CHAPTER 5
Fixed-Point Addition

They were rescued three days later, emaciated but beaming with joy:
they had just invented a new method, deduced from the logarithmic spiral,
for adding one-digit integers.

Topfer

Integer addition is without doubt the most versatile arithmetic operation. Integer and fixed-point adders will be used as building blocks in most chapters of this book. The present chapter reviews the main techniques for adder construction, from simple but slow adders to larger and faster ones. Integer addition also received special attention in most FPGA architectures, and this chapter reviews in depth the specificities of addition in FPGAs.

Application-specific arithmetic requires efficient adders of all sizes, from a few bits to a few hundred bits. Small adders are required for low-precision signal processing, but also, for instance, in counters used for datapath control and in the exponent processing of floating-point computations – exponents sizes typically vary between 5 bits, for a dynamic range of $[10^{-6}, 10^{6}]$, and 16 bits, for a dynamic range beyond $[10^{-10,000}, 10^{10,000}]$.

Larger adders (32 to 128 bits) are used everywhere. A modern microprocessor routinely performs several integer 32-bit or 64-bit additions per cycle: for address computation (on the program counter, the stack counter, the various addressing modes, etc); to process loop counters; and of course to process integer or fixed-point data. Besides, integer adders are used to build more complex operators, in particular multipliers and dividers of integer or floating-point data. Most techniques for the evaluation of numerical functions (reviewed in Part III of this book) also involve additions of various sizes. In short, integer or fixed-point addition is used as a building block in virtually all the coarser operators.

F. de Dinechin, M. Kumm, *Application-Specific Arithmetic*,
https://doi.org/10.1007/978-3-031-42808-1_5

Very large (more than 128 bits) adders are used in the construction of modular adders for cryptography. In cryptography based on the Rivest-Shamir-Adleman (RSA) algorithm, the acceptable key sizes are beyond 1024 bits. Elliptic-curve cryptography works with smaller data sizes, but even the acceptable security requires more than 150 bits [Oka+00].

Integer addition is so pervasive that it is generally supported by hardware description languages. One may just write a + in VHDL or Verilog, and synthesis tools will instantiate an adder.

In many cases (small additions, and in general additions that are not in the critical path), one does not need to worry about the implementation of such an addition. The most area-efficient adder implementation (a ripple-carry adder; see Sect. 5.2.2) will be adequate, and synthesis tools will infer just that. On FPGAs, the ripple-carry adder benefits from dedicated carry logic that makes it very efficient even in terms of speed.

Conversely, when addition speed is critical, it is possible to design faster (but more expensive) adders. The approaches used here are quite different when targeting ASIC or FPGA.

ASIC synthesis tools may use various techniques that reduce the addition latency at the expense of area. These techniques include transistor-level improvements to the carry-propagation path, but also a set of algorithmic techniques referred to in the literature as *fast adders*. These techniques will be reviewed in Sect. 5.3.

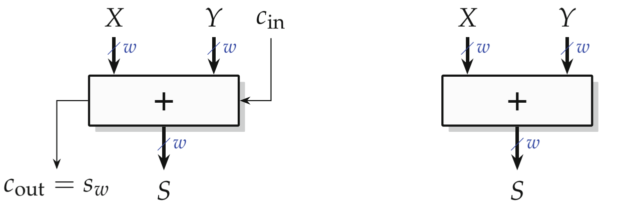
On FPGAs, classic fast adder techniques are mostly ineffective, due to the performance discrepancy between fast carry routing and general routing. The frequency of large ripple-carry adders may be improved by pipelining, which will be reviewed in Sect. 5.4. FPGA-specific fast addition techniques have also been studied and will be surveyed in Sect. 5.5.

5.1 Fixed-Point Considerations

In Chap. 2, integers have been presented as a special case of fixed-point representation. In this section, we first discuss integer addition and then generalize it to fixed-point addition.

5.1.1 *Unsigned Integer Addition*

An unsigned integer on w bits belongs to the interval $[0, 2^w - 0$. The sum of two such integers therefore belongs to $[0, 2^{w+1} - 2]$. It fits a $w + 1$-bit integer, and there is even room for adding one more unit to fill the interval $[0, 2^{w+1} - 0$. This defines the generic binary integer adder depicted in Fig. 5.1a: it inputs two w-bit unsigned integers $X = \sum_{i=0}^{w-1} 2^i x_i$ and $Y =$



(a) *generic adder* (b) *modulo* 2^w *adder*

Fig. 5.1 Functional views of the integer adder.

$\sum_{i=0}^{w-1} 2^i y_i$, and an input carry bit c_{in} of weight 1. It computes $X + Y + c_{\text{in}}$ exactly, as a $(w+1)$-bit integer $S = \sum_{i=0}^{w} 2^i s_i$. The most significant output bit s_w is usually called the *output carry* and denoted as c_{out}. It is the case in Fig. 5.1a.

When this carry-out bit is ignored, the operation computed by the adder is $X + Y + c_{\text{in}} \mod 2^w$. Figure 5.1b shows an adder with only two inputs and one output, all of the same size. It is a special case of Fig. 5.1a where $c_{\text{in}} = 0$ and c_{out} is dropped, and it computes $X + Y \mod 2^w$.

In some applications, in particular in signal processing, the interface seen in Fig. 5.1b is imposed, but *saturation* is preferred to this modulo behavior. A saturated adder returns the maximal representable value in case of overflow. It can be built on top of a generic adder: a multiplexer, controlled by c_{out}, outputs the w-bit sum if $c_{\text{out}} = 0$ and outputs 2^{w-1} if $c_{\text{out}} = 1$.

5.1.2 *Signed Integer Addition*

A signed integer on w bits belongs to the interval $[-2^{w-1}, 2^{w-1} - 0$. The sum of two such integers therefore belongs to $[-2^w, 2^w - 2]$. Again it fits a $w + 1$-bit signed integer, with the possibility to add a unit carry-in. The management of signs in integer addition will be detailed in Sect. 5.2.3, which will also show how subtraction also reduces to addition in two's complement.

5.1.3 *Fixed-Point Addition*

An unsigned fixed-point numbers in the format ufix(m, ℓ) is an integer of size $w = m - \ell + 1$ scaled by 2^ℓ (see Sect. 2.2.1, p. 38). Therefore, the addition of two unsigned fixed-point numbers X and Y of the same format ufix(m, ℓ) can be reduced to an integer addition:

$$S \;=\; 2^{\ell} X_{\text{int}} + 2^{\ell} Y_{\text{int}} \;=\; 2^{\ell}(X_{\text{int}} + Y_{\text{int}}) \tag{5.1}$$

Since $X_{\text{int}} + Y_{\text{int}}$ is a $w+1$-bit integer, the exact fixed-point sum is an unsigned number in ufix$(m+1, \ell)$: it has one bit more at the most significant bit (MSB) than the inputs.

The addition of two numbers of different formats $X \in \text{ufix}(m_X, \ell_X)$ and $Y \in \text{ufix}(m_Y, \ell_Y)$ is best understood by first converting them to a larger format which allows for an exact addition (see Fig. 5.2). This requires sign extension if the numbers are signed (see Sect. 2.2.3, p. 42). Defining $m = \max(m_X, m_Y) + 1$ and $\ell = \min(\ell_X, \ell_Y)$, then the exact sum can be represented in a ufix(m, ℓ) format.

If the sum S is expected in a smaller format ufix(m_S, ℓ_S), then overflow may happen if $m_S < m$ and rounding must happen if $\ell_S > \ell$.

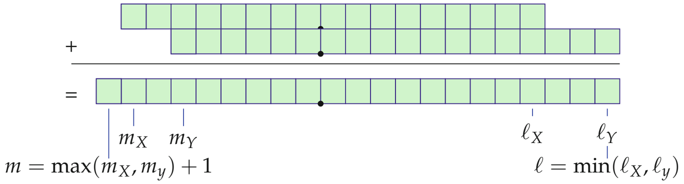


Fig. 5.2 Exact addition of two unsigned fix-point numbers of different formats.

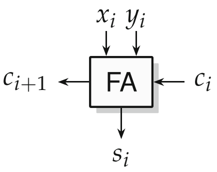
Since fixed-point addition can in all cases be reduced to integer addition, the remainder of this chapter focuses on integer addition.

5.2 Addition Basics

5.2.1 The Full Adder and Half Adder Cells

The full adder (FA) performs the analogous to a one-digit step of the paper-and-pencil addition algorithm. It computes the sum of its three input bits x_i, y_i, and c_i, where i denotes the bit position corresponding to weight 2^i. This sum is between 0 and 3 and may therefore be rewritten as a 2-bit binary number $c_{i+1}s_i$ where c_{i+1} is the most significant bit and s_i the least significant bit. Figure 5.3 describes the black box of the full adder (FA) and its truth table. In general c_i will be referred to as the input carry, and c_{i+1} is the output carry.

Looking at the truth table, the sum bit is the exclusive OR of the three inputs. The carry bit is the majority function, which is true when more than half of the inputs are true. This logic function can also be defined as the basic Boolean expressions



x_i	y_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Fig. 5.3 The full adder.

$$s_i = x_i \oplus y_i \oplus c_i \tag{5.2}$$
$$c_{i+1} = x_i c_i \vee y_i c_i \vee x_i y_i. \tag{5.3}$$

We will sometimes denote the full adder according to (5.2) and (5.3) as

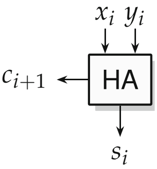
$$(c_{i+1}, s_i) = \mathrm{FA}(x_i, y_i, c_i). \tag{5.4}$$

As the FA performs the sum of its three inputs, these three inputs are functionally equivalent. However, in general, one wishes to reduce the delay of a complete w-bit addition, which happens when the carry is propagated. In the full adder cell, this means accelerating the path from c_i to c_{i+1}. Therefore, in terms of implementation, the three inputs are usually not equivalent.

There are many Boolean expressions equivalent to (5.3) and even more transistor-level implementations of these expressions. Actually, the FA cell is so important that founders provide several full-custom implementations for their technologies. Several papers and patents are published every year about the design of FA cells [ZW92; AB95; AB95; SB00; BWJ03; Bha+15].

In case one input is zero, the full adder reduces to a half adder (HA). Figure 5.4 shows the HA, and its truth table, which is identical to the FA for $c_i = 0$. Its sum becomes an XOR and its carry-out simplifies to an AND:

$$s_i = x_i \oplus y_i \tag{5.5}$$
$$c_{i+1} = x_i y_i. \tag{5.6}$$



x_i	y_i	c_{i+1}	s_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Fig. 5.4 The half adder.

5.2.2 *Ripple-Carry Addition*

The most area-efficient (but also slowest) adder is the ripple-carry adder (RCA) shown in Fig. 5.5.

The critical path of this design goes from one of the inputs of the rightmost FA (corresponding to the least significant bit) to one of the outputs of the leftmost one (most significant bit). In general, the latency of such an adder is linear in the number of bits w

$$\tau_{\text{add}}(w) = (w-1)\tau_{\text{cp}} + \max(\tau_{\text{cp}}, \tau_{\text{fa}}) \tag{5.7}$$

where τ_{fa} is the delay from inputs to outputs of a full adder cell and τ_{cp} is the carry propagate delay (from c_i to c_{i+1}). The RCA belongs to the class of carry propagate adders (CPAs). Other, faster types of CPAs are discussed in Sect. 5.3.

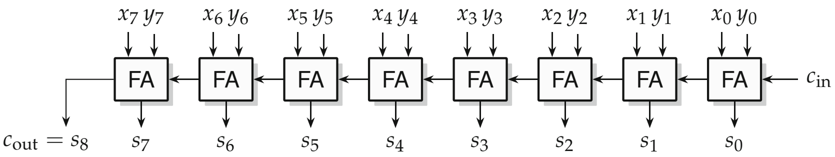


Fig. 5.5 An 8-bit ripple-carry adder.

5.2.3 *Addition of Signed Numbers in Two's Complement*

If the carry-in bit is set to 0 and the carry-out bit is ignored, the adder of Fig. 5.5 inputs two w-bit vectors X and Y and outputs a w-bit vector S. When these three vectors X, Y, and S are interpreted as unsigned w-bit integers (in $\text{ufix}(w-1,0)$ format), then this operator computes in S the sum modulo 2^w of X and Y.

This property also holds if X, Y, and S are interpreted as *signed* w-bit integers (in $\text{sfix}(w-1,0)$ format). Indeed, two's complement arithmetic on w bits is really modulo 2^w arithmetic (see Fig. 2.2), with the choice that the numbers with their MSB set represent negative numbers instead of large positive ones.

This is actually the main advantage of two's complement: the adder of Fig. 5.5 can be used unmodified to add signed numbers.

The only difference lies in the interpretation of the carry-out bit:

- For unsigned arithmetic, a carry-out of $c_{\text{out}} = 1$ indicates an overflow.
- For signed arithmetic, there will be an overflow if the two inputs have the same sign bit and the output has a different sign bit. Otherwise, if the two

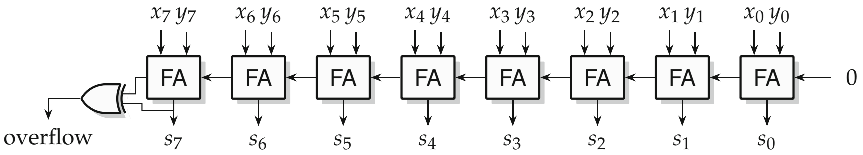


Fig. 5.6 An 8-bit adder for two's complement numbers. The FA inputting a zero carry on the right can be replaced by a less expensive HA.

inputs have different sign bits, then their sum cannot overflow. A classic result is that an overflow bit can be computed as the XOR of s_{w-1} (the sign of the result) and $s_w = c_{\text{out}}$, as illustrated by Fig. 5.6. The proof is by enumeration of the various cases.

