rounding logic	2.5	200
Increment logic	1.5	10
Selection	5.1	560
Total	**26.7**	**3580**
IEEE 754-2008 Decimal128		
Pre-correction	6.5	2140
135-bit bin. compound adder	12.7	4300
Rounding logic	2.5	200
Increment logic	1.5	10
Selection	5.9	1150
Total	**29.1**	**7800**

Table 5.4. *Delay-area figures for the significand BCD adder with rounding.*

of the binary sum.

5.4.2 Comparison

We first examine the critical path of the implementations proposed in [136] (separate significand addition and rounding) and in [157] for merging significand BCD addition with decimal rounding. Although both architectures were originally proposed for the Decimal64 format (16 digits), we consider also a precision of $p = 34$ digits.

For the separate sign-magnitude BCD adder and decimal rounder [136] (Fig. 5.3), the critical path (the discontinuous thick line) have $7 + \log_2(4p + 4) + 8$ logic levels (significand BCD adder) plus $\log_2(4p + 3) + 6$ (rounding unit, including the intermediate R1/R2-shifter). In addition, there are two chains of buffers with a load of $4p$ muxes each one. For Decimal64, the number of logic levels is 34 and 37 for Decimal128.

For the combined adder and rounding unit of [157] (Fig. 5.4), the critical path (the discontinuous thick line) goes through 2 logic levels for the pre-correction, $3 + \log_2(4p + 12)$ levels of the binary compound adder, 8 logic levels of the post-correction unit, 3 levels of the rounding logic, a chain of buffers loading $4p$ muxes and 2 levels for the final selection. This gives a total delay of 28 logic levels for Decimal64 and 29 gate levels for Decimal128.

Using our model, we have evaluated the area and delay of this architecture for both Decimal64 and Decimal128 formats. Table 5.5 shows these results and presents the corresponding ratios with respect to our proposals. The high hardware cost of implementations [136, 157] is mainly due to the decimal post-correction unit (85 NAND2 gates per digit). Thus, for the proposed architecture we estimate improvements between 11% and 16% in delay and area reductions of 25% and 27% with respect to the architecture with the best performance [157].

Adder	Delay		Area	
	(# FO4)	Ratio	(Nand2)	Ratio
IEEE 754-2008 Decimal64				
Proposed	26.1	1.00	3580	1.00
Thompson& Karra& Schulte [136]	37.5	1.44	4300	1.20
Wang& Schulte [157]	30.3	1.16	4490	1.25
IEEE 754-2008 Decimal128				
Proposed	29.1	1.00	7800	1.00
Thompson& Karra& Schulte [136]	43.5	1.50	9700	1.24
Wang& Schulte [157]	32.2	1.11	9950	1.28

Table 5.5. *Comparison figures for significand adders with rounding.*

We also have analyzed the critical path delay of the significand adder and rounding unit of a double precision (53-bit) binary floating-point adder [125]. It has 18 logic levels in the critical path with an estimated delay of 20.4 FO4 (28% faster) and a hardware complexity of 2280 NAND2 gates (57% less area).

5.5 Conclusions

A new method and architecture for merging significand (sign-magnitude) BCD addition and decimal rounding was presented. This is of interest to improve the efficiency of high performance IEEE 754-2008 DFP units, namely, DFP adders and multipliers. The IEEE 754-2008 Decimal64 (16 precision digits) and Decimal128 (34 precision digits) implementations present speedups of about 15% and 10% in performance while reduce the area of significand computation and rounding more than 25% with respect to a previous representative high-performance DFP adder [157]. Furthermore, the performance improvements are over 45% in the case of the DFP adder with separate significand addition and rounding [136] while the area is still reduced about 20%. With respect to the 53-bit binary floating-point adder, the significand BCD computation and decimal rounding for 16 precision digits is 20% slower and requires 55% more area.

This significant reduction in area lie in a simplification of the logic for decimal post-correction and rounding. To compute sign-magnitude BCD addition using a binary compound adder, the architecture implements an algorithm based on a conditional +6 digit addition of the BCD input digits [144]. This simplifies the post-correction logic required to obtain the BCD sum digits from the binary sums. Decimal rounding of the BCD magnitude result requires the computation of the BCD sums S, $S+1$ and $S+2$. To incorporate the rounding in the significand BCD addition we provide support for an additional +2 ulp increment of the sum result. This is performed by a previous binary carry-save addition of the +6 biased input BCD digits, which frees a 1-bit room in the lsb and allows the reuse of the binary compound adder to compute also $S+2$.

Chapter 6

Multioperand Carry-Free Decimal Addition

Multioperand addition is used in several algorithms for multiplication, division and square-root. In the previous two Chapters we considered methods and hardware implementations for addition of two decimal BCD operands. However, the recursive application of these algorithms is not efficient for multioperand addition, due to the carry propagation (logarithmic delay) of each two-operand addition.

Thus, several carry-free addition algorithms were proposed to speedup the addition of q ($q > 2$) decimal operands, reducing the q operands to a two-operand in a time independent of the word length. These algorithms consider decimal operands coded in BCD. However, the resulting implementations are quite complex due to the inefficiency of BCD for representing decimal numbers.

In this Chapter we introduce a decimal multioperand carry-save addition algorithm [148] that uses unconventional (non BCD) decimal coded number systems. We further detail this technique and present the new improvements to reduce the latency of the previous designs which include: optimized digit recoders for the generation of 2^n multiples (and $\times 5$ multiples), decimal 3:2 and 4:2 CSAs (carry-save adders) and carry-free adders implemented by special designed bit-counters. Moreover, we detail a design methodology that combines all these techniques to obtain efficient multioperand decimal CSA trees with different area and delay tradeoffs.

The Chapter is organized as follows. Section 6.1 outlines the previous work on multioperand decimal addition. In Section 6.2 we introduce the proposed technique for fast decimal carry-save addition. In Section 6.3 we describe the implementation of decimal 3:2 and 4:2 carry-save adders based on this method. Section 6.4 introduces a set of decimal carry-free adders implemented with counters. Different decimal and combined binary/decimal CSA trees are proposed in Section 6.5. First, we present a basic scheme and then we introduce two new proposals, one optimized for area and the other for delay. Section 6.6 presents the area and delay figures estimated for the proposed 64-bit (16 decimal digits) decimal and combined binary/decimal CSA trees. We also compare these results with some other representative works in decimal multioperand addition. We finally summarize the main conclusions and contributions of this Chapter in Section 6.7.

6.1 Previous Work

Proposals to perform a carry-free addition of several BCD operands were first suggested in [94] (carry-save) and [132] (signed-digit). Recently, several techniques have been proposed that improve these previous works. In [51] a signed-digit (SD) decimal adder based on [132] is used. Redundant binary coded decimal (RBCD) adders [127] can also perform decimal carry-free additions using a SD representation of decimal digits ($\in [-7, 7]$).

Decimal carry-save addition methods use a two BCD word to represent sum and carry [81, 82, 94, 109] or a BCD sum word and a carry bit per digit [50, 91]. The first group implements decimal addition mixing binary CSAs with combinational logic for decimal correction. In [109] a scheme of two levels of 3:2 binary CSAs is used to add the partial products iteratively. Since it uses BCD to represent decimal digits, a digit addition of +6 or +12 (Modulo 16) is required to obtain the decimal carries and to correct the sum digit.

In order to reduce the contribution of the decimal corrections to the critical path, three different techniques for multioperand decimal carry-save addition were proposed in [81]. Two of them perform BCD corrections (+6 digit additions) using combinational logic and an array of binary carry-save adders (speculative adders), although a final correction is also required. A sequential decimal multiplier using these techniques is presented in [82]. It uses BCD invalid combinations (overloaded BCD representation) to simplify the sum digit logic. The other approach (non-speculative adder) uses a binary CSA tree followed by a single decimal correction. In the non-speculative adder, preliminary BCD sum digits are obtained using a level of 4-bit carry propagate adders after the binary CSA tree. Finally, decimal carry and sum digit corrections are determined from the preliminary sum digit and the carries passed to the next more significant digit position in the binary CSA tree[20]. Decimal correction is performed using combinational logic (its complexity depends on the number of input operands added) and a 3-bit carry propagate adder per digit. Among these proposals, the non-speculative adders present the best area-delay figures and are suited for tree topologies.

A recent proposal [41] uses a binary carry-free tree adder and a subsequent binary to BCD conversion to add up to N p-digit BCD operands. The binary carry-free adder consists of a network of full adders and a level of 4-bit CLAs that add the values for each decimal column separately. Then, each binary column sum is converted to BCD, obtaining, after the alignment of the resultant BCD column sum digits, three p-digit BCD operands for $11 < N \leq 111$. These three rows are finally reduced to a two operand BCD word by applying the same algorithm (in this case, the network of full adders is just a 3:2 CSA).

The second group of methods [50, 91] uses different topologies of 4-bit radix-10 carry-propagate adders to implement decimal carry-save addition. Each digit adder takes two BCD digits and a 1-bit carry input and generates a 1-bit carry output and the BCD sum digit. This addition can be implemented using a direct decimal carry lookahead adder (CLA) [118]. In [50] a serial multiplier is implemented using an array of radix-10 CLAs. A CSA tree using these radix-10 CLAs is implemented in the decimal parallel multiplier proposed in [91]. To optimize the partial product reduction, they also use an array of decimal digit counters. Each counter adds 8 decimal carries of the same weight and produces a BCD digit.

[20] A +6 must be added each time a carry is passed to the next more significant digit position.

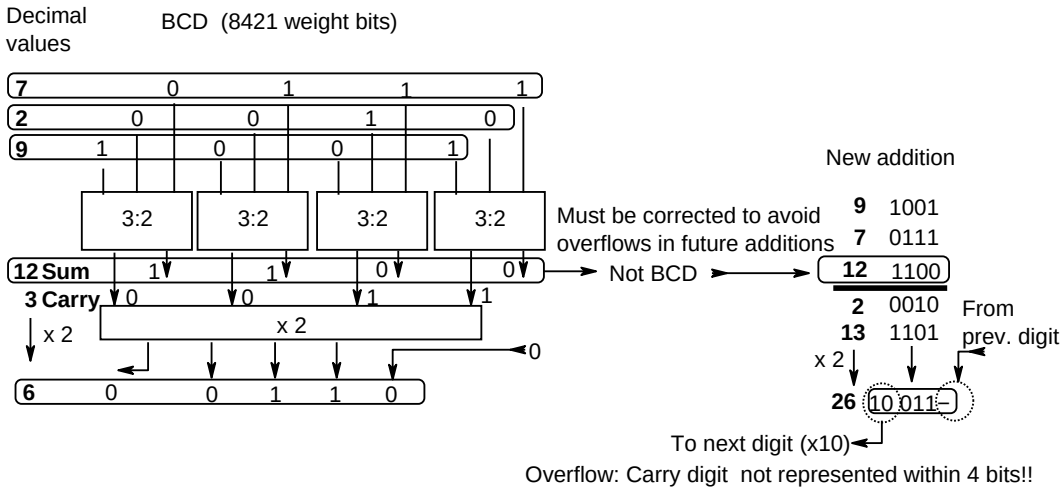


Figure 6.1. *BCD carry-save addition using a 4-bit 3:2 CSA.*

The addition of all decimal operands in parallel requires the use of efficient multioperand decimal tree adders. Among the different schemes, the most promising ones for fast parallel addition seem to be those using binary CSA trees or some parallel network of full adders [40, 81], due to their faster and simpler logic cells (full adders against SD adder cells or radix-10 CLAs). These methods assume that decimal digits are coded in BCD. However, BCD is highly inefficient for implementing decimal carry-save addition by means of binary arithmetic, because the need to correct the invalid 4-bit combinations (those not representing a decimal digit). Fig. 6.1 shows an example of the addition of 3 BCD digits using a 4-bit binary 3:2 CSA directly. In this case, the 4-bit representation (1100) of the decimal sum digit ('12') is an invalid BCD value and must be corrected to avoid overflows in subsequent BCD carry-save additions. The previous methods use different schemes to perform these BCD corrections. Moreover, the BCD carry digit must be multiplied by 2, which requires additional logic. We also implement multioperand decimal tree adders using a binary CSA tree, but with operands coded in decimal codings that are more efficient than BCD, as we show in the next Section. These multioperand decimal CSA trees are detailed in Section 6.5.

6.2 Proposed method for fast carry-save decimal addition

We propose the use of non-conventional (not BCD) decimal codings for an efficient implementation of decimal carry-save addition in a binary CSA tree or a full adder network. The use of these codes avoids the decimal corrections, so we only need to focus on the $\times 2$ decimal multiplications. We present first the set of preferred decimal codings, and next, the method for decimal carry-save addition.

Z_i	BCD	$Z_i(4221)$	$Z_i(5211)$	$Z_i(4311)$	$Z_i(3321)$
0	0000	0000	0000	0000	0000
1	0001	0001	0001 0010	0001 0010	0001
2	0010	0100 0010	0100 0011	0011	0010
3	0011	0101 0011	0101 0110	0100	0100 1000 0011
4	0100	0110 1000	0111	1000 0110 0101	1001 0101
5	0101	0111 1001	1000	1001 0111 1010	1010 0110
6	0110	1010 1100	1010 1001	1011	1100 1011 0111
7	0111	1011 1101	1011 1100	1100	1101
8	1000	1110	1110 1101	1110 1101	1110
9	1001	1111	1111	1111	1111

Table 6.1. *Decimal codings*

6.2.1 Alternative Decimal Digit Encodings

A decimal digit Z_i of an integer operand $Z = \sum_{i=0}^{p-1} Z_i \ 10^i$ is coded as a positive weighted 4-bit vector as

$$Z_i = \sum_{i=0}^{3} z_{i,j} \ r_j \tag{6.1}$$

where $Z_i \in [0, 9]$ is the i^{th} decimal digit, $z_{i,j}$ is the j^{th} bit of the i^{th} digit and $r_j \geq 1$ is the weight of the j^{th} bit. The previous expression represents a set of coded decimal number systems that includes BCD (with $r_j = 2^j$), shown in Table 6.1. We refer to these codes by the weight of each binary position as $(r_3 r_2 r_1 r_0)$. The 4-bit vector that represents the decimal value Z_i in a decimal code $(r_3 r_2 r_1 r_0)$ is denoted as $Z_i(r_3 r_2 r_1 r_0)$. Among all the possible decimal codes defined by expression (6.1), there is a subset more suitable for decimal carry-save addition. This family of coded decimal number systems verifies that the sum of their weight bits is nine, that is

$$\sum_{j=0}^{3} r_j = 9 \tag{6.2}$$

which includes the (4221), (5211), (4311) and (3321) codes, shown in Table 6.1. Some of these decimal codings are already known [163], but we use them in a different context, to design components for decimal carry-save arithmetic. Moreover, they are redundant codes, since two or more different 4-bit vectors may represent the same decimal digit. These codes have the following two properties:

- All the sixteen 4-bit vectors represent a decimal digit ($Z_i \in [0, 9]$). Therefore any bit-level logical operation (AND, OR, XOR,...) over the 4-bit vector representation of two or more input digits produces a 4-bit vector that represents a valid decimal digit (input and output digits represented in the same code).
- The 9's complement of a digit Z_i can be obtained by inverting their bits (as a 1's complement) since

$$9 - Z_i = \sum_{j=0}^{3} r_j - \sum_{j=0}^{3} z_{i,j} \ r_j = \sum_{j=0}^{3} (1 - z_{i,j}) \ r_j = \sum_{j=0}^{3} \overline{z_{i,j}} \ r_j \tag{6.3}$$

Negative operands can be obtained by the 2's complement of the bit vector representation, that is

$$-Z(r_3r_2r_1r_0) = \overline{Z(r_3r_2r_1r_0)} + 1 \tag{6.4}$$

Next, we show how these codes can be used to improve multioperand decimal carry-save addition/subtraction using these two properties.

6.2.2 Algorithm

Using the first property of these alternative decimal codings, we perform fast decimal carry-save addition using a conventional 4-bit binary 3:2 CSA as

$$A_i + B_i + C_i = \sum_{j=0}^{3}(a_{i,j} + b_{i,j} + c_{i,j})\ r_j = \sum_{j=0}^{3}(s_{i,j} + 2h_{i,j})\ r_j$$
$$= \sum_{j=0}^{3} s_{i,j}\ r_j + 2\ \sum_{j=0}^{3} h_{i,j}\ r_j = S_i + 2\ H_i$$

with $(r_3r_2r_1r_0) \in \Big\{(4221), (5211), (4311), (3321)\Big\}$, $s_{i,j}$ and $h_{i,j}$ are the sum and carry bit of a full adder and $H_i \in [0, 9]$ and $S_i \in [0, 9]$ are the decimal carry and sum digits at position i. No decimal correction is required because the 4-bit vector expressions of H_i and S_i represent valid decimal digits in the selected coding.

However, a decimal multiplication by 2 is required before using the carry digit H_i for later computations. Here we restrict the analysis of decimal carry-save addition to only (5211) and (4221) decimal codes, since the generation of multiples of 2 for operands coded in (4311) and (3321) seems, in principle, more complex. To simplify the implementation of the $\times 2$ computation we use the following property,

$$2Z(4221) = 2 \times Z(4221) = \mathbf{L1b}[\mathbf{Z}(\mathbf{5211})] \tag{6.5}$$

where $L1_{shift}$ denotes a 1-bit wired left shift. The resultant bit vector after shifting one bit to the left an operand Z coded in (5211) represents the double of the operand value ($2Z$) but coded in (4221), since their weight bits are multiplied by 2. Fig. 6.2 shows an example of $\times 2$ multiplication for decimal operands represented in (4221) and (5211) decimal codes. For a decimal operand Z represented by a bit vector $Z(4221)$, a $\times 2$ multiplication can be performed by a digit recoding of $Z_i(4221)$ into $Z_i(5211)$ followed by a $\mathbf{L1b}[Z(5211)]$. The resultant bit vector represents $2 \times Z$ coded in (4221). If the decimal operand Z is represented by $Z(5211)$, $2Z$ is obtained as $L1_{shift}[Z(5211)]$ but coded in (4221). To obtain $2Z(5211)$ a digit recoding from $2Z(4221)$ to $2Z(5211)$ is required.

To subtract a decimal operand coded in (4221) or (5211) using a carry-save adder, we first invert the bits of the operand and add one ulp (unit in the last place). This ulp can be placed in the free room at the least significant bit position that results from the left shift of the carry operand H.

In the following Sections, we describe how to design decimal CSAs of any number of input operands coded in (4221) or (5211). We first detail the implementation of decimal 3:2 and 4:2 CSAs using the proposed method. Next, in Section 6.4, we present a different family

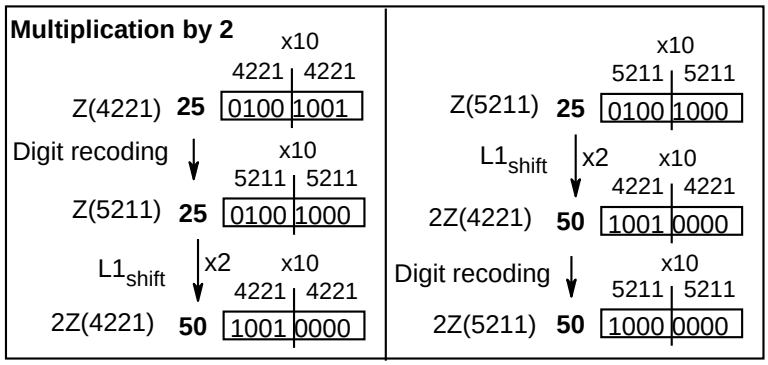


Figure 6.2. *Calculation of $\times 2$ for decimal operands coded in (4221) and (5211).*

of decimal carry-free adders made up of bit counters. Finally, in Section 6.5 we use these components as the building blocks of decimal and combined binary/decimal CSA trees of an arbitrary number of input operands.

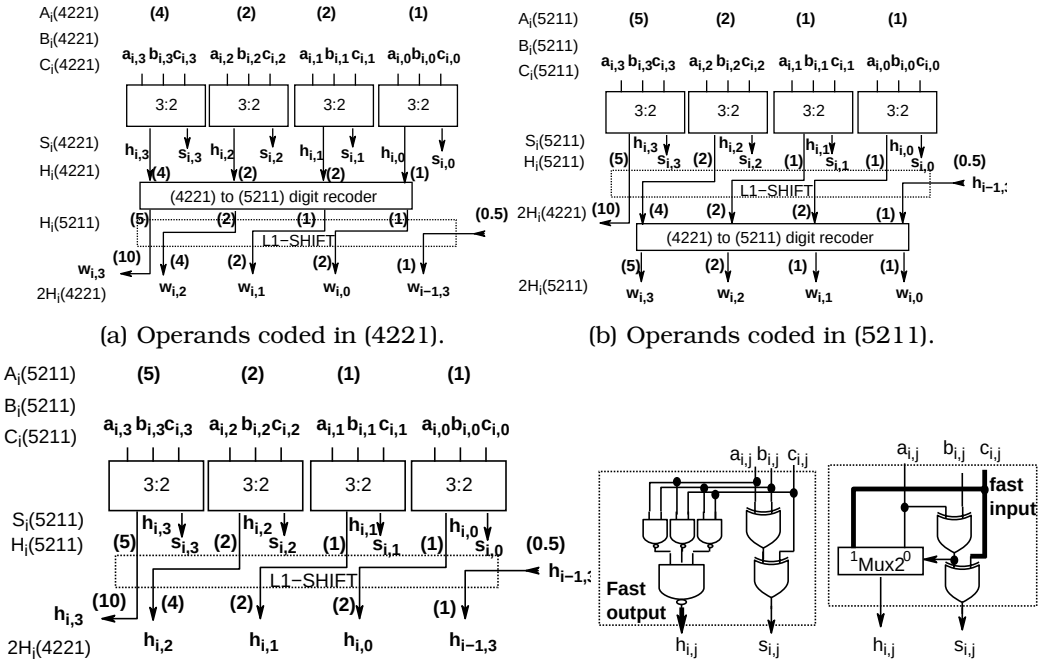
6.3 Decimal 3:2 and 4:2 CSAs

In this Section we detail the proposed implementations of a decimal 3:2 and 4:2 CSAs. We also describe the gate level implementation of the digit recoders required to perform conversions between different decimal codings. These recoders are the core logic components to compute $\times 2^n$ multiplications, which are also required for partial product generation in multiplication.

6.3.1 Gate level implementation

The proposed decimal 3:2 CSAs adds three decimal operands (A,B,C) coded in (4221) or (5211) and produce a decimal sum word (S) and a carry word (H) multiplied by 2 ($2 \times H$) coded in (4221) or (5211), such that $A + B + C = S + 2H$. Depending on the decimal coding of the operands, we have three possible implementations of a decimal digit 3:2 CSA using a 4-bit binary 3:2 CSA, as shown in Fig. 6.3:

- Input operands and output operands (S, H, $2H$) coded in (4221) (Fig. 6.3(a)). The weight bits in Fig. 6.3 are placed in brackets above each bit column. In this case, the decimal digit 3:2 CSA consists of a 4-bit binary 3:2 CSA and a digit recoder from (4221) to (5211). In Fig. 6.3 we show two gate level implementations of a 1-bit 3:2 CSA: one with a fast carry output (Fig. 6.3(d)) and one with a fast input (Fig. 6.3(e)). The output of the digit recoder ($H_i(5211)$) is then left shifted by one bit position ($L1_{shift}[H_i(5211)]$). The recoder is placed in the carry path, so choosing an appropriate gate implementation of the binary 3:2 CSA, in this case the fast carry output configuration (Fig. 6.3(d)), part of the recoder delay can be hidden.
- Input and output operands coded in (5211) (Fig. 6.3(b)). The implementation of the (5211) decimal digit 3:2 CSA is similar to the (4221) case, except that here the 4-bit carry vector $H_i(5211)$ is 1-bit left shifted before the digit recoding.



(a) Operands coded in (4221). (b) Operands coded in (5211).

(c) Mixed (5211)/(4221) coded output operands. (d) Full adder with fast carry output. (e) Full adder with fast input.

Figure 6.3. *Proposed decimal digit (4-bit) 3:2 CSAs.*

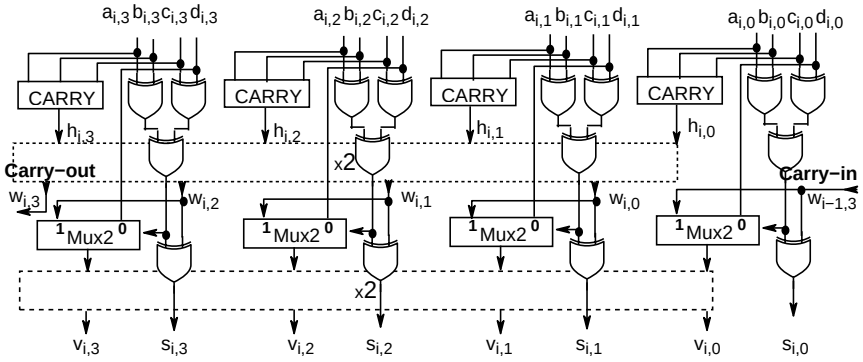
- Input operands coded in (5211), S, H coded in (5211) but $2H$ coded in (4221) (Fig. 6.3(c)). The decimal digit 3:2 CSA consists only of a level of 4-bit 3:2 CSA with the carry output shifted 1-bit to the left.

The gate level implementation of two decimal 4:2 CSAs for input and output operands coded in (4221) is shown in Fig. 6.4. The first decimal 4:2 CSA (Fig. 6.4(a)) uses a specialized gate configuration. The carry bit-vector H is computed as in binary from operands A, B and C coded in (4221). The intermediate decimal carry operand W is then obtained as $2 \times H$. The sum operand S (coded in (4221)) is obtained by XOR-ing the bits of A, B, C, D and W (approximately in 4 XOR gate delays). The decimal carry operand V is obtained (approximately in 6 XOR gate delays) by selecting the appropriate bits of D or W, depending on the xor of A, B, C and D, and multiplying the resulting bit vector (coded in (4221)) by 2.

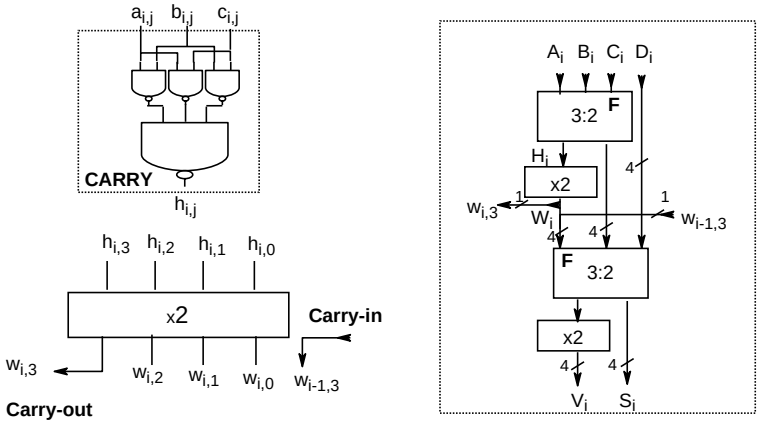
The second decimal 4:2 CSA (Fig. 6.4(b)) is designed by interconnecting two decimal 3:2 CSAs (Fig. 6.3(a)). The blocks labeled as 3:2 represent a 4-bit binary 3:2 CSA. The intermediate decimal carry W is connected to a fast input of the second full adder (indicated by a letter **F** in Fig. 6.4(b)) to reduce the delay of the critical path. Thus, both implementations present a similar critical path delay (6 XOR gate delays in the carry path).

6.3.2 Implementation of digit recoders

The design of efficient digit recoders is a critical issue, due to their high impact on the performance and area of a decimal multiplier. Due to the redundancy of (4221) and (5211) decimal



(a) Using a specialized gate configuration.



(b) Using two decimal $3:2$ CSAs.

Figure 6.4. *Proposed decimal (1-digit slice) 4:2 CSAs.*

codes, there are many choices for the digit recoding between (4221) and (5211). The sixteen 4-bit vectors of a coding can be mapped (recoded) into different subsets of 4-bit vectors of the other decimal coding representing the same decimal digit. These subsets of the (4221) and (5211) codes are also decimal codings. Among all the subsets analyzed, we have selected the non-redundant decimal codes (subsets of ten 4-bit vectors) shown in Table 6.2 to represent the recoded digits. These codes lead to two different configurations of digit recoders ($S1$ and $S2$) for the recoding from (4221) to (5211):

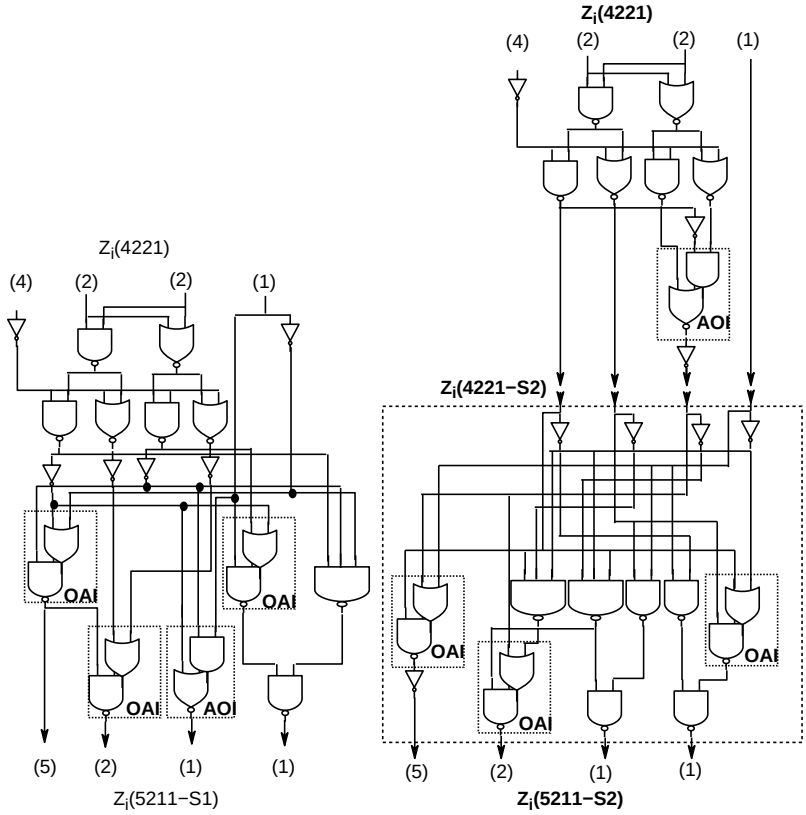
- The first group of codes, $S1 = \{(4221 - S1), (5211 - S1)\}$ leads to a simpler implementation of a digit recoder when all the sixteen 4-bit input combinations are possible. Therefore, in general, a $\times 2$ block is implemented by digit recoding $Z(4221)$ into $Z(5211 - S1)$ and shifting the output one bit to the left. The gate level implementation of a S1 digit recoder is shown in Fig. 6.5(a). This operation can be seen as a two-step digit recoding of $Z_i(4221)$ to $Z_i(4221-S1)$ and $Z_i(4221-S1)$ into $Z_i(5211-S1)$. The digit recoding between $Z_i(4221-S1)$ and $Z_i(5211 - S1)$ is very simple, since the 4-bit vectors representing each decimal digit value in both decimal codes are almost similar.

- The second group of codes, $S2 = \{(4221 - S2), (5211 - S2)\}$ verifies

$$2Z(4221 - S2) = L1_{shift}[Z(5211 - S2)] \tag{6.6}$$

Z_i	$Z_i(4221-S1)$	$Z_i(5211-S1)$	$Z_i(4221-S2)$	$Z_i(5211-S2)$
0	0000	0000	0000	0000
1	0001	0001	0001	0001
2	0100	0100	0010	0100
3	0101	0101	0011	0101
4	0110	0111	1000	0111
5	1001	1000	1001	1000
6	1010	1010	1010	1001
7	1011	1011	1011	1100
8	1110	1110	1110	1101
9	1111	1111	1111	1111

Table 6.2. *Selected decimal codes for the recoded digits.*



(a) S1 digit recoder. (b) S2 digit recoder.

Figure 6.5. *Gate level implementation of the (4221) to (5211) digit recoders.*

that is, after shifting one bit to the left an operand represented in $(5211-S2)$, the resultant digits are represented in $(4221-S2)$. This fact simplifies the implementation of $\times 2^n$ operations with $|n| > 1$. Specifically, $2^n \times Z$ can be implemented recoding each digit $Z_i(4221)$ to $Z_i(4221-S2)$ followed by n stages of $Z_i(4221-S2)$ to $Z_i(5211-S2)$ digit recoders. The implementation of this S2 digit recoder is shown in Fig. 6.5(b) (the $Z_i(4211-S2)$ to

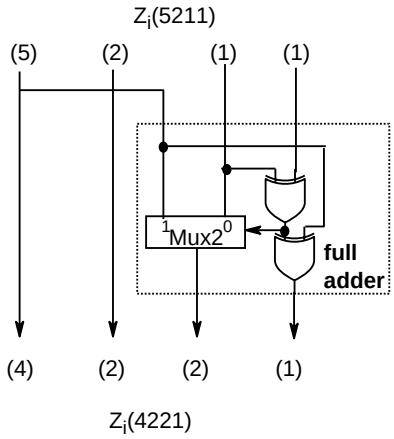


Figure 6.6. *Implementation of a (5211) to (4221) digit recoder.*

$Z_i(5211-S2)$ recoder is shown inside the dashed line box). Moreover, when input digits into a 4-bit binary 3:2 CSA are coded in a S2 decimal coding then the resultant carry digit H_i is represented in the same S2 coding. In this case, $2\times H$ is implemented as a row of the simpler $H_i(4211-S2)$ to $H_i(5211-S2)$ digit recoders with outputs or inputs 1-bit left shifted.

Additionally, the inverse digit recoding (from (5211) to (4221)) is easily implemented using a single full adder as shown in Fig. 6.6, since

$$Z_i(5211) = z_{i,3}\ (4+1) + z_{i,2}\ 2 + z_{i,1} + z_{i,0} = z_{i,3}\ 4 + z_{i,2}\ 2 + z^*_{i,1}\ 2 + z^*_{i,0} \tag{6.7}$$

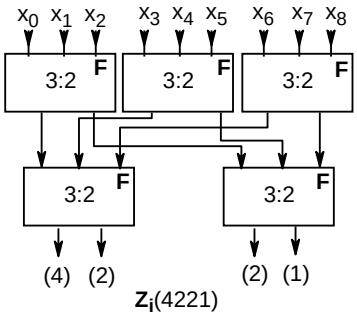
with $z^*_{i,1}\ 2 + z^*_{i,0} = (z_{i,3} + z_{i,1} + z_{i,0}) \leq 3$. This recoder is used in mixed (4221/5211) multioperand CSAs to convert a (5211) decimal coded operand into the equivalent (4221) coded one.

6.4 Decimal carry-free adders based on reduction of bit columns

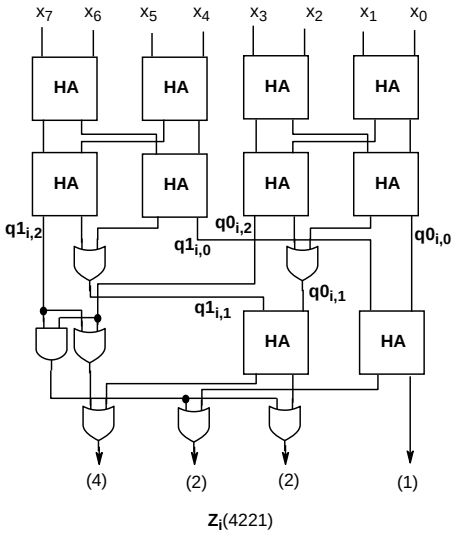
We have designed efficient 9:4, 8:4 and 7:3 decimal carry-free adders based on reduction by bit columns for decimal operands coded in (4221) or (5211). These adders consists of a row of bit counters which add a column of 9, 8 or 7 bits and produce a decimal digit coded in (4221). Next, we detail the gate level implementation of these bit counters and the architecture of the proposed carry-free decimal adders.

6.4.1 Bit counters

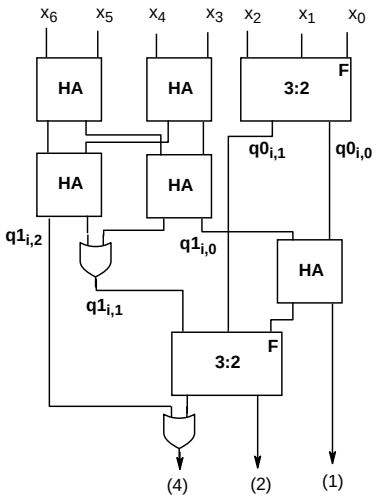
The proposed bit counters sum a column of up to $q = 9$ bits (same weight) and produce a t-bit vector ($t = \lceil \log_2(q+1) \rceil \leq 4$) with weights (4221) which represents a decimal digit $Z_i \in [0, 9]$. In Fig. 6.7(a) we show an implementation of a (4221) decimal counter that adds up to 9 bits using two levels of binary full adders (an **F** indicates the fast input). The binary weight of each output is indicated in brackets. The path delay varies approximately from 2 to 4 XOR gate delays for output (1), from 2 to 3 XORs for outputs (2) and is about 2 XORs for output (4). The



(a) (4221) counter for 9 bits.



(b) (4221) counter for 8 bits.



(c) 7:3 binary counter.

