Systolic convolution of arithmetic functions

Patrice Quinton, Yves Robert

Systolic convolution of arithmetic functions

Patrice Quinton, Yves Robert

UNITE DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 976

Programme 2

SYSTOLIC CONVOLUTION OF ARITHMETIC FUNCTIONS

Patrice QUINTON
Yves ROBERT

Février 1989

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

Convolution systolique de fonctions arithmétiques

Publication Interne n° 449 - Janvier 1989 - 30 Pages

Patrice QUINTON+ et Yves ROBERT++

+ IRISA
Campus de Beaulieu
35042 Rennes Cedex, France

++ LIP-IMAG
Ecole Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France

Résumé

Etant données deux fonctions arithmétiques f et g, leur convolution $h = f * g$ est définie par $h(n) = \sum_{kl=n,\ 1\le k,l\le n} f(k)\, g(l)$ pour tout $n\ge1$. Etant données deux fonctions arithmétiques g et h, le problème de la convolution inverse est de trouver f telle que $f * g = h$.

Dans ce rapport, on propose deux architectures systoliques linéaires pour le calcul en temps réel de la convolution et de la convolution inverse. Ces architectures étendent la solution de Verhoeff pour le calcul de la fonction de Möbius μ, définie comme la solution du problème de convolution inverse $\mu * g = \Delta$, où $g(n) = 1$ pour tout $n\ge1$ et $\Delta(n) = 1$ si $n\neq1$, $\Delta(n) = 0$ si $n>1$.

Systolic convolution of arithmetic functions

Abstract

Given two arithmetic functions f and g, their convolution $h = f * g$ is defined as $h(n) = \sum_{kl=n,\ 1\le k,l\le n} f(k)\, g(l)$ for all $n\ge1$. Given two arithmetic functions g and h, the inverse convolution problem is to determine f such that $f * g = h$.

In this paper, we propose linear arrays for the real-time computation of the convolution and the inverse convolution problem. These arrays extend the design of Verhoeff for the computation of the Möbius function μ, defined as the solution of the inverse convolution problem $\mu * g = \Delta$, where $g(n) = 1$ for all $n\ge1$ and $\Delta(n) = 1$ if $n\neq1$, $\Delta(n) = 0$ if $n>1$.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Systolic convolution of arithmetic functions

Patrice QUINTON+ et Yves ROBERT++

+ IRISA
Campus de Beaulieu
35042 Rennes Cedex, France

++ LIP-IMAG
Ecole Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, France

Abstract

Given two arithmetic functions f and g, their convolution $h = f * g$ is defined as $h(n) = \Sigma_{kl=n,\ 1\le k,l\le n} f(k)\, g(l)$ for all $n\ge 1$. Given two arithmetic functions g and h, the inverse convolution problem is to determine f such that $f * g = h$.

In this paper, we propose two linear arrays for the real-time computation of the convolution and the inverse convolution problem. These arrays extend the design of Verhoeff for the computation of the Möbius function μ, defined as the solution of the inverse convolution problem $\mu * g = \Delta$, where $g(n) = 1$ for all $n\ge 1$ and $\Delta(n) = 1$ if $n\neq 1$, $\Delta(n) = 0$ if $n>1$.

This work has been supported by the Coordinated Research Program C3 of CNRS

1. Introduction

Given two arithmetic functions f and g, their convolution $h = f * g$ is defined as $h(n) = \Sigma_{kl=n,\ 1\le k,l\le n}\ f(k)\ g(l)$ for all $n\ge 1$. Given two arithmetic functions g and h, the inverse convolution problem is to determine f such that $f * g = h$. The inverse convolution problem is not always solvable: see [Ken] for a review.

For instance, let $E_0(n) = 1$ for all $n\ge 1$ and $\Delta(n) = 1$ if $n\neq 1$, $\Delta(n) = 0$ if $n>1$. The Möbius function μ is defined as

$\mu(n) = 1$ if $n= 1$
$\mu(n) = (-1)^r$ if n is the product of r distinct primes
$\mu(n) = 0$ if n is divisible by a prime square

The Möbius function can be more easily computed using the Möbius inversion formula [Ken]

$$\Sigma_{1\le d\le n,\ d|n}\ \mu(d) = \Delta(n)$$

which states that μ is the solution to the inverse convolution problem $\mu * E_0 = \Delta$.

Verhoeff [Ver] proposes a systolic linear array for the real-time computation of the Möbius function μ. In this paper, we propose two systolic architectures for the real-time computation of the convolution and the inverse convolution of general arithmetic functions.

Kung [Kun81] introduces a systolic linear array for the computation of the sequence

$$h(n) = \Sigma_{k+l=n,\ 1\le k,l}\ f(k)\ g(l),\ n \ge 1$$

where $(f(k);\ k\ge 1)$ and $(g(l);\ l\ge 1)$ are two given sequences. Such a computation corresponds to the product of two polynomials, or series, of coefficients f(k) and g(l).

Informally, if we want to compute $h(n) = \Sigma_{d|n}\ f(d)$, we can use a design similar to that of [Kun81]. The problem is to inhibit the operation of the cells when they receive a pair of inputs (f(d),h(n)) where d is not a divisor of n, and this is the key to Verhoeff's design [Ver].

The general convolution problem $h = f * g$ is more difficult for two reasons:
- first we have to ensure that every component of f can meet every component of g
- when a cell receive a pair of inputs (f(d),h(n)) where d is not a divisor of n, its operation has still to be inhibited. But when d is a divisor of n, we have to organize the flow of g such that g(n/d) is also an input to the cell.

In this paper, we first propose a linear systolic architecture of O(n) cells which solves the problem of computing $(h(m);\ 1\le m\le n)$ in time O(n). This first solution is obtained using the dependence mapping method. The space-time complexity of the proposed architecture is $O(n^2 \log n)$. Then we propose another systolic architecture of only $O(n^{1/2})$ processing cells which also solves the problem in time O(n). This second architecture requires O(n log n) delay cells, leading to the same space-time complexity as for the first solution. Both architectures can be extended to the solution of the inverse convolution problem, with the same performances.

Throughout the paper, we assume the reader to be familiar with the systolic model. Systolic arrays have been introduced by Kung and Leiserson [KL] and consist of a large number of elementary processors (or cells) which are mesh-interconnected in a regular and modular way, and achieve high performance through extensive concurrent and pipeline use of the cells. We refer the reader to [Kun82] for a general presentation of the systolic model.

2. A first linear systolic array

The arithmetic convolution consists in the computation of the following equation :

$$\forall\, n \geq 1,\ h(n) = \Sigma_{kl=n,\ 1\leq k,l\leq n}\ f(k)\ g(l) \qquad (1)$$

where f(k), k≥1 and g(l), l≥ 1 are given integer functions.

Systolic arrays that compute the convolution

$$\forall\, n \geq 1,\ h(n) = \Sigma_{k=1..n}\ f(k).g(n-k+1) \qquad (2)$$

are well known (see for example [Kun82]). From the bidirectional design of Kung, one can easily derive a systolic array for the *unbounded convolution*, that is the convolution whose equation is

$$\forall\, n \geq 1,\ h(n) = \Sigma_{k=1..n}\ f(k).g(n-k+1) \qquad (3)$$

The only difficulty is the loading of the coefficients g, as one cannot assume that they are preloaded in the array, as in the case of the bounded convolution.

The arithmetic convolution (1) is much harder, as the summation has only to be done for product terms f(k).g(l) such that kl=n.

In the following, we shall derive and prove the correctness of a systolic array which has the following characteristics :

- the array is linear and bidirectionnal. The number of cells needed for the computation of h(m), $1 \leq m \leq n$ is n
- the period of the array is 3, that is, each cell works every three cycle.

We shall derive the design using a space-time transformation of a dependence graph of the algorithm, following the approach of Moldovan [Mol] and Quinton [Qui]. In section 2.1, we recall the principles of the dependence mapping method on the example of the unbounded convolution. In particular, we explain how the coefficients g(l) can be loaded in the cells. In section 2.2, we develop the arithmetic convolution.

2.1. Unbounded convolution

Figure 1 depicts a dependance graph for the unbounded convolution. For the moment, we ignore the loading of the g coefficients. On this diagram, a coefficient h(n) is computed along a diagonal line, by summing up the product terms f(k).g(n-k) in increasing order of k. The result h(n) is obtained at point (n,1). Deriving a systolic array from this dependence graph can be done easily using a space-time linear transformation (see [Mol,Qui] among others). First, we seek an affine (integral) schedule of the computations, that is a function t(n,m) such that if calculation at point

(n_1,m_1) depends on calculation at point (n_2,m_2), then $t(n_1,m_1) > t(n_2,m_2)$. Denote $t(n,m) = \lambda_1 n + \lambda_2 m + \alpha$, where λ_1, λ_2, and α belong to **Z**, the set of integers. Let us call *dependence vector* the quantity (n_1-n_2,m_1-m_2) for two dependent points. As the number of dependence vectors in this graph is finite (and independent of n), this condition amounts to a finite number of linear inequalities involving these vectors.

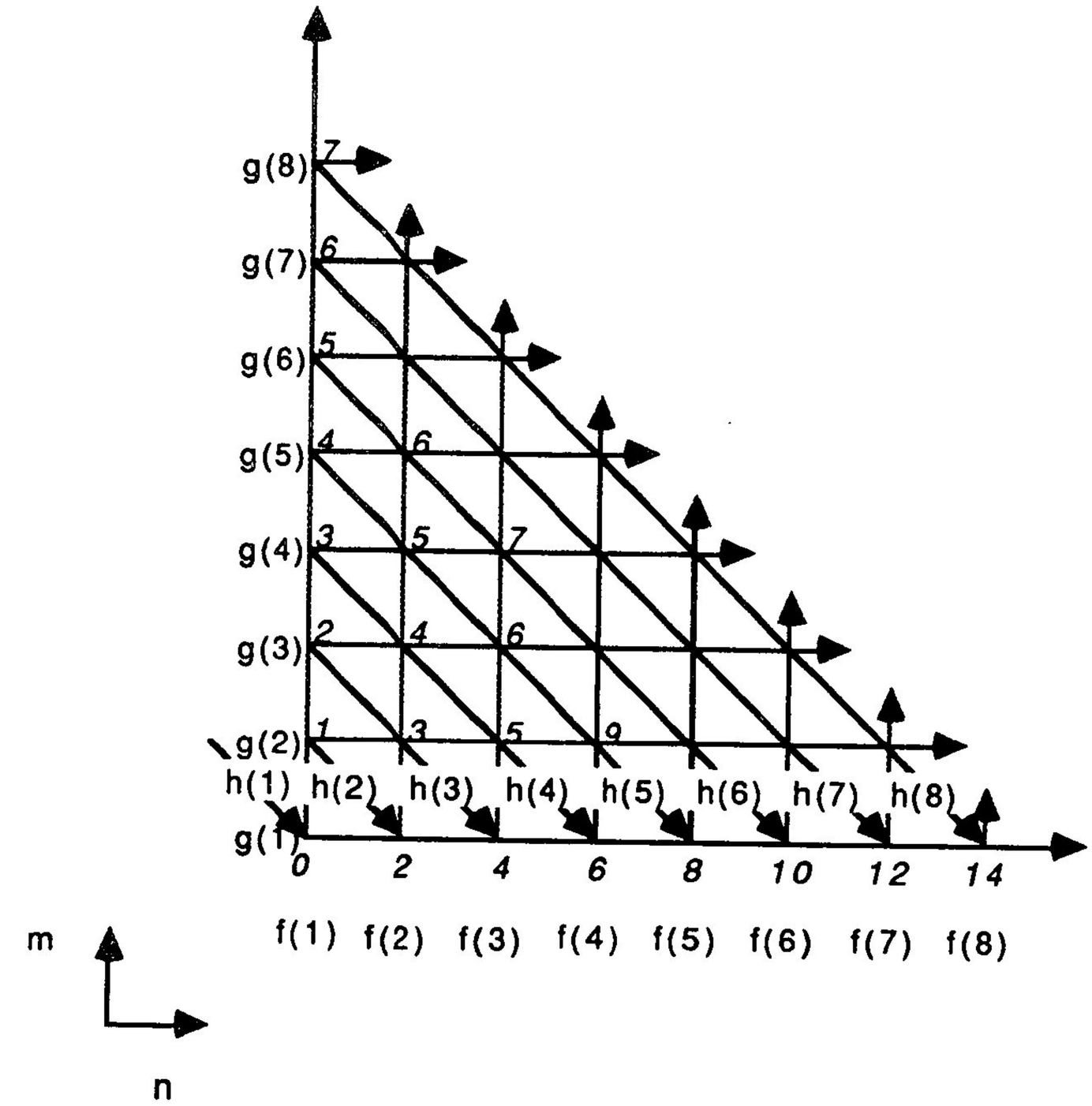


Figure 1 : Dependence graph and timing-function for the convolution

The set of possible dependence vectors is :

$$\Delta = \{ (0,1), (1,-1), (1,0) \}$$

We must therefore have :

$$\lambda_2 \geq 1; \lambda_1 - \lambda_2 \geq 1; \lambda_1 \geq 1$$

These conditions are met optimally with $\lambda_1 = 2$ and $\lambda_2 = 1$. The coefficient α can be chosen in such a way that the computation starts at time 0 by taking $\alpha = 3$. Thus, an optimal timing-function is $t(n,m) = 2n + m - 3$.

A well-known systolic array can be obtained by projecting the array along the n axis. Figure 2 shows this systolic array which uses n bidirectionnally connected cells, for the computation of h(k), $k \leq n$. The detail of the cells is shown in figure 3.

