Journal of VLSI Signal Processing 40, 143–152, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Reviewing 4-to-2 Adders for Multi-Operand Addition*

PETER KORNERUP
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Received August 15, 2003; Revised January 16, 2004; Accepted March 4, 2004

Abstract. Recently there has been quite a number of papers discussing the use of redundant 4-to-2 adders for the accumulation of partial products in multipliers, claiming one type to be superior to other types. This paper analyses a recent proposal of various 3- and 4-element redundant digit sets for radix 2, signed and unsigned, and compare their implementations using various encodings of the digits and carries. It is shown that theoretically they are equivalent, and differences in their implementations need only be very marginal. Another recent proposal for the use of the digit-set {0, 1, 2, 3}, with a special 3-bit encoding of digits, is analyzed and some optimizations are shown, including the possibility of using a 2-bit encoding, with a quite significant saving in the wiring of a multiplier tree. All these proposed designs are shown to be equivalent to a standard 4-to-2, carry-save adder, except possibly for a few signal inversions.

Keywords: redundant adders, digit sets, digit encodings, multiplier trees

1. Introduction

When implementing fast multipliers in VLSI, a major part of area and time is spent on accumulating partial products using some kind of tree structures. Originally these were based on the use of full-adders (and occasionally some half-adders), reducing the sum of three rows of bits to the sum of two, using either the Wallace- [1] or Dadda-organizations [2] of the tree structure. As these structures are not very regular to lay out due to the 3-to-2 structure, it was suggested to use 4-to-2 adders which allow the use of binary tree structures. Weinberger [3] seems to be the first to propose their use, based on the carry-save representation, where two addends are considered an encoding of a single operand represented using the redundant digit-set {0, 1, 2}. Thus the 4-to-2 adder can be considered an adder taking two such operands and adding them to produce the result in the same representation. This addition can be performed in digit parallel, and thus in constant time, using an array of such digit adders. Radix 2 signed-digits for 4-to-2 adders over the digit-set $\{-1, 0, 1\}$ were then later proposed in [4, 5]. Recently [6] proposed using the digit-set {0, 1, 2, 3} with a special 3-bit encoding of the digits, and most recently [7] compared various 3- and 4-element digit sets using some particular encodings, claiming significant differences in the implementation of these. Other publications discussing multiplier organizations are [8–15], and plenty more.

After an introduction in Section 2 on the standard carry-save 4-to-2 adder, and some results on the use of signals with negative weights, Section 3 goes through a detailed review of a series of 4-to-2 adders recently presented in [7]. These are all based on digit codings of the form $d = \pm 2d^h \pm d^l$ for redundant digit-sets with 3 or 4 elements, and using a special technique denoted *equal-weight grouping*, or EWG. Here columns of bits of the same weight are accumulated, but producing a sum represented in the various digit-sets. Results from simulations on the designs were presented, whereupon the authors then state some claims on the relative performances when using the investigated digit-sets.

The following Section 4 then shows that all these designs can be implemented using the very same basic 4-to-2 carry-save adder, just using alternative external

*This work was supported by the Danish Natural Science Research Council, grant no. 21-00-0679.

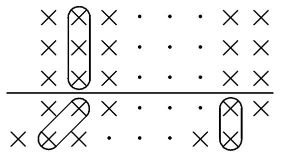
connections and possibly a few inverters. Hence there are no principal differences between the use of the various digit sets, as opposed to the claims of the paper, nor to the use of the digit encodings proposed or the 4-element digit-sets.

Section 5 reviews some other recent results from [6], where the use of the digit-set {0, 1, 2, 3} and a special 3-bit encoding of that digit-set is employed. The authors claim to achieve fairly significant speed-ups by this encoding. By a detailed analysis and identification of some basic building blocks, it is shown that it is not necessary to use the particular 3-bit encoding to obtain the same effect, there is an equivalent 2-bit encoding. It is found that a saving in the wiring of a multiplier tree can be obtained by reorganizing the building blocks. Finally it is shown that this design is also equivalent to one based on the standard 4-to-2 carry-save adder. In Section 6 conclusions are drawn.

2. A Basic Building Block

We will start by looking at the basic functionality of a 4-to-2 adder, whose purpose in a multiplier (and other multi-operand addition problems) is reducing the sum of four binary operands to the sum of two. The operands can be partial products as generated and delivered in parallel at the leaves of the tree, or at the internal nodes of the tree they can be the result delivered as pairs of operands, forming the result of other 4-to-2 adders. In general we shall assume that such operands and results, at a specific position in the tree, can have a specific positive or negative weight (is to be added or subtracted).

But let us start with the functionality of a full-adder in this context. It takes three bits of equal positive weight and delivers the sum of these as a two-bit number, the carry shifted one position over. An array of such full-adders (3-to-2 adders/compressors) then reduces the sum of three binary numbers into the sum of two:



where the last row represents the carries not being assimilated. The result is represented in the form of the last two rows of bits, hence the name *carry-save* representation. Pairing a column of two bits as indicated at the right in the figure, these can be interpreted as a digit in the set {0, 1, 2}, the *carry-save digit-set*, employing the *carry-save encoding*:

$$\begin{aligned} 0 &\sim 00 \\ 1 &\sim 01 \quad \text{or} \quad 10 \\ 2 &\sim 11 \end{aligned} \tag{1}$$

where it may be noted that the digit value 1 has two encodings.

