بسم الله الرحمن الرحيم

Sharif University of Technology

Frameworks for the Exploration and Implementation of Generalized Carry Free Redundant Number Systems

A dissertation submitted to the Department of Computer Engineering of Sharif University of Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Ghassem Jaberipur

Advisor: Dr. Mohammad Ghodsi
Associate professor, Computer Engineering Dept. Sharif Univ. of Technology

Co-advisor: Dr. Behrooz Parhami
Professor, Dept. Elec. & Computer Eng., Univ. of California at Santa Barbara

December 2004

To:

My Pentode of loved ones, *Wife, Parents, and Sons*

I thank

Allah, for all the guidance I have received in my life and during preparation of this dissertation,

many people, from Sharif University of Technology and Shahid Beheshti University, who helped me during the last seven years,

and particularly Professor Behrooz Parhami from University of California at Santa Barbara, and acknowledge and appreciate, with great pleasure, all the help and teaching received from him.

Abstract

Redundant number systems provide for carry-free arithmetic, where the result of arithmetic operations is achieved, in redundant format, without the need for latent carry propagation. However conversion of the result to a conventional nonredundant representation, always, requires carry propagation. Therefore, efficient use of redundant number systems is feasible when a series of arithmetic operations is to be performed before the need arises to obtain the result in a nonredundant representation. Redundant number systems have been used in several special purpose integrated designs (e.g., DSP applications) and also as intermediate number representation in complex arithmetic operations implemented for general purpose processors. But we have not encountered, in the literature, any general purpose application of redundant number systems in the sense that separate arithmetic operations, invoked by separate machine instructions of a general purpose executable program, accept redundantly represented operands and produce such results, in a carry-free manner or in constant time, independent of the length of the operands.

In this research we have established the characteristics of a general purpose carry-free arithmetic environment including a suitable redundant number representation system, a general purpose processor with carry-free arithmetic instructions, and special code optimizers to convert conventional arithmetic instructions to their carry-free counterparts. But the main trust is on development of the most suitable redundant number representation, which should provide for the most efficient (in terms of speed, area, regularity, etc.) *representationally closed* carry-free arithmetic. We start with investigation of previous works on signed digit number systems, namely the conventional signed digit number system of, the pioneer in the field, professor Avezienis, the generalized signed digit number systems of professor Parhami, and the hybrid signed digit number systems of professors Phatak and Koren, and offer some algorithmic improvement for carry-free addition of representation paradigms of signed digit number systems. Then, we gradually build up the desired redundant encoding system by introducing the class of *stored transfer* representation of redundant number systems with the idea of mixing the stored carry mechanism and signed digit number systems. Next, we introduce the class of *weighted bit-set* (WBS) encoding of redundant number systems as a unification of the generalized signed digit, and the hybrid signed digit number systems. A WBS-encoded number, in each binary position, has a collection of zero or more *posibits* (positively weighted bits) and *negabits* (negatively weighted bits). This generalization led us to develop the *extended hybrid redundant* number systems including an interesting symmetric subclass not foreseen in the hybrid redundancy scheme of Phatak and Koren. With the novel concept of *inverted encoding* of negabits, we manage to develop efficient and regular designs for universal hybrid redundant adders, based solely on standard full/half adders, with the possibility of employing conventional carry accelerating techniques. Further generalization was fruitful, and resulted in development of the new concept of *two-valued digits* (*twit*) and the *weighted twit-set* (WTS) encodings. A twit may assume any two integer values, and a WTS encoding has the same structure as a WBS encoding except that it may contain any twit besides posibits and negabits. The latter is believed to be the most comprehensive encodings covering all the redundant and nonredundant positional number systems, we have encountered, including those with noncontiguous digit sets, possibly, not including zero. Then we present high level designs for representationally closed multiplication and division of some selected redundant number representations, with the encodings studied, and show advantages of our designs over some of the state of the art multiplication and division methods. Our multiplier design includes a special Booth recoder for redundant multipliers, which produces one multiple per every two binary positions of the multiplier in spite of extra redundancy bits. Floating point arithmetic is another vital topic in our investigation of the desired redundant encodings, where we show the suitability and advantages of our selected encodings. To complete our study of the desired redundant representations, we design arithmetic support functions, for the selected encodings, such as negation, binary and radix shifts, zero and sign detection, and over/underflow detection and correction.

Keywords: Positional number systems, Redundant number systems, Carry-free computer arithmetic, Periodic number systems, Symmetric number systems, Hybrid redundancy, Signed digits, Number encodings and representations, Design and implementation of redundant arithmetic operations.

Contents

List of Symbols and Abbreviations

$'$ $''$ $'''$	Used to distinguish entities with the same symbols
▫	A transfer digit (Fig. 3.3, and 3.5)
●, ○	Dot notation for bit or posibit, negabit (Fig. 7.2)
®	Collection of weighted bits in a redundant position
▣	Dot notation for unibit (Fig. 7.2)
■, □	Dot notation for doublebit, negadoublebit (Fig. 7.2)
∇	Empty position lacking any twit
$\vert$	Logical or, + is reserved for addition in Chapter 10
$\oplus$	Exclusive or
!	Logical not
α, β	Parameters of digit set $\Delta = \{\alpha, \ldots \beta\}$
Γ	$\Gamma_{(j)} = \sum_{0\leq i<j} \gamma_i$ (Def. 6.8), $\Gamma_j = \sum_{0\leq i<mj} \gamma_i$, $\Gamma^+_i = \sum_{0\leq j\leq i-1} 2^j\Gamma_j$, $\Gamma^+ = \Gamma^+_k$ (Def. 7.4)
γ	Gap size for the twit $\{\lambda, \lambda + \gamma\}$ (Def. 7.1)
Δ	Digit set
Δ_s	Maximal symmetric subset of a digit set Δ (Def. 8.3)
δ	Range of transfers values (Chapter 3)
ε	Effective gap (Def. 7.4)
η	Width of the binary encoding of a digit set (Intro.)
φ, π, σ, ξ	Both cases: arbitrary bit values
ξ	Representational power coefficient (Chapter 8)
Λ	$\Lambda_{(j)} = \sum_{0\leq i<j} \lambda_i$ (Def. 6.8), $\Lambda_j = \sum_{0\leq i<mj} \lambda_i$, $\Lambda^+_i = \sum_{0\leq j\leq i-1} 2^j\Lambda_j$, $\Lambda^+ =\Lambda^+_k$ (Def. 7.4)
Λ	Range loss (Chapter 10)
λ	Lower value for the twit $\{\lambda, \lambda + \gamma\}$ (Def. 7.1)
μ	Number of outputs in (ν; μ)-compressor (Cor. 7.1)
ν	Number of inputs in (ν; μ)-compressor (Cor. 7.1)
ρ, ρ'	Redundancy index of a digit set, and of its main part (Chap. 3)
ρ_i	Redundancy index of position i (Def. 7.4)
τ, ω, σ', σ''	Arithmetic functions representing carry-free addition
υ	unit digit value (Def. 8.2)
Ω	Specific WBS encoding
Ψ	Cardinality of a set of integers
a, b	Both lower and upper case: arbitrary bit values
Biased	Twit encoding with λ encoded as 0 (Def. 7.2)
BSD	Binary signed-digit representation
C, c	Carry bit/twit
CCFAA	Conventional Carry-Free Addition Algorithm (Chapter 2)
CHRA	Compare with Half Radix Algorithm (Chapter 2)
D	Digit set of the main part of a stored transfer number (Chapter 1)
D, D_h, D_l	Divisor, Most significant and least significant halves of the divisor (Chapter 9)
d	Number of transfers in a transfer set $G = \{c_0, c_1, \ldots, c_{d-1}\}$
d^{max}	Maximum gap in a set of integers represented by a collection of twits
E, e, e_s	Encoding cost, encoding efficiency, symmetric encoding efficiency (Def. 7.4)
G	Transfer set
g	Width extension for WBS representation (Chapter 7), arbitrary digit (Chap. 10)
GSD	Generalized signed-digit representation
h	Period or power of 2 in $r = 2^h$ (Def. 7.10)

HSD	Hybrid Signed Digit
HRSD	High Radix Signed Digit (Chapter 2)
i, j, l	Arbitrary indices
k	Number of digit positions (Def. 7.3), or number of radix-r digits
LSD	Least significant digit (LSB is used in binary)
M_i, M, m	Partial multiplicity number, Multiplicity number, number of twits in a position
MSD	Most significant digit (MSB is used in binary)
N_i, N	$N_i = (n_{i-1} \ldots n_1 n_0)_{\text{two}}$; $N = N_k$ (Def. 7.3)
n_i	Number of negabits in position i (Def. 7.3)
Negabit	Two-valued digit in $\{-1, 0\}$ (Def. 7.1)
(n, p)	Digit composed of a negabit and a posibit
P_i, P	$P_i = (p_{i-1} \ldots p_1 p_0)_{\text{two}}$ (Def. 6.2); $P = P_k$ (Def. 7.3)
p_i	Position-sum in carry-free addition (Chap. 2)
Posibit	Two-valued digit in $\{0, 1\}$; same as bit (Def. 7.1)
PPA	Partial Product Array (Chapter 9)
Q	Quotient (Chapter 9)
R_i, R	$R_i = (\rho_{i-1} \ldots \rho_1 \rho_0)_{\text{two}}$ (Def. 7.4); $R = R_k$ (Def. 7.4)
r	Radix (base) of a number system
S, s	Sum bit/twit
$s^{\pm}$	Sink signals (Chapter 10)
SBC	Stored borrow-or-carry; digits in $[-1, 2]$
SC	Stored carry; digits in $[0, 2]$
SD	Signed Digit
SDB	Stored-double-borrow; digits in $[-2, 1]$
SDC	Stored-double-carry; digits in $[0, 3]$
Sink	The action of absorbing a visiting over/underflow value (Chapter 10)
SPT	Stored posibit transfer (Chapter 8)
STC	Stored-triple-carry
SUT	Stored-unibit-transfer (Def. 7.11)
T, t	Arbitrary transfer values
T_1, T_2	Arbitrary twits (Chapter 9)
Twit	Two-valued digit (Def. 7.1)
Unibit	Two-valued digit from the set $\{-1, 1\}$ (Def. 7.11)
u	Immediate real underflow signal (Chapter 10)
v	Immediate real overflow signal (Chapter 10)
u, v, w	Both lower and upper case: arbitrary twit logical values
WBS	Weighted bit-set encoding (Chapter 4)
WTS	Weighted twit-set encoding (Def. 7.4)
X	Don't care
x, y, z	Both lower and upper case: arbitrary twit logical values
Z	Dividend (Chapter 9)
$z^{\pm}$	Zip signals (Chapter 10)
Zip	Action of passing over to right of a visiting over/underflow value (Chapter 10)
ZSD	Zero and sign (zip and sink) detector (Chapter 10)

List of Definitions

List of Lemmas

List of Theorems

List of Corollaries

List of Examples

List of Algorithms

List of Figures

List of Tables

Chapter 1 Introduction

Computer arithmetic operations serve as an essential part of any executable program, whether referenced directly in the source program or generated as part of the execution of other programming features such as address calculation for array references or string manipulation. Improving the execution speed of arithmetic operations, individually or collectively, leads to better overall program execution efficiency. Arithmetic operations, in general, are defined in terms of the four basic operations, namely division, multiplication, subtraction, and addition. Division is performed either by repeated subtractions or through a converging series of multiplications. Multiplication is either simulated by repeated additions or involves at least one full word-length addition after the process of partial product reduction, where the latter is effectively a multi-operand addition. Finally subtraction is usually performed through adding the negative of the subtrahend to the subtractor. Addition operation has thus been recognized as the prime operation for implementing other computer arithmetic operations. Therefore improving the performance (i.e., cost/speed) of addition operation enhances the performance of all other arithmetic operations, leading, in general, to improved execution efficiency of executable programs.

Carry propagate adders exhibit the simplest and least costly designs for addition operation. Their latency and cost is characterized as $O(k)$, where $k = \lceil \log_2 \Psi \rceil$, is the minimum number of bits required for representing Ψ different values, and Ψ is the cardinality of the numbers representable by the underlying number system. A variety of carry accelerating techniques have been devised in the design of addition logics to gain sub linear latency. Carry skip [Lehm61, Kant91] and carry select adders [Bedr62] show an $O(k^{1/2})$ latency. A logarithmic latency (i.e., $O(\log_2 k)$) is achieved by carry look-ahead [Bren82, Ngai86, Dora88] and conditional sum schemes [Skla60]. The reduced latency is gained, naturally, in price of more costly circuits. Hybrid adders as a combination of two different carry accelerating techniques, such as carry look-ahead and carry select schemes [Lync92], show certain optimizing cost/speed trade-offs. The operands and result of the above addition schemes are invariably represented in non-redundant format, and usually in conventional binary representations, such as unsigned binary, sign-magnitude, one's or two's complement number representation systems.

There are unconventional number representation systems, which lead to addition with sub logarithmic latency, namely the residue number system [Garn59], and the class of positional redundant number systems. In residue number systems Ψ, the maximum size for the range of values represented, is factorized as $\Psi = \Psi_1 \Psi_2 \ldots \Psi_n$, where the n (≥ 2) factors are relatively prime. Each of the representable Ψ values, say v, is uniquely represented by the list of numbers $\{v \bmod \Psi_i \mid 1 \leq i \leq n\}$. The addition latency is thus characterized as $O(\lceil \log_2 \max_i (\Psi_i) \rceil)$, showing considerable improvement over the above cases, but still depending on Ψ, and not yet a constant time addition.

Redundant number systems have been widely used for representing one or both operands and the result of an addition operation, e.g., [Wall64, Vuil83, Taka85, Hara87, Kawa90, Kawa91, Kuni93, Maki96]. The main example is partial product reduction as a major part of multiplication. In the Wallace tree technique [Wall64] of partial product reduction one of the operands of each addition operation is a binary number, while the other operand and the result are represented in the binary carry-save redundant representation, except for the first and the last addition with both operands in binary. In partial product reduction using signed digit encoding for intermediate results, both operands and the result of each addition, except for the final result, are represented as signed digit redundant numbers. It is a well-known fact that when the result of addition is represented in a redundant number system, constant time addition (i.e., O (1)) with no *interdigit* carry propagation (i.e., carry-free addition) is possible. But conversion of the redundantly represented result to its equivalent conventional non-redundant form imposes a non constant delay, which is, at best, logarithmically proportional to width of the result. Therefore carry-free addition shows speed advantage, only when a series of additions is to be performed before the need arises to convert the result to a non-redundant representation.

The literature is replete with examples of using carry-free addition as part of a more complex operation embedded in a special purpose arithmetic algorithm often realized in hardware (e.g., see the previous references and also [Kame80] for a DSP example). However, we have not encountered any use of carry-free addition in a general-purpose manner in the design of conventional processors. For example a simple arithmetic expression like $a + b + c + d$ in a source program is normally translated to three separate two's complement hardware addition instructions executed consecutively by conventional processors. It is desirable to benefit from the O(1) latency of carry-free addition in a general purpose manner. In other words when executing any program on a regular computer, the possibility of performing all arithmetic operations using only carry-free additions, increases the overall speed of arithmetic. The speed gain is actually possible if, after each two-operand operation, one can avoid the conversion of redundant results to a conventional non-redundant representation, such as two's complement. Besides the need for special hardware, designed for carry-free addition, essentially with longer operand widths, avoiding the post operation conversion requires widening the data registers. The conversion, however, may be unavoidable in certain points of program execution, namely on storing a result due to an assignment statement or generating a numeric output in response to a write request. The reason for the latter is obvious, but storing a redundantly encoded result in the random access memory words (e.g., by an assignment), essentially requires extra bits per word compared to that of conventional non-redundant representations. Consuming the available bits of a memory word for providing such extra bits reduces the representational efficiency. But increasing the length of registers used for storing the intermediate results of arithmetic operations seems justifiable. When such longer registers are available, all the arithmetic operations, between two storing points of the execution flow, can be done in redundant mode. The slow redundant to non-redundant conversion operations are then restricted to assignment and write statements in the course of execution of a program. The former could even be avoided by widening the memory words or by designing special wide-word *arithmetic caches*, where any data transfer from the arithmetic cache to main memory would require a carry propagating conversion operation. Wherever carry-free arithmetic instructions and their related hardware are available on a processor, a special code optimizer is needed to, besides other optimizations, replace the conventional non-redundant arithmetic instructions with their carry-free counterparts, and insert a conversion operation just before a store operation; hence no need for any modification in the existing programming language compilers.

In this research we explore the characteristics of the most suitable (in terms of speed, cost, and suitability for VLSI design) redundant number system/encoding to be employed for general purpose carry-free arithmetic, design the relevant carry-free arithmetic algorithms and present high level hardware designs for the four basic arithmetic operations and some arithmetic support functions. In the following definitions we associate carry-free arithmetic with redundant number representation, and do not address residue number systems.

Definition 1.1 (Carry-free addition): Whenever addition is possible, in a small, constant time, independent of operand widths (i.e., with no carry propagation chain beyond a constant number of binary positions), the process is called *carry-free addition*. ■

For example addition of two operands represented in a radix-r ($r \geq 3$) signed digit number system, requires *intradigit* carry propagation, but no interdigit propagation (i.e., a carry generated in a bit position of a digit, propagates at most up to the next higher significant digit). The conventional carry-free addition algorithm for ordinary signed digit [Aviz61] and Generalized Signed Digit (GSD) operands as given in [Parh90] has three steps:

- Compute, in parallel, the sum of radix-r equally weighted digits x_i and y_i as
$$p_i = x_i + y_i.$$
- Derive the interim sum digit w_i and transfer digit t_{i+1} satisfying
$$w_i = p_i - rt_{i+1}.$$
- Form the final sum digit
$$s_i = w_i + t_i$$

This algorithm may be applied to any representation for redundant numbers, but there may be different, possibly more efficient, algorithms for special representations [Phat94, Jabe01, Jabe03].

Definition 1.2 (Carry-free arithmetic): If the implementations of basic arithmetic operations, namely division, multiplication, subtraction and addition accept redundant operands, and derive a redundant result through carry-free sub-operations, the calculations performed by a series of these operations is called *carry-free arithmetic*. ■

For example the calculations embedded in computing $sin(x)$, may be implemented by carry-free addition and carry-free-addition-based multiplications and divisions. The input x may be provided in a non-redundant representation (e.g., two's complement), which is normally convertible to the desired redundant form in constant time, and in a carry-free manner. After deriving the result (i.e., $sin(x)$) by carry-free arithmetic in some redundant representation, it may be converted back to the desired non-redundant representation, which of course requires interdigit carry propagation. There may be special purpose processors specially designed to perform carry-free computation of trigonometric functions as there are special purpose DSP processors [Moto92]. But carry-free computation of $sin(x)$ is not possible by programming the computation to be executed by a conventional general purpose processor.

Definition 1.3 (Carry-free arithmetic instruction): A *carry-free arithmetic instruction*, when executed, invokes a special carry-free arithmetic hardware to perform a carry-free arithmetic operation. The operands and result are all represented in redundant format. ■

Definition 1.4 (General purpose carry-free arithmetic environment): A g*eneral purpose carry-free arithmetic environment* is composed of:

a) An efficient redundant number system for representing all the intermediate results of arithmetic expressions of the executable programs, whether originated by the source program or generated by the system.
b) A general purpose processor whose instruction set includes a subset of carry-free arithmetic instructions.
c) Supporting wide data registers to accommodate wider words of redundant results.
d) Optional arithmetic cache to temporarily store the wider words of redundant results, assigned to a variable.
e) Special code optimizers to, besides other conventional optimizations, replace conventional arithmetic codes by their carry-free counterparts, and insert special conversion instructions when a result is to be stored in the main memory or copied to an output file. ■

Among the five components of the general purpose carry-free arithmetic environment, we extensively investigate the options available for the first component, and search for better representations leading to the most suitable number system for component a) above. The desired characteristics of a suitable number representation for component a) are listed below:

Definition 1.5 (Encoding efficiency): *Encoding efficiency* of a digit set Δ represented by h binary digits is $e = |\Delta| / 2^h$, where $|\Delta|$ is the cardinality of Δ. ■

One of the goals in designing the desired number representation is, naturally, maximizing the encoding efficiency (i.e., e approaches 1, the maximum possible encoding efficiency for a non-redundant representation).

Definition 1.6 (Representational closure property): A combination of a number representation and an arithmetic algorithm implementing an arithmetic operation has the *representational closure property*, if the result can be represented in the same number representation of the operands, without any post- or pre-operation conversion. ■

For example, conventional two's complement arithmetic is *representationally closed*. Another example is the combination of Generalized Signed Digit (GSD) number representation and its related carry-free addition algorithm [Parh90]. The implementation given in [Phat94], for addition of Hybrid Signed Digit (HSD) numbers, and some of other cases of hybrid redundancy studied in [Phat01], also have the property of representational closure. While encoding-algorithm combinations that are not representationally closed can be useful and are in fact used in practice (e.g., partial product reduction), when comparing a representationally closed scheme against a scheme that is not closed, fairness dictates that the overhead of conversion from the intermediate representation to the ultimate encoding be taken into account in any cost/speed comparisons. Particularly, in a general purpose carry-free environment, where the same circuits implementing redundant arithmetic are to be used for further manipulations on the redundant results, representational closure property is the rule, irrespective of the possible extra cost or declined speed.

Definition 1.7 (Digit-set preservation property): Whenever the arithmetic algorithms designed for a given number representation, are capable of producing results that cover the whole range of the digit-set used for the operands, the representation-algorithm combination has the *digit-set preservation property*. ■

For example two's complement arithmetic preserves the digit-set of the operands. Another example with the digit-set preservation property is two's complement representation of maximally redundant radix-2^h signed digit numbers with the compare with half radix addition algorithm, while the digit-set is not preserved in the less redundant cases [Jabe03].

Definition 1.8 (Symmetric number representation): If the negation of each value represented by a number representation is also representable, the number representation is *symmetric*. ■

For example one's complement number representation is symmetric, while two's complement is not. The HSD number system [Phat94] is highly asymmetric. The negative range of a radix-2^h HSD digit set is half that of the non-negative range [Jabe01]. In general-purpose arithmetic applications, symmetry of the number system is important, such that when, for any reason, an asymmetric number system is used; only the symmetric range is actually beneficial, leading to lower practical encoding efficiency.

Definition 1.9 (Symmetric-range encoding efficiency): The symmetric range of a digit set $\Delta = [-\alpha, \beta]$, where $\alpha \geq 0$ and $\beta \geq 0$, is $\Delta_s = [-\mathit{min}(\alpha, \beta), \mathit{min}(\alpha, \beta)]$. Then the *symmetric-range encoding efficiency* is $e_s = |\Delta_s| / 2^h$, where h is the number of bits required for representing a digit in Δ. When $|\Delta| = |\Delta_s| + 1$, Δ is a *minimally asymmetric* digit set. ■

For example a two's complement digit set is minimally asymmetric. As another example consider the stored BSD transfer [Jabe01] digit set with $\Delta = [-2^{h-1} - 1, 2^{h-1}]$. If a gain in cost/speed is desirable at the cost of less encoding efficiency, use of minimally asymmetric digit sets may be justified.

Definition 1.10 (Periodic number representation): Whenever the digit sets of all digit positions of a radix-r number system represent the same set of integer values, the number representation is said to be *periodic*. In a periodic representation with period h, each digit position occupies h binary positions leading to practical choice of the radix $r = 2^h$. ■

Periodicity of a representation is an important characteristic, for it leads to regularity in VLSI design.

In Chapter 2, we study the representation paradigms of symmetric signed digit number systems [Jabe03], where we investigate the suitability of conventional signed digit number systems for the general purpose carry-free environment. The Chapter is basically a reproduction of our paper [Jabe03] on the subject, where we consider three representation paradigms for radix-r signed digit number systems, and compare them on the speed of addition operation. We introduce a modified more efficient version of the conventional carry-free addition algorithm, called the *Compare with Half Radix Algorithm* (CHRA), and show speed improvements. This investigation leads to the conclusion that among the three representation paradigms of signed digits, namely sign magnitude, one's complement, and two's complement, the latter when implemented by CHRA, leads to fastest carry-free addition among the paradigms studied.

In search of more suitable representations for redundant number systems, in terms of addition speed, we present the *stored transfer* representation of redundant numbers in Chapter 3, mainly as a reproduction of our paper on the subject [Jabe01]. This representation of a redundant digit consist of a main digit, practically a two's complement number and a transfer digit from a small digit set. When adding two stored transfer operands, the last step of Algorithm 1.1 (i.e., the conventional carry-free addition algorithm above), is not necessary, hence less addition delay. We prove that for the stored transfer version of the conventional carry-free addition algorithm to be applicable, the transfer digit set should be at least 3-valued. Unfortunately, this leads to at least two-bit representation of transfer digits (i.e., two redundancy bits per digit), thus degrading the encoding efficiency.

Generalization, often leads to new discoveries, as we generalize the stored transfer scheme to lead to *Weighted Bit-Set* (WBS) encoding of redundant number systems. In Chapter 4, we address the WBS encoding (mainly through a reproduction of our paper on the subject [Jabe02]) as a unifying framework for representing the GSD number systems of Parhami [Parh90], and all variants of the hybrid redundancy scheme of Phatak and Koren [Phat01].

To show the advantages of WBS interpretation of hybrid redundancy over previous implementations in [Phat01] we describe, in Chapter 5 (again based on our paper on the subject [Jabe05a]), the high level design details of a universal addition scheme for all hybrid redundant number systems. The only building block of this adder is standard full adder, accepting both carries and inversely encoded borrows, leading to extreme regularity in VLSI design. Further elaboration on WBS representation of hybrid redundancy, leads to a new class of hybrid redundant numbers not studied before, which in particular includes new variants of symmetric hybrid redundant number systems with arbitrary digit sets allowing use of the same circuitry for addition and subtraction with minimal penalty for the latter. We take up this extended hybrid redundancy scheme in Chapter 6.

Further generalization of WBS encoding with our novel concept of *Two-valued digit* (*Twit*), leads to *Weighted Twit-Set* (*WTS*) encoding of redundant number systems. This is presented in Chapter 7, mainly by reproduction of our paper on this subject [Jabe05a]. We revisit, by refinement of our theories, the WBS encoding, and present the WTS encoding as a unifying representation of all possible positional redundant number systems including those with noncontiguous digit sets and digit sets not including 0. In particular we present the stored twit-transfer scheme, as a WTS encoding, with improved encoding efficiency over the WBS stored transfer scheme. We use full adders with a novel functionality as of receiving three equally weighted twits, and generating sum and carry twits, simplifying the representationally closed carry-free addition of stored twit-transfer operands.

In Chapter 8, we discuss the suitability of the redundant number representations considered in Chapters 2 to 7 for a general purpose carry-free arithmetic environment, and evaluate them against the criteria outlined above, namely the encoding efficiency, representational closure, digit set preservation, and symmetry as defined in Definitions 1.5 to 1.8, respectively. This evaluation leads us to pick three number representations; one from the class of stored twit transfer encodings, one from ordinary hybrid redundant number systems, and a symmetric one shared by the class of stored transfer encodings and the extended hybrid redundant number systems.

Having selected three candidates for the most suitable number system among the options studied, we next present high level designs for representationally closed carry-free multiplication and division in Chapter 9. Other arithmetic support functions, such as arithmetic shift, zero, sign, and overflow detection, are discussed in Chapter 10. Finally we draw our conclusions in Chapter 11.

Chapter 2 High Radix Signed Digit Number Systems

Redundant signed digit number systems are popular in computationally intensive environments particularly because of their carry-free property which allows for digit-parallel addition. The time required for addition is particularly important because other arithmetic operations heavily depend on it. Signed digit number systems with high radices are of particular interest because of less memory requirement to represent a given number. But, the time required to perform digit-parallel addition is, by a relatively large coefficient, logarithmically proportional to the radix. In this chapter, we investigate the possibilities aiming in reduction of this coefficient, where we emphasize on lowest cost implementations. We present a novel modification to the conventional carry-free addition algorithm for signed digit numbers, and study the impact of different representations of signed digits on reducing the time required to perform digit-parallel addition. Three representation paradigms are considered, namely, signed-magnitude, two's complement and one's complement. Following the common practice, and in order to achieve better results, we focus on power-of-two radices. With the new algorithm, the time required to derive the transfer digit reduces to a small constant value which does not depend on the radix.

Addition is widely recognized as a basis of other arithmetic operations. Adequate redundancy in a number system provides for digit-parallel addition, i.e., digit-wise addition of two numbers with no *inter-digit* carry propagation. The Signed Digit (SD) number system was first introduced by Avizienis [Aviz61] where he proved the carry-free property for radix-r ($r \geq 3$) SD numbers with a digit set $[-\alpha, \alpha]$. In a number system with the carry-free property, a carry generated in any digit position is absorbed in the next position. In any hardware realization of carry-free addition based on binary adders, a generated carry, in fact, propagates up to the most significant bit of the next digit, (i.e., the carry is absorbed by that digit), so we can say that there is no inter-digit carry propagation beyond a one-time transfer to the next higher position. Adequate redundancy for the carry-free property is assured by the following constraint on digit values [Hwan79]:

$$\lceil (r+1) / 2 \rceil \leq \alpha \leq r-1.$$

For example, in the Binary Signed Digit (BSD) number system (r = 2) [Parh00], there is not enough redundancy in the digit set {-1, 0, 1}, to provide for carry-free property. But BSD has the limited-carry property [Parh90]. In a number system with limited-carry property, a carry generated in any digit position propagates through a limited number of consecutive digit positions. The BSD number system, nevertheless, has been extensively used for implementing all basic arithmetic operations [EL94, KNET87, SP92]. The reason is that addition of two BSD numbers is possible with carry propagation limited to two binary digits, hence the possibility of very fast digit-parallel addition. But each binary signed digit is represented by two bits (twice the 1 bit needed to represent an unsigned binary digit), thus in BSD, the extra memory required is maximum (100%) as compared to SD systems with higher radices.

The Hybrid Signed Digit (HSD) number system provides a framework for trade-off between speed and area (memory requirement) [Phat94]. An HSD number is basically a binary number, except that some positions may hold a "−1" value as well (a BSD position). A carry generated in any position (BSD or binary) may propagate up to the next more significant BSD position. In the periodic HSD number systems, the number of binary positions between consecutive BSD positions is constant. The major drawback of the HSD number system is the high asymmetry that exists between the range of positive and negative values. For this reason, the HSD representation is not considered as one of the paradigms in this chapter.

High Radix Signed Digit (HRSD) number systems have the benefit of lower memory requirement, while providing full symmetry between representable positive and negative values. But, the time required to add two high radix signed digits is by a relatively large coefficient proportional (or logarithmically proportional when a carry accelerating technique [Parh00] is used) to the number of bits in the representation of one digit, where the latter is logarithmically proportional to the radix. We call this coefficient the *high radix coefficient*, and explore the possibilities for reducing it. The relative largeness of the high radix coefficient is due to the complexity of the carry-free addition algorithm [Kore93], which takes several steps to perform the addition. BSD, HRSD, and the periodic HSD are all special cases of the Generalized Signed Digit (GSD) number system that is introduced in [Parh90].

In this chapter, we look for the lowest-cost (i.e., minimal hardware) representation for signed digits with the least possible value for the high radix coefficient. To accurately define what we mean here by a minimal hardware implementation, we define an h-dependent cell as a hardware piece whose delay depends on h (linearly or logarithmically), where each signed digit is assumed to be represented by h+1 bits. Relevant examples relate to addition, or addition-like operations such as comparison or zero detection, where all can be implemented by a (h+1)-bit (or h−bit in the case of sign-magnitude representation) adder cell. A minimal hardware implementation is one that uses the minimum number of h-dependent cells, where the same cell may be reused as needed. On the other extreme a maximal hardware implementation is one that uses any number of h-dependent cells in parallel, and reuses an h-dependent cell only when it does not increase the total delay. We will show that the value of the high radix coefficient is actually equal to the number of h-dependent cells in the critical path of the implementation. Any implementation may have some condition control circuitry with constant delay (that does not depend on h). We study three different representations for signed digits and introduce a novel modification to the Conventional Carry-Free Addition Algorithm (CCFAA) for HRSD numbers [Parh90]. In Section 2.1, we note that the CCFAA has four steps, where each step includes a form of addition of two digits (i.e., addition, comparison, zero detection, increment, or decrement). The time required to perform each addition is dependent on the internal hardware representation of the signed digits. To have a basis for cost comparison of the cases studied in this chapter, we try to parallel the steps of the CCFAA to the extent possible in Section 2.2. In Section 2.3, we introduce our modification to the CCFAA and prove its validity. Our novel *Compare with Half Radix Algorithm* (CHRA), introduces some simplifications in the implementation of the carry-free addition algorithm, which leads to the reduction of the high radix coefficient especially in a minimal hardware approach. In Section 2.4, we examine the sign-magnitude representation of signed-digits, where we show that the value of the high radix coefficient, on a minimal hardware approach is as high as 5. In Sections 2.5 and 2.6 we show that with two's complement and one's complement representations of a signed digit, the high radix coefficient can be substantially reduced, without increasing the hardware cost.

2.1. Conventional Carry-Free Addition Algorithm (CCFAA)

The HRSD number systems provide for carry-free addition. Table 2.I depicts the different stages in addition of two HRSD numbers, where r is the radix and α denotes the maximum absolute value for a digit from the digit set $[-\alpha, \alpha]$. The addition algorithm, has four steps (as listed below), where each step may contribute to the value of the high radix coefficient.

1. Parallel addition of digits in the same position of two n-digit HRSD numbers A and B, which results in the position sum vector P.
2. Comparison of the magnitude of position sum digits with α in order to derive the transfer vector T, where each transfer belongs to {-1, 0, 1}, t_0 is assumed to be zero, a nonzero t_n denotes an overflow, and the expression $|t_{i+1}| = (|p_i| \geq \alpha)$ means that if $|p_i| \geq \alpha$ then the absolute value of t_{i+1} is 1, otherwise it is 0.
3. Derivation of the interim sum vector W, by possibly adding r or $-r$ to the position sums.
4. Parallel addition of the interim sum vector W and the transfer T which generates the sum vector S. The transfer selection mechanism in step 2, guarantees that no new transfer is generated in this step.

Table 2.I The CCFAA

	a_{n-1}	…	a_1	a_0 +	$A = \Sigma_{i=0,\, n-1}\, a_i r^i$	
	b_{n-1}	…	b_1	b_0	$B = \Sigma_{i=0,\, n-1}\, b_i r^i$	
P:	p_{n-1}	…	p_1	p_0	$P_i = a_i + b_i$	(1)
T:	t_{n-1}	…	t_1	t_0	$\lvert t_{i+1}\rvert = (\lvert p_i\rvert \geq \alpha)$, sign (t_{i+1}) = sign (p_i)	(2)
W:	w_{n-1}	…	w_1	w_0	$w_i = p_i - r\, t_{i+1}$	(3)
S:	s_{n-1}	…	s_1	s_0	$s_i = w_i + t_i$	(4)

Figure 2.1 depicts the derivation of t_{i+1} and w_i, where t_{i+1} is the transfer to the $(i+1)^{\text{th}}$ position, w_i is the $(i+1)^{\text{th}}$ element of the vector W, the solid slopes serve as a graphical representation of Equation (3) in Table 2.I, and the interval tags I_1, to I_6 will be referred to later.

2.1.1 The choice of α and preservation of digit set $[-\alpha, \alpha]$

For a given radix r, the choice of α in $[\lceil (r+1) / 2 \rceil, r-1]$, provides for several signed digit number systems from the minimally redundant system with the carry-free property ($\alpha = \lceil (r+1) / 2 \rceil$), to the maximally redundant ($\alpha = r-1$) system. The following lemma shows that for the practical case of $r = 2^h$ ($h>1$), and also two other impractical cases, the choice of α has no impact on the memory requirement (i.e., the number of bits needed) for representing a signed digit.

Lemma 2.1: For $2^h-2 \le r \le 2^h$, the memory requirement for the digit set $[-\alpha, \alpha]$, doesn't depend on α.

Proof: The number of digits in $[-\alpha, \alpha]$, is $2\alpha+1$. Using the constraint on α (i.e., $\lceil (r+1)/2 \rceil \le \alpha \le r-1$), we can find the range of $2\alpha+1$, as $2\lceil (r+1)/2 \rceil + 1 \le 2\alpha+1 \le 2r-1$. Combining the latter with the inequalities for r, in the Lemma's statement, leads to:

$$2^h + 1 \le 2\alpha + 1 \le 2^{h+1}-1.$$

From the inequalities for $2\alpha+1$, it is obvious that regardless of the value of α, the number of bits needed to represent a signed digit is exactly $h+1$. ■

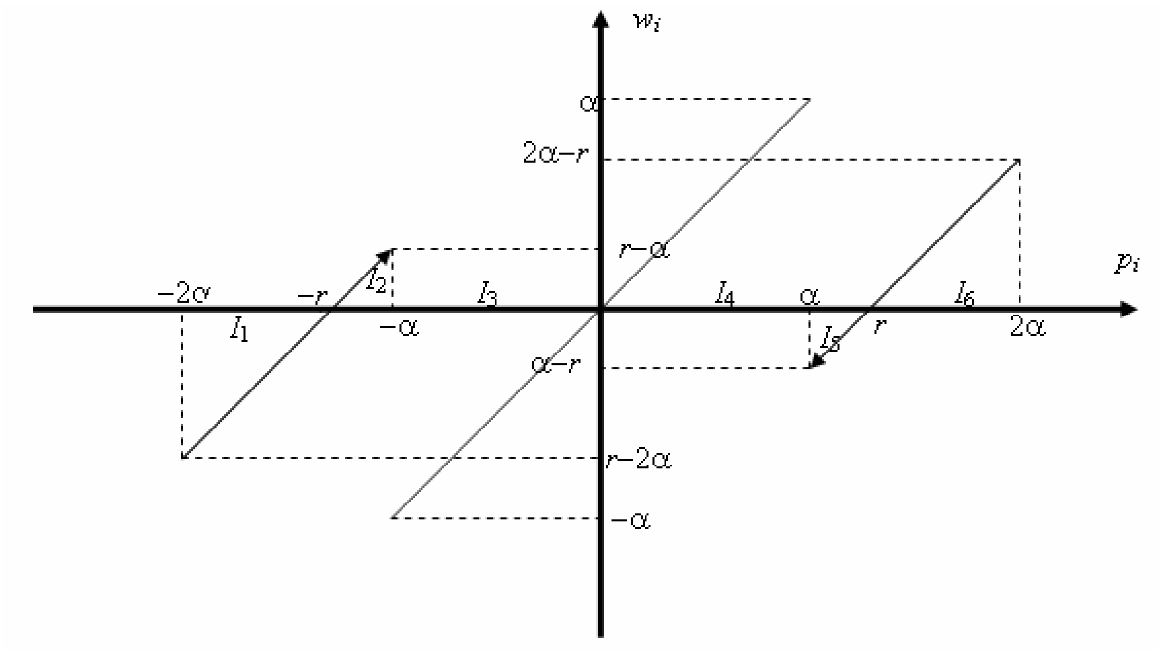


Fig. 2.1 Derivation of t_{i+1} and w_i

Next, we study the sources of preservation of the digit set $[-\alpha, \alpha]$ under the carry-free addition algorithm (i.e., the possibility that the range of s_i, is exactly equal to $[-\alpha, \alpha]$):

Lemma 2.2: Preserving the digit set $[-\alpha, \alpha]$ under carry-free addition is exclusively due to position sums p_i that satisfy $-\alpha < p_i < \alpha$, except for maximally redundant case ($\alpha = r-1$), where $|p_i| = 2\alpha$, also leads to $|s_i| = \alpha$.

Proof: For $-\alpha < p_i < \alpha$, we have $t_{i+1} = 0$, and thus $w_i = p_i$ and $-\alpha + 1 \le w_i \le \alpha - 1$. Therefore, the range of $s_i = w_i + t_i$ (where t_i belongs to $\{-1, 0, 1\}$), is $[-\alpha, \alpha]$. For $\alpha \le |p_i| \le 2\alpha$, by symmetry, we consider only $\alpha \le p_i \le 2\alpha$, where $t_{i+1} = 1$. We assume $\alpha = r - j$ for $1 \le j \le r - \lceil (r+1)/2 \rceil$, and show that the only value for j, leading to the preservation of the digit set is 1. Substitution of p_i by $w_i + r$ and α by $r - j$ in $\alpha \le p_i \le 2\alpha$ leads to $-j \le w_i \le r - 2j$. Now to let s_i reach α, we should let max $(w_i) = \alpha - 1$ or $r - 2j = r - j - 1$, i.e., $j = 1$. ■

2.1.2 Reduction of the high radix coefficient

Derivation of t_{i+1}, in equation (2) of the CCFAA, involves a comparison operation which generally have the same time complexity as that of an unsigned addition operation. Therefore, four digit-parallel addition-like operations are recognized in Table 2.I. The time required for each addition is dependent on h ($h = \lceil \log r \rceil$, where the number of bits in one digit is either h or $h + 1$ depending on the value of α), and so is the total addition time of two signed digits. Therefore, we can define the total addition time as a function of h, such as $\eta\delta(h) + c$, where η stands for the high radix coefficient, and c is a constant, which does not depend on h. $\delta(h)$ may be a linear function of h, where each digit-addition is implemented by a carry ripple technique or may be sub linear on h, where a carry accelerating technique, such as carry look-ahead, is used [Parh00].

To reduce the high radix coefficient, an obvious approach is to parallel the steps of the CCFAA to the extent possible, which considerably increases the hardware cost of the implementation. The first and second steps of the CCFAA (Table 2.I), cannot be paralleled, for obvious reason. But the rest of the computation can be done at the same time with step 2. The trick is to compute, in parallel, three groups of sum values depending on different values of t_i. In each group three values are computed in parallel depending on the three possible values of t_{i+1}. The groups for t_i in $\{-1, 0, 1\}$ are:

$$(p_i - 1, p_i - 1 + r, p_i - 1 - r), (p_i, p_i + r, p_i - r), (p_i + 1, p_i + 1 + r, p_i + 1 - r).$$

In each group, depending on the value of t_{i+1} three different values of the interim sum, are added to t_i. The position sum p_i is computed in step 1, and the other 8 values may be computed in parallel with step 2, by 8 extra adders. Next, one of the groups is selected by the value of t_i, and then the final sum is selected by the value of t_{i+1}. The selection process is done in constant time. Therefore in such a maximal hardware implementation, only steps 1 and 2 contribute to the value of the high radix coefficient. We will show in sections 2.4, 2.5, and 2.6, that contribution of steps 1 and 2 depend on the representation of the signed digits, specially, in sign-magnitude representation when implemented with minimal hardware, the contribution of step 1 is going to be more than 1. But by using considerable extra hardware, it is possible to limit the latter to 1.

To achieve the same effect of reducing the high radix coefficient, but with keeping the hardware cost as low as possible, we follow an algorithm optimization approach. In the next section we introduce our novel algorithm through which the derivation of the transfer in step 2 of the CCFAA can be done in constant time without using extra hardware. When we consider the impact of different representations of signed digits on the value of the high radix coefficient, we will show that the contribution of derivation of the interim sum in step 3 may also be reduced to zero, again without using any extra hardware. In the following sections we make the following assumptions for convenience and/or efficiency:

- $h = \lceil \log r \rceil$ and $r > 2$, where we assume that each signed digit is represented by (h+1) bits (zero padding or sign extension may be applied if necessary).

- $|p_i| = 2^h u_i + v_i x_i$, where u_i is the most significant bit of $|p_i|$, and $v_i x_i$ is the unsigned binary number composed of v_i, the second most significant bit of $|p_i|$ and x_i representing the (h-1) least significant bits of $|p_i|$ such that $0 \le x_i < 2^{h-1}$.

2.2 The Compare with Half Radix Algorithm (CHRA)

In the following theorem, we suggest that in step (2) of the CCFAA, $|p_i|$ may be compared with $\lceil r/2 \rceil$, instead of α. Then, for $r = 2^h$, the vector T may be derived with minimal delay after P is computed, such that the high radix coefficient is reduced by 1.

Theorem 2.1: In the carry-free addition algorithm, the transfer may be derived by comparing $|p_i|$, with $\lceil r/2 \rceil$ (instead of α), as:

$$\begin{aligned} t_{i+1} = {} & \text{if } (\lceil r/2 \rceil \le p_i \le 2\alpha) \text{ then } 1 \\ & \text{else if } (-2\alpha \le p_i \le -\lceil r/2 \rceil) \text{ then } -1 \\ & \text{else if } (-\lceil r/2 \rceil < p_i < \lceil r/2 \rceil) \text{ then } 0 \end{aligned}$$

Proof: It is sufficient to show that $|w_i| < \alpha$ for each of the above three intervals for p_i. Replacing p_i by $w_i + r\, t_{i+1}$ leads to:

$$-r + \lceil r/2 \rceil \le w_i \le 2\alpha - r, \text{ for } t_{i+1} = 1,$$
$$-2\alpha + r \le w_i \le -\lceil r/2 \rceil + r, \text{ for } t_{i+1} = -1, \text{ and}$$
$$-\lceil r/2 \rceil < w_i \le -\lceil r/2 \rceil, \text{ for } t_{i+1} = 0.$$

Enforcing the inequality $\lceil (r+1)/2 \rceil \le \alpha \le r-1$, in each of the above inequalities, leads to $|w_i| \le \alpha - 1$. ■

Note that for $p_i = \lceil r/2 \rceil$ and for even values of r, $t_{i+1} = 0$ is also valid. We will show later that this imprecision is indeed useful in the two's complement paradigm of representation of signed digits. The CHRA is particularly efficient in practice, where $r = 2^h$.

Corollary 2.1: For $r = 2^h$, the transfer is derived, with minimal delay, by comparing p_i with 2^{h-1}, i.e., sign (t_{i+1}) = sign (p_i) and $|t_{i+1}| = u_i \vee v_i$, where $\vee$ stands for logical or. ■

With the CHRA, contrary to Lemma 2.2, position sum values p_i, satisfying $-\alpha \le p_i \le \alpha$, do not contribute in preserving the digit set $[-\alpha, \alpha]$, except for the minimally redundant case $\alpha = \lceil (r+1)/2 \rceil$ with odd values of r, which is unfortunately not the case in Corollary 2.1. But, in the maximally redundant case ($\alpha = r - 1$), preservation of the digit set $[-\alpha, \alpha]$, always holds by Lemma 2.2 and the choice of $\alpha = r - 1$, where $2^{h-2} \le r \le 2^h$, does not introduce any inefficiency, as compared to less redundant cases (Lemma 2.1). The latter results are summarized in the following corollary.

Corollary 2.2: The compare with half radix algorithm preserves the digit set $[-\alpha, \alpha]$ in the maximally redundant signed digit number systems ($\alpha = r - 1$). Furthermore, for $2^{h-2} \le r \le 2^h$ and in particular for the practical case of $r = 2^h$, the choice of $\alpha = r - 1$ does not increase the memory requirement. ■

2.3 Sign-magnitude representation of HRSD numbers

Addition of two sign-magnitude digits, as described below, involves four steps by itself. All the four steps, in a maximal hardware approach may be paralleled such that the time required for a sign-magnitude addition is in the same order as the single step two's complement addition. In what follows, we consider the impact of the sign-magnitude representation of signed digits on different steps of the CCFAA, together with a time complexity analysis of a sign-magnitude addition.

2.3.1 Derivation of position sum

This step of the CCFAA, involves one sign-magnitude addition, whose contribution to the value of the high radix coefficient, by the following analysis, is 2(1) with the parenthesized figure relating to the maximal hardware approach. This is reflected in the first column and first row of Table 2.II.

2.3.1.1 Sign-magnitude addition

Addition of two sign-magnitude digits involves the following four steps where we assume that each digit is represented by a sign (1 bit) and an h-bit magnitude:

1) Possible complementation of the second operand:

   If the signs of the two operands are different, the magnitude of the second operand should be complemented before addition. Complementation involves an increment operation which may be deferred to be fused later in step 2 below, as an "always high" carry-in signal. As such, this step does not exclusively contribute in the total time needed for addition of two sign-magnitude digits, except for a sign-bit comparison and a conditional bit-wise inversion. That is, the contribution does not depend on h.

2) Addition of the magnitudes of the two operands.

   The contribution of this step to the total addition time depends on h.

3) Possible magnitude comparison of the two operands:

   If the two operands have different signs, then the sign of the result is the same as that of the operand with larger magnitude. In a minimal hardware approach, we may take advantage of the fact that magnitude comparison is necessary only when the signs are not alike, where the actual operation in Step 2 above is subtraction of magnitudes. For a non-zero result, the operand with larger magnitude can be determined from the subtraction result. For a zero result, the derived sign as such may be positive or negative, but unique zero representation requires a positive sign for zero magnitudes. We therefore need to determine if the subtraction result was zero or not. The time required for zero detection of an h-bit operand depends on h. The latter could be done in parallel with Step 2 [Vass89], but staying with our minimal hardware approach, we can reuse the adder cell of Step 2 for zero detection. The trick is to add 2^{h-1} to the subtraction result and check for the carry-out signal. A low signal indicates that the subtraction result was zero. We can conclude now that in a minimal hardware approach, the exclusive elapsed time of this step depends on h.

4) Possible complementation of the result:

   If the sign of complemented operand in step 1 was originally positive, the result of the addition in Step 2 should be complemented. The contribution of this step in the total addition time normally depends on h. But the post two's complement operation has been reported to be avoidable in [Vass89], without employing any extra h-dependent cell. The trick is to bit-wise complement the result, when is necessary, and instead of increment operation, as part of complementation, add to it the carry out of the magnitude addition.

The latter addition as a sort of end-around-carry addition does not actually introduce another h-dependent operation besides the magnitude addition. Therefore taking advantage of the latter clever technique, the time required for this step is not h-dependent, even in a minimal hardware approach.

Summing up the partial contributions of the steps above in the total sign-magnitude addition time, we conclude that in a minimal hardware approach, two h-dependent addition operations (due to those of Steps 2 and 3 above), contribute in the total addition time, while the h-dependent delay in a maximal hardware approach equals to that of only 1 addition.

2.3.2 Derivation of transfer and interim sum

Recalling Equation (2) of Table 2.I, we note that derivation of the transfer involves a magnitude comparison operation. The comparison operation has the same time complexity as that of a simple unsigned addition, and thus its contribution in the value of the high radix coefficient as reflected in the first column and second row of Table 2.V is 1.

To analyze the time complexity of the derivation of the interim sum by Equation (3) of Table 2.I, we recognize six cases depending on the six intervals of the values of p_i, denoted by I_1 to I_6, in Figure 2.1. In each case, as is shown in Table 2.II, we can derive w_i, by replacing $2^h u_i + v_i x_i$ for $|p_i|$ and 2^h for r, in $w_i = p_i - rt_{i+1}$, followed by substitution of the related values (with regards to the respected intervals) for u_i and t_{i+1}. The choice of $r = 2^h$, follows the common practice, and simplifies the derivation.

