ADDITION/ SUBTRACTION

"In the arithmetic of love, one plus one equals everything, and two minus one equals nothing."
MIGNON MCLAUGHLIN

"A man has one hundred dollars and you leave him with two dollars, that's subtraction."
MAE WEST, MY LITTLE CHICKADEE, 1940

ADDITION IS THE MOST COMMON ARITHMETIC OPERATION AND ALSO SERVES AS a building block for synthesizing many other operations. Within digital computers, addition is performed extensively both in explicitly specified computation steps and as a part of implicit ones dictated by indexing and other forms of address arithmetic. In simple arithmatic/logic units that lack dedicated hardware for fast multiplication and division, these latter operations are performed as sequences of additions. A review of fast addition schemes is thus an apt starting point in investigating arithmetic algorithms. Subtraction is normally performed by negating the subtrahend and adding the result to the minuend. This is quite natural, given that an adder must handle signed numbers anyway. Even when implemented directly, a subtractor is quite similar to an adder. Thus, in the following four chapters that constitute this part, we focus almost exclusively on addition:

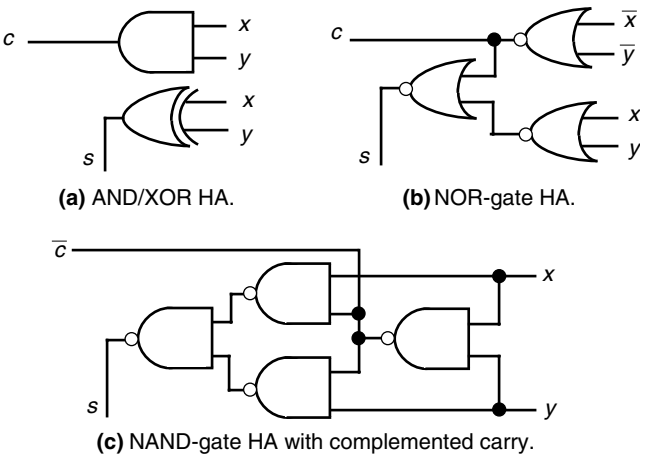
Basic Addition and Counting

"Not everything that can be counted counts, and not everything that counts can be counted."
ALBERT EINSTEIN

As stated in Section 3.1, propagation of carries is a major impediment to high-speed addition with fixed-radix positional number representations. Before exploring various ways of speeding up the carry-propagation process, however, we need to examine simple ripple-carry adders, the building blocks used in their construction, the nature of the carry-propagation process, and the special case of counting. Chapter topics include:

5.1 BIT-SERIAL AND RIPPLE-CARRY ADDERS

Single-bit half-adders (HAs) and full adders (FAs) are versatile building blocks that are used in synthesizing adders and many other arithmetic circuits. A HA receives two input bits x and y, producing a sum bit $s = x \oplus y = x\bar{y} \vee \bar{x}y$ and a carry bit $c = xy$. Figure 5.1 depicts three of the many possible logic realizations of a HA. A HA can be viewed as a single-bit binary adder that produces the 2-bit sum of its 1-bit inputs, namely,



(a) AND/XOR HA. **(b)** NOR-gate HA.

(c) NAND-gate HA with complemented carry.

Figure 5.1 Three implementations of a HA.

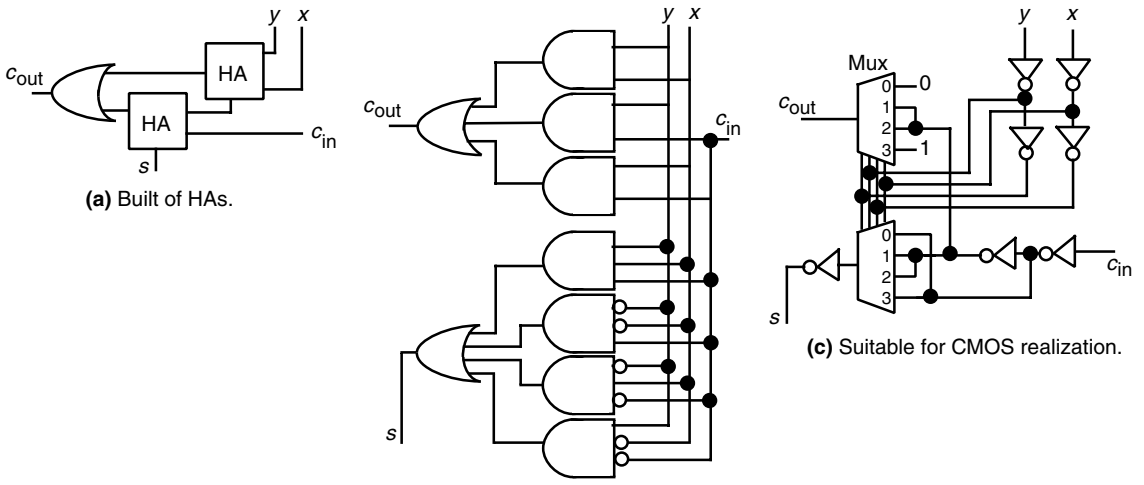
$x + y = (c_{\text{out}}\ s)_{\text{two}}$, where the plus sign in this expression stands for arithmetic sum rather than logical OR.

A single-bit FA is defined as follows:

Inputs:	Operand bits x, y and carry-in c_{in}	(or x_i, y_i, c_i for stage i)
Outputs:	Sum bit s and carry-out c_{out}	(or s_i and c_{i+1} for stage i)
	$s = x \oplus y \oplus c_{\text{in}}$	(odd parity function)
	$= xyc_{\text{in}} \vee \overline{x}\,\overline{y}c_{\text{in}} \vee \overline{x}y\overline{c}_{\text{in}} \vee x\overline{y}\,\overline{c}_{\text{in}}$	
	$c_{\text{out}} = xy \vee xc_{\text{in}} \vee yc_{\text{in}}$	(majority function)

An FA can be implemented by using two HAs and an OR gate as shown in Fig. 5.2a. The OR gate in Fig. 5.2a can be replaced with a NAND gate if the two HAs are NAND-gate HAs with complemented carry outputs. Alternatively, one can implement an FA as two-level AND-OR/NAND-NAND circuits according to the preceding logic equations for s and c_{out} (Fig. 5.2b). Because of the importance of the FA as an arithmetic building block, many optimized FA designs exist for a variety of implementation technologies. Figure 5.2c shows an FA, built of seven inverters and two 4-to-1 multiplexers (mux), that is suitable for complementary metal-oxide semiconductor (CMOS) transmission-gate logic implementation.

