NEW SELF-CHECKING BOOTH MULTIPLIERS

MARC HUNGER*, DANIEL MARIENFELD**

* Department of Electrical Engineering and Computer Science
University of Paderborn, 33098 Paderborn, Germany
e-mail: mhunger@date.upb.de

** Department of Computer Science
University of Potsdam, 14415 Potsdam, Germany
e-mail: dmarien@uni-potsdam.de

This work presents the first self-checking Booth-3 multiplier and a new self-checking Booth-2 multiplier using parity prediction. We propose a method which combines error-detection of Booth-3 (or Booth-2) decoder cells and parity prediction. Additionally, code disjointness is ensured by reusing logic for partial product generation. Parity prediction is applied to a carry-save-adder with the standard sign-bit extension. In this adder almost all cells have odd fanouts and faults are detected by the parity. Only one adder cell has an even fanout in the case of Booth-3 multiplication. Especially, for even-number Booth-2 multipliers parity prediction becomes efficient. Since that prediction slightly differs from previous work which describes CSA-folded adders, formulas to predict the parity are developed here. The proposed multipliers are compared experimentally with existing solutions. Only 102% of the area of Booth-2 without error detection is needed for the self-checking Booth-3 multiplier.

Keywords: Booth multiplier, self-checking, parity-prediction, carry-dependent adder, 1-out-of-5 code.

1. Introduction

Transient faults caused by electrical noise or external radiation are of growing importance and must be detected on-line. As (Shivakumar et al., 2002) describes, they result in soft errors in output latches of a combinational circuit if:

1. an output depends on the faulty subcircuit with respect to the input (logical condition);
2. a pulse, resulting from faults, has a significant duration and amplitude (electrical condition);
3. a pulse, resulting from faults, arrives at the latches at the clock transition (latching window).

Since there are a lot of masking effects, transient faults usually result in single bit errors. These effects become smaller for faults in latches or faults in the logic near latches. Therefore circuits which detect single input faults, single stuck-at-faults and multiple output faults are of interest.

This paper presents a self-checking Booth-3 and -2 multiplier. Since data paths usually use the parity code and/or double rail code for error detection, we developed a parity checked multiplier with duplicated output.

In Booth multipliers the number system of one operand is to be changed by some simple decoding steps. Therefore, effort for additions can be reduced, but depending on the algorithm, hard multiples of one factor have to be generated. The proposed self-checking Booth-3 multiplier extends the output of each decoder cell to the 1-out-of-5 code, which is used in combination with the parity of multiplier \( X \) to detect faults in decoder cells and input faults. To generate the hard multiple \( 3 \times Y \) and to check multiplicand \( Y \) by its parity, the sum-bit-duplicated look-ahead-adder (Ocheretnij et al., 2001) is modified. The proposed Booth-2 multiplier uses an existing solution presented in (Marienfeld et al., 2005) to check decoder cells.

In both multipliers, as a final Carry-Propagate-Adder (CPA), a sum-bit duplicated Carry-Ripple-Adder (CRA) conforming to (Marienfeld et al., 2004) is used together with parity prediction of a Carry-Save-Adder (CSA) consisting of carry-dependent adder cells and realizing the standard sign extension.
The following sections briefly describe basics of the Booth multiplier and self-checking adder networks. Section 4 proposes a new multiplier and Section 5 gives experimental results. Basics on computer arithmetics are presented in (Parhami, 2001), and (Lala, 2001) describes self-checking digital design.

2. Booth multiplier

Booth multipliers save costs (time and area) for adding partial products. They are reported, e.g., in (Booth, 1951; Al-Twaijry and Flynn, 1995). Figure 1 shows an example of signed Booth-2 multiplication. The multiplier $X = X_{n-1} \ldots X_0$ is transformed from the two’s complement to a radix-4 Booth code. The multiplicand $Y = Y_{n-1} \ldots Y_0$ is multiplied with these digits to generate partial products of the form $Y \times [\pm 0, \pm 1, \pm 2]$. These partial products are in two’s complement representation and the most significant bit (MSB) serves as the sign-bit. The numbers are weighted and added in a combinatorial or sequential fashion using an adder network to form the product $P = P_{2n-1} \ldots P_0$. With the higher radix the number of additions is reduced and the redundant Booth code reduces costs for generating partial products in a higher radix system. As Fig. 1 shows, the number of summands is halved in contrast to the classical binary multiplication. Additionally, each partial product can be generated via simple shifts and bit inversion.

