Chapter 11
Decimal Operations

In a number of computer arithmetic applications, decimal systems are preferred to the binary ones. The reasons come, not only from the complexity of coding/decoding interfaces but, mostly from the lack of precision in the results of the binary systems.

For the following circuits the input and output operators are supposed to be encoded in Binary Encoded Digits (BCD).

11.1 Addition

Addition is a primitive operation for most arithmetic functions, and then it deserves special attention. The general principles for addition are in Chap. 7, and in this section we examine special consideration for efficient implementation of decimal addition targeting FPGAs.

11.1.1 Decimal Ripple-Carry Adders

Consider the base-B representations of two n-digit numbers

$$x = x_{n-1} \cdot B^{n-1} + x_{n-2} \cdot B^{n-2} + \cdots + x_0 \cdot B^0,$$
$$y = y_{n-1} \cdot B^{n-1} + y_{n-2} \cdot B^{n-2} + \cdots + y_0 \cdot B^0.$$

The following (pencil and paper) Algorithm 11.1 computes the $(n + 1)$-digit representation of the sum $z = x + y + c_{in}$ where c_{in} is the initial carry.

J.-P. Deschamps et al., *Guide to FPGA Implementation of Arithmetic Functions*,
Lecture Notes in Electrical Engineering 149, DOI: 10.1007/978-94-007-2987-2_11,

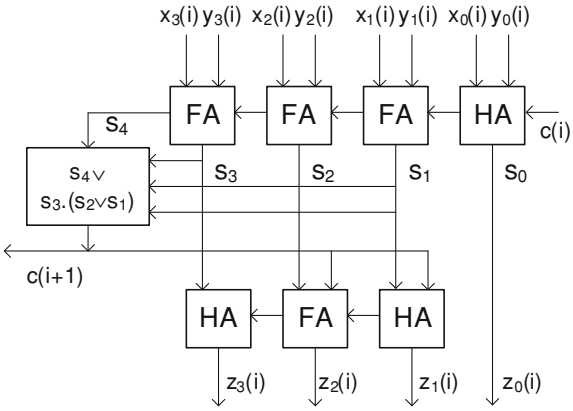

Fig. 11.1 Decimal ripple-carry adder cell

Algorithm 11.1: **Classic addition (ripple carry)**

```
c(0) := c_in;
for i in 0 .. n-1 loop
    if x(i) + y(i) + c(i) > B-1 then c(i+1) := 1;
    else c(i+1) := 0; end if;
    z(i) := (x(i) + y(i) + c(i)) mod B;
end loop;
z(n) := c(n);
```

For $B = 10$, the classic ripple-carry for a *BCD* decimal adder cell can be implemented as suggested in Fig. 11.1 The mod-10 addition is performed adding 6 to the binary sum of the digits, when a carry for the next digit is generated. The VHDL model *ripple_carry_adder_BCD.vhd* is available at the Authors' web page.

As described in Chap. 7, the na implementation of an adder (ripple-carry, Fig. 7.1) has a meaningful critical path. In order to reduce the execution time of each iteration step, Algorithm 11.1 can be modified as shown in the next section.

11.1.2 Base-B Carry-Chain Adders

In order to improve the ripple-carry adder, a better solution is the use of two binary functions of two B-valued variables, namely the *propagate* (P) and *generate* (G) functions.

$$p_i = p(x_i, y_i) = 1 \text{ if } x_i + y_i = B - 1, p(x_i, y_i) = 0 \text{ otherwise;}$$
$$g_i = g(x_i, y_i) = 1 \text{ if } x_i + y_i \geq B, g(x_i, y_i) = 0 \text{ if } x_i + y_i \leq B - 2, \text{ otherwise, any value.}$$

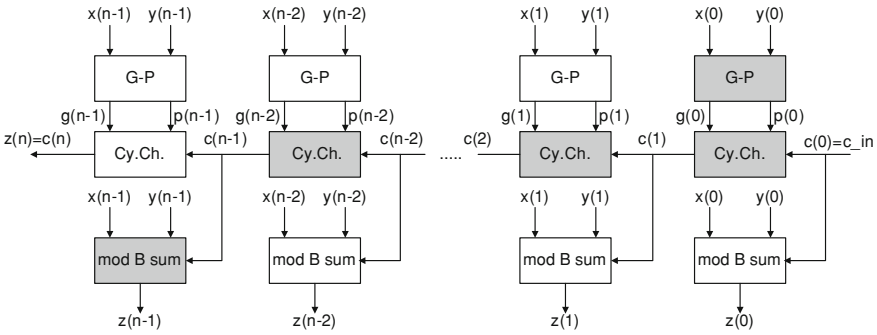

Fig. 11.2 n-digits carry-chain adder

So, c_{i+1} can be expressed under the following way:

$$c_{i+1} = p_i \cdot c_i + \text{not}\,(p_i) \cdot g_i.$$

The corresponding modified Algorithm 11.2 is the following one.

Algorithm 11.2: Carry-chain addition

```
--computation of the generation and propagation conditions:
for i in 0 .. n-1 loop
    g(i) := p(x(i), y(i)); p(i) := p(x(i) ,y(i));
end loop;
c(0) := c_in;  --carry computation:
for i in 0 .. n-1 loop
     if p(i) = 1 then c(i+1) := c(i); else c(i+1) := g(i); end if;
end loop;
for i in 0 .. n-1 loop --sum computation
    z(i) := (x(i) + y(i) + c(i)) mod B;
end loop;
z(n) := c(n);
```

The use of *propagate* and *generate* functions allow generating a *n*-digit adder carry-chain array of Fig. 11.1. It is based on the Algorithm 11.2. The *Generate-Propagate* (*G-P*) cell calculates the *Generate* and *Propagate* functions; and the *carry-chain* (*Cy.Ch*) cell computes the next carry. Observe that the carry-chain cells are binary circuits, whereas the *generate-propagate* and the *mod B sum* cells are *B*-ary ones. As regards the computation time, the critical path is shaded in Fig. 11.2. (It has been assumed that $T_{sum} > T_{Cy.\ Ch}$)

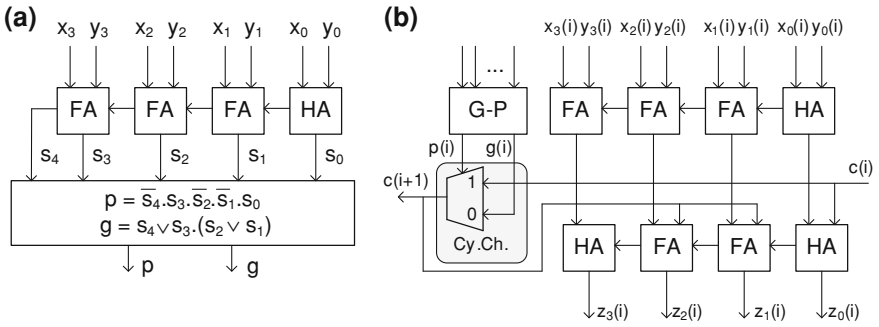

Fig. 11.3 **a** Simple *G-P* cell for *BCD* adder. **b** Carry-chain *BCD* adder *i*th digit computation

11.1.3 Base-10 Carry-Chain Adders

If $B = 10$, the carry-chain circuit remains unchanged but the P and G functions as well as the modulo-10 sums are somewhat more complex. In base 2 (Chap. 7), the P and G cells are respectively synthesized by *XOR* and *AND* functions, while in base 10, P *a*nd G are now defined as follows:

$$p_i = 1 \text{ if } x_i + y_i = 9,\ p_i = 0 \text{ otherwise;} \tag{11.1}$$

$$g_i = 1 \text{ if } x_i + y_i > 9,\ g_i = 0 \text{ otherwise;} \tag{11.2}$$

A straightforward way to synthesize P and G is shown at Fig. 11.3a. That is add the BCD numbers and detects if the sum is equal to nine and greater than nine respectively. Nevertheless, functions P and G may be directly computed from x_i, y_i inputs. The following formulas (11.3) and (11.4) are Boolean expressions of conditions (11.1) and (11.2).

$$p_i = P_0 \cdot [K_1 \cdot (P_3 \cdot K_2 \vee K_3 \cdot G_2) \vee G_1 \cdot K_3 \cdot P_2] \tag{11.3}$$

$$g_i = G_0 \cdot [P_3 \vee G_2 \vee P_2 \cdot G_1] \vee G_3 \vee P_3 \cdot P_2 \vee P_3 \cdot P_1 \vee G_2 \cdot P_1 \vee G_2 \cdot G_1 \tag{11.4}$$

where $P_j = x_j \oplus y_j$, $G_j = x_j \cdot y_j$ and $K_j = x_j^{'} \cdot y_j^{'}$ are the binary propagator, generator and carry-kill for the jth components of the *BCD* digits $x(i)$ and $y(i)$.

