Chapter 7

Fast Addition

In this chapter we present an asymptotically optimal adder design. The method we use is called parallel prefix computation. This is a quite general method and has many applications besides fast addition.

7.1 Reduction: sum-bits $\longmapsto$ carry-bits

In this section we review the reduction (presented in Section 6.3) of the task of computing the sum-bits to the task of computing the carry-bits.

The correctness of $\text{RCA}(n)$ implies that, for every $0 \leq i \leq n-1$,

$$S[i] = \text{XOR}_3(A[i], B[i], C[i]).$$

This implies a constant-time linear-cost reduction of the task of computing $S[n-1:0]$ to the task of computing $C[n-1:0]$. Given $C[n-1:0]$ we simply apply a bit-wise XOR of $C[i]$ with $\text{XOR}(A[i], B[i])$. The cost of this reduction is $2n$ times the cost of a XOR-gate and the delay is $2 \cdot d(\text{XOR})$.

We conclude that if we know how to compute $C[n-1:0]$ with $O(n)$ cost and $O(\log n)$ delay, then we also know how to add with $O(n)$ cost and $O(\log n)$ delay.

7.2 Computing the carry-bits

In this section we present a reduction of the problem of computing the carry-bits to a prefix computation problem (we define what a prefix computation problem is in the next section).

Consider the Full-Adder FA_i in an $\text{RCA}(n)$. The functionality of a Full-Adder implies that $C[i+1]$ satisfies:

$$C[i+1] = \begin{cases} 0 & \text{if } A[i] + B[i] + C[i] \leq 1 \\ 1 & \text{if } A[i] + B[i] + C[i] \geq 2. \end{cases} \tag{7.1}$$

The following claim follows directly from Equation 7.1.

Claim 26 *For every $0 \leq i \leq n-1$,*

$$\begin{aligned} A[i]+B[i]=0 &\implies C[i+1]=0 \\ A[i]+B[i]=2 &\implies C[i+1]=1 \\ A[i]+B[i]=1 &\implies C[i+1]=C[i]. \end{aligned}$$

Claim 26 implies that it is easy to compute $C[i+1]$ if $A[i]+B[i] \neq 1$. It is the case $A[i]+B[i]=1$ that creates the effect of a carry rippling across many bit positions.

The following definition is similar to the "kill, propagate, generate" signals that are often described in literature.

Definition 40 *The string $\sigma[n-1:-1] \in \{0,1,2\}^{n+1}$ is defined as follows:*

$$\sigma[i] \triangleq \begin{cases} 2 \cdot C[0] & \textit{if } i=-1 \\ A[i]+B[i] & \textit{if } i \in [0,n-1]. \end{cases}$$

Note that $\sigma[i]=0$ corresponds to the case that the carry is "killed"; $\sigma[i]=1$ corresponds to the case that the carry is "propagated"; and $\sigma[i]=2$ corresponds to the case that the carry is "generated". Instead of using the letters k, p, g we use the letters $0, 1, 2$. (One advantage of our notation is that this spares the need to define addition over the set $\{k, p, g\}$.)

7.2.1 Carry-Lookahead Adders

Carry-Lookahead adders are hierarchical designs in which addends are partitioned into blocks. The number of levels in the hierarchy in this family of adders ranges from two-three levels to $O(\log n)$ levels.

In this section we focus on the computation in a block in the topmost level. We explain the theory behind the design of a block. We then analyze the cost and delay associated with such a block. We prove that (i) the delay is logarithmic in the block size (if fanout is not considered), and (ii) the cost grows cubically as a function of the block size. This is why the common block size is limited to a small constant (i.e. 4 bits).

The correctness of Carry-Lookahead Adders is based on the following claim that characterizes when the carry bit $C[i+1]$ equals 1.

