Chapter 2
Basic Arithmetic Circuits

This chapter is devoted to the description of simple circuits for the implementation of some of the arithmetic operations presented in Chap. 1. Specifically, the design of adders, subtractors, multipliers, dividers, comparators and shifters are studied, with the objective of providing the design guidelines for these specific application circuits. The arithmetic circuits presented will be used in the next chapters for the implementation of algebraic circuits.

2.1 Introduction

This section presents the previous aspects related to the arithmetic circuits: differences between serial and parallel information, pipelining, or circuits multiplicity for increasing performance. Although these concepts will be probably known by the reader, they are included in order to provide an immediate reference.

2.1.1 Serial and Parallel Information

When transmitting or processing information, two extreme structures can be considered: serial and parallel information. Briefly, we have serial information when the bits integrating each of the information blocks are transmitted or processed at different times. On the contrary, we have parallel information when the bits composing each information block are transmitted or processed simultaneously.

The clearest example for discriminating between serial and parallel information resides on information transmission. Assuming the design of a system for performing some numerical calculations formed by several subsystems, and being each data 8-bit wide (i.e., 8-bit words must be processed), the information among the different subsystems can be transmitted using 8 wires. In this case, the 8 bits are transmitted simultaneously, at the same time, thus being parallel information. But this information can be also transmitted using only 1 wire, sending the 8 bits of

A. Lloris Ruiz et al., *Algebraic Circuits*, Intelligent Systems Reference Library 66,
DOI: 10.1007/978-3-642-54649-5_2, © Springer-Verlag Berlin Heidelberg 2014

a data block bit by bit, with a predetermined order and at 8 different times, constituting serial information. When using serial information, usually the first bit being transmitted and/or processed is the less significant one, but it could be also the most significant one.

Intermediate situation between serial and parallel structures can be considered. Each word can be divided into blocks (known as digits), being processed in parallel the bits corresponding to each digit, but being the different digits transmitted or processed in a serial way. As an example, a 64-bit word can be processed or transmitted in serial (taking 64 cycles), in parallel (taking only one cycle by using a 64-wire bus), in 16-digit of 4 bits (taking 16 cycles by using a 4-wire bus), in 8-digit of 8 bits (taking 8 cycles by using a 8-wire bus), etc.

2.1.2 Circuit Multiplicity and Pipelining

Every digital circuit **C** (Fig. 2.1a) establishes a correspondence between the inputs, E, and the outputs, S, $S = \mathrm{F}(E)$. Given an input at a given time, the most efficient circuit in terms of temporal response will be the combinational circuit capable of generating the output in the same cycle when the input has arrived. The complexity of this circuit depends mainly on the number of the input bits (input size). If the output is not needed in the same cycle, probably a simpler sequential circuit can be built generating the output S some clock cycles later from the input E arrival. However, in the case of a continuous input data flow, and being necessary generating a result in each cycle, the complexity of a circuit can produce large delays, preventing the output being generated in the same cycle than the input arrival. For maintaining a continuous data flow at the output, two alternatives can be considered: circuit multiplicity and pipelining, as detailed in the following.

Circuit multiplicity (Fig. 2.1b) consists on using m identical circuits (as many as the delay introduced by each one of the circuits), working in parallel. The inputs of the m circuits are connected to the outputs of a 1-to-m demultiplexer whose input is E. The function of the demultiplexer consists on driving the data to the circuit C_i being available in each time for starting the calculation. The outputs of the m circuits are connected with the inputs of an m-to-1 multiplexer, with output S. The function of the multiplexer consists on selecting at each time the output of the circuit being finished the calculation. In this way, during the first m cycles no result is generated, and from this moment, a calculation result will be generated at each cycle. Note that the result collected at the output in a given time corresponds to the inputs introduced m cycles before. This delay between input and output sequences is known as the **latency** of the system. Circuit multiplicity presents the advantage of simplicity because the design is reduced to placing as many circuits in parallel as indicated by the latency. As a drawback, the cost of the system can result excessive.

The **pipelining** of a combinational circuit (Fig. 2.1c), in their simplest version, consists on modifying the original circuit dividing it into n segments, each one completing the corresponding processing in one clock cycle. Each of the segments

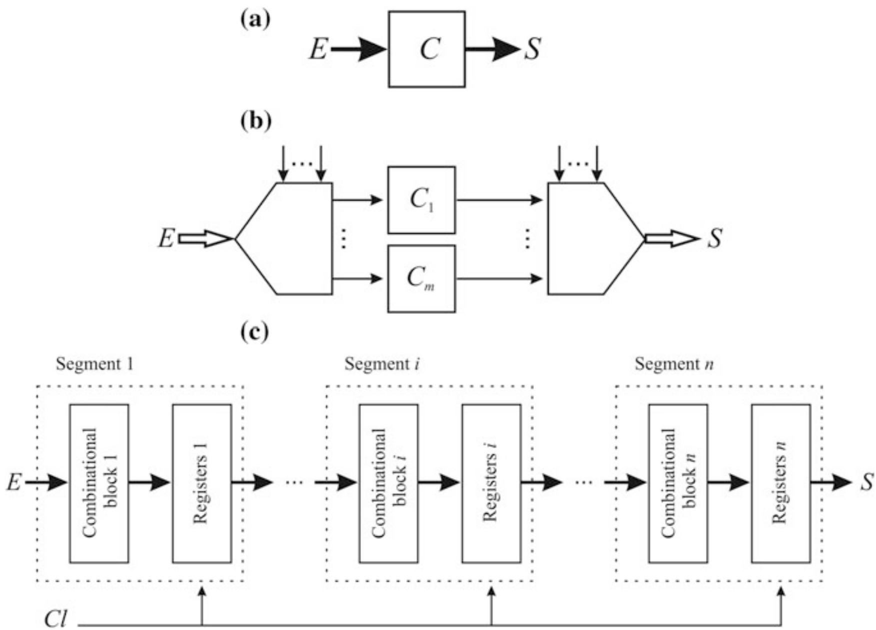


Fig. 2.1 **a** Circuit. **b** Multiplicity. **c** Pipelining

includes a register storing its output, making it available for the next segment. The registers at the different segments are controlled by the same clock signal. The pipelined circuit allows a continuous data flow at the input *E*, and after the initial *n* cycles delay due to the different segments (the latency of the pipelined circuit), a continuous data flow at the output *S*, is obtained. Thus, the output at a time corresponds with the input introduced into the pipelined circuit *n* clock cycles before. Each segment executes one part of the complete calculation, being then *n* data sets computed in parallel, each one with a different phase and at a different segment. Note that this structure is similar to the assembly line of a factory, where the global task is decomposed into simpler tasks, in such a way that each assembly machine (with the corresponding workers) performs only one of these elementary tasks.

When using pipelined circuits with the structure presented in Fig. 2.1c, each segment generates correctly its output in one cycle, and is used only once for generating each result. More complex circuits can be used where some or all of the segments are used more than once for generating each result. Also, pipelined circuits with each segment consisting on a sequential machine needing more than one clock pulse, can be defined.

Circuit multiplicity and pipelining can be combined, creating mixed solutions. Parallel units in the structure presented in Fig. 2.1b can be pipelined, generating a result every *m* clock cycles. Some of the segments in Fig. 2.1c structure can include element multiplicity.

2.2 Binary Adders

In this section, elementary circuits for adding two summands, using information in parallel or in series, are described. Also a pipelined adder is presented. First, half-adders are introduced, together with the full-adders, which will be the basic blocks for building adder circuits, and also will be widely used in the remainder of arithmetic circuits.

2.2.1 Parallel Adders

In the following, simplest binary adders are described. Thus, it will be assumed the situation of adding two positive numbers without sign bit. As an example, let consider the addition:

$$A + B = 1011 + 0011$$

Fig. 2.2 **a** Addition examples. Two bits addition tables: **b** Sum. **c** Carry

(a)	(b)			(c)		
1001	xy	0	1	xy	0	1
+0101	0	0	1	0	0	0
1110	1	1	0	1	0	1

Arranging these two summands as usual, one below the other, as it is done in the Fig. 2.2a, first the two bits corresponding to the position 2^0 are added, obtaining the bit of the result at the same position. For obtaining the bit at position 2^i ($i = 1, \ldots, n$) of the result, the two bits at this position are added together with the precedent carry. Partial sums and the carry for the next stage are obtained from addition tables (Table 1.1a), which are repeated in Fig. 2.2b, c using other arrangement. In our example, the result is: 1011 + 0011 = 1110.

The functions corresponding to the partial sum, s, and to the carry, a, are:

$$s = \bar{x}y + x\bar{y} = x \oplus y; \quad a = x \cdot y$$

Synthesizing these two functions as a combinational block (using two **AND-OR** gate levels or using **XOR** gates), in the way represented in Fig. 2.3a, it results the circuit known as half-adder. This block, represented in Fig. 2.3b, allows the less significant bit of a sum to be obtained, while the remainder of the bits require two half-adders to be calculated. Connecting in cascade several half-adders in the way represented in Fig. 2.3c, binary numbers with an arbitrary number of bits can be added. For obtaining the carry in a given stage, the **OR** operation must be performed over the carries generated by the two half-adders, because the two

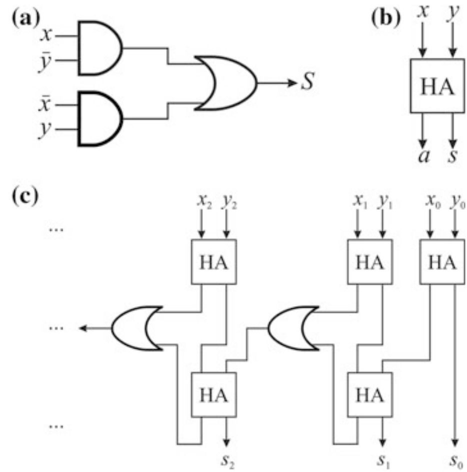


Fig. 2.3 Half-adder: **a** Circuit. **b** Representation. **c** Cascading

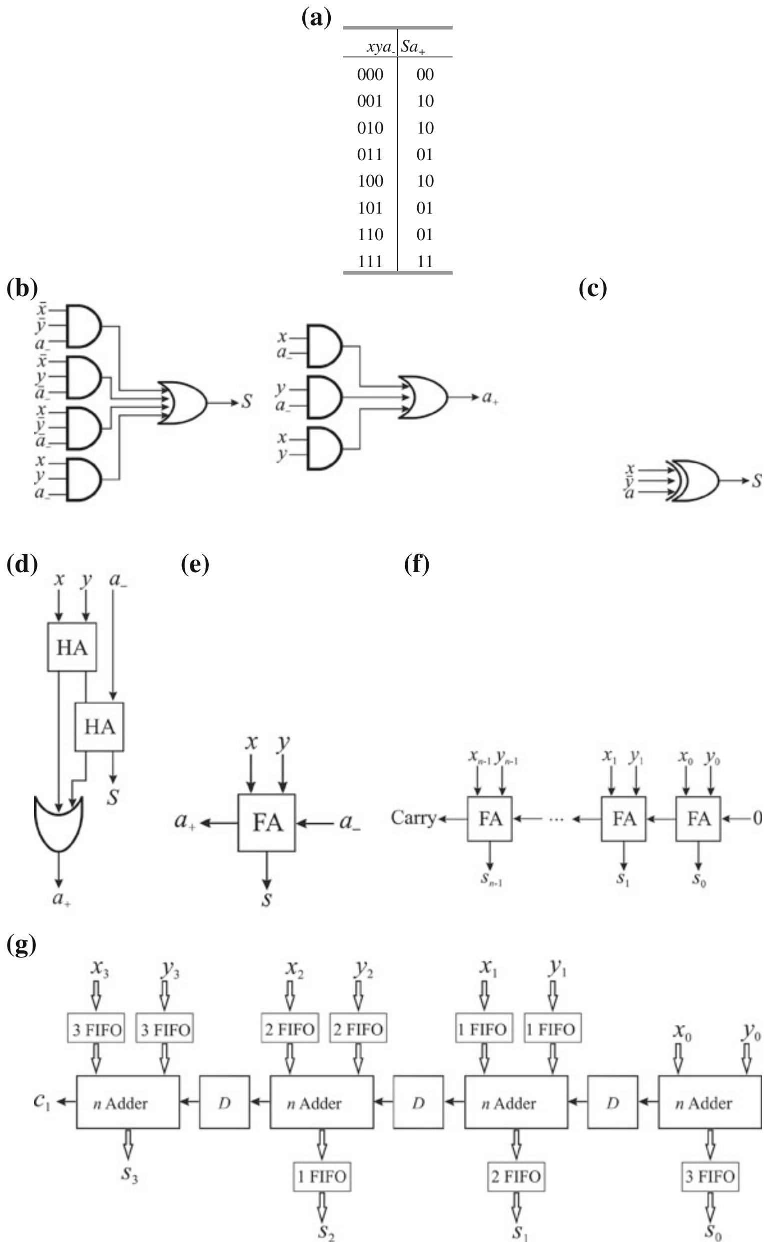
half-adders in a same stage cannot produce simultaneously carry ‘1’, as can be easily proved.

The calculation of sum and carry at each position can be also performed by means of a combinational block known as full-adder, with three inputs (the two summand bits at this position, x and y, plus the previous carry, a_-) and two outputs, S and a_+. From the truth table of the two functions (Fig. 2.4a) to be synthesized by this block, it results:

$$S = \bar{x}\bar{y}a_- + \bar{x}y\bar{a}_- + x\bar{y}\bar{a}_- + xya_- = x \oplus y \oplus a_-; \quad a_+ = xa_- + ya_- + xy$$

The full-adder block can be implemented using **AND-OR** synthesis (Fig. 2.4b) or using an **XOR** gate for the sum S (Fig. 2.4c), a_+ can be synthesized as shown in Fig. 2.4b or concatenating two adders plus an **OR** gate (Fig. 2.4d), and it is represented in Fig. 2.4e.

For adding n-bit numbers with parallel information, simply connect n full adders in cascade (Fig. 2.4f). The parallel n-bit adder resulting is known as **ripple carry adder**. This adder presents the drawback of the delay introduced by the carry propagation through the successive stages. In fact, the result at the carry output of the most significant bit of the sum must wait for any change at the carry input of the less significant bit being propagated. When the size of the summands (number of bits) is not excessive (from 4 to 16), or the circuit’s performance is not relevant, this drawback has no impact. However, when the size of the operands is large or a high operation speed is required, it may be that the result of the addition can not be generated correctly in one cycle. In this situation, alternative solutions accelerating carry propagation should be used, leading to carry look-ahead adders, or special procedures for adding. Pipelining of circuit detailed in Fig. 2.4f, the addition of more than one bit in each stage, and the addition of more than two summands at a time, are among the options for speeding up the adders operation.



xya_-	Sa_+
000	00
001	10
010	10
011	01
100	10
101	01
110	01
111	11

Fig. 2.4 Full adder: **a** Truth table. Synthesis: **b** AND-OR. **c** With an XOR gate. **d** With half-adders cascading. **e** Representation. **f** Ripple carry adder. **g** Pipelined ripple carry adder

When using biased representation, as shown in Sect. 1.4.3, and making $D = 2^{m-1}$, the same adders presented here can be used appending an inverter for the less significant bit. Similarly, if $D = 2^{m-1} - 1$, in addition to complementing the most significant bit, the initial carry must be 1.

2.2.2 Pipelined Adders

In several applications, like those involved in digital signal processing, a continuous data flow with multiple additions must be made. In this situation, the ripple carry adder results unsuitable because of its excessive delay, but it can be easily pipelined introducing registers in the appropriate locations. Assuming r is the delay corresponding to a full adder, and f is the clock frequency, then the maximum length m of the adder providing the result in each of the cycles will be:

$$m \leq \frac{1}{r \cdot f}$$

For building an n-bits adder, it must be divided into s segments, being:

$$s \geq \frac{n}{m}$$

Obviously, if $n \neq m \cdot s$, then one of the segments (usually the first one or the last one) can be shorter than the rest.

Each segment will be separated from the following by a **D** flip-flop in order to store the carry between stages. The inputs and outputs will be separated, in general, by means of register stacks (**FIFO** registers). All of the registers will have so many bits as the corresponding segment length (in the previous example, m bits), and the size or depth of each one of the stacks (i.e., the number of registers stacked) will depend on the segment position, with the objective of properly synchronizing inputs and outputs, as represented in Fig. 2.4g for an adder composed by four m-bits segments. The depth of each **FIFO** is indicated in Fig. 2.4g by the first digit in their name. The latency of these adders is 4.

