Bit-serial Systolic Squaring Algorithms

Mark R. Kramer

RUU-CS-88-20
April 1988

Bit-serial Systolic Squaring Algorithms

Mark R. Kramer

Technical Report RUU-CS-88-20
April 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.089, 3508 TB Utrecht
The Netherlands

Abstract

In this report a number of bit-serial systolic squaring algorithms are derived by systematic reasoning. Two algorithms are worked out in detail. The algorithms are especially well suited for hardware implementation.

The space and time complexity of all algorithms presented here is linear in the length of the numbers to be squared. This compares favourably to systolic bit-parallel squaring, where the space complexity is quadratic, while the time complexity is linear. Furthermore, the systolic arrays presented here are easy to expand to process numbers of greater length.

1 Introduction

With increasing densities of integrated circuits ever more complex functions may be offered in hardware at ever lower costs.

At the same time, to take advantage of integration, it is necessary to make regular designs using many identical simple elements. Although some global communication structures may be implemented efficiently, local communications on grids or arrays are necessary to exploit the full parallelism offered in VLSI. Kung introduced the idea of systolic algorithms to meet these requirements ([Ku79,KL79]).

A systolic algorithm consists of an array or grid of almost identical cells. These cells operate in lock step and communicate synchronously with neighbouring cells. Usually data is pipelined through the array like blood is flowing through the arteries (hence the name "systolic"), but in some systolic algorithms global broadcasting is used as well. External I/O is usually performed at the boundaries of the array by special cells that behave to the rest of the array as if they were ordinary systolic cells.

In [Hu86] a bit-parallel systolic squaring algorithm is presented. Its time complexity is linear in the length of the numbers to be squared, while its space complexity is quadratic. This algorithm was designed to be used as part of a 3D-graphics engine, but it turned out to be faster than necessary. The question arises whether it is possible to find a slower algorithm by saving a significant amount of hardware.

This is a special instance of the trade-off between time and space (area or memory) that occurs at many places in algorithm and hardware design. At the lower levels of hardware-design this trade-off is encountered as the choice between bit-parallel and bit-serial computations. In this report we present a number of bit-serial algorithms that use only a linear amount of hardware. The processing time is increased by a constant factor, compared to the bit-parallel algorithm.

Two of the algorithms are worked out in detail in section 5 (see also figures 8 and 9). In sections 3 and 4 these algorithms, together with a whole family of related approaches, are derived in a systematic manner. The following section gives some preliminaries from which we start the investigations.

Section 6 states the space and time complexity of the algorithms presented. Finally, in section 7 some possible other approaches are briefly described.

2 Squaring, multiplication and convolution

Taking the square of a number (in any representation) is just taking the product of that number with itself, so one could also use a multiplication algorithm to square a number. However, because of the additional information that the operands are identical, we can hope to save time and/or space by using a special squaring algorithm. The algorithm presented in [Hu86] in fact gives a reduction of a factor 2 in space and a factor 1.5 in time as compared to a parallel systolic multiplier.

As we will show now there is a relation between multiplication and convolution. This is particularly important because a number of serial systolic convolution algorithms have been published, on which we would like to build.

2.1 The relation with convolution algorithms

Multiplication of binary numbers (or numbers in any other positional representation) is related to convolution of vectors in the following way:

Convolution of a vector x ($=\{x_i\}$) by a vector w ($=\{w_i\}$) to form vector y ($=\{y_i\}$) is defined by

$$y_i = \sum_j w_j \cdot x_{i-j}$$

where all non-existing w_j's and x_{i-j}'s are treated as 0.

Alternatively this may be written as

$$y_k = \sum_{i+j=k} w_i \cdot x_j$$

A similar pattern is found in multiplication of binary numbers $a = \sum_i a_i \cdot 2^i$ and $b = \sum_i b_i \cdot 2^i$ to form the product $p = \sum_i p_i \cdot 2^i$. This product may be written as

$$p_k = \sum_{i+j=k} a_i \cdot b_j + c_k$$

In this formula c_k denotes the carry from lower order bits. In fact the p_k obtained has to be split in a result bit and a carry c_{k+1}, which in turn will be used in computing p_{k+1}. By the use of special full adders as in [Hu86] each temporary result, as it is generated, may be split in a least significant bit and a carry. Thus binary multiplication may be viewed as convolution plus carry-handling.