Coefficients h(k) are computed when they flow from the right to the left, and output by the left-most cell. Note that the period of the array is 2, and this can be clearly seen on the dependence graph.

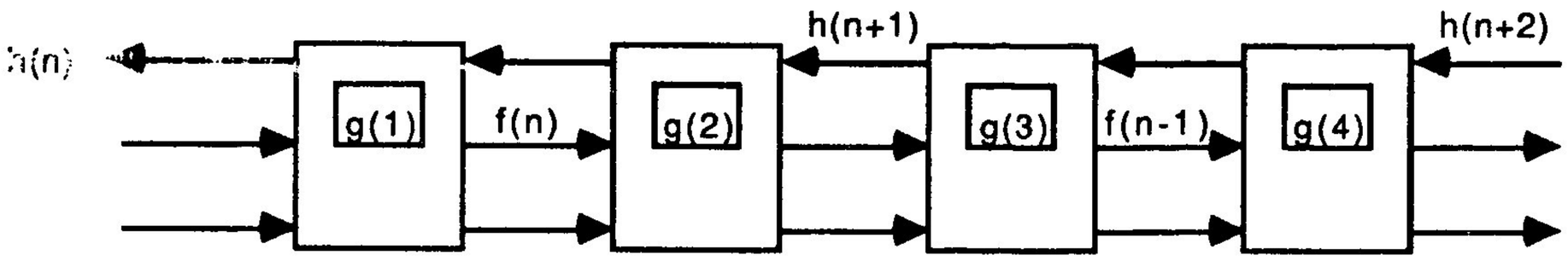
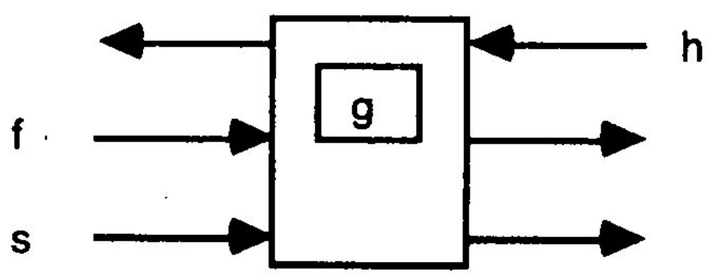
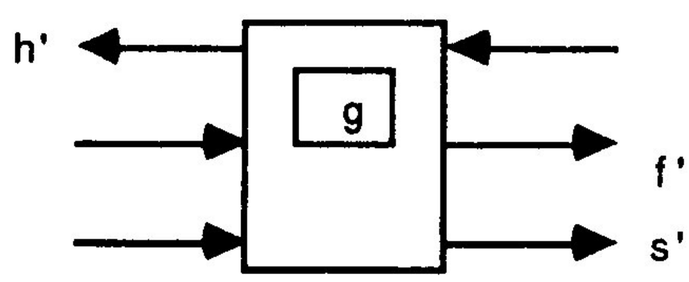


Figure2 : Systolic array for the usual convolution



time t



time t+1

f' := f; s' := s;
case
s → h' := f . g ; {first computation}
not s → h' := h + f . g ; {update}
esac

Figure 3 : Operation of the cells for the unbounded convolution

In this first design, coefficient g(I) is pre-loaded in cell I. The best way to solve this problem is to fold the dependence graph along the bissectrice of the first orthant, in order to let f and g play a symmetric role with regards to the projection direction. However, to do so, it is first necessary to change slightly the way the h coefficient is summed up, so that the direction of its movement remains the same once the domain is folded. The trick is to compute the summation starting by the middle, as shown in figure 4.

In the new dependence graph, g(I) and f(I) flow together, starting from point n=I, m=1, and are reflected along the line n=m. Therefore h(N), moving along the straight line n+m=N−1, meets successively the pairs (g(I),f(I)) and (g(N−I+1),f(N−I+1)), when I ranges from floor(N/2) downto 1.

By projecting the dependence graph along the n axis, one obtains the systolic array depicted by figure 5, whose cells are shown in figure 6.

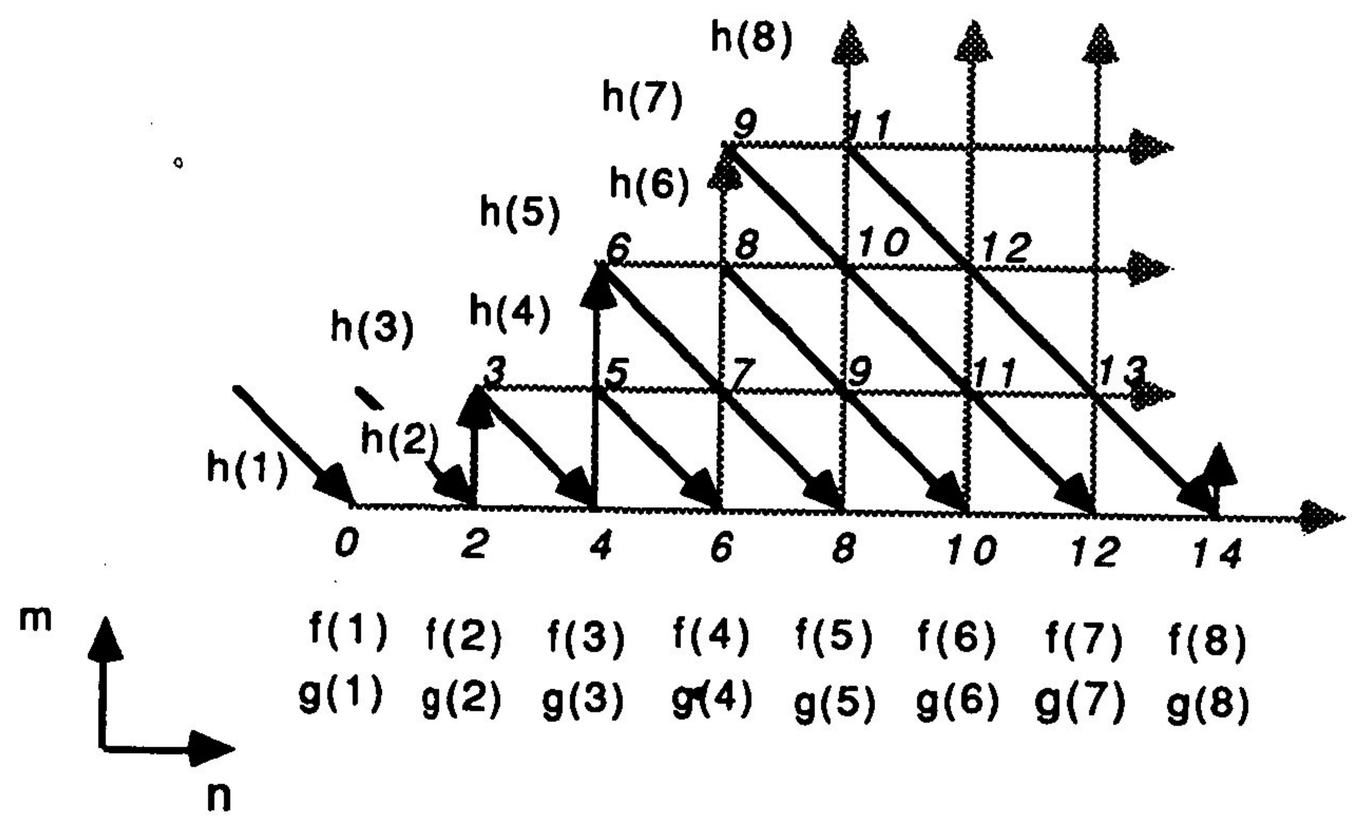


Figure 4 : Dependence graph and timing after folding

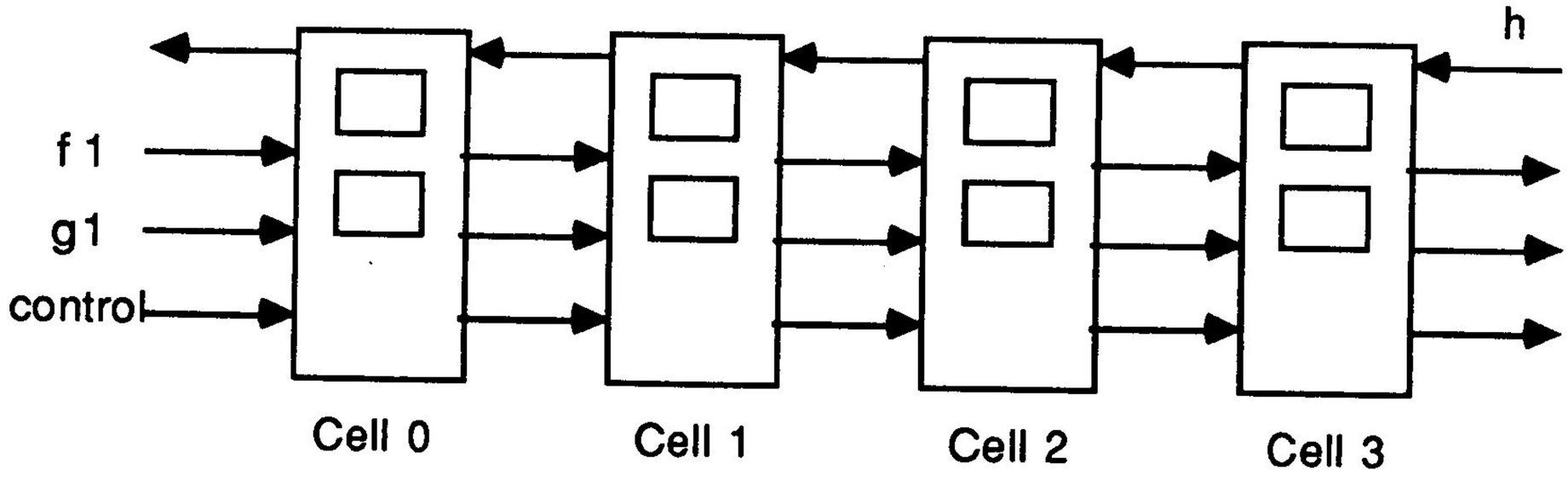
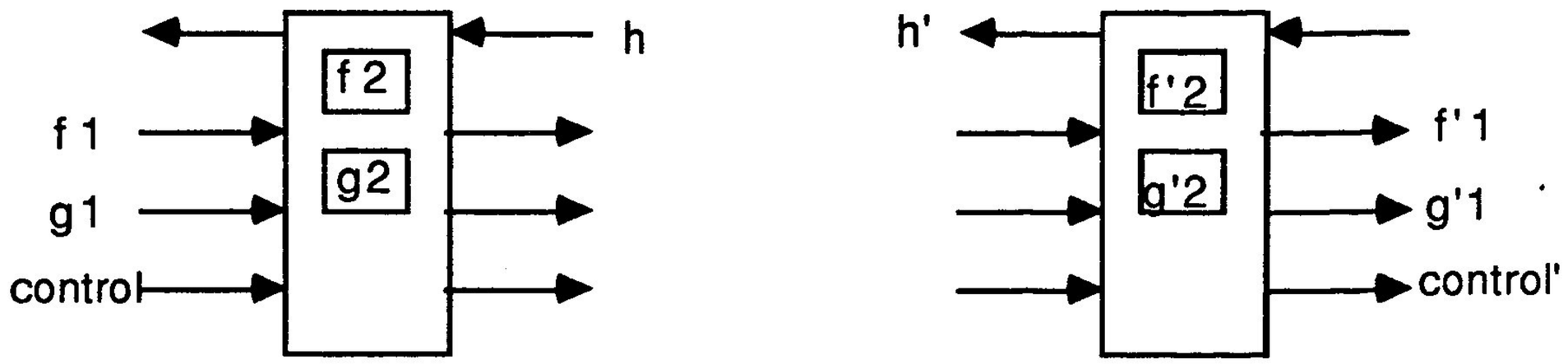


Figure 5 : Systolic array for the unbounded convolution

A few words of explanation about the control of the cells are in order. As shown in figure 4, depending on the parity of N, h(N) starts its computation in point ((N+1)/2,(N+1)/2) when N is odd, or in cell (N/2+1,N/2) when N is even. Moreover, the pair of coefficients g,f is reflected on point ((N+1)/2,(N+1)/2). These peculiarities of the operation of the cell are taken care of by introducing a signal named *control* which can take the value *initodd*, *initeven* or *normal*. When *control=initodd*, the coefficient f and g are loaded in the registers, and h is initialized with the value f*g. When *control=initeven*, h is initialized with f1*g2+f2*g1. Finally, in the normal case, h is incremented with f1*g2+f2*g1. The signal *control* flows along direction given by vector (1,1) on figure 4. Therefore, it visits a new cell every three cycles. Along diagonal n=m, *control = initodd*. Along line n=m+1, *control = initeven*. Otherwise, *control= normal*.



time t time t

```
case
    control = initodd → {this case for the points on line n=m}
    begin
        f2:=f1; g2:=g1; {load f1 and g1 in registers}
        h':=f1*g2; {compute middle term of the sum}
        control':=control; {transmit control}
    end
    control = initeven → {this case for the points on line n=m+1}
    begin
        f'1:=f1; g'1:=g1; control':=control {transmit f1, g1 and control}
        h':=f1*g2+ f2*g1; {compute middle term of the sum}
    end
    otherwise→ {normal case}
    begin
        f'1:=f1; g'1:=g1; control':=control {transmit f1, g1 and control}
        h':=h+f1*g1+ f2*g2; {update with next term of the sum}
    end
esac
```

Figure 6 : Detail of the cells of the systolic array for the unbounded convolution

2.2. Arithmetic convolution

The problem is to compute the convolution h = f * g. For all n≥1, we need evaluate the sum $h(n) = \sum_{kl=n,\ 1\le k,l\le n} f(k)\, g(l)$. The constraints are the following:

- communications with the host: there is only one cell in the array communicating with the host. This cell receives the input sequences (f(n); n≥1) and (g(n); n≥1) and delivers the output sequence (h(n); n≥1)
- real-time computation: for all n≥1, h(n) should be output by the array k units of time after the input of f(n) and g(n), where k is a fixed constant independent of n
- modularity: the operation of the cells should not depend upon their location in the array, nor should it depend on the indices of their inputs/outputs. Provided that this condition is met, the array can be used for the computation of an arithmetic convolution of any size.

