18.435/6.848 Theory of Parallel and VLSI Computation	**Fall 1997**

Solutions to Assignment #2

TA: Daniel Lewin — *Prof. Leighton*

Problem 1

Assume the a_i values are stored one per processor in order from left to right (so a_0 goes in the leftmost processor and a_{N-1} in the rightmost). Input and output will both be at the left end of the array. The input comes in every other step. Every processor P_i runs the following protocol: if there is input from the left, say x_j, and input from the right, say s_k, then output x_j to the right and output $s_k + a_i x_j$ to the left. Assume that P_n always sees 0 as its input from the right. It is easy to verify that x_t is input and $y_t = \sum_{k=0}^{N-1} a_k x_{t-k}$ is output at step $2t$ for all t.

The input needs to come in at every step so that inputs meet each other at the right time.

Problem 2

The basic idea is to have each processor in the $N/2$ processor linear array simulate two adjacent processors in the N processor linear array used in the solution to problem 1. The x's can now be input at every step, and it is easy to check that they meet in the correct processors. Since each processor can do two multiplies, the y's are output at every step.

Problem 3

Given y, we first normalize y by shifting the number left/right an *even* number of times until $1/4 \le y \le 1$. We shift an even number of times so that after computing the root of the normalized value we can shift the number back *half* the number of steps and get the root of the original number.

Now, we will compute the root x of the polynomial $f(x) = x^2 - y$ using Newton's rule. Since $f'(x) = 2x$, we get the following improved guess:

$$x_{i+1} = x_i - \frac{x_i^2 - y}{2x_i} = \frac{x_i}{2} + \frac{y}{2x_i}$$

We choose $x_0 = 3/4$. First we show that if we use infinite precision, the number of correct bits double each iteration.

Let $\epsilon_i = x_i - \sqrt{y}$.

$$
\begin{aligned}
\epsilon_{i+1} &= \frac{1}{2}\left(x_i + \frac{y}{x_i}\right) - \sqrt{y} \\
&= \frac{1}{2}\left[(x_i - \sqrt{y}) + \sqrt{y}\left(\frac{\sqrt{y}}{x_i} - 1\right)\right] \\
&= \frac{1}{2}\left[\epsilon_i - \frac{\epsilon_i\sqrt{y}}{x_i}\right]
\end{aligned}
$$

So, using $x_i = \epsilon_i + \sqrt{y}$ we have:

$$
\begin{aligned}
\epsilon_{i+1} &= \frac{\epsilon_i^2}{2(\epsilon_i + \sqrt{y})} \\
&= \frac{\epsilon_i^2}{2x_i}
\end{aligned}
$$

Now, for $i \geq 1$,:

$$x_i = \frac{1}{2}\left(x_{i-1} + \frac{y}{x_{i-1}}\right) \geq \sqrt{x_{i-1}\frac{y}{x_{i-1}}} = \sqrt{y} \geq 1/2$$

Since $y \geq 1/2$. So we have shown that $|\epsilon_{i+1}| \leq |\epsilon_i|^2$.

Now, we show that $|\epsilon_i| \leq 2^{2^{i+1}}$ by - you guessed it - induction. The base case $|\epsilon_0| = |x_0 - \sqrt{y}| \leq 1/4$ holds since $1/2 \leq \sqrt{y} \leq 1$. The inductive step follows immediately from $|\epsilon_{i+1}| \leq |\epsilon_i|^2$.

This implies that the number of correct bits doubles at every iteration. So, if we start with an initial guess of 3/4 and repeat for $\log N$ iterations, we get N bits of the square root. Each iteration involves a division and an addition which both can be done in $O(N)$ time on the linear array (as we saw in class). This gives a total time of $O(N \log N)$.

To get $O(N)$ time we use the same trick that we saw in class - keep less bits around and perform the addition and divisions on fewer bits. To show that this works, we need to show that if in the ith stage we use : $y_i = y$ rounded down to the $2^{i+2} + 1$ bit position instead of y and x_i to a precision of $2^{i+1} + 1$ bits, then we still have $|\epsilon_i| \leq 2^{-2^{i+1}}$.

