INFORMATION PROCESSING 86, H.-J. Kugler (ed.)
Elsevier Science Publishers B.V. (North-Holland)

VLSI ARCHITECTURES FOR BACK SUBSTITUTION*

Kam Hoi CHENG+ and Sartaj SAHNI
Computer Science Department, University of Minnesota
Minneapolis, Minnesota 55455, U.S.A.

VLSI architectures involving unidirectional, bidirectional and broadcast chains as well as unidirectional rings are examined for the back substitution problem. We develop designs with superior performance to earlier designs.

1. INTRODUCTION

VLSI architectures for a variety of problems have been proposed by several authors. A bibliography of over 150 research papers dealing with this subject appears in {6}. In this paper, we are concerned solely with the back substitution problem. The inputs to this problem are a non-singular lower triangular matrix A and a column vector b. The objective is to determine the unique column vector x with the property $Ax = b$. Throughout this paper, we assume that A is $n \times n$. So, x and b are $n \times 1$. The classical approach to obtain x is to use back substitution. x is obtained using the formula:

$$x_i = (b_i - \sum_{j=1}^{i-1} a_{ij} x_j)/a_{ii} , \qquad 1 \leq i \leq n$$

The x_is are computed in the order $x_1, x_2, \ldots, x_n$. VLSI architectures for this problem have been proposed earlier in {2}, {4}, {7}, and {8}.

The design of {2} employs a unidirectional chain of processors. The data flow is left to right. The design of Kung and Leiserson ({4}, {5} and {8}) employs a bidirectional chain. Here, data is permitted to flow both from left to right and from right to left. In {7}, a ring architecture. Data can flow in only a single direction (either clockwise or counterclockwise) around the ring.

We examine each of the above three architectures here. In addition, we consider the broadcast chain used in {3} and {1} (among others) for the matrix multiplication problem. To our knowledge, there has been no earlier research on the use of broadcast chains for back substitution. A broadcast line has the property that data put on this line becomes available at all PEs on the line in O(1) time.

In evaluating our designs, we assume that the VLSI system will be attached to a host processor via a bus. The evaluation of a VLSI design should take the following into account:

1. Bus bandwidth --- how much data is to be transmitted between the host and the VLSI system in any cycle? This figure is denoted by B.
2. Speed --- how much time does the VLSI system need to complete its task? This time may be decomposed into the times T_C (time for computations) and T_D (time for data transmissions both within the VLSI system and between the host and the VLSI system).

One may expect that by using a very high bandwidth B and a large number of processors P, we can make T_C and T_D quite small. So, T_C and T_D are not in themselves a very good measure of the effectiveness with which the resources B and P have been used. Let D denote the total amount of data that needs to be transmitted between the host and VLSI system. The ratio

$$R_D = B * T_D / D$$

measures the effectiveness with which the bandwidth B has been used. Clearly, $R_D \geq 1$ for every VLSI design.

Let C denote the time spent for computation by a single processor algorithm. The ratio

$$R_C = P * T_C / C$$

measures the effectiveness of processor utilization. Once again, we see that $R_C \geq 1$ for every VLSI design.

In evaluating a VLSI design, we shall be concerned with T_C and T_D and also with R_C and R_D. We would like R_C and R_D to be close to 1. Finally, we may combine the two efficiency ratios R_C and R_D into the single ratio $R = R_C * R_D$. A design that makes effective use of the available bandwidth and processors will have R close to 1.

The efficiency measure R as defined here is the same as that used in {1} to evaluate VLSI designs for matrix multiplication. This measure is also quite similar to that proposed in {3}. In fact, the two measures become identical when $T_C = T_D$.

For each of the designs considered in this paper, we compute R_C, R_D and R. In several cases, our designs have improved efficiency ratios than all earlier designs using the same model. In comparing different architectures for the same problem, one must be wary about over emphasizing the importance of R_C, R_D and R. Clearly, using $P = 1$ and $B = 1$, we can get $R_C = R_D = R = 1$ and no speed up at all. So, we are really interested in minimizing T_C and T_D while keeping R close to 1. Some of our designs reduce T_C at the expense of R.

2. BACK SUBSTITUTION

2.1. The Problem

Input:	A lower triangular matrix A and an $n \times 1$ vector b.
Output:	An $n \times 1$ vector x such that $Ax = b$.
Parameters:	$C = n(n+1)/2 \sim n^2/2$, $D = n(n+5)/2 \sim n^2/2$.