Figure 6.7. *Gate level implementation of digit counters*

8-bit counter of Fig. 6.7(b) only sums up to 8 bits but presents a similar critical path delay as a binary 4:2 CSA (3 XOR gate delay for output (1)). Basically, the first two levels of half adders (HA) and the two OR gates perform the computation

$$Q0_i = q0_{i,2}\ 4 + q0_{i,1}\ 2 + q0_{i,0}\ 1 = \sum_{k=0}^{3} x_k$$
$$Q1_i = q1_{i,2}\ 4 + q1_{i,1}\ 2 + q1_{i,0}\ 1 = \sum_{k=4}^{7} x_k \tag{6.8}$$

Since $Q0_i, Q1_i \in [0,4]$, the total sum $Z_i(4221) = Q1_i + Q0_i \in [0,8]$ is implemented in a simple way in the final logic level of Fig. 6.7(b) as

$$Z_i(4221) = \begin{cases} z_{i,3} = & q1_{i,2}\ q0_{i,2} \vee q1_{i,1}\ q0_{i,1} \\ z_{i,2} = & q1_{i,2}\ q0_{i,2} \vee q1_{i,0}\ q0_{i,0} \\ z_{i,1} = & q1_{i,2}\ q0_{i,2} \vee (q1_{i,1} \oplus q0_{i,1}) \\ z_{i,0} = & q1_{i,0} \oplus q0_{i,0} \end{cases}$$

In addition, a conventional $7:3$ binary counter, shown in Fig. 6.7(c), reduces a column of 7 bits into a 3-bit vector with weights (421). Other high-performance implementations of $7:3$

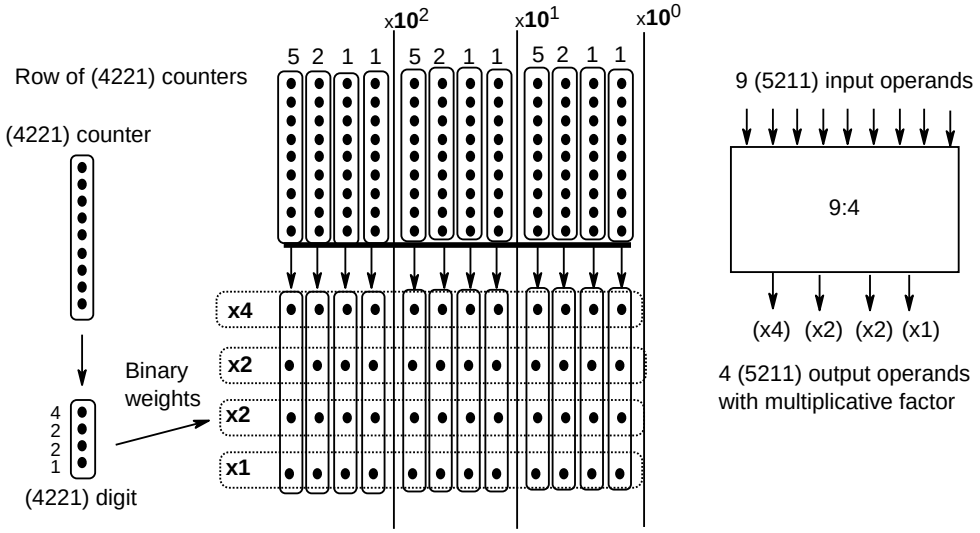


Figure 6.8. *9:4 reduction of (5211) decimal coded operands.*

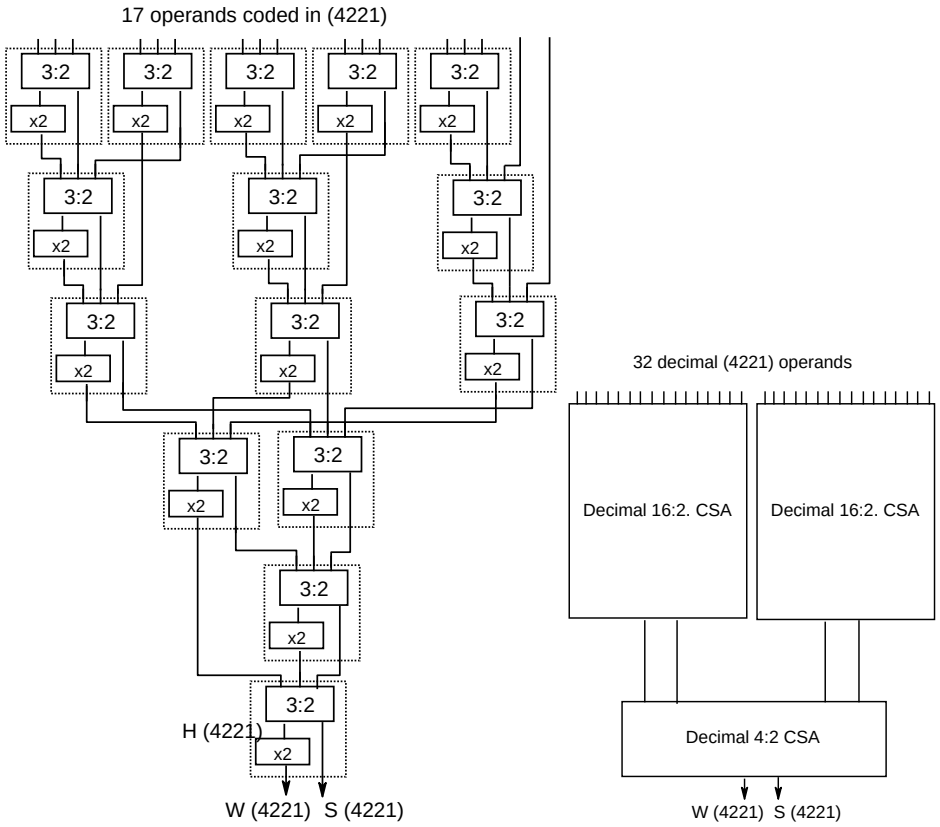
bit counters can be found in [48, 143]

6.4.2 Architecture

These counters can be used to reduce 9 or 8 p-digit decimal operands (coded in (4221) or (5211)) into 4 or 7 p-digit operands into 3. An example of this procedure is described in Fig. 6.8 for nine input operands coded in (5211). The procedure is similar for (4221) input operands. A row of p (4221) decimal counters of Fig. 6.7(a) sums the values of each bit column producing a (4221) digit per bit column. The four bits of each (4221) digit are placed in column from the most significant (top) to the least significant (bottom) and aligned in four rows according to the weight bit of their column ($5 \times 10^i, 2 \times 10^i, 1 \times 10^i, 1 \times 10^i$). This generates 4 decimal operands coded in (5211) that must be multiplied by a different factor (given by the binary weights of (4221)) before being added together. This organization causes all the bits of an output operand to have the same latency. The $\times 2$ and $\times 4$ operations can be performed as described in Section 6.3.2. The 9:4 decimal carry-free adder is represented by the labeled box of Fig. 6.8 where the multiplicative factor of each output is indicated in brackets. The 8:4 and 7:3 decimal carry-free adders are implemented using a row of counters of Fig. 6.7(b) and Fig. 6.7(c) respectively. By other hand, the 3:2 decimal CSA presented in Section 6.3.1 is a 3:2 decimal carry-free adder implemented using a row of 3-bit counters (a level of full adders) with the carry output multiplied by a factor $\times 2$.

6.5 Decimal and combined binary/decimal CSA trees

A $q : 2$ decimal CSA tree reduces q ($q > 2$) p-digit input operands coded in (4221) or (5211) into two decimal operands H and S, as $\sum_{l=0}^{q-1} Z[l] = 2H + S = W + S$, with $Z[l] = \sum_{i=0}^{p-1} Z_i[l]\ 10^i$. A diversity of multioperand decimal CSA trees with different area-delay trade-offs can be im-



(a) Decimal 17:2 CSA. (b) Decimal 32:2 CSA.

Figure 6.9. *Basic implementation of decimal (4221) q:2 CSAs.*

plemented combining binary 3:2 CSAs, digit recoders and the q:t decimal carry-free adders of Section 6.4. Next, we present several examples of multioperand decimal CSAs: basic implementations, optimized for area and optimized for delay. For the examples, we use 17 and 32 input operands coded in (4221), since these values correspond with the maximum number of operands to be added in the different 16-digit BCD parallel multipliers proposed in Chapter 7. Though we consider input and output operands coded in (4221), the resulting implementations are similar for the case of all decimal operands coded in (5211). We also present an example of a mixed delay-optimized decimal 32:2 CSA tree, with half of the operands coded in (4221) and the other half coded in (5211). Finally, we show the design of a combined binary/decimal CSA tree.

6.5.1 Basic implementations

Fig. 6.9 shows two examples of a q:2 decimal CSA trees that reduce q=17 and q=32 rows of decimal digits to 2. The blocks labeled as 3:2 are a $4p$-bit binary 3:2 CSA. The blocks labeled $\times 2$ represent a row of (4221) to (5211) S1 digit recoders (Fig. 6.5(a)) with the inputs (for 5211 coded operands) or outputs (for 4221 coded operands) 1-bit left shifted.

In this basic implementation, a multioperand decimal CSA tree is built using the decimal

3:2 CSAs of Fig. 6.3(a) (binary 3:2 CSA + $\times 2$ block). Since the carry path of a decimal 3:2 CSA has more delay than the sum path (3 vs. 2 XOR delays), it is connected to a fast input of the next level of decimal 3:2 CSAs to reduce the critical path delay of the CSA tree. We also use the most suitable implementation of a full adder (Fig. 6.3(d) or Fig. 6.3(e)) in every particular case to minimize the critical path delay. In this way, the 17:2 decimal CSA tree of Fig. 6.9(a) consists of 6 levels of decimal 3:2 CSAs with a critical path delay of 15 and 17 XOR delays for the sum S and carry W respectively.

This methodology allows to design decimal CSAs of a higher number of input operands in a hierarchical way. For example, the 32:2 decimal CSA tree of Fig. 6.9(b), is built using two decimal 16:2 CSA trees (very similar to the 17:2 CSA tree of Fig. 6.9(a)) and a level of the decimal 4:2 CSAs of Fig. 6.4. The critical path delay of this 32:2 decimal CSA tree is of about 20 XOR delays for the sum S and of 22 XOR delays for the carry W operand.

6.5.2 Area-optimized implementations

The area-optimized implementations reduce the hardware complexity by adding operands with the same multiplicative factor, that is

$$2^n\ A + 2^n\ B + 2^n\ C = 2^n\ (A + B + C) = 2^n\ S + 2^{n+1}\ H \tag{6.9}$$

This simplifies the hardware complexity since the overall number of $\times 2$ operations is reduced.

In Fig. 6.10 we show the architecture of an area-optimized 17:2 decimal CSA tree. Each intermediate operand is associated with a multiplicative factor power of two. For instance, the carry bit-vectors of the first level of binary 3:2 CSAs, before being multiplied by 2, have a multiplicative factor of (2). We indicate in brackets, beside each interconnection, the multiplicative factor of the corresponding intermediate (4221) decimal coded operand. Thus, it may be necessary to perform $\times 2^n$ ($n > 1$) operations for some paths. The implementation of these blocks is detailed in Section 6.3.2. For example, the 17:2 decimal CSA tree of Fig. 6.10 also requires $\times 8$ and $\times 4$ blocks. Moreover, since the total number of $\times 2$ operations is reduced in each path, the critical path delay of this area-optimized 17:2 decimal CSA is less (14 XOR delays for the sum S and 16 XOR delays for the carry W) than the one in the basic scheme.

The proposed implementation for an area-optimized 32:2 decimal CSA tree is shown in Fig. 6.11. The $\times 16$ block consists of 4 levels of the digit recoders of Fig. 6.5(b). Thus, each path goes through 5 levels of digit recoders at most, instead of 8, as in the basic scheme (Fig. 6.9(b)). Therefore, the critical path delay is also reduced to 19 XOR delays for the sum S and to 21 XOR delays for the carry W.

6.5.3 Delay-optimized implementations

We combine some of the strategies used for basic and area-optimized implementations to obtain delay-optimized implementations. In addition, we make use of the decimal carry-free adders of Section 6.4 to improve the latency of the decimal q:2 CSA trees. We aim to minimize the critical path delay by balancing the delay of the different paths.

A delay-optimized decimal 17:2 CSA tree for operands coded in (4221) is shown in Fig.

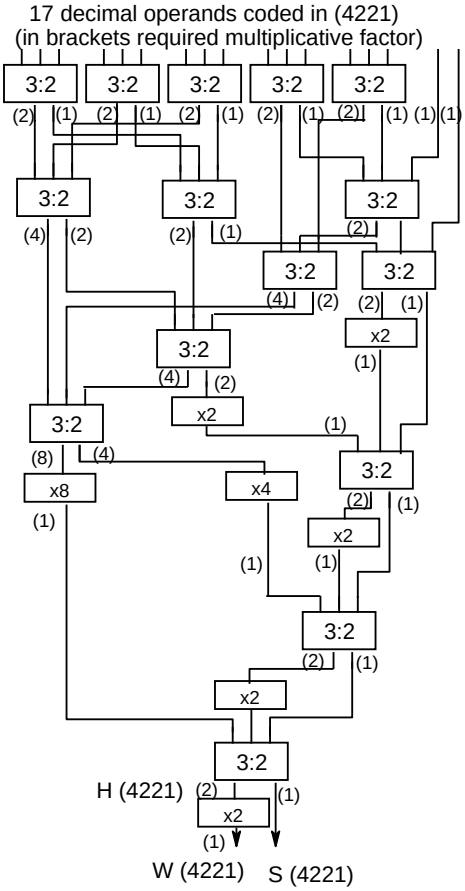


Figure 6.10. *Area-optimized implementation of a decimal (4221) 17:2 CSA.*

6.12. The 17 p-digit input operands $Z[l]$ are reduced in a first level of $\lfloor \frac{17}{k} \rfloor$ $k:t$ decimal carry-free adders and a decimal $mod_k(17):2$ CSA. The index $k \in \{7,8,9\}$ is set to keep $mod_k(17) \leq 4$ and balance the delay of the different paths. For 17 operands, a good choice is to use a $9:4$ and a $8:4$ decimal carry-free adders. The reduction of the resultant intermediate operands into H and S is performed by a decimal CSA tree. Each intermediate operand is associated with a multiplicative factor power of two. Operands may be multiplied by its factor before being added, or added in a binary 3:2 CSA with another two operands with the same factor. The critical path is reduced to 13 XOR delays for the sum path S and to 15 XOR delays for the carry path W.

Implementations of decimal CSA trees with operands in mixed (4221) and (5211) decimal codings may be used to speedup the reduction of partial products in decimal multiplication. For instance, the decimal SD radix-5 multiplier we introduce in Chapter 7 generates half of the decimal partial products coded in (5211) and the other half in (4221). In particular, the (5211) decimal coded operands are generated faster than the (4221) operands.

Fig. 6.13 shows an delay-optimized decimal 32:2 CSA with mixed coded operands: 16 input operands are coded in (5211) and the other half in (4221). For (5211) coded operands, $\times 2$ is implemented as a 1-bit wired left shift, with the result coded in (4221). The block

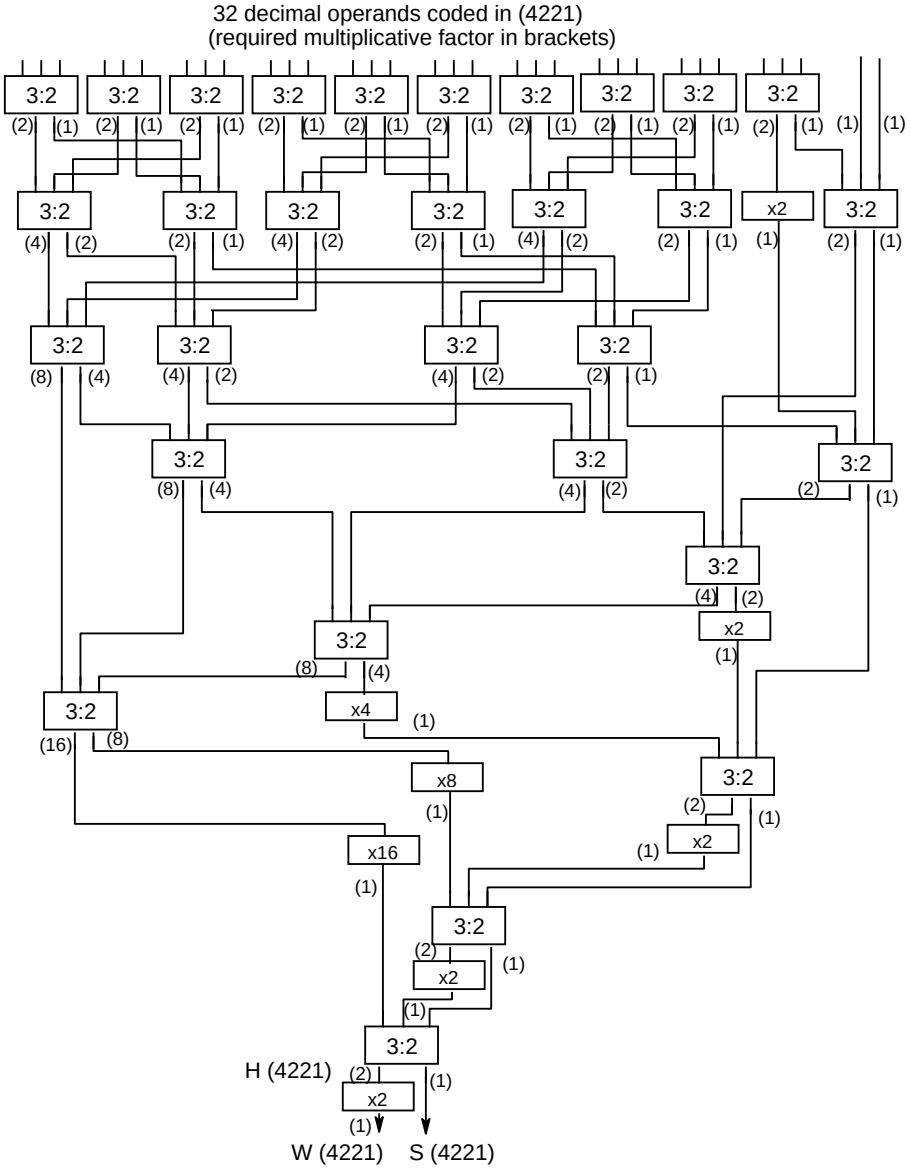


Figure 6.11. *Area-optimized implementation of a decimal (4221) 32:2 CSA.*

labeled 2:2 represents a row of half adders. A row of (5211) to (4221) digit recoders (Fig. 6.6) is required to reduce the remaining intermediate (5211) decimal coded operand with two (4221) coded operands in a binary 3:2 CSA.

We have used the same strategies as before to balance the delay of the different paths of the mixed decimal 32:2 CSA tree. Moreover, to speedup the evaluation, mixed (4221/5211) implementations use the fact that a 1-bit left shift (operation without delay) performed over an operand coded in (5211) is equivalent to multiplying it by $\times 2$ and recoding to (4221). Therefore, the critical path delay of this delay-optimized mixed (4221/5211) decimal 32:2 CSA is of about 18 XOR delays for the sum S and of 20 XOR delays for the carry W.

A different rearrangement of a decimal (4221) CSA tree optimized for delay was proposed

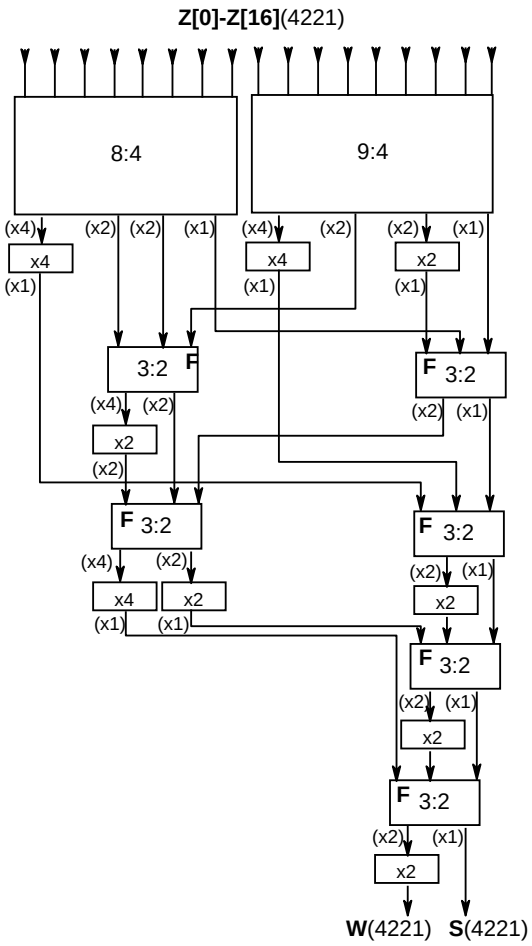


Figure 6.12. *Delay-optimized 17:2 decimal (4221) CSA tree.*

by Castellanos and Stine [23]. They use a regular structure of 4:2 decimal compressors (made of binary 3:2 CSAs and $\times 2$ blocks) to build 16:2 and 32:2 (4221) decimal CSAs. However, the resulting structures are of similar delay than our area-optimized proposals and have more area.

6.5.4 Combined binary/decimal implementations

The different decimal CSAs proposed in the previous Sections can be extended to support binary carry-save addition in a simple way. We assume decimal operands coded in (4221), but this is also valid for operands coded in (5211). An example of a combined binary/decimal 3:2 CSA is shown in Fig. 6.14. The only modification affects to the $\times 2$ block, detailed in Fig. 6.14(b). A 2:1 multiplexer controlled by a bit signal d_M ($d_M = 1$ for decimal operations) is used to select between the carry operand $W = 2 \times H$ coded in binary (W obtained as a 1-bit left shift of bit-vector $H = \sum_{i=0}^{p-1} H_i \ 16^i$) or in decimal (4221) code (W obtained as a digit recoding of $H = \sum_{i=0}^{p-1} H_i \ 10^i$ plus a 1-bit left shift). Note that for compatibility between the binary and decimal paths, the weight bits of decimal digits are reorganized as (2421), so $H_i = \sum_{j=0}^{3} h_{i,j} \ 2^j$

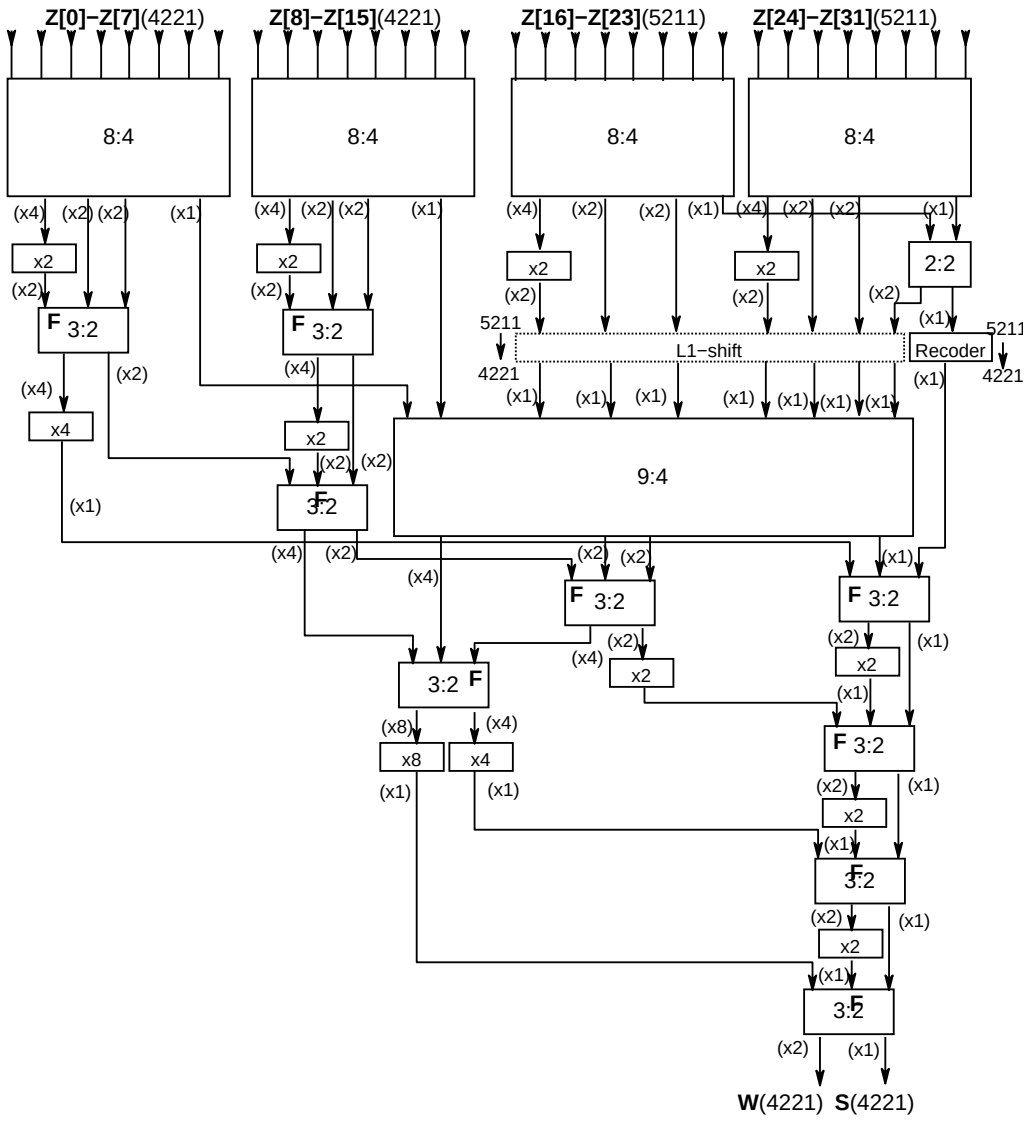


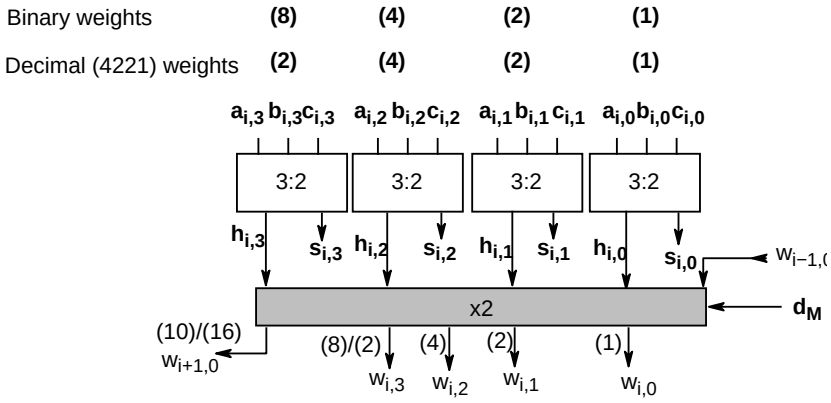
Figure 6.13. *Delay-optimized 32:2 decimal mixed (4221/5211) CSA tree.*

for binary and

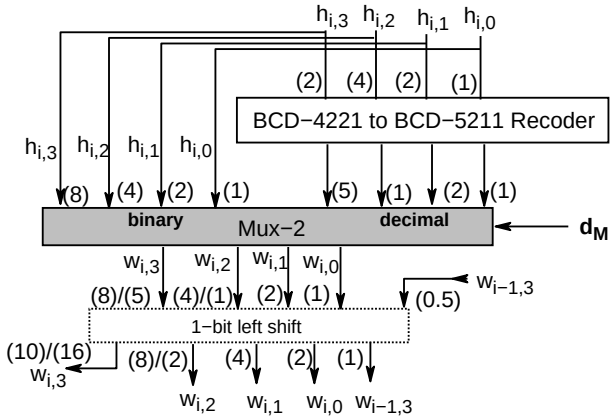
$$H_i = h_{i,3}\ 2 + h_{i,2}\ 4 + h_{i,1}\ 2 + h_{i,0}\ 1 \tag{6.10}$$

for decimal.

Hence, to transform the different decimal q:2 CSA trees into the equivalent combined binary/decimal CSA tree, we have to replace the $\times 2$ blocks with the implementation of Fig. 6.14(b). By other hand, the $\times 2^n$ ($n > 1$) blocks only require one level of 2:1 multiplexes, controlled by d_M, placed after the n levels of digit recoders. This scheme could also be applied to the delay-optimized implementations, but after introducing a slight modification, shown in Fig. 6.15, into the 9-bit and 8-bit counters of Section 6.4: the 4-bit vectors 1111 and 1110 representing the sum of columns of 9 and 8 bits must be replaced by 1001 and 1000 for binary operations.



(a) 4-bit (digit) slice.



(b) $\times 2$ block.

Figure 6.14. *Combined binary/decimal 3:2 CSA.*

6.6 Evaluation results and comparison

We present in this Section an estimation of the area and delay for the different 17:2 and 32:2 CSA trees shown in Section 6.5 using the evaluation model for CMOS technology described in Appendix A.

Also, we have analyzed several decimal carry-free tree adders for sixteen 64-bit (16-digit) operands based on different methods described in Section 6.2. We have evaluated and compared these implementations with our proposals. We show the results of these comparisons in Section 6.6.2.

6.6.1 Evaluation results

Table 6.3 shows the evaluation results for the proposed 17:2 and 32:2 CSA trees for 16-digit operands: the basic implementations (Fig. 6.9), the area-optimized implementations (Fig. 6.10 and Fig. 6.11) and the delay-optimized implementations (Fig. 6.12 and Fig. 6.13). We include the area and delay figures of the equivalent binary CSA trees for 64-bit operands and two basic implementations of combined binary/decimal 17:2 and 32:2 CSA trees. The 17:2

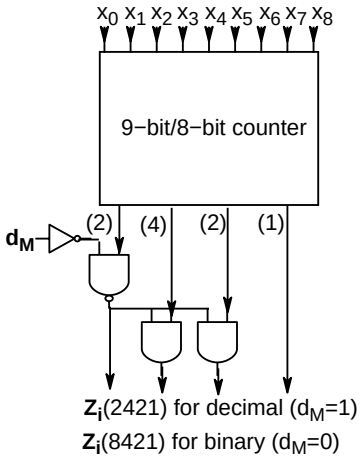


Figure 6.15. *Combined binary/decimal carry-free adders.*

Architecture	Delay (t_{FO4})	Area (NAND2)	Delay Ratio	Area Ratio
Bin. 17:2 CSA	21.0	10800	1	1
Basic Dec. 17:2 CSA	34.0	14500	1.60	1.35
Area-Opt. Dec. 17:2 CSA	32.0	13000	1.50	1.20
Delay-Opt. Dec. 17:2 CSA	30.5	14200	1.45	1.30
Area-Opt. Bin/Dec. 17:2 CSA	26.0/37.0	14300	1.25/1.75	1.30
Bin. 32:2 CSA	28.0	21200	1	1
Basic Dec. 32:2 CSA	44.0	30000	1.60	1.40
Area-Opt. Dec. 32:2 CSA	42.0	25500	1.50	1.20
Delay-Opt. Dec. 32:2 CSA	40.5	28000	1.45	1.30
Area-Opt. Bin/Dec. 32:2 CSA	34.0/47.0	27000	1.20/1.70	1.30

Table 6.3. *Evaluation results for the proposed CSAs (64-bit operands).*

binary CSA tree consists of 2 levels of binary 3:2 and 2 levels of binary 4:2 compressors. The binary 32:2 CSA is a tree of 4 levels of 4:2 compressors. Area and delay ratios for the proposed q:2 CSA trees are given with respect to the equivalent q:2 binary CSA tree.

Delay ratios are close to 1.45 (45% more latency) in the case of the delay-optimized decimal CSAs and 1.20 for binary vs. 1.70 for decimal in the case of combined binary/decimal CSA trees optimized for area. For area ratios, figures are close to 1.20 (20% more area) in the case of area-optimized decimal implementations and around 1.30 for combined implementations (area-optimized).

6.6.2 Comparison

The methods analyzed can be grouped in decimal signed-digit (SD) trees [51, 127], decimal 4-bit CLA trees [50, 91] and based on binary CSA trees [41, 81, 109]. We have also implemented a binary 16:2 CSA using a 3-level tree of binary 4:2 compressors. Table 6.4 shows the area and delay estimations for these different decimal tree adders and sixteen 64-bit operands. Area

Architecture	Delay (#FO4)	Area (NAND2)	Delay Ratio	Area Ratio
Binary CSA	21.0	10100	0.65	0.85
SD tree adder [51, 127]	61.0	31300	1.90	2.55
BCD CLA tree adders				
Ref. [50]	46.2	16700	1.45	1.40
Ref. [91]	42.9	16700	1.35	1.40
Based on binary tree adders				
Ref. [109]	48.5	31000	1.50	2.55
Ref. [81]	40.7	17600	1.30	1.45
Ref. [41]	38.4	18400	1.20	1.50
Proposed decimal CSAs				
Basic	34.0	13600	1.05	1.10
Delay-optimized	30.5	13400	0.95	1.10
Area-optimized	32.0	12200	1	1
Mixed (4221/5211)	31.0	12600	0.95	1.05
Results for 64-bit or 16-digit operands.				

Table 6.4. *Area-delay figures for 16:2 carry-free tree adders.*

and delay ratios are given with respect to our area-optimized decimal 16:2 CSA. Thus, a binary CSA tree of 4:2 compressors is 35% faster (delay ratio 0.65) and requires 15% less area (area ratio 0.85) than the proposed area-optimized decimal CSA. The proposed delay-optimized decimal CSA is slightly faster than the area-optimized decimal CSA (roughly 5% faster) but requires 10% more area. In the case of the (4221/5211) decimal mixed implementation, the figures are close to the area-optimized decimal CSA. The basic implementation presents the worst area and delay figures of our proposed decimal CSAs.

The complexity of the signed-digit decimal adder [132] leads to decimal signed-digit tree adders [51, 127] with high area and delay figures, inappropriate for high speed multioperand addition. The decimal CSA tree proposed in [109] also presents high area and delay figures, due to the multiple and complex corrections and digit additions performed in the critical path. BCD trees [50, 82] present good area/delay tradeoffs, but for high speed multioperand decimal addition the schemes based on binary trees [81] or [41] are a better choice. When compared with [41], our decimal CSA is at least 20% faster and requires about 50% less hardware and is, therefore, a good choice for high performance multioperand addition with moderate area.

6.7 Conclusions

We have developed a hardware algorithm to improve multioperand decimal carry-free addition. This method performs a decimal carry-save addition by means of a binary carry-save addition over three decimal operands represented not in BCD, but in non-conventional (4221) and (5211) decimal encodings. Each binary carry-save addition produces a sum and a carry operands represented in the same decimal code as the input operands, that is (4221) or (5211). The carry operand must be multiplied $\times 2$. This decimal operation is implemented as a digit recoding (no carry propagation between digits). This algorithm makes possible the

construction of q:2 decimal CSA trees that outperform the area and delay figures of existing proposals.

We have introduced three decimal carry-free adders to reduce 9 to 4, 8 to 4 and 7 to 3 decimal operands coded in (4221) or (5211). We use them to reduce the critical path delay of decimal q:2 CSAs. Hence, we have presented basic, area-optimized and delay-optimized implementations of decimal q:2 CSA trees combining different design strategies. We have also designed mixed CSAs admitting input operands coded in both (4221) and (5211) simultaneously. Moreover, the proposed method also allow the computation of combined binary/decimal multioperand addition with a moderate area and delay overhead.

The area and delay figures from a comparative study show that the proposed decimal CSA trees are a very interesting option for high performance with moderate area. With respect to a binary CSA tree, our decimal CSA trees are about 45% slower and have 20% more area.

Chapter 7

Decimal Multiplication

The new generation of high-performance DFUs (decimal floating-point units) is demanding efficient implementations of decimal multiplication. Although this is an important and frequent operation, current hardware implementations suffer from lack of performance. Parallel binary multipliers [108, 124] are used extensively in most of the binary floating-point units for high performance. However, decimal multiplication is more difficult to implement due to the complexity in the generation of multiplicand multiples and the inefficiency of representing decimal values in systems based on binary signals. These issues complicate the generation and reduction of partial products.

In this Chapter we present the architectures of two parallel DFX (decimal fixed-point) multipliers and a combined binary/decimal fixed-point multiplier. They were introduced in [148, 150], and use new techniques for partial product generation and reduction. The parallel generation of partial products is performed using SD (signed-digit) radix-10, radix-5 or radix-4 recodings of the multiplier and a simplified set of multiplicand multiples. The reduction of partial product is implemented in a tree structure based on the decimal multioperand carry-save addition methods presented in Chapter 6. Moreover, SD radix-4 and radix-5 recodings allow efficient implementations of parallel combined binary/decimal multipliers. Design decisions are supported by the area-delay model for static CMOS technology described in Appendix A.

All of the previous designs are combinational fixed-point multipliers. A pipelined IEEE 754-2008 compliant DFP (decimal floating-point) multiplier based on one of our DFX architectures [148] was proposed in [67]. We introduce two new different and faster proposals to support IEEE 754-2008 DFP multiplications using our parallel DFX multipliers and the sign-magnitude BCD adder with decimal rounding presented in Chapter 5. In addition, we also propose a parallel scheme for a decimal FMA (fused-multiply-add).

The rest of this Chapter is organized as follows. Section 7.1 outlines the previous (most representative) work on decimal multiplication. In Section 7.2 we introduce the proposed techniques for an efficient implementation of decimal parallel multiplication. The generation of decimal partial products is detailed in Section 7.3, while the reduction of partial products is discussed in detail in Section 7.4. We present the fixed-point architectures in Section 7.5 and the floating-point architectures in Section 7.6. We show the area and delay figures for the 64-bit (16-digit) combinational fixed-point architectures in Section 7.7. We also compare the

two proposed designs with some representative binary and decimal fixed point multipliers. We finally summarize the main conclusions in Section 7.8.

7.1 Overview of DFX multiplication and previous work

Integer and fixed-point multiplication (both binary and decimal) consists of three stages: generation of partial products, reduction (addition) of partial products to two operand words and a final conversion (usually a carry propagate addition) to a non redundant representation. Extension to decimal floating-point multiplication involves exponent addition, rounding of the integer product to fit the required precision and sign calculations.

Decimal multiplication is more complex than binary multiplication mainly for two reasons: the higher range of decimal digits ($[0,9]$), which increments the number of multiplicand multiples and the inefficiency of representing decimal values in systems based on binary logic using BCD (since only 9 out of the 16 possible 4-bit combinations represent a valid decimal digit). These issues complicate the generation and reduction of partial products.

Commercial implementations of decimal integer and fixed-point multiplication [19, 20, 45, 109] are based on iterative algorithms and therefore present low performance and a reduced area cost. Several sequential decimal multipliers were proposed in recent academical research [50, 51, 82], which improve the latency of commercial ones. More recently, the first known proposal of a parallel fixed-point decimal multiplier is described in [91].

Proposed methods for the generation of decimal partial products follow two approaches. The first alternative performs a BCD digit-by-digit multiplication of the input operands, using lookup table methods [94, 141] or combinational logic [77]. In a recent work [51], a magnitude range reduction of the operand digits by a signed-digit radix-10 recoding (from [0,9] to [-5,5]) is suggested. This recoding of both operands speeds-up and simplifies the generation of partial products. Then, signed-digit partial products are generated using simplified tables and combinational logic. This class of methods is only suited for serial implementations, since the high hardware demands make them impractical for parallel partial product generation (see [148]).

The second approach generates and stores all the required multiplicand multiples [113]. Next, multiples are distributed to the reduction stage through multiplexers controlled by the BCD multiplier digits ($[0,9]$). This approach requires several wide decimal carry-propagate additions to generate some complex BCD multiplicand multiples (3X,6X,7X,8X,9X). In [20], only even multiples $\{2X, 4X, 6X, 8X\}$ are computed and stored in the register file before multiplying. Odd multiples $\{3X, 5X, 7X, 9X\}$ are obtained on demand from the corresponding even multiple as $mX + X, \quad m = \{2, 4, 6, 8\}$. A reduced set of BCD multiples $\{X, 2X, 4X, 5X\}$ is precomputed in [50] without a carry propagation over the whole number. All the multiples can be obtained from the sum of two elements of this set. To avoid complicated multiples and reduce the complexity, the multiplier can be recoded to a signed digit (SD) representation. In [91] each multiplier digit is recoded as $Y_i = Y_H\ 5 + Y_L$, with $Y_H \in \{0, 1\}$ and $Y_L \in \{-2, -1, 0, 1, 2\}$. The $2X$ and $5X$ multiples are computed in few levels of combinational logic. Negative multiples require an additional 10's complement operation.

The reduction of decimal partial products is performed, in most of the commercial proces-

sors [19, 20, 45, 160], sequentially, by shifting and adding each partial product to the accumulated partial product sum, using a BCD carry-propagate adder [118]. To improve the reduction of decimal partial products, several sequential [50, 51, 82, 109] and parallel [91] multipliers use arrays or trees of decimal carry-free adders. Multioperand carry-free addition techniques were detailed in Chapter 6. The result of this carry-free reduction is finally assimilated to a non-redundant BCD operand (final product) in a BCD carry-propagate adder or in a conversion unit ($O(\log(p))$ delay).