The *BCD* carry-chain adder *i*th digit computation is shown at Fig. 11.3b and it is made of a first binary *mod* 16 adder stage, a carry-chain cell driven by the *G-P* functions, and an output adder stage performing a correction (adding 6) whenever the carry-out is one. Actually, a zero carry-out $c(i + 1)$ identifies that the *mod* 16 sum does not exceed 9, so no corrections are needed. Otherwise, the *add*-6 correction applies. Naturally, the *G-P* functions may be computed according to Fig. 11.3, using the outputs of the *mod* 16 stage, including the carry-out s_4.

The VHDL model *cych_adder_BCD_v1.vhd* that implements a behavioral model of the decimal carry-chain adder is available at the Authors' web page.

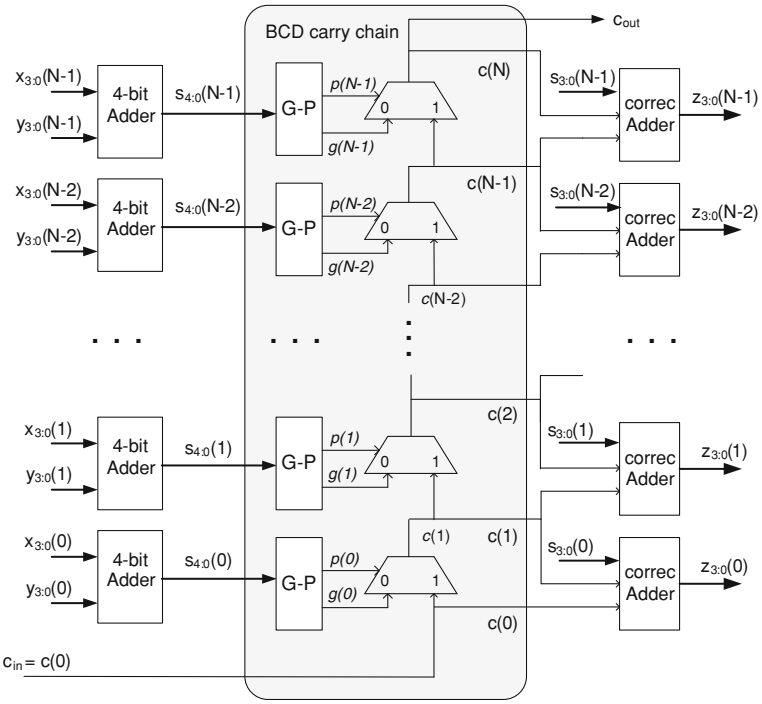

Fig. 11.4 *FPGA* implementation of an *N*-digit *BCD* Adder

11.1.4 FPGA Implementation of the Base-10 Carry-Chain Adders

The FPGA vendors provides dedicated resources to implement binary carry-chain adders [1, 2]. As mentioned in Chap. 7 a simple HDL description of a binary adder is implemented efficiently using the carry-logic. Nevertheless in order to use this resources in other designs it is necessary to instantiate the components manually.

Figure 11.4 shows a Xilinx implementation of the decimal carry-chain adder. The VHDL model *cych_adder_BCD_v2.vhd* that implements a low level component instantiation model of the decimal carry-chain adder is available at the Authors' web page.

Observe that the critical path includes a 4-bit adder, the G-P computation; the *n*-digits carry propagation and a final 3-bit correction adder. Xilinx 6-input/2-output *LUT* is built as two 5-input functions while the sixth input controls a 2-1 multiplexor allowing to implement either two 5-input functions or a single 6-input one; so *G* and *P* functions fit in a single *LUT* as shown at Fig. 11.5a.

Other alternatives to implement in FPGA the decimal carry-chain adders, include computing the functions *P* and *G* directly from the decimal digits ($x(i)$, $y(i)$ inputs) using the Boolean expressions, instead of the intermediate sum bits s_k [3].

For this purpose one could use formulas (11.3) and (11.4), nevertheless, in order to minimize time and hardware consumption the implementation of $p(i)$ and $g(i)$ is revisited as follows. Remembering that $p(i) = 1$ whenever the arithmetic sum $x(i) + y(i) = 9$, one defines a 6-input function $pp(i)$ set to be 1 whenever the arithmetic sum of the first 3 bits of $x(i)$ and $y(i)$ is 4. Then $p(i)$ may be computed as:

$$p(i) = (x_0(i) \oplus y_0(i)) \cdot pp(i). \tag{11.5}$$

On the other hand, $gg(i)$ is defined as a 6-input function set to be 1 whenever the arithmetic sum of the first 3 bits of $x(i)$ and $y(i)$ is 5 or more. So, remembering that $g(i) = 1$, whenever the arithmetic sum $x(i) + y(i) > 9$, $g(i)$, may be computed as:

$$g(i) = gg(i) \vee (pp(i) \cdot x_0(i) \cdot y_0(i)). \tag{11.6}$$

As Xilinx *LUT*s may compute 6-variable functions, then $gg(i)$ and $pp(i)$ may be synthesized using 2 *LUT*s in parallel while $g(i)$ and $p(i)$ are computed through an additional single *LUT* as shown at Fig. 11.5b.

11.2 Base-10 Complement and Addition: Subtration

11.2.1 Ten's Complement Numeration System

B's complement representation general principles are available in the literature. One restricts to 10's complement system to cope with the needs of this section. A one-to-one function $R(x)$, associating a natural number to x is defined as follows.

Every integer x belonging to the range $-10^n/2 \leq x < 10^n/2$, is represented by $R(x) = x \bmod 10^n$, so that the integer represented in the form $x_{n-1}x_{n-2}\cdots x_1x_0$ is

$$x_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \cdots + x_0 \text{ if } x_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \cdots + x_0 < 10^n/2, \tag{11.7}$$

$$x_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \cdots + x_0 - 10^n \text{ if } x_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \cdots + x_0 \geq 10^n/2. \tag{11.8}$$

The conditions (11.7) and (11.8) may be more simply expressed as

$$x_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \cdots + x_0 \text{ if } x_{n-1} < 5, \tag{11.9}$$

$$x_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \cdots + x_0 - 10^n \text{ if } x_{n-1} \geq 5. \tag{11.10}$$

Another way to express a 10's complement number is:

$$x'_{n-1} \cdot 10^{n-1} + x_{n-2} \cdot 10^{n-2} + \cdots + x_0 \tag{11.11}$$

where $x'_{n-1} = x_{n-1} - -10$ if $x_{n-1} \geq 5$ and $x'_{n-1} = x_{n-1}$ if $x_{n-1} < 5$, while the sign definition rule is the following one: if x is negative then $x_{n\_1} \geq 5$; otherwise $x_{n\_1} < 5$.

11.2.2 Ten's Complement Sign Change

Given an n-digit 10's complement integer x, the inverse $z = -x$ of x, is an n-digit 10's complement integer. Actually the only case $-x$ cannot be represented with n digits is when $x = -10^n/2$, so $-x = 10^n/2$. The computation of the representation of $-x$ is based on the following property. Assuming x to be represented as an n-digit 10's complement number $R(x)$, $-x$ may be readily computed as

$$-x = 10^{n+1} - R(x). \tag{11.12}$$

A straightforward inversion algorithm then consists of representing x with $n + 1$ digits, complementing every digit to 9, then adding 1. Observe that sign extension is obtained by adding a digit 0 to the left of a positive number, or 9 for a negative number, respectively.

11.2.3 10's Complement BCD Carry-Chain Adder-Subtractor

To compute $X + Y$ similar algorithm as in Algorithm 11.2 (Sect. 11.1.2) can be used. In order to compute $X - Y$, 10's complement subtraction algorithm actually adds $(-Y)$ to X.

