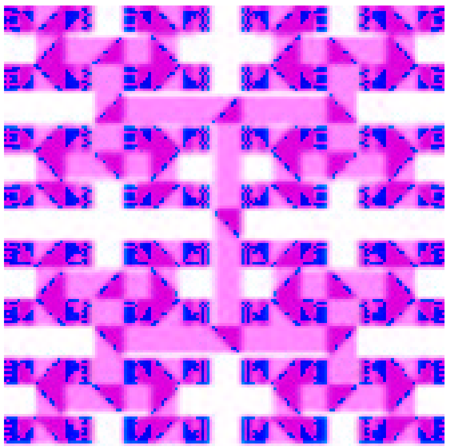

Ultrascalar Memo 1

Cyclic Segmented Parallel Prefix

Dana S. Henry and Bradley C. Kuszmaul

November 20, 1998

The cyclic segmented parallel prefix (CSPP) circuit is a varation on parallel prefix. Whereas ordinary parallel prefix computes prefix sums of a vector from the beginning, CSPP allows the starting point to move arbitrarily, with the data "wrapping around." The wraparound is widely useful. We have used CSPP to redesign many components of a superscalar processor to run in time logarithmic in the number of inputs.

Parallel-Prefix circuits and Segmented-Parallel-Prefix circuits are well understood. See for example, [1] for a discussion of log-depth Parallel-Prefix circuits. Segmented-Parallel-Prefix circuits are in an exercise of [1], and were implemented in the CM-5 supercomputer [7, 5, 4, 2].

The rest of this paper is organized as follows. Section 1 reviews the parallel-prefix problem along with the standard solutions.

Section 2 reviews the standard log-depth circuits for solving parallel prefix. Section 3 reviews segmented parallel prefix. Section 4 discusses a few minor variations on parallel prefix. Section 5 describes the cyclic segmented parallel prefix problem. Section 6 discusses more minor varations. Section 7 shows some examples.

1 Parallel Prefix

This section deines the parallel-prefix problem, and reviews the standard parallel-prefix circuits. Such solutions include linear-depth and quadratic-depth circuits.

The prefix problem is as follows. One is given an associative operator $\otimes$ with an identity value, I. Given some inputs $x_0, x_1, \ldots, x_{n-1}$ we need to compute $y_0, y_1, \ldots, y_n$ as:

$$y_i = x_0 \otimes x_1 \otimes \cdots \otimes x_{i-1},$$

where y_0 is defined to be the identity value for the $\otimes$. (For example, if $\otimes$ is addition (and the identity for addition is 0), then $y_i = \sum_{j=0}^{i-1} x_j$.)

Sometimes one wants special initial and final values. One can formulate the prefix problem as having an initial value z that is passed to the circuit. In this case we have

$$y_i = z \otimes x_0 \otimes x_1 \cdots \otimes x_{i-1}.$$

This can be viewed as the earlier case simply by renumbering the subscripts so that we have

$$x'_i = \begin{cases} z & \text{if } i = 0\text{, and} \\ x_{i-1} & \text{otherwise.} \end{cases}$$

and then performing a parallel prefix on the x' values. Similarly, one would like to get a final output value w from the circuit which is defined to be

$$w = z \otimes x_0 \otimes x_1 \cdots \otimes x_n.$$

Again, this can be implemented by the earlier case by manipulating subscripts. We simply extend the subscript range to $n+1$ and compute w as y_{n+1}.

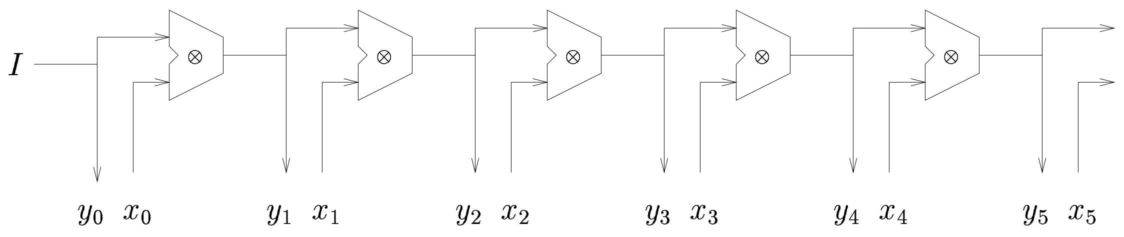

Figure 1: A linear-time prefix circuit.

The natural and easy thing to do is to compute the y_i's serially. First one computes each y_{i-1} and use that to compute y_i as

$$y_i = \begin{cases} \text{the identity value} & \text{if } i = 0\text{, and} \\ y_{i-1} \otimes x_{i-1} & \text{otherwise.} \end{cases}$$

Figure 1 shows a circuit that computes the prefix operation in linear time. Thus it is easy to see that prefix can be computed in time linear in n. It is surprising to many people that the prefix problem can be solved in time logarithmic in n by using a circuit structure known as *parallel prefix.* (Microprocessor designers are familiar with the fast carry-lookahead addition circuit to add two binary numbers in log time. It turns out that fast carry-lookahead is a special case in the class of parallel prefix circuits.)

2 Log-Time Parallel Prefix

Before reviewing the construction of parallel-prefix circuits in general, we present an example. Figure 2 shows a parallel-prefix circuit that takes eight inputs, $x_0, x_1, \ldots, x_7$, and computes their prefix sums $y_i = \sum_{j=0}^{i-1} x_j$. The inputs x_i are provided at the bottom of the circuit, and the outputs y_i come out the bottom, with output y_i coming out just to the left of where input x_i goes in. The identity (zero) and the sum of all the values (y_8) come out at the top. The critical-path length of this circuit is logarithmic in the number of inputs. This circuit can be layed out in VLSI using an H-tree layout [6] with a resulting area of about $A = O(n^2 b^2)$ where b is the number of bits in the result y_n. The resulting wire delay is about $O(\sqrt{A})$.

Note that one can further optimize the parallel-prefix sum circuit of Figure 2. If we use a redundant representation (such as the carry-save adder as used in Wallace-tree multipliers), with a single final sum at the end, we can perform the entire parallel-prefix sum in only $O(\log n)$ gate delays as opposed to $O(\log^2 n)$. Furthermore, often the width of the data values is smaller at the inputs than at the outputs (for example, when the inputs x_i to an sum are only one bit each, but the output it $\log n$ bits, a case which will come up later in this paper), then we can carefully size the ALUs so that they take just the right number of bits as inputs and produce the right number of bits as outputs, which will save area and power. One important special case is when the x_i's are one-bit each. The problem of summing one-bit inputs is often referred to as the *enumeration problem.* (Prior art: See [1, Exercise 29.2-8] for an exercise on building "tallying circuits" which have only $O(\log n)$ gate delays.)