In [Ku82] Kung gives a number of systolic convolution algorithms, all based on a linear array of cells. The weights (w, i.e. one operand), inputs (x, i.e. the other operand) and results (y) each constitute a stream of data elements (w_i, x_j and y_k respectively). Each of these streams may be moving through the array independently. In some of these algorithms one of the operands is stationary, however, while in some others the results are stationary. We could easily adapt any of these algorithms to the multiplication problem, provided that the results flow in a direction such that lower order results come first. This is necessary to be able to process the carries appropriately.

Unfortunately, in each of Kung's convolution algorithms where the results move in the desired direction, one of the operands is stationary. This would result in a multiplication algorithm in which one of the operands is to be preloaded in parallel. In a squaring algorithm preloading is not strictly necessary, since both operands are identical in that case. Instead, the stationary operand may be loaded from the moving operand. However, this would rise the need for additional control. Therefore such a convolution algorithm does not seem the right starting point for an efficient squaring algorithm.

In the next sections squaring algorithms are presented in which results as well as the two operands are moving. Because those algorithms rely on the fact that identical operands are used it may be hard to generalise them to convolution algorithms in which operands as well as results move.

2.2 Optimisations for squaring

To obtain an efficient bit-serial squaring algorithm we need something like a convolution algorithm in which the results as well as both operands are moving. To be able to process the carries appropriately, the streams should be moving in a direction such that lower order bits precede higher order bits.

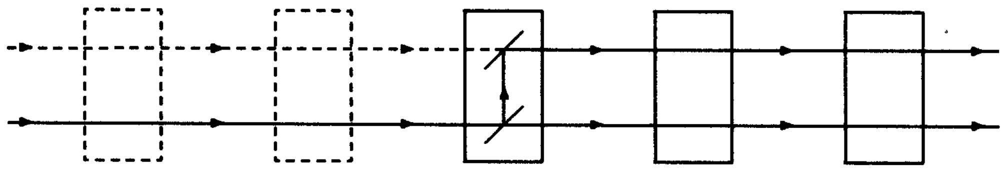
Furthermore we want to take advantage of the fact that the operands are identical. Firstly, we observe that the product $a_i \cdot a_j$ is computed twice unless $i = j$, so we might as well compute it only once, giving it double weight for $i \neq j$. Secondly, bit i of the first operand stream meets bit i of the second stream exactly once. Therefore we may obtain the second stream by copying each a_i from the first stream at the place where the a_i's would meet otherwise. Now a_i is not present in the second stream until it is copied, so a part of the products will not be computed. But, by the necessary regularity of the computation, those absent products will be products of a_i with bits a_j which either are all higher ordered or are all lower ordered. In either case these product bits will be computed from the copied a_j and the a_i from the complete stream.

Thus we obtain the following situation:

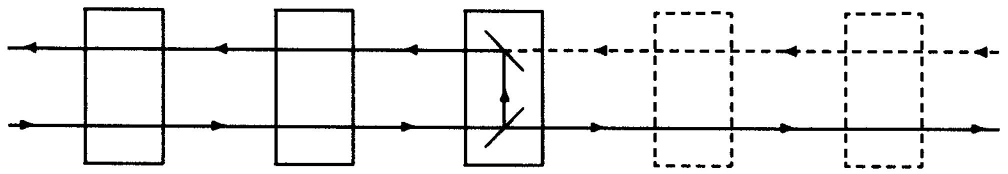
- One operand stream is moving such that the lower order bits precede the higher order bits.
- A second operand stream is constructed by copying from the first stream at appropriate positions.
- In any element where valid bits from both streams are available a product bit with double weight is computed, except for the element(s) where copying takes place.
- In the copying element(s) a product bit with single weight is computed.
- Finally, all product bits are collected in a result stream which should also move in a direction such that lower order bits precede higher order bits.