7.2 Proposed techniques for parallel DFX multiplication

We consider that multiplicand $X = \sum_{i=0}^{p-1} X_i \ 10^i$ and multiplier $Y = \sum_{i=0}^{p-1} Y_i \ 10^i$ are unsigned decimal integer p-digit BCD words. The final product

$$P = X \times Y = \sum_{i=0}^{2p-1} P_i \ 10^i \tag{7.1}$$

is a non-redundant $2p$-digit BCD operand.

We opt for recoding the BCD multiplier to compute only a reduced set of decimal multiplicand multiples. Each recoded digit produces a partial product by selecting the appropriate multiple. In this way, all the partial products are generated in parallel. We have developed three different SD (signed-digit) recodings for Y with good trade-offs between fast generation of partial products and the number of partial products generated.

A minimally redundant radix-10 recoding (with digits in $[-5, 5]$) produces only $p + 1$ partial products but requires a carry propagate addition to generate complex multiples $3X$ and $-3X$. Minimally redundant signed-digit (SD) radix-4 and radix-5 recodings (with digits in $[-2, 2]$) produce $2p$ partial products (two per each BCD digit of the multiplier) but multiplicand multiples are produced faster in a few levels of combinational logic.

For a fast carry-free reduction, we propose to represent the decimal partial products in the (4221) or (5211) decimal encodings. Thus, we can use the decimal CSA trees introduced in Chapter 6 to reduce the $p + 1$ or $2p$ partial products to 2. Partial product reduction is detailed in Section 7.4.

In this way, the multiplicand multiples are not generated in BCD, but in (4221) or (5211) decimal codes. Furthermore, another advantage of using (4221) or (5211) to represent the multiplicand multiples is that their 9's complement is obtained by bit inversion. This simplifies the generation of the negative multiples. The generation and selection of (4221) or (5211) decimal coded multiples is detailed in Section 7.3.

For the final decimal carry-propagate addition we use a $2p$-digit BCD Q-T (quaternary prefix tree) adder, similar to the one proposed in Section 3.3.2. Decimal Q-T adders based on conditional speculative decimal addition [144] present low latency (about 15% more than the fastest binary adders) and require less hardware than other alternatives.

7.3 Generation of partial products

We aim for a parallel generation of a reduced number of partial products coded in (4221) or (5211). In this Section we present three schemes for a fast generation of all partial products. These schemes are based on different SD recodings of the BCD multiplier and require a different set of decimal multiplicand multiples.

The minimally redundant SD radix-10 recoding of the multiplier uses the set of decimal multiples $\{-5X, -4X, \ldots, 0, \ldots, 4X, 5X\}$. It produces only $p+1$ partial products but requires a carry propagate addition to generate complex multiples $3X$ and $-3X$.

The second scheme, named SD radix-5 recoding, recodes each BCD digit Y_i of the multiplier into two digits $Y_U \in \{0,1,2\}$ $Y_L \in \{-2,-1,0,1,2\}$, such as $Y_i = Y_U\ 5 + Y_L$. It generates $2p$ partial products (2 digits per BCD digit), but the required decimal multiples $\{-2X, -X, 0, X, 2X, 5X, 10X\}$ are computed faster. Furthermore, some of the proposals based on the decomposition $Y_i = Y_U\ 5 + Y_L$ [91, 148] require combinational logic to generate the $5X$ multiple. We use mixed (4221/5211) decimal codings to remove this logic, so we only need to compute $\{-2X, -X, X, 2X\}$ coded in (4221), as proposed in [150].

Finally, the SD radix-4 recoding scheme also generates $2p$ partial products but uses a different set of multiples, that is, $\{-2X, -X, 0, X, 2X, 4X, 8X\}$. This recoding is quite similar as the conventional SD radix-4 recoding for binary (or Booth radix-4 recoding). Both SD radix-5 and radix-4 schemes could be of interest for combined binary/decimal implementations.

In the next subsections, we detail the generation of the decimal multiplicand multiples coded in (4221) or (5211) and the different schemes for the selection of multiples.

7.3.1 Generation of multiplicand's multiples

The BCD multiplicand X is recoded to the (4221) code in a simple way using the following logical equations:

$$\begin{aligned} w_{i,3} &= h_{i,3} \vee h_{i,2} \\ w_{i,2} &= h_{i,3} \\ w_{i,1} &= h_{i,3} \vee h_{i,1} \\ w_{i,0} &= h_{i,0} \end{aligned} \tag{7.2}$$

where $X_i(4221) = \sum_{j=0}^{3} w_{i,j}\ r_j$ and $X_i(BCD) = \sum_{j=0}^{3} h_{i,j}\ 2^j$. In particular, this recoding maps the BCD representation into the subset (4221-S2), shown in Table 6.2.

Decimal multiplicand multiples $2X$ and $5X$ are obtained in a few levels of logic using a digit recoding and a binary wired left shift. The generation sequence of $2X$ is as follows. Each BCD digit is first recoded to the (5421) decimal coding shown in Table 7.1 (the mapping is unique). The complexity of this digit recoder is similar to the (4221-S2) to (5211-S2) digit recoder (Fig. 6.5(b)). A 1-bit wired left shift is performed to the recoded multiplicand, obtaining the $2X$ multiple in BCD. Then, the $2X$ multiple is easily recoded from BCD to (4221) using the logical expressions (7.2).

The $5X$ multiple is obtained by a simple 3-bit wired left shift of the (4221) recoded multiplicand, with resultant digits coded in (5211). Then, a digit recoding from (5211) to (4221)

Z_i	BCD	$Z_i(5421)$
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Table 7.1. *Decimal codings*

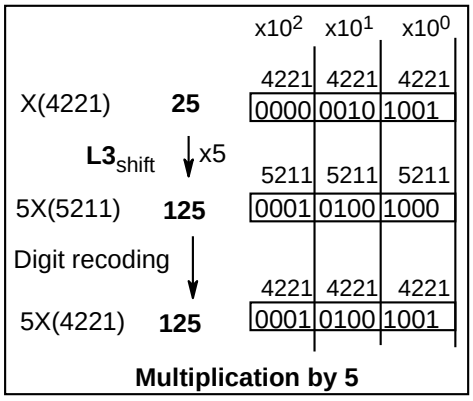
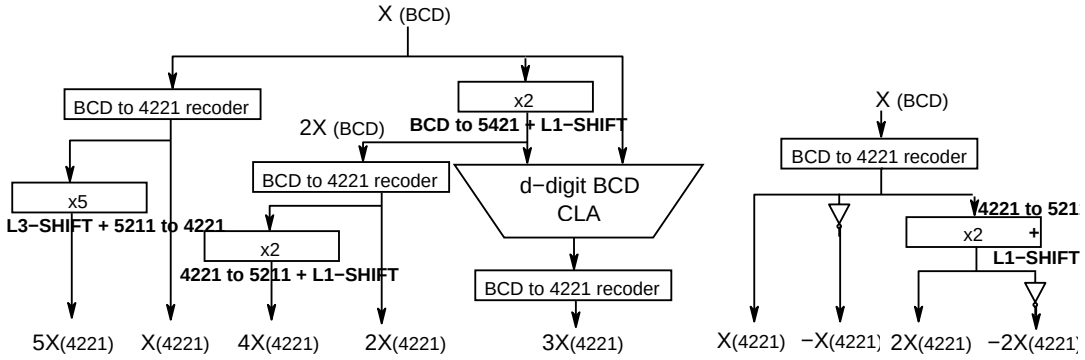


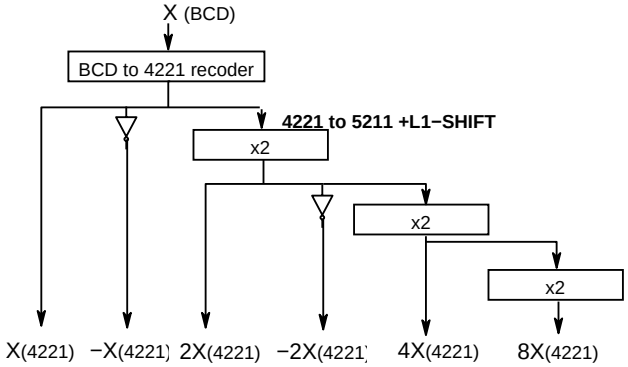
Figure 7.1. *Calculation of $\times 5$ for decimal operands coded in (4221).*

is performed (Fig. 6.6). Fig. 7.1 shows an example of this operation. Decimal multiple $4X$ is obtained as $2 \times 2X$. The second $\times 2$ operation is implemented as a digit recoding from (4221-S2) to (5211-S2) followed by a 1-bit left shift. Multiple $3X$ is evaluated by a carry propagate addition of multiples X and $2X$ in a p-digit BCD carry-propagate adder. The BCD sum digits are recoded to (4221-S2).

Fig. 7.2(a) shows the block diagram for the generation of positive multiplicand multiples for SD radix-10 recoding. The latency of the partial product generation for the SD radix-10 scheme is constrained by the generation of $3X$. The generation of multiples for the SD radix-5 recoding is shown in Fig. 7.2(b). The BCD multiplicand is first recoded to (4221-S2). The $2X$ multiple is implemented as a digit recoding from (4221-S2) to (5211-S2) followed by a 1-bit wired left shift. The negative multiples $\{-X, -2X\}$, coded in (4221), are obtained inverting the bits of the (4221-S2) decimal coded positive multiples and encoding the sign as described in Section 7.3.2. Fig. 7.2(c) shows the generation of multiples for the case of the decimal SD radix-4 recoding. Decimal multiple $8X$ is obtained as $2 \times 4X$, so the generation of multiplicand multiples is almost 3 times faster in the SD radix-5 scheme (Fig. 7.2(b)).



(a) Multiples for SD radix-10. (b) Multiples for SD radix-5.



(c) Multiples for SD radix-4.

Figure 7.2. *Generation of multiplicand multiples.*

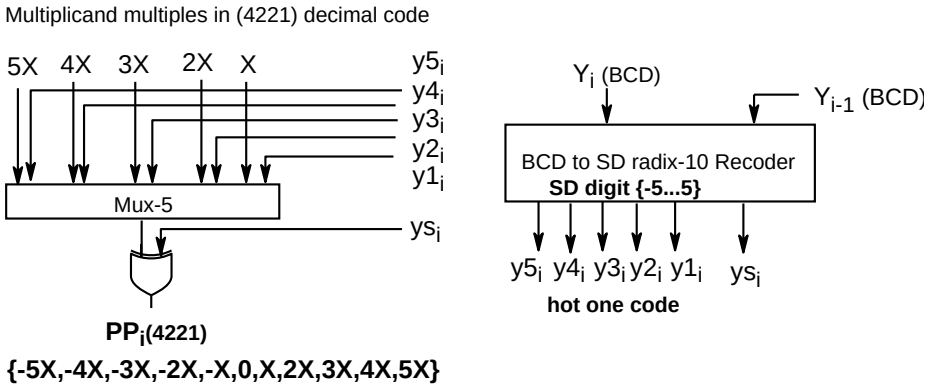


Figure 7.3. *Partial product generation for SD radix-10.*

7.3.2 Signed-digit multiplier recodings

A. SD Radix-10 Recoding.

Fig. 7.3 shows the block diagram of the generation of one partial product using the SD radix-10 recoding. This recoding transforms a BCD digit $Y_i \in \{0, \ldots, 9\}$ into a SD radix-10 $Yb_i \in \{-5, \ldots, 5\}$. As shown in Table 7.2, the value of the recoded digit Yb_i depends on the

decimal value of Y_i and on a signal ys_{i-1} (sign signal) that indicates if Y_{i-1} is greater or equal than 5. Thus, the p-digit BCD multiplier Y is recoded into the $(p+1)$-digit SD radix-10

Dec. value	BCD Y_i	$Y_{i-1} \geq 5$ ys_{i-1}	SD radix-10 digit Yb_i	Hot one code signals $ys_i y5_i y4_i y3_i y2_i y1_i$
0	0000	0	0	000000
		1	1	000001
1	0001	0	1	000001
		1	2	000010
2	0010	0	2	000010
		1	3	000100
3	0011	0	3	000100
		1	4	001000
4	0100	0	4	001000
		1	5	010000
5	0101	0	-5	110000
		1	-4	101000
6	0110	0	-4	101000
		1	-3	100100
7	0111	0	-3	100100
		1	-2	100010
8	1000	0	-2	100010
		1	-1	100001
9	1001	0	-1	100001
		1	0	100000

Table 7.2. *SD radix-10 selection signals.*

multiplier $Yb = \sum_{i=0}^{p} Yb_i \ 10^i$ with $Yb_p = ys_{p-1} \in \{0,1\}$[21]. Each digit Yb_i generates a partial product $PP[i]$ selecting the proper multiplicand multiple coded in (4221).

The selection of the multiplicand multiples $\{-5X, \ldots, 5X\}$ is then performed in a similar way to a modified Booth recoding: five 'hot one code' signals ($y1_i$, $y2_i$, $y3_i$, $y4_i$ and $y5_i$) are used as selection control signals for the 5:1 muxes to select the positive $(p+1)$-digit multiples $\{0, X, 2X, 3X, 4X, 5X\}$. These 'hot-one code' signals are obtained directly from the BCD multiplier digits Y_i using the following logical expressions:

$$
\begin{aligned}
ys_i &= y_{i,3} \vee y_{i,2} \cdot (y_{i,1} \vee y_{i,0}) \\
y5_i &= y_{i,2} \cdot \overline{y_{i,1}} \cdot (y_{i,0} \oplus ys_{i-1}) \\
y4_i &= ys_{i-1} \cdot y_{i,0} \cdot (y_{i,2} \oplus y_{i,1}) \vee \overline{ys_{i-1}} \cdot y_{i,2} \cdot \overline{y_{i,0}} \\
y3_i &= y_{i,1} \cdot (y_{i,0} \oplus ys_{i-1}) \\
y2_i &= \overline{ys_{i-1}} \cdot \overline{y_{i,0}} \cdot (y_{i,3} \vee \overline{y_{i,2}} \cdot y_{i,1}) \vee ys_{i-1} \cdot \overline{y_{i,3}} \cdot y_{i,0} \cdot \overline{y_{i,2} \oplus y_{i,1}} \\
y1_i &= \overline{y_{i,2} \vee y_{i,1}} \cdot (y_{i,0} \oplus ys_{i-1})
\end{aligned}
$$

Table 7.2 shows the value of the 'hot one code' selection signals for the SD radix-10 recoding. The sign signal ys_i determines if the negative multiple should be produced by the 10's complement of the corresponding positive multiple. This is performed simply by a bit inversion

[21]The most significant partial product is just 0 or the multiplicand X.

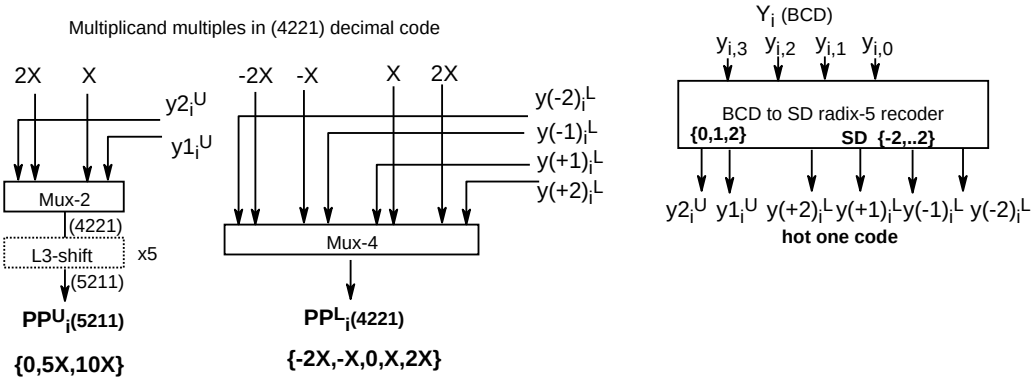


Figure 7.4. *Decimal partial product generation for SD radix-5.*

of the positive (4221) decimal coded multiple using a row of XOR gates controlled by ys_i. The addition of one ulp (unit in the last place) is performed enclosing a tail encoded bit ys_i to the next significant partial product $PP[i+1]$, since it is shifted a decimal position to the left from $PP[i]$. Therefore, the $p+1$ partial products generated have the following expression:

$$PP[i] = \begin{cases} -ys_0\ 10^{p+1} + \sum_{j=0}^{p}(|Yb_i\ X|_j \oplus ys_0)10^j) & if(i == 0) \\ -ys_i\ 10^{p+1} + \sum_{j=0}^{p}(|Yb_i\ X|_j \oplus ys_i)10^j + ys_{i-1}10^{-1} & if(0 < i < d) \\ ys_{p-1}\ X & if(i == p) \end{cases}$$

where $|Yb_i\ X|_j$ is the j^{th} digit of the positive multiple selected. To avoid a sign extension and thus to reduce the complexity of the partial product reduction tree, the partial product sign bits ys_i are encoded at each leading digit position as

$$PP[i]_{p+1} = \begin{cases} (0,0,0,\overline{ys_0})10 + (ys_0, ys_0, ys_0, ys_0) & If(i == 0) \\ (1,1,1,\overline{ys_i}) & If(0 < i < p-1) \\ (0,0,0,0) & If(i = p-1) \end{cases}$$

Therefore, each partial product is at most of $p+3$-digit length.

B. SD Radix-5 Recoding.

Fig. 7.4 shows the diagram of decimal partial product generation using the SD radix-5 recoding scheme.

Each BCD digit of the multiplier is encoded into two digits $Y_i^U \in \{0,1,2\}$ and $Y_i^L \in \{-2,-1,0,1,2\}$ as $Y_i = Y_i^U \cdot 5 + Y_i^L$. The specific mapping along with the 'hot one code' control signals for selection of multiples is shown in Table 7.3. SD radix-5 'hot one code' selection signals are obtained from the BCD input digits using the following expressions:

$$(Y_i^U)\begin{cases} y2_i^U = y_{i,3} \\ y1_i^U = y_{i,2} \vee y_{i,1} \cdot y_{i,0} \end{cases}$$

$$(Y_i^L)\begin{cases} y(+2)_i^L = y_{i,1} \cdot (y_{i,2} \cdot y_{i,0} \vee \overline{y_{i,2}} \cdot \overline{y_{i,0}}) \\ y(+1)_i^L = \overline{y_{i,3}} \cdot \overline{y_{i,2}} \cdot \overline{y_{i,1}} \cdot y_{i,0} \vee y_{i,2} \cdot y_{i,1} \cdot \overline{y_{i,0}} \\ y(-1)_i^L = y_{i,3} \cdot y_{i,0} \vee y_{i,2} \cdot \overline{y_{i,1}} \cdot \overline{y_{i,0}} \\ y(-2)_i^L = y_{i,3} \cdot \overline{y_{i,0}} \vee \overline{y_{i,2}} \cdot y_{i,1} \cdot y_{i,0} \end{cases}$$

Dec. value	BCD (Y_i)	Recoded digits (Y_i^U)	(Y_i^L)	Hot one code signals $y2_i^U$	$y1_i^U$	$y(+2)_i^L$	$y(+1)_i^L$	$y(-1)_i^L$	$y(-2)_i^L$	Sign ys_i^L
0	0000	0	0	0	0	0	0	0	0	0
1	0001	0	1	0	0	0	1	0	0	0
2	0010	0	2	0	0	1	0	0	0	0
3	0011	1	-2	0	1	0	0	0	1	1
4	0100	1	-1	0	1	0	0	1	0	1
5	0101	1	0	0	1	0	0	0	0	0
6	0110	1	1	0	1	0	1	0	0	0
7	0111	1	2	0	1	1	0	0	0	0
8	1000	2	-2	1	0	0	0	0	1	1
9	1001	2	-1	1	0	0	0	1	0	1

Table 7.3. *SD radix-5 selection signals.*

Each multiplier digit Y_i generates two partial products $PP[i]^U$ and $PP[i]^L$. Therefore, this scheme generates $2p$ partial products for a p-digit multiplier. The advantage of this recoding is that it uses a simple set of multiplicand multiples $\{-2X, -X, X, 2X\}$ coded in (4221). Partial product generation is comparable in latency with binary Booth radix-4, due to a faster generation of multiples.

Moreover, the generation of $PP[i]^U$ only requires positive multiples $\{X, 2X\}$. To obtain the correct value of $PP[i]^U$, the (4221) decimal coded multiple selected by Y_i^U ($\{0, X, 2X\}$) must be multiplied by 5 before being aligned and reduced. This is performed by shifting 3 bits to the left the bit vector representation of the selected multiple[22], producing a $\times 5$ operand coded in (5211).

The negative multiples $\{-X, -2X\}$ are the two's complement of the bit vector representation of $\{X, 2X\}$ coded in (4221). The sign bits ys_i^L, given by

$$ys_i^L = y_{i,3} \vee y_{i,2} \cdot \overline{y_{i,1}} \cdot \overline{y_{i,0}} \vee \overline{y_{i,2}} \cdot y_{i,1} \cdot y_{i,0} \tag{7.3}$$

are encoded to the left of $PP[i]^L$ and $PP[0]^U$ as

$$PP[i]_{p+1}^L = \begin{cases} (1,1,1,\overline{ys_i^L}) & If(0 \leq i < p-1) \\ (0,0,0,0) & If(i = p-1) \end{cases}$$
$$PP[0]_{p+1}^U = (0,0,0,ys_0^L) \tag{7.4}$$

The hot ones produced by the 10's complement of the partial products, $(0,0,0,ys_i^L)$, are just enclosed at the least significant digit of $PP[i]^U$ ($PP[i]_0^U$), which has a value of 0 or 5 coded in (5211). The $2p$ partial products generated are at most of $p+2$-digit length, p of them coded in (5211) ($PP[i]^U$) and the other half in (4221) ($PP[i]^L$).

A combined binary Booth radix-4/decimal SD radix-5 block diagram for the partial product generation is proposed in Fig. 7.5. Multiplexes controlled by d_M select the operands required by binary or decimal multiplications.

C. SD Radix-4 Recoding.

[22] Shifting three bits to the left a (4221) decimal coded bit vector is equivalent to multiply its binary weights by 5 to obtain a (5211) coded operand.

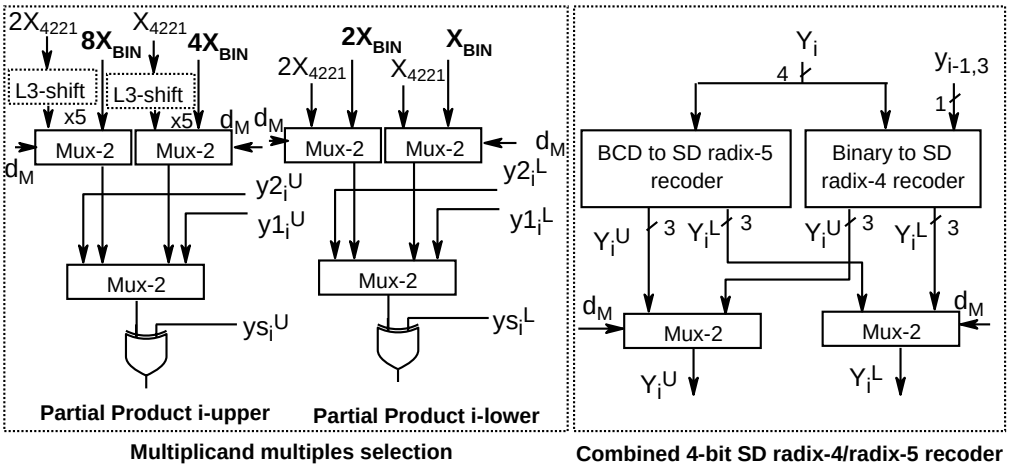


Figure 7.5. *Combined binary SD radix-4/decimal SD radix-5 scheme.*

Two SD radix-4 digits $Y_i^U \in \{0,1,2\}$ (upper), $Y_i^L \in \{-2,-1,0,1,2\}$ (lower) are generated per each BCD digit ($Y_i = Y_i^U \cdot 4 + Y_i^L$). We obtain the SD radix-4 selection signals directly from the BCD digits as

$$(Y_i^U) \begin{cases} ys_i^U = y_{i,3} \\ y2_i^U = \overline{y_{i,3}} \cdot y_{i,2} \cdot y_{i,1} \\ y1_i^U = \overline{y_{i,3}} \cdot y_{i,2} \oplus y_{i,1} \end{cases}$$

$$(Y_i^L) \begin{cases} ys_i^L = y_{i,3} \vee y_{i,1} \\ y2_i^L = ys_i^L \cdot \overline{y_{i,0}} \cdot \overline{y_{i-1,3}} \vee \overline{ys_i^L} \cdot y_{i,0} \cdot y_{i-1,3} \\ y1_i^L = y_{i,0} \oplus y_{i-1,3} \end{cases}$$

Though the SD radix-4 recoder is of similar complexity than the SD radix-5 recoder, the computation of decimal multiples $4X$ and $8X$ requires double and triple latency with respect to the evaluation of $2X$ (see Fig. 7.2). Therefore, this scheme only seems of interest for combined binary/decimal implementations.

The block diagram of a 4-bit combined binary/decimal recoder and the corresponding multiplicand multiple selector are shown in Fig. 7.6 where control signal d_M is true for decimal multiplication. The combined SD radix-4 recoder implements the decimal selection signals and the conventional Booth radix-4 selection signals. Upper signals select multiples $\pm 8X$ and $\pm 4X$ while lower signals select multiples $\{-2X, -X, X, 2X\}$.

7.4 Reduction of partial products

After the generation, the decimal partial products are aligned according to their decimal weights as

$$P = X \times Y = \sum_{i=0}^{p} PP[i]\ 10^i \tag{7.5}$$

and

$$P = X \times Y = \sum_{i=0}^{p} (PP^U[i] + PP^L[i])\ 10^i \tag{7.6}$$

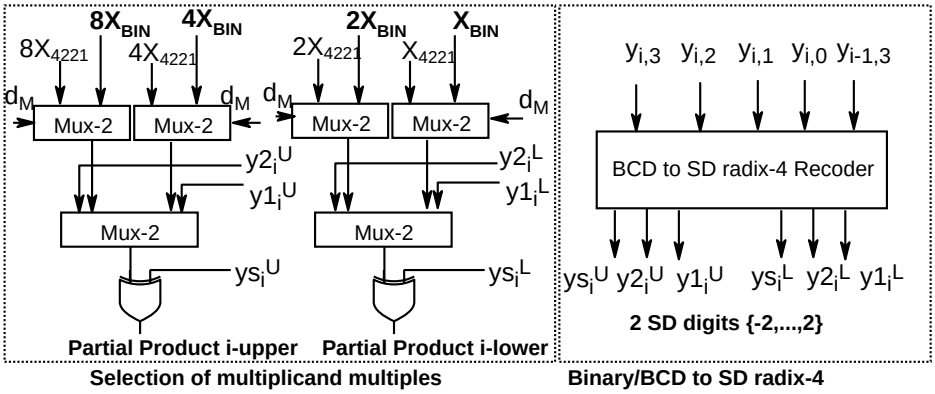
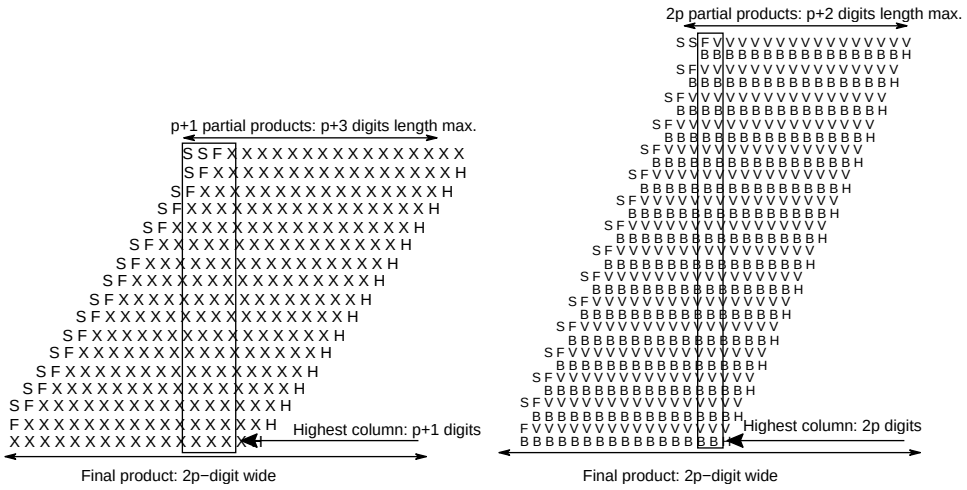


Figure 7.6. *Partial product generation for SD radix-4.*



(a) SD radix-10 DFX multiplier. (b) SD radix-5 DFX multiplier.

Figure 7.7. *Partial product arrays of the DFX multipliers.*

for the SD radix-10 and SD radix-5 schemes respectively. These alignments do not require any logic, since they are performed as $4i$-bit wired left shifts. The resultant partial product arrays after alignment are shown in Fig. 7.7. For the SD radix-10 architecture, the array of $p + 1$ decimal partial products is shown in Fig. 7.7(a). Each partial product, coded in (4221) is at most of $p + 3$-digit length. In the diagram, X indicates a (4221) decimal digit, S is an encoded sign digit, H is the 'hot one' encoding required for negative multiples and F represents the extra digit position required to represent the magnitude of the $(p + 1)$-digit multiplicand multiples (note that the multiplicand is a p-digit operand).

The decimal partial products are reduced using one of the proposed decimal CSA trees for input operands coded in (4221), described in Section 6.5. We had considered that the size of the decimal CSA tree was constant for each digit position. However, the height of the partial product array varies for each digit column: from (p+1) digits for the highest columns to 2. Therefore, the partial product reduction tree consists of a row of decimal q:2 CSA digit trees, where q is at most $p + 1$ for the SD radix-10 scheme.

For the decimal SD radix-5 architecture, the array of $2p$ partial products (after alignment) is shown in Fig. 7.7(b). The digits of the p decimal partial products coded in (5211) (the upper partial products $PP^U[i]$) are represented as **B**. The digits of the p partial products coded in (4221) are indicated with a V, while S is an encoded sign digit, H is the 'hot one' digit encoding and F is the extra digit required to support the length of the multiplicand multiples. In this case, the partial product reduction tree is implemented using mixed (4221/5211) decimal q:2 CSA digit trees, where q is at most $2p$.

In the binary case, the Booth radix-4 recoding generates an array of $\lceil \frac{n+1}{2} \rceil$ partial products for an n-bit binary multiplier, where $p = 4n$, so the number of partial products generated is $2p$. Therefore, we can use the same tree to reduce the partial products generated by the Booth radix-4 recoding for binary and by the SD radix-4 or SD radix-5 recodings for decimal. This reduction tree is implemented by a row of the combined binary/decimal q:2 CSA digit trees (with $q = 2p$ at most), detailed in Section 6.5.4.

7.5 Decimal fixed-point architectures

In this Section we present the architecture of two DFX parallel multipliers for 16-digit (64-bit) BCD operands based on the SD radix-10 and SD radix-5 recodings. We also detail the architecture of two combined binary/BCD fixed-point multipliers, which use a Booth radix-4 recoding for binary and the SD radix-4 or the SD radix-5 recodings for decimal.

7.5.1 Decimal SD radix-10 multiplier

The architecture of the 16-digit SD radix-10 multiplier is shown in Fig. 7.8. The generation of the 17 partial products is performed by an encoding of the multiplier into 16 SD radix-10 digits and an additional leading bit as described in Section 7.3.2. Each SD radix-10 digit controls a level of 76-bit 5:1 muxes and 76 XOR gates that select the corresponding multiple coded in (4221). The 17 partial products are aligned (see Fig. 7.7(a)) and reduced to two 32-digit (128-bit) operands S and H coded in (4221). The number of digits to be reduced varies from $q = 17$ to $q = 2$. The partial product reduction tree consists of a row of (4221) decimal coded digit q:2 CSAs, as described in Section 7.4. In particular, the highest columns can be reduced with a decimal 17:2 CSA digit tree: see Fig. 6.9(a) for a basic implementation, Fig. 6.10 for an area-optimized design, or Fig. 6.12 for a delay-optimized design. The final product is a 32-digit BCD word given by $P = W + S = 2H + S$. This addition is performed in a 128-bit BCD carry propagate adder. The adder architecture is described in Section 5.3. Note that the logic for decimal subtraction is not needed. One of the adder operands is represented in BCD and the other in BCD excess 6 (BCD value plus 6). Thus, S is recoded from (4221) to BCD excess 6 (this has practically the same logical complexity as recoding to BCD). The $W = 2 \times H$ multiplication is performed in parallel with the recoding of S. This $\times 2$ block uses a (4221) to (5421) digit recoder and a 1-bit wired left shift to obtain the operand coded in BCD.

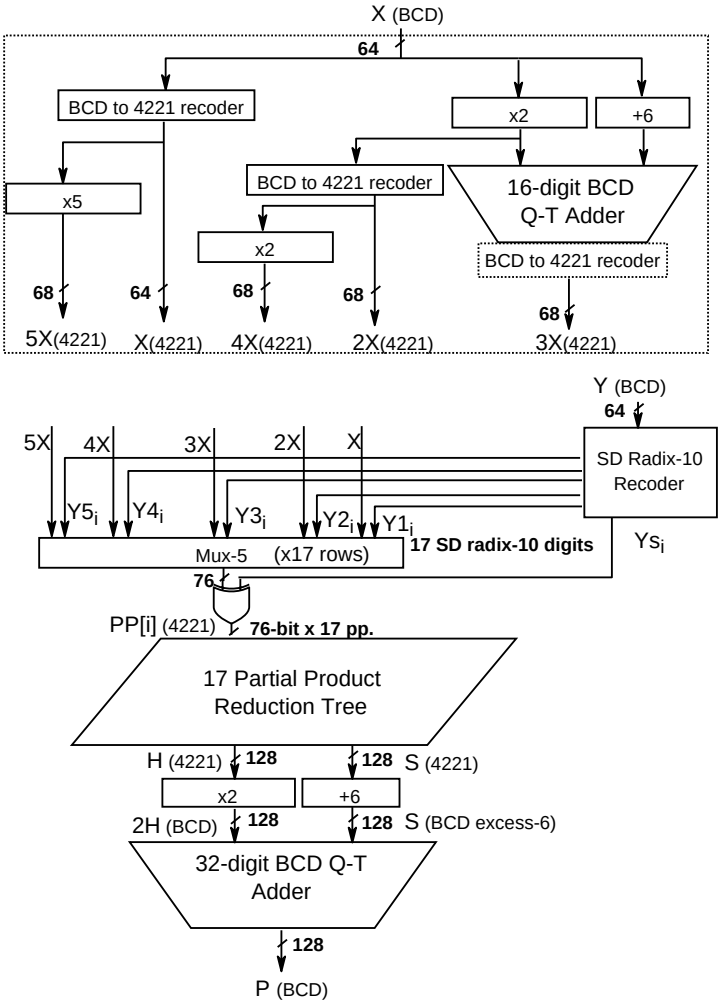


Figure 7.8. *Combinational SD radix-10 architecture.*

7.5.2 Decimal SD radix-5 multiplier

The dataflow of the 16-digit SD radix-5 architecture is shown in Fig. 7.9. It implements the SD radix-5 recoding described in Section 7.3.2, which generates 32 partial products, half coded in (4221) and the other half in (5211). After alignment, the reduction of digit columns is carried out using a row of the mixed (4221/5211) decimal digit q:2 CSAs ($2 \leq q \leq 32$), described in Section 6.5.3. The worst case corresponds to a column of 32 digits, which can be reduced using the delay-optimized decimal 32:2 CSA of Fig. 6.13. As in the SD radix-10 architecture, the 32-digit operands S and H coded in (4221) are assimilated in the 128-bit BCD carry-propagate adder as $P = S + 2 \times H$.

7.5.3 Combined binary/decimal SD radix-4/radix-5 multipliers

The proposed architecture for the combined binary/decimal multipliers is shown in Fig. 7.10. We have different multipliers depending on the scheme used to generate the partial products.

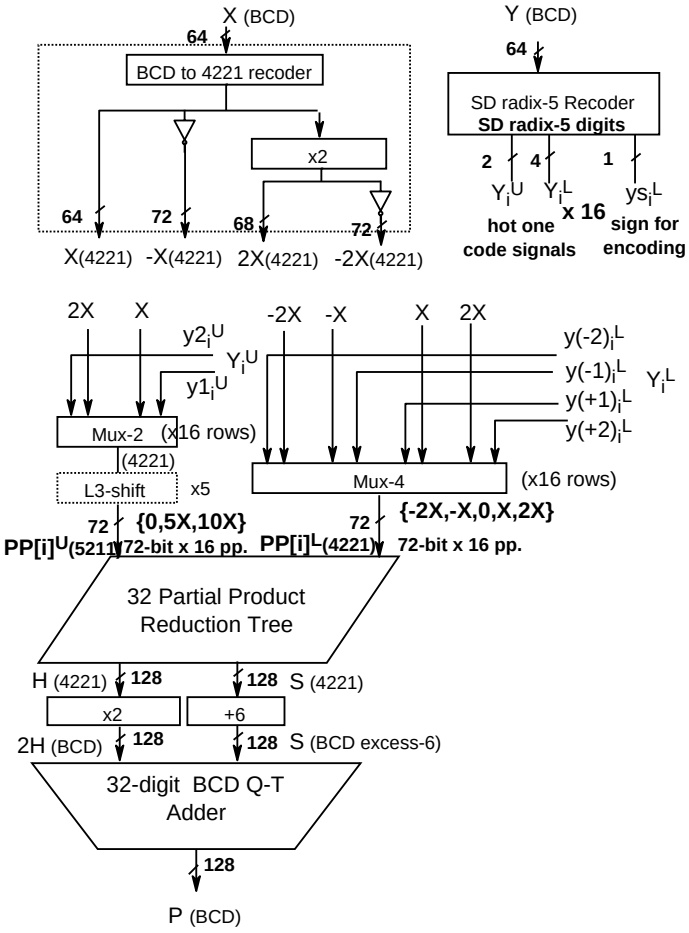


Figure 7.9. *SD radix-5 DFX multiplier.*

A combined binary Booth radix-4/decimal SD radix-4 multiplier can be implemented using the recoder of Fig. 7.6 and the multiplicand multiples of Fig. 7.2(c). The combined binary Booth radix-4/decimal SD radix-5 architecture is implemented using the partial product generation scheme of Fig.7.5, where the generation of multiples is shown in Fig.7.2(b).

In all cases, 32 partial products are generated. The array of 32 partial products is reduced using a combined binary/decimal 32:2 CSA digit tree (see Section 6.5.4) for the highest columns or simpler CSAs for the other columns. The final carry-propagate addition is performed with a 128-bit combined binary/BCD Q-T adder (see Section 3.3.2).