In general, a parallel prefix circuit consists of a tree as shown in Figure 3. The x_i values are inputed at the leaves of the tree (at the bottom of the figure). The results y_i are also outputed at the leaves, adjacent to the corresponding x_i's. The identity value I is inputed at the root of the tree (at the top of the figure) and the result y_a of combining all the y_i values is outputed at the root of the tree. The values along each signal wire have been labeled. (The signal wires may be several bits wide in order to encode the necessary information.) We use the notation $p_{i,j}$ to indicate that a particular wire carries the value $x_i \otimes x_{i+1} \otimes \cdots \otimes x_j$. Thus

$$p_{i,j} = \bigotimes_{k=i}^{j} x_k.$$

(If $j < i$ then $p_{i,j}$ is the identity value.) The circuit computes

$$y_j = p_{0,j-1}$$

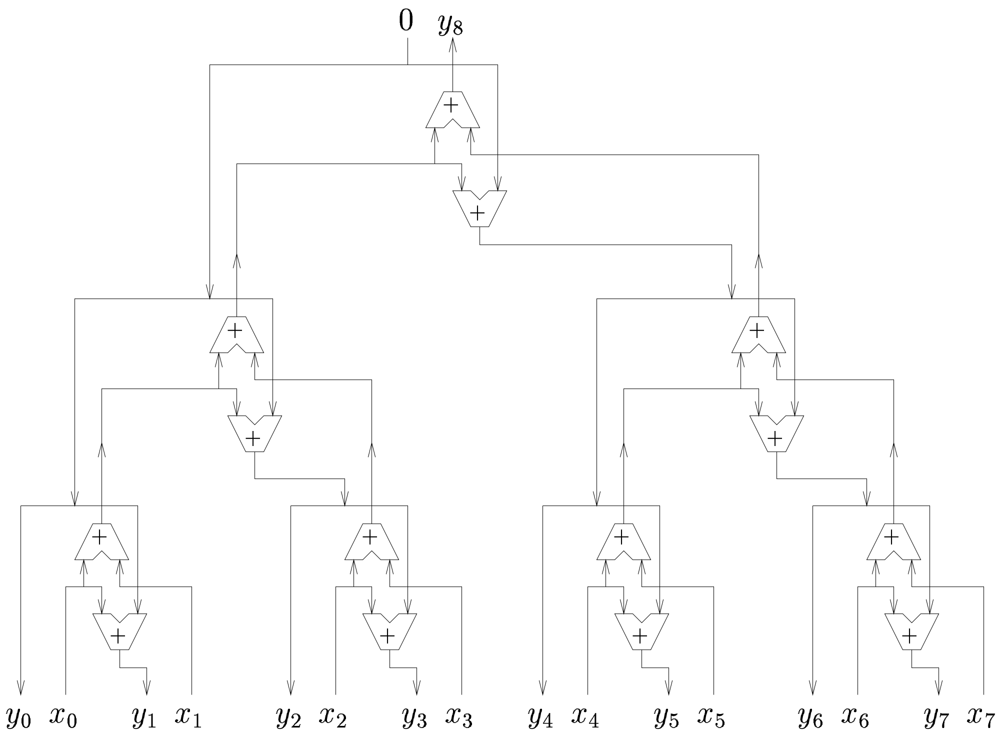
0
y_8
y_0 x_0 y_1 x_1 y_2 x_2 y_3 x_3 y_4 x_4 y_5 x_5 y_6 x_6 y_7 x_7
+

Figure 2: A parallel-prefix summation circuit.

for $0 \leq j \leq 7$. The vertices of the tree are called *treefix modules.* (See [1] for a discussion of how to adapt the circuit of Figure 3 to compute the special cases of w and z mentioned above.)

The treefix module of Figure 3 is shown in more detail in Figure 4. Each treefix module has three inputs and three outputs, arranged in pairs of an input and an output. One pair connects to the circuit above, one to the lower-left and one to the lower-right. There are some integers j, k, and m, with $j < k < m$, such that the data coming from above will be $p_{0,j-1}$, the data coming from the lower-left will be $p_{j,k-1}$ and the data coming from the lower-right will be $p_{k,m-1}$. The treefix module then produces $p_{j,m-1}$ outputed to above, it produces $p_{0,j-1}$ outputed to lower-left and produces $p_{0,k-1}$ outputed to lower-right. The reader can check that these are in fact the values carried on the wires of Figure 3. The circuit to compute these values is very easy to design since

$$\begin{aligned} p_{j,m-1} &= p_{j,k-1} \otimes p_{k,m-1}, \\ p_{0,j-1} &= p_{0,j-1}, \text{ and} \\ p_{0,k-1} &= p_{0,j-1} \otimes p_{j,k-1}. \end{aligned}$$

Although the tree in Figure 3 has a branching factor of two (that is, it is a binary tree), all the parallel-prefix circuits described in this paper can be implemented with an arbitrary branching factor. The choice of an appropriate branching factor depends on the parameters of a particular technology. For illustration, we will show all our circuits with a branching factor of two.

3 Segmented Parallel Prefix

We will also need *segmented* parallel-prefix circuits. The segmented-prefix problem is very similar to the prefix problem. A segmented-prefix operation consists of a collection of separate prefix operations over adjacent non-overlapping segments of the input $x_0, x_1, \ldots, x_{n-1}$. The way this works is that in addition to providing inputs x_i, we provide additional 1-bit inputs s_i called the *segment bits*. The segment bits indicate where a new segment is about to begin.

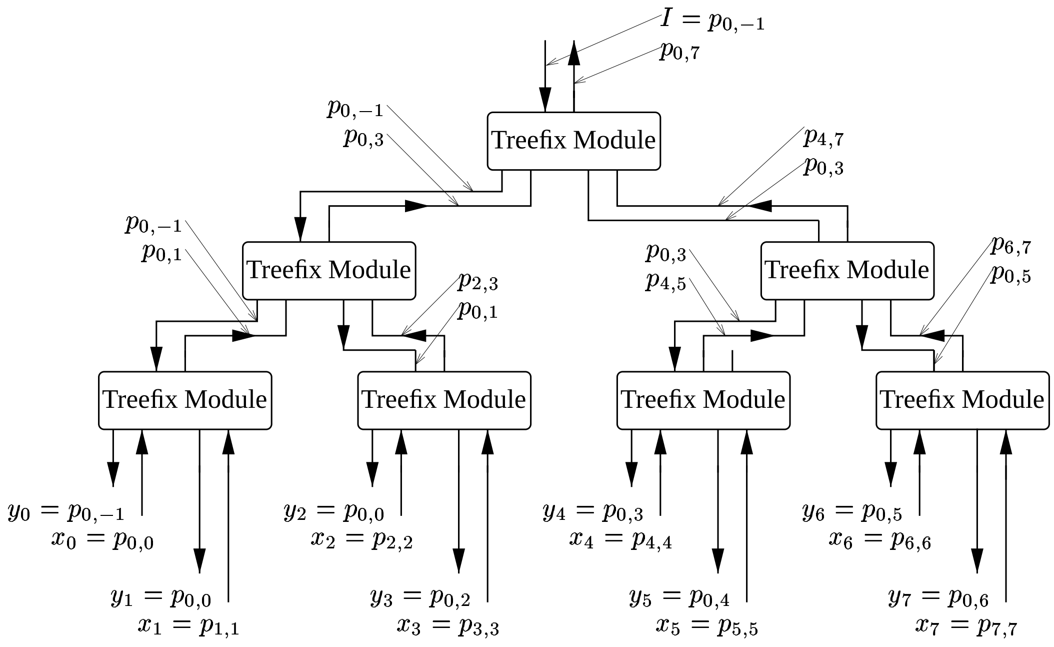

Figure 3: A parallel-prefix circuit.