Full and half-adders can be used for realizing a variety of arithmetic functions. We will see many examples in this and the following chapters. For instance, a bit-serial adder can be built from an FA and a carry flip-flop, as shown in Fig. 5.3a. The operands are supplied to the FA 1 bit per clock cycle, beginning with the least-significant bit, from a pair of shift registers, and the sum is shifted into a result register. Addition of k-bit numbers can thus be completed in k clock cycles. A k-bit ripple-carry binary adder requires k FAs, with the carry-out of the ith FA connected to the carry-in input of the $(i + 1)$th FA. The resulting k-bit adder produces a k-bit sum output and a carry-out; alternatively, c_{out} can be viewed as the most-significant bit of a $(k + 1)$-bit sum. Figure 5.3b shows a ripple-carry adder for 4-bit operands, producing a 4-bit or 5-bit sum.



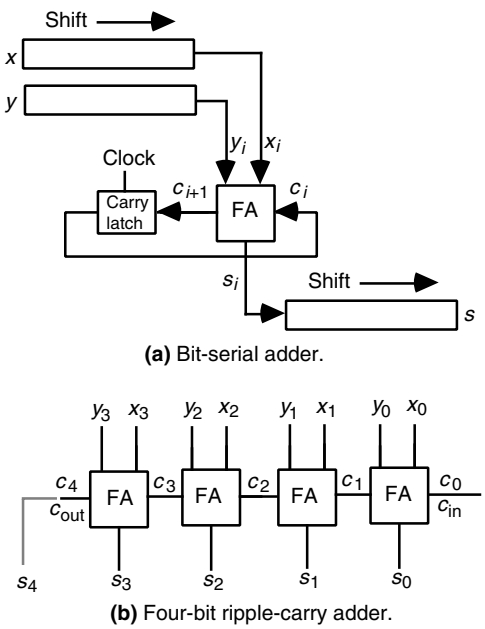
(a) Built of HAs.

(b) Built as an AND-OR circuit.

(c) Suitable for CMOS realization.

Figure 5.2 Possible designs for an FA in terms of HAs, logic gates, and CMOS transmission gates.

Figure 5.3 Using FAs in building bit-serial and ripple-carry adders.



(a) Bit-serial adder.

(b) Four-bit ripple-carry adder.

The ripple-carry adder shown in Fig. 5.3b leads directly to a CMOS implementation with transmission-gate logic using the FA design of Fig. 5.2c. A possible layout is depicted in Fig. 5.4, which also shows the approximate area requirements for the 4-bit ripple-carry adder in units of λ (half the minimum feature size). For details of this particular design, refer to [Puck94, pp. 213–223].

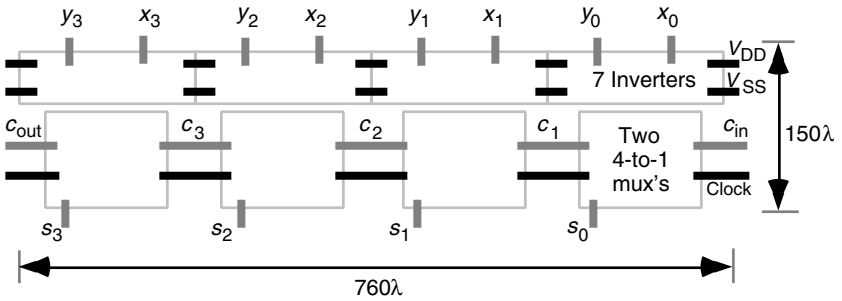


Figure 5.4 Layout of a 4-bit ripple-carry adder in CMOS implementation [Puck94].

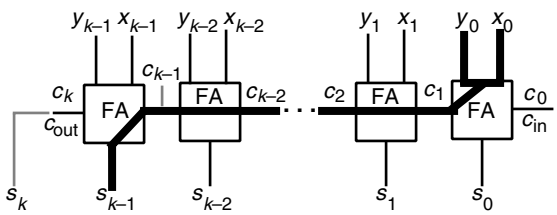


Figure 5.5 Critical path in a k-bit ripple-carry adder.

The latency of a k-bit ripple-carry adder can be derived by considering the worst-case signal propagation path. As shown in Fig. 5.5, the critical path usually begins at the x_0 or y_0 input, proceeds through the carry-propagation chain to the leftmost FA, and terminates at the s_{k-1} output. Of course, it is possible that for some FA implementations, the critical path might begin at c_0 and/or terminate at c_k. However, given that the delay from carry-in to carry-out is more important than from x to carry-out or from carry-in to s, FA designs often minimize the delay from carry-in to carry-out, making the path shown in Fig. 5.5 the one with the largest delay. We can thus write the following expression for the latency of a k-bit ripple-carry adder:

$$T_{\text{ripple-add}} = T_{\text{FA}}(x, y \rightarrow c_{\text{out}}) + (k-2) \times T_{\text{FA}}(c_{\text{in}} \rightarrow c_{\text{out}}) + T_{\text{FA}}(c_{\text{in}} \rightarrow s)$$

where $T_{\text{FA}}(\text{input} \rightarrow \text{output})$ represents the latency of an FA on the path between its specified input and output. As an approximation to the foregoing, we can say that the latency of a ripple-carry adder is kT_{FA}.

We see that the latency grows linearly with k, making the ripple-carry design undesirable for large k or for high-performance arithmetic units. Note that the latency of a bit-serial adder is also $O(k)$, although the constant of proportionality is larger here because of the latching and clocking overheads.

Full and half-adders, as well as multibit binary adders, are powerful building blocks that can also be used in realizing nonarithmetic functions if the need arises. For example, a 4-bit binary adder with c_{in}, two 4-bit operand inputs, c_{out}, and a 4-bit sum output can be used to synthesize the four-variable logic function $w \vee xyz$ and its complement, as depicted and justified in Fig. 5.6. The logic expressions written next to the arrows in

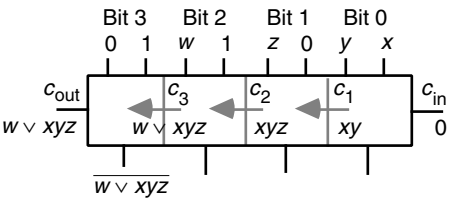


Figure 5.6 A 4-bit binary adder used to realize the logic function $f = w \vee xyz$ and its complement.

Fig. 5.6 represent the carries between various stages. Note, however, that the 4-bit adder need not be implemented as a ripple-carry adder for the results at the outputs to be valid.