Figures 2 and 3 show the logical architecture of combinatorial Booth-2 and -3 multipliers. These circuits include a Partial-Product-Generator (PPG), a Carry-Save-Adder (CSA) and a final Carry-Propagate-Adder (CPA). The PPG decodes operand $X$ into the Booth code with a higher radix. In the case of Booth-2, the radix is 4 and each decoded digit is in the set $[-2:2]$. Booth-3 uses a radix of 8 with digits $[-4:4]$. Each digit is decoded into selection signals $Sel_i$, which represent the multiple to select and the signal $N$, which is needed for inversions. Thus decoding partial products can be efficiently selected (Figs. 2 and 3 describe decoding and selection logic for Booth-2 and -3; these circuits result from Tables 2 and 3).

The hard multiple $3 \times Y$ needed in Booth-3 multi-
New self-checking Booth multipliers

Fig. 4. Booth-2 selection circuit and decoder.

Fig. 5. Booth-3 selection circuit and decoder.

Fig. 6. 16-bit Booth-3 with full sign-extension.

Fig. 7. 16-bit Booth-3 with CSA using correction terms.

Fig. 8. 16-bit Booth-2 with CSA using correction terms.

Fig. 9. 16-bit CSA-folded Booth-3.

pliers is generated using a CPA and all other multiples are generated by simple logical operations. The CSA reduces all partial products to two summands, which are added by the final CPA into the product \( P \). There exist various structures to perform these kinds of additions based on full or half adders. The CSA can be realized as a tree or an array (Al-Twaijry and Flynn, 1995). The methods differ in their regularity, delay, area usage, wiring effort or power consumption. The CPA (Parhami, 2001) can be realized as a simple chain of full adders (carry-ripple-adder). It is possible to fasten carry-propagation by additional look-ahead or skip units. Alternatively, carries can be generated speculatively. The correct result can be obtained using a multiplexer. In this work, the CPA is realized as a carry-ripple-adder and a linear array serves as the CSA. To generate the hard multiple \( 3 \times Y \), we choose a fast carry-look-ahead adder. This fastens the component with a small drawback in the area usage. But the described methods are independent of that choice.

Since in Booth multipliers the partial products are signed (including both integer and natural multiplications), they need to be extended to a \( 2 \times n \) bit length. In two’s complement representation an extension is performed by a sign-bit extension as shown in the DOT-scheme in Fig. 6. Here dots represent the bits generated via selector cells. The resulting DOT-scheme is in trapezoidal form and can be optimized by two known solutions reported in (Sparmann and Reddy, 1994). The standard sign-extension, or a “CSA with correction terms”, applies the equation

\[
SSSS \cdot \cdot \cdot = 111 \overline{S} \cdot \cdot \cdot + 00010000
\]  

(or a similar one) to each partial product. The resulting constants are added and Fig. 7 serves as an invariant for Booth-3. The resulting Booth-2 DOT-scheme is shown in Fig. 8. Special adder cells add remaining ones with the corresponding bits of partial products. The number of DOTS and therefore the number of full adders are reduced.

The second solution, described in (Nicolaidis and Duarte, 1998) and applied to Booth-3 in Fig. 9, consists of adder cells with multiple fanouts (CSA-folding). Here we use the fact that all half adders, adding the sign-bits of
the first two partial products, generate the same outputs. Therefore only one adder with multiple fanouts is needed. The same fact is true for adders in further rows. Thus in these rows redundant adder cells are saved, too. As in the first solution, the cost reduction is of significance. As described in Section 4.1, the parity of the partial products for standard sign-extension is different in comparison with a folded CSA.

3. Self-checking adders

To detect errors on-line, some kind of redundancy is needed. This includes time and information redundancy. In this work, with parity and the double rail code, information redundancy is used. Inputs \( I \) and outputs \( O \) for circuits are extended to code words, which are checked to detect errors. Codes for arithmetic circuits include, e.g., parity (the number of ones in \( I \) modulo 2), the Berger code (the number of zeros in \( I \)), as reported in (Lo et al., 1993), Bose-Lin Codes, described in (Gorshe and Bose, 1996), or the residue code reported in (Sparmann and Reddy, 1994).

Figure 10 shows a parity checked CSA with a final CRA. The input code includes the operands and the parity of all summands (or partial products in the multiplier). The output code includes the sum and both the generated parity \( P_{Out} \) and the predicted parity given by

\[ P_{Out} = P_{In} \oplus P_C, \]

where \( P_C \) is the parity of all carries.

For an on-line error detection, the following properties should be achieved, as described in (Goessel and Graf, 1993):

- **code-disjointness**: A circuit is called code-disjoint if each non code word is mapped to a non-output code word.

- **fault-security**: A circuit is called fault-secure if for all faults (in the fault model) there is no input code word that causes the faulty circuit to output an incorrect code word.