7.6 Decimal floating-point architectures

Older designs of DFP multipliers present low performance [12, 29] and do not conform to the IEEE 754-2008 standard. The IEEE 754-2008 DFP units incorporated in current commercial processors, such as the IBM Power 6 [45] and z10 [160], implement multiplication iteratively. These units implement a 36-digit BCD adder and some hardware shared with other decimal instructions, such a generator of $\times 2$ and $\times 5$ multiples and a 36-digit rotator (which performs

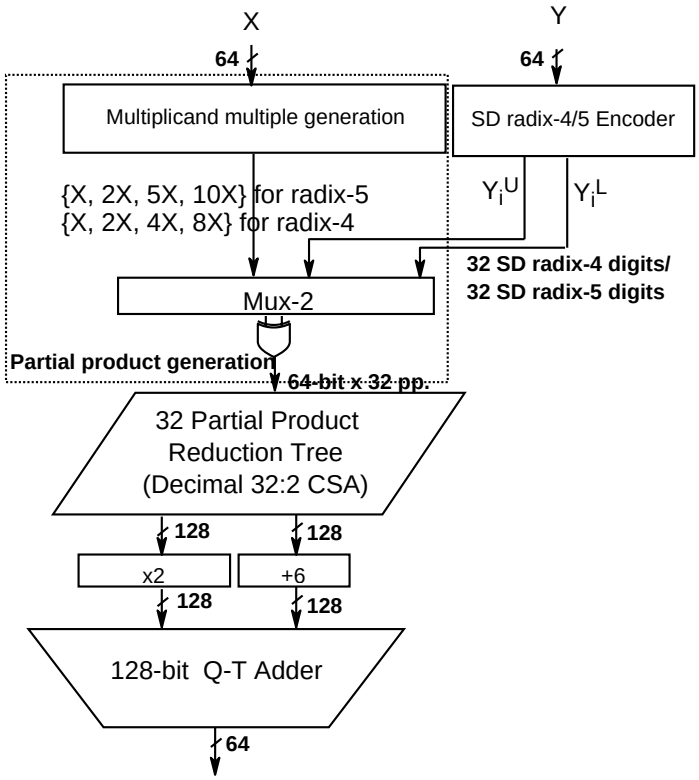


Figure 7.10. *Combined binary/decimal radix-4 (radix-5) multiplier.*

both left and right variable shifts). Therefore, the performance of a decimal multiplication is low (up to 35 cycles of 13 FO4 for Decimal64 and up to 89 cycles for Decimal128) when compared with a binary multiplication (7 cycles for a 64-bit multiplication in the fast parallel Power6 implementation [137]).

More recently, Erle, Schulte and Hickman reported two different IEEE 754-2008 compliant DFP multipliers [97, 67] for Decimal64 (16-digit) operands. The sequential DFP multiplier proposed in [97] is based in a previous fixed-point iterative design [50], described in Section 7.1. It performs a Decimal64 floating-point multiplication in 25 cycles of 15 FO4 (43 cycles with support for gradual underflow).

The scheme of the high-performance parallel DFP multiplier presented in [67] is shown in Fig. 7.11. The fixed-point multiplier block (DFX multiplier) implements our SD radix-10 parallel architecture [148], described in Section 7.5.1.

The decimal carry and sum operands from the partial product reduction must be normalized before rounding. This normalization is carried out as a left shift of $sla = min(lz_X + lz_Y, p)$ digits to remove the leading zeroes of the result (only p at most). The DFP dataflow uses the rounding scheme from the sequential implementation [97]. The (4221) carry and sum operands are recoded to BCD and added in a 2p-digit BCD adder before being normalized (2p-digit left shift). Thus, rounding is performed separately from the 2p-digit BCD carry-propagate addition, which requires another decimal carry propagation of p digits. The fully combinational implementation performs a Decimal64 multiplication in 80 FO4. It can be

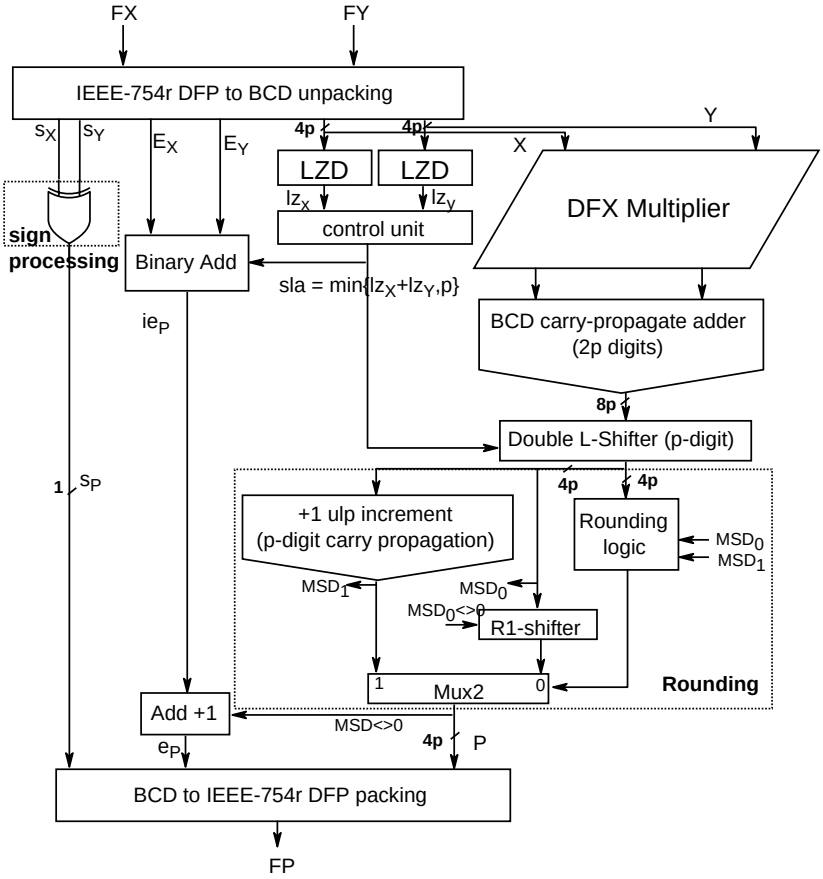


Figure 7.11. *Scheme of the DFP multiplier proposed in [67].*

pipelined in 12 stages of 15 FO4.

To reduce the latency of this scheme we propose to perform the normalization before the combined addition and rounding of the decimal carry-save product. We present two schemes (optimized for area and for delay) in Section 7.6.1.

Moreover, in Section 7.6.2 we propose a scheme for decimal FMA (fused-multiply-addition). This operation ($R = X \times Y \pm Z$) has two advantages over the case of a separate floating-point adder and multiplier: it is performed with only one rounding operation instead of two (reduces the overall delay and error), and several components are shared for two different instructions (area reduction). On the other hand, it is not possible to concurrently perform additions and multiplications (the overall throughput is reduced). However, separate add and multiply operations can be transformed in multiply-add instructions, following the Horner,s rule [120]. High-performance implementations of binary FMA units can be found in [137, 90, 120] among others. Specifically, our scheme is a DFP extension of the high-performance binary FMA unit proposed by Lang and Bruguera [90].

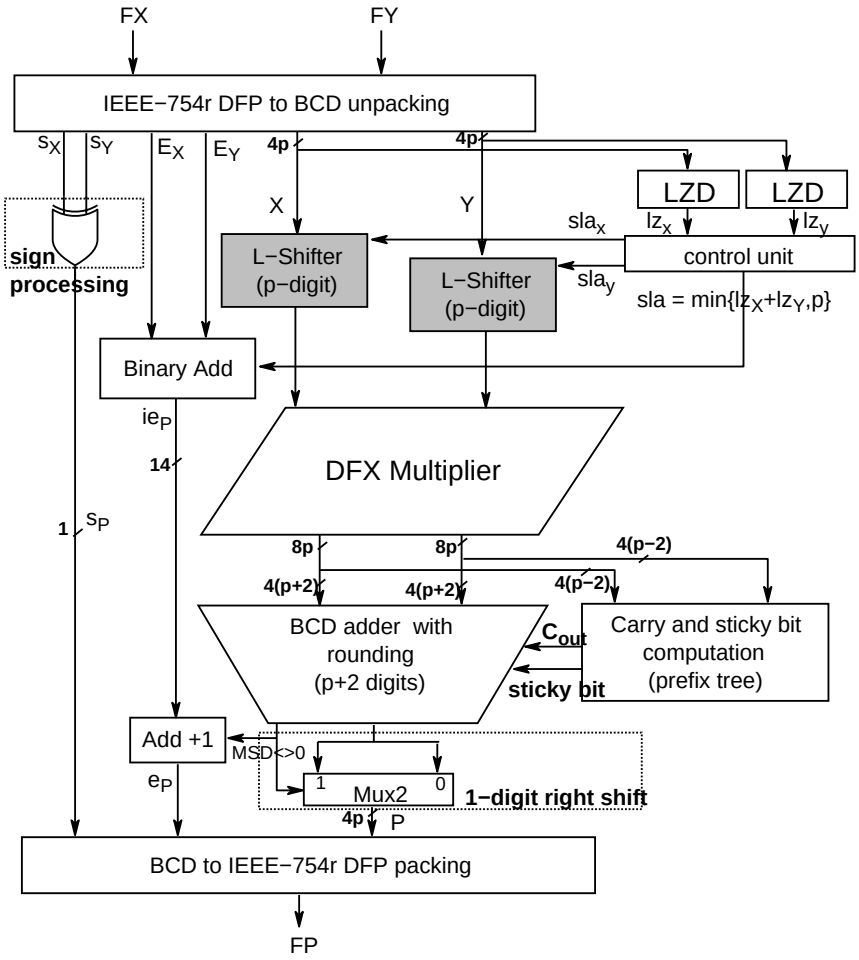


Figure 7.12. *Proposed area-optimized scheme for DFP multiplication.*

7.6.1 DFP multipliers

In a DFP multiplication $FP = FX \times FY = (-1)^s_P \ P \ 10^{E_P}$, the sign s_P, the biased exponent E_P and the product coefficient P are evaluated in three separate paths. Thus, the DFP IEEE 754-2008 multiplicand FX and multiplier FY are first unpacked as (s_X, X, E_X) and (s_Y, Y, E_Y). E_P is the least possible exponent for inexact computations and the closest to the preferred exponent $E_X + E_Y - bias$ for exact computations.

As in binary, the sign is computed straightforwardly as $s_P = s_X \oplus s_Y$. However, DFP multiplication presents a slight difference with respect to binary multiplication: since p-BCD digit coefficients X and Y are not normalized and have leading zeroes, the rounding position (the decimal point) is not fixed. Therefore, to combine the final decimal carry-propagate addition with rounding, a normalization of the $2p$-digit product must be previously performed. This normalization is carried out by removing the leading zeroes (up to p, due to the requirements of exact computation) from the input coefficients or the decimal carry-save product.

In Figs. 7.12 and 7.13 we present the two proposed high-performance schemes for DFP multiplication, one optimized for area and the other for delay.

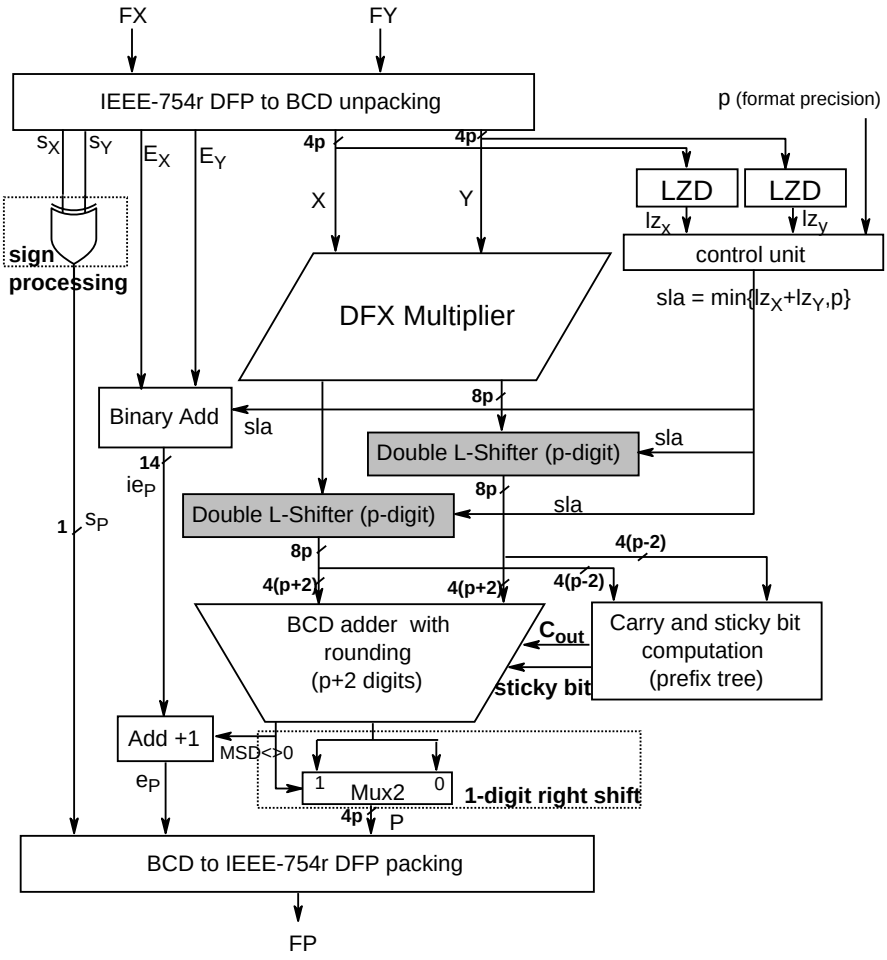


Figure 7.13. *Proposed delay-optimized scheme for DFP multiplication.*

The only difference between the two schemes is the placement of the left shifters for normalization. When they are placed before the DFX multiplication (area-optimized scheme, Fig. 7.12), the width of both shifters is of p digits. The shift left amounts for operands X and Y are computed as

$$\begin{aligned} sla_X &= min(p - lz_y, lz_x) \\ sla_Y &= lz_y \end{aligned} \tag{7.7}$$

where lz_X and lz_Y are the number of leading zeroes of X and Y determined using two LZDs (leading zero detectors). Thus, in this case, the evaluation of sla_X and sla_Y is in the critical path.

In the optimized-delay scheme (Fig. 7.13), the left shifters are placed after the reduction of partial products. The $2p$-digit decimal sum and carry operands are left shifted an amount of digits equal to

$$sla = min(lz_x + lz_y, p) \tag{7.8}$$

Thus, thought the width of the left-shifters is doubled ($2p$ digits) with respect to the area-

optimized scheme, the number of digit positions shifted is p at most. Moreover, the computation of sla is not in the critical path.

The final assimilation and rounding is performed using the combined (p+2)-digit BCD adder with decimal rounding presented in Chapter 5 (see Fig. 5.8). This architecture only requires a minor modification into the rounding logic to support correct rounding of decimal multiplications. For example, the C_{in} from the $2(p-2)$ least significant digits of the decimal carry-save product is incorporated as a carry-in into the BCD sum of the guard and round digits (see Figs. 5.9(a)-(b)). The sticky bit is also computed from the $(p-2)$ LSDs in a tree of OR gates. A final normalization (1-digit right shift) is required if the MSD of the rounded product P is not zero, that is

$$sr1 = \begin{cases} 0 & \texttt{If} MSD(P) = 0 \\ 1 & Else \end{cases} \tag{7.9}$$

The exponent E_P is computed in both schemes as

$$E_P = (E_X + E_Y - bias) - sla + sr1 \tag{7.10}$$

where sla and $sr1$ are given by expressions (7.8) and (7.9) respectively.

7.6.2 Decimal FMA: Fused-Multiply-Add

The objective of this Section is to describe, at high level, a feasible high-performance implementation of a decimal FMA. As we have commented before, the proposed scheme is based on a previous implementation of a high-performance binary FMA [90].

To reduce the latency we combine decimal addition/subtraction with rounding. The main problem in a (binary and decimal) FMA computation ($R = FX \times FY \pm FZ$) is that the radix point position is not known until normalization, due to a possible cancelation of some leading digits in the subtraction of coefficient $X \times Y$ and Z. Thus, we have to place normalization before rounding. Though this is also true for (binary and decimal) floating-point addition, in this case, the digits to the right of the LSD of the result are all zeroes.

Moreover, as we have seen for DFP multiplication, the product $X \times Y$ is not normalized. Therefore the leading zeroes of lz_X and lz_Y also have to be taken into account to align operand $P = X \times Y$ and Z before being added/subtracted. The different alignment cases are summarized in Fig. 7.14. The biased exponent of the result E_R is the least possible for inexact computation and the closest to the preferred exponent $min(E_Z, E_P)$ ($E_P = E_X + E_Y - bias$) for exact computations. Thus, when $E_Z \geq E_P$ the operand Z is shifted $E_Z - E_P$ positions to the left of the $2p$-digit product P. This shift is of $2p+2$ digits maximum (two extra positions are required for the guard and round digits when $E_Z - E_P > 2p$).

On the other hand, when $E_Z < E_P$, the alignment depends on the value $lz_P = lz_X + lz_Y$. The product P is shifted to the left $min(lz_P, p, E_P - E_Z)$ positions. If $E_P - E_Z > lz_P$ the operand Z is shifted $E_P - E_Z - lz_P$ positions to the right. In this way, the digits of Z shifted to the right of the decimal point only contributes to the computation of the sticky bit.

This alignment is equivalent to perform at most a $3p+2$-digit right shift of Z and a left shift of P of p digits at most. The normalization consists then on removing the leading zeroes

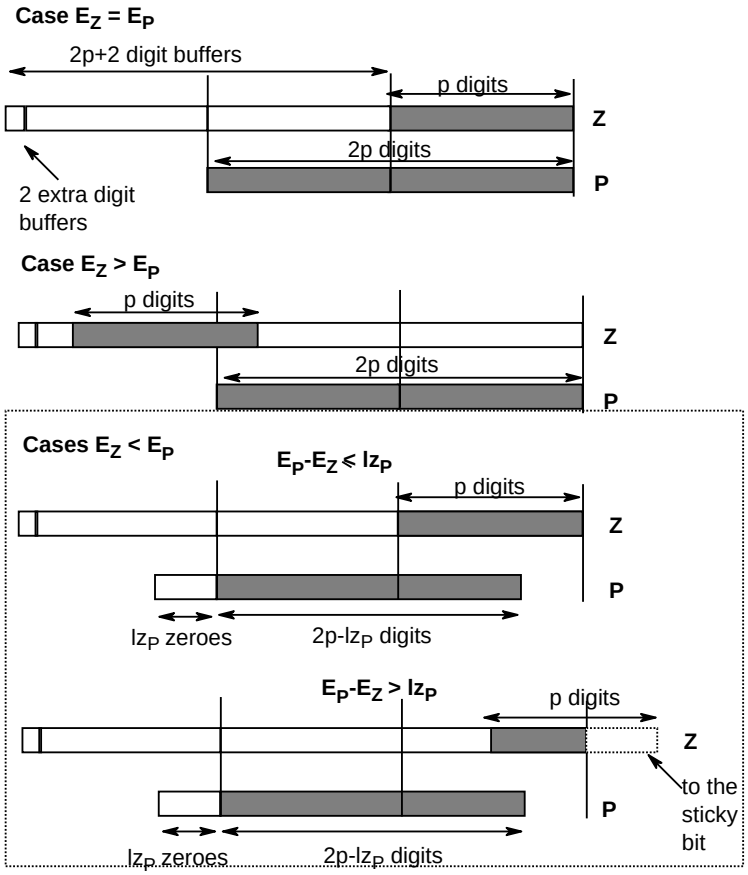


Figure 7.14. *Operand alignment for the decimal FMA operation.*

of R. Note that if $lz_P > E_P - E_Z > p$, then the result is exact and it has $(lz_P - p$ leading zeroes. Therefore, to be conformed to decimal IEEE 754-2008 standard specifications, this normalization is a left shift of $2p + 2$ digits at most.

To combine decimal addition and rounding we perform the normalization before the addition/subtraction. The resultant scheme is shown in Fig. 7.15. The complement of Z for effective subtractions $eop = 1$ and the subsequent wide $3p + 2$-digit right shifter are placed in parallel with the DFX multiplier.

As in [90], a LZA (leading zero anticipator) is used to determine the normalization amount (number of leading zeroes of the addition/subtraction) from the decimal carry-save product and the operand Z (or $\neg Z$). A decimal 3:2 CSA reduces in parallel these three operands to a two decimal operand.

To reduce the delay of the decimal FMA, some part of the logic of the decimal adder (the initial operand setup, see Fig. 5.8) could be placed before the normalization and overlapped with the LZA to reduce the overall delay (similar to [90]). We only detail the high level structure. A more detailed description would require a gate-level analysis to balance the delay of the different paths.

Note that for the IEEE 754-2008 Decimal128 format (34 digits or 136 bits) the datapath

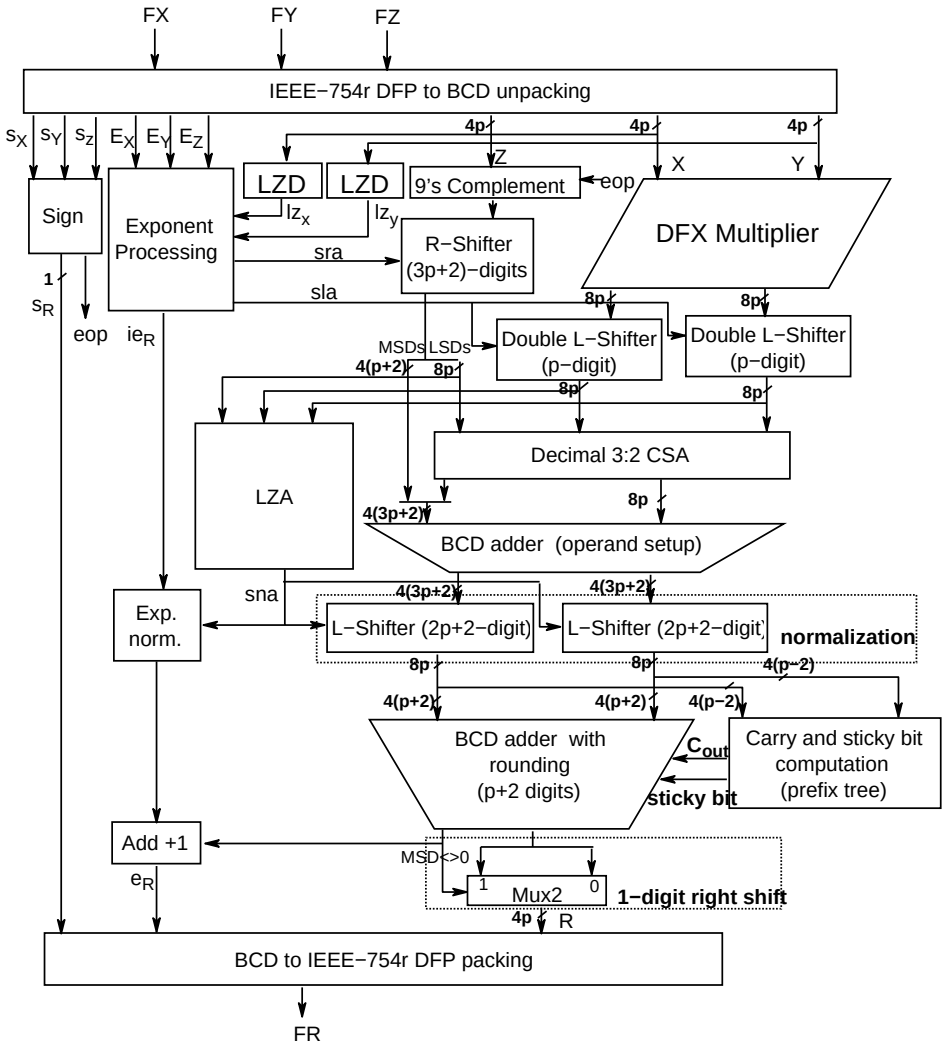


Figure 7.15. *Proposed high-performance scheme for decimal FMA operation.*

length of a parallel architecture is very huge for the standard of current implementations. For instance, the right shifter in Fig. 7.15 is of 576 bits (144 decimal digits), while commercial DFPUs (IBM Power6 [45] and z10 [160]) use a pipelined (two-stage) decimal rotator of 36 digits. Thus, the right shift of 144 digits could be performed sequentially in 4+1 cycles using a 36-digit rotator. Nevertheless, a parallel implementation of a decimal FMA could be possible in future technologies.

7.7 Evaluation results and comparison

We have used our area-delay evaluation model for static CMOS technology of Appendix A to estimate the area and delay figures of the proposed 16-digit (64-bit) fixed-point multipliers (decimal and combined binary/decimal) detailed in Section 7.5. We show these evaluation results in Section 7.7.1. We have also applied this model to compare our multipliers with other representative proposals for decimal fixed-point multiplication (sequential and parallel)

	SD Radix-10		SD radix-5	
Component	Delay (# FO4)	Area (#NAND2)	Delay (# FO4)	Area (#NAND2)
Multiplier recoding+buffering	4.8+3.3	450*	3.6+3.3*	550*
Generation of multiples	17.2*	2700*	3.5*	350*
Selection of multiples	3.8*	14850*	1.8	11500*
PPG stage	21.0	18000	8.7	12400
PPR tree	30.0	17000	38	27500
BCD operand's adder setup	4.2	1050	4.2	1050
128-bit BCD Q-T adder	14.1	3700	14.1	3700
Total stage	69.3	39750	65	44650

*These terms contribute to the area or the delay of PPG.

Table 7.4. *Area and delay for the proposed 16-digit BCD DFX multipliers.*

	Booth radix-4/SD Radix-4		Booth radix-4/SD radix-5	
Component	Delay (bin/dec) (# FO4)	Area (#NAND2)	Delay (bin/dec) (# FO4)	Area (#NAND2)
PPG stage	9.0/15.5	16500	10.5/11.0	16200
PPR tree	34/45	29500	34/45	29500
Operand's adder setup	4.4	1600	4.4	1600
Bin./BCD Q-T adder	14.1	3700	14.1	3700
Total stage	61.5/79	51300	63.0/74.5	51000

Table 7.5. *Area and delay for the combined binary/decimal architectures.*

and with two representative binary fixed-point parallel multipliers [108, 124]. In Section 7.7.2 we present the results of these comparisons.

7.7.1 Evaluation results

The area and delay figures for the 16-digit SD radix-10 and SD radix-5 DFX multipliers described in Section 7.5.1 and Section 7.5.2 are shown in Table 7.4.

The partial product generation (PPG) includes the recoding of the multiplier and the generation and the selection of multiples. For the SD radix-10 architecture, we provide the area and delay figures of an area-optimized implementation. Thus, the delay of the partial product reduction (PPR) is the critical path delay of the decimal $17:2$ CSA of Fig. 6.10 (the area-optimized design). For the SD radix-5 architecture, we opt for an area-delay tradeoff implementation. In this case, the delay of PPR corresponds to the critical path delay of the mixed (4221/5211) decimal 32:2 CSA of Fig. 6.13. The operand's BCD adder setup includes the $W = 2 \times H$ multiplication and the conversion of $S(4221)$ to BCD excess 6.

The evaluation results for the combined binary/decimal parallel multipliers are shown in Table 7.5. The delay figures are given as $\#FO4$ for binary/decimal for area-optimized implementations. When a combined binary/decimal implementation is required, the preferred

Architecture	Delay	Latency			Throughput	Area	
	#FO4	# Cycles	#FO4	Ratio	Mult./Cycle	NAND2	Ratio
Combinational multipliers:							
Bin. radix-4 [108]	51	1	51	1.00	1	43000	1.00
Bin. radix-8 [124]	57	1	57	1.15	1	39500	0.90
Dec. Ref. [91]	93	1	93	1.80	1	69000	1.60
Dec. SD Radix-5	65	1	65	1.25	1	45000	1.05
Dec. SD Radix-10	69	1	69	1.35	1	40000	0.90
BCD sequential multipliers:							
Ref. [51]	16	20	320	6.30	1/17	16000	0.40
Ref. [50]	14.7	20	294	5.80	1/17	18550	0.45
Ref. [82]	12.7	24	305	6.00	1/17	31500	0.75

Table 7.6. *Area-delay figures for 64-bit binary/16-BCD digit decimal fixed-point multipliers.*

option is the radix-5 architecture for low latency decimal multiplication (74.5 FO4 vs. 79 FO4) and the radix-4 architecture for low latency binary multiplication (61.5 FO4 vs. 63 FO4). With respect to the decimal SD radix-5 multiplier, the combined architectures have 14% more area and are between 15% and 20% slower for decimal multiplications.

7.7.2 Comparison

Table 7.6 shows the evaluation results for some representative sequential and combinational multipliers and the comparison ratios with respect to the binary Booth radix-4 multiplier.

So far, the other known implementation of a decimal fixed-point parallel multiplier is [91]. The recoding and the generation of partial products is similar to our SD radix-5 recoding scheme except that the 10's complement operation to obtain the negative BCD multiples $-2X$ and $-X$ is more complex than a simple bit inversion. Moreover, they require combinational logic to generate the $5X$ multiple. For 16 decimal digits, 32 partial products are generated. The partial product reduction tree uses seven levels of decimal CSAs (implemented by arrays of 4-bit decimal CLAs) in parallel with two levels of decimal digit counters. The final assimilation consists of a simplified direct decimal carry-propagate adder. Synthesis results given in [91] using a 90 nm CMOS standard cells library, show a critical path delay of 2.65ns (88 FO4) and an equivalent area of 68.000 NAND2 gates, while ratios are 1.90 for delay and 1.50 for area with respect to a radix-4 binary multiplier. Using our model we have obtained area and delay figures very close to their evaluation. We observe that our decimal multipliers have a speed-up between 1.25 and 1.40 with respect to [91] using at most 0.65 times its area.

To extract fair conclusions from the comparison between sequential and parallel implementations we have included the throughput of each multiplier. Sequential multipliers are more than two times smaller than parallel multipliers, but have higher latency and reduced throughput. For instance, the proposed SD radix-5 parallel multiplier is about 5 times faster than the best sequential implementation proposed in [51], but requires 2.8 times more area. In addition, it can issue a 16-digit BCD multiplication every cycle instead of one every 17 cycles.

7.8 Conclusions

We have presented several techniques to implement parallel decimal fixed-point multiplication in hardware. We have proposed three different SD encodings for the multiplier that lead to a fast and simple parallel generation of partial products in (4221) or (5211) decimal encodings. This makes possible the efficient reduction of all partial products using the proposed q:2 decimal CSA trees presented in Chapter 6. We have proposed two architectures for decimal SD radix-10 and SD radix-5 parallel multiplication and two combined binary Booth radix-4/decimal SD radix-4 and SD radix-5 fixed-point multipliers.

The area and delay figures from a comparative study including conventional parallel binary fixed-point multipliers and other representative decimal proposals show that our decimal SD radix-10 multiplier is an interesting option for high performance with moderate area. For higher performance the choice is the SD radix-5 architecture, although the SD radix-10 design has very close delay figures. For combined binary/decimal multiplications the choices are the Booth radix-4/SD radix-4 for low latency in binary multiplication or the Booth radix-4/SD radix-5 multiplier for low latency in decimal multiplication. Moreover, results can be further improved applying aggressive circuit and gate level techniques proposed for binary multipliers [109, 165].

Finally, we have proposed novel schemes for decimal floating-point multiplication and for decimal fused multiply-addition. The key components of these proposals are the BCD sign-magnitude adder with decimal rounding introduced in Chapter 5 and the SD radix-10 and SD radix-5 decimal fixed-point multipliers presented in this Chapter.

Chapter 8

Decimal Digit-by-Digit Division

We present the algorithm and architecture of a radix-10 floating-point divider based on a SRT non-restoring digit-by-digit algorithm. The algorithm uses conventional techniques developed to speed-up radix-2^k division such as signed-digit (SD) redundant quotient and digit selection by constant comparison using a carry-save estimate of the partial remainder. To optimize area and latency for decimal, we include novel features such as the use of alternative BCD codings to represent decimal operands, estimates by truncation at any binary position inside a decimal digit, a single customized fast carry propagate decimal adder for partial remainder computation, initial odd multiple generation and final normalization with rounding, and register placement to exploit advanced high fanin mux-latch circuits.

The Chapter is organized as follows: in Section 8.1 we outline the previous work on decimal division. A dataflow for decimal floating-point division is described in Section 8.2. In Section 8.3 we present the proposed algorithm and determine the selection constants and a suitable decimal digit encoding for a fast and a simple implementation of the selection function. In Section 8.4 we describe the architecture and the operation sequence of the divider for the preferred decimal encoding. We also detail the decimal adder used for residual assimilation, normalization and rounding. In Section 8.5 we present the area-delay evaluation results. We compare our design with two recent radix-10 SRT designs [92, 106] and with a software implementation [30]. In Section 8.6 we summarize the main conclusions of this work.

8.1 Previous work

Radix-10 division algorithms which are found in the literature are usually classified in multiplicative and restoring or non-restoring (SRT) digit-by-digit methods. Digit-by-digit methods have the characteristic of producing one digit per iteration. Good examples of radix-2^k based division algorithms can be found in [5, 47]. Recently, several algorithms have been proposed for decimal division [22, 83, 92, 106, 122] using a non-restoring digit-by-digit algorithm. These implementations outperform older restoring digit-by-digit proposals [19, 164], since they incorporate recent advances developed for binary division. On the other hand, decimal division based on multiplicative algorithms with quadratic convergence have been proposed in [26, 156]. These methods use a look-up table mechanism to obtain an initial approximation of

the decimal reciprocal of the divisor. Several Newton-Raphson iterations are then performed to obtain the reciprocal with the required accuracy. These iterations are computed using decimal multiply, square and subtract operations. A final decimal multiplication by the dividend is required to complete the division operation.

Regarding the digit-by-digit algorithms, two of the above mentioned designs are representative of the design space [92, 122]. Specifically, Schwarz and Carlough [122] proposed a low-cost implementation for the decimal floating-point unit of the IBM Power6 microprocessor. They use a radix-10 SRT algorithm with digit set $\{-5, \ldots, 0, \ldots, 5\}$ and prescaling of the operands so that the digit selection only depends on the residual. For a low cost implementation the recurrence is implemented with a carry-propagate BCD adder shared with the other floating-point operations (addition and multiplication among others). Therefore the residual has a non-redundant representation and the quotient digit is obtained by simple truncation.

Lang and Nannarelli [92] proposed a high-performance implementation. The radix-10 digits of the result are in the set $\{-7, -6, \ldots, 0, \ldots, 6, 7\}$ which are decomposed into two simpler digits to ease the implementation ($qu\ 5 + ql$, with $qu \in \{-1, 0, 1\}$ and $ql \in \{-2, -1, 0, 1, 2\}$). The implementation uses two overlapped stages with redundant carry-save decimal arithmetic to speedup the cycle time. Of particular interest is the use of redundant binary carry-save arithmetic in the most significant part of the datapath, which involves the digit selection, to have a cycle time comparable to a radix-16 binary implementation [93].

By other hand, the radix-10 SRT divider proposed by Nikmehr, Phillips and Lim [106] is also a high-performance implementation. They use the maximally redundant digit set $\{-9, \ldots, 0, \ldots, 9\}$ to represent both the quotient digits and the digits of the partial remainder. Thus, a decimal signed-digit adder is used to compute the partial remainder. The quotient digits are selected by comparing the truncated partial remainder with 18 selection constants. These constants are obtained from reduced precision multiples of the divisor. However, this implementation requires significantly more area than the radix-10 SRT divider proposed by Lang and Nannarelli [92] while the latency is not improved.

8.2 Decimal floating-point division

A IEEE 754-2008 DFP division $FQ = \frac{FX}{FD}$ is computed as shown in Fig. 8.1.

The DFP result (quotient) is of the form

$$FQ = (-1)^{s_Q}\ Q\ 10^{E_Q - bias} \tag{8.1}$$

and the DFP dividend and divisor have the following value:

$$\begin{aligned} FX &= (-1)^{s_X}\ X\ 10^{E_X - bias} \\ FD &= (-1)^{s_D}\ D\ 10^{E_D - bias} \end{aligned} \tag{8.2}$$

The biased exponent E_Q and the sign $s_Q = s_X \oplus s_D$ are computed separately from the coefficient quotient $Q = \frac{X}{D}$.

The coefficients of the DFP formats defined by the IEEE 754-2008 standard are decimal integer numbers not normalized (that is, they can have leading zeroes). However, to ensure

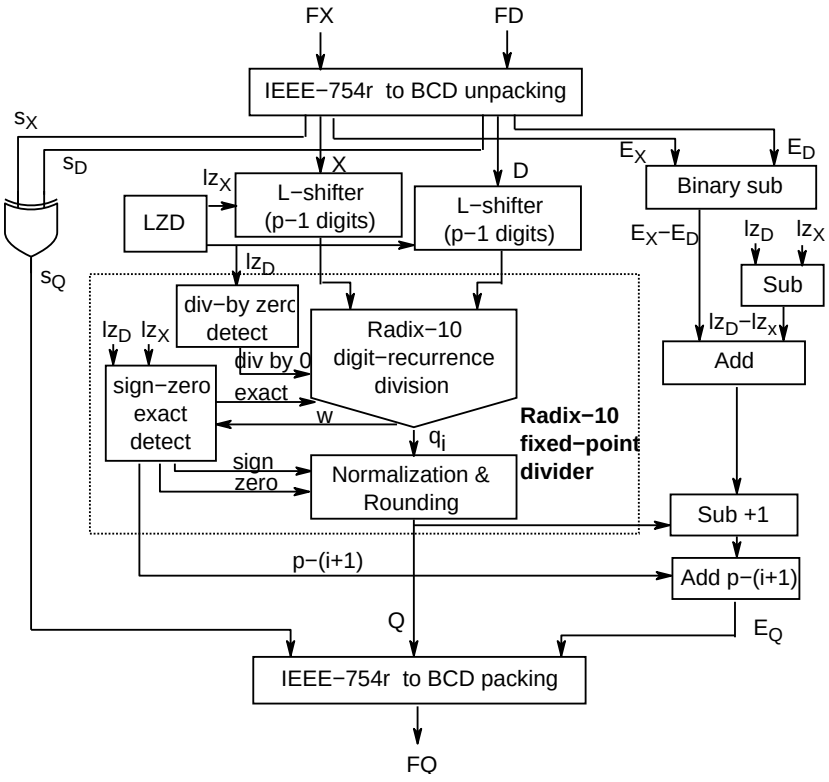


Figure 8.1. *Architecture of a IEEE 754-2008 DFP divider unit.*

convergence, the SRT digit-recurrence algorithms require fractional input operands normalized in a certain known range.

Therefore, a pre-processing stage is necessary to have the values of the BCD operands X and D normalized in the range $[1, 10)$. Operands X and D are shifted to the left an amount equal to the number of their leading zeroes, that is, lz_X and lz_D. The exponent E_Q is precomputed as $E_X - E_D + bias + lz_D - lz_X$. In case of exact result, E_Q is the closest possible exponent equal to the preferred exponent, $E_X - E_D + bias$.