Usually the 4-to-2 adder is considered an adder taking two such carry-save operands, and delivering the sum of these in the same carry-save representation. But alternatively interpreted at the digit level, it may also be considered a digit-set converter from the digit-set {0, 1, 2} + {0, 1, 2} = {0, 1, 2, 3, 4} into the set {0, 1, 2}, where such a conversion can be described by a two-level conversion diagram [16]:

$$\begin{tikzcd}
& {\{0,1,2,3,4\}} \arrow[d] \arrow[dl] & \\
{\{0,1\}} & {\{0,1,2\}} \arrow[d] & {\{0,1\}} \arrow[dl] \\
& {\{0,1,2,3\}} \arrow[d] \arrow[dl] & \\
{\{0,1\}} & {\{0,1\}} \arrow[d] & {\{0,1\}} \arrow[dl] \\
& {\{0,1,2\}} &
\end{tikzcd}$$

Each level shows the emission of a carry (the left-most, slanted arrow) and absorption of an incoming carry (the right-most, slanted arrow), together with the involved digit sets.

The converter or 4-to-2 adder can be realized by a combination of two full adders as in Fig. 1, where the tuples (i_1, i_2) and (i_3, i_4) are carry-save encodings of the operands, and (o_1, o_2) encodes the result. Note that carries $(c'_{\text{in}}, c''_{\text{in}})$ similarly constitutes a carry-save encoding of the incoming "double" carry, and $(c'_{\text{out}}, c''_{\text{out}})$ of the outgoing carry, corresponding to the previous conversion diagram.

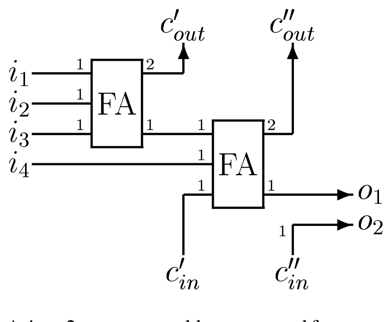


Figure 1. A 4-to-2 carry-save adder composed from two full adders.

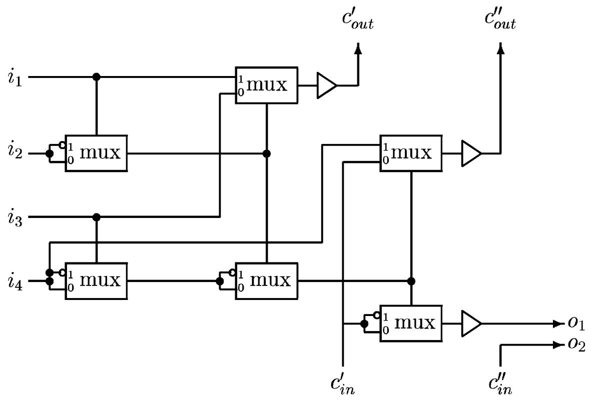


Figure 2. 4-to-2 carry-save adder built of multiplexers.

A particularly efficient realization of this 4-to-2 adder was shown in [11], as illustrated in Fig. 2, where the XOR-gates were implemented using pass-transistors.

Note that the addition of the carry c''_{in} is obtained at no cost in Figs. 1 and 2, corresponding to the second conversion (addition of the incoming carry) in the conversion diagram above being without cost in logic.

For the following discussion we will use the simplified Fig. 3, not being concerned with the actual implementation of the logic of the carry-save 4-to-2 adder.

In Figs. 1–3, and according to the carry-save representation, the signals $i_1, i_2, i_3, i_4, c'_{\text{in}}, c''_{\text{in}}, o_1$ and o_2 all have the weight 1, while c'_{out} and c''_{out} have weight 2, corresponding to the equation

$$\begin{aligned} &2(c'_{\text{out}} + c''_{\text{out}}) + (o_1 + o_2) \\ &\quad = i_1 + i_2 + i_3 + i_4 + c'_{\text{in}} + c''_{\text{in}}. \end{aligned} \tag{2}$$

We will in the following assume that $o_2 = c''_{\text{in}}$, since this allows c''_{in} to be added in at no cost. Thus we have the following defining equations

$$\begin{aligned} o_1 &= (i_1 + i_2 + i_3 + i_4 + c'_{\text{in}}) \bmod 2 \\ o_2 &= c''_{\text{in}} \\ c'_{\text{out}} + c''_{\text{out}} &= (i_1 + i_2 + i_3 + i_4 + c'_{\text{in}}) \text{ div } 2, \end{aligned} \tag{3}$$

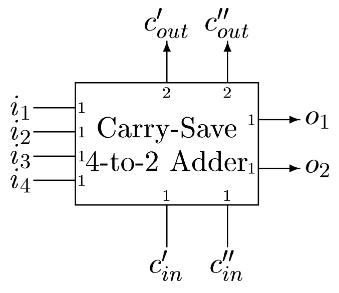


Figure 3. The 4-to-2 carry-save box.

where the pair $c'_{\text{out}}, c''_{\text{out}}$ provides a carry-save encoding of the combined carry in $\{0, 1, 2\}$.

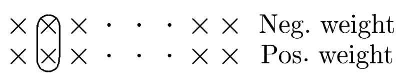
To allow changes in the weights for other digit sets, following [17] we note this lemma:

Lemma 1. *Let a signal $b \in \{0, 1\}$ have associated weight w, so that the value of the signal is $v = w\,b$. Inverting the signal into $1 - b$, while at the same time negating the sign of the weight, changes its value into $v' = v - w$, i.e., the value is being biased by the amount $-w$.*

Proof: Trivial since $v' = (-w)(1 - b) = wb - w = v - w$. □

When changing the interpretation of input signals as representation of values, it is then necessary to perform equivalent changes in the interpretation of the output signals. E.g., when the the total domain of input to an adder or a digit-set conversion is being biased, the output must be equivalently biased.