Claim 27 *For every $-1 \leq i \leq n-1$,*

$$C[i+1]=1 \qquad\iff\qquad \exists j \leq i \ : \ \sigma[i:j]=1^{i-j} \cdot 2.$$

Proof: We first prove that $\sigma[i:j]=1^{i-j} \cdot 2 \implies C[i+1]=1$. The proof is by induction on $i-j$. In induction basis, for $i-j=0$ is proved as follows. Since $i=j$, it follows that $\sigma[i]=2$. We consider two cases:

- If $i=-1$, then, by the definition of $\sigma[-1]$, it follows that $C[0]=1$.
- If $i \geq 0$, then $A[i]+B[i]=\sigma[i]=2$. Hence $C[i+1]=1$.

The induction step is proved as follows. Note that $\sigma[i:j] = 1^{i-j} \cdot 2$ implies that $\sigma[i-1:j] = 1^{i-j-1} \cdot 2$. We apply the induction hypothesis to $\sigma[i-1:j]$ and conclude that $C[i] = 1$. Since $\sigma[i] = 1$, we conclude that

$$\underbrace{A[i] + B[i]}_{\sigma[i]=1} + C[i] = 2.$$

Hence, $C[i+1] = 1$, as required.

We now prove that $C[i+1] = 1 \Rightarrow \exists j \leq i : \sigma[i:j] = 1^{i-j} \cdot 2$. The proof is by induction on i. The induction basis, for $i = -1$, is proved as follows. If $C[0] = 1$, then $\sigma[-2] = 2$. We set $j = i$, and satisfy the requirement.
The induction step is proved as follows. Assume $C[i+1] = 1$. Hence,

$$\underbrace{A[i] + B[i]}_{\sigma[i]} + C[i] \geq 2.$$

We consider three cases:

- If $\sigma[i] = 0$, then we obtain a contradiction.
- If $\sigma[i] = 2$, then we set $j = i$.
- If $\sigma[i] = 1$, then $C[i] = 1$.

We conclude that

$$\begin{aligned} C[i] = 1 \quad & \stackrel{\text{Ind. Hyp.}}{\Longrightarrow} \quad \exists j \leq i : \sigma[i-1:j] = 1^{i-j-1} \cdot 2 \\ & \stackrel{\sigma[i]=1}{\Longrightarrow} \quad \exists j \leq i : \sigma[i:j] = 1^{i-j} \cdot 2. \end{aligned}$$

This completes the proof of the claim. □

We wish to show that Claim 27 lays the foundation for Carry-Lookahead Adders. For this purpose, readers might be anxious to see a concrete definition of how $\sigma[i] \in \{0, 1, 2\}$ is represented. There are, of course, many ways to represent elements in $\{0, 1, 2\}$ using bits; all such reasonable methods use $2-3$ bits and incur a constant cost and delay. We describe a few options for representing $\{0, 1, 2\}$.

1. One could, of course, represent $\sigma[i]$ by the pair $(A[i], B[i])$, in which case the incurred cost and delay are zero.

2. Binary representation could be used. In this case $\sigma[i]$ is represented by a pair $x[1:0]$. Since the value 3 is not allowed, we conclude that $x[0]$ signifies whether $\sigma[i] = 1$. Similarly, $x[1]$ signifies whether $\sigma[i] = 2$. The cost and delay in computing the binary representation of $\sigma[i]$ from $A[i]$ and $B[i]$ is the cost and delay of a Half-Adder.

3. 1-out-of-3 representation could be used. In this case, we would need to add a bit to binary representation to signify whether $\sigma[i] = 0$. This adds the cost and delay of a NOR-gate.

Regardless of the representation that one may choose to represent $\sigma[i]$, note that one can compare $\sigma[i]$ with 1 or 2 with constant delay and cost. In fact, in binary representation and 1-out-of-3 representation, such a comparison incurs zero cost and delay.

Computation of $C[i+1]$ in a Carry-Lookahead Adder. Figure 7.1 depicts how the carry-bit $C[i+1]$ is computed in a Carry-Lookahead Adder. For every $-1 \leq j \leq i$, one compares the block $\sigma[i:j]$ with $1^{i-j} \cdot 2$. This comparison is depicted in the left hand side of Fig. 7.1. Each symbol $\sigma[i], \ldots, \sigma[j+1]$ is compared with 1 and the symbol $\sigma[j]$ is compared with 2. The results of these $i-j+1$ comparisons are fed to an AND-tree. The output is denoted by $\sigma[i:j] \stackrel{?}{=} 1^{i-j} \cdot 2$.

