Chapter 6

Addition

In this chapter we define binary adders. We start by considering a Ripple Carry Adder. This is a design with linear delay and linear cost. We then present designs with logarithmic delay but super-linear cost.

6.1 Definition of a binary adder

Definition 37 *A* binary adder *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$, *and* $C[0] \in \{0,1\}$.

Output: $S[n-1:0] \in \{0,1\}^n$ *and* $C[n] \in \{0,1\}$.

Functionality:

$$\langle \vec{S} \rangle + 2^n \cdot C[n] = \langle \vec{A} \rangle + \langle \vec{B} \rangle + C[0] \tag{6.1}$$

The inputs $\vec{A}$ and $\vec{B}$ are the binary representations of the addends. The input $C[0]$ is often called the *carry-in bit*. The output $\vec{S}$ is the binary representation of the sum, and the output $C[n]$ is often called the *carry-out bit*.

Question 33 *Verify that the functionality of* ADDER(n) *is well defined. Namely, for every* $A[n-1:0], B[n-1:0]$, *and* $C[0]$ *there exist* $S[n-1:0]$ *and* $C[n]$ *that satisfy Equation 6.1.*

Hint: Show that the set of numbers that can be represented by sums $\langle \vec{A} \rangle + \langle \vec{B} \rangle + C[0]$ *equals the set of numbers that can be represented by sums* $\langle \vec{S} \rangle + 2^n \cdot C[n]$.

There are many ways to implement an ADDER(n). In this chapter we present a few ADDER(n) designs.

Question 34 *Prove lower bounds on the cost and delay of combinational circuits that implement an* ADDER(n).

6.2 Ripple Carry Adder

Ripple Carry Adders are adders that are built of a chain of Full-Adders. We denote a Ripple Carry Adder that implements an ADDER(n) by RCA(n). A Full-Adder is a combinational circuit that adds three bits and represents their sum in binary representation.

Definition 38 (Full-Adder) *A* Full-Adder *is a combinational circuit with* 3 *inputs* $x, y, z \in \{0,1\}$ *and* 2 *outputs* $c, s \in \{0,1\}$ *that satisfies:*

$$2c + s = x + y + z.$$

The output s of a Full-Adder is often called *the sum output*. The output c of a Full-Adder is often called *the carry-out output*. We denote a Full-Adder by FA.

A Ripple Carry Adder, RCA(n), is built of a chain of Full-Adders. An RCA(n) is depicted in Figure 6.1. Note that the carry-out output of the ith Full-Adder is denoted by $c[i+1]$. The weight of $c[i+1]$ is 2^{i+1}. This way, the weight of every signal is two to the power of its index. One can readily notice that an RCA(n) adds numbers using the same addition algorithm that we use for adding numbers by hand.

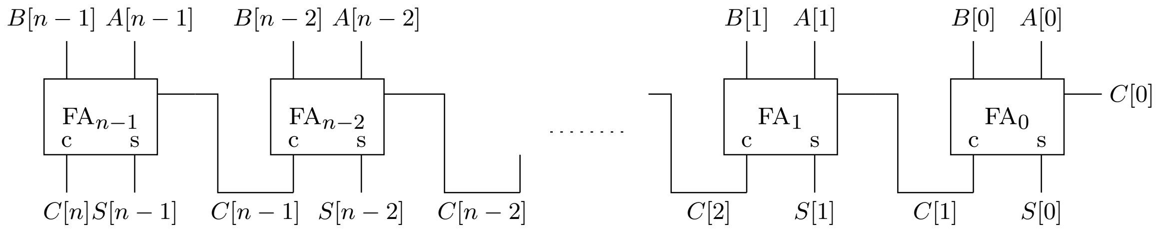

Figure 6.1: A Ripple Carry Adder RCA(n).

6.2.1 Correctness proof

In this section we prove the correctness of an RCA(n). To facilitate the proof, we use an equivalent recursive definition of RCA(n). The recursive definition is as follows.

Basis: an RCA(1) is simply a Full-Adder. Step: a recursive description of RCA(n), for $n \geq 1$, is depicted in Figure 6.2.

