Chapter 3.

Parallel Prefix

"What's one and one and one and one and one and one and one and one and one and one?" "I don't know," said Alice, "I lost count." "She can't do Addition," the Red Queen interrupted.

—Lewis Carroll, *Through the Looking Glass*

We present asynchronous circuits to solve the prefix problem with $O(N \log N)$ circuit size, $O(\log N)$ worst-case latency, and $O(1)$ cycle time. If the prefix operation has a right zero, the asynchronous solution has an average-case latency of $O(\log\log N)$. The construction can be used to obtain an $O(1)$ cycle time asynchronous adder with $O(N \log N)$ circuit size and $O(\log\log N)$ average-case latency. We prove that our circuits have optimal asymptotic average-case latency.

Let $\otimes$ be an associative operation. The *prefix problem* is to compute, given $x_1, x_2, \ldots, x_N$, the results $y_1, y_2, \ldots, y_N$, where $y_k = x_1 \otimes x_2 \otimes \cdots \otimes x_k$, for $1 \leq k \leq N$.[10]

We construct asynchronous solutions to the prefix problem that are similar to their synchronous counterparts. We improve the average-case performance of the asynchronous solution by using two competing methods for solving the prefix problem and picking the one that arrives earliest to produce the output. This technique reduces the average-case latency from $O(\log N)$ to $O(\log\log N)$ when the prefix operator has a right zero, a significant improvement. We show that our solutions have optimal asymptotic average-case latency.

A number of problems can be formulated as a prefix problem. Ladner and Fisher show how the prefix problem can be used to parallelize the computation of an arbitrary Mealy machine.[10] Leighton discusses a number of different problems that can be solved using prefix computations.[11] As a concrete application, we use the construction to obtain an asynchronous adder which has $O(1)$ cycle time, $O(N \log N)$ circuit size, $O(\log N)$ worst-case latency, and $O(\log \log N)$ average-case latency.

3.1. Traditional Solution

To formulate the prefix problem in terms of an asynchronous CHP program, we assume that the inputs $x_1, x_2, \ldots, x_N$ arrive on input channels $X_1, X_2, \ldots, X_N$ respectively, and that the outputs $y_1, y_2, \ldots, y_N$ are to be produced on output channels $Y_1, Y_2, \ldots, Y_N$ respectively. The problem can be restated in terms of reading the values x_i from the input channels, computing the y_i values, and sending these values on the appropriate output channels. In terms of CHP, the immediate solution that leaps to mind is the following program:

```
*[ $X_1?x_1$,  $X_2?x_2$,  $\ldots$,  $X_N?x_N$;
     $Y_1!x_1$,  $Y_2!(x_1 \otimes x_2)$,  $\ldots$,  $Y_N!(x_1 \otimes x_2 \otimes \cdots \otimes x_N)$
 ]
```

This program is very inefficient for a number of reasons, the most obvious being that there are $O(N^2)$ $\otimes$-operations, which correspond to $O(N^2)$ circuit elements; but it will serve as a specification for the problem.

For the purposes of this chapter, we will assume that the operation $\otimes$ has an identity e. This is merely an aid to clarity—it does not detract from the construction in any way.

Since we know input value x_i at position i, we can solve the prefix problem if we can determine $x_1 \otimes \cdots \otimes x_{i-1}$ at position i. Assume we had a method that computed the prefixes we needed for a problem of size n. We will extend it to compute the prefixes we need of size $2n$ as follows. We begin by using $x_{2i-1} \otimes x_{2i}$ as the input to the n-input prefix computation graph. The result of this operation would be to compute values $x_1 \otimes \cdots \otimes x_{2i}$ at output position $i+1$. We can now solve the prefix problem of size $2n$ by producing $x_1 \otimes \cdots \otimes x_{2i}$, and $x_1 \otimes \cdots \otimes x_{2i+1}$. The program to do this is described by

```
$UP(L, R, U, V, Ld, Rd) \equiv$
     *[ $L?x, R?y$;  $U!(x \otimes y)$;  $V?p$;  $Ld!p, Rd!(p \otimes x)$ ]
```

where the channels U and V correspond to the input and output stages of the prefix computation graph of half the size. From the structure of the solution, it is clear that the computation graph is a tree. Repeating this observation, all that remains is to provide a solution to the prefix problem of size 2—the root of the tree, and to read the inputs and produce the final outputs.