2.2.3 Serial Adders

When the summands (X and Y) are serially available bit by bit (being the first bit the less significant one), they can be added using a full adder and a **D** flip-flop in order to store the carry generated for the next stage, as shown in Fig. 2.5a. For a correct operation, the **D** flip-flop must be initialized to 0. At the output S, the addition is obtained serially. The final carry will remain at D, but it can be transferred to S introducing one '0' into each input after the most significant bits of both summands.

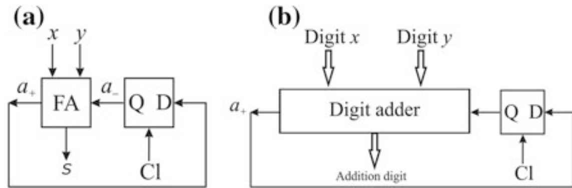


Fig. 2.5 Serial adder: **a** Bit by bit. **b** Digit by digit

For Serial operands digit by digit (the first digit is the less significant one, again) a parallel digit adder and a **D** flip-flop (initialized to '0') are required, as shown in Fig. 2.5b. The digit adder can be built using as many full adders as the size of the digit. Again, the final carry remains in the **D** flip-flop, but it can be transferred to S introducing one digit will all zeros into each input after the most significant digits of both summands.

Comparing serial processing with parallel processing, it is clear that the series circuits are simpler than the parallel, both in number of gates (less full adders in this case) and the number of inputs and outputs. With regard to the processing time, with serial structures as many computation cycles as blocks forming each word are required, whereas with parallel information only one cycle is sufficient. However, the serial adder, because it is simpler than the parallel, withstands higher speeds than parallel, i.e. the serial adder will require more cycles, but each cycle can be of shorter duration.

2.3 Binary Subtractors

Subtraction tables (Table 1.1b, and repeated in Fig. 2.6a) implementing the functions corresponding to the partial difference r, and the borrow, d, are:

$$r = \bar{x}y + x\bar{y} = x \oplus y$$
$$d = \bar{x}y$$

Synthesizing r and d functions (r fits with the partial sum S from the half-adder), half-subtractors are obtained, which can be cascaded in a similar way to that shown in Fig. 2.3c for half-adders, allowing the subtraction of binary numbers with any number of bits, as shown in Fig. 2.6b.

Also full-subtractors can be designed for 1-bit characters. In this case, the truth table corresponding to $x - y$, including the previous borrow, is given in Fig. 2.6c, resulting the following functions:

$$R = \bar{x}\bar{y}d_- + \bar{x}y\bar{d}_- + x\bar{y}\bar{d}_- + xyd_- = x \oplus y \oplus d_-$$
$$d_+ = \bar{x}d_- + \bar{x}y + yd_-$$

For subtracting unsigned binary n-bit numbers, $X - Y$, the *ripple-borrow subtractor* of Fig. 2.6d can be used. When $X \geq Y$, this subtractor generates the

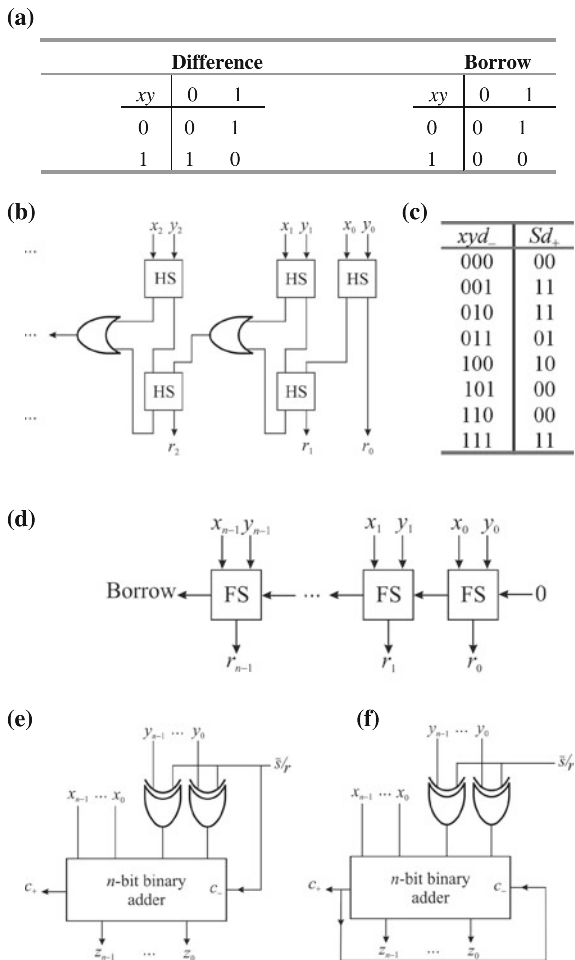


Fig. 2.6 **a** Subtraction table. **b** Half-subtractors cascading. **c** Full-subtractor table. **d** Full-subtractors cascading. Adder/subtractor: **e** Two's complement. **f** One's complement

correct result being the final borrow 0, as can be easily checked by the reader. When $X < Y$, the final borrow is 1, and the result is not correct. Thus, the result generated is correct only when it is positive. For taking into account negative results with this subtractor, a comparator must be included for detecting which operand is the greatest one, as was shown when introducing the **SM** representation. Other alternative is the use of complement adders/subtractors, as detailed in the following.

Because of the common part of the full-adder and the full-subtractor, are often built as a single adder/subtractor block, with a control input for selecting between the two operations.

As shown in Sect. 1.4.3, when using biased representations and making $D = 2^{m-1}$, the same subtractors described for **SM** can be used, adding an inverter for

the most significant bit. In a similar way, if $D = 2^{m-1} - 1$, an inverter must be added, with an initial borrow of 1.

About the subtraction using complement representations, when using two's complement, subtraction consists of adding to the minuend, the two's complement of the subtrahend. On the other hand, the complementation is performed by complementing all bits and adding 1 to the result. Joining these ideas, the circuit of Fig. 2.6e can be carried out for implementing a two's complement adder/subtractor. The control signal $\bar{s}/r$ must be 0 for adding, and 1 for subtracting (the detailed analysis of the circuit is left as an exercise for the reader). In this circuit, making $X = x_{n-1} \ldots x_0 = 0 \ldots 0$, and $\bar{s}/r = 1$, the two's complement of Y is obtained. With a similar idea, the Fig. 2.6f shows a one's complement adder/subtractor, as can be easily checked. In this situation, the end-around carry must be included, using the carry out as input carry. This end-around condition makes the two's complement advantageous with respect to one's complement representations, as seen comparing Fig. 2.6e, f.

2.4 Multipliers

In the following, some simple circuits for multiplication, both combinational and sequential, for integer binary numbers will be described. Also, the design of circuits for multiplying by a constant and for raising to an integer power will be approached.

2.4.1 Combinational Multipliers

To give an idea of how to build these circuits, without too much detail, we will first consider multipliers for binary coded positive integers (without sign bit). Such multipliers are widely used in signal processing applications and can be the core of multipliers when using signed binary numbers.

When multiplying an m-bit number A by an n-bit number B (both unsigned positive numbers), the product P will take $m + n$ bits. In fact:

$$A \leq 2^m - 1$$
$$B \leq 2^n - 1$$

thus $P = A \cdot B \leq (2^m - 1)(2^n - 1) = 2^{m+n} - 2^m - 2^n + 1$. Then, except in the situations $m = 1$ or $n = 1$, $m + n$ bits are required for representing P.

The most elemental multiplier is the one for one-bit characters, whose table is presented in Table 1.1c (and repeated in Fig. 2.7a). In this case, the operation is the **AND** function, and the result is represented by using only one bit.

When multiplying 2-bit integer positive numbers, $X = x_1x_0$ e $Y = y_1y_0$, 4 bits are required for representing the product $M = X \cdot Y$. This multiplier can be designed as a combinational circuit with four inputs and four outputs, which truth table and circuit are presented in Fig. 2.7b, c. This circuit can be also interpreted as a base-4 multiplier of two 1-digit characters, and synthesized by using elemental multipliers and adders. In fact, Fig. 2.7d details the X by Y multiplication, and Fig. 2.7e presents the circuit with this design strategy, using four 1-bit multipliers and two half-adders. With independence of the design used, a multiplier of two 2-bit characters (or two 1-digit base-4 characters) is represented as in Fig. 2.7f.

For building circuits enabling the multiplication of characters with any number of bits, the same techniques used for 2-bit numbers can be used. As an example, for multiplying 4-bit characters (a base-16 elemental multiplier), a combinational circuit with eight inputs and eight outputs can be synthesized and implemented using a programmable device (a **ROM** for example) or in any other way. Nevertheless, when the size of the characters to be multiplied increases, this synthesis technique leads to bulky and difficult to manage circuits. In this situation the multipliers are synthesized by combining elemental multipliers and adders. Figure 2.8a shows the method of operation for multiplying two 4-bit numbers. The circuit in Fig. 2.8b implements mimetically this method using 1-bit multipliers, half-adders and full-adders. Also, 2-bit multipliers and adders could be used as building blocks. In this case, being $X = X_1X_0$ ($X_1 = x_3x_2$, $X_0 = x_1x_0$), $Y = Y_1Y_0$ ($Y_1 = y_3y_2$, $Y_0 = y_1y_0$), multiplication can be performed as described in Fig. 2.8c, and the circuit of Fig. 2.8d is also a 4-bit characters multiplier.

Building methods based on the use of elementary multipliers and adders allow the design of combinational multipliers of any size.

In general, given two base b numbers (P and Q) to be multiplied, they can be decomposed into two or more pieces, and then processing these pieces using less complex resources. As an example, decomposing P and Q into two pieces it results:

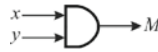
$$
\begin{aligned}
P &= p_{n-1}b^{n-1} + p_{n-2}b^{n-2} + \cdots + p_1b + p_0 = P_Hb^s + P_L \\
Q &= q_{n-1}b^{n-1} + q_{n-2}b^{n-2} + \cdots + q_1b + q_0 = Q_Hb^s + Q_L \\
P \cdot Q &= (P_Hb^s + P_L) \cdot (Q_Hb^s + Q_L) = P_HQ_Hb^{2s} + (P_HQ_L + P_LQ_H)b^s + P_LQ_L
\end{aligned}
$$

Note that s must be chosen close to $(n - 1)/2$ in order to make the circuits simpler.

With the expressions above, four multipliers and the corresponding adders are required for obtaining P_HQ_H, P_HQ_L, P_LQ_H and P_LQ_L. In order to reduce the number of multipliers, although at the expense of increasing the number of adders, the product $P \cdot Q$ can be expressed as:

$$
P \cdot Q = P_HQ_Hb^{2s} + ((P_H + P_L) \cdot (Q_H + Q_L) - P_HQ_H - P_LQ_L)b^s + P_LQ_L
$$

(a)



x	y	P
0	0	0
0	1	0
1	0	0
1	1	1

(b)

x_1	x_0	y_1	y_0	m_3	m_2	m_1	m_0
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

$$m_3 = x_1 x_0 y_1 y_0$$

$$m_2 = x_1 y_1 (\overline{x}_0 + \overline{y}_0)$$

$$m_1 = x_0 y_1 (\overline{x}_1 + \overline{x}_0) + x_1 y_0 (\overline{x}_0 + \overline{y}_1)$$

$$m_0 = x_0 y_0$$

(c)

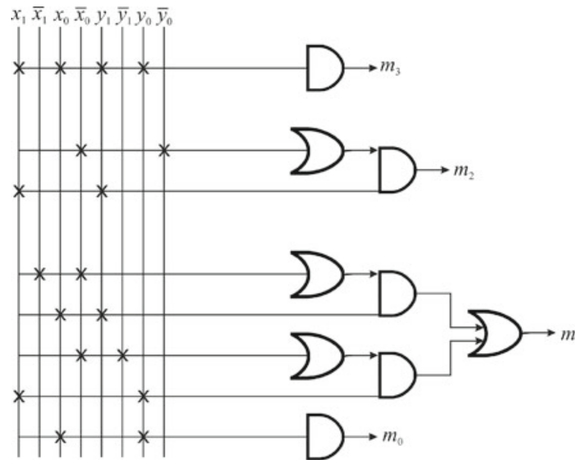


Fig. 2.7 **a** 1-bit multiplier. **b** Two-bits multiplying table. **c** Two-bits multiplier circuit. **d** *X* by *Y* multiplication. **e** Network for 2-bit character multiplying. **f** 2-bits multiplier

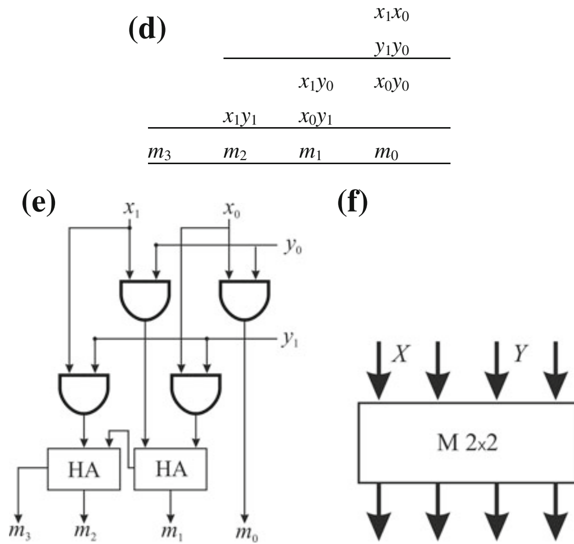


Fig. 2.7 continued

In this way, only three multipliers are required for getting $P_H Q_H$, $(P_H + P_L) \cdot (Q_H + Q_L)$ y $P_L Q_L$. Obviously, each one of the partial products can be computed using iteratively the same procedure for decomposing each operand into two or more chunks.

2.4.2 Sequential Multipliers

The designs above allow the multiplication of two unsigned binary numbers in only one clock cycle. It is possible that the resulting circuits are so much complex for the designer convenience or introduce excessive delay for responding into the clock cycle required for the general system performance. In this situation, pipelining of the circuits above or the design of the sequential circuit must be approached for providing simpler circuits, at the expense of more iterations for completing the multiplication operation. Let's consider the construction of a multiplier for n-bit unsigned binary numbers. If $X = x_{n-1} \ldots x_0$ is the multiplicand, being available in a parallel output register, $Y = y_{n-1} \ldots y_0$ is the multiplier, which is available in a shift register with serial output, and $R = r_{2n-1} \ldots r_0$, is the output which will be available in a $2n$-bit register (initialized to zero), we have the structure presented in Fig. 2.9a. With this circuit, the multiplication can be completed in n clock cycles (as many as bits in the multiplier operand), so that in each cycle, the partial sum corresponding to each multiplier bit is added to the previous result properly shifted. If the partial bit is '0', the corresponding partial sum will be zero, and when the multiplier bit is '1', the partial sum will be equal to the multiplicand. The operation can start from the most significant bit or from the less significant one,

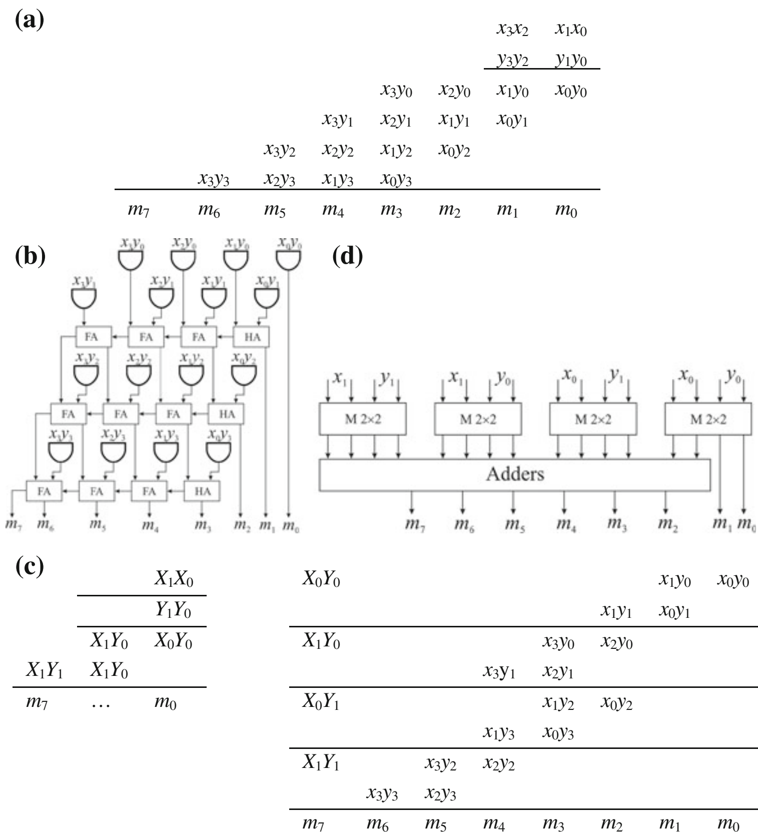


Fig. 2.8 **a** Four-bit characters multiplication. **b** Circuit for multiplication. **c** X by Y multiplication. **d** Network for multiplying X by Y

and in each case the previous result must be shifted in a different direction: to the left when starting from the most significant bit, and to the right when starting from the less significant one. As an example, when starting from the most significant bit of the multiplier, the multiplication algorithm will be:

Algorithm 2.1

First algorithm for sequential integer multiplication

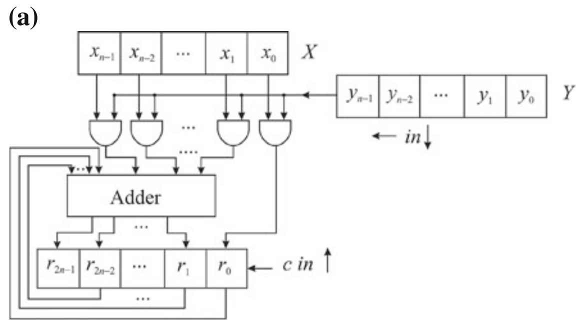
```
R ← 0, i ← 0
while i < n, do
    R ← ←R + y_{n-1-i} X, i ← i + 1;
end
```

End algorithm

where $\overleftarrow{R}$ is the previous content of R shifted one position to the left, and $y_{n-1-i}X$ is the current partial product.