Observe that $y \geq y_{i+1} \geq y_i$. y_0 is rounded down to the 5th bit position, so $y_0 \geq 1/4$, and $y_i \geq 1/4$ for all i. As before we have $1/2x_i \leq 1$.

$$
\begin{aligned}
\epsilon_{i+1} &= x_{i+1} - \sqrt{y} \\
&= \frac{1}{2}\left(x_i + \frac{y_i}{x_i}\right) - \sqrt{y} \\
&= \frac{\epsilon_i^2 - (y - y_i)}{2x_i}
\end{aligned}
$$

Now, if x_{i+1} is rounded up to the $2^{i+2}+1$th bit position, we have (using the inductive hypothesis and the fact that $\frac{y-y_i}{2x_i}$ is non negative) :

$$
|\epsilon_{i+1}| \le max\left(\frac{\epsilon_i^2}{2x_i}, \frac{y-y_i}{2x_i} + 2^{-2^{i+2}-1}\right)
$$

Now, $\frac{\epsilon_i^2}{2x_i} \le \epsilon_i^2 \le 2^{-2^{i+2}}$.

And also:

$$
\begin{aligned}
\frac{y-y_i}{2x_i} + 2^{-2^{i+2}-1} &\le (y-y_i) + 2^{-2^{i+2}-1} \\
&\le 2^{-2^{i+2}-1} + 2^{-2^{i+2}-1} \\
&= 2^{-2^{i+2}}
\end{aligned}
$$

So therefore, $|\epsilon_{i+1}| \le 2^{-2^{i+2}}$ even when we round the intermediate values in the algorithm. Now since division and addition of $O(2^i)$ bit numbers can be done in $O(2^i)$ steps on a folded linear array as we saw in class, we have total running time of $O(N)$.

Problem 4

We first show how to embed a N-cell ring into a N-cell linear array. Let

$$
f_N(i) = \begin{cases} 2i & 1 \le i \le \lfloor N/2 \rfloor \\ 2(N-i)+1 & \lfloor N/2 \rfloor + 1 \le i \le N \end{cases}
$$

Then processor i of the N-cell ring is embedded into processor $f_N(i)$ of the linear array. To show that our dilation is two, consider processors i and $i+1$, which are neighbors in the ring. It can be verified that $f_N(i)$ and $f_N(i+1)$ differ by at most 2. It should be clear that the mapping is one-to-one. Also, since each edge of the linear array simulates at most two edges of the ring, we have shown that a N-cell ring can be simulated on a N-cell linear array with slowdown two.

Now to simulate a $N \times N$ torus on a $N \times N$ array, embed node (i, j) of the torus into node $(f_N(i), f_N(j))$ of the array. It can be verified that two nodes which are neighbors in the torus are at most distance 2 in the array (this does need to be proved). Moreover, each horizontal (vertical) edge of the array simulates at most two horizontal (vertical) edges of the torus. Thus the simulation can be carried out with slowdown 2.

Problem 5

Let x_{ij}, for $1 \le i, j \le N$ be the nodes in a $N \times N$ mesh which calculate the product of two matrices in $O(N)$ time as we learned in class. In this algorithm, each node required only constant storage. We now use a $\sqrt{P} \times \sqrt{P}$ grid to emulate this $N \times N$ grid with a factor of N^2/P slowdown. In this emulation, node p_{ij} emulates nodes $x_{i'j'}$, where $(i-1)\lceil \frac{N}{\sqrt{P}} \rceil + 1 \le i' \le i \lceil \frac{N}{\sqrt{P}} \rceil$ and $(j-1)\lceil \frac{N}{\sqrt{P}} \rceil + 1 \le j' \le j \lceil \frac{N}{\sqrt{P}} \rceil$. This way, processors which are neighbors in the original $N \times N$ array either reside in the same processor in the $\sqrt{P} \times \sqrt{P}$ grid or reside in neighbors of this smaller grid. It is important that neighbors be maintained. Since each of the P processors will be emulating N^2/P processors, they will require N^2/P storage. Every N^2/P steps of the algorithm, a node in the $\sqrt{P} \times \sqrt{P}$ array emulates one step of each of the N^2/P processors in the $N \times N$ mesh that it is responsible for. Thus the slowdown is N^2/P, and the time is $O(N^3/P)$. Since the original algorithm was efficient (with respect to the $\Theta(N^3)$ sequential algorithm), and this algorithm is N^2/P times slower with a factor of N^2/P fewer processors, our new algorithm is efficient, as well.