The V channel at the root of the tree requires the empty prefix—the identity e. The output U of the root is not used by any other process. Thus, we simplify the root process to:

$$\begin{aligned}&ROOT(L,R,Ld,Rd)\equiv\\&\qquad *[\ L?x, R?y;\ \ Ld!e, Rd!x\ ]\end{aligned}$$

where e is the identity of $\otimes$. The leaves of the prefix computation tree read the inputs, their prefix (from the tree), and produce the appropriate output. A leaf process is written as:

$$\begin{aligned}&LEAF(X,U,V,Y)\equiv\\&\qquad *[\ X?x; U!x;\ \ V?y; Y!(y\otimes x)\ ]\end{aligned}$$

Part of the computation graph for the prefix problem when $N = 4$ is shown in Figure 3.1.

Observe that the sequencing between $U!(x\otimes y)$ and $V?p$ is enforced by the environment of the UP process. We can therefore split the process into two parts that execute in parallel. However, the obvious split would cause variable x to be shared between the two processes. We introduce a local channel C which is used to copy the value of x. The new UP process is:

$$\begin{aligned}&UP(L,R,U,V,Ld,Rd)\equiv\\&\qquad *[\ L?x, R?y;\ \ U!(x\otimes y),\ \ C!x\ ]\ \|\ *[\ C?c, V?p;\ \ Ld!p, Rd!(p\otimes c)\ ]\end{aligned}$$

These two processes are identical! Therefore, we write:

$$\begin{aligned}&UP2(A,B,C,D)\equiv\\&\qquad *[\ A?x, B?y;\ \ C!(x\otimes y), D!x\ ]\end{aligned}$$

$$UP(L,R,U,V,Ld,Rd)\equiv UP2(L,R,U,C)\ \|\ UP2(V,C,Rd,Ld)$$

Similarly, we can rewrite the $LEAF$ process as:

$$\begin{aligned}&LEAF(X,U,V,Y)\equiv\\&\qquad *[\ X?x; U!x, C!x]\ \|\ *[\ C?c, V?y; Y!(y\otimes c)\ ]\end{aligned}$$

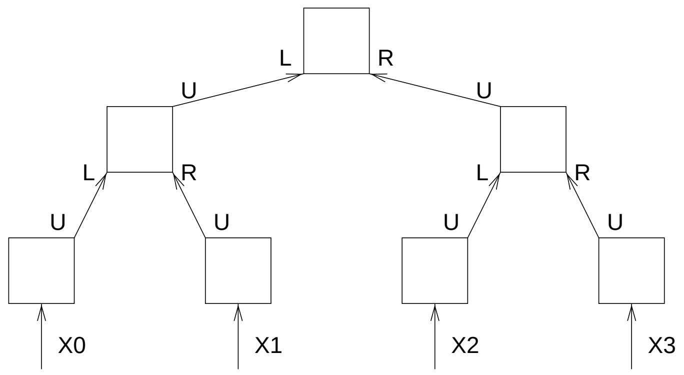

Figure 3.1. Solution to the prefix problem.

Since each node in the tree contains a constant number of $\otimes$ computations and there are $O(N)$ bounded fan-in nodes in the tree, there are $O(N)$ $\otimes$-computation circuits in the solution. Since the tree is of depth $O(\log N)$, the latency and cycle time of this solution is $O(\log N)$.

3.2. Pipelining

The solution presented above has a cycle time of $\Theta(\log N)$ since the prefix computation tree can only perform one prefix computation at a time. We can pipeline the computation to permit the tree to operate simultaneously on multiple inputs and reduce the cycle time to $O(1)$.

Consider a single *UP* node in the prefix computation tree. There are no pipeline stages between the two halves of process *UP*, since they communicate through a slack-zero channel C. However, the second process that is part of *UP* cannot complete its computation until it receives a value on channel V. This value is computed by a circuit which has a number of pipeline stages proportional to the depth of *UP* in the tree. Therefore, even though there are $O(\log N)$ pipeline stages on the computation for V, we cannot have $O(\log N)$ computations being performed by the tree since channel C has zero slack. Therefore, we introduce buffering on C proportional to the depth of the node in the tree. Logically, it is simpler to visualize the computation by "unfolding" the tree into two parts—the up-going phase, and down-going phase—as shown in Figure 3.2. The vertical arrows are the internal channels C, and two boxes connected by vertical arrows correspond to a single node in the tree.