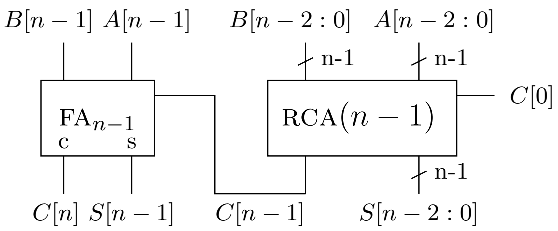

Figure 6.2: A recursive description of RCA(n).

The following claim deals with the correctness of RCA(n).

Claim 24 RCA(n) *is a correct implementation of* ADDER(n).

Proof: The proof is by induction on n. The induction basis, for $n = 1$, follows directly from the definition of a Full-Adder. The induction step is proved as follows.

The induction hypothesis, for $n-1$, is

$$\langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0] = 2^{n-1} \cdot C[n-1] + \langle S[n-2:0]\rangle. \tag{6.2}$$

The definition of a Full-Adder states that

$$A[n-1] + B[n-1] + C[n-1] = 2 \cdot C[n] + S[n-1]. \tag{6.3}$$

Multiply Equation 6.3 by 2^{n-1} to obtain

$$2^{n-1} \cdot A[n-1] + 2^{n-1} \cdot B[n-1] + 2^{n-1} \cdot C[n-1] = 2^n \cdot C[n] + 2^{n-1} \cdot S[n-1]. \tag{6.4}$$

Note that $2^{n-1} \cdot A[n-1] + \langle A[n-2:0]\rangle = \langle A[n-1:0]\rangle$. By adding Equations 6.2 and 6.4 we obtain:

$$2^{n-1} \cdot C[n-1] + \langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = 2^n \cdot C[n] + 2^{n-1} \cdot C[n-1] + \langle S[n-1:0]\rangle.$$

Cancel out $2^{n-1} \cdot C[n-1]$, and the claim follows. □

6.2.2 Delay and cost analysis

The cost of an RCA(n) satisfies:

$$c(\text{RCA}(n)) = n \cdot c(\text{FA}) = \Theta(n).$$

The delay of an RCA(n) satisfies

$$d(\text{RCA}(n)) = n \cdot d(\text{FA}) = \Theta(n).$$

The answer to Question 34 implies that the cost of RCA(n) is optimal, but its delay is exponentially far away from the optimum delay. The clock rates in modern microprocessors correspond to the delay of 15-20 gates. Most microprocessors easily add 32-bit numbers within one clock cycle. Obviously, adders in such microprocessors are not Ripple Carry Adders. In the rest of the chapter we present faster ADDER(n) designs.

6.3 Carry bits

The easiest way to define the carry bit associated with an addition

$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = \langle S[n-1:0]\rangle + 2^n \cdot C[n] \tag{6.5}$$

is by considering the internal signals $C[n-1:1]$ of an RCA(n). We extend this definition of the carry bits to the input $C[0]$ and the output $C[n]$. Hence there are $n+1$ carry-bits $C[n:0]$ associated with the addition defined in Equation 6.5.

We discuss a few issues related to the definition of the carry bits and binary addition.

1. Consider the functionality of an adder:
$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = 2^n \cdot C[n] + \langle S[n-1:0]\rangle.$$
Let $x = \langle \vec{A}\rangle + \langle \vec{B}\rangle + C[0]$. The above equality means that the sum x admits two representations. The representation of x on the right hand side is unique (this is the binary representation). Namely, given x, the bit $C[i+1]$ and the string $S[i:0]$ are uniquely determined. We refer to such a representation as a *non-redundant representation*. One characteristic of non-redundant representation is that if one wishes to compare two numbers x, y represented in the same non-redundant representation by X and Y, respectively, then it suffices to compare their representations X and Y.

   The left hand side represents the same value x but uses two binary strings and a carry-in bit. Given x, there are many possible combinations of values of $\langle \vec{A}\rangle, \langle \vec{B}\rangle$ and $C[0]$. For example: $8 = 4 + 3 + 1$, and also $8 = 5 + 3 + 0$.