Fig. 2.9 First serial-parallel multiplier: **a** Circuit. **b** Example ($X = 1011$, $Y = 1101$)

(a)



(b)

1) $y_3 = 1$	$\overleftarrow{R} \rightarrow$	00000000	
	$y_3x \rightarrow$	1011	
		00001011	$\rightarrow R$
2) $y_2 = 1$	$\overleftarrow{R} \rightarrow$	00010110	
	$y_2x \rightarrow$	1011	
		00100001	$\rightarrow R$
3) $y_1 = 0$	$\overleftarrow{R} \rightarrow$	01000010	
	$y_1x \rightarrow$	0000	
		01000010	$\rightarrow R$
4) $y_0 = 1$	$\overleftarrow{R} \rightarrow$	10000100	
	$y_0x \rightarrow$	1011	
		10001111	$\rightarrow R$

This algorithm can be implemented using the circuit of Fig. 2.9a. A latch, X, with parallel output, a shift register, Y, activated by falling edge, and a register R, with parallel output, parallel input and activated by rising edge, are used. In addition to these registers, n **AND** gates are used to generate the partial products. As the least significant bit of each partial product is directly the bit r_0 of the corresponding partial sum, this bit is stored directly in R and it is not an input of the adder. Thus, just an $(2n - 1)$-bit adder (usually one less than to be laid down for the final result), whose entries are, first, the n-product **AND** $x_{n-1}y_{n-1-i}$, $x_{n-2}y_{n-1-i,\ldots,}$ x_1y_{n-1-i} (see Fig. 2.9a), and secondly, the bits $r_{2n-2} \ldots r_0$ of the previous result. The $(2n - 1)$-bits of the adder output are stored in $r_{2n-1} \ldots r_1$. This allows the displacement to the left of the previous results. A counter modulo n would suffice to control the operation of this multiplier. As an example, the results that are generated in the four iterations that have to be executed by multiplying the 4-bit numbers $X = 1011$ by $Y = 1101$ are given in Fig. 2.9b.

If the multiplication starts by the least significant bit of the multiplier, the algorithm is as follows:

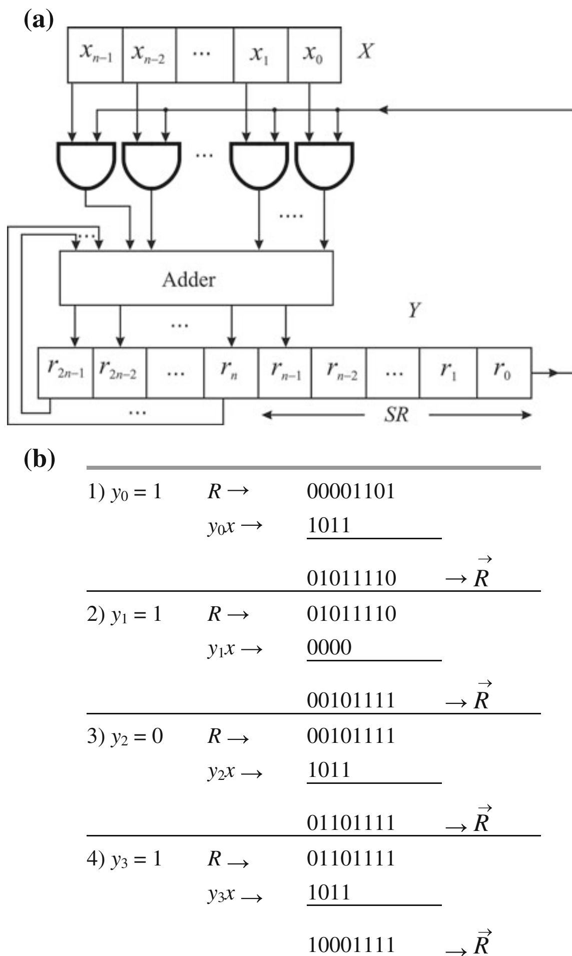


Fig. 2.10 Second serial-parallel multiplier: **a** Circuit. **b** Example ($X = 1011$, $Y = 1101$)

Algorithm 2.2

Second algorithm for sequential integer multiplication

```
R ← 0, i ← 0
while i < n, do
    R ← R⃗ + yᵢ X, i ← i + 1;
end
```

End algorithm

where $\vec{R}$ is the previous value of R shifted to the right, and y_iX is the present partial product.

This algorithm can be implemented using the circuit of Fig. 2.10a. A latch, X, with parallel output is used again for the multiplicand. However, the multiplier can be stored in the lower half of the register R, such that the most significant half of

R (n bits) forms a register with parallel output and parallel load, and the n-bit least significant of R form a shift register, Y. The register R is loaded or displaced in the falling edge of each clock pulse. In order to generate the partial products, n **AND** gates are used and an n-bits (as many as bits in the multiplicand) adder, whose inputs are, first, the bits corresponding to the partial product in each iteration, $x_{n-1}y_i$, …, x_0y_i, and otherwise, the r_{2n-1} …r_n bits of the previous result (see Fig. 2.10a). The $n + 1$ output bits of the adder are stored in r_{2n-1} …r_{n-1} (recall that r_{n-1} is the serial input of the shift register and, in each iteration there is a shift to the right of Y). With all this, the shift of the previous results is achieved. Again, to control the operation of the multiplier a module n counter is sufficed. As an example, the results generated in the four iterations by multiplying the 4-bit numbers, $X = 1011$ by $Y = 1101$, are given in Fig. 2.10b.

The circuits with the structures of Figs. 2.9a and 2.10a can be called **serial-parallel multiplier** due to the multiplier is serial data and the multiplicand is parallel data. A simpler but more expensive solution in terms of calculation time, would be the **serial-serial multiplier**, where in each iteration one bit of the multiplicand and one of the multiplier would be multiplied; it is left as an exercise.

In each iteration of the serial-parallel multiplier, a multiplier bit and the multiplicand, M, are multiplied. This circuit can be transformed into another allowing that M could be multiplied by more than one bit of the multiplier in each iteration. For example, multiplying in each iteration by two bits of the multiplier, for an n-bit multiplier, the multiplication would be available in $n/2$ iterations. Again, the design of these circuits is left as an exercise.

2.4.3 Multiplying by a Constant

The multiplication of a set of data for one or more constants is an operation that must be performed frequently. Of course, any multiplier can be used for this purpose, as described previously. However, in this case, when one of the operands is constant, simpler circuits can be designed for this specific purpose. For example, let suppose the case of a circuit for multiplying any unsigned 8-bit binary number, $X = x_7… x_0$, by 25. Given that $25_{10} = 11001_2$, to multiply by 25 is equivalent to adding the three terms given in Fig. 2.11a. Thus, using two 8-bit parallel adders, this multiplication can be implemented, as shown in the same Fig. 2.11a, generating a 13-bit result, $R = r_{12}… r_0$. Compared to a generic multiplier circuit, the reduction to be achieved with this specific circuit is evident. This idea of using parallel adders will be called **solution 1** for multiplying by a constant.

In general, both adders and subtractors can be used for the decomposition of the multiplier M. This is equivalent to use signed digits in the decomposition of M, and from the minimal representation of M a simple multiplier circuit may be obtained.

If full adders and half adders are used as building blocks, the circuit for multiplying by 25 can be reduced more. Specifically just 11 full adders and 5 half adders are required, as shown in Fig. 2.11b. This is the **solution 2** for multiplying

					x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
		x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0			
	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0				
r_{12}	r_{11}	r_{10}	r_9	r_8	r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0

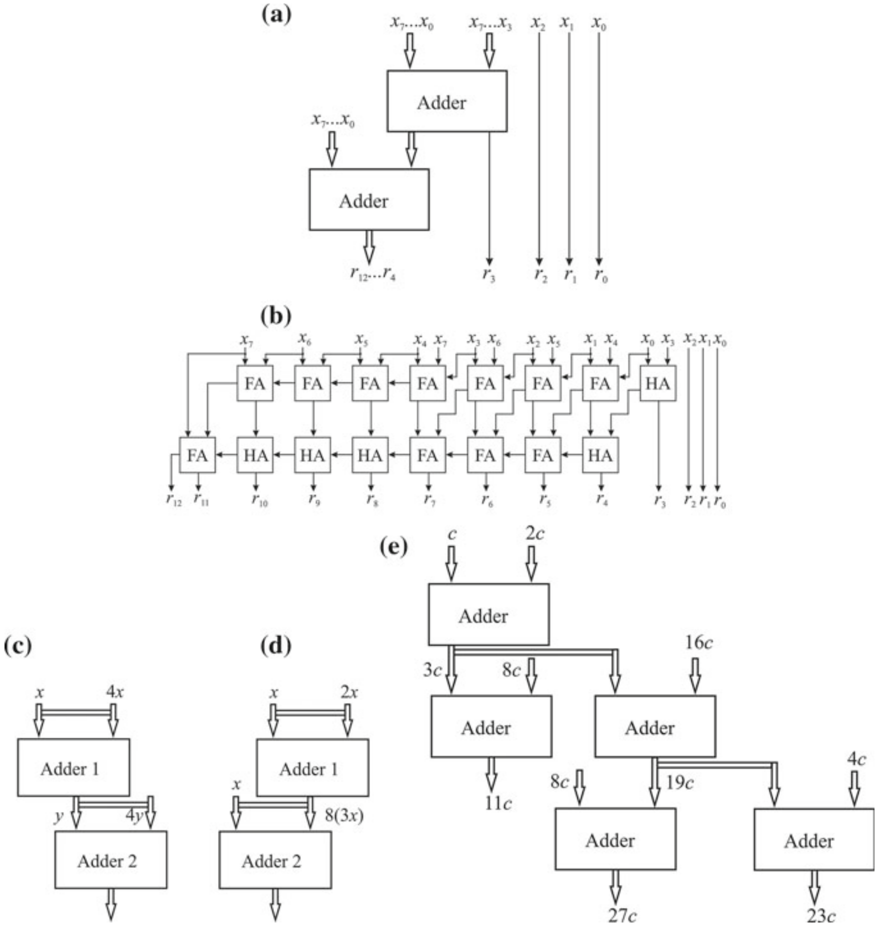


Fig. 2.11 Multiplying by 25: **a** First solution. **b** Second solution. **c** Third solution. **d** Other implementation for the third solution. **e** Multiplying by 11, 19, 23 and 27

by a constant. Solution 2, when considering the design at a lower level, usually produces simpler circuits than **solution 1**.

Another way to build specific multiplier when the multiplier is a constant, M, using adders digits (of adequate size in each situation), consists in decomposing the constant M in factors, which in turn are decomposed into sums of powers of two. For example, for $M = 25$, it results:

$$25 \cdot X = (4 + 1) \cdot (4 + 1) \cdot X = (4 + 1) \cdot (4 \cdot X + X) = 4 \cdot Y + Y$$

where $Y = 4 \cdot X + X$. Therefore, the multiplication is performed using two adders of appropriate length, as shown in Fig. 2.11c. Multiplication by a power of two is reduced to a displacement, which does not require circuitry. If X is of n bits, the adder 1 of Fig. 2.11c must be an n-bit adder, and the adder 2 an $(2n + 1)$-bit adder. This solution to multiply by a constant is called **solution 3**.

It is also possible to use that $25 = 3 \times 8 + 1$, and again this multiplication can be implemented using two adders, resulting in the circuit of Fig. 2.11d.

Developments that can be used to multiply by a constant, up to 100, are given in Table 2.1. The powers of 2 do not need one adder (obviously not included in the table); in this table, one adder/subtractor is enough for 31 constants; for 54 constants two adders/subtractors are needed; only for 8 constants three constants adder/subtractors are required.

Two factors products are only used in Table 2.1, since only reaches 100. Obviously, products with more than two factors can be used, which may make sense for constants greater than those shown in Table 2.1. For example, $5049_{10} = 9 \times 17 \times 33$ and, according with Table 2.1, it can be implemented with three adders, since each factor only needs one adder; using signed digit $5049_{10} = 101000\bar{1}00\bar{1}001_2$, and four adders/subtractors are required to build the multiplier.

Another decomposition of multiplier M of interest to explore consists of finding dividers on the form $2^i \pm 2^j$, which in some cases can lead to simpler circuits. For example, the case of multiply $17 \times 41 = 697 = (16 + 1)\ (1 + 8 + 32) = (16 + 1) + 8\ (16 + 1) + 32(16 + 1)$. With this decomposition, the multiplication can be done with three adders, while starting form the development $697 = 101100\bar{1}001$ four adders/subtractors are required.

When the same data have to be multiplied by multiple constants, it is possible to organize the process so that different partial products can be shared in the different calculations, as can be seen in the following example.