2.2. Chain with O(n) Bandwidth

An n PE chain may be used as in Figure 1 (dashes indicate no input). Each PE has one adder, multiplier and divider. PE(j) computes x_j. Row j of A (excluding the elements beyond the main diagonal) is input to PE(j) element by element beginning at time j. PE(j) inputs b_j at time j. The computed x values flow to the right and get output from the right end of the chain. A formal description of the algorithm executed by each PE is given in Algorithm 1. In this algorithm, the notation Y_i, $Y \in \{ c, x, A \}$ means register Y of PE(i). The correctness of this algorithm is readily established.

From Figure 1 and Algorithm 1, we see that maximum input/output occurs when $i = n$. At this time, a bandwidth of $\lceil n/2 \rceil + 1$ is needed. The performance figures for this scheme are $P = n$, $B = \lceil n/2 \rceil + 1$, $T_C = 2n - 1$, $T_D = 2n$, $R_C \sim 4$, $R_D \sim 2$ and $R \sim 8$.

* This research was supported in part by the National Science Foundation under grant MCS-83-05567.
+ Author's current address is: Computer Science Department, University of Houston, Houston, Texas, USA.

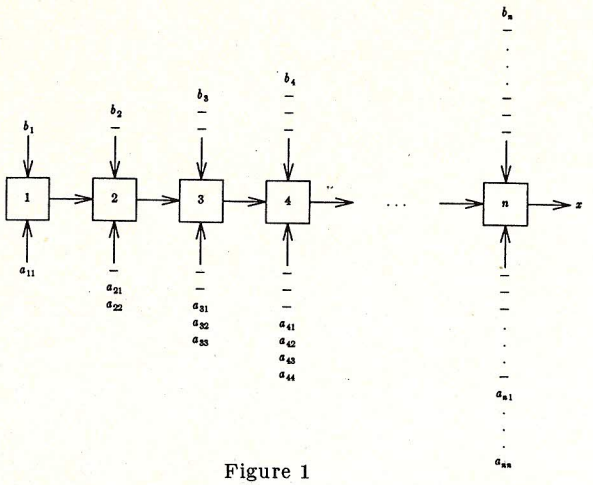


Figure 1

```
for i ←1 to 2n − 1 do
    do in parallel    {Input/Output}
        if i ≤ n then c_i ← b_i
                 else output x_n
        A_j ← a_{j, i−j+1}, (i + 1)/2 ≤ j ≤ min {i, n}
        x_j ← x_{j−1},          2 ≤ j ≤ n
    end
    do in parallel
        if i odd then [ x_{⌈i/2⌉} ← c_{⌈i/2⌉} / A_{⌈i/2⌉}
                        c_j ← c_j − A_j *x_j, j ≠ ⌈i/2⌉ ]
                 else [ c_j ← c_j − A_j *x_j, 1 ≤ j ≤ n ]
    end
end
output x_n
```

Algorithm 1

2.3. Ring with O(n) Bandwidth

An examination of Algorithm 1 reveals that at most $\lceil n/2 \rceil$ PEs do useful work at any time. Hence, we may double up the use of the first $\lceil n/2 \rceil$ PEs by requiring PE(i) to compute both x_i and $x_{i+\lceil n/2 \rceil}$. In case n is odd, PE($\lceil n/2 \rceil$) computes only $x_{\lceil n/2 \rceil}$. For simplicity in exposition, we assume that n is even (in case n is odd, an extra column and row may be added to A and an extra value added to b).

The input data pattern is shown in Figure 2 and the algorithm executed by each PE is given in Algorithm 2. Once again, correctness is easily established. The performance figures are $P = \lceil n/2 \rceil$, $B = \lceil n/2 \rceil + 1$, $T_C = 2n - 1$, $T_D = 2n$, $R_C \sim 2$, $R_D \sim 2$ and $R \sim 4$. So even though only half as many PEs are used, T_C and T_D are unchanged!

```
for i ←1 to 2n − 1 do
    do in parallel    {Input/Output}
        if i ≤ n then c_{(i−1) mod n/2+1} ← b_i
        A_j ← a_{j, i−j+1}, A_j ← a_{j+n/2, i−j−n/2+1},
                (i+1)/2 ≤ j+n/2 ≤ min {i, n}
        case
            1 < i ≤ n/2   : x_j ← x_{j−1},   2 ≤ j < n/2
            n/2 < i ≤ n   : x_j ← x_{j−1},   2 ≤ j < n/2
                            x_1 ← x_{n/2}
            i > n         : x_j ← x_{j−1},   2 ≤ j < n/2
                            output x_{n/2}
        endcase
    end
    do in parallel
        if i odd then [ k ←(⌈i/2⌉ − 1) mod n/2 + 1
                        x_k ← c_k / A_k
                        c_j ← c_j − A_j *x_j, j != k ]
                 else [ c_j ← c_j − A_j *x_j, 1 ≤ j ≤ n/2 ]
    end
end
output x_{n/2}
```