5.2 CONDITIONS AND EXCEPTIONS

When a k-bit adder is used in an arithmetic/logic unit (ALU), it is customary to provide the k-bit sum along with information about the following outcomes, which are associated with flag bits within a condition/exception register:

c_{out}	Indicating that a carry-out of 1 is produced
Overflow	Indicating that the output is not the correct sum
Negative	Indicating that the addition result is negative
Zero	Indicating that the addition result is 0

When we are adding unsigned numbers, c_{out} and "overflow" are one and the same, and the "sign" condition is obviously irrelevant. For 2's-complement addition, overflow occurs when two numbers of like sign are added and a result of the opposite sign is produced. Thus

$$\text{Overflow}_{\text{2's-compl}} = x_{k-1}y_{k-1}\overline{s}_{k-1} \vee \overline{x}_{k-1}\overline{y}_{k-1}s_{k-1}$$

It is fairly easy to show that overflow in 2's-complement addition can be detected from the leftmost two carries as follows:

$$\text{Overflow}_{\text{2's-compl}} = c_k \oplus c_{k-1} = c_k\overline{c}_{k-1} \vee \overline{c}_k c_{k-1}$$

In 2's-complement addition, c_{out} has no significance. However, since a single adder is frequently used to add both unsigned and 2's-complement numbers, c_{out} is a useful output as well. Figure 5.7 shows a ripple-carry implementation of an unsigned or 2's-complement adder with auxiliary outputs for conditions and exceptions. Because of the large number of inputs into the NOR gate that tests for 0, it must be implemented as an OR tree followed by an inverter.

When the sum of unsigned input operands is too large for representation in k bits, an overflow exception is indicated by the c_{out} signal in Fig. 5.5 and a "wrapped" value, which is 2^k less than the correct sum, appears as the output. A similar wrapped value may appear for signed addition in the event of overflow. In certain applications, a "saturated" value would be more appropriate than a wrapped value because a saturated value at

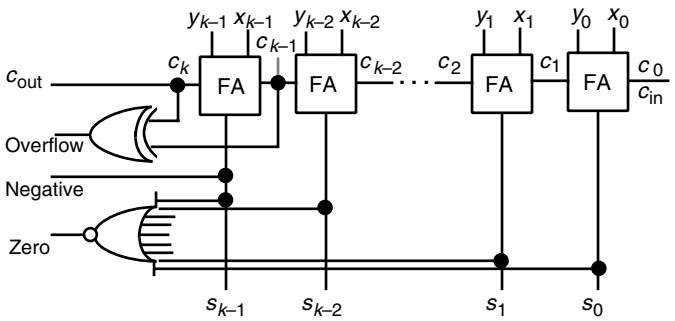


Figure 5.7 A 2's-complement adder with provisions for detecting conditions and exceptions.

least maintains the proper ordering of various sums. For example, if the numbers being manipulated represent the pixel intensities in an image, then an intensity value that is too large should be represented as the maximum possible intensity level, rather than as a wrapped value that could be much smaller. A saturating unsigned adder can be obtained from any unsigned adder design by using a multiplexer at the output, with its control input tied to the adder's overflow signal. A signed saturating adder can be similarly designed.

5.3 ANALYSIS OF CARRY PROPAGATION

Various ways of dealing with the carry problem were enumerated in Section 3.1. Some of the methods already discussed include limiting the propagation of carries (hybrid signed-digit, residue number system) or eliminating carry propagation altogether (redundant representation). The latter approach, when used for adding a set of numbers in carry-save form, can be viewed as a way of amortizing the propagation delay of the final conversion step over many additions, thus making the per-add contribution of the carry-propagation delay quite small. What remains to be discussed, in this and the following two chapters, is how one can speed up a single addition operation involving conventional (binary) operands.

We begin by analyzing how and to what extent carries propagate when adding two binary numbers. Consider the example addition of 16-bit binary numbers depicted in Fig. 5.8, where the carry chains of lengths 2, 3, 6, and 4 are shown. The length of a carry chain is the number of digit positions from where the carry is generated up to and including where it is finally absorbed or annihilated. A carry chain of length 0 thus means "no carry production," and a chain of length 1 means that the carry is absorbed in the next position. We are interested in the length of the longest propagation chain (6 in Fig. 5.8), which dictates the adder's latency.

Given binary numbers with random bit values, for each position i we have

Probability of carry generation $= 1/4$
Probability of carry annihilation $= 1/4$
Probability of carry propagation $= 1/2$

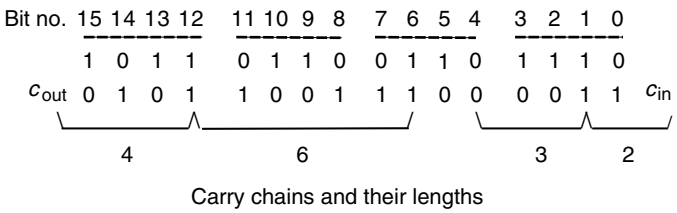


Figure 5.8 Example addition and its carry-propagation chains.

The probability that a carry generated at position i will propagate up to and including position $j-1$ and stop at position j ($j > i$) is $2^{-(j-1-i)} \times 1/2 = 2^{-(j-i)}$. The expected length of the carry chain that starts at bit position i is, therefore, given by

$$\sum_{j=i+1}^{k-1} (j-i)2^{-(j-i)} + (k-i)2^{-(k-1-i)} = \sum_{l=1}^{k-1-i} l2^{-l} + (k-i)2^{-(k-1-i)}$$

$$= 2 - (k-i+1)\,2^{-(k-1-i)} + (k-i)2^{-(k-1-i)} = 2 - 2^{-(k-i-1)}$$

where the simplification is based on the identity $\sum_{l=1}^{p} l2^{-l} = 2 - (p+2)2^{-p}$. In the preceding derivation, the term $(k-i)\,2^{-(k-1-i)}$ is added to the summation because carry definitely stops at position k; so we do not multiply the term $2^{-(k-1-i)}$ by 1/2, as was done for the terms within the summation.

The preceding result indicates that for $i \ll k$, the expected length of the carry chain that starts at position i is approximately 2. Note that the formula checks out for the extreme case of $i = k-1$, since in this case, the exact carry chain length, and thus its expected value, is 1. We conclude that carry chains are usually quite short.

On the average, the longest carry chain in adding k-bit numbers is of length $\log_2 k$. This was first observed and proved by Burks, Goldstine, and von Neumann in their classic report defining the structure of a stored-program computer [Burk46]. An interesting analysis based on Kolmogorov complexity theory has been offered in [Beig98]. The latter paper also cites past attempts at providing alternate or more complete proofs of the proposition.

Here is one way to prove the logarithmic average length of the worst-case carry chain. The reader can skip the rest of this section without any loss of continuity.