Table 2.II Derivation summary of w_i in addition of two sign-magnitude signed digits

Interval for p_i	p_i	Sign (p_i)	u_i	$\|t_{i+1}\|$	t_{i+1}	w_i	Sign (w_i)	$\|w_i\|$
$I_1 = [-2\alpha, -2^h]$	$-2^h u_i - v_i x_i$	1	1	1	-1	$-v_i x_i$	1	$v_i x_i$
$I_2 = [-2^h+1, -\alpha]$	$-2^h u_i - v_i x_i$	1	0	1	-1	$-v_i x_i + 2^h$	0	$!(v_i x_i) + 1$
$I_3 = [-\alpha+1, -1]$	$-2^h u_i - v_i x_i$	1	0	0	0	$-v_i x_i$	1	$v_i x_i$
$I_4 = [0, \alpha - 1]$	$2^h u_i + v_i x_i$	0	0	0	0	$v_i x_i$	0	$v_i x_i$
$I_5 = [\alpha, 2^h - 1]$	$2^h u_i + v_i x_i$	0	0	1	1	$v_i x_i - 2^h$	1	$!(v_i x_i) + 1$
$I_6 = [2^h, 2\alpha]$	$2^h u_i + v_i x_i$	0	1	1	1	$v_i x_i$	0	$v_i x_i$

In Table 2.II, we note that w_i is negative only when the number of "1"s in the three columns for sign(p_i), u_i, and $|t_{i+1}|$ is odd, i.e., sign (w_i) = sign (p_i) $\oplus$ $u_i \oplus$ $|t_{i+1}|$. To find an easy implementation for $|w_i|$, we note in Table 2.II that $|w_i| = v_i x_i$, except when $!u_i$ and $|t_{i+1}|$ are both "1" in which case $|w_i| = !(v_i x_i) + 1$, where $!(v_i x_i)$ is the bit-wise complement of $v_i x_i$. This observation can be summarized in the equation $|w_i|$ = multiplex ($v_i x_i$, $!u_i|t_{i+1}|$, $!(v_i x_i) + 1$), where multiplex (x, c, y) resolves to x when the bit-variable c is "0", and to y otherwise. The increment operation involved in the derivation of $|w_i|$ may be fused in step (4) of the CCFAA. Therefore, this step may be considered as not contributing in the value of the high radix coefficient, even in a minimal hardware approach.

Finally step (4) of the CCFAA as a sign-magnitude addition contributes another "2" (1 in the maximal hardware approach) to the value of the high radix coefficient, making η, as reflected in Table 2.V, equal to 5(2). Applying the CHRA reduces η, to 4(1).

2.4 Two's complement representation of high radix signed digits

Here, we consider representing each signed digit, as a two's complement number. The range $[-2^h, 2^{h-1}]$, of a $(h+1)$-bit two's complement digit, covers the digit set $[-\alpha, \alpha]$, for $\lceil (r+1)/2 \rceil \le \alpha \le r-1$ and $r = 2^h$.

2.4.1 Derivation of two's complement position sum

To derive the position sum, we sign-extend (one bit to the left) the two $(h+1)$-bit signed digits represented in two's complement format, and then perform two's complement addition. The result will be an $(h+2)$-bit position sum. The contribution of this operation in the value of the high radix coefficient, as is reflected in the third and fourth column and first row of Table 2.V is 1.

2.4.2 Derivation of transfer and two's complement interim sum

The outcome of applying the CHRA on two's complement signed digits (with $r = 2^h$) is shown in Figure 2.2 and also in Table 2.III.

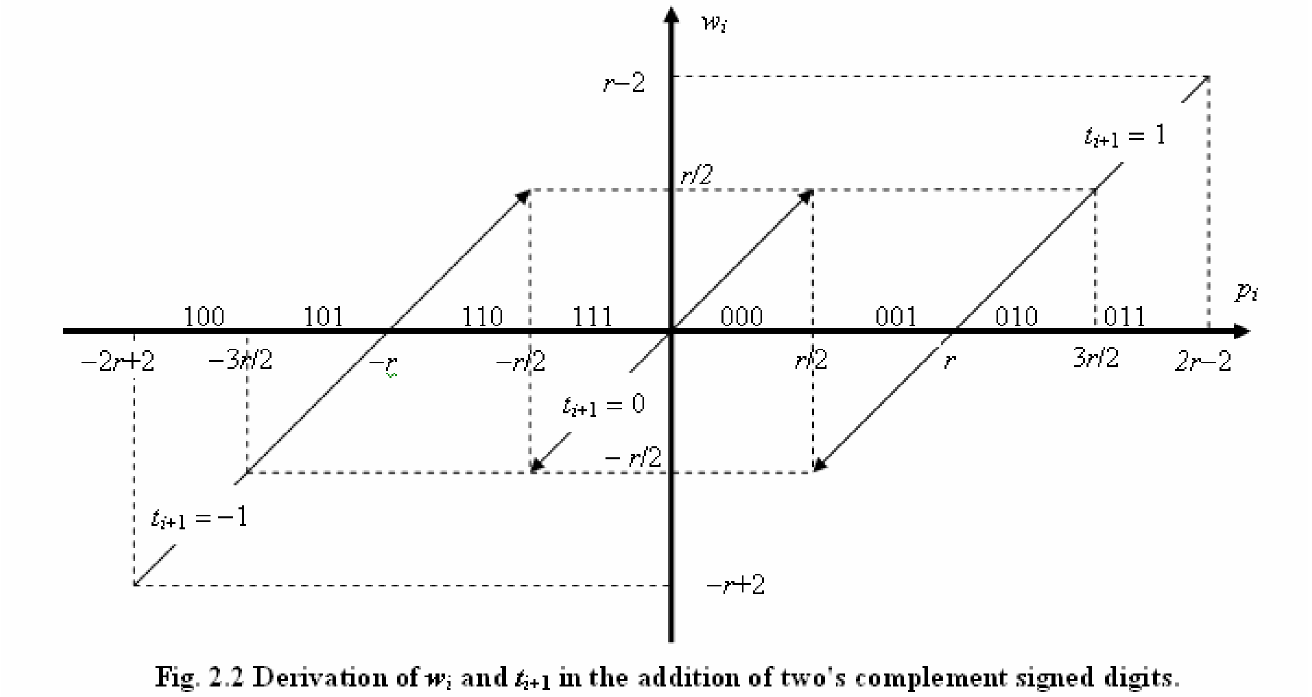


Fig. 2.2 Derivation of w_i and t_{i+1} in the addition of two's complement signed digits.

The Figure is drawn for the maximally redundant case $\alpha = r - 1$, in which the 3 bit numbers on the intervals for p_i (i.e., sign(p_i), u_i, and v_i), stand for the three most significant bits of p_i. In the Table, the columns 2-4 and 7-8 represent the three most significant bits of p_i and the two most significant bits of w_i respectively, x_i stands for the $(h - 1)$ least significant bits of p_i and the two's complement representation of t_{i+1} is shown in the rightmost two columns, where the superscripts denote the bit positions. Note that, by Theorem 2.1, the choice of $t_{i+1} = 0$ in the last row of Table 2.II includes the point with coordinates $(-r/2, -r/2)$ of Figure 2.2. As shown below, the latter choice is vital for simplification of the derivation of t_{i+1}. From Table 2.II, it can be easily verified that the transfer t_{i+1}, can be computed by a simple 3-input/2-output logic, as in the following logical equations:

$$t^1_{i+1} = \text{sign}(p_i)\ !(u_i v_i), \quad t^0_{i+1} = (!\text{sign}(p_i) + !u_i + !v_i)\ (\text{sign}(p_i) + u_i + v_i).$$

Table 2.III Derivation of w_i and t_{i+1} in the addition of two's complement signed digits.

p_i	Sign (p_i)	u_i	v_i	t_{i+1}	w_i	w_i^h	w_i^{h-1}	t^1_{i+1}	t^0_{i+1}
x_i	0	0	0	0	x_i	0	0	0	0
$2^{h-1} + x_i$	0	0	1	1	$-2^h + 2^{h-1} + x_i$	1	1	0	1
$2^h + x_i$	0	1	0	1	x_i	0	0	0	1
$2^h + 2^{h-1} + x_i$	0	1	1	1	$2^{h-1} + x_i$	0	1	0	1
$-2^{h+1} + x_i$	1	0	0	-1	$-2^h + x_i$	1	0	1	1
$-2^{h+1} + 2^{h-1} + x_i$	1	0	1	-1	$-2^h + 2^{h-1} + x_i$	1	1	1	1
$-2^{h+1} + 2^h + x_i$	1	1	0	-1	x_i	0	0	1	1
$-2^{h+1} + 2^h + 2^{h-1} + x_i$	1	1	1	0	$-2^h + 2^{h-1} + x_i$	1	1	0	0

The $(h-1)$ least significant bits of w_i, exactly, represent x_i (i.e., $(h-1)$ least significant bits of p_i), and also $w_i^{h-1} = p_i^{h-1}$, as can be easily seen in Table 2.II. What remains is w_i^h, which is computable by a simple 3-input logic, implementing the following equation:

$$w_i^h = \text{sign}(p_i)\ !u_i + \text{sign}(p_i)\ v_i + !u_i v_i.$$

From the above equations, we can see that derivation of the transfer and the interim sum do not contribute to the value of the high radix coefficient, as reflected in the fourth column and second and third row of Table 2.V. Finally, s_i can be derived by a simple two's complement increment/decrement logic, whose share in the value of the high radix coefficient is 1. The high radix coefficient for two's complement paradigm with the CCFAA and the CHRA is thus $\eta = 3(2)$ and $\eta = 2(1)$ respectively, where the figures in parenthesis refer to the maximal hardware approach.

2.5 One's complement representation of signed digits

A signed digit can be represented in one's complement format, pretty much the same as that shown in the previous section for two's complement signed digits. Following the same analysis as in the previous section, derivation of the position sum contributes a "1" to the value of the high radix coefficient. Then, Table 2.IV resembling the derivation of w_i and t_{i+1}, has been built up similar to Table 2.III, where there are two main differences between the two tables. First, one's complement encoding is used for t_{i+1} in the last two columns, and thus derivation of t^0_{i+1} is simpler as $t^0_{i+1} = !\text{sign}(p_i)\ (u_i + v_i)$. Second, the derivation of w_i, as seen in the second row and the row before last of Table 2.IV, requires an increment/decrement operation. But since t_i is available before it is possible to do the increment/decrement operation on w_i, the increment/decrement may be fused in the computation of $s_i = w_i + t_i$. Therefore, the high radix coefficient in this case is also $\eta = 3(2)$ and $\eta = 2(1)$ respectively. The value of the high radix coefficient in one's complement and two's complement paradigms is the same, but the two's complement representation of signed digits is naturally preferable. The reason is the popularity of the two's complement representation in general, availability of optimized standard adder cells for two's complement binary representation, and the ease of converting widely used two's complement numbers to their signed digit equivalent and vice versa.

Table 2.IV Derivation of w_i and t_{i+1} in addition of one's complement signed digits.

p_i	Sign (p_i)	u_i	v_i	t_{i+1}	w_i	w_i^h	w_i^{h-1}	t^1_{i+1}	t^0_{i+1}
x_i	0	0	0	0	x_i	0	0	0	0
$2^{h-1} + x_i$	0	0	1	1	$-2^h + 1 + 2^{h-1} + x_i$	1	1	0	1
$2^h + x_i$	0	1	0	1	x_i	0	0	0	1
$2^h + 2^{h-1} + x_i$	0	1	1	1	$2^{h-1} + x_i$	0	1	0	1
$-2^{h+1} + 1 + x_i$	1	0	0	-1	$-2^h + 1 + x_i$	1	0	1	0
$-2^{h+1} + 1 + 2^{h-1} + x_i$	1	0	1	-1	$-2^h + 1 + 2^{h-1} + x_i$	1	1	1	0
$-2^{h+1} + 1 + 2^h + x_i$	1	1	0	-1	$1 + x_i$	0	0	1	0
$-2^{h+1} + 1 + 2^h + 2^{h-1} + x_i$	1	1	1	0	$-2^h + 1 + 2^{h-1} + x_i$	1	1	0	0

Table 2.V Contribution of each step of carry-free addition in the value ofthe high radix coefficient η, where the parenthesized figures relate to the maximal hardware approach

	Sign-Magnitude		Two's Complement		One's Complement	
	CCFAA	CHRA	CCFAA	CHRA	CCFAA	CHRA
Position sum *P*	2 (1)	2 (1)	1 (1)	1 (1)	1 (1)	1 (1)
Transfer *T*	1 (1)	0 (0)	1 (1)	0 (0)	1 (1)	0 (0)
Interim sum *W*	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
Final sum *S*	2 (0)	2 (0)	1 (0)	1 (0)	1 (0)	1 (0)
High radix coefficient η	5 (2)	4 (2)	3 (2)	2 (1)	3 (2)	2 (1)

2.6 Summary

High radix signed digit number systems exhibit the carry-free property while economizing the memory requirement as compared to lower radix signed digit number systems. In this chapter, we introduced the high radix coefficient as a measure for comparing the time required to perform carry-free addition of HRSD numbers with different representations, where we emphasize on lowest-cost implementations, which is characterized by limiting the number of h-dependent cells to 1. An h-dependent ($h = \lceil \log r \rceil$, and r is the radix of the number system) cell is a (h+1)-bit (or h-bit) adder, comparator, or zero detector. We present a modification to the conventional carry-free addition algorithm for HRSD numbers, in order to reduce the high radix coefficient. One of the steps in carry-free addition involves comparing the magnitude of the position sum with the maximum absolute value α of the digit set. We present a theorem to prove that the comparison of the magnitude of the position sum with the half-radix $\lceil r/2 \rceil$, instead of α, will produce a valid transfer digit. We show that our modified algorithm, when applied for power-of-two radices ($r = 2^h$, $h > 1$), simplifies the comparison operation to a constant time derivation of a simple logical equation. We apply the modified algorithm to sign-magnitude, two's complement and one's complement representations of signed digits, and designate the proposed method the Compare with Half-Radix Algorithm (CHRA). We show that use of the CHRA, with two's complement or one's complement representation of signed digits in a minimal hardware (lowest-cost) approach has the same effect on reducing the high radix coefficient, as does the maximal hardware (most costly) implementation of the CCFAA or CHRA with sign-magnitude representation.

We present a comparison table (Table 2.V) of the application of the CHRA with that of the CCFAA on the three signed digit's representation paradigms studied in this chapter, for both minimal hardware and maximal hardware approaches. The Table shows that the two's complement and one's complement representations with the CHRA and the minimal hardware approach lead to a 60% lower value for the high radix coefficient (reducing from 5 to 2) over the sign-magnitude paradigm with the conventional carry-free addition algorithm. This is achieved for power-of-two radices ($r = 2^h$, $h > 1$) and the maximally redundant ($\alpha = r-1$) signed digit numbers (with the same memory requirement as any less redundant case), while the digit set $[-\alpha, \alpha]$ is fully preserved. The two's complement paradigm is preferred over one's complement because of the popularity of the two's complement representation in general. Some other even more efficient representation paradigms of signed digits are studied in the next chapters.

Chapter 3 | Stored Transfer Representation

Redundancy in number representation aims to improve the speed or efficiency of arithmetic units [Metz59], [Aviz60] and is commonly used in modern digital systems. One reason for speed improvement with redundancy is the possibility of carry-free addition; i.e., addition in a small, constant time, independent of operand widths [Parh90]. Another reason is that redundancy allows some imprecision in the decision processes (such as quotient or root digit selection [Parh00], [Parh01]); this tolerance for imprecision removes enough complexity from the computation's critical path to yield significant performance improvement. Here, we focus on mechanisms that facilitate carry-free addition and allow its implementation with even greater speed.

In carry-free addition, as illustrated in Figure 3.1.a, one performs the following steps on all k digit positions of the two radix-r operands in parallel, where x_i and y_i belong to the possibly redundant digit set $\Delta = [\alpha , \beta]$:

1. Compute the position sum digit $p_i = x_i + y_i$
2. Derive the interim sum digit w_i and transfer digit t_{i+1} satisfying $w_i = p_i - rt_{i+1}$
3. Form the final sum digit $s_i = w_i + t_i$

For step 3 to yield a valid digit in Δ without producing further transfers, w_i must be restricted in $D = [a , b]$, with the following holding for all possible values of t_i:

$$\alpha - t_i \le a < a + r - 1 \le b \le \beta - t_i$$

Note that the digit-size additions of steps 1 and 3, though quite fast compared to word-size additions required with nonredundant representations, are merely used for algorithm description and need not be explicitly performed in hardware. The addition in step 1 can be avoided, e.g., by noting that w_i and t_{i+1} are directly computable in hardware as functions of x_i and y_i. That is:

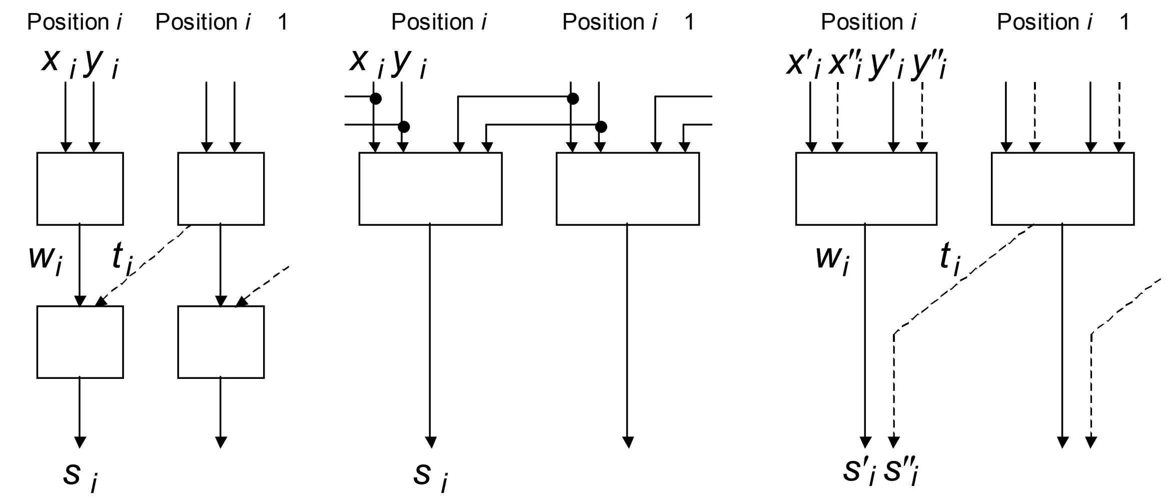
$$w_i = \omega(x_i, y_i) \qquad\qquad t_{i+1} = \tau(x_i, y_i)$$

This, in effect, as also depicted in Figure 3.1.b, fuses steps 1 and 2 and allows the designer to choose the best possible merged implementation. It may be the case, with certain digit sets and/or encodings, that some form of addition is still part of the best hardware implementation scheme for ω and τ, but this is not required. We are thus motivated to investigate methods for eliminating, or else simplifying, the addition in step 3.

3.1 The Notion of Stored-Transfer

In a manner similar to the stored-carry or carry-save representation of binary numbers [Metz59], [Jabe99], we study the implications of stored-transfer or transfer-save representations of redundant digits where the pair (w_i, t_i) is viewed as an encoding of the sum digit s_i, thus obviating the need for the final addition as depicted in Figure 3.1.c. We call w_i the *main part* and t_i the *transfer part* of a digit's stored-transfer encoding.

Example 3.1: A main part that is a 4-bit 2's-complement number and a 4-valued stored transfer in [−1, 2] constitute a 6-bit encoding of the digit set [−9, 9]. Direct encoding of the digit set requires 5 bits. ■



(a) Carry-free add with a final stage for transfer integration

(b) Carry-free add with lookahead in lieu of transfers

(c) Carry-free add with stored-transfer encoding of digits

Fig. 3.1 Carry-free addition paradigms

The latter scheme leads to a two-step formulation of carry-free addition. In the following, we assume that any digit $z \in \Delta$ has a transfer-save encoding (z', z''), with $z' \in D$ and $z'' \in G = \{c_0, c_1, \ldots, c_{d-1}\}$; that is, primed and double-primed variables are used to designate the main and transfer parts of a digit.

1. Compute the position sum digit $p_i = x_i' + x_i'' + y_i' + y_i''$
2. Derive $s_i = (s_i', s_i'')$, satisfying $s_i' = p_i - rs_{i+1}''$

Note that the generated transfer set $G = \{c_0, c_1, \ldots, c_{d-1}\}$, satisfying $c_0 < c_1 < \ldots < c_{d-1}$, is d-valued but does not necessarily contain a set of d consecutive integers. We take this more general view in anticipation that it may provide added flexibility for optimizations. We will see later that even though such generalized transfer sets do not provide additional benefits directly, they can be used with minor modifications to the carry-free addition algorithm. On the other hand, the main part of a digit belongs to an interval $D = [a, b]$ of values. Whereas gaps in this set are also admissible, provided that the values in the set contain one member from each of the r residue equivalence classes $j \bmod r$ $(0 \le j \le r - 1)$, we have not found this generality to lead to any speed or cost benefit. Of course, steps 1 and 2 in this new twe step process can again be fused, in the manner previously outlined, leading to a merged, or single-step, implementation: $s_i' = \sigma'(x_i', x_i'', y_i', y_i'')$, $s_{i+1}'' = \sigma''(x_i', x_i'', y_i', y_i'')$.

An objection may be raised that our scheme simply shifts the complexity of the original step 3 to the new step 1. That this is not the case will become clear as we describe our methods. Here, we just argue that the new scheme can, in principle, be faster than the original algorithm. For a 4-operand addition, where two of the operands (transfer parts) are fairly small, can indeed be faster and less complex than two separate additions [Koba85]. In such a case, the function pairs (ω, τ) and (σ', σ'') have comparable bit-level complexities.

3.2 Some General Requirements

Equating the boundaries of the original digit set $\Delta = [\alpha, \beta]$ and its stored-transfer representation, i.e., $[\alpha, \beta] = [a, b] + \{c_0, c_1, \ldots, c_{d-1}\}$, leads to the requirements:

$$\alpha = a + c_0 \qquad\qquad \beta = b + c_{d-1}$$

For convenience, we define redundancy indices associated with the two digit sets $[\alpha, \beta]$ and $[a, b]$ as $\rho = \beta - \alpha + 1 - r$ and $\rho' = b - a + 1 - r$, respectively. We also designate $\delta = c_{d-1} - c_0$ as the span of the transfer set. It is easy to show that $\rho = \rho' + \delta$. If, for the sake of representational efficiency, we set $\rho' = 0$, it is the case that $\rho = \delta$. Furthermore, we define $\Delta_i = [a + rc_i, b + rc_i] \cap [p_{\min}, p_{\max}]$ as the range of p, where c_i is a valid (or *useful*, per Definition 3.1 below) transfer value and $\lambda_i = b + rc_i - a - rc_{i+1} + 1 = \rho' + r - r\delta_i$ $(i < d - 1)$ as the overlap between Δ_i and Δ_{i+1}, where $\delta_i = c_{i+1} - c_i$.

Example 3.2: Stored-transfer representations of some redundant number systems are characterized in Table 3.I. In all cases, D is irredundant ($\rho' = 0$, $\rho = \delta$) and is taken to be the unsigned set $[0, r - 1]$, except for the last entry where D is $[-r/2, r/2 - 1]$ with r even. For the two hybrid signed-digit entries, $r = 2^h$. ■

We next explore constraints on the digit set and transfer values dictated by the requirements for carry-free addition, where we make use of the following definitions:

Table 3.I Stored transfer representation of familiar redundant number systems

Name of number system	Δ	G	D	$\rho = \delta$
Stored-carry	$[0, r]$	$\{0, 1\}$	$[0, r-1]$	1
Stored-borrow	$[-1, r-1]$	$\{-1, 0\}$	$[0, r-1]$	1
Stored-carry-or-borrow	$[-1, r]$	$\{-1, 0, 1\}$	$[0, r-1]$	2
	$[-1, r]$	$\{-1, 1\}$	$[0, r-1]$	2
Stored-double-carry	$[0, r+1]$	$\{0, 1, 2\}$	$[0, r-1]$	2
Hybrid S-D (h–1 B, 1 BSD, $r = 2^h$)	$[-1, 2^h-1]$	$\{-1, 0\}$	$[0, r-1]$	1
Hybrid S-D (1 BSD, h–1 B, $r = 2^h$)	$[-2^{h-1}, 2^h-1]$	$\{-2^{h-1}, 0\}$	$[0, r-1]$	2^{h-1}
Minimally redundant Asymmetric	$[-r/2-1, r/2]$	$\{-1, 0, 1\}$	$[-r/2, r/2-1]$	2
	$[-r/2-1, r/2]$	$\{-1, 1\}$	$[-r/2, r/2-1]$	2

Definition 3.1: A transfer value $c_i \in G$ is *useful* if the set Δ_i is nonempty; i.e., there exists some position sum value p that may be decomposed as $p = w + rc_i$, where $w \in [a, b]$. ■

Definition 3.2: A transfer value $c_i \in G$ is *necessary* if the set $\Delta_i - (\Delta_i \cap \Delta_{i+1}) - (\Delta_{i-1} \cap \Delta_i)$, where c_i constitutes the only valid choice of transfer digit value, is nonempty. ■

Definition 3.3: The *necessity range* of p for c_i, $0 < i < d - 1$, is the possibly empty interval $[b + rc_{i-1} + 1, a + rc_{i+1} - 1]$ where c_i is *necessary*, and neither c_{i-1} nor c_{i+1} is useful. ■

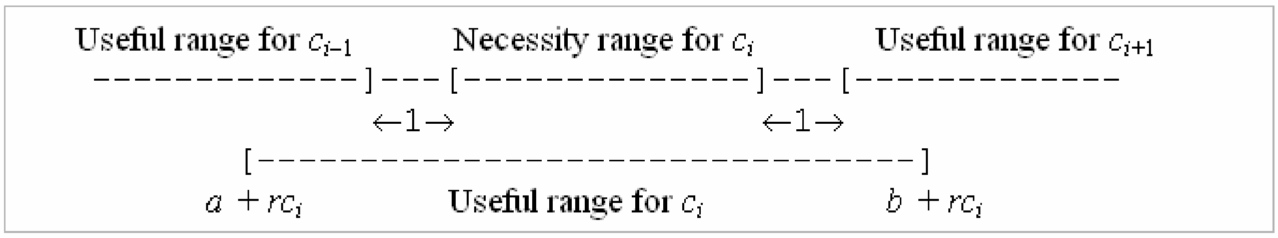

Useful range for c_{i-1}
Necessity range for c_i
Useful range for c_{i+1}
←1→
←1→
$a + rc_i$
Useful range for c_i
$b + rc_i$

Fig. 3.2 Illustrating Definitions 3.1-3.3.

For a representation system with the representational closure property under carry-free addition (see Figure 3.3 for an illustration), the range $[2a + 2c_0, 2b + 2c_{d-1}]$ of the position sum p should be totally contained within $[a, b] + \{rc_0, rc_1, \ldots, rc_{d-1}\}$. For digit set preservation property to hold, This leads to the following results.

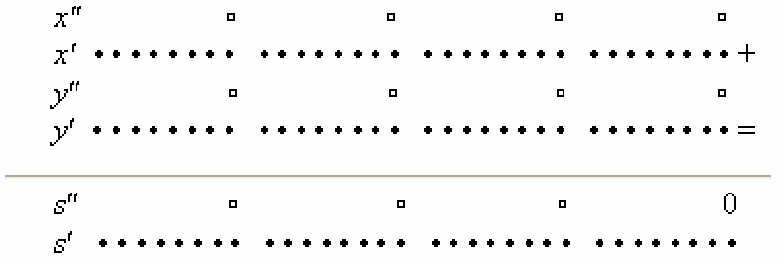


Fig. 3.3 Representationally closed carry-free addition of stored transfer operands.

Lemma 3.1: If $m = \max_i \delta_i$ is the maximum spacing of values in G, we must have $\rho' \geq (m - 1)r$ for carry-free addition to be possible with stored transfer representation.

Proof: Consider consecutive transfer values j and $j + m$ in G. The ranges of p for which these transfer values can be chosen are $\Delta_j = [a + jr, b + jr]$ and $\Delta_{j+m} = [a + (j + m)r, b + (j + m)r]$, respectively. To avoid gaps in the $p$values, Δ_j and Δ_{j+m} must overlap:

$$b + jr + 1 \geq a + (j + m)r$$

This is easily converted to $\rho' = b - a + 1 - r \geq (m - 1)r$. ■

Corollary 3.1: Given a value for ρ', the maximum allowed spacing of values in G is $\rho'/r + 1$ (i.e., $\delta_i \leq \rho'/r + 1$). ■

Corollary 3.2: Given a value for ρ', the overlap between Δ_i and Δ_{i+1} (cardinality of $\Delta_i \cap \Delta_{i+1}$) is $\lambda_i = \rho' + r - r\delta_i$, i.e., the overlap λ_i is minimized, for $\delta_i = \lfloor \rho'/r \rfloor + 1$. ■

Corollary 3.3: For $\rho' \le r - 1$, the transfer set G must contain an interval of integer values (i.e., $\delta_i = 1$ for all i). ■

Theorem 3.1: The transfer set G must be at least three-valued. Furthermore, a transfer set with four values is generally adequate, except for a few special cases.

Proof: The minimum and maximum transfer values, i.e., c_0 and c_{d-1}, should satisfy the following inequalities:

$$a + rc_0 \le 2a + 2c_0 \quad \Rightarrow \quad c_0 \le a/(r-2)$$
$$2b + 2c_{d-1} \le b + rc_{d-1} \quad \Rightarrow \quad b/(r-2) \le c_{d-1}$$

To minimize d, we aim to maximize the *necessity range* for each $c_i \in G$. We thus choose $c_0 = \lfloor a/(r-2) \rfloor$ and minimize λ_i by choosing $\delta_i = \lfloor \rho'/r \rfloor + 1$, for all i, as prescribed by Corollary 3.2. The value of c_{d-1} can then be derived as:

$$c_{d-1} = c_0 + \sum\delta_i = \lfloor a/(r-2) \rfloor + (d-1)(\lfloor \rho'/r \rfloor + 1)$$

This equation, along with the lower bound for c_{d-1}, yields:

$$\lfloor a/(r-2) \rfloor + (d-1)(\lfloor \rho'/r \rfloor + 1) \ge (a+\rho'+r-1)/(r-2)$$

Letting $a = (r-2)u + v$ and $\rho' = rq + y$, with $0 \le v \le r-3$ and $0 \le y \le r-1$, the condition above becomes:

$$u + (d-1)(q+1) \ge u + q + 1 + (v+y+2q+1)/(r-2)$$

Solving this inequality for d, we get:

$$d \ge 2 + \theta \quad \Rightarrow \quad d_{\min} = 2 + \lceil \theta \rceil$$

where $\theta = (v + y + 2q + 1) / [(r-2)(q+1)]$. Considering that $r \ge 3$, we next show that $\theta > 3$ ($d_{\min} > 5$) is impossible, and $\theta = 3$ ($d_{min} = 5$) is needed only for a few special cases. To show that $d_{min} > 5$ never holds, we note that

$$\theta = (v + y + 2q + 1) / ((r-2)(q+1)) > 3$$

implies $(3r-8)q < v + y - 3r + 7 \le 3 - r \le 0$, which is impossible given that $(3r-8)q < 0$ holds only if $q < 0$, whereas $q = \lfloor \rho'/r \rfloor \ge 0$. Similarly, setting $\theta > 2$ leads to:

$$(2r-6)q < v + y - 2r + 5$$

Given that the right-hand side of the inequality above is no greater than 1, we must have $q = 0$ for $r > 3$. This leads to the following special cases for which $d_{\min} = 5$:

$$r > 3,\ q = 0,\ v = r-3, \text{ and } y = r-1 \text{ or}$$
$$r = 3,\ v = 0, \text{ and } y = 2$$

For all other cases (i.e., $0 < \theta \le 2$), we have $3 \le d_{\min} \le 4$. ■

The undesirable cases in Theorem 3.1, where $\theta = 3$, are unlikely to be of practical interest. The radix-3 case (besides not being a power of 2) implies at least five values each for D and G, leading to 6 or more bits per digit. For radix 2^h, $h \ge 2$, the high redundancy implied by $\rho' \ge r-1$, coupled with 3 bits for the 5-valued stored transfer, can be easily avoided by suitable choice of a that ensures $v < r-3$ (e.g., $0 \le a \le r-4$ or $-r+2 \le a \le -2$).

Corollary 3.4: For $\rho' = 0$, we have $d_{\min} = 3$. In this case, $G_{\min} = \{c_0, c_0 + 1, c_0 + 2\}$ is adequate, where $c_0 = \lfloor a/(r-2) \rfloor$. ■

Corollary 3.5: For $0 < \rho' \le r-1$, we have $\theta \le 2$ and $d_{\min} \le 4$, except when $v = r-3$ and $y = r-1$, in which case $d_{\min} = 5$. ■

Corollary 3.6: For carry-free addition to be possible with digit set Δ, the condition $\rho \ge \delta \ge 2$ is necessary. ■

This last result is consistent with the fact that all the cases with $\rho = \delta = 1$ (e.g., some of those in Example 3.2) do not support carry-free addition [Parh90].

Lemma 3.2: The *necessity range* of p for $c_i \in G - \{c_0, c_{d-1}\}$ is nonempty iff $\delta_i + \delta_{i+1} > \rho'/r + 1$.

Proof: The requirement $b+1+rc_{i-1} \le a-1+rc_{i+1}$, with $c_{i+1}-c_{i-1} = \delta_i+\delta_{i+1}$ and $b - a + 1 = \rho' + r$ lead to the desired result. ■

Corollary 3.7: For $\rho' \le r-1$ ($\delta_i+\delta_{i+1}=2 > \rho'/r+1$), all $c_i \in G - \{c_0, c_{d-1}\}$, are *necessary* transfer values. ■

Corollary 3.8: When D is a signed digit set (i.e., $a<0<b$) and $\rho' \le r-1$, we have $c_0 \le a/(r-2) < 0 < b/(r-2) \le c_{d-1}$, implying that 0 is a necessary transfer value. Furthermore, $G = \{-1, 0, 1\}$ is adequate. ■

Because a four-valued G is always sufficient (except in a few practically insignificant special cases), compared to the binary encoding of the nonredundant digit set $[0, r-1]$, our stored-transfer representations need two bits of redundancy per digit. Virtually all practical redundant representations use power-of-two radices and thus imply at least one bit of redundancy. Therefore, the incremental cost of our scheme, in its initial form, and without the enhancement to be covered in Section 3.4, is one bit of redundancy per digit.

3.3 Speed and Cost Implications

The added cost of one bit per digit position buys us significant latency improvement in the basic operation of carry-free addition and all other arithmetic operations that use addition as a building block. In multioperand addition, and thus in multiplication, as well as in subtractive and multiplicative division, the per-add savings are compounded over many addition levels.

Because the main digit part can be in 2's-complement format with $\rho' = 0$, much of digit-level addition circuits can be based on readily available, and well optimized, binary adder cells. For example, a digit adder can be built from an h-bit binary adder, computing the $(h+1)$-bit sum $x_i' + y_i'$, followed by a special $(h+5)$-input, $(h+2)$-output circuit; the inputs are the aforementioned $(h+1)$-bit sum and two 2-bit stored transfers x_i'' and y_i'', while the outputs are the h-bit sum digit s_i' and a 2-bit generated transfer s_{i+1}''. Except for an $O(h)$-time digit addition, the rest of the computation may be performed in a small constant time, independent of the radix (see Section 3.4).

One way to compare the speed of addition in the stored-transfer scheme with other representations is to use the notion of *high-radix coefficient* introduced in [Jabθ3], where signed-magnitude/1's-/2's-complement encodings of redundant digits are studied. This coefficient corresponds to the number of simple digit-level addition and increment operations needed for adding two redundant numbers. As discussed above, stored-transfer representation has a high-radix coefficient of 1, where those of the other three representations are 2 for 2's-complement and 1's-complement, and 4 for signed-magnitude.

A comparison between our stored-transfer scheme and hybrid signed-digit representation [Phat94] will be provided in Section 3.5.

3.4 Two-Valued Stored Transfers

The representational efficiency of our stored-transfer scheme can be improved by using the following "trick". Consider a 3-valued transfer $x'' \in \{-1, 0, 1\}$ attached to a main digit $x' = 2u' + v'$, where $v' = x' \bmod 2$ and $u' = \lfloor x'/2 \rfloor$. We assume that x' is encoded in two parts: a single bit denoting v' and an arbitrary encoding of u'. A given stored-transfer digit $\langle 2u' + 0, 0\rangle$, as depicted in Figure 3.4, can be recoded as $\langle 2u' + 1, -1\rangle$, and $\langle 2u' + 1, 0\rangle$ as $\langle 2u' + 0, 1\rangle$, thus making it unnecessary to store the transfer value 0. The resulting 2-valued stored transfer renders the representational efficiency of our scheme competitive with the most efficient redundant representations. The cost of this recoding is small, given that it affects only a single bit v' in the encoding of x'. The case of a 3-valued transfer $x'' \in \{0, 1, 2\}$ is similar: recode $\langle 2u' + 0, 1\rangle$ as $\langle 2u' + 1, 0\rangle$, and $\langle 2u' + 1, 1\rangle$ as $\langle 2u' + 0, 2\rangle$.

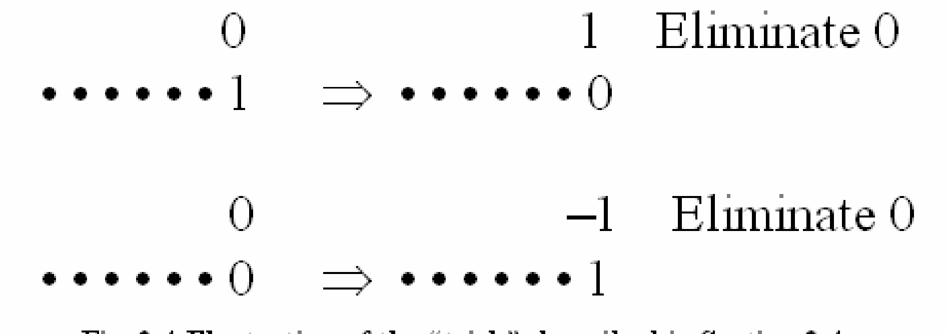


Fig. 3.4 Illustration of the "trick" described in Section 3.4

This scheme, which may be viewed as reintroducing step 3 of the carry-free addition process, but in much simpler form involving single-bit logical operations, can be applied after each carry-free addition operation to keep representations efficient in the arithmetic circuits and their associated registers or it can be applied only at the interface between the arithmetic unit and storage system.

Ad-hoc simplifications and efficient implementations for special cases of ρ' and G, may be derived. For example we give the following algorithm for addition of two stored-transfer digits x_i and y_i, where $\rho' = 0$ and $G = \{-1, 1\}$:

1. Form the h-bit 2's-complement value $z_i = x_i'' + y_i''$
2. Derive the carry-save sum $(u_i, v_i) = z_i + x_i' + y_i'$
3. Add u_i and v_i to form the binary position sum p_i
4. Derive s_i' and s_{i+1}'' satisfying $s_i' = p_i - r s_{i+1}''$
5. Adjust s_i'' and the least significant bit of s_i'

If we encode G as $\{0, 1\}$, the rightmost bit of z_i is always 0, the next bit is derived by an XNOR operation, and the identical leftmost $h - 2$ bits by a NOR operation. Standard full-adders may be used in step 2. Step 3 requires an h-bit ($h - 1$ if an extra half-adder is used in step 2) adder which can be of any suitable design. In step 4, s_i' and s_{i+1}'' are directly derived in constant time from p_i and its two most significant bits, respectively. Step 5 involves 1 gate delay, as previously discussed. Only step 3 has a latency that depends on h. Moreover, steps 1 & 2 and 3 & 4 may be partially overlapped to further reduce the constant-time component of the addition latency.

3.5 Very High Radix Representations

One context in which our scheme is particularly cost-effective is when the radix r is rather large. In this case, we have both lower relative redundancy and greater latency improvement over other radix-r redundant representations. In particular, our scheme can be viewed as a competitor for the hybrid redundancy scheme that provides a mechanism for high-radix redundant representation via incorporating binary signed-digit positions after each group of $h - 1$ ordinary binary positions [Phat94], [Phat99]. Our scheme shares many advantages of hybrid redundancy, while being capable of providing full symmetry in the number system (if desired), offering lower latency, and providing greater flexibility in circuit implementation.

We first compare the representation of k-digit radix-2^h numbers in the hybrid scheme, having 1 BSD and $h - 1$ ordinary bits per digit, with the two-valued stored transfer representation containing an h-bit main part, with $\rho' = 0$ and $G = \{-1, 1\}$. Both schemes require a total of $k(h + 1)$ bits. The range of a k-digit number in the hybrid scheme and in our scheme are $[-r/2, r - 1]\upsilon$ and $[-r/2 - 1, r/2]\upsilon$, respectively, where $\upsilon = (r^k - 1)/(r - 1)$. The maximal symmetric subrange is $[-r/2, r/2]\upsilon$ in both cases; that is, where symmetry is required, the two schemes exhibit the same representational efficiency.

Details regarding speed and circuit-cost comparisons will be dealt with in Chapter 8. Preliminary results indicate that, compared to hybrid redundancy, a few gates are saved in each digit position corresponding to a binary position in hybrid redundancy while a comparable number of extra gates are needed for each position corresponding to a BSD position. It thus seems that circuit-cost advantage exists for even moderate radices ($h > 2$) and the advantage becomes significant as we go to higher radices. These observations, along with the fact that any h-bit adder design can be used with stored-transfer representation, while hybrid redundancy implies a rather rigid realization, allows for experimentation with various design options and flexibility in optimizing implementation parameters. We will provide actual high level hardware designs in Chapters 5, 6, and 7.

3.6 Conversion to/from 2's Complement

To convert a 2's-complement number to a stored-transfer representation in radix 2^h, where $0, 1 \in G$, we deal with the h-bit groups of the 2's-complement number in parallel. We sign-extend (if necessary) the input number to an equivalent 2's-complement number whose width is a multiple of h. Then we use the ith group as the ith digit's main part and, except for the most significant group and $t_0 = 0$, set t_{i+1} equal to the most significant bit of the ith group, as depicted in Figure 3.5. If $t_{i+1} = 0$, the transfer clearly has no effect and the numerical value is preserved. When $t_{i+1} = 1$, its worth within the h-bit group is 2^{h-1} which is the same as 2^h (transfer) plus -2^{h-1} (negatively weighted bit in the 2's-complement main part). A constant-time postconversion adjustment, such as the one discussed in Section 3.4, is needed if G does not include $\{0, 1\}$.

For the reverse conversion, we add the main parts with their corresponding transfers, all in parallel. This yields a redundant number with 2's-complement digits. The rest of the process follows conventional redundant-to-binary conversion techniques [Parh00]. We note that converting a 2's-complement number to its stored-transfer equivalent requires little or no circuitry, since it is done by inserting a copy of some bits in place of the transfers. But the reverse conversion, as for any other redundant representation, involves word-width carry propagation.

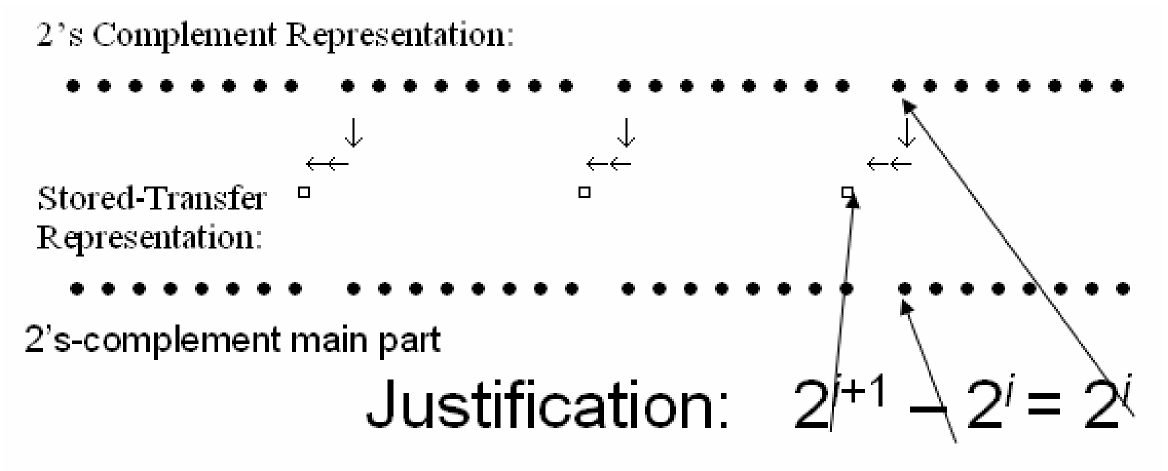

2's Complement Representation:
Stored-Transfer Representation:
2's-complement main part
Justification: $2^{i+1} - 2^i = 2^i$

Fig. 3.5 Two's complement to stored transfer conversion

3.7 Summary

We have shown that the stored-transfer representation of certain redundant numbers offers speed and cost benefits in the carry-free addition process. We proved the necessity of at least three transfer digit values and sufficiency of four values (in all practical situations), for carry-free addition. We further showed that by a simple adjustment in final stage of the carry-free addition algorithm, one can reduce the number of stored transfers to two values, thus requiring one bit for storage. Our stored transfer scheme is thus competitive with other practical redundant representations with regard to storage cost. In particular it has cost, speed, and symmetry advantages over hybrid redundancy.
We also demonstrated that converting a 2's-complement number to stored-transfer form implies virtually no cost or latency, while the reverse conversion needs the obligatory carry propagation. This affinity with 2's-complement numbers, in representation and circuit implementation, is a key strength of the stored-transfer scheme.

Derivation of algorithms for stored-transfer multiplication and division is quite feasible. Very-high-radix SRT division with signed-digit partial remainders and signed-digit quotient [Flyn01] can be modified to accept stored-transfer operands. A series of arithmetic operations can thus be performed without carry propagation by representing the inputs, intermediate results, and outputs in stored-transfer format.

Chapter 4 | Weighted Bit-Set Encodings

Contributions to redundant number representation are of two main types. In abstract studies, arithmetic algorithms are presented in terms of digit-level operations, specifying how each result digit is derived from operand digits and auxiliary quantities such as interdigit transfers [Parh00]. Implementation oriented studies, on the other hand, are often based on specific encodings for digit sets encountered in solving particular design problems; e.g., design of a high-speed 2's-complement full-tree multiplier [Taka85]. Some contributions of this latter type have dealt with limited classes of digit-set encodings without directly associating them with a specific design problem. Falling into the latter category are hybrid-redundant representation schemes [Phat94], [Phat01] and representation paradigms of high-radix signed digit numbers [Jab03].

In this Chapter we aim to fill the gap (see Fig. 4.1) between the aforementioned contributions by studying some efficient implementations for redundant arithmetic that are not tied to specific encodings, yet are not too removed from common hardware methods/structures used for arithmetic circuit implementations. When in carry-free addition, the transfer digit t_{i+1}, going from digit position i to digit position $i + 1$, is specified in terms of $x_i + y_i$ (e.g., by supplying comparison constants and their associated selection intervals [Parh90]), no specific encoding of the digit set is implied; it is also not implied that one must actually add the digits x_i and y_i in the conventional sense and then compare the resulting sum to the boundary constants. Specifying t_{i+1} in terms of the relationship between $x_i + y_i$ and comparison constants is simply an intuitive way of defining $t_{i+1} = \tau(x_i, y_i)$, where τ is the transfer function. This is akin to defining a logic function via a logic expression; even though the expression directly corresponds to a logic circuit, one is free to choose any other implementation of the same logic function. Typically, choices for the comparison constants to determine t_{i+1} are flexible, thus leaving room for imprecise comparisons and a variety of implementations based only on a subset of input bits.

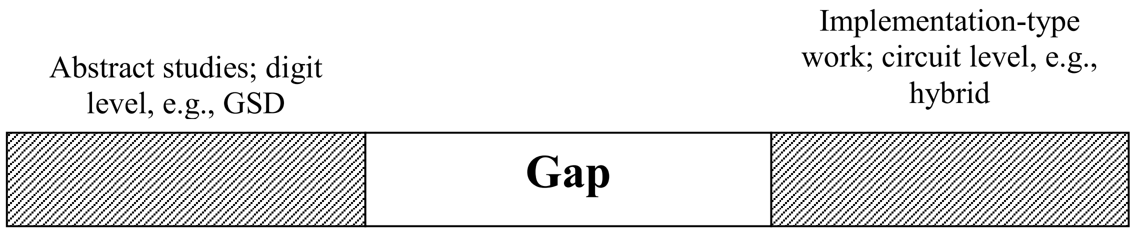

Abstract studies; digit level, e.g., GSD
Implementation-type work; circuit level, e.g., hybrid
Gap

Fig. 4.1 Spectrum of prior work on redundant number representation

We recognize that radices of practical interest are invariably powers of 2; thus, in practice, a redundant number can be viewed as a collection of digits, each weighted by a corresponding power of 2. Within each digit position, a digit value is also practically encoded as a collection of weighted bits. For example, the possibly asymmetric digit set $[-\alpha, \beta]$, with $\alpha \le 2^{\eta-1}$ and $\beta < 2^{\eta-1}$, might be encoded as an η-bit 2's-complement number, giving its bits the weights $-2^{\eta-1}$, $2^{\eta-2}$, . . . , 2, 1.

As another example, BSD numbers [Aviz61] are commonly encoded by representing the position-i digit as two bits weighted -2^i and 2^i; this is known as the (n, p) encoding [Parh90]. Under these conditions (i.e., power-of-2 radix and weighted-bit-set representation of each digit), the number as a whole is encoded by a collection of bits, each weighted by a positive or negative power of two.

