Asynchronous Parallel Prefix Computation

Rajit Manohar*
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125. U.S.A.
rajit@cs.caltech.edu

José A. Tierno
T.J. Watson Research Center
Yorktown Heights, NY 10598. U.S.A.
jose@watson.ibm.com

June 8, 1998

Abstract: *The prefix problem is to compute all the products $x_1 \otimes x_2 \otimes \cdots \otimes x_k$, for $1 \leq k \leq n$, where $\otimes$ is an associative binary operation. We start with an asynchronous circuit to solve this problem with $O(\log n)$ latency and $O(n \log n)$ circuit size, with $O(n)$ $\otimes$-operations in the circuit. Our contributions are:*

(i) A modification to the circuit that improves its average-case latency from $O(\log n)$ to $O(\log \log n)$ time; and

(ii) A further modification that allows the circuit to run at full-throughput i.e., with constant response time.

The construction can be used to obtain a asynchronous adder with $O(\log n)$ worst-case latency and $O(\log \log n)$ average-case latency.

Keywords: asynchronous circuits, binary addition, prefix computation, average-case latency

1. Introduction

There has been a renewal of interest in the design of asynchronous circuits, motivated by the potential benefits of designing circuits in an asynchronous fashion. Asynchronous circuits exhibit average case behavior and can therefore be optimized in a data-dependent fashion. We present asynchronous solutions to the parallel prefix problem that exploit this advantage of asynchronous circuits over their synchronous counterparts to reduce the average case latency of the prefix computation.

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 $1 \leq k \leq n$ [6].

The prefix problem can be used to solve a number of problems efficiently. Ladner and Fisher show how the prefix problem can be used to parallelize the computation of an arbitrary Mealy machine [6]. Leighton discusses a number of different problems that can be solved using prefix computations [7]. As a concrete application, we use the method presented in this paper to construct an asynchronous adder with an average-case latency of $O(\log \log n)$ steps.

A variety of synchronous solutions to the prefix problem are discussed by Leighton, all having a latency of $O(\log n)$ steps [7]. Winograd has shown that a lower bound on the worst-case time complexity for binary addition is $O(\log n)$, where n is the number of bits in the input [10]. Therefore, $O(\log n)$ is a lower bound on the latency of any synchronous adder. Gemmell and Harchol construct circuits for binary addition which add correctly with probability $1 - \epsilon$ and which have a latency of $O(\log \log(n/\epsilon))$ steps. They construct an asynchronous adder that always adds correctly with an average-case latency of $O(\log n)$ steps [3].

We begin with an asynchronous solution that is very similar to its synchronous counterpart. We improve the performance of this solution by the introduction of pipelining, and by using two competing methods for solving the prefix problem and picking the answer that arrives earliest to produce the output. All the solutions presented have $O(n \log n)$ hardware complexity, and worst-case $O(\log n)$ time complexity.

* Supported by the Advanced Research Projects Agency under the Office of Army Research, and in part by a National Semiconductor Corporation graduate fellowship.

We use CHP (communicating hardware processes), a variant of CSP [4], to give a high-level description of our circuits (we use circuits to mean asynchronous circuits in the paper). A brief description of CHP is provided in the appendix.

2. The prefix problem

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:

$$
\begin{array}{l}
*[\;\; X_1?x_1,\;\; X_2?x_2,\;\; \ldots,\;\; X_n?x_n;\\
\quad Y_1!x_1,\;\; Y_2!(x_1 \otimes x_2),\;\; \ldots,\;\; Y_n!(x_1 \otimes x_2 \otimes \cdots \otimes x_n)\\
\;]
\end{array}
$$

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 paper, 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.

Assume we had a method for constructing a circuit to compute $a_1 \otimes a_2 \otimes \cdots a_m$, for $m = \lfloor n/2 \rfloor$ and $m = \lceil n/2 \rceil$. We could use these circuits to compute $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ by adding a single process that read in the output of the two stages, and performed a single $\otimes$ operation (since the operation is associative). The process would read in two inputs on channels A and B, and produce the desired output on channel C, and is written as:

$$
\begin{array}{l}
UP(L, R, U) \equiv\\
\qquad *[\;\; L?x, R?y;\;\; U!(x \otimes y)\;\;]
\end{array}
$$

The value $x_1 \otimes \cdots \otimes x_n$ can be computed using a tree of these UP processes, as shown in Fig. 1.