Let $\eta_k(h)$ be the probability that the longest carry chain in a k-bit addition is of length h or more. Clearly, the probability of the longest carry chain being of length exactly h is $\eta_k(h) - \eta_k(h+1)$. We can use a recursive formulation to find $\eta_k(h)$. The longest carry chain can be of length h or more in two mutually exclusive ways:

a. The least-significant $k-1$ bits have a carry chain of length h or more.
b. The least-significant $k-1$ bits do not have such a carry chain, but the most significant h bits, including the last bit, have a chain of the exact length h.

Thus, we have

$$\eta_k(h) \leq \eta_{k-1}(h) + 2^{-(h+1)}$$

where $2^{-(h+1)}$ is the product of 1/4 (representing the probability of carry generation) and $2^{-(h-1)}$ (probability that carry propagates across $h-2$ intermediate positions and stops in the last one). The inequality occurs because the second term is not multiplied by a probability as discussed above. Hence, assuming $\eta_i(h) = 0$ for $i < h$:

$$\eta_k(h) = \sum_{i=h}^{k} [\eta_i(h) - \eta_{i-1}(h)] \leq (k - h + 1)\, 2^{-(h+1)} \leq 2^{-(h+1)} k$$

To complete our derivation of the expected length λ of the longest carry chain, we note that

$$\begin{aligned}
\lambda &= \sum_{h=1}^{k} h[\eta_k(h) - \eta_k(h+1)] \\
&= [\eta_k(1) - \eta_k(2)] + 2[\eta_k(2) - \eta_k(3)] + \cdots + k[\eta_k(k) - 0] \\
&= \sum_{h=1}^{k} \eta_k(h)
\end{aligned}$$

We next break the final summation above into two parts: the first $\gamma = \lfloor \log_2 k \rfloor - 1$ terms and the remaining $k - \gamma$ terms. Using the upper bound 1 for terms in the first part and $2^{-(h+1)}k$ for terms in the second part, we get

$$\lambda = \sum_{h=1}^{k} \eta_k(h) \leq \sum_{h=1}^{\gamma} 1 + \sum_{h=\gamma+1}^{k} 2^{-(h+1)} k < \gamma + 2^{-(\gamma+1)} k$$

Now let $\varepsilon = \log_2 k - \lfloor \log_2 k \rfloor$ or $\gamma = \log_2 k - 1 - \varepsilon$, where $0 \leq \varepsilon < 1$. Then, substituting the latter expression for γ in the preceding inequality and noting that $2^{\log_2 k} = k$ and $2^{\varepsilon} < 1 + \varepsilon$, we get

$$\lambda < \log_2 k - 1 - \varepsilon + 2^{\varepsilon} < \log_2 k$$

This concludes our derivation of the result that the expected length of the worst-case carry chain in a k-bit addition with random operands is upper-bounded by $\log_2 k$. Experimental results verify the $\log_2 k$ approximation to the length of the worst-case carry chain and suggest that $\log_2(1.25k)$ is a better estimate [Hend61].

5.4 CARRY-COMPLETION DETECTION

A ripple-carry adder is the simplest and slowest adder design. For k-bit operands, both the worst-case delay and the implementation cost of a ripple-carry adder are linear in k.

However, based on the analysis in Section 5.3, the worst-case carry-propagation chain of length k almost never materializes.

A carry-completion detection adder takes advantage of the $\log_2 k$ average length of the longest carry chain to add two k-bit binary numbers in O(log k) time on the average. It is essentially a ripple-carry adder in which a carry of 0 is also explicitly represented and allowed to propagate between stages. The carry into stage i is represented by the two-rail code:

$$\begin{array}{rl} (b_i, c_i) = (0,0) & \text{Carry not yet known} \\ (0,1) & \text{Carry known to be 1} \\ (1,0) & \text{Carry known to be 0} \end{array}$$

Thus, just as two 1s in the operands generate a carry of 1 that propagates to the left, two 0s would produce a carry of 0. Initially, all carries are (0, 0) or unknown. After initialization, a bit position with $x_i = y_i$ makes the no-carry/carry determination and injects the appropriate carry $(b_{i+1}, c_{i+1}) = (\overline{x_i \vee y_i}, x_i y_i)$ into the carry-propagation chain of Fig. 5.9 via the OR gates. The carry $(\overline{c}_{\text{in}}, c_{\text{in}})$ is injected at the right end. When every carry has assumed one of the values (0, 1) or (1, 0), carry propagation is complete. The local "done" signals $d_i = b_i \vee c_i$ are combined by a global AND function into *alldone*, which indicates the end of carry propagation.

In designing carry-completion adders, care must be taken to avoid hazards that might lead to a spurious *alldone* signal. Initialization of all carries to 0 through clearing of input bits and simultaneous application of all input data is one way of ensuring hazard-free operation.

Excluding the initialization and carry-completion detection times, which must be considered and are the same in all cases, the latency of a k-bit carry-completion adder ranges from 1 gate delay in the best case (no carry propagation at all: i.e., when adding a number to itself) to $2k + 1$ gate delays in the worst case (full carry propagation from c_{in}

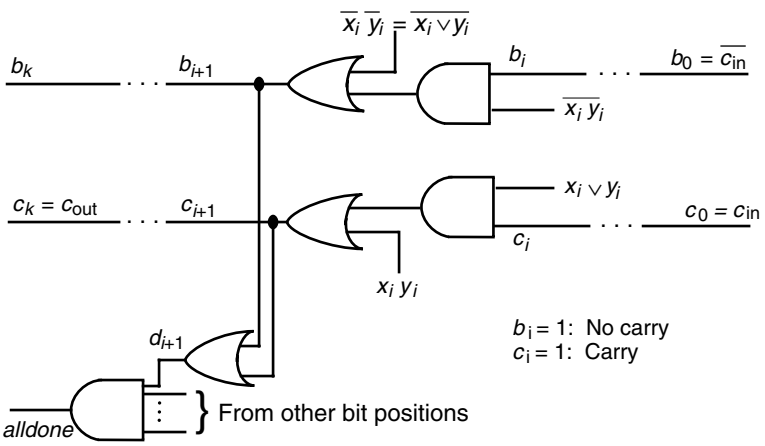


Figure 5.9 The carry network of an adder with two-rail carries and carry-completion detection logic.

to c_{out}), with the average latency being about $2 \log_2 k + 1$ gate delays. Note that once the final carries have arrived in all bit positions, the derivation of the sum bits is overlapped with completion detection and is thus not accounted for in the preceding latencies.

Because the latency of the carry-completion adder is data-dependent, the design of Fig. 5.9 is suitable for use in asynchronous systems. Most modern computers, however, use synchronous logic and thus cannot take advantage of the high average speed of a carry-completion adder.