5.2.4 Basic Subtraction

Similar to the FA, one can derive a primitive circuit for performing subtractions which is called a *full subtractor*. This cell computes the difference as well as a *borrow* bit that works similar to the carry out, but has a negative weight.

However, most subtractors are built using a different approach. The subtraction operation can also be seen as the addition of the negative subtrahend, i.e., $X - Y = X + (-Y)$.

In two's complement, $-Y$ is computed as $\overline{Y} + 1$ where $\overline{Y}$ denotes the bitwise complement of Y. Technically, this is realized by inverting Y and adding a "1" to the least significant bit (LSB) position. This "1" can be added for free by using the carry-in of the adder, as illustrated by Fig. 5.7. Overflow detection logic does not appear on this figure, because it depends on the signedness of the inputs.

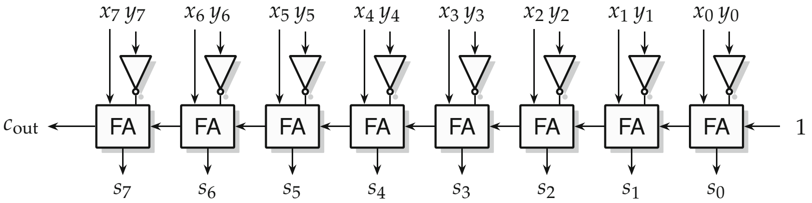


Fig. 5.7 An 8-bit subtractor computing a result in two's complement.

5.2.5 Reconfigurable Adder/Subtractor

The idea of the subtractor can be further generalized to an adder/subtractor. An adder/subtractor inputs an additional select signal sub and computes

$$S = \begin{cases} X + Y & \text{when sub} = 0 \\ X - Y & \text{when sub} = 1 \end{cases} . \tag{5.8}$$

In the case of subtraction, the two's complement of the Y input has to be computed: Y has to be bit-wise inverted, and a "1" has to be added at the LSB. The conditional inversion can be realized by a bit-wise XOR between y_i and sub. The conditional +1 can be realized by adding the "sub" signal using the carry-in as shown in Fig. 5.8.

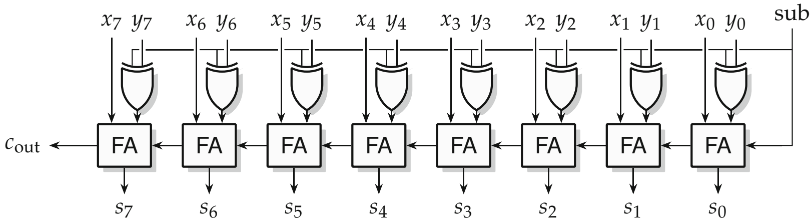


Fig. 5.8 Adder/subtractor computing $S = X + (-1)^{\text{sub}}Y$.

The main take-away message of Sects. 5.2.3, 5.2.4, and 5.2.5 is that thanks to two's complement, both addition and subtraction of signed or unsigned numbers reduce to the basic ripple-carry adder. They can therefore all benefit from the fast addition techniques discussed in the sequel.

5.2.6 Carry-Save Addition

The simplest variant of fast addition is actually a simplification of the ripple-carry adder: the *carry-save* adder depicted in Fig. 5.9 outputs all the carries instead of propagating them from one FA cell to the next. The unused carry-in signals of each FA cell together form a third binary input $Z = \sum_{i=0}^{w-1} 2^i z_i$. Thus, a w-bit carry-save adder inputs three w-bit numbers and outputs their sum as two w-bit numbers S and C.

In Fig. 5.9, the indices of all the bits correspond to their bit positions. There is no c_0 in Fig. 5.9, because the carry output of the rightmost FA cell already has weight 2. If we define $c_0 = 0$, then the bits c_i together define an integer $C = \sum_{i=0}^{w} 2^i c_i$. The relationship between the inputs and outputs of

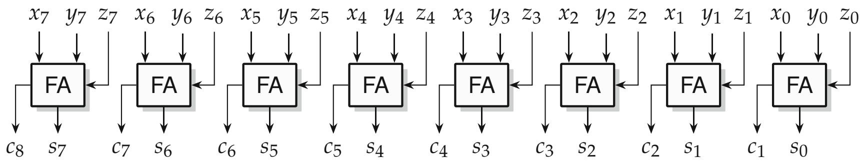

x_7 y_7 z_7 x_6 y_6 z_6 x_5 y_5 z_5 x_4 y_4 z_4 x_3 y_3 z_3 x_2 y_2 z_2 x_1 y_1 z_1 x_0 y_0 z_0
FA
c_8 s_7 c_7 s_6 c_6 s_5 c_5 s_4 c_4 s_3 c_3 s_2 c_2 s_1 c_1 s_0

Fig. 5.9 An 8-bit carry-save adder.

the carry-save adder is then

$$C + S = X + Y + Z. \tag{5.9}$$

This addition is obviously not complete, since the output $C + S$ is still an unevaluated sum. However, the carry-save adder has definitely performed one addition, since there were two additions in $X + Y + Z$, and only one remains in the result $C + S$. Besides, this addition was performed with minimal hardware and with a delay independent of the addition size.

A number kept as an unevaluated sum $C + S$ is said to be in carry-save form. Carry-save is not a very interesting final format to express numbers, because it requires twice as many bits as standard binary to express the same range and because it is very redundant: there are many ways to write most integers in carry-save form,[1] which makes some operations such as comparisons very difficult. Actually the best way to compare two carry-save numbers is to first convert them to binary (by performing the addition $C + S$).

However, carry-save can be very useful for intermediate results, in particular in iterative algorithms: the use of carry-save form may allow for an iteration without carry propagation, thus dramatically reducing the iteration time. Classic examples include multipliers (see Chap. 8) and dividers (see Chap. 9).

A variant of carry-save consists in keeping α-bit carry propagation, outputting carries every α bits only, as illustrated in Fig. 5.10. This is called high-radix carry-save (HRCS), since the carries that are output are those of a radix-2^α addition (see Sect. 2.3). For instance, for $\alpha = 4$, what we have is a hexadecimal carry-save adder. This is also called *partial redundant form* [FO01].

HRCS is less redundant than carry-save[2] by having fewer carry bits. This may be beneficial if these carries are to be stored in registers. However, the addition has a longer critical path (now that of a α-bit addition). Thus, it offers a trade-off between area and speed, which will be used, for instance, in Chap. 21.

[1] Referring to Sect. 2.4, carry-save is a binary system with the redundant digit set $\{0, 1, 2\}$ since each digit is represented by two bits.

[2] HRCS is a radix-2^α system with the redundant digit set $\{0, 1, ..., 2^\alpha\}$ since each digit is represented by one standard radix-2^α digit, plus one unit bit.

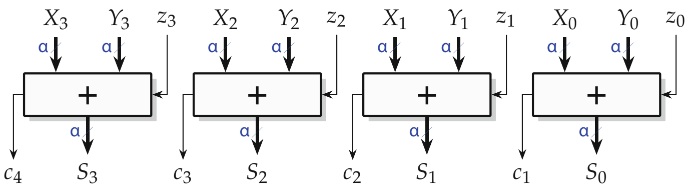


Fig. 5.10 A high-radix carry-save adder.

Carry-save arithmetic is not limited to the addition of three numbers: all these ideas will be generalized in Chap. 7.

5.3 Fast Adders for VLSI

The key observation behind the construction of fast adders is that the bulk of the delay comes from the carry propagation. Hence, a fast adder should accelerate the carry propagation. For that, it helps to study carry propagation as a function of the inputs x_i and y_i of the FA. There are three cases which are given in Table 5.1, a condensed version of the truth table of Fig. 5.3. In the first case, when $x_i = y_i = 0$, the output carry c_{i+1} will be zero, whatever the value of the input carry c_i. As this stops carry propagation, this is called the *kill* case. Similarly, in the last case when $x_i = y_i = 1$, a carry will be always generated whatever the value of c_i. The only cases of actual carry propagation are when exactly one of x_i or y_i is equal to 1. In this *propagate* case, the output carry is equal to the input carry.

Table 5.1 Cases in carry propagation in a full adder.

x_i	y_i	c_{i+1}	Case		
0	0	0	kill		
0	1	c_i		propagate	alive
1	0	c_i			
1	1	1	generate		

These cases can be encoded by the following Boolean expressions:

- kill: $k_i = \overline{x_i} \wedge \overline{y_i} = \overline{x_i \vee y_i}$
- propagate: $p_i = x_i \oplus y_i$
- generate: $g_i = x_i y_i$

All these signals are independent from the carry propagation and can be obtained in parallel from the x_i and y_i only. Now, the carry and sum computations reduce to

$$c_{i+1} = g_i \vee p_i c_i \tag{5.10}$$
$$s_i = p_i \oplus c_i. \tag{5.11}$$

An alternative for the propagate case combines the propagate and generate cases as the *alive* case

$$a_i = \overline{k}_i = x_i \vee y_i \tag{5.12}$$

still enabling the definition of carry propagation as

$$c_{i+1} = g_i \vee a_i c_i. \tag{5.13}$$

An alternative expression for the generate can be obtained by considering that the generate case is only possible when not in the propagate case and one of the inputs is true (compare with Table 5.1):

$$g_i = \overline{p_i}\, x_i = \overline{p_i}\, y_i \tag{5.14}$$

This is useful for multiplexer (MUX)-based implementations of the carry computation, where p_i serves as the select input:

$$c_{i+1} = \overline{p_i}\, x_i \vee p_i c_i = \overline{p_i}\, y_i \vee p_i c_i \tag{5.15}$$

It is clear from (5.11) that each sum bit can be obtained from the corresponding carry and propagate bits. If we are able to compute all the carries, then we also have all the sums, at the cost of just another exclusive-or (XOR) delay. All the approaches discussed in the following are based on a fast computation of all the carries, where "fast" means "faster than in the ripple-carry adder."

5.3.1 Switched Ripple-Carry Adder

The idea of this adder is to use fast switches to realize the three cases of carry propagation discussed above. The speedup here comes from the fact that pure switches may be much faster than complex logic gates. Its structure is identical to the RCA, but each full adder is realized using fast switches. The FA cell is shown in Fig. 5.11. First, the GKSSP block computes the generate, kill, and propagate signals. These mutually exclusive signals now control the fast switches to either pass the input carry to the output ($p_i = 1$) or to clear ($k_i = 1$) or set ($g_i = 1$) the output carry by pulling or pushing the signal to a logic high (supply voltage, Vcc) or low (ground, GND).

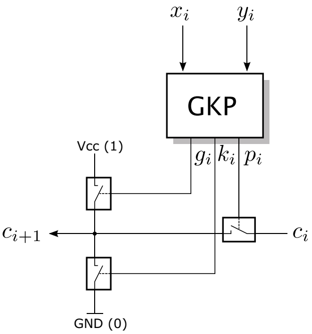


Fig. 5.11 The FA of a switched ripple-carry adder.

The benefit of this approach is very technology dependent. For instance, it was used by K. Zuse in the relay-based Z3 [Zus84; Roj97]: relays switched in parallel to compute k_i, g_i, and p_i, and this established a carry-propagation electrical circuit (see Fig. 5.11) through which the propagation of the carry itself involved no mechanical movement anymore, enabling a complete addition in one cycle. This relay-based design was later (and independently) improved by H. Aiken [Bau98].

In modern VLSI, the equivalent of a relay is the transmission gate, and a full adder cell may be designed around this idea [SB00; Bha+15]. However, transmission gates do not regenerate the signal. Therefore, this approach does not scale well to arbitrary long carry propagations (and (5.7) is not even valid in this case [AP02]).

We now look into generic methods that can be applied at the Boolean algebra level in order to speed up the carry propagation.

5.3.2 *Carry-Select Adder*

Another possibility to speed up the carry propagation is to split the input arguments in groups, each group computing m bits. As the input carry of each group is unknown (except for the group computing the least significant bits), the idea is to just compute both cases in parallel by using two adders and to select the correct result once the carry computation is finished. This is illustrated in Fig. 5.12 for a 16-bit adder with group size of $m = 4$ bits. The CPAs can be realized as RCAs or any other fast adder approach presented in this section. From the timing point of view, all CPAs start independent from each other. Once the carry of the first CPA is computed (c_4), it selects the correct sum bits and output carry (c_8) of the next group. This carry selects the result of the following ones, etc.

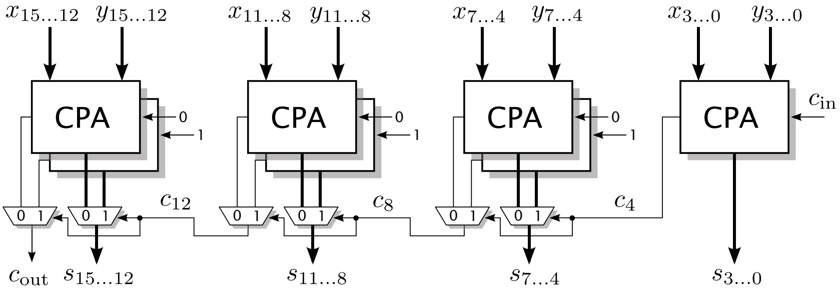


Fig. 5.12 Example of a 16 bit carry-select adder with group size of $m = 4$ bit.

The total delay is then the sum of the delay of one m-bit CPA plus the delay of $\lceil w_s/m \rceil - 1$ multiplexers required to select the carries. A slight delay improvement may be achieved by allowing for increasingly larger groups from right to left, as the leftmost groups have more time to perform their carry propagation than the rightmost ones.