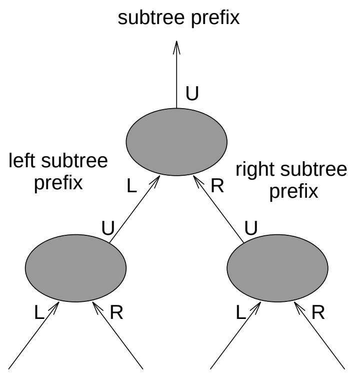

Fig. 1: Tree of "UP" processes.

For any input x_k, we need the prefix $x_1 \otimes \cdots \otimes x_{k-1}$ to compute the output y_k. Observe that the input to UP at a particular node in the tree is the prefix of the inputs at the leaves of the left and right subtree of the node. The prefix required by the first (leftmost) node of the right subtree can be computed if the prefix required by the first node of the left subtree is known. Assume that this prefix is obtained on another input channel V. Process UP can now be augmented to send the appropriate subtree prefixes back down the tree. The modified UP process is:

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

$*[\ L?x, R?y;\ \ U!(x \otimes y);$

$\quad V?p;\ \ Ld!p, Rd!(p \otimes x)$

$]$

Notice that the Ld and Rd channels provide exactly the inputs needed by the V channel of the children of a particular UP process, so this collection of processes indeed solves the prefix problem. All that remains is to provide an input to the root of the prefix computation tree, and to read the inputs and produce the final outputs.

The V channel at the root of the tree requires the null prefix, which is the identity e and the output of the root is not used by any other process. We can simplify the root process to:

$ROOT(L, R, Ld, Rd) \equiv$

$*[\ L?x, R?y;\ \ Ld!e, Rd!x\ ]$

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:

$LEAF(X, U, V, Y) \equiv$

$*[\ X?x; U!x;\ \ V?y; Y!(y \otimes x)\ ]$

A complete solution for the problem when $n = 4$ is shown in Fig. 2. Since each node in the tree contains a constant number of $\otimes$ computations, and since there are $O(n)$ nodes in the tree, and each node is of bounded fanin, there are $O(n)$ $\otimes$-computation circuits in the solution. The tree is of depth $O(\log n)$, and therefore the time complexity of this solution is $O(\log n)$.

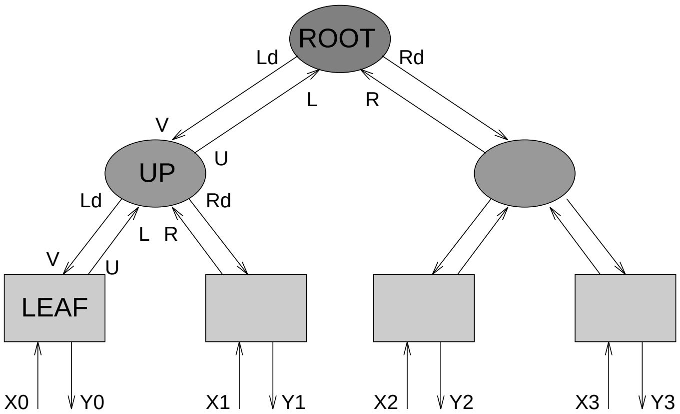

Fig. 2: Solution to the prefix problem.

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:

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

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

These two processes are identical! We therefore can write:

$UP2(A, B, C, D) \equiv$

$*[\ A?x, B?y;\ \ C!(x \otimes y), D!x\ ]$

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

Using a similar technique, we can rewrite the *LEAF* process as:

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

We compile each process in the tree using standard techniques introduced by Martin [9]. We begin by rewriting the processes using *handshaking expansions*. This transformation eliminates all communication on channels, and replaces them with handshake protocols that implement the synchronization and data communication [8].