The significand division of the normalized operands is performed in a decimal fixed-point digit-by-digit divider. We detail the algorithm and the architecture of this unit in Section 8.3 and Section 8.4 respectively. In case of division by zero ($lz_D = p$), the computation is stopped and an exception is signaled. The radix-10 fixed-point divider produces iteratively a quotient $Q \in [0.1, 10)$ of $p + 2$ digits (an extra round and guard digits are required). The quotient is normalized to the range $[1, 10)$ and rounded to p digits. The final reminder is used to compute the sticky bit and to determine when the computation is exact.

In case of exact result, we have to shift the normalized quotient $min(max(lz_D - lz_X, 0), tr_Q)$ positions to the right, where tr_Q is the number of trailing zeroes of Q. To avoid this right shift, we stop the computation when $i \geq p - (lz_D - lz_X) - 1 > 0$ if the partial remainder $w[i+1]$ is zero (exact result). The exponent is incremented an amount $p - (i + 1)$. This is computed on-the-fly together with the quotient digits q_i. Finally, the result is packed to an IEEE 754-2008 DFP format.

8.3 SRT radix-10 digit-recurrence division

The starting point of our proposal is the application of well-known radix-2^k methods to improve radix-10 division. Among these methods are the use of a symmetrical redundant digit set for the quotient digits ($-a \leq q_i \leq a$, with $a \in \{5, \ldots, 10\}$), quotient digit selection using estimates of the residual and the divisor and preloaded constants, and the use of a carry-save format to represent the residual.

This work is not just a simple transposition to radix-10 of radix-2^k methods, at least because while on one hand the algorithm operating radix is 10, on the other side, the representation index is binary. For this reason, it is necessary to bridge this gap and, in line of principle it is not ensured that the currently well known methodologies of the literature correspond to the best solution.

To adapt these radix-2^k techniques to radix-10 division we follow a different approach than [92] (implementation with the highest performance to date). Instead of splitting the digit selection and residual updating into two overlapped stages, we opt for a digit set, similar to the one used in [122], that minimizes the complexity of generating divisor multiples. Specifically, this work presents the following contributions:

• Implementation with a suitable digit set that minimizes the complexity of generating divisor multiples.

• A study of alternative BCD codings to represent decimal digit operands. The dividend X, divisor D and quotient Q require a decimal range-complement non redundant representation, while the residual $w[i]$ requires a redundant representation (carry-save or signed digit). The preferred coding leads to the simpler implementation of the selection function and divisor multiples generation in terms of area-latency trade-offs.

• Obtention of the selection constants as estimates of different truncated multiples of the divisor, avoiding tables.

• Design of a decimal adder to compute the required odd divisor multiples, the assimilation of the previous residual (in parallel with the selection of the next quotient digit), and normalization and rounding, thus sharing the same hardware for different parts of the floating-point division.

Before going to the representation issues, let us tackle the problem by exploiting the common points with radix-2^k based division algorithms, which are directly derived from the general concept of SRT non restoring division.

8.3.1 SRT non restoring division

The division algorithm is regulated by the following recurrence

$$w[i+1] = rw[i] - q_{i+1}\ D \tag{8.3}$$

and

$$Q[i+1] = Q[i] + q_{i+1}\ r^{-(i+1)} \tag{8.4}$$

where $w[i]$ is the partial remainder at iteration i, r is the radix of the algorithm, D is the divisor, q_{i+1} is the digit of the quotient with weight $r^{-(i+1)}$ and $Q[i]$ is the partial result after i

iterations. A convenient choice is to have the residual $w[i]$ in redundant representation, since this implies a simpler and faster updating hardware. For binary based division algorithms, common representations are carry-save and signed-digit.

The determination of the digit q_{i+1} is carried out by inspecting the value of the residual $w[i]$. The use of a redundant digit set for q_{i+1}, avoids full length comparisons and then, allows an estimate of the residual $\widehat{w[i]}$ and of the divisor $\widehat{D}$ to be used in place of the full precision residual $w[i]$ and divisor D. The choice of the representation used for the residual is directly related to the value of the truncation error, which, in turn, affects the working parameters of the algorithm, such as number of digits of residual and divisor to be considered for performing correct computations.

As the computations continue, it is possible on-the-fly (see [46] for binary representations) to convert the partial result $Q[i]$ into non-redundant form, in order to have the conventional representation of the final quotient value Q ready after just one additional iteration.

8.3.2 Decimal representations for the operands

The coding of decimal digits is a key issue to provide fast implementations. By extending to radix-10 the classical requirements of radix-2^k division, we see that:

- The dividend X, divisor D and quotient Q require a decimal nonredundant representation.
- The quotient digit q_{i+1} requires a signed-digit decimal (redundant) representation.
- The residual $w[i]$ requires a signed-digit or carry-save (redundant) representation, having the possibility to represent also negative values.

The choice of the type of representation for the residual is the most critical. By assuming a BCD 10's complement carry-save representation of the residual we observe that there are basically two alternatives for the representation:

- Digit carry-save: in this case the sum word is composed of BCD digits in the full range $\{0, ..., 9\}$, and the carry word is composed of digits in the range $\{0, 1\}$.
- Full carry-save: in this case, both the sum and carry words are composed of BCD digits in the full range $\{0, ..., 9\}$.

We use the second approach (full carry-save), since this allows to obtain faster division units with a small increase in hardware cost.

In addition, the use of an estimate $\widehat{w[i]}$ to obtain the quotient digits, instead of the full precision residual $w[i]$, introduces an error which plays a relevant role in the definition of the other implementation parameters of the algorithm. This error has to be the smallest as possible, related to the truncation position.

If we limit us by assuming that a truncation of the residual can occur only in correspondence of exactly the k-th fractional decimal digit, then it is easy to compute a bound for the truncation error due to the sum word, as 10^{-k}. It should be noted that this resulting error

is independent on the type of representation used for the decimal digits. Clearly, to have the truncation in correspondence of the decimal digits, except for the fact that it is straightforward, presents very low flexibility, especially if we consider that we have access to the single bits of each binary digit, and therefore theoretically the truncation point could be everywhere.

Let us relax this constraint and compute the error in case the truncation occurs inside a decimal digit. For this purpose, we define $-t$ the fractional bit position where the truncation occurs, and $-(k+1)$ the decimal weight of the decimal digit hosting the truncation. Then, excluding the trivial case $k = t/4$, since it corresponds with a decimal digit, we have three possible cases: $t - 4k = 3, 2, 1$.

Unfortunately, there are not only the fractional bits of the truncated residual, but also the need to be able to represent the integer part of the residual with the smallest number of bits. With respect to the problem of having a small truncation error, this is a symmetrical issue (but in the opposite direction), in the sense that in order to have a small number of integer bits, then it is necessary that the possibility to represent values has to be the largest as possible. A uniform behavior as binary number system would imply that each integer bit that we add on the left (integer part) it is expected that we almost double the range of values which can be represented.

Finally, there is also another aspect that has to be considered when examining a potential coding for representing decimal digits: the efficiency of the coding, defined as the number of different useful representations divided by the number of possible representations. For BCD this is a relatively small value and therefore it is more than justified to look for different representations with increased efficiency of representation.

In Table 8.1 we list the alternatives that we have explored, sorted by decreasing efficiency. Starting from left to right, here is the explanation of the different columns: in column 1 we report the coding weight of the 4 bits that we consider. In column 2 it is shown the efficiency. In columns from 3 to 7 we have the maximum truncation error (refereed only to the sum word, must be multiplied by $10^{-(k+1)}$), for different positions of the truncation point, i.e. values of $t - 4k$. Again, with reference to columns 3 to 7, to have $t - 4k = 0$ means that the truncation holds exactly in correspondence of decimal digit of weight 10^{-k}, $t - 4k = 1, 2, 3$ stand for 1, 2, or 3 bits on the right with respect to that point, and $t - 4k = 4$ (only for further reference) stands for a truncation in correspondence of decimal weight $10^{-(k+1)}$. On the rightmost part of the table, i.e. columns 8 to 12, we report the maximum possible integer value which can be represented in each case, when the number of integer bits is from 0 to 4 (i.e. a whole decimal digit).

It is easy to observe and also straightforward to prove mathematically, that the five columns, 3 to 7, are symmetrical to columns 8 to 12. This means that if a coding has good capabilities to be able to represent a "large" integer value with a small number of bits, then the same coding has large truncation error in some cases. For this reason, although a uniform behavior could be a good tradeoff, it is still not sufficient to justify a choice of a coding different from BCD (8421).

From the results of Table 8.1, we have chosen to analyze the codings with the largest values for efficiency and uniform error behavior. The attention has therefore fallen to (3321), (4221), (5211), being the latter the one with a uniform behavior, well balanced and with

Code	Effic.	Max. truncation error					Max. representable int.					not used codings
	$\times 16$	= 0	=1	=2	=3	=4	= 0	=1	=2	=3	=4	
8421	10	10	8	4	2	1	1	2	4	8	10	
3321	16	10	7	4	2	1	1	2	4	7	10	
4221	16	10	6	4	2	1	1	2	4	6	10	
5211	16	10	5	3	2	1	1	2	3	5	10	
4321	15	10	7	4	2	1	1	2	4	7	10	
5221	15	10	6	4	2	1	1	2	4	6	10	
5311	15	10	6	3	2	1	1	2	3	6	10	
4321	14	10	6	4	2	1	1	2	4	6	10	0111
5221	14	10	5	4	2	1	1	2	4	5	10	0111
5321	14	10	7	4	2	1	1	2	4	7	10	
5311	14	10	5	3	2	1	1	2	3	5	10	
6221	14	10	6	4	2	1	1	2	4	6	10	
5321	13	10	6	4	2	1	1	2	4	6	10	0111
6311	13	10	6	3	2	1	1	2	3	6	10	
6321	13	10	7	4	2	1	1	2	4	7	10	
5221	12	10	5	3	2	1	1	2	3	5	10	0111, 0011, 1011
5321	12	10	5	4	2	1	1	2	4	5	10	0111, 0110
6321	12	10	6	4	2	1	1	2	4	6	10	

Table 8.1. *Decimal digit encodings and their characteristics*

good capabilities to represent integer values. We have shown (see Chapter 6) that the (4221) and (5211) codes lead to efficient decimal carry-save adder implementations, which might be important for the implementation of the division algorithm. The implementations with (3321) codes will not be discussed since we have verified that they are less efficient. While doing the implementations we will then consider the (4221) and (5211) codes.

If the truncation of the residual occurs only in correspondence of exactly the k-th fractional decimal digit, then for digit carry-save representation, it is easy to determine a bound for the truncation error due to the carry word as $(1/9)10^{-k}$. In case the truncation occurs inside a decimal digit, we have $k = \lfloor t/4 \rfloor$ and the same expression as above still holds. Thus, we have to add $(1/9)10^{-\lfloor t/4 \rfloor}$ to the truncation error due to the sum word indicated in Table 8.1 to obtain the global error.

For full carry-save representation the truncation error due to the carry word is the same as the truncation error due to the sum word. Then, the values found in the Table 8.1 have to be multiplied by $2 \times 10^{-(k+1)}$ in order to obtain the exact value of the maximum truncation error.

8.3.3 Proposed algorithm

We assume that the dividend X, the divisor D and the quotient Q are in the range $[1, 10)$. The radix-10 division algorithm implements the following recurrence

$$w[i+1] = 10\ w[i] - q_{i+1}\ D \tag{8.5}$$

where $w[i]$ is the partial remainder at iteration i, D is the divisor and q_{i+1} is the digit of the quotient with weight $10^{-(i+1)}$. In order to converge, it is necessary to select q_{i+1} so that the

resulting residual is bounded by

$$-\rho D \leq w[i+1] \leq \rho D \tag{8.6}$$

where $\rho = a/9$ is the redundancy factor.

The main drawback of recurrence (8.5) is the generation of odd multiples of D. One simple approach consists of implementing the recurrence as two simpler overlapped recurrences of lower radix [92]. In this work we explore an alternative, a direct implementation of (8.5) with the minimally redundant set[23] $\{-5, \ldots, 0, \ldots, 5\}$ ($\rho = 5/9$). This choice minimizes the complexity of the generation of decimal divisor multiples ($\{-5D, -4D, \ldots, -D, 0, D, \ldots, 5D\}$) while having a single recurrence. We only need to compute the odd multiple $3D$ using a decimal carry-propagate adder. The other multiples can be generated with simple digit recodings.

Since we need a carry-propagate adder for multiple generation, we designed the algorithm and the architecture to reuse this adder. Specifically, the addition required by the recurrence (8.5) is implemented with this adder, so that we keep the residual in non-redundant form. To make the determination of q_{i+1} independent of this carry-propagate addition (which would result in a large cycle time) we perform the digit selection using an estimation of $w[i]$, $\widehat{w[i]}$, obtained from the leading digits of $10\ w[i-1]$ and $-q_i\ D$ (this is allowed by the redundancy of the digit set for q_{i+1}). From the point of view of the estimation, this is similar to the standard practice of keeping the residual in carry-save.

For convergence it is necessary to assure $-(5/9)D \leq w[0] \leq (5/9)D$. This is achieved by the following initialization

$$w[0] = \begin{cases} X/20 & \text{if } (X-D) \geq 0 \\ X/2 & \text{else} \end{cases} \tag{8.7}$$

and $q_0 = 0$. Under this initialization $0.05 \leq Q < 0.5$, so it is necessary to multiply the resultant quotient by 20 to produce the normalized quotient in the appropriate interval $[1, 10)$. For a p-digit precision rounded quotient the algorithm requires $p+2$ quotient digits q_i ($i > 0$) (including a guard and round digits). This initialization requires the computation of the sign of $X - D$, which is performed by the decimal carry-propagate adder. Moreover, the same adder is also used for the final conversion from redundant to non-redundant representation and rounding of the result quotient.

To convert the $p+2$ signed-digit quotient into the non-redundant p-digit rounded quotient, each q_i value is recoded as $q_i = -10\ s_i + q_i^*$ where

$$(s_i, q_i^*) = \begin{cases} (0, q_i) & q_i \geq 0 \\ (1, 10 - q_i) & else \end{cases} \tag{8.8}$$

$(s_i, q_i^*) \in (\{0,1\}, \{0, \ldots, 9\})$. This digit recoding is performed after each digit selection, so after $p+2$ iterations of (8.5) we have $Q^* = \sum_{i=1}^{p+2} q_i^*\ 10^i$ and $S = \sum_{i=2}^{p+3} s_i\ 10^i$, where $s_{p+3} = sign(w[p+2])$ is introduced to correct the quotient when the last residual is negative. The quotient is obtained as $Q = round[Q^* - 10 \cdot S]_p$, which requires a decimal subtraction and rounding to p digits according to the rounding specifications. This operation is performed by the decimal carry-propagate adder, adapted to include the rounding (see Section 8.4.4).

[23] This digit set is also used in the IBM Power6 decimal divider [122].

8.3.4 Selection function

As mentioned before, the quotient digit q_{i+1} is obtained from an estimation $\widehat{10w[i]}$ of $10w[i]$ by truncating $10w[i-1]$ and $-q_i\ D$ and input them to the digit selection. This estimation is compared to selection constants (m_k, with $k = -4, ..., 5$) which are dependent on the leading digits of D. Specifically,

$$q_{i+1} = k \quad \texttt{if} \ \ m_k \leq \widehat{10\ w[i]} < m_{k+1} \tag{8.9}$$

As in [92] we implement the selection function by comparing the estimation of the residual with each of the selection constants. Since the estimation is composed of two words, the comparisons are performed by obtaining the sign of the difference between the estimation and the selection constant. This is performed by a 3:2 reduction using a decimal carry-save adder and a decimal sign detector.

The method used to obtain the selection constants is well-known [92]. However, in this work, we introduce two innovations:

• New decimal codings to reduce the estimation error.

• Estimations by truncating at the bit level instead of digit level (number of bits multiple of four), to reduce the number of bits of the estimations.

In the following we briefly outline the procedure to determine the selection constants and number of bits of the estimations.

The starting point is the definition of the selection intervals for convergence. It is well-known that for digit-recurrence division [92], we may choose $q_{j+1} = k$, assuring convergence, if the partial remainder $10w[j]$ takes values within the interval $[L_k(D), U_k(D)]$, with

$$\begin{aligned} L_k(D) &= (k - \rho)\ D = (k - 5/9)\ D \\ U_k(D) &= (k + \rho)\ D = (k + 5/9)\ D \end{aligned} \tag{8.10}$$

As mentioned before, we use selection constants m_k with a finite number of fractional bits. The comparison of the estimate of $10w[j]$ with the constants has an error, and therefore certain conditions should be met to assure convergence. Specifically, for an estimation (positive) error in $10w[j]$ of $\Delta\epsilon_w$, and denoting by $h(m_k)$ the distance between m_k and the next lower value within the same precision granularity, the selection constants should verify the following conditions

$$\begin{aligned} m_k(D) - h(m_k) + \Delta\epsilon_w &\leq U_{k-1}(D) \\ L_k(D) &\leq m_k(D) \end{aligned} \tag{8.11}$$

Note that $m_k(D) - h(m_k)$ is the value of the upper limit for the estimate $\widehat{10w[i]}$ to select $q_{j+1} = k - 1$, and therefore, the corresponding upper bound in the value of $10w[j]$ (obtained adding the estimation error) should be less or equal to $U_{k-1}(D)$. Obviously, since the estimation error is never negative, the constant should be greater or equal to $L_k(D)$.

Therefore the following condition results for $m_k(D)$

$$L_k(D) \leq m_k(D) \leq U_{k-1}(D) + h(m_k) - \Delta\epsilon_w \tag{8.12}$$

To have a finite and practical set of constants we use also estimations of D (denoted by $\widehat{D}$, and with positive error bounded by $\Delta\epsilon_D$). Therefore we use the same constant for the interval of D $[\widehat{D}, \widehat{D} + \Delta\epsilon_D)$. Thus, condition (8.12) should be met along all the corresponding interval of D. For an estimation $\widehat{D}$, the worst case of (8.12) for the whole interval $[\widehat{D}, \widehat{D} + \Delta\epsilon_D)$ is

$$\begin{aligned} &\max\{L_k(\widehat{D}), L_k(\widehat{D} + \Delta\epsilon_D)\} \leq m_k(\widehat{D}) \leq \\ &\min\{U_{k-1}(\widehat{D}), U_{k-1}(\widehat{D} + \Delta\epsilon_D)\} + h(m_k) - \Delta\epsilon_w \end{aligned} \tag{8.13}$$

The maximum and minimum functions take different values depending on the sign of k. Specifically for $k \geq 0$ the condition results in

$$L_k(\widehat{D} + \Delta\epsilon_D) \leq m_k(\widehat{D}) \leq U_{k-1}(\widehat{D}) + h(m_k) - \Delta\epsilon_w \tag{8.14}$$

For $k < 0$ we have

$$L_k(\widehat{D}) \leq m_k(\widehat{D}) \leq U_{k-1}(\widehat{D} + \Delta\epsilon_D) + h(m_k) - \Delta\epsilon_w \tag{8.15}$$

Moreover, as pointed out before, m_k should be representable with the number of fractional bits used for the estimation of $10\ w[j]$.

Using (8.14) and (8.15), we determined the minimum required number of bits for the estimations of $10\ w[j]$ and D, and the intervals of possible constants values for m_k, under the following conditions:

- Range of the estimation of $10w[j]$: since for convergence $-10\rho D \leq 10w[j] \leq 10\rho D$, the range of the estimation is
$$-\Delta\epsilon_w - 10\rho D \leq \widehat{10w[j]} \leq 10\rho D \tag{8.16}$$
For D in the range $[1, 10)$ and $\rho = 5/9$, this leads to a range of the estimation of $-\Delta\epsilon_w - 500/9 \leq \widehat{10w[j]} \leq 500/9$.

- Decimal representation of $10w[j]$ and D: we considered codes BCD, (4221) and (5211). The truncation error for the estimates depends on the code and the weight of the position of truncation. Moreover, the value of $h(m_k)$ might be dependent of the value of the constant.

We obtained that by using (5211) coding, one less fractional bit is needed for the estimation of the residual compared to BCD or (4221). Moreover, the decimal carry-save adder is effectively implemented with the (5211) code. Therefore we use (5211) for representing the estimation of the residual $\widehat{w[i]}$ and the selection constants m_k. By other hand, the residual $w[i]$ is coded in (5421) (see Table 7.1) due to the fast conversion between (5421) and (5211) decimal codes and the efficient implementation of a decimal (5421) carry-propagate adder (similar to a BCD carry-propagate adder).

For (5211) representation, the selection constants are obtained from the leading 12 bits of D (one integer and two fractional decimal digits). The estimation of the residual requires 9 integer (including sign) and 6 fractional bits. The selection constants require the following number of bits (integer+fractional): m_1 and $m_0 \rightarrow$ 3+6, m_2 and $m_{-1} \rightarrow$ 5+6, m_3, m_4, m_{-2} and $m_{-3} \rightarrow$ 6+6, m_5 and $m_{-4} \rightarrow$ 7+6.

A straightforward implementation to obtain the constants consists in using a look-up table with the 12 bits of D as input. Our synthesis results indicate that this approach is very

costly in terms of area and time (we want to determine the constants in just one cycle). An efficient implementation is obtained by computing the constants as follows.

• Compute simple multiples of $\widehat{D}$ that lie within the interval $[L_k(D), U_{k-1}(D)]$. Since $\rho = 5/9$, the multiples may be computed as $(k - 0.5) \times \widehat{D}$. These multiples might be out (but close) of the required range for the constants since the requirements for the constant are more restrictive than just being in the interval $[L_k(D), U_{k-1}(D)]$. Therefore, the computed multiples must be perturbed to fit within the interval of possible selection constants. We have determined that the following schemes allow to obtain valid selection constants:

• For m_k with $k = 1, 2, 3, 4, 5$ the perturbation of the computed value $(k-0.5) \times \widehat{D}$ consist in rounding up to six fractional bits (note that since $\widehat{D}$ has eight fractional bits, $(k-0.5) \times \widehat{D}$ may have nine fractional bits). Since the constants should have six fractional bits, and the code is (5211), the least significant digit of the constant should have values 0, 2, 5 or 7. Therefore the rounding up of $(k - 0.5) \times \widehat{D}$ is done to have the least significant digit with values 0, 2, 5 or 7. Moreover, the sign detectors require the addition of $-m_k$, therefore it is necessary to complement the digits of m_k (bit inversion for 5211 code) and add a 1 in the least significant position. In Section 8.4.3 we show the detailed implementation of this method.

• For m_k with $k = -4, -3, -2, -1, -0$, we obtain $-m_k$ (a positive value) by truncating $(k - 0.5) \times \widehat{D}$ to six fractional bits and adding five to the resultant least significant digit. In Section 8.4.3 we also show the detailed implementation of this method.

Fig. 8.2 shows an instance (a few intervals of $\widehat{D}$) of the $\widehat{rw[j]}$ vs. $\widehat{D}$ space for $k = 5, -4$ illustrating these concepts. Therefore, we compute the constants using simple arithmetic methods, avoiding large and slow lookup tables.

8.4 Decimal fixed-point architecture

In this Section we present the divider architecture, which uses the (5421) code to represent the residual $w[i]$ and the (5211) code for the estimation of the residual and the selection constants. We only detail the architecture for significand (fixed-point) computation, since other issues of floating-point division were detailed in Section 8.2. We show both the datapath (including the decimal (5421) carry-propagate adder, the generation of the divisor multiples for the selection constants) and the implementation of the selection function. A description of the sequence of operations is also presented.

8.4.1 Implementation of the datapath

For a p-digit precision quotient, we need at least bit-vectors of length $l = (4p+6)$ bits (including a sign bit and 5 guard bits for the initial scaling by 20) to represent the decimal operands X, D, Q and $w[i]$ coded in (5421). We assume that X is unpacked to BCD and D to (5421) code. The architecture is shown in Fig. 8.3. The division unit consists mainly of a (p+2)-digit decimal (5421) adder and rounding unit (see Section 8.4.4), a block implementing the quotient digit selection (detailed in Section 8.4.3), a generator of divisor multiples coded in (5421) and a generator of $(k - 0.5) \times \widehat{D}$ multiples ($k = \{1, \ldots, 5\}$).

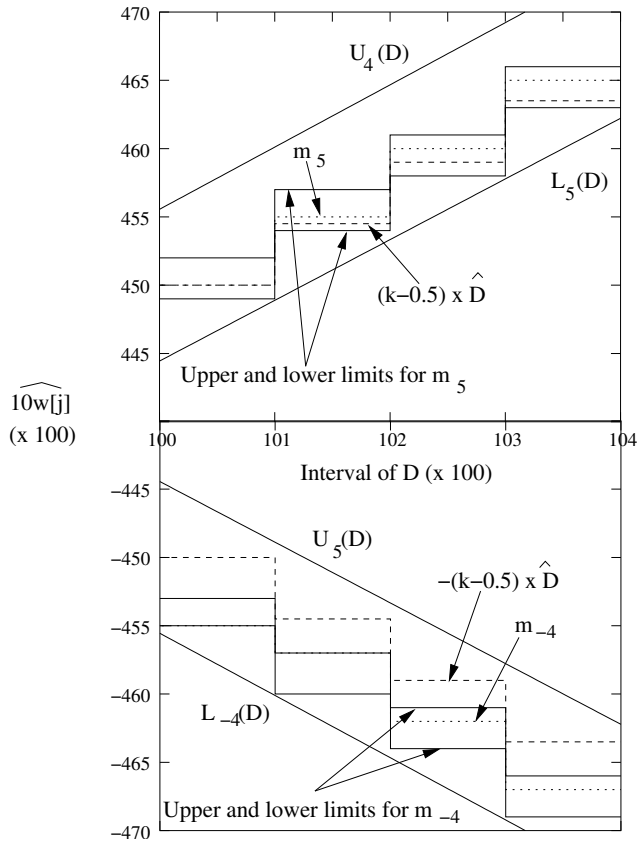


Figure 8.2. *Selection constants for some intervals of D for $k = 5$ and $k = -4$.*

Fig. 8.4(a) shows the generation of positive full length divisor multiples (D, $2D$, $3D$, $4D$, $5D$). These multiples are precomputed and are not needed until after the selection of the first quotient digit q_1. Since the divisor D is unpacked in (5421), the multiplication by 2 consists of a 1-bit wired left shift (it does not require logic) followed by a digit recoding from BCD to (5421) using combinational logic with no carry propagation between digits (see Section 8.3.1).

The $4D$ multiple is obtained by performing this sequence twice. Multiple $5D$ is generated first recoding from (5421) to BCD and then performing a 3-bit wired left shift. The generation of $3D = 2D + D$ requires a decimal carry-propagate addition that is performed in the (p+2)-digit decimal (5421) adder. The result is stored in a latch to be available for the next iterations.

In Section 8.3.4 we obtained that the selection function requires 15 leading bits of $-q_i D$ and so they need to be buffered. For a reduced latency implementation, the datapath (latches and multiplexes) for these leading bits is replicated and the sign bit of q_i is buffered and latched.

Fig. 8.4(b) shows the block to precompute the positive $Dk = (k - 0.5) \times \widehat{D}$ multiples ($k = 1, 2, 3, 4, 5$), required to obtain the selection constants $\{m_{-4}, \ldots, m_0, \ldots, m_5\}$ as described in Section 8.3.4. We obtain $D1 = 0.5 \times \widehat{D}$ ($k = 1$) and $D2 = 2.5 \times \widehat{D}$ ($k = 3$) coded in (5211) by shifting 1-bit to the right the representations of $\widehat{D}$ and $5 \times \widehat{D}$ in (4221) code. Since the 12-bit estimate of D is coded in (5421), we perform an initial digit recoding of $\widehat{D}$ to BCD followed by a second digit recoding from BCD to (4221), obtaining $\widehat{D}$(4221). By other hand, $5 \times \widehat{D}$(4221)

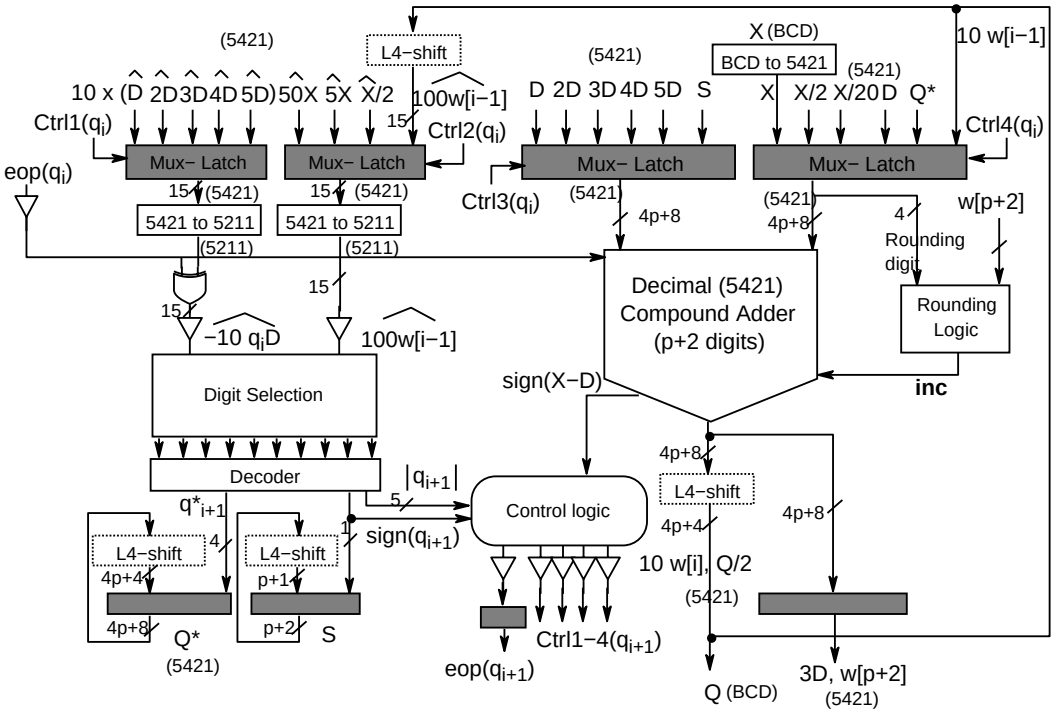
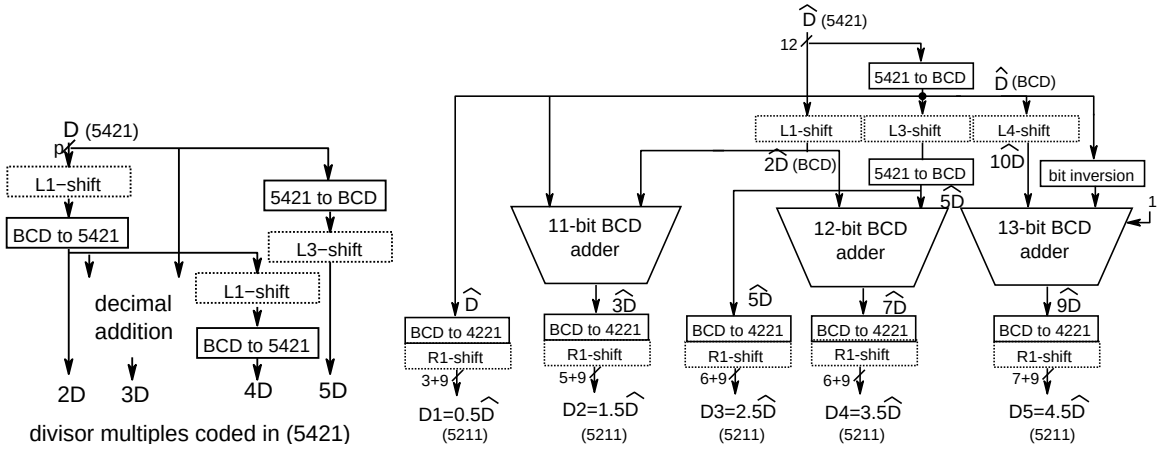


Figure 8.3. *Datapath for the proposed radix-10 divider.*



(a) Full length divisor multiples. (b) Multiples of $\widehat{D}$.

Figure 8.4. *Generation of D and $\widehat{D}$ multiples.*

results from a 3-bit left shift of $\widehat{D}$(BCD).

The other three (5211) decimal coded multiples Dk are obtained by a 1-bit right shift of $3\widehat{D}$ ($k = 2$), $7\widehat{D}$ ($k = 4$) and $9\widehat{D}$ ($k = 5$), coded in (4221). These values are computed using 11-bit, 12-bit and 13-bit BCD carry-propagate adders as $3\widehat{D} = 2\widehat{D} + \widehat{D}$, $7\widehat{D} = 5\widehat{D} + 2\widehat{D}$ and $9\widehat{D} = 10\widehat{D} - \widehat{D}$. The resulting BCD digits requires a digit conversion to (4221).

Finally, each selection constant $-m_k$ ($k = -4, \ldots, 0, \ldots, 5$) is obtained on-the-fly by per-

forming a perturbation of the corresponding precomputed multiple Dk inside each decimal comparator (see Section 8.4.3).

Other key components are the mux-latches that combine a wide multiplexer and a latch [107]. They are used to select and store the corresponding divisor multiples for the next iteration, the results coming from the decimal adder and the different values of initialization. The architecture was designed to fit the latches after the multiplexes to reduce latency. We now describe the operation of the divider.

8.4.2 Operation sequence

The sequence of operations of the proposed architecture is as follows:

• Cycle 1 (initialization). We assume $q_0 = 0$. We preloaded the narrow mux-latches with 0 and $\widehat{X/20}$ and the wide mux-latches with D and X. First, the decimal adder performs $X - D$. Then, the narrow mux-latch load 0 and $\widehat{X/20}$ (if $X - D \geq 0$, determined by examining the carry-out of the adder) or $\widehat{X/2}$ in other case. The wide mux-latches load $2D$ and D. Note that $X/2$ and $X/20$ coded in (5421) are obtained by shifting respectively 1 and 5 bits to the right the BCD operand X.

• Cycle 2 (recurrence iteration $i = 0$). The quotient digit q_1 is obtained decoded in a sign bit s_1, a (5421) digit q_1^* and a 5-bit signal $|q_1|$, which represents the absolute value of q_1. Depending on $|q_{i+1}|$, s_{i+1} and the cycle number, a control unit computes the hot-one code selection signals ($Ctrl1(q_{i+1})$ to $Ctrl4(q_{i+1})$) for the mux-latches, and an effective operation ($eop(q_{i+1})$) for the next cycle. In parallel, the decimal adder performs $2D + D = 3D$. The $3D$ multiple is stored in a dedicated latch. The wide mux-latches load $|q_1|D$ and $X/20$ ($X - D \geq 0$) or $X/2$ ($X - D < 0$). The narrow mux-latches load $\widehat{|q_1|D}$ and $\widehat{w[0]}$.

• Cycle 3 to $p + 3$ (iterations $i = 1$ to $i = p + 1$). In each iteration, the selection function performs $q_{i+1} = selection(\widehat{100\ w[i-1]}, \widehat{-10\ q_i D})$ while in parallel the decimal adder computes $w[i] = 10w[i-1] - q_i\ D$. At the end of the cycle $p+3$, the quotient digits (q_1^*, s_1) to (q_{p+2}^*, s_{p+2}) are available.

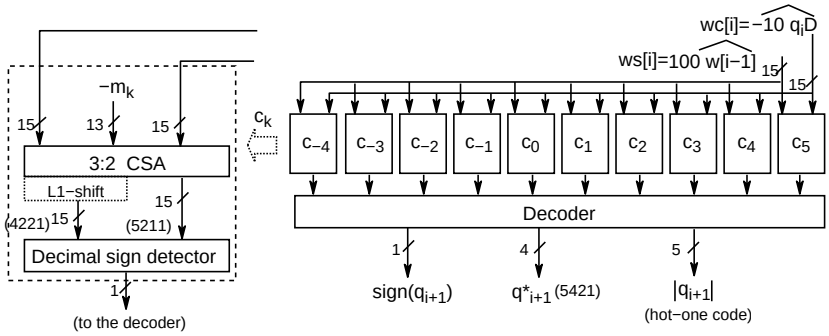
• Cycle $p + 4$. The residual $w[p + 2] = 10w[p + 1] - q_{p+2}\ D$ is assimilated in the decimal adder. This cycle could be avoided by implementing a sign and zero detector at the expense of additional hardware cost.

• Cycle $p + 5$. The decimal adder performs the subtraction $Q = Q^* - 10 \cdot S$ and rounds the result to p-digit precision (the round position is known). The rounding logic determines the +1 ulp conditional increment from the round digit, the rounding mode and the sticky bit. The (5421) quotient is multiplied by 20 (5-bit left wired shift) to produce the normalized decimal quotient Q in BCD. Note that only the most significant bit of q_{n+2}^* is needed, so the subtraction fits in the (p+2)-digit (5421) carry-propagate adder.

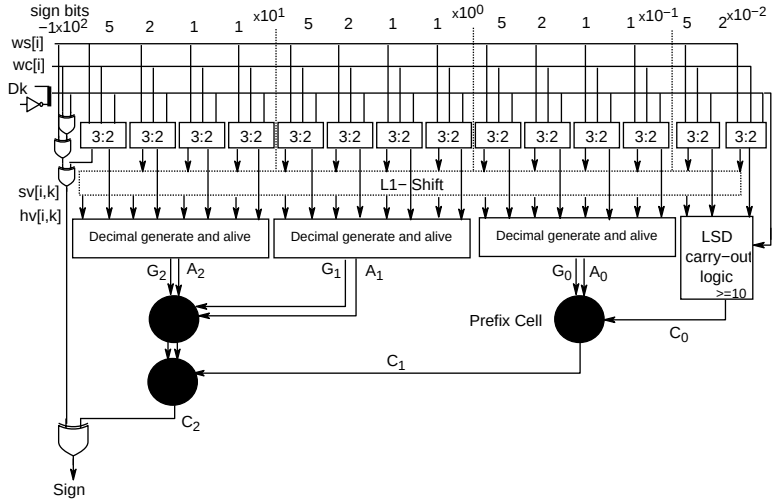
In summary, a p-digit decimal significand (fixed-point) division is performed in $p+5$ cycles. The significand division is completed in 21 cycles for the IEEE 754-2008 Decimal64 format (16 BCD digit operands), and in 39 cycles for the IEEE 754-2008 Decimal128 format (34 BCD digit operands). In the following Sections we detail the implementation of the selection function and the decimal adder.

8.4.3 Implementation of the selection function

Fig. 8.5(a) shows the detail corresponding to the selection function part. The selection func-



(a) Digit selection block diagram.



(b) Bit level implementation of a decimal comparator.

Figure 8.5. *Implementation of the digit selection function.*

tion is directly implemented using 10 decimal comparators. Each one compares the estimate of the residual $\widehat{10w[i]}$ coded as a decimal (5211) carry-save operand ($ws[i] = 10\widehat{0w[i-1]}$, $wc[i] = \widehat{-10q_iD}$) with one selection constant $-m_k$, $k = \{-4, \ldots, 0, \ldots, 5\}$. If $\widehat{10w[i]} \geq m_k$ then the output of the c_k comparator is 1. The value of q_{i+1} is obtained decoding the output of the comparators. The decoder produces a sign bit $s_{i+1} = sign(q_{i+1})$ and the absolute value of q_{i+1} ($|q_{i+1}|$) in hot-one code for the control signals of the mux-latches. It also provides the q^*_{i+1} quotient digits coded in (5421).