The outcomes of the $i+2$ comparisons of blocks of the form $\sigma[i:j]$ with $1^{i-j} \cdot 2$ are fed to an OR-tree. The OR-tree outputs the carry-bit $C[i+1]$.

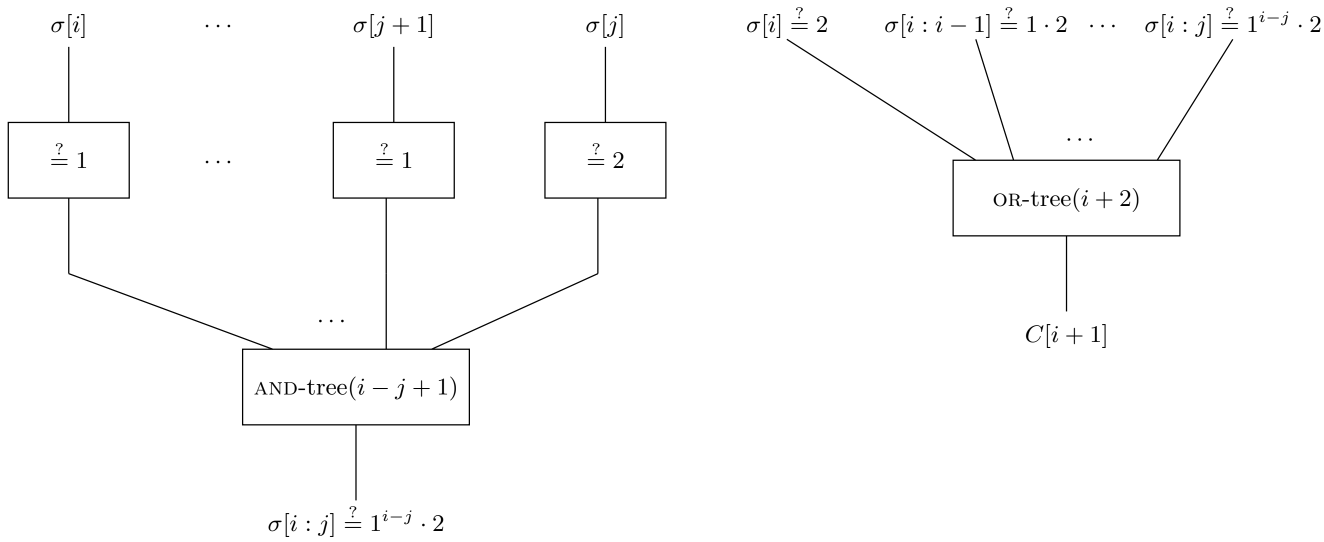

Figure 7.1: Computing the carry-bit $C[i+1]$ in a Carry-Lookahead Adder.

Note that the computation of the carry bits in a Carry-Lookahead Adder is done in this fashion only within a block (hence n here denotes the size of a block).

Cost and delay analysis. The delay associated with computing $C[i+1]$ in this fashion is clearly logarithmic. The cost of computing $C[i+1]$ in the fashion depicted in Fig 7.1 is

$$\begin{aligned}
\sum_{i=1}^{n} c(\text{look-ahead } C[i+1]) &= \sum_{i=1}^{n} \left(\sum_{j=-1}^{i} c(\text{AND-tree}(i-j+1)) + c(\text{OR-tree}(i+2)) \right) \\
&= \sum_{i=1}^{n} \left(\sum_{j=-1}^{i} O(i-j) \right) \\
&= \sum_{i=1}^{n} O(i^2) \\
&= O(n^3).
\end{aligned}$$

We conclude that the cost per block is cubic in the length of the block.

7.2.2 Reduction to prefix computation

Definition 41 *The dyadic operator* $* : \{0,1,2\} \times \{0,1,2\} \longrightarrow \{0,1,2\}$ *is defined by the following table.*

$*$	0	1	2
0	0	0	0
1	0	1	2
2	2	2	2

Claim 28 *For every* $a \in \{0,1,2\}$*:*

$$\begin{aligned} 0 * a &= 0 \\ 1 * a &= a \\ 2 * a &= 2. \end{aligned}$$

We use the notation $a * b$ to denote the result of applying the function $*$ to a and b.