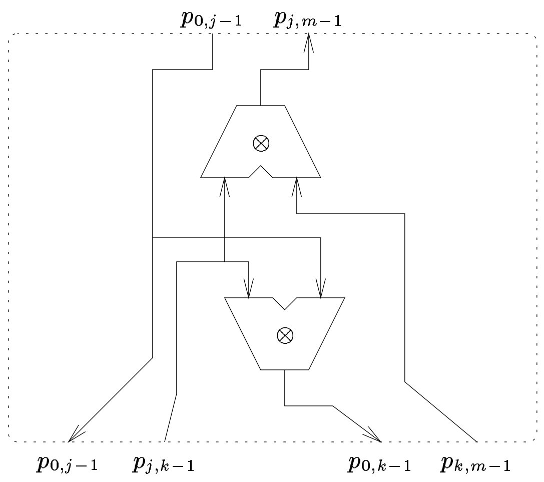

Figure 4: The treefix module.

The output is

$$y_i = \bigotimes_{j=k_i}^{i-1} x_i,$$

where

$$k_i = \max\{0, \max\{k < i : s_k = 1\}\}.$$

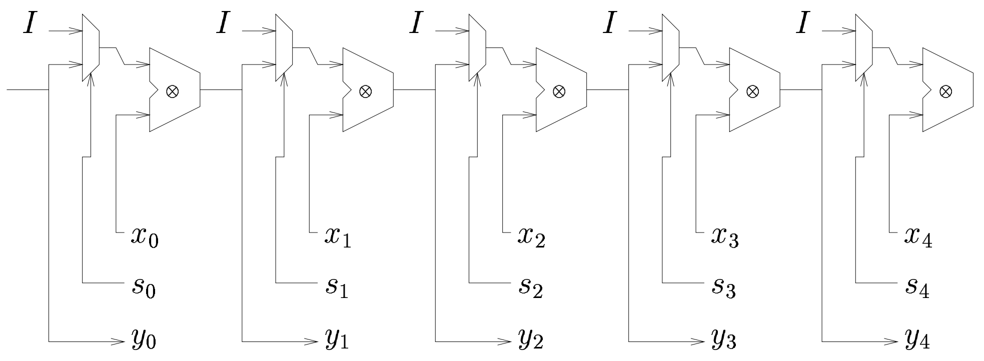

Figure 5: A linear-time segmented prefix circuit.

Thus if we have

$$\begin{array}{lcllllllllll}
x & = & \langle x_0, & x_1, & x_2, & x_3, & x_4, & x_5, & x_6, & x_7, & x_8, & x_9\rangle \\
 & = & \langle\ 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10\rangle \\
s & = & \langle s_0, & s_1, & s_2, & s_3, & s_4, & s_5, & s_6, & s_7, & s_8, & s_9\rangle \\
 & = & \langle\ 0, & 0, & 1, & 0, & 0, & 0, & 1, & 0, & 1, & 0\rangle \\
\text{then} \\
k & = & \langle k_0, & k_1, & k_2, & k_3, & k_4, & k_5, & k_6, & k_7, & k_8, & k_9\rangle \\
 & = & \langle\ 0, & 0, & 0, & 2, & 2, & 2, & 2, & 6, & 6, & 8\rangle \\
\text{and} \\
y & = & \langle y_0, & y_1, & y_2, & y_3, & y_4, & y_5, & y_6, & y_7, & y_8, & y_9\rangle \\
 & = & \langle\ 0, & 1, & 1+2, & 3, & 3+4, & 3+4+5, & 3+4+5+6, & 7, & 7+8, & 9\rangle \\
 & = & \langle\ 0, & 1, & 3, & 3, & 7, & 12, & 18, & 7, & 15, & 9\rangle.
\end{array}$$

A linear-time segmented parallel-prefix circuit is shown in Figure 5. This is similar to the circuit of Figure 1 except that MUXes have been added to handle the segment bits.

The segmented parallel-prefix circuit has the same structure as the ordinary parallel-prefix tree, except that we modify the treefix module to compute an additional *segmentation signal* $s_{j,k-1}$, which is passed up the tree. The value $s_{j,k-1}$ indicates if any of the segments bits $s_j, \ldots, s_{k-1}$ are equal to one. Figure 6 shows a segmented parallel-prefix circuit with eight leaf nodes ($n = 8$). The tree uses the slightly modified treefix module shown in Figure 7. An OR-gate within the module computes the segmentation signal that will be passed up. Two multiplexers (that is *MUXes*) serve the purpose that if there is a segment bit in a certain subtree, no value will be added in from the left subtree.

Note that a segmented parallel-prefix can be viewed as a prefix computation on values which are pairs: $\langle P, S\rangle$ where P is the value being operated on by the original operator, and S is a segmentation bit. We have the following operator:

$$\langle P_l, S_l\rangle \otimes\text{seg}\ \langle P_r, S_r\rangle = \begin{cases} \langle P_r, 1\rangle & \text{if } S_r = 1, \\ \langle P_l \otimes P_r, S_l\rangle & \text{otherwise.} \end{cases}$$

Note that this operator is associative. To show this we show that

$$(\langle P_a, S_a\rangle \otimes \langle P_b, S_b\rangle) \otimes \langle P_c, S_c\rangle = \langle P_a, S_a\rangle \otimes (\langle P_b, S_b\rangle \otimes \langle P_c, S_c\rangle).$$

Proof: If $S_c = 1$ then

$$\begin{array}{rcl}
(\langle P_a, S_a\rangle \otimes \langle P_b, S_b\rangle) \otimes \langle P_c, S_c\rangle & = & (\langle P_a, S_a\rangle \otimes \langle P_b, S_b\rangle) \otimes \langle P_c, 1\rangle \\
 & = & \langle P_c, 1\rangle \\
 & = & \langle P_c, S_c\rangle \\
 & \text{and} & \\
\langle P_a, S_a\rangle \otimes (\langle P_b, S_b\rangle \otimes \langle P_c, S_c\rangle) & = & \langle P_a, S_a\rangle \otimes (\langle P_b, S_b\rangle \otimes \langle P_c, 1\rangle) \\
 & = & \langle P_a, S_a\rangle \otimes \langle P_c, 1\rangle
\end{array}$$

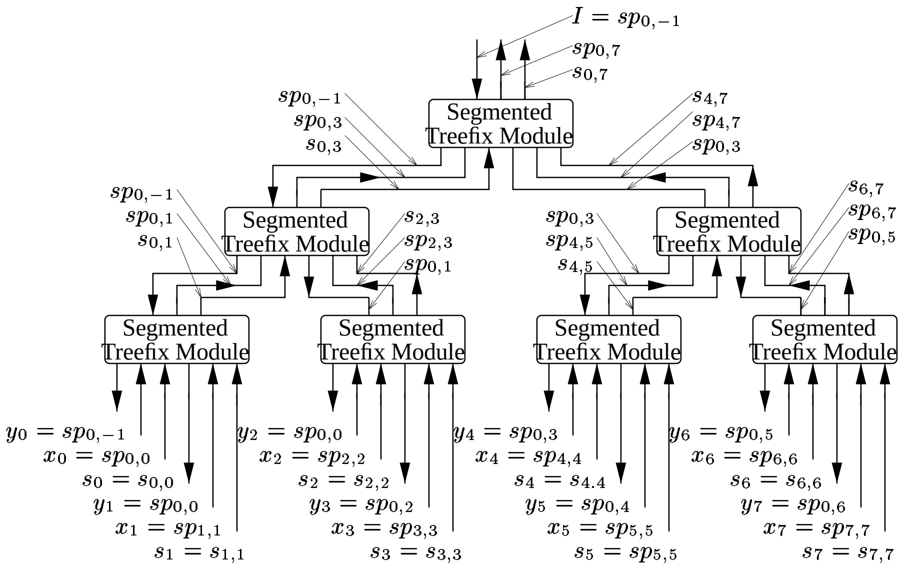

Figure 6: A segmented parallel-prefix circuit.