Now consider the base 2 digit-set $\{-1, 0, 1\}$ using the *signed-digit* encoding (also denoted *borrow-save*), where two bit strings are considered a string of digits:



obtained by pairing bits using the digit encoding:

$$\begin{aligned} -1 &\sim 10 \\ 0 &\sim 00 \quad \text{or} \quad 11 \\ 1 &\sim 01, \end{aligned} \tag{4}$$

where the left-most bit has negative weight. The conversion diagram for the addition of two signed-digit operands is then:

$$\begin{tikzcd}
 & \{-2,-1,0,1,2\} \arrow[dl, "\{-1,0\}"'] \arrow[d, "\{0,1,2\}"] & \arrow[dl, "\{-1,0\}"] \\
 & \{-1,0,1,2\} \arrow[dl, "\{0,1\}"'] \arrow[d, "\{-1,0\}"] & \arrow[dl, "\{0,1\}"] \\
 & \{-1,0,1\} &
\end{tikzcd}$$

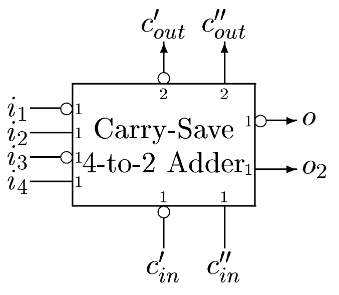


Figure 4. A 4-to-2 signed-digit adder obtained from a carry-save adder.

If in Fig. 3 we change the input so that (i_1, i_2) and (i_3, i_4) represent signed-digits, with i_1 and i_3 having negative weight, and the signals thus are delivered inverted, then the input is being biased by -2. The output must then according to Eq. (2) be equivalently biased, which is possible by changing the sign of c'_{out} by inverting its signal value. But then c'_{in} must also change sign, which changes the bias in the input to -3. Finally, to compensate we can change the sign of the weight of o_1, corresponding to the output (o_1, o_2) now representing a signed-digit, hence the signals must now satisfy the following equations:

$$\begin{aligned}&2((1 - c'_{\text{out}}) + c''_{\text{out}}) + ((1 - o_1) + o_2)\\&\quad = (1 - i_1) + i_2 + (1 - i_3) + i_4 + (1 - c'_{\text{in}}) + c''_{\text{in}}\end{aligned}$$

or

$$\begin{aligned}&2(-c'_{\text{out}} + c''_{\text{out}}) + (-o_i + o_2)\\&\quad = -i_1 + i_2 - i_3 + i_4 - c'_{\text{in}} + c''_{\text{in}}\end{aligned}$$

and the equivalent defining equations derived from (3). Inverting the signals appropriately in Fig. 3 we obtain Fig. 4.

Theorem 2. *In a computational model where inversion is without cost in area and time, radix* 2 *signed-digit addition can be realized at exactly the same cost as carry-save addition.*

The transistor implementation of the XOR-gates in Fig. 2 from [11] uses a “dual-rail” representation of signals, these being provided and used in true as well as in inverted form. This implies that inversion of a signal here can be realized by “twisting wires”, and thus at no significant cost in time and area. Also, in many implementations some signals are produced in complemented form anyway, thus some inversions may actually be avoided.

Observe that if an array of such adders are connected (here vertically) to form an n-digit adder, then no inversions are needed between the individual adders. Similarly if a tree of such arrays are formed to perform multi-operand addition, then all the inversions internally in the tree can be eliminated. We have thus shown the following:

Theorem 3. *Multi-operand addition of radix* 2 *signed-digit operands can be implemented by an array or tree of carry-save adders, by inverting all negatively weighted signals on input, as well as on output from the array/tree, but with no internal changes.*

3. Codings of the Form $\pm 2d^h \pm d^l$

Recently in [7] various radix 2 redundant digit-sets, with encodings of the form (d^h, d^l) with

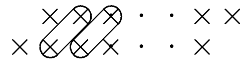
$$d = \pm 2d^h \pm d^l,$$

were suggested and analyzed, employing a particular way of implementation denoted *equal-weight grouping* or EWG, to be described below. The digit-sets investigated were

$$\begin{aligned}\mathcal{D}^{(SD)} &= \{-1, 0, 1\} & \mathcal{D}^{(CS2)} &= \{0, 1, 2\}\\ \mathcal{D}^{(SD3^{(-)})} &= \{-2, -1, 0, 1\} & \mathcal{D}^{(CS3)} &= \{0, 1, 2, 3\}\\ \mathcal{D}^{(SD3^{(+)})} &= \{-1, 0, 1, 2\}. & & \end{aligned} \qquad (5)$$

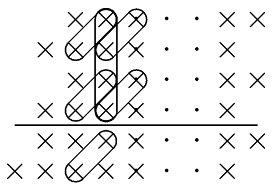
The digit-sets $\mathcal{D}^{(SD)}$ and $\mathcal{D}^{(SD3^{(-)})}$ are coded as $d = -2d^h + d^l$, corresponding to a 2’s complement encoding of the digit d. Since the digit-set $\mathcal{D}^{(SD)}$ does not include -2, the bit-pattern $(d^h, d^l) = (1, 0)$ is not valid in the SD representation. The set $\mathcal{D}^{(SD3^{(+)})}$ is realized by changing the sign of both components, i.e., $d = 2d^h - d^l$. The $\mathcal{D}^{(CS3)}$ and $\mathcal{D}^{(CS2)}$ digit-sets are coded with $d = 2d^h + d^l$, where the bit-pattern $(d^h, d^l) = (1, 1)$ is invalid in the $\mathcal{D}^{(CS2)}$ representations.

Since the encodings employ weights differing by a factor of 2, neighboring digits d_i and d_{i-1} in a radix representation overlap one another, i.e., d_i^l has the same weight as d_{i-1}^h,



but accumulation of several bit-vectors from the encoding of digit vectors can still be performed in vertical columns. This is what the authors of [7] denote *equal-weight grouping* or EWG-form.