Claim 29 *The function* $*$ *is associative. Namely,*

$$\forall a, b, c \in \{0,1,2\} : (a * b) * c = a * (b * c).$$

Question 38 *(i) Prove claim 29. (ii) Is the function* $*$ *commutative?*

We refer to the outcome of applying the $*$ function by the $*$-product. We also use the notation

$$\prod[i,j] \triangleq \sigma[i] * \cdots * \sigma[j].$$

Associativity of $*$ implies that for every $i \leq j < k$:

$$\prod[i,k] = \prod[i,j] * \prod[j+1,k].$$

The reduction of the computation of the carry-bits to a prefix computation is based on the following claim.

Claim 30 *For every* $-1 \leq i \leq n-1$,

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \prod[i:-1] = 2.$$

Proof: From Claim 27, it suffices to prove that

$$\exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2 \qquad \Longleftrightarrow \qquad \prod[i:-1] = 2.$$

($\Rightarrow$) Assume that $\sigma[i:j] = 1^{i-j} \cdot 2$. It follows that

$$\prod[i:j] = 2.$$

If $j = -1$ we are done. Otherwise, by Claim 28

$$\begin{aligned}\prod[i:-1] &= \underbrace{\prod[i:j]}_{=2} * \prod[j-1:-1] \\ &= 2.\end{aligned}$$

($\Leftarrow$) Assume that $\prod[i:-1] = 2$. If, for every $\ell \leq i$, $\sigma[\ell] \neq 2$, then $\prod[i:-1] \neq 2$, a contradiction. Hence

$$\{\ell \in [-1,i] \ : \ \sigma[\ell] = 2\} \ \neq \ \emptyset.$$

Let

$$j \triangleq \max\{\ell \in [-1,i] \ : \ \sigma[\ell] = 2\}.$$

By Claim 28, $\prod[j:-1] = 2$. We claim that $\sigma[\ell] = 1$, for every $j < \ell \leq i$.

By the definition of j, $\sigma[\ell] \neq 2$, for every $j < \ell \leq i$. If $\sigma[\ell] = 0$, for $j < \ell \leq i$, then $\prod[i:\ell] = 0$, and then $\prod[i:-1] = \prod[i:\ell] * \prod[\ell-1:-1] = 0$, a contradiction.

Since $\sigma[i:j+1] = 1^{i-j}$, we conclude that $\sigma[i:j] = 1^{i-j} \cdot 2$, and the claim follows. $\square$

A prefix computation problem is defined as follows.

Definition 42 *Let Σ denote a finite alphabet. Let* $\text{OP} : \Sigma^2 \longrightarrow \Sigma$ *denote an associative function. A* prefix computation *over Σ with respect to* OP *is defined as follows.*

Input $x[n-1:0] \in \Sigma^n$.

Output: $y[n-1:0] \in \Sigma^n$ *defined recursively as follows:*

$$\begin{aligned} y[0] &\leftarrow x[0] \\ y[i+1] &= \text{OP}(x[i+1], y[i]). \end{aligned}$$

Note that $y[i]$ can be also expressed simply by

$$y_i = \text{OP}_i(x[i], x[i-1], \ldots, x[0]).$$

Claim 30 implies a reduction of the problem of computing the carry-bits $C[n:1]$ to the prefix computation problem of computing the prefixes $\prod[0:-1], \ldots, \prod[n:-1]$.

7.3 Parallel prefix computation

In this section we present a general asymptotically optimal circuit for the prefix computation problem.

As in the previous section, let $\text{OP} : \Sigma^2 \longrightarrow \Sigma$ denote an associative function. We do not address the issue of how values in Σ are represented by binary strings. We do assume that some fixed representation is used. Moreover, we assume the existence of a OP-gate that given representations of $a, b \in \Sigma$ outputs a representation of $\text{OP}(a, b)$.