It is clear that one must add $2d - 1$ stages of buffering on the internal channel C for a node that is d steps away from the root for the circuit to be pipelined in

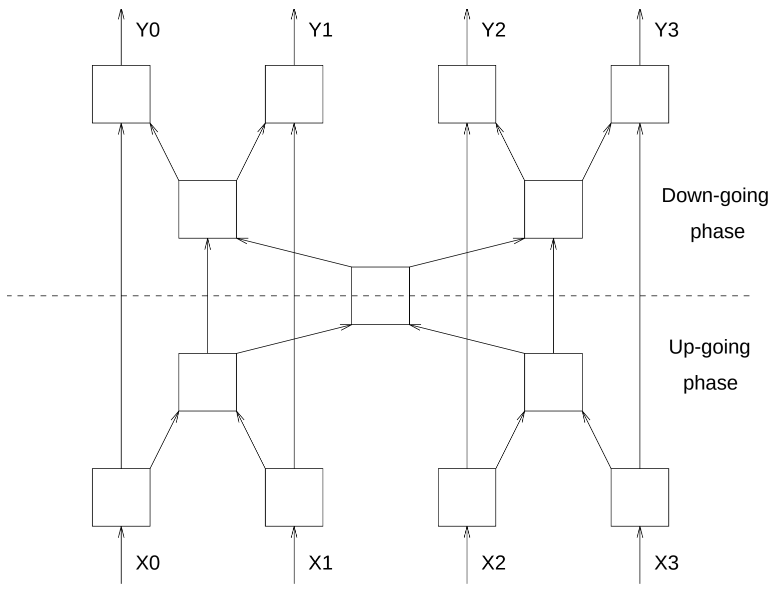

Figure 3.2. Unpipelined prefix computation.

a manner that permits $2 \lg N + 1$ prefix operations to be performed simultaneously. Figure 3.3 shows the tree after the appropriate buffers have been introduced.

The cycle time of the pipelined prefix computation with buffers does not depend on the number of inputs, but on the time it takes to perform the $\otimes$ operation. The latency of the computation block is proportional to the number of stages, and is therefore $2 \lg N + 1$ stages both with and without the buffers. However, we have increased the circuit size from $O(N)$ to $O(N \log N)$ since we have introduced $O(N \log N)$ buffers.

3.3. Reducing the Average-Case Latency

If the prefix computation is not used very often, the observed performance depends on the latency of the prefix computation—a quantity that is not reduced by adding buffers to the computation tree. In this section, we present a technique that reduces the average-case latency of the prefix computation in certain cases. We begin by considering a simple solution to the prefix problem.

The simplest way to perform the prefix computation is in a sequential fashion. Since we have n different input channels, we use n processes, one for each input channel, connected in a linear fashion as shown in Figure 3.4.

The stage for x_k receives y_{k-1} on channel L from the previous stage and x_k on

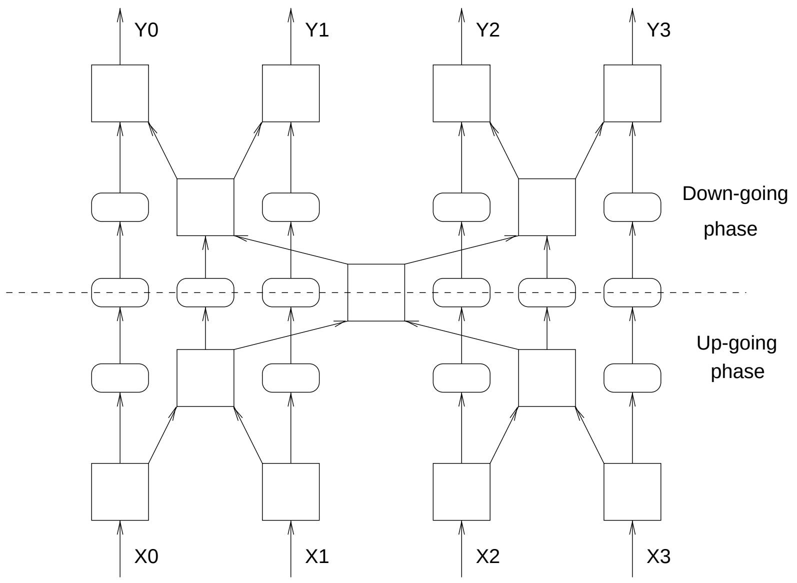

Figure 3.3. Pipelined prefix computation with buffers.