Normally redundant adders are used to reduce the sum of two digit-vectors to a single redundant digit-vector, absorbing and emitting suitable carries. With additions of the form $\mathcal{D} + \mathcal{D} \rightarrow \mathcal{D}$, where $\mathcal{D}$ is one of the above listed CS digit sets, e.g., with $\mathcal{D} = \mathcal{D}^{(CS3)}$, then $\mathcal{D} + \mathcal{D} = \{0, 1, 2, 3, 4, 5, 6\}$. But with EWG, bits from the encoding of two neighboring positions are added, the summation taking bits in a column and forming a sum in $\{0, 1, 2, 3, 4\}$, but producing the resulting digit in $\mathcal{D}^{(CS3)} = \{0, 1, 2, 3\}$, as shown in the following diagram (without the carries).



The coding used for $\mathcal{D}^{(CS2)}$ and $\mathcal{D}^{(CS3)}$ is $d = 2d^h + d^l$. Since the bit-vectors all have positive weight, it is a digit-set conversion of the form:

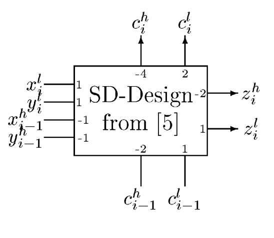
$$\begin{tikzcd}
 & \{0,1,2,3,4\} \arrow[d] & \\
 & \{0,1,2\} \arrow[d] \arrow[dl, "\{0,1\}"'] & {} \arrow[dl, "\{0,1\}"'] \\
{} & \{0,1,2,3\} &
\end{tikzcd}$$

for the case of addition in $\mathcal{D}^{(CS3)}$, where using EWG the conversion is from $\{0, 1, 2, 3, 4\}$ back into $\mathcal{D}^{(CS3)} = \{0, 1, 2, 3\}$. When the coding is $d = -2d^h + d^l$, two of the bit-vectors have negative weight, and by EWG the mapping is from $\{-2, -1, 0, 1, 2\}$ to $\{-1, 0, 1\}$.

For comparison the authors consider the following four cases:

1. $SD + SD \rightarrow SD$:[1] Here they employ the signed-digit, 4-to-2 adder from [5] using the encoding $-1 \sim 11$, $0 \sim 00$ and $1 \sim 01$, corresponding to 2's complement (but it could also be interpreted as a sign-magnitude) encoding. The carry-set is $\mathcal{C}^{(SD)} = \{-1, 0, 1\}$.

$$\begin{tikzcd}
 & \{-2,-1,0,1,2\} \arrow[d] & \\
 & \{0,1,2\} \arrow[d] \arrow[dl, "\{-1,0\}"'] & {} \arrow[dl, "\{-1,0\}"'] \\
{} & \{-1,0,1,2\} \arrow[d] & \\
 & \{-1,0\} \arrow[d] \arrow[dl, "\{0,1\}"'] & {} \arrow[dl, "\{0,1\}"'] \\
{} & \{-1,0,1\} &
\end{tikzcd}$$

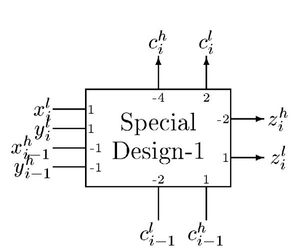


2. $SD3^{(-)} + SD3^{(-)} \rightarrow SD3^{(-)}$: The carry-set used is $\mathcal{C}^{(SD^{(-)})} = \{-1, 0, 1\}$, despite the fact that $\{0, 1\}$ is sufficient, as can be seen from this conversion diagram:

$$\begin{tikzcd}
 & \{-2,-1,0,1,2\} \arrow[d] & \\
 & \{-2,-1,0\} \arrow[d] \arrow[dl, "\{0,1\}"'] & {} \arrow[dl, "\{0,1\}"'] \\
{} & \{-2,-1,0,1\} &
\end{tikzcd}$$

The signals in their mux-based design have weights as shown in the diagram:

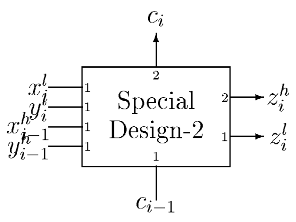
$$\begin{tikzcd}
 & \{-2,-1,0,1,2\} \arrow[d] & \\
 & \{0,1,2\} \arrow[d] \arrow[dl, "\{-1,0\}"'] & {} \arrow[dl, "\{-1,0\}"'] \\
{} & \{-1,0,1,2\} \arrow[d] & \\
 & \{-1,0\} \arrow[d] \arrow[dl, "\{0,1\}"'] & {} \arrow[dl, "\{0,1\}"'] \\
{} & \{-1,0,1\} &
\end{tikzcd}$$



It is noted that the addition $SD3^{(+)} + SD3^{(+)} \rightarrow SD3^{(+)} = \{-1, 0, 1, 2\}$ can be realized by interchanging the positive and negative inputs.