Definition 43 *A* Parallel Prefix Circuit, $\text{PPC–OP}(n)$, *is a combinational circuit that computes a prefix computation. Namely, given input $x[n-1:0] \in \Sigma^n$, it outputs $y[n-1:0] \in \Sigma^n$, where*

$$y_i = \text{OP}_i(x[i], x[i-1], \ldots, x[0]).$$

Example 13 *A Parallel Prefix Circuit with (i) the alphabet $\Sigma = \{0, 1\}$ and (ii) the function $\text{OP} = \text{OR}$ is exactly the Unary Priority Encoder, $\text{U-PENC}(n)$.*

Example 14 *Consider $\Sigma = \{0, 1, 2\}$ and $\text{OP} = *$. The Parallel Prefix Circuit with (i) the alphabet $\Sigma = \{0, 1, 2\}$ and (ii) the function $\text{OP} = *$ can be used (according to Claim 30) to compute the carry-bits.*

Our goal is to design a $\text{PPC–OP}(n)$ using only OP-gates. The cost of the design is the number of OP-gates used. The delay is the maximum number of OP-gates along a directed path from an input to an output.

Question 39 *Design a $\text{PPC–OP}(n)$ circuit with linear delay and cost.*

Question 40 *Design a $\text{PPC–OP}(n)$ circuit with logarithmic delay and quadratic cost.*

Question 41 *Assume that a design $C(n)$ is a $\text{PPC–OP}(n)$. This means that it is comprised only of OP-gates and works correctly for every alphabet Σ and associative function $\text{OP} : \Sigma^2 \rightarrow \Sigma$. Can you prove a lower bound on its cost and delay?*

7.3.1 Implementation

In this section we present a linear cost logarithmic delay $\text{PPC–OP}(n)$ design. The design is recursive and uses a technique that we name "odd-even".

The design we present is a recursive design. For simplicity, we assume that n is a power of 2. The design for $n = 2$ simply outputs $y[0] \leftarrow x[0]$ and $y[1] \leftarrow \text{OP}(x[0], x[1])$. The recursion step is depicted in Figure 7.2. Adjacent inputs are paired and fed to an OP-gate. Observe that wires carrying the inputs with even indexes are sent over the $\text{PPC–OP}(n/2)$ so that the even indexed outputs can be computed. The $n/2$ outputs of the OP-gates are fed to a $\text{PPC–OP}(n/2)$. The outputs of the $\text{PPC–OP}(n/2)$ circuit are directly connected to the odd indexed outputs $y[1], y[3], \ldots, y[n-1]$. The even indexed outputs are obtained as follows: $y[2i] \leftarrow \text{OP}(x[2i], y[2i-1])$.