channel X_k and produces y_k on channel Y_k as well as channel R which connects it to the next stage. The CHP for an intermediate stage of such a solution is given by:

```
$SERIAL(X, Y, L, R) \equiv$
        *[ $X?x, L?p$;  $Y!(p \otimes x), R!(p \otimes x)$ ]
```

However, we know that the input on channel X arrives much sooner than the input on channel L. Given this information, is it possible to produce the outputs on Y and R before receiving the input on L?

Suppose we know that a is a right zero of the prefix operation, i.e., $x \otimes a = a$ for all values of x. Now, if the input on channel X is equal to a, we can produce the output on Y and R *before* reading the value on L. We rewrite *SERIAL* as:

```
$SERIAL(X, Y, L, R) \equiv$
        *[ $X?x$;  [ $x = a \longrightarrow Y!a, R!a, L?p$
                   [] $x \neq a \longrightarrow L?p$;  $Y!(p \otimes x), R!(p \otimes x)$
                   ]
         ]
```

The time taken for this solution to produce the output is *data-dependent*. In the best case (when all inputs are a), the time from receiving the inputs to producing the output is constant—much better than the prefix computation tree, and in the worst

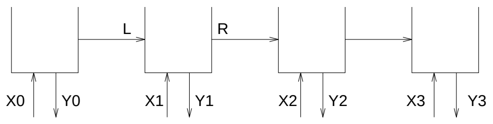

Figure 3.4. Serial prefix computation.

case, the time taken is $O(N)$—much worse than the prefix computation tree which only takes $O(\log N)$ time.

The solution we adopt is to *combine* both the prefix computation tree and the serial computation into a single computation. The two computations compete (in time) against one another, and we can pick the solution that arrives first. This technique has a worst-case latency of $O(\log N)$, but a best-case latency of $O(1)$.

We begin with the unpipelined prefix computation corresponding to Figure 3.2. The CHP for the *LEAF* process used by the prefix computation tree is:

```
$LEAF(X, U, V, Y) \equiv$
        *[ $X?x; U!x, C!x$] $\|$ *[ $C?c, V?y; Y!(y \otimes c)$ ]
```

Observe that the value received along channel V for a leaf which receives x_k as input is the same as the value received along channel L by the corresponding process in the serial computation shown in Figure 3.4.

We introduce channels L and R from the serial computation into the prefix computation tree. The output Y from the leaf process is simply copied on outgoing channel R. Since the values received on L and on the corresponding V are the same, we combine these two channels externally using a merge process that picks the first input that arrives, as follows:

```
$MERGE(L, V, M) \equiv$
        *[[$\overline{L} \longrightarrow L?y; M!y, V?yy$
          |$\overline{V} \longrightarrow V?y; M!y, L?yy$
         ]]
```

The new *LEAF* process is:

```
$LEAF(X, U, V, L, R, Y) \equiv$
        *[ $X?x; U!x, C!x$] $\|$ $MERGE(L, V, M)$ $\|$ $SERIAL(C, Y, M, R)$
```

The compilation of *SERIAL* depends on the structure of $\otimes$. The compilation of the *MERGE* procedure that picks the first input is given below:

```
$*[ [\neg Ma]; [v(L) \vee v(V)]; M \Uparrow; ([v(L)]; La\uparrow), ([v(V)]; Va\uparrow);$
   $[Ma]; M \Downarrow; ([n(L)]; La\downarrow), ([n(V)], Va\downarrow)$
 $]$
```

This circuit has an efficient implementation because we know that the value being received on both L and V will be the same.