5.5 ADDITION OF A CONSTANT: COUNTERS

When one input of the addition operation is a constant number, the design can be simplified or optimized compared with that of a general two-operand adder. With binary arithmetic, we can assume that the constant y to be added to x is odd, since in the addition $s = x + y_{\text{even}} = x + (y_{\text{odd}} \times 2^h)$, one can ignore the h rightmost bits in x and add y_{odd} to the remaining bits. The special case of $y = 1$ corresponds to standard counters, while $y = \pm 1$ yields an up/down counter.

Let the constant to be added to $x = (x_{k-1} \cdots x_2 x_1 x_0)_{\text{two}}$ be $y = (y_{k-1} \cdots y_2 y_1\ 1)_{\text{two}}$. The least-significant bit of the sum is $\overline{x}_0$. The remaining bits of s can be determined by a $(k-1)$-bit ripple-carry adder, with $c_{\text{in}} = x_0$, each of its cells being a HA ($y_i = 0$) or a modified HA ($y_i = 1$). The fast-adder designs to be covered in Chapters 6 and 7 can similarly be optimized to take advantage of the known bits of y.

When $y = 1(-1)$, the resulting circuit is known as an *incrementer* (*decrementer*) and is used in the design of up (down) counters. Figure 5.10 depicts an up counter, with parallel load capability, built of a register, an incrementer, and a multiplexer. The design shown in Fig. 5.10 can be easily converted to an up/down counter by using an incrementer/decrementer and an extra control signal. Supplying the details is left as an exercise.

Many designs for fast counters are available [Ober81]. Conventional synchronous designs are based on full carry propagation in each increment/decrement cycle, thus

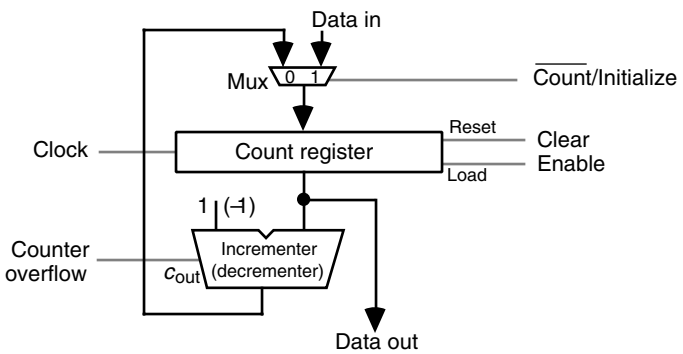


Figure 5.10 An up (down) counter built of a register, an incrementer (decrementer), and a multiplexer.

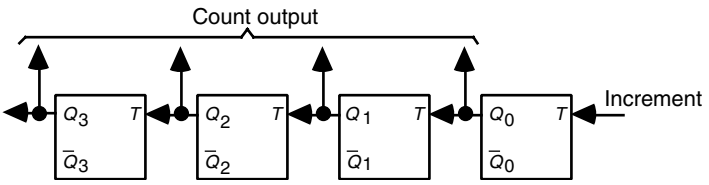


Figure 5.11 A 4-bit asynchronous up counter built only of negative-edge-triggered T flip-flops.

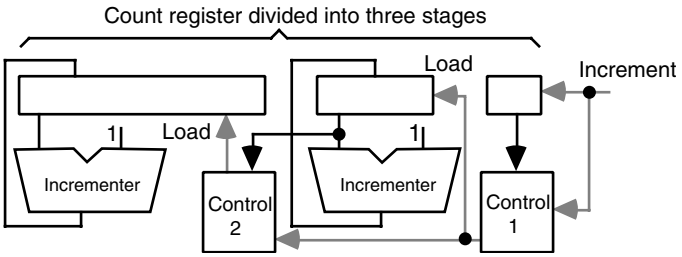


Figure 5.12 Fast three-stage up counter.

limiting the counter's operating speed. In some cases, special features of the storage elements used can lead to simplifications. Figure 5.11 depicts an asynchronous counter built of cascaded negative-edge-triggered T (toggle) flip-flops. Each input pulse toggles the flip-flop at the least significant position, each 1-to-0 transition of the least-significant bit flip-flop toggles the next flip-flop, and so on. The next input pulse can be accepted before the carry has propagated all the way to the left.

Certain applications require high-speed counting, with the count potentially becoming quite large. In such cases, a high-speed incrementer must be used. Methods of designing fast adders (Chapters 6 and 7) can all be adapted for building fast incrementers. When even the highest-speed incrementer cannot keep up with the input rate or when cost considerations preclude the use of an ultrafast incrementer, the frequency of the input can be reduced by applying it to a prescaler. The lower-frequency output of the prescaler can then be counted with less stringent speed requirements. In the latter case, the resulting count will be approximate.

Obviously, the count value can be represented in redundant format, allowing carry-free increment or decrement in constant time [Parh87]. However, with a redundant format, reading out the stored count involves some delay to allow for conversion of the internal representation to standard binary. Alternatively, one can design the counter as a cascade that begins with a very narrow, and thus fast, counter and continues with increasingly wider counters [Vuil91]. The wider counters on the left are incremented only occasionally and thus need not be very fast (their incremented counts can be precomputed by a slow incrementer and then simply loaded into the register when required). Figure 5.12 shows this principle applied to the design of a three-stage counter. Some details of this design, as well as its extension to up/down counting, will be explored in the end-of-chapter problems.

5.6 MANCHESTER CARRY CHAINS AND ADDERS

In the next three chapters, we will examine methods for speeding up the addition process for two operands (Chapters 6 and 7) and for multiple operands (Chapter 8). For two operands, the key to fast addition is a low-latency carry network, since once the carry into position i is known, the sum digit can be determined from the operand digits x_i and y_i and the incoming carry c_i in constant time through modular addtition:

$$s_i = (x_i + y_i + c_i) \bmod r$$

In the special case of radix 2, the relation above reduces to

$$s_i = x_i \oplus y_i \oplus c_i$$

So, the primary problem in the design of two-operand adders is the computation of the k carries c_{i+1} based on the $2k$ operand digits x_i and $y_i, 0 \leq i < k$.