Now, by shifting the result stream, we may compute all product bits with single weight, giving the special product bits half weight. In fact these special product bits have to be added at the preceding position in the result stream.

Now there are still many possibilities, so we may make additional assumptions.



a. Streams moving in same direction



b. Streams moving in opposite directions

Figure 1: The main approaches

3 Data stream considerations

We will now first concentrate on those aspects of the squaring algorithms that do not depend on the details of the computation steps. In fact in this section we will consequently argue about *digits* instead of *bits*. In particular this has the effect that if technological considerations lead to the conclusion that e.g. byte-wise operations are more efficient, only details of the resulting algorithms need to be changed.

The discussion in the previous section led to the situation of one operand stream that is copied to a second (identical) operand stream at appropriate positions in a linear array of cells. Because this leaves many possibilities we make the (more or less arbitrary) choice that there is just one single cell in which copying occurs. Thus we need only one special element in the resulting algorithm. It is possible, however, that other choices result in less complex cells. We will briefly discuss other possibilities in section 7.

Consequently, in the following algorithms there is only one cell in which each digit meets its copy. So this will be the only place to take care of weights of partial results. We have to distinguish between the case that both streams move in the same direction and the case that they move in different directions. Figure 1 gives the two resulting situations. The special cell is indicated by small mirrors that split the original stream into two copies. We see that the special cell is always at an end (either the "start" or the "end") of the resulting array.

We will investigate the two situations separately. Before analysing these two cases, we make some observations that apply equally to both situations.

In what follows we will consider the speeds of data streams. The *data rate* of a stream at some point is the amount of data passing that point per time unit (usually a clock period). The *speed* of a stream is the mean distance (in number of cells) the data elements move per time unit.

Now, for any stream that passes unaltered through a cell, the rate of data entering the cell should be equal to the rate of data leaving the cell, since there is only a fixed amount of memory in a cell. For the same reason the data rates of the two operand streams should be equal by our assumption that copying takes place at one fixed cell.

However, in the following we will conclude that the operand streams should have different speeds. Then the only way to maintain the equal data rates is to pass one stream through more pipeline latches than the other.

From the speeds of the operand streams the speed of the result stream may be obtained as will be shown at the appropriate points in the discussions. Here we show that the *data rate* of the result stream should however always be twice the data rate of the operand streams. This is a consequence of the fact that the length of the result is twice the length of the operand.

Let X and Y be the number of latches per cell in the first (original) and second (copied) stream respectively. At a certain moment a cell will contain the elements $a_k, \ldots, a_{k+X-1}$ and $a_l, \ldots, a_{l+Y-1}$ giving partial product digits $p_{k+l}, \ldots, p_{k+l+X+Y-2}$ (or a subset thereof). After the operand streams have moved by one position that cell will contain $a_{k+1}, \ldots, a_{k+X}$ and $a_{l+1}, \ldots, a_{l+Y}$ giving $p_{k+l+2}, \ldots, p_{k+l+X+Y}$. So when the operands have moved by one position, the results must have moved by two positions. This is another way of saying that the data rate of the result stream is always twice the data rate of the operand stream.

Now we turn our attention to the two distinct relative directions of movement for the operand streams. Because at first sight it seems easier to find an algorithm in which the streams move in different directions, we will discuss that approach first.

3.1 Different directions

If both streams move with the same speed the first problem will be to make sure that every element of the first stream meets every element of the second stream. This is not really a serious problem. But even if this problem is solved, another problem remains: When a_0 meets a_i, then at that same moment a_1 meets a_{i-1}, a_2 meets a_{i-2} and so on, all giving a contribution to the same result digit. But then it is far from straightforward (or even impossible) to assemble these product digits in a regular way by means of local communications.

Therefore the streams should have different speeds, so one stream should pass through more delays (i.e. latches) per cell. Now we still have to find a suitable number of delays such that results may be assembled in a regular way.

Now we return to the problem that every digit has to meet every other digit. Suppose that consecutive digits from a stream are stored in consecutive latches in the array of cells. Furthermore suppose that all these latches are clocked at the same moment. Then the situation occurs that in some two neighbouring cells A and B there is an element a_k in A to be transferred to B in the next clock-cycle, and an element a_l in B to be transferred to A at the same time. But then a_k and a_l will never meet.