For the circuit to be quasi-delay-insensitive, it must function correctly even if the inputs to the circuit do not arrive at the same time. Therefore, each input must be encoded using a delay-insensitive (unordered) code. In such a code, the value of the input changes from a *neutral* value to a *valid* value without any intermediate values that are valid or neutral [9]. Different valid values are used to encode different inputs. We use the functions $v(\cdot)$ and $n(\cdot)$ to denote the validity and neutrality of the code. $C \Uparrow$ is the concurrent assignment of some bits of C such that the result is an appropriate valid value without any intermediate value being valid or neutral, and $C \Downarrow$ is the concurrent assignment of some bits of C such that the result is a neutral value without any intermediate value being neutral or valid. The exact nature of these operations depends on the encoding scheme and operation $\otimes$.

A prefix computation is initiated by the environment by setting the inputs to some valid value. The environment then waits for the outputs to become valid, after which the inputs are reset to a neutral value. The next input is supplied after the outputs are reset to a neutral value.

The handshaking expansions for the processes that comprise the prefix computation are:

$$UP2(A, B, C, D) \equiv$$
$$*[\; [v(A) \wedge v(B)];\;\; C \Uparrow, D \Uparrow;\;\; [n(A) \wedge n(B)];\;\; C \Downarrow, D \Downarrow \;]$$

$$LEAF(X, U, V, Y) \equiv$$
$$*[\; [v(L)];\, U \Uparrow, C \Uparrow;\;\; [n(L)];\;\; U \Downarrow, C \Downarrow \;]$$
$$\|\; *[\; [v(C) \wedge v(V)];\;\; Y \Uparrow;\;\; [n(C) \wedge n(V)];\;\; Y \Downarrow \;]$$

$$ROOT(L, R, Ld, Rd) \equiv$$
$$*[\; [v(L) \wedge v(R)];\;\; Ld \Uparrow, Rd \Uparrow;\;\; [n(L) \wedge n(R)];\;\; Ld \Downarrow, Rd \Downarrow \;]$$

The handshaking expansions given above can be compiled into a quasi-delay-insensitive asynchronous circuit by the techniques outlined by Martin [9]. The resulting circuits are very similar to those shown in [6], and can be used in synchronous implementations as well.

In the programs presented here, we use a binary tree for the prefix computation. The method presented can be easily extended so that the tree is k-ary.

3. Pipelining

The solution presented above has the drawback that the tree can only perform one prefix computation at a time. To permit the tree to operate simultaneously on multiple inputs, we can pipeline the prefix computation.

Observe that since we must wait for the output to become valid before resetting the input, the protocol we have chosen earlier cannot be pipelined. To circumvent this problem, we introduce an additional acknowledge signal for each input and each output. The environment is now permitted to reset the inputs after receiving an acknowledge from the circuit, and can send the next input after the acknowledge signal has been reset.

With this small modification, the handshaking expansion for each stage can be written as:

$UP2(A, B, C, D) \equiv$
$\quad *[\ [\neg Ca \wedge \neg Da \wedge v(A) \wedge v(B)];\ \ C \Uparrow, D \Uparrow, Aa\uparrow, Ba\uparrow;$
$\quad\quad [Ca \wedge Da];\ \ C \Downarrow, D \Downarrow;\ \ [n(A) \wedge n(B)];\ \ Aa\downarrow, Ba\downarrow$
$\quad]$

$LEAF(X, U, V, Y) \equiv$
$\quad *[\ [\neg Ua \wedge \neg Ca \wedge v(L)];\ U \Uparrow, C \Uparrow, La\uparrow;\ \ [Ua \wedge Ca];\ \ U \Downarrow, C \Downarrow;\ \ [n(L)];\ \ La\downarrow\ ]$
$\quad \|\ *[\ [\neg Ya \wedge v(C) \wedge v(V)];\ \ Y \Uparrow, Ca\uparrow, Va\uparrow;\ \ [Ya];\ \ Y \Downarrow;\ \ [n(C) \wedge n(V)];\ \ Ca\downarrow, Va\downarrow\ ]$

$ROOT(L, R, Ld, Rd) \equiv$
$\quad *[\ [\neg Lda \wedge \neg Rda \wedge v(L) \wedge v(R)];\ \ Ld \Uparrow, Rd \Uparrow, La\uparrow, Ra\uparrow;$
$\quad\quad [Lda \wedge Rda];\ \ Ld \Downarrow, Rd \Downarrow;\ \ [n(L) \wedge n(R)];\ \ La\downarrow, Ra\downarrow$
$\quad]$

The signals which end in “a” are the acknowledge signals for the various channels.

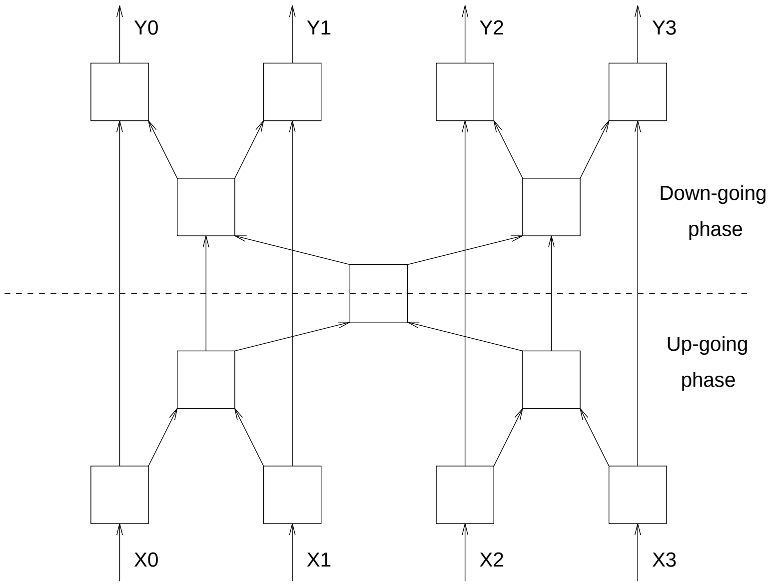

Fig. 3: Pipelined prefix computation.

Consider a single UP node in the prefix computation tree. There are no pipeline stages between the two halves of process UP. However, the down-going phase of the computation cannot begin until a value is received on channel V. This value is computed by a circuit which has pipeline stages proportional to the depth of the node in the tree. Therefore, even though there are $O(\log n)$ pipeline stages, we cannot have $O(\log n)$ computations being performed by the tree. To permit this, we must introduce buffering on the channel connecting the two halves of UP (and $LEAF$). This buffering is 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 Fig. 3. (The vertical arrows are the internal channels C.)

It is clear that one must add $2d-1$ stages of buffering on the internal channel C (shown as vertical arrows in Fig. 3) for a node that is d steps away from the root for the circuit to be pipelined in a manner that permits $2\log n+1$ simultaneous prefix operations to be performed. Fig. 4 shows the tree after the appropriate buffers have been introduced.

The *throughput* (the number of operations that can be performed per second) 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* (the time for the output to be produced when the pipeline is empty) of the computation block is proportional to the number of stages, and is therefore $2\log n+1$ stages both with and without the buffers. This circuit has $O(n\log n)$ components with bounded fan-in and fan-out, and $O(n)$ circuits that perform a $\otimes$-computation.

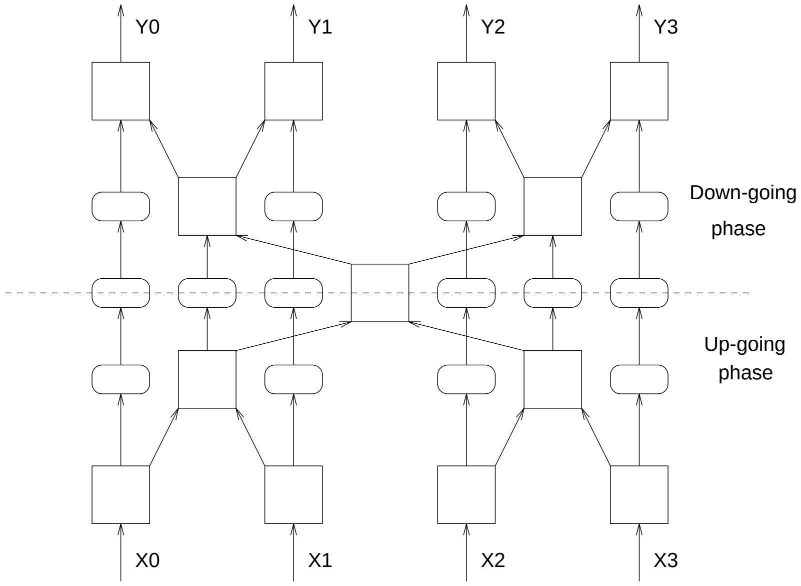

Fig. 4: Pipelined prefix computation with buffers.

4. Improving the average case latency

A pipelined prefix computation with buffers is useful when the the prefix computation is repeatedly performed. However, if the prefix computation is not used very often, then adding buffers to the computation tree does not improve the performance of the prefix computation. However, we may still be interested in minimizing the latency through the prefix computation tree. We begin by considering a simple solution to the prefix computation problem.

The simplest way to perform the prefix computation is in a bit-serial fashion. However, since we have n different input channels, we will use n processes, one for each input channel. The stages are connected linearly as shown in Fig. 5.

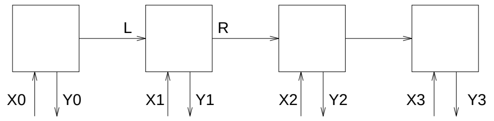

Fig. 5: Serial prefix computation.

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

$$
\begin{array}{l}
\mathit{SERIAL}(X, Y, L, R) \equiv \\
\qquad *[\;\; X?x, L?p;\;\; Y!(p \otimes x), R!(p \otimes x)\;\;]
\end{array}
$$

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 more about the prefix computation operator $\otimes$. In particular, suppose that $x \otimes a = a$ for all values of x. Now, if the input on channel X is equal to a, then we can produce the output on Y and R *before* reading the value on L! We can rewrite the *SERIAL* process as follows:

$$
\begin{array}{l}
\mathit{SERIAL}(X, Y, L, R) \equiv \\
\qquad *[\;\; X?x;\;\; [\;\; x = a \longrightarrow Y!a, R!a, L?p \\
\qquad\qquad\qquad\;\; [\!]\;\; x \neq a \longrightarrow L?p;\;\; Y!(p \otimes x), R!(p \otimes x) \\
\qquad\qquad\qquad\;\;] \\
\qquad\;\;]
\end{array}
$$

The time taken for this solution to produce the output is *data-dependent*. In the best case, the time from receiving the inputs to producing the output is constant—much better than the prefix computation tree—, and in the worst 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 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 modify the *LEAF* processes to include the serial computation in them. The original *LEAF* process used by the prefix computation tree is:

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

To add the serial computation phase, we add channels L and R to this process. Note that the value received along channel V is the same as the value received along channel L. We can combine these two channels externally using a merge process as follows:

$$
\begin{array}{l}
\mathit{MERGE}(L, V, M) \equiv \\
\qquad *[\;\; L?y, V?yy;\;\; M!y\;\;]
\end{array}
$$

Using this process, the new *LEAF* process is:

$$
\begin{array}{l}
\mathit{LEAF}(X, U, V, L, R, Y) \equiv \\
\qquad *[\;\; X?x; U!x, C!x]\;\; \|\;\; \mathit{MERGE}(L, V, M)\;\; \|\;\; \mathit{SERIAL}(C, Y, M, R)
\end{array}
$$

Finally, we modify *MERGE* so that it picks the first input to produce the output.

$$
\begin{array}{l}
\mathit{MERGE}(L, V, M) \equiv \\
\qquad *[[\overline{L} \longrightarrow L?y; M!y, V?yy \\
\qquad\;\;\; | \overline{V} \longrightarrow V?y; M!y, L?yy \\
\qquad\;\;]]
\end{array}
$$

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

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

(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 ripple-carry prefix computation. The resulting computation has this ripple-carry stage *at every level* in the prefix computation tree.

The prefix computation given above cannot run at full throughput, because there is $O(1)$ buffering between stages in the serial part of the computation. To improve the throughput of the prefix computation, we can once again introduce $(2d-1)$ buffers between each stage of the prefix computation at depth d in the tree. These buffers can be implemented using a folded fifo, which has $O(1)$ latency.

5. 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.

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

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.

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 additional delay introduced by adding one level to the tree (i.e., h is the delay of going upward and downward through one stage of the tree). The first part of the formula comes from the rippling 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. This is because the inputs to the next stage of the tree are constructed by grouping the inputs into blocks of size k; as a result, propagate sequences now occur between blocks of size k, and the longest such sequence will have length m/k. 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:

$$\langle L(n)\rangle \leq s + \frac{h}{\log k}\langle \log m\rangle$$

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

$$\langle \log m\rangle \leq \log\langle m\rangle$$

The reason is that the expected value of the logarithm of a random variable is the log 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 $\langle L(n)\rangle$ if we can determine $\langle m\rangle$.

When $p = 1/2$, we know that $\langle m\rangle \leq \log_2 n$ (cf. [1]). A simple extension of the proof shows that

$$\langle m\rangle \leq \lceil \log_{1/(1-p)} n\rceil + \frac{1}{np} = O(\log n)$$

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

$$\begin{aligned}\langle L(n)\rangle &\leq s + \frac{h}{\log k}\log\left(\lceil \log_{1/(1-p)} n\rceil + \frac{1}{np}\right)\\ &= O(\log\log n)\end{aligned}$$

which establishes that the average-case latency of the prefix computation is $O(\log\log n)$.

When the prefix computation operates at full-throughput, 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 (which can be implemented using a cache buffer or a folded fifo), the value of s is bounded above by a function that is $O(1)$. Therefore, the full-throughput modification does not increase the order of the average-case latency.

6. Application to binary addition

The prefix computation can be used to construct a binary *kpg*-adder [6]. To perform binary addition at bit position k, 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 k 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 1. Prefix operator for addition.

Observe that the *kpg* code has the desirable property outlined in the preceding section, namely that $x \otimes k = k$ and $x \otimes g = g$ for all values of x. Therefore, we can use the techniques of the previous section to reduce the latency of binary addition.

From the previous section, we observe that the average latency through such an adder is $O(\log\log n)$.

A 32-bit pipelined adder based on Section 3 (PA1-32) and 32-bit and 64-bit pipelined adders with $O(\log \log n)$ average-case latency from Section 4 (PA2-32 and PA2-64) were simulated for various input values using a gate-level simulator as well as HSPICE. We consider a gate to be any circuit that can be directly implemented in CMOS (such as a NAND, NOR, or C-element), with fan-in and fan-out restrictions that ensure that a gate delay is between 0.1ns and 0.2ns in HP's $0.6\mu m$ CMOS technology. We used $k = 4$ for the branching factor in the prefix computation tree, except where it would violate our gate delay restriction ($k = 2$ was used in that case). The quasi-delay-insensitive asynchronous circuits obtained as a result of compilation from handshaking expansions correspond closely to precharged quasi-static domino logic.

Kinniment compares the latencies of 32-bit ripple-carry asynchronous adders (ASY-32) to synchronous ripple-carry adders (SPA-32) and 32-bit synchronous carry select adders (CSA-32) [5]. To be able to compare our results to those reported in [5], we normalized our gate delays so that one gate delay is the delay through a two-input NAND gate (which is the assumption made in [5]). We extrapolated the delay for a 64-bit synchronous carry select adder (CSA-64) using the expression for delay given in his paper.

Two input distributions for random numbers were used: (a) A uniform distribution; (b) A distribution that corresponds to more realistic workloads for 32-bit adders [2]. The results of our simulation are shown in Table 2. The table includes the best delays for various adder implementations described in [5] as well. The throughput for the circuits PA1-32, PA2-32, and PA2-64 was one transfer every 16.3 gate delays.

Adder type	Worst Delay (gates)	Average Delay (a) (gates)	Average Delay (b) (gates)
SPA-32	55		
CSA-32	11.3		
CSA-64	14.1		
ASY-32	40.1	17.2	19.2
PA1-32	14.0	13.2	13.8
PA2-32	14.2	10.4	11.2
PA2-64	17.5	11.5	

Table 2. Delay through asynchronous and synchronous adders. (a) uniform; (b) workload

The difference in average-case and worst-case delay for PA1-32 is due to variations in the number of series transistors that switch in the logic. In terms of delay, CSA-32 is better than the simple prefix computation adder PA1-32. However, PA2-32 performs better than both PA1-32 and CSA-32 for random inputs as well as under more realistic workloads. PA2-64 performs significantly better than CSA-64 on random inputs. However, the PA2 adders have a slightly higher worst-case delay compared to PA1 adders because of increased fan-out of internal signals. We expect the difference between PA2-n and CSA-n adders to increase as n increases. However, the performance of CSA-n for small n suggests that a better asynchronous adder could be constructed using a combination of carry select and prefix computation techniques. Both PA1-32 and PA2-32 have larger area than CSA-32 due to pipelining overhead, and overhead introduced by circuitry that generates the acknowledge signals.

7. A variant of the prefix problem

The prefix computation can be used to determine the location of a leading one in a binary string. Such an operation is useful for both binary multiplication and division—pre-shifting the input reduces the number of stages required to compute the desired product or quotient.

Given an n-bit input, the prefix computation produces n-bits of output. Each bit of the input will know whether or not it is the position of the leading one. After the prefix computation, the number is required to be shifted by 0 to n bit positions to move the leading one to a fixed bit position. However, the input to a typical variable length shifter is encoded in binary, and not in the form produced by the prefix computation.

Observe that the sequence of decisions made by the nodes along the path taken by the leading one in the down-going phase of the prefix computation *is* the binary encoding of the bit-position of the leading one, assuming the prefix computation tree is balanced. (If the number of inputs is not a power of 2, then the tree can be balanced using dummy processes.) Therefore, if we augment the nodes in the tree to produce this bit, then we have produced the appropriate input for the shifter as part of the prefix computation itself. This is done by introducing a bus at each level of the down-going phase of the prefix computation tree.

In addition, note that the output of the buses are produced one after another, most significant bit first. Therefore, if we are using a pipelined logarithmic shifter, the shifting can be completely overlapped with the down-going phase of the prefix computation.

8. Conclusions

This paper presented a number of asynchronous solutions to the prefix problem. The first solution had $O(\log n)$ latency, and $O(\frac{1}{\log n})$ throughput using a circuit of size $O(n)$. The pipelined prefix solution had $O(\log n)$ latency and $O(1)$ throughput. The circuit size was increased to $O(n \log n)$, although it still contained $O(n)$ $\otimes$-computation blocks.

The latency of the prefix computation block was improved by using two competing prefix computations and picking the result that arrived first. The circuits had $O(\frac{1}{\log n})$ throughput and $O(n)$ circuit size. The circuits had a data-dependent latency with a worst-case latency of $O(\log n)$ and a best-case latency of $O(1)$. Under very general assumptions, the average-case latency of the prefix computation was shown to be $O(\log \log n)$.

Appendix

A1. Notation

The notation we use is based on Hoare's CSP [4]. A full description of the notation and its semantics can be found in [8]. What follows is a short and informal description of the notation we use.

- Assignment: $a := b$. This statement means "assign the value of b to a." We also write $a\uparrow$ for $a := \mathit{true}$, and $a\downarrow$ for $a := \mathit{false}$.
- Selection: $[G1 \to S1 \;[]\; ... \;[]\; Gn \to Sn]$, where Gi's are boolean expressions (guards) and Si's are program parts. The execution of this command corresponds to waiting until one of the guards is *true*, and then executing one of the statements with a *true* guard. The notation $[G]$ is short-hand for $[G \to \mathit{skip}]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Repetition: $*[G1 \to S1 \;[]\; ... \;[]\; Gn \to Sn]$. The execution of this command corresponds to choosing one of the *true* guards and executing the corresponding statement, repeating this until all guards evaluate to *false*. The notation $*[S]$ is short-hand for $*[\mathit{true} \to S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is *true* iff a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$

- Parallel Composition: $S \parallel T$ or S, T.

A2. Average case analysis

Given an n-input prefix computation, let c_n be the length of the longest sequence of propagate inputs. We would like to determine the expected value of c_n, assuming that the n inputs are independent, identically distributed random variables and that the probability of an input being of propagate type is $(1-p) = q$. We use a simple generalization of the reasoning presented by Burks et al. [1]. Clearly, the expected value of c_n is given by:

$$\sum_{k=1}^{n} \Pr[c_n \geq k] \qquad (*)$$

where $\Pr[c_n \geq k]$ is the probability that the length of the longest sequence of propagate inputs is at least k.

The probability $\Pr[c_n \geq k]$ consists of two parts: (a) the probability that the first $(n-1)$ inputs have a sequence of propagate inputs at least k; (b) the probability that the first $n-1$ don't have such a sequence but adding the nth input produces a sequence of length k. We can therefore write:

$$\Pr[c_n \geq k] = \Pr[c_{n-1} \geq k] + q^k(1-q)\,(1 - \Pr[c_{n-k-1} \geq k])$$

The second term (which corresponds to part b) is obtained by observing that of the n inputs, the last k inputs are of type propagate, and the input at position $n-k$ is not of type propagate. We also need to take into account the fact that the first $n-k-1$ positions do not have a propagate sequence of length at least k. Repeatedly expanding the first term, we obtain:

$$\begin{aligned}
\Pr[c_n \geq k] &= \Pr[c_k \geq k] + q^k(1-q) \sum_{i=k+1}^{n} (1 - \Pr[c_{i-k-1} \geq k]) \\
&= q^k + q^k(1-q)(n-k) - \sum_{i=k+1}^{n} \Pr[c_{i-k-1} \geq k] \\
&\leq q^k + q^k(1-q)(n-k) \\
&\leq q^k + q^k(1-q)n
\end{aligned}$$

To complete the proof, we note that $\Pr[c_n \geq k] \leq 1$. We split the range of the summation $(*)$ into two parts.

$$\begin{aligned}
\sum_{k=1}^{n} \Pr[c_n \geq k] &= \sum_{k=1}^{K-1} \Pr[c_n \geq k] + \sum_{k=K}^{n} \Pr[c_n \geq k] \\
&\leq K - 1 + \sum_{k=K}^{n} (q^k + q^k(1-q)n) \\
&= K - 1 + (1 + (1-q)n) \sum_{k=K}^{n} q^k \\
&= K - 1 + \frac{q^K(1+(1-q)n)(1-q^{n-K+1})}{1-q} \\
&\leq K - 1 + \frac{q^K}{1-q} + nq^K
\end{aligned}$$

Pick K such that $nq^K \leq 1$, i.e., pick $K = \lceil \log_{1/q} n \rceil$. We obtain:

$$\sum_{k=1}^{n} \Pr[c_n \geq k] \leq \lceil \log_{1/q} n \rceil + \frac{1}{n(1-q)}$$
$$= O(\log n)$$

References

[1] A.W. Burks, H.H. Goldstein, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Institute for Advanced Study, Princeton, N.J. June 1946. Also available in A.H. Taub, Ed., *Collected works of John von Neumann*, vol. 5. New York: Macmillan, 1963, pp. 34–79.

[2] J.D. Garside. A CMOS VLSI implementation of an asynchronous ALU. *Proceedings of the IFIP Workshop on Asynchronous Design Methodologies*, Manchester, UK, 1993.

[3] Peter Gemmell and Mor Harchol. Tight Bounds on Expected Time to Add Correctly and Add Mostly Correctly. *Information Processing Letters*, **49**:77–83, 1994.

[4] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–677, 1978

[5] D.J. Kinniment. An Evaluation of Asynchronous Addition. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **4**(1):137–140, March 1996.

[6] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the Association for Computing Machinery*, **27**(4):831–838, October 1980.

[7] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan-Kaufmann, 1992.

[8] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.

[9] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in Systems Design*, **1**(1):119–139, 1992.

[10] S. Winograd. On the Time Required to Perform Addition. *Journal of the Association for Computing Machinery*, **12**(2):277–285, April 1965.