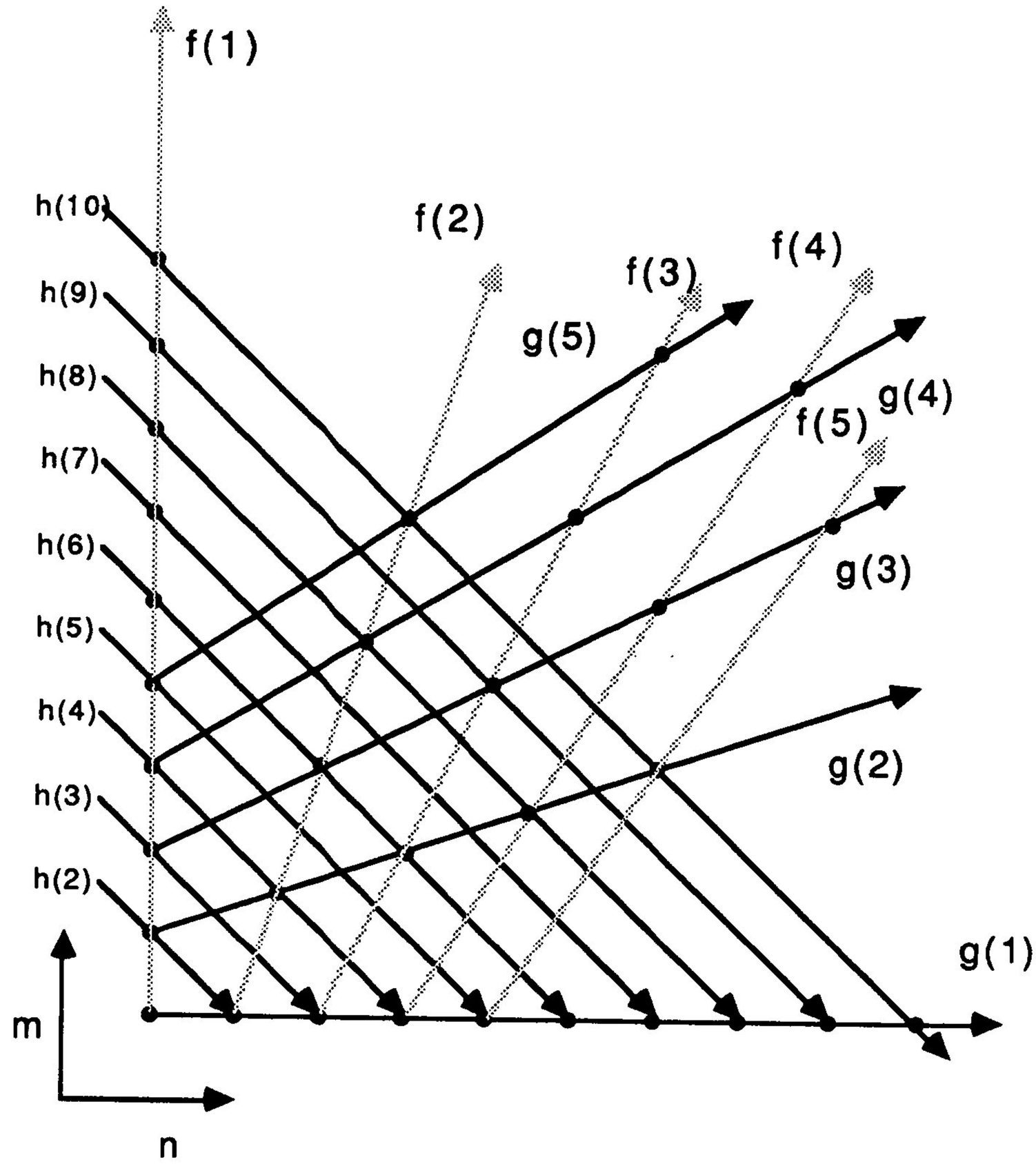


Figure 7 : Dependence graph for the arithmetic convolution

Figure 7 shows a dependence graph for the arithmetic convolution. As in the usual convolution, coefficient h(N) is computed along the diagonal line, whose equation is n+m=N+1. The problem is to "route" coefficients g and f used for computing the product terms, in such a way that the graph be uniform and that these coefficients meet on the right line. The choice that is made here consists in routing coefficients g and f in a *symmetric way*, so that g(n) and f(n) meet on the first bisectrice. For example, g(2) and f(2) meet at point (5/2, 5/2), just on the line where h(4) is summed up, and coefficients g(3) and f(3) at point (5,5), which is just on the line m=10−n where h(9) is computed. The diagram shows intuitively that when the g's and the f's flow on straight lines, they meet on and only on the lines where they are used. This is confirmed by the following lemma :

Lemma 1 : Consider the family of points

$X(k,l) = (G_l(k), F_k(l)) = ((k-1)(l+1)/2 + 1,\ (l-1)(k+1)/2 + 1)$, where $k \geq 1$ and $l \geq 1$

Then the curves $\{G_l(k)\}_{k\geq 1}$ and $\{F_k(l)\}_{l\geq 1}$ are straight lines. Moreover, they meet on the line $m=kl+1-n$.

Proof : Obvious.

Lemma 1 clearly states that the routing scheme shown in figure 7 is correct. However, as it is, this dependence graph cannot be used for deriving a systolic array, for three reasons :

1. the underlying lattice of the graph is not integral,
2. there exists an unbounded number of dependence vectors,
3. we must provide a mechanism to avoid loading the g coefficients.

Problem 1 can be solved by a dilation of the graph by 2 along both directions. We shall also solve problem 3 by applying the same trick as for the unbounded convolution, that is to say, by folding the dependence graph.

Problem 2 is more involved. The solution consists in replacing the straight lines where the coefficients g and f flow by a piecewise linear curve that meets the following conditions :

a) the curve follows a finite number of directions,
b) all the intersection points are on the curve,
c) the number of curves passing through a given point of the plane is bounded.

Condition a) is necessary if one wants to obtain a finite number of dependence vectors. Condition b) is obviously necessary to keep the properties of the previous dependence graph. Finally, condition c) is needed for the resulting systolic array to have only a finite amount of data to be transmitted from one cell to another. This last condition is the reason why we have chosen the symmetric routing scheme (other simpler routing schemes are possible, for example, to have f(k) flow on line n=k and route g(l) to the point (k,kl+1), but in this case, condition c) could not be met).

Let us first ignore the problem of loading the g coefficients. Figure 8 shows the dependence graph after dilation by 2 along both axis (in order for the intersection points to be on an integral lattice), once the straight lines are replaced by a piecewise approximation. This approximation can be explained as follows.

Let $\{G'_l(k,i)\}_{k\geq 1,\ 0\leq i<l+1}$ be the curve. parameterized by k and i, defined by :

$$G'_l(k,i) = \begin{cases} \text{if } 0\leq i\leq 2 & \text{then } \big((k-1)(l+1)+i+2,\ (l-1,k+1)+2\big) \\ \text{if } 2\leq i<l+1 & \text{then } \big((k-1)(l+1)+4,\ (l-1,k+1)+i\big) \end{cases}$$

Informally, the points of this curve are obtained by starting at point (2,2l), and moving once along vector (2,0) then l–1 times along vector (1,1).

Similarly, let $\{F'_k(l,i)\}_{l\geq 1,\ 0\leq i<k+1}$ be the curve, parameterized by l and i, defined by :

$$F'_k(l,i) = \begin{cases} \text{if } 0\leq i\leq 2 & \text{then } \big((k-1)(l+1)+i+2,\ (l-1,k+1)+2\big) \\ \text{if } 2\leq i<k+1 & \text{then } \big((k-1)(l+1)+4,\ (l-1,k+1)+i\big) \end{cases}$$

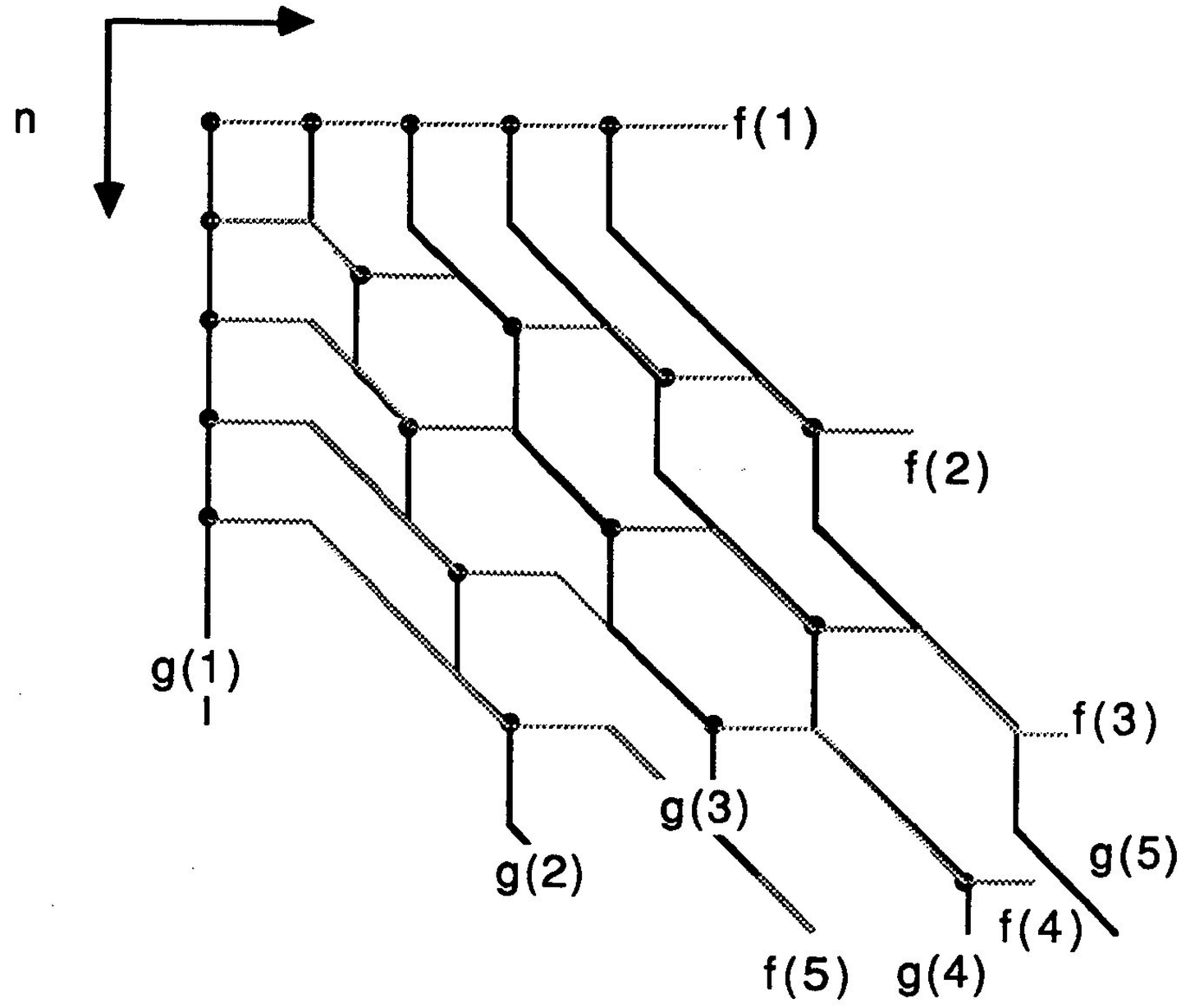


Figure 8 : Piecewise approximation of g and f movement

The following lemma proves that the piecewise approximation meets the previously stated conditions :

Lemma 2 For all $l,k \geq 0$,

1) $\{G'_l(k,i)\}_{k\geq 1,\ 0\leq i<l+1}$, $\{F'_k(l,i)\}_{l\geq 1,\ 0\leq i<k+1}$, and the straight line $n+m=2(kl+1)$ are concurrent

2) The family of curves $\{G'_l(k,i)\}$ (respectively $\{F'_k(l,i)\}$) has no intersection point

Proof : 1) is easily seen, as $G'_l(k,0) = F'_k(l,0) = \big((k-1)(l+1)+2,\ (l-1,k+1)+2\big)$, which is just twice the coordinates of the point X that we have already seen in Lemma 1. The proof of 2) is also very simple.