5.3.3 Recursive Carry-Select Adder (Conditional-Sum Adder)

The carry-select idea can also be used recursively to build a first logarithmic-time adder ($\mathcal{O}(\log w)$ in Landau "big-O" notation [GKP94]) as follows. Assume for simplicity that w is a power of two, say $w = 2^q$. Let us build a carry-select adder with only $m = 2$ groups of size $w/2$ bits. The corresponding architecture is the rightmost half of Fig. 5.12. It consists of three adders of size $w/2$ which can operate in parallel. If we ignore for now fanout issues, the $w/2$-bit wide multiplexer can also operate in a bit-parallel way: its delay is independent of w. The delay of the addition in this architecture can therefore be written

$$\tau_{\text{add}}(w) = \tau_{\text{add}}(w/2) + \tau_{\text{mux}} \quad . \tag{5.16}$$

Now, we may use the carry-select idea recursively for each of the three sub-adders: each is itself built as three adders of size $w/4$ followed by a multiplexer, and all these adders can operate in parallel. Thus, (5.16) becomes

$$\begin{aligned} \tau_{\text{add}}(w) &= \tau_{\text{add}}(w/4) + \tau_{\text{mux}} + \tau_{\text{mux}} \\ &= ... \\ &= \tau_{\text{add}}(1) + q\tau_{\text{mux}} \quad . \end{aligned} \tag{5.17}$$

Since $q = \log_2 w$, we have derived a first adder architecture that computes the addition in logarithmic time with respect to the addition size. It is actu-

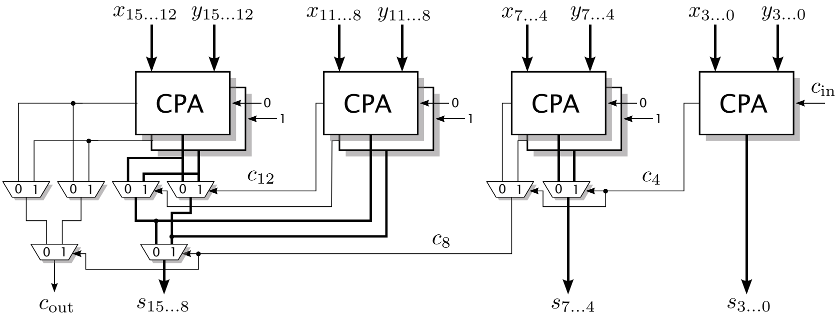


Fig. 5.13 Example of a 16 bit conditional sum adder with group size of $m = 4$ bit.

ally possible to stop the recursion earlier, typically when the adders at the leaves are of size $m = 4$ or $m = 8$.

It is important to understand that this recursive formulation does not increase the final number of CPAs compared to a carry-select adder: even though the recursive formulation seems to imply that the area is multiplied by at least 3/2 with each level, at the leaves of the recursion, there are only two choices for the CPAs: either the input carry is 0, or it is 1. Another point of view is that a leaf CPA is shared by all the adders of the upper levels. For instance, if the recursion is stopped when the addition size is 4, we obtain an architecture with the same CPA blocks as Fig. 5.12, but with a tree of multiplexers replacing the linear sequence of multiplexers. The area thus still grows linearly with w. An architecture exploiting this observation is called a conditional-sum adder [EL04]. Figure 5.13 illustrates a 16-bit conditional-sum adder with group size of $m = 4$ bits. Note again that only the MUX routing is different compared to Fig. 5.12.

In an actual implementation of this idea, fanout issues should not be neglected: there is one carry bit that is input to a $w/2$-bit multiplexer, hence to $w/2$ gates inside this multiplexer (e.g., c_8 on Fig. 5.13). Due to the combined electrical capacity of all these gate, for large w, this will entail a slower operation of the multiplexer. VLSI and FPGA synthesis tools are able to handle this situation automatically, but the reader should be aware that this may add some area and delay. Section 5.3.6 will present a family of adders where the fanout can be controlled.

5.3.4 *Carry-Lookahead Adder*

The idea of the carry-lookahead (CLA) adder is, as the name suggests, to directly compute the carry without computing intermediate carries first. For that, the recursive definition of the carry propagation

$$c_{i+1} = g_i \vee p_i c_i \tag{5.18}$$

is applied for a certain number of iterations, yielding

$$c_1 = g_0 \vee p_0 c_0 \tag{5.19}$$
$$c_2 = g_1 \vee p_1 c_1 = g_1 \vee p_1 g_0 \vee p_1 p_0 c_0 \tag{5.20}$$
$$c_3 = g_2 \vee p_2 c_2 = g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0 \tag{5.21}$$
$$c_4 = g_3 \vee p_3 c_3 = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 p_1 p_0 c_0 \tag{5.22}$$
$$\vdots$$
$$c_k = g_{k-1} \vee p_{k-1} g_{k-2} \vee p_{k-1} p_{k-2} g_{k-3} \vee \cdots \vee p_{k-1} p_{k-2} \cdots p_0 c_0. \tag{5.23}$$

Now, we have a formula with just two levels of logic, one level of wide AND gates connected to a wide OR. However, the number of inputs to these gates quickly grows with k. In practice, this will introduce more levels of logic as well as a high area complexity.

One way to deal with this is to restrict the use of the lookahead scheme to a certain amount of carry propagation. Figure 5.14 shows an example of a 16-bit adder which uses groups of $m = 4$ bits. On each group, the CLA-4 module uses (5.19), (5.20), (5.21), and (5.22) for the carry computations together with (5.11) to compute the sum bits.

With this architecture, the delay still grows linearly with the word size of the adder, but with a smaller factor.

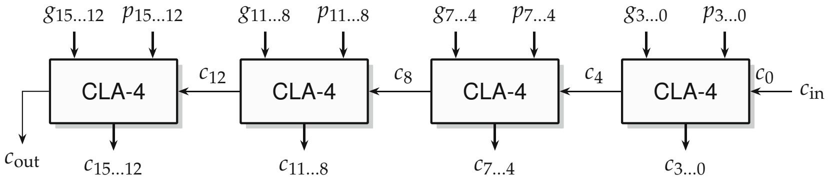


Fig. 5.14 Carry generation of a 16-bit adder with 4-bit carry-lookahead groups.

5.3.5 Recursive Carry-Lookahead Adder

The carry-lookahead idea can also be applied recursively, leading to a second family of logarithmic-time adder architectures. Here, the idea is that each CLA block of Fig. 5.14 can compute at no extra cost two additional signals $g_{i:j}$ and $p_{i:j}$, respectively, the carry generate and carry propagate of the group between indices i and j (with $i > j$), such that

$$c_{i+1} = g_{i:j} \vee p_{i:j} c_j \; . \tag{5.24}$$

This formulation is simply a reparenthesizing of the carry-lookahead equations. For instance, (5.22) can be rewritten

$$\begin{aligned} c_4 &= g_{3:0} \vee p_{3:0} c_0 && (5.25)\\ \text{with} \quad p_{3:0} &= p_3 p_2 ... p_1 p_0 && (5.26)\\ \text{and} \quad g_{3:0} &= g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 . && (5.27) \end{aligned}$$

These group generate and propagate signals can be computed by a tree of components similar to CLA-m. In this tree, the maximum number of levels is $\log_m(w)$, leading to an overall time complexity which scales logarithmically with w. The details can be found in [EL04].

Compared to the conditional-sum adder, we have a shallower tree – $\log_m(w)$ levels instead of $\log_2(w)$ – but with more complex nodes.

We now present a family of fast adders that generalizes the previous ideas and offers a wider implementation space.

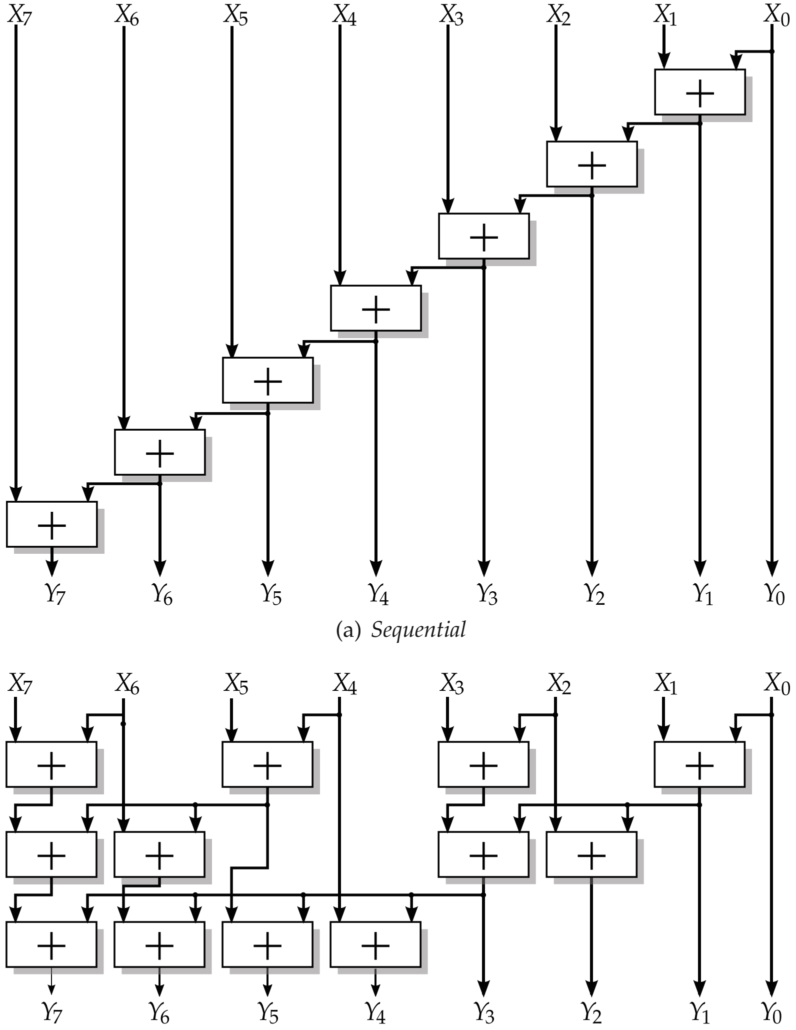
5.3.6 Parallel Prefix Adders

The main idea here is to express the carry propagation problem as a prefix operation.

To illustrate what is a prefix operation and why it is interesting in terms of design space expressivity, let us consider a different problem: prefix sums. Assume we have a vector of n values $(X_0, X_1, ...X_{n-1})$ and we have to compute *all* the arithmetic sums:

$$\begin{aligned} Y_0 &= X_0 && (5.28)\\ Y_1 &= X_0 + X_1 && (5.29)\\ Y_2 &= X_0 + X_1 + X_2 && (5.30)\\ Y_3 &= X_0 + X_1 + X_2 + X_3 && (5.31)\\ &\vdots \\ Y_{n-1} &= \sum_{i=0}^{n-1} X_i . && (5.32) \end{aligned}$$

Clearly, given that Y_1 is computed, one can rewrite (5.30) to $Y_2 = Y_1 + X_2$, then (5.31) to $Y_3 = Y_2 + X_3$, etc. This is a resource-efficient way to compute all the prefixes, but also the slowest as it is completely sequential, with a succession of n additions. Figure 5.15a illustrates this computation scheme for the sum of $n = 8$ values.



(a) *Sequential*

(b) *Parallel due to Sklansky [Skl60]*

Fig. 5.15 Examples of computations of a parallel prefix sum.

On the contrary, if the objective is to achieve the maximum speed, one can build one adder tree for each of the outputs. All these adder trees will work in parallel, which guarantees that all the computations will be achieved in $\log_2(n)$ additions. However, the hardware cost will be high. One way to reduce it is to exploit the associativity of addition: many intermediate results can be shared while still achieving a worst-case $\log_2(n)$ time, as illustrated in Fig. 5.15b [Skl60]. The reader may check that Fig. 5.15b still computes the same as Fig. 5.15a, with a moderate increase in the number of adders (12 adders in Fig. 5.15b compared to 8 adders of Fig. 5.15a).

We have exposed a trade-off between time and area. This problem is an instance of a *parallel prefix operation*, where the objective is to compute all the prefixes according to some associative operation, here $+$. There are actually many other intermediate solutions with different trade-offs; see [Zim97] for an extensive review including cost evaluations for area, time, and area·time product.

Back to the subject of this chapter, let us show that carry propagation can also be expressed as a prefix computation, for an operation $\circ$ defined on 2-bit tuples formed of a generate bit and an alive[3] bit (see (5.10))

$$X = (g_X, a_X) \tag{5.33}$$
$$Y = (g_Y, a_Y) \tag{5.34}$$

which are related by a prefix operation

$$Z = (g_Z, a_Z) = X \circ Y = (g_X, a_X) \circ (g_Y, a_Y) \tag{5.35}$$

with

$$g_Z = g_X \vee a_X g_Y \tag{5.36}$$
$$a_Z = a_X a_Y. \tag{5.37}$$

This operation is associative, i.e.,

$$(U \circ V) \circ W = U \circ (V \circ W) \tag{5.38}$$

which can be easily proven by setting (5.36) and (5.37) into the left- and right-hand side of (5.38) and showing equality:

$$g_{(U\circ V)\circ W} = g_{U\circ V} \vee a_{U\circ V} g_W = g_U \vee a_U g_V \vee a_U a_V g_W \tag{5.39}$$
$$g_{U\circ(V\circ W)} = g_U \vee a_U g_{V\circ W} = g_U \vee a_U (g_V \vee a_V g_W) \tag{5.40}$$
$$= g_U \vee a_U g_V \vee a_U a_V g_W \tag{5.41}$$

$$a_{(U\circ V)\circ W} = a_{U\circ V} a_W = a_U a_V a_W \tag{5.42}$$
$$a_{U\circ(V\circ W)} = a_U a_{V\circ W} = a_U a_V a_W. \tag{5.43}$$

With this operation, we can obtain the carries by computing

[3] It is also possible to use the generate and propagate bits, e.g., in [Zim97].

$$(g_0, a_0) \circ (c_0, c_0) = (g_0 \vee a_0 c_0, a_0 c_0) = (c_1, a_0 c_0) \tag{5.44}$$

$$(g_1, a_1) \circ (c_1, a_0 c_0) = (g_1 \vee a_1 c_1, a_1 a_0 c_0) = (c_2, a_1 a_0 c_0) \tag{5.45}$$

$$(g_2, a_2) \circ (c_2, a_1 a_0 c_0) = (g_2 \vee a_2 c_2, a_2 a_1 a_0 c_0) = (c_3, a_2 a_1 a_0 c_0) \tag{5.46}$$

$$\vdots$$

$$(g_{i-1}, a_{i-1}) \circ (c_{i-1}, a_{i-1} a_{i-2} \ldots a_0 c_0) = (c_i, a_{i-1} a_{i-2} \ldots a_0 c_0). \tag{5.47}$$

So, the carry at position i can be computed by evaluating

$$(c_i, a_{i-1} a_{i-2} \ldots a_0 c_0) = \bigcirc_{j=-1}^{i-1} (g_j, a_j) \tag{5.48}$$

when defining $g_{-1} = a_{-1} = c_0$.