Example 2.1 **Let suppose the case of multiply simultaneously by 11, 19, 23 and 27.**

Developing these constants as follows $11 = 8 + 3$, $19 = 16 + 3$, $23 = 19 + 4$, $27 = 19 + 8$ or $27 = 11 + 16$, the multiplier can be made using five adders, sharing intermediate results, as it is depicted in Fig. 2.11e. □

2.5 Exponentiation

To raise a number N to a power P (P integer) consists on multiplying the number N by itself P times. Therefore, with an appropriate multiplier, any integer number N can be raised to any power. First the calculation of the square of N is considered, where N is an unsigned integer in base 2. As illustrative example, let suppose a 8-bit number, $N = x_7x_6x_5x_4x_3x_2x_1x_0$. The $N \times N$ multiplication is shown in Fig. 2.12a, where it has been applied that $x_ix_i = x_i$. Moreover, when a column has

Table 2.1 Multipliers 1–100

N°	N° A/S	Develop	N°	N° A/S	Develop
3	1	2 + 1	53	3	32 + 16 + 4 + 1
5	1	4 + 1	54	2	6 × 9; 64 − 8 − 2
6	1	4 + 2	55	2	64 − 8 − 1
7	1	8 − 1	56	1	64 − 8
9	1	8 + 1	57	2	64 − 8 + 1
10	1	8 + 2	58	2	64 − 8 + 2
11	2	8 + 2 + 1	59	2	64 − 4 − 1
12	1	8 + 4	60	1	64 − 4
13	2	8 + 4 + 1	61	2	64 − 4 + 1
14	1	16 − 2	62	1	64 − 2
15	1	16 − 1	63	1	64 − 1
17	1	16 + 1	65	1	64 + 1
18	1	16 + 2	66	1	64 + 2
19	2	16 + 2 + 1	67	2	64 + 2 + 1
20	1	16 + 4	68	1	64 + 4
21	2	16 + 4 + 1	69	2	64 + 4 + 1
22	2	16 + 4 + 2	70	2	64 + 4 + 2
23	2	16 + 8 − 1	71	2	64 + 8 − 1
24	1	16 + 8	72	1	64 + 8
25	2	16 + 8 + 1; 5 × 5	73	2	64 + 8 + 1
26	2	16 + 8 + 2	74	2	64 + 8 + 2
27	2	3 × 9; 32 − 4 − 1	75	2	15 × 5
28	1	32 − 4	76	2	64 + 8 + 4
29	2	32 − 4 + 1	77	3	64 + 8 + 4 + 1
30	1	32 − 2	78	2	5 × 16 − 2
31	1	32 − 1	79	2	5 × 16 − 1
33	1	32 + 1	80	1	5 × 16
34	1	32 + 2	81	2	5 × 16 + 1; 9 × 9
35	2	5 × 7; 32 + 2 + 1	82	2	5 × 16 + 2
36	1	32 + 4	83	3	64 + 16 + 2 + 1
37	2	32 + 4 + 1	84	2	5 × 16 + 4
38	2	32 + 4 + 2	85	2	17 × 5
39	2	32 + 8 − 1	86	3	64 + 16 + 4 + 2
40	1	5 × 8; 32 + 8	87	3	64 + 16 + 8 − 1; 3 × 29
41	2	32 + 8 + 1	88	2	8 × 11; 5 × 16 + 8; 3 × 32 − 8
42	2	32 + 8 + 2	89	3	64 + 16 + 8 + 1
43	3	32 + 8 + 2 + 1	90	2	3 × 30; 5 × 18
44	2	32 + 8 + 4	91	3	64 + 32 − 4 − 1
45	2	5 × 9	92	2	3 × 32 − 4
46	2	32 + 16 − 2	93	2	3 × 31
47	2	32 + 16 − 1	94	2	3 × 32 − 2

(continued)

Table 2.1 (continued)

N°	N° A/S	Develop	N°	N° A/S	Develop
48	1	32 + 16	95	2	3 × 32 − 1
49	2	32 + 16 + 1	96	1	3 × 32
50	2	32 + 16 + 2	97	2	3 × 32 + 1
51	2	3 × 17	98	2	3 × 32 + 2
52	2	32 + 16 + 4	99	2	3 × 33
			100	2	3 × 32 + 4; 5 × 20

$x_ix_j + x_jx_i = 2x_ix_j$ obviously it can be moved to the next column as x_ix_j. Also, when a column is $x_i + x_ix_j$:

$$x_i + x_ix_j = x_i\left(x_j + \bar{x}_j\right) + x_ix_j = 2x_ix_j + x_i\bar{x}_j$$

and $2x_ix_j$ can be moved to the next column as x_ix_j. Considering all these replacements, the summands to be used for calculating the square can remain as in Fig. 2.12a.

With respect to the implementation of the different products, the products expressed as $x_{j+1}\bar{x}_j$ and $x_{j+1}x_j$ that appear in adjacent columns (highlighted in Fig. 2.12a) can be synthesized simultaneously with a demultiplexer, such as shown in Fig. 2.12b. A possible implementation of the squaring circuit for 8-bit integers is given in Fig. 2.12c, using 7 demultiplexers, 21 **AND** gates, 7 half adders and 20 full adders. The **AND** gates are shown in Fig. 2.12c with a circle that includes the sub indexes of the two input gate.

Obviously, for an integer N with any number of bits, a combinational circuit for squaring can be designed as was done for eight bits.

If it is useful in some situation, the product of two numbers can be calculated using addition, subtraction and square, from the following expression:

$$XY = \frac{1}{4}\left\{(X+Y)^2 - (X-Y)^2\right\}$$

2.5.1 Binary Methods

To raise N to any other integer power, P, square and multiplier circuits can be used. To obtain a starting expression suitable, P is developed as a binary number:

$$P = p_{m-1}2^{m-1} + p_{m-2}2^{m-2} + \cdots + p_1 2 + p_0 \quad (2.1)$$

$$= ((\ldots(p_{m-1}2 + p_{m-2})2 + \cdots)2 + p_1)2 + p_0 \quad (2.2)$$

Thus, using the development (2.2) it results:

$$N^P = (\ldots((N^{p_{m-1}})^2 \cdot N^{p_{m-2}})^2 \cdot \cdots \cdot N^{p_1})^2 \cdot N^{p_0}$$

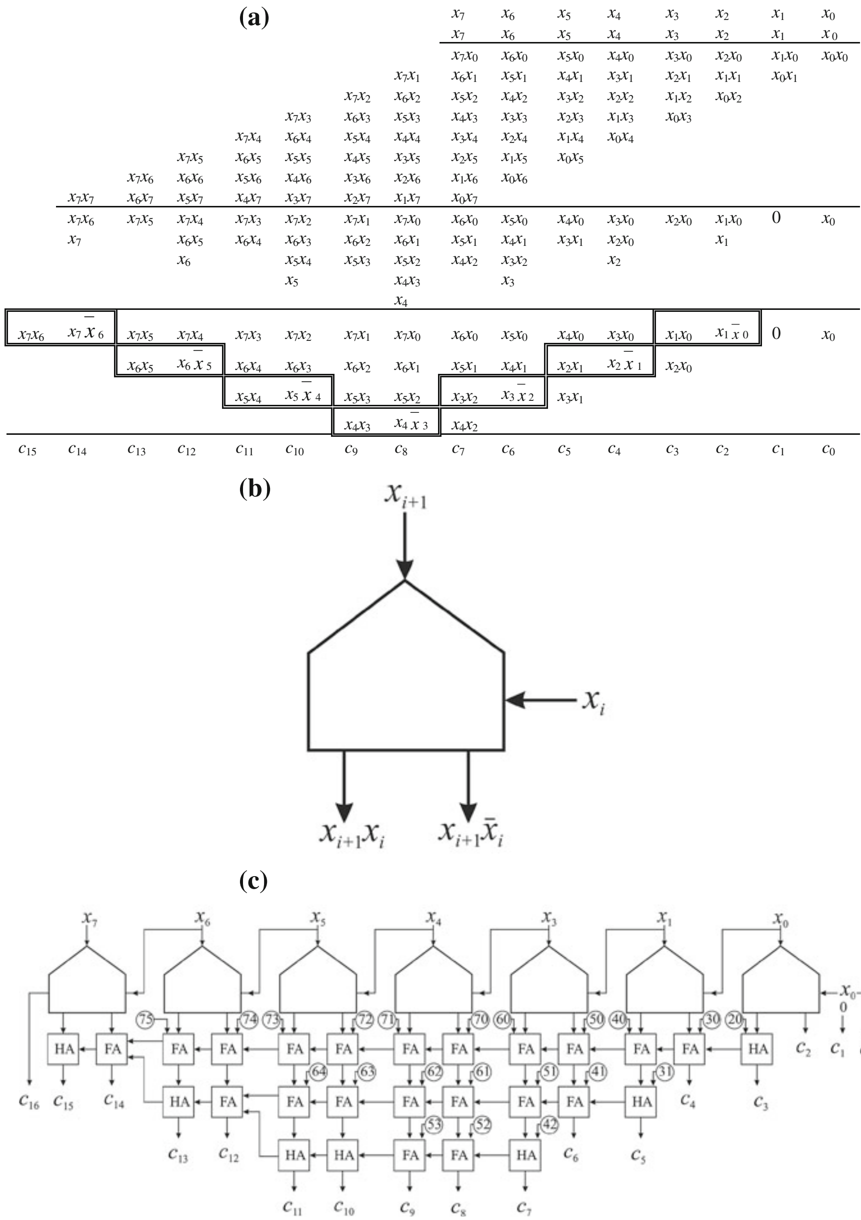


Fig. 2.12 **a** Square. **b** Demultiplexer. **c** Circuit for squaring

With this development for N^P, the calculation involves squaring and multiplication, iteratively. The core of the calculation would be:

```
1) R ← R²;
2) if pi=1, R ← RN;
```

The result remains in the register R, which initially must be $R \leftarrow 1$. The complete algorithm could be as follows:

Algorithm 2.3

First algorithm for exponentiation

```
R ← 1, i ← m-1;
while i > 0, do
  begin
    1) if pi=1 then R ← RN;
    2) R ← R², i ← i - 1;
  end
if p0=1 then R ← RN;
```

End algorithm

The bits p_i of the binary development of P are processed in this algorithm starting with the most significant one, hence, this method is generally known as *binary method from left to right*.

A possible processing unit for exponentiation using the above algorithm (the control signals are not included) is represented in Fig. 2.13a. This circuit includes a register R, a multiplier and a squarer.

Other development of N^P using (2.1) is the following:

$$N^P = \left(N^{2^0}\right)^{p_0} \cdot \left(N^{2^1}\right)^{p_1} \cdot \cdots \cdot \left(N^{2^{m-2}}\right)^{p_{m-2}} \cdot \left(N^{2^{m-1}}\right)^{p_{m-1}} \tag{2.3}$$

From this development another algorithm for the exponentiation can be designed. Again, the calculation consists on to square and to multiply, according to the following core:

```
1) N ← N²;
2) if pm-1-i=1, R ← RN;
```

Initially $R \leftarrow 1$. The result remains in R. The algorithm for this case could be the following one:

Algorithm 2.4

Second algorithm for exponentiation

```
R ← 1, i ← m-1;
while i > 0, do
  begin
    1) if pm-1-i=1 then R ← RN;
    2) N ← N², i ← i - 1;
  end
```

End algorithm

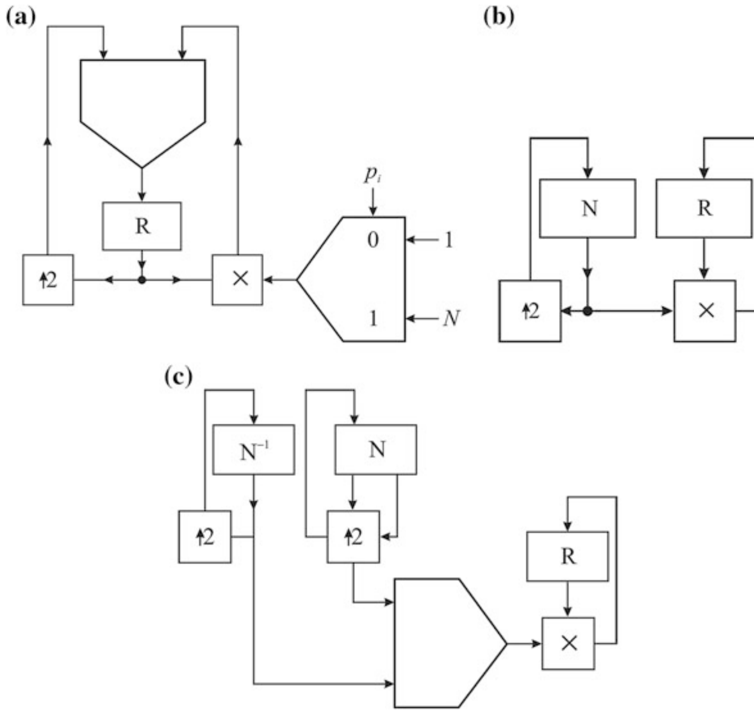


Fig. 2.13 Exponentiation: **a** First solution. **b** Second solution. **c** Exponentiation using the canonic development

The bits p_i of the binary development of P are processed in this algorithm starting with the least significant one, hence, this method is generally known as *binary method from right to left*.

A possible circuit for exponentiation using the above algorithm (the control signals are not included) is represented in Fig. 2.13b. This circuit includes two registers, a multiplier and a squarer.

When operating in the context of certain algebraic structures can be arranged and easily operate with both N and N^{-1}. If this is the case, a canonical development can be used for the exponent P, in which they appear, in general, both +1 and −1 (that is, both positive and negative exponents), but the number of operations will be in average, smaller. The core of the calculation, using a development similar to (2.3) would be in this case:

```
1) N ← N²; N⁻¹ ← N⁻²;
2) if p_{m-1-i}=1, then R ← RN;
   else if p_{m-1-i}=-1, then R ← RN⁻¹;
```

The algorithm in this case may be that which is given below, the corresponding circuit could be the one in Fig. 2.13c.

Algorithm 2.5
Third algorithm for exponentation

```
R ← 1, i ← m−1;
while i > 0, do
  begin
    1)  if p_{m−1−i}=1 then R ← RN;
        else if p_{m−1−i}=−1, then R ← RN^{-1};
    2) N ← N^2, N^{-1} ← N^{-2}, i ← i − 1;
  end
```

End algorithm

If the exponent P can be factorized, $P = Q \cdot R$, then the exponentiation can be decomposed into two phases:

$$N^P = N^{Q \cdot R} = \left(N^Q\right)^R$$

If the exponent P is developed using any base, b:

$$\begin{aligned} P &= p_{m-1}b^{m-1} + p_{m-2}b^{m-2} + \cdots + p_1 b + p_0 \\ &= ((\ldots(p_{m-1}b + p_{m-2})b + \cdots)b + p_1)b + p_0 \end{aligned}$$

the binary method, both from left to right and from right to left, can be extended to the base b, with appropriate modifications. In the algorithm in base b from left to right it must be calculated:

$$N^P = \left(\ldots\left(\left(N^{p_m-1}\right)^b \cdot N^{p_m-2}\right)^b \cdot \cdots \cdot N^{p_1}\right)^b \cdot N^{p_0}$$

Since the coefficients p_j are not only 0 or 1, it is necessary to multiply by N^j ($j = 1, 2, \ldots, b - 1$) and to raise to the power b.

In the algorithm in base b from right to left it must be calculated:

$$N^P = \left(N^{b^0}\right)^{p_0} \cdot \left(N^{b^1}\right)^{p_1} \cdot \cdots \cdot \left(N^{b^{m-2}}\right)^{p_{m-2}} \cdot \left(N^{b^{m-1}}\right)^{p_{m-1}}$$

Thus, it is required to raise to the powers 1, 2, …, b.

2.5.2 Additive Chains

The developments (2.1) and (2.2) transform the exponent P into an addition and, applying that the exponents are additive, the binary developments emerge. This same idea is used for additive chains.

Given P a positive integer, an additive chain for P is a sequence of integers, p_0, $p_1, \ldots, p_n$, such that $p_0 = 1$, $p_n = P$, $p_i = p_j + p_k$, $i > j \geq k$, $p_i \neq p_j$ for

$i \neq j$. Then, the two first elements of each additive chain are always 1 and 2; the third element can be only 3 or 4, and the remainder elements are obtained adding two previous elements (that can be a previous repeated element).

A particular case of additive chains are known as Brauer chains [Bra39]; for these additive chains $p_i = p_{i-1} + p_k$, $i - 1 \geq k$. Thus, using a Brauer chain, to obtain the next element of the chain, the present element is used in the involved addition. For implementation purposes, it is obvious that using always the previous result is very interesting. A procedure for constructing this type of additive chains, which are the most used, is described in [Bra39]. This procedure is not the only possibility. To apply this method an integer e is chosen and P is developed in base $b = 2^e$:

$$P = p_{m-1}b^{m-1} + p_{m-2}b^{m-2} + \cdots + p_1 b + p_0$$

The following Brauer additive chain can be constructed for P, being composed by m sections that have to be adequately linked. The first section is $\{1, 2, 3, \ldots, 2^e - 1\}$; the second section is $\{2p_{m-1}, 4p_{m-1}, 8p_{m-1}, \ldots, bp_{m-1}\,(bp_{m-1} + p_{m-2})\}$; the third section is $\{2(bp_{m-1} + p_{m-2}), 4(bp_{m-1} + p_{m-2}), 8(bp_{m-1} + p_{m-2}), \ldots, b(bp_{m-1} + p_{m-2}), b(bp_{m-1} + p_{m-2}) + p_{m-3}\}$; …; the last section is $\{2(b(\ldots(bp_{m-1} + p_{m-2}) \cdots + p_1), 4(b(\ldots(bp_{m-1} + p_{m-2}) \cdots + p_1), 8(b(\ldots(bp_{m-1} + p_{m-2}) \cdots + p_1), \ldots, b(b(\ldots(bp_{m-1} + p_{m-2}) \cdots + p_1), b(b(\ldots(bp_{m-1} + p_{m-2}) \cdots + p_1) + p_0\}$, as it is done in the following example.