4.1 The Notion of Weighted Bit-Sets

Definition 4.1 (WBS-encoded numbers): A *weighted bit-set* (WBS) encoding of width k is characterized by k integer pairs $(p_{k-1}, n_{k-1}), \ldots, (p_1, n_1), (p_0, n_0)$, where the representation has k radix-2 positions, indexed 0 to $k-1$, and digit position i $(0 \le i < k)$ of weight 2^i is comprised of n_i negatively weighted and p_i positively weighted bits. We require that $p_{k-1} + n_{k-1} > 0$. The most negative (positive) value represented by a WBS encoding is $-N$ (P), where $N = (n_{k-1} \ldots n_1 n_0)_{\text{two}}$ and $P = (p_{k-1} \ldots p_1 p_0)_{\text{two}}$. A given integer represented as $(v_{k-1} \ldots v_1 v_0)_{\text{two}}$, with $-n_i \le v_i \le p_i$, may have other WBS representations as well. ■

Definition 4.2 (Characterization of WBS encodings): The *encoding multiplicity* of position i in a WBS encoding is the total number $m_i = n_i + p_i$ of bits in that position. The ordered collection $m_{k-1} \ldots m_1 m_0$ of the positional multiplicities is the *multiplicity pattern* and $M = N + P$ is the *total multiplicity number*, which may be represented as the possibly redundant radix-2 number $(m_{k-1} \ldots m_1 m_0)_{\text{two}}$. Similarly, the ith *partial multiplicity number* M_i is $M_i = (m_{i-1} \ldots m_1 m_0)_{\text{two}} = N_i + P_i$ where $-N_i$ (P_i) is the most negative (positive) representable value by the rightmost i positions in the encoding. The total encoding cost is $E = \sum_{0 \le i < k} m_i$, leading to the encoding efficiency $e = \lceil \log_2(M+1) \rceil / E$. ■

Example 4.1 (A WBS-encoded number): An 8-position WBS-encoded number is shown in Fig. 4.2, where mi, n_i, and p_i values for each position are indicated, and other parameters are computed below:

$$N = (n_{k-1} \ldots n_1 n_0)_{\text{two}} = (2\ 2\ 1\ 0\ 3\ 1\ 2\ 0)_{\text{two}} = 448,\ P = (p_{k-1} \ldots p_1 p_0)_{\text{two}} = (2\ 0\ 1\ 0\ 1\ 2\ 1\ 2)_{\text{two}} = 308$$

$$v = (v_{k-1} \ldots v_1 v_0)_{\text{two}} = (1\ {}^{-}1\ 1\ 0\ {}^{-}3\ 2\ 1\ 1)_{\text{two}} = 83,\ M = N + P = (m_{k-1} \ldots m_1 m_0)_{\text{two}} = 758$$

$$E = \textstyle\sum_{0 \le i < k} m_i = 4 + 2 + 2 + 0 + 4 + 3 + 3 + 2 = 20,\ e = \lceil \log_2(M+1) \rceil / E = \lceil \log_2(759) \rceil / 20 = 0.5 \ ■$$

i	7	6	5	4	3	2	1	0
	1	–1	–0		–1	1	1	0
	–1	–0	1		0	–0	–0	1
	–0				–1	1	–0	
	1				–1			
(p_i,n_i)	(2,2)	(0,2)	(1,1)	(0,0)	(1,3)	(2,1)	(1,2)	(2,0)
m_i	4	2	2	0	4	3	3	2

Fig. 4.2 Characteristics of a 7-position WBS-encodednumber

Definition 4.3 (Negabits and posibits): We use *negabit*to denote a negatively weighted bit in [–1, 0] and *posibit* for a normal bit in [0, 1]. Graphically, • (○) stands for posibit (negabit) in a natural extension of standard dot notation. ■

Example 4.2 (Extended dot notation): Fig. 4.3 shows the extended dot notation of a WBS encoding with partial multiplicity numbers compared with that of the related nonredundant representation. The WBS-encoded number of Example 4.1 is an instance of the WBS encoding of Fig. 4.3. ■

i	7	6	5	4	3	2	1	0
	●	○	○		○	●	●	●
	○	○	●		●	○	○	●
	○				○	●	○	
	●				○			
M_{i+1}	756	244	116	52	52	20	8	2
$2^{i+1}-1$	255	127	63	31	15	7	3	1

Fig. 4.3 Extended dot notation for an 8-position WBS encoding

Example 4.3 (Familiar WBS-encoded numbers): The number representation systems whose descriptions follow are depicted in extended dot notation in Fig. 4.4. For unsigned carry-save representation, we have $m_i = p_i = 2$, $n_i = 0$, $\forall i$. Binary signed-digit (BSD) numbers with (n, p)-encoded digits have $n_i = p_i = 1$, $m_i = 2$. This represents, in effect, the 1's-complement encoding of the digit set $[-1, 1]$. Nonredundant 2's-complement number representation has $m_i = 1$, $\forall i$, $n_{k-1} = 1$, $p_i = 1$ for $i < k - 1$. For 2's-complement carry-save representation, we have $m_i = 2$, $\forall i$, with $n_{k-1} = 2$ and $p_i = 2$ for $i < k - 1$. In hybrid redundancy, with every fourth position being an (n, p)-encoded BSD digit, we have $m_i = p_i = 1$ and $n_i = 0$, except in positions whose index is 3 mod 4, for which $m_i = 2$, $n_i = 1$. ■

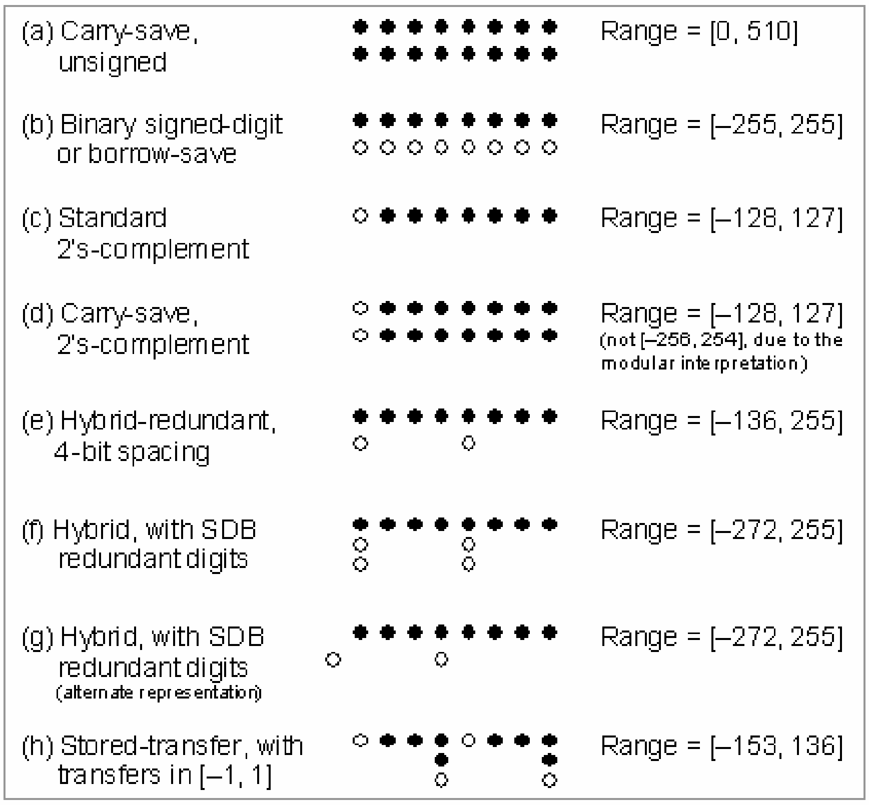

(a) Carry-save, unsigned
Range = [0, 510]
(b) Binary signed-digit or borrow-save
Range = [–255, 255]
(c) Standard 2's-complement
Range = [–128, 127]
(d) Carry-save, 2's-complement
Range = [–128, 127] (not [–256, 254], due to the modular interpretation)
(e) Hybrid-redundant, 4-bit spacing
Range = [–136, 255]
(f) Hybrid, with SDB redundant digits
Range = [–272, 255]
(g) Hybrid, with SDB redundant digits (alternate representation)
Range = [–272, 255]
(h) Stored-transfer, with transfers in [–1, 1]
Range = [–153, 136]

Fig. 4.4 Dot-notation representations for several familiar 8-position WBS-encoded number systems.

4.2 General WBS Encodings

In this section, we prove some general properties of WBS encodings. These general results are useful, because they cover, and tie together, numerous practical instances.

Definition 4.4 (Equivalent WBS encodings): WBS encodings representing precisely the same set of integer values are *equivalent*. *Strongly equivalent* WBS encodings are equivalent and have the same width k. ■

Example 4.4 (Equivalent WBS encodings): The 8-position WBS encoding shown at the top of Fig. 4.5 is equivalent to the 7-position WBS encoding shown below it, and strongly equivalent to the 8-position encoding appearing at the bottom of Fig. 4.5. ■

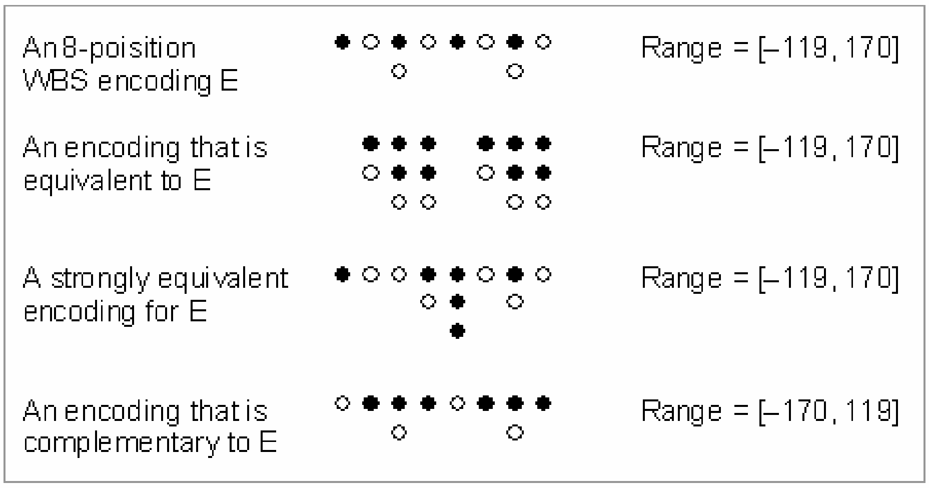

An 8-poisition WBS encoding E
Range = [–119, 170]
An encoding that is equivalent to E
Range = [–119, 170]
A strongly equivalent encoding for E
Range = [–119, 170]
An encoding that is complementary to E
Range = [–170, 119]

Fig. 4.5 Equivalent WBS encodings.

Theorem 4.1: An interval $[-N_k, P_k]$ of integer values containing $M_k + 1$ consecutive integers is representable by a WBS encoding with multiplicity pattern $m_{k-1} \ldots m_1 m_0$ iff for all i in the range $0 < i < k$, we have $M_i \geq 2^i - 1$.

Proof: The necessity part is easy to prove. If $M_i < 2^i - 1$ for some i, then positions 0 to $i - 1$ collectively represent fewer than 2^i distinct values. At least one of the 2^i mod-2^i equivalence classes must be unrepresented among these values. Given that bits in positions i and higher can only represent multiples of 2^i, there must be gaps in the representation. We prove the sufficiency part by induction on k. Recall that the multiplicity m is nonzero for the most-significant position of our postulated WBS representation. This leads to $m_0 > 0$, because either position 0 is the only position or else the condition of the theorem statement guarantees $M_1 = m_0 \geq 2^1 - 1$. The base case is $k = 1$; a one-position WBS representation with $m_0 > 0$, and clearly covers all integers from $-N_1 = -n_0$ to $P_1 = p_0$. Now suppose that the theorem holds for any WBS representation with $k - 1$ or fewer positions. Let a k-position WBS representation be obtained by extending a $(k - g)$-position representation, where $g \geq 1$, with $m_{k-1} > 0$ and $m_j = 0$ for $k - g \leq j < k - 1$; i.e., the leftmost g components of multiplicity pattern are $m_{k-1}0\ 0 \ldots 0$. Then, by our assumptions, $M_{k-1} = M_{k-2} = \ldots = M_{k-g} \geq 2^{k-1} - 1$. This implies that positions 0 to $k - 2$ can collectively represent a continuous interval of integers with at least 2^{k-1} consecutive values. These values combined with multiples of 2^k representable by the bit(s) in position $k - 1$ yield a continuous interval of integers overall. ■

Theorem 4.1 suggests that even though it is possible to avoid having any posibit or negabit in a particular position j of a WBS encoding, doing so would require additional bits in lesser significant positions (two in position j–1, or four in position j–2, etc.). Thus, for encoding efficiency, it is advantageous to enforce $m_i > 0$ for all i. On the other hand, replacement of a pair of bits of the same polarity in position j by one bit in position $j + 1$, through the substitutions outlined in Fig. 4.6, keeps $m_i \leq 2$, and further improves encoding efficiency. These observations lead us to define the class of canonical WBS encodings.

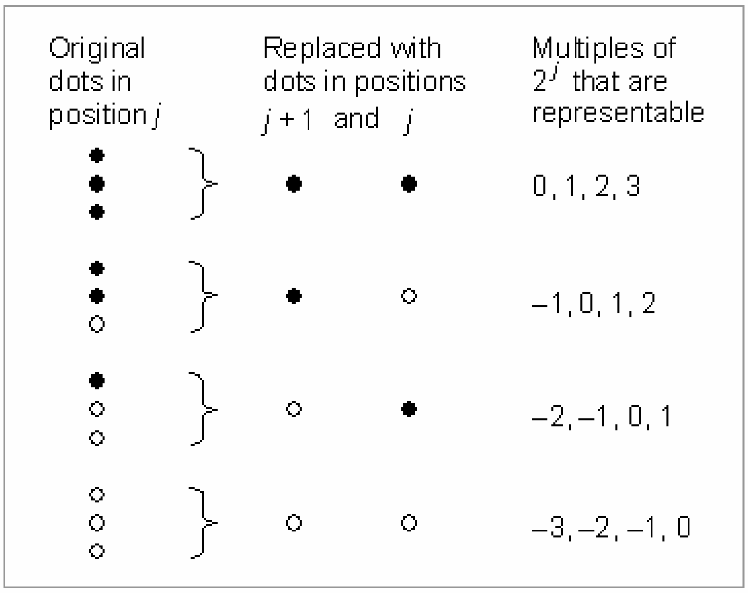

Original dots in position j
Replaced with dots in positions $j+1$ and j
Multiples of 2^j that are representable
0, 1, 2, 3
−1, 0, 1, 2
−2, −1, 0, 1
−3, −2, −1, 0

Fig. 4.6 Substitutions used in the proof of Theorem 4.2.

Definition 4.5 (Canonical WBS encodings): A k-position WBS encoding is *canonical* iff $1 \leq m_i \leq 2$ for $0 \leq i \leq k-2$. ■

Theorem 4.2: Any WBS encoding with the multiplicity pattern $m_{k-1} \ldots m_1 m_0$ satisfying $M_i \geq 2^i - 1$ for $0 < i < k$, and thus representing a continuous interval of integers in view of Theorem 4.1, is strongly equivalent to some k-position canonical WBS encoding.

Proof: We describe the process for deriving the desired canonical encoding from a given WBS encoding. Scan the multiplicities m_i from the right until you find $m_j \geq 3$ for some $j < k - 1$. If no such position exists, the encoding is already in the desired canonical form. If you find $m_j \geq 3$, take three of the bits in position j and make the substitution shown in Fig. 4.6. This does not change the set of values representable, and it reduces m_j by 2. Repeating this process eventually leads to $m_j \leq 2$ for $0 \leq j < k - 1$. To show that the resulting multiplicities satisfy $m_j \geq 1$, $0 \leq j < k - 1$, we note that $M_j = (0m_{j-2} \ldots m_0)_{\text{two}}$ has a value of $2^j - 2$ when all the multiplicities assume the maximal value of 2.

Corollary 4.1: A given WBS encoding is *redundant* iff in its equivalent canonical forms, $m_j > 1$ for some $j < k$. ■

Example 4.5 (Deriving the canonical encoding): Fig. 4.7 depicts the derivation of a canonical encoding, strongly, equivalent to the WBS encoding of Fig. 4.3. ■

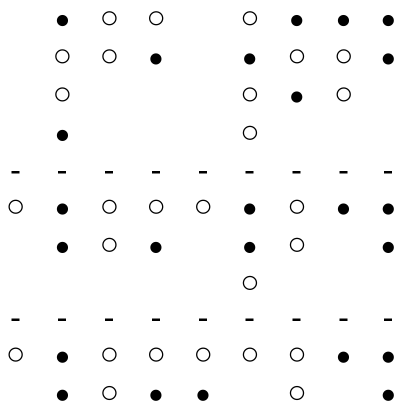


Fig. 4.7 Derivation of a canonical WBS encoding, strongly, equivalent to the encoding of Fig. 4.3

4.3 Peiodic WBS Encodings

Whereas arbitrary WBS encodings can be envisaged and used, circuit implementation in VLSI favors regularity in the number of bits associated with the various positions. Thus, we define the class of periodic WBS encodings.

Definition 4.6 (Periodic WBS encodings): A k-position WBS encoding is *periodic* iff there exist $h < k$ with $n_{i+jh} = n_i$ and $p_{i+jh} = p_i$ for all j; the smallest such h is the *period*. ■

Assuming k to be a multiple of h, a periodic WBS encoding represents a generalized signed-digit (GSD) number system in radix 2^h utilizing the digit set $[\alpha, \beta]$, with $\alpha = -(n_{h-1} \ldots n_1 n_0)_{\text{two}}$ and $\beta = (p_{h-1} \ldots p_1 p_0)_{\text{two}}$.

Given that full and half-adder cells, which are widely available and efficient, can be used to combine a set of bits with power-of-2 weights into another set of similarly weighted bits, periodic WBS encodings may be viewed as practically desirable GSD representations. In fact, all GSD representations that we have encountered in the literature are based on WBS encodings. Some examples are shown in Table 4.I. For those digit sets in Table 4.I that are symmetric, signed-magnitude encoding could conceivably be used, leading to a non-weighted-bit representation. However, we have been unable to find an actual implementation that is based on such a representation.

Theorem 4.3: For an interval $[-N, P]$ of integers, that includes 0, and integer k satisfying $1 \le k \le \log_2(N + P + 1)$, there exists a unique k-position canonical WBS encoding representing exactly $[-N, P]$.

Proof: A trivial one-position WBS encoding with the given range has $n_0 = N$, $p_0 = P$, and $M = m_0 = N + P$. The unique k-position canonical encoding equivalent to the above can be easily derived by the construction of Theorem 4.2. ■

Corollary 4.2: For a radix-2^h GSD number system with digit set $[-\alpha, \beta]$, there is a unique periodic canonical WBS encoding with period h, where $1 \le h \le \log_2(\alpha + \beta + 1)$. ■

Table 4.I Some commonly used periodic WBS redundant number system encodings.

Digit set	Encoding name	# bits	Bit weights
[–1, 1]	(n, p)-encoded binary signed-digit	2	1, –1
[–1, 1]	2's-complement-encoded binary signed-digit	2	–2, 1
[–2, 2]	2's-complement-encoded minimally redundant radix-4	3	–4, 2, 1
[0, 2^h]	Radix-2^h stored-carry	$h + 1$	2^{h-1}, . . ., 2, 1, 1
[–1, 2^h–1]	Radix-2^h stored-borrow	$h + 1$	2^{h-1}, . . ., 2, 1, –1
[–1, 2^h]	Radix-2^h stored-carry-or-borrow	$h + 2$	2^{h-1}, . . ., 2, 1, 1, –1
[–2^{h-1}, 2^h–1]	Radix-2^h hybrid, with (n, p)-encoded BSD position	$h + 1$	–2^{h-1}, 2^{h-1}, . . . , 2, 1
[–2^h, 2^h–1]	Radix-2^h hybrid, with redundant position in [–2, 1]	$h + 1$	–2^h, 2^{h-1}, . . ., 2, 1
[–2^{h-1}–1, 2^{h-1}]	Radix-2^h stored-transfer, with transfers in [–1, 1]	$h + 2$	–2^{h-1}, 2^{h-2}, . . ., 2, 1, 1, –1

The next to last entry in Table 4.I exemplifies a case where the bits in the encoding of adjacent digits overlap in terms of their weights. Such overlaps are avoidable by simply regrouping the bits. Figure 4.4 shows an example where the bits in a periodic WBS encoding with $h = 6$ are grouped in three different ways, each leading to a distinct digit set in radix-64 interpretation. Such variations are indeed useful for optimizing circuit implementations. Note that in the second and third groupings in Fig. 4.8, the boundary groups in the least- and most-significant end need special treatment, but this is generally not problematic. Note also that if two digits in [–5, 65] are added, the obtained sum in [–10, 130] is representable by the third digit set in Fig. 4.8. Hence, these two options in Fig. 4.8 collectively represent a *stored-transfer* scheme for carry-free addition [Jabe01]. This observation leads to the following general result.

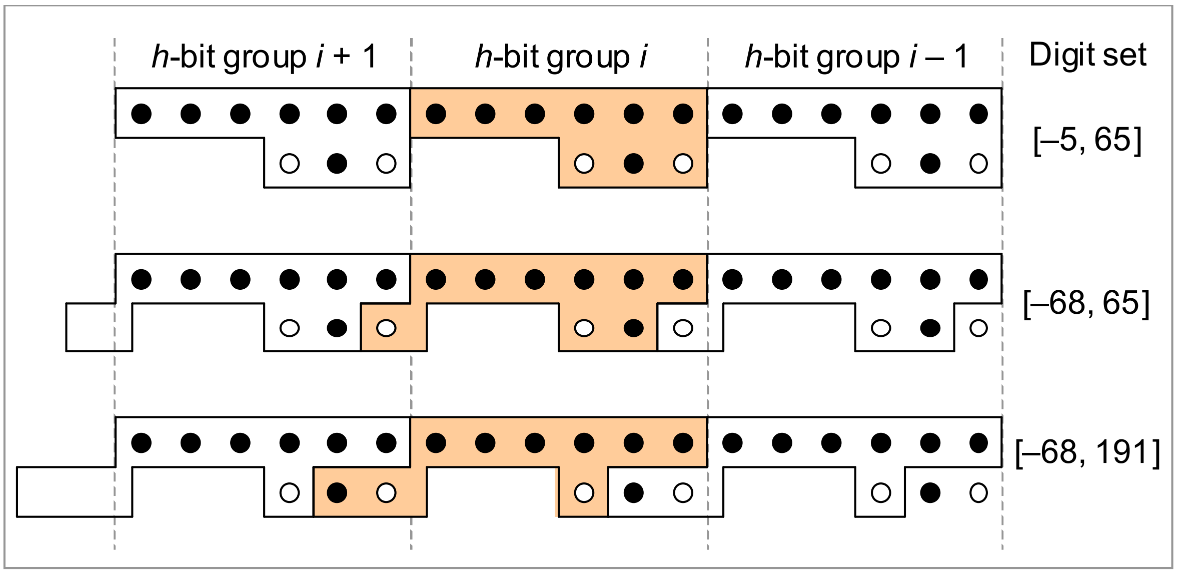


Fig. 4.8 Three different interpretations of the same periodic WBS encoding.

Theorem 4.4: Any stored-transfer scheme for radix-2^h GSD addition, where transfers are encoded as a set of posibits and negabits, can be explained in terms of bit grouping in a suitably chosen WBS encoding.

Proof: A stored-transfer scheme [Jabe01] is characterized by a main digit set $[a, b]$ and a transfer set $\{c_0, \ldots, c_{d-1}\}$, together constituting the radix-2^h digit set $[\alpha, \beta]$. If the transfer values are encoded as a set of posibits and negabits, as assumed, and the main digit set is encoded likewise, the overall representation is a periodic WBS encoding whose parameters m_j, n_j, and p_j within one period or radix-2^h digit, $0 \le j < h$, are obtained by adding the respective parameters of the main digit set and the transfer set. ■

4.4 Framework for WBS Arithmetic

Numbers with arbitrary digit sets can be added digitwise to produce a sum with a digit set whose range is the sum of the ranges of the operands. This wider digit set can be kept intact and the result used as an operand in further arithmetic operations. It is also possible to convert the wider digit set to another, more convenient, one for further processing. Often, however, it is required to obtain results with the same digit set as inputs. Such representationally closed arithmetic is desirable for reusability of the arithmetic cells and regularity in VLSI circuit implementation. We note that when comparing a representationally closed scheme against a scheme that is not closed, fairness dictates that the overhead of conversion from the intermediate representation to the ultimate encoding be taken into account in any cost/speed comparison.

In circuit implementations, posibits are more easily dealt with than a mix of posibits and negabits, because they can be combined and regrouped using standard full adder, half-adder, and parallel counter cells. This motivates us to define 2's-complement-like WBS encodings in which negabits appear only in the most significant position $k - 1$, with all other positions holding only posibits.

Definition 4.7 (Two's-complement-like WBS encodings): A k-position WBS encoding is 2's-complement-like (2CL) if $m_i = p_i$, $0 \le i \le k - 2$. In a canonical 2CL-WBS encoding, we have $1 \le m_i = p_i \le 2$, $0 \le i \le k - 2$. ■

Theorem 4.5: For any k-position WBS encoding, there exists a unique $(k + 1)$-position canonical 2CL-WBS encoding. Furthermore, the latter can be constructed efficiently.

Proof: We describe the process for deriving the canonical 2CL-WBS encoding from a WBS encoding Ω. Consider a WBS encoding Ω' with the same multiplicity pattern as Ω, but with $p_i = m_i$, $\forall i$. Clearly, the range of Ω' is $[0, N + P]$. Now form $(k + 1)$-bit 2CL representation of the constant $-N$ with a single posibit in each of the positions 0 through $k - 1$ and one or more negabits in position k. Obtain the WBS encoding Ω'' by adding to each position of Ω' a posibit (one or more negabits in the case of position k) where the 2CL representation of $-N$ contains 1s. Clearly, the range of Ω'' includes $[-N, P]$. The desired canonical 2CL-WBS encoding is obtained by applying the first substitution of Fig. 4.6 to positions 0 to $k - 1$ that have more than 2 posibits until each of them holds 1 or 2 posibits. The process of converting a WBS number to a 2CL-WBS encoding can be implemented in parallel using time that is logarithmic in the depth d of the starting representation. ■

Example 4.6 (Conversion to 2-CL WBS): Fig. 4.9 shows the 8-position WBS-encoding of – 448 with position 4 being empty, and its equivalent 9-position 2-CL WBS encoding. ■

```
        0  –1  –1      –1   0   0   0
       –1  –1   0       0  –1  –1   0
       –1              –1   0  –1
        0              –1
  -     -   -   -   -   -   -   -   -
 –1     0   1   0   0   0   0   0   0
 –1
```

Fig 4.9 4-deep WBS-encoded representation of – 448 and its 2-CL WBS equivalent

4.5 WBS Addition and Multiplication

In this section, we briefly describe algorithms for addition, subtraction, and multiplication of canonical 2CL-WBS numbers. Arithmetic algorithms for other operations, perhaps with a different encoding for each operand, can be developed by using either pre- or post-operation conversion. With preconversion, operands are changed to 2CL-WBS format before an operation. Postconversion allows an intermediate result (e.g., juxtaposition of bits for addition or matrix of bitwise products for multiplication) to be formed based on the original operand bits.

Addition of two 2CL-WBS operands is performed by conceptually copying the bits of the 2-deep operands in the bit placeholders of a 4-deep WBS representation. This is then followed by conversion to canonical 2CL-WBS representation. Subtraction is similar, except that the posibits (negabits) of the second operand become negabits (posibits) in the intermediate 4-deep result.

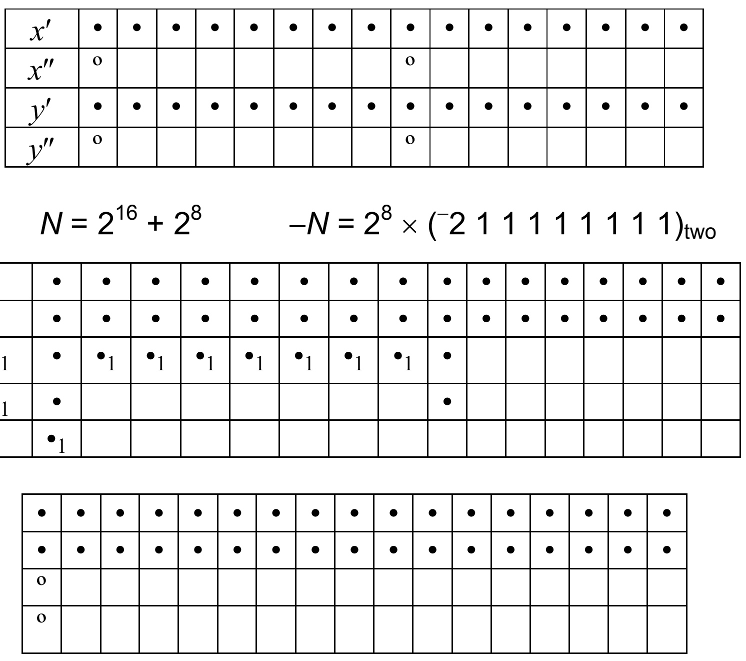


Fig 4.10 Conversion to 2CL-WBS

Example 4.7 (WBS addition): Fig. 4.10 depicts the addition procedure of two 2CL-WBS numbers, where $\bullet_1$ means a 1-valued posibit, and $^{o}{}_1$ means a (–1)-valued negabit. ■

To multiply two 2CL-WBS encoded numbers, we might first derive a partial product bit matrix, and then reduce it through compression. The number of bitwise products to be dealt with can be 4 times greater than in standard binary multiplication, given the depth of two for each operand. One way to reduce the complexity of our multiplier is to reduce the number of positions holding 2 posibits through partial carry assimilation. For example, if 4-bit segments of each 2-deep operand are combined to yield 5-bit binary numbers, with the MSB of one number aligned under the LSB of the next higher number, a radix-16 carry-save representation results for which efficient multiplication circuits have been studied [Ferg99].

4.6 WBS Conversions

Conversions of interest are: (1) 2's complement to WBS, (2) WBS to 2's complement, and (3) One WBS form to another. Because the last category is quite varied, with conversion strategies differing depending on the source/target formats, we do not discuss it here in any detail except to note that any WBS-to-WBS conversion between formats of the same period can be viewed as digit-set conversion which can be performed in parallel and carry-free manner. A possible conversion strategy is to use the 2CL-WBS format as an intermediate format, thus needing to supply only a method for converting from 2CL-WBS to an arbitrary WBS.

Conversion from 2's-complement to 2CL-WBS is trivial, while conversion to a periodic WBS format can be done either directly as digit-set conversion or by first going to 2CL-WBS as an intermediate format. In either case, circuit implementation will be parallel and regular (consisting of identical cells), except in the most-significant end where the number sign must be processed differently. Conversion from WBS to 2's complement can similarly go through 2CL-WBS as an intermediate representation. The first phase (arbitrary WBS to 2CL-WBS) is carry-free, but the second phase, like all redundant to nonredundant conversions, requires full carry propagation and is thus a logarithmic-time process at best.

4.7 Summary

In this chapter, we introduced the class of weighted bit-set (WBS) redundant number representations that can lead to a fairly general strategy for obtaining efficient circuit implementations for redundant arithmetic using readily available, and highly optimized, building blocks developed for conventional binary arithmetic. For a given generalized signed-digit or hybrid-redundant representation, one can derive a suitable WBS encoding. The resulting encoding has the advantage that its intradigit propagation can be limited to posibit transfers, while in other instances, including hybrid redundancy, positive and negative carries coexist, leading to slower circuit implementations.

Extended WBS encodings that allow general two-valued digits, dubbed *twits* (e.g., having values in {–1, 1}, {0, 2}, or {0, –2}), will be investigated in Chapter 7 as a natural extension of WBS encoding. This generalization not only enhances the encoding efficiency but also leads to speed gains in many instances. We will show that twits can be processed by essentially the same circuits that are applied to bits or negabits in this chapter and will develop more complex twit-based arithmetic algorithms in the next chapters.

Chapter 5 Universal Addition Scheme for Hybrid redundancy

Redundant number representations are used extensively to speed up arithmetic operations within both general-purpose and special-purpose digital systems [Aviz61], [Parh90]. The speed advantage resulting from carry-free arithmetic with redundant representations is often large enough to offset the format conversion overheads, even in signal processing and other applications with moderate frequency of arithmetic operations. When the conversion and reconversion circuitry can be shared among multiple function units, redundant representations also lead to savings in VLSI area and power dissipation, thus making them even more attractive. Like conventional digit sets, redundant digit sets can be encoded in any desired way. However, in practice, encodings comprised of weighted positive and negative bits have been found to offer advantages in implementation simplicity and modularity, including the applicability of standard cells used in binary arithmetic [Jabe02].

Uniform treatment of negatively weighted and normal bits is responsible for the simplicity and widespread application of 2's-complement arithmetic [Baug73], [Koba85]. We use *negabits* in {–1, 0} for the former and *posibits* in {0, 1} for the latter [Jabe02]. Negabits have been widely used in redundant number representations. For example, binary signed-digit (BSD) numbers [Aviz61] are commonly encoded by using two bits weighted -2^i and 2^i for the position-i digit; viz. the (n, p) encoding [Parh90]. Similarly, in some variants of radix-2 hybrid-redundant numbers [Phat01], redundant digits such as stored-double-borrow (SDB), in [–2, 1], or stored-borrow-or-carry (SBC), in [–1, 2], may be represented by a collection of posibits and negabits, leading to *weighted bit-set* (WBS) encodings [Jabe02]. For example, the WBS encoding of a redundant SDB digit consists of two negabits and one posibit in the same position, or equivalently, of a negabit in position $i + 1$ and a posibit in position i. Other possibly useful variants of digits in redundant positions of a hybrid-redundant number, as enumerated in [Phat01], are stored-carry (SC), in [0, 2], and stored-double-carry (SDC), in [0, 3]. The latter digit set has also been used in the design of redundant adders [Erce97].

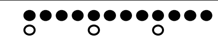
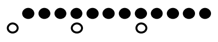
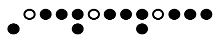
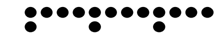
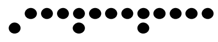
Table 5.I depicts symbolic representations for BSD, SDB, SBC, SC, and SDC digits, where a posibit (negabit) appears as ● (○). The double-position representations of these redundant digits have been used in Table 5.II, which depicts five variants of radix-2^h hybrid representations for $h = 4$. The WBS encodings of Table 5.II are all 2-deep encodings (i.e., contain no more than 2 dots in any position) with no empty position; these are known as canonical WBS encodings [Jabe02]. The third entry of Table 5.II is an example of allowing a negabit in a nonredundant position. By allowing negabits to appear in arbitrary nonredundant positions, canonical WBS encodings, which include all the variants of hybrid redundancy studied by Phatak et al [Phat01], offer new hybrid-redundant systems not explored before. This nonredundant use of negabits can be seen in 2's-complement numbers and, more recently, in certain stored-transfer representations of redundant numbers [Jab01]. In Section 5.2, we show that this possibility leads to interesting new symmetric variants of hybrid-redundant digit sets.

Table 5.I Single/double-position WBS representations

Digit	Single-position encoding	Double-position encoding
BSD	● ○	N/A
SDB	○ ○	○ ●
SBC	○ ●	● ○
SC	● ●	N/A
SDC	● ●	● ●

Addition of two canonical WBS operands is performed by conceptually copying the bits of the 2-deep operands in the bit placeholders of a 4-deep WBS representation. However, since the operands are 2 deep, it is desirable to convert the sum to a 2-deep encoding as well. In Section 5.1, we explore an efficient and uniform implementation for constant-time addition of two hybrid redundant numbers with 2-deep result, where the operands need not belong to the same hybrid-redundant number system (i.e., redundant positions of the result are shifted one position to the left of the redundant position of the operands). We offer representationally closed addition schemes for all the previously studied variants of hybrid-redundant number systems and the new symmetric variants in Section 5.3. In these implementations the results belong to the same number system as the operands.

Table 5.II Five hybrid-redundant number systems

Composition (digit pattern)	WBS encoding with 3 digits
1 BSD, $h - 1$ binary	
1 SDB, $h - 1$ binary	
1 SBC, $h - 1$ binary	
1 SC, $h - 1$ binary	
1 SDC, $h - 1$ Binary	

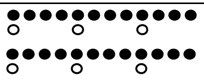
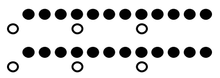
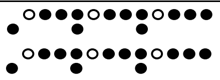
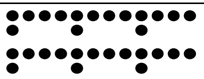
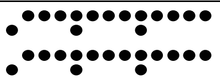
To multiply two canonical WBS encoded numbers, we might first derive a partial product bit matrix, composed of posibits and negabits, and then reduce it through compression. In Section 5.4, we show that by inverted encoding of negabits we can use the standard compressors, such as (3; 2) and (4; 2) counters, for partial product reduction. The number of bitwise products to be dealt with can be 4 times greater than in standard binary multiplication, given the depth of two for each operand. But the second component of each hybrid redundant operand is relatively sparse compared to the first component.

Therefore, one way to reduce the complexity of our multiplier is to reduce the number of positions holding 2 posibits through partial carry assimilation. For example, if 4-bit segments of each 2-deep operand are combined to yield 5-bit binary numbers, with the MSB of one number aligned under the LSB of the next higher number, a radix-16 carry-save representation results for which efficient multiplication circuits have been studied [Ferg99].

5.1 Adding Hybrid Redundant Numbers

The first step in our addition scheme for WBS encoding of hybrid-redundant numbers is to construct a 4-deep WBS number by aligning the two operands one under the other as in Table 5.III. The equal weight grouping offered in [Phat01] may be considered as a special case. Next we need to reduce the 4-deep result to an equivalent 2-deep result. In the case of SC and SDC hybrid numbers (Table 5.I), any conventional reduction scheme may be used for this purpose [Parh00].

Table 5.III Addition of 2-deep operands with 4-deep results

Composition (digit pattern)	4-deep addition result
1 BSD, $h - 1$ binary	
1 SDB, $h - 1$ binary	
1 SBC, $h - 1$ binary	
1 SC, $h - 1$ binary	
1 SDC, $h - 1$ binary	

For example, one full-adder (FA) per nonredundant position and two FAs in redundant positions are all we need to reduce the 4-deep interim sum of two SC hybrid operands to a 2-deep result (Fig. 5.1). Note that the sum in Fig. 5.1 is encoded slightly differently from the operands in that its least-significant group is one position longer (i.e., has $h + 1$ positions). It is easily seen that a reduction scheme similar to that of Fig 5.1 is applicable to the addition of SDC hybrid numbers.

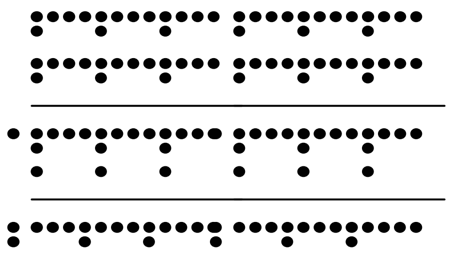


Fig. 5.1 Reduction of the addition result to a 2-deep result.

The second, third, and fifth rows of Table 5.I depict two equivalent encodings for SDB, SBC, and SDC digits. The equivalent 3-deep and 1-deep representations for an SDC digit bring to mind the functionality of a binary full-adder and suggests that similar devices for 3-deep to 1-deep conversions for SDB and SBC digits might also be feasible. For example, consider the PPM cell used in the design of a borrow-save adder [Mign00], a dual-purpose (rather complex) logic for addition of two SDB digits or two SBC digits offered in [Phat01], and four variants of half adders, reducing alternate combinations of equally weighted posibits and negabits to equivalent carry and sum posibits and negabits, proposed in [Daum00]. It turns out, however, that a full-adder is all that we need, provided that we use an inverted encoding for a negabit; that is, encoding –1 as 0 and 0 as 1, which is exactly the opposite of the conventional encoding for negabits.

Table 5.IV (5.V), shows the functionality of a conventional FA as reducing a collection of two negabits (posibits) and one posibit (negabit), all in position i, to a negabit (posibit) in position $i + 1$, and a posibit (negabit) in position i. We have used the convention of [Jabθ2] for variable names: uppercase letters for negabits, lowercase for posibits. The contents of the first three and the last two columns of each table are identical to the truth table for a full-adder, hence the functionality of full-adders for reducing any set of three posibits and inversely encoded negabits; the case of three negabits is obvious.

Table 5.IV Reduction of two negabits and one posibit

X_i	Y_i	c_i	$X_i + Y_i + c_i$	C_{i+1}	s_i
0	0	0	–2	0	0
0	0	1	–1	0	1
0	1	0	–1	0	1
0	1	1	0	1	0
1	0	0	–1	0	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	1	1	1

Table 5.V Reduction of two posibits and one negabit

X_i	y_i	c_i	$X_i + y_i + c_i$	c_{i+1}	S_i
0	0	0	–1	0	0
0	0	1	0	0	1
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	2	1	1

To reduce a 4-deep sum of two hybrid redundant operands to a 2-deep one, we use one full-adder per nonredundant position, and two full-adders for each redundant position. Figures 5.2a and 5.2b depict adder cells for redundant and nonredundant positions, respectively, where following the convention in [Jabe02], primed variables belong to the first components of the operands and the result, double-primed variables represent bits in the second components, and unprimed variables indicate intermediate carries.

A full-adder in a nonredundant position receives two inputs from the same nonredundant position of the operands, and a carry from the previous full-adder, producing a nonredundant sum bit and a carry to the next position (Fig. 5.2b). In a redundant position, the top full-adder, as in Fig. 5.2a, reduces three of the bits to a sum bit feeding the lower full-adder, and a carry to the next higher positioned full-adder. The lower full-adder absorbs the carry from the last position, receives the sum bit from the top full-adder, and the fourth bit of the redundant position, and produces a nonredundant sum bit and a carry to rest as the second bit of the now left-shifted redundant position. These adder cells may be used for all five hybrid-redundant representations of Table 5.II, which coincide with those covered in [Phat01].

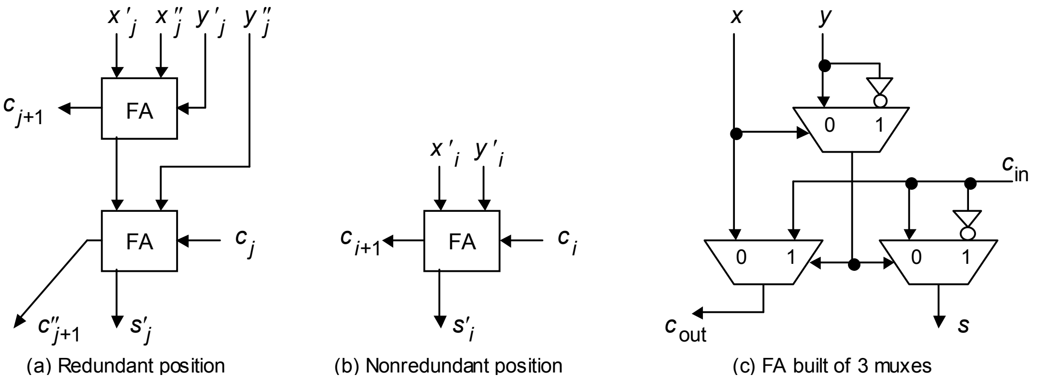


Fig. 5.2 Adder cells for hybrid representations of Table 5.II, and a 3-multiplexer full-adder.

It is interesting to note that in the preceding discussion, the operands need not belong to the same hybrid-redundant number representation. Moreover, it can be easily verified that they work for addition of any two canonical WBS numbers. This includes hybrid-redundant numbers with negabits in their nonredundant positions, which we call *extended hybrid-redundant numbers*. However, the result pattern may be slightly different from either operand (i.e., with redundant positions of the result shifted one position to the left of the corresponding redundant positions of the operands). If the extended dot notation [Jabe02] of two 4-deep WBS numbers (possibly resulting from the first step of addition of two canonical WBS operands), irrespective of the bit polarities, are identical then the reduction circuitry is exactly the same.

The total adder delay is equal to that of $d + 2$ full-adders, where d is the longest spacing (in terms of the number of nonredundant positions) between two redundant positions. Our universal adder has a number of advantages over previous implementations. The only building block required in our design is full-adder, which leads to more regularity, possibility of using highly optimized FA cells, and employing any standard carry acceleration technique to achieve an O(log d) total delay.

The cost per nonredundant position is minimal (i.e., one FA, as in nonredundant addition), while for redundant positions there is only one extra FA. Given that each FA can be implemented with three multiplexers (Fig. 5.2c), our adder cell for redundant positions costs six multiplexers, while the one propsed in [Phat01] for SDB and SBC hybrid cases is made of seven multiplexers, plus a few other gates. This is a pleasant surprise, because the use of standard cells often implies an increase in component count (layout area) or a sacrifice in performance.

5.2 Symmetric WBS Hybrid Redundancy

Hybrid signed-digit (HSD) representations, introduced in [Phat94] and extended in [Phat01] to allow alternate digit sets in redundant positions, are essentially asymmetric, except for the limiting case that coincides with the fully redundant BSD number system. The reason is that in the three of the variants where redundant digits include negative values, there is one equally weighted posibit for each negabit, while other positions hold only posibits. For example, radix-2^h digit sets associated with the hybrid representations shown in Table 5.II are $[-2^{h-1}, 2^h - 1]$, $[-2^h, 2^h - 1]$, $[-2^{h-1}, 3 \times 2^{h-1} - 1]$, $[0, 3 \times 2^{h-1} - 1]$, and $[0, 2^{h+1} - 1]$, respectively. We will show, in Chapter 6, that besides the BSD number system, the ordinary hybrid redundancy (i.e., allowing nonredundant positions to hold only posibits) provides for only one other 2-deep symmetric representation, which is the minimally redundant radix-4 SD number system. Fig. 5.3 shows a classification of redundant representations based on weighted bits and, in particular, depicts the place of various hybrid-redundant representations.

A canonical WBS digit set is redundant if and only if there is at least one position holding a set of more than one posibits and/or negabits [Jabe02]. In other words a position with only one posibit or negabit is nonredundant, while any other position is a redundant one, given the fact that in a canonical WBS encoding there is no empty position. This flexibility further extends the hybrid redundancy scheme to allow negabits both in redundant and arbitrary nonredundant positions. We use this extension, which will be more elaborated upon in Chapter 6, to design symmetric hybrid-redundant representations with arbitrary different spacing between consecutive redundant positions. For example consider the periodic radix-16 extended hybrid redundant number of Fig. 5.4, where the digit set is $[-8, 8]$. The adder cells as in Figs. 5.2a and 5.2b work for this number system as well, but the addition process is not representationally closed; the pattern of dots in the sum is shifted to the left by one binary position relative to the input operands.

5.3 Representationally Closed Addition

Numbers with arbitrary digit sets can be added digitwise to produce a sum with a digit set whose range is the sum of the ranges of the operand digits. This wider digit set can be kept intact and the result used as an operand in further arithmetic operations. It is also possible to convert the wider digit set to another, more convenient, one for further processing. Often, however, it is required to obtain results with the same digit set as inputs [Korn99]. Such representationally closed arithmetic is desirable for storage efficiency, reusability of the arithmetic cell designs, and regularity in VLSI circuit implementation.

While encoding-algorithm combinations that are not representationally closed can be useful and are in fact used in practice, when comparing a representationally closed scheme against a scheme that is not closed, fairness dictates that the overhead of conversion from the intermediate representation to the ultimate encoding be taken into account in any cost/speed comparisons. We explore representationally closed constant-time addition schemes for practical cases where the double primed components of the canonical WBS operands are relatively sparse. We present a general addition algorithm below and subsequently apply it to specific cases.

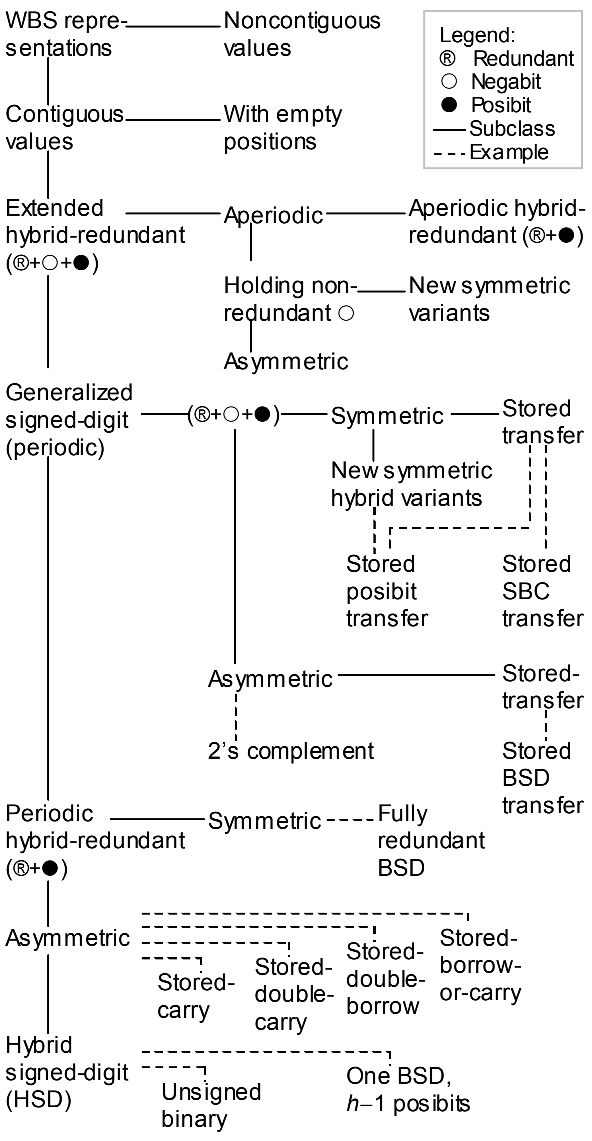


Fig. 5.3 The hierarchy of number representations using weighted components (tree branches go from left to right and top to bottom).

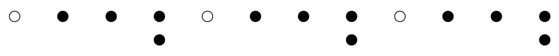


Fig. 5.4 A symmetric hybrid-redundant number system.

Algorithm 5.1 (Extended hybrid-redundant addition):

Step 1: Add the equally weighted double-primed bits of the second component for the two operands to form a 1-deep sum, possibly left-extended to the next redundant position to preserve sign information.

Step 2: Using one binary FA cell per digit position, reduce the 3- or 4-deep WBS number composed of the two full components of the original operands, and the component produced by step 1, to a 2- or 3-deep WBS number. Depth of 4 may occur only in redundant positions.

Step 3: Add the equally weighted digits (where the leftmost position of each digit holds its only redundant binary digit) of the two components of the latter result, in parallel, with special treatment of the redundant positions. ■

We next demonstrate, in detail, the application of Algorithm 5.1 to addition with SDB hybrid representation. We also briefly examine the use of this algorithm for other variants. We show that steps 1 and 2 take constant time, while step 3, which needs intradigit carry propagation, can be performed in O(log d) time at best, where d is the longest distance between two redundant positions.