Algorithm 2

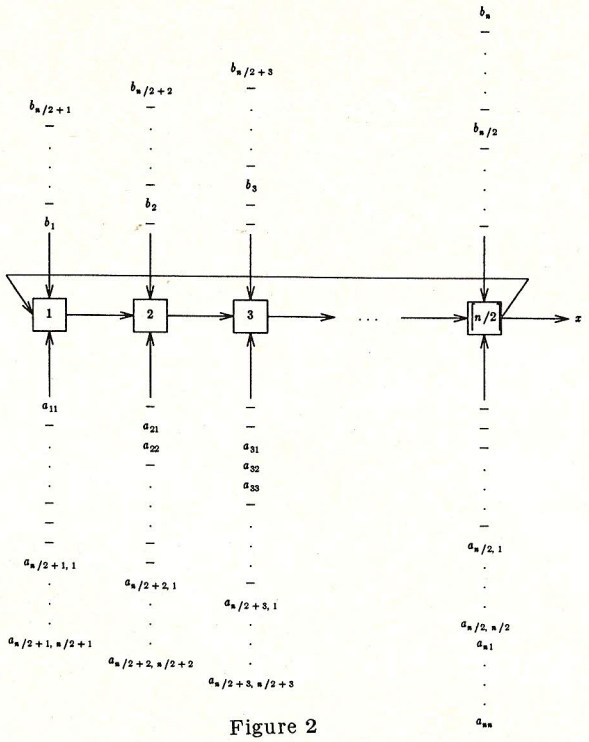


Figure 2

2.4. Bidirectional Chain With O(n) Bandwidth

Kung {4} has proposed an n PE bidirectional chain architecture for the back substitution problem. The performance figures for this architecture are: $P = \lceil n/2 \rceil$, $B = \lceil n/2 \rceil + 2$, $T_C = 2n - 1$, $T_D = 2n$, $R_C \sim 2$, $R_D \sim 2$ and $R \sim 4$.

It is possible to improve the throughput of the bidirectional chain by providing each PE with its own divider that can function in parallel with the rest of the PE. Our design requires n PEs and is as shown in Figure 3. The output is generated from the middle PE. The PEs to the left of the middle PE compute all but the last term needed for the even x_is. Similarly, the PEs on the right compute all terms but the last needed for the odd x_is.

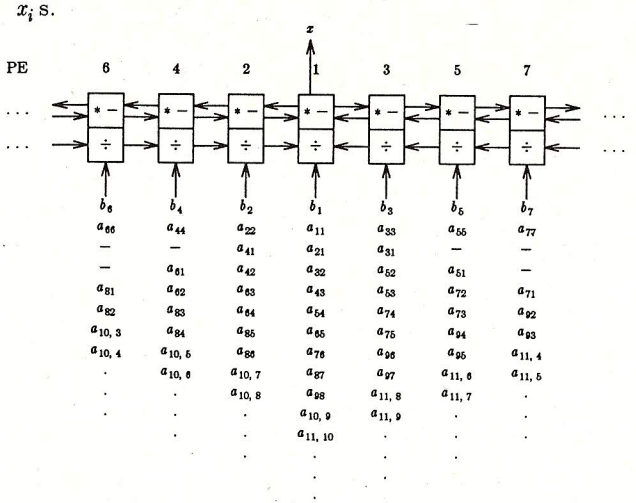


Figure 3

The input data pattern is quite regular. Assume that the PEs are indexed as in Figure 3 with the middle PE given the index 1. The first input to PE(i) is b_i and the second a_{ii}. Let $A(i,j)$ denote the $(j+2)$th input to PE(i). Then, $A(1,1) = a_{21}$, $A(1,2) = a_{32}$, $A(1,3) = a_{43}$, $A(2,1) = a_{41}$, $A(5,1) = 0$, etc. Formally, we have:

$$A(i,j)=\begin{cases} a_{j+1,j}, & i=1 \\ a_{i+2\lceil (j+1-\lceil i/2\rceil)/2\rceil,\, j+1-\lfloor i/2\rfloor}, & i>1 \end{cases}$$

where $a_{ik}=0$ for $k<1$.

Each x_i is computed using the formula

$$x_i = \frac{b_i}{a_{ii}} - \frac{a_{i1}}{a_{ii}}x_1 - \frac{a_{i2}}{a_{ii}}x_2 - \cdots - \frac{a_{i,i-1}}{a_{ii}}x_{i-1} \qquad (1)$$

Each term of the form $\frac{a_{ij}}{a_{ii}}x_j$ is obtained by computing a_{ij}/a_{ii} in one cycle and then multiplying by x_j in the next cycle.