10's complement sign change algorithm may be implemented through a digitwise 9's complement stage followed by an add-1 operation. It can be shown that the 9's complement binary components w_3, w_2, w_1, w_0 of a given *BCD* digit y_3, y_2, y_1, y_0 are expressed as

$$w_3 = y'_3 \cdot y'_2 \cdot y'_1; \quad w_2 = y_2 \oplus y_1; \quad w_1 = y_1; \quad w_0 = y'_0 \tag{11.13}$$

An improvement to the adder stage could be carried out by avoiding the delay produced by the 9's complement step. Thus, this operation may be carried out within the first binary adder stage, where $p(i)$ and $g(i)$ are computed as

$$\begin{aligned}
&p_0(i) = x_0(i) \oplus y_0(i) \oplus \left(A'/S\right); \; p_1(i) = x_1(i) \oplus y_1(i); \\
&p_2(i) = x_2(i) \oplus y_2(i) \oplus y_1(i).\left(A'/S\right) \\
&p_3(i) = x_3(i) \oplus \left(y_3(i)' \cdot y_2(i)' \cdot y_1(i)'\right) \cdot \left(A'/S\right) \oplus y_3(i) \cdot \left(A'/S\right)' \qquad (11.14) \\
&g_k(i) = x_k(i), \; \forall k. \qquad (11.15)
\end{aligned}$$

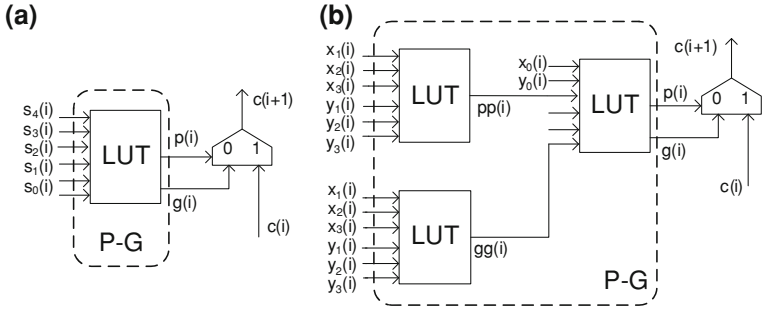

Fig. 11.5 *FPGA* carry-chain for decimal addition. **a**. P-G calculation using an intermediate addition. **b**. P-G calculation directly from BCD digits

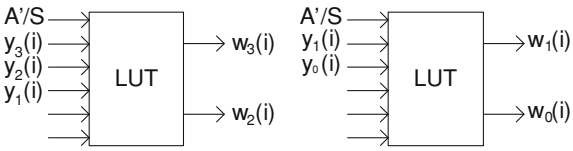

Fig. 11.6 *FPGA* 9's complement circuit for *BCD* digit

A third alternative is computing G and P directly from the input data. As far as addition is concerned, the P and G functions may be implemented according to formulas (11.4) and (11.5). The idea is computing the corresponding functions in the subtract mode and then multiplexing according to the add/subtract control signal A'/S.

11.2.4 FPGA Implementations of Adder Subtractors

To compute $X - Y$, 10's complement subtraction algorithm actually adds $(-Y)$ to X. So for a first implementation, Fig. 11.6 presents a 9's complement implementation using 6-input/2-output *LUT*s, available in the Xilinx (Virtex-5, Virtex-6, spatan6, 7-series) technology. A'/S is the add/subtract control signal; if $A'/S = 1$ (subtract), formulas (11.13) apply, otherwise $A'/S = 0$ and $w_j(i) = y_j(i) \forall i, j$.

The complete circuit is similar to the circuit of Fig. 11.4, but instead of input y, the input w as produced by the circuit of Fig. 11.5.

The better alternative intended to avoid the delay produced by the 9's complement step, embeds the 9's complementation within the first binary adder stage, as depicted in Fig. 11.7a, where $p(i)$ and $g(i)$ are computed as explained in (11.14) and (11.15).

The VHDL model *addsub_1BDC.vhd* that implements the circuit of Fig. 11.7a is available at the Authors' web page. The VHDL model has two architectures a behavioral and a low level that instantiates components (Luts, muxcy, xorcy, etc.).

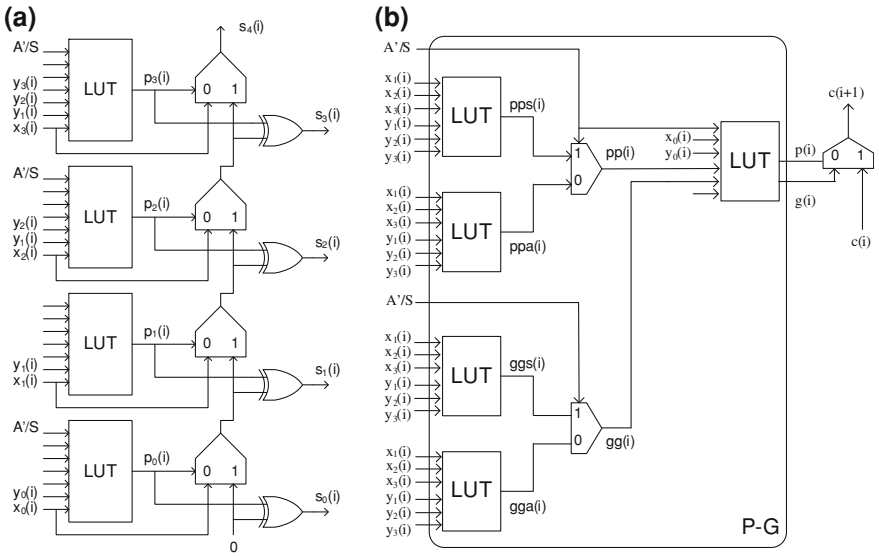

Fig. 11.7 *FPGA* implementation of adder-subtractor. **a** Adder-subtractor for one BCD digit. **b** Direct computation of P-G function

Then we can use the circuit of Fig. 11.6a to compute the P-G function using the previous computed addition of BDC digits. The VHDL model *addsub_BDC_v1.vhd* that implements a complete decimal adder-subtractor and it is available at the Authors' web page. To complete the circuit, a final correction adder (correct_add.vhd) corrects the decimal digit as a function of the carries.

The third alternative is computing G and P directly from the input data. For this reason, assuming that the operation at hand is $X + (\pm Y)$, one defines on one hand $ppa(i)$ and $gga(i)$ according to (11.4) and (11.5) (Sect. 11.1.4), i.e. using the straight values of Y's *BCD* components. On the other hand, $pps(i)$ and $ggs(i)$ are defined using $w_k(i)$ as computed by the 9's complement circuit (11.13). As $w_k(i)$ are expressed from the $y_k(i)$ both $pps(i)$ and $ggs(i)$ may be computed directly from $x_k(i)$ and $y_k(i)$. Then the correct $pp(i)$ and $gg(i)$ signal is selected according to the add/subtract control signal A'/S. Finally, the *propagate* and *generate* function are computed as:

$$p(i) = (x_0(i) \oplus y_0(i) \oplus (A'/S)) \cdot pp(i), \tag{11.16}$$

$$g(i) = gg(i) \vee (pp(i) \cdot x_0(i) \cdot (y_0(i) \oplus (A'/S))). \tag{11.17}$$

Figure 11.7b shows the Xilinx LUT based implementation. The multiplexers are implemented using dedicated muxF7 resources.

The VHDL model *addsub_BDC_v2.vhd* that implements the adder-subtractor using for the P-G computation from the direct inputs (*carry-chain_v2.vhd*) is available at the Authors' web page.

Fig. 11.8 Binary to BCD arithmetic reduction

	80	40	20	10	8	4	2	1
		p_6	p_5	p_4	p_3	p_2	p_1	p_0
+			p_6	p_5		p_4	p_4	
						p_6	p_5	
	d_3	d_2	d_1	d_0	c_3	c_2	c_1	c_0

11.3 Decimal Multiplication

This section presents approaches for *BCD* multipliers. It starts by discussing algorithmic alternatives to implement a 1×1 *BCD* digit multiplier. Then, this section proposes an implementation for an $N \times 1$ *BCD* multiplier. This operation can be used to carried out an $N \times M$ multiplier, but by itself, the $N \times 1$ multiplication appears to be a primitive for other algorithms, such as logarithm or exponential functions.

11.3.1 One-Digit by One-Digit BCD Multiplication

11.3.1.1 Binary Arithmetic with Correction

The decimal product can be obtained through a binary product and a post correction stage [4, 5]. Let A and B be two *BCD* digits $(a_3\ a_2\ a_1\ a_0)$ and $(b_3\ b_2\ b_1\ b_0)$ respectively. The *BCD* coded product consists of two *BCD* digits D and C such that:

$$A*B = D*10 + \mathrm{C} \tag{11.18}$$

A * B is first computed as a 7-bit binary number $P(6{:}0)$ such that

$$A*B = P = p_6\,p_5\,p_4\,p_3\,p_2\,p_1\,p_0 \tag{11.19}$$

Although a classic binary-to-*BCD* decoding algorithm can be used, it can be shown that the *BCD* code for P can be computed through binary sums of correcting terms described in Fig. 11.8. The first row in Figure shows the *BCD* weights. The weights of p_3, p_2, p_1 and p_0 are the same as those of the original binary number "$p_6\,p_5\,p_4\,p_3\,p_2\,p_1\,p_0$". But weights 16, 32 and 64 of p_4, p_5, and p_6 have been respectively decomposed as (10, 4, 2), (20, 10, 2) and (40, 20, 4). Observe that "p_3 p_2 p_1 p_0" could violate the interval [0, 9], then an additional adjust could be necessary.

First the additions of Row 1, 2, 3, and correction of "$p_3\,p_2\,p_1\,p_0$" are completed (least significant bit p_0 is not necessary in computations). Then the final correction is computed.

One defines (Arithmetic I):

1. the binary product $A*B = P = p_6\,p_5\,p_4\,p_3\,p_2\,p_1\,p_0$
2. the Boolean expression: $adj_1 = p_3 \wedge (p_2 \vee p_1)$,

3. the arithmetic sum: $dcp = p_6\,p_5\,p_4\,p_3\,p_2\,p_1\,p_0 + 0\,p_6\,p_5\,0\,p_4\,p_3 + 0\,0\,0\,0\,p_6\,p_5\,0 + 0\,0\,0\,0\,adj_i\;adj_1\,0$,

4. the Boolean expression: $adj_2 = (dcp_3 \wedge (dcp_2 \vee dcp_1)) \vee (p_5 \wedge p_4 \wedge p_3)$.

One computes

$$dc = dcp + 0\,0\,0\,0\,adj_2\,adj_2\,0. \tag{11.20}$$

Then

$$D = dc_7\,dc_6\,dc_5\,dc_4 \text{ and } C = dc_3\,dc_2\,dc_1\,dc_0$$

A better implementation can be achieved using the following relations (Arithmetic II):

1. the product $A*B = P = p_6\,p_5\,p_4\,p_3\,p_2\,p_1\,p_0$
2. compute:

$$cc = p_3\,p_2\,p_1\,p_0 + 0\,p_4\,p_4\,0 + 0\,p_6\,p_5\,0,$$
$$dd = p_6\,p_5\,p_4 + 0\;p_6\,p_5$$

(cc is 5 bits, dd has 4 bits, computed in parallel)

3. define:

$$cy1 = 1 \text{ iff } cc > 19,\; cy0 = 1 \text{ iff } 9 < cc < 20$$

($cy1$ y $cy2$ are function of $cc_3\;cc_2\;cc_1\;cc_0$, and can be computed in parallel)

4. compute:

$$c = cc_3\,cc_2\,cc_1\,cc_0 + cy1(cy1 \text{ or } cy0)\,cy\,0\,0, \tag{11.21}$$

$$d = dd_3\,dd_2\,dd_1\,dd_0 + 0\,0\,cy_1\,cy_0$$

(c and d calculated in parallel)

Compared with the first approach, the second one requires smaller adders (5 and 4-bit vs. 8-bit) and the adders can operate in parallel as well. The VHDL models *bcd_mul_arith1.vhd* and *bcd_mul_arith2.vhd* that described the previous method for digit by digit multiplication are available at the Authors' web page.

11.3.1.2 Using ROM

Actually a (100 × 8)-bit *ROM* can fit to store all the $A*B$ multiplications. However, as A and B are two 4-bit operands (BCD digits) the product can be mapped into a 2^8 × 8-bit *ROM*. In FPGA devices there mainly two main memory recourses: Block and distributed RAMS. For the examples in Xilinx devices two

main possibilities are considered: Block RAMs *BRAM*'s or distributed *RAM*'s (LUT based implementation of *RAM*'s).

***BRAM*-based implementation**: Xilinx Block *RAM*'s are 18-kbit (or 36-kbits) configurable and synchronous dual-port *RAM* (or *ROM*) blocks. They can be configured into different layouts. The $2^{11} \times 8$-bit configuration option has been selected, though wasting some memory capacity. As *BRAM* is dual-port, two one-by-one digit multiplications can be implemented in a single *BRAM* and in a single clock cycle. However the main characteristic to cope with is that *BRAM*'s are synchronous, so either the address or the output should be registered.

Distributed *RAM* (*LUT*-based): 6-input *LUT*'s can be configured as 64×1-bit RAM's. Moreover the four 6-LUTs in a slice has additional multiplexers to implement 256×1 bit RAMs. Then the $2^8 \times 8$-bit ROM can be implemented using 8 slices (32 *LUT*'s).

A trivial optimization reduces the area considering the computation of the *BCD* final result components c_0 and d_3 straightforward. Actually $c_0 = a_0 \wedge b_0$ and $d_3 = a_0 \wedge b_0 \wedge a_3 \wedge b_3$. That is, c_0 is related to the parity while d_3 emphasizes that most significant bit is set for one in only one case (decimal $9 \times 9 = 81$). It is thus possible to reduce the required memory size to a $2^8 \times 6$-bit *ROM* only plus two LUTs to implement c_0 and d_3.

Comments

1. *BRAM*-based design is fast but synchronous. It is useless for combinational implementations, but suitable for sequential and pipelined ones.
2. The existence of "do-not-care" conditions in the memory definition allows the synthesizer to reduce the effective memory requirement.

The VHDL model *bcd_mul_bram.vhd* implements the BRAM based implementation for the digit by digit multiplication. Additionally, and *bcd_mul_mem1.vhd* and *bcd_mul_mem2.vhd* provides the LUT based implementation of decimal BCD multiplication. These models with the corresponding test bench (*test_mul_1by1BCB.vhd*) are available at the Authors' web page.

11.3.2 N by One BCD Digit Multiplier

A $N \times 1$ *BCD* digit multiplier is readily achieved through N 1×1-digit multiplications followed by a *BCD* decimal addition. Fig. 11.9 shows how the partial products are arranged to feed the *BCD N*-digit adder stage. The carry-chain adder of Sect. 13.1 can be used.

The VHDL model *mult_Nx1_BCD.vhd* and *mult_Nx1_BCD_bram.vhd* that describe the N by one decimal multiplier are available at the Authors' web page.

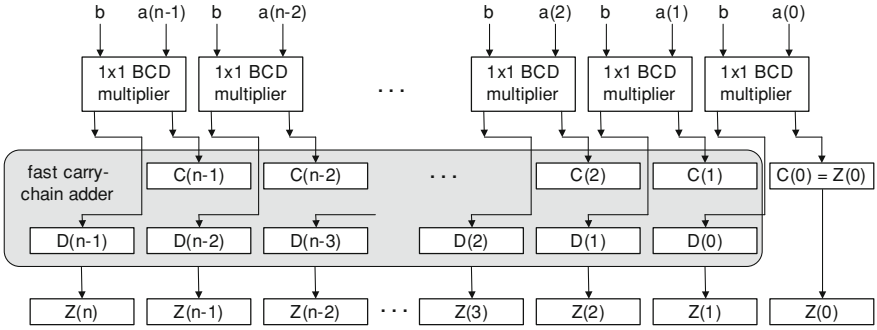

Fig. 11.9 N by one digit circuit multiplication. It includes *N* one by one digit multipliers and a fast carry-chain adder

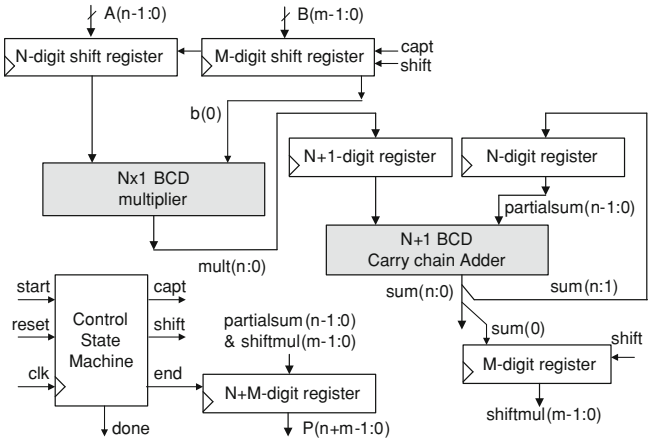

Fig. 11.10 A sequential $N \times M$ multiplier

11.3.3 N by M Digits Multiplier

Using the previously designed *N* by one digit multiplier it is possible to perform the $N \times M$ digits multiplication. The best area compromise is obtained using a sequential implementation that uses an $N \times 1$ digits multiplier and an $N + 1$ digit adder. Figure 11.10 show the scheme of the Least Significant Digit (LSD) first algorithm implemented. The *B* operand is shifted right at each clock cycle. The LSD digit of *B* is multiplied by the *A* operand and accumulated in the next cycle to shorten the critical path. After $M + 1$ cycles, the $N + M$ digit result is available.

The VHDL models *mult_BCD_seq.vhd* and *mult_BCD_bram_seq.vhd* that describes the *N* by *M* digits decimal multiplier and the matching test bench (*test_mult_BCB_seq.vhd*) is available at the Authors' web page.

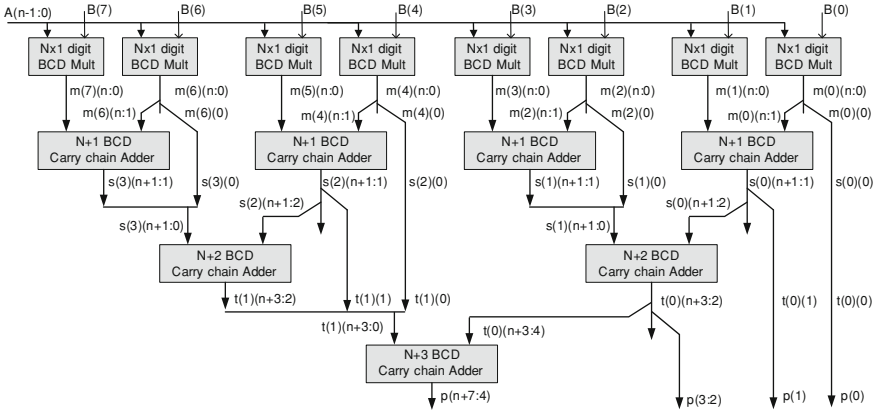

Fig. 11.11 An example of a paralell $N \times M$ multiplier ($N \times 8$)

Fully combinational implementations of $N \times M$ digits multipliers are possible also based on the $N \times 1$ digit multiplication cell. After the first multiplication, an adder tree sums up the intermediate result and gives the $N + M$ digit result. Figure 11.11 show an $N \times 8$ digit multiplication example.

The VHDL model *mult_BCD_comb.vhd* that describes the N by M digits decimal multiplier and the test bench (*test_mult_BCB_comb.vhd*) are available at the Authors' web page.

11.4 Decimal Division

As described in Chap. 9, the basic algorithms are based on digit recurrence. Using Eqs. (9.1), (9.2) and (9.3) at each step of (9.2) $q_{-(i\,+\,1)}$ and r_{i+1} are computed in function of r_i and y in such a way that $10 \cdot r_i = q_{-(i+1)}y + r_{i+1}$, with $-y \leq r_{i+1} < y$, that is

$$r_{i+1} = 10 \cdot r_i - q_{-(i+1)}y, \text{ with } -y \leq r_{i+1} < y. \tag{11.22}$$

The *Robertson diagram* applied to radix-10 ($B = 10$) is depicted at Fig. 11.12. It defines the set of possible solutions: the dotted lines define the domain $\{(10 \cdot r_i, r_{i+1}) | -10 \cdot y \leq 10 \cdot r_i < 10 \cdot y \text{ and } -y \leq r_{i+1} < y\}$, and the diagonals correspond to the equations $r_{i+1} = 10 \cdot r_i - ky$ with $k \in \{-10, -9, \ldots, -1, 0, 1, \ldots, 9, 10\}$. If $ky \leq 10 \cdot r_i < (k+1)y$, there are two possible solutions for $q_{-(i\,+\,1)}$, namely k and $k + 1$. To the first one corresponds a non-negative value of $r_{i\,+\,1}$, and to the second one a negative value.

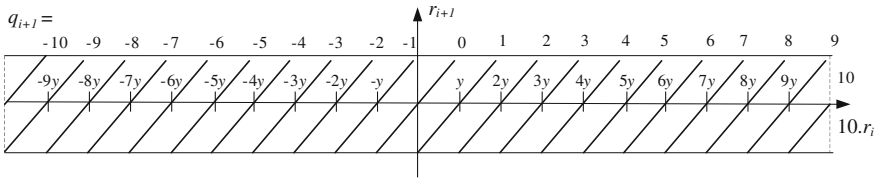

Fig. 11.12 Robertson diagram for radix 10

11.4.1 Non-Restoring Division Algorithm

A slightly modified version of the base-10 non-restoring algorithm (Sect. 9.2.1) is used. For that y must be a normalized n-digit natural, that is $10^{n-1} \le y < 10^n$. The remainders r_i satisfy the condition $-y \le r_i < y$ and belong to the range $-10^n < r_i < 10^n$. Define $w_i = 10 \cdot r_i$, so that

$$-10^{n+1} < w_i < 10^{n+1}. \tag{11.23}$$

Thus, w_i is an $(n + 2)$-digit 10's complement number. The selection of every digit $q_{-(i+1)}$ is based on a truncated value of w_i, namely $w' = \lfloor w_i / 10^{\alpha} \rfloor$, for some α that will be defined later, so that $w_i - 10^{\alpha} < w' \cdot 10^{\alpha} \le w_i$ and

$$w' \cdot 10^{\alpha} \le w_i < w' \cdot 10^{\alpha} + 10^{\alpha}. \tag{11.24}$$

According to (11.23) and (11.24), $-10^{n+1} - 10^{\alpha} < w' \cdot 10^{\alpha} < 10^{n+1}$, so that

$$-10^{n+1-\alpha} \le w' < 10^{n+1-\alpha}. \tag{11.25}$$

Thus, w' is an $(n + 2 - \alpha)$-digit 10's complement number. Assume that a set of integer-valued functions $m_k(y)$, for k in $\{-10, -9, \ldots, -1, 0, 1, \ldots, 8, 9\}$, satisfying

$$k \cdot y \le m_k(y) \cdot 10^{\alpha} < (k + 1) \cdot y - 10^{\alpha} \tag{11.26}$$

has been defined. The interval $[k \cdot y, (k + 1) \cdot y - 10^{\alpha}]$ must include a multiple of 10^{α}. Thus, y must be greater than or equal to $2 \cdot 10^{\alpha}$. Taking into account that $y \ge 10^{n-1}$, the condition is satisfied if $\alpha \le n - 2$.

The following property is a straightforward consequence of (11.24) and (11.26):

Property 11.1 If $m_k(y) \le w' < m_{k+1}(y)$, then $k \cdot y \le w_i < (k + 2) \cdot y$.

According to the Robertson diagram of Fig. 11.12, a solution $q_{-(i+1)}$ can be chosen as follows:

if $w' < m_{-9}(y)$ then $q_{-(i+1)} = -9$,
if $w' \ge m_8(y)$ then $q_{-(i+1)} = 9$,
if $m_k(y) \le w' < m_{k+1}(y)$ for some k in $\{-9, -8, \ldots, -1, 0, 1, \ldots, 7\}$, then $q_{-(i+1)} = k + 1$.

Thus, this non-restoring algorithm generates a p-digit decimal quotient $0.q_{-1}$ $q_{-2}\ldots q_{-p}$ and a remainder r_p satisfying

$$x = \left(0 \cdot q_{-1}q_{-2}\ldots q_{-p}\right)y + r_p \cdot 10^{-p}, \text{ with } -y \cdot 10^{-p} \leq r_p \cdot 10^{-p} < y \cdot 10^{-p}, \tag{11.27}$$

where every q_{-i} is a signed decimal digit. It can be converted to a decimal number by computing the difference between two decimal numbers

$$pos = q'_{-1} \cdot 10^{-1} + q'_{-2} \cdot 10^{-2} + \cdots + q'_{-p} \cdot 10^{-p} \text{ and}$$
$$neg = q''_{-1} \cdot 10^{-1} + q''_{-2} \cdot 10^{-2} + \cdots + q''_{-p} \cdot 10^{-p},$$
$$\text{with } q'_{-i} = q_i \text{ if } q_i > 0, q'_{-i} = 0 \text{ if } q_i < 0,\ q''_{-1} = q_i \text{ if } q_i < 0, q''_{-i} = 0 \text{ if } q_i > 0.$$

It remains to define a set of integer-valued functions $m_k(y)$ satisfying (11.26). In order to simplify their computation, they should only depend on the most significant digits of y. The following definition satisfies (11.26):

$$m_k(y) = \lfloor k \cdot y'/10 \rfloor + bias \text{ where } y' = \left\lfloor y/10^{\alpha-1} \right\rfloor \text{ if } k \geq 0 \text{ and}$$
$$m_k(y) = -\lfloor -k \cdot y'/10 \rfloor + bias \text{ if } k < 0,$$

where *bias* is any natural belonging to the range $2 \leq bias \leq 6$. With $\alpha = n\text{-}2$, y' as a 3-digit natural, and w' and $m_k(y)$ are 4-digit 10's complement numbers. In the following Algorithm 1 $m_k(y)$ is computed without adding up *bias* to $\lfloor k \cdot y'/10 \rfloor$or $-\lfloor -k \cdot y'/10 \rfloor$, and w' is substituted by w'—*bias*.

Algorithm 11.3: Non-restoring algorithm for decimal numbers

```
yy := y / (10**(n-3));
plus_0_y := 0;  plus_1_y := yy / 10; plus_2_y := 2*yy / 10;
...; plus_9_y := 9*yy  / 10;
r := x; power := 10**(p-1);
quotient_pos := 0; quotient_neg := 0;
for i in 0 .. p-1 loop
  ww := (10*r)/(10**(n-2));
  ww := ww - bias;
  if ww >= 0 then
    sum1 := ww - plus_1_y;
    ...
    sum8 := ww - plus_8_y;
    if sum1 < 0 then q := 1;
    elsif sum2 < 0 then q := 2;
    ...
    elsif sum8 < 0 then q := 8;
    else q := 9;
    end if;
```

```
    q_y := q*y;
    quotient_pos := quotient_pos + q*power;
  else --ww < 0
    sum1 := ww + plus_1_y;
    ...
    sum9 := ww + plus_9_y;
    if sum1 >= 0 then q := 0;
    elsif sum2 >= 0 then q := 1;
    ...
    elsif sum9 >= 0 then q := 8;
    else q := 9;
    end if;
    q_y := -q*y;
    quotient_neg := quotient_neg + q*power;
  end if;
  r := 10*r - q_y;
  power := power / 10;
end loop;
quotient := quotient_pos - quotient_neg;
```

The structure of the data path corresponding to Algorithm 11.3 is shown in Fig. 11.13. Additionally, two decimal shift registers storing *pos* and *neg* and an output subtractor are necessary. Alternatively the conversion could be done *on-the fly*. The computation time and complexity of the *k_y's generation* and *range detection* components are independent of n. The execution time of the iteration step is determined by the n-digit by 1-digit multiplier and by the $(n + 1)$-digit adder. The total computation time is $O(p{\cdot}n)$.

The simplified VHDL model *decimal_divider_nr.vhd* that describes the N by N digits divider that produces P digits of decimal result and the test bench (*test_dec_div_seq.vhd*) are available at the Authors' web page.

11.4.2 An SRT-Like Division Algorithm

In order to make the computation time linear, a classical method consists of using carry-save adders and multipliers instead of ripple-carry ones. In this way, the iteration step execution time can be made independent of the number n of digits. Nevertheless, this poses the following problem: a solution $q_{-(i+1)}$ of Eq. (9.2) must be determined in function of a carry-stored representation (s_i, c_i) of r_i, without actually computing $r_i = s_i + c_i$. An algorithm similar to the SRT-algorithms for radix-2^K dividers can be defined for decimal numbers (Sect. 9.2.3). Once again the divisor y must be assumed to be a normalized n-digit natural.

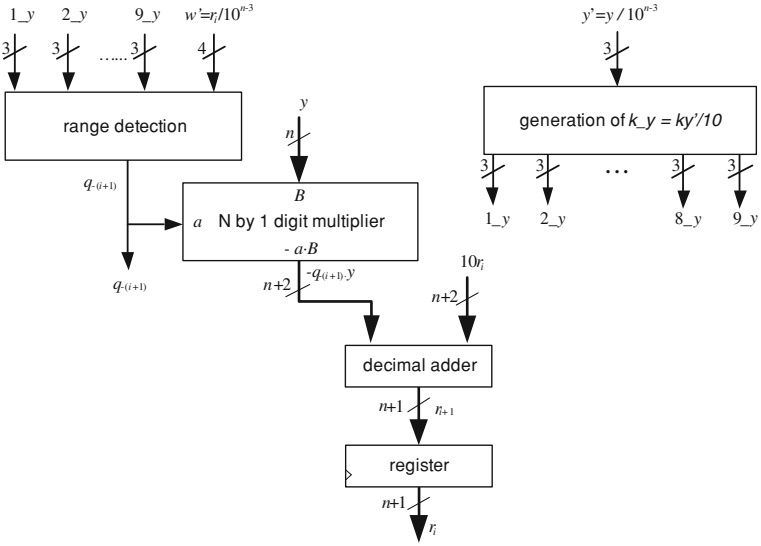

Fig. 11.13 Non-restoring radix 10 algorithm data path

All along the algorithm execution, r_i will be encoded in the form $w_i = s_i + c_i$ (stored-carry encoding) where s_i and c_i are 10's-complement numbers. Define w_i and w' as in previous section, that is $w_i = 10 \cdot r_i$, and $w' = \lfloor w_i/10^{\alpha} \rfloor$. Thus, $w_i = 10 \cdot s_i + 10 \cdot c_i$ and $w' = \lfloor (10 \cdot s_i + 10 \cdot c_i)/10^{\alpha} \rfloor$. Define also truncated values s_t and c_t of $10{\cdot}s_i$ and $10{\cdot}c_i$, that is $s_t = \lfloor 10 \cdot s_i/10^{\alpha} \rfloor$ and $c_t = \lfloor 10 \cdot c_i/10^{\alpha} \rfloor$, and let w'' be the result of adding s_t and c_t. The difference between $w' = \lfloor (10 \cdot s_i + 10 \cdot c_i)/10^{\alpha} \rfloor$ and $w'' = \lfloor 10 \cdot s_i/10^{\alpha} \rfloor + \lfloor 10 \cdot c_i/10^{\alpha} \rfloor$ is the possible carry from the rightmost positions, so that $w' - 1 \le w'' \le w'$, and thus

$$w'' \le w' \le w'' + 1. \tag{11.28}$$

According to (11.25) and (11.28),

$$-10^{n+1-\alpha} - 1 \le w'' < 10^{n+1-\alpha}, \tag{11.29}$$

so that w'' is an $(n + 3-\alpha)$-digit 10's-complement number. The relation between w_i and the estimate $w''{\cdot}10^{\alpha}$ of w_i is deduced from (11.24) and (11.28):

$$w'' \cdot 10^{\alpha} \le w_i < w'' \cdot 10^{\alpha} + 2 \cdot 10^{\alpha}. \tag{11.30}$$

Assume that a set of integer-valued functions $M_k(y)$, for k in $\{-10, -9, \ldots, -1, 0, 1, \ldots, 8, 9\}$, satisfying

$$k \cdot y \le M_k(y) \cdot 10^{\alpha} < (k+1) \cdot y - 2 \cdot 10^{\alpha}, \tag{11.31}$$

have been defined. The interval $[k{\cdot}y, (k + 1){\cdot}y - 2{\cdot}10^{\alpha}[$ must include a multiple of 10^{α}. Thus y must be greater than or equal to $3{\cdot}10^{\alpha}$. Taking into account that

$y \geq 10^{n-1}$, once again the condition is satisfied if $\alpha \leq n-2$. The following property is a straightforward consequence of (11.30) and (11.31).

Property If $M_k(y) \leq w'' < M_{k+1}(y)$, then $k \cdot y \leq w_i < (k+2) \cdot y$.

Thus, according to the Robertson diagram of Fig. 11.12, a solution $q_{-(i+1)}$ can be chosen as follows:

if $w'' < M_{-9}(y)$ then $q_{-(i+1)} = -9$,
if $w'' \geq M_8(y)$ then $q_{-(i+1)} = 9$,
if $M_k(y) \leq w'' < M_{k+1}(y)$ for some k in $\{-9, -8, \ldots, -1, 0, 1, \ldots, 7\}$, then $q_{-(i+1)} = k+1$.

This SRT-like algorithm generates a p-digit decimal quotient $0.q_{-1}\, q_{-2} \ldots q_{-p}$ and a remainder r_p satisfying (11.27), and can be converted to a decimal number by computing the difference between two decimal numbers as in non-restoring algorithm.

It remains to define a set of integer-valued functions $M_k(y)$ satisfying (11.31). In order to simplify their computation, they should only depend on the most significant digits of y. Actually, the same definition as in Sect. 11.3.1 can be used, that is

$$M_k(y) = \lfloor k \cdot y'/10 \rfloor + bias \text{ where } y' = \lfloor y/10^{\alpha-1} \rfloor \text{ if } k \geq 0 \text{ and}$$
$$M_k(y) = -\lfloor -k \cdot y'/10 \rfloor + bias \text{ if } k < 0.$$

In this case the range of *bias* is $3 \leq bias \leq 6$. With $\alpha = n-2$, y' as a 3-digit natural, w' and $m_k(y)$ are 4-digit 10's complement numbers, and w'' is a 5-digit 10's complement number. In the following Algorithm 2 $M_k(y)$ is computed without adding up *bias* to $\lfloor k \cdot y'/10 \rfloor$ or $-\lfloor -k \cdot y'/10 \rfloor$, and w'' is substituted by w''—*bias*.

Algorithm 11.4: SRT like algorithm for decimal numbers

```
--computation of k_y for k in 0 to 9:
yy := y / (10**(n-3));
plus_0_y := 0; plus_1_y := yy/10; ...; plus_9_y := 9*yy/10;
--digit-recurrence algorithm:
s := x; c := 0; quotient_pos := 0; quotient_neg := 0;
power := 10**(p-1);
for i in 0 .. p-1 loop
  --estimate of w:
  ww := (10*s)/(10**(n-2)) + (10*c)/(10**(n-2));
  --detection of the range of ww:
  ww := ww - bias;
  if ww >= 0 then
    sum1 := ww - plus_1_y;
    sum2 := ww - plus_2_y;
    ...
```

```
    sum8 := ww - plus_8_y;
    if sum1 < 0 then q := 1; q_y := - y;
    elsif sum2 < 0 then q := 2; q_y := - 2*y;
    ...
    elsif sum8 < 0 then q := 8; q_y := - 8*y;
    else q := 9; q_y := - 9*y;
    end if;
    quotient_pos := quotient_pos + q*power;
  else
    sum1 := ww + plus_1_y;
    sum2 := ww + plus_2_y;
    ...
    sum9 := ww + plus_9_y;
    if sum1 >= 0 then q := 0; q_y := 0;
    elsif sum2 >= 0 then q := 1; q_y := y;
    ...
    elsif sum9 >= 0 then q := 8; q_y := 8*y;
    else q := 9; q_y := 9*y;
    end if;
    quotient_neg := quotient_neg + q*power;
  end if;
  --carry-save addition of 10s, 10c and -qy:
  csa (10*s, q_y, 10*c, next_s, next_c);
  s := next_s; c := next_c;
  power := power / 10;
end loop;
quotient := quotient_pos - quotient_neg;
--stored-carry decoding:
r := s + c;
```

The structure of the data path corresponding to Algorithm 11.4 is shown in Fig. 11.4. The carry-free multiplier is a set of 1-digit by 1-digit multipliers working in parallel. Each of them generates two digits $p_{1,\,j\,+\,1}$ and $p_{0,\,j}$ such that $q_{-(i+1)} \cdot y_j = 10 \cdot p_{1,j+1} + p_{0,j}$, and the product $q_{-(i+1)} \cdot y$ is obtained under the form $p_1 + p_0$. The 4-to-2 counter computes $10{\cdot}s_i + 10{\cdot}c_i - (p_1 + p_0)$, that is $10{\cdot}r_i - q_{-(i\,+\,1)}{\cdot}y$, under the form $s_{i\,+\,1} + c_{i\,+\,1}$. Two decimal shift registers storing *pos* and *neg* and an output ripple-carry subtractor are necessary. Another output ripple-carry adder is necessary for computing the remainder $r_p = s_p + c_p$. Thus, all the components, but the output ripple-carry components, have computation times independent of n and p. The total computation time is $O(p + n)$.

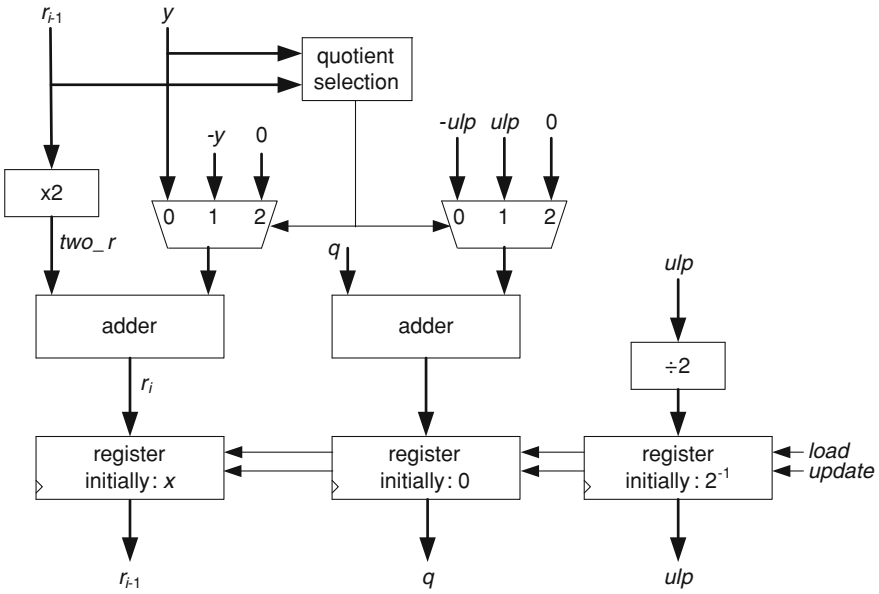

Fig. 11.14 Binary digit-recurrence data path for decimal division

The VHDL model *decimal_divider_SRT_like.vhd* that describes the *N* by *N* digits divider that produces *P* digits decimal result with several modules (*decimal_shift_register.vhd, mult_Nx1_BCD_carrysave.vhd, special_5digit_adder.vhd, range_detection3.vhd, bcd_csa_addsub_4to2.vhd*) and the test bench (*test_dec_div_seq.vhd*) are available at the Authors' web page.

11.4.3 Other Methods for Decimal Division

Other methods of division could be considered such as the use digit recoding in dividend and or divisor and also use extra degree of pre-normalization for the operands. The idea behind these methods is to ease the digit selection process.

Another idea is the use of the binary digit recurrence algorithm of describe in Chap. 9 but using decimal operands.

Observe that the Algorithm 9.1 can be executed whatever the representation of the numbers. If *B*'s complement radix-*B* representation is used, then the following operations must be available: radix-*B* doubling, adding, subtracting and halving.

Algorithm 11.5: Binary digit-recurrence algorithm for decimal division

```
r0 := x; q := 0; ulp := 2^-1;
for i in 1 .. p loop
  two_r := doubling(ri-1);
  sum := adding(two_r, y);
  dif := subtracting(two_r, y);
  if quotient_selection(ri-1, y) = 1 then
    ri := dif;
    q := adding(q, ulp);
  elsif quotient_selection(ri-1, y) = -1 then
    ri := sum;
    q := subtracting(q, ulp);
  else
    ri := two_r;
  end if;
  ulp := halving(ulp);
end loop ;
r := rp;
```

In order to obtain m decimal digits results, you need to choose p so that $2^{-p} \cong 10^{-m}$, that is $p \cong m \cdot log_2 10 \cong 3.3 \cdot m$. A possible data path is shown in Fig. 11.14. The implementation of this architecture leads to a smaller circuit, but slower than the decimal digit recurrence due to the difference in the amount of cycles to be executed.

11.5 FPGA Implementation Results

The circuits have been implemented on Xilinx Virtex-5 family with speed grade -2 [6]. The Synthesis and implementation have been carried out on *XST* (Xilinx Synthesis Technology) [7] and Xilinx Integrated System environment (*ISE*) version 13.1 [2]. The critical parts were designed using low level components instantiation (lut6_2, muxcy, xorcy, etc.) in order to obtain the desired behavior.

11.5.1 Adder-Subtractor Implementations

The adder and adder-subtractor implementation results are presented in this section. Performances of different N-digit *BCD* adders have been compared to those of an M-bit binary carry chain adder (implemented by *XST*) covering the same range, i.e. such that $M = \lfloor N.log_2(10) \rfloor \cong 3.322N$.

Table 11.1 Delays in *ns* for decimal and binary adders and adder-subtractor

N (digits)	RpCy add	CyCh add	AddSub V1	AddSub V2	M (bits)	Binary add-sub
8	12.4	3.5	3.5	3.4	27	2.1
16	24.4	3.8	3.8	3.7	54	2.6
32	48.5	4.5	4.6	4.8	107	3.8
48	72.3	5.1	5.2	5.3	160	5.2
64	95.9	5.2	5.5	5.5	213	6.6
96	–	5.9	6.1	6.1	319	8.8

Table 11.2 Area in 6-input *LUT*s for different decimal adders and adders-subtractors

Circuit	# LUTs
Ripple carry adder	$7 \times N$
Carry chain adder	$9 \times N$
Binary	$\lceil 3.32 \times N \rceil$
Adder-subtractor V2 (PG from binary addition)	$9 \times N$
Adder-subtractor V2 (PG direct form inputs)	$13 \times N$
Binary adder and adder-subtractor	$\lceil 3.32 \times N \rceil$

Table 11.1 exhibits the post placement and routing delays in *ns* for the decimal adder implementations *Ad*-I and *Ad*-II of Sect. 7.1; and the delays in *ns* for the decimal adder-subtractor implementations *AS*-I and *AS*-II of Sect. 7.2. Table 11.2 lists the consumed areas expressed in terms of 6-input look-up tables (6-input *LUT*s). The estimated area presented in Table 11.2 was empirically confirmed.