We use the same technique discussed in the previous paragraph to simulate an $N \times N$ array on a $M \times M$ array. Divide the N processors in each row into M blocks, each with $\lceil N/M \rceil$ processors. (The last block may contain fewer processors.) Do the same for each of the N columns. Now index each block according to its row and column block number. Simulate the (i, j) block $1 \le i, j \le M$ by the processor in the ith row and jth column of the $M \times M$ array. Figure PS2.1 illustrates the scheme for simulating a 8×8 array by a 3×3 array.

The load (the maximum number of processors simulated by any processor in the $M \times M$ array) is at most $\lceil N/M \rceil^2$. Moreover, processors that were neighbors in the $N \times N$ array are either simulated on the same processor or neighboring processors in the $M \times M$ array. Also, an edge in the $M \times M$ array simulates at most $\lceil N/M \rceil$ edges. Thus, the slowdown of our simulation is $\lceil N/M \rceil^2$.

Can we do better? In general, the answer is no, shown here by contradiction. Assume that a $M \times M$ array simulates a $N \times N$ array with slowdown of $\lceil N/M \rceil^2 - 1$. Consider the case when $M \mid N$. Then, each processor in the $M \times M$ array simulates at most $\lceil N/M \rceil^2 - 1 = (N/M)^2 - 1$ processors of the $N \times N$ array. But this totals only $N^2 - M^2 < N^2$ processors on the $M \times M$ array, contradicting the fact that our simulation was faithful.

Problem 6

The Seidel relaxation rule takes the form:

$$x_i(t+1) = -\frac{1}{a_{ii}} \left(\sum_{j<i} a_{ij} x_j(t+1) + \sum_{j>i} a_{ij} x_j(t) - b_i \right)$$

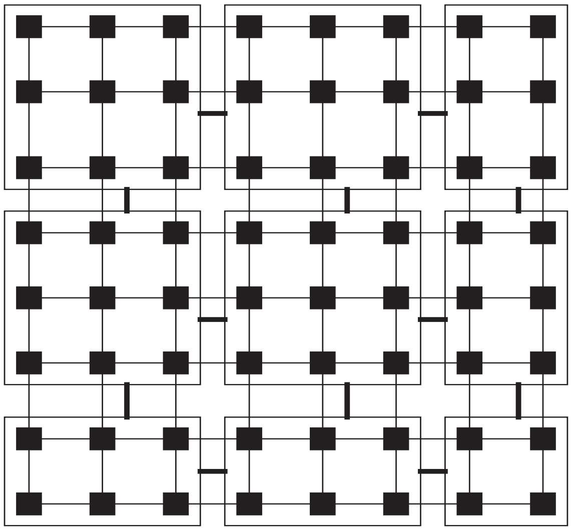

Figure PS2.1: Simulating 8×8 array on 3×3 array. Thick edges represent edges on the smaller array.

This is equivalent to:

$$X(t+1) = LX(t+1) + UX(t)$$

where $X(i) = \begin{pmatrix} x_1(i) \\ x_2(i) \\ \vdots \\ x_N(i) \\ 1 \end{pmatrix}$,

$$L = \begin{pmatrix} 0 & 0 & \dots & 0 & 0 \\ -\frac{a_{21}}{a_{22}} & 0 & \dots & 0 & 0 \\ \vdots & & & & \\ -\frac{a_{N1}}{a_{NN}} & -\frac{a_{N2}}{a_{NN}} & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix} \text{ and } U = \begin{pmatrix} 0 & -\frac{a_{12}}{a_{11}} & \dots & -\frac{a_{1N}}{a_{11}} & \frac{b_1}{a_{11}} \\ 0 & 0 & \dots & -\frac{a_{2N}}{a_{22}} & \frac{b_2}{a_{22}} \\ \vdots & & & & \\ 0 & 0 & \dots & 0 & \frac{b_N}{a_{NN}} \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}.$$

So, $(I-L)X(t+1) = UX(t)$. Or, equivalently, if we let $H = (I-L)^{-1}U$, $X(t+1) = HX(t)$. Note that since the diagonal components of L are strictly zero, $I-L$ is invertible.

Problem 7

In the case where the input and output hosts are separate, there are no cycles at all so no slowdown is needed. We compute the lag function from $G-1$ and then apply this lag function to G producing our systolic network — the lag function and retimed network are shown in Figure PS2.2.