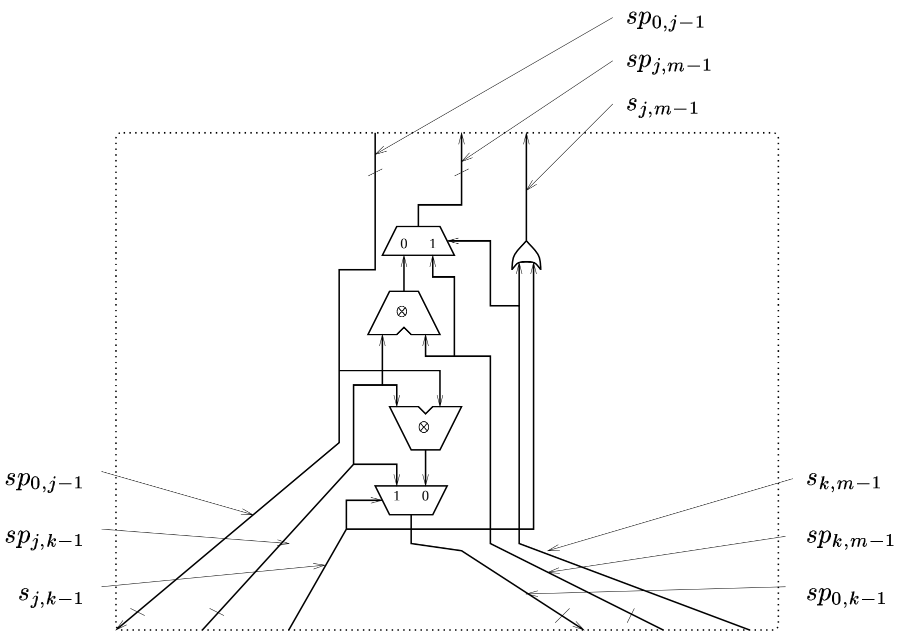

Figure 7: A segmented treefix module.

$$\begin{aligned} &= \langle P_c, 1\rangle \\ &= \langle P_c, S_c\rangle. \end{aligned}$$

Otherwise

$$\begin{aligned} ((\langle P_a, S_a\rangle \otimes \langle P_b, S_b\rangle) \otimes \langle P_c, S_c\rangle &= ((\langle P_a, S_a\rangle \otimes \langle P_b, S_b\rangle) \otimes \langle P_c, 0\rangle \\ &= ((\langle P_a, S_a\rangle \otimes \langle P_b, S_b\rangle) \\ \text{and} & \\ \langle P_a, S_a\rangle \otimes ((\langle P_b, S_b\rangle \otimes \langle P_c, S_c\rangle) &= \langle P_a, S_a\rangle \otimes ((\langle P_b, S_b\rangle \otimes \langle P_c, 0\rangle) \\ &= \langle P_a, S_a\rangle \otimes ((\langle P_b, S_b\rangle) \\ &= \langle P_a, S_a\rangle \otimes \langle P_b, S_b\rangle. \end{aligned}$$

Thus, the segmentation operator is associative, and our tree switch circuit can be viewed as an ordinary parallel-prefix circuit with a certain associative operator. (See [1, Exercise 30-1].)

4 Variations on Prefix Circuits

Often, the prefix is modified slightly from the formulae given above. For example, an *inclusive* prefix has

$$y_i = \bigotimes_{k=0}^{i} x_k$$

instead of

$$y_i = \bigotimes_{k=0}^{i-1} x_k.$$

An inclusive segmented prefix has

$$y_i = \bigotimes_{j=k_i+1}^{i} x_i,$$

instead of

$$y_i = \bigotimes_{j=k_i}^{i-1} x_i.$$

Sometimes it is useful to have a backwards prefix operation. An exclusive backwards prefix operation is

$$y_i = \bigotimes_{k=i+1}^{N-1} x_k.$$

Similarly, an inclusive version can be made with and without segmentation.

The practitioner must be careful to get the “fencepost” conditions right when implementating these circuits. That is, the lower and upper bounds to the summation index must be thought through carefully to avoid designed-in errors in the circuits.

Note: Sometimes prefix circuits are called “scan chains” (See, e.g., [3, 8].)

5 Cyclic Segmented Parallel Prefix

If we are guaranteed that at all times some segment bit is equal to one, then we can define the cyclic segmented prefix problem. This problem is the same as the segmented prefix, except that we want the values to “wrap around”. We can define this by using modulo arithmetic for the indices of x and s. We let

$$y_i = \bigotimes_{j=k_i}^{i-1} x_{(i \bmod n)},$$

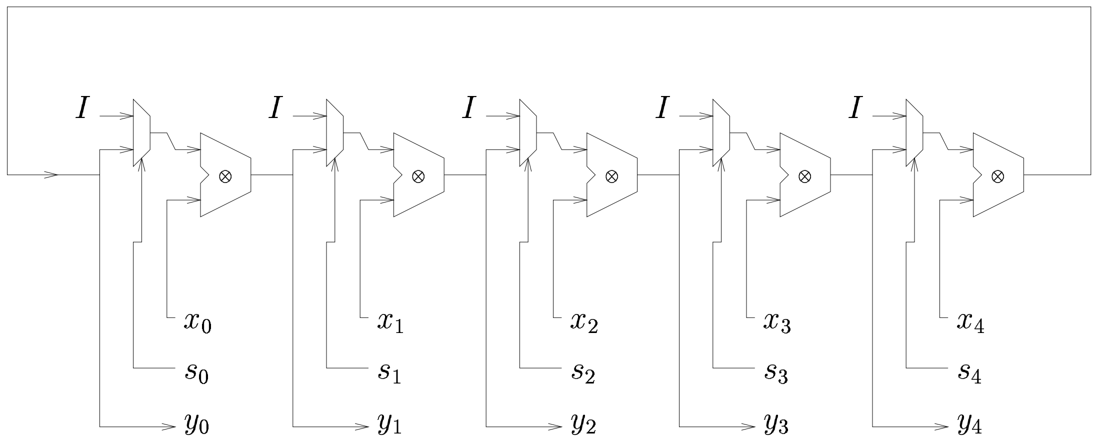

Figure 8: A linear-time cyclic segmented prefix circuit.

where

$$k_i = \max\{k \text{ any integer} : k < i \text{ and } s_{(k \bmod n)} = 1\},$$

and

$$i \bmod n = \min\{m \geq 0 : (i - m) \text{ is a multiple of n.}\}$$

Thus $-4 \bmod 10 = 6$. Note that k_i may be negative.