Without loss of generality we show the application of Algorithm 5.1 for radix-2^h periodic SDB hybrid-redundant operands, where each digit includes a full h-posibit primed component, extending from position 0 to $h-1$, and one inverted-negabit double-primed component in position h, overlapping with the least-significant primed posibit component of the next higher digit.

$$\begin{array}{ccccccccccccccccc}
 & a'_{15} & a'_{14} & a'_{13} & a'_{12} & a'_{11} & a'_{10} & a'_{9} & a'_{8} & a'_{7} & a'_{6} & a'_{5} & a'_{4} & a'_{3} & a'_{2} & a'_{1} & a'_{0} \\
A''_{16} & & & & A''_{12} & & & & A''_{8} & & & & A''_{4} & & & & \\
 & b'_{15} & b'_{14} & b'_{13} & b'_{12} & b'_{11} & b'_{10} & b'_{9} & b'_{8} & b'_{7} & b'_{6} & b'_{5} & b'_{4} & b'_{3} & b'_{2} & b'_{1} & b'_{0} \\
B''_{16} & & & & B''_{12} & & & & B''_{8} & & & & B''_{4} & & & & \\
\hline
 & a'_{15} & a'_{14} & a'_{13} & a'_{12} & a'_{11} & a'_{10} & a'_{9} & a'_{8} & a'_{7} & a'_{6} & a'_{5} & a'_{4} & a'_{3} & a'_{2} & a'_{1} & a'_{0} \\
 & b'_{15} & b'_{14} & b'_{13} & b'_{12} & b'_{11} & b'_{10} & b'_{9} & b'_{8} & b'_{7} & b'_{6} & b'_{5} & b'_{4} & b'_{3} & b'_{2} & b'_{1} & b'_{0} \\
t_{16} & t_{15} & t_{14} & t_{13} & t_{12} & t_{11} & t_{10} & t_{9} & t_{8} & t_{7} & t_{6} & t_{5} & t_{4} & & & & \\
T_{16} & & & & T_{12} & & & & T_{8} & & & & 1 & & & &
\end{array}$$

Fig. 5.5. Symbolic representation of step 1 in adding two SDB hybrid-redundant numbers.

Table 5.VI Combining of the double-primed components for SDB hybrid addition

A''_{ih}	B''_{ih}	Sum	$T_{(i+1)h}$	$t_{(i+1)h-1} \ldots t_{ih+1}$	t_{ih}
0	0	–2	0	1 . . . 1	0
0	1	–1	0	1 . . . 1	1
1	0	–1	0	1 . . . 1	1
1	1	0	1	0 . . . 0	0

Fig. 5.5 depicts step 1 of Algorithm 5.1 for 4-digit radix-16 SDB hybrid operands, where $T_{(i+1)h}$, $t_{(i+1)h-1} \ldots t_{ih}$ ($i = 1, 2, 3$, and $h = 4$), represent the sign extended 2's-complement sum of two inverted negabits in position ih. For uniformity in treating positions whose indices are multiples of 4, we have placed a 1 in position 4 as the code for an inverted negabit with arithmetic value 0.

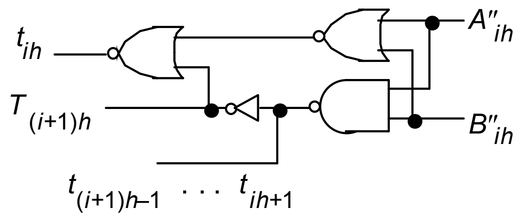


Fig. 5.6 Circuit for reducing the second components of Fig. 5.5

Table VI and Fig. 5.6 depict the truth table and logic implementation (actually a half adder) for deriving the 2's-complement sum. The result of applying step 2 on the 4-deep WBS number of Fig. 5.5 is shown as the 3-deep WBS number in Fig. 5.7. The first row of full-adders in Fig. 5.8 constitutes the required hardware, whose operation can start at the same time as that of the circuit of Fig. 5.6. Step 3 is performed by an (h – 1)-bit carry-propagate adder in the second row of Fig. 5.8. The full-adder in position ih receives two posibits and one inverted negabit and generates an inverted negabit sum along with a posibit carry. The posibit carry out of the full-adder in position ih – 1 (i.e., s'_{ih} in Fig. 5.8) is held in position ih and will not propagate beyond there. This bit, together with the inverted negabit sum S''_{ih} of the full-adder in position ih, form the SDB redundant digit of the result in the same position as that of the operands; hence the representational closure property.

$$
\begin{array}{ccccccccccccccccc}
 & w_{15} & w_{14} & w_{13} & w_{12} & w_{11} & w_{10} & w_{9} & w_{8} & w_{7} & w_{6} & w_{5} & w_{4} & w_{3} & w_{2} & w_{1} & w_{0} \\
c_{16} & c_{15} & c_{14} & c_{13} & c_{12} & c_{11} & c_{10} & c_{9} & c_{8} & c_{7} & c_{6} & c_{5} & c_{4} & c_{3} & c_{2} & c_{1} & \\
T_{16} & & & & T_{12} & & & & T_{8} & & & & 1 & & & & \\
\hdashline
s'_{16} & s'_{15} & s'_{14} & s'_{13} & s'_{12} & s'_{11} & s'_{10} & s'_{9} & s'_{8} & s'_{7} & s'_{6} & s'_{5} & s'_{4} & s'_{3} & s'_{2} & s'_{1} & s'_{0} \\
S''_{16} & & & & S''_{12} & & & & S''_{8} & & & & S''_{4} & & & &
\end{array}
$$

Fig. 5.7 Step 3 of the SDB hybrid addition.

The total delay of the adder above is equal to that of h + 1 full-adders, which is the same as that of our simpler implementation in Section 5.1, given that $h = d + 1$. Note that any carry acceleration method can be applied in a straightforward manner to reduce the delay due to h cascaded FAs within the second row in the design of Fig. 5.8.

An implementation of SDB hybrid redundancy is offered in [Phat01], where intradigit borrow (as well as carry) propagation and the look-back mechanism complicate the adder cells for nonredundant and redundant positions, respectively, and standard carry acceleration logic is not directly applicable.

The implementation above works equally well for BSD hybrid numbers, for it is the same as SDB hybrid except that the second component is right shifted one position. As for the SDC hybrid case, we can use the circuit in Fig. 5.6 to get a 2-bit sum of the double primed posibits (no extension is needed).

The remaining steps can be followed in Fig. 5.9. Due to the limited extension in step 1, some positions remain 2-deep. Therefore the corresponding FAs of the first row of Fig. 5.8 may be replaced by HAs. The SC hybrid representation can be handled similarly due to its resemblance to SDC hybrid.

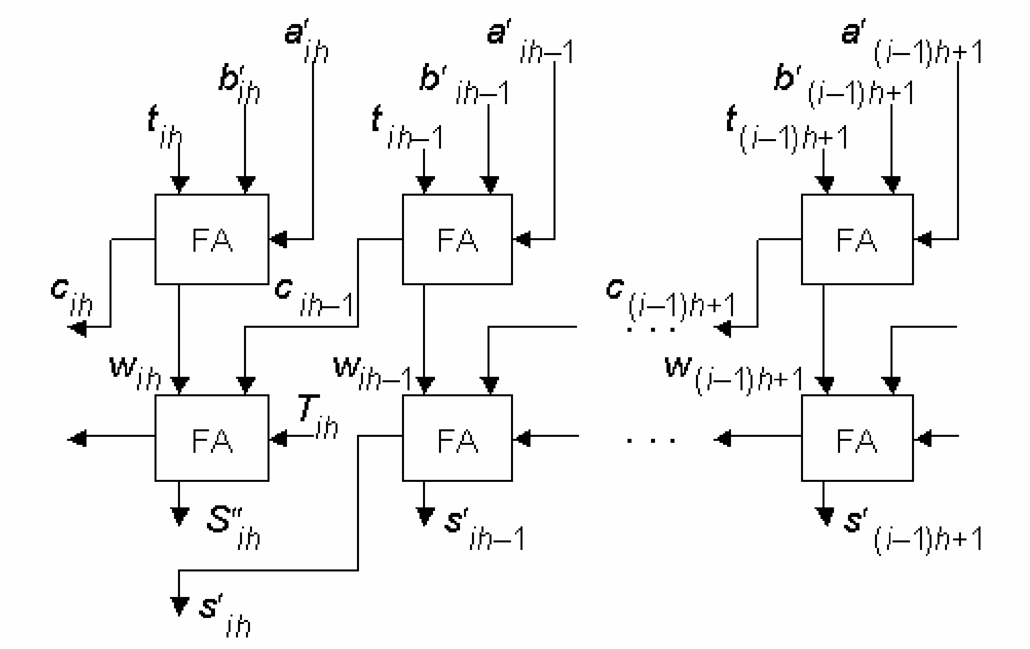


Fig. 5.8 SDB hybrid-redundant representationally closed adder. *T* and *t* come from Fig. 5.6

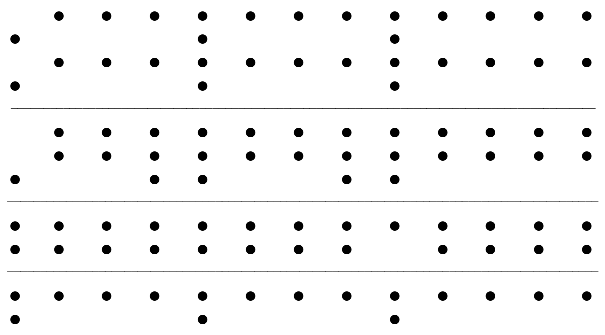


Fig. 5.9 SDC hybrid representationally closed addition.

For SBC hybrid (with double position redundant digit) and symmetric hybrid numbers, due to existence of negabits in nonredundant positions, step 1 of Algorithm 5.1 needs to be applied somewhat differently. Fig. 5.10 depicts the situation for symmetric hybrid numbers, where 0 (1) indicates a posibit (negabit) with constant value 0. In step 1, we make a 1-deep sum of the negabits as well as that of double-primed posibits in redundant positions. Moreover, the reduction to a 2-deep WBS number takes two steps. The generated bits in the leftmost column have been discarded in the final result. A collective nonzero value of those bits indicates an overflow/underflow.

The same scheme works for SBC hybrid case, for the encoding is the same except that the double-primed components have been left-shifted to the next redundant position. The latency is equal to that of *h* + 1 FAs and 1 HA. Given that the circuit of Fig. 5.6 is actually a half-adder, the complexity of the symmetric hybrid adder amounts to three FAs per posibit nonredundant position, two FAs plus two HAs per redundant position, and two FAs plus one HA per negabit nonredundant position. Recall that our uniform representationally unclosed adder of Section 5.1 had one FA per nonredundant position and two FAs per redundant position. The added complexity is the price paid for symmetry and representational closure. The delay penalty, however, is minimal, given that the total adder delay is increased only by that of a half-adder.

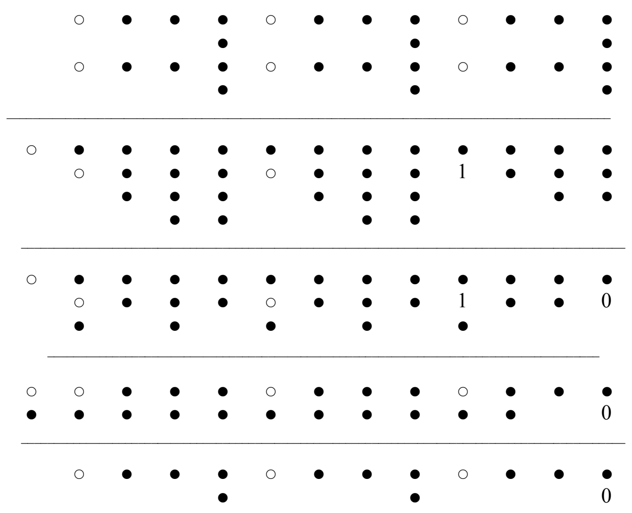


Fig. 5.10 Symmetric hybrid representationally closed addition.

5.4 Multiplication of Hybrid-Redundant Numbers

The first step in multiplication of two extended hybrid redundant numbers (or canonical WBS numbers) is to derive the partial product bit-matrix composed of posibits and negabits. Fig. 5.11 depicts the required gates in this step, for three different possible combinations of posibits and negabits, where upper (lower) case variables indicate negabits (posibits).

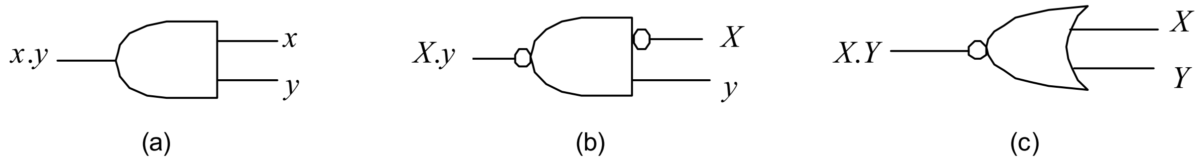


Fig. 5.11 Basic gates for derivation of the partial products.

We will show in Chapter 7 that all (ν; μ)-compressors receiving ν equally weighted posibits and negabits in position *i* produce μ posibits and negabits in positions *i* through $i + \mu - 1$, such that inputs and outputs have the same collective values. Here we show a similar result for the popular (4; 2) compressor. A conventional (4; 2) compressor receives 5 equally weighted bits in position *i*, (one of them normally being a carry from position $i - 1$), producing two equally weighted bits in position $i + 1$ and one bit in the same position *i* (Fig. 5.12a). The compression process is governed by the following equation [Kore02]: $x_1' + x_2' + x_3' + x_4' + x_5' = 2(c' + c'') + s'$

The arithmetic value $\alpha(X)$ of an inversely encoded negabit X can be expressed in terms of its logical value as $\alpha(X) = X - 1$. Replacing any of the posibits in the above equation by a negabit will add -1 to the left hand side of the equation, which should be compensated for by adding -1 to the right-hand side. The appearance of one, three, or five negabits on the left-hand side, as is depicted in Fig. 5.12, causes the same number of -1s to be added to the right-hand side. These -1s could be absorbed by the sum bit s', and zero, one, or two carry bits, respectively, thus turning to negabits with the same logical values. Two or four negabits on the left would similarly turn one or two of the carry bits to negabits, respectively. Note that usability of a conventional (4; 2) compressor to reduce any collection of 5 negabits and posibits is independent of the hardware implementation of the compressor.

Any partial product bit-matrix, can be reduced to a 2-deep WBS number, by using (4; 2) compressors, and also (3; 2) counters if needed. The resulting 2-deep WBS number can be reduced to a nonredundant 2's-complement number through carry acceleration circuits. It can also be converted to a desired WBS encoding (e.g., that of the input operands) through conversion process given in [Jabe02].

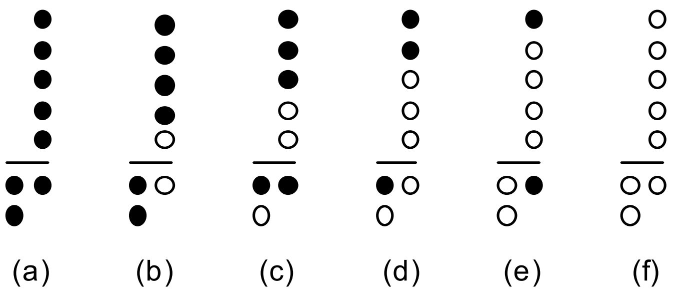


Fig. 5.12 Reduction of alternate collections of 5 negabits and posibits

5.5 Summary

In this Chapter, we have revisited the previously studied classes of hybrid-redundant numbers by viewing them as subclasses of weighted bit-set (WBS) redundant representations. We showed that the class of canonical WBS numbers covers all the variants of the hybrid-redundant numbers previously considered. Moreover the class of canonical WBS numbers with a single negabit in some positions represents a new variant of hybrid-redundant numbers, where arbitrary nonredundant positions may hold negabits; this is in contrast to standard hybrid redundancy which is restricted to containing only posibits in all nonredundant positions. We noted that this possibility allows for designing new variants of symmetric hybrid redundant numbers with arbitrary spacing of redundant positions, which have not been considered before. The new variants, and the flexibility in choosing the encodings for existing systems, allow for optimizations not previously possible.

We showed that inverted encoding of negabits leads to the use of conventional full-adders for the reduction of any set of three equally weighted posibits and negabits to two bits, one with the same weight and the other with double the weight. Using this fact, we provided the high-level design for a universal hybrid-redundant adder capable of adding two extended hybrid-redundant numbers (or canonical WBS numbers) with advantages over previous implementations of hybrid redundancy in terms of circuit regularity, possibility of using standard carry acceleration techniques, shorter critical-path delay, and lower complexity. With regard to the latter, 1 (2) full-adder(s) per nonredundant (redundant) position is required. We further explored representationally closed addition schemes, with additional advantage of greater reusability, for all variants of hybrid redundant numbers including the new symmetric variants. Finally we showed a new functionality of the popular (4; 2) compressors in reducing any collection of five equally weighted posibits and negabits, and used it in the high level design of a multiplier for extended hybrid redundant numbers.

Chapter 6 Extended-Hybrid Redundant Number Systems

Redundant number systems enable us to perform digit-parallel addition with a small, constant latency, which is independent of operand widths [Aviz61], [Gonz00]. The redundancy ρ of a digit set $[-\alpha, \beta]$ is defined as the difference between the number of digit values (i.e., $\alpha + \beta + 1$) and the radix r of the number system. The higher the redundancy index, the greater the number of bits needed to represent each digit, and the longer the potential delays in digit-parallel arithmetic. In many cases, a redundancy index of $\rho = 2$ is adequate, and one never needs to go beyond $\rho = 3$, for carry-free addition in radices higher than 2 [Parh90]. However, proper choice of the redundancy index ρ, coupled with suitable encoding of the resulting digit set, may allow for a more efficient (faster and/or more compact) VLSI implementation. Such variations in redundancy indices and associated digit-set encodings is the main focus of this chapter. Much of what we present deals, directly or indirectly, with facilitating area-time tradeoffs in the VLSI implementation of arithmetic operations on redundant operands.

Stored-transfer representations [Jabe01], weighted bit-set encodings for digit sets [Jabe02], and representation paradigms of high-radix signed-digit number systems [Jabe03] are all motivated by area-time trade-off concerns, improvement in the representation coverage, speed of arithmetic, and/or regularity in VLSI implementation. Similarly, hybrid-redundant number systems introduced in [Phat94], and extended in [Phat01], provide a framework for the efficient design and implementation of digit-parallel addition for a class of redundant number systems. Briefly, a hybrid-redundant number is composed mostly of normal, positively-weighted bits (posibits), with some radix-2 positions holding redundant digits. Unfortunately, the design and implementation of redundant arithmetic based on the original notion of hybrid redundancy engenders some limitations such as the following:

- Considerable difference in the range of positive and negative numbers, leading to inefficiencies in the implementation of subtraction.
- Inapplicability of standard carry acceleration methods, and the associated highly optimized circuits, due to the use of nonstandard adder cells.
- Inability to faithfully cover, as a representation paradigm, almost all symmetric digit sets as well as many other useful digit sets.

To circumvent these problems, which are more fully explained in Section 6.2, we reformulate and extend the hybrid redundancy, within the framework of weighted bit-set encodings, in Section 6.3.

Our quest for more efficient and VLSI-friendly carry-free addition schemes for hybrid-redundant numbers leads us to a scheme for the encoding of negabits (i.e., negatively weighted bits) in Section 6.4, where we explore different functionalities for standard full-adders in the summation of any collection of three negabits and posibits. This leads to the design of efficient adder cells for both nonredundant and redundant positions in a hybrid-redundant representation. Section 6.5 demonstrates the power of extended hybrid redundancy scheme in deriving symmetric hybrid-redundant number systems with arbitrary spacing of redundant positions. This is followed by implementation details of an efficient and regular adder/subtractor for symmetric operands, where a representationally closed version of the adder is also provided. Finally, Section 6.6 provides a summary of this chapter.

6.1. Limitations of Ordinary Hybrid Redundancy

A hybrid-redundant number system has k radix-2 positions numbered 0 to $k - 1$, from the least to the most significant position, respectively. Each position may be nonredundant, holding a posibit (i.e., a normal bit in [0, 1]), or redundant with a digit in $[-n, p]$, where $n, p \geq 0$. The digit in position i ($0 \leq i < k$) has the weight 2^i. Some practical values for n and p have been reproduced in Table 6.I from [Phat01]. At the extreme of no redundant position, a hybrid-redundant number system represents unsigned binary integers. Efficient adder cells for computing $x_i + y_i + t_i$ are offered in [Phat94] and [Phat01], where x_i and y_i, the digits in position i of the two operands, belong to the same digit set from Table 6.I, and t_i, the transfer digit coming from the right context, is restricted to [–1, 1]. The latter has been made possible through equal-weight grouping and look-back mechanisms [Phat01]. For other possible digit sets and arbitrary pairing of digit sets in position i, the outgoing transfer t_{i+1} from position i may assume larger values (e.g., $t_{i+1} \in [-2, 2]$), leading to more complex adder cells.

A hybrid-redundant number system is periodic if the number of posibit place-holders between two redundant positions remains constant and digit sets associated to redundant positions are the same, with the period h being one more than the constant distance. Such periodic hybrid-redundant systems can be viewed as efficient encodings for special classes of GSD representations. However, there exist useful GSD number systems, periodic by definition, that cannot be represented via ordinary hybrid redundancy. For example, the radix-10 GSD representation with digits in [−9, 9] has no counterpart in hybrid redundancy. We will show later (Theorem 6.1) that the subclass of symmetric ordinary hybrid-redundant representations is very limited and that efficient implementations exist only for fully redundant binary signed-digit (BSD) and minimally redundant radix-4 number systems, both of which had been studied and used prior to, and in contexts other than, hybrid redundancy.

Table 6.I. Redundant digit sets in the hybrid redundancy schemes of [Phat01]

Type of redundant digit	Digit set: $[-n, p]$
Binary signed-digit (BSD)	[–1, 1]
Stored double borrow (SDB)	[–2, 1]
Stored borrow or carry (SBC)	[–1, 2]
Stored carry (SC)	[0, 2]
Stored double carry (SDC)	[0, 3]

Definition 6.1 (Right-side and left-side periodic hybrid redundancy): In a hybrid-redundant representation of period h, the position index for the redundant binary digit in $[-n, p]$ is either 0 or $h - 1$ (mod h). We refer to the former (latter) as right-side (left-side) hybrid redundancy. Taking each period of the hybrid-redundant representation as a radix-2^h GSD position, the corresponding digit set of a right-side (left-side) redundant representation is $[-n, 2^h + p - 2]$ ($[-2^{h-1}n, 2^{h-1}p + 2^{h-1} - 1]$). ■

Lemma 6.1 (Symmetry of digit sets associated with periodic hybrid-redundant representations): Left-side hybrid-redundant digit sets cannot be symmetric except for $h = 1$, while symmetric right-side hybrid redundancy is possible for all $h \geq 1$.

Proof: For the left-side hybrid redundant digit set $[-2^{h-1}n, 2^{h-1}p + 2^{h-1} - 1]$ to be symmetric, we must have $2^{h-1}n = 2^{h-1}p + 2^{h-1} - 1$ or $n = p + 1 - 1/2^{h-1}$. It is obvious that the latter equation has integer solutions for n and p only if $h = 1$. The corresponding equation for right-side hybrid redundancy is $n = p + 2^h - 2$, which has a solution for any $h \geq 1$. ■

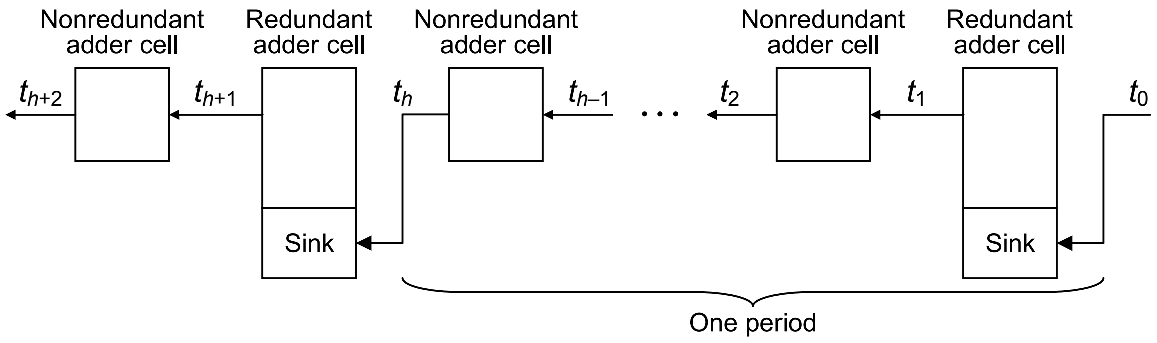


Fig. 6.1. Hybrid -redundant adder with right-side redundant digit positions.

In a hybrid-redundant adder, as described in [Phat94] and [Phat01], the adder cell of a redundant position does not propagate the incoming transfer (e.g., carry or borrow). Transfers generated by redundant or nonredundant positions may propagate up to the next redundant position, where they sink. This process is depicted in Fig. 6.1, where the larger boxes representing adder cells in redundant positions are intended to reflect the greater complexity of those cells relative to adder cells in nonredundant positions.

To keep the complexity of the adder cells in check, the range of transfer values in [Phat01] has been chosen to be $[-1, 1]$, which is as narrow as possible. This is achieved through the constraint $2 \leq n + p \leq 3$ (see Table 6.I), along with two techniques. One technique is equal-weight grouping, which effectively leads to encoding a 4-valued redundant binary digit (i.e., $n + p = 3$) as a radix-4 digit, where the doubly weighted bit of a redundant digit, together with the posibit in the next higher position, rise to a new redundant position with $n + p = 2$. The other technique is the lookback mechanism, where the value of the transfer generated by the adder cell of a redundant position is made dependent on the values of the posibits in the previous nonredundant positions. This dependency guarantees that the redundant position will be able to absorb an incoming carry or borrow from the right context. The constraint $n + p = 2$ for redundant positions, enforced by equal-weight grouping, when applied to the result in Lemma 6.1, leads to severe restrictions on designing symmetric hybrid-redundant digit sets, as explained by the following lemma.

Lemma 6.2 (Restricted symmetry in ordinary hybrid redundancy): There are only two possible symmetric ordinary hybrid-redundant digit sets meeting the constraint $n + p = 2$.

Proof: According to Lemma 6.1, for left-side hybrid redundancy, a symmetric digit set is possible only if $h = 1$. This, when combined with the constraint $n + p = 2$, yields $n = p = 1$ and leads to the BSD representation. For right-side hybrid redundancy, Lemma 6.1 dictates $n = p + 2^h - 2$. The latter, combined with the constraint $n + p = 2$, leads to $n = 2^{h-1}$ and $p = 2 - 2^{h-1}$. Because $p \geq 0$, we must have $p = n = 1$ or $p = 0$ with $n = 2$. The former solution again represents the BSD number system, while the latter leads to the minimally redundant radix-4 representation. ■

The adder cells of [Phat94] and [Phat01] for redundant and nonredundant positions do not use standard full-adders as building blocks. This design decision is justified, even for nonredundant positions, by the fact that redundant positions may generate borrows as well as carries, which must then ripple through nonredundant positions. In Section 6.5, we will show that the addition of posibits in a nonredundant position and the incoming borrow or carry can indeed be delegated to a common full-adder. The benefits of such a design are the use of highly optimized standard full-adder cells and the possibility of carry acceleration within multiple nonredundant positions; neither of these applies to the adder cells of [Phat94] and [Phat01].

6.2. WBS Encodings and Hybrid Redundancy

WBS encoding, as introduced in [Jab02], is capable of representing any GSD digit set, including those of hybrid-redundant systems. Furthermore, aperiodic hybrid-redundant number systems, not covered by the GSD paradigm, can also be represented by WBS encoding. Canonical WBS encodings, where each redundant radix-2 digit set is 3-valued and a proper subset of [–2, 2], are particularly useful for efficient carry-free addition.

Definition 6.2 (Canonical WBS encoding): The digit set in each radix-2 position of a canonical WBS encoding is [–2, 0], [–1, 0], [–1, 1], [0, 1], or [0, 2], which is representable by two equally weighted negabits, one negabit, a negabit and a posibit, one posibit, or two posibits, respectively. The multiplicity (i.e., the number of bits) of each position of a canonical WBS encoding is either 1 or 2. When multiplicities for all radix-2 positions are equal to 1, the encoding is called 1-deep; otherwise it is 2-deep. ■

Example 6.1 (Canonical WBS encoding): Figure 6.2 depicts the dot-notation representation of a canonical 2-deep WBS encoding of an 8-position redundant digit corresponding to the digit set [–73, 227], where ● (○) stands for a posibit (negabit). ■

```
● ● ○ ● ● ● ● ●
    ○   ○ ●   ○
```

Fig. 6.2. Dot -notation representation of a canonical WBS encoding.

Lemma 6.3 (1-deep WBS encoding of $[-n, p]$): A redundant radix-2 digit set $[-n, p]$ can be faithfully represented by a 1-deep WBS encoding iff $n + p = 2^g - 1$ for some $g > 0$.

Proof: Assume that there exist a g-position 1-deep WBS encoding representing exactly $[-n, p]$. It can be shown that for such an encoding, n (p) is the value of a g-bit binary number with 1s where the WBS encoding holds a negabit (posibit) and 0s elsewhere. That $n + p = 2^g - 1$ follows immediately. For the sufficiency part, we construct a WBS encoding Ω with a negabit (posibit) in any position where the unsigned binary representation of n (p) has a 1. Because $n + p = 2^g - 1$, each constituent bit of Ω has its unique position, hence a 1-deep WBS encoding. Furthermore, the most negative (positive) value in Ω results from assigning 1s (0s) to negabits and 0s (1s) to posibits. The latter assignment establishes the range as being $[-n, p]$. ■

Lemma 6.4 (Sparse 2-deep WBS encoding of a digit set): A redundant radix-2 digit set $[-n, p]$ can be faithfully represented by a 2-deep WBS encoding, where the second tier of dots consists of a single bit in position j, iff $n + p + 1 = 2^g + 2^j$ for some $g > 0$ and $0 \leq j < g$.

Proof: The digit set $[-n, p]$ of a 2-deep WBS encoding, as described in this Lemma's statement, may be decomposed into $[-n', p'] + \{0, \pm 2^j\}$, where the first component represents the primary digit set corresponding to the first tier of dots and $\pm$ relates to the second component holding a posibit or a negabit. The primary component is actually a 1-deep WBS encoding of $[-n', p']$, leading (by Lemma 6.3) to $n' + p' = 2^g - 1$. The latter equality, along with the decomposition above, lead to $n + p + 1 = n' + p' + 1 + 2^j = 2^g + 2^j$. ■

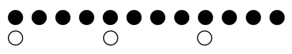
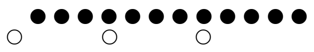
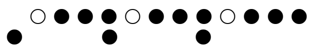
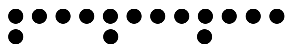
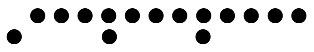
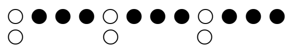
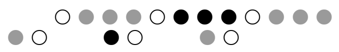
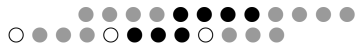
Corollary 6.1 (2-deep WBS encoding with 1-bit right-side second component): The redundant radix-2 digit set $[-n, p]$ can be faithfully represented by a 2-deep WBS encoding, with a 1-bit second component in position 0, iff $n + p = 2^g$ for some $g > 0$. ■

Theorem 6.1 (Canonical WBS encoding of hybrid-redundant numbers): The interval of integers represented by a k-position hybrid-redundant number system is equivalently and faithfully representable by a k-position canonical WBS encoding iff for every redundant position that represents $[-n, p]$, the following holds, where d is the distance to the next higher redundant position: $n + p = 2^g - 1$ for some g in the range $0 < g \leq d + 2$, or $n + p = 2^g$ for some g satisfying $0 < g \leq d + 1$.

Proof: We construct a k-position canonical WBS encoding Ω with a single posibit in each nonredundant position of the given k-position hybrid-redundant system, and augment it with additional bits for faithful representation of all the redundant digits. To ensure that Ω is 2-deep, the maximum number of bits available for representing a redundant digit is $d + 2$; one in each of the d nonredundant positions to its immediate left and 2 in the redundant position itself. Note that to represent the full required range, the additional bits must be able to represent all integers in the interval $[-n, p]$. If $n + p = 2^g - 1$, then a 1-deep g-position WBS encoding representing $[-n, p]$ exists by Lemma 6.2. It follows that $0 < g \leq d + 2$ must hold for a 2-deep overall encoding. Otherwise, given the 2-deep constraint, the only other possibility is 2-deep g-bit WBS encoding of $[-n, p]$ with $g \leq d + 1$ and a 1-bit right-side second component, which by Corollary 6.1 requires $n + p = 2^g$. ■

Example 6.2 (WBS encoding for hybrid-redundant number systems): Table 6.II depicts WBS encodings for some hybrid-redundant number systems, where the first five entries coincide with those studied in [Phat01]. The posibits in the most and least significant digit positions of the last two entries are shown in gray for better visualization of the periodic structure. ■

Table 6.II Some hybrid-redundant number systems.

Composition (digit pattern)	Conditions of Theorem 6.1	WBS encoding with 3 digits
1 BSD [–1, 1], h – 1 binary	$n + p = 2, g = 1 \leq d + 1 = h = 4$	
1 SDB [–2, 1], h – 1 binary	$n + p = 3, g = 2 \leq d + 2 = 5$	
1 SBC [–1, 2], h – 1 binary	$n + p = 3, g = 2 \leq d + 2 = 5$	
1 SC [0, 2], h – 1 binary	$n + p = 2, g = 1 \leq d + 1 = 4$	
1 SDC [0, 3], h – 1 binary	$n + p = 3, g = 2 \leq d + 2 = 5$	
1 in [–2, 0], h – 1 binary	$n + p = 2, g = 1 \leq d + 1 = 4$	
1 in [–3, 4] , h – 1 binary	$n + p = 7, g = 3 \leq d + 2 = 5$	
1 in [–8, 8] , h – 1 binary	$n + p = 16, g = 4 \leq d + 1 = 4$	

Theorem 6.1 and Table 6.II show that any of the five ordinary hybrid-redundant number systems efficiently implemented in [Phat01] is representable by a 2-deep WBS encoding. These canonical WBS representations may be regarded as hybrid-redundant number systems with redundant positions restricted by the constraint $n + p \leq 3$. Thus, the redundant binary digit sets of canonical WBS encodings are the same as those studied in [Phat01], with the addition of [–2, 0]. Other hybrid-redundant number systems with redundant positions of wider range (e.g., those in the last two entries of Table 6.II), when represented by canonical WBS encoding, can be alternatively regarded as having other redundant digit sets with $n + p = 2$. Therefore, we can design adders for any hybrid-redundant system meeting the conditions of Theorem 6.1 based on the adder cells of [Phat01] along with a similar adder cell for the digit set [–2, 0].

An alternative approach for addition of canonical WBS numbers is offered in [Jabθ2], where the negabits of each 2-deep operand are temporarily viewed as being posibits. The value represented by such an all-posibit interpretation is biased by a nonpositive constant relative to the value originally represented. Addition is, thus, reduced to standard multioperand addition of binary numbers, where two of the operands are the bias constants. The advantage here is that the adder is composed of only standard full-adders. Taking advantage of our new result [Jabe05a] that standard full-adder cells are capable of reducing any collection of three posibits and negabits, we provide a direct implementation (i.e., without pre-addition conversion to all-posibit operands) of adder cells for redundant and nonredundant positions in Section 6.5.

Ordinary hybrid redundancy, as defined in [Phat94] and extended in [Phat01], does not allow single negabits in nonredundant positions. For example two’s complement numbers, with a negabit in their most significant bit position, may not be viewed as a special case of ordinary hybrid redundancy.

The third entry of Table 6.II, with a single negabit in its WBS encoding may be thought of as a counter example to the latter claim. However, one must note that in the implementations offered in [Phat01], this single negabit together with a posibit in the next higher position forms an SBC digit in the same (redundant) position as the negabit, and is thus not considered and manipulated by itself as a nonredundant binary digit.

Since a negabit represents the nonredundant radix-2 digit set [–1, 0], we are motivated to extend the hybrid redundancy scheme to allow for negabits in nonredundant positions. This implies that, in designing the required adder cells, the negabit would be considered by itself as a nonredundant binary digit and not as part of a redundant digit.

Definition 6.3 (Extended hybrid redundancy): A k-position *extended hybrid-redundant number system* has k radix-2 digits in positions 0 to k – 1, weighted 2^0 to 2^{k-1}, respectively. Each digit is from a contiguous redundant or nonredundant digit set with 0 as a member. Graphically a redundant position is shown as ®, or by a collection of two or more posibits (●) and negabits (○); a nonredundant position contains exactly one posibit or one negabit. ■

Example 6.3 (Extended hybrid-redundant number system): In dot notation, the structure of an extended hybrid-redundant number system may appear, for example, as ® ● ○ ® ○ ® ○ ● ● ® ●, where the positions marked ® may use any redundant digit set with 0 as a member. Any WBS encoding of redundant digits may be used. Examples include the five digit sets in Table 6.I, whose single- and double-position WBS encodings are shown in Fig. 6.3. ■

For redundant digit sets meeting the conditions of Theorem 6.1, leading to 2-deep canonical WBS representations, implementation of hybrid-redundant addition is tantamount to reducing a 4-deep WBS encoding to an equivalent 2-deep encoding. This is taken up in Section 6.4.

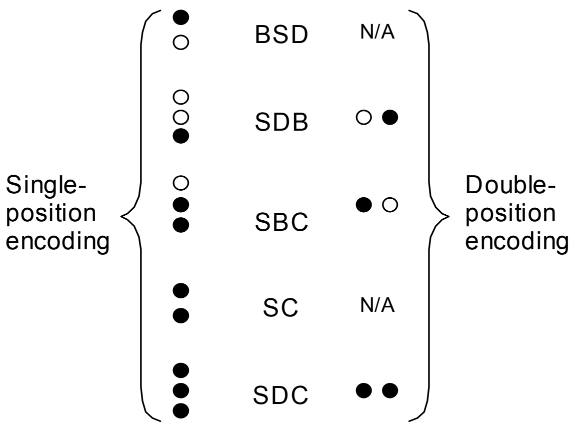


Fig. 6.3. Single/double-position WBS representations

6.3 Inverted Encoding of Negabits

The relative complexity of the adder cells proposed in [Phat94] and [Phat01] is mainly due to carry and borrow propagation within the same circuit. It is well known that inverting all three inputs of a full-adder will result in inverted sum and carry. This hints at using a standard full-adder for addition of three equally weighted negabits, and possibly extending it to any collection of three posibits and negabits. This intuition is also supported by the value-preserving transformations depicted in Fig. 6.4, which shows how any collection of three posibits and negabits with the same weight may be replaced by a bit with the same weight and a doubly weighted one, without affecting the representable range.

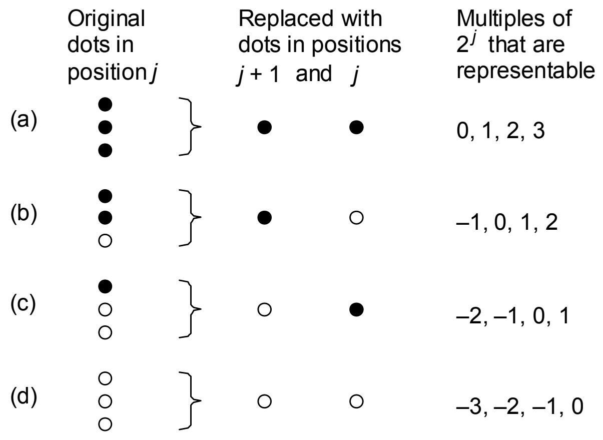

Original dots in position j
Replaced with dots in positions $j+1$ and j
Multiples of 2^j that are representable
(a)
0, 1, 2, 3
(b)
–1, 0, 1, 2
(c)
–2, –1, 0, 1
(d)
–3, –2, –1, 0

Fig. 6.4. Replacement of three equally weighted posibits and negabits.

It is easily verified, by examining the four cases depicted in Fig. 6.4, that a standard full-adder receiving a combination of posibits and negabits with conventional encoding of negabits does not always produce the correct sum and carry values. However, inverted encoding of negabits (representing –1 as 0 and 0 as 1) does allow the use of standard full-adders as universal adder cells for any collection of three negabits and posibits. The concept is more formally defined and justified below.

Definition 6.4 (Inverted encoding of negabits): Inverted encoding of negabits is exactly the opposite of the conventional encoding, as used in, for example, in the most significant position of standard two's complement representation of binary integers. The lower (higher) value of a negabit, that is, –1 (0), is inversely encoded as 0 (1). We use uppercase (lowercase) letters to designate the logical value of a negabit (posibit). Then the arithmetic value of a negabit X (a posibit x) would be $X - 1$ (x). ■

Figure 6.5 depicts the universal functionality of a standard full-adder in (3; 2) compression of any equally weighted collection of three negabits and posibits. For a justification, let x_1, x_2, and x_3 denote the logical values of three equally weighted posibits and inversely encoded negabits. Recall from Definition 6.4 that the arithmetic value of a posibit with logical value x is x but that of a negabit with logical value Y is $Y - 1$. Given c and s as the carry and sum outputs of a standard full-adder receiving x_1, x_2, and x_3 as inputs, the justification of universality of full adders is summarized in Table 6.III, where negabits are shown in uppercase.

Similarly, one could use half-adders to convert any set of two equally weighted posibits and negabits to an arithmetically equivalent 1-deep two-bit result. This functionality of half-adders is justified by the contents of Table 6.IV. It has been shown [Jabθ5 a] that conventional compressors present similar functionality in reducing more complex collections of posibits and negabits.

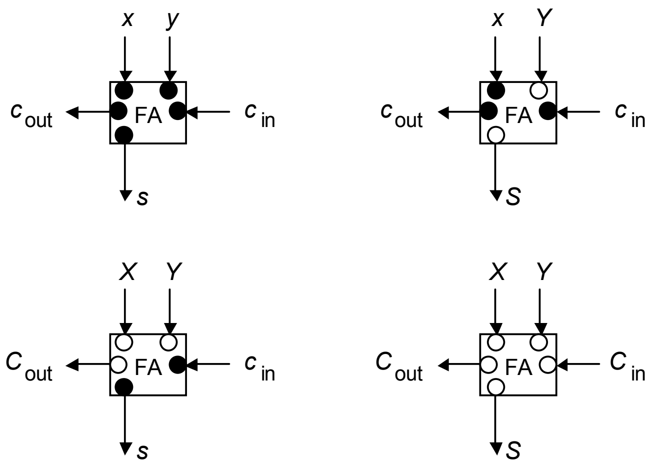


Fig. 6.5. Univ ersality of a binary full-adder for adding equally weighted posibits (shown as lowercase variables) and negabits (uppercase).

Table 6.III. Justifying the universality of a full-adder as shown in Fig. 6.5.

Input-collection	**Arithmetic equivalence**
Three posibits x_1, x_2, x_3	$x_1 + x_2 + x_3 = 2c + s$
Two posibits x_1, x_2, and one negabit X_3	$x_1 + x_2 + (X_3 - 1) = 2c + (S - 1)$
Two negabits X_1, X_2, and one posibit x_3	$(X_1 - 1) + (X_2 - 1) + x_3 = 2(C - 1) + s$
Three negabits X_1, X_2, X_3	$(X_1 - 1) + (X_2 - 1) + (X_3 - 1) = 2(C - 1) + (S - 1)$

Table 6.IV. Half-adder functionality with posibit and negabits as inputs.

Logical input	**Sum for the three cases**			**Logical output**
● ● ○ ● ○ ○				●● ●○ ○●
0 0	0	−1	−2	00
0 1	1	0	−1	01
1 0	1	0	−1	01
1 1	2	1	0	10

Addition of two canonical WBS operands is performed by conceptually copying the bits of the 2-deep operands in the bit placeholders of a 4-deep WBS representation. This is then followed by conversion (or reduction) to canonical WBS representation. In fact, only redundant WBS positions produce 4-deep results, with nonredundant positions yielding 2-deep results. But if the canonical WBS encodings of the two operands are not exactly alike, a nonredundant position of one may align with a redundant position of the other, thus leading to 3-deep positions as well. There is never a 1-deep position. Note that addition, as formulated above, can be viewed as a special case of digit-set conversion [Korn99]. The reduction of a 4-deep WBS number with no empty position can be done in two steps:

1. Reduce the 4-deep result to a 3-deep one, using full- and half-adders as appropriate.
2. Use a chain of full-adders for carry-propagate addition starting at an intermediate 3-deep position (or position 0), followed by a full-adder chain for 2-deep positions up to, but not including, the next 3-deep position. The carry-out of the full-adder for the last 2-deep position in a chain will stop at the following 3-deep position, where it joins the sum bit generated in that position. These two bits compose a redundant position *i* of the result.

When the number of bits in the like positions of the two operands are the same (i.e., they have the same redundancy pattern), the 4-deep intermediate sum contains 2- and 4-deep positions, and there will be no 1- or 3-deep positions. Then it may be desired to have the final result with the same redundancy pattern as that of the operands. Based on whether the redundancy pattern is to be preserved, one of the following may be applied as the reduction step:

- **Preserved redundancy pattern**: Reduce every 2-, 3- (if applicable), and 4-deep position to at most 2-, 2-, and 3-deep positions, respectively, as an intermediate step by using a half-adder (full-adder) for each 2- (3-, 4-) deep position. In case of identical redundancy patterns for the two operands, where no 3-deep position exists at the outset, it is easy to see that after the second step above, redundant positions in the result are the same as that of the operands. The required universal reduction cells (i.e., independent of input and output polarities) for arbitrary positions *i* (4-deep), *j* (3-deep) and *k* (2-deep) are shown in the first row of Figs. 6.6a, 6.6b, and 6.6c, respectively, where the primed and double primed variables reflect the depth 2 of the operands and non-primed variables denote intermediate results. The second row full-adders perform the second step above.
- **Shifted redundancy pattern**: leave the 2-deep positions of the 4-deep representation intact by removing the half-adder in Fig. 6.6c, but reduce 3- and 4-deep positions as above. Therefore, in case of identical redundancy patterns of the two operands, positions related to nonredundant positions of the operands immediately to the left of redundant positions will be 3-deep after the first step and 2-deep (i.e., redundant) after the second step; hence shifting in the redundancy pattern. The simplified adder cells are depicted in Fig. 6.7.

The approach that preserves the redundancy pattern does not necessarily lead to representational closure, because the latter requires not only a match in the redundancy patterns of the operands and the result but also identical polarity combinations for like positions. Where the polarity sets match in case of a shifted redundancy pattern, we have a representationally shifted result. The adder cells provided in [Phat01] for double-position redundant digits (i.e., SDB, SDC, and SBC) lead to representationally shifted outputs, which may be a desirable result in some applications.

Often, however, one needs a result with exactly the same encoding as that of the operands. Such representationally closed arithmetic enhances regularity and reusability of arithmetic cells in VLSI design. Although the adder cells of Fig. 6.6, when applied to operands with the same encodings, do preserve the bit multiplicity in each position (i.e., they preserve the redundancy pattern), they do not possess the representational closure property. In Section 6.5, we use the cells of Fig. 6.7 to design a VLSI-friendly carry-free adder for a symmetric extended hybrid redundant adder, with and without the representational closure property.

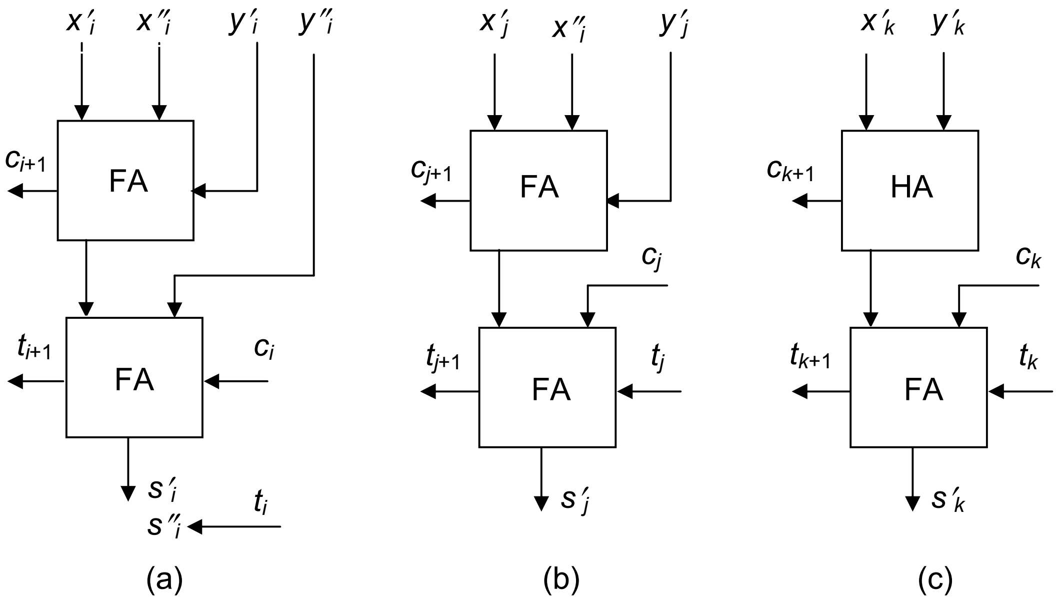


Fig. 6.6. Universal reduction cells for 4-, 3-, and 2-deep positions.

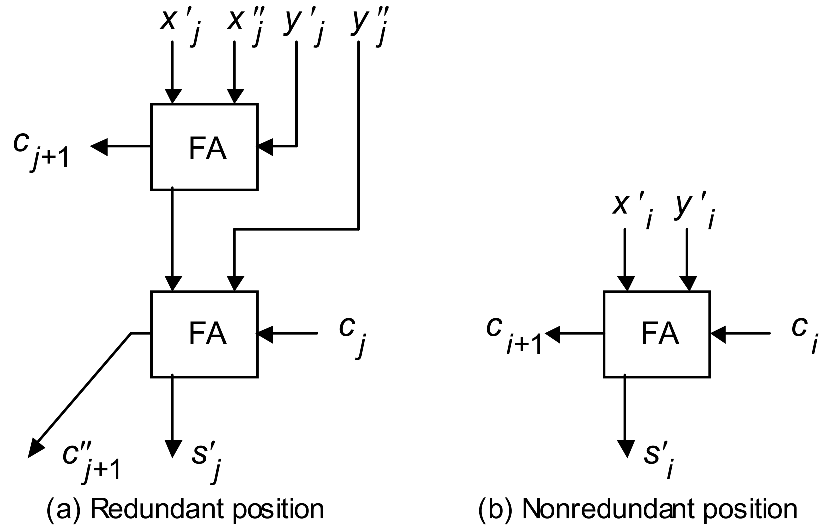


Fig. 6.7. Adder cells leading to shifted redundancy pattern relative to those of the operands.

6.4. Symmetric Extended Hybrid Redundancy

Recalling our discussion in Section 6.2, variants of symmetric ordinary hybrid redundancy are:

- **Left side redundant position:** By Lemma 6.1, this is possible only for $h = 1$ and leads to fully redundant number systems, where all positions hold redundant digits (e.g., BSD for 2-deep encodings).
- **Right-side redundant position:** The redundant digit must be in $[-(p+2^h-2), p+2^h-2]$ for all $h > 0$, where p is the maximum positive value which can be represented by the right-side redundant position (see Lemma 6.1). A WBS encoding for such a digit set would have at least $2^h - 2$ negabits in its redundant position. This means that the representation depth in redundant positions grows exponentially with the distance between redundant positions (i.e., $h-$ 1).