The implementation of one decimal comparator is shown in Fig. 8.5(b). Operands $ws[i]$, $wc[i]$ and $-m_k$, coded in (5211), have 6 fractional bits. However, instead of the 10 selection constants $-m_k$ $k = \{-4, \ldots, 0, \ldots, 5\}$, we precomputed the positive multiples $Dk = (k - 0.5) \times \widehat{D}$ ($k = \{1, 2, 3, 4, 5\}$). To obtain the correct result in the comparators, we truncate the values Dk to six fractional bits and then we perturb them as detailed in Section 8.3.4. This perturbation is incorporated into the logic to generate a decimal carry-out C_0 from the LSD position, as we show later.

First, a level of binary 3 to 2 CSAs reduces the three input operands to a two operand, a sum operand $sv[i,k]$, coded in (5211), and a decimal carry operand $hv[i,k] = 2\ h[i,k]$ coded in (4221), that is

$$ws[i] + wc[i] + Dk = sv[i,k] + hv[i,k] \tag{8.17}$$

The decimal carry operand $hv[i,k] = \sum_{j=-2}^{1} hv_j\ 10^j$ is obtained by a 1-bit left shift of the binary carry bit-vector $h[i,k]$. The digit $hv_2[i,k] \in \{0,1\}$ is a decimal carry into the sign position.

The decimal addition of $sv[i,k]$ and $hv[i,k]$ generates a carry-out C_2. The sign of the comparison is determined by the XOR of the sign bits of the input operands, $hv_2[i,k]$, and the decimal carry-out C_2. For each decimal position, we compute a decimal carry generate G_j and decimal carry alive A_j from the digits of $sv[i,k]$, coded in (5211), and $hv[i,k]$, coded in (4221). We use the notation $(sv_{j,3}, sv_{j,2}, sv_{j,1}, sv_{j,0})$ and $(hv_{j,3}, hv_{j,2}, hv_{j,1}, hv_{j,0})$ to represent the bits of $sv[i,k]$ and $hv[i,k]$. Fig. 8.6 shows the gate level implementation required to compute G_j and A_j. We express the digits $sv_j[i,k]$ and $hv_j[i,k]$ in two parts as $sv_j[i,k] = sv_{j,3}\ 5 + sv_j^L$

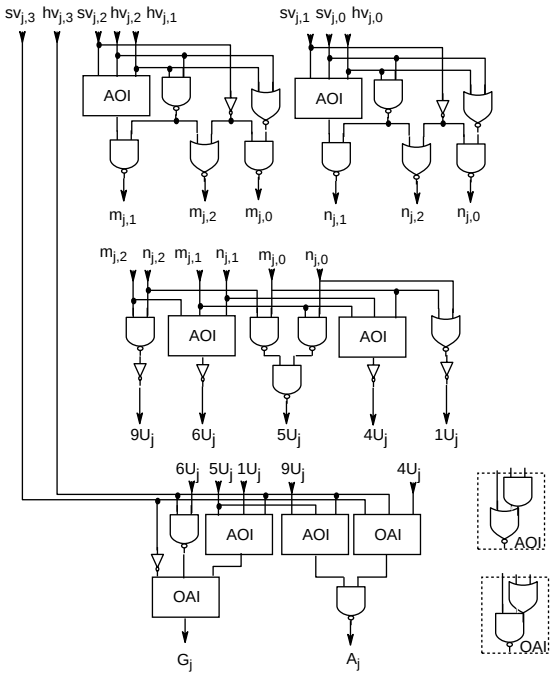


Figure 8.6. *Decimal carry generate (G) and alive (A) block.*

and $hv_j[i,k] = hv_{j,3}\ 4 + hv_j^L$, so that $sv_j^L + hv_j^L < 10$. From their definition, G_j and A_j can be computed as

$$\begin{aligned} G_j &= sv_{j,3}\ (hv_{j,3}\ 1U_i \vee 5U_i) \vee hv_{i,3}\ 6U_i \\ A_j &= sv_{j,3}\ (hv_{j,3} \vee 4U_i) \vee (hv_{j,3}\ 5U_i \vee 9U_i) \end{aligned} \tag{8.18}$$

where the boolean functions $nU_j,\ n \in \{1,4,5,6,9\}$ are true if $sv_j^L + hv_j^L \geq n$. To simplify the computation of these nU_j's, we define $sv_j^L + hv_j^L = M_j\ 2 + N_j$, where $M_j = m_{j,2}\ 2 + m_{j,1} + m_{j,0}$ with

$$m_{j,2} = sv_{j,2}\ hv_{j,1}\ hv_{j,0}$$

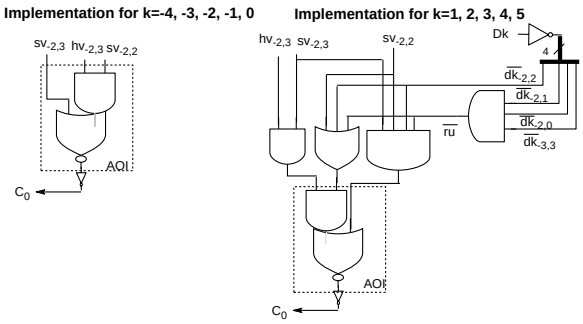


Figure 8.7. *Implementation of the LSD carry-out block.*

$$\begin{aligned} m_{j,1} &= (hv_{j,1}\ hv_{j,0}) \vee (hv_{j,1} \vee hv_{j,0})\ sv_{j,2} \\ m_{j,0} &= sv_{j,2} \vee hv_{j,1} \vee hv_{j,0} \end{aligned} \tag{8.19}$$

and $N_j = n_{j,2}\ 2 + n_{j,1} + n_{j,0}$ with

$$\begin{aligned} n_{j,2} &= sv_{j,1}\ sv_{j,0}\ hv_{j-1,3} \\ n_{j,1} &= (sv_{j,1}\ sv_{j,0}) \vee (sv_{j,1} \vee sv_{j,0}) \cdot hv_{j-1,3} \\ n_{j,0} &= sv_{j,1} \vee sv_{j,0} \vee hv_{j-1,3} \end{aligned} \tag{8.20}$$

The nU_j's are obtained from the bits of N_j and M_j as

$$\begin{aligned} 1U_j &= m_{j,0}\ n_{j,0} \\ 4U_j &= m_{j,1} \vee n_{j,1}\ m_{j,0} \\ 5U_j &= m_{j,0}\ n_{j,2} \vee m_{j,1}\ n_{j,0} \\ 6U_j &= m_{j,2} \vee m_{j,1}\ n_{j,1} \\ 9U_j &= m_{j,2}\ n_{j,2} \end{aligned} \tag{8.21}$$

Next, we compute C_2 as a direct decimal carry-propagation $C_{j+1} = G_j \vee A_j\ C_j$ (see Section 3.1) using a 2-level prefix tree. The decimal carry-out C_0 of the comparator c_k should be 1 when the sum of the bits of the LSD of $sv[i,k]$ and $hv[i,k]$ is equal or greater than 10, that is

$$sv_{-2,3}\ 5 + hv_{-2,3}\ 4 + sv_{-2,2}\ 2 \geq 10 \tag{8.22}$$

However, the perturbation of values Dk requires a slight modification of the logic to generate the carry-out C_0 (the block 'LSD carry-out logic' of Fig. 8.5(b)). The gate-level implementation of this block, shown in Fig. 8.7, is obtained as follows:

- For $k = \{-4, -3, -2, -1, 0\}$, we only have to sum +5 to the LSD of Dk truncated to six fractional bits, as shown in Section 8.3.4. Therefore, we have to detect the condition

$$(sv_{-2,3} + 1)\ 5 + hv_{-2,3}\ 4 + sv_{-2,2}\ 2 \geq 10 \tag{8.23}$$

This is implemented using combinational logic as

$$sv_{-2,3} \vee hv_{-2,3}\ sv_{-2,2} \tag{8.24}$$

- For $k = \{1, 2, 3, 4, 5\}$ we have to round up Dk to six fractional bits and obtain its 10's complement. Thus, we have to calculate first a round up bit **ru** examining the three least significant bits of Dk discarded, that is, $\mathbf{ru} = dk_{-2,1} \vee dk_{-2,0} \vee dk_{-3,3}$. To perform the round up of Dk to 6 fractional bits, $Dk(ru)$, we have to add $ru\ 2 + (dk_{-2,2}\ ru)$ to the LSD of Dk truncated to six bits. If we represent the LSD of $Dk(ru)$ in (5221) code as $(dk_{-2,3}, dk_{-2,2}, ru, dk_{-2,2}\ ru)$, then the 10's complement of $Dk(ru)$ is obtained inverting its bits. Observe that the weight bits (5221) sum 10, so $\overline{Dk(ru)_{-2}} = 10 - Dk(ru)_{-2}$. Summarizing, the constants $-m_k$ ($k = \{1, 2, 3, 4, 5\}$) are obtained inverting the bits of Dk before the binary 3:2 carry-save addition, and adding a value $\overline{ru}\ 2 + \overline{Dk_{-2,2}\ ru}$ to the LSD position. Therefore, the condition for the generation of C_0 is stated as

$$sv_{-2,3}\ 5 + hv_{-2,3}\ 4 + (sv_{-2,2} + \overline{ru})\ 2 + \overline{dk_{-2,2}\ ru} \geq 10 \tag{8.25}$$

This is implemented at logic level as

$$(sv_{-2,3}\ hv_{-2,3})(sv_{-2,2} \vee \overline{ru} \vee \overline{dk_{-2,2}}) \vee sv_{-2,3}\ sv_{-2,2}\ \overline{ru}\ \overline{dk_{-2,2}} \tag{8.26}$$

8.4.4 Implementation of the decimal (5421) coded adder

The algorithm for decimal addition is based on the conditional speculative method (see Section 3.2), but with digits in a (5421) code instead of the more conventional BCD. We use the (5421) decimal code since recoding from (5421) to (5211) is much simpler than from BCD to (5211).

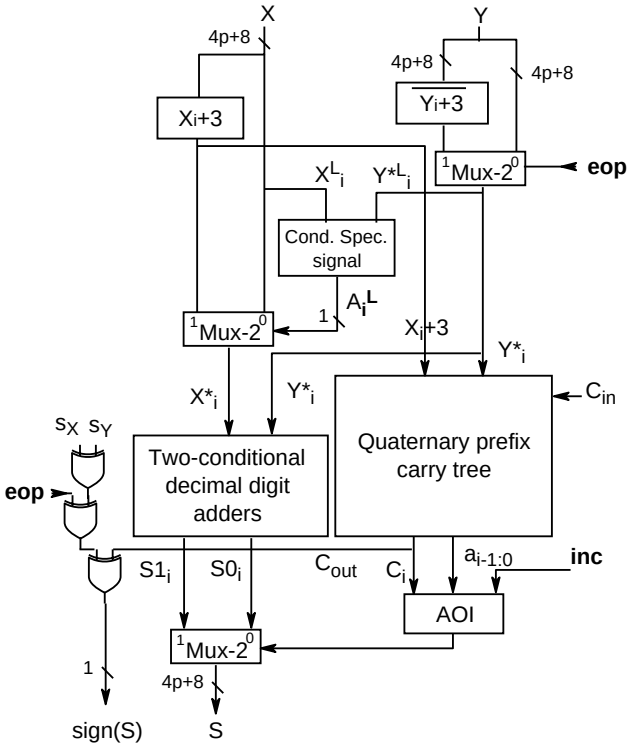
The implementation of the hybrid carry-select/prefix quaternary-tree for signed operands is shown in Fig. 8.8. The sign bits of the input operands are represented as s_X, s_Y, while X and Y are the (p+2)-digit magnitudes coded in (5421). The adder performs both two-operand 10's complement additions and subtractions $S = X + (-1)^{eop} Y$. The digits of X are incremented +3 units. This operation is carried out as a digit recoding from (5421) to (5421) excess-3 of one input operand. This allows the computation of the decimal carries using conventional 4-bit binary carry-propagate additions, since decimal carries are equal to the binary carries at decimal positions. The decimal carries C_i and the carry-alive groups $a_{i-1:0}$ are computed in a quaternary prefix tree. The complement operation for subtractions is performed by inverting the bits of Y coded in (5421) excess-3 and setting up the carry input to 1. The resultant operand Y^* is coded in (5421).

As in the conditional speculative decimal addition method of Section 3.2, we compute a control signal A_i^L for each digit position i given by

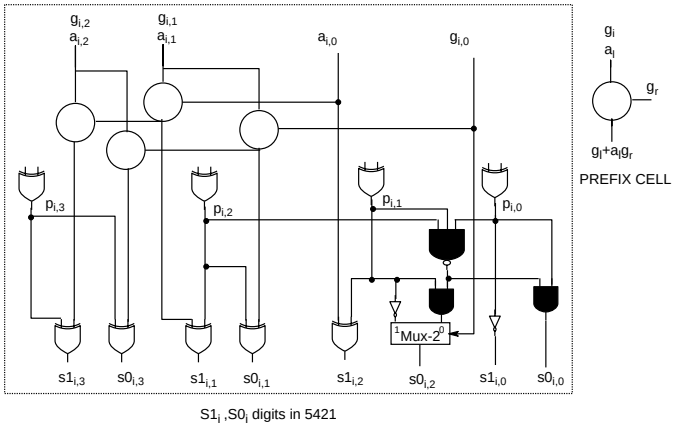
$$A_i^L = \begin{cases} 1 & \texttt{If } X_i^L + (Y_i^*)^L \geq 5 \\ 0 & \texttt{Else} \end{cases} \tag{8.27}$$

where X_i^L, $(Y_i^*)^L$ represents the 3 least significant bits of digits X_i and Y_i^*. This signal indicates when the digit X_i should be incremented +3 ($X_i^* = X_i + 3\ A_i^L$) to obtain the correct decimal sum digit S_i coded in (5421). This speculation fails when $X_i^* + Y_i^* \in \{7, 12\}$ ('0111', '1111' in (5421)) and the late carry LC_i is zero.

The presum digits $S1_i = X_i^* + Y_i^* + 1$, $S0_i = X_i^* + Y_i^*$ are computed in the 4-bit two-conditional decimal (5421) adder of Fig. 8.8(b). The sum digit $S0_i$ must be corrected when $S1_i \in \{7, 15\}$. The correct decimal sum values are $S0_i \in \{4, 9\}$ ('0100', '1100' in (5421)). The decimal correction is performed by the black gates of Fig. 8.8(b).



(a) Q-T decimal adder (digit slice).



(b) Two-conditional decimal digit sum cell.

Figure 8.8. *Diagram block of the Q-T decimal (5421) adder.*

Sum digits S_i (coded in (5421)) are obtained as $S_i = S1_i\ LC_i \vee S0_i\ \overline{LC_i}$, where the decimal late carries are given by $LC_i = C_i \vee a_{i-1:0}\ inc$ using the level of And-Or-Inverter (AOI) gates placed after the prefix tree, where signal inc controls a request from the rounding logic to increment the sum. This allows for a +1 conditional increment of the result for rounding operations.

By other hand, the sign bit of the result is computed as $sign(S) = s_X \oplus s_Y \oplus C_{out} \oplus eop$ (without overflow).

Block	Area (NAND2)	Stage delay (FO4)
IEEE 754-2008 Decimal64 (16 digits) Cycles/division= 21		
Sel. function (Fig. 8.5)	2800	22.3
Mult. generator (Fig. 8.4)	2000	18.5
Adder datapath (Fig. 8.8)	2600	17.5
Mux/latches	2700	3.0
Total	10100	25.3*
IEEE 754-2008 Decimal128 (34 digits) Cycles/division= 39		
Sel. function (Fig. 8.5)	2800	22.3
Mult. generator (Fig. 8.4)	4500	21.5
Adder datapath (Fig. 8.8)	5800	20.5
Mux/latches	6000	3.0
Total	19100	25.3*

*Critical path delay (sel. function +mux/latch)

Table 8.2. *Delay and area of the proposed divider.*

8.5 Evaluation and comparison

In this Section we show the area and delay figures estimated for the proposed architecture using our evaluation model for CMOS technology (see Appendix A). We have compared it with two recent proposals of decimal dividers based on a radix-10 digit-recurrence algorithm [92, 106] and with the radix-10 divider implemented in the IBM Power6 [122]. As a reference, we include the division latency figures of a DFP software library [30].

8.5.1 Evaluation results

Table 8.2 summarizes the evaluation results for the IEEE 754-2008 Decimal64 (16 precision digits) and Decimal128 (34 precision digits) dividers. Note that these figures correspond to the architecture for significand radix-10 division (Fig. 8.3). In both 16-digit and 34-digit implementations, the evaluation of the selection function (critical path) determines the latency of the divider. The area and delay of this block is independent of the widths of the operands, so the latency of the proposed divider is given by $(p+5) \times 25.3$ FO4, where $p \in \{16, 34\}$ is the number of precision digits. The adder datapath includes the decimal (5421) carry-propagate adder and the logic for rounding.

The area of the divider can be reduced between 25% and 30% by reusing an existing decimal $p+2$-digit carry-propagate adder with logic for rounding. However, commercial units [19, 20, 45, 122] do not implement decimal (5421) adders but BCD adders. For instance, the IBM Power6 DFPU includes a 36-digit BCD adder. To reuse this BCD adder, we should modify the proposed divider by coding in BCD the dividend X, the divisor multiples $\{-5D, \ldots, 0, \ldots, 5D\}$ and the residual $w[i]$. For a fast implementation of the selection unit, the estimate of the residual and the selection constants should be coded in (4221). This incre-

Divider	Cycle time (FO4)	Cycles #	Latency (FO4 /Ratio)	Area (NAND2/Ratio)
IEEE 754-2008 Decimal64 (16 digits)				
Proposed (Fig. 8.3)	25.3	21	531/1.00	10100/1.00
Ref. [92]	26.8	20	536/1.00	13500/1.35
IBM Power6 [122]	13	82	1066/2.00	6600/0.65
Ref. [30]*	≈ 22	294	6468/12.20	–
IEEE 754-2008 Decimal128 (34 digits)				
Proposed (Fig. 8.3)	25.3	39	987/1.00	19100/1.00
Ref. [106]	35.8	37	1325/1.35	48100/2.50
IBM Power6 [122]	13	154	2002/2.00	15600/0.80
Ref. [30]*	≈ 22	679	14938/15.10	–
*Software library running in an Itanium 2 @ 1.4 GHz				

Table 8.3. *Comparison results for area-delay.*

ments in one bit the length of the path for the selection of the quotient digits. In addition, the decimal comparator for operands coded in (4221) is slightly slower (about 3%) than the proposed comparator for (5211) coded operands of Fig. 8.5(b).

8.5.2 Comparison

Table 8.3 presents the area-delay figures and ratios for the architectures analyzed. We also include the delay figures of a software implementation of IEEE 754-2008 DFP division [30].

Since the area-delay figures for the Decimal64 ($p = 16$) divider proposed in [92] are comparable to ours we have estimated its critical path delay using our area-delay model in order to provide fair comparison results. The hardware complexity in NAND2 gates was provided by the authors. For the other reference [106], we use the figures provided by its synthesis results for Decimal128 ($p = 34$) and expressed in terms of equivalent FO4 gate delays and number of equivalent NAND2 gates.

The radix-10 division algorithm implemented in the IBM Power6 [122] uses a similar recurrence and the same divisor multiples than our proposal. Basically, they differ in the selection of the quotient digits and that the IBM Power6 decimal divider keeps the partial remainder in a non-redundant form. Thus, to obtain (roughly) the hardware cost of the IBM Power6 decimal divider we considered the area occupied by a (p+2)-digit BCD adder, a generator of decimal divisor multiples and the mux-latches for selection of the divisor multiples.

From this comparison we conclude that our proposal is comparable in terms of latency to the best up-to-date implementation [92] and is more advantageous in terms of hardware complexity (about 1.35 area ratio). Moreover, our implementation for Decimal128 requires less than half the area of the Decimal128 radix-10 divider proposed in [106] and presents an speedup of 1.35. In addition, we obtain a speedup of 2 with respect to the decimal division of the IBM Power6 processor at the cost of about 20% more area. Finally, observe that the software implementation analyzed [30] is an order of magnitude slower than the SRT radix-10 hardware implementations.

With respect to hardware implementations based on multiplicative algorithms such as Newton-Raphson [26, 156], comparison is difficult, since different issues should be taken into account: the use of a serial or a parallel decimal multiplier, the reuse of an existing multiplier or replication and the impact on the performance of other instructions that share the same multiplier. However, since efficient decimal parallel multipliers have been recently reported [148], we expect that the design decisions for decimal division are close to those considered for binary division.

8.6 Conclusions

We have presented the algorithm and architecture of a decimal division unit. The proposed radix-10 algorithm is based on the SRT digit-recurrence methods with a minimally redundant signed-digit set (ρ=5/9). The resultant implementation combines conventional methods used for high-performance radix-2^k division (SD redundant quotient and digit selection using selection constants and an estimate of the carry-save residual) with novel techniques developed in this work to exploit the particularities of radix-10 division. Among these new techniques are the use of non-conventional decimal codings to represent decimal operands, estimates by truncation at any binary position of a decimal digit and a customized decimal adder for several computations. We have designed 16 and 34 decimal digit dividers that fit the IEEE 754-2008 decimal64 and decimal128 formats. Evaluation results show that our 16-digit implementation presents comparable delay figures with respect to the best up-to-date implementation of a radix-10 digit-recurrence divider [92] but using less hardware complexity (1.35 area ratio). Moreover, the proposed architecture can be efficiently implemented in current commercial DFUs to reduce $\times 2$ the latency of the decimal divisions at the cost of about 20% more area.

Conclusions and Future Work

We have presented a new family of high-performance decimal fixed and floating-point units which implement the decimal arithmetic operations of most frequent use in numerical processing. These units are based on novel algorithms developed to improve the efficiency (performance, hardware cost and power consumption) of previous high-performance decimal units.

We detailed the functionality and the gate-level implementation of the different architectures. Moreover, we estimated the hardware costs and latencies using an area-delay evaluation model for static CMOS technology independent of the scale of integration. Specifically, the main contributions of this thesis are the following:

1. A new carry-propagate algorithm to compute 10's complement BCD additions and subtractions (Chapter 3). We have implemented the proposed method using different high-performance binary Ling and prefix tree adders, only requiring, in addition to the binary adder, a fast (constant delay) decimal pre-correction and a minor modification of the binary sum cell. This leads to very efficient implementations of combined 2's complement binary/10's complement decimal integer or fixed-point adders, and quite competitive with other academical and commercial proposals.

2. A new method and a high-performance architecture to compute sign and magnitude BCD addition/subtraction (Chapter 4). This operation is used in many applications, such as decimal floating-point addition to add/subtract the BCD coefficients. Unlike other proposals which use a decimal post-correction, the BCD magnitude result is directly obtained from a binary sign-magnitude addition after a slightly more complex pre-correction stage. Hence, the proposed architecture presents better area and delay than these previous implementations.

3. A unit to detect soft errors in 10's complement and mixed 2's/10's complement adders (Chapter 3) and in sign-magnitude BCD adders (Chapter 4). These units present a reduced hardware complexity (half the area) of other solutions used in commercial microprocessors (such as replication of arithmetic units), while it does not affect to the performance of the processor (unlike parity checking).

4. A new method and the architecture for merging significand (sign-magnitude) BCD addition and IEEE 754-2008 decimal rounding (Chapter 5). This is of interest to improve the efficiency of high-performance DFP adders, DFP multipliers and decimal FMA (fused-multiply-add) units. The IEEE 754-2008 Decimal64 (16 precision digits) and Decimal128

(34 precision digits) implementations present speedups of about 15% in performance while the area of the proposed unit is reduced more than 25% with respect to the best performance significand BCD adder and decimal rounding unit reported to date. This significant reduction in area and latency comes from a simplification of the logic required for decimal post-correction and rounding. The proposed method implements a pre-correction stage of reduced constant delay to allow the computation of both significand BCD addition and decimal rounding using a binary compound adder and little additional hardware.

5. An algorithm to improve multioperand decimal carry-free addition (Chapter 6) based on a binary carry-save addition over decimal operands represented in unconventional (4221) and (5211) decimal encodings. This makes possible the construction of multioperand decimal CSA trees that outperform the area and delay figures of existing proposals and only have 20% more area than a equivalent binary CSA tree. This makes viable the implementation of efficient commercial units which use these kind of structures to increase their performance, such as decimal parallel multipliers and FMA (fused multiply-add) units.

6. Several techniques for parallel decimal fixed-point multiplication (Chapter 7). We have proposed three different schemes for fast and parallel generation of partial products coded in (4221) or (5211) decimal encodings. This makes possible the efficient reduction of all partial products using the proposed multioperand decimal CSA trees.

7. Two architectures for decimal parallel multiplication and two combined binary/decimal fixed-point multipliers (Chapter 7). The decimal SD radix-10 multiplier is an interesting option for high performance with moderate area. For higher performance a better alternative is the SD radix-5 architecture. For combined binary/decimal multiplications the choices are the Booth radix-4/SD radix-4 for low latency in binary multiplication or the Booth radix-4/SD radix-5 multiplier for low latency in decimal multiplication. These architectures present better latency and less area than the only parallel decimal fixed-point multiplier previously reported.

8. Novel schemes for decimal floating-point multiplication and for decimal fused multiply-addition (Chapter 7). The key components of these designs are the proposed BCD sign-magnitude adder with decimal rounding and the SD radix-10 and SD radix-5 decimal fixed-point multipliers.

9. The algorithm and the architecture of a radix-10 digit-by-digit division unit with a minimally redundant signed-digit set (Chapter 8). The resultant implementation combines conventional methods used for high-performance radix-2^k division with novel techniques developed in this work to exploit the particularities of radix-10 division, namely, the use of (5421) and (5211) decimal codings to represent operands, an estimate of the residual by truncation at any binary position of a decimal digit and a customized decimal adder reused for several computations. Our unit is comparable in terms of latency with the radix-10 digit-recurrence divider that presents the highest performance to date, but requires 30% less area.

Our future work will be focused on the following issues:

1. Implement a delay optimized version of the significand BCD adder with rounding of Chapter 5. A further objective will be the integration of this adder in the DFU (decimal floating-point unit) of a high-performance microprocessor.

2. Optimize the decimal fixed-point parallel multipliers to provide pipelined implementations that fit adequately in the dataflow and cycle time of current commercial DFUs. Part of this work has already been done thanks to a research project funded by IBM and a five-month on-site collaboration with a design team of IBM Boeblingen (Germany).

3. Provide a detailed implementation of the DFP multiplier and decimal FMA designs presented in Chapter 7 for IEEE-754 Decimal64 and Decimal128 formats.

4. Extend the radix-10 digit-by-digit divider of Chapter 8 to support also decimal square-root computations.

5. Implementation of transcendental functions (exponentials, logarithms, trigonometric functions, ...) using decimal CORDIC units [146]. These units are a low-cost solution and therefore a feasible option to incorporate these operations in current DFUs. Moreover, these functions are recommended to be implemented correctly rounded in the new standard for floating-point arithmetic [74].

Appendix A

Area and Delay Evaluation Model for CMOS circuits

This Appendix describes the evaluation model used for area and delay estimations of CMOS circuits. The delay model is based on a simplification of the logical effort method [131] that allows for faster hand calculations. It is independent from the technology and only depends on the transistor level design of the components. The main characteristics of this delay model are the following:

- The path delay is obtained as the sum of the delays of the gate stages in this path. The total stage delay is given by the critical path delay assuming equal input and output capacitances. This total stage can be either a pipelined stage or the whole design.
- Delay values are given in FO4 units (delay of an minimum sized (1x) inverter with a fanout of four 1x inverters).
- It does not consider the delay of the wiring interconnect [68].
- The delay of a gate is obtained as the sum of the parasitic delay (a function of the gate's intrinsic internal capacitance) and the effort delay (due to the load on the gate's output). The effort delay is the product of the logical effort (depends on the logic gate's topology that drives the load) and the electrical effort (the ratio between the output load and the input gate capacitance).
- For a given path with fixed number of gate stages, the minimum path delay is achieved when each gate stage in the path has the same stage effort f_i. The optimum number of gates $\hat{N}$ in a path for minimum delay depends on the total path effort F. For practical purposes, we estimate that the optimum number of gate stages is $\hat{N} = \log_\rho F$ with $\rho \approx 4$.
- It does not consider gate sizing optimizations. The logical effort method uses gate sizing to equilibrate the gate efforts and thus minimize the path delay. Instead, we assume gates with the drive strength of the 1x inverter (minimum sized gates) to simplify the calculation of the total stage delay. To ensure that minimum path delays are close to the optimum values, we use other optimization techniques, such as buffering or cloning (gate replication) to drive high loads.

The area of a design is the sum of the area of their components. The cost related to the area of a gate is a function of the number of transistors and its size (width). To measure the

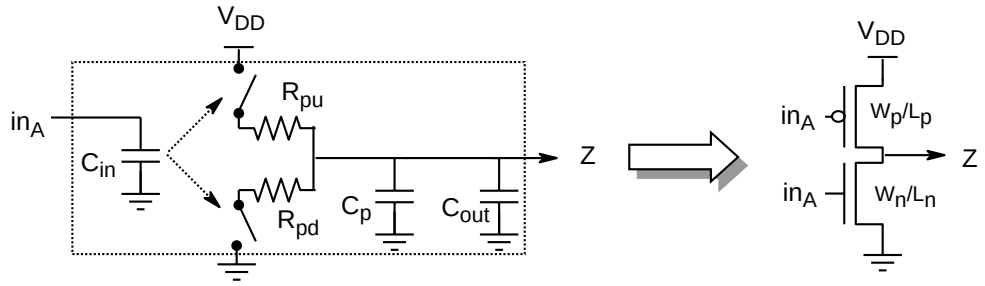


Figure A.1. *RC model of a logic gate (for one input).*

total hardware cost in terms of number of gates, the area of the different gates is normalized using a reference gate, in this case a minimum sized (1x) two-input NAND gate (NAND2). Thus, the area of a design is estimated as the number of equivalent NAND2 gates. This value is just a rough approximation of the actual physical size of a design, since extra space is needed for chip pins and wiring. However, the values obtained are good enough to compare the relative sizes of the different designs.

Before evaluating delays and areas it is necessary to determine the model gate parameters. Section A.1 introduces the static CMOS gate library used for the gate level description of the architectures. Gate model parameters are obtained characterizing the static CMOS gate transistor circuit as a network of capacitors and resistors. For dynamic CMOS circuits, such as the dynamic CMOS mux-latches used in the radix-10 divider (Chapter 7), we use the area and delay values provided by the developers (normalized to NAND2 and FO4 units). In Section A.2 we detail the methodology for path delay evaluation and for critical path delay optimization. Section A.3 covers two of the techniques used for delay path optimization: buffering and balancing of loads for branching paths. Finally, we present in Section A.3 an example of delay an area estimations for some basic components.

A.1 Parametrization of the Static CMOS Gate Library

The logical effort method characterizes the logic gate delay by means of three parameters: logical effort, electrical effort, and parasitic delay. These parameters are obtained by modeling logic gates as capacitors, which represent the transistor gates and stray capacitances, charging and discharging through resistors, which model networks of transistors connected between the power supply voltages and the output of the logic gate. This RC model of a gate is shown in Fig. A.1. It relates easily to the design of an inverter. The pMOS pull-up transistor, with width W_p and length L_p, is modeled by the switch and resistor R_{pu} forming a path from the output Z to the positive power supply. The nMOS pull-down transistor, with width W_n and length L_n is modeled by the switch and resistor R_{pd} that form a path from the output Z to ground. The delay in a logic gate modeled by Fig. A.1 is just the RC delay associated with charging and discharging the capacitance attached to the output node Z. The equation of the output voltage Z(t) is stated as

$$Z(t) = V_{DD} \cdot e^{\frac{-t}{R_p(C_p+C_{out})}} \tag{A.1}$$

where R_p represents the pull-up (R_{pu}) or pull-down resistance (R_{pd}) of the charging/discharging path, C_p is the internal gate capacitance and C_{out} is the external load. From this equation we obtain the following delay expression:

$$t_p = R_p(C_p + C_{out}) \ln\left(\frac{V_{DD}}{Z(t)}\right) \tag{A.2}$$

where t_p is either the propagation delay for pull-up t_{pu} or pull-down t_{pd} transitions. Taking as a reference for high and low states the 65% and 35% of V_{DD} respectively, the propagation delay transitions are approximated as

$$\begin{aligned} t_{pu} &= R_{pu}(C_p + C_{out}) \\ t_{pd} &= R_{pd}(C_p + C_{out}) \end{aligned} \tag{A.3}$$

An additional delay term t_q would be necessary to account for other nonideal factors such as:

- The dependency of the delay on the input ramps.
- Capacitances in the internal nodes.
- Finite time to produce a charge/discharge path.
- Capacitance of interconnection loads.
- Gate capacitance variable with the applied voltage.

so the gate delay is given by $t_{gate} = max\{t_{pu}, t_{pd}\} + t_q$. To obtain a simplified expression for the gate delay we adopt the following criteria for the transistor level design:

- Both pMOS and nMOS transitions have the same delay. Subsequently, pMOS transistors have double width compared to nMOS transistors.
- For any gate we perform a scaling of the width of the transistor (normalization of the gate) in order to obtain (in the worst case) the same drive capability than the minimum sized inverter.
- For our rough model we do not consider nonideal terms, so $t_q = 0$.

Thus, assuming that both transitions have the same delay, equations (A.3) are summarized in

$$t_{gate} = t_{pu} = t_{pd} = R_p(C_p + C_{out}) \tag{A.4}$$

and therefore $R_{pu} = R_{pd} = R_p$. Expression (A.4) represents the delay of a 1x gate. For a sx gate (equivalent scaled width of transistors by s), the delay is given by

$$t_{gate}(s) = \frac{R_p}{s}(C_{out} + s\ C_p) \tag{A.5}$$

We assume that the capacitance C_{in} of an input is proportional to the area of both pMOS and nMOS transistor gates. Thus, the input gate capacitance of the scaled sx gate is $Cs_{in} = s\ C_{in}$, where $C_{in} = k(W_p\ L_p + W_n\ L_n)$ is the input capacitance of the 1x gate (k is a constant

that depends on the fabrication process). Introducing the input capacitances in expression (A.5), the propagation delay results in

$$t_{gate} = R_p\ C_p + R_p C_{in}(\frac{C_{out}}{Cs_{in}}) \tag{A.6}$$

The logical effort method represents the gate delay as a function of a fixed part, called the parasitic delay, p, and a part that is proportional to the load on the gate's output, called the effort delay or stage effort, f. The effort delay is further defined as $f = g\ h$, where g is the logical effort, and h is the electrical effort. Thus, in terms of a delay unit $\tau = R_{inv}\ C_{inv}$, expression (A.6) is parameterized as

$$t_{gate} = \tau(p + f) = \tau(p + g\ h) \tag{A.7}$$

where

$$\begin{aligned} g &= \frac{R_p C_{in}}{\tau} \\ h &= \frac{C_{out}}{Cs_{in}} \\ p &= \frac{R_p C_p}{\tau} \end{aligned} \tag{A.8}$$

The product $\tau = R_{inv}\ C_{inv}$ is a characteristic delay for a technology, where C_{inv} and R_{inv} are the corresponding channel resistance and the total gate capacitance for a 1x inverter.

Instead of using absolute delay units, delay values are usually normalized to the fanout-of-4 inverter (FO4) metric. A FO4 delay represents the delay of a 1x inverter with an output load of four other 1x inverters. The FO4 metric can be used to express delays in a process-independent way since it is fairly stable through a wide range of process technologies, temperatures, and voltages [68], and also because most designers know the FO4 delay in their process and can therefore estimate how your circuit will scale to their process.

A FO4 delay is related to τ units as follows:

$$t_{FO4} = R_{inv}\ C_{inv} + R_{inv}\ C_{out} = R_{inv}(C_{inv} + 4C_{inv}) = 5\tau \tag{A.9}$$

The gate delay in FO4 units is given by the following expression

$$d_{gate}(\#FO4) = \frac{t_{gate}}{t_{FO4}} = \frac{\tau}{t_{FO4}}(p + g\ h) = \frac{1}{5}(p + g\ h) \tag{A.10}$$

Therefore, the FO4 delay of any CMOS gate is estimated by determining its parameters p, g and h. We show next how to compute these terms.

Computation of g (logical effort)

As we have mentioned, we consider that the 1x gates have the same drive capability than the 1x inverter. Therefore, in this case, $R_p = R_{inv}$, resulting in

$$g = \frac{R_p\ C_{in}}{R_{inv}\ C_{inv}} = \frac{C_{in}}{C_{inv}} \tag{A.11}$$

which corresponds to the ratio of the input (gate) capacitances between the 1x gate and the 1x inverter, provided that $R_p = R_{inv}$. We consider that the gate capacitances are proportional to the width of the corresponding gate. Therefore

$$C_{in} = \frac{W_{gp} + W_{gn}}{W_{invp} + W_{invn}}\ C_{inv} \tag{A.12}$$

where W_{gp}, W_{gn} are the widths of the pMOS and nMOS transistors of the corresponding gate input and W_{invp}, W_{invn} are the widths of the inverter transistors. Then, the logical effort is

$$g = \frac{W_{gp} + W_{gn}}{W_{invp} + W_{invn}} \tag{A.13}$$

Computation of h (electrical effort)

Introducing relations $Cs_{in} = s\ C_{in}$ and $C_{in} = g\ C_{inv}$ in the corresponding equation for h of expression (A.8), we have

$$h = \frac{C_{out}}{s\ g\ C_{inv}} \tag{A.14}$$

The output gate capacitance C_{out} in terms of C_{inv} is given by

$$C_{out} = \frac{\sum_{i=0}^{k-1}(W_{gp} + W_{gn})_i}{W_{invp} + W_{invn}} C_{inv} = \sum_{i=0}^{k-1} g_i\ C_{inv} \tag{A.15}$$

where the terms $(W_{gp} + W_{gn})_i$ correspond to the gate load for a fanout of k gates. We consider only the load due to input gate capacitances, obtained as the sum of the widths of the input gate transistors.