- **self-testing**: A circuit is called self-testing if for all faults (in the fault model) there is at least one input code word such that the corresponding output is not a code word.

- **self-checking**: A circuit is called totally self checking if it is self-testing and fault-secure.

Code-disjointness ensures that faults in input latches are detectable. The fault-secure property ensures that the circuit under a fault either generates correct outputs or detects the fault. In self-testing circuits each fault can be tested by applying an input vector. To achieve this property, redundancies in circuits should be carefully analyzed.

For arbitrary adder cells, the adder described in Fig. 10 is not self-checking. Not detected are stuck-at-faults in adder cells, which affects only the carry-output. That faulty signal affects the generated parity via a sum output. The predicted parity is affected through \( P_C \). Therefore the following redundant adder cells are needed in a self-checking adder:

1. A **carry-duplicated adder**, presented in (Nicolaidis et al., 1997), duplicates the carries (Fig. 11). The propagate signal \( P \) can be shared by the sum-output and both carries. One of the carries is needed for carry-propagation and the other generates \( P_C \) and therefore predicts \( P_{Out} \). That structure ensures that always odd or no outputs of the cell are erroneous.

2. A **carry-dependent adder**, proposed in (Hsiao and Sellers, 1963), forces a faulty carry to affect the corresponding sum-output. Here the sum-bit is realized as \( S = f \oplus C \) (with \( f \) having a suitable truth table and a structure to meet the self-checking property). If \( f \) and \( C \) share no logic, then each fault affects either the signal \( f \) or \( C \), or none of them. If \( C \) and \( f \) share an OR-gate 1, as shown in Fig. 12, the same property is saved. Also the special half adder, shown in Fig. 13, and the trivial cell adding \( A + 1 \) (here \( f \) is logical
New self-checking Booth multipliers

4. Proposed Booth multiplier

This section describes the proposed self-checking Booth-2 and Booth-3 multipliers. Caused by the sign-extension of $Y$ and the decoding of $X$, parity prediction is not code-disjoint and so both inputs must be checked locally. Faults in decoder cells can generate multiple erroneous bits of one partial product. Faults in the $3 \times Y$-CPA can generate multiple erroneous bits of multiple partial products. So both components need to be checked locally, too.

In this work a carry-dependent adder is used for the CSA since it gave better results in delay and area overhead.

Figure 15 shows a sum-bit duplicated carry-dependent adder cell used in the CPA given in Fig. 16. Note that the logic near latches is more sensitive to soft errors. Therefore that logic is partially duplicated and one of the sums is realized as carry-dependent. Since the carries are checked, a look-ahead adder without duplicated look-ahead units can be used. The parity of the propagated signals $P$ is checked by the input parity. As shown in (Marienfeld et al., 2004), a parity-checked carry-save-adder can be combined efficiently with a final sum-bit duplicated carry-dependent CPA to increase the error detection capability of a multiplier.

4.1. Self-checking Booth-2 multiplier. Figure 17 shows the architecture of the proposed Booth-2 multiplier. This multiplier is almost the same as that presented in (Marienfeld et al., 2005), but a different CSA, and therefore different parity prediction, are applied. Operand $Y$ is checked via a parity tree. The decoder is partially duplicated and checked by parity trees. These trees output needed signals to predict the output parity of the adder network. The following describes the code-disjoint and self-checking decoder (Fig. 18):

1. The parity $P_{Sel1}$ of all $Sel1_i$ is checked by the parity and the MSB of $X$.
2. The signals $Sel2$ of each decoder are duplicated and their parities $P_{Sel2}$ and $P_{Sel2D}$ are compared.

With these preparations, odd input faults and single stuck-at-faults are detected. At the same time, with $P_{Sel1}$ and $P_{Sel2}$, signals to predict the partial products parity are computed. Additionally, $P_Y$ is checked by a parity tree.
In contrast to (Nicolaidis and Duarte, 1998), duplicating the decoder (and using a two-rail-checker) is avoided and costs are saved.

The CSA realizes the standard sign-extension instead of CSA-folding, as performed in (Marienfeld et al., 2005; Nicolaidis and Duarte, 1998). The adder network is described in Fig. 19 and includes adder cells described in Section 3. For the standard sign-extended CSA, there are no cells with even fan-out, including adder and selector cells. Since in a CSA-folded Booth-2 multiplier some cells have even fan-outs, in (Nicolaidis and Duarte, 1998) the sum-circuit, including selector cells, of all most significant adder cells was duplicated to avoid even “sum-path parities”. So the proposed method saves a parity tree and a set of selector cells.