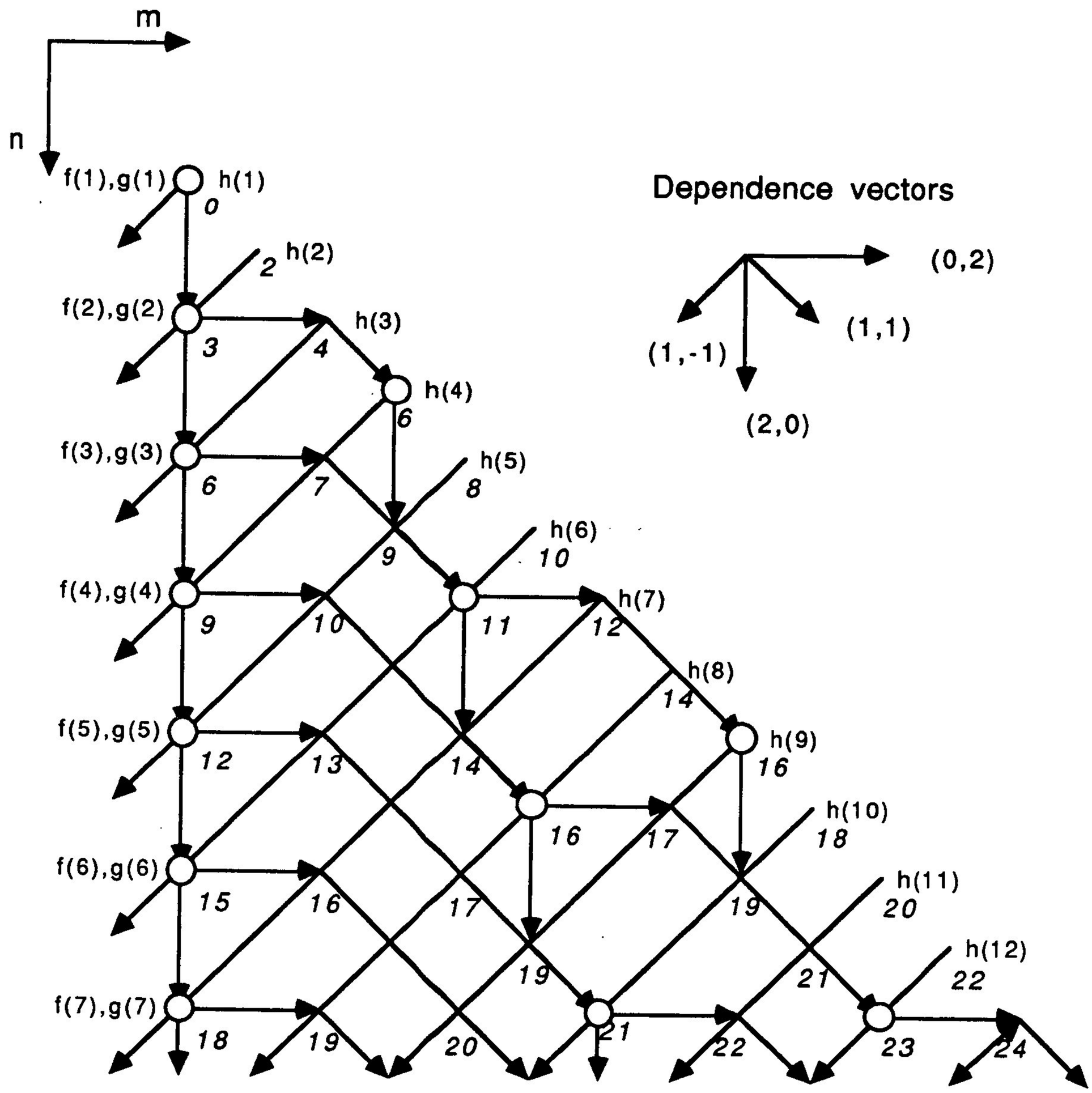


Figure 9 : Final dependence graph for the arithmetic convolution

The loading of the g coefficients is solved in a similar way as already seen in the unbounded convolution, that is to say, by folding the dependence graph along the line n=m. The resulting dependence graph is shown in figure 9. The pairs (f(n),g(n)) are input along the n axis, and are reflected when they meet the line n=m. The dependence vectors are also shown in figure 9. They are

$$\Delta = \{(0,2), (1,1), (2,0), (-1,1)\}$$

Therefore, the parameters λ_1 and λ_2 of the timing function must satisfy to :

$$2\lambda_2 \geq 1 \qquad \lambda_1 + \lambda_2 \geq 1$$

$$2\lambda_1 \geq 1 \qquad \lambda_1 - \lambda_2 \geq 1$$

The optimal (rational) solution is $\lambda_1 = 3/2$ and $\lambda_2 = 1$, which gives the timing function t(n,m) = floor(3n/2+m-2), as shown in figure 9 (the interested reader is refered to [Qui] for the use of rational timing functions).

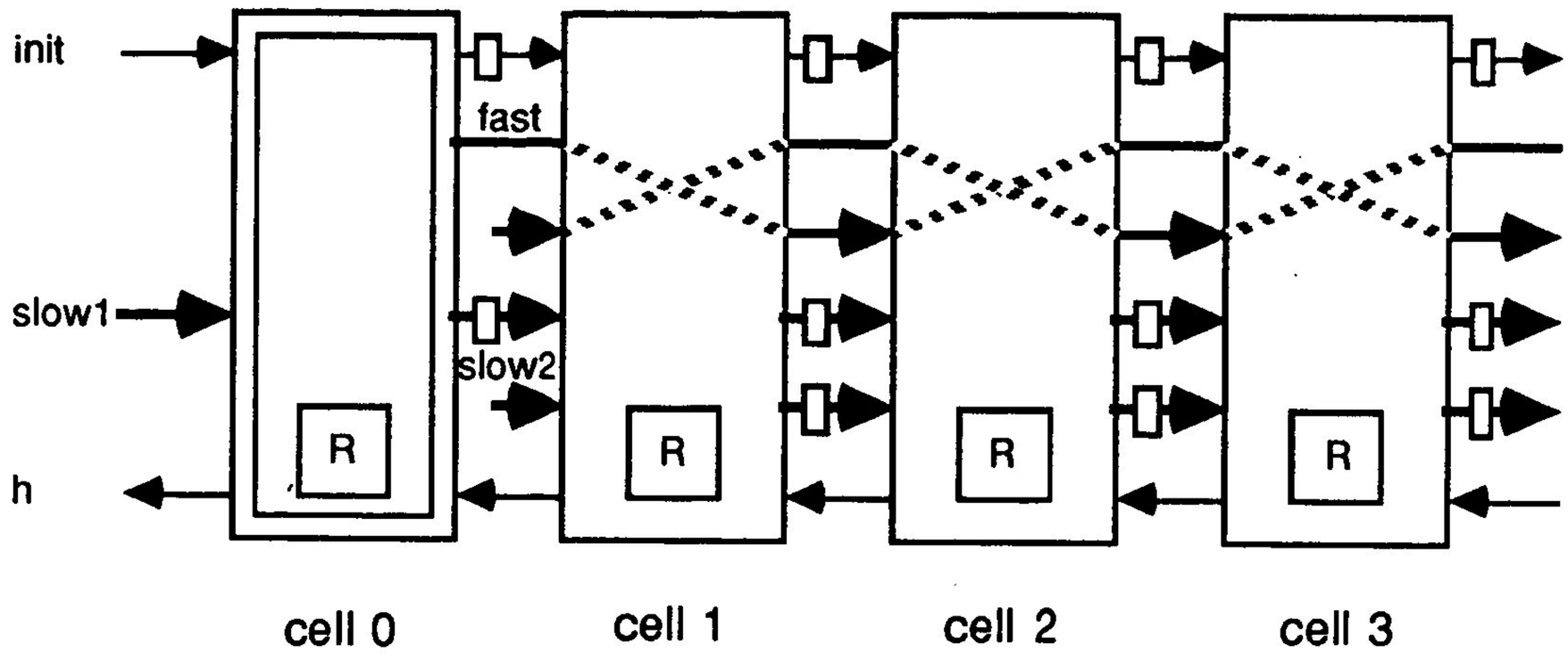


Figure 10 : Systolic array for the arithmetic convolution

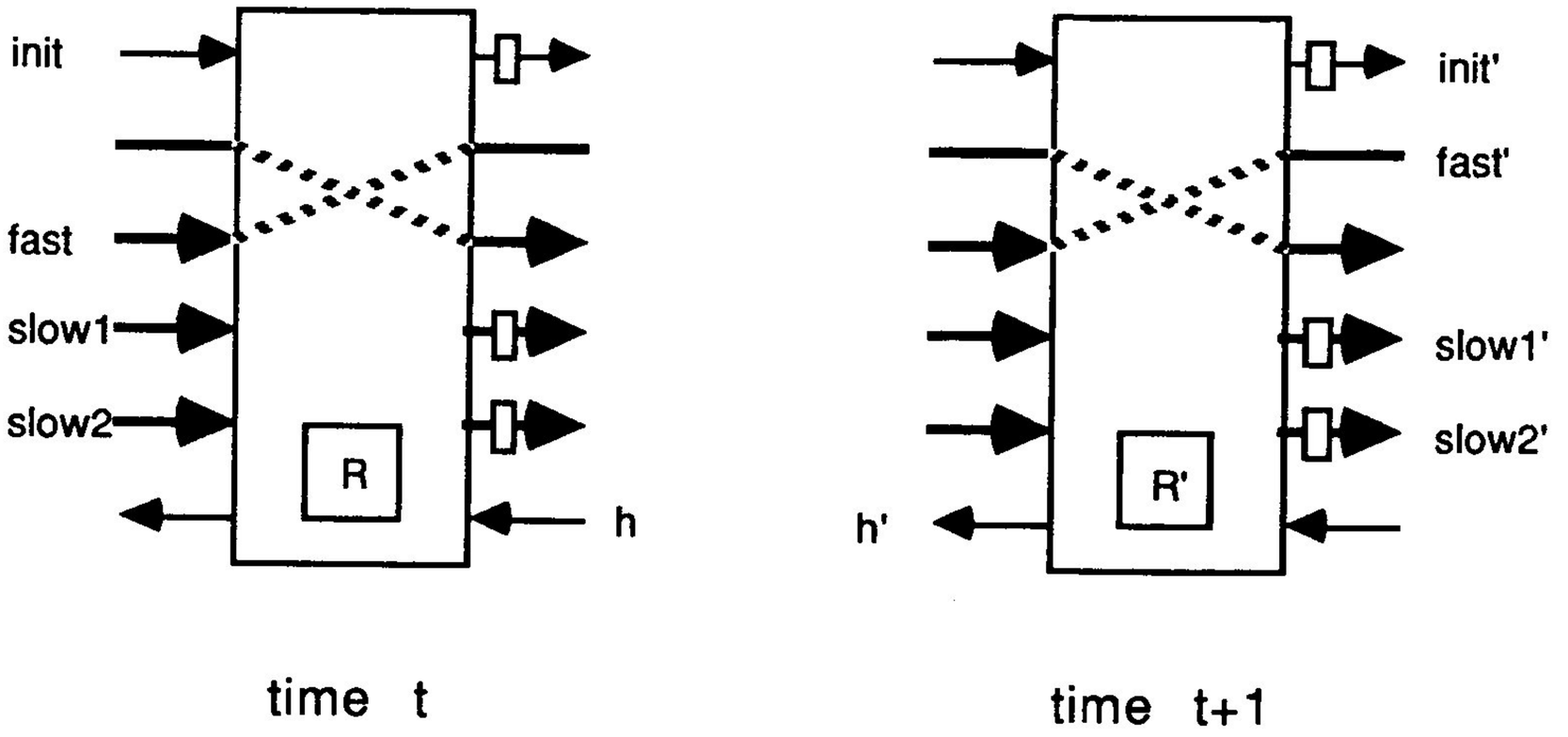
The final architecture (figure 10) is obtained by projecting the dependence graph along axis n. We can immediately see that the number of cells needed for computing the sequence h(k), $1 \leq k \leq n$ is n. Each cell of the array works only every three cycles. Each cell but the first one has four left-to-right, and one right-to-left links :

- *init* carries a initialization signal which follows the line n=m on the dependence graph of figure 9. This signal reaches a new cell every other time;
- *fast* carries the f and g coefficients moving along direction (0,2) on figure 9, as well as other informations that will be detailed later on. After being processed by a cell, these values skip the next cell and reach the second next one;
- *slow1* and *slow2* carry two pairs of f and g coefficients (as well as other informations) moving along direction (1,1);
- *h* carries the value of the h coefficient under computation.

Finally, each cell is provided with a register R which keeps the pair f and g flowing along direction (2,0) in figure 9 : this direction being projected in the same cell, it corresponds to the storage of a value in one cell. The first cell (cell 0) is special. It has only a *slow1* , a *h* and a *init* link.