Example 2.2 **Obtain Brauer additive chains for 26221, with e = 1, 2, 3, 4 and 5.**

Choosing $e = 1$ ($b = 2^e = 2$) it is:

$$26221 = 2^{14} + 2^{13} + 2^{10} + 2^9 + 2^6 + 2^5 + 2^3 + 2^2 + 1$$

Thus, $p_{14} = 1$, $p_{13} = 1$, $p_{12} = 0$, $p_{11} = 0$, $p_{10} = 1$, $p_9 = 1$, $p_8 = 0$, $p_7 = 0$, $p_6 = 1$, $p_5 = 1$, $p_4 = 0$, $p_3 = 1$, $p_2 = 1$, $p_1 = 0$, $p_0 = 1$. The first section of the Brauer chain is 1; the second section is 2 and 3; the third section is 6; the fourth section is 12; the fifth section is 24, 25; the sixth section is 50, 51; the seventh section is 102, the eighth section is 204; the ninth section is 408, 409; the tenth section is 818, 819; the eleventh section is 1638, the twelfth section is 3276, 3277; the thirteenth section is 6554, 6555; the fourteenth section is 13110; and the fifteenth section is 26220, 26221. Thus the additive chain is formed by 23 elements {1, 2, 3, 6, 12, 24, 25, 50, 51, 102, 204, 408, 409, 818, 819, 1638, 3276, 3277, 6554, 6555, 13110, 26220, 26221}.

Choosing $e = 2$ ($b = 2^e = 4$) it results:

$$26221 = 1 \times 4^7 + 2 \times 4^6 + 1 \times 4^5 + 2 \times 4^4 + 1 \times 4^3 + 2 \times 4^2 + 3 \times 4 + 1$$

Therefore $p_7 = 1$, $p_6 = 2$, $p_5 = 1$, $p_4 = 2$, $p_3 = 1$, $p_2 = 2$, $p_1 = 3$, $p_0 = 1$. The first section of the chain Brauer is 1, 2, 3; the second section is 2, 4, 6; the third section is 12, 24, 25; the fourth section is 50, 100, 102; the fifth section is 204, 408, 409; the sixth section is 818, 1636, 1638; the seventh section is 3276,

6552, 6555; and the eighth section is 13110, 26220, 26221. In the second section the 2 should be removed, which is already in the first section; the 4 can also be removed, since it is not needed to build the following elements. Thus the additive chain has 22 elements: {1, 2, 3, 6, 12, 24, 25, 50, 100, 102, 204, 408, 409, 818, 1636, 1638, 3276, 6552, 6555, 13110, 26220, 26221}.

Choosing $e = 3$ ($b = 2^e = 8$) it is:

$$26221 = 6 \times 8^4 + 3 \times 8^3 + 8^2 + 5 \times 8 + 5$$

Therefore $p_4 = 6$, $p_3 = 3$, $p_2 = 1$, $p_1 = 5$, $p_0 = 5$. The first section of the Brauer chain is 1, 2, 3, 4, 5, 6, 7; the second section is 12, 24, 48, 51; the third section is 102, 204, 408, 409; the fourth section is 818, 1636, 3272, 3277; the fifth section is 6554, 13108, 26216, 26221. From the first section 4 and 7 can be suppressed, which are not used lately. Thus the additive chain is 21 elements {1, 2, 3, 5, 6, 12, 24, 48, 51, 102, 204, 408, 409, 818, 1636, 3272, 3277, 6554, 13108, 26216, 26221}.

Choosing $e = 4$ ($b = 2e = 16$) it results:

$$26221 = 6 \times 16^3 + 6 \times 16^2 + 6 \times 16 + 13$$

Thus, $p_3 = 6, p_2 = 6, p_1 = 6, p_0 = 13$. The first section of the Brauer chain is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15; the second section is 12, 24, 48, 96, 102; the third section is 204, 408, 816, 1632, 1638; and the fourth section is 3276, 6552, 13104, 26208, 26221. From the first section 3, 5, 7, 8, 9, 10, 11, 14 and 15 can be suppressed, which are not subsequently used, although 13 is used later, can also be delete building it like 12 + 1, thereby facilitating the link of the first section to the second; making this, the fourth section would be 3276, 6552, 13104, 26208, 26220, 26221. In this way the additive chain is 20 elements {1, 2, 4, 6, 12, 24, 48, 96, 102, 204, 408, 816, 1632, 1638, 3276, 6552, 13104, 26208, 26220, 26221}.

Choosing $e = 5$ ($b = 2^e = 32$) it results:

$$26221 = 25 \times 32^2 + 19 \times 32 + 13$$

Therefore $p_2 = 25, p_1 = 19, p_0 = 13$. The first section of the Brauer chain is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31; the second section is 50, 100, 200, 400, 800, 819, and the third section is 1638, 3276, 6552, 13104, 26208, 26221. From the first section 3, 5, 7, 8, 9, 10, 11, 14, 15, 16, 17, 18, 20, 21, 22, 23, 24, 26, 27, 28, 29, 30 and 31 can be suppressed, since they are not subsequently used. In this way the additive is 20 elements string {1, 2, 4, 6, 12, 13, 19, 25, 50, 100, 200, 400, 800, 819, 1638, 3276, 6552, 13104, 26208, 26221}.

Using an additive chain with P elements, $P - 1$ operations must be performed (multiplications and squares) to calculate the corresponding power, since the element 1 does not involve any operation. Thus, with a chain of 20 elements, 19 operations are required.

Considering $26221 = 2017 \times 13$, 26221 can be expressed as the concatenation of two additive chains, corresponding to 2017 and to 13. For 2017 the additive Brauer chain with 15 elements can be constructed {1, 2, 3, 6, 12, 24, 31, 62, 124, 248, 252, 504, 1008, 2016, 2017}, for 13 the Brauer additive chain with 6 elements can be constructed {1, 2, 3, 6, 12, 13}. Using these two additive chains, 19 operations are required again to calculate the corresponding power, as the initial 1 of each chain does not involve any operation. □

From Example 2.2 it is clear that a Brauer chain with $e = 1$ corresponds to the binary method from left to right described above. It can be considered that a Brauer chain with $e > 1$ is only a generalization of the binary method from left to right; in fact the exponentiation method using a Brauer chain is also known as *the method* 2^e *from left to right*.

There is no available algorithm that guarantees the shortest chain.

The same circuit proposed for the binary method from left to right (Fig. 2.13), with minor modifications, can be used to calculate any power using additive chains.

2.6 Division and Square Root

This section will consider simple circuits to divide integers and to extract the integer square root of integer numbers.

2.6.1 Combinational Divisors

Let consider two unsigned binary numbers: D (m-bit dividend), and d (n-bit divisor). In this case, the division consists on finding two unsigned binary numbers, C (quotient) and r (remainder), $r < d$, such that:

$$D = C \cdot d + r$$

The division is defined for $d \neq 0$. Therefore, in what follows it is assumed that this condition is met, i.e., before dividing, it is checked if $d \neq 0$, and only in this affirmative case, the division is performed.

With the condition $r < d$, C and r are unique, and to calculate them, an immediate combinational solution might be thought obviously. To implement the division it is sufficed, for example, a **ROM** of $m + n$ address bits and m outputs (2^{m+n} words of m bits), in which the quotient and the remainder corresponding to every possible (D, d) are written. For real cases this is not a feasible combinational solution since $m + n$ will result almost always too large to directly synthesize the corresponding functions (for example, to include all possible outputs in a **ROM**).

Another combinational solution is possible that attempts to mimic the division algorithm as a series of subtractions and shifts. Before addressing this alternative,

more aspects about the operands have to be established. Specifically it is assumed that, as usual, the relationship between the lengths of the dividend and the divisor is $m = 2n - 1$, and that the most significant bit of the divisor, d, is 1. Neither of these assumptions implies restriction, because on the one hand, the size of the operand can always be adjusted by adding zeros, and, second, by shifting the divisor it is possible to make 1 the most significant bit; after division, the shifts made in the divisor must be properly transferred to the quotient and the remainder to obtain correct results. With these assumptions, with n-bits for both the quotient and to the remainder, all possible results may be represented, and the division is made in n-steps, in each of which a bit of the quotient is obtained.

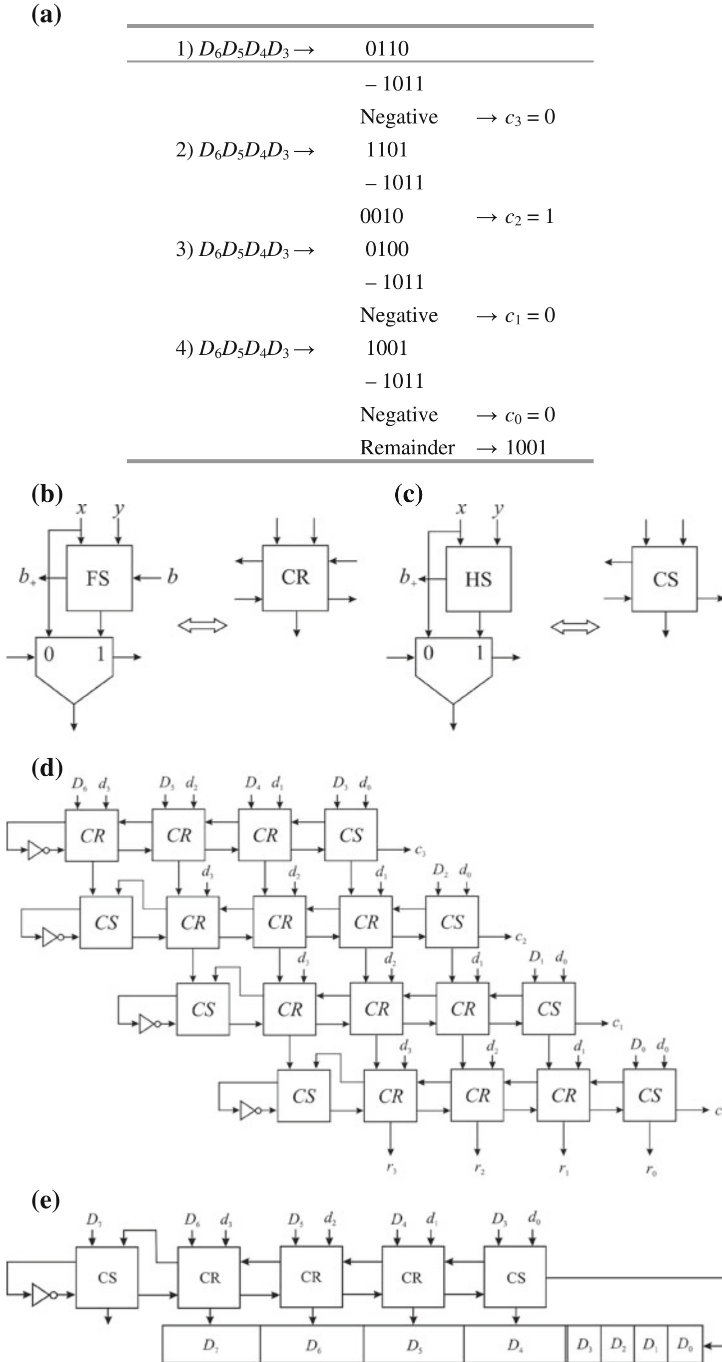
In what follows, an example will be used to reach a combinational divider circuit: let $n = 4$, $D = 0110101$, $d = 1011$. The four stages of calculation for this case are detailed in Fig. 2.14a. In the first stage d is subtracted from the four most significant bits of D ($D_6D_5D_4D_3$); if the result is positive (and therefore no output borrow), the quotient bit is 1, and the difference $D_6D_5D_4D_3 - d$ passes to the next stage as the most significant bits of the modified dividend. If the result is negative (i.e., there is output borrow), the quotient bit is 0, and the dividend unchanged passes to the next stage. In other words, the quotient bit is the complement of the borrow of the subtractor output, and if the quotient bit is 0, D without changing is selected for the next stage, while if the quotient bit is 1, the most significant bits of the dividend bit must be modified selecting $D_6D_5D_4D_3 - d$. Therefore, with a full subtractor, **FS**, to take into account the possible borrow of the previous bit, plus one 2-to-1 multiplexer, the circuit necessary for processing each bit can be constructed, as shown with cell **CR** of Fig. 2.14b. If for a given bit (as with the least significant bit) no input borrows are to be considered, the full subtractor **FS** can be replaced by a half subtractor, **HS**, resulting in the **CS** cell, simpler than the **CR**, Fig. 2.14c.

The second and subsequent iterations consist on repeating the same as the first iteration, using in each case the unmodified or modified dividend which has resulted in the previous iteration. Then, by subtracting, the divisor is shifted one position to the right in each iteration. The remainder, $r_3 \ldots r_0$, is obtained in the fourth iteration.

The circuit for dividing a number of seven bits by other of four bits is detailed in Fig. 2.14d, in which 12 **CR** cells and 7 **CS** cells are used (or 19 **CR** cells, if only one single type of cells want to be used). As it has been already indicated, the divisor has to be adjusted to get that always the most significant bit is a 1, and after division, these movements have to be translated to the results. It is straightforward to extend these design divisors for any value of n.

2.6.2 Sequential Divisors

The most common divisors are the sequential. The ideas that led to the divisor of Fig. 2.14d can be used to construct a divisor that divides D, of $2n - 1$ bits, by d, of n bits, using n clock pulses. As a particular case it is still assumed that $n = 4$.



(a)

1) $D_6D_5D_4D_3 \rightarrow$	0110	
	– 1011	
	Negative	$\rightarrow c_3 = 0$
2) $D_6D_5D_4D_3 \rightarrow$	1101	
	– 1011	
	0010	$\rightarrow c_2 = 1$
3) $D_6D_5D_4D_3 \rightarrow$	0100	
	– 1011	
	Negative	$\rightarrow c_1 = 0$
4) $D_6D_5D_4D_3 \rightarrow$	1001	
	– 1011	
	Negative	$\rightarrow c_0 = 0$
	Remainder	$\rightarrow$ 1001

Fig. 2.14 **a** Division example; **b** CR cell; **c** CS cell; **d** Combinational divisor of 7 by 4 unsigned bits. **e** Sequential divisor of 7 by 4 unsigned bits

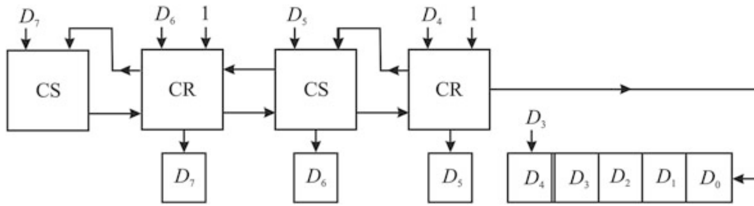


Fig. 2.15 Sequential divisor by 10_{10}

Figure 2.14e shows a circuit using three **CR** cells, two **CS** cells, one 4-bit latch to store the divisor, d (this register it is not shown in Fig. 2.14e), and an 8-bit register for the dividend, D. This register D consists of two 4-bit register: the first register ($D_7D_6D_5D_4$) must be simultaneous reading and writing (i.e., master-slave), the second ($D_3D_2D_1D_0$) must be a shift register with serial input and output. The registers d and D have to be loaded with the data to be processed before starting operation. Obviously always $D_7 = 0$ before starting to divide, and the divisor will have been shifted so the most significant bit of d is 1. The shift register ($D_3D_2D_1D_0$) is used to store the bits of the quotient. It is easily verified that this circuit in Fig. 2.14e does the same as the one in Fig. 2.14d, unless using four clock pulses. Therefore, after four iterations, the quotient is stored in $D_3D_2D_1D_0$ and the remainder of the division is stored in $D_7D_6D_5D_4$. Again, this circuit can be extended immediately to any value of n.

2.6.3 Dividing by a Constant

In some applications, as in the scaling or in the change of base or in the modular reduction, it is necessary to divide a data set by the same constant. For this purpose different specific circuits can be used. In what follows it is assumed that integer data are going to be divided by an integer constant.

It is easy to see that, strictly, just dividers to divide by odd numbers have to be designed. Indeed, the division of a number of m bits by 2^n is simplified to n shifts: the n least significant bits are the remainder of the division, and the $m - n$ most significant bits are the quotient. Moreover, any even number can be decomposed into the product of an odd number by a power of two:

$$C = I \times 2^n \Rightarrow \frac{N}{C} = \frac{N}{I}\frac{1}{2^n}$$

A first solution to design a divider by a constant, even or odd, consists of particularizing the generic circuits of Fig. 2.14d, e for the divisor to be used. For example, the sequential divisor of the Fig. 2.14e is shown in Fig. 2.15; this sequential divisor is particularized to divide by 10 ($10_{10} = 1010_2$) any unsigned 7-bit integer data. Obviously, the same result is obtained when dividing by 5, and

then by 2. In any case, these particularized circuits provide both the quotient and the remainder of the division.