   We refer to such a representation as *redundant representation*. Comparison of values represented in redundant representation is not as easy as it is with non-redundant representation. For example, assume that
$$x = \vec{A} + \vec{B}$$
$$x' = \vec{A}' + \vec{B}'.$$
It is possible that $x = x'$ even though $A \neq A'$ and $B \neq B'$. Namely, inequality of the representations does not imply inequality of the represented values.

   An alternative way to interpret an $\textsc{rca}(n)$ (or an $\textsc{adder}(n)$, in general) is to say that it translates a redundant representation to a non-redundant binary representation.

2. The correctness proof of $\textsc{rca}(n)$ implies that, for every $0 \le i \le n-1$,
$$\langle A[i:0]\rangle + \langle B[i:0]\rangle + C[0] = 2^{i+1} \cdot C[i+1] + \langle S[i:0]\rangle. \tag{6.6}$$
This equality means that the cone of the outputs $C[i+1]$ and $S[i:0]$ is a subset of the inputs $A[i:0] \bigcup B[i:0] \bigcup C[0]$. In fact, this is the cone of $S[i]$ and $C[i+1]$.

3. Equation 6.6 implies that, for every $0 \le i \le n-1$,
$$\langle S[i:0]\rangle = \mathrm{mod}(\langle A[i:0]\rangle + \langle B[i:0]\rangle + C[0], 2^{i+1}).$$

4. The correctness of $\textsc{rca}(n)$ implies that, for every $0 \le i \le n-1$,
$$S[i] = \textsc{xor}(A[i], B[i], C[i]). \tag{6.7}$$
This immediately implies that, for every $0 \le i \le n-1$,
$$C[i] = \textsc{xor}(A[i], B[i], S[i]). \tag{6.8}$$

5. Equations 6.7 and 6.8 imply constant-time linear-cost reductions between the problems of computing the sum bits $S[n-1:0]$ and computing the carry bits $C[n-1:0]$. The task of computing the sum bits is the task of an adder. We will later design an optimal adder that will first compute the carry bits, and then obtain the sum bits from the carry-bit by applying bit-wise XOR.

6.4 Conditional Sum Adder

A Conditional Sum Adder is an adder design that is based on divide-and-conquer.

6.4.1 Motivation

Imagine a situation in which Alice positioned on Earth holds the strings $A[k-1:0], B[k-1:0], C[0]$. Bob, stationed on the Moon holds the strings $A[n-1:k], B[n-1:k]$. The goal of Alice and Bob is to jointly compute the sum $\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0]$. They don't care who holds the sum bits and $C[n]$, as long as one of them does. Now, sending information from Alice to Bob is costly. How many bits must Alice send to Bob? It suffices to send $C[k]$ to Bob. Now, sending information from Alice to Bob takes time. Even at the speed of light, it takes a second, which is a lot compared to the time it takes to compute the sum. Suppose Bob wants to finish his task as soon as possible after receiving $C[k]$ from Alice. How what should Bob do during the second it takes $C[k]$ to travel to the Moon? Since the message may be either one or zero, an industrious Bob will compute two sums; one under the assumption that $C[k] = 0$, and one under the assumption that $C[k] = 1$. Finally, when $C[k]$ arrives, Bob only needs to select between the two sums he has pre-computed. This story captures the idea behind a conditional sum adder.

6.4.2 Implementation

A Conditional Sum Adder is designed recursively using divide-and-conquer. A $\text{CSA}(1)$ is simply a Full-Adder. A $\text{CSA}(n)$, for $n > 1$ is depicted in Figure 6.3. The input is partitioned into a lower part consisting of the bits in positions $[k-1:0]$ and an upper part consisting of the bits in positions $[n-1:k]$. The lower part (handled by Alice in our short tale) is fed to a $\text{CSA}(k)$ to produce the sum bits $S[k-1:0]$ and the carry bit $C[k]$. The upper part (handled by Bob) is fed to two $\text{CSA}(n-k)$ circuits. The first one is given a carry-in of 0 and the second is given a carry-in of 1. These two $\text{CSA}(n-k)$ circuits output $n-k+1$ bits each. A multiplexer selects one of these outputs according to the value of $C[k]$ which arrives from the lower part.