As Fig. 19 shows, in case of “correction terms” the parity of the $i$-th partial product ($i > 0$) is given by (3). The first term computes the parity of the lower bits (given by the selector cells and using the fact that both $Sel_{1i}$ and $Sel_{2i}$ are never equal to 1). The second term computes the parity of the sign-bit, which is inverted. The term $N_i$ represents the signal added to the partial product in the least significant position and the last term 1 is added in the most significant position. This equation simplifies to

$$P_{Part_{i>0}} = \sum_{j=0}^{n-1} (Sel_{1i} * Y_j + Sel_{2i} * Y_{j-1} \oplus N_i) \oplus (Sel_{1i} * Y_{n-1} + Sel_{2i} * Y_{n-1} \oplus N_i \oplus 1) \oplus N_i \oplus 1 = (Sel_{1i} \oplus Sel_{2i}) \oplus P_Y \oplus Sel_{1i} * Y_{n-1}. \quad (3)$$
New self-checking Booth multipliers

The parity of the first partial product is

\[ P_{\text{Part}_{i=0}} = \sum_{j=0}^{n-1} (\text{Sel}_{1_i} \cdot Y_j \oplus \text{Sel}_{2_i} \cdot Y_{j-1} \oplus N_i) \]

\[ \oplus (\text{Sel}_{1_i} \cdot Y_{n-1} \oplus \text{Sel}_{2_i} \cdot Y_{n-1} \oplus N_i) \]

\[ \oplus (\text{Sel}_{1_i} \cdot Y_{n-1} \oplus \text{Sel}_{2_i} \cdot Y_{n-1} \oplus N_i) \]

\[ \oplus (\text{Sel}_{1_i} \cdot Y_n \oplus \text{Sel}_{2_i} \cdot Y_n \oplus N_i \oplus 1) \]

\[ \oplus N_i \]

\[ = (\text{Sel}_{1_i} \oplus \text{Sel}_{2_i}) \cdot P_Y \oplus \text{Sel}_{1_i} \cdot Y_{n-1} \oplus 1 \]. (4)

Here the last term 1 is saved. The last equality results from the fact that, since the sign-bit is tripled, it is added once to the parity (as in all other partial products).

The partial product parity is given by

\[ P_{\text{Part}} = \sum_{i=0}^{n/2-1} ((\text{Sel}_{1_i} \oplus \text{Sel}_{2_i}) \cdot P_Y \oplus \text{Sel}_{1_i} \cdot Y_{n-1}) \]

\[ \oplus 1 \]

\[ = (P_{\text{Sel1}} \oplus P_{\text{Sel2}}) \cdot P_Y \oplus P_{\text{Sel1}} \cdot Y_{n-1} \oplus 1 \]. (5)

This parity is independent of \( P_N \). As shown in (Nicolaidis and Duarte, 1998), parity prediction in a CSA-folded Booth-2 multiplier for even length operands is performed by

\[ P_{\text{Part}} = (P_{\text{Sel1}} \oplus P_{\text{Sel2}}) \cdot P_Y \oplus P_{\text{Sel1}} \cdot Y_{n-1} \oplus 1 \]. (6)

Since this equation depends on \( P_N \), the proposed method does not require the generation (by a parity tree or a two-rail-checker) of that signal.

Since a “CSA with correction terms” differs only in the cells given in Figs. 13 and 14 from a folded CSA the self-checking properties are preserved.

4.2. Self-checking Booth-3 multiplier. Figure 20 shows the proposed Booth-3 multiplier. Since Booth-3 decoders are more complex and need more area, (partially) duplicating becomes more expensive. Additionally, the parity of \( X \) is not directly computable by decoder outputs.

The following method, presented in (Hunger, 2006), makes the Booth-3 decoder code-disjoint and self-checking.

1. Decoder outputs are extended by \( \text{Sel}_{0} \) to the 1-out-of-5 code (Fig. 21 and Tables 2 and 3). To combine code checking and parity prediction, for each \( \text{Sel}_{i} \) an own parity tree forms \( P_{\text{Sel}_{i}} \). These parities are checked by the equation

\[ P_{\text{Sel0}} \neq P_{\text{Sel1}} \oplus P_{\text{Sel2}} \]

\[ \oplus P_{\text{Sel3}} \oplus P_{\text{Sel4}} \]. (7)

where the equal sign is used if the number of decoder cells is even and the unequal sign is used otherwise.