Finally, using a similar transformation, we can replace process *UP* in the prefix computation tree by one that also has a serial computation phase. The original *UP* process was:

```
$UP(L, R, U, V, Ld, Rd) \equiv$
        $*[\ L?x, R?y;\ \ U!(x \otimes y),\ \ C!x\ ] \ \|\ *[\ C?c, V?p;\ \ Ld!p, Rd!(p \otimes c)\ ]$
```

The value to be sent along the “right” channel for the serial computation, namely *SR*, is given by $p \otimes x \otimes y$. We therefore introduce an additional internal channel C', along which the value $x \otimes y$ is sent. Finally, the “left” channel for the serial computation, namely *SL*, is merged with V using the same *MERGE* process shown above. We obtain:

```
$UP(SL, SR, L, R, U, V, Ld, Rd) \equiv$
          $*[\ L?x, R?y;\ \ U!(x \otimes y),\ \ C'!(x \otimes y),\ \ C!x\ ]$
        $\|\ MERGE(SL, V, M)$
        $\|\ *[\ C?c, M?p;\ \ M1!p, Ld!p, Rd!(p \otimes c)\ ]$
        $\|\ *[\ C'?d;\ \ [d = a \longrightarrow SR!a, M1?p\ \ [\!]\ d \neq a \longrightarrow M1?p; SR!(p \otimes d)]\ ]$
```

Since this solution follows from the unpipelined version of the prefix computation, its cycle time is $O(\log N)$. To improve its cycle time this time, we need to add buffering to both channels C and C'. This transformation will once again increase the circuit size from $O(N)$ to $O(N \log N)$. For reasons to be discussed in the following section, we use binary tree buffers to implement the buffering on channels C and C' instead of linear buffers.

3.4. Analysis of the Average Case

The latency of the prefix computation is data-dependent. We therefore need some information about the input distribution to determine the average-case latency. Consider process *SERIAL* shown below that is part of the prefix computation.

```
$SERIAL(X, Y, L, R) \equiv$
        *[ $X?x$;  [ $x = a \longrightarrow Y!a, R!a, L?c$
                   [] $x \neq a \longrightarrow L?c;\ \ Y!(c{\otimes}x), R!(c{\otimes}x)$
                   ]
         ]
```

When $x \neq a$, the output on Y and R depends on the input c. We call this the *propagate* case, since the output of the process depends on the input c. Let the probability of a particular input being a be p, and let this distribution be independent across all the n inputs. If the inputs remain independently distributed, the analysis below is applicable even if the probability of the input being a at input position i might vary (as long as it remains non-zero), since we can pick p to be the smallest value as a conservative approximation.

Theorem 3.1.
If the inputs of the prefix computation are independently distributed with non-zero probability of an input being a right zero, the average-case latency of the modified asynchronous prefix computation is $O(\log\log N)$, where N is the input size.

Proof: Let $L(N)$ be the latency through a prefix computation with N inputs. We assume that the prefix computation uses a k-ary tree for the purpose of this analysis. We can write:

$$L(N) = \min\left(ms, L\left(\frac{N}{k}\right) + h\right)$$

where m is the length of the longest sequence of "propagate" inputs, s is the delay through a single stage of the serial "propagate" chain at the leaves of the tree, and h is the delay through one stage of the tree. The first part of the formula comes from the serial computation, and the latter from the tree computation. To expand $L(\frac{N}{k})$, observe that at the next stage in the tree, m will be replaced by m/k since we are considering the *same* input. Applying this expansion recursively, we obtain:

$$L(N) = \min_{m \geq k^i}\left(\frac{ms}{k^i} + ih\right)$$

In particular, choosing $m = k^i$ we obtain:

$$L(N) \leq s + \frac{h}{\log k}\log m$$

The average latency is bounded above by:

$$\mathrm{E}[L(N)] \leq s + \frac{h}{\log k}\mathrm{E}[\log m]$$

To compute the expected value of $\log m$, observe that

$$\mathrm{E}[\log m] \leq \log \mathrm{E}[m]$$

since the expected value of the logarithm of a random variable is the logarithm of the geometric mean of the variable. Since the arithmetic mean is always at least the geometric mean and log is increasing (m is always non-negative), the above inequality follows. We can bound $\mathrm{E}[L(N)]$ from above if we determine $\mathrm{E}[m]$.