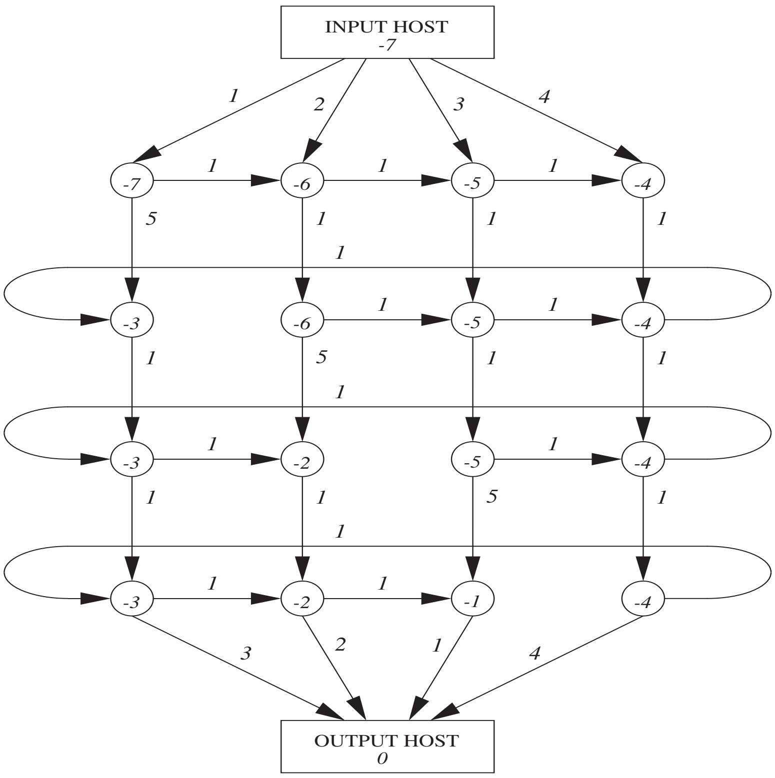

Figure PS2.2: Retiming when the input and output hosts are separate. Values of the lag function are given for each processor, and the final register counts in the systolic network are given for each wire.

When the input and output hosts are merged we have to slow the network down. Figure PS2.3 shows a cycle of 11 edges which has only 4 registers in the semisystolic model; thus the network needs to be slowed by a factor of 3. The lag function is computed from $3G - 1$ and then applied to $3G$ producing the systolic network shown in Figure PS2.3.

Problem 8

The idea is to let the first processor be a filter which removes punctuation symbols from the input and passes non-punctuation symbols on to the palindrome recognizer. With this modification, the palindrome recognizer no longer receives a symbol at every other step; it may wait long interludes between inputs. During these interludes, we cannot allow the palindrome recognizer to keep passing symbols on to the right. When the filter removes a punctuation symbol from the input, it should broadcast a signal to the processors of the palindrome recognizer telling them to not do anything. This seems to require both broadcasting and accumulating. Unfortunately, if we add broadcast edges to the original palindrome recognizer (before retiming), the resulting semisystolic circuit has zero-weight cycles which cannot be removed by retiming.