Returning to our previous example, if the input to a cyclic parallel prefix is

$$\begin{array}{rcrrrrrrrrrrr}
x & = & \langle & x_0, & x_1, & x_2, & x_3, & x_4, & x_5, & x_6, & x_7, & x_8, & x_9\rangle \\
 & = & \langle & 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10\rangle \\
s & = & \langle & s_0, & s_1, & s_2, & s_3, & s_4, & s_5, & s_6, & s_7, & s_8, & s_9\rangle \\
 & = & \langle & 0, & 0, & 1, & 0, & 0, & 0, & 1, & 0, & 1, & 0\rangle \\
\multicolumn{13}{l}{\text{then}} \\
k & = & \langle & k_0, & k_1, & k_2, & k_3, & k_4, & k_5, & k_6, & k_7, & k_8, & k_9\rangle \\
 & = & \langle & -2, & -2, & -2, & 2, & 2, & 2, & 2, & 6, & 6, & 8\rangle \\
\multicolumn{13}{l}{\text{and}} \\
y & = & \langle & y_0, & y_1, & y_2, & y_3, & y_4, & y_5, & y_6, & y_7, & y_8, & y_9\rangle \\
 & = & \langle & 9+10, & 9+10+1, & 9+10+1+2, & 3, & 3+4, & 3+4+5, & 3+4+5+6, & 7, & 7+8, & 9\rangle \\
 & = & \langle & 19, & 20, & 22, & 3, & 7, & 12, & 18, & 7, & 15, & 9\rangle.
\end{array}$$

A linear-time cyclic segmented prefix circuit is shown in Figure 8. This is similar to the linear-time prefix circuits of Figure 5 except the circuit is wrapped around to be a ring.

A cyclic segmented parallel-prefix circuit can be implemented by starting with a segmented parallel prefix tree, and modifying it as follows: connect the output at the root of the tree to the input at the root, discarding the segment bit. Figure 9 shows a cyclic segmented parallel prefix circuit. The individual treefix modules are the same as for the segmented acyclic circuit in Figure 7. The only difference from the acyclic segmented circuit is the root of the tree. The treefix module at the root of the acyclic tree is removed, and the outputs of the two subtrees are simply connected together (the input of the left subtree connected to the output of the right subtree, and the input of the right subtree connected to the output of the left subtree, and the segment summaries coming out of each subtree ignored..) Note that the tree produces some value even if none of the segment bits are equal to one (there is no cycle in the combinational logic), but the value produced is not the indicated by the formula above.

Figure 10 shows a cyclic segmented parallel-prefix circuit layed out in an H-tree layout. The subtree that would be drawn on the left in Figure 9 is at the top of Figure 10, and the right tree is at the bottom of Figure 10. The root node of the tree has been optimized to take advantage of the fact that at least one of the segment bits must be set to one.

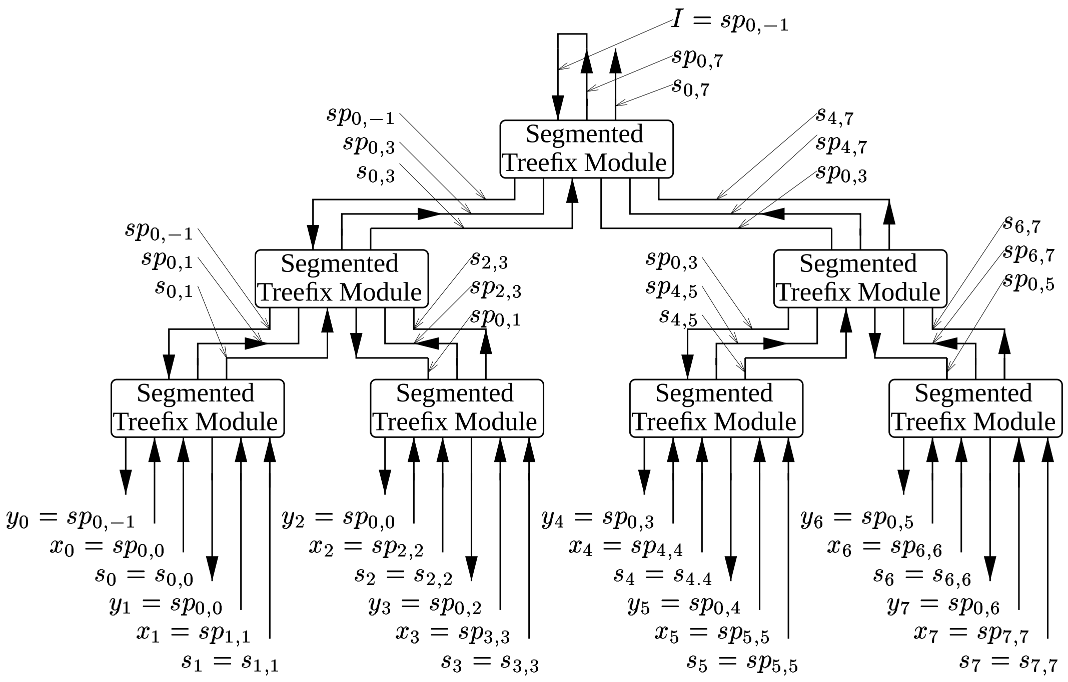

Figure 9: CSPP.

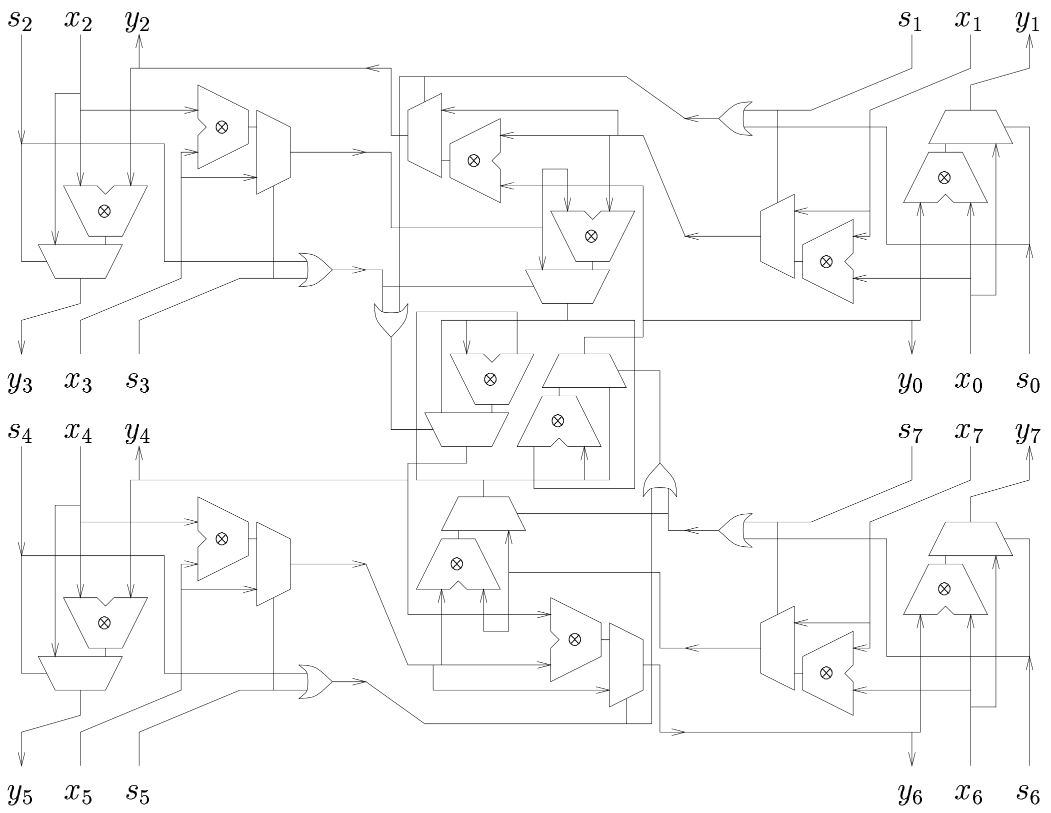

Figure 10: H-Tree Layout of CSPP.