So we conclude that either there have to be empty slots in the streams or not all latches should be clocked at the same moment. Unless a multi-phase clocking scheme is used, the second option is impractical. On the other hand, if a multi-phase clocking scheme is used we could as well view every phase as a separate clock period (from an algorithmic point of view). As a result empty slots in the streams are created between latches that are clocked in different phases. In the following discussion we therefore assume that all latches are clocked at the same time, and that in the streams dummy elements are included to ensure that all proper elements meet each other.

By the necessary regularity of the systolic algorithm to result, we get a stream in which proper elements and dummy elements are interleaved in a regular way. The simplest form conceivable consists of a stream in which proper elements and dummy elements (denoted by Δ henceforth) alternate. This approach was also used in some of the algorithms presented in [Ku79]. It turns out that this simplest form leads to an acceptable result. Because more complex forms will lead to more complex cells as well, we will not consider other possibilities.

It is easy to see that the number of delays per cell should be odd for at least one of the operand streams. Now, if the other stream has an even number of delays per cell, the same argument as before shows that at one of the boundaries elements pass without giving a contribution to the final product. Therefore the number of delays per cell should be odd for both operand streams. Furthermore, the two streams should have different speeds, so the simplest approach has one delay per cell for one stream and three for the other.

Now there are again two possibilities: either the three delays are in the original stream, or they are in the reflected stream. This is not a serious difference however. Suppose we forgot which stream was the original one, then there is no means to find out except by looking at either end of the array of cells. In all intermediate cells there is no noticeable difference between the two cases. We will now first study the common part in these two solutions.

Without loss of generality we assume that the stream that passes through three delays per cell moves to the left. Consider two cells A and B, with B right of A (see figure 2). Suppose at time t cell A contains the elements a_{k-1}, Δ, a_k and B contains Δ, a_{k+1}, Δ. Then B should contain a dummy element from the other stream, because a proper element a_l in B would still pass a_k without a contribution to the product. So cell A contains a_l from the second stream whereas B contains Δ.

Therefore at time t the partial results p_{k+l-1} and p_{k+l} may be computed in A, while B gives no results. At time $t+1$ cell A contains Δ, a_k, Δ and Δ, while B contains a_{k+1}, Δ, a_{k+2} and a_l, giving partial results p_{k+l+1} and p_{k+l+2} in B, while A gives no results. At time $t+2$ cell A contains a_k, Δ, a_{k+1} and a_{l+1}, while B contains Δ, a_{k+2}, Δ and Δ, giving partial results p_{k+l+1} and p_{k+l+2} in A, while again B gives no results.