Hence, as we need all the carries, we can perform the same rearrangement done for computing the sums in the example above. This is shown in Figs. 5.17, 5.18, and 5.19. In the figures, two different notations are used which are introduced in Fig. 5.16. The reason is that in some prefix additions, no a_R is given and thus no a_{out} result is computed. Therefore we can omit its computation. Those simplified prefix computations are illustrated using a filled circle[4] (see Fig. 5.16).

The obtained carries have to be XORed with the propagate signal to obtain the final sum. These XORs are not shown on the figures.

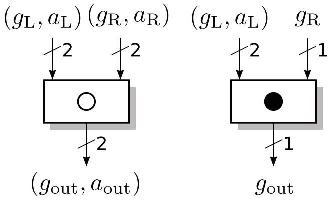


Fig. 5.16 The prefix operator and its simplified version.

The only rule to consider when organizing the prefix operators is (5.48). This allows many other variants with different properties. Regarding the delay, the stage count is the main metric to consider, but another important one is the *fanout* of each operator. The fanout of an output counts the number of inputs that are connected to this output. Due to the combined electrical capacity of these inputs, a high fanout entails slow transitions. This can be partially compensated in very large scale integrated circuit (VLSI) circuits by using a stronger driver, but this also leads to a larger power consumption. Hence, designs with low fanout are desirable. Figure 5.19 shows the parallel prefix adder according to Kogge and Stone [KS73], which limits the fanout to two (compared to $w/2$ in the Sklansky structure of Fig. 5.18) while still

[4] Different notations are used in the literature; we use the notation of [EL04] here while, e.g., [Zim97] uses a different notation.

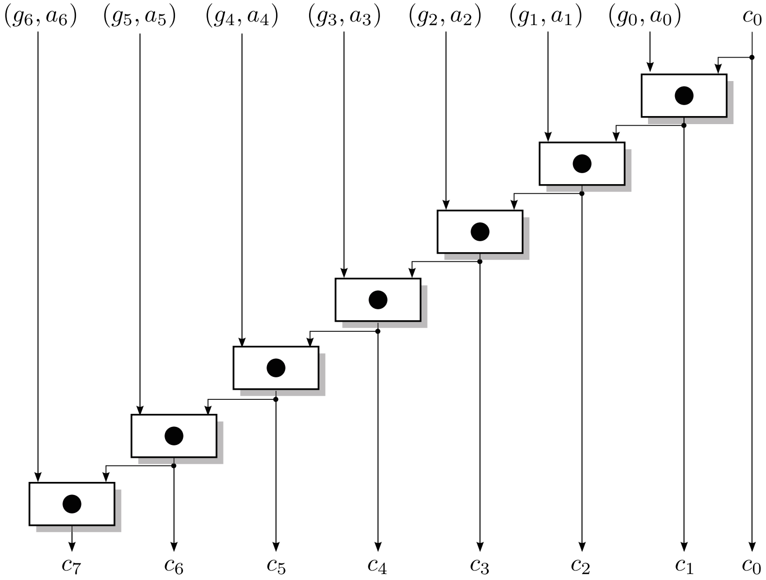


Fig. 5.17 The carry-propagate adder as a special case of parallel-prefix adder.

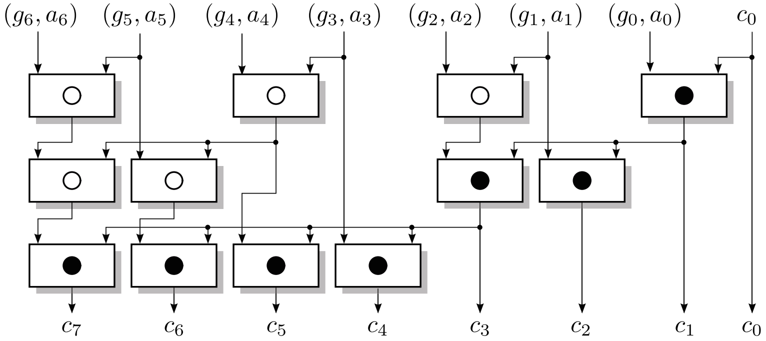


Fig. 5.18 The Sklansky parallel-prefix adder.

providing the minimum number of stages. This comes at the cost of more resources than the Sklansky structure.

Another example is the Brent-Kung adder [BK1], which uses very low resources while having a slightly larger stage count. Generalizations have also been made toward prefix operators accepting more than two inputs [BL01].

An excellent review of various prefix structures comparing area, delay, wire cost, and fanout is given in Zimmermann's PhD thesis [Zim97]. Another good overview, including a taxonomy comparing the stage count (logic depth), the fanout, as well as the complexity of the wire tracks was presented by Harris [Har03]. The issue of power consumption in fast adders has been studied by Patil et al. [Pat+07]. To address the huge design space of parallel prefix adders, an NVIDIA team has used reinforcement learning [Roy+21]. Prefix adders are also well suited to adders with specific timing requirements, i.e., input bits arriving at different times or output bits having different timing constraints [Zim97, Chapter 5].

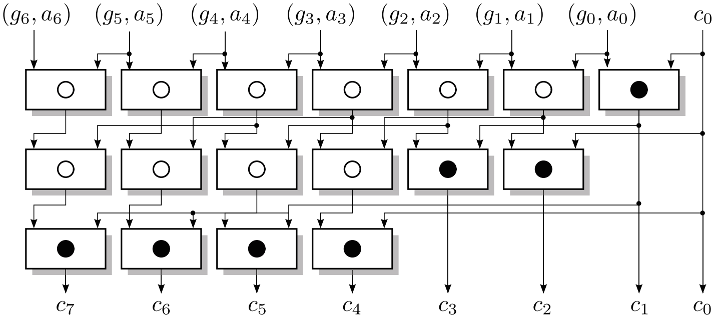


Fig. 5.19 A Kogge-Stone parallel prefix adder.

5.3.7 *Compound Fast Adder*

In a parallel prefix adder, the bulk of the operator computes all the immediate generate and alive signals. The computation that actually depends on c_0, the input carry, is only the bottom line of simplified prefix operators (see Figs. 5.18 and 5.19). Its size is linear in w; therefore it is relatively cheap to duplicate it for $c_0 = 0$ and $c_0 = 1$. This way, a fast parallel prefix adder computing $X + Y$ may also compute $X + Y + 1$ for little additional cost, as illustrated in Fig. 5.20.

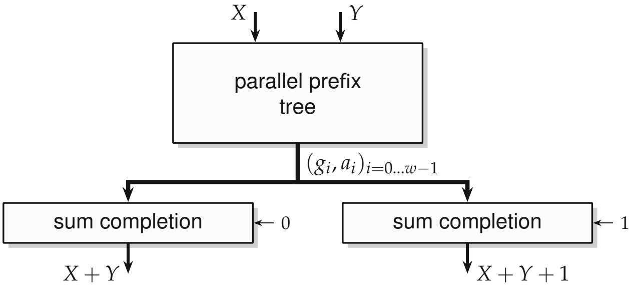


Fig. 5.20 Compound adder computing both $X + Y$ and $X + Y + 1$.

Such a *compound adder* is very useful for floating-point operators, where the rounded significand is either a sum or the successor of this sum [SE01].

5.3.8 Fast Absolute Difference Operator

A variation of the compound adder, depicted in Fig. 5.21, computes $|X - Y|$ when X and Y are two positive numbers. With two's complement, this operator needs to compute either $X - Y = X + \overline{Y} + 1$ or $Y - X = \overline{X + \overline{Y}}$.[5] The selection is done in constant time by a multiplexer controlled by the sign bit of one of the differences ($Y - X$ on our figure). The two bitwise negations in these equations do not necessarily entail additional delay: $\overline{Y}$ can be merged in the initial (g_i, a_i) computation, and the last negation of $\overline{X + \overline{Y}}$ can be merged in the sum completion logic.

Note that the operator of Fig. 5.21 may also output (for free) the sign bit of $Y - X$, or the sign bit of $X - Y$, or both. The two sign bits will be different, except when $X = Y$: at the cost of an additional NOR gate, this operator may also provide an $X = Y$ bit.

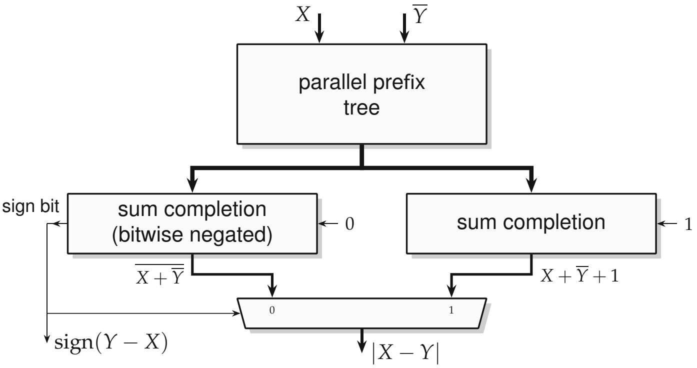


Fig. 5.21 Compound adder computing $|X - Y|$.

5.3.9 Fast Compound Triple Sum

Some fast floating-point adders [Soh+22] require the computation of $X + Y$, $X + Y + 1$, and $X + Y + 2$ in parallel. Here is a simple way to do implement

[5] Proof that $Y - X = \overline{X + \overline{Y}}$: we have $X - Y = X + \overline{Y} + 1$, hence, $X - Y - 1 = X + \overline{Y}$; the opposite of this value is $-(X - Y - 1) = \overline{X + \overline{Y}} + 1$ hence $Y - X = \overline{X + \overline{Y}}$.

this [Soh+22]. First, the two addends X and Y are input into a row of half adders (HAs) that rewrite the sum $X + Y$ into sum and carry vectors $S + C$ (Fig. 5.22). The advantage of this rewriting is that the sum bit at position 0 is reduced to a single bit s_0.

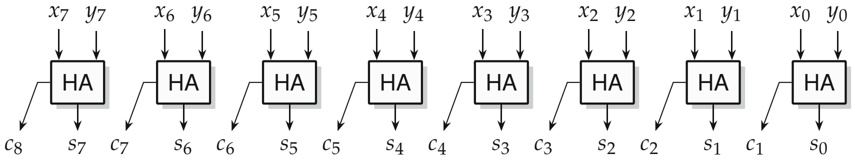

x_7 y_7
x_6 y_6
x_5 y_5
x_4 y_4
x_3 y_3
x_2 y_2
x_1 y_1
x_0 y_0
HA
c_8 s_7 c_7 s_6 c_6 s_5 c_5 s_4 c_4 s_3 c_3 s_2 c_2 s_1 c_1 s_0

Fig. 5.22 Rewriting $X + Y = S + C$ using a row of half adders.

Then, a compound adder on the upper bits (positions 1 to $w + 1$, i.e., assuming $s_0 = 0$) computes two intermediate sums Z and $Z' = Z + 2$. With this, the triple sum can be expressed (at the cost of a multiplexer) as the following concatenations:

$$\text{if } s_0 = 0 \begin{cases} X + Y = Z0 \\ X + Y + 1 = Z1 \\ X + Y + 2 = Z'0 \end{cases} \quad \text{else} \quad \begin{cases} X + Y = Z1 \\ X + Y + 1 = Z'0 \\ X + Y + 2 = Z'1 \end{cases} . \tag{5.49}$$

In short, the overhead of the fast triple adder with respect to the compound fast adder is a row of w half adders and three $w + 2$-bit multiplexers.

5.4 Adders on FPGAs

Fixed-point adders on FPGAs have two particularities that are worth studying. The first is that carry propagation benefits from dedicated hardened (i.e., non-programmable) hardware and routing resources (collectively named *fast carry logic*; see Sect. 4.1, p. 91) which make it very fast compared to generic programmable logic. The second is that additions can be merged with other functionalities at no additional cost. This section studies these two points in more details for the two mainstream FPGA families from AMD and Intel.

5.4.1 Addition Support on AMD FPGAs

On AMD field programmable gate arrays (FPGAs), the adder implementation depends on the series. On series 4 to 7 and UltraScale/UltraScale+ FPGAs, the carry chain logic consists of an XOR gate, and a MUX per basic logic element (BLE) (see Fig. 4.5b in Sect. 4.1.2, p. 91) can be used to build ripple-carry adders (RCAs). For this, the look-up table (LUT) has to be configured to compute the propagate signal $p_i = x_i \oplus y_i$. The sum output is computed by XORing p_i with c_i with the carry chain XOR. The carry computation uses the MUX formulation obtained in (5.15). Hence, each BLE computes the Boolean relations

$$s_i = p_i \oplus c_i \tag{5.50}$$

$$c_{i+1} = p_i c_i \vee \overline{p_i}\, x_i \tag{5.51}$$

which correspond to the full adder (FA) relations (5.2) and (5.15). The BLE configuration is shown in Fig. 5.23.