Question 35 *Prove the correctness of the* $\text{CSA}(n)$ *design.*

6.4.3 Delay and cost analysis

To simplify the analysis we assume that $n = 2^\ell$. To optimize the cost and delay, we use $k = n/2$.

Let $d(\text{FA})$ denote the delay of a Full-Adder. The delay of a $\text{CSA}(n)$ satisfies the following recurrence:

$$d(\text{CSA}(n)) = \begin{cases} d(\text{FA}) & \text{if } n = 1 \\ d(\text{CSA}(n/2)) + d(\text{MUX}) & \text{otherwise.} \end{cases}$$

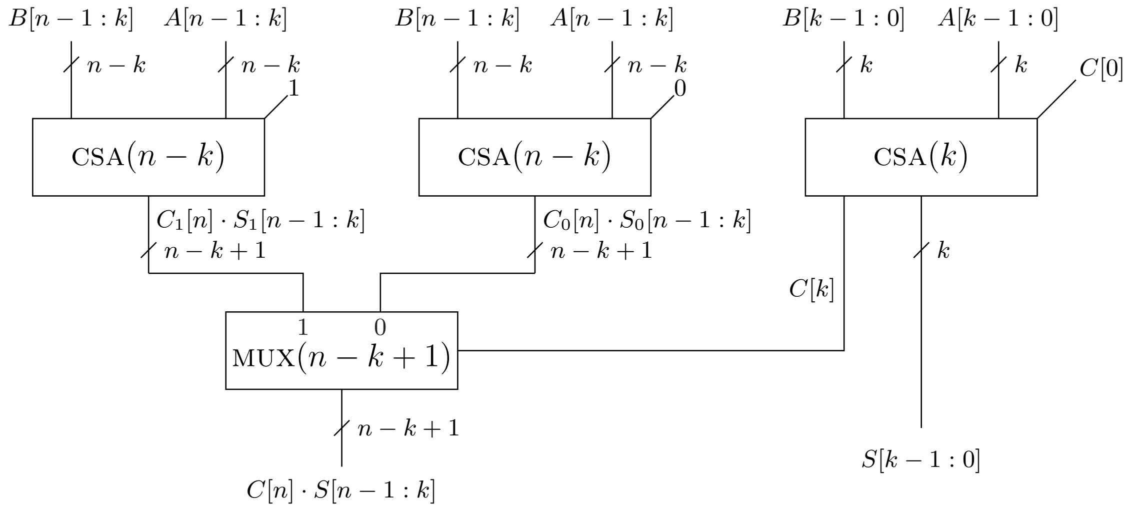

Figure 6.3: A Conditional Sum Adder $\text{CSA}(n)$.

It follows that the delay of a $\text{CSA}(n)$ is

$$\begin{aligned} d(\text{CSA}(n)) &= \ell \cdot d(\text{MUX}) + d(\text{FA}) \\ &= O(\log n). \end{aligned}$$

Let $c(\text{FA})$ denote the cost of a Full-Adder. The cost of a $\text{CSA}(n)$ satisfies the following recurrence:

$$c(\text{CSA}(n)) = \begin{cases} c(\text{FA}) & \text{if } n = 1 \\ 3 \cdot c(\text{CSA}(n/2)) + (n/2+1) \cdot c(\text{MUX}) & \text{otherwise.} \end{cases}$$

Those of you familiar with the master theorem for recurrences can use it to solve this recurrence. We will solve this recurrence from scratch.
We open two steps of the recurrence to get a feeling of how the different terms grow.

$$\begin{aligned} c(n) &= 3 \cdot c\left(\frac{n}{2}\right) + c\left(\text{MUX}\right) \cdot \left(\frac{n}{2} + 1\right) \\ &= 3 \cdot \left(3 \cdot c\left(\frac{n}{4}\right) + c(\text{MUX}) \cdot \left(\frac{n}{4} + 1\right)\right) + c(\text{MUX}) \cdot \left(\frac{n}{2} + 1\right) \\ &= 3^2 \cdot c\left(\frac{n}{4}\right) + c(\text{MUX}) \cdot \frac{n}{2} \cdot \left(1 + \frac{3}{2}\right) + (1+3) \cdot c\left(\text{MUX}\right) \end{aligned}$$