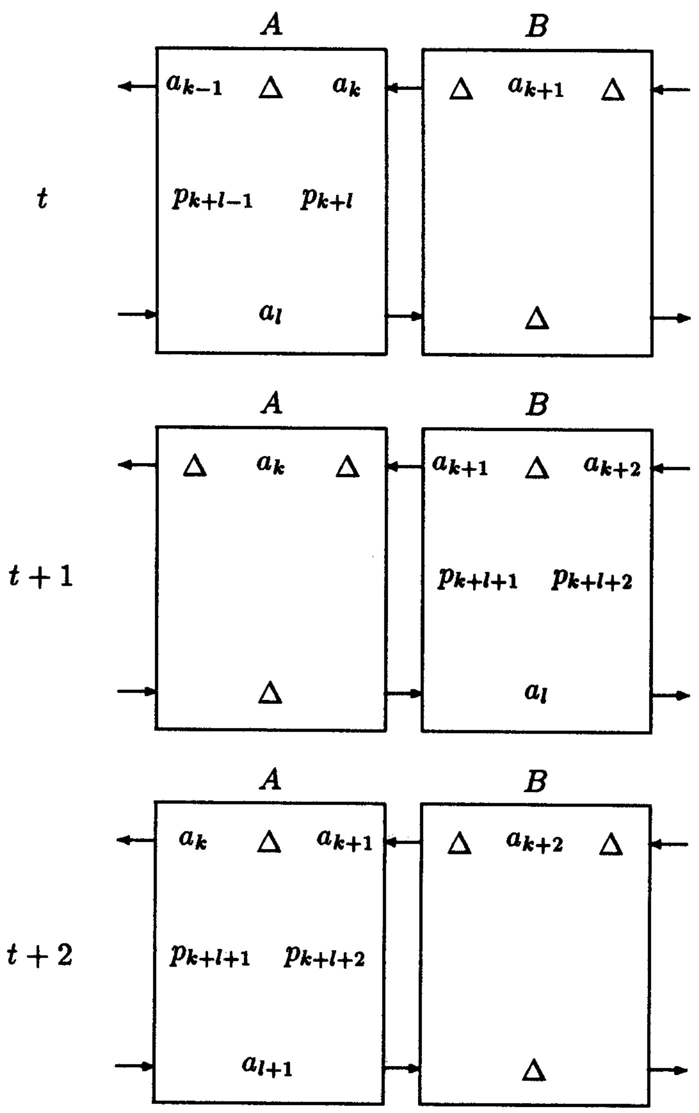


Figure 2: Snapshots for arrays with operands moving in different directions. Note: p_{k+l} stands for a contribution of $a_k \cdot a_l$ to p_{k+l} etc.

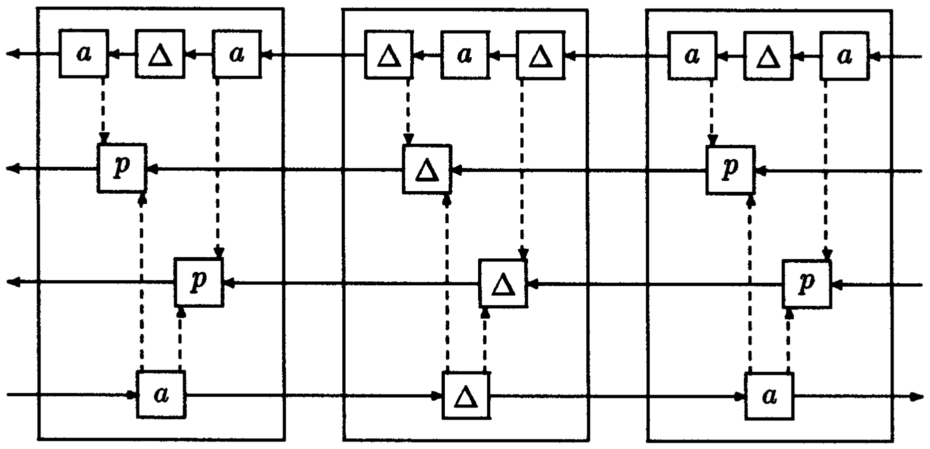


Figure 3: Symbolic representation of the algorithms with data moving in different directions

We see that the result digits move in the same direction as the slower stream, where two digits at a time move one cell per time unit. So the *speed* of the result stream is 1, while the *data rate* is also 1 (being twice the effective data rate of the operand streams). Hence the number of delays per cell for the result stream should be 1 too.

Now it is convenient to incorporate dummy elements in the result stream as well, i.e. the cells not giving valid results compute dummy results. But then the number of delays has to be doubled, while the speed of the result stream should not be altered. This can be achieved by splitting the result stream into two independent streams, one for result digits with odd index and one for result digits with even index. The data flow of this algorithm is depicted in figure 3.

The discussion up to this point was independent of the choice of which operand stream was the slower one. As indicated before the difference occurs at the ends of the array. We have shown that the result stream moves in the same direction as the slower of the two operand streams. If the original stream is the slower one, then we conclude that the results travel towards the cell where the stream is reflected. So results have to leave the array at the other end than where operands are input in this case. In the other case results leave the array at the same end where operands are input. Because the latter solution needs only one special I/O-cell, it has some preference over the other solution. In section 4.2 we will find a second reason why the latter solution is better.

So we prefer the case where the result stream and the original operand stream move in different directions. This solution is obtained if the original stream passes through one delay per cell, while the reflected stream passes through three delays.