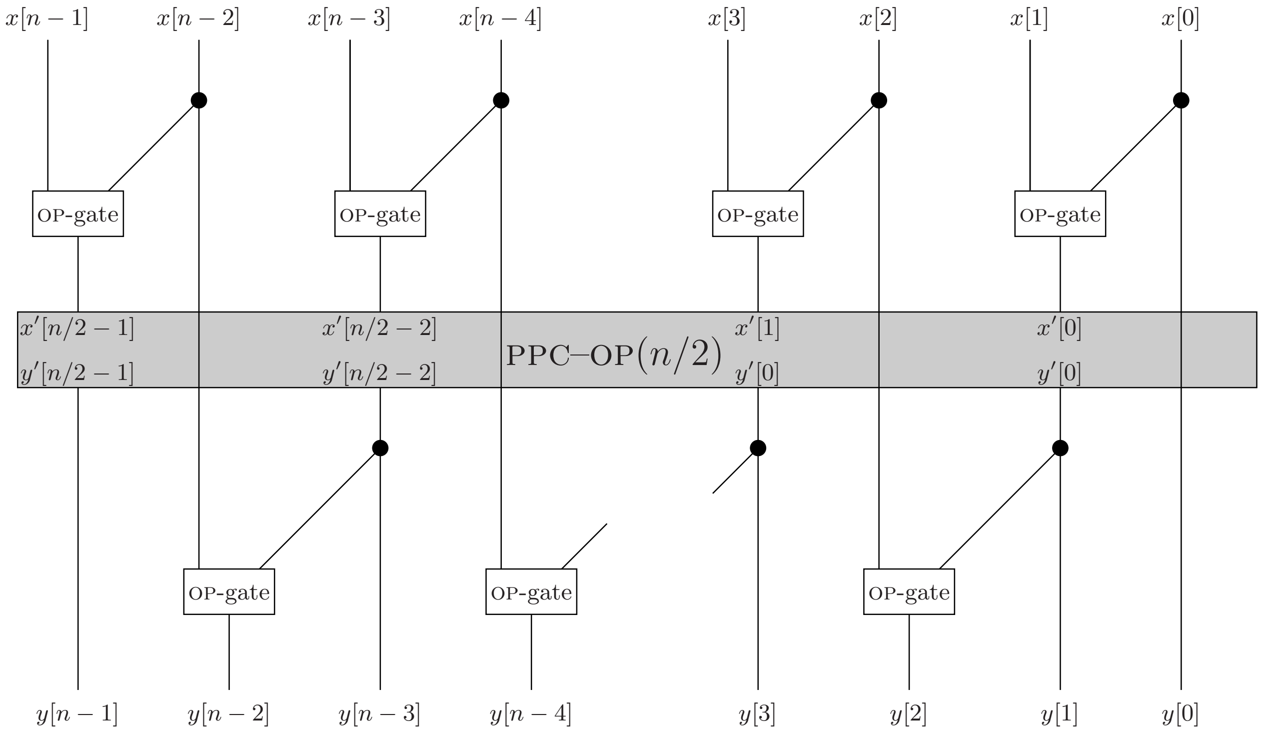

Figure 7.2: A recursive design of PPC–OP(n).

7.3.2 Correctness

Claim 31 *The design depicted in Fig. 7.2 is correct.*

Proof: The proof of the claim is by induction. The induction basis holds trivially for $n = 2$. We now prove the induction step. Consider the PPC–OP$(n/2)$ used in a PPC–OP(n). Let $x'[n/2-1:0]$ and $y'[n/2-1:0]$ denote the inputs and outputs of the PPC–OP$(n/2)$, respectively. The ith input $x'[i]$ equals $\text{OP}(x[2i+1], x[2i])$. By the induction hypothesis, the ith output $y'[i]$ satisfies:

$$\begin{aligned} y'[i] &= \text{OP}_i(x'[i], \ldots, x'[0]) \\ &= \text{OP}_{2i}(x[2i+1], \ldots, x[0]). \end{aligned}$$

Since $y[2i+1]$ equals $y'[i]$, it follows that the odd indexed outputs $y[1], y[3], \ldots, y[n-1]$ are correct. Finally, $y[2i]$ equals $\text{OP}(x[2i], y'[i])$, and hence $y[2i] = \text{OP}(x[2i], y[2i-1])$. It follows that the even indexed outputs are also correct, and the claim follows. □

7.3.3 Delay and cost analysis

The delay of the PPC–OP(n) circuit satisfies the following recurrence:

$$d(\text{PPC–OP}(n)) = \begin{cases} d(\text{OP-gate}) & \text{if } n = 2 \\ d(\text{PPC–OP}(n/2)) + 2 \cdot d(\text{OP-gate}) & \text{otherwise.} \end{cases}$$

If follows that

$$d(\textsc{ppc–op}(n)) = (2\log n - 1) \cdot d(\textsc{op}\text{-gate}).$$

The cost of the $\textsc{ppc–op}(n)$ circuit satisfies the following recurrence:

$$c(\textsc{ppc–op}(n)) = \begin{cases} c(\textsc{op}\text{-gate}) & \text{if } n = 2 \\ c(\textsc{ppc–op}(n/2)) + (n-1) \cdot c(\textsc{op}\text{-gate}) & \text{otherwise.} \end{cases}$$