The next three sections are also devoted to the implementation of the division by a constant, but considering those cases in which only one of the results is of interest: the quotient or the remainder.

2.6.4 Modular Reduction

In some cases only one of the two results of the division is of interest. If only the remainder is of interest, it is a modular reduction, as shown in Sect. 1.2.4. After obtaining the remainder, $R = N\mathbf{mod}C$, the difference $N - R$ is a multiple of C. The following example shows a case study of modular reduction based on calculating the remainder corresponding to the different powers of the base, as developed in Sect. 1.2.4.

Example 2.3 **Let suppose the case of calculating the remainder resulting from dividing by 5 any 8-bit unsigned binary numbers. Let suppose $N = ABCDEFGH$ is the 8-bit binary number to be processed. It results:**

$$\begin{aligned} N\mathbf{mod}5 &= \left(A2^7 + B2^6 + C2^5 + D2^4 + E2^3 + F2^2 + G2^1 + H\right)\mathbf{mod}5 \\ &= \{A\left(2^7\mathbf{mod}5\right) + B\left(2^6\mathbf{mod}5\right) + C\left(2^5\mathbf{mod}5\right) + D\left(2^4\mathbf{mod}5\right) \\ &\quad + E\left(2^3\mathbf{mod}5\right) + F\left(2^2\mathbf{mod}5\right) + G\left(2^1\mathbf{mod}5\right) + H\}\mathbf{mod}5 \end{aligned}$$

Calculating the remainders of the different powers it results:

$$\begin{aligned} &2^7\mathbf{mod}5 = 3; 2^6\mathbf{mod}5 = 4; 2^5\mathbf{mod}5 = 2; 2^4\mathbf{mod}5 = 1; \\ &2^3\mathbf{mod}5 = 3; 2^2\mathbf{mod}5 = 4; 2^1\mathbf{mod}5 = 1 \end{aligned}$$

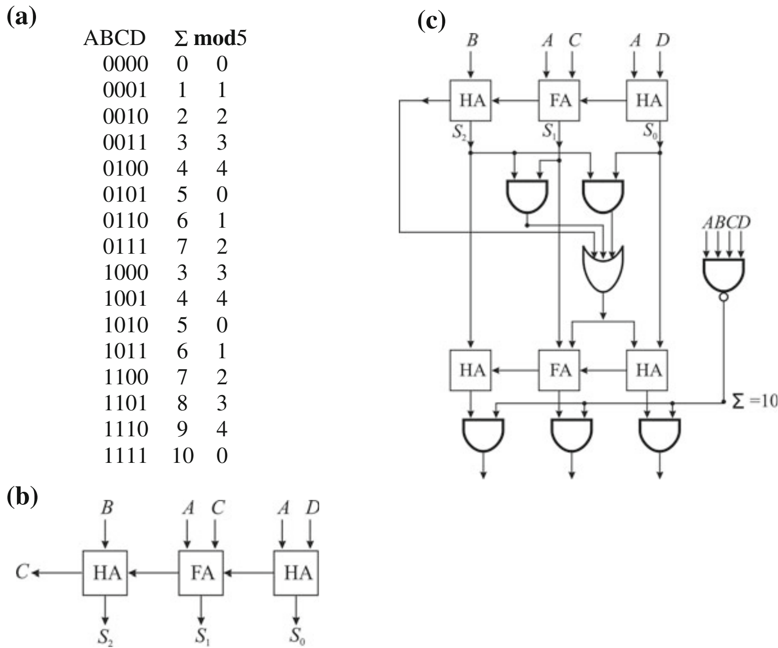
Thus:

$$N\mathbf{mod}5 = (3A + 4B + 2C + D + 3E + 4F + 2G + H)\mathbf{mod}5$$

Applying this expression, the modular reduction can be made using three blocks:

$$\begin{aligned} L &= (3A + 4B + 2C + D)\mathbf{mod}5; \\ M &= (3E + 4F + 2G + H)\mathbf{mod}5; \\ N &= (L + M)\mathbf{mod}5 \end{aligned}$$

The calculations for L and M are identical, and in what follows reference to L will be made. The sum ($\Sigma = 3A + 4B + 2C + D$) and the remainder (**mod**5) for each combination of the inputs are shown in the table of Fig. 2.16a. It is immediate that the value of Σ can be obtained with the circuit of Fig. 2.16b. It can be probed that to get the remainder it is just necessary adding 3 to Σ when Σ is



(a)

ABCD	Σ	**mod5**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	0
0110	6	1
0111	7	2
1000	3	3
1001	4	4
1010	5	0
1011	6	1
1100	7	2
1101	8	3
1110	9	4
1111	10	0

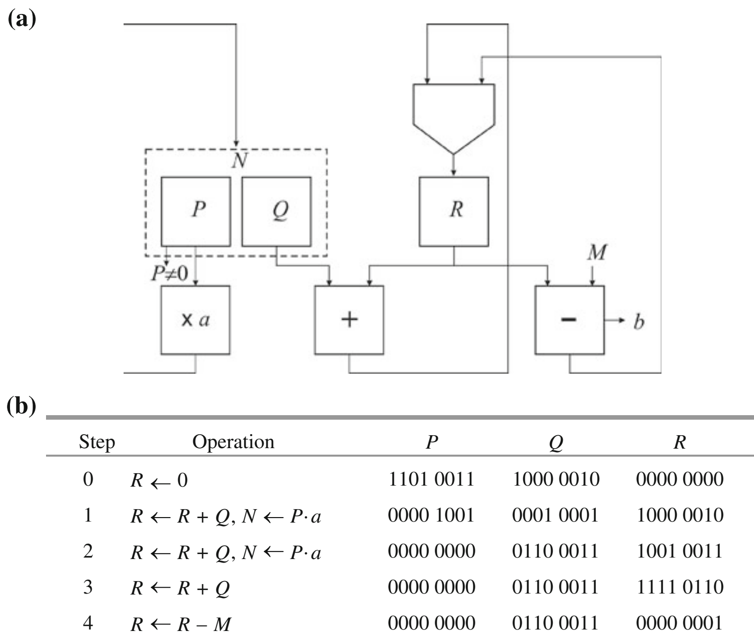
Fig. 2.16 **a** Table with additions and remainders. **b** Circuit for Σ. **c** Circuit for **mod**5

equal to 5, 6 or 7, or the carry is $c = 1$, i.e., it must add 3 when the function $F = s_2 s_0 + s_2 s_1 + c$ is equal to 1. This correction, when $\Sigma = 10_{10}$, gives 5 as remainder, when it should be 0. Therefore the result should be correct in this situation, to be 0 instead of 5. Since $\Sigma = 10_{10}$ only for ABCD = 1111, with a **NAND** gate this exceptional situation can be controlled. Joining the two successive corrections it results the circuit of Fig. 2.16c.

It is easy to see that to calculate N the same circuit for L or M can be used, although in the case of N it is not necessary to correct the value 10, since it can not appear.

In short, with three blocks as that in Fig. 2.16b the remainder from dividing by 5 any 8-bit binary number can be calculated. □

The modular reduction algorithm based on successive modular multiplications (modular multiplicative reduction, see Sect. 1.2.4) can also be used. Let $M = 2^k - a$, $1 \le a < 2^k - 1$. To calculate N**mod**M, being N an n-bit integer number, an n-bit register N can be used, which is the concatenation of P, of $n - k$ bits, and Q, of k bits, by applying the following algorithm:



Step	Operation	P	Q	R
0	$R \leftarrow 0$	1101 0011	1000 0010	0000 0000
1	$R \leftarrow R + Q, N \leftarrow P \cdot a$	0000 1001	0001 0001	1000 0010
2	$R \leftarrow R + Q, N \leftarrow P \cdot a$	0000 0000	0110 0011	1001 0011
3	$R \leftarrow R + Q$	0000 0000	0110 0011	1111 0110
4	$R \leftarrow R - M$	0000 0000	0110 0011	0000 0001

Fig. 2.17 Multiplicative modular reduction. **a** Processing unit. **a** Example N = 1101 0011 1000 0010

Algorithm 2.6

First algorithm for modular reduction

```
R ← 0;
while P ≠ 0, do
    R ← R+Q, N ← P·a;
    R ← R+Q;
while R ≥ M, do
    R ← R–M;
```

End algorithm

The register R stores the result.

Example 2.4 **Design a circuit to operate with 16-bit binary numbers for calculating *N*mod245.**

Since $245 = 256 - 11 = 2^8 - 11 = 2^k - a$ (it results $k = 8$, $a = 11$), the 16-bit register N is the concatenation of two 8-bit registers, P and Q. An auxiliary register, R, is used to store intermediate results, and which is initially set to zero. In each iteration R and Q have to be added, and P has to be multiplied by 11. The number N to be reduced has to be initially introduced into the register N. With all this, the circuit of Fig. 2.17a can be used as the processing unit for this calculation.

As an example, the contents of the various registers when N = 1101 0011 1000 0010 are shown in Fig. 2.17b. □

2.6.5 *Calculating the Quotient by Undoing the Multiplication*

When it is known that the remainder is zero (i.e., it is an exact division) then iterative procedures that try to "undo" the multiplication [Sri94] can be used, such as dividing by 3, as detailed in the following example. For division by 5, see [Sit74].

Example 2.5 **Let suppose the case of the division of any 8-bit integer number N = $QRSTUVWZ$ multiple of 3, by 3 (zero remainder).**

The 6-bit quotient is $C = abcdef$. The task consists on obtaining *abcdef* from *QRSTUVWZ*. The bits *QRSTUVWZ* are related to the bits *abcdef*. In fact, multiplying *abcdef* by $3_{10} = 11_2$, as follows, it results:

		a	b	c	d	e	f
					x	1	1
		a	b	c	d	e	f
	a	b	c	d	e	f	
Q	R	S	T	U	V	W	Z

It is clear that $f = Z$. As $W = e + f$, e is obtained by subtracting f to *QRSTUVW*; specifically it is the least significant bit of $M = QRSTUVW - f$. After deletion of the least significant bit of M with a right shift, d is obtained by subtracting e, and so on for the remaining bits of the quotient. In short, by shifts and subtractions the quotient bits are obtained, one each time, from the least significant to the most significant, with an algorithm, whose core may be obtained as follows:

$$C \leftarrow \overset{n_0 \rightarrow}{C}, N \leftarrow \overrightarrow{N - n_0}$$

where C (where it will be the quotient) is a shift register in which, at each iteration, the least significant bit in register N (n_0 bit) is entered. This division can be implemented with the circuit of Fig. 2.18. □

The procedure for dividing by a constant when the remainder is zero applied in Example 2.5 can be easily extended to any divisor, with appropriate modifications. For example, repeating this procedure for each case, to divide by $5_{10} = 101_2$, two bits of the quotient can be obtained in each iteration, and to divide by $9_{10} = 1001_2$, three bits of the quotient can be obtained in each iteration.

When the binary divider development has more than two ones, the intermediate operations can be more complex, even the minimal signed digit development can be used. For example, for $7 = 111 = 100\overline{1}$ a procedure involving additions instead of subtractions can be designed. In general, both sums and subtractions may appear.

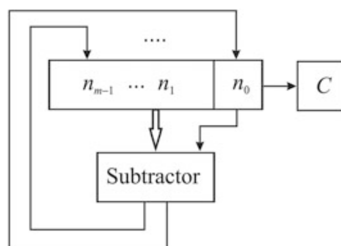


Fig. 2.18 Divider by 3

Table 2.2 Inverses of the first integers

$1/3 = 0.\overline{01}$
$1/5 = 0.\overline{0011}$
$1/7 = 0.\overline{001}$
$1/9 = 0.\overline{000111}$
$1/11 = 0.\overline{0001011101}$
$1/15 = 0.\overline{000100111011}$

2.6.6 Calculating the Quotient by Multiplying by the Inverse of the Divisor

Another method for dividing by a constant, particularly when only the quotient is of interest, consists on multiplying by its inverse. The inverse of the first odd integers are given in Table 2.2; the inverse of an even number is simply obtained by shifts of the inverse of the greater odd which it is a multiple.

When the task consists of dividing by a constant whose inverse is a periodic fraction, to obtain the quotient a process in which simply multiplying the dividend by the first period or by the first two periods (usually more than two periods are not necessary) of the inverse of the divisor can be designed, and then adding 1 to the least significant position, as it is demonstrated in the following example.

Example 2.6 **Design a circuit to divide any unsigned 8-bit integer number *N*, multiple of 5 (it means, with remainder 0), by $5_{10} = 101_2$.**

The largest multiple of 5 with 8 bits is $255_{10} = 11111111_2$. The quotient in this case is $51_{10} = 110011_2$. According to Table 2.2, the inverse of 5 is $1/5 = 0.\overline{0011}$. Multiplying 11111111 by 0.0011 it results the following integer:

$$11111111 \times 0.0011 = 101111 = 47_{10}$$

which differs in three of 51. Therefore it does not suffice to multiply by the first period to generate a result that differs in one from the correct. Using two periods, it results the following integer:

$$11111111 \times 0.00110011 = 110010 = 50_{10}$$

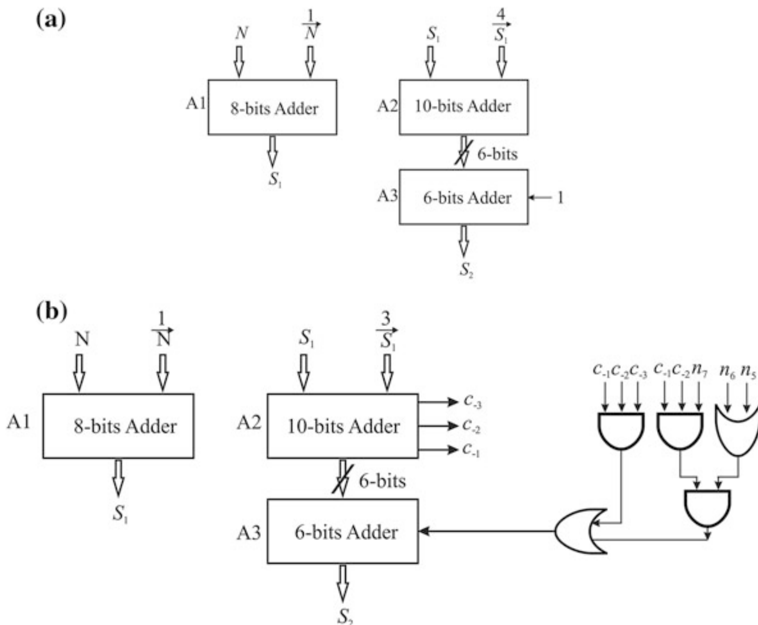


Fig. 2.19 **a** Divider by 5 for multiples of 5. **b** Divider by 5

which differs in one of 51. Therefore, to calculate the quotient of dividing by 5 any 8-bit multiple of 5 (non-zero), it is enough to multiply by 0.00110011, and to add 1 to the integer part of the result. Moreover, the multiplication by 0.00110011 can be done in two iterations, with a single adder, or in one iteration, with two adders.

A circuit for dividing by 5, in one iteration, including the correction of adding 1 to the result of the multiplication by 0.00110011, is shown in Fig. 2.19a. A first adder of 8 bits, A1, whose inputs are $E_1 = N$ and $E_2 = \overrightarrow{N}$ (that is, N shifted right one position) and whose 10-bit output is S1, is used. S1 (unshifted and shifted four positions) is the input to a second adder A2; only the most significant 6 bits of the output of A2 are used; to add these 6 bits and 1 the adder A3 is used, consisting of six half adders. □

The procedure for obtaining the quotient in an exact division using the multiplication by the inverse consists of to analyze the behavior of the largest possible multiple of the divisor, and to decide how many periods of the inverse should be used. This particular procedure outlined in Example 2.6 can be easily refined and extended to obtain the integer quotient when dividing any number by 5, whether be or not a multiple of 5.

With the multiplication by the inverse there is a procedure for scaling values within a predetermined range. When it is applied without refinements to any value, not necessarily a multiple of the constant scaling, the maximum error that can be committed is 1.

With a more detailed analysis of each case, it is possible to design circuits to divide by means of the multiplication by the inverse and that produce a correct result in all cases, as can be seen in the following example.