We now have a good guess (which can be proved by induction) that

$$c(n) = 3^\ell \cdot c\left(\frac{n}{2^\ell}\right) + c(\text{MUX}) \cdot \frac{n}{2} \cdot \left(1 + \frac{3}{2} + \cdots \left(\frac{3}{2}\right)^{\ell-1}\right) + (1 + 3 + \cdots + 3^{\ell-1}) \cdot c(\text{MUX}).$$

Since $\ell = \log_2 n$, it follows that

- $3^\ell = n^{\log_2 3}$

- $(1+3+\cdots+3^{\ell-1}) < 3^\ell/2 = \frac{1}{2}\cdot n^{\log_2 3}$
- $\frac{n}{2}\cdot\left(1+\frac{3}{2}+\cdots\left(\frac{3}{2}\right)^{\ell-1}\right) < n^{\log_2 3}$.

We conclude that

$$\begin{aligned} c(n) &< n^{\log_2 3}\cdot\left(c(\text{FA})+\frac{3}{2}\cdot c(\text{MUX})\right)\\ &\approx \Theta\left(n^{1.58}\right). \end{aligned}$$

We conclude that a $\text{CSA}(n)$ is rather costly - although, for the time being, this is the only adder we know whose delay is logarithmic. We do point out that the $\text{CSA}(n)$ design does allow us to use three half-sized adders (i.e. addends are $n/2$ bits long) in order to implement a full sized adder (i.e. addends are n bits long).

Question 36 (effect of fanout on $\text{CSA}(n)$) *The fanout of the carry-bit $C[k]$ is $n/2+1$ if $k=n/2$. Suppose that we associate a delay of $\log_2(f)$ with a fanout f. How would taking the fanout into account change the delay analysis of a $\text{CSA}(n)$?*

Suppose that we associate a cost $O(f)$ with a fanout f. How would taking the fanout into account change the cost analysis of a $\text{CSA}(n)$?

6.5 Compound Adder

The Conditional Sum Adder uses two adders in the upper part, one with a zero carry-in and one with a one carry-in. This motivates the definition of an adder that computes both the sum and the incremented sum. Such an adder is called a Compound Adder and is defined as follows.

Definition 39 *A* Compound Adder *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$.

Output: $S[n:0], T[n:0] \in \{0,1\}^{n+1}$.

Functionality:

$$\begin{aligned} \langle\vec{S}\rangle &= \langle\vec{A}\rangle + \langle\vec{B}\rangle\\ \langle\vec{T}\rangle &= \langle\vec{A}\rangle + \langle\vec{B}\rangle + 1. \end{aligned}$$

Note that a Compound Adder does not have carry-in input. To simplify notation, the carry-out bits are denoted by $S[n]$ for the sum and by $T[n]$ for the incremented sum.
We denote a compound adder with input length n by $\text{COMP-ADDER}(n)$.