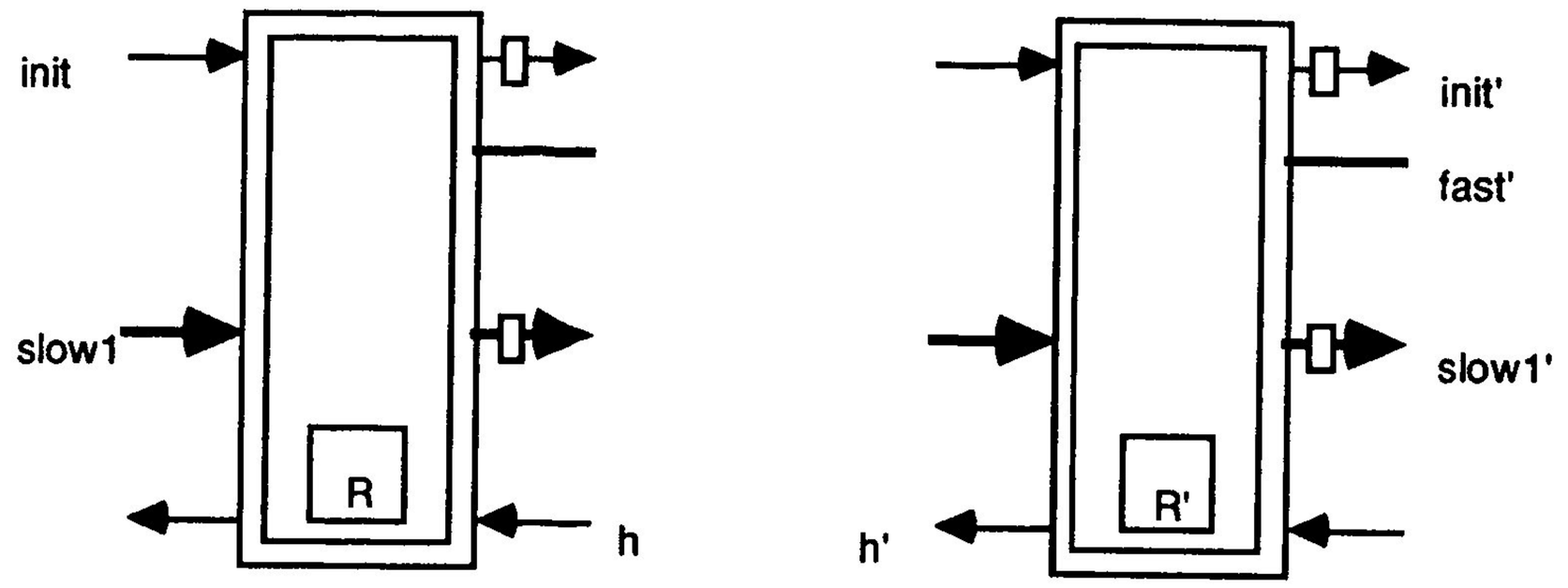
The detail of the operation of the cells is shown in figure 11. Actually, each one of the *fast, slow1* and *slow2* link carries a 5-tuple (f,g,valid,rank,count), where f and g are coefficients, valid is a marker indicating that the link effectively carries significant values, rank is the number of the coefficients, and count is an integer which will be used to determine the movement of the pair f,g. To understand the operation of the cell, it is best to refer again to the dependence graph of figure 9. As already seen, a pair (f(k),g(k)) moves once along direction (0,2) then k times along direction (1,1). After the pair is reflected along the line n=m, it moves once along direction (2,0) and k times along direction (1,1). The rank parameter is used to remember the k value, and

count is decreased when doing the k movements along direction (1,1). When count reaches 0, then two pairs of coefficients are available on *slow1* and *slow2* and the h coefficient is modified.




```
case
        init and slow1.count ≠ 0 → {initialization and no calculation}
        begin h':=0; init':=init end;
        init and slow1.count = 0 → {initialization and calculation}
        begin
                h':=h + slow1.f * slow1.g; R:=slow1; init':=init {store slow1 in R}
        end;
        not init and fast.valid→ {send fast on slow1, reset count}
        begin
                slow1:=fast; slow1'.count:=rank-1; init':=init
        end;
        not init and slow1.valid and slow1.count ≠ 0 and not slow2.valid→
        begin {Keep on moving on slow lines, decrease counts}
                slow1':=slow1; slow1'.count:=slow1'.count-1;
                slow2':=R; slow2'.count:=slow2'.rank; init':=init
        end;
        not init and slow1.valid and slow1.count ≠ 0 and slow2.valid→
        begin {Keep values moving on slow lines, decrease counts}
                slow1':=slow1; slow1'.count:=slow1'.count-1;
                slow2':=slow2; slow2'.count:=slow2'.count-1; init':=init;
        end;
        not init and slow1.valid and slow1.count = 0 →
                {Note that slow2 is necessarily valid, and slow2.count is also 0}
                begin {Compute, send slow1 on fast and keep slow2 in R}
                        h':=h + slow1.f * slow2.g + slow2.f * slow1.g;
                        R:=slow2; fast:= slow1; init':=init;
                end
esac
```

Figure 11 : Operation of the cells



case

init → {initialization: compute and store in R}
begin
h':=slow1.f * slow1.g; R':=slow1; {store in R}
init':=init
end;
not init → {first cell, normal case}
begin
h':=h + slow1.f * R.g + R.f * slow1.g;
R':=R {store in R}; init':=init
end;
esac

Figure 11 (continued) : Operation of the first cell

2.3. Performances

In fact, although each cell is activated every three cycles, the real efficiency of the network is less than 1/3, as the number of calculations to be done (in sequential) for computing h(n) is not of the order of n: computing (h(m); $1 \le m \le n$) requires $\sum_{1 \le m \le n} \text{div}(m) = O(n \log n)$ multiplications, where div(m) is the number of divisors of m.

In summary, this first solution uses O(n) cells for processing in time O(n) the computation of the sequence h(k), $1 \le k \le n$. However, as each cell has to make use of a counter initialized to k, the area complexity of this design is O(n log n).

3. Another systolic design

In this section, we design another systolic array of processors for the parallel computation of the convolution of two arithmetic functions. We address the inverse convolution problem in section 3.4.

The second array will look like the one shown on figure 12:

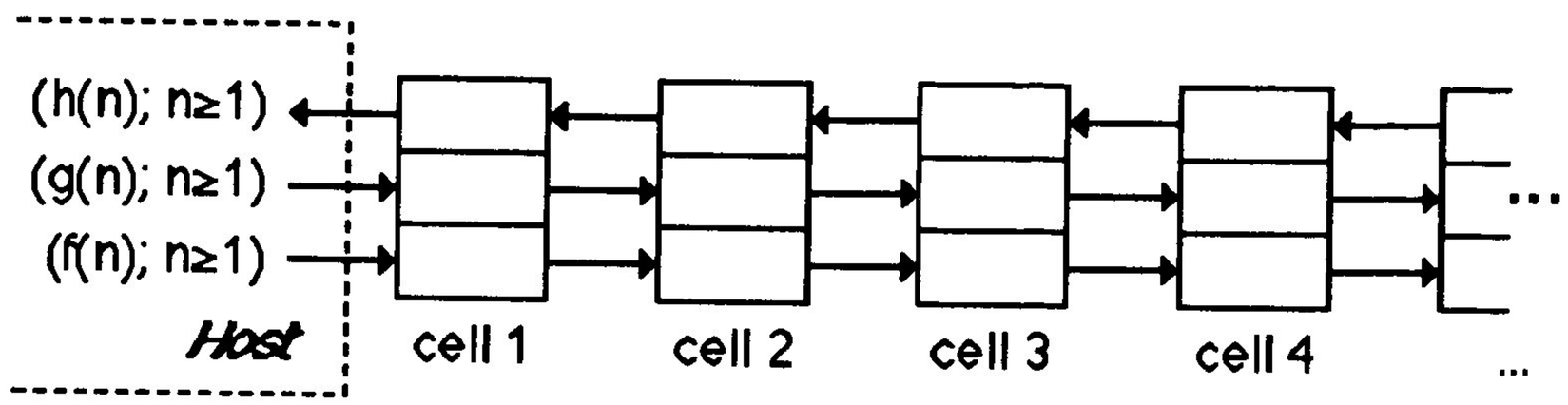


Figure 12 : The second linear array

3.1. Half computation

First, we show how to compute the sum $h_1(n) = \sum_{kl=n,\ k\geq l} f(k)\ g(l)$. The other half of the arithmetic convolution will be computed similarly. We use a linear array of cells, as in figure 12, and number the cells from left to right.

- **Input and output format**

  For all n≥1, f(n) and g(n) enter the array at time n; $h_1(n)$ is output at time n (see figure 13)

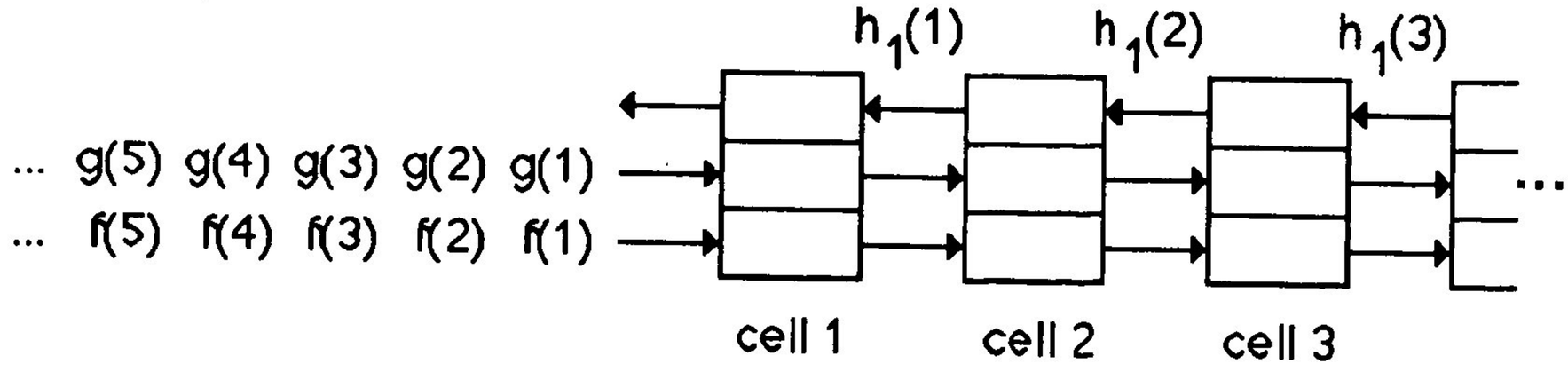


Figure 13 : Input/output format

- **Flow of g(l)**

  For all l≥1. g(l) flows rightwards until it reaches cell l. It is then stored in the internal register *g_reg* of cell l. The systolic mechanism to realize such a flow is well-known: cells are marked when their register g_reg is filled, and each g(l) flows rightwards until it finds a nonmarked cell.

- **Flow of f(k)**

  For all $k \geq 1$, f(k) flows rightwards until cell k, where it initializes the operation of the cell. It is then marked nonactive. Therefore f(k) is the first active f-input of cell k for all k. When it reaches cell k, f(k) meets g(k) and the first product f(k)*g(k) is computed for $h_1(k^2)$. According to the I/O format, $h_1(k^2)$ is an input of cell 1 at time k^2, so the product f(k)*g(k) must be computed at time $k^2-(k-1)$ in cell k. The flow of the f's will be organized to meet this requirement: f(k) will reach cell k at time k^2-k+1. We detail herafter the organization of the flow of f.

- **Flow of $h_1(n)$**

  Since $h_1(n)$ is in cell 1 at time n, we can conceptually say it is in cell n at time 1 and moves leftwards from cell to cell at speed 1. In fact the first product of $h_1(n)$ is computed in cell l, where l is the largest divisor of n such that $l^2 \leq n$, at time $n-l+1$. As it moves leftwards, $h_1(n)$ accumulates partial products f(k)*g(l) in decreasing order with respect to l, in all cells l such that l divides n.

Now we need organize the flow of the f's. We examine the first meetings that are required. In table 1, we report the times when, and the cells where, products are computed.

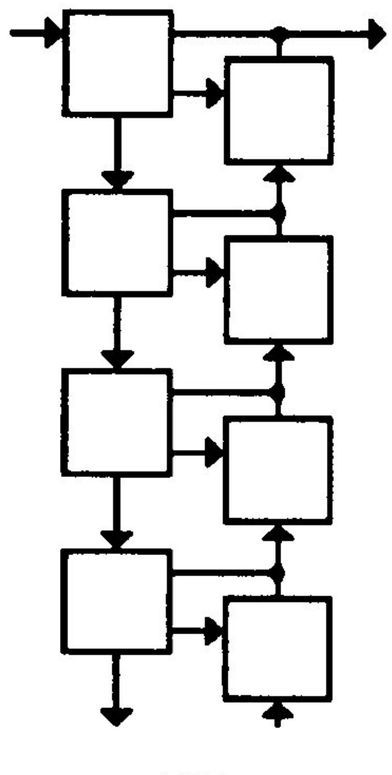
Times	Cell 1	Cell 2	Cell 3	Cell 4	Cell 5
1	f(1).g(1)				
2	f(2).g(1)				
3	f(3).g(1)	f(2).g(2)			
4	f(4).g(1)				
5	f(5).g(1)	f(3).g(2)			
6	f(6).g(1)				
7	f(7).g(1)	f(4).g(2)	f(3).g(3)		
8	f(8).g(1)				
9	f(9).g(1)	f(5).g(2)			
10	f(10).g(1)		f(4).g(3)		
11	f(11).g(1)	f(6).g(2)			
12	f(12).g(1)				
13	f(13).g(1)	f(7).g(2)	f(5).g(3)	f(4).g(4)	
14	f(14).g(1)				
15	f(15).g(1)	f(8).g(2)			
16	f(16).g(1)		f(6).g(3)		
17	f(17).g(1)	f(9).g(2)		f(5).g(4)	
18	f(18).g(1)				
19	f(19).g(1)	f(10).g(2)	f(7).g(3)		
20	f(20).g(1)				
21	f(21).g(1)	f(11).g(2)		f(6).g(4)	f(5).g(5)

Table 1 : Space-time diagram for the computation of partial products

From table 1 we see that cell k is activated every k-th step after receiving its first f-input f(k) at time k^2-k+1. In other words, f(j) is input cell k at time $kj-k+1$ for all $j \geq k$. For $j \geq k+1$, f(j) is input to cell k+1 at time $(k+1)j-(k+1)+1$, so that f(j) should be delayed j-2 units of time in cell k before being output towards cell k+1.