When $p = 1/2$, we know that $\mathrm{E}[m] \leq \log_2 N$.[1] A simple extension of the proof shows that

$$\mathrm{E}[m] \leq \lceil \log_{1/(1-p)} N \rceil + \frac{1}{pN} = O(\log N)$$

when $0 < p < 1$ (a complete proof is given in Appendix 2). Therefore, the average latency through the prefix computation is bounded above by:

$$\begin{aligned}\mathrm{E}[L(N)] &\leq s + \frac{h}{\log k}\log\left(\lceil \log_{1/(1-p)} N \rceil + \frac{1}{pN}\right)\\ &= O(\log\log N)\end{aligned}$$

concluding the proof. □

When the prefix computation operates with $O(1)$ cycle time, the value of s given above is a function of N. Since we add $2d-1$ stages of buffering at depth d in the tree for the serial computation part as well, the value of s is bounded above by a function that depends on the latency of a buffer of size $O(\log N)$. Since we have used a binary tree buffer to implement the slack on the internal channels, the latency of a buffer of size $O(\log N)$ is $O(\log\log N)$. Therefore, the additional buffering required to reduce the cycle time of the circuit does not increase the order of the average-case latency.

3.5. Reducing the Area Overhead

The $O(\log\log N)$ average-case latency adder has $O(N\log N)$ additional circuit size because of the additional buffering required. In this section we show how the area overhead of the prefix computation circuit can be reduced by using the fact that the input distribution is independent.

On examination of the analysis for average-case latency, we make the following observation. The way we achieve an average-case latency of $O(\log\log N)$ is as follows. We traverse up the tree computation $O(\log\log N)$ steps. At this point, the average propagate-chain length is $O(1)$, and we use the serial part of the computation. In another $O(\log\log N)$ steps, we propagate the results down the tree. This permits us to complete the prefix computation with a latency of $O(\log\log N)$ steps. Therefore, we should be able to achieve the same average-case latency with lower area overhead by using the serial part of the computation only at one stage of the prefix computation tree.

Assume that we add the serial phase of the computation at only one level of the prefix computation that is d steps away from the leaves of the tree. The latency is given by:

$$L(N) = \min\left(d\cdot h + ms/k^d, h\log_k N\right)$$

On average, the latency would be:

$$\begin{aligned}\mathrm{E}[L(N)] &\leq \min\left(d\cdot h + s/k^d\cdot \mathrm{E}[m], h\log_k N\right)\\ &= \min\left(d\cdot h + s/k^d\cdot \log_{1/(1-p)} N, h\log_k N\right)\end{aligned}$$

We attempt to determine the minimum value of this function by differentiating the first part of the minimum expression with respect to d. We obtain:

$$\begin{aligned}d_{min} &= \log_k\left(s/h\cdot \ln k \log_{1/(1-p)} N\right)\\ &= O(\log\log N)\end{aligned}$$

When we add a serial phase to this stage of the prefix computation tree, the average-case latency is given by:

$$\begin{aligned}\mathrm{E}[L_{min}(N)] &\leq \frac{h}{\ln k} + \log_k\left(s/h\cdot \ln k\log_{1/(1-p)} N\right)\\ &= O(\log\log N)\end{aligned}$$

Since we added a serial phase $O(\log\log N)$ steps away from the leaves, the additional area required to permit the computation to run at full throughput is $O(N)$ since we have $O(N/\log N)$ nodes, with $O(\log(N/\log N))$ buffering required for each of them.

If we are willing to sacrifice throughput, we can reduce the area overhead even further and still have $O(\log\log N)$ average-case latency. Observe that we no longer need the tree computation beyond d_{min}. If we simply eliminate the tree after that depth, we still have the same average-case latency! However, we have increased the worst-case latency to $O(N/\log N)$, which may or may not be acceptable in practice. However, we have a significant savings in area—we save an additional $O(N)$ in circuit size, compensating for the $O(N)$ area overhead for adding the serial phase of the computation at depth d_{min}. The actual area necessary will depend on the exact circuit implementation used in either case.

3.6. Application to Binary Addition

The prefix computation can be used to construct a binary *kpg*-adder.[10] To perform binary addition at bit position i, the carry-in for that bit-position must be known. The carry-in computation can be formulated as a prefix computation as follows.

Suppose bit i of the two inputs are both zero. Then no matter what the carry-in is, the carry-out of the stage is zero—a *kill* (k). Similarly, if the two inputs are both one, the carry-out is always one—a *generate* (g). Otherwise, the stage *propagates* (p) the carry-in. To determine the carry-out of two adjacent stages, one can use the following $\otimes$ operation. The vertical column represents the *kpg* code for the least significant bit.