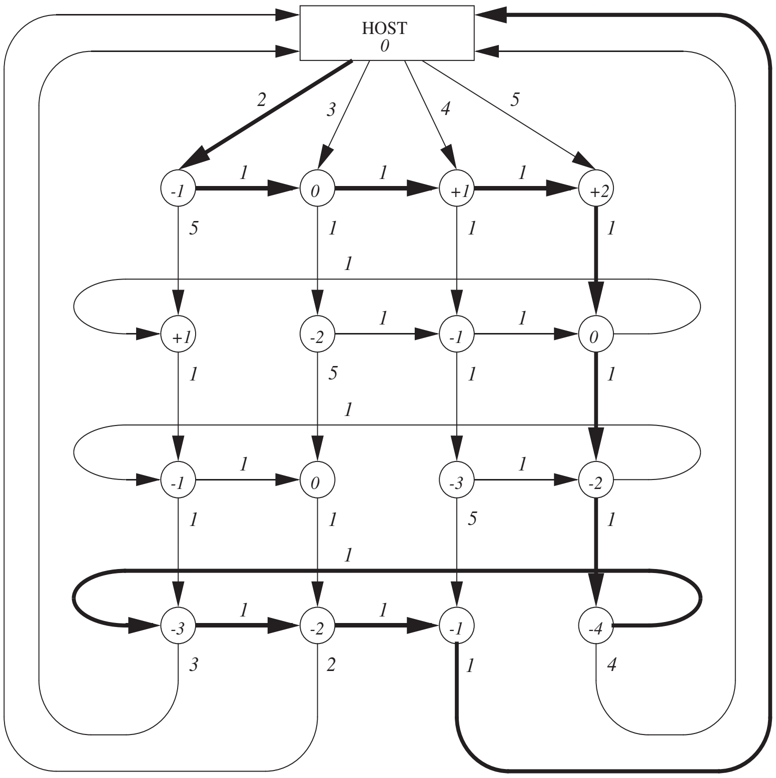

Figure PS2.3: Retiming when the input and output hosts are merged. The bold lines illustrate a cycle of 11 edges with only 4 registers in the semisystolic model, so the network needs to be slowed by a factor of 3. Values of the lag function are given for each processor, and the final register counts in the systolic network are given for each wire.

The next obvious step to solving this problem is to add the broadcast edges to the *retimed* circuit. In class we saw that this trick did not work in the case of adding a FIND command to a priority queue that already had a DELETE command. The problem is that after retiming the DELETE command, it is no longer possible to add accumulate edges to make a query like "is data item 6 in the queue at time 5?" because some nodes in the queue have been lagged and are not yet at time 5. Fortunately, this trick *does* work in the special case of adding broadcast edges to broadcast the signal "wait for one step". In fact, if we add this broadcast to any systolic circuit the circuit will perform as if the ignored input character had never appeared. The network with added broadcast is shown in Figure PS2.4.

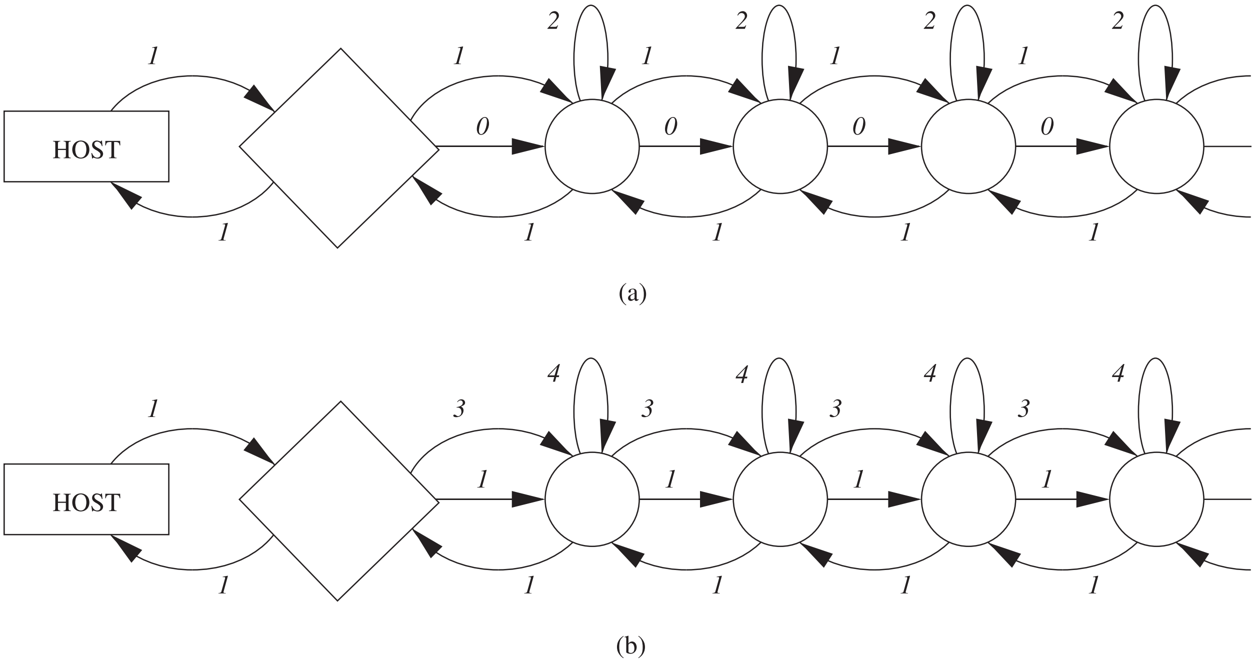

Figure PS2.4: Adding broadcast to the systolic palindrome recognizer. (a) Before retiming. (b) After retiming.