Comments

1. Observe that for large operands, the decimal operations are faster than the binary ones.
2. The delay for the carry-chain adder and the adder-subtractor are similar in theory. The small difference is due to the placement and routing algorithm.
3. The overall area with respect to binary computation is not negligible. In Xilinx 6-input LUT family an adder-subtractor is between 3 and 4 times bigger.

11.5.2 Multiplier Implementations

The decimal multipliers use the one by one digit multiplication described in Sect. 11.2.1 and the decimal adders of Sect. 11.1. The results are for the same Virtex 5 device speed grade −2.

Table 11.3 Results of BCD N × 1 multipliers using LUTs cells and BRAM cells

N	Mem in LUTs cells		BRAM-based cells		
	T (ns)	# LUT	T (ns)	# LUT	# BRAM
4	5.0	118	5.0	41	1
8	5.1	242	5.1	81	2
16	5.3	490	5.4	169	4
32	6.1	986	6.1	345	8

Table 11.4 Results of sequential implementations of $N \times M$ multipliers using one by one digit multiplication in LUTs

N	M	T (ns)	# FF	# LUT	# cycles	Delay (ns)
4	4	5.0	122	243	4	25.0
8	4	5.1	186	451	5	25.5
8	8	5.1	235	484	9	45.9
8	16	5.1	332	553	17	86.7
16	8	5.3	363	921	9	47.7
16	16	5.3	460	986	17	90.1
32	16	5.7	716	1,764	17	96.9
16	32	5.3	653	1,120	33	174.9
32	32	5.7	909	1,894	33	188.1

11.5.2.1 Decimal N × 1 Digits Implementation Results

Comparative figures of merit (minimum clock period (T) in ns, and area in LUTs and BRAMS) of the $N \times 1$ multiplier are shown in Table 11.3, for several values of N; the result are shown for circuits using LUTs, and BRAMS to implement the digit by digit multiplication respectively.

11.5.2.2 Sequential Implementations

The sequential circuit multiplies N digits by 1 digit per clock cycle, giving the result $M + 1$ cycles later. In order to speed up the computation, the addition of partial results is processed in parallel, but one cycle later (Fig. 11.10). Results for sequential implementation using LUT based cells for 1×1 *BCD* digit multiplication are given in Table 11.4. If the *BRAM*-based cell is used, similar periods (T) can be achieved, by means of less LUTs but using BRAMs blocks.

11.5.2.3 Combinational Implementations of N by M Multipliers

For the implementation of the $N \times M$-digit multiplier, the $N \times 1$ *mux*-based multiplication stage has been replicated M times: it is the best choice because *BRAM*-

Table 11.5 Results of combinational implementations of $N \times M$ multipliers

N	M	Delay (ns)	# LUT
4	4	10.2	719
8	4	10.7	1,368
8	8	13.4	2,911
8	16	15.7	6,020
16	8	13.6	5,924
16	16	16.3	12,165

based multipliers are synchronous. Partial products are inputs to an addition tree. For all *BCD* additions the fast carry-chain adder of Sect. 11.1.4 has been used.

Input and output registers have been included in the design. Delays include *FF* propagation and connections. The amount of *FF*'s actually used is greater than $8_*(M + N)$ because the ISE tools [2] replicate the input register in order to reduce fan-outs. The most useful data for area evaluation is the number of *LUT*'s (Table 11.5).

Comments

1. Observe that computation time and the required area are different for N by M than M by N. That is due mainly by the use of carry logic in the adder tree.

11.5.3 Decimal Division Implementations

The algorithms have been implemented in the same Xilinx Virtex-5 device as previous circuits in the chapter. The adders and multipliers used in division are the ones described in previous sections. The non-restoring like division circuit correspond to Algorithm 11.4 (Fig. 11.13), meanwhile the SRT-like to Algorithm 11.5 (Fig. 11.15). In tables the number of decimal digits of dividend and divider is expressed as *N* and the number of digits of quotient as *P*, meanwhile the period and latency in *ns* (Table 11.6, 11.7).

Comments

1. Both types of dividers have approximately the same costs and similar delay. They are also faster than equivalent binary dividers. As an example, the computation time of an SRT-like 48-digit divider (n = p = 48) is about 50·10.9 = 545 ns, while the computation time of an equivalent binary non-restoring divider, that is a 160-bit one (48/log102 ≅ 160), is more than 900 ns.
2. On the other hand, the area of the 48-digit divider (4,607 LUTs) is about five times greater than that of the 160-bit binary divider (970 LUTs). Why so a great difference? On the one hand it has been observed that decimal computation resources (adders and subtractors) need about three times more slices than binary resources (Sects. 11.4.1 and 11.4.2), mainly due to the more complex definition of the carry propagate and carry generate functions, and to the final

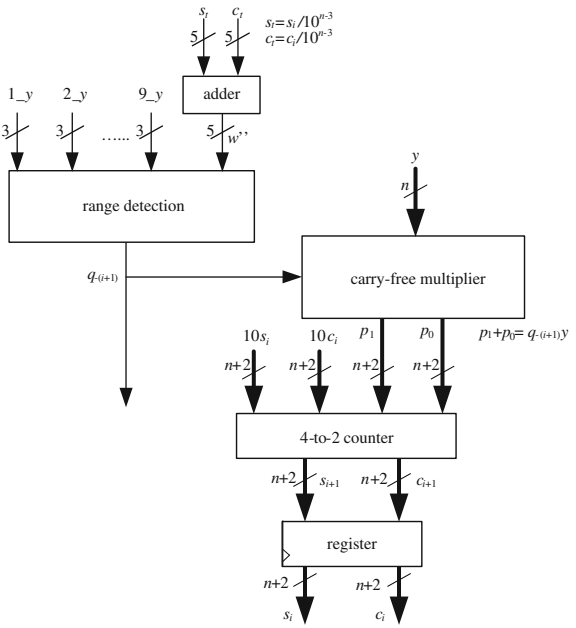

Fig. 11.15 SRT-like radix-10 algorithm data path

Table 11.6 Result for non-restoring division algorithm

N, P	FF	LUTS	Period	Latency
8	106	1,082	11.0	110.0
16	203	1,589	11.3	203.4
32	396	2,543	11.6	394.4
48	589	3,552	12.1	605.0

Table 11.7 Result for SRT-like division algorithm

N, P	FF	LUTS	Period	Latency
8	233	1,445	10.9	109.0
16	345	2,203	10.9	196.2
32	571	3,475	10.9	370.6
48	795	4,627	10.9	545.0

mod 10 reduction. On the other hand, the computation of the next quotient digit is much more complex than in the binary case.

11.6 Exercises

1. Implement a decimal comparator. Based on the decimal adder subtractor architecture modify it to implement a comparator.
2. Implement a decimal "greater than" circuit that returns '1' if A $\geq$ B else '0'. Tip: Base your design on the decimal adder subtractor.
3. Implement a $N \times 2$ digits circuit. In order to speed up computation analyze the use of a 4 to 2 decimal reducer.
4. Implement a $N \times 4$ digits circuit using a 8 to 2 decimal reducer and only one carry save adder.
5. Design a N by M digit multiplier using the $N \times 2$ or the $N \times 4$ digit multiplier. Do you improve the multiplication time? What is the area penalty with respect to the use of a $N \times 1$ multiplier?
6. Implement the binary digit-recurrence algorithm for decimal division (Algorithm 11.5). The key point is an efficient implementation of radix-B doubling, adding, subtracting and halving.

References

1. Altera Corp (2011) Advanced synthesis cookbook. http://www.altera.com
2. Xilinx Inc (2011b) ISE Design Suite Software Manuals (v 13.1). http://www.xilinx.com
3. Bioul G, Vazquez M, Deschamps J-P, Sutter G (2010) High speed FPGA 10's complement adders-subtractors. Int J Reconfigurable Comput 2010:14, Article ID 219764
4. Jaberipur G, Kaivani A (2007) Binary-coded decimal digit multiplier. IET Comput Digit Tech 1(4):377–381
5. Sutter G, Todorovich E, Bioul G, Vazquez M, Deschamps J-P (2009) FPGA implementations of BCD multipliers. V International Conference on ReConFigurable Computing and FPGAs (ReConFig 09), Mexico
6. Xilinx Inc (2010) Virtex-5 FPGA Data Sheet, DS202 (v5.3). http://www.xilinx.com
7. Xilinx Inc (2011) XST User Guide UG687 (v 13.1). http://www.xilinx.com