Example 2.7 **Design a circuit to obtain the quotient of the division by $5_{10} = 101_2$ of any unsigned 8-bit integer number N ($n_7 \ldots n_0$).**

According to Example 2.6, if N is a multiple of 5, it is necessary to add 1 to the result of multiplying N by 0.00110011. It is easy to see that if N is not a multiple of 5, the integer part of $N \times 0.00110011$ is the correct quotient. Therefore, if for this case is intended to construct a similar circuit to that in Fig. 2.19a, it is necessary to separate the multiples of 5 from the other values. This can be done by analyzing the fractional part of $N \times 0.00110011$. With a detailed analysis of the different cases, it is concluded that N is a multiple of 5 if the three most significant bits of the fractional part $c_{-1}c_{-2}c_{-3}$ are equal to 111 (this is true for $N \leq 160$), or if the two most significant bits $c_{-1}c_{-2}$ are equal to 11, and $N > 160$. As $160_{10} = 10100000_2$,

$$N \geq 160 \rightarrow n_7(n_6 + n_5) = 1$$

Therefore, the condition for adding 1 is that the following function F be equal to 1:

$$F = c_{-1}c_{-2}c_{-3} + c_{-1}c_{-2}n_7(n_6 + n_5)$$

From all this, the circuit of Fig. 2.19b will generate the correct quotient of any 8-bit unsigned integer, when it is divided by 5. □

Other procedures for dividing by an integer are based on the following expressions, which can be easily verified making the corresponding divisions:

$$(1 - 2^{-n})^{-1} = 1 + 2^{-n} + 2^{-2n} + 2^{-3n} + \cdots$$
$$(1 + 2^{-n})^{-1} = 1 - 2^{-n} + 2^{-2n} - 2^{-3n} + \cdots$$

Given an integer p, it is possible to find two integers q and n, such that:

$$p \times q = 2^n - 1 \Rightarrow \frac{1}{p} = \frac{q}{2^n - 1}$$

but

$$(2^n - 1)^{-1} = 2^{-n}(1 - 2^{-n})^{-1} = 2^{-n}\left(1 + 2^{-n} + 2^{-2n} + 2^{-3n} + \cdots\right)$$

From this equation:

$$\frac{1}{p} = q \times 2^{-n}\left(1 + 2^{-n} + 2^{-2n} + 2^{-3n} + \cdots\right)$$

Table 2.3 Applied products to the first integers

p	$p \times q$	$2^n \pm 1$	n
3	3×1	3	2
5	5×1	5	2
7	7×1	7	3
9	9×1	9	3
11	11×3	33	5
13	13×5	65	6
15	15×1	15	4
17	17×1	17	4
19	19×27	513	9
21	21×3	63	6
23	23×89	2047	11
25	25×41	1025	10
27	27×19	513	9
29	29×565	16385	14
31	31×33	1023	10
33	33×31	1023	10
35	35×117	4095	12
37	37×7085	262145	18
39	39×105	4095	12
41	41×25	1025	10
43	43×3	129	7
45	45×91	4095	12
47	47×178481	8388607	23
49	49×42799	2097151	21
51	51×5	255	8

Another option is:

$$p \times q = 2^n + 1 = \frac{1}{p} = \frac{q}{2^n + 1}$$

$$(2^n + 1)^{-1} = 2^{-n}(1 + 2^{-n})^{-1} = 2^{-n}\left(1 - 2^{-n} + 2^{-2n} - 2^{-3n} + \cdots\right)$$

$$\frac{1}{p} = q \times 2^{-n}\left(1 - 2^{-n} + 2^{-2n} - 2^{-3n} + \cdots\right)$$

Therefore, for dividing by p it is enough to multiply by q, that it is shifted n places, and for the corresponding sum $(1 + 2^{-n} + 2 - 2^n + 2 - 3^n + \cdots)$ or $(1 - 2^{-n} + 2 - 2^n - 2 - 3^n + \cdots)$. Only the first summands of each addition are used, as can be seen in the following example. Of these two last possibilities, the integer q leading to a simpler procedure is chosen in each case. Possible products applicable to the first integers are shown in Table 2.3.

Example 2.8 **Obtain the expression corresponding to the division by 5 and by 7 using the Table 2.3.**

The development of $(2^n + 1)^{-1}$ is used for the division by 5, where $q = 1$, $n = 2$. It results:

$$\frac{1}{5} = 1 \times 2^{-2}\left(1 - 2^{-2} + 2^{-4} - 2^{-6} + \cdots\right) = 0.01(1 - 0.01 - 0.0001 - 0.0000001 + \cdots)$$
$$= 0.01(0.11 + 0.000011 + \cdots) = 0.\overline{0011}$$

The development of $(2^n - 1)^{-1}$ is used for the division by 7, where $q = 1$, $n = 3$. It results:

$$\frac{1}{7} = 1 \times 2^{-3}(1 + 2^{-3} + 2^{-6} - 2^{-9} + \cdots) = 0.001(1 + 0.001 + 0.000001$$
$$+ 0.000000001 + \cdots) = 0.\overline{001}$$

Of course, the obtained expressions are identical to those given in Table 2.2.□

2.6.7 Modular Reduction (Again)

The idea developed in the previous section to obtain the quotient multiplying by the inverse of the divisor can be used to implement the modular reduction, N**mod**m. It involves using a good approximation for the value of the quotient of N divided by m. As seen in Sect. 2.6.6, using an appropriate value for $M = 1/m$ and multiplying by N, the correct value of the quotient is obtained, or a sufficiently approximate value, c_a, so that the following algorithm can calculate $R = N$**mod**m:

Algorithm 2.7

Second algorithm for modular reduction

```
R ← N-N·M·m;
while R ≥ m, do
    R ← R-m;
```

End algorithm

If n digits are used for operating in the base b, the $N \cdot M$ product can be expressed as:

$$c_a = \left\lfloor \frac{N}{b^k} \frac{b^n}{m} \frac{1}{b^{n-k}} \right\rfloor$$

so that, pre calculating $M = \frac{b^n}{m}$, it is enough to multiply M by the $n-k$ most significant digits of N to obtain c_a, as it is probed in the following example. The Barrett modular reduction method [Bar87] basically consists of this.

Example 2.9 **Design a procedure to obtain Nmod13, with N of 8 bits.**

With these data $2^8/13 \approx 19_{10} = 10011_2$ can be used. Let consider the extreme case $N = 1111\ 1111$. It is straightforward to check that $c = 10011$.

- For $k = 4$: c_a = five most significant bits of 1111 × 10011 = 10001. Therefore $c - c_a = 2$ (should subtract twice).
- For $k = 3$: c_a = five most significant bits of 11111 × 10011 = 10010. Therefore $c - c_a = 1$ (should subtract once).
- For k = 2, 1 and 0, the same value is obtained for c_a (10010).

Using $2^8/13 \approx 19.5_{10} = 10011.1_2$ again for the extreme case N = 1111 1111 (will remain c = 10011), it results:

- For $k = 4$: c_a = five most significant bits of 1111 × 100011.1 = 10010. Therefore $c - c_a = 1$ (should subtract once).
- For $k = 3$ it results the same value for c_a (10010).
- For $k = 2$: c_a = five most significant bits of 1111 × 100011.1 = 10010. Therefore $c = c_a$ (subtraction is not necessary).

In conclusion, for this application works well using $2^8/13 \cdot 19.5_{10} = 10011.1_2$ and do $k = 2$. □

2.6.8 Square Root

The square root can be extracted by successive subtractions, such as seen in Sect. 1.6.1. The obtained circuits are very similar to those implementing the division. For example, the circuit of Fig. 2.20a, which uses the same **CR** and **CS** cells from the divider (Fig. 2.14b, c) extracts the integer binary square root of any 8-bit integers, $a_7 \ldots a_0$. In this case, the combinational circuit for calculating the square root has four stages or rows, each one calculating $D^+ = D - (4R_1 + 1)2^{2i}$. If $D^+ \geq 0$, then $r_i = 1$ and D is substituted by D^+; if $D^+ < 0$, then $r_i = 0$ and D is unchanged. The result is a square root of four bits, $r_3r_2r_1r_0$, and a remainder of five bits, $b_4b_3b_2b_1b_0$.

The integer square root of a binary number of 8 bits, $A = a_7 \ldots a_0$, may be calculated with the sequential circuit of Fig. 2.20b using four iterations. A shift register is used to store A, called SR1, with double shift at each iteration, so that in the a_i and a_{i-1} outputs are successively obtained a_7 and a_6, a_5 and a_4, a_3 and a_2, a_1 and a_0. The successive bits of the result are written to a normal shift register called SR2. The results of successive subtractions are written to a read-write parallel register, R3. Initially R3 must be zero. After four iterations, the root is stored in SR2 and the remainder is stored in R3. It is easy to verify that, with the specified operating conditions, the circuit of Fig. 2.20b performs in each iteration the same action as the corresponding row of the circuit of Fig. 2.20a.

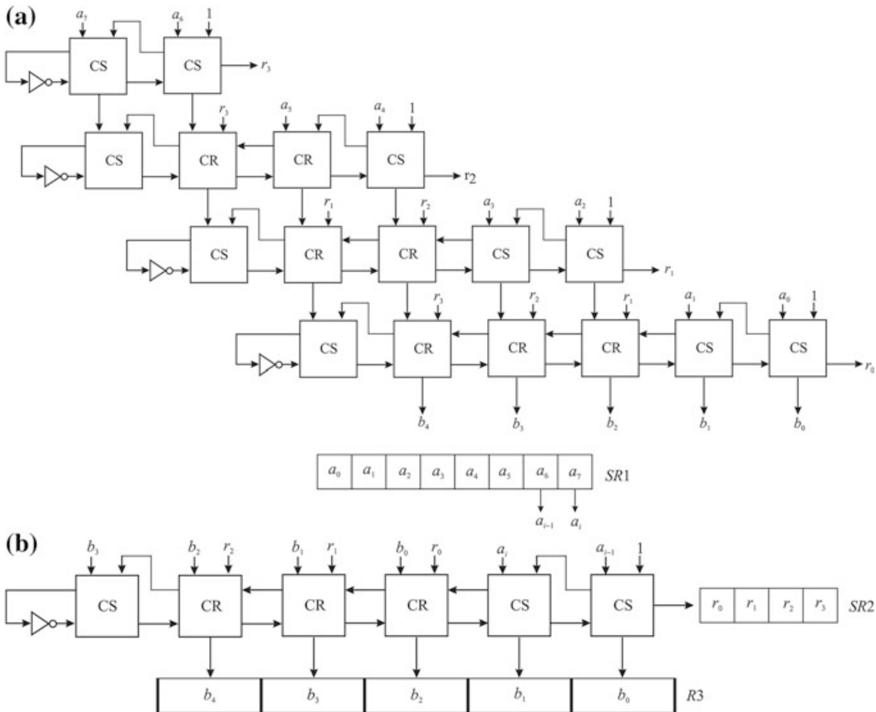


Fig. 2.20 Square root **a** Combinational circuit. **b** Sequential circuit

2.7 BCD Adder/Subtracter

From Sect. 1.7.1 it results that a circuit to add two **BCD** characters, $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$, can be constructed with four binary adders plus the correction circuit for adding 6 when appropriate. Calling $R = r_3r_2r_1r_0$ to the partial result generated by the four binary adders, and calling a_+ to the partial carry, 6 must be added when $F = 1$, for which two expressions are given:

$$F = r_3r_2 + r_3r_1 + a_+ = a_+ + a_{++}$$

This F function also gives the final carry. Therefore, the circuit of Fig. 2.21a or b is an adder for **BCD** digits. Unsigned decimal numbers of n digits can be added by cascading n adder circuits of **BCD** digits as depicted in Fig. 2.21c.

The sign digit has to be included if the subtraction has to be implemented. The **SM** representation is not recommended for subtraction, since prior to the operation itself, the two operands should be compared. However, if the 9's complement representation is used, basically the same structure in Fig. 2.21c can be used to add and subtract. Just it is necessary to change the sign of the subtrahend and 9's complement each of its digits. The truth table for the 9's complement generation of

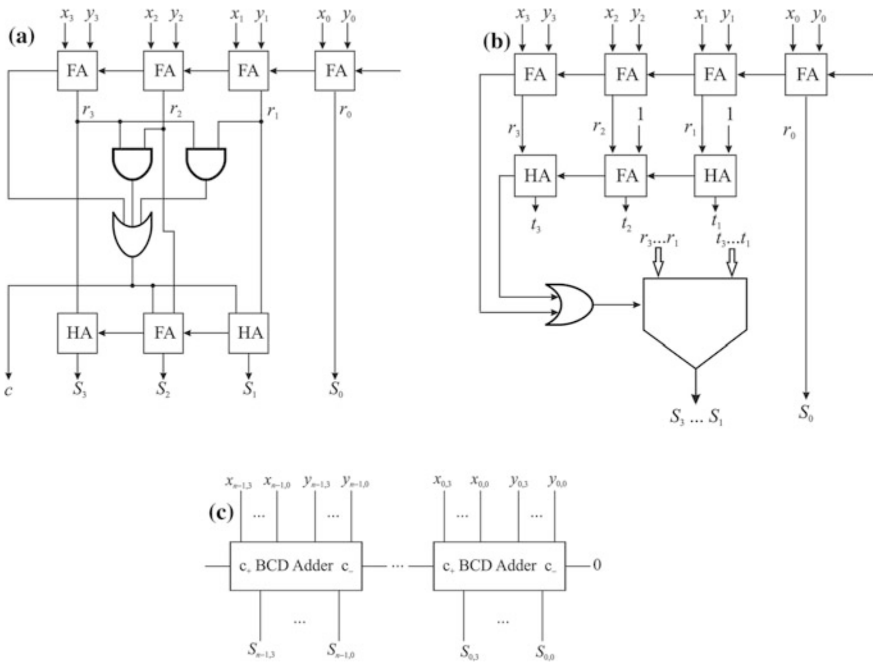


Fig. 2.21 **BCD** adder. **a** For digits. **b** For digits by using a multiplexer. **c** For numbers of length *n*

each digit is shown in Fig. 2.22a, and the corresponding circuit is shown in Fig. 2.22b; to change the sign it is enough to invert the bits with which it is encoded. Using the control signal $\bar{s}/r$, to be 0 for the sum and to 1 for the subtraction, in Fig. 2.22c has an adder/subtractor for two **BCD** numbers of $n - 1$ digits plus a sign digit represented in 9's complement; in this circuit it is taken into account the end-around carry.

Regarding the 10's complement representation, it is important to remember that the 10's complement of a **BCD** number can be obtained from the 9's complement, by adding 1. Using this idea, and considering that in this case there is no end-around carry, the circuit of Fig. 2.22d is an adder/subtractor for **BCD** numbers represented in 10's complement.

Comparing the circuits of Fig. 2.22c, d, it is obvious that it is preferable the 10's complement representation versus the 9's complement representation.

2.8 Comparators

In the processing of the information it is common to have to compare two words or data in general. For example, ordering from lowest to highest a table of numbers or alphabetize a series of words, the elements are compared in pairs and sorted

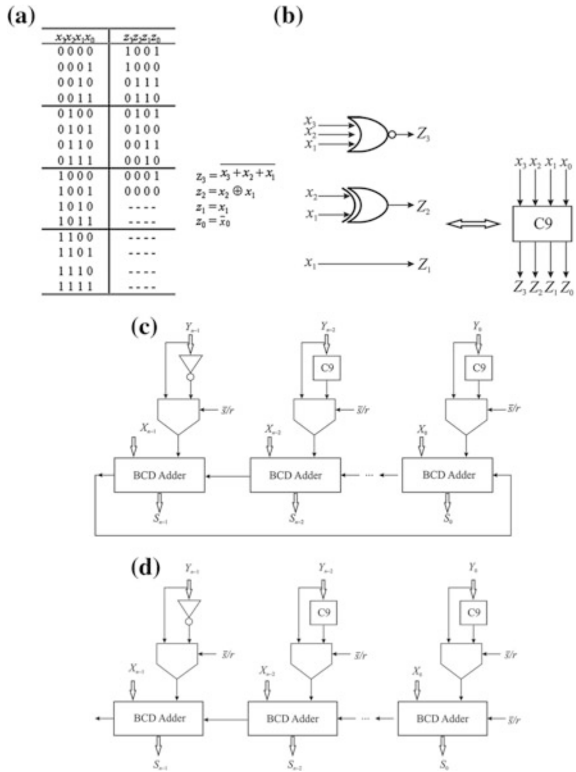


Fig. 2.22 **a** Truth table for the 9's complement. **b** Circuit to calculate 9's complement. **c** 9's complement adder/subtractor. **d** 10's complement adder/subtractor

accordingly; also, in many arithmetic operations different numerical results have to be compared.