The c registers hold the partial sums for (1); the Z registers hold terms of the form a_{ij}/a_{ii}; the D registers hold the divisors a_{ii}; and the e registers hold a single term of the form $\frac{-a_{ij}}{a_{ii}}x_j$. The e registers are needed as, often, a PE computes two terms of an x_i. The first term is saved in the e register and then combined on the next cycle with the second term and the contents of the c register. The functioning of the systolic system is described formally in Algorithm 3.

```
zero all registers
c_i ← b_i ,            1 ≤ i ≤ n
D_i ← a_ii ,           1 ≤ i ≤ n
c_i ← c_i / D_i ,      1 ≤ i ≤ n
{ Let U = {3, 4, 7, 8, 11, 12, ...},
  V = {2, 5, 6, 9, 10, 13, 14, ...} }
for j ←1 to n do
    do in parallel    {Input/Output}
        A_1 ← a_{j+1, j}
        A_i ← a_{i+2⌈(j+1-⌈i/2⌉)/2⌉, j+1-⌊i/2⌋},
                  i > 1
        x_2 ← x_1 ,          j ≥ 2
        x_i ← x_{i-2} ,      i ≥ 3,  j ≥ 2
        output x_1 ,         j ≥ 2
        if j odd then [ c_1 ← c_3,  j ≥ 3 ; D_1 ← D_2 ]
                  else [ c_1 ← c_2; D_1 ← D_3 ]
        c_{i-2} ← c_i ,      i ≥ 4,  ⌊(i+1)/2⌋ < j
        D_{i-2} ← D_i ,      i ≥ 4,  ⌊(i-1)/2⌋ ≤ j
    end
    do in parallel
        x_1 ← c_1 - Z_1 * x_1
        if j odd then [c_i ← (c_i + e_i) - Z_i * x_i ,
                          i ∈ V and ⌊i/2⌋ < j
                       e_i ← - Z_i * x_i ,
                          i ∈ U and ⌊(i+1)/2⌋ < j]
                  else [c_i ← (c_i + e_i) - Z_i * x_i ,
                          i ∈ U and ⌊i/2⌋ < j
                       e_i ← - Z_i * x_i ,
                          i ∈ V and ⌊(i+1)/2⌋ < j]
        Z_i ← A_i / D_i ,    1 ≤ i ≤ n
    end
end
output x_1
```

Algorithm 3

From Algorithm 3 and Figure 3, we see that $P=n$, $B=n+1$, $T_C=n+1$, $T_D=n+3$, $R_C\sim 2$, $R_D\sim 2$ and $R\sim 4$.

The throughput cannot be further improved using formula (1), as to compute x_i, we need to know x_{i-1}. Hence x_i can be computed, at best, one cycle after x_{i-1} has been computed. We can bring T_C down to $o(n/2)$ by computing two x_is each cycle using the formulae:

$$x_i = \frac{b_i}{a_{ii}} - \sum_{j=1}^{i-2}\frac{a_{ij}}{a_{ii}}x_j - \frac{a_{i,i-1}}{a_{ii}}x_{i-1} \qquad (2)$$

$$= \frac{b_i}{a_{ii}} - \frac{a_{i,i-1}}{a_{ii}\,a_{i-1,i-1}}b_{i-1} - \sum_{j=1}^{i-2}\left(\frac{a_{ij}}{a_{ii}} - \frac{a_{i,i-1}\,a_{i-1,j}}{a_{ii}\,a_{i-1,i-1}}\right)x_j$$

for $i>1$ and odd, and

$$x_i = \frac{b_i}{a_{ii}} - \frac{a_{i,i-1}}{a_{ii}\,a_{i-1,i-1}}b_{i-1} \qquad (3)$$

$$- \sum_{j=1}^{i-3}\left(\frac{a_{ij}}{a_{ii}} - \frac{a_{i,i-1}\,a_{i-1,j}}{a_{ii}\,a_{i-1,i-1}}\right)x_j - \left(\frac{a_{i,i-2}}{a_{ii}} - \frac{a_{i,i-1}\,a_{i-1,i-2}}{a_{ii}\,a_{i-1,i-1}}\right)x_{i-2}$$

$$= \frac{b_i}{a_{ii}} - \frac{a_{i,i-1}}{a_{ii}\,a_{i-1,i-1}}b_{i-1} - \frac{Z_i\,b_{i-2}}{a_{i-2,i-2}} - \sum_{j=1}^{i-3}\left(\frac{a_{ij}}{a_{ii}} - \frac{a_{i,i-1}\,a_{i-1,j}}{a_{ii}\,a_{i-1,i-1}} - \frac{Z_i\,a_{i-2,j}}{a_{i-2,i-2}}\right)x_j$$

where $Z_i = \left(\frac{a_{i,i-2}}{a_{ii}} - \frac{a_{i,i-1}\,a_{i-1,i-2}}{a_{ii}\,a_{i-1,i-1}}\right)$ for $i>2$ and even.