$\otimes$	k	p	g
k	k	k	g
p	k	p	g
g	k	g	g

Table 4.1. Prefix operator for kpg addition

Observe that the *kpg* code has the property that both k and g are right zeros of the prefix operator. Therefore, we can use the techniques discussed above to reduce the latency of binary addition. From the previous section, we observe that the average latency through such an adder is $O(\log\log N)$.

3.7. Area Optimality

We have designed a parallel prefix computation block which has $O(1)$ cycle time and $O(\log N)$ worst-case latency. The circuit has $O(N \log N)$ size. In this section, we show that we cannot do any better in the general case.

We assume that the prefix computation circuit has the following properties:

1. It can be used repeatedly;
2. It does not store information about its history, i.e., it cannot use information from any previous input to compute its next output;
3. Output y_k cannot be generated without knowledge of x_k.

Under these assumptions, we conclude:

Theorem 3.2.
Let $\mathcal{C}(N)$ be a family of circuits that solve an N-input prefix problem, with $r(N)$ being the worst-case ratio of their latency and cycle time over all possible input values. Then the size of the circuits, $S(N)$, is $\theta(N \max(1, r(N)))$.

Proof: The circuit cannot have less than $\theta(N)$ size since it has N inputs and N outputs, and must store at least one bit per output.

Consider a consecutive sequence of inputs all of which have the worst-case ratio of latency to cycle time. Let the latency for the input be l, and the cycle time be τ. If the cycle time is τ, then after τ seconds, the circuit must be able to accept its next input. Since the latency is l, the circuit must have $\frac{l}{\tau}$ pending prefix computations internally. Since each prefix computation requires $\theta(N)$ size to store information for N different outputs, we conclude that the circuit must have $\theta(N\frac{l}{\tau})$ size. □

Corollary 3.3.
A full-throughput N-input parallel prefix computation circuit has $\theta(N \log N)$ size.

Proof: The worst-case latency of any parallel prefix computation circuit is $\theta(\log N)$. Since the cycle time is constant, $r(N) = \theta(\log N)$, concluding the proof. □

From these two observations, we conclude that all the circuits we presented to solve the prefix problem have asymptotically optimal circuit size. The unpipelined circuits have $O(N)$ circuit size, and the full-throughput circuits have $O(N \log N)$ circuit size.

3.8. Latency Optimality

Let V be the set of values that x_i might take. To analyze the delay through a prefix computation circuit, we partition V into two parts: a subset consisting of *propagate*-type values, and one consisting of non-propagate values. The set P of propagate-type values is the maximal set characterized by the property that $|V \otimes p| > 1$ for all $p \in P$, i.e., $x \otimes p$ depends on the value x, where $V \otimes x$ is the set $\{s \otimes x \mid s \in V\}$. In the case of a binary adder, the input p is the only input of propagate type (see Table 4.1).

Given an input vector $\mathbf{x} = (x_1, \ldots, x_n)$, a propagate sequence is a subvector $(x_i, x_{i+1}, \ldots, x_j)$ such that $x_i \otimes x_{i+1} \otimes \cdots \otimes x_j \in P$. We define $m(\mathbf{x})$ to be the length of the longest propagate sequence in $\mathbf{x}$. For example, $m(k, k, p, g, p, p) = 2$ since there are two consecutive p values in the vector.

Theorem 3.4.
The average-case latency through any prefix computation circuit is $\theta(\mathrm{E}[\log m(\mathbf{x})])$, *where* m *is defined as above.*

Proof: Given an input vector, let the longest propagate sequence in it be at positions i through j. This implies that the outputs at positions i through j must depend on the input at position i. Therefore, the information content in the input at position i must be communicated to $j - i + 1 = m(\mathbf{x})$ different output positions. This information cannot propagate faster than $\log m(\mathbf{x})$, concluding the proof. □

By Theorem 3.4, the prefix computation circuit we have designed has asymptotically optimal average-case latency. Note that the result does not depend on the input distribution.

Consider the case of binary addition. The argument used in the proof of Theorem 3.4 was based on an analysis of the input to output dependencies; this analysis holds no matter how the binary adder is constructed, and therefore the result also applies to binary addition. In particular, this implies that an adder constructed in this manner has the best possible asymptotic average-case latency characteristics for any input distribution.