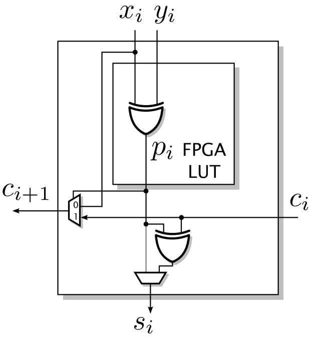


Fig. 5.23 Realization of a full adder on Xilinx/AMD's series 4 to 7 and UltraScale/UltraScale+.

It can also be seen as a switched ripple-carry adder (see Sect. 5.3.1) where the XOR in the LUT computes the propagate signal p_i. Now, if $p_i = 1$, the carry in is propagated to the output. Otherwise, if $p_i = 0$, it is sufficient to pass either x_i or y_i, as it is known that both are equal (as their XOR obtained zero due to $p_i = 0$).

This multiplexer is very fast compared to the surrounding logic, which is why this is called the *fast carry chain*. To give some example values, consider the timing data of the Virtex 6 FPGA architecture (similar ratios can be assumed for other families). It comes in different performance grades, which are called *speed grades*. For the fastest speed grade (3), the carry-in to carry-out delay of a complete slice with four BLEs is specified to be at most 0.06 ns [Xil14]. Hence, each BLE contributes with $\tau_{\text{cp}} = 0.015$ ns to the carry delay. The relevant combinational delay of a LUT ranges between

$\tau_{\text{LUT}} = 0.14 \ldots 0.32\,\text{ns}$, depending which input influences which output [Xil14]. The delay of the XOR (from carry-in to one of the BLE outputs within a slice) is $\tau_{\text{XOR}} = 0.21 \ldots 0.25\,\text{ns}$. This leads to a worst-case full adder delay of $\tau_{\text{fa}} = 0.57$, which is 38× the delay of the carry chain. Besides, there is also a dedicated link for the transmission of a carry bit from one CLB to the next, which is also much faster than random wiring through the generic routing fabric. The worst-case delays of adders with $w = 8$, $w = 16$, and $w = 32$ bit can be obtained using (5.7) and are about 1, 1.5, and 2.5 ns, respectively.

On the Versal devices, the carry chain logic of each BLE does not contain the XOR gate anymore, but an equivalent of the MUX is included in the fast carry-lookahead logic (see Fig. 4.5c in Sect. 4.1.2, p. 91). The fast carry-lookahead logic computes every second carry from all the propagate and intermediate carries computed in the LUTs. From the eight BLEs in a slice which are labeled from `A` to `H`, but only the even carry ouputs `COUTB`, `COUTD`, `COUTF`, and `COUTH` are computed from the carry logic in a lookahead manner. In consequence, the remaining odd carry outputs have to be computed by using the LUTs.

Figure 5.24 shows the configuration of three BLEs together with the carry-lookahead logic at the bottom. The propagate signals p_i are computed using LUTs; in this case, each BLE computes a propagate in the LUT4 that is connected to the `PROSSP` input of the carry-lookahead logic, called the `LOOKAHEAD8` unit. As no XOR is available in the carry chain anymore, we have to utilize the LUTs. For that, the carry c_i of the neighboring BLE is routed by casc_{in} to the MUX that corresponds to `O5_1`. With that, sum signal is computed by

$$s_i = p_i \overline{c_i} \vee \overline{p_i}\, c_i \tag{5.52}$$

$$= p_i \oplus c_i, \tag{5.53}$$

where $\overline{p_i}$ is computed in the second LUT4.

For BLEs that compute inputs with even indices (c_0, c_2, …), the carry is either given as input (c_0) or computed by the carry-lookahead logic (c_2, c_4, …) and provided at the casc_{in} of the BLE. For BLEs that compute inputs with odd indices (c_1, c_3, …), the carry has to be computed in the previous BLE and is passed by casc_{in} of the BLE. Hence, the odd indexed carry outputs are computed by

$$c_{i+1} = x_i y_i \overline{c_i} \vee (x_i \vee y_i) c_i \tag{5.54}$$

$$= x_i y_i \overline{c_i} \vee x_i c_i \vee y_i c_i \tag{5.55}$$

$$= x_i y_i \vee x_i c_i \vee y_i c_i. \tag{5.56}$$

This is realized in the remaining two LUT4 and outputted at `O5_2`, `O6`, and finally casc_{out}. In the even indexed BLEs, one of the inputs have to be passed to the carry-lookahead logic according to (5.15). With that, we will only have a carry propagation from even indexed BLEs to odd indexed BLEs. Together with the carry-lookahead scheme of the LOOKAHEAD8 unit, this will result in fast adders that are very flexible in size.

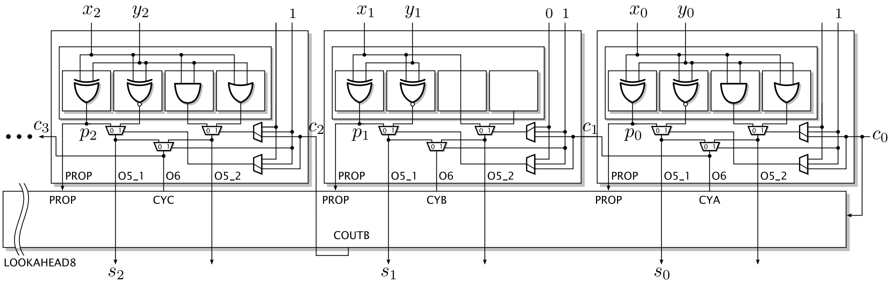


Fig. 5.24 Realization of the first bits of an adder using AMD's Versal BLEs.

All this is exploited transparently by synthesis tools: the take-away message here is that an FPGA designer should not worry too much about the carry propagation delay of additions of sizes up to 32-bit. However, another important remark is that the addition logic is not fully using the BLE. We will shortly see (Sect. 5.4.3 and the subsequent sections) how this remark can be exploited.

5.4.2 *Addition Support on Intel FPGAs*

Things are simpler on Intel hardware, as each BLE includes a complete FA with dedicated carry propagation wires to its neighbors (see Sect. 4.1.3). The implementation of a RCA is straightforward. There are only a few constraints on the start and stop of the carry chain: there are ten adaptative logic modules (ALMs) per logic array block (LAB) (each ALM consisting of two BLEs with partially shared inputs). A carry chain can start at in the first or the sixth ALM of an LAB.

Here also, carry propagation is very fast compared to generic programmable logic thanks to the fact that these FA are hardened and that the generic programmable routing is avoided.

5.4.3 *Merging Additional Functionality in an Adder*

Implementing adders on FPGAs will only use a fraction of the LUT resources. On Intel FPGAs, the inputs of the FA can be fed from the LUTs (see Fig. 4.6, p. 93), which can be used for additional logic without additional cost. This enables an RCA computing

$$(c_{i+1}, s) = \mathrm{FA}(p(a,b,c,d), q(b,c,d,e), c_i), \tag{5.57}$$

using the FA notation introduced in (5.4) where p and q are two bit-parallel functions implemented as parallel LUT4 sharing most of the inputs.

We have a similar case for AMD FPGAs as only a fraction of the LUT is utilized in Fig. 5.23. Hence, the remaining logic can be used for additional logic at no cost. The BLE of Fig. 4.5b can realize two independent LUT5 with shared inputs in front of each of the FA inputs computing

$$(c_{i+1}, s) = \mathrm{FA}(p(a,b,c,d,e), q(a,b,c,d,e), c_i) \text{ (Fig. 5.25a).} \tag{5.58}$$

Here, two parallel LUT5 with shared inputs are possible. Alternatively, there can be a single LUT6 on one of the FA inputs, allowing to perform

$$(c_{i+1}, s) = \mathrm{FA}(p(a,b,c,d,e,f), a, c_i) \text{ (Fig. 5.25b).} \tag{5.59}$$

The BLE of the Versal architecture in Fig. 4.5c can realize two independent LUT4 with shared inputs in front of each of the FA inputs computing

$$(c_{i+1}, s) = \mathrm{FA}(p(a,b,c,d), q(a,b,c,d), c_i) \text{ (Fig. 5.25c).} \tag{5.60}$$

This additional logic can be used, e.g., to realize

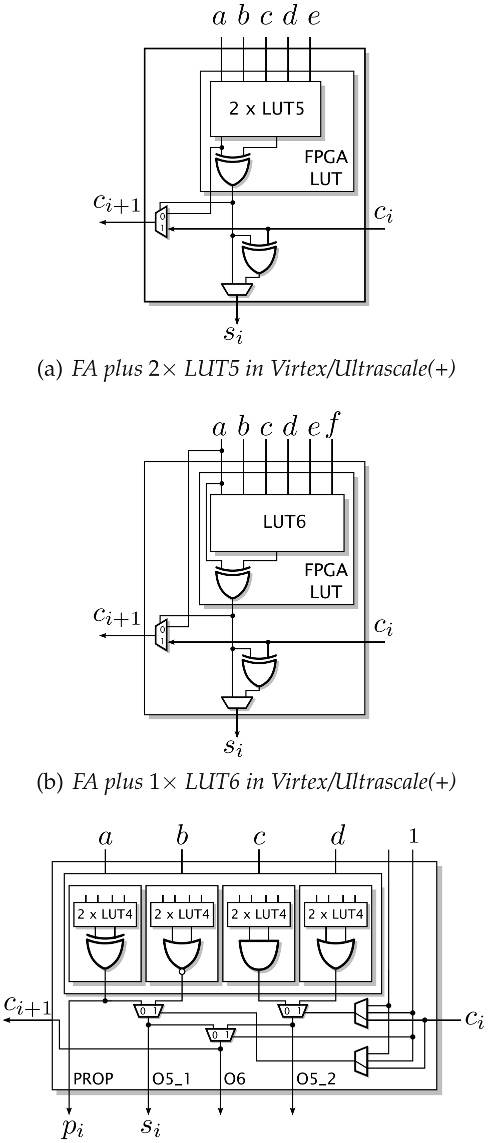
- an additional full adder adding a third input (see Sect. 5.4.4),
- an adder/subtractor, where one input bit selects between addition and subtraction at runtime (see Sect. 5.4.5),
- a multiplexer selecting between different sources for the addition (see, e.g., Fig. 9.8 in the dividers of Sect. 9.2.4),
- the addition of a value read from a table indexed with fewer than 5 bits inputs (see, e.g., table-based constant multiplications [Wir04] in Sect. 12.2.2 or optimizations to an arctangent implementation [TVC17]),
- the generation of partial products in multiplications (see Sect. 8.4.2).

This is one example of *target-specific optimization*, one of the techniques for application-specific arithmetic discussed in the introduction.

The remainder of this section gives two examples where additional functionality is merged in the LUTs implementing the addition: Sect. 5.4.4 shows the construction of a ternary adder, and Sect. 5.4.5 discusses the construction of several adder/subtractor variants.

5.4.4 Ternary Adders on FPGAs

The logic available for free in front of the RCA can be used to realize additional full adders to be able to add a third number with essentially the same



(a) *FA plus* 2× *LUT5 in Virtex/Ultrascale(+)*

(b) *FA plus* 1× *LUT6 in Virtex/Ultrascale(+)*

(c) *FA plus* 1× *LUT6 in Versal*

Fig. 5.25 Additional logic that can be implemented in front of a full adder using AMD's BLEs (see Fig. 4.5b and c).

logic resources as for common 2-input adders. Hence, three variables can be added in one row of BLEs, computing

$$S = X + Y + Z. \tag{5.61}$$

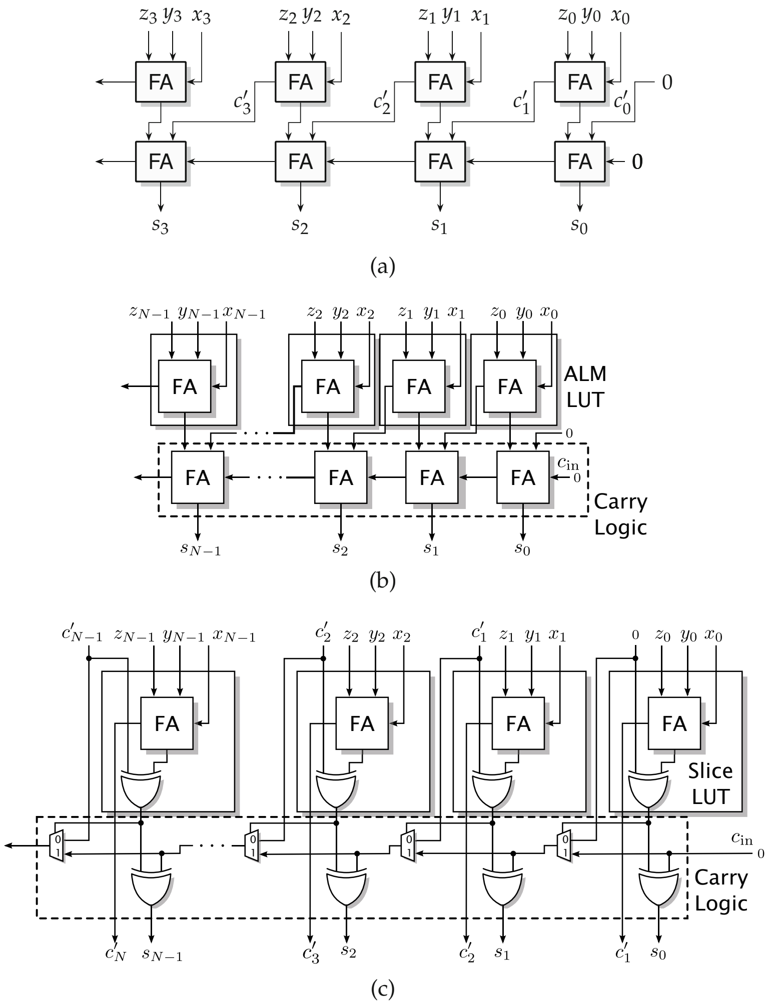


Fig. 5.26 Realization of ternary adders (a) generic, (b) mapping to Intel Stratix II-V ALMs (c) mapping to Xilinx/AMD Virtex 5-7 slices.