First, x_1 is computed as $x_1 = b_1/a_{11}$ and x_2 as $x_2 = \frac{b_2}{a_{22}} - \frac{a_{21}\,b_1}{a_{11}}$. Once x_1 and x_2 have been computed, x_3 and x_4 can be computed. To compute x_3, x_1 is needed and to compute x_4, x_1 is needed. So both x_3 and x_4 can be output one cycle after x_1 and x_2. In the meantime, x_5 and x_6 can start computation using x_1 and x_2. In the next cycle, the computation of x_5 and x_6 can be completed using x_3 from the previous cycle. The VLSI system that incorporates this uses more hardware than ure 2.4 and is quite a bit more complex. We shall not present the details here. We note that the method may be extended to get a T_C of $o(n/k)$ for any fixed k.

2.5. Broadcast Chain With O(n) Bandwidth

The architecture and data pattern are shown in Figure 4. The algorithm used is given in Algorithm 4. The performance figures for the broadcast chain are $P=n$, $B=n$, $T_C=n$, $T_D=n+2$, $R_C\sim 2$, $R_D\sim 2$ and $R\sim 4$.

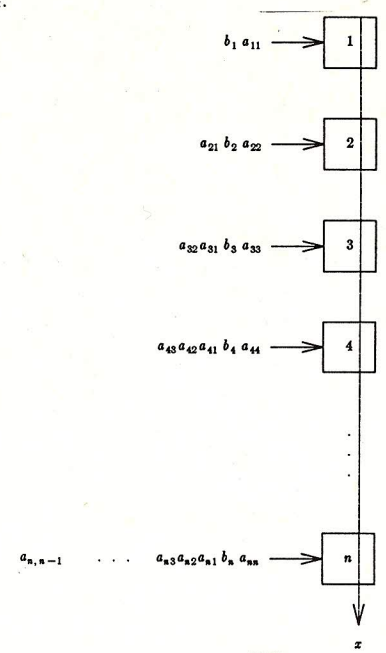


Figure 4

One should note that each computation of Figure 4 takes $t_{SUB}+t_{MUL}+t_{DIV}$ time, while in all of our previous designs, a computation required only $\max\{t_{DIV}, t_{MUL}+t_{SUB}\}$ where t_{DIV} is the time to divide, t_{MUL} is the time to multiply, and t_{SUB} is the time to subtract. The performance of the broadcast chain may be improved slightly by overlapping divisions with other operations. The new design is as in Figure 5. Equation (1) is used to compute the x_is. The dividers are loaded with a_{ii}. Each input to PE(i) is divided by a_{ii} at the divider and then transmitted to the remainder

```
D_j ← a_jj      {input}
c_j ← b_j       {input}
x_1 ← c_1 / D_1
for i ← 1 to n − 1 do
    do in parallel
        broadcast x_i to x_j ,     j > i
        output x_i
        A_j ← a_ji ,               j > i    {input}
    end
    do in parallel
        x_j ← (c_j − A_j * x_j)/D_j ,    j = i + 1
        c_j ← c_j − A_j * x_j ,          j > i + 1
    end
end
output x_n
```

Algorithm 4

of the PE. The divisions take place in parallel with a multiply and subtract operation in the same PE. As a result of this, each computation step takes $\max\{t_{DIV}, t_{SUB} + t_{MUL}\}$ when Figure 5 is used rather than $t_{DIV} + t_{SUB} + t_{MUL}$. The performance figures using Figure 5 are $P = n$, $B = n$, $T_C = n + 1$, $T_D = n + 3$, $R_C \sim 2$, $R_D \sim 2$ and $R \sim 4$.