2. The interior XOR-layer is checked by the parity of \( X \). For a 8-bit decoder we use

\[ P_X = P_{XOR2} \oplus P_N \oplus X_{n-1} \], (8)

\[ P_X = P_{XOR1,3} \oplus P_N \]. (9)

The following analyses the self-testing property of the decoder. The interior XOR-layer and the trees to compute \( P_{XOR2} \) and \( P_{XOR1,3} \) can be tested, since all inputs of \( X \) can be applied. Also the NOR-layer and the parity of
Fig. 24. 16-bit self-checking Booth-3 CSA using correction terms.

Table 1. Area overhead in % of the multiplier.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booth-2</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Booth-3</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Booth-2 acc. to Sec. 4</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>88</td>
</tr>
<tr>
<td>Booth-3 acc. to Sec. 4</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>121</td>
</tr>
<tr>
<td>Booth-2 acc. to (Nicolaidis and Duarte, 1998) ext. by a parity tree</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>165</td>
</tr>
<tr>
<td></td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>142</td>
</tr>
</tbody>
</table>
New self-checking Booth multipliers

lowing terms represent the two most significant bits (the last one is inverted). The last summand $N_i$ is the signal added to the LSB. That equation simplifies to (11).

The first partial products parity is given by

$$P_{\text{Part}_{i=0}} = (Sel1_0 \oplus Sel2_0 \oplus Sel4_0) * P_Y$$
$$\oplus (Sel1_0 \oplus Sel4_0) * Y_{n-1}$$
$$\oplus Sel3_0 * (3Y + (3Y)_{n+1} + 1).$$

(12)

If $n \equiv 1 \mod 3$, as in 16- or 64-bit multipliers, the parity of the last partial product is given by

$$P_{\text{Part}_{L}} = \sum_{j=0}^{n-1} (Sel1_L * Y_j \oplus N_L) \oplus 1 \oplus 1 \oplus N_L$$
$$= Sel1_L * P_Y \oplus N_L.$$ (13)

In 8- and 32-bit multipliers that parity slightly differs from this equation. We have

The parity of all partial products for $n \equiv 1 \mod 3$ is computed by

$$P_{\text{Part}} = P_{\text{Part}_{0}} + P_{\text{Part}_{L}} + \sum_{i=1}^{L-1} P_{\text{Part}_{i}}$$
$$= (P_{Sel1} \oplus P_{Sel2} \oplus P_{Sel4}) * P_Y$$
$$\oplus P_{Sel3} * P_{3Y} \oplus Sel3_0 * (Y_{n-1})$$
$$\oplus (P_{Sel2} \oplus Sel1_0 \oplus Sel2_0 \oplus Sel4_0) * Y_{n-1}$$
$$\oplus P_{N} \oplus N_0 \oplus 1.$$ (14)

It depends on the number of partial products; the remaining ones in the partial products are added to one or zero (the last term in (14)). So (14) is valid for $n = 16$ or 64.

5. Experimental results

The multiplier was implemented in VHDL, tested by ModelSim and mapped to the vttlib25 cmos library (Sulistyo and Ha, 2002, 2003) by the Synopsis design compiler. Optimization was performed for the area and delay, such that all multipliers have almost the same delay. Results for the operand sizes 8, 16, 32 and 64 are given in Table [I]

By increasing operand sizes, Booth-3 becomes smaller than Booth-2. For 64 bits the method needs 88% of the area of Booth-2.

For 64 bits the proposed self-checking Booth-3 multiplier only needs 102% of the area of Booth-2 without error detection and 115% of Booth-3 without error detection. For the same configuration, the proposed self-checking Booth-2 multiplier needs 122% of Booth-2 without error detection. Thus the higher performance for

Booth-3 in the linear carry-save adder is preserved and slightly strengthened.

The Booth-2 multiplier presented in (Nicolaidis and Duarte, 1998) and extended by one parity tree has an overhead of 142%. Thus, a significant cost reduction in comparison with that multiplier could be achieved. We emphasize that the small extra parity tree was inserted to check the parity of $Y$ (the parity of $X$ can be checked by some two-rail-checker outputs), to meet the code-disjoint property and to make all multipliers comparable.

6. Conclusions

An efficient self-checking code-disjoint Booth-3 multiplier was proposed based on a linear CSA and a final CRA. The multiplier detects single input faults, single stuck-at-faults at logical gates and all errors in an output register. An error detection scheme without duplicating decoder cells was presented and combined with parity prediction. The code-disjoint-property could be combined with forming partial products and no extra parity tree is needed. Parity prediction for Booth-2 and -3 multipliers was developed using the standard sign-extension.

Acknowledgments

The authors would like to thank Prof. Michael Goessel from the Fault Tolerant Computing Group at the University of Potsdam for stimulating discussions.
References


Received: 25 January 2007
Revised: 26 May 2007
Re-revised: 9 May 2008