These adders are commonly called ternary adders (not to be confused with adders using a ternary *number format*, i.e., $\{-1, 0, 1\}$ [EL04]).

The ternary adder is realized by implementing one layer of full adders to form a carry-save adder (CSA) which compresses the three input vectors into two vectors (sum vector and carry vector). More details about carry-save arithmetic can be found in Chap. 7. These two vectors are added to get the final sum by using an RCA built from the fast carry chain. This principle is shown in Fig. 5.26a. The first row of full adders realizes the CSA; the second row of full adders corresponds to the RCA. Note that there is no carry-propagation in the CSA. In VLSI, it would not save any hardware as the same number of FAs is used. However, on FPGAs the first layer of FAs can be mapped to the LUTs in front of the RCA. The detailed implementation for the different FPGA vendors are briefly given in the following.

5.4.4.1 Realization on Intel FPGAs

The ALMs of Intels Arria I, II, and V and Stratix II-V FPGAs support a special *shared arithmetic mode* for the fast implementation of ternary adders [Bae+09; Alt13]. Unfortunately, the shared arithmetic mode has been dropped with the Stratix 10 generation [Int17; Int18a]. For FPGAs supporting the shared arithmetic mode, the FAs of the CSA can be directly mapped to the LUTs as shown in Fig. 5.26b. Note that a 2-input adder with the same output word size requires exactly the same number of ALMs, but will be slightly faster [Kum+13]. For FPGAs not supporting the shared arithmetic mode, the ternary adder requires additional LUTs [Int18a].

The Quartus II tool automatically detects ternary adders from a VHDL statement of the form `S <= X+Y+Z`. Unfortunately, ternary operations when one or more inputs are negated (i.e., $X - Y + Z$, $X + Y - Z$, and $X - Y - Z$) are not directly supported from a high-level description. To overcome this problem, the corresponding input(s) must be inverted, and the carry-in signal(s) must be set to 1. To set the carry bit(s), the word size of the adder has to be extended by one bit for $X - Y + Z$ and $X + Y - Z$ and by two bits for $X - Y - Z$, consuming additional ALMs.

5.4.4.2 Realization on AMD FPGAs

Ternary adders can be efficiently mapped to all modern Xilinx/AMD FPGA families where the BLE provides a 6-input LUT that can be fractioned as two 5-input LUTs [SP06], namely, the Virtex 5-7, Spartan 6, Kintex 7, Artix 7 families, as well as the Zync System-on-Chips (SoCs) (including UltraScale and UltraScale+).

The slice configuration is shown in Fig. 5.26c. The full adder for the CSA may be realized in the same LUT as the XOR gate of Fig. 5.23. The carry output c_i' are all computed in parallel, but unfortunately they have to be routed to the next higher FA through the FPGA routing fabric. As a result, for low word sizes [Kum+13], a ternary adder is up to 50% slower than a binary one. Still, the carry propagation of the second adder only uses the fast carry dedicated routing (dashed box of Fig. 5.26c).

Currently, there is no guarantee that ternary adders from an hardware description language (HDL) description are mapped in that efficient way by the AMD tools. Thus, it has to be built by using the AMD primitives. Note that the ternary subtract operations ($X - Y + Z$, $X + Y - Z$ and $X - Y - Z$) can be realized without additional resources.

Hands on: Ternary adder for AMD/Xilinx FPGAs
The FloPoCo core generator can be used to build ternary adders for AMD/Xilinx FPGAs. The following FloPoCo call generates VHDL code for a ternary adder with 16 bit input word size:

```
flopoco target=virtex6 XilinxTernaryAddSub wIn=16
```

An optional bit mask can be passed to select inputs to be subtracted (with the LSB corresponding to input X and the MSB corresponding to Z). For instance, the bit mast $011_2 = 3_{10}$ selects the $-X - Y + Z$ operation:

```
flopoco target=virtex6 XilinxTernaryAddSub wIn=16 \
   AddSubBitMask=3
```

5.4.5 *Reconfigurable Adder/Subtractor*

The structure of an adder/subtractor as introduced in Sect. 5.2.5 is easy to map to the current FPGA architectures. It comes at no additional cost compared to a simple adder or subtractor as the required XORs can be mapped to the same LUTs used for the RCA.

The idea can be even further extended such that both inputs may be negated. Cleary, adding a select signal sub_X which is XORed with the X input and ORed with the carry-input would also allow a selective $-X + Y$. However, to realize $-X - Y$, one would have to add constant $1 + 1 = 2$ for complementing both inputs. This is, of course, not possible just by using the carry-in. However, the structure of the ternary adder described above can be used to perform this [KKZ16]. Such a generalized adder computes

$$S = \begin{cases} X + Y & \text{when } \text{sub}_X = 0,\ \text{sub}_Y = 0 \\ X - Y & \text{when } \text{sub}_X = 0,\ \text{sub}_Y = 1 \\ -X + Y & \text{when } \text{sub}_X = 1,\ \text{sub}_Y = 0 \\ -X - Y & \text{when } \text{sub}_X = 1,\ \text{sub}_Y = 1 \end{cases} . \tag{5.62}$$

Its FPGA mappings to Intel and AMD FPGAs are shown in Figs. 5.27 and 5.28, respectively. Only when exactly one of the select inputs is negated, a $+1$ has to be added which is realized by an XOR gate. The $+2$ is added in the case that both select inputs are true, which is realized by an AND gate.

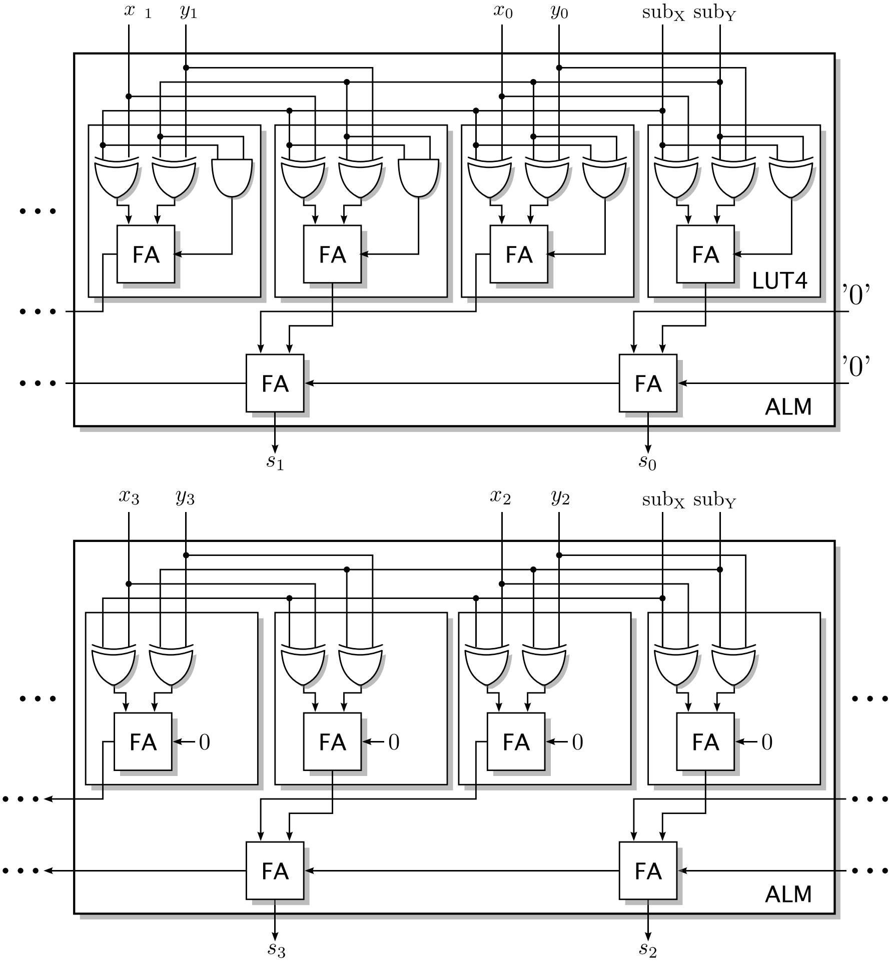


Fig. 5.27 Intel ALM mapping of the adder/subtractor computing $S = (-1)^{\mathrm{sub}_X} X + (-1)^{\mathrm{sub}_Y} Y$.

This generalized adder/subtractor is especially interesting for adding up absolute values (by using the sign bit of X and Y as sub$_X$ and sub$_Y$, respectively). This is, for example, used in the sum of absolute difference (SAD) metric widely found in image processing [KKZ16].

5.5 Fast Adders on FPGAs

Fast adder architectures as discussed in Sect. 5.3 are useless when targeting FPGAs for typical sizes (up to 32 bits). However, several FPGA-specific

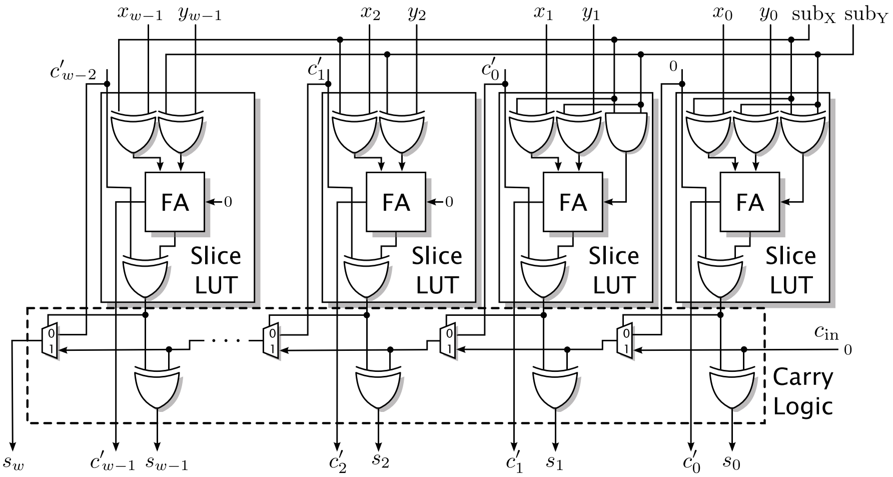


Fig. 5.28 AMD Slice mapping of the adder/subtractor computing $S = (-1)^{\text{sub}_X} X + (-1)^{\text{sub}_Y} Y$.

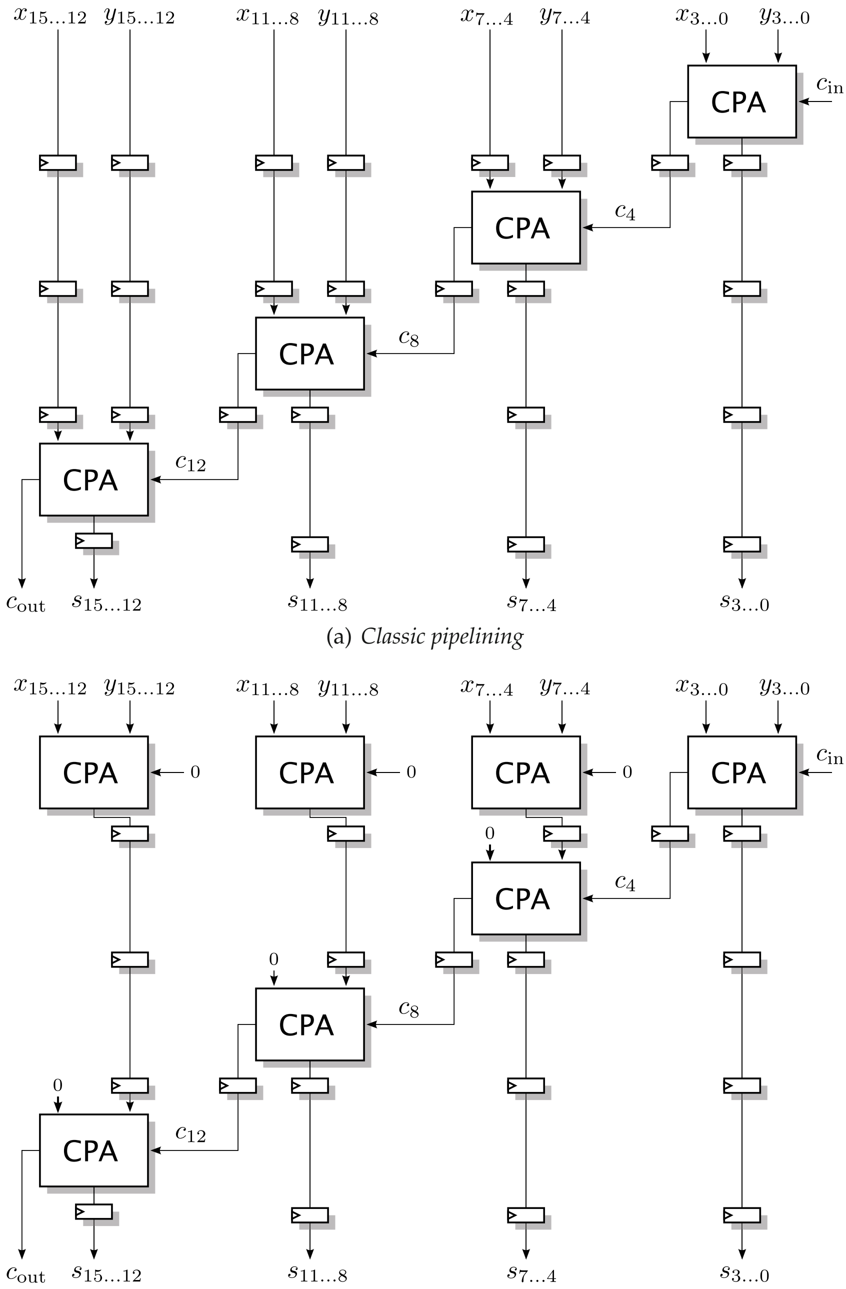
schemes have been suggested to accelerate larger additions on FPGAs [XY98; YN08; DNP10; NPP11; RHG14; KG16; LPB19]. Accelerating fast addition is an active research topic, as there are plenty of applications for very large word sizes, in particular in cryptography [RHG14; LPB19].