Let $n = 2^k$, it follows that

$$\begin{aligned} c(\textsc{ppc–op}(n)) &= \sum_{i=2}^{k} (2^i - 1) \cdot c(\textsc{op}\text{-gate}) + c(\textsc{op}\text{-gate}) \\ &= (2n - 4 - (k-1) + 1) \cdot c(\textsc{op}\text{-gate}) \\ &= (2n - \log n - 2) \cdot c(\textsc{op}\text{-gate}). \end{aligned}$$

Corollary 32 *If the delay and cost of an* OP*-gate is constant, then*

$$\begin{aligned} d(\textsc{ppc–op}(n)) &= \Theta(\log n) \\ c(\textsc{ppc–op}(n)) &= \Theta(n). \end{aligned}$$

Corollary 32 implies that $\textsc{ppc–op}(n)$ with $\Sigma = \{0, 1\}$ and $\textsc{op} = \textsc{or}$ is an asymptotically optimal $\textsc{u-penc}(n)$. It also implies that we can compute the carry-bit corresponding to an addition in linear cost and logarithmic delay.

Remark 3 *This analysis ignores fanout effects. Note, however, that if we insert a buffer in every branching point of the* $\textsc{ppc–op}(n)$ *design (such branching points are depicted by filled circles), then the fanout is constant. Such an insertion only affects the constants in the analysis of the cost and delay.*

Question 42 *What is the maximum fanout in the* $\textsc{ppc–op}(n)$ *design. Analyze the effect of inserting buffers to the cost and delay of* $\textsc{ppc–op}(n)$*.*

7.4 Putting it all together

In this section we assemble an asymptotically optimal adder. The stages of the construction are as follows.

Compute $\sigma[n-1 : -1]$: In this step the symbols $\sigma[i] \in \{0, 1, 2\}$ are computed. This step can be regarded as an encoding step; the sum $A[i] + B[i]$ is encoded to represent the corresponding value in $\{0, 1, 2\}$. The cost and delay of this step depend on the representation used to represent values in $\{0, 1, 2\}$. In any case, the cost and delay is constant per $\sigma[i]$, hence, the total cost is $O(n)$ and the total delay is $O(1)$.

PPC$-*(n)$: In this step the products $\prod[i:-1]$ are computed from $\sigma[i:-1]$, for every $i \in [n-1:0]$. The cost and delay of this step are $O(n)$ and $O(\log n)$, respectively.

Extraction of $C[n:1]$: By Claim 30, it follows that $C[i+1] = 1$ iff $\prod[i:-1] = 2$. In this stage we compare each product $\prod[i:-1]$ with 2. The result of this comparison equals $C[i+1]$. The cost and delay of this step is constant per carry-bit $C[i+1]$. It follows that the cost of this step is $O(n)$ and the delay is $O(1)$.

Computation of sum-bits: The sum bits are computed by applying

$$S[i] = \text{XOR}_3(A[i], B[i], C[i]).$$

The cost and delay of this step is constant per sum-bit. It follows that the cost of this step is $O(n)$ and the delay is $O(1)$.

By combining the cost and delay of each stage we obtain the following result.

Theorem 33 *The adder based on parallel prefix computation is asymptotically optimal; its cost is linear and its delay is logarithmic.*

Remark 4 *There are representations of* $\{0, 1, 2\}$ *in which* XOR$(A[i], B[i])$ *is computed in the stage in which* $\sigma[i]$ *is computed. In this case, we can use this value again in the last stage when the sum-bits are computed.*

7.5 Summary

In this section we presented an adder with asymptotically optimal cost and delay. The adder is based on a reduction of the task of computing the sum-bits to the task of computing the carry bits. We then reduce the task of computing the sum bits to a prefix computation problem.

A prefix computation problem is the problem of computing $\text{OP}_i(x[i-1:0])$, for $0 \leq i \leq n-1$, where OP is an associative operation. We present a linear cost logarithmic delay circuit for the prefix computation problem. We refer to this circuits as PPC–OP(n). Using a PPC–OP(n) we were also able to design an optimal Unary Priority Encoder.

In Section 7.2.1 we describe the top level of Carry-Lookahead Adders. It is possible to design asymptotically optimal adders based on Carry-Lookahead Adders, however, the design is less systematic and is less general.