If the set P is closed under $\otimes$ then inputs from P will result in long propagate sequences, slowing down the prefix computation. Note that $p, q \in P$ implies $p \otimes q \in P$ is quite a natural property for a prefix operator to have since it is associative. Suppose $x \otimes a$ depends on x and $x \otimes b$ depends on x. Then $x \otimes (a \otimes b) = (x \otimes a) \otimes b$. Since

$x \otimes a$ depends on x, it is natural to expect that $(x \otimes a) \otimes b$ would depend on x. The example in Table 4.2 shows that this is not true in general.

$\otimes$	0	1	2
0	0	0	0
1	0	0	1
2	0	1	2

Table 4.2. P may not be closed under $\otimes$

In this case, the elements 1 and 2 are contained in the set P for this operator. However, $1 \otimes 1 = 0 \notin P$. Since the set $\{2\}$ is closed under $\otimes$, the input can have long sequences of 2's in it, which will slow down the prefix computation. We formalize this observation below.

Let Q_1, ..., Q_q be maximal subsets of P that are closed under $\otimes$. Intuitively, the members of these Q-sets make the prefix computation slow since long sequences of values from a fixed set Q_i will result in large values of $m(\cdot)$. The operator in Table 4.2 has only one such Q-set, namely $\{2\}$. The fact that we may have more than one maximal Q-set is illustrated by Table 4.3, whose prefix operator has two such sets $\{1\}$ and $\{2\}$.

$\otimes$	0	1	2
0	0	0	0
1	0	1	0
2	0	0	2

Table 4.3. $\otimes$ may have more than one Q-set

By the definition of Q-sets, there exists at least one such set since the trivial set $\emptyset \subseteq P$ is closed under $\otimes$.

Lemma 3.5.

$|Q_1| = 0$ if and only if $x_1 \otimes \cdots \otimes x_{|P|+1} \notin P$ for all $x_i \in V$.

Proof: If the RHS holds, then clearly only the empty set is closed under $\otimes$. Assume that the RHS does not hold, i.e., $x_1 \otimes \cdots \otimes x_{|P|+1} \in P$ for some $x_i \in V$. Then, $x_1 \otimes \cdots \otimes x_j \in P$ for all j, $1 \leq j \leq |P| + 1$. Since we have $|P| + 1$ possible values for j, we are guaranteed that $x_1 \otimes \cdots \otimes x_a \otimes \cdots \otimes x_b = x_1 \otimes \cdots \otimes x_a$ for some $a < b$. Let $x = x_1 \otimes \cdots \otimes x_a$, and $y = x_{a+1} \otimes \cdots \otimes x_b$. This shows that $x \otimes y^k = x$ for all values $k \geq 0$. Therefore, $y^k \in P$ for all values $k \geq 1$, showing the existence of a set $\{y\} \subseteq P$ that is closed under $\otimes$, concluding the proof. $\square$

If $|Q_1|$ is empty, then Lemma 3.5 shows that no matter what vector we pick from $V^{|P|+1}$, multiplying the elements from the vector results in an element that is not in P. If this is the case, then we can solve the prefix problem in constant time by splitting the input into blocks of size $|P|+1$ and solving the problem for each block independently. Indeed, Lemma 3.5 can be used to determine if this is the case since the RHS of the equivalence stated in Lemma 3.5 can be easily checked.

3.9. Related Work

Asynchronous adders were originally studied by Burks et al.[1] who showed that the average-case latency through a ripple-carry binary adder (assuming that the inputs were independently distributed and the zero-one probabilities were equal) was bounded by $\log_2 N$, where N is the number of bits being added. Winograd[28] showed that a lower bound on the worst-case time complexity for binary addition is $O(\log_2 N)$, where N is the number of bits in the input. The prefix problem and the formulation of binary addition as a prefix problem was proposed by Ladner and Fischer.[10] Gemmell and Harchol[5] present a method for adding two binary numbers "mostly correctly," with an error probability ϵ. They show lower bounds on the latency of such adders to be $O(\log\log(N/\epsilon))$. Our circuits always produce the correct answer with $O(\log\log N)$ latency. Gemmell and Harchol also claim an $O(\log N)$ lower bound on the average-case latency of binary addition in their abstract, which we have shown to be incorrect in this chapter by providing a construction for a $O(\log\log N)$ binary adder; closer inspection reveals that their lower bound only applies to "VRTC" (variable running time correct) circuits, showing that asynchronous circuits for addition have better latency characteristics than those constructed by their method.