6 Variations of CSPP

Just as for ordinary prefix circuits one can have inclusive and/or backwards prefix operations, so can one implement inclusive and/or backwards prefix operations with cyclic segmented parallel-prefix circuits.

It is clear that there are many alternatives for building prefix circuits. One can build a linear-time prefix chain, one can build a separate tree to compute each output, possibly sharing some common subexpressions, or one can build a single bidirectional tree as described here. One can vary the degree of the tree, so that instead of each node having two children, each node could have three or more children. One could vary the timing methodology. For example, in a self-timed system data travelling along the tree links would carry its own clocking information. Or one could build a globally clocked system. Other timing schemes would work as well. One could vary the technology so that the circuits are implemented in CMOS or Bi-CMOS or other logic families, and the communications links could be implemented with traces on circuit boards, or twisted pair cables, or coaxial cables, or fiber optic cables, or other technologies. One could use standard circuit optimization techniques to improve the speed of the circuit. For example, in some technologies, NAND gates are faster than any other gates. In such technologies, the circuits using AND and OR gates could be transformed into circuits using only NAND gates. Another example will discussed in Section 7 and is shown in Figure 13, which shows a cyclic segmented parallel-prefix of the AND function. In that case, the MUX has been simplified to an OR gate.

7 Examples of CSPP

This section describes a number of CSPP circuits that we find useful in processor design. At block-level, all the CSPP circuits that we are about to describe are identical to each other. For example, Figure 9 showed a CSPP circuit with 8 leaf nodes ($n = 8$). The different CSPP circuits only differ in their implementation of treefix modules. In this section, we will describe several useful CSPP circuits in terms of their function and their treefix module implementation.

The first example is the “oldest” CSPP circuit. The “oldest” CSPP circuit passes to each leaf node in the CSPP tree the input value of the oldest node in its segment. This circuit can be used to to pass data between producer and consumer instructions in a wrap-around instruction window, for example. The “oldest” CSPP circuit uses the operator $a \otimes b = a$. Figure 11 shows the circuity inside each treefix module for the “oldest” CSPP circuit.

Another very useful type of a CSPP circuit is the conjunction CSPP circuit. The conjunction CSPP circuit tells each leaf node in the CSPP tree whether all preceding leaf nodes within its segment have met a certain condition. For example, a conjunction CSPP circuit can tell each instruction within an instruction window whether all preceding instructions have computed. The conjunction CSPP circuit uses the operator $a \otimes b = a \wedge b$. Figure 12 shows the circuity inside each treefix module for the conjunction CSPP circuit. The circuit shown takes a k-bit input, and performs bitwise logical AND operations on those inputs. This can be viewed as performing k independent AND operations using a single, shared, segment bit. Of course, the circuitry within each treefix module can be optimized. For example, Figure 13 shows an optimized, 8-leaf-node conjunction CSPP with input and output width of 1 ($k = 1$).

The disjunction CSPP circuit indicates whether any previous node within the segment has met a certain condition. This can be implemented by inverting the x_i inputs and the y_i outputs for the conjunction CSPP circuit, or it can be implemented directly. Figure 14 shows the unoptimized treefix module for a disjunction CSPP circuit. Of course this module can also be optimized from

$$f(a,b,s) = \begin{cases} a \vee b & \text{if } \overline{s} \\ b & \text{otherwise,} \end{cases}$$

to

$$f(a,b,s) = b \vee (a \wedge \overline{s}).$$

A third important type of a CSPP circuit is the summation CSPP circuit. The summation CSPP circuit tells each leaf node the sum of the preceding leaf nodes’ inputs within its segment. A summation CSPP can be used, for example, to sum up the number of requests for a resource. A straightforward implementation of a summation CSPP uses a lookahead adder for each $\otimes$ operator. A more efficient implementation can be achieved by using carry-save adders. Figure 15 shows one possible implementation of the treefix module for a summation CSPP circuit. This CSPP circuit represents each sum by two partial sums that add up to the sum. For each $\otimes$, the module uses two cascaded carry-save adders (CSA) that reduce four partial sums to two partial sums. Each CSA takes three n bit inputs and produces two $n + 1$ bit outputs. As partial sums propagate up the tree, they increase in size by one bit at each treefix module. (In some situations, it is known that the output requires at most n bits to represent, in which case the high

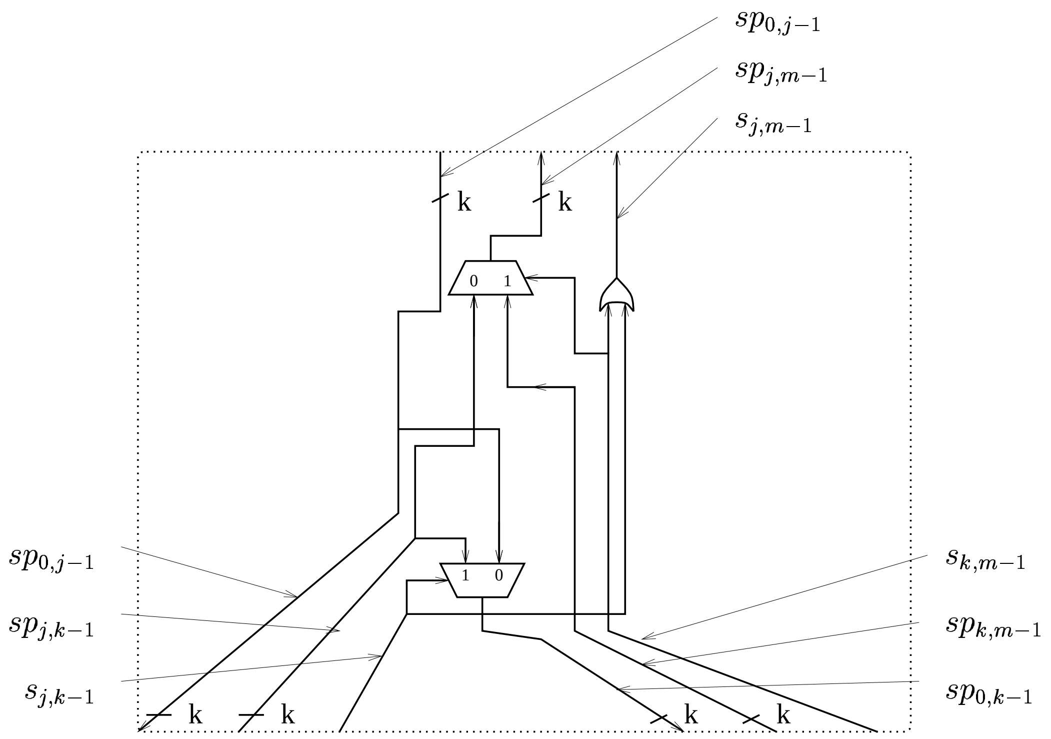

Figure 11: An “Oldest” Treefix Module.