3. $CS2 + CS2 \rightarrow CS2$: The carry-set here is proven to be $\mathcal{C}^{(CS2)} = \{0, 1\}$, as the two outgoing carries in the diagram is shown never simultaneously to be 1. Another special mux-based design is used in [7]:

$$\begin{tikzcd}
 & \{0,1,2,3,4\} \arrow[d] & \\
 & \{0,1,2\} \arrow[d] \arrow[dl, "\{0,1\}"'] & {} \arrow[dl, "\{0,1\}"'] \\
{} & \{0,1,2,3\} \arrow[d] & \\
 & \{0,1\} \arrow[d] \arrow[dl, "\{0,1\}"'] & {} \arrow[dl, "\{0,1\}"'] \\
{} & \{0,1,2\} &
\end{tikzcd}$$



4. $CS3 + CS3 \rightarrow CS3$: The carry-set is again $\mathcal{C}^{(CS3)} = \{0, 1\}$, hence a simplified conversion is sufficient. To implement the adder they employ the 4-to-2 compressor (carry-save based) from [11] as here described in Fig. 2, but using the digit encoding $d = 2d^h + d^l$:

$$\begin{tikzcd}
 & \{0,1,2,3,4\} \arrow[d] & \\
 & \{0,1,2\} \arrow[d] \arrow[dl, "\{0,1\}"'] & {} \arrow[dl, "\{0,1\}"'] \\
{} & \{0,1,2,3\} &
\end{tikzcd}$$

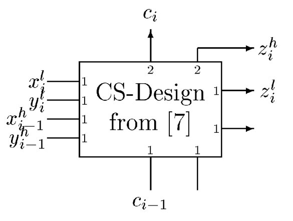


Table 1. TSMC SCN025 0.25 micron technology design timings from [7].

Adder cell	Critical path delay (ns)
SD	0.78750
$SD3^{(-)}$	0.96025
$CS2$	0.66100
$CS3$	0.46580

where the lower right-hand connectors are not used (they are anyway just connected by a "jumper".)

Note that for all four designs above, input is provided in the EWG-form by combining signals (bits of the same absolute value weight) from two neighboring digit positions, but output is delivered in the proper encoding for the particular digit-set.

The authors of [7] performed SPICE simulations of the four designs above, yielding Table 1 (their Table 8):

On page 1276 of [7] it is stated:

"*multipliers based on $CS3$ can be expected to outperform multipliers based on other redundant representations*"

and furthermore using arguments on digit-set cardinalities:

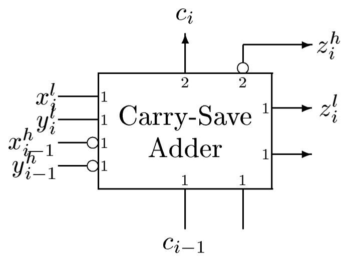
"*Therefore, cells such as 1 and 3 from [the listing above] are fundamentally more complex, hence, bigger and slower.*"

4. Alternative Implementations

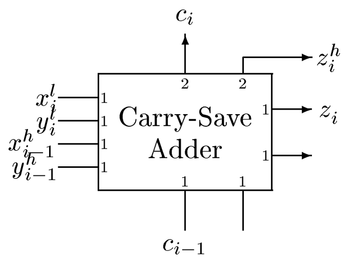
Using the results of Section 2, *and contrary to the above statements*, we shall now show that all the four adder cases presented above can be implemented by the same hardware (e.g., the efficient design from [11] as used in Fig. 2 and case $CS3$ above), just using alternative interconnections. It will then also be apparent that there are no advantages in using the particular codings of the form $\pm 2d^h \pm d^l$, as opposed to the form $d^h \pm d^l$, traditionally used for carry-save and signed-digit encodings.

We will do this by showing that in all of the above four cases there exist alternative implementations, using any generic 4-to-2 carry-save adder. In particular we may employ the mux-based design from [11], where it is possible at no cost in transistor count to add properly placed signal inversions. This was the design found in [7] to be the fastest, the one listed in Table 1 as $CS3$.

1. $SD + SD \rightarrow SD$: Can be realized by the circuit:



where the lower right-hand connectors are not used (they are anyway just connected by a "jumper".)
2. $SD3^{(-)} + SD3^{(-)} \rightarrow SD3^{(-)}$: Obviously the diagram above can also be used here, since it delivers a result in $\{-1, 0, 1\} \subset SD3^{(-)} = \{-2, -1, 0, 1\}$.
3. $CS2 + CS2 \rightarrow CS2$: The standard carry-save adder can directly be used, just using different output connections to deliver the result in the $2d^h + d^l$ encoding.



4. $CS3 + CS3 \rightarrow CS3$: The very same 4-to-2, carry-save adder as above can be used since $\{0, 1, 2\} \subset \mathcal{D}^{(SD3)} = \{0, 1, 2, 3\}$. Note that this was also the solution used in [7].

Observe again, that in a tree-structure of arrays of such adders (e.g., in a multiplier tree), there is no need for inversions internally in the tree. We can thus conclude that it is possible to implement addition in these five digit-sets with the same logic, except possibly for a few inversions, which may even be trivially realized at no cost in logic.

5. Another Example Using the Digit-Set $\{0, 1, 2, 3\}$

In [6] another implementation was proposed employing the digit-set $\{0, 1, 2, 3\}$, albeit with a 3-bit encoding of

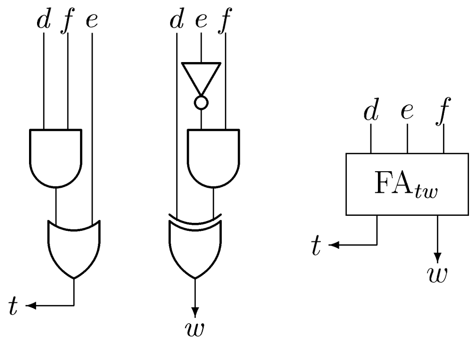


Figure 5. Mapping from (d, e, f) into (t, w).