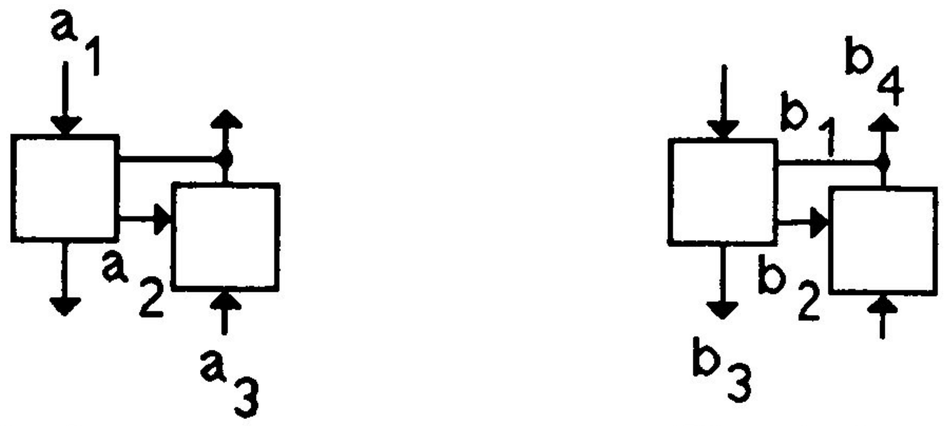
There is a special treatment for f(1) by cell 1, which acts slightly differently from the other cells: rather than marking f(1) nonactive, it deletes it (equivalently, it can mark it with some special code). As a consequence, the i-th f-input to all cells except the first one should be delayed of i-1 units of time. The first input, namely f(2), is transmitted without delay. The second input, namely f(3), is delayed of one unit of time, and so on. After having deleted f(1), cell 1 operates exactly as the other cells.

Right now, we only need design a special systolic mechanism to generate these delays: then we add one of them to each cell, and the flow of the f's will be correct. Such a mechanism cannot be implemented using counters, because of the modularity constraint. We use a design similar to that described in [RT].



...

Figure 14: Delay mechanism




```
case status of
        first: begin b1 := a1 ; status := second ; end
        second:  begin b2 := a1 ; status := marked ; end
        marked : begin b3 := a1 ; end
b4 := a2 OR  a3 ;
```

Figure 15 : Operation of the delay mechanism

The delay mechanism is depicted figure 14 It is a two-columns array of delays cells. The first column is composed of cells with one input and three outputs. The operation of the cells in the first column is very simple (figure 15):
- the first input is output rightwards on the fast channel
- the second input is output rightwards on the slow channel
- all following inputs are output downwards to the next cell in the column

The cells of the second column simply transmit their valid input, if any (to implement this, they perform an OR-operation on their two inputs, where non specified variables have the default value *nil*). Some consecutive time-steps of the mechanism are illustrated figure 16.

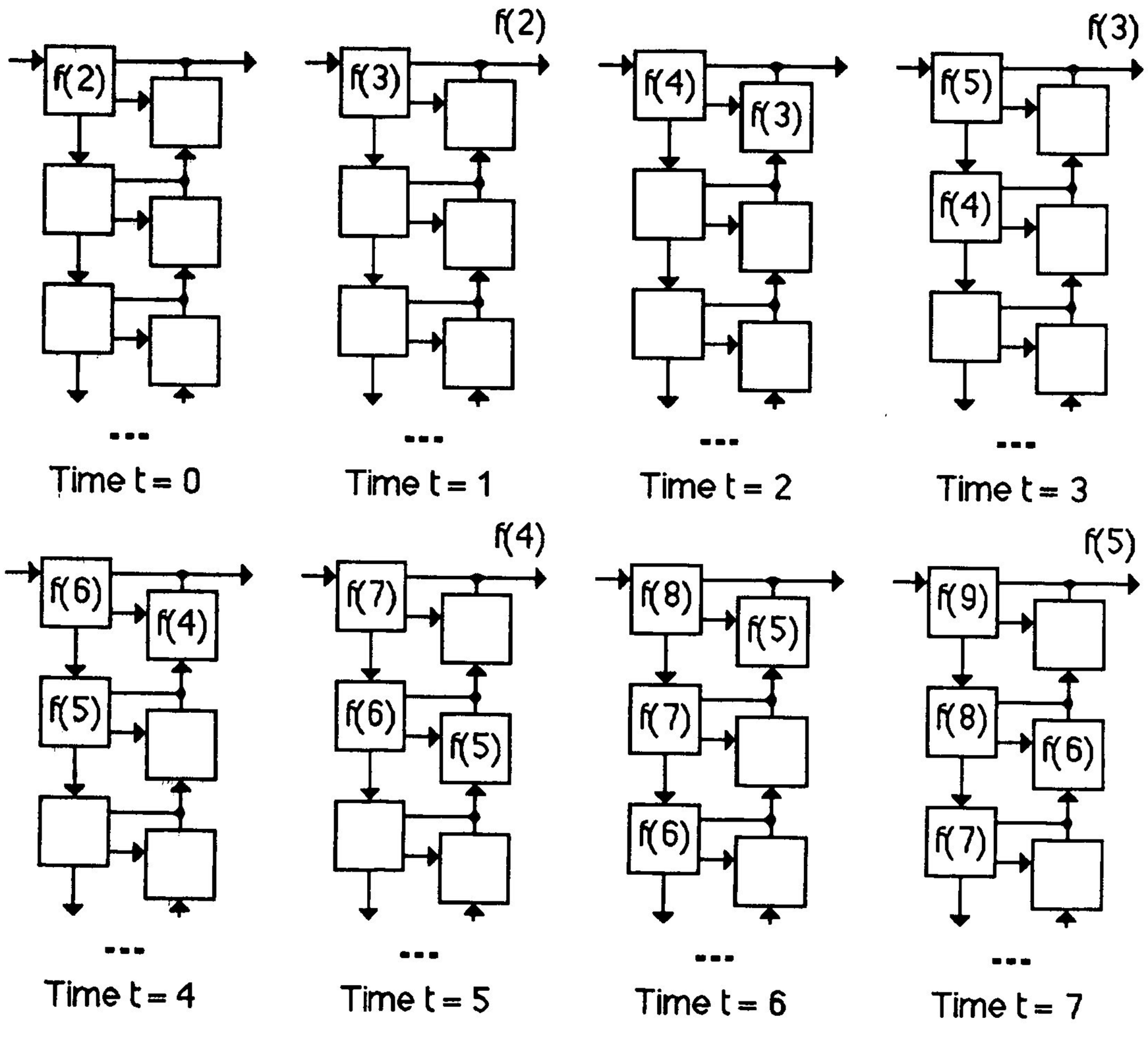
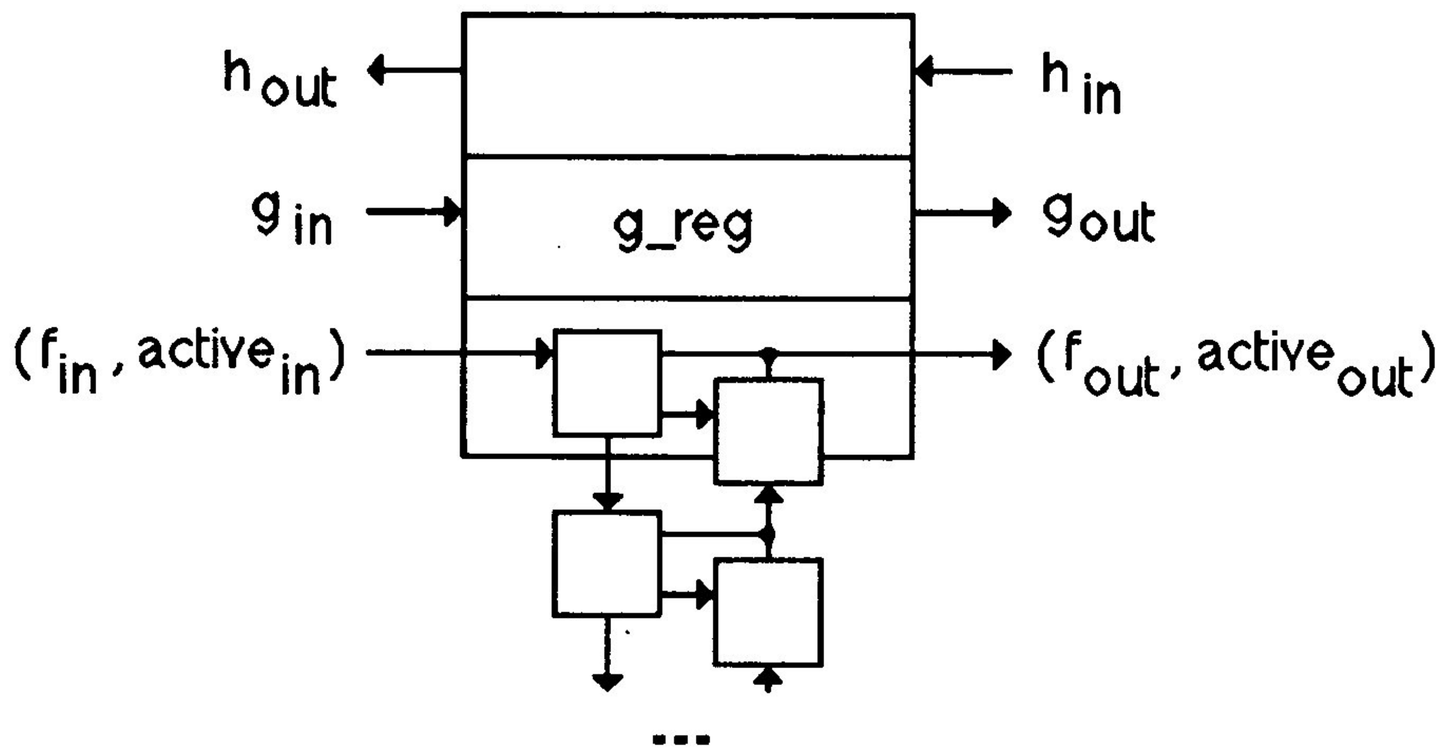


Figure 16: Some consecutive time-steps for the delay mechanism

The full operation of the cells for the computation $h_1(n) = \sum_{kl=n,\ k \geq l} f(k)\ g(l)$ is described figure 17, where non specified variables have the default value *nil* . As stated above, the operation of cell 1 is slightly different, since its first input is deleted.

Proof of correctness

We know that f(k), k≥1, reaches cell I at time kI+k-1. If k≥I, f(k) is active cell I, and the product f(k)*g(I) is computed. Consider the computation of $h_1(n)$: $h_1(n)$ is in cell I at time n-I+1. It meets some f(k) there if and only if kI-I+1 = n-I+1, that is kI = n. Then $h_1(n)$ is updated into $h_1(n) := h_1(n) + f(k).g(I)$ if and only if f(k) is active, or equivalently k≥I. Therefore the final value of $h_1(n)$ is $h_1(n) = \sum_{kI=n,\ k\geq I} f(k)\ g(I)$ as expected.




```
{ store g_in in g_reg if nonmarked }
if nomarked then
      begin g_reg := g_in ; nonmarked := false ; end
else g_out := g_in ;
{ active f-input }
if active_in then
      begin
            { inactivate first f_input }
            if first_f then begin active_in := false ; first_f := false ; end
            { update h_in }
            h_out := h_in + g_reg * f_in ;
      end
{ delay all f_inputs }
(f_out, active_out) := Delay_Mechanism(f_in, active_in) ;
```

Figure 17 : Operation of the cells of the array

3.2. Systolic arithmetic convolution

For computing $h(n) = \sum_{kI=n} f(k)\ g(I)$, we use two copies of the previous array. In the first array, we compute $h_1(n) = \sum_{kI=n,\ k\geq I} f(k)\ g(I)$ as before.

In the second array, we compute $h_2(n) = \sum_{kl=n,\ k>l} g(k)\ f(l)$. We interchange the flows of f and g in the scond array: the f's are stored in the cells, and the g's move rightwards with delays. The only modification is that the second array should not compute products f(k)*g(k), as they are already computed by the first array. We simply modify the operation of the cells (except the first one) as follows: the first time they receive an active g-input, they let $h_2(k^2) := 0$ rather than $h_2(k^2) := f(k)*g(k)$. In fact, we can make things simpler by coalescing the corresponding cells of both arrays, as described figure 18. Again, the first cell is slightly different because it deletes its first f-input (bottom part) and its first g-input (top part). Also, it duplicates f-and g-inputs. See figure 19 for the operation of all cells but the first one, and figure 20 for the operation of the first cell.