From the point of view of carry propagation and the design of a carry network, the actual operand digits are not important. What matters is whether in a given position a carry is generated, propagated, or annihilated (absorbed). In the case of binary addition, the *generate*, *propagate*, and *annihilate* (*absorb*) signals are characterized by the following logic equations:

$$\begin{aligned} g_i &= x_i y_i \\ p_i &= x_i \oplus y_i \\ a_i &= \overline{x}_i \overline{y}_i = \overline{x_i \vee y_i} \end{aligned}$$

It is also helpful to define a *transfer* signal corresponding to the event that the carry-out will be 1, given that the carry-in is 1:

$$t_i = g_i \vee p_i = \overline{a}_i = x_i \vee y_i$$

More generally, for radix r, we have

$$\begin{aligned} g_i &= 1 \quad \text{iff } x_i + y_i \geq r \\ p_i &= 1 \quad \text{iff } x_i + y_i = r - 1 \\ a_i &= 1 \quad \text{iff } x_i + y_i < r - 1 \end{aligned}$$

Thus, assuming that the signals above are produced and made available, the rest of the carry network design can be based on them and becomes completely independent of the operands or even the number representation radix.

Using the preceding signals, the *carry recurrence* can be written as follows:

$$c_{i+1} = g_i \vee c_i p_i$$

The carry recurrence essentially states that a carry will enter stage $i+1$ if it is generated in stage i or it enters stage i and is propagated by that stage. Since

$$\begin{aligned} c_{i+1} &= g_i \vee c_i p_i = g_i \vee c_i g_i \vee c_i p_i \\ &= g_i \vee c_i (g_i \vee p_i) = g_i \vee c_i t_i \end{aligned}$$

the carry recurrence can be written in terms of t_i instead of p_i. This latter version of the carry recurrence leads to slightly faster adders because in binary addition, t_i is easier to produce than p_i (OR instead of XOR).

In what follows, we always deal with the carry recurrence in its original form $c_{i+1} = g_i \vee c_i p_i$, since it is more intuitive, but we keep in mind that in most cases, p_i can be replaced by t_i if desired.

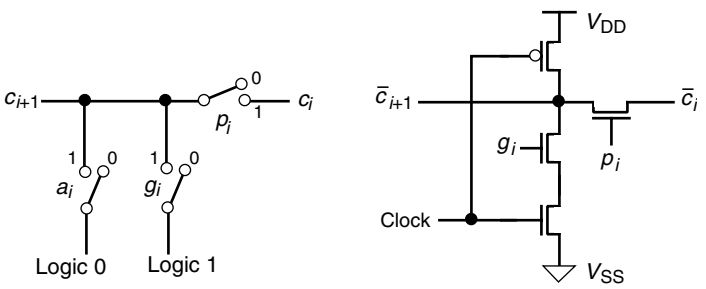
The carry recurrence forms the basis of a simple carry network known as *Manchester carry chain*. A *Manchester adder* is one that uses a Manchester carry chain as its carry network. Each stage of a Manchester carry chain can be viewed as consisting of three switches controlled by the signals p_i, g_i, and a_i, so that the switch closes (conducts electricity) when the corresponding control signal is 1. As shown in Fig. 5.13a, the carry-out signal c_{i+1} is connected to 0 if $a_i = 1$, to 1 if $g_i = 1$, and to c_i if $p_i = 1$, thus assuming the correct logical value $c_{i+1} = g_i \vee c_i p_i$. Note that one, and only one, of the signals p_i, g_i, and a_i is 1.

Figure 5.13b shows how a Manchester carry chain might be implemented in CMOS. When the clock is low, the c nodes precharge. Then, when the clock goes high, if g_i is high, c_{i+1} is asserted or drawn low. To prevent g_i from affecting c_i, the signal p_i must be computed as the XOR (rather than OR) of x_i and y_i. This is not a problem because we need the XOR of x_i and y_i for computing the sum anyway.

For a k-bit Manchester carry chain, the total delay consists of three components:

1. The time to form the switch control signals.
2. The setup time for the switches.
3. Signal propagation delay through k switches in the worst case.

The first two components of delay are small, constant terms. The delay is thus dominated by the third component, which is at best linear in k. For modern CMOS technology, the delay is roughly proportional to k^2 (as k pass transistors are connected in series),



(a) Conceptual representation. **(b)** Possible CMOS realization.

Figure 5.13 One stage in a Manchester carry chain.

making the method undesirable for direct realization of fast adders. However, when the delay is in fact linear in k, speed is gained over gate-based ripple-carry adders because we have one switch delay rather than two gate delays per stage. The linear or superlinear delay of a Manchester carry chain limits its usefulness for wide words or in high-performance designs. Its main application is in implementing short chains (say, up to 8 bits) as building blocks for use with a variety of fast addition schemes and certain hybrid designs.

We conclude this chapter by setting the stage for fast addition schemes to follow in Chapters 6 and 7. Taking advantage of generate and propagate signals defined in this section, an adder design can be viewed in the generic form of Fig. 5.14. Any adder will have the two sets of AND and XOR gates at the top to form the g_i and p_i signals, and it will have a set of XOR gates at the bottom to produce the sum bits s_i. It will differ, however, in the design of its carry network, which is represented by the large oval block in Fig. 5.14. For example, a ripple-carry adder can be viewed as having the carry network shown in Fig. 5.15. Inserting this carry network into the generic design

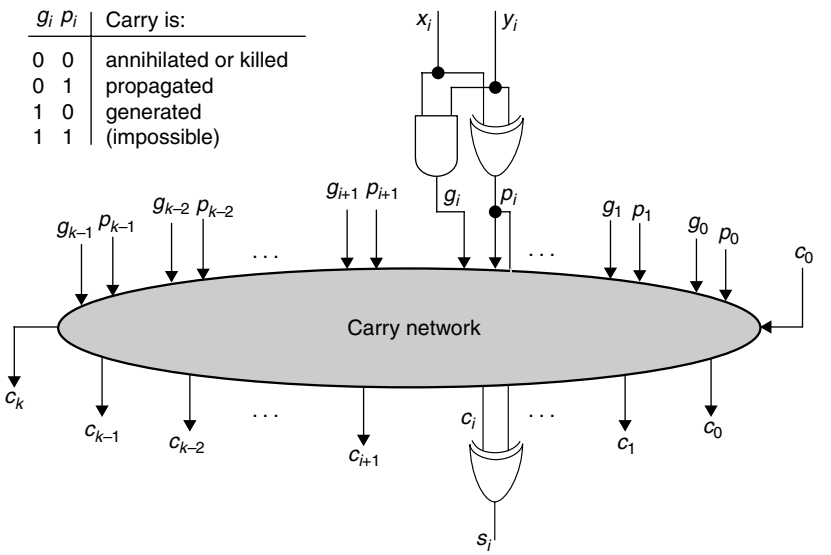


Figure 5.14 Generic structure of a binary adder, highlighting its carry network.