There is a subtle difficulty to this approach involving the interleaving of problems. Recall that the retimed palindrome recognizer solved two interleaved problems simultaneously. Consider the case where the input string is $\mathfrak{a}_1\mathfrak{b}_1\mathfrak{a}_2\mathfrak{b}_2 * \mathfrak{b}_3\mathfrak{a}_4\mathfrak{b}_4$, where $*$ is a punctuation mark. The palindrome-with-punctuation circuit will behave as if the input string were $\mathfrak{a}_1\mathfrak{b}_1\mathfrak{a}_2\mathfrak{b}_2\mathfrak{b}_3\mathfrak{a}_4\mathfrak{b}_4$. Now the two problems are no longer interleaved correctly in the input! The solution is to always ignore characters in pairs; both $*$ and $\mathfrak{b}_3$ should be ignored by the circuit. Now the circuit can only solve one problem at a time, unless we are guaranteed to have the punctuation in both $\mathfrak{a}$ and $\mathfrak{b}$ come in the same places.

Problem 9

The problem is slightly more difficult than the palindrome recognizer, but the network is exactly the same. The key to recognizing strings of the form ww is that we can use the accumulate wires to pass along different values at different time steps. At even steps, we want to accumulate the "equality" value, i.e., whether the input value in each register is equal to the value of that register. At odd steps, we want to pass the last value received in the rightmost register "back" to the leftmost register. See Figure PS2.5.

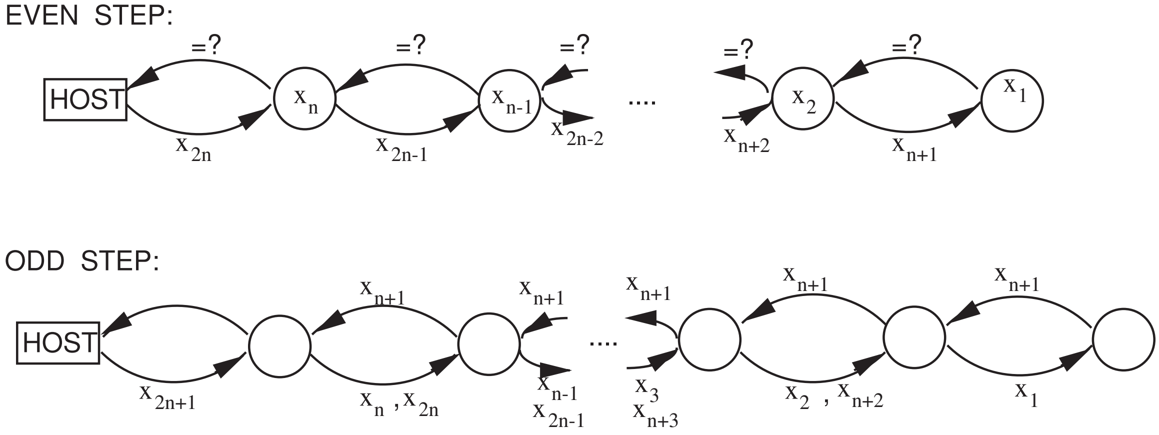

Figure PS2.5: Semisystolic circuit for recognizing strings of type ww

Problem 10

The array for convolution has precisely the same structure as the palindrome recognizer. The differences are:

1. Pairs (a_i, b_i) are entered as input rather than x_i.
2. Rather than comparing the two pairs (a_i, b_i) and (a_j, b_j) as in the palindrome recognizer, the value $a_i b_j + a_j b_i$ is calculated. (If there is only one pair in the processor, the value $a_i b_i$ is calculated).
3. The accumulation is used to sum all these values rather than to perform a global AND across the array.

The array for integer multiplication uses nearly the same algorithm as the convolution above where the a_i's and b_i's are 0 or 1, except it must concern itself with the carry bits generated. In order to deal with this problem, each processor holds an extra value C which represents the carries generate by previous steps. At each step:

1. Each processor calculates $d = a_i b_j + b_i a_j$ as above (note: d can be no more than 2).
2. Let p be the value coming in along the accumulation line from the right. This value represents the low order bit of the partial sum thus far calculated. The processor

calculates $r = C + d + p$ and sends the low order bit of this sum down the accumulation line.

3. Removing the low order bit from r we get the value of the new carry C, to be used for the next step.

C can be represented by 2 bits, because in the worst case, $C = 2, d = 2, p = 1$, so that $r = 5$, leaving $C = \lfloor \frac{r}{2} \rfloor = 2$ for the next step.