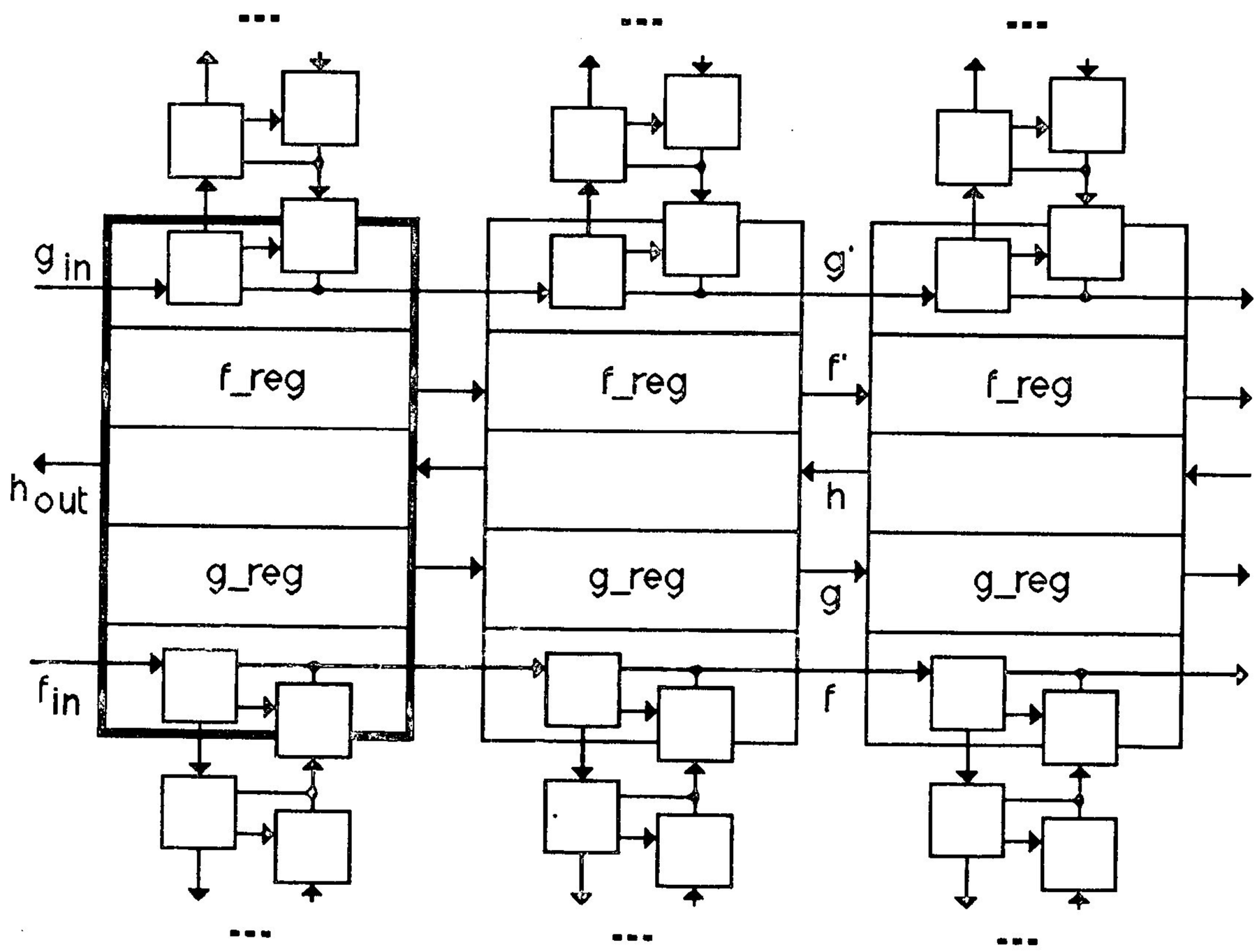
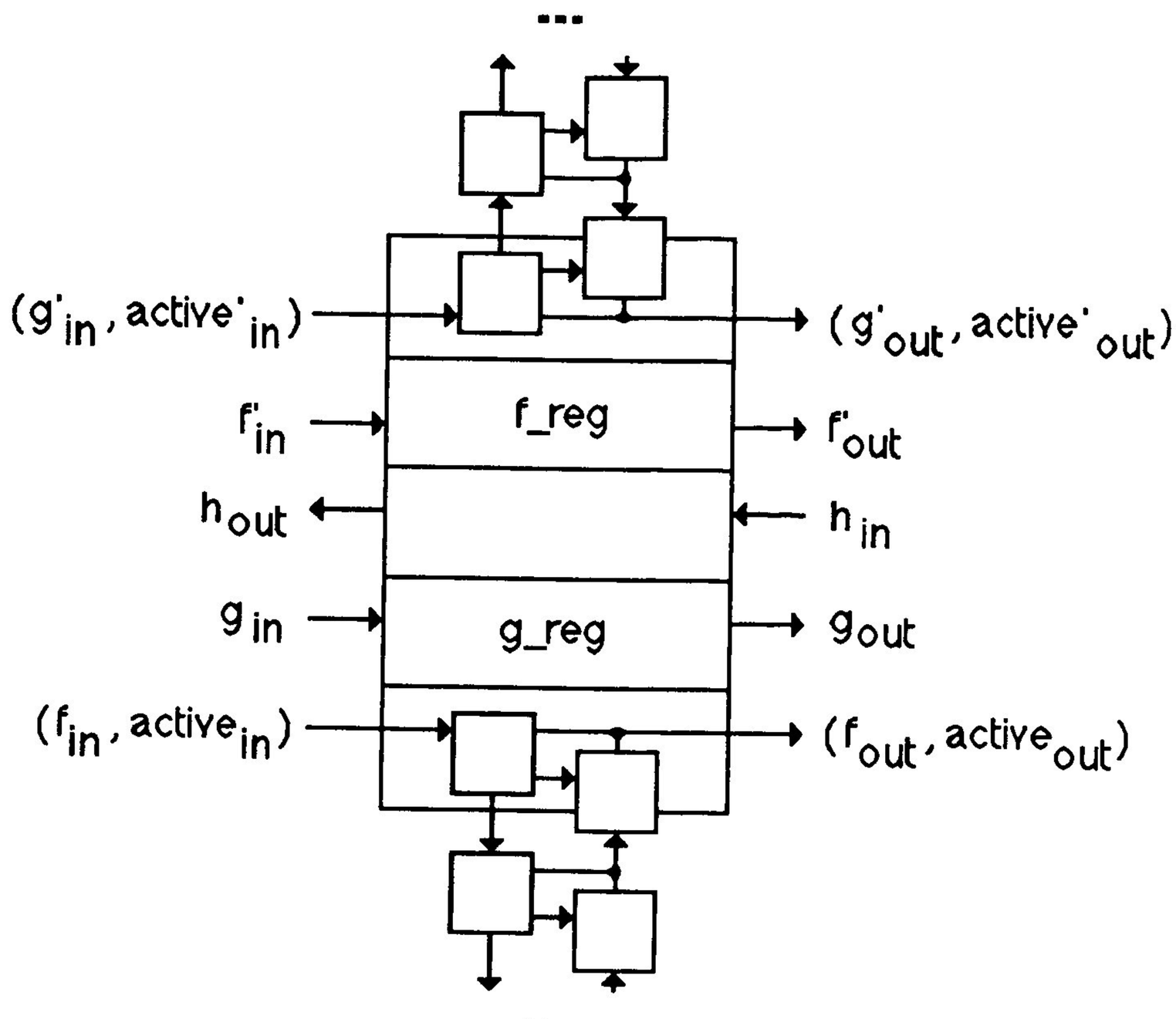


Figure 18 : Systolic array for the arithmetic convolution h = f * g

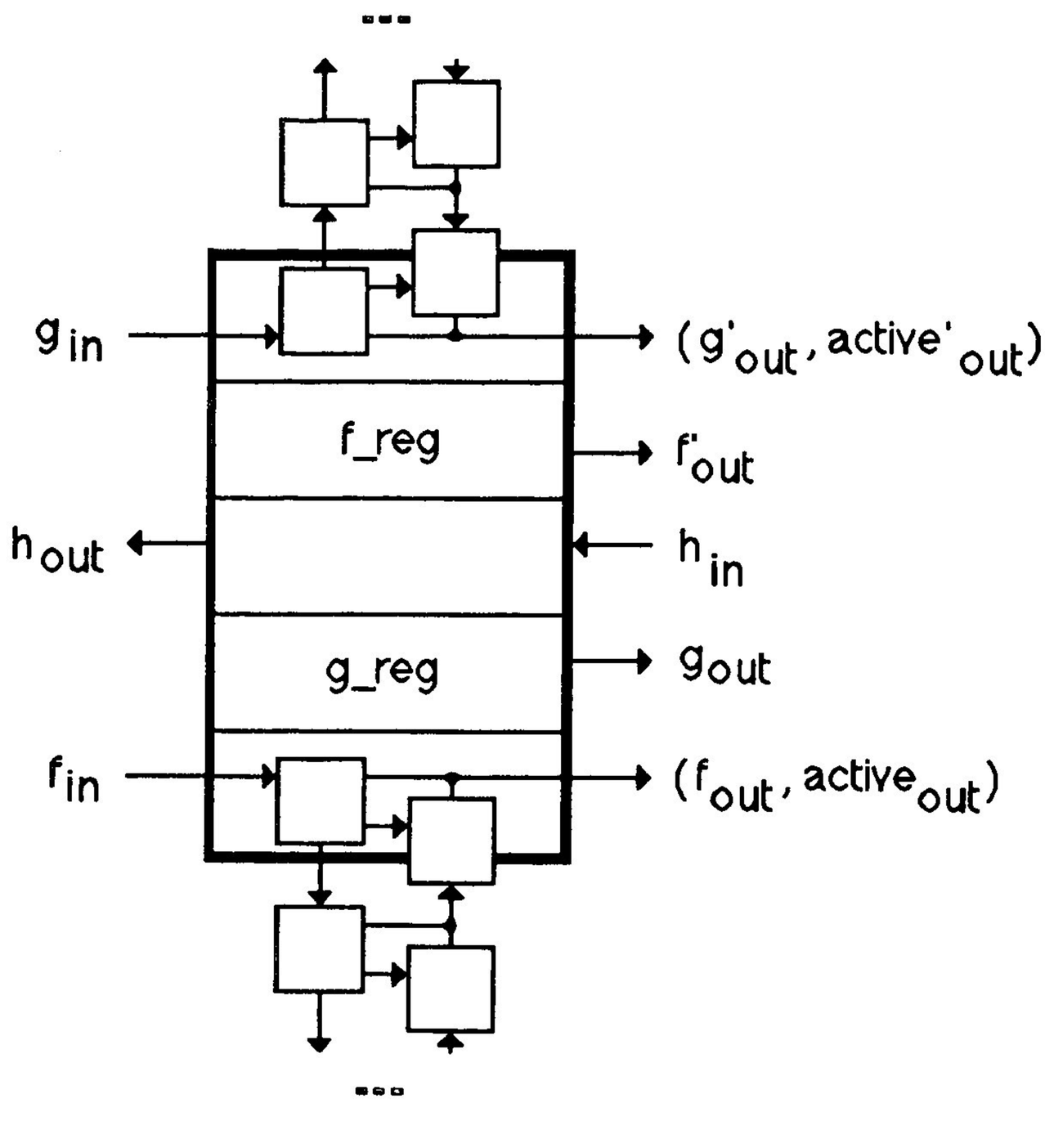
3.3. Performances

For the computation of (h(m); 1≤m≤n), we have designed an array of $n^{1/2}$ processing cells (which perform multiply-and-adds). Note that the total number of delays is proportional to n log n, and not to n, although we have only O(n) inputs. To see this, consider for instance the first array. The last element we need to consider in cell k is f(n/k), to be multiplied by g(k). f(n/k) has been delayed n/k units of time in cells 1, 2,..., k-1 before reaching cell k. So that we need n/2 delays in cell 1 (due to f(n/2)), n/3 delays in cell 2 (due to f(n/3)), n/4 delays in cell 3 (due to f(n/4)), and so on up to $n/n^{1/2}$ delays in the cell before the last one (due to $f(n^{1/2})$).




```
{ store gin in g_reg and f'in in f_reg if nonmarked }
if nonmarked then
      begin g_reg := gin ; f_reg := f'in ; nomarked := false ; end
else
      begin gout := gin ; f'out := f'in ; end
{ active f-input and g'-input. Note that activein = active'in by symmetry }
if activein then
      begin
            { inactivate f- and g'- inputs }
            if first_fg' then
                  begin activein := false ; active'in := false ; first_fg' := false ; end
            { update hin }
            hout :=  hin + g_reg * fin  + f_reg * g'in ; end
{ delay all f- and g'-inputs }
(fout, activeout) := Delay_Mechanism(fin, activein) ;
(g'out, active'out) := Delay_Mechanism(g'in, active'in) ;
```

Figure 19 : Operation of the cells for the arithmetic convolution h = f * g




```
if nonmarked then
      begin
            { gin = g(1) ; fin = f(1); store and mark cell }
             f_reg := fin ; g_reg := gin ; nonmarked := false ;
            { delete fout and g'out } fout := nil ; g'out := nil ;
            { compute h(1) } hout := fin * gin ;
      end
else
      begin
            { transmit gin and f'in } gout := gin ; f'out := fin;
            { update hin } hout :=  hin + g_reg * fin  + f_reg * g'in ;
            { activate f-input and g'-input } activein := true ; active'in := true ;
            { delay all f- and g'- inputs }
            (fout, activeout) := Delay_Mechanism(fin, activein) ;
            (g'out, active'out) := Delay_Mechanism(g'in, active'in) ;
end
```

Figure 20 : Operation of first cell for the arithmetic convolution h = f * g

3.4. The inverse arithmetic convolution problem

The previous array can be very easily modified to solve the inverse arithmetic problem. This is quite similar to the technique used for moving from FIR filtering to IIR filtering [Kun82] or from polynomial multiplication to polynomial division [Kun81].

To compute (whenever possible) the function f such that f * g = h, we observe that

$f(1) = h(1) / g(1)$

$f(n) = [\, h(n) - \Sigma_{kl=n,\ 1\le k,l\le n,\ k\ne n} f(k).g(l) \,] / g(1)$ if $n\ge 1$

We input to the array the sequence (g(n); n≥1) in the same format as before. We replace the input sequence (f(n); n≥1) by the sequence (h(n); n≥1), with the same format (figure 21). All cells operate exactly as before, except the first one whose program is given figure 22.