Therefore, h can be computed as

$$h = \sum_{i=0}^{k-1} \frac{g_i}{s\ g} \tag{A.16}$$

where the $g_i's$ are the logical efforts of the k gates that contribute to the gate load. Moreover, the effort delay is given by

$$f = h\ g = \sum_{i=0}^{k-1} \frac{g_i}{s} \tag{A.17}$$

Computation of p (electrical effort)

For a gate with scaled width for transistors so that its drive capability is the same as the 1x inverter, we have that $R_p = R_{inv}$, and therefore,

$$p = \frac{C_p}{C_{inv}} \tag{A.18}$$

We consider C_p proportional to the width of transistors connected to the output. Thus, the ratio of parasitic capacitances between a gate and a 1x inverter is

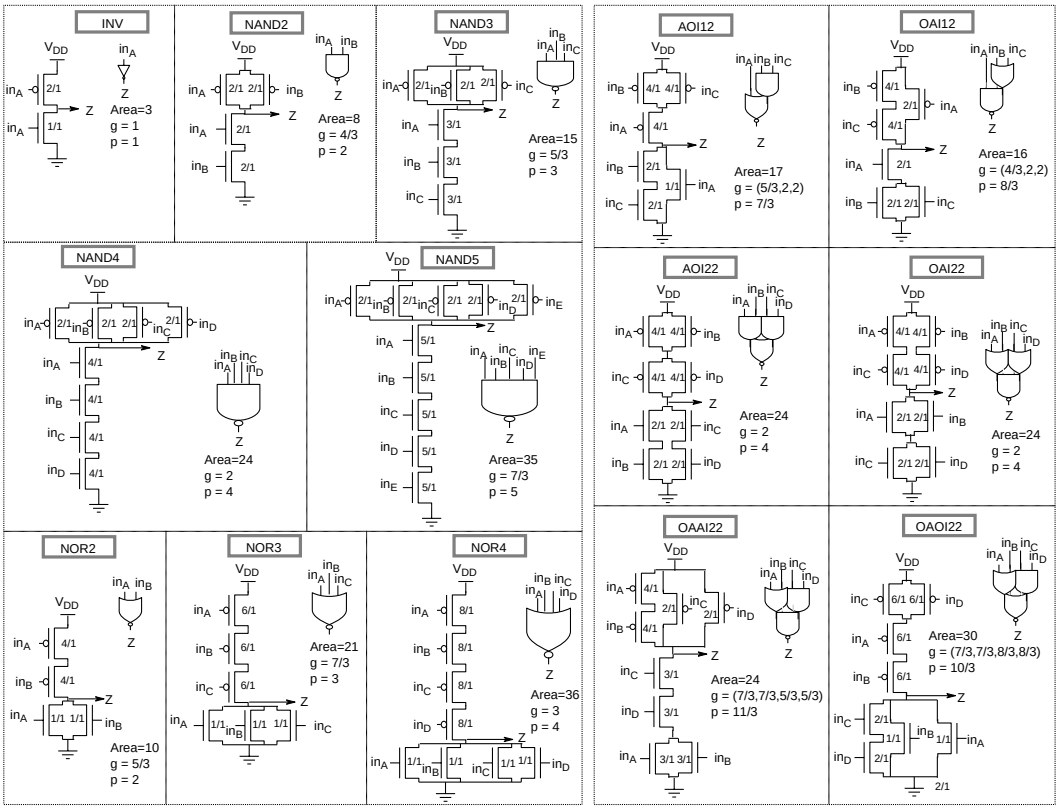
$$\frac{C_p}{C_{pinv}} = \frac{W_{og}}{W_{oinv}} \tag{A.19}$$

where W_{og} and W_{oinv} are the sum of the widths of the gate transistors connected to the output for a gate and an 1x inverter respectively. Therefore, we express p as follows

$$p = \frac{C_p}{C_{inv}} = \frac{W_{og}}{W_{oinv}} \frac{C_{pinv}}{C_{inv}} = \frac{W_{og}}{W_{oinv}} \frac{p_{inv} C_{inv}}{C_{inv}} = \frac{W_{og}}{W_{oinv}} p_{inv} \tag{A.20}$$

Note that the parasitic delays does not depend on the size of the gate. It is difficult to provide values of p_{inv} (electrical effort of an inverter) for a wide range of technologies. For our rough model we assume $p_{inv} = 1$, so the parasitic delay is estimated as

$$p = \frac{W_{og}}{W_{oinv}} \tag{A.21}$$



(a) Basic gates. (b) Complex gates.

Figure A.2. *Library of minimum size static CMOS gates.*

Therefore, the calculation of p, g and h for gate delay evaluation is simply reduced to determine the widths of the transistors for each gate. Fig. A.2 shows the library of parameterized static CMOS gates we have used for the gate level description of the architectures. We represent both the transistor circuit and the schematics for each gate. The logic gates selected for the library were limited to single inverting stages, to keep the range of gate stage effort around 4. As we have commented, we need to normalize the geometries of the transistors to obtain gates with the same drive capability as the 1x inverter, that is, we only use minimum sized (1x) gates. Thus, the scaling factor is s=1 for all the CMOS gates of the library.

This simplifies further path delay computations, since h is simply determined by the g's of the gate and its load (given by equation A.16 with s=1), while the gate effort delay f is just given by the g's of the gate load (equation A.17 with s=1). Therefore, the FO4 gate delay is computed as

$$d_{gate} = \frac{1}{5}(p_{gate} + f_{gate}) = \frac{p}{5} + 0.2L_{out} \tag{A.22}$$

where we have defined the gate load $L_{out} = \sum_{i=0}^{k-1} g_i$ for a fanout of k gates. L_{out} represents the gate load capacitance normalized to the 1x inverter capacitance. Thus, to determine the FO4 delays we only need to characterize the p and g parameters for each gate of the library.

Since the drive capability depends on the W/L ratio of the transistors, we need to adjust this ratio for the transistors of the corresponding gate. The W/L units for each transistor are

determined so that the gate is normalized, that is, the equivalent W/L ratios of each of the nMOS and pMOS parts are the same as in the 1x inverter. The rules for this adjustment are as follows:

- For a branch of transistors in serial configuration, the equivalent W/L is given by the ratio of the W of each transistor and the addition of the L values.
- For branches of transistors in parallel, we consider the worst case of conduction, that is, only one branch conducts. Therefore, the equivalent W/L ratio is the maximum W/L ratio of the branches.

We consider relative units for W and L normalized to the values of the nMOS transistor in a 1x sized inverter. Therefore, for the nMOS transistor of the inverter we have W/L =1/1. Moreover, another important parameter is m, the ratio of the widths of the nMOS and pMOS transistors to have symmetrical rise/fall characteristics. A typical value used for current static CMOS technology would be m = 2 [126, 131]. Therefore, W/L =2/1 for the pMOS transistor of the minimum sized inverter.

For instance, the p and g parameters and the FO4 delay equation for the AOI12 (asymmetric (1+2)-input AND-OR-INVERTER) gate of Fig. A.2(b) are computed as follows:

- Computation of p. As defined in equation (A.21), we need to compute the normalized widths of the transistors connected to the output of both the gate and the 1x inverter. For the AOI12 gate, the sum of the normalized widths is W_{og} = 7, and W_{oinv}=3 for the inverter. Therefore, p = 7/3.
- Computation of g. We give the value of g for each input in vector form, that is, $g = (g_{inA}, g_{inB}, g_{inC})$. From equation (A.13), we just need to compute the ratio of the normalized widths of the nMOS and pMOS transistors for each gate input to the width of the nMOS and pMOS transistors of the 1x inverter. Therefore, g=(5/3,2,2).
- Computation of gate delay. From expression (A.22), we obtain that the estimated FO4 delay of the AOI12 gate is
$$d_{AOI12} = \frac{7}{15} + 0.2L_{out} \tag{A.23}$$
for an output load of L_{out}.

The cost of each gate is estimated as the logical area. The logical area measures the number of square units in the active areas of the transistors of the gate. Therefore, the cost of a gate is given by the sum of the normalized W/L ratios of all nMOS and pMOS transistors of the gate. Note that this method takes into account the number of transistors and its size. Fig. A.2 also shows the logical area for each gate of the library. To normalize the results to the cost of the NAND2 gate (with logical area of 8 units), we compute the ratio Area$_{gate}$/8. For instance, the estimated area of the AOI12 gate is 17/8= 2.125 NAND2.

A.2 Path delay evaluation and optimization

Computation of path delay

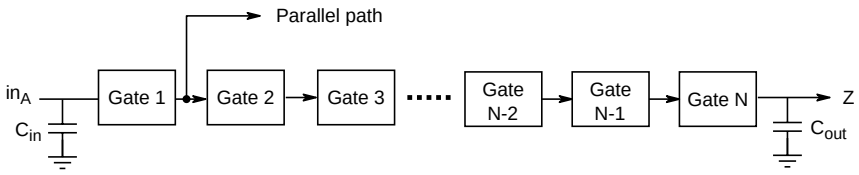


Figure A.3. *Path to be optimized.*

The path delay D (normalized to FO4 units) is the sum of the delays of the logic gates in the path. The total stage delay is given by the critical path delay, D_{max}. D is decomposed in a path parasitic delay P, and in a path effort delay, D_F, as

$$D(\#FO4) = \sum_i d_i = \frac{1}{5}(P + D_F) = \frac{1}{5}\sum_i (p_i + f_i) = \frac{1}{5}(\sum_i p_i + \sum_i g_i h_i) \tag{A.24}$$

where the subscript i index the logic gate stages along the path. The path effort delay D_F is not easy to compute using the electrical efforts h_i when branching occurs within a logic network. In this case, some of the available drive current is directed along the path we are analyzing, and some is directed off the path. The logical effort method introduces the branching effort B, to account for fanout within a network.

The use of only minimum sized gates, as we propose, simplifies the computation of D_F by means of the gate stage efforts $f_i = g_i h_i$. In our case, from equation (A.17) with s=1, we obtain that the stage effort f_i is given by the normalized load of the gate at stage i, L_i, with fanout k_i, that is,

$$f_i = L_i = \sum_{j=0}^{k_i - 1} g_j(i) \tag{A.25}$$

so D_F is computed as

$$D_F = \sum_{i=1}^{N} f_i = \sum_{i=1}^{N-1} L_i + L_{out} \tag{A.26}$$

For the whole design or pipelined stages, we assume that the output load of the path C_{out} is equal to the input load C_{in}, so in our case, $L_{out} = L_{in} = \sum_{j=0}^{k_0 - 1} g_j(0)$, where the $g_j(0)'s$ are the logical efforts of the k_0 gates connected to a given input.

Optimization of path delay

We want to obtain the conditions to optimize a logical path of a multi-stage design as shown in Fig. A.3. An overall delay optimization usually requires balancing the delay of parallel paths, which is beyond the scope of this analysis due to its high complexity. We only consider a simple but very common case of parallel path optimization in Section A.3: a buffer fork.

The method of logical effort provides the best number of stages to minimize the path delay and it shows how to get least overall delay by balancing the delay among the stages. For a given path we define the path effort as

$$F = GH = \prod_{i=1}^{N} h_i g_i \tag{A.27}$$

where $H = \prod_{i=1}^{N} h_i = \frac{C_{out}}{C_{in}}$ is the path electrical effort and $G = \prod_{i=1}^{N} g_i$ is the path logical effort.

The optimization process consists in calculating h_i to minimize D with the constrain $H = \frac{C_{out}}{C_{in}}$ (a detailed calculation is shown in Chapter 3 of [131]). For a given fixed number of gate stages in a path, N, the minimum path delay is achieved when each gate in the path has the same effort, that is, $f_i = h_i g_i = \rho = F^{\frac{1}{N}}$ and the FO4 delay of the path is

$$D_{min}(N) = \frac{1}{5}(P + NF^{\frac{1}{N}}) = \frac{1}{5}(P + N(GH)^{\frac{1}{N}}) \tag{A.28}$$

The optimum path delay is obtained for a number of gate stages equal to $N = \hat{N}$, that is, $D_{opt}(N) = D_{min}(\hat{N})$ in FO4 delays. A good estimate of the optimum number of gate stages for an inverter chain, as detailed in Section A.3, is $\hat{N} \approx \frac{\ln F}{\ln \rho} = \log_\rho F$ with $\rho \approx 3.6$. The optimum path delay is $D_{opt} \approx \frac{1+\rho}{5} log_\rho F$ in FO4 units.

For paths with more complex gates, the parasitics have more importance, so a better value for ρ is approximately around 4, so $D_{opt} \approx \log_4 F$. This estimate is useful for rough comparisons among different circuit topologies by computing only the path effort [131].

To keep close to the optimum path delay, we use gate loads $f_i = L_{out}(i)$ around $\rho = 4$. The logical effort method use gate sizing to adjust the stage efforts. Devices with higher transistor sizes are faster but have more input capacitance (i.e. presents more load to the previous stage), so optimal gate sizing is a complex tradeoff process. Instead of gate sizing, we use other equivalent optimization techniques to reduce the number of logic stages (increasing the stage effort) or to decrease the gate load (and therefore to reduce the stage effort), such as:

- Buffering. A buffering tree, or equivalently, a chain of inverters, is used to drive the load and reduce the effort stage. The number and size of buffers required to drive an output capacitance C_{out} is determined in Section A.3.
- Gate cloning. Involves duplicating a gate and dividing the fanout between the copies.
- Delay balancing of buffer forks. We apply this technique to optimize the delay of true complement control signals that drive high loads. This is detailed in Section A.3.
- Gate stage expansion. In this case, a complex gate in the critical path is converted into a simpler gate and other additional gates out of the critical path.
- Gate stages compression. A two-logic stage of simpler gates is compressed into a single complex logic stage.
- Inverter pushing. An inverter at an input in the critical path can be pushed to other input out of the critical path.

A.3 Optimization of buffer trees and forks

This Section covers two of the techniques used for path delay minimization: delay optimization of buffer trees (or equivalently buffer chains) and buffer forks, which consist of a pair of buffer (inverter) chains with a common input used to drive high loads that require a true complement input, e.g. the control signal of a 2:1 mux.

Optimum delay of a buffer tree

We use expression (A.28) to determine the optimum delay of a buffer tree (or a buffer chain) with an input capacitance C_{in} (or L_{in}) that drives an output capacitance C_{out} (or L_{out}). Since for inverters, $g_i = 1$, we have

$$F = GH = H = h^N \tag{A.29}$$

Therefore, from (A.28) we obtain

$$D_{opt} = \frac{1}{5}(Np_{inv} + Nh) = \frac{N}{5}(h + p_{inv}) = \frac{\ln H}{5 \ln h}(h + p_{inv}) \tag{A.30}$$

that is, $N = \frac{\ln H}{\ln h}$ and $H = \frac{C_{out}}{C_{in}}$ is a constant.

The minimum of this expression occurs for $\ln h = \frac{h+p_{inv}}{h}$, so after introducing this value in (A.30), the optimum delay is $D_{opt} = \frac{h}{5}\ln(H)$.

For $p_{inv} = 1$, the value of h which verifies $\ln h = \frac{h+p_{inv}}{h}$ is $h = 3.591$, and therefore the optimum delay of the buffer tree with an input capacitance C_{in} driving a load C_{out} is

$$D_{opt}(\#FO4) = 0.72 \ln(\frac{C_{out}}{C_{in}}) \tag{A.31}$$

The estimated logical area of a tree buffer is

$$Area_{buffer} = N \times h \times Area_{inv} = 8.43 \times \ln(\frac{C_{out}}{C_{in}}) \tag{A.32}$$

or $1.05 \times \ln(\frac{C_{out}}{C_{in}})$ NAND2 units.

Optimum delay of a buffer fork

One of the complications of buffer forks is that one of the paths has an additional inverting stage. To avoid an increased overall delay, the true and complement output signals must emerge at the same time. Therefore, the delay of each buffer chain should be the same. These kind of signals are typically used as control inputs for high loads of 2:1 multiplexers. We also assume that both paths drive the same load $\frac{C_{out}}{2}$. Fig. A.4 shows the general structure of a buffer fork driving n 2:1 muxes ($L_{out} = \frac{4n}{3}$). The input capacitance C_{in} is divided between the two paths as

$$C_{in} = C_{Ain} + C_{Bin} = (1 - \beta)C_{in} + \beta C_{in} \tag{A.33}$$

where $\beta \in [0, 1]$ denotes the fraction of input capacitance allocated to the branch with more stages.

The optimal FO4 delay for each separate path is given by

$$\begin{aligned} D_{Aopt} &= \frac{(N-1)}{5} H_A^{\frac{1}{N-1}} + \frac{(N-1)}{5} p_{inv} = \frac{(N-1)}{5}((\frac{H}{2(1-\beta)})^{\frac{1}{N-1}} + p_{inv}) \\ D_{Bopt} &= \frac{N}{5} H_B^{\frac{1}{N}} + \frac{N}{5} p_{inv} = \frac{N}{5}((\frac{H}{2\beta})^{\frac{1}{N}} + p_{inv}) \end{aligned} \tag{A.34}$$

To minimize the total delay, both path delays should be equal, that is, $D_{Aopt} = D_{Bopt}$. The electrical effort of each path H_A and H_B must be close to the electrical effort of the whole fork, $H = \frac{C_{out}}{C_{in}}$, for optimal delay. To reduce the delay of the slowest path, we reduce slightly

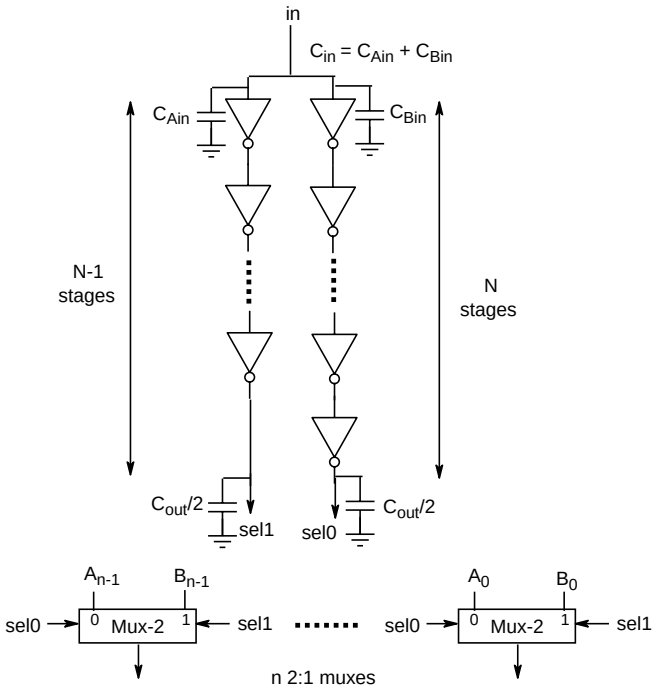


Figure A.4. *An amplifier fork with a load of n 2:1 muxes.*

its electrical effort by incrementing the input capacitance of that path. Since C_{in} and C_{out} are fixed, this implies a variation in β, making slower the other path of the fork.

By other hand, the optimum number of inverter stages for a buffer chain with an equivalent electrical effort H, is given by $\hat{N} = \lceil 0.78 \ln H \rceil$ (obtained replacing $h = 3.591$ in $N = \frac{\ln H}{\ln h}$). For minimum delay one path must have $\hat{N}$ inverting stages. To determine if $\hat{N}$ corresponds to the N stage path or to the N-1 stage path, we compute the optimum electrical stage efforts $\rho = H^{\frac{1}{N}}$ of the buffer chain for $N = \hat{N} + 1$ stages and for $N = \hat{N} - 1$ stages (note that $H \approx H_A \approx H_B$). The selected value ($\hat{N} + 1$ or $\hat{N} - 1$) is the one with ρ closer to 3.591, the optimum value for a buffer chain.

We replace N by $\hat{N}$ or $\hat{N} + 1$ in (A.34) and compare both delay equations (with $p_{inv} = 1$), that is,

$$(\hat{N} - 1)(1 + (\frac{H}{2(1-\beta)})^{\frac{1}{\hat{N}-1}}) = \hat{N}(1 + (\frac{H}{2\beta})^{\frac{1}{\hat{N}}}) \tag{A.35}$$

or

$$\hat{N}(1 + (\frac{H}{2(1-\beta)})^{\frac{1}{\hat{N}}}) = (\hat{N} + 1)(1 + (\frac{H}{2\beta})^{\frac{1}{\hat{N}+1}}) \tag{A.36}$$

obtaining a value of β for a given H. The optimum FO4 delay is obtained replacing the corresponding β and and $\hat{N}$ or $\hat{N} + 1$ in one of the delay equations (A.34).

The most usual number of stages for forks are $N = \{2, 3\}$. Actually, for higher loads, it is preferred to use a buffer tree and a fork of inverters with N=3. For lower loads, it is better to use a fork with N=2 by replicating the gate stage preceding the fork (gate cloning) rather than using a fork with N=1.

As an example, we present in Table A.1 the FO4 delay figures of the optimum forks for a

n	Opt. fork type	Delay (# FO4)
1 to 8	2-1	0.85 to 1.95
9 to 36	3-2	2.05 to 3.05
37 to 152	4-3	3.10 to 4.15
152 to 598	5-4	4.20 to 5.15

Table A.1. *Optimum delay of buffer forks for a load of n 2:1 muxes.*

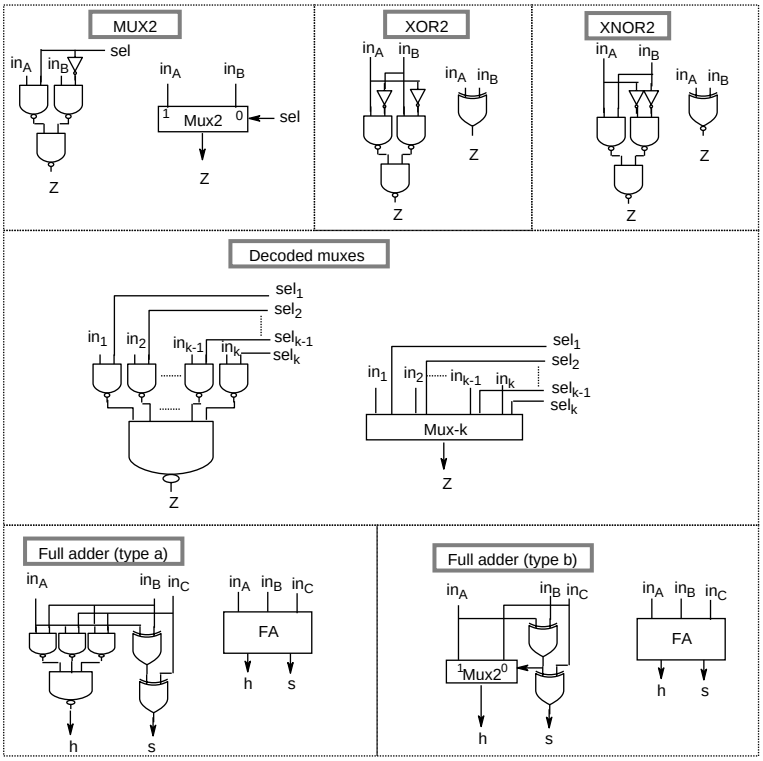


Figure A.5. *Basic multi-stage gate components.*

load of n 2:1 muxes. We consider an electrical effort of $H = \frac{2n}{3}$ for the whole path ($L_{in} = 4$ and $L_{out} = 2\frac{4n}{3}$). The optimum path delay of this fork is computed as

$$D_{opt} = \frac{N}{5}\left[1 + (\frac{n}{\beta})^{\frac{1}{N}}\right] \tag{A.37}$$

A.4 Area and delay estimations of some basic components

We use our delay model to estimate the delay of multi-stage gate components of common use in hardware architectures, such as the multi-stage gate components shown in Fig. A.5.

Table A.2 summarizes the equivalent area, FO4 delay equation and normalized input load (L_{in}) of these common components. The cost in area of each component is just the sum of the equivalent NAND2 values of its gates. The FO4 stage delay of each component was obtained adding the delay of the gates in its critical path, and the normalized output load

Component	Delay (# FO4)	L_{in}	Area (# NAND2)
Xor2	1.53+0.2 L_{out}	$\frac{7}{3}$	3.750
Xnor2	1.53+0.2 L_{out}	$\frac{7}{3}$	3.750
Mux2(not decoded)	(1.10, 1.53)+0.2 L_{out}	$(\frac{4}{3}, \frac{7}{3})$	3.375
Mux2	1.10+0.2 L_{out}	$\frac{4}{3}$	3.000
Mux3	1.36+0.2 L_{out}	$\frac{4}{3}$	4.875
Mux4	1.60+0.2 L_{out}	$\frac{4}{3}$	7.000
Mux5	1.86+0.2 L_{out}	$\frac{4}{3}$	9.375
Full adder (type a)	(1.33, 3.53)+0.2 L_{out}	5	12.375
Full adder (type b)	4.0+0.2 L_{out}	$(\frac{11}{3}, \frac{7}{3}, \frac{11}{3})$	11.250
*D-latch (1-bit)	3.00	$\frac{4}{3}$	4
* m:1 Mux-latch (1-bit)	(3.00,4.50)	$\frac{4}{3}$	4+m
* Area and delay figures obtained from reference [107]			

Table A.2. *Area, delay and input capacitance estimations of some multi-stage gates.*

L_{out}. The area and delay values for the D-latches and the dynamic CMOS mux-latches (up to a 8:1 multiplexer) are extracted from reference [107]. We use all these values to estimate the cost and FO4 delay of designs in a hierarchical way. For instance, both full adders (1-bit 3:2 CSAs) shown in Fig. A.5 are built of other multi-stage gate components (xor gates and 2:1 muxes). Thus, the total delay stage of both full adders was estimated using the delay equations for the xor and 2:1 mux.

Resumen

Los microprocesadores de propósito general incorporados en los sistemas de computación de altas prestaciones anteriores a 2007 solo implementan unidades de punto flotante binarias. Esto es debido, fundamentalmente, al hecho de que los datos binarios pueden ser almacenados y manipulados eficientemente en los computadores actuales (basados en la electrónica de transistores de dos estados), lo que los hace adecuados para cálculos científicos.

Las actuales aplicaciones financieras, comerciales y orientadas al usuario necesitan procesar una gran cantidad de números en representación decimal. Sin embargo, las unidades de aritmética binaria no pueden representar exactamente la mayoría de las fracciones decimales (por ejemplo, 0.1 no tiene una representación binaria exacta), lo que genera errores de precisión al tratar directamente los datos decimales. Además, en muchas aplicaciones comerciales, incluyendo análisis financiero, transacciones bancarias, cálculo de aranceles, tasas e impuestos, conversiones monetarias, seguros, contabilidad y e-comercio, los errores introducidos al convertir entre números decimales y binarios pueden incumplir los requisitos de precisión legales.

Las aplicaciones software de aritmética decimal corrigen estos errores, pero son significativamente más lentas que las unidades hardware. Por otra parte, el continuo escalamiento de las tecnologías de integración, permite obtener circuitos con una complejidad de millones de transistores con un coste moderado. Así, existe un interés cada vez mayor por parte de los fabricantes de microprocesadores para incorporar unidades hardware dedicadas de aritmética decimal en sus futuros productos.

Este interés se ha visto apoyado por los esfuerzos para definir un estándar para la aritmética decimal de punto flotante. En concreto, la revisión del estándar IEEE 754-1985 para punto flotante (IEEE 754-2008) incorpora especificaciones para aritmética decimal. El estándar puede ser implementado totalmente en hardware, en software o usando cualquier combinación de ambos.

De este modo, las primeras unidades de punto flotante decimal ya están disponibles desde Junio de 2007. Los microprocesadores IBM Power6 de doble núcleo, e IBM z10, de cuatro núcleos, incorporan una unidad decimal en cada núcleo. Estos procesadores constituyen la base de la línea de servidores UNIX de alto rendimiento de IBM destinados a aplicaciones financieras. Otros fabricantes, como Intel, han optado por implementaciones software, pero planean incorporar algún tipo de soporte hardware para procesamiento de aritmética decimal. Además, se están publicando un gran número de trabajos académicos centrados en diferentes aspectos del diseño de unidades decimales de altas prestaciones.

Siguiendo esta línea de trabajo, esta tesis doctoral se centra en la investigación y diseño de nuevos algoritmos y unidades hardware de altas prestaciones para aritmética decimal de punto fijo (entera) y de punto flotante (real).

En primer lugar, hemos determinado cual es el conjunto de operaciones aritméticas decimales de punto fijo y de punto flotante definidas en el estándar IEEE 754-2008 que deberían ser aceleradas en hardware. Para esta estimación se han tenido en cuenta diversos aspectos, como el rendimiento de las unidades decimales existentes, el margen de mejora respecto a las unidades binarias equivalentes y la frecuencia relativa de cada operación en programas comerciales. De este modo, hemos desarrollado diferentes unidades hardware decimales (para punto fijo y punto flotante) de alto rendimiento para procesar estas operaciones, que incluyen suma/resta, multiplicación (o multiplicación acumulación) y división.

Con el objetivo de obtener arquitecturas eficientes, competitivas con otras propuestas tanto académicas como industriales, se ha investigado el uso de nuevos algoritmos, diferentes codificaciones numéricas decimales, técnicas de suma decimales libres de propagación de acarreo, especulación hardware y otros métodos que permiten un mayor grado de paralelismo. Con la aplicación de estos conceptos, las unidades resultantes presentan una latencia reducida. Además se han abierto nuevas vías que sirven de guia en el diseño de los futuros procesadores comerciales. En este sentido, la tendencia parece ser la necesidad de incorporar mayor precisión (número de dígitos) en las unidades decimales que en las binarias para satisfacer los requerimientos de los usuarios. Esto impone demandas adicionales en el diseño de estos procesadores en cuanto a optimización del área.

Otro aspecto de la tesis se ha centrado en mejorar la calidad (fiabilidad) de las computaciones. Los usuarios de servicios financieros y de e-comercio demandan un alto grado de fiabilidad en sus transacciones. Por otro lado, los errores debidos a fallos temporales de los circuitos causados por radiaciones y partículas cósmicas son cada vez más significativos debido a las altas densidades de integración y los reducidos tamaños de los transistores en un chip. Los servidores incorporan diversos tipos de soportes para detectar errores. Para proteger a las unidades aritméticas contra estos errores se replican, lo que conlleva un aumento del área (coste) del procesador. Así, se ha desarrollado una técnica de detección y corrección de errores para varios tipos de sumadores decimales de acarreo propagado con un coste en área reducido.

También hemos desarrollado un modelo de evaluación del área (coste de fabricación) y retardo (relacionado con el rendimiento y velocidad del procesador) de las diferentes arquitecturas basado en el esfuerzo lógico, que permite obtener resultados de forma rápida y sencilla, y que, a la vez, permite establecer comparaciones realistas entre diferentes arquitecturas y propuestas independientemente de la tecnología de fabricación.

Un último aspecto que hemos contemplado es la aplicación de los conceptos y arquitecturas desarrollados a implementaciones reales. Por ejemplo, hemos propuesto un conjunto de modificaciones necesario para adaptar la unidad de división de alto rendimiento diseñada a una unidad decimal de punto flotante de un procesador comercial como el IBM Power6 o el IBM z10.

Entre las unidades decimales de punto fijo y punto flotante desarrolladas se encuentran las siguientes arquitecturas:

1. Sumadores enteros decimales (BCD) de acarreo propagado.

Hemos propuesto un nuevo algoritmo para la suma/resta de dos operandos enteros decimales en complemento a 10. Este método, denominado suma decimal mediante especulación condicional, y las arquitecturas resultantes, están detallados en el Capítulo 3.

El método analiza el valor de la suma de los 3 bits más significativos para cada dígito de los operandos de entrada, codificados en BCD (4 bits por dígito, con pesos 8, 4, 2, y 1), e incrementa en +6 unidades cada posición decimal si dicha suma es mayor o igual que 8. Este incremento se realiza en un tiempo de computación constante, independiente de la precisión requerida (número máximo de dígitos significativos representables en un formato), puesto que no se produce propagación de acarreo entre dígitos.

De esta forma, se genera un acarreo '1' al siguiente dígito cuando la suma en una posición decimal es mayor o igual que 16. Esto permite utilizar cualquier técnica convencional de suma/resta binaria para acelerar la propagación de acarreo, ya que los acarreos decimales coinciden con los acarreos binarios en posiciones hexadecimales (esto es, 1 cada 4).

Además, la suma binaria coincide con la representación BCD de la suma decimal excepto en aquellas posiciones decimales en las que la suma de los dígitos de entrada es 8(+6) o 9(+6) y el acarreo decimal de entrada es '0'. La corrección se realiza mediante lógica binaria muy sencilla reemplazando los dígitos de suma 14 ('1110') o 15 ('1111'), por 8 ('1000') o 9 ('1001') respectivamente. Esta lógica se puede incorporar en el sumador binario sin incrementar el retardo de la propagación de acarreo (su vía crítica o de máximo retardo).

Las arquitecturas resultantes constan básicamente de dos bloques: un primer módulo de pre-corrección decimal y un sumador binario modificado. Para obtener implementaciones de alto rendimiento hemos seleccionado un conjunto de sumadores binarios paralelos basados en el cálculo de prefijos ("parallel prefix adders"), puesto que las implementaciones resultantes en tecnología VLSI (muy alta escala de integración) presentan estructuras muy regulares y de muy baja latencia. Entre las topologías más eficientes están los sumadores paralelos cuaternarios (de área reducida, solo evalúa 1 de cada 4 acarreos) con esquema de Ling (adelantan la obtención del acarreo en una etapa lógica). Además, el método propuesto también permite obtener sumadores combinados binarios/decimales muy eficientes respecto a otras propuestas.

El algoritmo para suma/resta decimal mediante especulación condicional se ha extendido en el Capítulo 4 para operandos enteros decimales (BCD) representados en signo-magnitud. Las arquitecturas resultantes son más complejas que las equivalentes para operandos en complemento a 10, pero su utilidad está justificada, ya que se requieren para suma o resta de coeficientes enteros (BCD) en unidades de punto flotante decimal (sumadores, multiplicadores, etc...).

Del análisis comparativo efectuado, usando el modelo de área-retardo descrito en el Apéndice A, se concluye que los sumadores decimales propuestos presentan mejores valores de área y latencia en relación a otras arquitecturas de alto rendimiento basadas en algoritmos alternativos, puesto que evitan la necesidad de implementar un módulo de corrección decimal posterior al sumador binario.

2. Unidades de detección de errores en sumadores enteros decimales.

Se han desarrollado unidades para detectar errores en sumadores BCD en complemento a 10 y sumadores combinados binarios/decimales, detalladas en el capítulo 3, y en sumadores BCD de signo-magnitud, detalladas en el Capítulo 4.

Estas unidades comparan los dos operandos de entrada con el resultado de la suma a nivel de bit, produciendo un bit de error en caso de que el vector de suma no sea el esperado. Consisten en un sumador libre de propagación de acarreo (tiempo de computación constante) y un detector de paridad. Requieren la mitad de área que otras soluciones empleadas en microprocesadores comerciales (como la replicación de unidades), mientras que no afectan al rendimiento del procesador, puesto que la detección tiene lugar fuera de la vía crítica: el resultado de la instrucción de suma es especulativo por uno o más ciclos, mientras el microprocesador verifica las dependencias con intrucciones previas. En este tiempo, la unidad aritmética puede detectar errores en la suma.

3. Sumador de punto flotante decimal.

En el Capítulo 5 presentamos una nueva arquitectura de altas prestaciones para sumar coeficientes BCD (en signo-magnitud) que incorpora una unidad de redondeo decimal conforme al estándar IEEE 754-2008. Este es un componente fundamental para acelerar en hardware diferentes operaciones decimales de punto flotante como suma/resta, multiplicación y suma-multiplicación combinada.

Una de las dos implementaciones de altas prestaciones publicadas hasta la fecha realiza el redondeo decimal después de la suma de coeficientes BCD en signo-magnitud, por lo que son necesarias dos propagaciones de acarreo, una para la suma de coeficientes y otra para el redondeo. La suma/resta de magnitudes BCD consiste en una pre-corrección de los operandos de entrada, una suma binaria y una corrección posterior compleja, dependiente de los acarreos decimales, para obtener la representación BCD correcta de la suma/resta de magnitudes. El redondeo consiste en la suma adicional de una unidad en el dígito menos significativo de la suma BCD truncada (el numero máximo de dígitos representable depende del formato) si las condiciones para el modo de redondeo decimal lo requieren.

La otra propuesta existente combina parte de la suma BCD en signo magnitud con el redondeo decimal solapando la corrección decimal posterior a la suma binaria con la evaluación de ciertas señales requeridas para el redondeo. De esta forma se reduce un 20% la latencia de la propuesta previa. Sin embargo, esta solución presenta un alto coste hardware debido a la introducción de árboles de puertas lógicas (tiempo de computación logarítmico) para adelantar el cálculo del redondeo y a la complejidad de la unidad de correción decimal posterior a la suma binaria.

La arquitectura presentada en esta tesis utiliza el mismo sumador binario para realizar tanto suma BCD en signo-magnitud como el redondeo decimal. Para ello se realiza una pre-corrección de los operandos BCD de entrada (tiempo de computación constante), lo que facilita realizar en el sumador binario incrementos de +1 o +2 unidades en la posición menos significativa del formato del operando de suma, sin aumentar significativamente la latencia del sumador. Esto permite realizar la suma en signo-magnitud y el redondeo simultáneamente, y

además simplifica significativamente la unidad de corrección decimal posterior.

De esta forma, hemos mejorado en un 15% el rendimiento del sumador IEEE 754-2008 de punto flotante decimal de menor latencia presentado hasta la fecha y reducido el área de la unidad combinada de suma de coeficientes BCD y redondeo decimal un 25% .

4. Sumadores decimales libres de propagación de acarreo.

La mayoría de los algoritmos para implementar multiplicación, división y raíz cuadrada decimales en unidades de alto rendimiento requieren sumar múltiples operandos decimales. Los métodos para sumar dos operandos decimales usan propagación de acarreo (tiempo de computación logarítmico). Sin embargo, la aplicación recursiva de este tipo de algoritmos no es eficiente para la suma de múltiples operandos.

Así, de forma similar que en binario, para acelerar la suma de múltiples (más de dos) operandos decimales se han propuesto varios algoritmos que realizan sumas libres de propagación de acarreo. Estos métodos reducen q ($q > 2$) operandos decimales a dos operandos en un tiempo independiente del tamaño de palabra (tiempo de computación constante), retrasando la propagación del acarreo hasta la suma final de los dos operandos.

Estos algoritmos asumen que los operandos decimales están representados en BCD. Por lo tanto, las implementaciones resultantes son bastante complejas debido a la ineficiencia de la codificación BCD para representar los dígitos decimales (solo utiliza 10 de las 16 posibles combinaciones de 4 bits). Esto implica la introducción de mecanismos adicionales para corregir las combinaciones inválidas (aquellas que no representan un dígito decimal).

En el Capítulo 6 introducimos un nuevo algoritmo para sumar múltiples operandos decimales representados en sistemas de codificación decimales no convencionales (distintos de BCD), denominados (4221), (5211), (3321) y (4311). Estos códigos de 4 bits tienen la característica común de que la suma de los pesos de sus bits es 9. Por ejemplo, el dígito $A \in [0, 9]$ representado en código (4221) tiene el valor $A = a_3\ 4 + a_2\ 2 + a_1\ 2 + a_0$, donde $a_i \in \{0, 1\}$ son los bits asociados a cada peso.

El uso de estos códigos permite emplear aritmética binaria para realizar sumas (o restas) decimales libres de acarreo sin necesidad de efectuar correcciones, puesto que cada una de las 16 combinaciones de 4 bits representa un número decimal. Así, podemos utilizar un sumador binario sin propagación de acarreo (por ejemplo de acarreo almacenado, o carry-save adder o CSA, en inglés), o una estructura en árbol de CSAs para acelerar la suma de múltiples operandos decimales. Además, esto simplifica la implementación de sumadores combinados binario/decimal libres de propagación de acarreo.

Un CSA binario reduce tres vectores de bits (A, B, C) a un vector de suma y otro de acarreo C, de forma que $A + B + C = S + 2C$. La multiplicación binaria $2C$ se realiza mediante un desplazamiento aritmético de C de 1 bit a la izquierda. Sin embargo, la multiplicación de un operando decimal $\times 2$ es más compleja. Para la codificación (4221) esta multiplicación se realiza mediante una conversión de los dígitos de C a la representación (5211) y un desplazamiento de 1 bit a la izquierda del vector de bits resultante, con lo que los pesos se duplican (5211)$\rightarrow$ (10 4 2 2) y el vector resultante es el doble de C codificado en (4221).