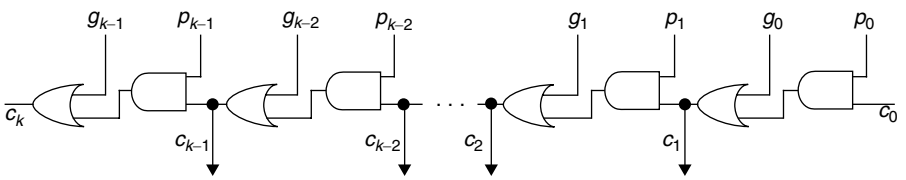


Figure 5.15 Alternate view of a ripple-carry network in connection with the generic adder structure shown in Fig. 5.14.

of Fig. 5.14 will produce a complete adder. Thus, in our subsequent discussions, we will focus on different designs for the carry network, and we will compare adders with respect to latency and cost of the carry network only.

PROBLEMS

5.1 Bit-serial 2's-complement adder

Present the complete design of a bit-serial 2's-complement adder for 32-bit numbers. Include in your design the control details and provisions for overflow detection.

5.2 Four-function ALU

Extend the design of Fig. 5.2c into a bit-slice for a four-function ALU that produces any of the following functions of the inputs x and y based on the values of two control signals: Sum, OR, AND, XOR. *Hint:* What happens if c_{in} is forced to 0 or 1?

5.3 Subtractive adder for 1's-complement numbers

Show that the alternate representation of 0 in 1's complement, which is obtained only when x and $-x$ are added, can be avoided by using a "subtractive adder" that always complements y and performs subtraction to compute $x + y$.

5.4 Digit-serial adders

a. A radix-2^g digit-serial adder can be faster than a bit-serial adder. Show the detailed design of a radix-16 digit-serial adder for 32-bit unsigned numbers and compare it with respect to latency and cost to bit-serial and ripple-carry binary adders.
b. Design a digit-serial binary-coded decimal (BCD) adder to add decimal numbers whose digits are encoded as 4-bit binary numbers.
c. Combine the designs of parts a and b into an adder than can act as radix-16 or BCD adder according to the value of a control signal.

5.5 Binary adders as versatile building blocks

A 4-bit binary adder can be used to implement many logic functions besides its intended function. An example appears in Fig. 5.6. Show how a 4-bit binary adder, and nothing else, can be used to realize the following:

a. A 3-bit adder, with carry-in and carry-out.
b. Two independent single-bit FAs.
c. A single-bit FA and a 2-bit binary adder operating independently.
d. A 4-bit odd parity generator (4-bit XOR).
e. A 4-bit even or odd parity generator under the control of an even/$\overline{\text{odd}}$ signal.
f. Two independent 3-bit odd parity generators.
g. A five-input AND circuit.
h. A five-input OR circuit.

i. A circuit to realize the four-variable logic function $wx \vee yz$.
j. A circuit to realize the four-variable logic function $wx\bar{y} \vee wx\bar{z} \vee \bar{w}yz \vee \bar{x}yz$.
k. A multiply-by-15 circuit for a 2-bit number x_1x_0, resulting in a 6-bit product.
l. A circuit to compute $x + 4y + 8z$, where x, y, and z are 3-bit unsigned numbers.
m. A five-input "parallel counter" producing the sum $s_2s_1s_0$ of five 1-bit numbers.

5.6 Binary adders as versatile building blocks

Show how an 8-bit binary adder can be used to realize the following:

a. Three independent 2-bit binary adders, each with carry-in and carry-out.
b. A circuit to realize the six-variable logic function $uv \vee wx \vee yz$.
c. A circuit to compute $2w + x$ and $2y + z$, where w, x, y, z are 3-bit numbers.
d. A multiply-by-85 circuit for a number $x_3x_2x_1x_0$, resulting in an 11-bit product.
e. A circuit to compute the 5-bit sum of three 3-bit unsigned numbers.
f. A seven-input "parallel counter" producing the sum $s_2s_1s_0$ of seven 1-bit numbers.

5.7 Decimal addition

Many microprocessors provide an 8-bit unsigned "add with carry" instruction that is defined as unsigned addition using the "carry flag" as c_{in} and producing two carries: carry-out or c_8, stored in the carry flag, and "middle carry" or c_4, stored in a special flag bit for subsequent use (e.g., as branch condition). Show how the "add with carry" instruction can be used to construct a routine for adding unsigned decimal numbers that are stored in memory with two BCD digits per byte.

5.8 2's-complement adder

a. Prove that in adding k-bit 2's-complement numbers, overflow occurs if and only if $c_{k-1} \neq c_k$.
b. Show that in a 2's-complement adder that does not provide c_{out}, we can produce it externally using $c_{\text{out}} = x_{k-1}y_{k-1} \vee \bar{s}_{k-1}(x_{k-1} \vee y_{k-1})$.

5.9 Carry-completion adder

The computation of a k-input logic function requires O(log k) time if gates with constant fan-in are used. Thus, the AND gate in Fig. 5.9 that generates the *alldone* signal is really a tree of smaller AND gates that implies O(log k) delay. Wouldn't this imply that the addition time of the carry completion adder is O($\log^2 k$) rather than O(log k)?

5.10 Carry-completion adder

a. Design the sum logic for the carry-completion adder of Fig. 5.9.
b. Design a carry-completion adder using FAs and HAs plus inverters as the only building blocks (besides the completion detection logic).

c. Repeat part a if the sum bits are to be obtained with two-rail (z, p) encoding whereby 0 and 1 are represented by (1, 0) and (0, 1), respectively. In this way, the sum bits are independently produced as soon as possible, allowing them to be processed by other circuits in an asynchronous fashion.

5.11 Balanced ternary adder

Consider the balanced ternary number system with $r = 3$ and digit set $[-1, 1]$. Addition of such numbers involves carries in $\{-1, 0, 1\}$. Assuming that both the digit set and carries are represented using the (n, p) encoding of Fig. 3.7:

a. Design a ripple-carry adder cell for balanced ternary numbers.
b. Convert the adder cell of part a to an adder/subtractor with a control input.
c. Design and analyze a carry-completion sensing adder for balanced ternary numbers.

5.12 Synchronous binary counter

Design a synchronous counterpart for the asynchronous counter shown in Fig. 5.11.

5.13 Negabinary up/down counter

Design an up/down counter based on the negabinary (radix -2) number representation in the count register. *Hint:* Consider the negabinary representation as a radix-4 number system with the digit set $[-2, 1]$.

5.14 Design of fast counters

Design the two control circuits in Fig. 5.12 and determine optimal lengths for the three counter segments, as well as the overall counting latency (clock period), in each of the following cases. Assume the use of ripple-carry incrementers.

a. An overall counter length of 48 bits.
b. An overall counter length of 80 bits.