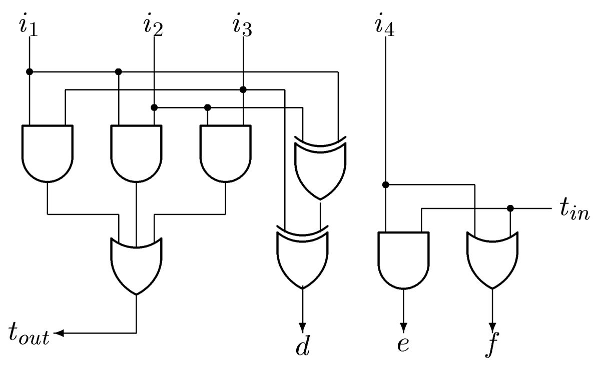


Figure 6. Adder taking four binary signals into (d, e, f) encoding.

digits, where the value of the code can be obtained by adding the values of the three bits (d, e, f) for all the valid combinations, $v = d + e + f$, where $(e, f) \neq (1, 0)$. Internally a unique 2-bit coding (t, w) was also used:

Value	0	1	–	2	1	2	–	3
d, e, f	000	001	010	011	100	101	110	111
t, w	00	01	–	10	01	10	–	11

Observe that (t, w) is the same encoding as used in [7] for the digit-sets $\mathcal{D}^{(CS2)} = \{0, 1, 2\}$ and $\mathcal{D}^{(CS3)} = \{0, 1, 2, 3\}$, with digit value $d = 2d^h + d^l$. Mapping from (d, e, f) into (t, w) is performed by the logic in Fig. 5 (slightly reorganized from [6], here using exclusively AND, OR and XOR gates), shown together with a simplified block symbol of it.

This logic is functionally equivalent to a full-adder, except for the restrictions on the input (d, e, f) (which we will denote the *def*-encoding), and coding of the output in the (t, w)-form (denoted the tw-encoding). Ercegovac and Lang [6] then describes various adders for combinations of operands in the redundant *def*-encoding and ordinary binary, with output in *def*-encoding. We shall only go into details with two of these. The first we describe in Fig. 6 is an adder taking four ordinary binary operands, and delivering their sum in the *def*-encoding, again slightly modified from [6].

Observe here that the left-most part of Fig. 6, computing (t_{out}, d), is a standard full-adder, and the right-most part is a recoding where arithmetically $e + f = i + t$, but assures that only legal combinations of d, e, f are produced ($(e, f) \neq (1, 0)$). This is the logic needed at the first level of a multiplier tree, the critical path here is through two XOR gates, and the authors claim a speedup compared to an ordinary 4-to-2 adder.

For the case of two redundant operands the critical path is two XORs, plus an AND and an OR gate, where the authors also found a speedup. The following conversion diagram shows how the addition is performed:

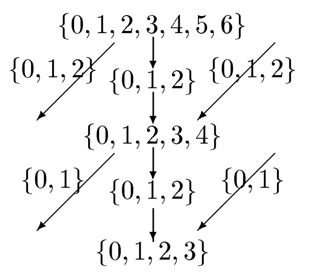


Figure 7, reproduced with the same kind of slight modifications from [6], shows their implementation, using a carry in $\{0, 1, 2\}$ in carry-save representation (t^1, t^2), plus another carry u in $\{0, 1\}$.

Observe that the dot-framed part is again a standard full-adder, and the logic producing e and f is again the recoding assuring $(e, f) \neq (1, 0)$. Thus Fig. 7 can also be described in a simplified way by the block-diagram in Fig. 8.

Now note that instead of using the *def*-coding at input and output, we may as well remove the two FA_{tw} adders at the top, and add one at the bottom, thus using the tw-coding on input and output, as illustrated in Fig. 9.

We have thus shown that it is not necessary to use the *def*-encoding at the external interface to the adder, and that the 4-to-2 adder in tw-encoding can be implemented with only about two thirds of the logic needed for the adder using def-encoding, as presented in [6].

We finally need the adder-type to use at the first level of an adder tree, i.e., taking four binary signals and

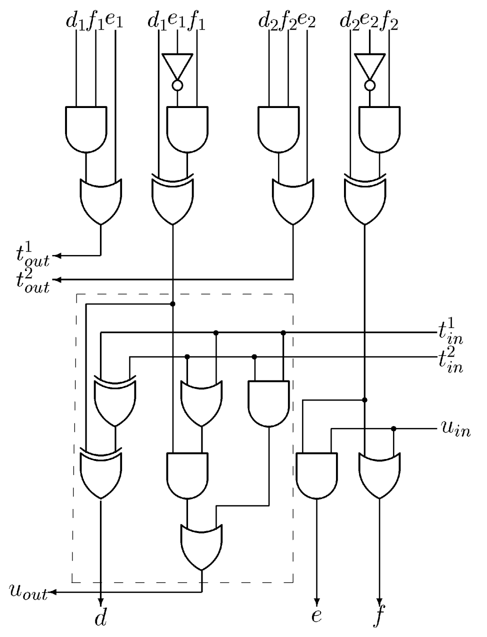


Figure 7. Adder with two redundant operands.

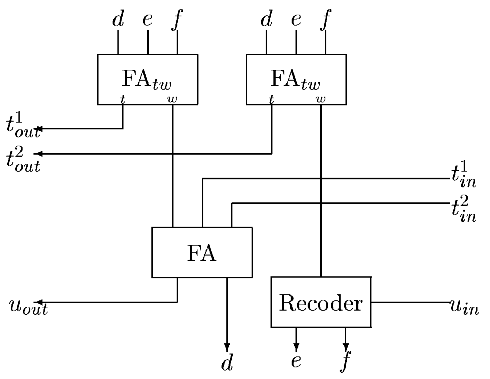


Figure 8. Block diagram of Fig. 7.