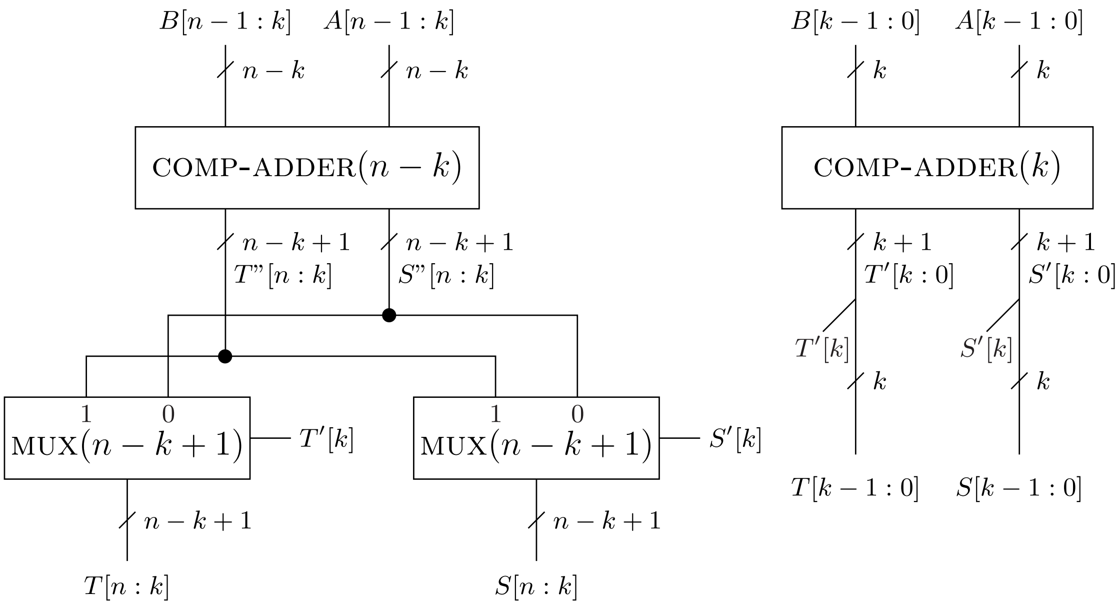

Figure 6.4: A Compound Adder COMP-ADDER(n).

6.5.1 Implementation

We apply divide-and-conquer to design a COMP-ADDER(n). For $n = 1$, we simply use a Full-Adder and a Half-Adder (one could optimize this a bit). The design for $n > 1$ is depicted in Figure 6.4.

Example 12 *Consider a* COMP-ADDER(4) *with input* $A[3:0] = 0110$ *and* $B[3:0] = 1001$. *The lower part computes* $S'[2:0] = 011$ *and* $T'[2:0] = 100$. *The two lower bits of the outputs are simply* $S[1:0] = S'[1:0] = 11$ *and* $T[1:0] = T'[1:0] = 00$. *The upper part computes* $S''[4:2] = 011$ *and* $T''[4:2] = 100$. *The output* $S[4:2]$ *is selected to be* $S''[4:2]$ *since* $S'[2] = 0$. *The output* $T[4:2]$ *is selected to be* $T''[4:2]$ *since* $T'[2] = 1$. *Hence* $S[4:0] = 01111$ *and* $T[4:0] = 10000$.

Question 37 *Present an example for* COMP-ADDER(4) *in which* $T[4:2]$ *is selected to be* $S''[4:2]$. *Is it possible that* $S'[k] = 1$ *and* $T'[k] = 0$? *Which combinations of* $S'[k]$ *and* $T'[k]$ *are possible?*

6.5.2 Correctness

We prove the correctness of COMP-ADDER(n).

Claim 25 *The* COMP-ADDER(n) *design depicted in Figure 6.4 is a correct adder.*

Proof: The proof is by induction on n. The case of $n = 1$ follows from the correctness of a Full-Adder and a Half-Adder. We prove the induction step for the output $S[n:0]$; the correctness of $T[n:0]$ is proved in a similar fashion.

The induction hypothesis implies that

$$\langle S'[k:0]\rangle = \langle A[k-1:0]\rangle + \langle B[k-1:0]\rangle. \tag{6.9}$$

Note that (i) the output $S[k-1:0]$ equals $S'[k-1:0]$, and (ii) $S'[k]$ equals the carry bit $C[k]$ corresponding to the addition $\langle A[k-1:0]\rangle + \langle B[k-1:0]\rangle$.

The induction hypothesis implies that

$$\begin{aligned}\langle S''[n:k]\rangle &= \langle A[n-1:k]\rangle + \langle B[n-1:k]\rangle\\ \langle T''[n:k]\rangle &= \langle A[n-1:k]\rangle + \langle B[n-1:k]\rangle + 2^k.\end{aligned} \tag{6.10}$$

It follows from Equations 6.9 and 6.10 that

$$\langle S''[n:k]\rangle + \langle S'[k:0]\rangle = \langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle \tag{6.11}$$

We consider two cases of the carry bit $C[k]$: $C[k]=0$ and $C[k]=1$.

1. If $C[k]=0$, then $S'[k]=0$. Equation 6.11 then reduces to

$$\begin{aligned}\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle &= \langle S''[n:k]\rangle + \langle S'[k-1:0]\rangle\\ &= \langle S[n:k]\rangle + \langle S[k-1:0]\rangle = \langle S[n:0]\rangle.\end{aligned}$$

2. If $C[k]=1$, then $S'[k]=1$. Equation 6.11 then reduces to

$$\begin{aligned}\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle &= \langle S''[n:k]\rangle + 2^k + \langle S'[k-1:0]\rangle\\ &= \langle T''[n:k]\rangle + \langle S[k-1:0]\rangle = \langle S[n:0]\rangle.\end{aligned}$$

In both cases, the output $S[n:0]$ is as required, and the claim follows. □

6.5.3 Delay and cost analysis

To simplify the analysis we assume that $n=2^\ell$. To optimize the cost and delay, we use $k=n/2$.

The delay of a COMP-ADDER(n) satisfies the following recurrence:

$$d(\textsc{comp-adder}(n)) = \begin{cases} d(\textsc{fa}) & \text{if } n=1\\ d(\textsc{comp-adder}(n/2)) + d(\textsc{mux}) & \text{otherwise.}\end{cases}$$

It follows that the delay of a COMP-ADDER(n) is

$$\begin{aligned}d(\textsc{comp-adder}(n)) &= \ell\cdot d(\textsc{mux}) + d(\textsc{fa})\\ &= \Theta(\log n).\end{aligned}$$

This recurrence is identical to that of $d(\textsc{csa}(n))$, hence $d(\textsc{comp-adder}(n)) = \Theta(\log n)$.

Note that the fanout of $S'[k]$ and $T'[k]$ is $n/2+1$. If the effect of fanout on delay is taken into account, then, as in the case of CSA(n), the delay is actually $\Theta(\log^2 n)$.

The cost of a $\text{COMP-ADDER}(n)$ satisfies the following recurrence:

$$c(\text{COMP-ADDER}(n)) = \begin{cases} c(\text{FA}) + c(\text{HA}) & \text{if } n = 1 \\ 2 \cdot c(\text{COMP-ADDER}(n/2)) + (n/2 + 1) \cdot c(\text{MUX}) & \text{otherwise.} \end{cases}$$

We are already familiar with such a recurrence and conclude that $c(\text{COMP-ADDER}) = \Theta(n \log n)$.

6.6 Summary

We started by defining binary addition. We reviewed the Ripple Carry Adder. We proved its correctness rigorously and used it to define the carry bits associated with addition.

We showed that the problems of computing the sum bits and the carry bits are equivalent modulo a constant-time linear-cost reduction. Since the cost of every adder is $\Omega(n)$ and the delay is $\Omega(\log n)$, we regard the problems of computing the sum bits and the carry bits as equivalently hard.

We presented an adder design called Conditional Sum Adder ($\text{CSA}(n)$). The $\text{CSA}(n)$ design is based on divide-and-conquer. Its delay is asymptotically optimal (if fanout is not taken into account). However, its cost is rather large, approximately $\Theta\left(n^{1.58}\right)$.

We then considered the problem of simultaneously computing the sum and incremented sum of two binary numbers. We presented a design called Compound Adder ($\text{COMP-ADDER}(n)$). This design is also based on divide-and-conquer. The asymptotic delay is also logarithmic, however, the cost is $\Theta(n \cdot \log n)$.

This result is rather surprising: a $\text{COMP-ADDER}(n)$ is much cheaper asymptotically than a $\text{CSA}(n)$! You should make sure that you understand the rational behind this magic. Moreover, by adding a line of multiplexers, one can obtain an $\text{ADDER}(n)$ from a $\text{COMP-ADDER}(n)$. So the design of a $\text{COMP-ADDER}(n)$ is a real improvement over the $\text{CSA}(n)$.

In the next chapter we present an adder design that is asymptotically optimal both with respect to delay and with respect to cost. Moreover, the asymptotic delay and cost of this optimal design is not affected by considering fanout.