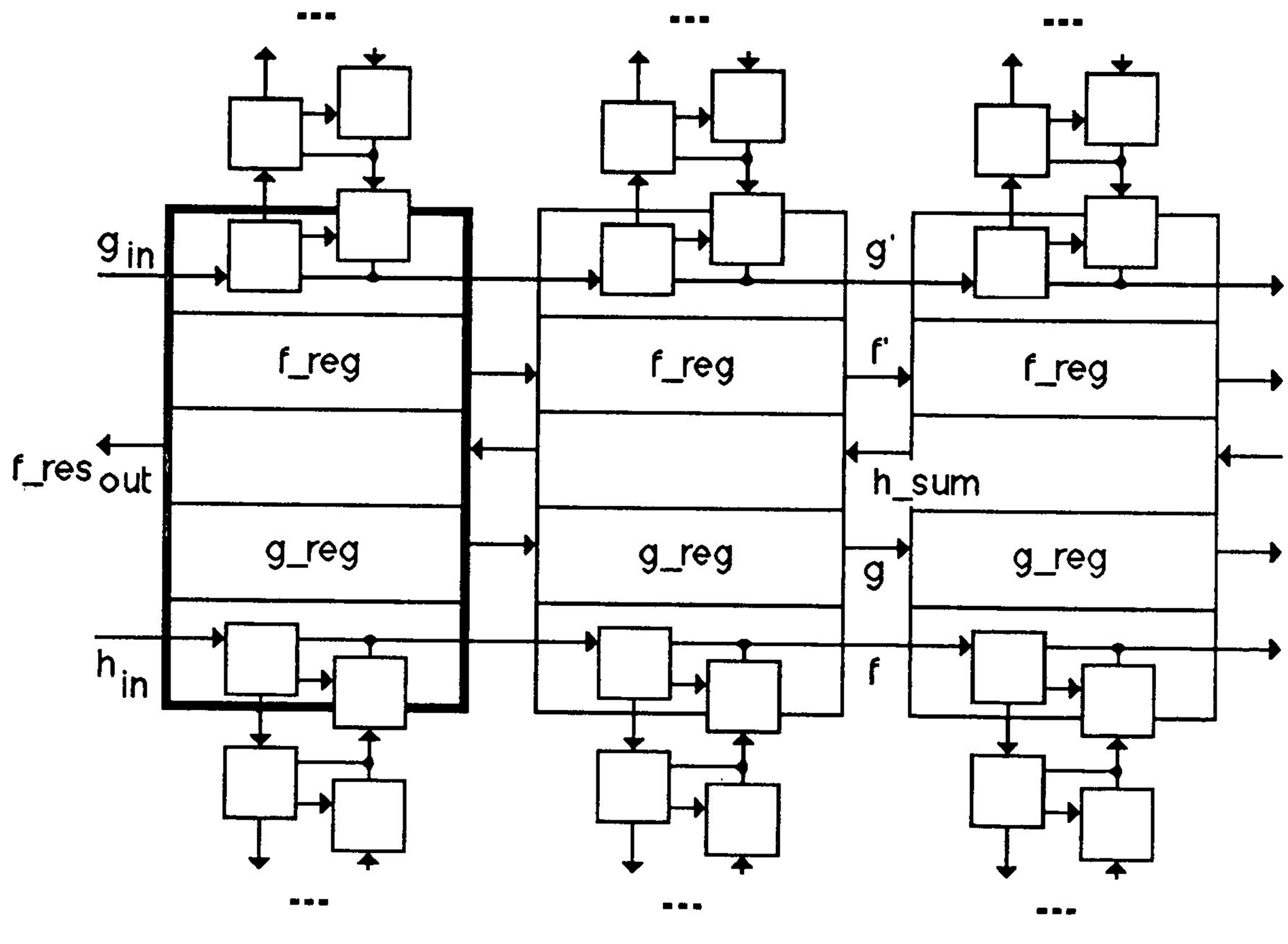
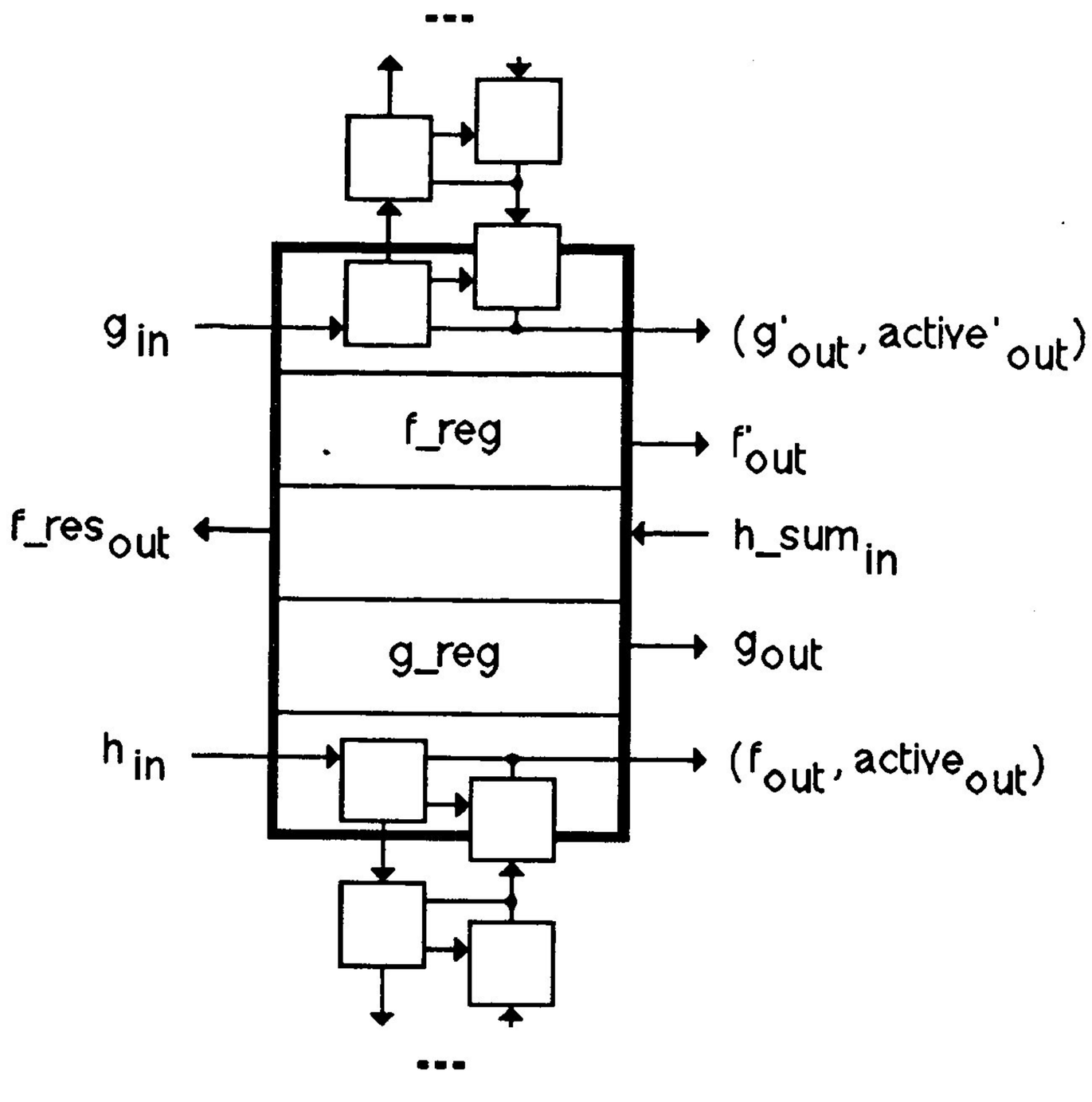


Figure 21 : Systolic array for the inverse arithmetic convolution problem

The performances are the same as for the direct arithmetic convolution. We use $n^{1/2}$ processing cells and O(n log n) delays for the computation of the sequence (f(m); $1\le m\le n$) within n units of time.




```
if nonmarked then
      begin
      { gin = g(1) ; hin = h(1) ; compute f_resout = f(1) } f_resout  := hin / gin  ;
      { store and mark cell } f_reg := f_resout ; g_reg := gin ; nonmarked := false ;
      { delete fout and g'out } fout := nil ; g'out := nil ;
      end
else
      begin
            { compute f_resout } f_resout := (hin - h_sumin - f_reg*gin ) / g_reg ;
            { transmit gin and f'in } gout := gin ; f'out := f_resout ;
            { activate f-input and g'-input } activein := true ; active'in := true ;
            { delay all f- and g'- inputs }
            (fout, activeout) := Delay_Mechanism(f_resout, activein) ;
            (g'out, active'out) := Delay_Mechanism(g'in, active'in) ;
end
```

Figure 22 : Program of first cell for the inverse convolution problem

4. Conclusion

We have presented two linear systolic arrays for the real-time solution of the arithmetic convolution and of the inverse arithmetic convolution problem. We believe it would be an interesting challenge to derive the second designcompletely automatically using the synthesis methods of [Che] [DI] [Mol] [Qui] or the parallel constructs of [SS] [Ver].

Acknowledgements

The authors would like to thank Tom Verhoeff for bringing the arithmetic convolution problem to their attention. Interesting discussions were conducted with Marina Chen, Young-il Choo, Martin Rem, Jan van de Snepscheut, and Tom Verhoeff during a workshop organized by Alain Martin in La Jolla, California, USA, on February 22-26, 1988. Thanks to all of them !

Note added in proof

Since the beginning of this work, we have been aware of two other solutions to the arithmetic convolution problem, by Chen and Choo [CC] and by Duprat [Dup]. Both are variations on the first design presented in this paper, in that they are linear arrays of O(n) cells that solve the arithmetic computation problem in time O(n).

References

[Che] M. CHEN, Synthesizing VLSI architectures: dynamic programming solver, Proc. 1986 Int. Conf. on Parallel Processing, K. Hwang et al. eds., IEEE Computer Society Press (1986), 776-784

[CC] M. CHEN, Y. CHOO, Synthesis of a systolic Dirichlet product using non-linear contraction domain, in Parallel and Distributed Algorithms, M. Cosnard et al. eds., North Holland (1989)

[DI] J.M. DELOSME, I.C.F. IPSEN, Systolic array synthesis: computability and time cones, in Parallel Algorithms and Architectures, M. Cosnard et al. eds., North Holland (1986), 295-312

[Dup] J. DUPRAT, Private communication

[Ken] H.L. KENG, Introduction to number theory, Springer Verlag, Berlin Heidelberg New York (1982)

[Kun81] H.T. KUNG, Use of VLSI in algebraic computations: some suggestions, in Proc. of the 1981 ACM Symposium on Symbolic and Algebraic Computation, ACM Press (1981), 218-222

[Kun82] H.T. KUNG, Why systolic architectures, IEEE Computer 15, 1 (1982), 37-46

[KLe] H.T. KUNG, C.E. LEISERSON, Systolic arrays for (VLSI), Proc. of the Symposium on Sparse Matrices Computations, I.S. Duff et al. eds, Knoxville, Tenn. (1978), 256-282

[LS] C.E. LEISERSON, J.B. SAXE, Optimizing synchronous systems, in Proc. 22-th Annual Symposium on Foundations of Computer Science, IEEE Press (1981), 23-36

[Mol] D.I. MOLDOVAN, On the design of algorithms for VLSI systolic arrays, Proceedings of the IEEE 71, 1 (1983), 113-120

[Qui] P. QUINTON, The systematic design of systolic arrays, in Automata networks in computer science: theory and applications, F. Fogelman et al eds., Manchester University Press (1987), 229-260

[RT] Y. ROBERT, M. TCHUENTE, Réseaux systoliques pour des problèmes de mots, RAIRO Informatique Théorique 19, 2 (1985), 107-123

[SS] J. L.A. van de SNEPSCHEUT, J.B. SWENKER, On the design of some systolic algorithms, Computing Science Note 87/05, University of Groningen, The Netherlands (1987)

[Ver] T. VERHOEFF, A parallel program that generates the Möbius sequence, Computing Science Note 88/01, University of Groningen, The Netherlands (1988)

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 438 - A PROPOS DE LA RESOLUTION D'UN SYSTEME LINEAIRE DANS UN CORPS FINI : ALGORITHMES ET MACHINES PARALLELES
Hervé LE VERGE, Patrice QUINTON, Yves ROBERT, Gilles VILLARD
22 Pages, Novembre 1988.

PI 439 - ALPHA DU CENTAUR : A PROTOTYPE ENVIRONMENT FOR THE DESIGN OF PARALLEL REGULAR ALGORITHMS
Pierrick GACHET, Patrice QUINTON, Christophe MAURAS, Yannick SAOUTER
20 Pages, Novembre 1988.

PI 440 - CONSTRUCTION METHODIQUE D'UN ALGORITHME REPARTI DE DETECTION DE LA TERMINAISON
Jean-Michel HELARY, Michel RAYNAL
18 Pages, Décembre 1988.

PI 441 - LES GRAPHES A MOTIF
Didier CAUCAL
46 Pages, Décembre 1988.

PI 442 - CAUSAL TREES
Philippe DARONDEAU, Pierpaolo DEGANO
44 Pages, Décembre 1988.

PI 443 - TROIS IMPLANTATIONS DU RECUPERATEUR DE MEMOIRE DE LA MACHINE MALI
Michel LE HENAFF, Hervé SANSON
118 Pages, Décembre 1988.

PI 444 - ANALYSE FACTORIELLE LISSEE ET ANALYSE FACTORIELLE DES DIFFERENCES LOCALES
Brigitte ESCOFIER, Habib BENALI
34 Pages, Décembre 1988.

PI 445 - MULTISCALE STATISTICAL SIGNAL PROCESSING
Michèle BASSEVILLE, Albert BENVENISTE
16 Pages, Décembre 1988.

PI 446 - MODELES STATISTIQUES TEMPS-ECHELLE EN TRAITEMENT DU SIGNAL
Michèle BASSEVILLE, Albert BENVENISTE
60 Pages, Décembre 1988.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique