Table 2.4. IEEE floating-point standard.

	Single	Double
Word length	32	64
Mantissa	23	52
Exponent	8	11
Bias	127	1023
Range	$2^{138} \approx 3.8 \cdot 10^{38}$	$2^{1024} \approx 9 \cdot 10^{307}$

normalized so that both numbers are adjusted to the larger exponent. Then the two mantissas can be added. For both multiplication and addition a final normalization is necessary to achieve again the fractional $1.m$ representation for the mantissa.

The single IEEE floating-point format uses FPGA resources intensively due to the 23×23-bit fast integer multiplier. Therefore Shirazi et al. [42] have developed a modified format to implement various algorithms on their custom computing machine called SPLASH-2, a multiple-FPGA board based on Xilinx XC4010 devices (see Table 1.5, p. 16). They used an 18-bit format so that they can transport two operands over the 36-bit wide system bus of the multiple-FPGA board. The 18-bit format has a 10-bit mantissa, 7-bit exponent and a sign bit, and can represents a range of $3.7 \cdot 10^{19}$. They implemented an adder and a multiplier with three pipeline stages. The data for a single floating-point multiplier and adder [42] can be seen from Table 2.5.

Table 2.5. Custom 18-bit floating-point design using Xilinx XC4010.

	Adder	Multiplier
Function Generator (i.e., LE)	224	352
Flip-flops	112	112
Speed	8.6 MHz	4.9 MHz

2.3 Binary Adders

A basic binary N-bit adder/subtractor consists of N full adders (FA). A full adder implements the following Boolean equations

$$s_k = x_k \quad \text{XOR} \quad y_k \quad \text{XOR} \quad c_k \tag{2.23}$$
$$= x_k \oplus y_k \oplus c_k \tag{2.24}$$

that define the sum-bit. The carry (out) bit is computed with:

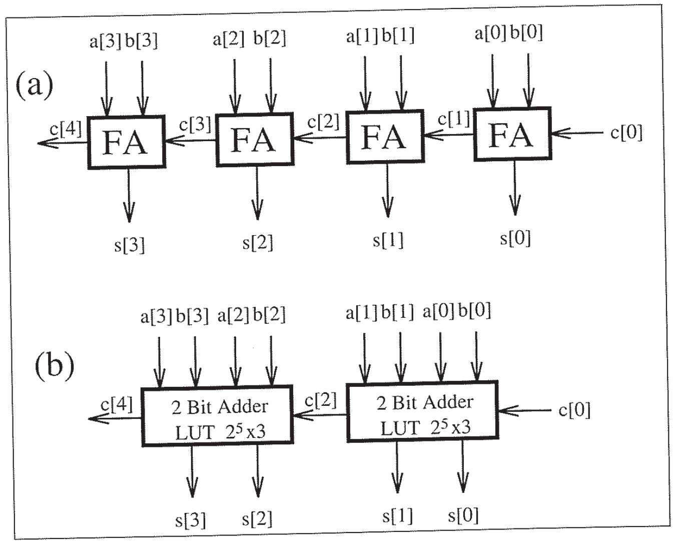

Fig. 2.6. Two's complement adders.

$$c_{k+1} = (x_k \text{ AND } y_k) \text{ OR } (x_k \text{ AND } c_k) \text{ OR } (y_k \text{ AND } c_k) \tag{2.25}$$
$$= (x_k \cdot y_k) + (x_k \cdot c_k) + (y_k \cdot c_k) \tag{2.26}$$

In the case of a 2C adder, the LSB can be reduced to a half adder because the carry input is zero.

The simplest adder structure is called the "ripple carry adder" as shown in Fig. 2.6(a) in a bit-serial form. If larger tables are available in the FPGA, several bits can be grouped together into one LUT, as shown in Fig. 2.6(b). For this "two bit at a time" adder the longest delay comes from the ripple of the carry through all stages. Attempts have been made to reduce the carry delays using techniques such as the carry-skip, carry lookahead, conditional sum, or carry-select adders. These techniques can speed up addition and can be used with older generation FPGA families (e.g., XC 3000 from Xilinx) since these devices do not provide internal fast carry logic. Modern families, such as the Xilinx XC4K or Altera Flex, posses very fast "ripple carry logic" that is about a magnitude faster than the delay through a regular logic LUT [1]. Altera uses fast tables (see Fig. 1.12, p. 17), while the Xilinx XC4K uses hardwired decoders for implementing carry logic based on the multiplexer structure shown in Fig. 2.7. The presence of the fast carry logic in modern FPGA families removes the need to develop hardware intensive carry look-ahead schemes.

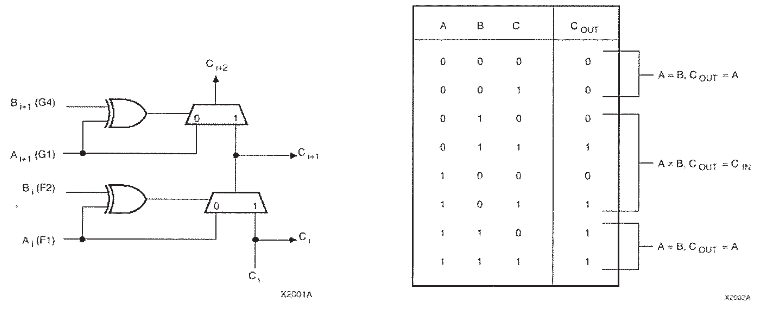

Fig. 2.7. XC4000 fast carry logic (©1993 Xilinx).

Fig. 2.8 summarizes the size and `Registered Performance` of N-bit binary adders, if implemented with the `lpm_add_sub` megafunction component. If the operands are applied through I/O cells, the delays through the busses of a FLEX device are dominant and performance decreases if the registers of the I/O cells are used (Option: `Assign`→ `Global Project Logic Synthesis`→`Automatic Fast I/O`). If the data are routed from local registers, performance improves. This type of additional LC register allocation will appear (in the project report file) as increased LC use by a factor of three. A synchronous registered design would not consume any additional resources. A typical design will achieve a speed between these two cases.

2.3.1 Pipelined Adders

Pipelining is extensively used in DSP solutions due the intrinsic data-flow regularity of DSP algorithms. Programmable digital signal processor MACs [14, 15, 6] typically carry at least four pipelined stages. The processor:

1) Decodes the command
2) Loads the operands in registers
3) Performs multiplication and stores the product, and
4) Accumulates the products, all concurrently.

The pipelining principle can be applied to FPGA designs as well, at little or no additional cost since each logic element contains a flip-flop, which is otherwise unused, to save routing resources. With pipelining it is possible to break an arithmetic operation into small primitive operations, save the carry and the intermediate values in registers, and continue the calculation in the next clock cycle. Such adders are sometimes called carry save adders (CSAs) [1] in the literature. Then the question arises: In how many pieces should we

[1] The name carry save adder is also used in the context of Wallace-multiplier, see Exercise 2.1, p. 75.

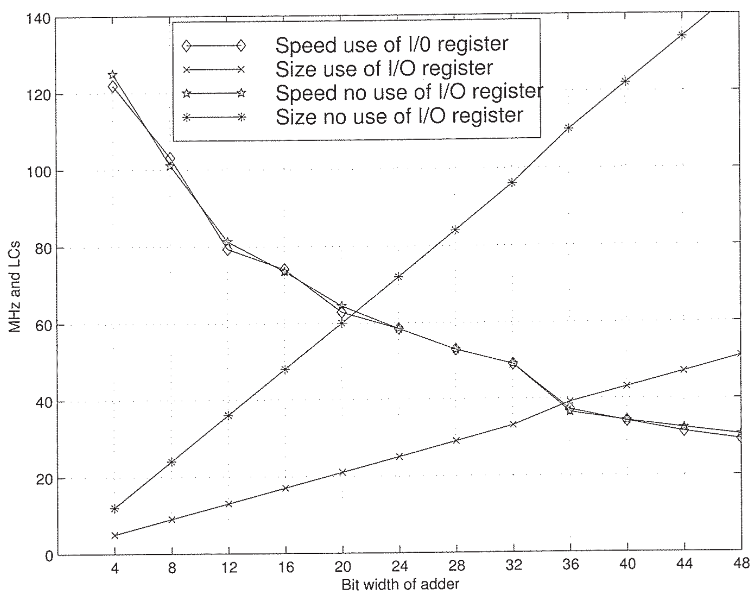

Fig. 2.8. Adder speed and size for Flex 10K.

divide the adder? Should we use bit level? For Altera's Flex 10K devices a reasonable choice will be always using an LAB with 8 LCs and 8 FFs for one pipeline element. In fact it can be shown that if we try to pipeline (for instance) a 5-bit adder, the performance drops, as reported in Table 2.6, because the pipelined 5-bit adder does not fit in one LAB.

Table 2.6. Performance of a 5-bit pipelined adder using synthesis of predefined LPM modules with pipeline option.

Pipeline-stages	Use I/O register		No use of I/O register	
	MHz	LCs	MHz	LCs
0	123.45	6	120.48	15
1	121.95	10	123.45	18
2	112.35	14	123.45	24
3	112.35	20	120.48	32
4	112.35	28	123.45	41
5	103.91	31	121.95	45

Table 2.7. Performance of pipelined adders. Size and speed are for the maximum bit width, for 15-, 22-, and 29-bit adders.

Bit width	Use I/O register MHz	Use I/O register LCs	No use of I/O register MHz	No use of I/O register LCs	Pipeline stages	Design file name
9-15	97.08	26	94.33	61	1	`add_1p.vhd`
16-22	94.33	58	91.74	113	2	`add_2p.vhd`
23-29	91.74	105	90.90	180	3	`add_3p.vhd`

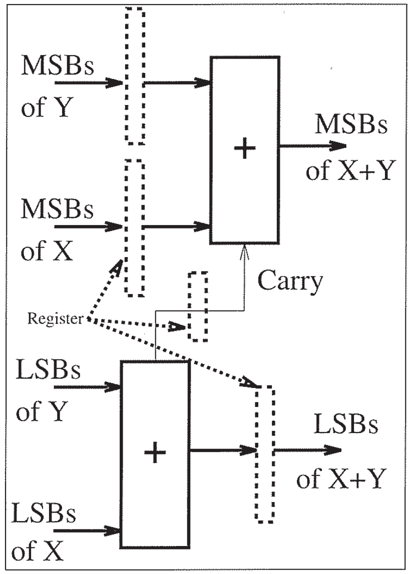

Fig. 2.9. Pipelined adder.

Because the number of flip-flops in one LAB is 8 and we need an extra flip-flop for the carry-out, we should use a maximum block size of 7 bits for maximum `Registered Performance`. Only the blocks with the MSBs can be 8 bits wide, because we do not need the extra flip-flop for the carry. This observation leads to the following conclusions:

1) With one additional pipeline stage we can build adders up to a length $7+8=15$.
2) With two pipeline stages we can build adders with up to $7+7+8=22$-bit length.

3) With three pipeline stages we can build adders with up to $7+7+7+8=29$-bit length.

Table 2.7 shows the `Registered Performance` and LC utilization of this kind of pipelined adder. From Table 2.7 it can be concluded that although the bit width increases the `Registered Performance` remains almost the same if we add the appropriate number of pipeline stages.

The following example shows the code of a 15-bit pipelined adder which is graphically interpreted by Fig. 2.9.

Example 2.14: VHDL Design of 15-bit Pipelined Adder

Consider the VHDL code[2] of a 15-bit pipelined adder that is graphically interpreted in Fig. 2.9. Depending on the synthesis style, the design runs at 90 to 104 MHz.

```
LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY add_1p IS
  GENERIC (WIDTH  : INTEGER := 15; -- Total bit width
           WIDTH1 : INTEGER := 7;  -- Bit width of LSBs
           WIDTH2 : INTEGER := 8;  -- Bit width of MSBs
           ONE    : INTEGER := 1); -- 1 bit for carry reg.
  PORT (x,y : IN  STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
                                                 -- Inputs
        sum : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
                                                 -- Result
        clk : IN  STD_LOGIC);
END add_1p;

ARCHITECTURE flex OF add_1p IS
  SIGNAL l1, l2, r1, q1                -- LSBs of inputs
                       : STD_LOGIC_VECTOR(WIDTH1-1 DOWNTO 0);
  SIGNAL l3, l4, r2, q2, u2, h2        -- MSBs of inputs
                       : STD_LOGIC_VECTOR(WIDTH2-1 DOWNTO 0);
  SIGNAL s             : STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
                                             -- Output register
  SIGNAL cr1, cq1      : STD_LOGIC_VECTOR(ONE-1 DOWNTO 0);
                                           -- LSBs carry signal


BEGIN
  PROCESS  -- Split in MSBs and LSBs and store in registers
  BEGIN
    WAIT UNTIL clk = '1';
    -- Split LSBs from input x,y
    FOR k IN WIDTH1-1 DOWNTO 0 LOOP
```

[2] The equivalent Verilog code `add_1p.v` for this example can be found in Appendix A on page 345.

```
      l1(k) <= x(k);
      l2(k) <= y(k);
    END LOOP;
    -- Split MSBs from input x,y
    FOR k IN WIDTH2-1 DOWNTO 0 LOOP
      l3(k) <= x(k+WIDTH1);
      l4(k) <= y(k+WIDTH1);
    END LOOP;
  END PROCESS;
-------------- First stage of the adder ------------------
  add_1: lpm_add_sub                      -- Add LSBs of x and y
         GENERIC MAP ( LPM_WIDTH => WIDTH1,
                       LPM_REPRESENTATION => "UNSIGNED",
                       LPM_DIRECTION => "ADD")
         PORT MAP ( dataa => l1, datab => l2,
                    result => r1,  cout => cr1(0));
  reg_1: lpm_ff            -- Save LSBs of x+y and carry
         GENERIC MAP ( LPM_WIDTH => WIDTH1 )
         PORT MAP ( data => r1, q => q1,clock => clk );
  reg_2: lpm_ff
         GENERIC MAP ( LPM_WIDTH => ONE )
         PORT MAP ( data => cr1, q => cq1, clock => clk );

  add_2: lpm_add_sub                      -- Add MSBs of x and y
         GENERIC MAP ( LPM_WIDTH => WIDTH2,
                       LPM_REPRESENTATION => "UNSIGNED",
                       LPM_DIRECTION => "ADD")
         PORT MAP (dataa => l3, datab => l4, result => r2);
  reg_3: lpm_ff                    -- Save MSBs of x+y
         GENERIC MAP ( LPM_WIDTH => WIDTH2 )
         PORT MAP ( data => r2, q => q2, clock => clk );
------------ Second stage of the adder --------------------
  -- One operand is zero
  h2 <= (OTHERS => '0');

  -- Add result from MSBs (x+y) and carry from LSBs
  add_3: lpm_add_sub
         GENERIC MAP ( LPM_WIDTH => WIDTH2,
                       LPM_REPRESENTATION => "UNSIGNED",
                       LPM_DIRECTION => "ADD")
         PORT MAP ( cin => cq1(0), dataa => q2,
                    datab => h2, result => u2 );

  PROCESS              -- Build a single registered output
  BEGIN                -- word of WIDTH=WIDTH1+WIDHT2
    WAIT UNTIL clk = '1';
    FOR k IN WIDTH1-1 DOWNTO 0 LOOP
      s(k) <= q1(k);
    END LOOP;
    FOR k IN WIDTH2-1 DOWNTO 0 LOOP
      s(k+WIDTH1) <= u2(k);
    END LOOP;
  END PROCESS;
```

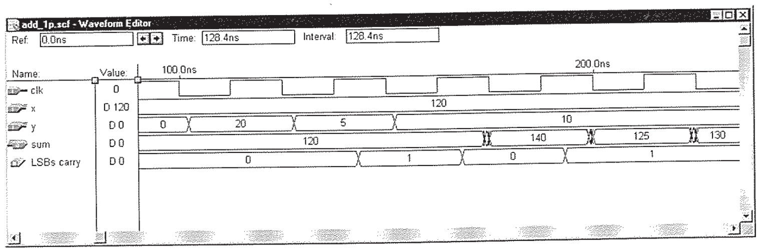

Fig. 2.10. Simulation results for a pipelined adder.

```
    sum <= s ;    -- Connect s to output pins
  END flex;
```

The simulated performance of the 15-bit pipelined adder is shown in Fig. 2.10. Note that the addition of 140 and 130 produces a carry from the lower 7-bit adder, but there is no carry for $120 + 5 = 125 < 127$. $\boxed{2.14}$

2.3.2 Modulo Adders

Modulo adders are the most important building blocks in RNS-DSP designs. They are used for both additions and, via index arithmetic, for multiplications. We wish to describe some design options for FPGAs in the following discussion.

A wide variety of *modular* addition designs exists [43]. Using LEs only, the design of Fig. 2.11(a) is viable for FPGAs. The Altera FLEX devices contain a small number of 2Kbit ROMs or RAMs (EABs) which can be configured as $2^8 \times 8, 2^9 \times 4, 2^{10} \times 2$ or $2^{11} \times 1$ tables and can be used for modulo m_l correction. The next table shows size and `Registered Performance` 6, 7, and 8-bit modulo adder [44].

	Pipeline	Bits		
	stages	6	7	8
MPX	0	41.3 MSPS 27 LE	46.5 MSPS 31 LE	33.7 MSPS 35 LE
MPX	2	76.3 MSPS 16 LE	62.5 MSPS 18 LE	60.9 MSPS 20 LE
MPX	3	151.5 MSPS 27 LE	138.9 MSPS 31 LE	123.5 MSPS 35 LE
ROM	3	86.2 MSPS 7 LE 1 EAB	86.2 MSPS 8 LE 1 EAB	86.2 MSPS 9 LE 2 EAB

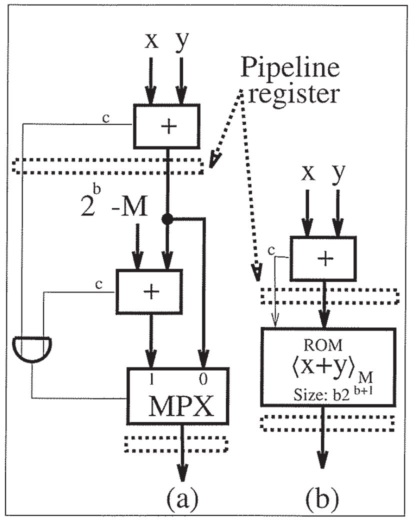

Fig. 2.11. Modular additions. **(a)** MPX-Add and MPX-Add-Pipe. **(b)** ROM-Pipe.

Although the ROM shown in Fig 2.11 provides high speed, the ROM itself produces a four-cycle pipeline delay and the number of ROMs is limited. ROMs, however, are mandatory for the scaling schemes discussed before. The multiplexed-adder (MPX-Add) has a comparatively reduced speed even if a carry chain is added to each column. The pipelined version usually needs the same number of LEs as the unpipelined version but runs about twice as fast. Maximum throughput occurs when the adders are implemented in two blocks within 6-bit pipelined channels.

2.4 Binary Multipliers

The product of two N-bit binary numbers, say X and $A = \sum_{k=0}^{N-1} a_k 2^k$, is given by the "pencil and paper" method as:

$$P = A \cdot X = \sum_{k=0}^{N-1} a_k 2^k X \qquad (2.27)$$

It can be seen that the input X is successively shifted by k positions and whenever $a_k \neq 0$, then $X2^k$ is accumulated. If $a_k = 0$, then the corresponding shift-add can be ignored (i.e., nop). The following VHDL example uses this "pencil and paper" scheme to multiply two 8-bit integers.

Example 2.15: 8-bit Multiplier

The VHDL description[3] of an 8-bit multiplier is developed below. Multiplication is performed in three stages. First, the 8-bit operands are "loaded" and the product register reset. In the second stage, `s1`, the actual serial-parallel multiplication takes place. In the third step, `s2`, the product is transferred to the output register `y`.

```
PACKAGE eight_bit_int IS                -- User defined types
  SUBTYPE BYTE IS integer RANGE -128 TO 127;
  SUBTYPE TWOBYTES IS integer RANGE -32768 TO 32767;
END eight_bit_int;

LIBRARY work;
USE work.eight_bit_int.ALL;

LIBRARY ieee;                           -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY mul_ser IS                               ------> Interface
  PORT ( clk  : IN  STD_LOGIC;
         x    : IN  BYTE;
         a    : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
         y    : OUT TWOBYTES);
END mul_ser;

ARCHITECTURE flex OF mul_ser IS

  TYPE STATE_TYPE IS (s0, s1, s2);
  SIGNAL state    : STATE_TYPE;

BEGIN

  States: PROCESS       ------> Multiplier in behavioral style
    VARIABLE  p, t  : TWOBYTES;             -- Double bit width
    VARIABLE count  : integer RANGE 0 TO 7;
  BEGIN
    WAIT UNTIL clk = '1';
    CASE state IS
      WHEN s0 =>              -- Initialization step
        state <= s1;
        count := 0;
        p := 0;            -- Product register reset
        t := x;            -- Set temporary shift register to x
      WHEN s1 =>              -- Processing step
        IF count = 7 THEN -- Multiplication ready
          state <= s2;
        ELSE
        IF a(count) = '1' THEN
          p := p + t;           -- Add 2^k
        END IF;
```

[3] The equivalent Verilog code `mul_ser.v` for this example can be found in Appendix A on page 353.

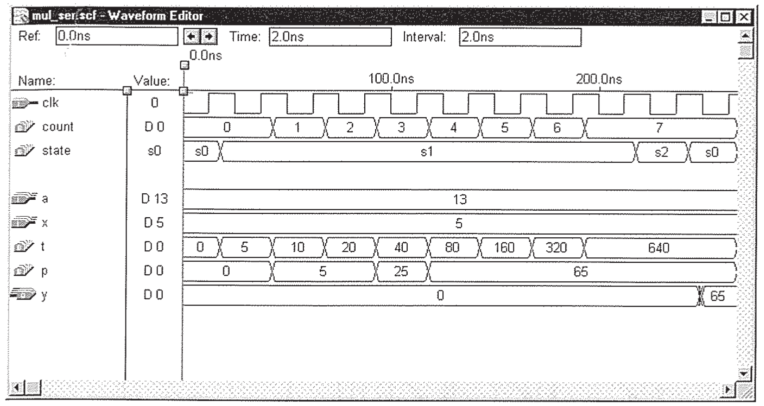

Fig. 2.12. Simulation results for a shift add multiplier.

```
              t := t * 2;
              count := count + 1;
              state <= s1;
            END IF;
          WHEN s2 =>          -- Output of result to y and
            y <= p;           -- start next multiplication
            state <= s0;
        END CASE;
      END PROCESS States;

    END flex;
```

Fig. 2.12 shows the simulation result of a multiplication of 13 and 5. The register t shows the partial product sequence of $5, 10, 20, \ldots$. Since $13_{10} = 00001101_{2C}$, the product register p is updated only three times in the production of the final result, 65. In state s2 the result 65 is transferred to the output y of the multiplier. $\boxed{2.15}$

Because one operand is used in parallel (i.e., X) and the second operand A is used bitwise, the multipliers we just described are called serial/parallel multipliers. If both operands are used serial, the scheme is called a serial/serial multiplier [45], and such a multiplier only needs one full adder, but the latency of serial/serial multipliers is high $\mathcal{O}(N^2)$, because the state machine needs about N^2 cycles.

Another approach, which trades speed for increased complexity, is called an "array," or parallel/parallel multiplier. A 4×4-bit array multiplier is shown in Fig. 2.13. Notice that both operands are presented in parallel to an adder array of N^2 adder cells.

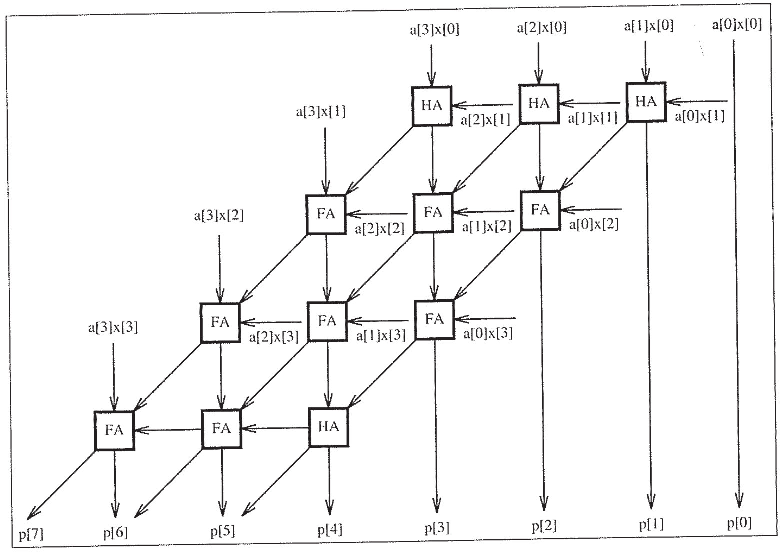

Fig. 2.13. A 4-bit array multiplier.

This arrangement is viable if the time required to complete the carry and sum calculations are the same. For a modern FPGA, however, the carry computation is performed faster than the sum calculation and a different architecture is more efficient for FPGAs. The approach for this array multiplier is shown in Fig. 2.14, for an 8×8-bit multiplier. This scheme combines in the first stage two neighboring partial products $a_n X 2^n$ and $a_{n+1} X 2^{n+1}$ and the results are added to arrrive at the final output product. This is a direct array form of the "pencil and paper" method and must therefore produce a valid product.

We recognize from Fig. 2.14 that this type of array multiplier gives the opportunity to realize a (parallel) *binary tree* of the multiplier with a total:

$$\text{number of stages in the binary tree multiplier} = \log_2(N). \qquad (2.28)$$

This alternative architecture also makes it easier to introduce pipeline stages after each tree level. The necessary number of pipeline stages, according to (2.28), to achieve maximum throughput is:

Bit width	2	3 − 4	5 − 8	9 − 16	17 − 32
Optimal number of pipeline stages	1	2	3	4	5

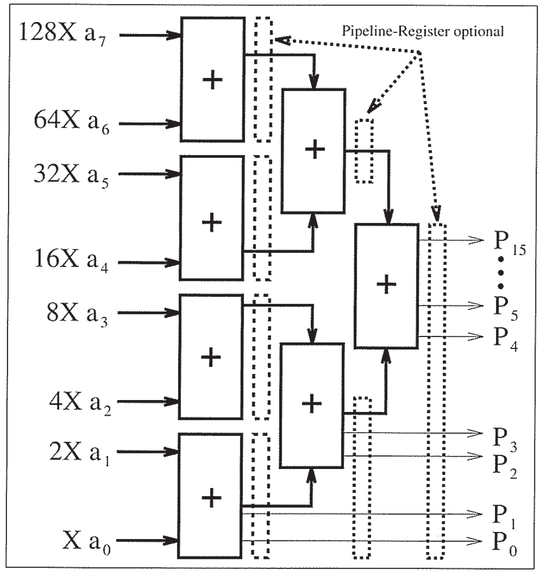

Fig. 2.14. Fast array multiplier for FPGAs.

Fig. 2.15 reports the `Registered Performance` of three pipelined $N \times N$-bit multipliers, using the MaxPlusII `lpm_mult` function, in a range from 4×4 to 16×16 bits without (dotted line) and with (solid line) the use of I/O cell registers. Fig. 2.16 shows the effort for the multiplier, with (solid line) and without (dotted line) using the I/O cell registers. Placing the input register close to the multiplier (turn off option: `Assign`→`Global Project Logic Synthesis`→`Automatic Fast I/O`), produces a slight gain in performance. The maximum size of multiplier that fits in the FLEX10K20 is a 20×20-bit unit that uses 872 LCs and runs at 37.03 MHz `Registered Performance` with four pipeline stages.

Other multiplier architectures typically used in the ASIC world include Booth multipliers and Wallace-tree multipliers. They are discussed in Exercises 2.2 (p. 76) and 2.1 (p. 75) but are rarely used in connection with FPGAs.

2.4.1 Multiplier Blocks

A $2N \times 2N$ multiplier can be defined in terms of an $N \times N$ multiplier block. The resulting multiplication is defined as:

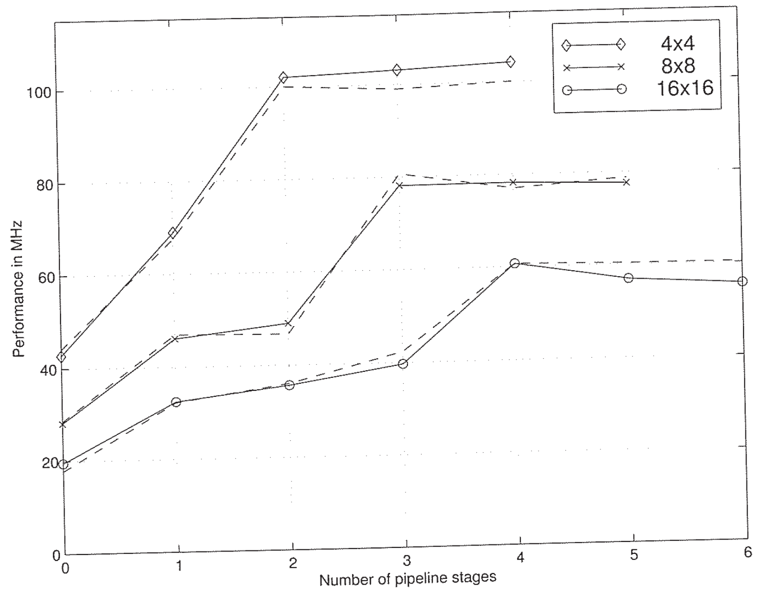

Fig. 2.15. Performance of array multiplier for FPGAs.

$$\begin{aligned} P = Y \cdot X &= (Y_2 2^N + Y_1)(X_2 2^N + Y_1) \\ &= Y_2 Y_2 2^{2N} + (Y_2 X_1 + Y_1 X_2) 2^N + Y_1 X_1 \qquad (2.29) \end{aligned}$$

where the indices 2 and 1 indicate the most significant half and least significant N-bit halves respectively. This partitioning scheme can be used if the capacity of the FPGA is insufficient to implement a multiplier of desired size, or used to implement a multiplier using 2Kbit EAB blocks. The number of EAB blocks in the FLEX10K20 are limited to six, and therefore the maximum symmetric multiplier is 8×8. Table 2.8 shows the data for an EAB-based multiplier. Comparing the data of Table 2.8 with the data from Fig. 2.15 (p. 58) and 2.16 (p. 59), it can be seen that the EAB-based multiplier reduces the number of LCs but does not improve the `Registered Performance`.

2.5 Multiply-Accumulator (MAC) and Sum of Product (SOP)

DSP algorithms are known to be multiply-accumulate (MAC) intensive. To illustrate, consider the linear convolution sum given by

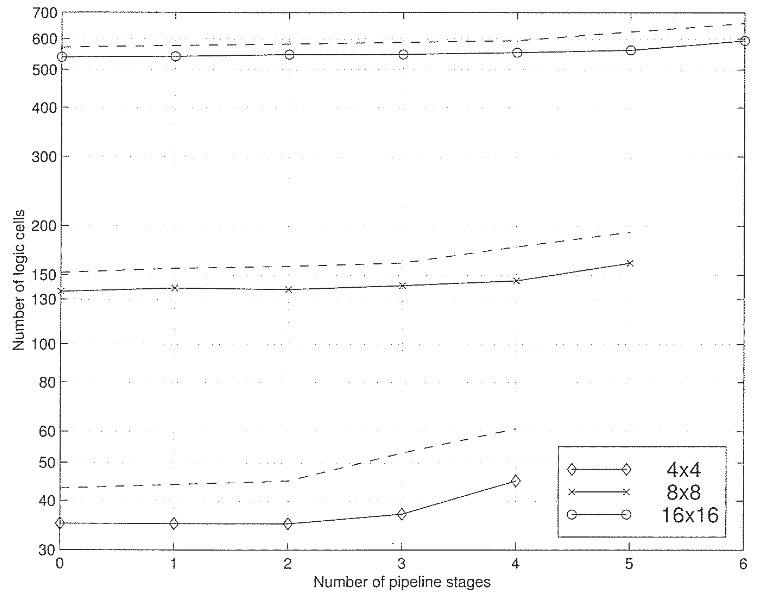

Fig. 2.16. Effort in LCs for array multipliers.

Table 2.8. Data for EAB-based multipliers.

Size	Use I/O register	LCs	EABs	Pipeline stages	`Registered Performance`
4×4	✓	0	1	0	35.58 MHz
4×4	✓	0	1	1	51.54 MHz
4×4	✓	0	1	2	76.33 MHz
4×4	–	16	1	0	37.87 MHz
4×4	–	16	1	1	51.81 MHz
4×4	–	16	1	2	76.33 MHz
8×8	✓	29	4	0	23.80 MHz
8×8	✓	29	4	1	40.81 MHz
8×8	✓	29	4	2	40.81 MHz
8×8	–	49	4	0	24.15 MHz
8×8	–	49	4	1	40.81 MHz
8×8	–	49	4	2	40.81 MHz

$$y[n] = f[n] * x[n] = \sum_{k=0}^{L-1} f[k]x[n-k] \tag{2.30}$$

requiring L consecutive multiplications and $L-1$ addition operations per sample $y[n]$ to compute the sum of products (SOPs). This suggests that $N \times N$-bit multipliers need to be fused together with an accumulator. A full-precision $N \times N$-bit product is $2N$ bits wide. If both operands are (symmetric) signed numbers, the product will only have $2N-1$ significant bits, i.e., two sign bits. The accumulator, in order to maintain sufficient dynamic range, is often designed to have an extra K-bits in width, as demonstrated in the following example.

Example 2.16: The Analog Devices PDSP family ADSP21xx contains a 16×16 array multiplier and an accumulator with an extra 8-bit width (for a total accumulator width of $32 + 8 = 40$ bits). With this 8 extra bits, at least 2^8 accumulations are possible without sacrificing the output. If both operands are signed 2^9 accumulation can be performed. In order to produce the desired output format, such modern PDSPs include also a barrelshifter, which allows the desired adjustment within one clock cycle. $\boxed{2.16}$

This overflow consideration in fixed-point PDSP is important to mainstream digital signal processing, which requires that DSP objects be computed in real-time without unexpected interruptions. Recall that checking and servicing accumulator overflow interrupts the data flow and carries a significant temporal liability. By choosing the number of guard bits correctly the liability can be eliminated.

An alternative approach to the MAC of a conventional PDSP for computing a sum of product will be discussed in the next section.

2.5.1 Distributed Arithmetic Fundamentals

Distributed arithmetic (DA) is an important FPGA technology. It is extensively used in computing the sum of products

$$y = \langle\, \boldsymbol{c}, \boldsymbol{x} \,\rangle = \sum_{n=0}^{N-1} c[n]x[n]. \tag{2.31}$$

Besides convolution, correlation, DFT computation and the RNS inverse mapping discussed earlier can also be formulated as such a "sum of products" (SOPs). Completing a filter cycle, when using a conventional arithmetic unit, would take approximately N MAC cycles. This amount can be shortened with pipelining but can, nevertheless, be prohibitively long. This is a fundamental problem when general purpose multipliers are used.

In many DSP applications, a general purpose multiplication is technically not required. If the filter coefficients $c[n]$ are known apriori, then technically the partial product term $c[n]x[n]$ becomes a multiplication with a constant (i.e., scaling). This is an important difference and is a prerequisite for a DA design.

The first discussion of DA can be traced to a 1973 paper by Croisier [46] and DA was popularized by Peled and Liu [47]. Yiu [48] extended DA to signed numbers, and Kammeyer [49] and Taylor [50] studied quantization effects in DA systems. DA tutorials are available from White [51] and Kammeyer [52]. DA also is addressed in textbooks [53, 54]. To understand the DA design paradigm, consider the "sum of products" inner product shown below:

$$\begin{aligned} y &= \langle\, \boldsymbol{c}, \boldsymbol{x}\, \rangle = \sum_{n=0}^{N-1} c[n] \cdot x[n] \\ &= c[0]x[0] + c[1]x[1] + \ldots + c[N-1]x[N-1], \end{aligned} \tag{2.32}$$

Assume further that the coefficients $c[n]$ are known constants and $x[n]$ is a variable. An unsigned DA system assumes that the variable $x[n]$ is represented by:

$$x[n] = \sum_{b=0}^{B-1} x_b[n] \cdot 2^b \quad \text{with } x_b[n] \in [0, 1], \tag{2.33}$$

where $x_b[n]$ denotes the b^{th} bit of $x[n]$, i.e., the n^{th} sample of $\boldsymbol{x}$. The inner product y can, therefore, be represented as:

$$y = \sum_{n=0}^{N-1} c[n] \cdot \sum_{b=0}^{B-1} x_b[k] \cdot 2^b \tag{2.34}$$

Re-distributing the order of summation (thus the name "distributed arithmetic") results in:

$$\begin{aligned} y = \; & c[0] \left(x_{B-1}[0]2^{B-1} + x_{B-2}[0]2^{B-2} + \ldots + x_0[0]2^0\right) \\ & +c[1] \left(x_{B-1}[1]2^{B-1} + x_{B-2}[1]2^{B-2} + \ldots + x_0[1]2^0\right) \\ & \vdots \\ & +c[N-1] \left(x_{B-1}[N-1]2^{B-1} + \ldots + x_0[N-1]2^0\right) \\ = \; & \left(c[0]x_{B-1}[0] + c[1]x_{B-1}[1] + \ldots + c[N-1]x_{B-1}[N-1]\right) 2^{B-1} \\ & + \left(c[0]x_{B-2}[0] + c[1]x_{B-2}[1] + \ldots + c[N-1]x_{B-2}[N-1]\right) 2^{B-2} \\ & \vdots \\ & + \left(c[0]x_0[0] + c[1]x_0[1] + \ldots + c[N-1]x_0[N-1]\right) 2^0 \end{aligned}$$

or in more compact form

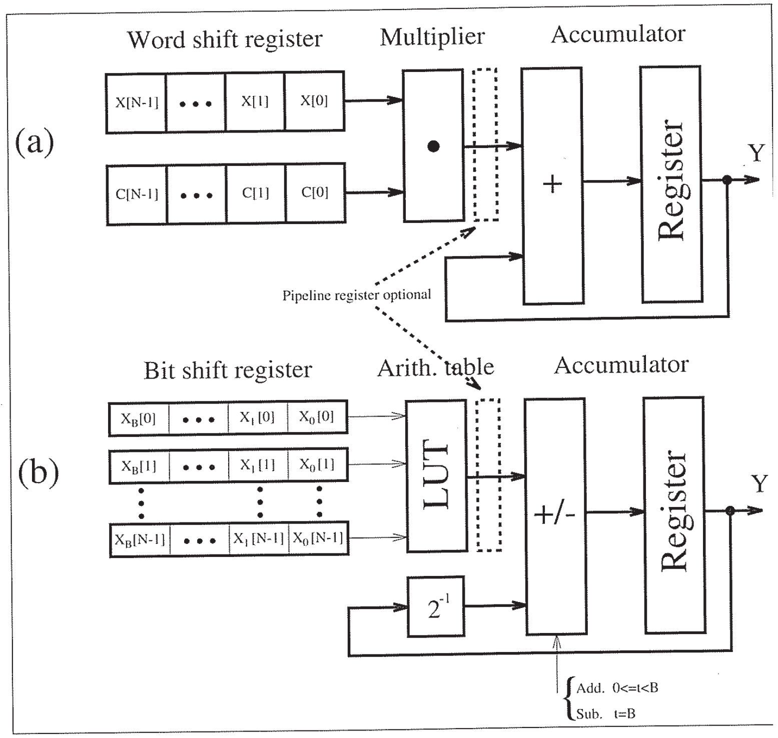

Fig. 2.17. Conventional PDSP and Shift-Adder DA Architecture.