The most important characteristic of ordinary hybrid redundancy is the design flexibility in allowing an arbitrary number of nonredundant positions between redundant positions for area-time trade-off, as it is this number that defines the area requirement and the associated latency for the design. With exponential growth of area for the redundant positions when symmetry is a requirement, any attempt to increase h would be ineffective as an area-time trade-off measure. For example for $h = 3$, corresponding to a rather short distance of 2 between redundant positions, the encoding depth of redundant positions will be $p + 6$ (at least 6). Converting such a deep WBS encoding to a 2-deep (canonical) encoding reduces the number of nonredundant positions, which is counterproductive as regards to the main advantage of hybrid redundancy.

Example 6.4 (Deep symmetric hybrid redundancy): Figure 6.8 depicts the WBS encoding of a 6-deep symmetric hybrid redundant number system, and its equivalent canonical WBS encoding. Each radix-8 digit belongs to [–6, 6]. ■

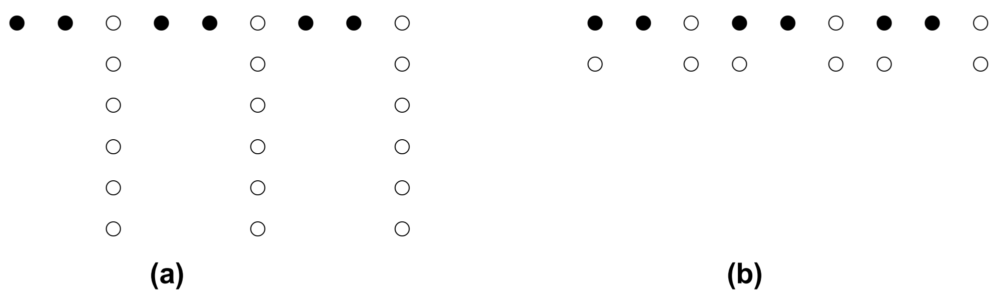


Fig. 6.8. A deep (a), and an equivalent canonical WBS encoding (b) for a symmetric ordinary hybrid redundant number system.

Example 6.4, as an evidence of the result of Lemma 6.1, shows that ordinary 2-deep hybrid redundancy [Phat94], [Phat01] provides for only two different symmetric digit sets; BSD and minimally redundant radix-4 digit set. This intrinsic restriction does not allow for arbitrary spacing of redundant positions in symmetric number systems. This result is formally stated in the following theorem.

Theorem 6.2 (Restricted spacing in symmetric ordinary hybrid-redundant representations): The maximum spacing between redundant positions in 2-deep symmetric ordinary hybrid-redundant number systems is 1.

Proof: Because negabits are disallowed in nonredundant positions, they must be present in redundant positions in a way to completely counterbalance the posibits in nonredundant positions. Only two arrangements accomplish this. With no nonredundant position, the only 2-deep symmetric redundant digit is a binary signed digit; hence the BSD number system is the only symmetric ordinary hybrid-redundant system with a period of 1 (spacing of 0). The contribution of a nonredundant posibit in position i to the positive range of the represented values is 2^i. Given the maximum depth of 2, and absence of negabits in nonredundant positions, the only possible compensation in the negative range is the use of two negabits in position $i - 1$. This observation leads to the only other possibility for a symmetric digit set with 2-deep representation: single binary positions alternating with redundant positions containing two negabits. This representation with period of 2 (spacing of 1) corresponds to the minimally redundant radix-4 signed digit number system. ■

Theorem 6.2 establishes that ordinary hybrid-redundant representations are mostly asymmetric, thus essentially denying designers the flexibility of spacing variations to trade off speed for economy (smaller VLSI area) in cases where symmetry is desired.

To reduce the depth of a high-radix symmetric ordinary hybrid redundant representation, it is possible to use more than one position for representation of the redundant binary digit set, as was suggested by the equal weight grouping in [Phat01].

Example 6.5 (Shallow encoding of symmetric hybrid redundancy): Figure 6.9 depicts a 10-deep (a), and a 3-deep (b) equivalent WBS encoding of a radix-8 hybrid-redundant number system with the digit set [–8, 8], where ® stands for an equally weighted collection of 2 posibits and 8 negabits. ■

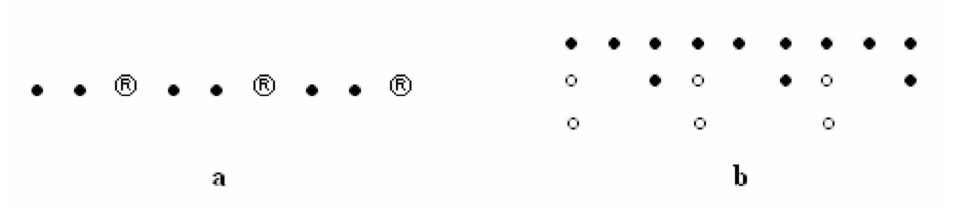


Fig. 6.9. Equivalent encodings of a hybrid -redundant number system.

The symmetric ordinary hybrid-redundant number system of Fig. 6.9b is not a 2-deep WBS encoding; it is thus unsuitable for the efficient universal addition scheme based on the adder cells of Fig. 6.7. The process of deriving its equivalent canonical WBS encoding, through transformations of Fig. 6.4, leaves a single negabit in each of the originally redundant positions with two posibits. The canonical WBS encoding thus derived no longer represents an ordinary hybrid-redundant number system (Fig. 6.10). This is indeed consistent with Theorem 6.2, and suggests a general method for constructing a 2-deep WBS encoding to represent a given symmetric range [$-\alpha$, α]. We begin with a one-position WBS encoding with α posibits, and α negabits, and repeat the transformations of Fig. 6.4, until no other transformation step is possible. A formal correctness proof for this method may be found elsewhere [Jabe05a].

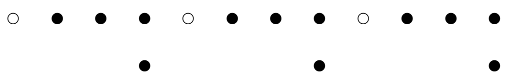


Fig. 6.10. A canonical WBS encoding of an exten ded hybrid-redundant number system with digit set [–8, 8]

Theorem 6.2 and the latter construction of 2-deep symmetric encodings reinforce the superiority of extended hybrid redundancy over ordinary hybrid redundancy in designing useful 2-deep (i.e., low redundancy in the WBS context) symmetric hybrid-redundant number systems having arbitrary spacing, with the possibility of using the universal addition scheme, where the adder cells of Fig. 6.7 are the only cells needed.

Numbers with arbitrary digit sets can be added digitwise to produce a sum with a digit set whose range is the sum of the ranges of the operand digits. This wider digit set can be kept intact and the result used as an operand in further arithmetic operations. It is also possible to convert the wider digit set to another, more convenient, one for further processing. Often, however, it is required to obtain results with the same digit set as inputs [Korn99]. Such representationally closed arithmetic is desirable for storage efficiency, reusability of the arithmetic cell designs, and regularity in VLSI circuit implementation. While encoding-algorithm combinations that are not representationally closed can be useful and are in fact used in practice, when comparing a representationally closed scheme against a scheme that is not closed, fairness dictates that the overhead of conversion from the intermediate representation to the ultimate encoding be taken into account in any cost/speed comparisons.

Where the two operands in addition are represented with the same canonical WBS encoding, the reduction cells of Figs. 6.6a and 6.6c may be used to produce a 2-deep result with the same redundancy pattern of the operands. Preserving the redundancy pattern is a necessary condition for representational closure, but it is not sufficient; the number of posibits and negabits of the like positions of the result and the operands should be the same as well. One obvious case, in which the latter property is sufficient, is when the encoding consists of only posibits (e.g., SC digit) or negabits. The adder cells of Fig. 6.7, however, preserve representational closure, except for a one position left shift of the result, that is, the number of posibits and negabits of any position $i + 1$ of the result is equal to that of position i of either operand.

Figure 6.11 depicts, in dot notation, representationally closed addition of two 3-digit symmetric hybrid-redundant operands with the digit set [–8, 8]. Figure 6.12 shows a regular adder design for an arbitrary radix-2^h digit i extending from position ih to $(i + 1)h - 1$, where the only building blocks are full-adders and half-adders (shaded with dots) and cells drawn with dashed lines belong to position $ih - 1$.

The following steps explain the addition process:

1. Replace the 2-deep equal-weight negabits by an $(h + 1)$-position 1-deep 2's-complement number of the same value. This produces a new negabit in the next redundant position. According to Table 6.IV, a standard half-adder can produce the 2-bit 2's-complement sum of two negabits. Sign-extending this to h bits produces the desired result; however, due to our inverted encoding of negabits, an inversion is required. The required circuitry for this step, a half-adder in the leftmost position of each radix-2^h digit and two inverters, can be seen in Fig. 6.12.

2. At the same time, reduce the 4- (2-) deep posibit positions by one full- (half-) adder. The intermediate result thus derived will be 3-deep. Zero valued posibit constants (**bold 0**), and zero valued negabit constant (1), have been added in the least significant digit position of Fig. 6.11 for regularity. The delay for this step is equal to that of one full-adder.
3. Use one full-adder per position to reduce the 3-deep result to one with depth 2. The latency of this step is again equal to the delay of one full-adder.
4. Use a chain of h full-adders per every h positions to derive the final result. The delay of this step is equal to that of h cascaded full adders. For large h (say, ≥ 4), one may use carry acceleration techniques to gain a delay of O(log h).

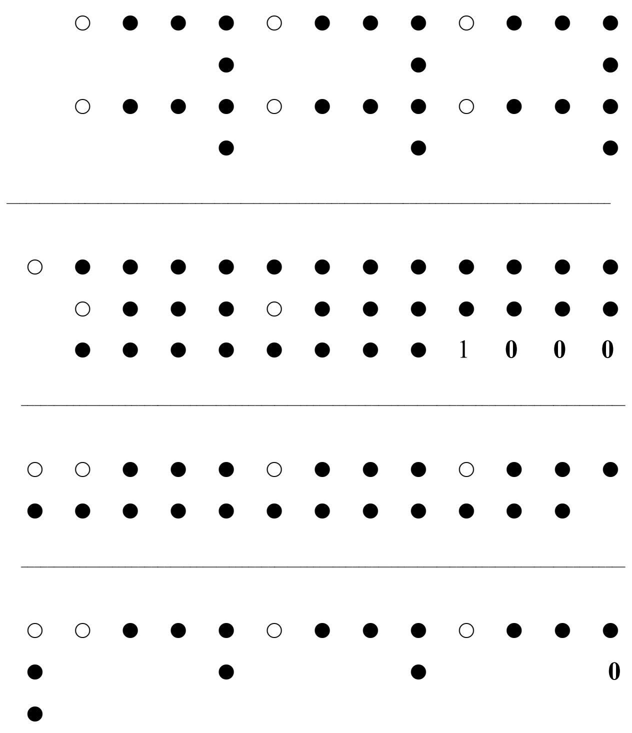


Fig. 6.11. Representationally closed addition of symmetric hybrid -redundant operands.

The extra cost for subtraction is minimal. We negate the subtrahend by bitwise inversion of each digit, and then perform addition as above. That a simple bit-wise inversion of each digit negates that digit, and thus the whole number is negated, is justified by the following equations for an h position symmetric digit.

Arithmetic value of an h position symmetric digit D, given that of a negabit X is $X - 1$, is:

$$V(D) = 2^{h-1}(X_{h-1} - 1) + 2^{h-2}x_{h-2} + \ldots + 2x_1 + x_0 + x'_0.$$

Arithmetic value of D', obtained by bitwise inversion of D, may be computed as follows. Derive the logical representation of D' by replacing each variable x by $1 - x$ in the representation of D:

$D' = 1 - X_{h-1}\ 1 - x_{h-2}\ \ldots\ 1 - x_1\ 1 - x_0$

$1 - x'_0$

Then compute the arithmetic value $V(D')$ as above:

$V(D') = 2^{h-1}(1 - X_{h-1} - 1) + 2^{h-2}(1 - x_{h-2}) + \ldots + 2(1 - x_1) + (1 - x_0) + (1 - x'_0) =$

$-(2^{h-1}X_{h-1} + 2^{h-2}x_{h-2} + \ldots + 2x_1 + x_0 + x'_0) + 2^{h-2} + \ldots + 2 + 1 + 1 =$

$-(2^{h-1}(X_{h-1} - 1) + 2^{h-2}x_{h-2} + \ldots + 2x_1 + x_0 + x'_0) = -V(D).$

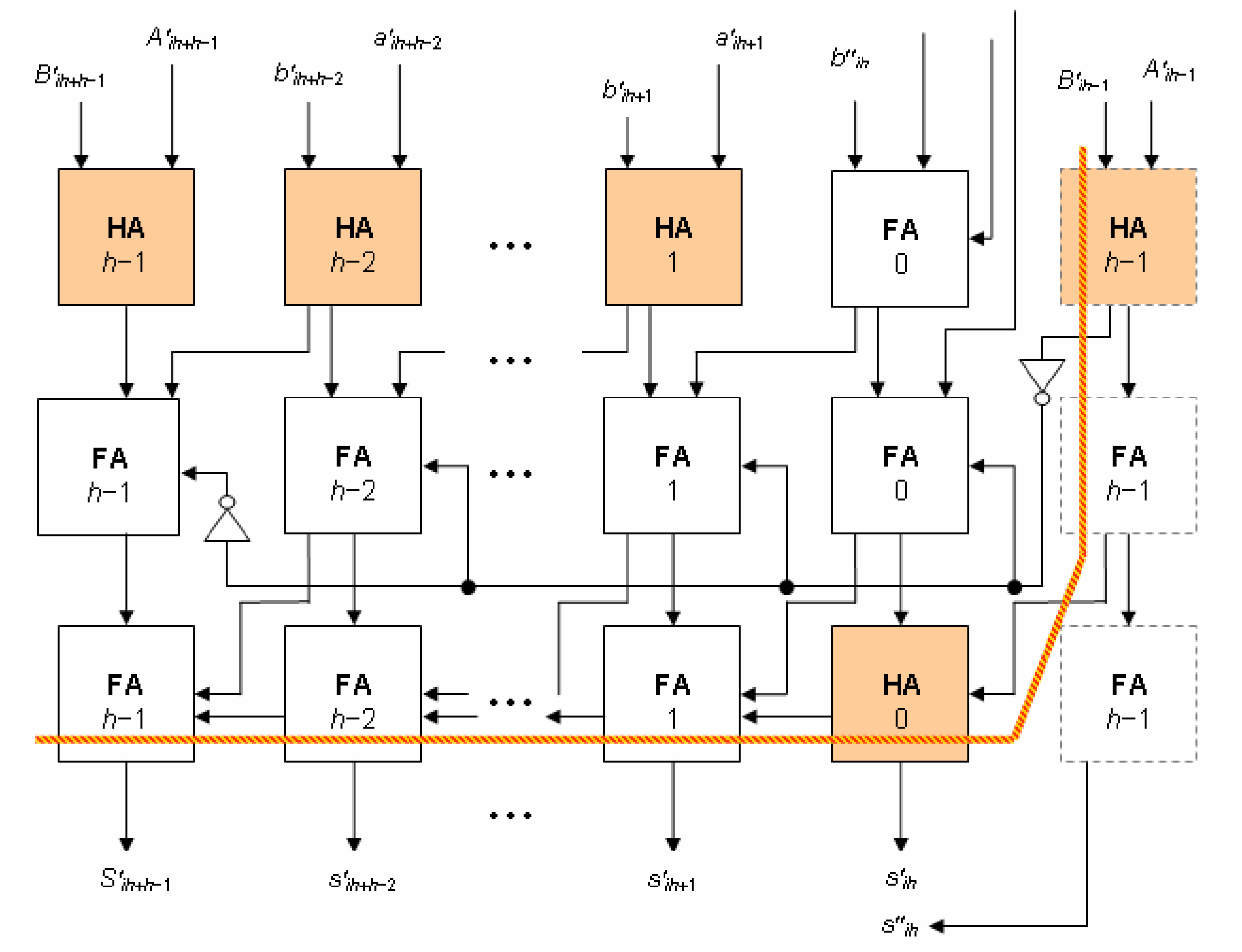


Fig. 6.12. Representationally closed adder for digit *i* of radix-2^h symmetric hybrid redundant numbers

The overall adder circuitry, as depicted in Fig. 6.12, amounts to two full-adders and one half-adder per radix-2 position. An inverter per bit and a multiplexer is the minimum possible penalty for subtraction, which is fortunately realizable in this case, as noted above. The total addition delay, corresponding to the critical path of Fig. 6.12 (the heavy bold line) is equal to that of h full-adders and two half-adders. With a carry acceleration circuit, an O(log h) delay can be easily achieved. Note that a representationally shifted adder, based on the adder cells of Fig. 6.7, consumes one (two) full-adder(s) per nonredundant (redundant) position, that is, a total of $h + 1$ full-adders per radix-2^h digit. The delay, in this case, is equal to that of $h + 1$ full-adders, almost the same as in the case of representationally closed adder. However, the hardware penalty for representational closure is rather substantial; the equivalent of one extra half-adder (and one extra full-adder) per redundant (nonredundant) position.

6.5. Summary

The hybrid redundancy scheme of [Phat01] constitutes as an easily understood concept leading to straightforward management of area-time trade-off in the design of hybrid-redundant number systems. The designer has the option of considering as many posibits between the redundant positions as required by the area-time targets. The redundant positions are practically restricted to at most 4-valued digit sets to enhance addition speed. The latter, with the help of equal-weight grouping, has led to 2-deep encodings (using the terminology of WBS encodings) of hybrid redundant number systems. However, the ordinary hybrid redundancy scheme as defined in [Phat01] fails to offer the latter design flexibility when shallow symmetric number systems are desired, does not directly allow the use of carry acceleration techniques, and fails to support subtraction by means of the same circuitry used for addition.

In this chapter, we provided an in-depth analysis of limitations of ordinary hybrid redundancy and showed that these problems can be overcome by two innovations:

- Allowing single negabits in nonredundant positions: This possibility, which led to definition of extended hybrid-redundant number systems, helps in designing shallow symmetric hybrid redundant number systems, which would become impractically deep otherwise (the depth increases exponentially with the spacing of redundant positions).

- Inverted encoding of negabits: This simple idea leads to universal functionality of conventional full/half-adders and compressors in reducing any combination of posibits and negabits and renders carry acceleration techniques directly applicable. Conventional binary full/half-adders have been studied extensively with regard to area, speed, and energy efficiency; hence, using them in our designs allows a wide choice of predesigned and highly optimized cells. Furthermore negation operation is quite efficient, leading to direct reusability of addition circuitry for subtraction. For example negation, in the case of popular symmetric number systems is done by bitwise inversion.

We showed that when representationally shifted results are acceptable, as is generally the case in ordinary hybrid redundancy, a universal adder may be designed with one (two) full-adders per nonredundant (redundant) position. The adder delay for radix-2^h periodic number systems equals to that of $h + 1$ full-adders. As shown in the representationally closed adder of Fig. 6.12, the hardware penalty for the coexistence of symmetry and representational closure, both desired in practice, is the equivalent of one extra half-adder (and one extra full-adder) per redundant (nonredundant) position. Fortunately, however, the addition delay is almost the same (that of h full-adders and two half-adders in series), so the speed penalty is not serious.

Further research on the extended hybrid redundancy schemes may proceed by considering the design of multipliers and dividers as well as efficient circuits for converting from various extended hybrid-redundant formats to standard 2's-complement binary format.

Chapter 7 Weighted Two-Valued Digit-Set Encodings

Contributions to redundant number representation and associated arithmetic systems are of two main types. In abstract studies (e.g., [Matu82], [Parh90], [Korn94]), arithmetic algorithms are presented in terms of digit-level operations, specifying how each result digit is derived from operand digits and auxiliary quantities such as interdigit transfers. Implementation-oriented studies, on the other hand, are often based on specific encodings for the digit sets encountered in solving particular design problems; for example, construction of a high-speed 2's-complement full-tree multiplier [Taka85], design of high-throughput floating-point units [Matu97], [Niel97], or enhanced implementation of floating-point addition and rounding [Fahm03]. Some contributions of this latter type have dealt with limited classes of digit-set encodings without directly associating them with a specific design problem or application. Examples include the hybrid redundancy scheme [Phat94], [Phat01] and representation paradigms for high-radix signed-digit number systems [Jabθ3].

This chapter aims to fill the gap between the aforementioned contributions. We note that radices of practical interest are invariably powers of 2; thus, in practice, a redundant number is formed by a collection of digits, each associated with a power-of-2 weight. Within each digit position, a digit value is also practically encoded as a collection of weighted bits. For example, the possibly asymmetric digit set $[\alpha, \beta]$, with $\alpha \geq -2^{\eta-1}$ and $\beta < 2^{\eta-1}$, might be encoded as an η-bit 2's-complement number, giving its bits the weights $-2^{\eta-1}, 2^{\eta-2}, \ldots, 2, 1$. Similarly, binary signed-digit (BSD) numbers [Aviz61] are commonly represented by using two bits weighted -2^i and 2^i for the position-i digit, leading to the (n, p) encoding [Parh90]. Also in carry-save [Metz59] and stored-transfer [Jabe01] redundant representations, the stored carry or transfer digit is composed of bits with the same weights as those of the main digit. Finally, a hybrid-redundant representation [Phat01] may have redundant positions with stored-double-borrow (SDB) digits in $[-2, 1]$, each of which is encoded using two bits of weight -2^i and one bit of weight 2^i or with a pair of bits of weights -2^{i+1} and 2^i. Under such conditions (i.e., power-of-2 radix and weighted bit-set representation of each digit), the number as a whole is encoded by a collection of bits; posibits in $\{0, 1\}$ or negabits in $\{-1, 0\}$, each weighted by a positive or negative power of two, respectively.

The *weighted bit-set* (WBS) encoding [Jabθ2] has been studied based on the observation just made. Any addition scheme for WBS-encoded operands entails the problem of combining bits with potentially opposite polarities. Some studies have presented variations of full- and half-adders as a solution to the latter problem. Examples include the PPM cell, proposed in connection with redundant representations of complex numbers [Dupr91] and later used in the design of a borrow-save adder [Mign00], and four half-adder variants that reduce various combinations of equally weighted posibits and negabits [Daum03]. A rather complex dual-purpose logic [Phat01] for addition of two stored-double-borrow (SDB) or stored-borrow-or-carry (SBC) digits has addressed a similar problem. Inverted encoding of negabits (representing −1 by 0, and 0 by 1, which is exactly the opposite of conventional encoding) allows standard full- and half-adders to be applied for deriving the sum and carry bits of either polarity for any collection of two or three posibits and negabits [Jabθ5 a].

Given that two-valued digit sets other than {0, 1} and {–1, 0} have found applications in practice (e.g., {–1, 1} in representing stored-transfer numbers [Jabe01]), we are motivated to generalize binary digits to *two-valued digits* (*twits*) and to extend WBS encoding to allow twits in any position. This is taken up in Sections 7.1 and 7.2, where we define twits and *weighted twit-set* (WTS) encodings and examine their properties. These include the bias encoding of twits (as a generalization of the aforementioned inverted encoding of negabits), which leads to the possibility of twit manipulations by means of standard full/half-adders. This reliance on the use of standard building blocks makes our results imminently practical. WBS-like encodings, as a subclass of WTS encodings with immediate practical interest, are introduced in Section 7.3, where we establish necessary and sufficient conditions for contiguity of digit sets and existence of equivalent canonical forms. In WBS-like encodings, each binary position, possibly including noncontiguous twits, represents a contiguous digit set with 0 as a member. WTS interpretation of previously studied redundant number systems (such as generalized signed-digit [Parh90], hybrid-redundant [Phat94], [Phat01], and stored-transfer [Jabe01] representations) is taken up in Section 7.4, where we also provide a general arithmetic framework for WBS-like encoded numbers, based primarily on the notion of digit-set conversion [Korn94], [Korn99], and offer a representationally closed addition/subtraction high level design, for a subclass of WTS encodings. Section 7.5 provides a summary of the chapter and offers a comprehensive hierarchical classification of all redundant number representations that the authors have encountered in the literature as instances of WTS encodings.

Various properties of twits and of WTS encodings cited in this chapter are stated as theorems and associated corollaries, and supported by formal proofs, in Appendix 7.A.

7.1. Two-Valued Digits (Twits)

Besides negabits and posibits used in the WBS definition [Jabe02], other two-valued digits, such as transfer digits in {–1, 1}, have been found useful in practice [Jabe01]. Also, one could think of an SDC, or stored-double-carry [Parh 90], digit in [0, 3] as being represented, with improved encoding efficiency, by a pair of equally weighted two-valued digits in {0, 1} and {0, 2}, respectively, instead of by 3 equally weighted posibits. Digit sets not including 0, such as [1, 3], cannot be faithfully represented by any collection of posibits and/or negabits. However a posibit and a two-valued digit in {1, 2}, both of the same weight, can represent [1, 3] precisely. This motivates us to generalize binary digits to two-valued digits in Definition 7.1 and to extend WBS encoding to include any two-valued digit (see Definition 7.3 at the beginning of Section 7.2).

Definition 7.1 (Two-valued digit or twit): A *twit* has two possible values, λ and $\lambda + \gamma$. A twit encoded as a bit x represents the value $\lambda + \gamma x$, with λ (*lower value*) and $\gamma > 0$ (*gap size*) being the twit parameters. If $\gamma = 1$ ($\gamma > 1$), the twit is *contiguous* (*noncontiguous*). If $\lambda \neq 0$ ($\lambda = 0$), the twit is *biased* (*unbiased*). The least (highest) representable value by a collection of m equally weighted twits is $\Lambda_{(m)}$ ($\Lambda_{(m)} + \Gamma_{(m)}$), where $\Lambda_{(m)} = \sum_{0 \le i < m} \lambda_i$ and $\Gamma_{(m)} = \sum_{0 \le i < m} \gamma_i$. □

For notational convenience, we use letters to denote twits according to the following conventions. Regular (**boldface**) type is used to denote contiguous (noncontiguous) twits, while lower (UPPER) case is used for unbiased (biased) twits having $\lambda = 0$ ($\lambda \neq 0$). When 0 is not one of the two twit values, we underline the twit's symbolic name. Twits in the same digit position are distinguished by using prime, double-prime, triple-prime, and so on. For ease of reference, these conventions are illustrated in Fig. 7.1, and some special twits, along with their representations in dot and symbolic notations, are depicted in Fig. 7.2.

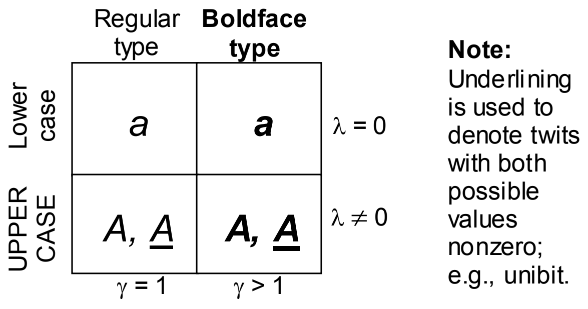

Regular type
Boldface type
Lower case
UPPER CASE
a
a
A, A̲
A, A̲
λ = 0
λ ≠ 0
γ = 1
γ > 1
Note: Underlining is used to denote twits with both possible values nonzero; e.g., unibit.

Fig. 7.1. Conventions for twit symbolic names .

	Name	Lower value	Upper value	Gap size	Dot notation	Symbolic notation
(a)	Bit or posibit	0	1	1	●	x', y'', z'''
(b)	Negabit	–1	0	1	○	X', Y'', Z'''
(c)	Unibit	–1	1	2	▣	$\underline{\boldsymbol{X}}', \underline{\boldsymbol{Y}}'', \underline{\boldsymbol{Z}}'''$
(d)	Doublebit	0	2	2	■	$\boldsymbol{x}', \boldsymbol{y}'', \boldsymbol{z}'''$
(e)	Negadoublebit	–2	0	2	□	$\boldsymbol{X}', \boldsymbol{Y}'', \boldsymbol{Z}'''$

Fig. 7.2. Some examples of two -valued digits or twits.

It has been shown elsewhere [Jabe05a] that a standard full-adder is capable of correct (3; 2) reduction of posibits and negabits, provided that negabits are inversely encoded, with the lower −1 value of a negabit encoded as 0 and the upper 0 value encoded as 1. In other words, the arithmetic value of a negabit with the same logical value as a posibit is biased by −1. This observation may be generalized as follows.

Definition 7.2 (Bias encoding of twits): Encoding of the lower value λ and higher value λ + γ of a twit as 0 and 1, respectively, is called *bias encoding* (the lower value λ is biased relative to lower value of a posibit). *Twit bias* is then synonymous with the lower value λ. □

Twit property 1 (Twit-FA): The sum and carry outputs of a standard full-adder, receiving three bias encoded twits with equal gaps, can represent arithmetically correct sum and carry twits with the same gaps. This property, which stems from our special bias encoding of twits, is justified by Theorem 7.1 in Appendix 7.A. □

The bias λ for a negabit is –1; that is, for a bias-encoded negabit, logical 0 means –1 and logical 1 means 0, which is the opposite of the convention used for the negabit in the most significant position of a 2's-complement number. Similarly, for a bias-encoded unibit, logical 0 means –1 and logical 1 means 1; again the opposite of the universally adopted sign convention. However, given that each of the two possible encodings of a twit is the logical inverse of the other, any required conversion is trivial.

Example 7.1 (Twit-FA): Figure 7.3a shows the functionality of a standard full-adder as twit-FA for different collections of three posibits and negabits. The functionality of the full-adder of Fig. 7.3b in adding a unibit $(-1 + 2\underline{\boldsymbol{X}})$, a doublebit $(0 + 2\boldsymbol{C}_{\mathbf{in}})$, and a negadoublebit $(-2 + 2\boldsymbol{y})$ is justified by:

$$-3 + 2\,(\underline{\boldsymbol{X}} + \boldsymbol{y} + \boldsymbol{C}_{\mathbf{in}}) = 2\ \ (-1 + 2\underline{\boldsymbol{C}}_{\mathbf{out}}) + (-1 + 2\underline{\boldsymbol{S}})$$

where $2\underline{\boldsymbol{C}}_{\mathbf{out}} + \underline{\boldsymbol{S}} = \ \underline{\boldsymbol{X}} + \boldsymbol{y} + \boldsymbol{C}_{\mathbf{in}}$ represents the normal full-adder functionality. □

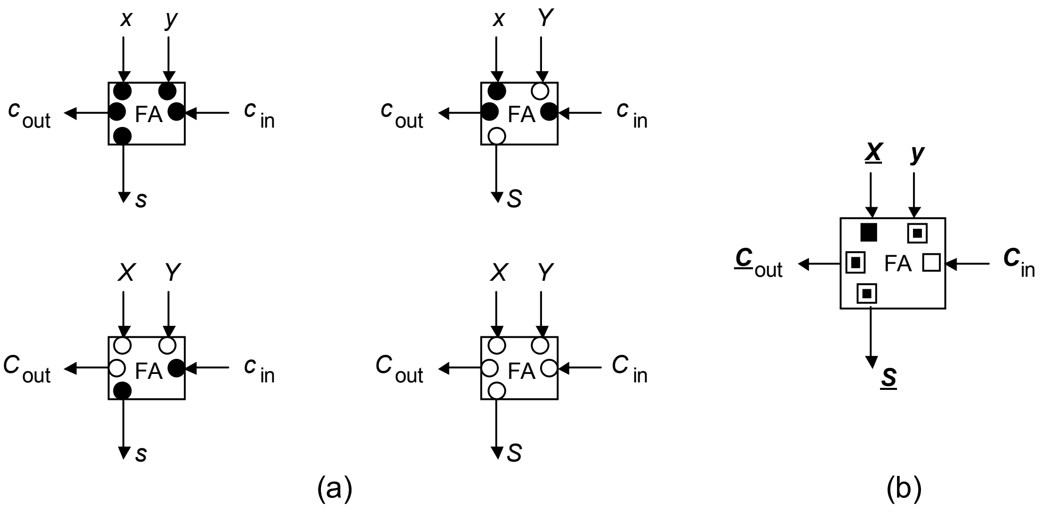


Fig. 7.3. Twit-FA used for adding various collections of three twits.

Twit property 2 (Twit compressor): A standard compressor, normally implemented by a collection of standard full-adders, may receive equigap twits in lieu of input posibits and produce twits with the same gap where one normally sees output posibits. This property is justified by Corollary 7.1 in Appendix 7.A. □

Twit property 3 (In-place reduction of twits): Three equally weighted, equigap, bias-encoded twits may be reduced by a full-adder to two equally weighted twits, one with a doubled gap (the carry output) and one with the original gap (the sum output). Furthermore, two equally weighted twits can be replaced by two other twits with the same weights, where the bias of one is increased by a constant and that of the other is decreased by the same constant. This property is justified by Corollary 7.2 in Appendix 7.A. □

Example 7.2 (Twit reductions): A collection of two posibits and one negabit may be reduced to a doubly weighted posibit and one negabit (per twit property 1), a doublebit and a negabit, a {1, 2} twit and a negadoublebit (per the first part of twit property 3), or a unibit and a posibit (per the second part of twit property 3). The last three in-place reductions are depicted in Fig. 7.4. We will make good use of the latter reduction in generating the unibit transfer of a stored transfer addition in Section 7.4. □

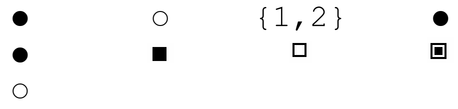


Fig. 7.4. Two posibits and one negabit, along with three possible in-place reductions.

Twit property 4 (Gaps in representation): Consider for each twit in a collection of equally weighted twits, the difference between its gap and sum of the gaps of all twits with smaller gap sizes. The largest of these differences equals the maximum distance between consecutive integer values in the ordered collection of integers representable by the twit collection. When the largest difference is 1, the twit collection can represent a contiguous interval of integers. Furthermore, if the representable contiguous interval is $[-\alpha, \beta]$ and includes 0, it may equivalently be represented by an equally weighted collection of α negabits and β posibits. Justifications are provided by Theorem 7.2 and Corollaries 7.3 and 7.4 in Appendix 7.A. □

Example 7.3 (Representational efficiency of twits): The set of three twits $\{0, 1\}$, $\{-2, 0\}$, and $\{1, 5\}$, with gaps of $\gamma_0 = 1$, $\gamma_1 = 2$, and $\gamma_2 = 4$, meets the conditions of twit property 4. The set represents integers in $[-1, 6]$, which can equivalently be represented by a collection of 1 negabit and 6 posibits. This example demonstrates the representational power of twit collections in enhancing the overall encoding efficiency of redundant binary digits (3 twits versus 7 negabits/posibits). This observation is generalized as twit property 5 below. □

Twit property 5 (Size of twit representation): A contiguous interval $[\alpha, \beta]$ of integers is representable by the minimum number $m = \lceil \log_2(\beta - \alpha + 1) \rceil$ of equally weighted twits whose gaps are $1, 2, 4, \ldots, 2^{m-2}, \beta - \alpha + 1 - 2^{m-1}$. For $\beta - \alpha + 1 = 2^m$, encoding efficiency of the resulting representation (see Definition 7.4 in Section 7.2) is maximal. This property is justified by Theorem 7.3 and Corollary 7.5 in Appendix 7.A. □

7.2. Weighted Twit-Set (WTS) Encodings

Having defined twits and examined some of their properties, we proceed to introduce a very general twit-based encoding scheme as a tool for unifying, evaluating, and comparing redundant number representations.

Definition 7.3 (WTS-encoded numbers): A k-position *weighted twit-set* (WTS) encoding is characterized by k integers $m_{k-1}, \ldots, m_1, m_0$, where the representation has k radix-2 positions indexed 0 to $k - 1$ and the multiplicity of digit position i ($0 \le i < k$) of weight 2^i is m_i (i.e., it is comprised of m_i twits). We postulate for the most significant position that $m_{k-1} > 0$. Other positions may be empty, that is, $m_i \ge 0$ for $0 \le i < k - 1$. □

Note that WBS encodings [Jabe02], elaborated upon in Section 7.3, constitute special cases of WTS encodings, where the twits are restricted to posibits and/or negabits.

Definition 7.4 (Characteristics of WTS encodings): The lowest (highest) value collectively representable by the twits in position i is Λ_i ($\Lambda_i + \Gamma_i$), where $\Lambda_i = \sum_{0 \le j < m_i} \lambda_j$ and $\Gamma_i = \sum_{0 \le j < m_i} \gamma_j$. *Positional bias* is a synonym for the lowest positional value Λ_i. The maximum distance for position i (see twit property 4) is denoted as d_i^{max}. The *effective gap* of the twit $\{\lambda, \lambda + \gamma\}$ in position i is $\varepsilon = 2^i\gamma$. The lowest (highest) value collectively representable by the i rightmost positions of the WTS encoding is Λ^+_i ($\Lambda^+_i + \Gamma^+_i$), where $\Lambda^+_i = \sum_{0 \le j < i} 2^j\Lambda_j$ is the *partial encoding bias* and $\Gamma^+_i = \sum_{0 \le j < i} 2^j\Gamma_j$. The lowest (highest) value representable by such a k-position encoding as a whole is Λ^+ ($\Lambda^+ + \Gamma^+$), where Λ^+ is the *total encoding bias*. The redundancy index of position i is defined as $\rho_i = \Gamma_i - 1$, where a negative ρ_i of -1 occurs in empty positions (denoted by ∇ in our extended dot notation) and $\rho_{k-1} \ge 0$ by Definition 7.3.

The ordered collection $\rho_{k-1} \dots \rho_1\rho_0$ of the k positional redundancy indices is the *redundancy pattern* and $R = \Gamma^+ + 1 - 2^k$ is the *total redundancy index*, which may be represented as the possibly redundant radix-2 number $(\rho_{k-1} \dots \rho_1\rho_0)_{\text{two}}$. Similarly, the i^{th} *partial redundancy index* is defined as $R_i = (\rho_{i-1} \dots \rho_1\rho_0)_{\text{two}} = \Gamma^+_i + 1 - 2^i$, with $R_0 = 0$. The total encoding cost is $E = \sum_{0 \le i < k} m_i$, leading to the encoding efficiency $e = \lceil \log_2(\Gamma^+ + 1) \rceil / E = \lceil \log_2(2^k + R) \rceil / E$. □

Definition 7.5 (Strongly contiguous WTS encoding): A *strongly contiguous* WTS encoding is one where each digit position represents a nonempty interval of integers (see twit property 4) and, consequently, so does the entire encoding. □

Definition 7.6 (Equivalent WTS encodings): WTS encodings representing precisely the same set of integer values are *equivalent*. *Strongly equivalent* WTS encodings are equivalent and equiwidth (have the same number k of positions). □

Definition 7.7 (Complementary WTS encodings): If the negation of every integer representable by a WTS encoding is representable by another WTS encoding, and vice versa, the two encodings are *complementary*. If each twit of a given WTS encoding is replaced by an inverted twit (e.g., posibits by negabits, negabits by posibits, and doublebits by negadoublebits), with possible swapping of placements in the same position, the encoding that results is *strongly complementary* to the original one. □

Equivalent or complementary WTS encodings that are equiwidth have the same total redundancy indices, but their redundancy patterns may be different in general; redundancy patterns are the same in case of strong complementation. Complementary equiwidth WTS encodings are not necessarily strongly complementary.

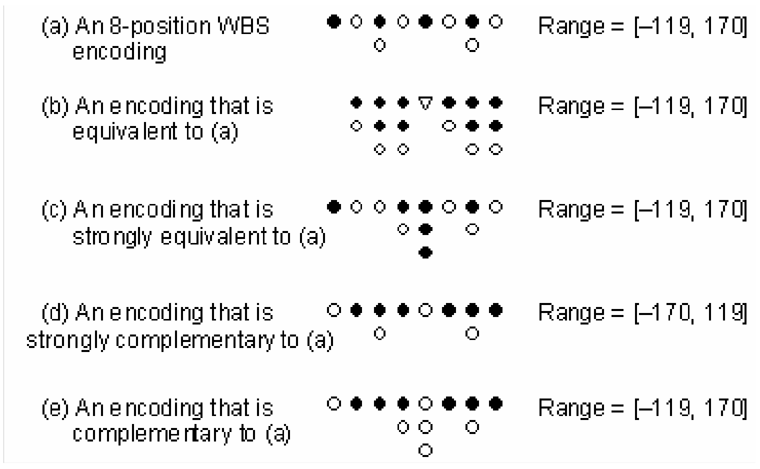

(a) An 8-position WBS encoding
Range = [–119, 170]
(b) An encoding that is equivalent to (a)
Range = [–119, 170]
(c) An encoding that is strongly equivalent to (a)
Range = [–119, 170]
(d) An encoding that is strongly complementary to (a)
Range = [–170, 119]
(e) An encoding that is complementary to (a)
Range = [–119, 170]

Fig. 7.5. Equivalent and complementary WTS encodings .

Example 7.4 (equivalent WTS encodings): The 8-position WBS encoding (a) in Fig. 7.5 is equivalent to the 7-position encoding (b). Furthermore, encoding (a) is strongly equivalent to the 8-position encoding (c), strongly complementary to the 8-position encoding (d), and complementary to the 8-position encoding (e). □

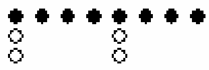
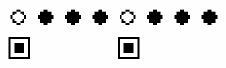
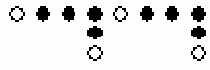
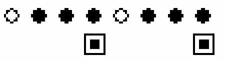
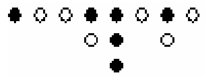
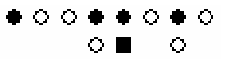
7.3. WBS-Like Encodings

The digit sets encountered in practice are, almost always, contiguous and include 0 as a member. These contiguous zero-included digit sets may be represented by a collection of equally weighted posibits and negabits in a straightforward manner, leading to a WBS encoding. However, a noncontiguous and/or zero-excluded twit may contribute in the representation of the same digit set and enhance the encoding efficiency (see Example 7.2). The equivalence of the two representations (i.e., with and without twits other than posibits and negabits) hints that any result obtained for WBS encodings [Jabe02] might be valid for WTS encodings with contiguous zero-included digit sets in each nonempty position. Therefore we define the class of WBS-like encodings and review the properties of WBS encodings that are applicable to WBS-like encodings.

Definition 7.8 (WBS-like encoding): A *WBS-like* encoding is a strongly contiguous WTS encoding that meets the conditions of the last part of twit property 4 in every digit position; that is, each digit position represents an interval of integers including 0 or, equivalently, is representable by a collection of equally weighted posibits and negabits. □

Example 7.5 (WBS-like encodings): Table 7.I depicts some WBS encodings along with their equivalent WBS-like encodings illustrating the advantage of noncontiguous twits in improving the encoding efficiency. The first entry represents a two digits radix-16 periodic hybrid redundant number system [Phat01] with stored-double-borrow (SDB) redundant positions using digits in [−2, 1]. The second one is a stored transfer representation [Jabe01] with transfer digits in [−1, 1]. The third one is a made-up example intended to illustrate the generality of our encodings in that they need not be regular or periodic. □

Table 7.I. Some WBS and equivalent WBS-like encodings.

Encoding name	WBS encoding	WBS-like encoding	Range
SDB hybrid			–272, 255
Stored transfer			–153, 136
Not named			–119, 170

WBS Property 1 (Contiguity): A WBS encoding is said to be contiguous iff the set of integers represented by the encoding exactly coincides with a contiguous interval of integers. Obviously, A WBS encoding with no empty position is contiguous. But if the right context of an empty position is deep enough to compensate for the missing range caused by the empty position, then the whole encoding could still be contiguous. Formal description of this property is provided by Theorem 7.4 in Appendix 7.A. □

WBS property 1 suggests that even though it is possible to avoid having any posibit or negabit in a particular position j, doing so would require additional bits in less significant positions (two in position $j - 1$, four in position $j - 2$, and so on). Thus, for encoding efficiency, it is advantageous to enforce $m_i > 0$ for all i. On the other hand, replacement of a pair of bits of the same polarity in position j by one bit in position $j + 1$, through the substitutions outlined in Fig. 7.6, keeps $m_i \leq 2$, and further improves encoding efficiency. These observations lead us to define the class of canonical WBS encodings.

Definition 7.9 (Canonical WBS encodings): A k-position WBS encoding is *canonical* iff it is strongly contiguous (Definition 7.5) and has $\rho_i \leq 1$ (i.e., $1 \leq m_i \leq 2$) for $0 \leq i \leq k - 2$. □

Several strongly equivalent canonical encodings may exist for a given WBS encoding Ω. For example, if Ω is symmetric, any strongly equivalent canonical encoding Ω′ leads to another strongly equivalent encoding Ω″ which is strongly complementary to Ω′. Interestingly, these encodings have the same redundancy pattern.

WBS property 2 (Uniqueness of redundancy patterns among strongly equivalent canonical encodings): For all equivalent canonical WBS encodings with the same number of positions, the numbers of bits in the like positions are the same. Theorem 7.5 in Appendix 7.A provides justification for this property. □

WBS property 2, which is established through the transformations depicted in Fig. 7.6, has led to the possibility of designing a universal adder circuit for all such encodings [Jab05 a].

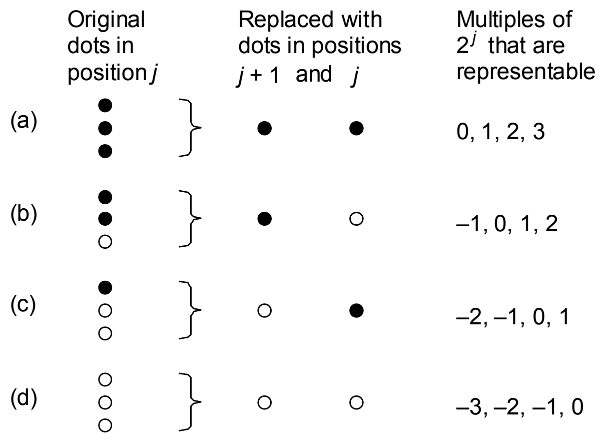

Original dots in position j
Replaced with dots in positions $j + 1$ and j
Multiples of 2^j that are representable
(a)
0, 1, 2, 3
(b)
–1, 0, 1, 2
(c)
–2, –1, 0, 1
(d)
–3, –2, –1, 0

Fig. 7.6. Substitutions used in the proof of WBS Property 2 (Theorem 7.5 in Appendix7.A)

WBS property 3 (Redundancy of a WBS encoding): A given k-position WBS encoding is *redundant* iff in any of its strongly equivalent canonical forms, $\rho_j > 0$ for some $j < k$. □

WBS property 3 is a direct consequence of WBS property 2. We have deliberately associated the redundancy of a WBS encoding with the redundancy of its strongly equivalent canonical forms because existence of a redundant position by itself does not imply a positive total redundancy index. For example, the 3-position WBS encoding having the redundancy pattern 0 –1 2 (i.e., with position 1 empty) is nonredundant, even though its position 0 is redundant.

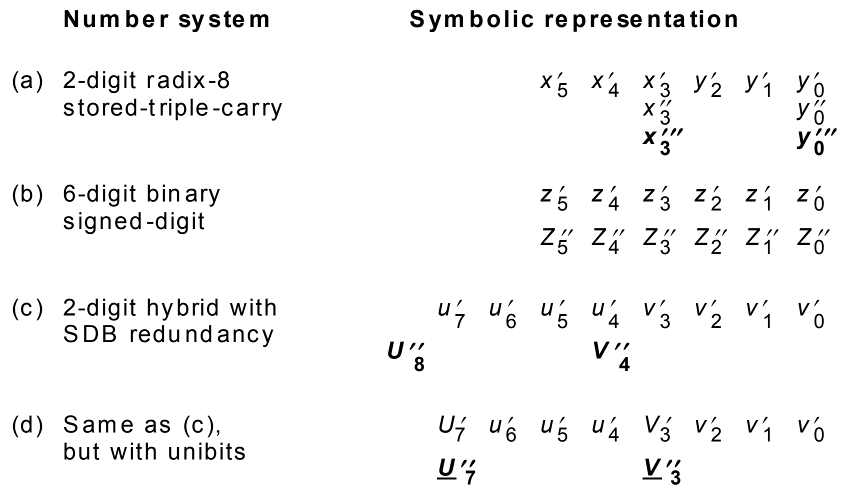
WBS Property 4 (Efficiency of canonical WBS encodings): The encoding efficiency of a canonical encoding Ω is maximal among all WBS encodings strongly equivalent to Ω. This property is established by Theorem 7.6 in Appendix 7.A. □

7.4. Arithmetic on WTS-Encoded Operands

While arbitrary WTS encodings can be envisaged and used, circuit implementation of arithmetic functions in VLSI favors regularity in the numbers and kinds of twits associated with the various positions. Thus, we define the class of periodic WTS encodings.

Definition 7.10 (Periodic WTS encodings): A k-position WTS encoding is deemed *periodic* iff there exists $h < k$, such that the twit collection of position $i + jh$ is precisely the same as that of position i, for $0 \le i \le h- 1$ and $0 < j \le \lceil k/h \rceil - 1$; the smallest such h is the *period.* □

Assuming k to be a multiple of h, a periodic WBS-like-encoded number represents a generalized signed-digit (GSD) number system in radix 2^h utilizing the digit set $[\alpha, \beta]$, with $\alpha = \Lambda^+_h$, and $\beta = \Lambda^+_h + \Gamma^+_h$, where $\Lambda^+_h = (\Lambda_{h-1} \ldots \Lambda_1 \Lambda_0)_{\text{two}}$ and $\Gamma^+_h = (\Gamma_{h-1} \ldots \Gamma_1 \Gamma_0)_{\text{two}}$.



Number system
Symbolic representation
(a) 2-digit radix-8 stored-triple-carry
(b) 6-digit binary signed-digit
(c) 2-digit hybrid with SDB redundancy
(d) Same as (c), but with unibits

Fig. 7.7. Symbolic representation of periodic WTS-encoded numbers.

Example 7.6 (Symbolic representation of WTS encodings): The symbolic representations for a 2-digit radix-8 stored-triple-carry (STC) number, a 6-digit BSD number, and a 2-digit radix-16 stored double-borrow (SDB) hybrid-redundant number are depicted in Fig. 7.7. Note that the digit sets for these WTS-encoded GSD number systems are [0, 10], [–1, 1], [–16, 15], and [–16, 15], respectively. □

A general framework for arithmetic operations with WTS-encoded operands may be established following the general framework of arithmetic for WBS encodings [Jabe02]. Given that addition operation may be viewed as a special case of digit-set conversion [Korn99], and arithmetic functions on WTS operands can always be reduced to one or more addition operations, the central problem in WTS arithmetic is recognized as conversion of a deep digit set to a one with less depth. This is where bias encoding of twits helps in using standard compressors for reducing the representation depth to a desired level. When the input operands and the derived results have the same WTS encodings, the arithmetic is said to be representationally closed, where a key example and its associated advantages, has been explained elsewhere [Jab05 a] in connection with SDB hybrid-redundant operands.