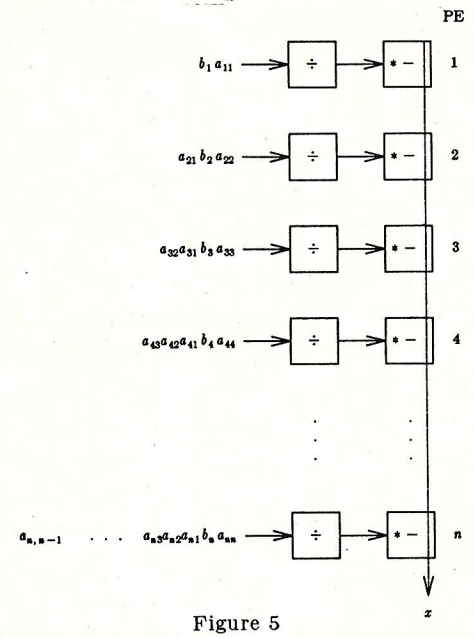


Figure 5

The R value of both the preceding designs may be reduced to $\sim 9/4$ by using half as many PEs. For example, the design of Figure 4 translates into that of Figure 6. The $n/2$ PE broadcast chain is first used to compute $x_1, x_2, \ldots, x_{n/2}$ in exactly the same way as Figure 4 is used when A is an $n/2 \times n/2$ matrix. Next, the remaining $n/2$ xs are computed by having the $n/2$ PEs behave like the last $n/2$ PEs of Figure 4 with the required x_is being re-input and broadcast. To compute the first $n/2$ xs, $n/2$ compute steps and $n/2 + 2$ data move steps are needed. For the remaining $n/2$ xs, an additional n compute steps and $n + 2$ data move steps are needed. Hence, $P = n/2$, $B = n/2 + 1$, $T_C = 3n/2$, $T_D = 3n/2 + 4$, $R_C \sim 3/2$, $R_D \sim 3/2$ and $R \sim 9/4$. This reduction in the R value has been obtained at the expense of T_C and T_D which are now both 50% larger than before.

This idea may be extended to the case of n/k PEs for k any constant. For example when $k = 4$, we get $P = n/4$, $B = n/4 + 1$, $T_C = 5n/2$, $T_D = 5n/2 + 8$, $R_C \sim 5/4$, $R_D \sim 5/4$ and $R \sim 25/16$. A further reduction in R has occurred at the expense of T_C and T_D! Certainly, when $k = n$, $P = 1$ and the scheme becomes the normal one processor scheme with $R = 1$.

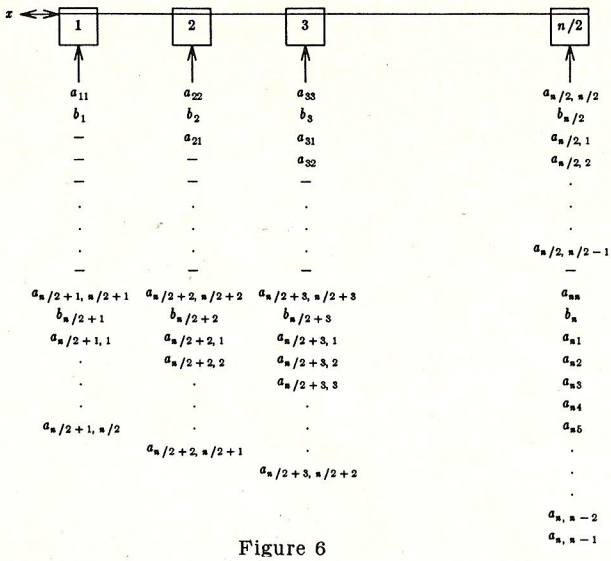


Figure 6

2.6. $o(n)$ Throughput With Unidirectional Data Flow

The unidirectional chain of Figure 1 cannot be modified to produce the n x_is in $o(n)$ time. This is because it takes $o(n)$ time for x_1 to reach PE(n) and another $o(n)$ time to compute x_n using one PE. While we have seen several architectures that provide for $o(n)$ computation of the x_is, these either employed a broadcast capability or required data to flow in two directions. Neither of these capabilities is necessary for this.

We may compute the x_is in $o(n)$ time using $2n - 2$ PEs as in Figure 7. One PE is assigned to the computation of x_1 and another to that of x_2. Each of the remaining x_is is computed using two PEs. The upper PE of each such pair computes the terms involving x_j for i odd while the lower PE computes the even terms. Predivision of the a_{ij}s by a_{ii} as suggested by formula (1) is done by each PE. The upper PE in the pair for x_i computes $U_i = \dfrac{b_i}{a_{ii}} - \sum_{\substack{1 \le j < i \\ j\ odd}} \dfrac{a_{ij}}{a_{ii}} x_j$ when i is even and $U_i = \sum_{\substack{1 \le j < i \\ j\ odd}} \dfrac{a_{ij}}{a_{ii}} x_j$ when i is odd. The lower PE computes $L_i = \dfrac{b_i}{a_{ii}} - \sum_{\substack{1 \le j < i \\ j\ even}} \dfrac{a_{ij}}{a_{ii}} x_j$ when i is odd and $L_i = \sum_{\substack{1 \le j < i \\ j\ even}} \dfrac{a_{ij}}{a_{ii}} x_j$ when i is even. L_i and U_i are combined in the lower (upper) PE when i is odd (even). This involves a subtraction and this subtraction is performed in the same cycle as the multiplication of $\dfrac{a_{i,i-1}}{a_{ii}}$ and x_{i-1}. Hence, the cycle time is $\max\{t_{DIV}, \max\{t_{MUL}, t_{SUB}\} + t_{SUB}\}$ (assuming the add time to be roughly equal to the subtract time). The working of the systolic system is described formally in {9}.