Let X and Y be two elements to sort; the comparators can be used for this task. A comparator for n-bit words has $2n$ inputs and m outputs so that each of the elements to be compared is encoded with n bits, and the comparison is made based on the value in binary (unsigned) of these encodings. The m outputs give the result of the comparison; the most frequent is $m = 3$, in which case the outputs are $X > Y$, $X = Y$, $X < Y$, each being activated as they fulfill the corresponding condition.

The simplest comparator is that including 1-bit words ($n = 1$). In this case the three functions to be synthesized, as can be easily checked in the table in Fig. 2.23a, are:

For $X > Y$: $f_2(x, y) = x\bar{y}$
For $X = Y$: $f_9(x, y) = xy + \bar{x}\bar{y}$
For $X < Y$: $f_4(x, y) = \bar{x}y$

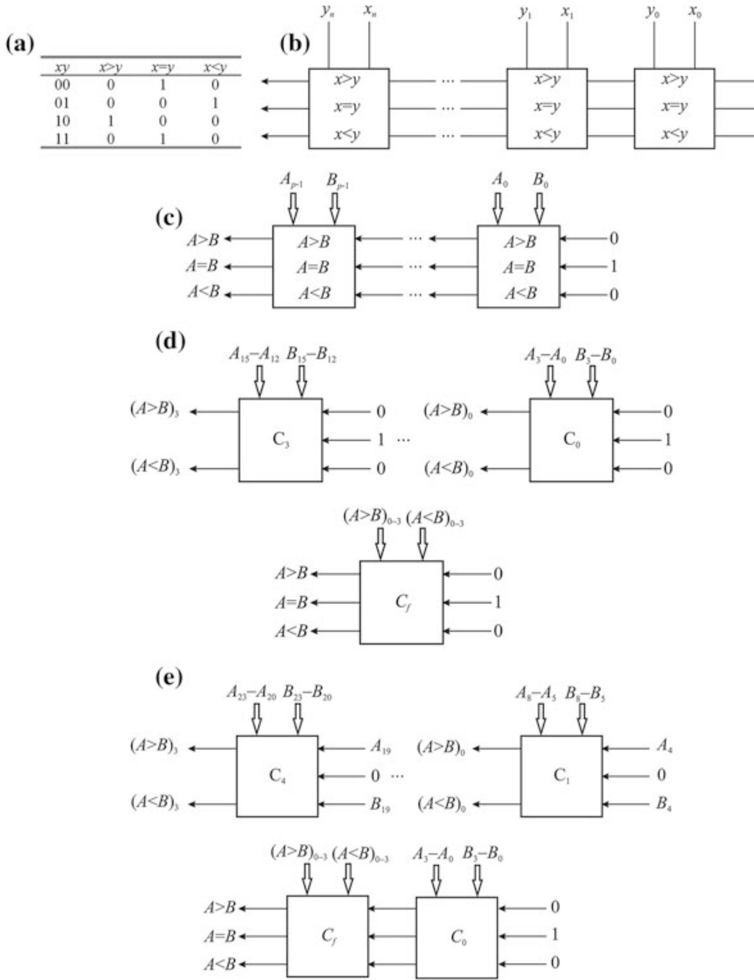


Fig. 2.23 Comparators. **a** Table for 1-bit comparator. **b** Cascade connection. **c** Cascade connection of digit comparator. **d** Parallel-serial connection of digit comparators. **e** Comparator of 24-bit words with parallel-serial connection

A common value for n is 4 ($X = x_3 \ldots x_0$, $Y = y_3 \ldots y_0$). For this case, the output $X = Y$ will be 1 when the corresponding bits of each input are equal; this means:

$$F(X = Y) = f_9(x_3, y_3) \cdot f_9(x_2, y_2) \cdot f_9(x_1, y_1) \cdot f_9(x_0, y_0)$$

The output $X > Y$ will be 1 if $x_3 > y_3$, or $x_3 = y_3$ and $x_2 > y_2$, or $x_3 = y_3$ and $x_2 = y_2$ and $x_1 > y_1$, or $x_3 = y_3$ and $x_2 = y_2$ and $x_1 = y_1$ and $x_0 > y_0$; it means:

$$\begin{aligned}F(X > Y) &= f_2(x_3, y_3) + f_9(x_3, y_3) \cdot f_2(x_2, y_2) + f_9(x_3, y_3) \cdot f_9(x_2, y_2) \\ &\quad \cdot f_2(x_1, y_1) + f_9(x_3, y_3) \cdot f_9(x_2, y_2) \cdot f_9(x_1, y_1) \cdot f_2(x_0, y_0) \\ &= f_2(x_3, y_3) + f_9(x_3, y_3) \cdot (f_2(x_2, y_2) \\ &\quad + f_9(x_2, y_2) \cdot (f_2(x_1, y_1) + f_9(x_1, y_1) \cdot f_2(x_0, y_0)))\end{aligned}$$

The expression for the function corresponding to the output $X < Y$ is parallel to the one for $X > Y$, substituting $>$ by $<$ (it means, f_2 by f_4). It is obvious that once that two of the comparator outputs are known, the third can be obtained from these two. Concretely,

$$\begin{aligned}X = Y &\Leftrightarrow (\overline{X > Y}) \cdot (\overline{X < Y}) \\ X > Y &\Leftrightarrow (\overline{X = Y}) \cdot (\overline{X < Y}) \\ X < Y &\Leftrightarrow (\overline{X = Y}) \cdot (\overline{X > Y})\end{aligned}$$

Thus, it is suffices to synthesize two of the output functions and to construct the third as the products of its complements (**NOR** function).

The commercially available comparators are ready for possible cascade connection, for which they include three inputs ($X > Y_{in}$, $X = Y_{\text{in}}$, $X < Y_{in}$), through which the outputs of the preceding stage are introduced, thereby allowing to build comparators for words of any length, as shown in Fig. 2.23b for the case of 1-bit comparators. For comparator of digits of m bits, this way of cascade connection can be used to construct comparators of pm bit words, as shown in Fig. 2.23c. This cascading connection can be slow since the overall delay accumulates the delay of all the comparators.

Parallel-serial structures with several comparators can be used to accelerate the response of the comparator of words, which partial results are globalized at a final comparator, as shown in Fig. 2.23d for the case of constructing a comparator for 16-bit words using 4-bit digit comparators. In this case four parallel comparators are used, C_3 … C_0. Two digits are composed with the outputs of these parallel comparators, $A > B$ and $A < B$, that are compared in a final comparator, C_f, which provides the final result of the comparison. This parallel-serial structure can be improved by using the inputs provided to the cascade connection; in the structure of Fig. 2.23d are set to the neutral values 010. For example, using five comparators in parallel, C_4 …C_0, the final comparator C_f, and the inputs for the cascading connection, a comparator for 24-bit words, as shown in Fig. 2.23e, can be built.

2.9 Shifters

A k-positions shifter is a circuit whose input is an n-bit character, $E = e_{n-1}\ldots e_0$, and whose output is also an n-bit character, $S = s_{n-1}\ldots s_0$, which, when the shifting have to be made, it is obtained from the input E by means of k-shifts,

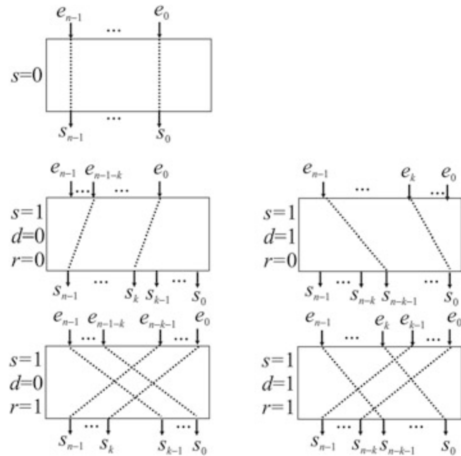


Fig. 2.24 Actions of the shifters

either to the right or to the left, as stated, as shown in Fig. 2.24; if no displacement has to be made, then $S = E$.

In a shift of k positions there will be k bits of S to which no bit of E is applied: the k most left bits when moving to the right, or the k most right bits on the left shifts. For these k bits of S the values to be assigned has to be established, usually using one of the following two options:

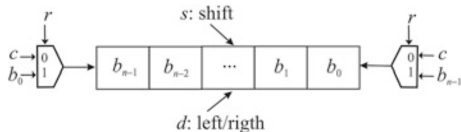
(a) filled with constant (all zero or all one, although other combinations are possible);
(b) filled with the k bits of E that would be unmatched (i.e., the most right on the right shifts, or the most left on the left shifts; in both cases it consists on rotating the input in the sense that apply). For example, in a shifter with two positions ($k = 2$) for 8-bit characters ($n = 8$), with zero padding, if $E = 10011101$, a shift to the right will result $S = 00100111$, and a shift to the left be $S = 01110100$; if the padding were with remaining input bits (i.e., a rotation) in a shift to the right will be $S = 01100111$, and in a shift to the left $S = 01110110$.

Therefore, to define the action of a shifter it must be specified:

- the size n of the characters to be shifted,
- whether it has to perform or not the shift, with the variable s,
- the magnitude of the shift k,
- if the shift is to the right or to the left, with the variable d,
- if the padding is a constant value or by rotation, with the variable r,
- and finally, the value of the constant filling, where appropriate, with the c variable.

The size n of the characters are supposed to be pre-established, usually equal to the size of the characters being processed. The decision to perform the shift is

Fig. 2.25 Shifter built using a shift register



described, for example, with $\boldsymbol{s} = \mathbf{1}$ ($\boldsymbol{s} = \mathbf{0}$, no shift). The magnitude of the shift k can range from $\boldsymbol{k} = \mathbf{1}$ in **simple shifter** until $\boldsymbol{k} \leq \boldsymbol{n}$ in a **general shifter**, called by some authors as a **barrel shifter**. The sense of the shift is encoded with $\boldsymbol{d}$ (0 on the left and 1 to the right, for example); in the simplest case the shift is one-way, in which case it is not necessary that variable. With $\boldsymbol{r}$ the fill type (0 stuffing constant, 1 for rotation, for example) is encrypted. The constant filling, $\boldsymbol{c}$, can match (this would be the easiest) the value 0 or 1 to use.

2.9.1 Shifters Built with Shift Registers

The most obvious and simplest solution to construct a shifter is to use a shift register. The register depends on the features desired for the shifter. Using a bidirectional universal shift register, a shifter with all possible benefits can be built, as shown in Fig. 2.25. A standard shift register can shift a position (to the left or to the right) on each clock pulse.

The drawback of this solution is the time it can take to make a shift. In effect, a shift of k-position takes k clock pulses (each position takes a pulse), and, sometimes, it is an unbearable delay due to the performance degradation involved. Therefore a strictly combinational solution is the option, without using memory elements, as shown below.

2.9.2 Combinational Shifters

Using multiplexers as building blocks it is very easy to get a shifter with any performance. Let consider first the design of a shifter for k fixed and a default value of n. In this case the control variables are: s, d, r and c. It is straightforward to check that the circuit of Fig. 2.26a acts as a fixed shifter of k-positions; it is sufficient to obtain the outputs $s_{n-1} \ldots s_0$ from the multiplexers for all combinations of s, d, and r, and comparing them to the outputs generated in Fig. 2.24.

The k-position shifter of Fig. 2.26a consists of three levels of multiplexing (with 2-to-1 multiplexers), which can be replaced by a single level of multiplexing using 4-to-1 multiplexers, as shown in Fig. 2.26b. In this case the selection is done with signals f_1 and f_0 obtained from s, d, and r as follows (it is left as an exercise to check these functions):

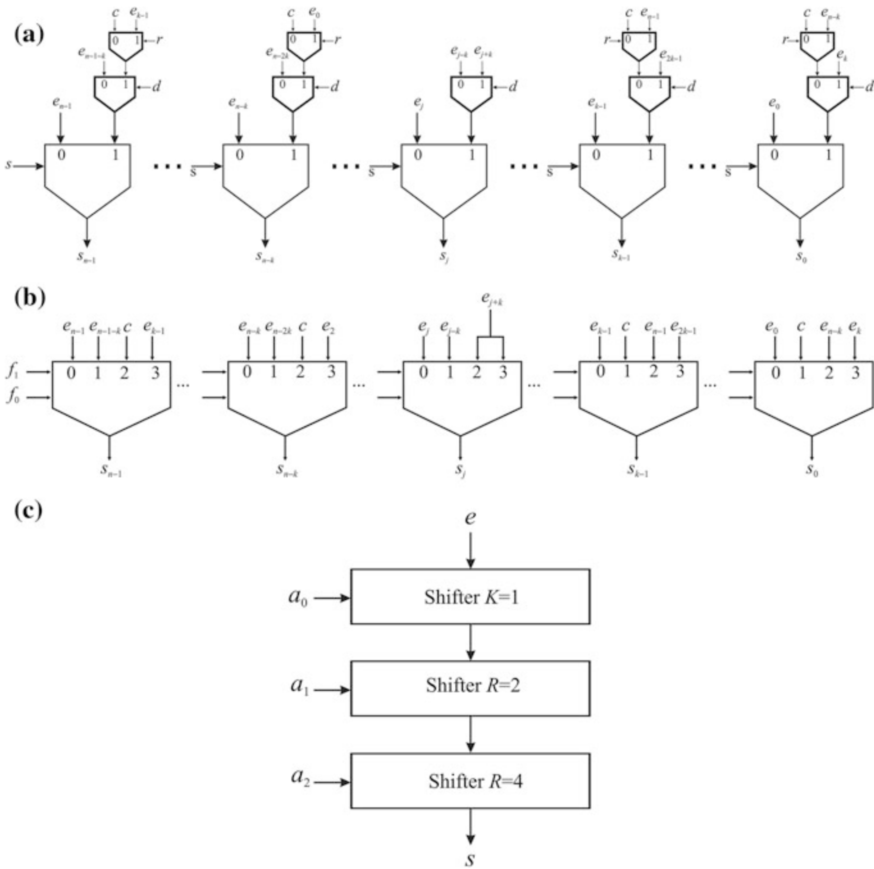


Fig. 2.26 **a** *k*-position shifter. **b** Other *k*-position shifter. **c** Barrel shifter up to 7 positions

$$f_1 = sd \quad f_0 = s\left(\overline{d}\overline{r} + dr\right)$$

Let suppose that p fixed shifters with different values of k are used, such that $k = 2^n$, $n = 0, \ldots, p - 1$. Each of these shifters has its own control input s_i, for deciding whether or not made the corresponding shift; all other control inputs (d, r and c) are common to all shifters. It is easy to check that with these p shifters, acting in cascade, each one over the output of the previous one, any shift $k < 2^p$ can be get. It is sufficient to write k in binary, $k = a_{p-1}\ a_{p-2}\ \ldots\ a_1\ a_0$, and do $s_i = a_i$. The obtained shifter from this structure is sometimes also known as **barrel shifter**. For example, with three shifters (1, 2 and 4 positions) any shift between 0 and 7 can be accomplished, as shown in Fig. 2.26c.

It is obvious that all shifter circuits described above could be simplified if the shifts were in one direction, or if the fill were of a single type, etc.

2.10 Conclusion

This chapter has presented the arithmetic circuits that are used in the following chapters, for the implementation of the algebraic circuits.

References

[Bar87] Barrett, P.: Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology—CRYPTO'86 Proceedings, LNCS, vol. 263, pp. 311–323. Springer, Berlin (1987)

[Bra39] Brauer, A.: On addition chain. Bull. Am. Math. Soc. **45**, 736–739 (1939)

[Sit74] Sites, R.L.: Serial binary division by ten. IEEE Trans. Comp. 1299–1301 (1974)

[Sri94] Srinivasan, P., Petra, F.E.: Constant-division algorithms. IEE Proc. Comput. Tech. **141**(6), 334–340 (1994)

http://www.springer.com/978-3-642-54648-8

Algebraic Circuits
Lloris Ruiz, A.; Castillo Morales, E.; Parrilla Roure, L.; García Ríos, A.
2014, XXIV, 394 p. 110 illus., Hardcover
ISBN: 978-3-642-54648-8