There are in principle two ways to accelerate the speed: pipelining and optimizations of the combinatorial paths (like discussed above in Sect. 5.3 for VLSI adders). Of course, both can be combined. They are discussed in the following.

5.5.1 *Pipelined Adders*

In VLSI, the cost of a flip flop is roughly comparable to that of a full adder; therefore pipelining an integer addition is not competitive compared to the fast adder techniques of Sect. 5.3 (all the more as pipelining delays the output by one or several cycles, which may incur more flip-flops (FFs) for synchronization of the output with other signals). Conversely, FPGAs offer a LUT/FF ratio close to 1: FFs are a resource that is wasted if it is not used. Besides, FFs are integrated in the BLEs and directly connected to the addition output (see Sect. 4.1), so pipelining is an attractive way to speed up integer addition on FPGAs.

Figure 5.29a shows an RCA after applying classic pipelining. The RCA is divided into groups of m bits, and each group is computed in a separate pipeline stage. Hence, each pipeline stage except stage 0 requires $2m$ FFs for the inputs, m FFs for the output, and a single FF for the carry. Note that this



(a) *Classic pipelining*

(b) *FPGA-optimized pipelining*

Fig. 5.29 Pipelined ripple-carry adder with 16 bits and a group size of $m = 4$ bit.

FF is the only one that actually cuts the critical path; the other $3m$ FFs are just to balance the pipeline.

Here, all of the output registers use the registers of the BLEs computing the addition and can be considered to come for free. However, the registers for balancing the inputs come at a cost: they have to come from other BLEs. In case of several FFs in a sequence, they may use the shift register LUT (SRL) as discussed in Sect. 4.3.2. In any case, the number of BLEs for pipeline balancing may be higher than the BLEs required for the actual summation.

Figure 5.29b shows a solution presented in [DNP10] that avoids about half of the BLEs required for input balancing. It uses the fact that in terms of BLE consumption, a FF has a similar cost as a FF + FA. Hence, m bit adders are used in the first stage to compress the amount of bits to store within the following pipeline stages. Then, the following adders are just used to add the carries of the previous stage.

The pipelined adder is a very generic way to speed up adders to any word size at a reasonable cost, but at the expense of a potentially high latency. To reduce this latency, further logic can be used to reduce the delay.

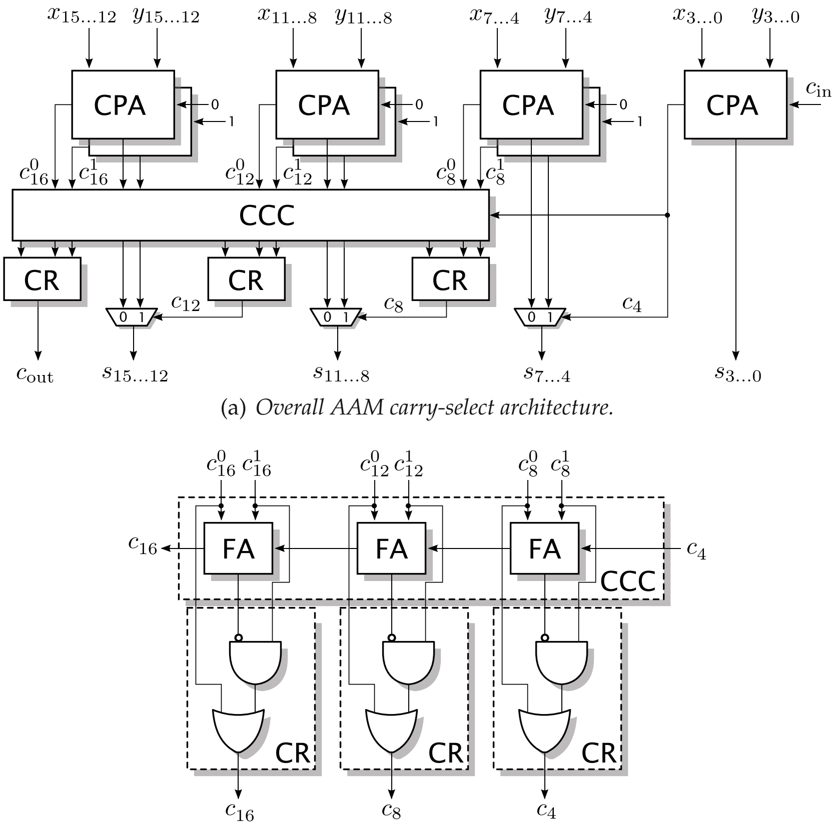
Hands on: Pipelined adders
The FloPoCo core generator can be used to build pipelined adders that meet a given target frequency. The following FloPoCo call will generate a pipelined adder according to Fig. 5.29a with 1024 bit input word size:

```
flopoco frequency=300 IntAdder wIn=1024
```

The reader is invited to play with the optional parameters of `IntAdder` and also with the `target` and `frequency` parameters to observe their impact on the group size and pipeline depth.

5.5.2 *Fast Combinatorial Adders*

One method for reducing the delay was first presented in [DNP10] and later improved in [NPP11; RHG14; LPB19]. We present the idea of the AAM carry-select architecture of [NPP11] in the following. This architecture basically applies the carry-select idea, but uses a different selection scheme. The architecture is shown in Fig. 5.30a for the example of a 16-bit adder split into groups of $m = 4$ bits. Note that this is a toy size for explaining the concept, as a delay benefit will only occur for much larger sizes (256 bits or more [NPP11]).



(a) *Overall AAM carry-select architecture.*

(b) *Carry computation circuit (CCC) with carry recovery (CR).*

Fig. 5.30 add-add multiplex (AAM) carry-select architecture for a 16-bit adder.

Like in the carry-select scheme (see Sect. 5.3.2), two adders are applied per segment in the first stage, computing both possible results for carry-in equals 0 and 1 in parallel. The resulting carry candidates c_i^0 and c_i^1 of the first stage are now used to select the correct carries. In contrast to the classic approach, this is not performed by MUXes but by using fast prefix computations that can be efficiently mapped to the fast carry chain. These are shown as a carry computation circuit (CCC) block in Fig. 5.30a and a small LUT-based carry recovery (CR) circuit.

The CCC as well as the CR circuits are shown in Fig. 5.30b. The CCC is nothing less than a ripple-carry adder. Table 5.2 shows the truth table how the actual carry c_i is computed from both carry candidates c_i^0 and c_i^1 computed in the first stage and the carry of the previous group c_{i-m}. The same cases like in a full adder occur, except that the propagate case $c_i^0\overline{c_i^1}$ is obviously impossible. Hence, the output carry c_i can be computed by

Table 5.2 Cases in carry-propagation in the carry-select scheme.

c_i^0	c_i^1	c_i	Case
0	0	0	kill
0	1	c_{i-m}	propagate
1	0	–	*impossible*
1	1	1	generate

$$c_i = c_i^0 \vee c_{i-m} c_i^1. \tag{5.63}$$

To avoid the slow propagation using the general routing, the idea is to use an RCA, in which each FA computes the sum

$$s_i = c_i^0 \oplus c_i^1 \oplus c_{i-m} \tag{5.64}$$

$$= \begin{cases} c_{i-m} & \text{kill or generate case} \\ \overline{c_{i-m}} & \text{propagate case} \end{cases} . \tag{5.65}$$

Hence, the term c_{i-m} in (5.63) can now be replaced by $\overline{s_i}$ as obtained in the CCC block

$$c_i = c_i^0 \vee \overline{s_i}\, c_i^1. \tag{5.66}$$

The remaining logic is computed in the CR block (see Fig. 5.30b). Note that the logic of the CR block can be implemented together with the MUX in a single LUT5 per output bit.

The complete architecture is also cheap to pipeline as most of the FFs are otherwise not used in the BLEs that implement the logic. The results presented in [NPP11] clearly indicate that this architecture consumes similar resources compared to the pipelined adder discussed in Sect. 5.5.1 when targeting the same speed, but provides a single cycle latency for large adders beyond 256 bits.

This core idea consisting of a carry-select stage with selection units utilizing the prefix computations in the fast carry chain has been further improved in [LPB19]. Here, several variants of computing the partial additions in the first stage, the prefix network (CCC block in Fig. 5.30), as well as the selection/carry recovery in the last stage have been systematically analyzed. Results were presented for different prefix networks, including Brent-Kung, Han-Carlson, Kogge-Stone, and Sklansky for huge adders computing 1024 to 8192 bits. These show a “quasi linear relationship between prefix tree complexity and performance” [LPB19]. Hence, the data presented in [LPB19] should help to select the right architecture that fits best a given application.

References

[AB95] E. Abu-Shama and M. A. Bayoumi. “A New Cell for Low Power Adders”. In: *International Midwest Symposium on Circuits and Systems*. 1995, pp. 1014–1017. (cit. on p. 107).

[Alt13] *Stratix V Device Handbook*. Altera Corporation. 2013. (cit. on p. 132).

[AP02] Massimo Alioto and Gaetano Palumbo. “Analysis and Comparison on Full Adder Block in Submicron Technology”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10.6 (2002), pp. 806–823. (cit. on p. 114).

[Bae+09] Gregg Baeckler, Martin Langhammer, James Schleicher, and Richard Yuan. “Logic Cell Supporting Addition of Three Binary Words”. U.S. pat. 7565388. Altera Corporation. 2009. (cit. on p. 132).

[Bau98] Friedrich L. Bauer. “Zuse, Aiken und der einschrittige Übertrag”. In: *Informatik-Spektrum* 21.5 (1998), pp. 279–281. (cit. on p. 114).

[Bha+15] Partha Bhattacharyya, Bijoy Kundu, Sovan Ghosh, Vinay Kumar, and Anup Dandapat. “Performance Analysis of a Low-Power High-Speed Hybrid 1-bit Full Adder Circuit”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23.10 (2015), pp. 2001–2008. (cit. on pp. 107, 114).

[BK1] R.P. Brent and H.T. Kung. “A Regular Layout for Parallel Adders”. In: *IEEE Transactions on Computers* C-31.3 (1), pp. 260–264. (cit. on p. 122).

[BL01] Andrew Beaumont-Smith and Cheng-Chew Lim. “Parallel Prefix Adder Design”. In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 218–225. (cit. on p. 122).

[BWJ03] Hung Tien Bui, Yuke Wang, and Yingtao Jiang. “Design and Analysis of Low-Power 10-Transistor Full Adders Using Novel XORXNOR Gates”. In: *IEEE Transactions On Circuits And Systems II: Analog And Digital Signal Processing* 49.1 (2003). (cit. on p. 107).

[DNP10] Florent de Dinechin, Hong Diep Nguyen, and Bogdan Pasca. “Pipelined FPGA Adders”. In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2010, pp. 422–427. (cit. on pp. 135, 137).

[EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004. (cit. on pp. 116, 118, 121, 131).

[FO01] Michael J. Flynn and Stuart F. Oberman. *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001. (cit. on p. 111).

[GKP94] Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. *Concrete Mathematics – A Foundation for Computer Science*. Addison-Wesley Professional, 1994. (cit. on p. 115).

[Har03] David Harris. "A Taxonomy of Parallel Prefix Networks". In: *Asilomar Conference on Signals, Circuits and Systems*. IEEE, 2003, pp. 2213–2217. (cit. on p. 122).

[Int17] *Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide*. Intel Corporation. 2017. (cit. on p. 132).

[Int18a] *Intel Stratix 10 High-Performance Design Handbook*. Intel Corporation. 2018. (cit. on p. 132).

[KG16] Petter Källström and Oscar Gustafsson. "Fast and Area Efficient Adder for Wide Data in Recent Xilinx FPGAs". In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2016. (cit. on p. 135).

[KKZ16] Martin Kumm, Marco Kleinlein, and Peter Zipf. "Efficient Sum of Absolute Difference Computation on FPGAs". In: *International Conference on Field Programmable Logic and Application (FPL)*. IEEE, 2016, pp. 1–4. (cit. on pp. 133, 134).

[KS73] Peter M. Kogge and Harold S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". In: *IEEE Transactions on Computers* C-22.8 (1973), pp. 786–793. (cit. on p. 121).

[Kum+13] Martin Kumm, Martin Hardieck, Jens Willkomm, Peter Zipf, and Uwe Meyer-Baese. "Multiple Constant Multiplication with Ternary Adders". In: *International Conference on Field Programmable Logic and Application (FPL)*. 2013, pp. 1–8. (cit. on p. 132).

[LPB19] Martin Langhammer, Bogdan Pasca, and Gregg Baeckler. "High Precision, High Performance FPGA Adders". In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 298–306. (cit. on pp. 135, 137, 139).