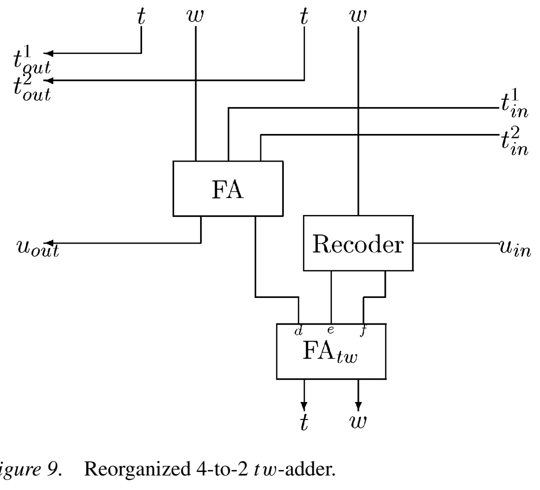


Figure 9. Reorganized 4-to-2 tw-adder.

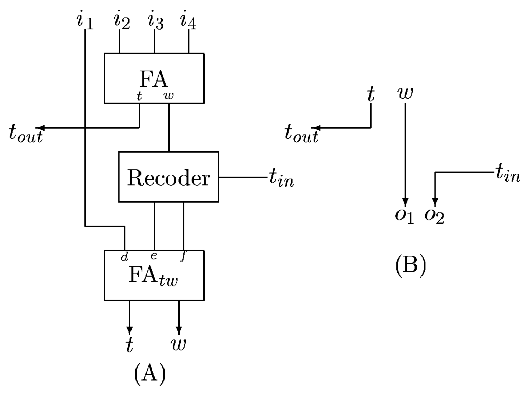


Figure 10. (A) First level adder for tw-encoded result. (B) Root conversion.

producing their sum in tw-encoding, and a recoding to use at the root of the adder tree. Both of these are trivially found as shown in Fig. 10.

However, the simplifications of the nodes do not help in an adder tree, since there still is a *def*- to tw-conversion between each level in the tree, and at the first level an extra similar conversion is needed now as seen in Fig. 10(A). Note though that at the root of the tree, only the simple re-wiring in Fig. 10(B) is needed to deliver the ordinary carry-save representation of the result. Thus the critical path is unchanged.

But much more important, the wiring of the tree is now significantly reduced, since only two signals instead of three are needed in each connection of the adder tree.

Now observe that in all places where signals e, f are fed into an FA$_{tw}$-adder, there is a recoding taking place, to assure that the pattern $(e, f) \neq (1, 0)$. If the logic of that recoding is combined with the logic of the FA$_{tw}$-adder, it appears that the result is an ordinary full-adder.

Just consider Fig. 9 with input (d, w, u) to the combination of a recoder and FA$_{tw}$-adder, with output denoted (t', w'), then from the combined logic in Fig. 11 we find

$$\begin{aligned} t' &= dw + du + wu \\ w' &= d \oplus w \oplus u, \end{aligned}$$

which is the functionality of a standard full-adder.

When used in an adder tree, we have thus shown that the proposed adder is equivalent to the standard carry-save, 4-to-2 adder based on the digit-set {0, 1, 2}, and the speed differences observed are due to different implementations of the full-adder-equivalent circuits used.

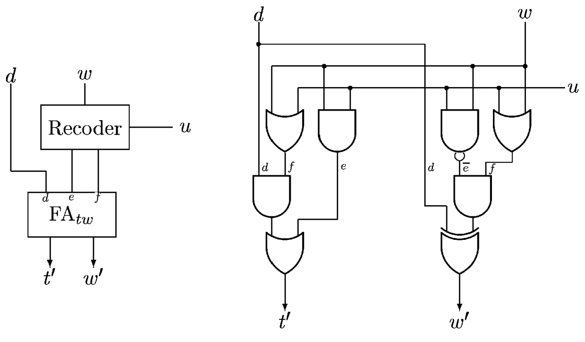


Figure 11. A "hidden" full adder.

6. Conclusions

Using some trivial results on signal weights it has been shown that there are no fundamental differences between the redundant, radix 2, 4-to-2 adders based on carry-save, and those based on signed-digit representations, i.e., digit sets $\{0, 1, 2\}$ and $\{-1, 0, 1\}$ with their usual encodings. There need only be very small differences between implementations, as only the addition or removal of a few inverters is needed to change one type of adder into the other. In a computational model where inversion is without cost in area and time, signed-digit arithmetic can be performed at exactly the same cost as carry-save.

For multi-operand addition, as in multiplication, it is shown that inversions are only needed on appropriate input and output signals to/from the adder tree or array, no changes are necessary internally to the tree or array structure. Any single, or two or more alternating, optimal adder design(s) may thus be used internally in a tree or array structure. Only at the external interface to the array will it be necessary to add or remove inverters, adapting to the digit set used in the external environment.

After a review of some redundant adder designs from [7] for a variety of digit-sets (3- and 4-element), for which claims were made about principal differences and relative speeds, it was shown that they can in fact all be implemented by the same basic adder, just using some other interface wiring to the environment. Hence there is no principal difference between the use of, say, the digit-sets $\{0, 1, 2, 3\}$ and $\{0, 1, 2\}$, the difference is just a question of interpretation of which signal pairs constitute digit encodings.

Finally another proposal in [6] to use a radix-2, 4-element digit-set with a 3-bit digit encoding, has been reviewed. It was shown that this design is only marginally different from the adder in [7] also employing the digit-set $\{0, 1, 2, 3\}$, and thus can also be interpreted as a normal 4-to-2, carry-save adder. It was furthermore shown that the 3-bit encoding is not necessary, a standard 2-bit encoding can be used, thus significantly reducing the complexity of the wiring of an adder tree. Actually, the difference is only in the interpretation of where the boundaries between the levels in a multiplier tree are positioned.

All the proposed 4-to-2 adder designs, employing digit or carry encodings of the form $d' \pm d''$ or $\pm 2d' \pm d''$ (including also the sign-magnitude encoding), have thus been shown to be equivalent to a 4-to-2, carry-save adder over the standard carry-save digit-set $\{0, 1, 2\}$ with encoding $d' + d''$, except possibly for a few signal inversions. It is conjectured that there are no other fundamentally different digit encodings that will allow for faster implementations.

Note

1. This is the notation used in [7], note that due to EWG this is **not** set addition.

References

1. C.S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, 1964, pp. 14–17. Reprinted in [18].
2. L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Freq.*, vol. 34, 1965, pp. 349–356. Reprinted in [18].
3. A. Weinberger, "4-2 Carry-Save Adder Module," *IBM Technical Disclosure Bulletin*, vol. 23, 1981.
4. N. Takagi, H. Yasuura, and S. Yajima, "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," *IEEE Transactions on Computers*, vol. C-34, no. 9, 1985, pp. 789–796.
5. S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi, "Design of High Speed MOS Multiplier and Divider Using Redundant Binary Representation," in *Proc. 8th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, 1987, pp. 80–86.
6. M.D. Ercegovac and T. Lang, "Effective Coding for Fast Redundant Adders using the Radix-2 Digit Set $\{0, 1, 2, 3\}$," in *Proc. 31st Asilomar Conf. Signals Systems and Computers*, pp. 1163–1167, 1997.
7. D.S. Phatak, T. Goff, and I. Koren, "Constant-Time Addition and Simultaneous Format Conversion Based on Redundant Binary Representations," *IEEE Transactions on Computers*, vol. 50, no. 11, 2001, pp. 1267–1278.
8. J. Vuillemin, "A Very Fast Multiplication Algorithm for VLSI Implementation," *INTEGRATION, the VLSI Journal*, vol. 1, 1983, pp. 39–52. Reprinted in [19].

9. M.R. Santoro and M.R. Horowitz, "A Pipelined 64 × 64b Iterative Array Multiplier," in *Proc. IEEE International Solid-State Circuit Conference*, 1988, pp. 36–37.
10. Z.J. Mou and F. Jutand, "'Overturned-Stairs' Adder Trees and Multiplier Design," *IEEE Transactions on Computers*, vol. 41, no. 8, 1992, pp. 940–948.
11. N. Ohkubo, M. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome, "A 4.4 ns CMOS 54 × 54-b Multiplier Using Pass Transistor Multiplexer," *IEEE Journal of Solid State Circuits*, vol. 30, no. 3, 1995, pp. 251–257.
12. V.G. Oklobdzija and D. Villeger, "Improving Multiplier Design by Using Improved Column Compression Tree and Optimized Final Adder in CMOS Technology," *IEEE Transactions on VLSI*, vol. 3, no. 2, 1995, pp. 292–301.
13. V.G. Oklobdzija, D. Villeger, and S.S Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach," *IEEE Transactions on Computers*, vol. 45, no. 3, 1996, pp. 294–306.
14. P.F. Stelling, C.U. Martel, V.G. Oklobdzija, and R. Ravi, "Optimal Circuits for Parallel Multipliers," *IEEE Transactions on Computers*, vol. 47, no. 3, 1998, pp. 273–285.
15. W.-C. Yeh and C.-W Jen, "High-Speed Booth Encoded Parallel Multiplier Design," *IEEE Transactions on Computers*, vol. 49, no. 7, 2000, pp. 692–701.
16. P. Kornerup, "Digit-Set Conversions: Generalizations and Applications," *IEEE Transactions on Computers*, vol. C-43, no. 6, 1994, pp. 622–629.
17. T. Aoki, Y. Sawada, and T. Higuchi, "Signed-Weight Arithmetic and its Application to a Field-Programmable Digital Filter Architecture," *IEICE Trans. Electron.*, vol. E82, no. 9, 1999, pp. 1687–1698.
18. Earl E. Swartzlander (Ed.), *Computer Arithmetic*, Vol I. Dowden, Hutchinson and Ross, Inc., 1980. Reprinted by IEEE Computer Society Press, 1990.
19. Earl E. Swartzlander (Ed.), *Computer Arithmetic*, Vol II. IEEE Computer Society Press, 1990.

Peter Kornerup was born in Aarhus, Denmark, 1939. He received the mag.scient. degree in mathematics from Aarhus University, Denmark, in 1967. After a period with the University Computing Center, from 1969 involved in establishing the computer science curriculum at Aarhus University, where he helped found the Computer Science Department in 1971. Through most of the 70's and 80's he served as Chairman of that department. He spent a leave during 1975/76 with the University of Southwestern Louisiana, Lafayette, LA, four months in 1979 and shorter stays in many other years with Southern Methodist University, Dallas, TX, one month with Université de Provence i Marseille in 1996 and two months with Ecole Normale Supérieure de Lyon i 2001. Since 1988 he has been Professor of Computer Science at Odense University, now University of Southern Denmark, where he has also served a period as the Chairman of this department. His interests include compiler construction, microprogramming, computer networks and computer architecture, but in particular his research has been in computer arithmetic and number representations, with applications in cryptology and digital signal processing.

Prof. Kornerup has served on the program committees for numerous IEEE, ACM and other meetings, in particular he has been on the Program Committees for the 4th through the 16th IEEE Symposium on Computer Arithmetic, and served as Program Co-Chair for these symposia in 1983, 1991 and 1999. He has been guest editor for a number of journal special issues, and also served as an associate editor of the IEEE Transactions on Computers during 1991–95. He is a member of the IEEE.

kornerup@imada.sdu.dk