To illustrate the advantages of WTS encoding and of the use of twit-FAs in enhancing encoding efficiency and regularity of VLSI design as well as addition speed, we adapt the representationally closed WBS addition algorithm of [Jab05 a] to a WBS-like encoded stored-transfer representation. One such encoding can be seen in the second entry of Table 7.I with unibit transfers. In the following, we shift each of the unibit transfer digits h positions to the left as depicted in Fig. 7.8a in order to achieve a wider representation range. Because the most significant position then holds a single unibit, it violates the WBS-like restriction on positional contiguity (Definition 7.8) and also results in a representation gap by Theorem 7.4 (see Appendix 7.A). A simple fix is to replace the single unibit in the most-significant position by a posibit and a negabit (Fig. 7.8b). With these modifications, the representation remains periodic, except for the most-significant digit whose transfer is now a shifted binary signed digit instead of a shifted unibit.

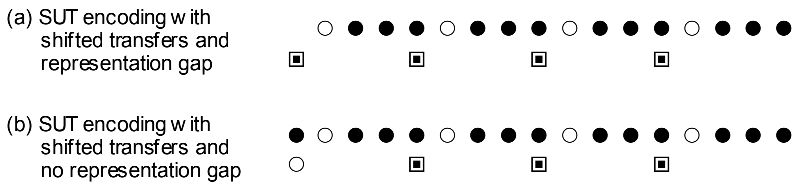

(a) SUT encoding with shifted transfers and representation gap
(b) SUT encoding with shifted transfers and no representation gap

Fig. 7.8. Stored unibit encodings with shifted transfers .

Definition 7.11 (SUT representation): The digit set Δ of a radix-2^h periodic stored-unibit-transfer (SUT) representation with shifted transfers is composed of a radix-2^h main part $\Delta' = [-2^{h-1}, 2^{h-1} - 1]$ in 2's-complement form and a twit transfer part $G = \{-2^h, 2^h\}$, except in the most significant position where the transfer set is $\{-2^h, 0, 2^h\}$. □

It is interesting to note that, due to use of noncontiguous twits (i.e., unibits) in the SUT definition, Δ is not contiguous, but with the most significant position modified to hold a contiguous digit set, the number system as a whole is contiguous. Also, the SUT representation may be regarded as an extended hybrid-redundant number system [Jabe05b] with the digit set $[-1, 2]$ in redundant positions, but it has no equivalent in ordinary hybrid-redundant scheme [Phat01]. For, the redundant digit set, composed of a negabit and a doubly weighted unibit, is not contiguous; it exactly represents $\{-3, -2, 1, 2\}$.

We now proceed to provide the high-level design for an adder for SUT operands. Figure 7.9 depicts a symbolic representation of addition steps for two 4-digit radix-16 SUT operands, where $T_{(i+1)h-1}$ and $t_{(i+1)h-2} \ldots t_{ih}$ ($i = 1, 2, 3$, $h = 4$), denote sign-extended 2's-complement sum of two unibits in position ih. The required circuit, actually very similar to a half-adder per each redundant position, is shown in Fig. 7.10, with a justification provided in Table 7.II. The overall SUT adder is depicted in Fig. 7.11, where the adder cells in the first row serve as reduction units. The full-adders in the second row are serially interconnected and perform a standard h-bit ripple-carry addition. This part of the circuit can be replaced by any desired fast adder design incorporating carry accleration. A second-row full-adder in position ih (except in the most significant digit position) generates a sum bit and a unibit transfer by in-place reduction of two posibits and one negabit (see twit property 3, and Corollary 7.2 in Appendix 7.A). Position kh needs a different treatment (that is why the details for that position are left out in Fig. 7.9); there are 3 negabits and three posibits to be added: a'_{kh}, b'_{kh}, A''_{kh}, B''_{kh}, and transfers coming from the first (C_{kh}) and second rows of full-adders. Figure 7.12 depicts the required hardware in position kh, along with overflow detection logic, where the overflow bits do not always indicate a real overflow [see Chapter 10]. This condition of " apparent overflow" is pretty much the norm in redundant number representation schemes [Parh93].

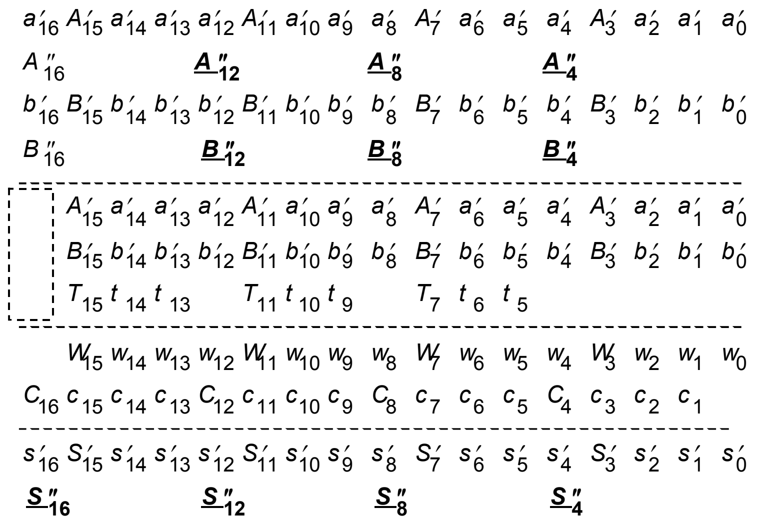

$a'_{16}\ A'_{15}\ a'_{14}\ a'_{13}\ a'_{12}\ A'_{11}\ a'_{10}\ a'_9\ a'_8\ A'_7\ a'_6\ a'_5\ a'_4\ A'_3\ a'_2\ a'_1\ a'_0$
$A''_{16}\ \underline{A}''_{12}\ \underline{A}''_8\ \underline{A}''_4$
$b'_{16}\ B'_{15}\ b'_{14}\ b'_{13}\ b'_{12}\ B'_{11}\ b'_{10}\ b'_9\ b'_8\ B'_7\ b'_6\ b'_5\ b'_4\ B'_3\ b'_2\ b'_1\ b'_0$
$B''_{16}\ \underline{B}''_{12}\ \underline{B}''_8\ \underline{B}''_4$
$A'_{15}\ a'_{14}\ a'_{13}\ a'_{12}\ A'_{11}\ a'_{10}\ a'_9\ a'_8\ A'_7\ a'_6\ a'_5\ a'_4\ A'_3\ a'_2\ a'_1\ a'_0$
$B'_{15}\ b'_{14}\ b'_{13}\ b'_{12}\ B'_{11}\ b'_{10}\ b'_9\ b'_8\ B'_7\ b'_6\ b'_5\ b'_4\ B'_3\ b'_2\ b'_1\ b'_0$
$T_{15}\ t_{14}\ t_{13}\ T_{11}\ t_{10}\ t_9\ T_7\ t_6\ t_5$
$W_{15}\ w_{14}\ w_{13}\ w_{12}\ W_{11}\ w_{10}\ w_9\ w_8\ W_7\ w_6\ w_5\ w_4\ W_3\ w_2\ w_1\ w_0$
$C_{16}\ c_{15}\ c_{14}\ c_{13}\ C_{12}\ c_{11}\ c_{10}\ c_9\ C_8\ c_7\ c_6\ c_5\ C_4\ c_3\ c_2\ c_1$
$s'_{16}\ S'_{15}\ s'_{14}\ s'_{13}\ s'_{12}\ S'_{11}\ s'_{10}\ s'_9\ s'_8\ S'_7\ s'_6\ s'_5\ s'_4\ S'_3\ s'_2\ s'_1\ s'_0$
$\underline{S}''_{16}\ \underline{S}''_{12}\ \underline{S}''_8\ \underline{S}''_4$

Fig. 7.9. Representationally closed SUT addition .

For subtraction, one usually negates the subtrahend, and then performs addition. However, we can apply a more efficient approach based on negating each digit independently. An SUT digit has a 2's-complement number as the main part, which can be negated via 2's complementation, and a unibit transfer which is negated by inversion. To negate the main digit, we simply invert its bits and let $t_{ih} = sub$, where *sub* (Fig. 7.10) is the subtraction control signal (1 for subtraction, 0 for addition). The hardware modification to accommodate subtraction is minimal and consists of replacing the half-adder in the first row of position *ih* by a full-adder. The most significant transfer being a BSD, again needs special treatment: to negate it, we simply invert its posibit and negabit components and do not apply the *sub* control as in other positions. Based on the description above, the time penalty for negation is minimal and consists of a single inverter delay.

Table 7.II. Combining of the unibit transfers for SUT addition.

$\underline{A}''_{ih}$	$\underline{B}''_{ih}$	Sum	$T_{(i+1)h-1}$	$t_{(i+1)h-2} \ldots t_{ih+2}$	t_{ih+1}	t_{ih}
0	0	–2	0	1 . . . 1	1	0
0	1	0	1	0 . . . 0	0	0
1	0	0	1	0 . . . 0	0	0
1	1	2	1	0 . . . 0	1	0

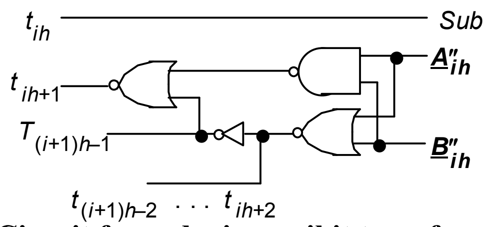


Fig. 7.10. Circuit for reducing unibit transfers of Fig. 7.9.

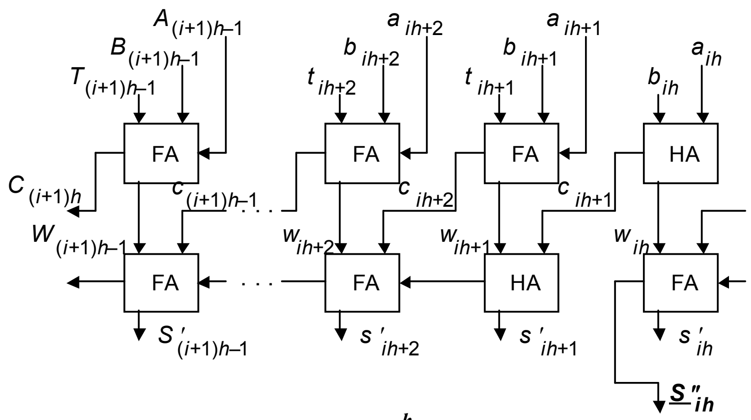


Fig. 7.11. SUT radix -2^h redundant adder.

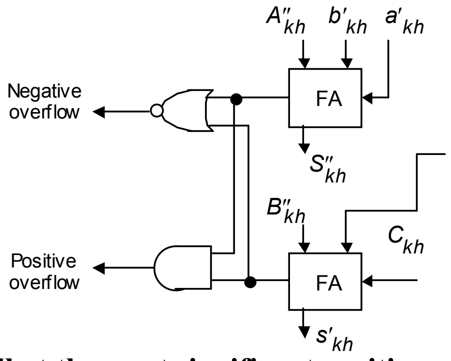


Fig. 7.12. The c ell at the most significant position of our SUT adder.

7.5. Summary

In this chapter, we introduced the use of general two-valued digits, or twits, that include posibits and negabits as special cases. We showed that weighted twit-set (WTS) encodings cover all positional redundant number systems that have appeared in the literature, including those employing subranges of integers (perhaps excluding zero) and noncontiguous digit sets. Figure 7.13 presents a hierarchical classification of all redundant representations that can be obtained from WTS encodings at the root. We showed how bias encoding of twits, as a generalization of inverted encoding of negabits, leads to new functionalities for standard full/half-adders and compressors in reducing equally weighted, equigap twits. The latter possibility led to use of standard reduction (e.g., Wallace tree) and carry acceleration techniques to implement arithmetic on WTS-encoded operands. Focusing on a subclass of WTS representations, those that possess equivalent WBS encodings, a twit-based representationally closed adder design for stored-unibit-transfer (SUT) representation was described. This twit-based design offers advantages over a similar WBS-based implementation of SDB hybrid redundancy [Jab05 a] in speed and time/logic penalty for subtraction relative to addition. Unified descriptions of these and other diverse implementations of redundant arithmetic can be viewed as evidence for the generality and usefulness of the WTS paradigm. Research on the representational power of twit-based encodings and their various applications is continuing. Problems currently being addressed include developing theories for general WTS representations (including twit-based formulation of digit-set conversions, necessary and sufficient conditions for constant-time WTS conversion, and representability of arbitrary digit sets), and deriving design details for twit-based multipliers, dividers, and other arithmetic circuits. Design of application-specific units for DSP applications and cryptography is also envisaged.

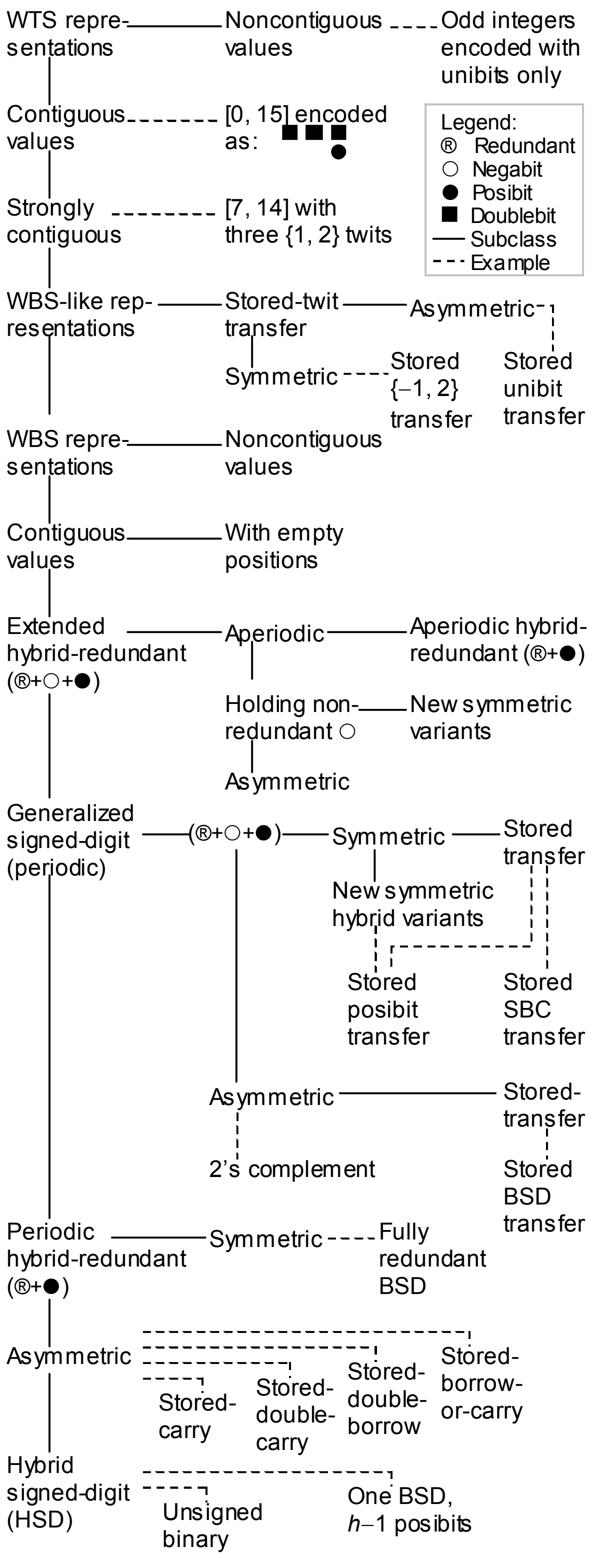


Fig. 7.13. The hierarchy of number representations resulting from WTS encoding (tree branches go from left to right and top to bottom).

Appendix 7.A

Theorem 7.1 (Twit FA): A standard full-adder cell receiving as inputs any three equally weighted, equigap, bias-encoded twits with values in $\{\lambda_i, \lambda_i + \gamma\}$, $i = 1, 2, 3$, produces carry and sum twits which have the same gap as the inputs and with bias values λ_c and λ_s satisfying $2\lambda_c + \lambda_s = \lambda_1 + \lambda_2 + \lambda_3$.

Proof: We describe the operation of an equigap twit full-adder and show that it can be implemented by a standard binary full-adder. Form the sum of three equally weighted equigap bias-encoded twits $\lambda_1 + \gamma x_1$, $\lambda_2 + \gamma x_2$, $\lambda_3 + \gamma x_3$:

$$(\lambda_1 + \gamma x_1) + (\lambda_2 + \gamma x_2) + (\lambda_3 + \gamma x_3) = (\lambda_1 + \lambda_2 + \lambda_3) + \gamma\,(x_1 + x_2 + x_3)$$

Let c and s be the carry and sum outputs of a standard full-adder, with the encoding bits x_1, x_2, and x_3 as inputs, and select biases λ_c and λ_s such that $2\lambda_c + \lambda_s = \lambda_1 + \lambda_2 + \lambda_3$. Substituting $2\lambda_c + \lambda_s$ for $\lambda_1 + \lambda_2 + \lambda_3$ and $2c + s$ for $x_1 + x_2 + x_3$ in the right-hand side of the equation above, we get:

$$(2\lambda_c + \lambda_s) + \gamma(2\,c + s) = 2(\lambda_c + \gamma c) + (\lambda_s + \gamma s)$$

Note that selection of λ_c and λ_s is always possible; if $\lambda_1 + \lambda_2 + \lambda_3$ is even (odd) , say equal to $2j$ $(2j + 1)$, then $\lambda_s = 2i$ $(2i + 1)$, and $\lambda_c = j - i$, for all integers i and j. Design peculiarities may guide the latter choices. For example with equibias input twits, the output carry and sum twits can be made to have the same bias. □

Corollary 7.1 (Twit compressor): A standard binary $(\nu;\ \mu)$-compressor receiving ν equally weighted, equigap twits in position i produces μ twits with the same gap in positions i to $i + \mu - 1$, such that inputs and outputs have the same collective values. Moreover, because any (multicolumn) posibit compressor can be implemented by a collection of standard full-adders, such a compressor may receive equigap twits, in lieu of input posibits, and produce twits with the same gap where one normally would see output posibits. □

Corollary 7.2 (In-place reduction of twits): Three equally weighted, equigap, bias-encoded twits $\lambda_1 + \gamma x_1$, $\lambda_2 + \gamma x_2$, and $\lambda_3 + \gamma x_3$, may be reduced, by a full-adder, to two equally weighted twits, one with a doubled-gap (i.e., the carry output), and one with the original gap (the sum output), such that:

$$(2\lambda_c + 2\gamma c)\ + \lambda_s + \gamma s\ = (\lambda_1 + \lambda_2 + \lambda_3) +\ \gamma(x_1 + x_2 + x_3)$$

Furthermore, the equally weighted twits $\{2\lambda_c, 2\lambda_c + 2\gamma\}$, having an even bias, and $\{\lambda_s, \lambda_s + \gamma\}$ can be replaced with $\{2\lambda_c + 1, 2\lambda_c + 1 + 2\gamma\}$ and $\{\lambda_s - 1, \lambda_s - 1 + \gamma\}$, respectively. □

Theorem 7.2 (Gaps in representation): The maximum distance between consecutive integer values in an ordered collection of integers representable by a set of $m \geq 1$ equally weighted twits, given the twit gaps in descending order $\{\gamma_{m-1}, \ldots, \gamma_1, \gamma_0\}$, is:

$$d^{max} = max\{(\gamma_j - \Gamma_{(j)}) \mid 0 \leq j < m\}, \text{ where } \Gamma_{(0)} = 0 \text{ and } \Gamma_{(j)} = \Sigma_{0\leq i<j}\, \gamma_i$$

Proof (by induction on m): Let $\{\lambda_i, \lambda_i + \gamma_i\}$ denote the values of a twit. The base case $m = 1$ is obvious, given that the formula yields $d^{max} = max\{(\gamma_j - \Gamma_{(j)}) \mid 0 \leq j < 1\} = \gamma_0$. Now assume that $m - 1$ twits represents the ordered set of integers $\Psi_{m-1} = \{\Lambda_{(m-1)}, \ldots, \Lambda_{(m-1)} + \Gamma_{(m-1)}\}$, per the theorem's statement, with $d^{max} = max\{(\gamma_j - \Gamma_{(j)}) \mid 0 \leq j < m - 1\}$, where $\Lambda_{(m-1)} = \Sigma_{0\leq i<m-1}\, \lambda_i$ and $\Gamma_{(m-1)} = \Sigma_{0\leq i<m-1}\, \gamma_i$. We now include $\{\lambda_{m-1}, \lambda_{m-1} + \gamma_{m-1}\}$ in the set of twits. The represented values for the m-twit collection are:

$$\Psi_m = \{(\lambda_{m-1} + v) \mid v \in \Psi_{m-1}\} \cup \{(\lambda_{m-1} + \gamma_{m-1} + v) \mid v \in \Psi_{m-1}\}$$

The value of d^{max} within each of the subcollections in Ψ_m remains the same as that of $\Psi_{m-1.}$ If the ranges of values in the two parts of Ψ_m overlap, then d^{max} for Ψ_m remains the same as that of Ψ_{m-1}, which together with $\gamma_{m-1} \leq \Gamma_{(m-1)}$ (due to the overlap), meet the condition of the theorem's statement. Otherwise the new d^{max} is the maximum of the old one, and the distance between the minimum value of the second part of Ψ_m and the maximum value of the first part, i.e., $\lambda_{m-1} + \gamma_{m-1} + \Lambda_{(m-1)} - (\lambda_{m-1} + \Lambda_{(m-1)} + \Gamma_{(m-1)}) = \gamma_{m-1} - \Gamma_{(m-1)}$. □

Corollary 7.3 (Representational contiguity of twit sets): A nonempty set of equally weighted twits represents an interval of integers (i.e., $d^{max} = 1$) iff, given the gaps in descending order $\{\gamma_{m-1}, \ldots, \gamma_1, \gamma_0\}$, $\gamma_0 = 1$ and $\gamma_j \leq 1 + \Gamma_{(j)}$ for $0 < j \leq m - 1$. □

Corollary 7.4 (WBS-like twit collections): The interval of integers represented by m twits meeting the conditions of Corollary 7.3, and $\Lambda_{(m)} \leq 0 \leq \Lambda_{(m)} + \Gamma_{(m)}$, is representable by a collection of $-\Lambda_{(m)}$ negabits and $\Lambda_{(m)} + \Gamma_{(m)}$ posibits, where the redundancy index (Definition 7.4) of the interval is $\Gamma_{(m)} - 1$. □

Theorem 7.3 (Size of twit representation): A contiguous interval $[\Lambda_{(m)}, \Lambda_{(m)} + \Gamma_{(m)}]$ of integers is representable by at least $m = \lceil \log_2(\Gamma_{(m)} + 1) \rceil$ equally weighted twits $\{\lambda_i, \lambda_i + 2^i\}$, $0 \leq i \leq m - 2$, and $\{\lambda_{m-1}, \lambda_{m-1} + \gamma_{m-1}\}$ with $\gamma_{m-1} = \Gamma_{(m)} + 1 - 2^{m-1}$.

Proof: We have $2^{m-1} \leq \Gamma_{(m)} = \Sigma_{0\leq i<m}\, \gamma_i \leq 2^m - 1$ by the Theorem's conditions. We choose λ_i, for $0 \leq i \leq m - 1$, such that $\Lambda_{(m)} = \Sigma_{0\leq i\leq m-1}\, \lambda_i$. Such a set of m twits represents the interval $[\Lambda_{(m)}, \Lambda_{(m)} + \Gamma_{(m)}]$ for it meets the conditions of Corollary 7.4. We prove that m is minimal by contradiction. Suppose there is a collection of $(n < m)$ twits with gaps in descending order $\{\gamma'_{n-1}, \ldots, \gamma'_1, \gamma'_0\}$, collectively representing $[\Lambda_{(m)}, \Lambda_{(m)} + \Gamma_{(m)}]$. Then, we have: $2^{m-1} \leq \Sigma_{0\leq i<n}\gamma'_i = \Gamma_{(m)} \leq 2^m - 1$.

For the latter inequality to hold, there must be at least one twit gap satisfying $\gamma'_j > 2^j$. Assume that γ'_j is the first such twit gap (the one with the smallest index). But by Corollary 7.4 we have:

$$\gamma'_j \le 1 + \gamma'_0 + \gamma'_1 + \gamma'_2 + \ldots + \gamma'_{j-1} \le 1 + 1 + 2 + 4 + \ldots + 2^{j-1} = 2^j$$

The derived constraint $\gamma'_j \le 2^j$ clearly contradicts the requirement $\gamma'_j > 2^j$. □

Corollary 7.5 (Maximal efficiency twit set): A set of m equally weighted twits with gaps $\gamma_i = 2^i$, for $0 \le i \le m - 1$, represents an interval of 2^m integers with maximal encoding efficiency $e = 1$ (see Definition 7.4). □

Theorem 7.4 (WBS representation of intervals): An interval $[\Lambda^+, \Lambda^+ + \Gamma^+]$ of integer values containing $\Gamma^+ + 1$ consecutive integers is representable by a WBS encoding with redundancy pattern $\rho_{k-1} \ldots \rho_1\rho_0$ iff for all indices i in the range $0 < i < k$, we have $R_i \ge 0$.

Proof: The necessity part is easy to prove. If $R_i < 0$ for some i, then positions 0 to $i - 1$ collectively represent fewer than 2^i distinct values. At least one of the 2^i mod-2^i equivalence classes must be unrepresented among these values. Given that bits in positions i and higher can only represent multiples of 2^i, there must be gaps in the representation. We prove the sufficiency part by induction on k. Recall that m_{k-1} is nonzero by Definition 7.1. This leads to $m_0 > 0$, because either position 0 is the only position or else the assumed condition $R_i \ge 0$ guarantees $R_1 = \rho_0 = m_0 - 1 \ge 0$. The base case is $k = 1$; a one-position WBS representation with $m_0 > 0$ covers all integers from Λ_0 to $\Lambda_0 + \Gamma_0$. Suppose that the theorem holds for any WBS representation with at most $k-1$ positions. Let a k-position representation Ω be obtained by extending a $(k - g)$-position representation, where $g \ge 1$, with $m_{k-1} > 0$ and $m_j = 0$ for $k - g \le j < k - 1$, that is, assume that the leftmost g components of redundancy pattern are ρ_{k-1} -1 -1 . . . -1. Then, by our assumptions, $R_{k-1} = R_{k-2} = \ldots = R_{k-g} \ge 0$. In particular, $R_{k-1} = (-1\ -1 \ldots -1\ \rho_{k-g-1} \ldots \rho_1\ \rho_0)_{\text{two}} \ge 0$ leads to $-2^{k-1} + 2^{k-g} + R_{k-g} \ge 0$, or $R_{k-g} + 2^{k-g} \ge 2^{k-1}$. This implies that the interval represented by the rightmost $k - g$ positions of Ω contains at least 2^{k-1} consecutive values. These values combined with multiples of 2^{k-1} representable by the bit(s) in position $k - 1$ yield a continuous interval of integers overall. □

Theorem 7.5 (Uniqueness of redundancy pattern for strongly equivalent canonical WBS encodings): Any WBS encoding with total redundancy index R and the redundancy pattern $\rho_{k-1} \ldots \rho_1\rho_0$ satisfying $R_i \ge 0$ for $0 < i < k$, and thus representing a continuous interval of integers by Theorem 7.1, is strongly equivalent to one or more canonical WBS encodings with a common redundancy pattern and the same total redundancy index R.

Proof: We describe the process for deriving a canonical encoding from a given WBS encoding. Scan the redundancy indices ρ_i from the right until you find $\rho_j \geq 2$ for some $j < k$–1. If no such position exists, the encoding is already in the desired canonical form; we will show later that $\rho_i \geq 0$ for $0 \leq i \leq j - 1$. If you find $\rho_j \geq 2$, take three of the bits in position j and make the substitution shown in Fig. 7.6. This does not change the set of values representable (which preserves the total redundancy index R) and reduces ρ_j by 2. Repeating this process eventually leads to $\rho_j \leq 1$ for $0 \leq j < k - 1$. To show that the resulting redundancy indices satisfy $\rho_j \geq 0$, $0 \leq j < k - 1$, we note that $R_j = (-1\ \rho_{j-2}\ .\ .\ .\ \rho_0)_{\text{two}}$ has a value of –1 when all the redundancy indices assume the maximal value of 1. We can prove the uniqueness of the redundancy pattern by contradiction. Suppose that another equivalent canonical encoding with a different redundancy pattern exist, and let l be the leftmost (most significant) position in which redundancy indices differ. If R' is the total redundancy index for this second canonical encoding, $R - R'$ (that is, the difference between the sizes of intervals representable by the two encodings) will be nonzero, given that $R - R' \geq 2^l - (1\ 1\ .\ .\ .\ 1)_{\text{two}} = 1$. □

Corollary 7.6 (WBS redundancy): A given k-position WBS encoding is *redundant* iff in any of its strongly equivalent canonical forms, $\rho_j > 0$ for some $j < k$. □

Theorem 7.6 (Efficiency of canonical WBS encodings): Among all strongly equivalent WBS encodings, canonical encodings have the highest encoding efficiency.

Proof: We show, by contradiction, that the encoding cost $E = \sum_{0 \leq i < k} m_i$ is minimal for canonical encodings. If a canonical encoding does not have the lowest cost among all strongly equivalent WBS encodings, uniqueness of the redundancy pattern for canonical encodings implies that the lowest-cost strongly equivalent encoding must be noncanonical. This is impossible, however, because the process of transforming a WBS encoding to a canonical form (described in the proof of Theorem 7.5) is solely composed of repeated applications of the substitutions shown in Fig. 5, and each such substitution reduces the encoding cost E by 1. □

Theorem 7.7 (Canonical encoding with a given range): For an interval $[-N, P]$ of integers, that includes 0, and integer k in $[1, \lceil \log_2 (N + P + 1) \rceil]$, a k-position canonical WBS encoding representing exactly $[-N, P]$ exists.

Proof: A trivial one-position WBS encoding with the given range has N negabits and P posibits, and $R = m_0 - 1 = N + P - 1$. A k-position canonical encoding equivalent to the above can be easily derived by the construction of Theorem 7.5. □

Corollary 7.7 (WBS encoding for a GSD representation): For any radix-2^h GSD number system with the digit set $[\alpha, \beta]$, there exists a periodic canonical WBS encoding with period h, where $1 \leq h \leq \lceil \log_2(\beta - \alpha + 1) \rceil$.□

Chapter 8 Suitable Number Systems and Encodings

We have studied in Chapters 2 to 7, a variety of redundant number systems, and their representations. We now examine them against the desired properties for a suitable number system for a general purpose carry-free arithmetic environment. The desired properties, besides minimal cost and delay, are maximal encoding efficiency, representational closure, digit set preservation, and symmetry, as defined in Definitions 1.5 to 1.8. Regularity of VLSI design is also an important desired property, which is fortunately shared by all of our high level designs based on standard full and half adders. Table 8.I shows a comparison of several redundant number representations, where the period for **boldface** entries is $h-1$, and is h otherwise. The contents of the first three columns come from our discussion on various number systems/representations and their implementations in previous chapters. We discuss the last four columns below.

Table 8.I Comparison of redundant number representations.

System	**Closed**	**Preserved**	**Digit set Δ**	ξ	**Delay** coefficient	**Cost** coefficient	**Subtraction delay penalty**
Sign magnitude SD	Yes	Yes	$[-r+1, r-1]$	2	$4^{+}\log_2 h$	$1^{++}h$	XOR
2's complement SD	Yes	Yes	$[-r+1, r-1]$	2	$2^{+}\log_2 h$	$1^{+}h$	XOR
HSD [Phat94]	Yes	Yes	$[-r/2, r-1]$	1	h	$1^{+}h$	XOR^{+}
SDB hybrid [Phat01]	No	Yes	$[-r, r-1]$	2	h	$1^{+}h$	XOR^{+}
SDB hybrid [Jab05 a]	No	Yes	$[-r, r-1]$	2	$1^{+}\log_2 h$	$1^{+}h$	XOR^{+}
SDB hybrid [Jab05 a]	Yes	Yes	$[-r, r-1]$	2	$1^{+}\log_2 h$	$2^{+}h$	XOR^{+}
Augmented SDB hybrid	Yes	Yes	$[-r, r-1]$	4	$1^{+}\log_2 h$	$2^{+}h$	XOR^{+}
Stored BSD transfer	Yes	Yes	$[-r/2-1, r/2]$	2^{-k}	$1^{+}\log_2 h$	$2^{+}h$	XOR^{+}
Stored shifted unibit transfer (SUT)	Yes	Yes	$[-3r/2, -r/2-1]$, $[r/2, 3r/2-1]$	3	$1^{+}\log_2 h$	$2^{+}h$	XOR
Stored SBC transfer	Yes	No	$[-r/2-1, r/2+1]$	2^{-k}	$1^{+}\log_2 h$	$2^{+}h$	XOR^{+}
Stored $\{-1, 2\}$ transfer	Yes	No	$[-r/2-1, r/2+1]$	1	$1^{+}\log_2 h$	$2^{+}h$	XOR^{+}
Stored posibit transfer	No	Yes	$[-r/2, r/2]$	1	$1^{+}\log_2 h$	$1^{+}h$	XOR
Stored posibit transfer	Yes	Yes	$[-r/2, r/2]$	1	$1^{++}\log_2 h$	$2.5^{+}h$	XOR

8.1 Representational power

Without loss of generality, and for the ease of comparison we assume periodic number systems, where each number is represented by k digits, and each digit is represented by $h + 1$ twits. In the number representations we have studied, each digit has 1 (2) redundancy twit(s). This remains h ($h - 1$) twits in h ($h - 1$) consecutive positions for the rest of each digit, which leads to 2^h (2^{h-1}), for the representation's radix, and h ($h - 1$) for the period. Therefore the total number of twits are the same (i.e., $k(h + 1)$) in all the representations. For a better comparison of the *representational powers* of the number systems studied, we define the representational power ratio ξ as the ratio of the cardinality of a number system to the cardinality of a kh bit nonredundant number system (i.e., 2^{kh}).

Definition 8.1 (Cardinality): The cardinality of a WBS-like digit set $\Delta = [-n, p]$ is $|\Delta| = n + p + 1$. The cardinality of a periodic k digit radix-r number representation with digit set Δ is $|\Delta^k| = N + P + 1$, where $P = p\Sigma_{i=0}^{k-1} r^i$ ($-N = -n\Sigma_{i=0}^{k-1} r^i$), is the most positive (negative) number represented. ■

Definition 8.2 (Unit digit value): The *Unit digit value*, of a periodic k digit radix-r number representation is the value $\upsilon = \Sigma_{i=0}^{k-1} r^i$ of a k digit radix-r number, where the value of each digit is 1. ■

Using Definitions 8.1 and 8.2, we derive the following:

$$|\Delta^k| = N + P + 1 = (n + p)\upsilon + 1 = 1 + (|\Delta| - 1)\upsilon.$$

For ease of comparison between symmetric and asymmetric number systems, we use cardinality of the maximum symmetric range.

Definition 8.3 (Cardinality of the maximum symmetric range): The maximum symmetric range of a digit set $\Delta = [-n, p]$ is $\Delta_s = [-min(n, p), min(n, p)]$ as was per Definition 1.9. The maximum symmetric range of a periodic k-digit radix-r number representation is likewise defined as $\Delta^k_s = [-min(N, P), min(N, P)]$. The cardinalities of Δ_s and Δ^k_s are $|\Delta_s| = 2\, min(n, p) + 1$ and $|\Delta^k_s| = 2\, min(N, P) + 1$. ■

Given that $min(N, P) = min(n, p)\,\upsilon$, the symmetric range cardinality is derived as:

$$|\Delta^k_s| = 2\, min(n, p)\,\upsilon + 1 = 1 + (|\Delta_s| - 1)\upsilon.$$

Definition 8.4 (Representational power coefficient for the maximum symmetric range): The *representational power coefficient* ξ for the maximum symmetric range of a periodic k-digit radix-r number is the ratio of the symmetric range cardinality $|\Delta^k_s|$, and the cardinality of a nonredundant kh bit number representation, which is 2^{kh}, leading to $\xi = |\Delta^k_s|/2^{kh}$. ■

Combining the equations for $|\Delta^k_s|$ and ξ, with $\upsilon = \Sigma_{i=0}^{k-1} r^i = (r^k - 1)/(r - 1)$ and $r = 2^h$ leads to:

$$\xi = r^{-k} + (|\Delta_s| - 1)(1 - r^{-k})/(r - 1).$$

For $r = 2^{h-1}$, ξ is divided by 2^k. We derive the approximate ξ values for different cases of Table 8.I, as follows:

- Signed digit and SDB hybrid number systems: The symmetric range for the signed digit number system (First two entries) and the SDB hybrid representations is $\Delta_s = [-2^h +1, 2^h-1]$, with $|\Delta_s| = 2^{h+1} - 1 = 2r - 1$, leading to:

$$\xi = r^{-k} + (2r - 2)\,(1 - r^{-k}) \,/\, (r - 1) = 2 - r^{-k} \cong 2.$$

- HSD, stored BSD transfer, and stored posibit number systems: The symmetric range for These three cases is $\Delta_s = [-2^{h-1}, 2^{h-1}]$, with $|\Delta_s| = 2^h + 1 = r + 1$, leading to:

$$\xi = r^{-k} + r\,(1 - r^{-k}) \,/\, (r - 1) = 1 + (1 - r^{-k}) \,/\, (r - 1) \cong 1.$$

- Stored {-1, 2} transfer and stored SBC transfer representations: In these cases, with period h and $h - 1$, we have $|\Delta_s| = 2^{h-1} +3 = r + 3$, and $|\Delta_s| = 2^h + 1 = r + 1$, leading to $\xi \cong 1$ and $\xi \cong 2^{-k}$, respectively. The considerable degradation of the representational power, in the latter case, is due to an extra redundancy twit per digit.

- SUT: Definitions 8.1 and 8.2 do not apply for the SUT case, for the digit set is not continuous, and is actually composed of two intervals $[-3\times2^{h-1}, -2^{h-1}-1]$ and $[2^{h-1}, 3\times2^{h-1}-1]$, as depicted by the solid lines in Figure 8.1. But note that the SUT number representation as a whole represents an interval of integers. Therefore we derive the $|\Delta^k_s|$ directly as 2 *min* $(N, P) + 1$. There are a negabit in position $h-1$ and a unibit (negabit in MSD) in position h of each digit, leading to $N = (2^h + 2^{h-1})\upsilon$. For the positive range, observe that there are $h-1$ posibits in positions 0 to $h-2$ and a unibit (posibit in MSD) in position h of each digit leading to $P = (2^h + 2^{h-1} - 1)\upsilon = N - \upsilon$, and thus *min* $(N, P) = P$. Therefore we have $|\Delta^k_s| = 2\,(2^h + 2^{h-1} - 1)\,\upsilon + 1 = (3r - 2)\upsilon$, which leads to $\xi = 3 - 2r^{-k} + (1 - r^{-k}) \,/\, (r - 1) \cong 3$.

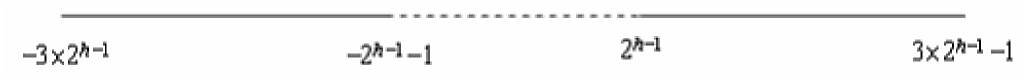


Fig. 8.1 The noncontiguous digit set for the SUT representation

For a fair comparison of the representational powers of different redundant representations, we should allow for the possibility of augmenting other representations, with an extra twit in position kh, as in SUT. But it can be easily verified that the only other representation where the augmentation increases the symmetric range is the SDB hybrid, which may be augmented by a unibit in position kh, for which case we have added an entry in Table 8.I, under the name *Augmented SDB hybrid*. For this case we have $N = 2^h\,\upsilon + 2^{kh}$ and $P = (2^h - 1)\,\upsilon + 2^{kh}$, leading to *min* $(N, P) = P$ and $|\Delta^k_s| = 2(2^h -1)\,\upsilon + 2^{kh+1} + 1 = 4\times2^{kh} - 1$, leading to $\xi \cong 4$.

8.2 Delay (cost) comparison

For easy comparison of the addition delay (cost), for redundant representations studied, we use the *delay* (*cost*) *coefficient*, which is the approximate ratio between the actual delay (cost) and the delay (cost) of one full adder. For example for an h-bit carry propagate adder where the critical path of the addition logic consists of h full adders, the delay (cost) coefficient is h (h). When a logarithmic carry accelerating logic (such as carry look ahead) is used, it is $1^+\log_2 h$ ($1^{++}h$). We use i^+ ($i > 0$) to indicate more than i times and less than $i + 1$ times. Similarly i^{++} indicates more than i^+ times, but still less than $i + 1$ times. For comparison of the subtraction delay, we consider only the delay penalty imposed for complementation. For example in a nonredundant subtraction the penalty is equal to that of the delay of an XOR gate used for complementation of each bit. In the following, we analyze the delay (cost) coefficients of Table 8.I:

- **Simple signed digit (first two entries)**: 4^+ and 2^+, in the column for delay coefficient, come from the relevant high radix coefficients computed in Chapter 2 (Table 2.V). For both entries carry acceleration logic is applicable, where some extra control logic is necessary for the sign magnitude paraidgm. The delay penalty for subtraction is equal to that of an XOR gate used for complementing the sign bits in sign magnitude representation, and complementing all the bits in two's complement representation.
- **Hybrid redundant (old implementations)**: The carry acceleration techniques are not directly applicable in the implementations given in [Phat94] and [Phat01], which are based on look-back mechanism and both carry and borrow propagation. Therefore the delay coefficients in these cases are linearly proportional to h. The adder cells in the old implementations of hybrid redundancy are more complex than the new full adder implementations of [Jab05 a]; hence the 1^+h cost coefficients. Subtraction of hybrid redundant numbers has not been discussed by Phatak and Koren [Phat94, Phat01]. But it is easy to see that besides the XOR gate per bit, needed for bit-wise complementation, the following increment operation (as a part of two's complementation), complicates the derivation of the carry out of a redundant position; hence XOR$^+$ in subtraction penalty column.
- **Hybrid redundant (new implementations and augmented entries)**: Before applying carry acceleration, we have the delay of one level of full adders (Figure 6.16) hence the $1^+\log_2 h$ entries. The 2^+h entries in the cost coefficient column are due to the second row full adders in nonredundant positions of representationally closed implementations. Subtraction starts with reduction of a 5-deep WBS number, which requires for one extra full adder in redundant positions; hence XOR$^+$ for subtraction penalty.
- **Stored transfer entries**: The $1^+\log_2 h$ and 2^+h entries, are justified as in the previous case. The 1^+h entry, for the representationally unclosed stored posibit transfer case is due to the delay of the universal addition scheme used here. The $1^{++}\log_2 h$ and 3^+h entries for representationally closed stored posibit transfer case is due to one extra level of full adders (Figure 6.22). The increment part of two's complementation does not complicate the subtraction operation in the SUT and stored posibit transfer cases; hence simple XOR entries.

8.3 Choosing the best number system

Where the representational closure property is not required, the best choice in Table 8.I is naturally the one before the last entry, where the symmetric stored posibit transfer representation with the universal addition scheme exhibits the lowest cost and fastest carry-free arithmetic.

The best choice, for a general purpose carry-free arithmetic environment, should support representational closure. Among the three symmetric entries of Table 8.I, with representational closure property (i.e., the stored SBC transfer, its two-valued version, and the stored posibit transfer), the stored posibit transfer number system exhibits, symmetry, digit set preservation, $\xi = 1$, regularity, and reasonable delay and cost. But the two cases with minimally asymmetric digit sets and highest representational power coefficients (i.e., the augmented SDB hybrid and the SUT number systems) offer better figures. They both provide for much wider symmetric range, less delay, and less cost. The advantage of the augmented SDB hybrid is its wider symmetric range ($\xi = 4$), but the SUT number system ($\xi = 3$) shows less delay, less cost, and less subtraction delay and cost penalty (details in Chapter 7). These observations tend to distinguish three of the representations as the best choices, namely:

- Stored posibit transfer (SPT)
- Stored shifted unibit transfer (SUT)
- Augmented SDB hybrid redundant

In Chapter 9, we study WBS-like multiplication and division, with examples from the above choices. Note that addition and subtraction operations for these number representations were studied, in detail, in Chapter 5, 6, and 7. Here we study the problem of conversion between two's complement and any of the three representations, and vise versa.

8.4 Conversion of WBS-like representation to (from) two's complement

Conversion of WBS-like numbers to canonical WBS encoding was discussed in Chapters 4 and 7. The particular 2-deep bit pattern of each canonical WBS encoding may lead to its special conversion method to and from two's complement representation. We show, below, three special conversion methods for our three selected encodings.

8.4.1 Conversion of stored posibit transfer to (from) two's complement

We show, in Fig. 8.2, the steps of conversion of a four digit radix-16 stored posibit transfer encoding to its equivalent two's complement representation. In the first step we replace each negabit X'_j (except for the MSB), by a posibit x'_j, with the same logical value, and an equally weighted constant -1. Then we replace each -1, with a string of a -1, followed by 1111. Clearly these conversions do not change the value of the number as a whole.

$$\begin{array}{cccccccccccccccc}
X'_{15} & x'_{14} & x'_{13} & x'_{12} & X'_{11} & x'_{10} & x'_{9} & x'_{8} & X'_{7} & x'_{6} & x'_{5} & x'_{4} & X'_{3} & x'_{2} & x'_{1} & x'_{0} \\
 & & & x''_{12} & & & & x''_{8} & & & & x''_{4} & & & & x''_{0} \\
\hline
X'_{15} & x'_{14} & x'_{13} & x'_{12} & x'_{11} & x'_{10} & x'_{9} & x'_{8} & x'_{7} & x'_{6} & x'_{5} & x'_{4} & x'_{3} & x'_{2} & x'_{1} & x'_{0} \\
 & & & x''_{12} & -1 & & & x''_{8} & -1 & & & x''_{4} & -1 & & & x''_{0} \\
\hline
X'_{15} & x'_{14} & x'_{13} & x'_{12} & x'_{11} & x'_{10} & x'_{9} & x'_{8} & x'_{7} & x'_{6} & x'_{5} & x'_{4} & x'_{3} & x'_{2} & x'_{1} & x'_{0} \\
-1 & 1 & 1 & x''_{12} & & 1 & 1 & x''_{8} & & 1 & 1 & x''_{4} & 1 & & & x''_{0} \\
 & & & 1 & & & & 1 & & & & 1 & & & &
\end{array}$$

Fig. 8.2 The first two steps of the conversion of an SPT encoding to 2's complement

Now we can derive the result by a conventional carry look-ahead adder with block size 4, but with simplified logic due to special 0 and 1 constants of the second component. The simplified carry propagate (p_i), carry generate (g_i), positional carry (c_i), interim sum (w_i), block carry propagate (P_i), and block carry generate (G_i) signals are:

$$p_0 = x'_0 + x''_0,\ g_0 = x'_0x''_0,\ c_0 = 0,\ w_0 = x'_0 \oplus x''_0,$$
$$p_1 = x'_1,\ g_1 = 0,\ c_1 = x'_0x''_0,\ w_1 = x'_1,$$
$$p_2 = x'_2,\ g_2 = 0,\ c_2 = x'_0x''_0x'_1,\ w_2 = x'_2,$$
$$p_3 = 1,\ g_3 = x'_3,\ c_3 = x'_0x''_0x'_1x'_2,\ w_3 = !x'_3,$$
$$p_i = 1,\ g_i = x'_i + x''_i,\ w_i = !(x'_i \oplus x''_i), \text{ for } i = 4, 8, \text{ and } 12,$$
$$p_i = 1,\ g_i = x'_i,\ c_i = c_{i-1} + x'_{i-1} + x''_{i-1},\ w_i = !x'_i, \text{ for } i = 5, 9, \text{ and } 13,$$
$$p_i = 1,\ g_i = x'_i,\ c_i = c_{i-2} + x'_{i-2} + x''_{i-2} + x'_{i-1},\ w_i = !x'_i, \text{ for } i = 6, 10, \text{ and } 14,$$
$$p_i = x'_i,\ g_i = 0,\ c_i = c_{i-3} + x'_{i-3} + x''_{i-3} + x'_{i-2} + x'_{i-1},\ w_i = x'_i, \text{ for } i = 7 \text{ and } 11,$$
$$p_i = X'_i,\ g_i = 0,\ c_i = c_{i-3} + x'_{i-3} + x''_{i-3} + x'_{i-2} + x'_{i-1},\ w_i = X'_i, \text{ for } i = 15,$$
$$P_0 = (x'_0 + x''_0)x'_1x'_2,\ G_0 = x'_0x''_0x'_1x'_2 + x'_3,\ c_4 = G_0,$$
$$P_1 = x'_7,\ G_1 = x'_7\,(x'_4 + x''_4 + x'_5 + x'_6),\ c_8 = G_1 + G_0\,x'_7,$$
$$P_2 = x'_{11},\ G_2 = x'_{11}\,(x'_8 + x''_8 + x'_9 + x'_{10}),\ c_{12} = G_2 + G_1\,x'_{11} + G_0\,x'_7\,x'_{11},$$
$$P_3 = X'_{15},\ G_3 = X'_{15}\,(x'_{12} + x''_{12} + x'_{13} + x'_{14}).$$

The encoding of –1 in position 15, as a negabit, would be 0. The latest delivered signal is:

$$S_{15} = w_{15} \oplus c_{15} = X'_{15} \oplus (c_{12} + x'_{12} + x''_{12} + x'_{13} + x'_{14}) = c_{12}\ !X'_{15} + X'_{15}\ !x!c_{12} + x\ !X'_{15},$$

where $x = x'_{12} + x''_{12} + x'_{13} + x'_{14}$. The latency of S_{15} is 2 gate levels more than that of c_{12}, which is 4 gate levels. Thus the total latency of the 4-digit radix-16 SPT to 2's complement converter amounts to 6 gate levels. Note that the latency of a 16-bit carry look-ahead adder with 4-bit blocks is 9 gate levels [Parh00].

Conversion of a two's complement number to its stored posibit transfer equivalent is easy. In our four digit radix-16 example above, we need a negabit in positions $4j + 3$, instead of the posibits x'_{4j+3}, and an extra posibit in every position $4j$, for $0 \le j \le 3$. A posibit x_i, with value x_i may indeed be replaced by a negabit $!X_i$, with value $-x_i$ in the same position i, and a posibit x_i in position $i + 1$, with value $2x_i$, where the replacement does not change the represented value. Therefore the required conversion is done by only copying each posibit in positions $4j + 3$ into immediate next positions, and regarding the original as a negabit (see Fig. 3.5 for a justification).

8.4.2 Conversion of SDB hybrid and SUT to (from) two's complement

Figures 8.3 and 8.4 depict the first two steps of conversion to two's complement for augmented SDB hybrid and SUT numbers, respectively. Note that in Fig. 8.4 a unibit is replaced by a posibit with the same encoding in the next higher position and a –1 in the same position. The actual logical equations for the required carry look-ahead adders are similar to those in the previous section.