As drawn, the design of Figure 7 has a bandwidth of $2n$. This is easily reduced to n by delaying the inputs of both the upper row of a_{ii}s and the row following the lower row of a_{ii}s by one time unit. This is equivalent to inserting one row of dashes ("–") before row 2 of the upper input and after row 2 of the lower input. The performance figures for this design are $P = 2n - 2$, $B = n$, $T_C = n + 1$, $T_D = n + 4$, $R_C \sim 4$, $R_D \sim 2$ and $R \sim 8$.

2.7. O(1) Bandwidth Designs

Horowitz {2} presents an n PE chain with O(1) bandwidth for the back substitution problem. While his original design employs $2n$ PEs, n of these are used solely for input and may be eliminated. Figure 8(a) shows the input pattern for the case $n = 4$ while Figure

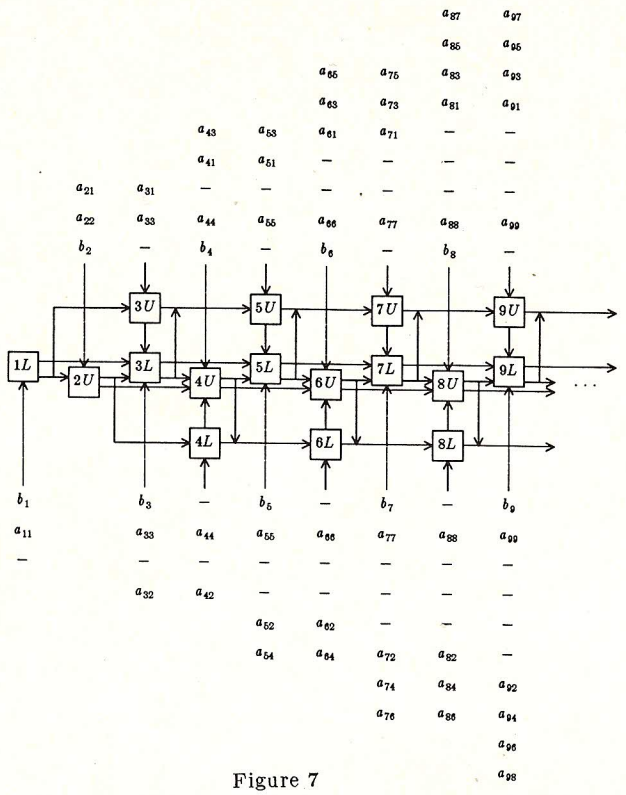
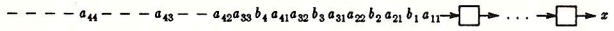
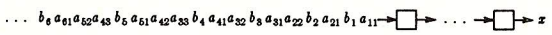


Figure 7

8(b) shows the general input pattern. PE(i) computes x_i as well as all terms that involve x_i. Hence PE(i) computes $[\, b_j - \sum_{k<i} a_{jk}\, x_k \,] - a_{ji}\, x_i$ for $j > i$. The term $[\, b_j - \sum_{k<i} a_{jk}\, x_k \,]$ is transmitted to PE(i) from PE($i-1$) and PE(i) in turn transmits $[\, b_j - \sum_{k<i} a_{jk}\, x_k \,] - a_{ji}\, x_i$ to PE($i+1$). The formal algorithm is given in Algorithm 5. The performance figures for the design of {2} are $P = n$, $B = 1$, $T_C = 2n - 1$, $T_D = n(n+3) - 1$, $R_C \sim 4$, $R_D \sim 2$ and $R \sim 8$.