Por consiguiente, es necesario introducir recodificadores de dígitos en el árbol de CSAs

binarios para obtener la suma decimal correcta. Así, con respecto a un árbol CSA binario, los árboles CSA decimales propuestos son un 45% más lentos y tienen un 20% más área. Sin embargo, respecto a otras implementaciones de sumadores decimales libres de propagación de acarreo, los sumadores propuestos son al menos un 20% más rápidos y requieren un 40% menos área. Por lo tanto, constituyen una alternativa muy competitiva para implementar sumadores decimales o sumadores combinados binarios/decimales de múltiples operandos.

5. Multiplicadores paralelos de punto fijo decimal.

Aunque la multiplicación decimal es una operación importante y muy frecuente en las aplicaciones comerciales, las implementaciones hardware actuales presentan un rendimiento muy bajo respecto a las unidades de multiplicación binaria.

La multiplicación de enteros y en punto fijo (tanto binaria como decimal) consiste en tres etapas: generación de productor parciales, reducción (suma) de los productos parciales a dos operandos y conversión final (mediante una suma con propagación de acareo) a una representación no redundante.

Las unidades binarias de altas prestaciones implementan multiplicadores paralelos. Sin embargo, la multiplicación decimal es más compleja de implementar debido a la complejidad para generar los múltiplos decimales y la ineficiencia para representar valores decimales en sistemas basados en señales binarias. Estos problemas complican la generación y reducción de productos parciales.

Las implementaciones comerciales de multiplicadores decimales están basadas en algoritmos iterativos (serie) y presentan bajo rendimiento. Para obtener un multiplicador decimal de altas prestaciones es necesario generar los productos parciales de forma paralela y reducirlos eficientemente usando un sumador libre de propagacion de acarreo (de baja latencia y con un coste de área asumible).

En el Capítulo 7 hemos propuesto dos arquitecturas para enteros o punto fijo de multiplicadores paralelos decimales y dos multiplicadores paralelos binario/decimal combinados.

Para generar los productos parciales hemos optado por recodificar los dígitos BCD del multiplicador y precomputar un conjunto reducido de múltiplos decimales del multiplicando. Cada dígito recodificado del multiplicador da lugar a un producto parcial seleccionando el múltiplo indicado por el valor de dicho dígito (que puede ser negativo). De este modo, todos los productos parciales son generados en paralelo. Hemos desarrollado tres recodificaciones diferentes para el multiplicador: la primera (SD radix-10) transforma los dígitos BCD $Y_i \in \{0,9\}$ a dígitos en el conjunto $Y_i^b \in \{5,\ldots,0,5\}$. Las otras dos recodificaciones, SD radix-5 y SD radix-4, separan cada dígito BCD Y_i del multiplicador en dos dígitos, de la forma $Y_i = Y_i^H + Y_i^L$, donde $Y^L \in \{-2,-1,0,1,2\}$ y $Y_i^H \in \{0,5,10\}$ (SD radix-5) o $Y_i^H \in \{0,4,8\}$ (SD radix-4).

Los productos parciales se representan en código (4221) o (5211) para poder ser reducidos de forma eficiente usando los sumadores decimales en árboles propuestos en el Capítulo 6.

Del estudio comparativo de área-latencia realizado, incluyendo al único multiplicador decimal paralelo propuesto con anterioridad, hemos concluido que los multiplicadores propuestos para 16 dígitos de precisión (formato IEEE 754-2008 Decimal64) pueden ser suficien-

temente competitivos para ser implementados en una unidad decimal comercial: presentan entre un 25% y 40% menos de latencia y un 35% menos área que el diseño con el mejor rendimiento propuesto hasta el momento. Además tienen un área similar que un multiplicador binario paralelo de 64 bits y solo un 25% más de latencia.

6. Multiplicador y sumador-multiplicador combinado de punto flotante decimal.

En el Capítulo 7 hemos propuesto varios esquemas para implementar unidades de punto flotante de multiplicación decimal y suma-multiplicación decimal combinada conformes al estándar IEEE 754-2008. Dichos esquemas emplean los multiplicadores paralelos de punto fijo propuestos y el sumador decimal con redondeo presentado en el Capítulo 6 para reducir la latencia de otra propuesta anterior. Un objetivo inmediato será proporcionar una implementación detallada de estas unidades y estudiar su adaptación a las unidades decimales IEEE 754-2008 de punto flotante.

7. Unidad de división decimal basada en dígito-recurrencia.

En el Capítulo 8 presentamos una nueva arquitectura de altas prestaciones para división decimal de punto flotante basada en un algoritmo de dígito recurrencia sin restauración en base 10. Este tipo de algoritmos proponen una recurrencia para una función, denominada residuo, del que se obtiene un dígito del cociente en cada iteración (convergencia lineal), de tal forma que si el residuo se mantiene en un cierto rango de valores, el resto de la división converge a 0.

El algoritmo usa técnicas convencionales desarrolladas para acelerar la división binaria en base 2^k, como la representación del cociente en forma redundante usando dígitos con signo. En concreto, para base 10, se ha usado el conjunto de dígitos mínimamente redundante $\{-5, -4, \ldots, 0, \ldots, 4, 5\}$. Otra técnica empleada ha sido la selección de dígitos del cociente mediante la comparación con constantes de una estimación del residuo, representado de forma redundante mediante una doble palabra (suma y acarreo).

Con el fin de optimizar área y latencia para la implementación en base 10, hemos empleado nuevas técnicas como el uso de codificaciones binarias alternativas (distintas de BCD) de los dígitos decimales, estimación mediante truncamiento del residuo decimal en una posición binaria dentro de un dígito, diseño específico de un sumador decimal de acarreo propagado con redondeo adaptado para realizar diversas tareas, computación previa de los múltiplos decimales del divisor, y recolocación de los registros con el fin de explotar las características especiales de registros con multiplexación incorporada. Además, la conversión del cociente desde una representación redundante a no redundante se realiza conjuntamente con el redondeo en una misma iteración del sumador decimal.

Para minimizar el tamaño (en bits) de la estimación redundante (suma y acarreo) del residuo y las constantes de selección y reducir así la latencia y el coste hardware de la selección de dígitos, hemos usado una codificación (5211) para representar los dígitos de la estimación redundante del residuo.

Hemos implementado divisores decimales para 16 y 34 dígitos de precisión que se corresponden con los formatos Decimal64 y Decimal128 del estándar IEEE 754-2008. Los resultados de la evaluación usando nuestro modelo de área-retardo muestran que la arquitectura propuesta presenta latencias comparables a las del divisor decimal que presenta el mejor rendimiento hasta la fecha, pero usando un 30% menos de área. Además, la arquitectura propuesta puede ser incorporada de forma simple y eficiente a las unidades de punto flotante decimal de los microprocesadores de IBM Power6 y z10, reduciendo la latencia de las instrucciones de división decimal en un factor 2, mientras que el área de la unidad aumenta menos de un 20%.

Bibliography

[1] P. H. ABBOTT ET AL., *Architecture and Software Support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point Arithmetic*, IBM Journal of Research and Development, 43 (1999), pp. 723–760.

[2] G. M. AMDAHL, G. A. BLAAUW AND F. P. BROOKS, *Architecture of the IBM System/360*, IBM Journal of Research and Development, 8 (1964), pp. 87–53.

[3] S. F. ANDERSON, J. G. EARLE, R. E. GOLDSCHMIDT AND D. M. POWERS, *The IBM System/360 Model 91: Floating-point Execution Unit*, IBM Journal of Research and Development, 11 (1967), pp. 34–53.

[4] H. ANDO, Y. YOSHIDA, A. INOUE ET AL., *A 1.3-GHz Fifth-Generation SPARC64 Microprocessor*, IEEE Journal of Solid-State Circuits, 38 (2003), pp. 1896–1905.

[5] E. ANTELO, T. LANG, P. MONTUSCHI AND A. NANNARELLI, *Digit-Recurrence Dividers with Reduced Logical Depth*, IEEE Transactions on Computers, 54 (2005), pp. 837–851.

[6] A. BAJWA AND R. STECK, *A Fast Floating Point Unit in the i960 General-Purpose Embedded Processor Family*, in Proc. of the Wescon '90 Conference, Anaheim, CA, USA, November 1990, pp. 218–222.

[7] C. BAUGH AND B. WOOLEY, *A Two's Complement Parallel Array Multiplication Algorithm*, IEEE Transactions on Computers, C-22 (1971), pp. 1045–1047.

[8] A. BEAUMONT-SMITH AND C. LIM, *Parallel-Prefix Adder Design*, in Proceedings of the 15^{th} IEEE Symposium on Computer Arithmetic, June 2001, pp. 218–225.

[9] O. BEDRIJ, *Carry-Select Adder*, IRE Transactions on Electronic Computers,, EC-11 (1962), pp. 340–346.

[10] M. BHAT, J. CRAWFORD, R. MORIN AND K. SHIV, *Performance Characterization of Decimal Arithmetic in Commercial Java Workloads*, in IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS 2007), Apr. 2007, pp. 54–61.

[11] D. BOGGS ET AL., *The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology*, Intel Technology Journal, 8 (2004), pp. 1–17.

[12] G. BOHLENDER AND T. TEUFEL, *A Decimal Floating-Point Processor for Optimal Arithmetic*, in Computer Arithmetic: Scientific Computation and Programming Languages, B. G. Teubner Stuttgart, 1987, pp. 31–58.

[13] W. BUCHHOLZ, *Fingers or Fists? (The Choice of Decimal or Binary representation)*, Communications of the ACM, 2 (1959), pp. 3–11.

[14] W. BULTMANN, W. HALLER, H. WETTER AND A. WÖRNER, *Binary and Decimal Adder Unit, US Patent No. 6292819*, Sept. 2001.

[15] N. BURGESS, *The Flagged Prefix Adder and its Applications in Integer Arithmetic*, Journal of VLSI Signal Processing, 31 (2002), pp. 263–271.

[16] ——, *Prenormalization Rounding in IEEE Floating-Point Operations Using a Flagged Prefix Adder*, IEEE Trans. on VLSI Systems, 13 (2005), pp. 266–277.

[17] A. W. BURKS, H. H. GOLDSTINE AND J. VON NEUMANN, *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*, Inst. for Advanced Study, Princeton, N. J., Jun. 1946.

[18] BURROUGHS CORPORATION, *Burroughs B5500 Information Processing Systems Reference Manual*, tech. report, Burroughs Corporation, Detroit, MI, USA, Jun. 1964.

[19] F. Y. BUSABA, C. A. KRYGOWSKI, W. H. LI, E. M. SCHWARZ AND S. R. CARLOUGH, *The IBM z900 Decimal Arithmetic Unit*, in Conference Record of the Asilomar Conference on Signals, Systems and Computers, vol. 2, Nov. 2001, pp. 1335–1339.

[20] F. Y. BUSABA, T. SLEGEL, S. CARLOUGH, C. KRYGOWSKI AND J. G. RELL, *The Design of the Fixed Point Unit for the z990 Microprocessor*, in Proceedings of the 14th ACM Great Lakes Symposium on VLSI 2004, Apr. 2004, pp. 364–367.

[21] R. P. CASE AND A. PADEGS, *Architecture of the IBM System/370*, Journal of the Association for Computing Machinery (ACM), 21 (1978), pp. 73–96.

[22] I. D. CASTELLANOS AND J. E. STINE, *Experiments for Decimal Floating-Point Division by Recurrence*, in 40^{th} Asilomar Conference on Signals, Systems and Computers, 2006. ACSSC '06, Oct.-Nov. 2006, pp. 1716–1720.

[23] ——, *Compressor Trees for Decimal Partial Product Reduction*, in 18th ACM Great Lakes Symposium on VLSI, May 2008, pp. 107–110.

[24] P. E. CERUZZI, *A History of Modern Computing, 2nd Ed.*, The MIT Press, 2003.

[25] M. A. CHECK AND T. J. SLEGEL, *Custom S/390 G5 and G6 Microprocessors*, IBM Journal of R&D, 43 (1999), pp. 671–680.

[26] D. CHEN AND S.-B. KO, *Design and Implementation of Decimal Reciprocal Unit*, in Canadian Conference on Electrical and Computer Engineering (CCECE 2007), Apr. 2007, pp. 1094–1097.

[27] W. D. CLINGER, *How to Read Floating Point Numbers Accurately*, in Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, June 1990, pp. 92–101.

[28] J. CLOUSER, M. MATSON, R. BADEAU, R. DUPCAK, S. SAMUDRALA, R. ALLMON AND N. FAIRBANKS, *A 600-MHz Superscalar Floating-Point Processor*, IEEE Journal of Solid-State Circuits, 34 (1999), pp. 1026–1029.

[29] M. S. COHEN, T. E. HULL AND V. C. HAMACHER, *CADAC: A Controlled-Precision Decimal Arithmetic Unit*, IEEE Transactions on Computers, C-32 (1983), pp. 370–377.

[30] M. CORNEA, C. ANDERSON, J. HARRISON, P. TANG, E. SCHNEIDER AND C. TSEN, *A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format*, in 18^{th} Symposium on Computer Arithmetic, June 2007, pp. 29–37.

[31] M. CORNEA AND J. CRAWFORD, *IEEE 754R Decimal Floating-Point Arithmetic: Reliable and Efficient Implementation for Intel Architecture Platforms*, Intel Technology Journal, 11 (2007), pp. 91–94.

[32] J. CORTADELLA AND J. M. LLABERRÍA, *Evaluation of A+B=K Conditions Without Carry-Propagation*, IEEE Transactions on Computers, 41 (1992), pp. 1484–1488.

[33] M. COWLISHAW, *IBM General Decimal Arithmetic Website*. http://speleotrove.com/decimal/.

[34] M. F. COWLISHAW, *Densely Packed Decimal Encoding*, IEE Proceedings Computers and Digital Techniques, 149 (2002), pp. 102–104.

[35] ——, *Decimal Floating-Point: Algorism for Computers*, in Proceedings of the 16^{th} IEEE Symposium on Computer Arithmetic, Jul. 2003, pp. 104–111.

[36] ——, *The DecNumber ANSI C Library, Version 3.32*. http://www2.hursley.ibm.com/decimal/decnumber.html, Dec. 2005.

[37] ——, *General Decimal Arithmetic Specification, Version 1.66*. http://www2.hursley.ibm.com/decimal, Mar. 2007.

[38] M. F. COWLISHAW, J. BLOCH AND J. DARCY, *Fixed, floating, and exact computation with Java's BigDecimal*, Dr. Dobb's Journal, 29 (2004), pp. 22–27.

[39] M. F. COWLISHAW, E. M. SCHWARZ, R. M. SMITH AND C. F. WEBB, *A Decimal Floating-Point Specification*, in Proceedings of the 15^{th} IEEE Symposium on Computer Arithmetic, June 2001, pp. 147–154.

[40] L. DADDA, *Some Schemes for Parallel Multipliers*, Alta Frequenza, 34 (1965), pp. 349–356.

[41] ——, *Multioperand Parallel Decimal Adder: A Mixed Binary and BCD Approach*, IEEE Transactions on Computers, 56 (2007), pp. 1320–1328.

[42] DIGITAL EQUIPMENT CORPORATION, *Software Product Description: COBOL-81/RSTS/E, Version 3.1*, Dec. 1990.

[43] G. DIMITRAKOPOULOS AND D. NIKOLOS, *High-Speed Parallel-Prefix VLSI Ling Adders*, IEEE Transactions on Computers, 54 (2005), pp. 225–231.

[44] A. Y. DUALE, M. H. DECKER, H.-G. ZIPPERER, M. AHARONI AND T. J. BOHIZIC, *Decimal Floating-Point in Z9: An Implementation and Testing Perspective*, IBM Journal Research and Development, 51 (2007), pp. 217–227.

[45] L. EISEN ET AL., *IBM POWER6 accelerators: VMX and DFU*, IBM Journal Research and Development, 51 (2007), pp. 663–684.

[46] M. D. ERCEGOVAC AND T. LANG, *On-the-Fly Conversion of Redundant into Conventional Representations*, IEEE Transactions on Computers, 36 (1987), pp. 895–897.

[47] ——, *Algorithms for Division and Square Root*, Kluwer Academic Publishers, 1994.

[48] ——, *Digital Arithmetic*, Morgan Kaufmann Publishers, Inc., 2004.

[49] M. A. ERLE, J. M. LINEBARGER AND M. J. SCHULTE, *Potential Speedup Using Decimal Floating-Point Hardware*, in 36^{th} Asilomar Conference on Signals, Systems and Computers, Nov. 2002, pp. 1073–1077.

[50] M. A. ERLE AND M. J. SCHULTE, *Decimal Multiplication Via Carry-Save Addition*, in IEEE International Conference on Application-Specific Systems , Architectures, and Processors, I. C. S. Press, ed., The Hague, Netherlands, June 2003, pp. 348–358.

[51] M. A. ERLE, E. M. SCHWARZ AND M. J. SCHULTE, *Decimal Multiplication With Efficient Partial Product Generation*, in 17^{th} IEEE Symposium on Computer Arithmetic, June 2005, pp. 21–28.

[52] EUROPEAN COMMISSION DIRECTORATE GENERAL II, *The Introduction of the Euro and the Rounding of Currency Amounts*, II/28/99-EN Euro Papers, DGII/C-4-SP(99) (1999), p. 32.

[53] G. EVEN AND P.-M. SEIDEL, *A Reduced-Area Scheme for Carry-Select Adders*, IEEE Trans. on Computers, C-42 (1993), pp. 1163–1170.

[54] ——, *A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication*, IEEE Trans. on Computers, 49 (2000), pp. 638–650.

[55] B. FU, A. SAINI AND P. P. GELSINGER, *Performance and Microarchitecture of the i486 Processor*, in Proc. of the IEEE International Conference on Computer Design ICCD'89, October 1989, pp. 182–187.

[56] D. M. GAY, *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions, Numerical Analysis Manuscript 90-10*, tech. report, AT&T Bell Laboratories, Nov. 1990.

[57] G. GERWIG, H. WETTER, E. M. SCHWARZ, J. HAESS, C. A. KRYGOWSKI, B. M. FLEISCHER AND M. KROENER, *The IBM eServer z990 Floating-Point Unit*, IBM Journal of Research and Development, 48 (2004), pp. 311–322.

[58] S. GOCHMAN, A. MENDELSON, A. NAVEH AND E. ROTEM, *Introduction to Intel Core Duo Processor Architecture*, Intel Technology Journal, 10 (2006), pp. 89–97.

[59] D. GOLDBERG, *Computer Arithmetic. Appendix H of Computer Architecture. A Quantitative Approach - J.L.Hennessy and D.A.Patterson.*, Morgan Kaufmann, 2002.

[60] H. H. GOLDSTINE, *The Computer from Pascal to von Neumann*, Princeton University Press, 1972.

[61] H. H. GOLDSTINE AND A. GOLDSTINE, *The Electronic Numerical Integrator and Computer (ENIAC)*, IEEE Annals of the History of Computing, 18 (1996), pp. 10–16.

[62] J. GRAD AND J. E. STINE, *A Hybrid Ling Carry-Select Adder*, in Conference Record of the 38th Asilomar Conference on Signals, Systems and Computers, vol. 2, Nov. 2004, pp. 1363–1367.

[63] W. HALLER, U. KRAUCH, T. LUDWIG AND H. WETTER, *Combined Binary/Decimal Adder Unit, US Patent No. 5928319*, Jul. 1999.

[64] P. HARTMAN, O. RUTZ AND P. SHAH, *Decimal Floating Point Computations in SAP NetWeaver 7.10*. White Paper available at http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101104, October 2007.

[65] A. HENINGER, *Zilog's Z8070 Floating Point Processor*, Mini Micro Systems, (1983), pp. 16/2/1–7.

[66] HEWLETT PACKARD, *Chapter 13: Data representations*, in Software Internal Design Specification for the HP-71, Vol. 1, Hewlett Packard Company, Palo Alto, CA, USA, December 1983, pp. 13.1–13.17.

[67] B. J. HICKMAN, A. KRIOUKOV, M. A. ERLE AND M. J. SCHULTE, *A Parallel IEEE P754 Decimal Floating-Point Multiplier*, in XXV IEEE Conference on Computer Design, Oct. 2007, pp. 296–303.

[68] R. HO, K. W. MAI AND M. A. HOROWITZ, *The Future of Wires*, Proceedings of the IEEE, 89 (2001), pp. 490–504.

[69] C. HUNTSMAN AND D. CAWTHRON, *The MC68881 Floating-Point Coprocessor*, IEEE Micro, 3 (1983), pp. 44–54.

[70] R. HYDE, *The Art of Assembly Language*, No Starch Press, September 2003.

[71] W. H. INMON, *Building the Data Warehouse, 4th Ed.*, Wiley, 2005.

[72] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS, INC., *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*, Aug. 1985.

[73] ——, *IEEE Standard for Radix-Independent Floating-Point Arithmetic, ANSI/IEEE Std 854-1987*, Mar. 1987.

[74] ——, *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008*, Aug. 2008.

[75] INTEL, *8080/ 8085 Floating-Point Arithmetic Library User's Manual*, Intel Corporation, Santa Clara, CA, USA, 1979.

[76] ——, *The iAPX 286 Programmer's Reference Manual*, Intel Corporation, Santa Clara, CA, USA, 1985.

[77] G. JABERIPUR AND A. KAIVANI, *Binary-Coded Decimal Digit Multipliers*, IET Comput. Digit. Tech., 1 (2007), pp. 377–381.

[78] F. B. JONES AND A. W. WYMORE, *Floating Point Feature On The IBM Type 1620*, IBM Technical Disclosure Bulletin, 62 (1962), pp. 43–46.

[79] W. KAHAN, *Floating-Point Arithmetic Besieged by Business Decisions*, in Proc. of the 17^{th} IEEE Symposium on Computer Arithmetic, Invited Keynote Address, Jun. 2005.

[80] C. N. KELTCHER, K. J. MCGRATH, A. AHMED AND P. CONWAY, *The AMD Opteron Processor for Multiprocessor Servers*, IEEE Micro, 23 (2003), pp. 66–76.

[81] R. D. KENNEY AND M. J. SCHULTE, *High-Speed Multioperand Decimal Adders*, IEEE Transactions on Computers, 54 (2005), pp. 953–963.

[82] R. D. KENNEY, M. J. SCHULTE AND M. A. ERLE, *High-Frequency Decimal Multiplier*, in Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, Oct. 2004, pp. 26–29.

[83] Y.-D. KIM, S.-Y. KWON, S.-K. HAN, K.-R. CHO AND Y. YOU, *A Hybrid Decimal Division Algorithm Reducing Computational Iterations*, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E89-A (2006), pp. 1807–1812.

[84] S. KNOWLES, *A Family of Adders*, in Proc. of the 14^{th} IEEE Symposium on Computer Arithmetic, Jun. 1999, pp. 30–34.

[85] D. E. KNUTH, *Volume 2: Seminumerical Algorithms. Third Edition*, in The Art of Computer Programming, Addison-Wesley Professional, Reading, Massachusetts, 1997, p. 762.

[86] P. KOGGE AND H. STONE, *A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations*, IEEE Transactions on Computers, C-22 (1973), pp. 786–793.

[87] L. KOHN AND N. MARGULIS, *The i860 64-bit Supercomputing Microprocessor*, in Proc. of Supercomputing '89, Reno, Nevada, November 1989, pp. 450–456.

[88] P. KONGETIRA, K. AINGARAN AND K. OLUKOTUN, *Niagara: A 32-Way Multithreaded SPARC Processor*, IEEE Micro, 25 (2005), pp. 21–29.

[89] R. LADNER AND M. FISCHER, *Parallel Prefix Computation*, Journal of the Association for Computing Machinery (ACM), 27 (1980), pp. 831–838.

[90] T. LANG AND J. D. BRUGUERA, *Floating-Point Multiply-Add-Fused with Reduced Latency*, IEEE Transactions on Computers, 53 (2004), pp. 988–1003.

[91] T. LANG AND A. NANNARELLI, *A Radix-10 Combinational Multiplier*, in 40th Asilomar Conference on Signals, Systems, and Computers, Oct. 2006, pp. 313–317.

[92] ——, *A Radix-10 Digit-Recurrence Division Unit: Algorithm and Architecture*, IEEE Transactions on Computers, 56 (2007), pp. 727–739.

[93] ——, *Combined Radix-10 and Radix-16 Division Unit*, in 41st Asilomar Conference on Signals, Systems, and Computers, Nov. 2007, pp. 967–971.

[94] R. H. LARSON, *High-Speed Multiply Using Four Input Carry-Save Adder*, IBM Tech. Disclosure Bulletin, 16 (1973), pp. 2053–2054.

[95] T. LING, *High-Speed Binary Adder*, IBM Journal of R&D, 25 (1981), pp. 156–166.

[96] K. LONEY AND G. KOCH, *Oracle 8i: The Complete Reference*, Mc. Graw-Hill, 2000.

[97] M. A. ERLE AND M. J. SCHULTE AND B. J. HICKMAN, *Decimal floating-point multiplication via carry-save addition*, in 18^{th} IEEE Symposium on Computer Arithmetic, June 2007, pp. 46–55.

[98] M. J. MACK, W. M. SAUER, S. B. SWANEY AND B. G. MEALEY, *IBM POWER6 reliability*, IBM Journal of Research and Development, 51 (2007), pp. 763–774.

[99] B. MARKS AND N. MILSTED, *ANSI X3.274-1996: American National Standard for Information Technology Programming Language REXX*, tech. report, ANSI, Feb. 1996.

[100] S. MATHEW, M. ANDERS, R. KRISHNAMURTHY AND S. BORKAR, *A 4Ghz 130nm Address Generation Unit with 32-bit Sparse-Tree Adder Core*, IEEE Journal of Solid-State Circuits, 38 (2003), pp. 689–695.

[101] C. MCNAIRY AND R. BHATIA, *Montecito: A Dual-Core, Dual-Thread Itanium Processor*, IEEE Micro, 25 (2005), pp. 10–20.

[102] P. MEANEY, S. SWANEY, P. SANDA AND L. SPAINHOWER, *IBM z990 Soft Error Detection and Recovery*, IEEE Trans. on Device and Materials Reliability, 5 (2005), pp. 419–427.

[103] S. MICROSYSTEMS, *The BigDecimal Java Class*, Java2 Platform, Standard Edition, (2003).

[104] MOTOROLA, *M68000 Family Programmer's Reference Manual*, Motorola Corporation, Phoenix, AZ, USA, 1992.

[105] A. NAINI, A. DHABLANIA, W. JAMES AND D. D. SARMA, *1-GHz HAL SPARC64 Dual Floating Point Unit with RAS Features*, in 15^{th} IEEE Symposium on Computer Arithmetic, Jun. 2001, pp. 173–183.

[106] H. NIKMEHR, B. PHILLIPS AND C.-C. LIM, *Fast Decimal Floating-Point Division*, IEEE Trans. VLSI Systems, 14 (2006), pp. 951–961.

[107] H.-J. OH ET AL., *A Fully Pipelined Single-Precision Floating-Point Unit in the Synergistic Processor Element of a CELL Processor*, IEEE Journal of Solid-State Circuits, 41 (2006), pp. 759–771.

[108] N. OHKUBO ET AL., *A 4.4 ns CMOS 54x54-bit Multiplier Using Pass-Transistor Multiplexer*, IEEE Journal of Solid State Circuits, 30 (1995), pp. 251–256.

[109] T. OHTSUKI ET AL., *Apparatus for Decimal Multiplication*, U.S. Patent No. 4,677,583, (1987).

[110] J. F. PALMER AND S. P. MORSE, *The 8087 Primer*, Wiley, 1984.

[111] D. PATIL, O. AZIZI, M. HOROWITZ, R. HO AND R. ANANTHRAMAN, *Robust Energy-Efficient Adder Topologies*, in Proc. 18th IEEE Symposium on Computer Arithmetic, Jun. 2007, pp. 16–25.

[112] B. RANDELL, *The Origins of Computer Programming*, IEEE Annals of the History of Computing, 16 (1994), pp. 6–13.

[113] R. K. RICHARDS, *Arithmetic Operations in Digital Computers*, D. Van Nostrand Company, Inc., New Jersey, 1955.

[114] L. RUBINFIELD, *A Proof of the Modified Booth's Algorithm for Multiplication*, IEEE Transactions on Computers, C-22 (1975), pp. 1014–1015.

[115] R. SACKS-DAVIS, *Applications of Redundant Number Representations to Decimal Arithmetic*, The Computer Journal, 25 (1982), pp. 471–477.

[116] M. R. SANTORO, G. BEWICK AND M. A. HOROWITZ, *Rounding Algorithms for IEEE Multipliers*, in 9^{th} IEEE Symposium on Computer Arithmetic, Sep. 1989, pp. 176–183.

[117] H. SCHMID, *Decimal Computation*, John Wiley & Sons, 1974. Reprinted in 1983 by Krieger Publishing Company (ISBN 0898743184).

[118] M. SCHMOOKLER AND A. WEINBERGER, *High Speed Decimal Addition*, IEEE Trans. on Computers, c-20 (1971), pp. 862–866.

[119] M. J. SCHULTE, N. LINDBERG AND A. LAXMINARAIN, *Performance Evaluation of Decimal Floating-Point Arithmetic*, in Proceedings of the 6th IBM Austin Center for Advanced Studies Conference, Feb. 2005.

[120] E. SCHWARZ, *Binary Floating-Point Unit Design*, in High Performance Energy Efficient Microprocessor Design, R. Krishnamurthy and V. G. Oklobdzija, eds., Springer, Mar. 2006, pp. 189–208.

[121] ——, *Future Research in Computer Arithmetic*, in The Future of Computing, essay in memory of Stamatis Vassiliadis, Sep. 2007, pp. 114–120.

[122] E. SCHWARZ AND S. R. CARLOUGH, *Power6 Decimal Divide*, in Proc. of the 18th IEEE International Conference on Application-Specific Systems, Architectures and Processors, Jul. 2007, pp. 128–133.

[123] E. M. SCHWARZ ET AL., *The Microarchitecture of the IBM eServer z900 Processor*, IBM Journal of Research and Development, 46 (2002), pp. 381–395.

[124] E. M. SCHWARZ, R. M. A. III AND L. J. SIGAL, *A Radix-8 CMOS S/390 Multiplier*, in 13^{th} IEEE Symposium on Computer Arithmetic, July 1997, pp. 2–9.

[125] P. M. SEIDEL AND G. EVEN, *Delay-Optimized Implementation of IEEE Floating-Point Addition*, IEEE Transaction on Computers, 53 (2004), pp. 97–113.

[126] K. L. SHEPARD ET AL., *Design Methodology for the S/390 Parallel Enterprise Server G4 Microprocessors*, IBM Journal of Research and Development, 41 (1997), pp. 515–547.

[127] B. SHIRAZI, D. Y. Y. YUN AND C. N. ZHANG, *RBCD: Redundant Binary Coded Decimal Adder*, in IEE Proceedings - Computers and Digital Techniques, vol. 136, Mar. 1989, pp. 156–160.

[128] J. SKLANSKY, *Conditional-Sum Addition Logic*, IRE Transactions on Electronic Computers,, EC-9 (1960), pp. 226–231.

[129] W. STALLINGS, *Computer Organization and Architecture. Designing for Performance, 6th edition*, Prentice Hall, 2003.

[130] G. L. STEELE JR. AND J. L. WHITE, *How to Print Floating-Point Numbers Accurately*, in Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, June 1990, pp. 112–126.

[131] I. SUTHERLAND, R. SPROULL AND D. HARRIS, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann, 1999.

[132] A. SVOBODA, *Decimal Adder with Signed-Digit Arithmetic*, IEEE Trans. on Computers, C (1969), pp. 212–215.

[133] P. TANG, *Binary-Integer Decimal Encoding for Decimal Floating-Point*. http://754r.ucbtest.org/issues/decimal/bid_rationale.pdf, Jul. 2005.

[134] G. S. TAYLOR AND D. A. PATTERSON, *VAX Hardware for the Proposed IEEE Floating-Point Standard*, in Proc. of the 5^{th} IEEE Symposium on Computer Arithmetic, 1981, pp. 190–196.

[135] TEXAS INSTRUMENTS, *TI-89/TI-92 Plus Developers Guide, Beta Version .02*, Texas Instruments, 2001.

[136] J. THOMPSON, N. KARRA AND M. J. SCHULTE, *A 64-bit Decimal Floating-Point Adder*, in IEEE Computer Society Annual Symposium on VLSI, Feb. 2004, pp. 297–298.

[137] S. D. TRONG, M. SCHMOOKLER, E. M. SCHWARZ AND M. KROENER, *P6 Binary Floating-Point Unit*, in 18th IEEE Symposium on Computer Arithmetic, Jun. 2007, pp. 77–86.

[138] A. TSANG AND M. OLSCHANOWSKY, *A Study of DataBase 2 Customer Queries, TR. 03.413*, tech. report, IBM Santa Teresa Laboratory, San Jose, CA, USA, Apr. 1991.

[139] C. TSEN, S. GONZÁLEZ-NAVARRO AND M. SCHULTE, *Hardware Design of a Binary Integer Decimal-based Floating-point Adder*, in XXV IEEE International Conference on Computer Design, Oct. 2007, pp. 288–295.

[140] C. TSEN, M. SCHULTE AND S. GONZLEZ-NAVARRO, *Hardware Design of a Binary Integer Decimal-based IEEE P754 Rounding Unit*, in IEEE International Conference on Application-Specific Systems, Architectures, and Processors, Jul. 2007, pp. 115–121.

[141] T. UEDA, *Decimal Multiplying Assembly and Multiply Module. U.S. Patent No. 5,379,245*, January 1995.

[142] S. VASSILIADIS, D. S. LEMON AND M. PUTRINO, *S/370 Sign-Magnitude Floating-Point Adder*, IEEE Journal of Solid-State Circuits, 24 (1989), pp. 1062–1070.

[143] S. VASSILIADIS AND E. SCHWARZ, *Generalized 7/3 Counters. U.S. Patent No. 5,187,679*, Feb. 1993.

[144] A. VÁZQUEZ AND E. ANTELO, *Conditional Speculative Decimal Addition*, in 7^{th} Conference on Real Numbers and Computers (RNC 7), July 2006, pp. 47–57.

[145] ——, *Decimal Arithmetic Units for Financial and e-Commerce Servers: A Case Study of High-Performance Decimal Addition*, in XVII Jornadas de Paralelismo, Sept. 2006.

[146] ——, *Constant Factor Cordic for Decimal BCD Input Operands*, in 8^{th} Conference on Real Numbers and Computers (RNC 8), Jul. 2008, pp. 83–91.

[147] ——, *New Insights on Ling Adders*, in 19th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP), Jul. 2008, pp. 233–238.

[148] A. VÁZQUEZ, E. ANTELO AND P. MONTUSCHI, *A New Family of High-Performance Parallel Decimal Multipliers*, in 18^{th} IEEE Symposium on Computer Arithmetic, June 2007, pp. 195–204.

[149] ——, *A Radix-10 SRT Divider Based on Alternative BCD Codings*, in XXV IEEE International Conference on Computer Design (ICCD 2007), Oct. 2007, pp. 280–287.

[150] ——, *Improved Design of High-Performance Parallel Decimal Multipliers*, Submitted to IEEE Transactions on Computers. Under Revision., (2008).

[151] J. VON NEUMANN, *First Draft of a Report on the EDVAC*, Tech. Report Reprinted in IEEE Annals of History of Computing, no. 4, 1993, Moore School, University of Pennsilvania, 1945.

[152] C. D. WAIT, *IBM PowerPC 440 FPU with Complex-Arithmetic Extensions*, IBM Journal of Research and Development, 49 (2005), pp. 249–254.

[153] C. S. WALLACE, *A Suggestion for a Fast Multiplier*, IEEE Transactions on Computers, EC-13 (1964), pp. 14–17.

[154] L.-K. WANG AND M. J. SCHULTE, *Decimal Floating-Point Division Using Newton-Raphson Iteration*, in 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), September 2004, pp. 84–95.

[155] ——, *Decimal Floating-Point Square Root Using Newton-Raphson Iteration*, in 16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP), July 2005, pp. 309–315.

[156] ——, *A Decimal Floating-Point Divider Using Newton-Raphson Iteration*, The Journal of VLSI Signal Processing, 49 (2007), pp. 3–18.

[157] ——, *Decimal Floating-Point Adder and Multifunction Unit with Injection-Based Rounding*, in 18^{th} IEEE Symposium on Computer Arithmetic, June 2007, pp. 56–68.

[158] L. K. WANG, C. TSEN, M. J. SCHULTE AND D. JHALANI, *Benchmarks and Performance Analysis of Decimal Floating-Point Applications*, in XXV IEEE International Conference on Computer Design, Oct. 2007, pp. 164–170.

[159] Y. WANG, C. PAI AND X. SONG, *The Design of Hybrid Carry-Lookahed/Carry-Select Adders*, IEEE Trans. Circuits Syst. II, 49 (2002), pp. 16–24.

[160] C. F. WEBB, *IBM z10 - The Next-Generation Mainframe Microprocessor*, IEEE Micro, 28 (2008), pp. 19–29.

[161] M. H. WEIK, *A Third Survey of Domestic Electronic Digital Computing Systems, Report No. 1115*, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland, Mar. 1961.

[162] A. WEINBERGER AND J. L. SMITH, *A One-Microsecond Adder Using One-Megacycle Circuitry*, IRE Transactions on Electronic Computers, EC-5 (1956), pp. 65–73.

[163] G. S. WHITE, *Coded Decimal Number Systems for Digital Computers*, Proceedings of the IRE, 41 (1953), pp. 1450–1452.

[164] H. YABE, Y. OSHIMA, S. ISHIKAWA, T. OHTSUKI AND M. FUKUTA, *Binary Coded Decimal Number Division Apparatus, U.S. Patent 4635220*, Jan. 1987.

[165] W. YEH AND C. JEN, *High-Speed Booth Encoded Parallel Multiplier Design*, IEEE Trans. on Computers, 49 (2000), pp. 692–701.

[166] M. YILMAZ, A. MEIXNER, S. OZEV AND D. J. SORIN, *Lazy Error Detection for Microprocessor Functional Units*, in 22^{nd} International Symposium on Defect and Fault Tolerance in VLSI Systems, Sep. 2007, pp. 361–369.

[167] R. K. YU AND G. B. ZYNER, *167 MHz Radix-4 Floating Point Multiplier*, in 12^{th} IEEE Symposium on Computer Arithmetic, Jul. 1995, pp. 149–154.

[168] B. ZEYDEL, T. KLUTER AND V. OKLOBDZIJA, *Efficient Mapping of Addition Recurrence Algorithms in CMOS*, in Proc. 17th IEEE Symposium on Computer Arithmetic, June 2005, pp. 107–113.

[169] R. ZLATANOVICI ET AL., *Power-Performance Optimal 64-Bit Carry-Lookahead Adders*, in Proc. ESSCIRC 2003, Sep. 2003, pp. 321–324.