To convert a two's complement number to its equivalent radix-16 SDB hybrid representation, we simply insert a 1 as a zero valued negabit in positions $4j$ ($j > 0$) of the second component of the desired SDB hybrid encoding. For the MSD, we have a negabit in the MSB of the two's complement number, which should be replaced by a posibit in the same position and a negabit/unibit pair in the next higher position. For example a negabit y_{15}, of a 16-bit two's complement number, would be replaced by a posibit x'_{15}, and a negabit/unibit pair $X''_{16}/\underline{\boldsymbol{X}}''_{16}$, in the next higher position, where $x'_{15} = X''_{16} = !\underline{\boldsymbol{X}}''_{16} = y_{15}$. Table 8.II justifies the replacement. Note that conventional encoding of negabits has been assumed for y_{15}.

$$\begin{array}{ccccccccccccccccc}
\underline{\boldsymbol{X}}'_{16} & x'_{15} & x'_{14} & x'_{13} & x'_{12} & x'_{11} & x'_{10} & x'_{9} & x'_{8} & x'_{7} & x'_{6} & x'_{5} & x'_{4} & x'_{3} & x'_{2} & x'_{1} & x'_{0} \\
X''_{16} & & & & X''_{12} & & & & X''_{8} & & & & X''_{4} & & & & \\
\hline
\underline{\boldsymbol{X}}'_{16} & x'_{15} & x'_{14} & x'_{13} & x'_{12} & x'_{11} & x'_{10} & x'_{9} & x'_{8} & x'_{7} & x'_{6} & x'_{5} & x'_{4} & x'_{3} & x'_{2} & x'_{1} & x'_{0} \\
X''_{16} & & & & x''_{12} & & & & x''_{8} & & & & x''_{4} & & & & \\
 & & & & -1 & & & & -1 & & & & -1 & & & & \\
\hline
X'_{16} & x'_{15} & x'_{14} & x'_{13} & x'_{12} & x'_{11} & x'_{10} & x'_{9} & x'_{8} & x'_{7} & x'_{6} & x'_{5} & x'_{4} & x'_{3} & x'_{2} & X'_{1} & x'_{0} \\
X'_{16} & 1 & 1 & 1 & x''_{12} & 1 & 1 & 1 & x''_{8} & 1 & 1 & 1 & x''_{4} & & & & \\
X''_{16} & & & & & & & & & & & & 1 & & & &
\end{array}$$

Fig. 8.3 First 2 steps of the conversion of an augmented SDB hybrid encoding to 2's complement

$$\begin{array}{ccccccccccccccccc}
X'_{16} & X'_{15} & x'_{14} & x'_{13} & x'_{12} & X'_{11} & x'_{10} & x'_{9} & x'_{8} & X'_{7} & x'_{6} & x'_{5} & x'_{4} & X'_{3} & x'_{2} & x'_{1} & x'_{0} \\
X''_{16} & & & & \underline{\boldsymbol{X}}''_{12} & & & & \underline{\boldsymbol{X}}''_{8} & & & & \underline{\boldsymbol{X}}''_{4} & & & & \\
\hline
X'_{16} & x'_{15} & x'_{14} & x'_{13} & x'_{12} & x'_{11} & x'_{10} & x'_{9} & x'_{8} & x'_{7} & x'_{6} & x'_{5} & x'_{4} & x'_{3} & x'_{2} & X'_{1} & x'_{0} \\
x''_{16} & -1 & & x''_{12} & -1 & -1 & & x''_{8} & -1 & -1 & & x''_{4} & -1 & -1 & & & \\
\hline
X'_{16} & x'_{15} & x'_{14} & x'_{13} & x'_{12} & x'_{11} & x'_{10} & x'_{9} & x'_{8} & x'_{7} & x'_{6} & x'_{5} & x'_{4} & x'_{3} & x'_{2} & X'_{1} & x'_{0} \\
X''_{16} & & 1 & X''_{12} & & & 1 & x''_{8} & & & 1 & x''_{4} & & 1 & & & \\
 & & & 1 & & & & 1 & & & & 1 & & & & &
\end{array}$$

Fig. 8.4 The first two steps of the conversion of an SUT encoding to 2's complement

Table 8.II Conversion of the MSB of a 2's complement number to the most significant hybrid digit and the augmented unibit of the equivalent augmented SDB hybrid encoding

y_{15}	Value	X'_{15}	$\underline{\boldsymbol{X}}''_{16}$	X''_{16}
0	0	0	1	0
1	–1	1	0	1

Next we consider the conversion of a 16-bit two's complement number to its equivalent 4 digit radix-16 SUT number. Here we need to replace $y_{4j}y_{4j-1}$ bit pairs of the two's complement number ($j > 0$) by a posibit x'_{4j}, a unibit $\underline{\boldsymbol{X}}''_{4j}$, and a negabit X'_{4j-1}. Table 8.III helps in deriving the required logic equations:

$$x'_{4j} = !(y_{4j} \oplus y_{4j-1}),\ \underline{\boldsymbol{X}}''_{4j} = y_{4j} + y_{4j-1},\ X'_{4j-1} = !y_{4j-1}.$$

For the MSB of the 2's complement number, we just invert it due to biased encoding of the target. The Most significant transfer will be 0 (i.e., a negabit 1 and a posibit 0).

Table 8.III Conversion of two's complement to SUT

y_{4j}	y_{4j-1}	Value	x'_{4j}	$\underline{\boldsymbol{X}}''_{4j}$	X'_{4j-1}
0	0	0	1	0	1
0	1	1	0	1	0
1	0	2	0	1	1
1	1	3	1	1	0

8.5 Floating Point Redundant Arithmetic

Operations on floating point operands, normally, consist of few steps, some of which could possibly execute in parallel. Floating point addition/subtraction consists of equalizing the exponent components of the two operands, adjusting the significand of the operand whose exponent has been changed, by an appropriate shift operation, adding the significands, and normalizing the result. Floating point multiplication (division), requires performing multiplication (division) on the significands, and addition (subtraction) of exponents, followed by a post normalization of the result.

In a floating point redundant number both the significand and the exponent may be kept in redundant form. For the execution efficiency of the shifts required for exponent equalization and the post normalization operations, a sound decision is to choose the base of the exponent of the floating point representation equal to the radix of underlying redundant representation. The reason is that a one binary position shift, by standard shift registers, on WBS-like operands does not preserve the representational closure property. But an h binary positions shift is representationally closed, where the base of the exponents and the radix of the underlying periodic WBS-like encoding are both equal to 2^h (More on shifts in Section 9.4).

Equalizing the exponents requires exponent comparison, and post normalization operation needs detection of the first nonzero-digit of the significand. Embedded in both operations is the comparison of a redundant digit with zero. We take up the zero detection issue in Section 9.2.

8.6 Summary

In this chapter we selected three WBS-like encodings as suitable number representation for representationally closed redundant arithmetic. The selection criteria besides the speed and cost considerations were symmetry, digit-set preservation, representational closure, and representational power. The three selected encodings, namely the SPT, SUT, and augmented SDB hybrid, all show maximal speed and lowest cost as compared to other possible choices enumerated in Table 8.I. They all, exhibit representational closure and digit-set preservation properties. The stored posibit transfer encoding is fully symmetric, while the symmetric range representational power of the SUT and augmented SDB hybrid are 3 and 4 times more, respectively. The latter figures should not be misleading in comparison of representational powers. Augmenting the stored posibit transfer encoding by 1 twit in position *kh*, does not increase the symmetric range, unless we augment with a unibit, in which case the number system would become noncontiguous. But augmenting with a negabit/posibit pair, while preserving contiguity, would raise the representational power to 4, i.e., it can bit the SDB hybrid encoding if one extra bit in the whole encoding is tolerated.

Chapter 9 WBS-Like Multiplication & Division

Redundant number representations have been widely used for representing intermediate results of a multiplication or a division operation, while the original operands and the result may be nonredundant. Stored carry and binary signed digit representations are usually used to represent intermediate results in partial product reduction [Gonz00]. Producing redundant quotient digits and redundant partial remainders is common in fast division techniques [Parh00]. In this Chapter we study multiplication and division of WBS-like numbers, and compare the performance of our methods with that of conventional nonredundant multiplication and division algorithms.

9.1 WBS-like multiplication

Multiplication operation is generally composed of three steps performed in a sequence:

- Derivation of partial products
- Reduction of partial products
- Computing the result by a final addition

We study WBS-like multiplication, for each of the three parts in the following sections

9.1.1 Partial product derivation for WBS-like operands

Derivation of partial products may be achieved through twit by twit multiplication (e.g., the AND matrix of conventional multipliers), digit by digit multiplication (e.g., a decimal digit multiplier), or by a table look-up for digit multipliers. WBS-multiplication does not introduce any special problem for the latter method. For the other two methods, we need to design special twit multipliers for each kind of twit pairs. Since a WBS-like number can be encoded as an equivalent WBS number, we need to design only three gates (or Boolean element creators, as named in [Flyn01]) as in Figure 9.1, where the outputs of the gates a, b, and c, are posibit, negabit, and posibit respectively, also shown in Table 9.I for multiplication of twits T_1 and T_2.

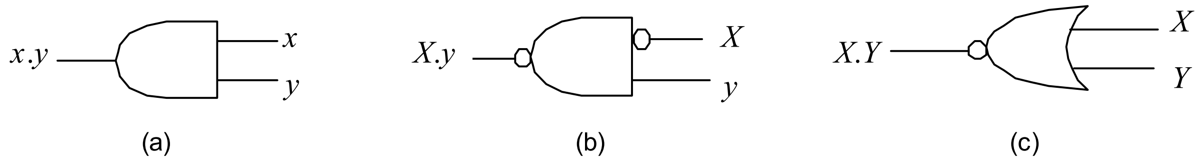


Fig. 9.1 Basic gates for derivation of the partial products.

Table 9.I. Multiplication of two twits

T_1	T_2	T_1T_2
●	●	●
●	○	○
○	○	●

9.1.2 Partial product reduction

In the second step of a WBS-like multiplication, we have a twit matrix composed of posibits and negabits. A conventionally encoded negabit has been named, elsewhere, as a signed Boolean element, and a partial product array (PPA) of signed and unsigned Boolean elements as signed PPA [Flyn01]. To reduce a signed PPA, special adders has been designed [Peza71, Schw91, Schw92, Mand96], specially in the context of function approximation using multipliers. The principal idea for reduction of a signed PPA [Flyn01] is similar to our method for converting a WBS encoding to an equivalent 2CL encoding described in Theorem 4.6. But here we take advantage of inverted encoding of negabits, which leads to a more efficient signed PPA reduction.

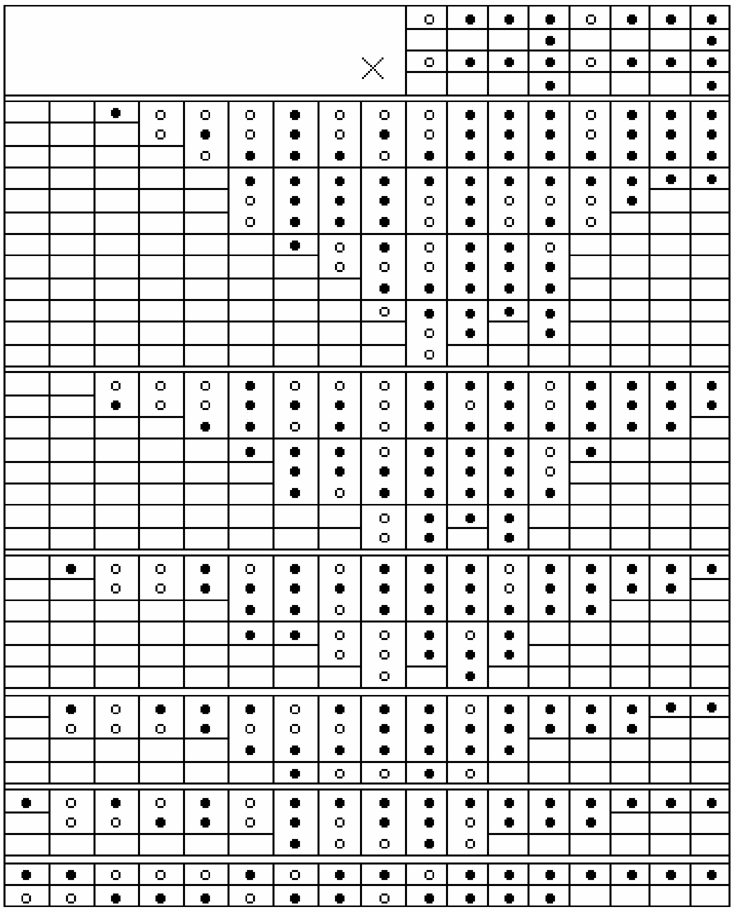


Fig. 9.2 Canonical partial product reduction to a 2-deep result

A signed PPA is, in fact, a WBS encoding of the product, which should be reduced to a 2-deep equivalent WBS encoding. Recalling Theorem 7.1 (Twit FA) and Corollary 6.2 (Twit compressors), we can reduce the partial products, pretty much the same way as in standard partial product reduction methods [Parh00]. Figure 9.2 depicts, in our extended dot notation, the first and second parts of multiplication of two-digit radix- 16 stored posibit transfer (SPT) numbers. The second box shows the non reduced partial product. The next four boxes show the hierarchy of partial product reduction leading to a 2-deep result in the 7th box. The partial products are computed by a program (see appendix 1), where the inputs to full/half adders are always picked from top to bottom. We name such a partial product reduction as a *canonical reduction*. The partial product reduction takes five levels, while the same process for multiplication of two 8-bit non-redundant binary numbers will take four levels. This means that the delay for partial product reduction in the redundant case is 25% more than that of the non redundant case. However we should note that the range of the integers represented by the redundant representation (i.e., [– 136, 136]) is 6.25% more than that of the non-redundant case (i.e., [0, 255] or [– 128, 127]). Moreover Table 9.II, shows that for very high radix cases, such as radix 256, the number of levels for both redundant and non-redundant operands are the same.

Table 9.II Comparison of partial product reduction levels

		Reduction Levels	
Digits	**Radix**	**Redundant**	**Nonredundant**
4	16	7	6
4	32	7	6
4	64	8	7
4	128	8	7
4	256	8	8

9.1.3 Derivation of the final product

The 2-deep WBS encoding of the product in Fig.9.2 is not equivalent to that of the multiplication operands. It may be converted to a desired WBS encoding following the process discussed in Chapter 4. However, to reduce the total delay of a representationally closed WBS multiplication, we may reconfigure the assignments of twits to full/half adders such that the reduction process directly leads to a 2-deep result with the same encoding as that of the operands.

Figure 9.3 depicts a heuristically non-canonical partial product reduction of the same multiplication as in Fig. 9.1, leading to a 2-deep WBS result in the 6th level, which is convertible to the desired encoding with only intra-digit carry propagation limited to the width of a radix-16 digit. The total delay for partial product reduction and conversion to the desired stored posibit transfer representation amounts to the delay of nine full adders, five for reduction to the level 6 and four full adders for deriving the most significant digit of the result. Note that the MSD is derived through a virtual borrow propagation, which is actually a carry propagation due to inverted encoding of negabits. In three places in the 5th level of reduction, we have used constant posibits (0) and negabits (1), without changing the value of the partial product in that level. A constant negabit in the 6th level has also helped in the derivation of the desired result. A similar design (i.e., with simple carry propagation in the last stage) for 8 bit nonredundant operands would result in a delay equal to that of 14 full adders and two half adders.

9.1.4 Booth recoding of the multiplier

Booth recoding [Boot51] and its variations has been used in many multiplier designs [Parh00], in order to reduce the number of originally generated partial products. In this section we examine a Booth recoding of stored posibit transfer numbers and its possible application in the design of redundant multipliers.

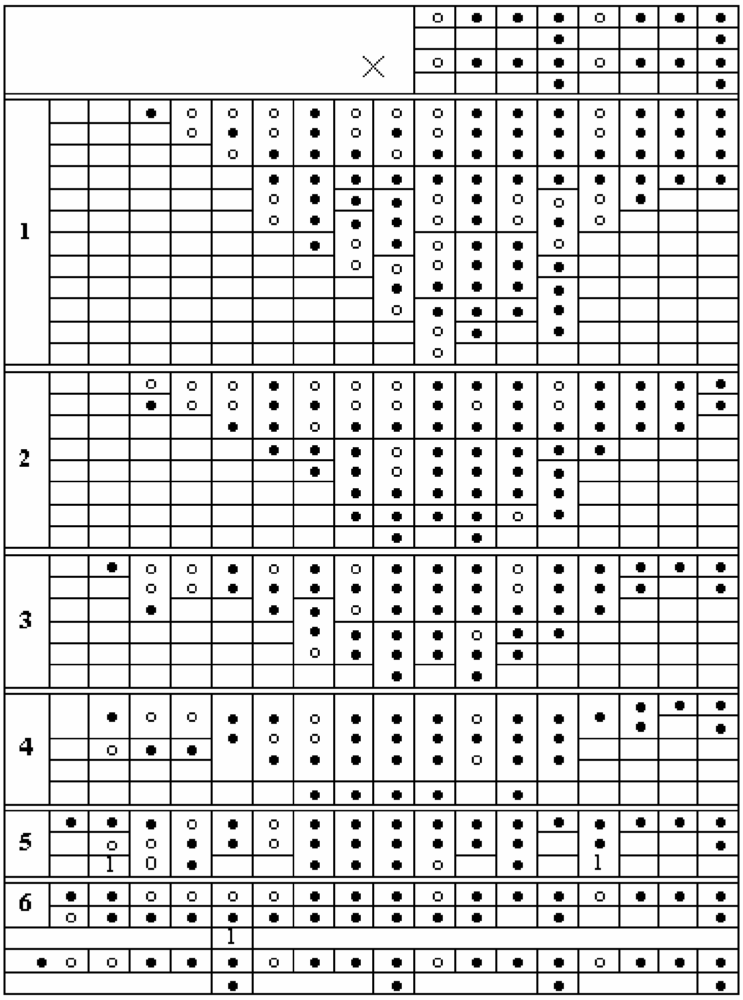


Fig. 9.3 Representationally closed WBS multiplication

In the conventional modified Booth recoding, a two's complement multiplier is converted to an equivalent minimally redundant radix-4 signed digit number [Parh00]. A stored posibit transfer number can similarly be converted. Fig. 9.4 depicts the conversion of a radix-256 stored posibit transfer digit to an equivalent four digit minimally redundant radix-4 signed digit number. Let $|x|$ denote the arithmetic value of twit *x*, and assume that twits with the same name irrespective of the letter case and font style (bold or underlined), have the same logical code in {0, 1}. For example the logical code for x'_1 and X'_1 are always the same, i.e., either both 0 or both 1, leading to $|!X'_j| = 1 - |X'_j| = 1 - (|x'_j| - 1) = - |x'_j|$.

In Figure 9.4 each posibit x'_{2i-1} in position $2i-1$, for $i = 1, 2, 3$, is replaced by a negabit $!X'_{2i-1}$, in the same position, with value $-|x'_{2i-1}|$, and same posibit in position $2i$, with value $2|x'_{2i-1}|$, such that the replacement does not change the total value represented. As is clear in Fig. 9.4 each radix-4 digit consisting of a negabit and two posibits in its immediate right position, is actually a minimally redundant radix-4 signed digit in $[-2, 2]$. It is easy to verify that a standard modified Booth recoder receiving the three twits of such a radix-4 digit produces the required signals needed for selecting the relevant multiples of the multiplicand. Nevertheless we provide the required logic in Fig. 9.6 in the next Section. The conversion of Fig. 9.4 is not actually needed as a pre-recoding operation; we just used it to explain the inputs to the recoder. Note that for any stored posibit transfer digit with $2j$ positions, the Booth recoding method, just described, will produce j multiples, which is equal to the number of multiples in the nonredundant case. But without Booth recoding, there would be $2j + 1$ multiples per one $2j$-position digit of the multiplier.

$$\begin{array}{cccccccc}
X'_7 & x'_6 & x'_5 & x'_4 & x'_3 & x'_2 & x'_1 & x'_0 \\
 & & & & & & & x''_0 \\
\hline
X'_7 & x'_6 & !X'_5 & x'_4 & !X'_3 & x'_2 & !X'_1 & x'_0 \\
 & x'_5 & & x'_3 & & x'_1 & & x''_0
\end{array}$$

Fig. 9.4 Conversion of a stored posibit transfer digit to radix-4 signed digits

We redesign the multiplier of Fig. 9.3, with Booth recoding of the multiplier. In the design time, we don't know which of the output signals of the Booth recoder would be selected for a given radix-4 digit of the multiplier. Therefore, we have to arrange the extended dot notation of partial products such that the reduction logic implemented based on our design would accept either multiples (i.e., zero, ± the multiplicand, or ± twice the multiplicand). Fig. 9.5a shows the first level partial products for the same multiplication as that of Fig. 9.3, but with Booth recoding applied. For better illustration, we have deliberately not filled the empty places among the dots of the first partial product by those of the second one, and so on. Fig. 9.5b shows the equivalent partial products with the gaps filled, leading to an 8-row partial product matrix. Note that the first level partial products of Fig 9.3 (without Booth recoding), had 12 rows for 8 partial products, but Fig. 9.5b has 8 rows for 4 Booth partial products. We will show, below, that for practically wider operands, the reduction ratio due to Booth recoding approaches the same value as for the nonredundant operands.

For wider operands, the Booth recoding of the redundant multiplier exhibits a much better effect on reduction of the rows in first level partial products, leading to less total reduction levels. Table 9.III compares the number of levels, derived by a simulation program (appendix 1), for nonredundant and redundant Booth multiplications. As is evident, for wider stored posibit transfer operands, Booth recoding is more beneficial. The reason is that in the nonredundant case, each two bits of the multiplier provide one Booth signal, while in the stored posibit transfer case, in the vicinity of redundant positions, each three twits provide one Booth signal. The latter observation leads to the pleasant result that for practical wide operands with sparse second component (e.g. 32 positions or more and one redundant position in 8), our stored posibit transfer multiplier design leads to less overall delay, compared to nonredundant multipliers.

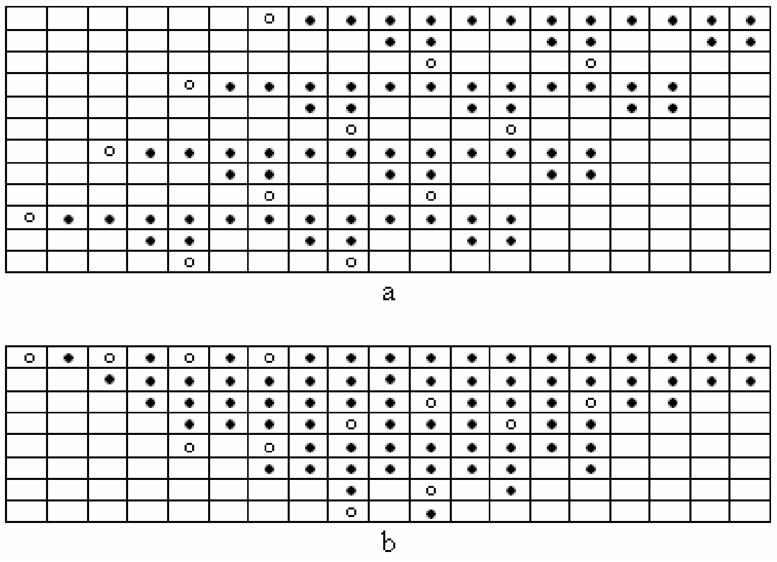


Fig. 9.5 The first level partial products of a nonredundant Booth multiplication

Table 9.III Comparison of partial product reduction levels, when Booth recoding applied

		Reduction Levels	
Digits	**Radix**	**Redundant**	**Nonredundant**
4	16	5	4
4	32	5	5
4	64	6	5
4	128	6	6
4	256	6	6

9.2 WBS-like Division

Division is generally believed to be the most complex arithmetic operation among the four basic ones. Several division methods, implemented by software, firmware, or hardware have been invented during past decades. Division methods may be divided in two basic categories, namely those based on repeated subtractions, and multiplicative methods. Both categories include radix-2 and higher-radix versions. The subtractive methods are further divided to restoring and nonrestoring methods, where the latter has synchronous and asynchronous (i.e., the well known SRT [Robe58] method) versions. The multiplicative methods, too, are divided into two groups; one converges, the divisor to 1 and the dividend to the desired quotient, through repeated multiplications of both the dividend and divisor by the same multipliers [Ande67]. The other one [Fowl89] computes the reciprocal of the divisor through, for example, extracting the route of equation $1/X - D = 0$, where D is the divisor. Extraction of the route X is normally done by repeated multiplication and addition operations, based on the Newton-Raphson iteration or through table look up [Parh87]. A final multiplication of the dividend by the reciprocal of the divisor derives the quotient. Both subtractive and multiplicative methods may use redundant representations, such as carry save, for intermediate results. A comprehensive coverage of all these methods may be found in [Erce94] and [Parh00]. Some advanced designs are offered in [Flyn01].

Our concern in this section is to design a representationally closed division method for redundant operands, where the dividend, divisor, quotient, and the remainder are all represented in the same redundant number system. Several approaches may be considered:

1) The simplest approach is to use the redundant to two's complement converter of Section 8.4, to convert both the dividend and divisor to their equivalent two's complement representations, and then use the most appropriate division hardware to derive the two's complement quotient and remainder. This should be followed by converting the results back to redundant encoding of the operands. The pre-division, redundant to two's complement, conversion, as discussed in Section 8.4, requires at most one add cycle, and the time for post-division conversion is negligible and can be part of the last cycle. Given that the conventional two's complement division requires several cycles, the extra one cycle for pre-division conversion may be justified in computations where division does not frequently occur, i.e. the extra cycle may be well paid for by the times saved in executing several non-division redundant operations between two divisions.

2) One may keep the dividend in the original redundant encoding, and only convert the divisor. The quotient digit selection in this case could be similar to the approach in [Parh00], where the partial remainders are kept in stored carry representation, and the number represented by only the few most significant positions of the partial remainder is enough for quotient digit selection. The extra add cycle, mentioned in the first approach is still needed for conversion of the divisor.

3) Looking for a division method which does not require the divisor to be restricted to two's complement representation, we picked an advanced algorithm in [Flyn01]. This algorithm is based on the following equation, where Z, D, D_h, and D_l, are the dividend, divisor, its most significant, and least significant halves respectively.

$$Z / D = Z / (D_h + D_l) = Z\,(D_h - D_l) / (D_h^2 - D_l^2)$$

If, as usual, we assume a normalized fractional devisor, D_l^2 in the denominator of the above equation would be negligible (reason to be given below), and can be omitted leading to the following simpler equation.

$$Z / D = Z(D_h - D_l) / D_h^2$$

We can look up for $1 / D_h^2$ in a table, while at the same time performing the computation of $Z\,(D_h - D_l)$, followed by a final multiplication. We can keep all the operands and intermediate results in redundant form, and derive the final quotient and remainder ($Z - QD$, where Q is the derived quotient) in redundant format as well. Note that no inter-digit carry propagation would be necessary. In the next section we provide the details of our design.

9.2.1 Representationally closed carry-free division of stored posibit transfer operands

In this section, we provide the details of our design based on approach 3 above. Assume that the dividend Z and the divisor D are normalized fractional $2m$-digit stored posibit transfer operands of the division operation, where each stored posibit transfer digit has h binary positions ($h = 4$ and $-2m \leq i \leq -1$, in the example below). Z and D being normalized, lead to $|z_{-1}| > 0$ and $|d_{-1}| > 0$.

$Z = .z_{-1}\, z_{-2} \ldots z_{-2m+1}\, z_{-2m}$ $\qquad$ $D = .d_{-1}\, d_{-2} \ldots d_{-m}\, d_{-m-1} \ldots d_{-2m+1}\, d_{-2m}$

$$z_i = \begin{matrix} Z'_3 & z'_2 & z'_1 & z'_0 \\ & & & z''_0 \end{matrix} \qquad d_i = \begin{matrix} D'_3 & d'_2 & d'_1 & d'_0 \\ & & & d''_0 \end{matrix}$$

To see that D_l^2 is negligible in the expression $D_h^2 - D_l^2$, we compare the range of D_h^2 and D_l^2. Each digit d_i belongs to $[-r/2, r/2]$, where $r = 2^h$, is the radix, and h is the number of positions in each digit. Then we have: $D_h^2 \geq r^{-1}$, due to D being normalized and

$$D_l^2 \leq r^2/4\,(r^{-m-1} + r^{-m-2} + \ldots + r^{-2m+1} + r^{-2m})^2 = r^2/4\ \ (r^{-m} - r^{-2m})^2/(r-1)^2.$$

It can be shown that the right hand side of the latter inequality is less than $r^{-2m}/2$, for $r > 2 + 2^{1/2} > 3$, where r, being practically a power of two, is at least 4. Therefore deletion of $D_l^2 < ulp/2$, from the expression $D_h^2 - D_l^2$ does not introduce any error in the computation [Flyn01], hence the suitability of the equation

$$Q = Z\,/D = Z(D_h - D_l)/D_h^2.$$

For $D_h - D_l$, we don't need to actually perform any subtraction. The latter expression serves as the multiplier in $Z(D_h - D_l)$. We feed Z, as the multiplicand, and D as the input to the Booth recoder of the multiplication logic. We negate the least significant $mh/2$ (h is normally even in practice) Booth signals to take care of subtraction in $D_h - D_l$.

Fig. 9.6 shows the required Booth recoder cell, where X resembles the negabits in the odd positions (after the conversion of Fig. 9.4, where the complementation is fused in the logic). Inputs y and z, resemble the posibits in even positions. The most significant negabit of each stored transfer digit should be complemented before feeding, as X, into the logic. Note that the conversion of Fig. 9.4 does not actually take place. The X and y signals of the i^{th} Booth recoding cell ($0 \leq i \leq h/2 - 1$, for each digit) are provided by the posibits in positions $2i + 1$ and $2i$ of the original stored posibit transfer digit, respectively. The z signal of the i^{th} cell comes from the posibit in position $2i - 1$. There are two exceptions however; for $i = 0$, the second component posibit in position 0 provides z, and for $i = h/2 - 1$, X is provided by the complement of the negabit in position $h - 1$. S_1 is the conventional sign signal of the Booth encoder and its complement S_0, is provided for inverted Booth recoding of D_l. The computation (actually a single multiplication) of the nominator $Z(D_h - D_l)$ and looking up $1\,/\,D_h^2$, in a pre-computed table, can be done in parallel. Then another multiplication derives the quotient. The size of the look-up table would be more than that for the nonredundant division of [Flyn01], because an m digit radix-2^h stored posibit transfer operand has a total of $(m + 1)h$ twits, while a similar nonredundant operand has mh bits.

9.3 Summary

In this chapter, we provided high level designs for representationally closed stored posibit transfer multiplication and division. Similar designs for other WBS-like encodings, such as SUT and SDB hybrid representations, are feasible. We showed that in spite of existence of negabits in arbitrary positions of the multiplicand, Booth recoding can be applied to reduce the number of first level partial products. Moreover the extra second component sparse twits in the multiplier do not increase the number of Booth multiples as compared to a nonredundant multiplier with the same number of positions. Furthermore it turns out, as a pleasant surprise, that for practically wide redundant operands, with practically sparse second components, the number of reduction levels, where Booth recoding is applied, is the same as that of nonredundant operands with the same width. Thus the overall delay of our redundant multiplier is less than that of a nonredundant one with the same operand width. The overall delay is composed of three components; first one is the delay of Booth recoding, which is the same for both redundant and nonredundant operands, second one is the delay for partial product reduction, which was also shown to be the same. However the delay for final product derivation, in our redundant multiplier is less than that of the nonredundant case. The carry propagation chain of the last stage of the redundant multiplier is limited to the width of one redundant digit, while an operand width carry propagation chain is necessary to derive the nonredundant product. As was shown by the heuristic used in deriving the final product in Fig. 9.3, representational closure of our redundant multiplier does not introduce any extra delay.

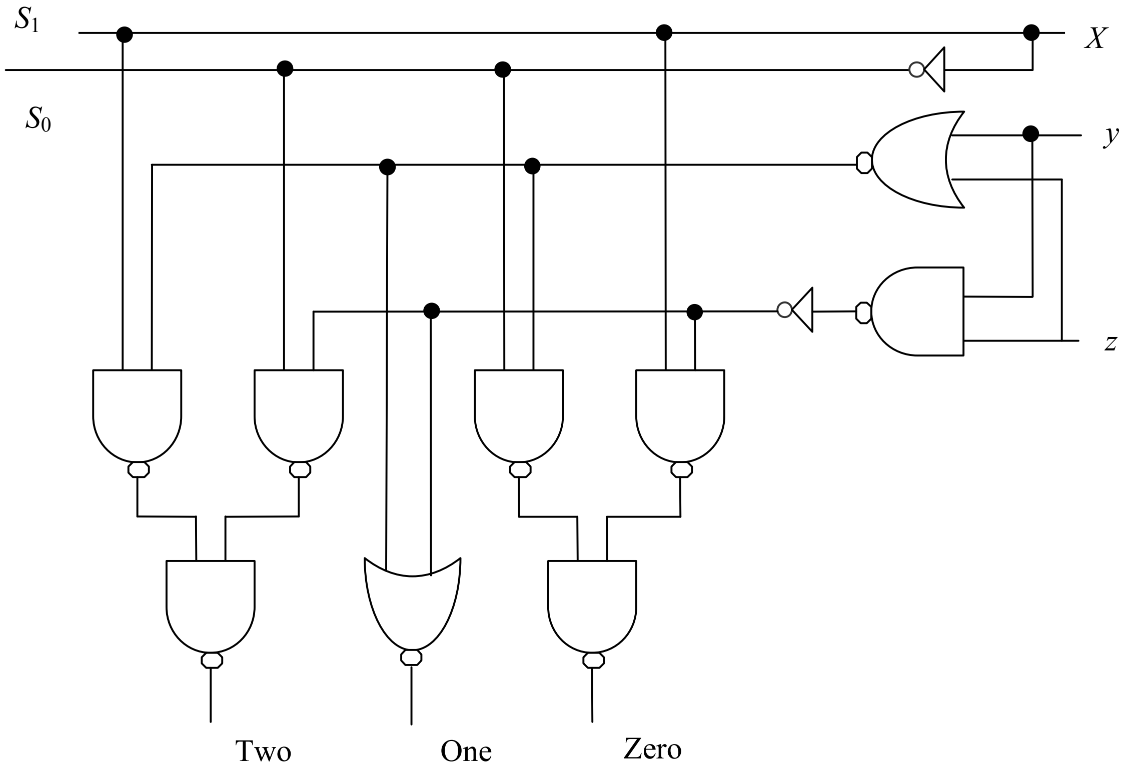


Fig. 9.6 Booth recoder for D_h (sign is S_1) and D_l (sign is S_0). X weights twice as y and z.

We observed that the subtractive division methods are not suitable for representationally closed redundant division. Thus we extended the high level multiplicative division design of [Flyn01], for stored posibit transfer operands. We showed that it is possible to keep the dividend and the divisor in redundant form, and use our redundant multiplier for the two multiplications embedded in the division algorithm, where the reciprocal of square of the most significant half of the divisor is looked up in a table implemented by a ROM or a PLA. Due to extra second component twits of the stored posibit transfer encoded divisor, our look up table imposes extra delay. But the limited carry propagation of the last stage of the multiplication is faster than the full carry propagation of the nonredundant version, and well pays off the extra look up delay, leading to overall less delay for our redundant division scheme.

Chapter 10 Arithmetic Support Functions

In the previous chapters we focused on the four basic arithmetic operations without explicitly addressing, in detail, the negation operation (or sign change) and the three standard detection operations, namely zero, sign, and over/underflow detection. Professor Parhami provides a comprehensive study of these operations on GSD [Parh90] number systems in [Parh93]. Negation can always be performed through subtracting from zero, but better performance is possible with specific tailor designed logics. We show that among the three selected encodings of Chapter 8, negation of SPT-encoded numbers is the simplest; just a twit-wise inversion of the number to be negated. Negation for the other two encodings, namely the SDB hybrid and SUT, require intradigit propagation as in all asymmetric GSD number systems [Parh93]. But it turns out that interdigit propagation is unavoidable for the other three detection operations. Whereas sequential linear latency algorithms are offered in [Parh93] for GSD numbers in general, we provide concrete solutions, with logarithmic latency, for detection operations in our three selected encodings. We present high level design of a tree structured logic shared by zero, sign, and over/underflow detection operations. Finally, we study the design of arithmetic shift operations (binary and radix) for the three selected encodings.

10.1 Negation of WTS-encoded numbers

It is naturally desirable to negate a twit by inverting its logical value.

Definition 10.1 (Negated twit): The negation of a twit with logical value x, characteristic $\{\lambda, \gamma\}$, and arithmetic value $\lambda + \gamma x$, is a twit with logical value $!x$ and arithmetic value $-\lambda - \gamma x$. ■

Lemma 10.1 (Negated twit): Given a twit $\{\lambda, \gamma\}$, with logical value x, the characteristic of its negation as per Definition 10.1 is $\{-\lambda - \gamma, \gamma\}$.

Proof: The following equation shows that the value $-\lambda - \gamma x$, of the negated twit can be represented by a twit $\{-\lambda - \gamma, \gamma\}$ with logical value $!x$.

$$-\lambda - \gamma x = -\lambda - \gamma + \gamma(1 - x) = -\lambda - \gamma + \gamma(!x). \blacksquare$$

The above Definition and Lemma lead to a minimal cost/delay negation operation of WTS-encoded numbers. We simply invert the logical value of each twit, and regard it as a twit whose lower value has been negated and then reduced by its gap value. Unfortunately, the encoding of the negation result is not, in general, the same as the original encoding, but there are special cases where, simple inversion preserves the representational closure property. Negation of three special case twits, which have been used in the three selected encodings of Chapter 8, is shown in Table 10.I, where a negated twit is characterized by $\{\lambda_n, \gamma_n\}$.

Table 10.I Negation of special case twits

Twit symbol	λ	γ	$\lambda_n = -\lambda - \gamma$	$\gamma_n = \gamma$	Negated twit symbol
●	0	1	-1	1	○
○	-1	1	0	1	●
▣	-1	2	-1	2	▣

Representational closure is, obviously, preserved for WTS-encodings, where the twits are symmetric (e.g., unibit), and/or there are equal number of complementary twits (e.g., posibits and negabits) in each position. Whenever the minimal cost/delay representationally closed negation is not possible, negation with no interdigit transfer propagation may be possible. It has indeed been shown that the latter is always true for all GSD number systems irrespective of the encoding used [Parh93]. Here we examine the negation operation on our three selected encodings.

10.1.1 Negation of SDB hybrid redundant numbers

Each SDB hybrid redundant digit, where double position encoding of SDB digit (see Table. 5.II) is used in a redundant position, may be viewed as a two's complement number. Therefore negation may be performed as a simple, digit parallel, two's complement operation. Two's complementing of each digit includes an increment operation possibly causing intradigit carry propagation. But due to asymmetry of two's complement representation, negation of 00…0 (representing $-r$ for say the i^{th} digit extending from position $(i-1)h$ to position ih) produces a posibit constant 1 in position ih, where it can't be held by the existing negabit in that position. To make room for this posibit 1, we use the following trick:

a) Do not complement the negabit in position ih when it has logical value 0 (i.e., when it represents arithmetic value -1), and add posibit constant 1 to that position. The net effect is the same as complementing the negabit.
b) Feed the added posibit 1 in position ih, as a carry-in to the increment logic of two's complementing digit $i+1$.

The preserved negabit 0 (representing -1) in position ih absorbs any possible coming posibit 1 through increment logic of position i, hence avoiding interdigit propagation. Then the whole negation operation, effectively, is composed of:

- Inverting all the posibits
- Setting all the negabits to 0
- Provide the carry-in of the two's complementing increment logic with the complements of the negabits, except for the LSD.

The negabit of the most significant digit should be treated differently, for there is no increment logic starting in that position. We do invert that negabit, and a carry 1 out of that position indicates an apparent overflow, which does not necessarily mean that the negation result cannot be represented [Parh93] (see also 10.3).

10.1.2 Negation of SUT-encoded numbers

An SUT digit is composed of a main two's complement part and a unibit as the second component. The unibit is negated by inversion as shown in Table 10.I. Negation of the main part, as a two's complement number is naturally done by twit-wise inversion, followed by an increment operation. The latter may generate a carry to the next more significant digit, where it should be absorbed for a digit parallel negation to be possible. To make room for this carry (or posibit transfer), instead of directly performing the increment operation, we do the following (for the i^{th} digit, $0 \leq i < k$, where k is the total number of digits):

a) Add a constant 1 to the inverted unibit $\underline{\mathbf{X''}_{ih}}$, leading to a double-bit with the same logical value, and replace it with a posibit x''_{ih+1} in position $ih + 1$
b) Add x''_{ih+1}, to the twit-wise inverted main part, generating a posibit transfer $c_{(i+1)h}$
c) Merge $c_{(i+1)h}$, with the first component posibit $w'_{(i+1)h}$, deriving a new posibit with logical value $!(w'_{(i+1)h} \oplus c_{(i+1)h})$ and a unibit in the same position with logical value $w'_{(i+1)h} \mid c_{(i+1)h}$

Step a) is explained by $1 + \underline{\mathbf{X''}_{ih}} = 1 + (-1 + 2x''_{ih}) = 2x''_{ih}$, where x''_{ih} is logical value of $\underline{\mathbf{X''}_{ih}}$. Table 10.II, depicts a truth table justifying Step c). Besides the initial inversion of all the twits, the negation operation uses one half adder per nonredundant positions, for the increment, and the logic required for Step c), is as simple as a half adder (one per each redundant position).

Table 10.II Truth table for the equations of Step c of SUT negation

$w'_{(i+1)h}$	$c_{(i+1)h}$	$w'_{(i+1)h} + c_{(i+1)h}$	•	▣
0	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	1

10.1.3 Negation of Stored Posibit Transfer (SPT) numbers

For symmetric digit sets, negation of each digit can be represented independent of other digits, for it always exists in the digit set. In other words, as noted in [Parh93], for a more general case, negation of a number with digits belonging to a symmetric digit set, does not generate transfers to more significant digits. The stored posibit transfer (SPT) encoding is symmetric, and as shown below, a simple twit-wise inversion negates a SPT number, while preserving representational closure.

Recall Table 10.I, where we showed that to negate the arithmetic value represented by a negabit (posibit), it is sufficient to invert its logical valuet, and regard it as a posibit (negabit). Assuming that the i^{th} digit of a SPT number is shown as $X'_{(i+1)h-1}\ x'_{(i+1)h-2} \ldots \mathrm{x}'_{ih+1}\ x'_{ih} + x''_{ih}$,, the negated digit can be represented as:

$$-\left(X'_{(i+1)h-1}\ x'_{(i+1)h-2} \ldots \mathrm{x}'_{ih+1}\ x'_{ih} + x''_{ih}\right) = y'_{(i+1)h}\ Y'_{(i+1)h-1} \ldots Y'_1\ Y'_0 + Y''_0.$$

Each y twit, in the latter equation, is the logical inversion of the corresponding x twit. Where is the representational closure then? The following theorem presents an interesting property of the SPT encoding, and answers the question.

Theorem 10.1 (Symmetry and complementarily): A symmetric contiguous WBS encoding is equivalent to its strongly complementary encoding.

Proof: Recalling Definition 6.5, to derive the strongly complementary encoding of a WBS encoding we replace each negabit (posibit) by a posibit (negabit). Assume that the original encoding represents an interval of integers $[-N, P]$. Replacing all posibits (negabits), by negabits (posibits), shifts the interval by $-P$ $(+N)$. This leads to $[-P, N]$, as the representable interval for the equivalent strongly complementary encoding. That in a symmetric WBS encoding we have $N = P$ concludes the proof. ■

Applying the result of Theorem 10.1 to the SPT encodings, we can regard the above y (Y) twit(s) as negabit (posibits), thus restoring the representational closure. This can be formally stated as follows:

Corollary 10.1 (Negation of SPT-encoded numbers): To negate an SPT-encoded number, it is sufficient to invert logical values of all the twits in the representation. ■

10.2 Zero and sign detection of periodic contiguous WBS-like encoded numbers

Recalling Corollary 6.7 and Theorem 6.10, we may consider a periodic contiguous WBS-like encoding with digit set $[-\alpha, \beta]$, as a representation of a GSD number system with the same digit set. Thus the relevant results in [Parh93] apply here as well. But we try to concretize them by studying the zero and sign detection problems for our three selected encodings, where $\alpha > 0$, $\beta > 0$, and practically the number of digits $k > 1$. With these conditions, Theorems 8 and 9 in [Parh93] reduce to the following theorem.

Theorem 10.2 (Zero and sign detection in the three selected encodings): Zero is represented solely by all zero radix-r digits in SUT, SPT, and alternatively interpreted SDB hybrid redundant encodings, where max $(\alpha, \beta) < r$, and sign of such numbers is the same as that of most significant nonzero digit. ■

In the SDB hybrid encoding, where max $(\alpha, \beta) = r$, Theorem 10.2 does not apply, but in the next section we present an alternative interpretation of SDB hybrid encoding, where max $(\alpha, \beta) < r$, leading to application of Theorem 10.2 as well.

10.2.1 Zero and sign detection of SDB hybrid redundant numbers

Two consecutive SDB hybrid digits may collectively represent zero in two ways; either each of them, independently, represent zero or the more significant one represents 1, and the other one represents $-r$ (given that $\alpha = r$). Therefore existence of a nonzero digit does not necessary imply that the whole number represents a nonzero value, and the zero detection techniques developed for nonredundant number representation systems do not apply for SDB hybrid. But a k digit radix 2^h periodic SDB hybrid encoding, where consecutive digits overlap in positions whose index is a nonzero multiple of h, may be viewed as a k digit radix 2^h periodic WBS encoding with an augmenting negabit in position kh and an enforced constant negabit, with logical value 1, in position 0 of the second component, where consecutive digits don't overlap.

This alternate interpretation of the same encoding leads to max $(\alpha, \beta) = 2^h - 1 < r$, which in turn implies, by Theorem 10.2, unique zero representation. Therefore once the zero and nonzero digits are distinguished, conventional algorithms for zero detection apply. Fortunately each digit in the WBS encoding described above has unique zero representation within a digit, where posibits are all 0, and the single second component negabit is 1, with arithmetic value 0. Sign of a single SDB hybrid redundant digit is easily detected by inspection of the logical value of the negabit. However, in the alternative interpretation of SDB hybrid encoding, sign of a single digit could depend on the logical values of all twits in a digit. In fact, only the value of an all zero digit is negative. Unique zero representation within, and sign detection of, a single digit could be equally important in cost/latency considerations for zero and sign detection logic. In the next two sections we study the same problem for SUT and SPT digits, and show that single digit zero detection is more difficult, while single digit sign detection is as easy as in a two's complement digit.

10.2.2 Zero and sign detection of SUT and SPT numbers

With the application of Theorem 10.2 for SUT and SPT numbers, standard zero detection techniques used for nonredundant numbers, with logarithmic latency at best, may be used here as well. Sign detection, however, is not as easy as the immediate sign detection for e.g., two's complement numbers. Algorithms 8 and 9 of [Parh93] may be directly used for our SUT, SPT, and SDB hybrid (alternative interpretation of Section 10.2.1) numbers for sign detection, and combined zero and sign detection respectively. These sequential algorithms show a linear latency. However, we provide, below, a high level design of some logarithmic latency logic for combined zero and sign detection. Consider two consecutive digits g_{i+1} and g_i, with z_{i+1} and z_i, indicating whether their value is zero or not, and s_{i+1} and s_i, showing their signs, respectively. A z (s) signal with logical value 0 indicates that the corresponding digit g is nonzero (nonnegative). Then the following equations compute $z\ (g_{i+1}g_i)$ and $s\ (g_{i+1}g_i)$, the z and s signals for the composite digit $g_{i+1}g_i$, respectively: $z\ (g_{i+1}g_i) = z_{i+1}z_i$, $s\ (g_{i+1}g_i) = !(z_{i+1})s_{i+1} |\ z_{i+1}s_i$

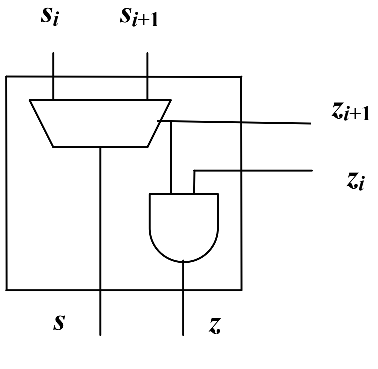


Fig. 10.1 A ZSD cell

Fig. 10.1 shows the Zero and Sign Detector (ZSD) cell for two consecutive digits. A binary tree structure of the ZSD cells, as in Fig. 10.2, derives the sign of a redundant number, and determines whether it is zero, both with logarithmic latency. One last point to be noted for SUT and SPT encodings is that neither of them have single digit unique zero representation (Fig. 10.3), hence more complex logic for their z signals. But the s signal of an SUT or an SPT digit is determined only by their most significant twit (i.e., the only negabit in the digit's encoding); a zero negabit causes the digit to be nonpositive.

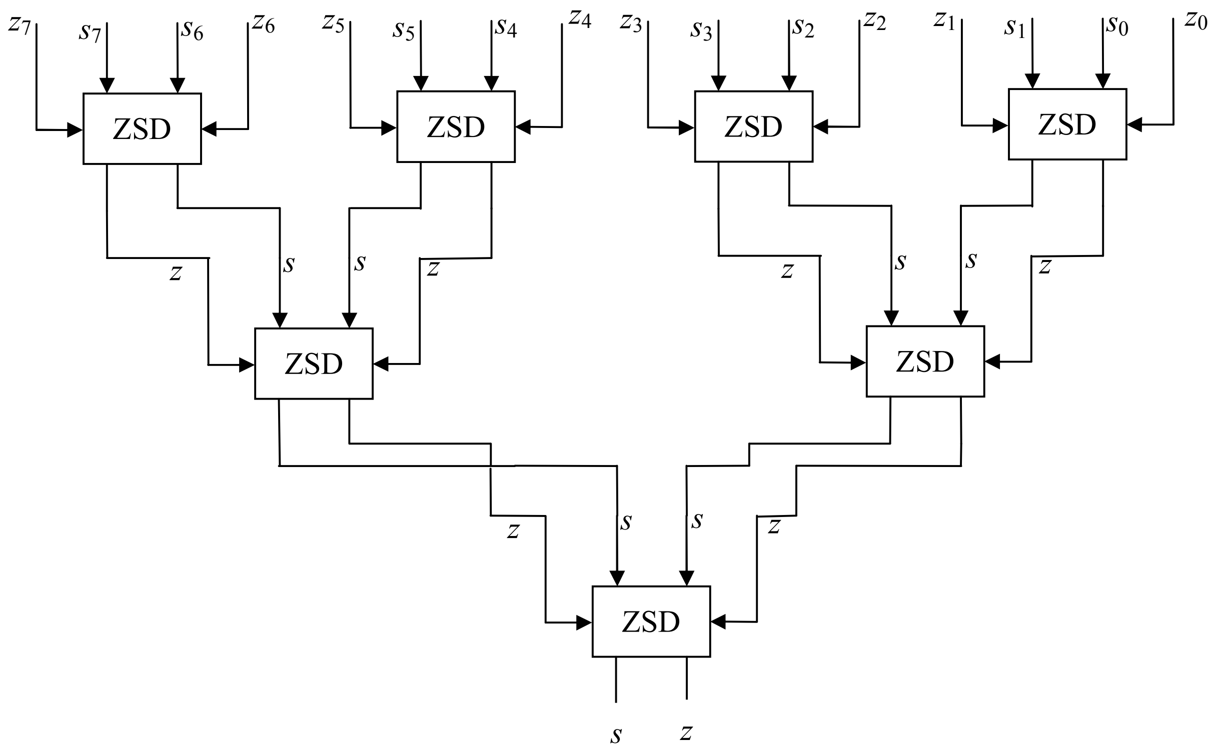


Fig. 10.2 Zero/sign detector for 8 digit redundant numbers

10.3 Over/Underflow Detection of WBS-like-encoded results

A general treatment of over/underflow detection of GSD numbers is offered in [Parh93], where after easy detection of an apparent over/underflow, a sequential algorithm, with linear latency, either detects real over/underflow, or corrects the apparent over/underflow. The linear latency performance may not be desirable for very long data paths (e.g., 128 twits) with small periods (e.g., h = 4). Following a similar approach as in Section 10.2, by focusing on the three selected encodings, we show that the hardware of Fig. 10.2 can be used to either detect real over/underflow, or with a slight modification correct the apparent over/underflow. Nevertheless, some real over/underflow conditions may be detected faster, by inspection of the twits in position kh.