In section 4 we will study these solutions in more detail.

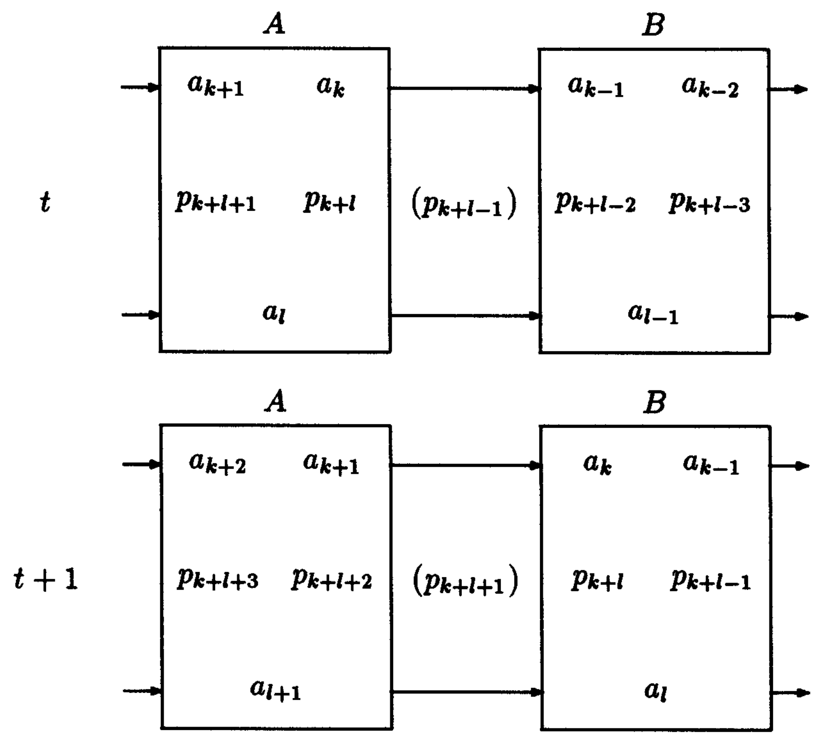


Figure 4: Snapshots for arrays with operands moving in the same direction

3.2 Same direction

Now we turn our attention to the other main approach, where the two operand streams move in the same direction.

It is easy to see that two streams moving at the same speed in the same direction will not give all product digits. Again the data rate should be the same for both streams. So one of the operand streams has to pass through more delays, as in the case for streams moving in different directions, and again we have to find a suitable number of delays per cell.

In contrast to the previous situation, now there is no a priori reason to incorporate dummy elements in the operand streams. If there are no dummy elements at all, it is not necessary that there is an odd number of delays per cell in each stream. Consequently, the simplest case when the operand streams move in the same direction is the situation where each cell has one delay for one stream and two for the other.

Now, since both streams move in the same direction, there is no difference whatsoever between original stream and copied stream. In fact the copying cell is the cell where data enter the array, and the two streams are just two copies of the incoming data. Again we have to find out the speed of the result stream. So consider two cells A and B, with B to the right of A (see figure 4). We assume operands move to the right.

Suppose at time t cell A contains the elements a_{k+1}, a_k from one stream and a_l from the other stream. Then B contains a_{k-1}, a_{k-2} and a_{l-1}. So at time t the partial results p_{k+l+1} and p_{k+l} may be computed in A, while B gives p_{k+l-2}, p_{k+l-3}. At time $t+1$ cell A contains a_{k+2}, a_{k+1} and a_{l+1}, while B contains a_k, a_{k-1} and a_l, giving partial results p_{k+l+3}, p_{k+l+2} in A and p_{k+l}, p_{k+l-1} in B.

From this information we may calculate the speed of the result stream and so obtain the number of delays to be included per cell in the result stream. The only contributions to the same product digit here are those to p_{k+l}, so we are tempted to conclude that the speed of the result stream is 1. In reality p_{k+l} has only travelled $\frac{2}{3}$ cell in one time unit, however.