$$y = \sum_{b=0}^{B-1} 2^b \cdot \sum_{n=0}^{N-1} \underbrace{c[n] \cdot x_b[n]}_{f(c[n],x_b[n])} = \sum_{b=0}^{B-1} 2^b \cdot \sum_{n=0}^{N-1} f\left(c[n], x_b[n]\right) \tag{2.35}$$

Implementation of the function $f(c[n], x_b[n])$ requires special attention. The preferred implementation method is to realize the mapping $f(c[n], x_b[n])$ using one LUT. That is, a 2^N-word LUT is preprogrammed to accept an N-bit input vector $\boldsymbol{x}_b = [x_b[0], x_b[1], \cdots, x_b[N-1]]$, and output $f(c[n], x_b[n])$. The individual mappings $f(c[n], x_b[n])$ are weighted by the appropriate power-of-two factor and accumulated. The accumulation can be efficiently implemented using a shift-adder as shown in Fig. 2.17(b). After N look-up cycles, the inner product y is computed.

Example 2.17: Unsigned DA Convolution

A third-order innerproduct is defined by the inner product equation $y = \langle \boldsymbol{c}, \boldsymbol{x} \rangle = \sum_{n=0}^{2} c[n]x[n]$. Assume that the 3-bit coefficients have the values $c[0] = 2$, $c[1] = 3$, and $c[2] = 1$. The resulting LUT, which implements $f(c[n], x_b[n])$, is defined below:

$x_b[2]$	$x_b[1]$	$x_b[0]$	$f(c[n], x[n])$
0	0	0	$1 \cdot 0{+}3 \cdot 0{+}2 \cdot 0{=}0_{10}{=}000_2$
0	0	1	$1 \cdot 0{+}3 \cdot 0{+}2 \cdot 1{=}2_{10}{=}001_2$
0	1	0	$1 \cdot 0{+}3 \cdot 1{+}2 \cdot 0{=}3_{10}{=}011_2$
0	1	1	$1 \cdot 0{+}3 \cdot 1{+}2 \cdot 1{=}5_{10}{=}101_2$
1	0	0	$1 \cdot 1{+}3 \cdot 0{+}2 \cdot 0{=}1_{10}{=}001_2$
1	0	1	$1 \cdot 1{+}3 \cdot 0{+}2 \cdot 1{=}3_{10}{=}011_2$
1	1	0	$1 \cdot 1{+}3 \cdot 1{+}2 \cdot 0{=}4_{10}{=}100_2$
1	1	1	$1 \cdot 1{+}3 \cdot 1{+}2 \cdot 1{=}6_{10}{=}110_2$

The inner-product, with respect to $x[n] = \{x[0] = 1_{10} = 001_2, x[1] = 3_{10} = 011_2, x[2] = 7_{10} = 111_2\}$, is obtained as follows:

Step t	$x_t[2]$	$x_t[1]$	$x_t[0]$	$f[t]$	$+ACC[t-1]$	$=ACC[t]$
0	1	1	1	$6 \cdot 2^0 +$	0	$=$ 6
1	1	1	0	$4 \cdot 2^1 +$	6	$=$ 14
2	1	0	0	$1 \cdot 2^2 +$	14	$=$ 18

As a numerical check, note that

$$y = \langle c, x \rangle = c[0]x[0] + c[1]x[1] + c[2]x[2]$$
$$= 2 \cdot 1 + 3 \cdot 3 + 1 \cdot 7 = 18. \checkmark$$

$\boxed{2.17}$

For a hardware implementation, instead of shifting each intermediate value by b (which will demand an expensive barrelshifter) it is more appropriate to shift the accumulator content itself in each iteration one bit to the right. It is easy to verify that this will give the same results.

The bandwidth of an N^{th}-order B-bit linear convolution, using general purpose MACs and DA hardware, can be compared. Fig. 2.17 shows the architectures of a conventional PDSP and the same realization using distributed arithmetic.

Assume that a LUT and a general purpose multiplier have the same delay $\tau = \tau(\text{LUT}) = \tau(\text{MUL})$. The computational latencies are then $B\tau(\text{LUT})$ for DA and $N\tau(\text{MUL})$ for the PDSP. In the case of small-bit-width B, the speed of the DA design can therefore be significantly faster than a MAC-based design. In Chapter 3, comparisons will be made for specific filter design examples.

2.5.2 Signed DA Systems

In the following we wish to discuss how (2.32) should be modified, in order to process a signed two's complement number. In two's complement, the MSB is used to distinguish between positive and negative numbers. For instance, from Table 2.1 (p. 32) we see that decimal -3 is coded as $101_2 = -4{+}0{+}1 = -3_{10}$. We use, therefore, the following $(B+1)$ bit representation

$$x[n] = -2^b \cdot x_B[n] + \sum_{b=0}^{B-1} x_b[n] \cdot 2^b. \tag{2.36}$$

Combining this with (2.34), the outcome y is defined by:

$$y = -2^b \cdot f(c[n], x_B[n]) + \sum_{b=0}^{B-1} 2^b \cdot \sum_{n=0}^{N-1} f\left(c[n], x_b[n]\right). \tag{2.37}$$

To achieve the signed DA system we therefore have two choices to modify the unsigned DA system. They are

- An accumulator with add/subtract control
- Using a ROM with one additional input

Most often the switchable accumulator is preferred, because the additional input bit in the table requires a table with twice as many words. The following example demonstrates the processing steps for the add/sub switch design.

Example 2.18: Signed DA Inner Product

Consider again a third-order inner product defined by the convolution sum $y = \langle \boldsymbol{c}, \boldsymbol{x} \rangle = \sum_{n=0}^{2} c[n]x[n]$. Assume that the data is given a $N = 4$-bit two's complement encoding and that the coefficients are $c[0] = -2$, $c[1] = 3$, and $c[2] = 1$. The corresponding LUT table is given below:

$x_b[2]$	$x_b[1]$	$x_b[0]$	$f(c[k], x[n])$
0	0	0	$1 \cdot 0{+}3 \cdot 0{-}2 \cdot 0{=}\ 0_{10}$
0	0	1	$1 \cdot 0{+}3 \cdot 0{-}2 \cdot 1{=}{-}2_{10}$
0	1	0	$1 \cdot 0{+}3 \cdot 1{-}2 \cdot 0{=}\ 3_{10}$
0	1	1	$1 \cdot 0{+}3 \cdot 1{-}2 \cdot 1{=}\ 1_{10}$
1	0	0	$1 \cdot 1{+}3 \cdot 0{-}2 \cdot 0{=}\ 1_{10}$
1	0	1	$1 \cdot 1{+}3 \cdot 0{-}2 \cdot 1{=}{-}1_{10}$
1	1	0	$1 \cdot 1{+}3 \cdot 1{-}2 \cdot 0{=}\ 4_{10}$
1	1	1	$1 \cdot 1{+}3 \cdot 1{-}2 \cdot 1{=}\ 2_{10}$