SUT								SPT							
1	0	0	1	0	1	1	1	1	0	0	0	0	1	1	1
			0				1				0				1

Fig. 10.3 Alternative zero representations (h = 4)

In the following definitions, we assume a k position periodic WTS encoding with period h.

Definition 10.2 (Over/underflow twits): A collection of extra twits, besides the ones defined in the encoding, that are generated in position kh, and beyond, of the result of an arithmetic operation is called over/underflow twits. ■

Definition 10.3 (Apparent over/underflow): A positive (negative) collective value of over/underflow twits indicates an apparent overflow (underflow). ■

Definition 10.4 (Real over/underflow): When there is no possibility to make room, while preserving representational closure, for the value represented by the apparent overflow twits in the valid positions and twits of the result, a real over/underflow occurs.■

Definition 10.5 (Apparent over/underflow correction): When the collective value of apparent over/underflow twits plus the value represented by the valid twits of the encoding fall in the valid range of the underlying redundant representation, the apparent over/underflow is said to be correctable, otherwise there exist a real over/underflow. ■

In apparent over/underflow correction a valid result should be obtained by *back-propagating* the value represented by the apparent over/underflow twits. Note that when representational closure is not required, the over/underflow twits can be kept as they are in position *kh*, and beyond, of the result. For each of the three selected encodings, we study the peculiarities of the apparent over/underflow twits, and their back propagation, absorption (correction), or rejection (real overflow).

10.3.1 Apparent Over/Underflow Detection for WTS encodings

We provide, below, the details of apparent over/underflow detection for our three selected encodings. Then we study the problem of real over/underflow detection and apparent over/underflow correction for the selected encodings.

- **SPT**: Recalling the representationally closed SPT addition of Fig. 6.23, there are three apparent over/underflow twits, one negabit and two posibits, in position *kh* ($k = h = 4$). To make the task of over/underflow handling easier, we use another full adder to reduce the over/underflow twits to a_{kh+1}, a posibit in position *kh*+1, and A_{kh}, a negabit in position *kh* (Fig. 10.4).

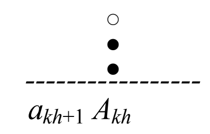


Fig. 10.4 Reduction of SPT apparent overflow twits

- **SUT**: Recall Fig. 6.18, which depicts a representationally closed *k* digit radix 2^h SUT addition, for $k = h = 4$. The two unibits in position *kh*, are the two apparent over/underflow twits $\underline{A}'_{kh}$ and $\underline{A}''_{kh}$.

- **SDB hybrid**: Fig. 10.5 depicts a representationally closed addition of two augmented SDB hybrid redundant operands, where the apparent over/underflow twits are shown to be a collection of one posibit, one negabit, and two unibits. Note that we have used the result expressed in Corollary 6.3 to reduce, in place, the twits in position *kh*.

It would make the real over/underflow detection easier, if we had fewer number of apparent over/underflow twits. Fig. 10.6 shows an alternative design for position kh, leading to one apparent over/underflow twit $\underline{\boldsymbol{A}}_{kh+1}$, in position kh+1, a posibit a_{kh} (last in the carry chain of MSD), and a negabit A_{kh} (directly from T_{kh}), both in position kh.

$\underline{\boldsymbol{A}}'_{16}$	a'_{15}	a'_{14}	a'_{13}	a'_{12}	a'_{11}	a'_{10}	a'_{9}	a'_{8}	a'_{7}	a'_{6}	a'_{5}	a'_{4}	a'_{3}	a'_{2}	a'_{1}	a'_{0}
A''_{16}				A''_{12}				A''_{8}				A''_{4}				
$\underline{\boldsymbol{B}}'_{16}$	b'_{15}	b'_{14}	b'_{13}	b'_{12}	b'_{11}	b'_{10}	b'_{9}	b'_{8}	b'_{7}	b'_{6}	b'_{5}	b'_{4}	b'_{3}	b'_{2}	b'_{1}	b'_{0}
B''_{16}				B''_{12}				B''_{8}				B''_{4}				
A''_{16}	a'_{15}	a'_{14}	a'_{13}	a'_{12}	a'_{11}	a'_{10}	a'_{9}	a'_{8}	A'_{7}	a'_{6}	a'_{5}	a'_{4}	a'_{3}	a'_{2}	a'_{1}	a'_{0}
B''_{16}	b'_{15}	b'_{14}	b'_{13}	b'_{12}	b'_{11}	b'_{10}	b'_{9}	b'_{8}	b'_{7}	b'_{6}	b'_{5}	b'_{4}	b'_{3}	b'_{2}	b'_{1}	b'_{0}
$\underline{\boldsymbol{A}}'_{16}$	t_{15}	t_{14}	t_{13}	t_{12}	t_{11}	t_{10}	t_{9}	t_{8}	t_{7}	t_{6}	t_{5}	t_{4}				
$\underline{\boldsymbol{B}}'_{16}$				T_{12}				T_{8}								
T_{16}																
A''_{16}	w_{15}	w_{14}	w_{13}	w_{12}	w_{11}	w_{10}	w_{9}	w_{8}	w_{7}	w_{6}	w_{5}	w_{4}	w_{3}	w_{2}	w_{1}	w_{0}
c_{16}	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_{9}	c_{8}	c_{7}	c_{6}	c_{5}	c_{4}	c_{3}	c_{2}	c_{1}	
B''_{16}				T_{12}				T_{8}				1				
$\underline{\boldsymbol{A}}'_{16}$																
$\underline{\boldsymbol{B}}'_{16}$																
T_{16}																
$\underline{\boldsymbol{S}}'_{16}$	s'_{15}	s'_{14}	s'_{13}	s'_{12}	s'_{11}	s'_{10}	s'_{9}	s'_{8}	s'_{7}	s'_{6}	s'_{5}	s'_{4}	s'_{3}	s'_{2}	s'_{1}	s'_{0}
S'_{16}				S''_{12}				S''_{8}				S''_{4}				
▣																
▣																
○																
●																

Fig. 10.5 Apparent over/underflow twits of representationally closed SDB hybrid addition

10.3.2 Real over/underflow detection

In all the three selected encodings, digits belong to redundant digit sets. Redundancy of the digit set provides for possibility of apparent over/underflow correction. One possible correction strategy is back propagation of the apparent over/underflow value.

Definition 10.6 (Back propagation): An existing apparent over/underflow value ω in the (i+1)th digit g_{i+1} is back propagated through the i^{th} digit g_i of the result of an arithmetic operation, by correcting g_{i+1} to $g_{i+1} - \omega$, and g_i to $g_i + 2^h\omega$. ■

Definition 10.7 (Sink): During the back propagation, when an over/underflow value ω visits the i^{th} digit g_i, if the corrected digit value stays in the range of the digit set, we say that the over/underflow value *sinks* in g_i. We use signal s^+_i (s^-_i) to indicate that g_i absorb the visiting overflow (underflow) value. ■

Definition 10.8 (Zip): The corrected digit value (i.e., $g_i + 2^h\omega$) due to back propagation, may be slightly over (under) the maximum (minimum) value represented by the digit set, in which case the excess value may *zip* over to the next right digit. We use signal z^+_i (z^-_i) to indicate the latter situation for digit g_i. ■

Definition 10.9 (Reject): If the excess value after correction of a digit due to back propagation is too large to be absorbed by the right context, both the sink and zip signals will be set to 0, and we say that the visited digit rejects the visiting value. ■

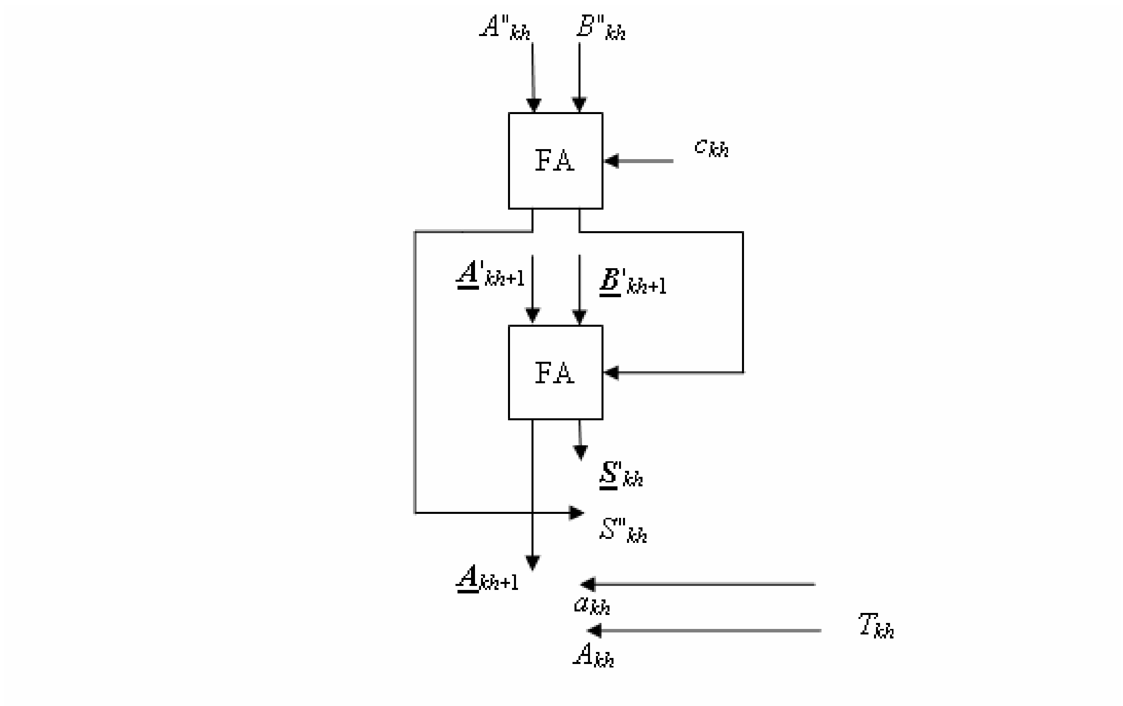


Fig. 9.6 Alternative design for position *kh* of augmented SDB hybrid addition

Table 10.III shows the sink/zip characteristic of a digit.

Table 10.III sink/zip characteristic of a redundant digit

s	*z*	**Action**
0	0	Reject
0	1	Zip over
1	0	Sink
1	1	X

When an apparent over/underflow value during back propagation is rejected, or zips over the LSD, there is a real over/underflow. In the following subsections we examine the sink/zip properties of the digit sets of our three selected encodings, and derive the equations for *s* and *z* signals. But first we show that once these signals are derived for all digits of a result, they may be fed into the circuit of Fig. 10.2 to derive the overall *s* and *z* signals in the root of the ZSD tree.

We define $z\ (g_{i+1}g_i)$ and $s\ (g_{i+1}g_i)$, as the composite *z* and *s* signals for two consecutive digits g_{i+1} and g_i. The following equations obviously hold. But Note that they are, given that *s* and *z* may not both be 1, logically equivalent to those in Section 10.2.2 above. Therefore the ZSD cell of Fig. 10.1 and the tree structure of Fig. 10.2 can be reused for over/underflow detection, as well as sign and zero detection. A real overflow (underflow) is detected if there is an apparent overflow (underflow), and the *s* signal, in the root of the sign, zero, and over/underflow detection tree, is low.

$$z\ (g_{i+1}g_i) = z_{i+1}z_i \qquad\qquad s\ (g_{i+1}g_i) = s_{i+1}\,|\,z_{i+1}s_i$$

10.3.2.1 Derivation of *s* and *z* signals for SDB hybrid redundant digits

An SDB hybrid redundant digit belongs to $[-2^h, 2^h-1]$, where the period is h and $r = 2^h$ is the radix. A collective value 2 for the apparent overflow can be corrected if, for example, the value of the two most significant digits are -2^h, leading to corrected digit values 2^h-1 and 0, from left to right. Over/underflow twits with absolute collective values more than 1, complicate overflow handling. An alternative approach is to use the alternative interpretation of SDB hybrid encoding as we did in Section 10.2.1 above, where the digit set is $[-1, 2^h-1]$. A digit with value -1 (0), can absorb (zip over to the next right digit) a 1 back propagating from its next left digit, turning itself to 2^h-1. But it cannot absorb or pass a back propagating 2. For, receiving a constant 2 from left, is equivalent to adding 2^{h+1} to the receiving digit, and turning it to 2^h-1+2^h. The extra 2^h cannot sink in the right context. With similar argument we can see that a digit of the alternative interpretation of the SDB hybrid encoding cannot absorb or pass the back propagating twit (s) with collective value less than -1. Therefore when the collective value of the apparent over/underflow twits (i.e., $\underline{\boldsymbol{A}}_{kh+1}$, A_{kh}, and a_{kh} of Fig. 10.6) is neither 1, nor -1, given that it cannot be zero, we may have an *immediate* real over/underflow. Immediate overflow (underflow) occurs, exactly, when the apparent overflow (underflow) value is 3 or 2 (-3 or -2), and the augmenting digit is nonnegative or 1 (negative or -2), respectively. Otherwise the real over/undrflow detection is left to the ZSD tree. The equation for immediate real overflow v (underflow u) is as follows, and $!v$ ($!u$) denote *reduced* apparent overflow (underflow):

$$v = \underline{\boldsymbol{S}}'_{kh}\, \underline{\boldsymbol{A}}_{kh+1}\, (A_{kh}\ \ a_{kh} \mid (A_{kh} \oplus a_{kh})\, S''_{kh}),$$

$$u = \underline{!\boldsymbol{S}}'_{kh}\, !\underline{\boldsymbol{A}}_{kh+1}\, (!A_{kh}\ \ !a_{kh} \mid (A_{kh} \oplus a_{kh})\, !S''_{kh}).$$

We recognize the overflow (underflow) handling capability of the i^{th} digit by two signals s_i^+ (s_i^-) and z_i^+ (z_i^-). s_i^+ (s_i^-) is high when its corresponding digit's value is -1 (2^h-1), absorbing a back propagating 1 (-1), and is low otherwise. z_i^+ (z_i^-) is similarly defined to represent the passing capability of the corresponding digit on a back propagating 1 (-1), when the digit's value is 0 (2^h-2). Describing the i^{th} digit of the result by $s'_{(i+1)h-1}\ s'_{(i+1)h-2} \ldots s'_{ih+1}\, s'_{ih} + S''_{ih}$, we can derive the equations for $s_i^{\pm}$ and $z_i^{\pm}$, as shown below for digits indexed 0 to $k-1$.

$$-1\!: 00\ldots00 + 0 \longrightarrow s_i^+ = !(s'_{(i+1)h-1} \mid s'_{(i+1)h-2} \mid \ldots \mid s'_{ih+1} \mid s'_{ih})!S''_{ih}$$
$$0\!: 00\ldots00 + 1 \text{ or } 00\ldots01 + 0 \longrightarrow z_i^+ = !(s'_{(i+1)h-1} \mid s'_{(i+1)h-2} \mid \ldots \mid s'_{ih+1})\, (s'_{ih} \oplus S''_{ih})$$

$$2^h-1\!: 11\ldots11 + 1 \longrightarrow s_i^- = s'_{(i+1)h-1}\, s'_{(i+1)h-2} \ldots s'_{ih+1}\, s'_{ih}\, S''_{ih}$$
$$2^h-2\!: 11\ldots1 + 0 \text{ or } 11\ldots10 + 1 \longrightarrow z_i^- = s'_{(i+1)h-1}\, s'_{(i+1)h-2} \ldots s'_{ih+1}\, (s'_{ih} \oplus S''_{ih})$$

10.3.2.2 Derivation of *s* and *z* signals for SUT digits

To simplify over/underflow handling, as we did for SDB-hybrid redundant encoding we use alternative interpretation (i.e., with unshifted unibit transfers) of SUT digits, where a digit value belongs to $[-2^{h-1}-1, 2^{h-1}]$. A back propagating value with absolute value of more than 1, visiting an alternatively interpreted SUT digit, is not welcome, and will be rejected. The collective value of the apparent over/underflow twits for SUT encoding (i.e., $\underline{\boldsymbol{A}}'_{kh}$ and $\underline{\boldsymbol{A}}''_{kh}$ of Section 10.3.1) belongs to $\{-2, 0, 2\}$. Assume that the twits of the SUT encoding of the result in position kh, are denoted by s'_{kh} and S''_{kh}.

The collective value of valid SUT twits in position *kh* (i.e., that of a posibit and a negabit) belongs to [–1, 1]. Therefore an apparent overflow (underflow) value 2 (–2), is rejected, i.e., turns to immediate real overflow (underflow) by position *kh*, iff the latter represents 1 (–1). These observations lead to the following equations for immediate real overflow (underflow) *v* (*u*).

$$v = \underline{\mathbf{A}}'_{kh}\ \underline{\mathbf{A}}''_{kh}\, s'_{kh}\, S''_{kh}, \qquad\qquad u = !\underline{\mathbf{A}}'_{kh}\ !\underline{\mathbf{A}}''_{kh}\, !s'_{kh}\, !S''_{kh}.$$

A visiting value 1 (–1) sinks in an alternatively interpreted SUT digit, only when the digit value is $-2^{h-1}-1$ (2^{h-1}) and -2^{h-1} ($2^{h-1}-1$), and it zips over, only when the digit value is $-2^{h-1}+1$ ($2^{h-1}-2$). Assuming that an alternatively interpreted SUT digit is denoted as $S'_{(i+1)h-1}\, s'_{(i+1)h-2} \ldots s'_{ih+1}\, s'_{ih} + \underline{\mathbf{S}}''_{ih}$, the latter observation leads to the following equations for $s_i^{\pm}$ and $z_i^{\pm}$, as shown below for digits indexed 0 to *k*–1.

$$-2^{h-1}-1\text{: } 00\ldots00 + 0 \text{ and } -2^{h-1}\text{: } 00\ldots01 + 0 \rightarrow s_i^{+} = !(S'_{(i+1)h-1} | s'_{(i+1)h-2} | \ldots | s'_{ih+1} | \underline{\mathbf{S''_{ih}}})$$
$$-2^{h-1}+1\text{: } 00\ldots00 + 1 \text{ or } 00\ldots10 + 0 \rightarrow z_i^{+} = !(S'_{(i+1)h-1} | s'_{(i+1)h-2} | \ldots s'_{ih+2} | s'_{ih})(s'_{ih+1} \oplus \underline{\mathbf{S''_{ih}}})$$

$$2^{h-1}\text{: } 11\ldots11 + 1 \text{ and } 2^{h-1}-1\text{: } 11\ldots10 + 1 \rightarrow s_i^{-} = S'_{(i+1)h-1}\, s'_{(i+1)h-2} \ldots s'_{ih+1}\, \underline{\mathbf{S''_{ih}}}$$
$$2^{h-1}-2\text{: } 11\ldots01 + 1 \text{ or } 11\ldots11 + 0 \rightarrow z_i^{-} = S'_{(i+1)h-1}\, s'_{(i+1)h-2} \ldots s'_{ih+2}\, s'_{ih}\, (s'_{ih+1} \oplus \underline{\mathbf{S''_{ih}}})$$

10.3.2.3 Derivation of *s* and *z* signals for SPT digits

An SPT digit rejects any visiting value whose absolute is more than 1. The apparent over/underflow twits of an SPT result was denoted, in Section 10.3.1 above, as a_{kh+1} and A_{kh}. The collective apparent over/underflow value belongs to [–1, 2]. Therefore there is no immediate underflow. But immediate overflow occurs when the apparent overflow value is 2. This leads to the following equations:

$$v = a_{kh+1} A_{kh}, \qquad\qquad u = 0.$$

The range of an SPT digit is $[-2^{h-1}, 2^{h-1}]$. A visiting value 1 (– 1) sinks only when the digit value is -2^{h-1} (2^{h-1}), and zips over only when the digit value is $-2^{h-1}+1$ ($2^{h-1}-1$). These observations lead to the following equations:

$$-2^{h-1}\text{: } 00\ldots00 + 0 \rightarrow s_i^{+} = !(S'_{(i+1)h-1} | s'_{(i+1)h-2} | \ldots | s'_{ih+1} | s'_{ih} | s''_{ih})$$

$$-2^{h-1}+1\text{: } 00\ldots01 + 0 \text{ or } 00\ldots00 + 1 \rightarrow z_i^{+} = !(S'_{(i+1)h-1} | s'_{(i+1)h-2} | \ldots | s'_{ih+1})(s'_{ih} \oplus s''_{ih})$$

$$2^{h-1}\text{: } 11\ldots11 + 1 \rightarrow s_i^{-} = S'_{(i+1)h-1}\, s'_{(i+1)h-2} \ldots s'_{ih+1}\, s'_{ih}\, s''_{ih}$$

$$2^{h-1}-1\text{: } 11\ldots11 + 0 \text{ or } 11\ldots10 + 1 \rightarrow z_i^{-} = S'_{(i+1)h-1}\, s'_{(i+1)h-2} \ldots s'_{ih+1}\, (s'_{ih} \oplus s''_{ih})$$

10.3.3 Apparent over/underflow correction

An apparent overflow (underflow) can be corrected when there is at least one digit with high s_i^{+} (s_i^{-}), such that all its preceding digits, if any, have either high s_i^{+} (s_i^{-}) or high z_i^{+} (z_i^{-}). This leads to a high s^{+} (s^{-}) in the root of the ZSD tree. When there is a digit *g* with neither of s_i^{+} (s_i^{-}) nor z_i^{+} (z_i^{-}) being high, before the leftmost digit with high s_i^{+} (s_i^{-}), a back propagating 1 (–1) caused by an apparent overflow (underflow), will not sink in, nor zip over *g*. The latter situation indicates a real overflow (underflow).

To correct the result in case of a high $s^{\pm}$ at the root of the tree, we should find the first digit with high $s^{\pm}$ from the left. To search for this digit, we can follow the high $s^{\pm}$ children, giving priority to left child, until we reach at the desired high $s^{\pm}$, which should be a leaf. The required logic for this kind of search is offered in [Parh99]. Having found the rightmost digit to be corrected, we set all the digit values, from the MSD to the found digit, to an appropriate correcting value, which is to be determined, below, for each of the three selected encodings.

10.3.3.1 Correcting value for SDB hybrid redundant encoding

The alternatively interpreted digit set of the SDB hybrid redundant encoding is $[-1, 2^h-1]$. A visiting 1 (–1) sinks only when the digit value is –1 (2^h-1), correcting the digit value to 2^h-1 (–1), and zips over, only when the digit value is 0 (2^h-2), correcting the digit value to 2^h-1 (–1). Therefore when the correctable digits are recognized by the logic described above, they should all be set to 2^h-1, encoded as 11…11 + 1 (–1, encoded as 00…00 + 0), for apparent overflow (underflow) correction.

10.3.3.2 Correcting values for SUT encoding

The alternatively interpreted digit set of the SUT encoding is $[-2^{h-1}-1, 2^{h-1}]$. A visiting 1 (–1) sinks only when the digit value is $-2^{h-1}-1$ or -2^{h-1} (2^{h-1} or $2^{h-1}-1$), and zips over only when the digit value is $-2^{h-1}+1$ ($2^{h-1}-2$). The correcting value for z^{+}(z^{-}) digits is 2^{h-1} ($-2^{h-1}-1$), encoded as 11…11 + 1 (00…00 + 0). But there are two kinds of sink digits, with different values leading to different correcting values. The correction can be performed by adding the bias 2^h (-2^h) to s^{+}(s^{-}) digits, but the following solution is less costly. For s^{+}(s^{-}) digits, either 00…00 + 0 (11…11 + 1) is corrected to 11…10 + 1 (00…01 + 0), or 00…01 + 0 (11…10 + 1) changes to 11…11 + 1 (00…00 + 0). Therefore it is easily seen that, for apparent overflow (underflow) correction all we have to do is to set all the twits of correctable digits to 1(0), except for the least significant posibit of the sink digit, which stays unchanged. Recall that only the rightmost digit to be corrected is a sink one, and the rest of the correctable digits are zip ones.

10.3.3.3 Correcting value for SPT encoding

The SPT digit set is $[-2^{h-1}, 2^{h-1}]$. A visiting 1 (–1) sinks only when the digit value is -2^{h-1} (2^{h-1}), correcting the digit value to 2^{h-1} (-2^{h-1}), and zips over only when the digit value is $-2^{h-1}+1$ ($2^{h-1}-1$), correcting the digit value to 2^{h-1} (-2^{h-1}). Therefore when the correctable digits are recognized by the logic described above, they should all be set to 2^{h-1} (-2^{h-1}) encoded as 11…11 + 1 (00…00 + 0), for apparent overflow (underflow) correction.

10.3.4 Trusting the apparent over/underflow

For large periods (e.g. $h = 8$) and practical number of digits ($k \leq 8$), the ZSD tree is not very deep (e.g., 3 levels for $k = 8$), leading to moderate extra delay for completion of arithmetic operations by real over/underflow detection. Nevertheless with a sacrifice in the range of representable numbers, the extra delay can be reduced to a minimum amount independent of k. Considering k-digit, alternatively interpreted , SDB hybrid redundant and SUT numbers, regardless of their augmenting twits (see 10.3.2.1 and 10.3.2.2, respectively), and the SPT encoded numbers, observe that the only apparent overflow (underflow) value visiting the kth digit is 1 (–1). Assume that the digit set is $[-\alpha, \beta]$, and the radix is r.

Then in case of apparent overflow (underflow) the minimum (maximum) representable number not leading to a real overflow (underflow) is $\mu = 1 -\alpha\ldots -\alpha = r^k - \alpha\ \alpha \ldots \alpha$ ($\lambda = -1\ \beta\ldots\ \beta = -r + \beta\ \beta\ldots\ \beta$), or

$$\mu = r^k - \alpha\upsilon, \qquad\qquad \lambda = -r^k + \beta\upsilon,$$

where $\upsilon = \Sigma_{i=0}^{k-1}\ r^i$ is the unit digit value (see Definition 7.2). If we trust the apparent over/underflow detection, and always regard it as real over/underflow, we may loose some valid results in the range $[-\alpha\upsilon, \beta\upsilon]$. The positive (negative) loss is $\Lambda^+ = \beta\upsilon - \mu + 1$ ($\Lambda^- = \lambda + \alpha\upsilon + 1$). Replacing for μ (λ) in the latter expression, we derive:

$$\Lambda^+ = \Lambda^- = (\alpha + \beta)\upsilon - r^k + 1.$$

Recalling redundancy index $\rho = \alpha + \beta - r + 1$, defined for GSD number systems [Parh90], and applying the latter to above range loss equation, while replacing $\upsilon = \Sigma_{i=0}^{k-1}\ r^i$, by $(r^k - 1)/(r - 1)$, leads to:

$$\Lambda^+ = \Lambda^- = \rho\upsilon + (r - 1)\upsilon\ - r^k + 1 \quad\longrightarrow\quad \Lambda^+ = \Lambda^- = \rho\upsilon.$$

The cardinality of the representable range $[-\alpha\upsilon, \beta\upsilon]$ is $(\alpha + \beta)\upsilon + 1 = \rho\upsilon + (r - 1)\upsilon$. Therefore the percentage of the total range loss $\Lambda = 2\rho\upsilon$ to the representable range $\rho\upsilon + (r - 1)\upsilon$ is derived, after some manipulations as:

$$\%\Lambda_{\mathrm{SPT}} = 200\rho/(\rho + r - 1)$$

The percentage of the range loss for SPT encoding with $\rho = 1$, is $200 / r$. For the SUT encoding, we should consider the contribution of the augmenting twits in the value of μ and λ. μ is actually increased by r^k, and λ is decreased by r^k. But the maximum and minimum representable numbers change likewise leading to no change in total loss Λ. The loss percentage for the SUT encoding is thus derived as:

$200\rho\upsilon/((\rho + r - 1)\upsilon + 2r^k) = 200\rho/\ (\rho + r - 1 + 2r^k(r - 1)/(r^k - 1)) = 200\rho/\ (\rho + (r - 1)\ (3r^k - 1)/(r^k - 1))$, leading to $\%\Lambda\mathrm{sut} \approx 200\rho/\ (\rho + 3\ (r - 1))$.

For SDB hybrid redundant encoding, given that the collective value of augmenting twits belongs to $[-2, 1]$, the loss percentage can likely be derived leading to $\%\ \Lambda_{\mathrm{SDB}} \approx 200\rho/(\rho + 4(r - 1))$.

For practical values of period h and r (e.g., $h = 8$ and $r = 256$) , the range loss percentage is less than 0.8% for the SPT, and slightly more than 0.5% for the SUT, and less than 0.4% for the SDB hybrid redundant encodings.

10.4 Arithmetic shift operations on the three selected encodings

We recognize two kinds of shift operations; binary shift is the same as standard shift operation, but radix shift is defined for practical cases where the radix is a power of two ($r = 2^h$). In a representationally closed radix-2^h shift, each twit in position i is moved $\pm h$ positions. The latter is easy to implement by standard shift registers, and is most suitable for floating point arithmetic as discussed in Chapter 8. But a representationally closed binary shift is not as easy, and requires intradigit propagation. Standard shift registers can be used for a binary shift, where the result should be modified for preserving the representational closure property.

10.5 Summary

In this Chapter we studied the usual arithmetic support operations for WTS encodings focusing on the three selected encodings of Chapter 8 as candidates for general purpose representationally closed redundant arithmetic. We showed that twit-wise inversion of an SPT encoded number negates its value, but negation of the other two encodings requires intradigit propagation. We provided high level logarithmic latency logic to be used for sign, zero, and over/underflow detection operations, while presenting details of feeding the shared logic in case of each of the encodings. To keep the latency overhead of over/underflow detection to a minimum constant delay, we showed that trusting the apparent over/underflow signals, which are virtually available immediately after an arithmetic operation, sacrifices less than 0.8%, little over 0.5%, and less than 0.4% of the representable range of SPT, SUT, and the SDB hybrid redundant encodings, respectively. Finally we noted that the radix shift needed for redundant floating point operations can be performed in constant time, while other shifts (e.g., binary) require intradigit propagation.

Chapter 11 Conclusions

This research aimed at developing representationally closed redundant number encodings with efficient high level design of redundant arithmetic operations, which are suitable for a general purpose carry free arithmetic processor. A number of properties were enumerated for the desired number representations and the arithmetic algorithms to manipulate them, namely:

- Symmetry, or minimal asymmetry
- Digit set preservation
- Maximal encoding efficiency
- Representational closure
- Periodicity

The last two properties are essential for regularity and reusability in VLSI implementation.

In Chapter 2 we examined conventional signed digit number systems, and offered two's complement encoding of signed digits with a novel modification in the carry-free addition algorithm, where a position sum is compared with $r/2$ (r being the radix) instead of α (representing the digit set $[-\alpha, \alpha]$). This modification was shown to lead to fairly efficient implementation of carry-free addition.

Searching for even more efficient representations and algorithms, we developed the class of stored transfer encodings in Chapter 3, where each digit is composed of a main two's complement number and a stored transfer value. The concept of storing the transfer instead of conventional fusing of the transfer with the next digit of the result, helped in reducing the addition latency. We proved some theorems on the properties of this new class, specially the necessity of at least three, and sufficiency of four transfer values for carry-free addition of stored transfer encoded numbers. The basics of a virtually two-valued stored transfer scheme with increased encoding efficiency were established.

Generalization, usually leads to discoveries of special useful cases. In Chapter 4, we presented a generalization of the stored transfer scheme called the weighted bit-set (WBS) encoding. We showed that this generalization is indeed a unified representation of previously explored redundant number systems including the generalized signed digit number systems, the hybrid redundancy scheme, and the stored transfer representation. We developed general arithmetic algorithms for WBS encoding based on using readily available and highly optimized, building blocks developed for conventional binary arithmetic. In particular by limiting the propagation to posibits, we showed advantages over the hybrid redundancy scheme, where coexistence of borrow and carry propagation slows arithmetic operations.

In Chapters 5 and 6 we explored an interesting class of WBS encoding as an extension to hybrid redundancy scheme, which allowed for designing symmetric hybrid redundant number systems with arbitrary spacing of redundant positions such as the stored posibit transfer (SPT) encoding. In our extended hybrid redundancy scheme, negabits are allowed in nonredundant positions. We introduced the concept of inverted encoding of negabits (contrary to conventional encoding of negabits), and proved interesting properties of this novel encoding. The inverted encoding of negabits, surprisingly allowed for greater efficiency, while only standard cells, such as full/half adders, are used in the implementation of arithmetic operations. Possibility of using standard carry acceleration techniques for hybrid redundancy with inverted encoding of negabits is another important advantage.

The WBS encoding with all its interesting properties failed to exactly represent the virtually two-valued stored transfer encoding introduced in Chapter 3. We were thus motivated to further generalize the WBS encoding by introducing the concept of two-valued-digit (twit) in Chapter 7, which provides, as a special case, the two-valued unibit needed for two-valued stored transfer scheme. We also developed the bias encoding of twits, of which the inverted encoding of negabits is a special case, and showed that standard full/half adders, conventional compressors, and counters may be used to manipulate twits. We introduced our, super general, weighted twit-set encoding covering all the previously explored redundant number systems and allowing for new systems not explored before including those with discrete digit sets such as the shifted stored unibit transfer (SUT) encoding. General arithmetic algorithms for WTS-encoded numbers were offered, and concrete high level designs for special cases from stored transfer, hybrid redundant, and extended hybrid redundant schemes were provided.

In Chapter 8 we reviewed the number systems and encodings described in Chapter 2-7, and provided a table summarizing all the properties of the encodings discussed as a tool to help in selection of a desired encoding meeting the needs of a particular task. Using this table, we selected three of the encodings as candidates for general purpose carry free arithmetic. The selected encodings are the symmetric SPT encoding, minimally asymmetric SUT, and SDB hybrid redundant representations. The two latter encodings were augmented with extra twits in their most significant positions leading to greater encoding efficiency. For better comparison of symmetric SPT encoding and the other two asymmetric encodings we derived the symmetric range of the asymmetric ones. Subtraction delay penalty was another measure considered in the comparison. Conversion of the selected encodings to (from) two's complement encoding, were considered in detail for the three selected encodings. Conversion from two's complement was shown to be almost immediate, but the reverse conversion requires obligatory carry propagation. We designed special carry look-ahead logics to reduce the latency of conversions. Peculiarities of floating point arithmetic on the three selected encodings were discussed. Choosing the radix of the exponent equal to that of the encodings prohibits the introduction of any time penalty for implementing the required shift operations for preliminary alignment and post-normalizations needed in floating point arithmetic.

To complete the set of arithmetic operations on the three candidate encodings, we provided, in Chapter 9, high level carry free representationally closed multiplication and division algorithms, where interdigit carry propagation was not allowed, and WTS encodings were used for intermediate results. We showed that standard optimization techniques used in the design of efficient nonredundant multipliers, including the Booth recoding of the multiplier, and the popular (4; 2) compression of partial products can be directly used in the design of redundant multipliers, all achieved through the magic of inverted encoding of negabits. We found subtractive division methods not suitable for our redundant encodings, while multiplicative division proves to be appropriate. We then adopted an instance of advanced division designs of Flynn et. al [Flyn01] for our selected encodings.

Arithmetic shifts and negation operations, zero, sign, and over/underflow detection are arithmetic support operations that should necessarily be considered for a full treatment of a general purpose carry free arithmetic. We covered these topics in Chapter 10, where we provided special logarithmic latency logic shared between zero, sign, and over/underflow detection units. We noted that trusting the apparent over/underflow signals, which are immediately available after arithmetic operations on redundant operands, prohibits the delay penalty for real over/underflow detection. The latter is achieved at the cost of minor sacrifice in representation range of the underlying encoding. The figures for radix-256 encodings (i.e., with every 8th position being redundant), were 0.8%, 0.5%, and 0.4% range losses for SPT, SUT, and SDB hybrid redundant encodings, respectively.

To summarize the main results, we have achieved in this research:

- ➢ A platform for general purpose carry-free arithmetic consisting of:
  - ✓ number system encoding (Chapters 3, 4,and 7)
  - ✓ Efficient high level design of basic arithmetic operations and arithmetic support operations (Chapters 2, 5, 6, 9, and 10)
  - ✓ Guidelines for code optimization techniques for efficient use of carry-free instructions
- ➢ Development of new concepts in computer arithmetic such as:
  - ✓ Representational closure
  - ✓ Inverted encoding of negabits
  - ✓ Two-valued digit (twit) arithmetic
  - ✓ Bias encoding of twits leading to enhanced regularity in VLSI design
- ➢ Introduction of novel redundant number representations, and their efficient implementation such as:
  - ✓ Stored transfer encoding of redundant number systems
  - ✓ Weighted bit-set encoding, which provides for:
    - Unification of GSD and hybrid redundant number systems
    - Extension of hybrid redundancy scheme to include symmetric number systems with arbitrary digit sets
  - ✓ Weighted twit-set encoding covering all possible redundant number systems including those with discrete digit sets

Some possible topics for continuing research are:

- Twit interpretation of digit-set conversion
- Necessary and sufficient conditions for constant time WTS conversion
- Representability of any digit set by twits
- Impacts of WTS arithmetic on computer architecture
- Impacts of WTS arithmetic on compiler and code optimization techniques

References

[Ande67] Anderson, F. S. et al., “The IBM System/360 Model 91: Floating-Point Execution Unit,” *IBM Journal Res. And Dev.*, Vol. 11, pp. 34-53, Jan. 1667.

[Aviz61] Avizienis, A., “Signed-Digit Number Representations for Fast Parallel Arithmetic,” *IRE Trans. Electronic Computers*, Vol. 10, pp 389-400, Sep. 1961.

[Baug73] Baugh, C.R., and B.A. Wooley, “A Two’s Complement Parallel Array Multiplication Algorithm,” *IEEE Trans. Computers*, Vol. 22, pp 1045-1047, December 1973.

[Bedr62] Bedrij, O.J., “Carry Select Adder,” *IRE Trans. Electronic Computers*, Vol. 11, pp. 340-346, 1962.

[Boot51] Booth A. D., “A Signed Binary Multiplication Technique,” *Quartely Journal of Mechanics and Applied Mathmatics*, Vol. 4, Pt. 2, pp 236-240, June 1951.

[Bren82] Brent, R. P., and H. T. Kung, “A Regula Layout for Parallel Adders,” *IEEE Trans. On Computers*, Vol. C-31, pp 260-264, 1982.

[Daum03] Daumas, M., and D. W. Matula, “Further reducing the redundancy of a notation over a minimally redundant digit set,” *Journal of VLSI signal processing*, Vol 33, pp. 7-18, 2003.

[Dora88] Doran, R. W., “Variants of an Improver Carry Look-Ahead Adder,” *IEEE Trans. On Computers*, Vol. 37, pp 1110-1113, 1988.

[Dupr91] Duprat, J., Y. Herreros, and S. Kla, “New Redundant Representations of Complex Numbers and Vectors,” *Proc. 10th IEEE Symp. Computer Arithmetic*, pp. 2-9, June 1991.

[Erce94] Ercegovac M. D., and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer, 1994.

[Erce97] Ercegovac, M.D., “Effective Coding for Fast Redundant Adders Using the Radix-2 Digit Set {0, 1, 2, 3},” *Proc. 31st Asilomar Conf. Signals Systems and Computers*, November 1997, pp 1163-1167.

[Fahm03] Fahmy, H., and M.J. Flynn, “The Case for a Redundant Format in Floating Point Arithmetic,” *Proc. 16th IEEE Symp. Computer Arithmetic*, pp. 95-102, 2003.

[Ferg99] Ferguson, M.I. and M.D. Ercegovac, “A Multiplier with Redundant Operands,” *Proc. 33rd Asilomar Conf. Signals Systems and Computers*, Oct. 1999, pp 1322-1326.

[Flyn01] Flynn, M.J. and S.F. Oberman, *Advanced Computer Arithmetic Design*, Wiley, 2001.

[Fowl89] Fowler D. L., and J. E. Smith, “An Accurate High Speed Implementation of Division by Reciprocal Approximation,” *Proc. Of 9th Symposium on Computer Arithmetic*, pp. 60-67, 1989.

[Garn59] Garner, H. L., “The Residue Number System,” *IRE Trans. Electronic Computers*, Vol. 8, pp. 140-147, June 1959.

[Gonz00] Gonzalez, A. F., and P. Mazumder, "Redundant arithmetic, algorithms and implementations," *INTEGRATION, the VLSI journal*, 30, pp 13-53, 2000.

[Hara87] Harata, Y., Y. Nakamura, H. Nagase, M. Takigava, and N. Takagi, "A High-Speed Multiplier using a Redundant Binary Adder Tree," *IEEE j. Solid State Circuits*, SC-22, pp. 28-33, 1987.

[Jabe99] Jaberipur, G., "A Generalization of Carry Save Adders for Higher Radix Multi-Operand Addition," Public (unclassified) summary of a 1985 confidential technical report, Ref. # 1976, Iran Telecommunication Research Center, Nov. 1999.

[Jabe01] Jaberipur, G., B. Parhami, and M. Ghodsi, "A Class of Stored-Transfer Representations for Redundant Number Systems," *Proc. 35th Asilomar Conf. Signals Systems and Computers*, Nov. 2001, pp 1304-1308.

[Jabe02] Jaberipur, G., B. Parhami, and M. Ghodsi, "Weighted Bit-Set Encodings for Redundant Digit Sets: Theory and Applications," *Proc. 36th Asilomar Conf. Signals Systems and Computers*, November 2002, pp 1629, 1633.

[Jabe03] Jaberipur, G. and Ghodsi, M., "High Radix Signed Digit Number Systems: Representation Paradigms," *Scientia Iranica*,Vol. 10, No. 4, Oct. 2003, pp 383-391

[Jabe05a] Jaberipur, G., B. Parhami, and Ghodsi, M., "An Efficient Universal Addition Scheme for All Hybrid-Redundant Representations with Weighted Bit-Set Encoding," to appear in *Journal of VLSI Signal Processing.*

[Jabe05b] Jaberipur, G., B. Parhami, and M. Ghodsi, "Weighted Two-Valued Digit-Set Encodings: Unifying Efficient Hardware Representation Schemes for Redundant Number Systems," to appear in *IEEE Transactions on Circuits and Systems* I.

[Kame80] Kameyama, M., T. Higuchi, "Design of a Radix-4 Signed-Digit Arithmetic Circuit for Digital Filtering," *Proc. Of International Symposium on Multiple-Valued Logic*, IEEE, New York, pp. 272-277, 1980.

[Kant91] Kantaburta, V., "Designing Optimum Carry-Skip Adders," *Proc. 10th Symp. On Computer Arithmetic*, IEEE, pp. 146-153, 1991.

[Kawa90] Kawahito, S., M. Kameyama, and T. Higuchi, "Multiple-Valued Radix-2 Signed-Digit Arithmetic Circuits for High Performance VLSI Systems," *IEEE J. Solid State Circuits*, Vol 25, pp. 125-131, 1990.

[Kawa91] Kawahito, S., K. Mizuno, M. Ishida, and T. Nakamura, "Multiple-Valued Current-Mode Arithmetic Circuits Based on Redundant Positive-Digit Number Representations," *Proc. Of International Symposium on Multiple-Valued Logic*, IEEE, New York, pp. 330-339, 1991.

[Koba85] Kobayashi, H., "A Mutioperand Two's Complement Addition Algorithm," *Proc. 7th IEEE Symp. Computer Arithmetic*, pp 16-19, June 1985.

[Kore02] Koren, I., *Computer Arithmetic Algorithms*, 2nd edition, A.K. Peters, 2002.

[Korn94] Kornerup, P., "Digit-Set Conversions: Generalizations and Applications," *IEEE Trans. Computers*, Vol. 43, No. 5, pp 622 629, May 1994.

[Korn99] Kornerup, P., “Necessary and Sufficeint Conditions for Parallel, Constant Time Conversion and Addition,” *Proc. 14th IEEE Symposium on Computer Arithmetic* (ARITH-14), pp 152-155, April 1999, IEEE Computer Society.

[Kuni93] Kuninobu, S., T. Nishiyama, and T. Taniguchi,“High Speed Mos Multiplier and Divider using Redundant Binary Representation and Their Implementation in a Microprocessor,” *IEICE Trans. Electron.*, E76-C, pp. 436-444, 1993.

[Lehm61] Lehman, T., and N. Burla, “Skip Techniques for High-Speed Carry Propagation in Binary Arithmetic Units,” *IRE Trans. Electronic Computers*, Vol. 10, pp. 691-698, December 1961.

[Lync92] Lynch, T., and E. E. Swartzlander,Jr., “A Spannig Tree Carry Look-Ahead Adder,” *IEEE Trans. Computers*, Vol. 41, pp 931- 939, August 1992.

[Maki96] Makino, H., Y. Nakase, H. Suzuki, H. Morinaka, H. Shinohara, and K. Mashiko, “An 8.8-ns 54x54-bit Multiplier with High-Speed Redundant Binary Architecture,” *IEEE J. Solid State Circuits*, Vol. 31, pp 773- 783, 1996.

[Mand96] Mandellbaum, D. M., “A fast efficient parallel-acting method of generating functions defined by power series, including logarithm, exponential and sine, cosine,” *IEEE Transactions on Parallel and distributed systems,* Vol 7, No. 1, pp 33-45, Jan. 1996.

[Matu82] Matula, D. W., “Basic Digit Seys for Radix Representation,” *Journal of ACM*, Vol. 29, No. 4, October, 1982, pp 1131, 1143.

[Matu97] Matula, D. W., and A.M. Nielsen, “Pipelined Packet-Forwarding Floating Point: I. Foundations and a Rounder,” *Proc. 13th IEEE Symposium on Computer Arithmetic* (ARITH-13), pp 140-147, July 1997, IEEE Computer Society.

[Metz59] Metze, G. and J.E. Robertson, “Elimination of Carry Propagation in Digital Computers,” *Proc. Int’l Conf. Information Processing*, Paris, pp. 389- 396, 1959.

[Mign00] Mignotte, A., J.M. Muller and O.Peyran, “Synthesis for mixed arithmetic,” *Design Automation for Embedded Systems*, Vol. 5 No 1, pages 29-60, Feb. 2000.

[Moto92] Motorola Inc. “DSP56xxx Digital Signal Processor: Family Manual,” 1992.

[Ngai86] Ngai, T. F., M. J. Irwin, and S. Rawat, “Regular, Area-Time Efficient Carry Look-Ahead Adders,”*J. of parallel and distributed computing*, Vol. 3, pp 92-105, 1986.

[Niel97] Nielsen, A. M., and D. W. Matula, “Pipelined Packet-Forwarding Floating Point: II. An Adder,” *Proc. 13th IEEE Symposium on Computer Arithmetic* (ARITH-13), pp. 148-155, July 1997, IEEE Computer Society.

[Parh87] Parhami, B., “On the Complexity of Table Look-Up for Iterative Division” *IEEE Trans. Computers*, Vol. 36, No. 10, pp 1233-1236, 1987.

[Parh90] Parhami, B., “Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations,” *IEEE Trans. Computers*, Vol. 39, No. 1, pp. 89-98, Jan. 1990.

[Parh99] Parhami, B., *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum Press, New York, 1999

[Parh00] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford, 2000.

[Parh02] Parhami, B., "Number Representation and Computer Arithmetic," in *Encyclopedia of Information Systems*, Academic Press, Vol 3, pp 317-333, 2002.

[Peza71] Pezaris, S. D., "A 40-ns 17 bit by 17-bit array multiplier," *IEEE Trans. Computers*, pp. 442-447, April 1971.

[Phat94] Phatak, D.S. and I. Koren, "Hybrid Signed-Digit Number Systems: A Unified Framework for Redundant Number Representations with Bounded Carry Propagation Chains", *IEEE Trans. Computers*, Vol. 43, pp 880-891, Aug. 1994.

[Phat99] Phatak, D.S., T. Goff, and I. Koren, "Redundancy Management in Arithmetic Processing via Redundant Binary Representations", *Proc. 33rd Asilomar Conf. Signals Systems and Computers*, Oct. 1999, pp 1475-1479.

[Phat01] Phatak, D.S. and I. Koren, "Constant-Time Addition and Simultaneous Format Conversion Based on Redundant Binary Representations," *IEEE Trans. Computers*, Vol. 50, No. 11, pp 1267-1278, Nov. 2001.

[Robe58] Robertson, J. E., "A New Class of Digital Division Methods," *IRE Trans. Electronic Computers*, Vol. 7, pp. 218-222, September 1958.

[Schw91] Schwarz, E. M., and M. J. Flynn, "Cost-efficient high radix division," *Journal of VLSI Signal Processing*, pp 293-305, Oct. 1991.

[Schw92] Schwarz, E. M., and M. J. Flynn, "Approximating the sine function with combinational logic," in 26th Asilomar Conference on Signals, Systems, and Computers, 1992.

[Skla60] Sklansky, J., "Conditional-Sum Addition Logic," *IRE Trans. Electronic Computers*, Vol. 9, pp 226-231, June 1960.

[Taka85] Takagi, N., H. Yasuura, and S. Yajima, "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," *IEEE Trans. Computers*, Vol. 34, No. 9, pp. 789-796, Sep. 1985.

[Vuil83] Vuillemin, J. "A very Fast Multiplication Algorithm for VLSI Implementation," *Integration, VLSI Journal*, pp 39-52, 1983

[Wall64] Wallace, C.S., "A Suggestion for a Fast Multiplier," *IEEE Trans. Electronic Computers,* Vol 13, pp 14-17, 1964.