(a)



(b)

Figure 8

```
for i ←1 to 2n − 1 do
    for k ←1 to ⌈ i/2 ⌉ do
        do in parallel
            A_1 ← a_{⌊ i/2 ⌋ + k, ⌈ i/2 ⌉ − k + 1},   {input}
            A_j ← A_{j−1},                          2 ≤ j ≤ k
        end
    end
    do in parallel
        if i ≤ n then c_1 ← b_i,                    {input}
        c_j ← c_{j−1},                              2 ≤ j ≤ ⌈ i/2 ⌉
    end
    do in parallel
        if i odd then [ x_{⌈ i/2 ⌉} ← c_{⌈ i/2 ⌉} / A_{⌈ i/2 ⌉}
                        c_j ← c_j − A_j * x_j,  j ≠ ⌈ i/2 ⌉ ]
                 else [ c_j ← c_j − A_j * x_j,
                        1 ≤ j ≤ ⌈ i/2 ⌉ ]
    end
end
output x_1,...,x_n
```

Algorithm 5

Improved performance can be obtained using a somewhat simpler algorithm, Algorithm 6, and the data pattern of Figure 9. This time the chain is a bidirectional chain. The number of PEs used is $n - 1$. The performance figures for this design are $P = n - 1$, $B = 1$, $T_C = 2n - 1$, $T_D = n(n+5)/2$, $R_C \sim 4$, $R_D \sim 1$ and $R \sim 4$.

```
c_1 ← b_1                          {input}
A_1 ← a_11                         {input}
x_1 ← c_1 / A_1
output x_1
for i ←2 to n do
    do in parallel
        c_1 ← b_{n−i+2},           {input}
        c_j ← c_{j−1},             2 ≤ j < n
    end
end
for i ←1 to n − 1 do
    for k ←1 to n − i do
        do in parallel
            A_1 ← a_{n−k+1, i},    {input}
            A_j ← A_{j−1},         2≤ j ≤n−i
            x_j ← x_{j−1},         2≤ j ≤n−i
        end
    end
    c_j ← c_j − A_j * x_j,         2 ≤ j ≤ n − i
    A_1 ← a_{i+1, i+1},            {input}
    x_1 ← c_1 / A_1
    do in parallel
        c_j ← c_{j+1},             1 ≤ j ≤ n − i − 1
        output x_1
    end
end
```

Algorithm 6

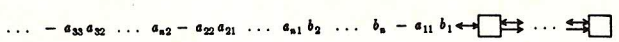


Figure 9

Further improvement may be obtained by simulating the $\lceil n/2 \rceil$ PE version of the bidirectional chain algorithm of Section 2.4. The simulation involves sending all the input needed for the next computation rightwards from PE(1) and then performing the computation in all PEs simultaneously. P, T_C and R_C are unchanged. However, now, $B = 1$, $T_D = n(n+5)/2$, $R_D = 1$ and $R \sim 2$.

An O(1) bandwidth bidirectional chain with improved throughput can also be obtained by simulating the improved throughput design of Section 2.4. The input for each computation step is provided through the middle PE. When this is done, we get $P = n$, $B = 1$, $T_C = n + 2$, $T_D = n(n+5)/2$, $R_C \sim 2$, $R_D = 1$ and $R \sim 2$.

2.8. Summary

Our designs are the first VLSI systems that require less than $o(2n)$ computational steps. This has been accomplished without sacrificing on R. In fact, the R value of our O(1) bandwidth design for the case of $T_C = o(n)$ is half that of the best designs with $T_C = o(2n)$.

Finally, we note that the comparisons among the different designs are not entirely fair as some designs require each PE to have a multiplier, a divider, and a subtracter; while other designs require two kinds of PEs: one with a multiplier and a subtracter and the other with just a divider.

3. REFERENCES

{1} K.H. Cheng and S. Sahni, *VLSI Systems For Matrix Multiplication,* Foundations of software technology and theoretical computer science, Lecture Notes in Computer Science, Vol 206, Springer Verlag, 1985.

{2} E. Horowitz, *VLSI architectures for matrix computations,* IEEE International Conference On Parallel Processing, 1979, pp. 124-127.

{3} K.H. Huang and J.A. Abraham, *Efficient parallel algorithms for processor arrays,* IEEE International Conference On Parallel Processing, 1982, pp. 271-279.

{4} H.T. Kung and C.E. Leiserson, *Systolic arrays for VLSI,* Department of Computer Science, Carnegie-Mellon University, April 1978.

{5} H.T. Kung, *Let's design algorithms for VLSI systems,* Proceedings CALTECH Conference on VLSI, Jan. 1979, pp. 65-90.

{6} H.T. Kung, *A Listing of Systolic Papers,* Department of Computer Science, Carnegie-Mellon University, May 1984.

{7} H.T. Kung and M. Lam, *Wafer scale integration and two level pipelined implementations of systolic arrays,* Journal of Parallel and Distributed Processing, Vol. 1, #1, 1984

{8} C.E. Leiserson, *Area-Efficient VLSI Computation,* MIT Press, 1983.

{9} K.H. Cheng and S. Sahni, *VLSI archtectures for back substitution,* University of Minnesota, Technical Report 84-30, 1984.