5.15 Fast up/down counters

Extend the fast counter design of Fig. 5.12 to an up/down counter. *Hint:* Incorporate the sign logic in "Control 1," use a fast 0 detection mechanism, and save the old value when incrementing a counter stage.

5.16 Manchester carry chains

Study the effects of inserting a pair of inverters after every g stages in a CMOS Manchester carry chain (Fig. 5.13b). In particular, discuss whether the carry-propagation time can be made linear in k by suitable placement of the inverter pairs.

5.17 Analysis of carry-propagation

In deriving the average length of the worst-case carry-propagation chain, we made substitutions and simplifications that led to the upper bound $\log_2 k$. By deriving an O(log k) lower bound, show that the exact average is fairly close to this upper bound.

5.18 Binary adders as versatile building blocks

Show how to use a 4-bit binary adder as:

a. A multiply-by-3 circuit for a 4-bit unsigned binary number.
b. Two independent 3-input majority circuits implementing 2-out-of-3 voting.
c. Squaring circuit for a 2-bit binary number.

5.19 Negabinary adder or subtractor

Derive algorithms and present hardware structures for adding or subtracting two negabinary, or radix-(-2), numbers.

5.20 Carry-propagation chains

Consider the addition of two k-bit unsigned binary numbers whose bit positions are indexed from $k-1$ to 0 (most to least significant). Assume that the bit values in the two numbers are completely random.

a. What is the probability that a carry is generated in bit position i?
b. What is the probability that a carry generated in bit position i affects the sum bit in position j, where $j > i$? The answer should be derived and presented as a function of i and j.
c. What is the probability that a carry chain starting at bit position i will terminate at bit position j? *Hint:* For this to happen, position j must either absorb the carry or generate a carry of its own.
d. What is the probability that the incoming carry c_{in} propagates all the way to the most significant end and affects the outgoing carry c_{out}?
e. What is the expected length of a carry chain that starts in bit position i? Fully justify your answer and each derivation step.

5.21 FA hardware realization

Realize a FA by means of a minimum number of 2-to-1 multiplexers and no other logic component, not even inverters [Jian04].

5.22 Latency of a ripple-carry adder

A ripple-carry adder can be implemented by inserting the FA design of Fig. 5.2a or Fig. 5.2b into the k-stage cascade of Fig. 5.5. This leads to four different designs, given that HAs can take one of the three forms shown in Fig. 5.1. A fifth design can be based on Figs. 5.14 and 5.15. Compare these implementations with respect to latency and hardware complexity, using reasonable assumptions about gate delays and costs.

5.23 Self-dual logic functions

The dual of a logic function $f(x_1, x_2, \ldots, x_n)$ is another function $g(x_1, x_2, \ldots, x_n)$ such that the value of g with all n inputs complemented is the complement of f with uncomplemented inputs. A logic function f is self-dual if $f = g$. Thus, complementing all inputs of a logic circuit implementing the self-dual logic function f will lead to its output being complemented. Self-dual functions have applications in the provision of fault tolerance in digital systems via time redundancy (recomputation with complemented inputs, and comparison).

a. Show that binary HAs and FAs are self-dual with respect to both outputs.
b. Is a k-bit 1's-complement binary adder, with $2k + 1$ inputs and $k + 1$ outputs, self-dual?
c. Repeat part b for a 2's-complement adder.

5.24 FA with unequal input arrival times

Show that a FA can be designed such that if its 3 input bits arrive at times u, v, and w, with $u \leq v \leq w$, it will produce the sum output bit at time $max(v + 2, w + 1)$ and its carry-out bit at time $w + 1$, where the unit of time is the delay of an XOR gate [Stel98].

REFERENCES AND FURTHER READINGS

[Beig98] Beigel, R., B. Gasarch, M. Li, and L. Zhang, "Addition in $\log_2 n + O(1)$ Steps on Average: A Simple Analysis," *Theoretical Computer Science*, Vol. 191, Nos. 1–2, pp. 245–248, 1998.

[Bui02] Bui, H. T., Y. Wang, and Y. Jiang, "Design and Analysis of Low-Power 10-Transistor Full Adders Using Novel XOR-XNOR Gates," *IEEE Trans. Circuits and Systems II*, Vol. 49, No. 1, pp. 25–30, 2002.

[Burk46] Burks, A. W., H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Institute for Advanced Study, Princeton, NJ, 1946.

[Gilc55] Gilchrist, B., J. H. Pomerene, and S. Y. Wong, "Fast Carry Logic for Digital Computers," *IRE Trans. Electronic Computers*, Vol. 4, pp. 133–136, 1955.

[Hend61] Hendrickson, H. C., "Fast High-Accuracy Binary Parallel Addition," *IRE Trans. Electronic Computers*, Vol. 10, pp. 465–468, 1961.

[Jian04] Jiang, Y., A. Al-Sheraidah, Y. Wang, E. Sha, and J.-G. Chung, "A Novel Multiplexer-Based Low-Power Full Adder," *IEEE Trans. Circuits and Systems II*, Vol. 51, No. 7, pp. 345–353, 2004.

[Kilb60] Kilburn, T., D. B. G. Edwards, and D. Aspinall, "A Parallel Arithmetic Unit Using a Saturated Transistor Fast-Carry Circuit," *Proc. IEE*, Vol. 107B, pp. 573–584, 1960.

[Laps97] Lapsley, P., *DSP Processor Fundamentals: Architectures and Features*, IEEE Press, 1997.

[Lin07] Lin, J. F., Y.-T. Hwang, M.-H. Sheu, and C.-C. Ho, "A Novel High-Speed and Energy Efficient 10-Transistor Full Adder Design," *IEEE Trans. Circuits and Systems I*, Vol. 54, No. 5, pp. 1050–1059, 2007.

[Ober81] Oberman, R. M. M., *Counting and Counters*, Macmillan, London, 1981.

[Parh87] Parhami, B., "Systolic Up/Down Counters with Zero and Sign Detection," *Proc. Symp. Computer Arithmetic*, pp. 174–178, 1987.

[Puck94] Pucknell, D. A., and K. Eshraghian, *Basic VLSI Design*, 3rd ed., Prentice-Hall, 1994.

[Stel98] Stelling, P. F., C. U. Martel, V. G. Oklobdzija, and R. Ravi, "Optimal Circuits for Parallel Multipliers," *IEEE Trans. Computers*, Vol. 47, No. 3, pp. 273–285, 1998.

[Vuil91] Vuillemin, J. E., "Constant Time Arbitrary Length Synchronous Binary Counters," *Proc. Symp. Computer Arithmetic*, pp. 180–183, 1991.