[NPP11] Hong Diep Nguyen, Bogdan Pasca, and Thomas B. Preußer. "FPGA-Specific Arithmetic Optimizations of Short-Latency Adders". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2011, pp. 232–237. (cit. on pp. 135, 137, 139).

[Oka+00] Souichi Okada, Naoya Torii, Kouichi Itoh, and Masahiko Takenaka. "Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA". In: *Cryptographic Hardware and Embedded Systems*. Springer, 2000, pp. 25–40. (cit. on p. 104).

[Pat+07] Dinesh Patil, Omid Azizi, Mark Horowitz, Ron Ho, and Rajesh Ananthraman. "Robust Energy-Efficient Adder Topologies". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2007, pp. 29–37. (cit. on p. 122).

[RHG14] Marcin Rogawski, Ekawat Homsirikamol, and Kris Gaj. "A Novel Modular Adder for One Thousand Bits and More Using Fast Carry Chains of Modern FPGAs". In: *International Con-*

ference on Field Programmable Logic and Application (FPL). IEEE, 2014, pp. 1–8. (cit. on pp. 135, 137).

[Roj97] Raúl Rojas. "Konrad Zuse's Legacy: The Architecture of the Z1 and Z3". In: *IEEE Annals of the History of Computing* 19.2 (1997). (cit. on p. 114).

[Roy+21] Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. "PrefixRL: Optimization of Parallel Prefix Circuits using Deep Reinforcement Learning". In: *Design Automation Conference (DAC)*. ACM/IEEE, 2021, pp. 853–858. (cit. on p. 123).

[SB00] Ahmed M. Shams and Magdy A. Bayoumi. "A Novel High-Performance CMOS 1-Bit Full-Adder Cell". In: *IEEE Transactions On Circuits And Systems II: Analog And Digital Signal Processing* 47.5 (2000). (cit. on pp. 107, 114).

[SE01] P.-M. Seidel and G. Even. "On the Design of Fast IEEE Floating-Point Adders". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2001, pp. 184–194. (cit. on p. 124).

[Skl60] J. Sklansky. "Conditional-Sum Addition Logic". In: *IEEE Transactions on Electronic Computers* EC-9.2 (1960), pp. 226–231. (cit. on p. 119).

[Soh+22] Jongwook Sohn, David K. Dean, Eric Quintana, and Wing Shek Wong. "Enhanced Floating-Point Adder with Full Denormal Support". In: *Symposium on Computer Arithmetic (ARITH)*. IEEE, 2022. (cit. on pp. 124, 125).

[SP06] James M. Simkins and Brian D. Philofsky. "Structures and Methods for Implementing Ternary Adders/Subtractors in Programmable Logic Devices". U.S. pat. 7274211. Xilinx Inc. 2006. (cit. on p. 132).

[TVC17] V. Torres, J. Valls, and M.J. Canet. "Optimised CORDIC-based atan2 computation for FPGA implementations". In: *Electronics Letters* 53.19 (2017), pp. 1296–1298. (cit. on p. 129).

[Wir04] Michael J. Wirthlin. "Constant Coefficient Multiplication Using Look-Up Tables". In: *Journal of VLSI Signal Processing* 36.1 (2004), pp. 7–15. (cit. on p. 129).

[Xil14] *Virtex-6 FPGA Data Sheet: DC and Switching Characteristics (DS152)*. Xilinx. 2014. (cit. on pp. 126, 127).

[XY98] Shanzhen Xing and W W H Yu. "FPGA Adders: Performance Evaluation and Optimal Design". In: *IEEE Design & Test of Computers* 15.1 (1998), pp. 24–29. (cit. on p. 135).

[YN08] Romana Yousuf and Najeeb-ud-din. "Synthesis of carry select adder in 65 nm FPGA". In: *IEEE Region 10 Conference (TENCON)*. 2008, pp. 1–6. (cit. on p. 135).

[Zim97] Reto Zimmermann. "Binary Adder Architectures for Cell-Based VLSI and their Synthesis". PhD thesis. Swiss Federal Institute of Technology, Zurich, 1997. (cit. on pp. 120, 121, 122, 123).

[Zus84] Konrad Zuse. *Der Computer – Mein Lebenswerk*. Springer, 1984. (cit. on p. 114).

[ZW92] N. Zhuang and H. Wu. “A new design of the CMOS full adder”. In: *IEEE Journal on Solid-State Circuits* 27 (1992), pp. 840–844. (cit. on p. 107).

CHAPTER 6

Fixed-Point Comparison

All animals are equal, but some animals are more equal than others.
G. Orwell, *Animal Farm*

This chapter studies the comparison of two binary numbers in order to rank them. In a generic-purpose processor, such a comparison is usually performed by subtraction, using an existing integer adder. Assuming that the synthesis tools will remove all the hardware that computes the sum bits, this approach is valid in an application-specific architecture. However, devising integer comparison from first principles allows for specific optimizations in some contexts.

6.1 Introduction

This chapter addresses the construction of the operator whose interface is shown in Fig. 6.1. It compares two w-bits unsigned binary integers X and Y and outputs up to three mutually exclusive Boolean signals $X < Y$, $X = Y$, and $X > Y$.

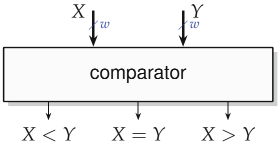


Fig. 6.1 Interface to an integer comparator (it may also output a subset of the comparison bits).

F. de Dinechin, M. Kumm, *Application-Specific Arithmetic*,
https://doi.org/10.1007/978-3-031-42808-1_6

This operator is needed for the floating-point comparator of Sect. 11.5, but also inside many other operators, for instance, to compare exponents in floating-point addition and other summations. Its specialization to a constant Y is used in several places in Chap. 15.

6.1.1 Fixed-Point Considerations

The title of this chapter mentions fixed-point comparison, and Fig. 6.1 only shows a comparator of unsigned integers. Indeed, fixed-point comparison is trivially (i. e., at no cost) reduced to unsigned integer comparison:

- Two unsigned fixed-point numbers X and Y in the same format ufix(m, ℓ) compare the same as the integers $2^\ell X$ and $2^\ell Y$. In other words, the position of the point is irrelevant in the comparison.
- Two unsigned fixed-point numbers X and Y of different formats ufix(m_X, ℓ_X) and ufix(m_Y, ℓ_Y) can be compared by first converting them to the smallest larger common format ufix(m, ℓ) where $m = \max(m_X, m_Y)$ and $\ell = \min(\ell_X, \ell_Y)$ (see Sect. 2.2.3). This conversion does not change the value of either number and therefore does not change the comparison result. Since it involves padding one number with constant zeroes, it enables some of the optimizations related to comparison to a constant operand that will be studied in Sect. 6.3.3.
- In two's complement data, the most significant bit (MSB) has the same weight as it would have in an unsigned representation, but with a negative sign. A consequence is that two signed numbers X and Y of the same format compare the same as the unsigned numbers obtained by exchanging the MSBs of X and Y. Of course this bit exchange is at no cost.

Therefore, the remainder of this chapter focuses on unsigned integer comparison.

6.1.2 Area and Delay Considerations

In principle $X = Y$ can be computed as the logical AND of the individual equality bits $x_i = y_i$. It costs w NXOR gates (which can operate in parallel) and a tree of AND gates that can operate in $\log w$ time.

The Boolean $X < Y$ is the sign bit of the subtraction $X - Y$ and is also its carry out. It can therefore also be computed in logarithmic time (using only the leftmost tree of any of the prefix trees of Chap. 5). Its negation provides $X \geq Y$, and of course $X > Y$ and $X \leq Y$ can be obtained by simply swapping X and Y.

The technique described below does not fundamentally improve these complexities. However, the focus on the comparison operation makes it easier to optimize a comparator for a given context, especially when targeting FPGA technology.

6.2 Basic Binary Tree Comparison

The core idea is simple: given $k = \lceil w/2 \rceil$, X is split into its higher bits X_H and lower bits X_L, or $X = 2^k X_H + X_L$. Similarly Y is split as $Y = 2^k Y_H + Y_L$. Then we have two cases:

- If $X_H = Y_H$ then the order of X and Y is the order of X_L and Y_L.
- If $X_H < Y_H$ or $X_H > Y_H$, we do not need to consider the comparison of X_L and Y_L to decide the ordering of X and Y.

Table 6.1 summarizes these situations. It is then possible to recursively split X_H, Y_H, X_L, and Y_L. This leads to a binary tree. At the leaves are individual bit comparisons.

It remains to encode the three cases of Table 6.1 in binary, which requires two bits. A natural idea is to choose the encoding that we have at the leaves, so that no logic is needed at the leaves: $X > Y$ is encoded by 10, $X < Y$ is encoded by 01, and $X = Y$ is encoded by either 00 or 11.

The previous tree provides the three comparison output bits encoded as two bits. If only $X = Y$ is needed, then the problem obviously becomes simpler, as the equality can be encoded on one bit only (and we obtain the AND tree of XNORs already evoked). However, the problem is not simpler if only $X < Y$ is needed: with the exception of the root node, all nodes in the tree must still distinguish between the three cases $X < Y$, $X_H = Y_H$, and $X_H > Y_H$, which require two bits to encode.

Table 6.1 Divide-and-conquer comparison when X and Y are split in two.

X cmp Y	$X_H < Y_H$	$X_H = Y_H$	$X_H > Y_H$
$X_L < Y_L$	$X < Y$	$X < Y$	$X > Y$
$X_L = Y_L$	$X < Y$	$X = Y$	$X > Y$
$X_L > Y_L$	$X < Y$	$X > Y$	$X > Y$

6.3 FPGA-Specific Implementations

6.3.1 *Exploiting Fast Carry Logic*

It is trivial that the $X < Y$ bit is the carry-out of $X - Y$ and can therefore be computed in w LUTs through the fast carry chain. On most moderns FPGAs, however, it can be further improved to $w/2$ LUT4s [PZC10] by exploiting the observations made in Sect. 5.4, p. 125. As shown there, most basic logic elements (BLEs) of modern FPGAs allow to compute the full addition of two functions, i. e., $\mathrm{FA}(p(a,b,c,d), q(a,b,c,d), c_i)$. The idea is then to have (a,b,c,d) consisting of two aligned 2-bit chunk X_i and Y_i, respectively, of X and of Y. Then p and q, respectively, compute one bit x'_i and one bit y'_i that compare the same as X_i and Y_i (see Table 6.2). Finally, a subtracter based on the fast carry logic computes $X'_i - Y'_i$, whose sign is that of $X - Y$.

This is the most area-efficient solution if only the $X < Y$ bit is required. It is also the default method used by synthesis tools, which thus implement $X < Y$ on w-bit integers using $\lceil w/2 \rceil$ LUTs.

Similarly, the $X = Y$ bit can be computed as a wide AND of equality comparisons on large chunks. On recent AMD devices, it is possible to chain LUT6 that each compare 3-bit chunks, for a total cost of $\lceil w/3 \rceil$ LUTs.

These two approaches are the most area-efficient, and with fast carry logic, they also provide the best possible latency for sizes up to 64 bits.

Table 6.2 Encoding the comparison of two integers X_i and Y_i as two bits that compare similarly.

	$X_i < Y_i$	$X_i = Y_i$	$X_i > Y_i$
x'_i	0	0	1
y'_i	1	0	0

6.3.2 *Two-Level Tree of Fast Carry Logic*

For comparisons of very large numbers, the latency of the previous approach (a $w/2$-bit carry propagation) may be too large. In this case, a good option is to split X and Y in chunks of k bits and perform in parallel the comparison of each chunk. The result of this comparison is then encoded as two k'-bit synthetic integers X' and Y' (see Table 6.2), and these two integers are compared again using a fast carry comparison. In terms of delay, the optimal choice of k and k' is to use $k \approx k' \approx \sqrt{w}$. In a pipeline design, it is also possible to choose k or k' in order to balance the pipeline stages. This is the choice made by FloPoCo when a comparator is part of a larger pipelined design: k is chosen in such a way that the first level completes the remaining delay available in the current pipeline stage.

This solution is fast, but for a single comparison (e. g., $X < Y$), its cost is much higher than the $w/2$ LUTs of the plain fast carry solution: the area overhead of the second level can be kept small (about $\sqrt{w}/2$), but the first level also adds the cost of the computation of the chunk equality bits (about $w/3$ LUTs).

Other solutions are possible, for instance, a LUT6 allows for a ternary comparison tree instead of the binary tree of Sect. 6.2, while fast carry logic allows for a p-level tree of $\sqrt[p]{w}$-bit comparators. These solutions can be mixed and matched, but for practical sizes, the plain fast carry approach is always the most area-efficient, while a two-level tree provides very low latency for sizes of several thousand bits.

Hands on: Large comparators in FloPoCo
The following command implements a two-level 1000 bit comparator, each level using fast carry logic:

```
flopoco IntComparator w=1000
```

6.3.3 *Comparing to a Constant*

In a tree-based approach, the logic optimizer of a synthesis tool will remove parts of the tree if Y is a constant. The main benefit is in LUT-based optimizations: each LUT-α may now compare α bits of the input with the constant. Then, the fast carry logic may be used as previously.

At the time of writing this book, AMD tools do not implement this optimization: they use the logic optimizer for small w and default to a standard comparator for larger w.

Hands on: Large constant comparators in FloPoCo
The following command implements a constant comparator that decomposes the 64-bit input in 13 chunks of 5 bits and links the comparison of these chunks using fast carry logic:

```
flopoco IntConstantComparator flags=1 w=64 \
    c=17979737894628297144
```

References

[PZC10] Stefania Perri, Paolo Zicari, and Pasquale Corsonello. “Efficient Absolute Difference Circuits in Virtex-5 FPGAs”. In: *5th IEEE Mediterranean Electrotechnical Conference (Melecon)*. IEEE. 2010, pp. 309–313. (cit. on p. 148).