The values of $x[k]$ are $x[0] = 1_{10} = 0001_{2C}$, $x[1] = -3_{10} = 1101_{2C}$, and $x[2] = 7_{10} = 0111_{2C}$. The output at sample index k, namely y, is defined as follows:

Step t	$x_t[2]$	$x_t[1]$	$x_t[0]$	$f[t] \cdot 2^t$	$+Y[t-1]$	$=Y[t]$
0	1	1	1	$2 \cdot 2^0\ +$	0	$=$ 2
1	1	0	0	$1 \cdot 2^1\ +$	2	$=$ 4
2	1	1	0	$4 \cdot 2^2\ +$	4	$=$ 20
	$x_t[2]$	$x_t[1]$	$x_t[0]$	$-f[t] \cdot 2^t$	$+Y[t-1]$	$=Y[t]$
3	0	1	0	$-3 \cdot 2^3\ +$	20	$=-4$

A numerical check results in $c[0]x[0]+c[1]x[1]+c[2]x[2] = -2{\cdot}1+3{\cdot}(-3)+1{\cdot}7 = -4$ ✓ $\boxed{2.18}$

2.5.3 Modified DA Solutions

In the following we wish to discuss two interesting modifications to the basic DA concept, where the first variation reduces the size, and the second increases the speed.

If the number of coefficients N is too large to implement the full word with a single LUT (recall that input LUT bit width = number of coefficients), then we can use partial tables and add the results. If we also add pipeline registers, this modification will not reduce the speed, but can dramatically reduce the size of the design, because the size of a LUT grows exponentially with the address space, i.e., the number of input coefficients N. Suppose the length LN inner product

$$y = \langle \boldsymbol{c}, \boldsymbol{x} \rangle = \sum_{n=0}^{LN-1} c[n]x[n] \tag{2.38}$$

is to be implemented using a DA architecture. The sum can be partitioned into L independent N^{th} parallel DA LUTs resulting in

$$y = \langle \boldsymbol{c}, \boldsymbol{x} \rangle = \sum_{l=0}^{L-1} \sum_{n=0}^{N-1} c[Ll+n]x[Ll+n] \tag{2.39}$$

This is shown in Fig. 2.18 for a realization of a $4N$ DA design requiring three post-additional adders. The size of the table is reduced from one $4N \times 2^B$ LUT to four $N \times 2^B$ tables.

Another variation of the DA architecture increases speed at the expense of additional LUTs, registers, and adders. A basic DA architecture, for a length N^{th} sum-of-product computation, accepts one bit from each of N words. If two bits per word are accepted, then the computational speed can be essentially doubled. The maximum speed can be achieved with the fully pipelined word-parallel architecture shown in Fig. 2.19. Here, a new result of a length four sum-of-product is computed for 4-bit signed coefficients at each LUT cycle. For maximum speed, we have to provide a separate ROM (with identical content) for each bit vector $x_b[n]$. But the maximum speed can become expensive: If we double the input bit width, we need twice as many LUTs, adders and registers. If the number of coefficients N is limited to four or eight this modification gives attractive performance, essentially outperforming all commercially available programmable signal processors, as we will see in Chapter 3.

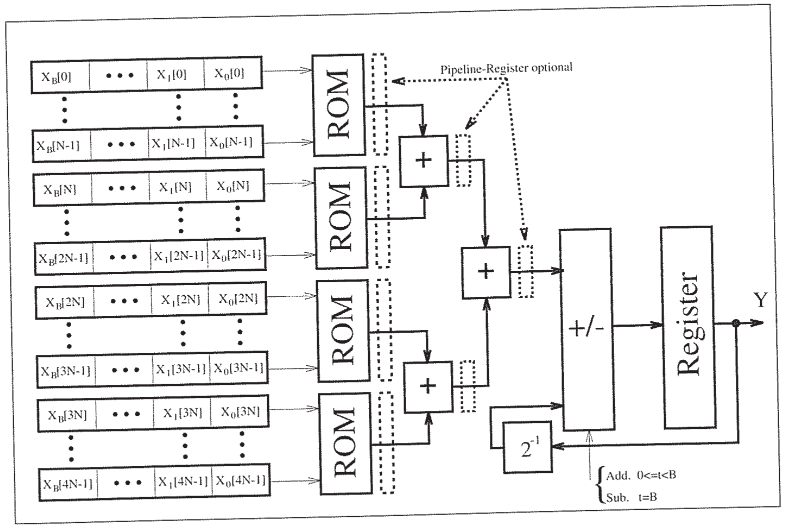
X_B[0] ••• X_1[0] X_0[0]
X_B[N-1] ••• X_1[N-1] X_0[N-1]
X_B[N] ••• X_1[N] X_0[N]
X_B[2N-1] ••• X_1[2N-1] X_0[2N-1]
X_B[2N] ••• X_1[2N] X_0[2N]
X_B[3N-1] ••• X_1[3N-1] X_0[3N-1]
X_B[3N] ••• X_1[3N] X_0[3N]
X_B[4N-1] ••• X_1[4N-1] X_0[4N-1]
ROM
Pipeline-Register optional
+
+/-
Register
Y
2^-1
Add. 0<=t<B
Sub. t=B

Fig. 2.18. Distributed arithmetic with table partitioning to yield a reduced size.

2.6 Computation of Special Functions Using CORDIC

If a digital signal processing algorithm is implemented with FPGAs and the algorithm uses a nontrivial (transcendental) algebraic function, like $\sqrt{x}$ or $\arctan y/x$, we can always use the Taylor series to approximate this function, i.e.,

$$f(x_0) = \sum_{k=0}^{K} \frac{\mathrm{d}f^{(k)}(x - x_0)}{\mathrm{d}x} x^k \bigg|_{x=x_0} \tag{2.40}$$

and the problem is reduced to a sequence of multiply and add operations. A more efficient, alternative approach, based on the *Coordinate Rotation Digital Computer* (CORDIC) algorithm can also be considered. The CORDIC algorithm is found in numerous applications, such as pocket calculators [55], and in mainstream DSP objects, such as adaptive filters, FFTs, DCTs [56], demodulators [57], and neural networks [36]. The basic CORDIC algorithm can be found in two classic papers by Volder [58] and Walther [59]. Some theoretical extensions have been made, such as the extension of range in the hyperbolic mode, or the quantization error analysis by Hu et al. [60], and Meyer-Bäse et al. [57]. VLSI implementations have been discussed in Ph.D. theses, such as those by Timmermann [61] or Hahn [62]. The first FPGA implementations were investigated by Meyer-Bäse et al. [4, 57]. The realization of the CORDIC algorithm in distributed arithmetic was investigated by Ma