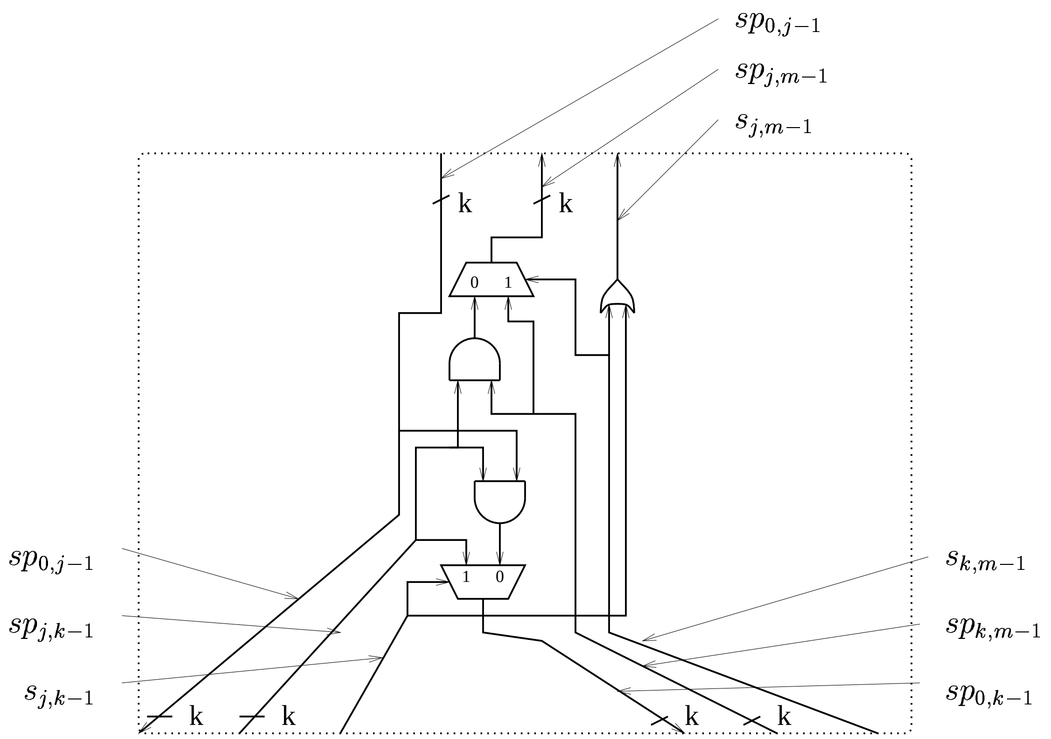

Figure 12: A Conjunction Treefix Module.

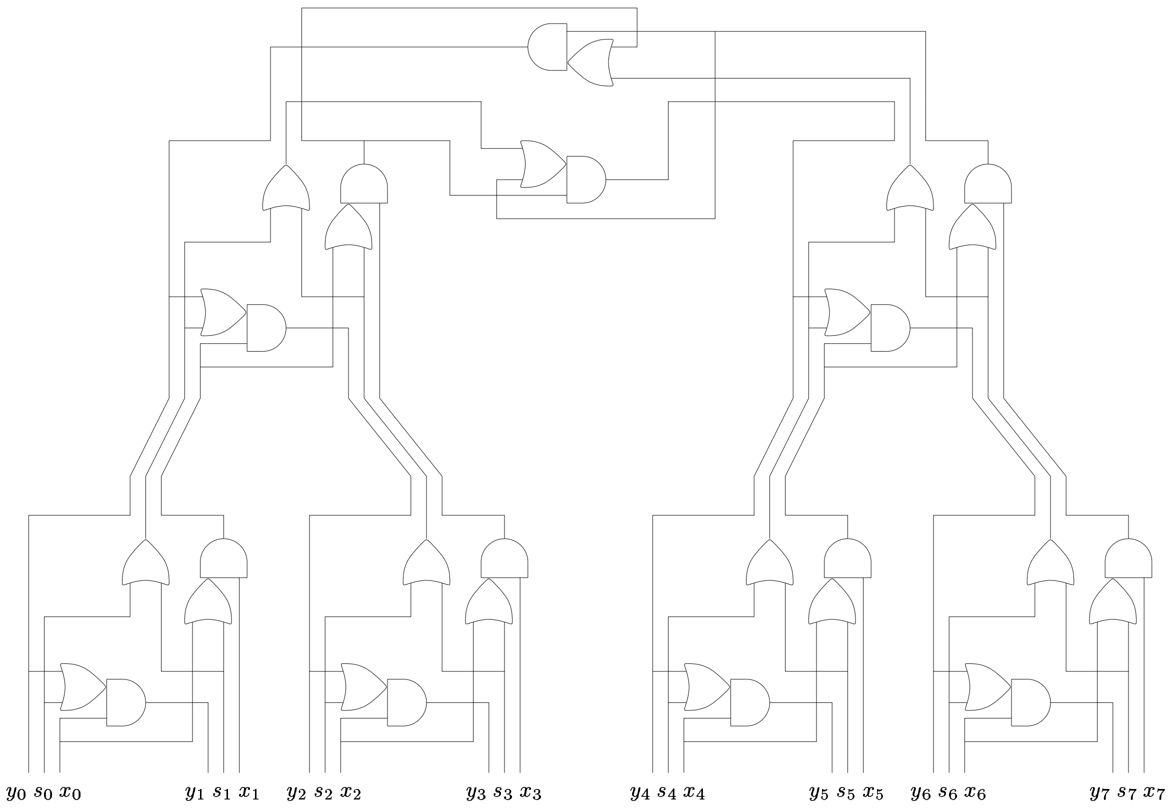

Figure 13: An optimized Conjunction CSPP.

order bits can be removed from the circuit. An example of such a situation is adding together 32-bit numbers to get a 32-bit sum for an address calculation.)

The advantage of using CSA circuitry is that the entire summation of n numbers to produce a k-bit result can be performed with critical-path delay $\Theta(\log n + \log k)$ instead of $\Theta(\log n \cdot \log k)$ for adders using a nonredundant representation.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1990.

[2] David C. Douglas, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Charles E. Leiserson, David S. Wells, Monica C. Wong, Shaw-Wen Yang, , and Robert C. Zak. Parallel computer system. U.S. Patent 5,333,268, issued July 26, 1994.

[3] John G. Favor, Amos Ben-Meir, and Jeffrey E. Trull. Scan chain for rapidly identifying first or second objects of selected types in a sequential list. U.S. Patent 5,745,724, 28 April 1998.

[4] W. Daniel Hillis, David C. Douglas, Charles E. Leiserson, Bradley C. Kuszmaul, Mahesh N. Ganmukhi, Jeffrey V. Hill, and Monica C. Wong-Chan. Parallel computer system with physically separate tree networks for data and control messages. U. S. Patent 5,590,283, issued December 31, 1996.

[5] Bradley C. Kuszmaul, Charles E. Leiserson, Shaw-Wen Yang, Carl R. Feynman, W. Daniel Hillis, David Wells, and Cynthia J. Spiller. Parallel computer system including arrangement for quickly draining messages from message router. U.S. Patent 5,390,298, issued February 14, 1995.

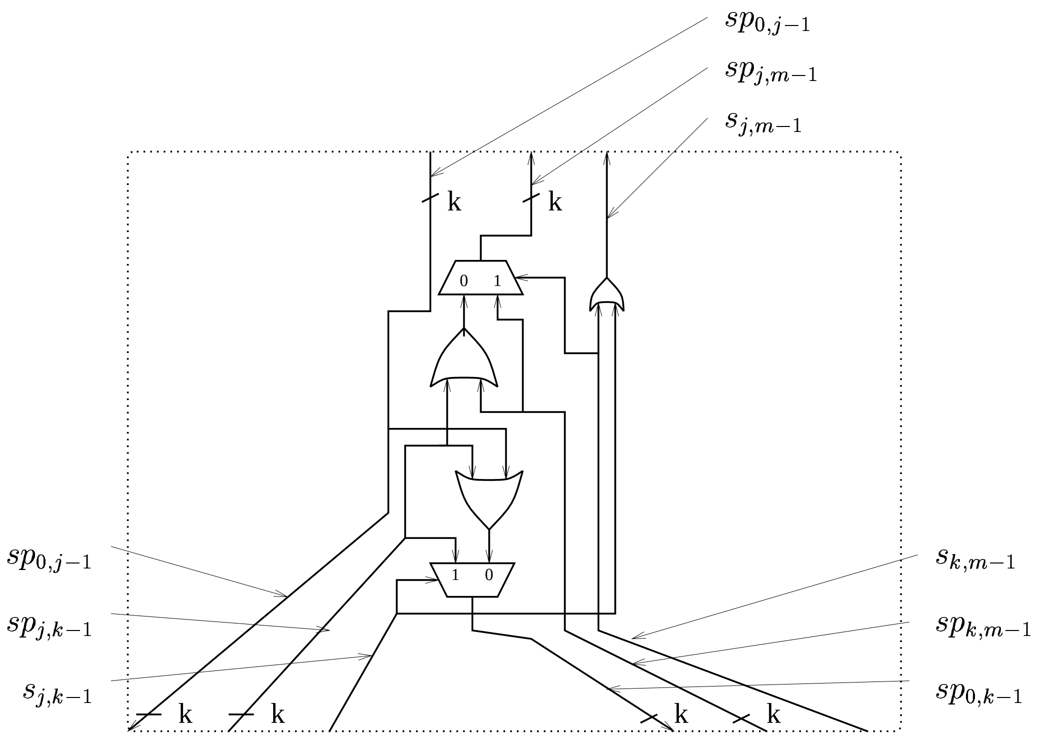

Figure 14: A Disjunction Treefix Module.

[6] Charles E. Leiserson. *Area-Efficient VLSI Computation*. The MIT Press, 1982. ACM Doctoral Dissertation Award 1982.

[7] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. In *4th Annual Symposium on Parallel Algorithms and Architectures (SPAA '92)*, pages 272–285, June 1992. `ftp://theory.lcs.mit.edu/pub/cel/spaa92.ps.Z` ftp://theory.lcs.mit.edu/pub/cel/spaa92.ps.Z.

[8] Robert W. Martell and Glenn J. Hinton. Method and apparatus for scheduling the dispatch of instructions from a reservation station. U.S. Patent 5,519,864, 21 May 1996.

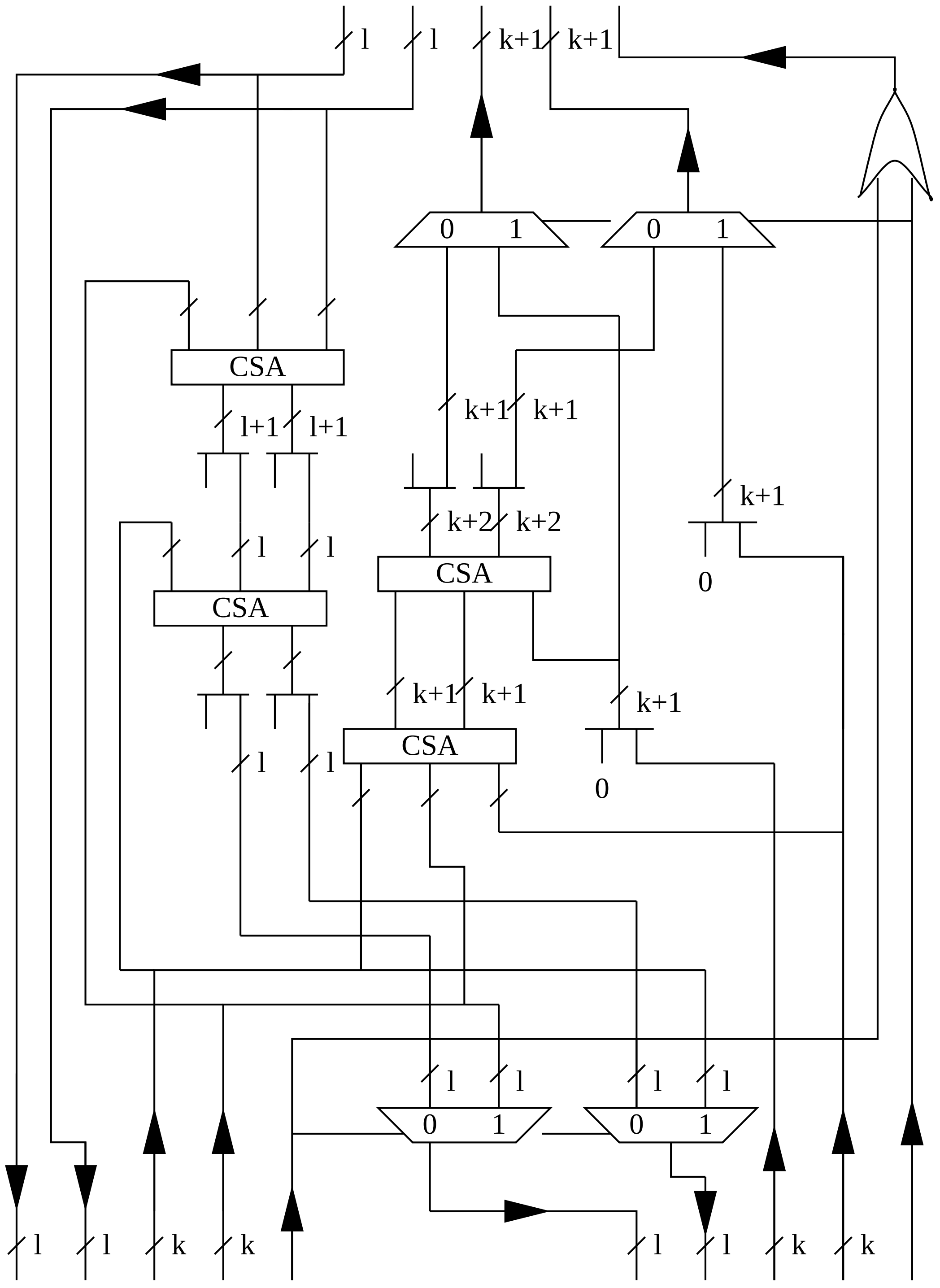

Figure 15: A Summation Treefix Module.