To find the speed of the result stream we might either extend the foregoing analysis to three cells and three time steps, or trace all product digits, including 'invisible' ones. In the latter approach there is an invisible product digit between A and B: at time t this is p_{k+l-1}, at time $t+1$ it is p_{k+l+1} (see figure 4). Now, in the resulting row of product digits all digits move two positions to the right in a time step, while there are three product digits per cell (two visibles and an invisible). So the distance a digit moves per unit time is two-thirds of a cell.

As we have seen before, the data rate of the result stream should be 2, so the number of delays per cell should be 3 for the result stream. Note that in fact in the analysis we just calculated the speed from the data rate and the number of delays, which we derived to be 2 and 3, respectively.

4 Further exploration of the algorithms

Now we have found three algorithm skeletons based on the assumption that stream copying is done in only one cell. We still have to fill in all details concerning the exact computational operations.

In this section we will refer to the different algorithms as algorithm Ia, Ib and II. Algorithms Ia and Ib have operand streams moving in different directions, with the reflected stream passing through three delays per cell in Ia and one delay per cell in Ib. Where the difference between Ia and Ib is not relevant we will simply write algorithm I.

In algorithm II the operand streams move in the same direction.

4.1 Finding all contributions to the product

So far we have not even confirmed that every bit meets every other bit. We have only considered necessary conditions but it is not yet clear that these conditions are also sufficient. Therefore we have to prove that in all proposed skeletons every bit a_i meets every bit a_j at least once. If $i = j$ then clearly a_i meets a_j, namely in the copying cell. Now suppose without loss of generality that $i > j$. For $i - j$ sufficiently small it is always possible to force a_i and a_j to meet by taking the special cell complex enough. So suppose that we already know that a_i meets a_j for some $i \geq j$. Now we have to prove that a_{i+1} meets a_j and a_i meets a_{j-1}. In fact it is sufficient to prove either one of these.

Looking back at figures 2 and 4 we consider each of the proposed skeletons separately:

- Let in algorithm Ia a_i and a_j meet in cell A at time t.
  Then $a_j = a_l$ and either $a_i = a_{k-1}$ or $a_i = a_k$.
  In the first case $a_{i+1} = a_k$ meets $a_j = a_l$ at time t in cell A.
  In the second case $a_{i+1} = a_{k+1}$ meets $a_j = a_l$ at time $t+1$ in cell B

- Let in algorithm Ib a_i and a_j meet in cell B at time $t+1$.
  Then $a_i = a_l$ and either $a_j = a_{k+1}$ or $a_j = a_{k+2}$.
  In the first case $a_i = a_l$ meets $a_{j-1} = a_k$ at time t in cell A.
  In the second case $a_i = a_l$ meets $a_{j-1} = a_{k+1}$ at time $t+1$ in cell B.

- Let in algorithm II a_i and a_j meet in cell A at time t.
  Because the lower stream moves faster, a_l is newer than a_{k+1} and a_k, i.e. $l \geq k+1$. So $a_i = a_l$ and either $a_j = a_{k+1}$ or $a_j = a_k$.
  In either case $a_i = a_l$ meets $a_{j-1} = a_k$ or $a_{j-1} = a_{k-1}$ at time $t+1$ in cell B.

So by induction over $i-j$ we prove that indeed every a_i meets every a_j.

Furthermore from this analysis we find the number of cells needed to square an n-bit number. Assume that the copying cell is as simple as possible. So the only a_i and a_j that meet there have $i = j$. Then $n-1$ steps of the foregoing analysis are needed. In algorithm Ia as well as algorithm Ib the elements meet in the next cell at every other step of the analysis. Therefore $n/2$ cells are needed in total. In algorithm II a new cell is encountered at every step, so in that case n cells are needed. If the special cell is more complex such that more a's meet there then the number of cells needed decreases by a (small) constant.

Now we know that every operand bit meets every other operand bit at least once. But of course we need only one contribution of $a_i \cdot a_j$ to the final result for any i and j. From figure 2 it is easy to see that every a_i meets every a_j at most once and hence exactly once. We conclude that in algorithm I at every occasion a product bit should be computed.

But for algorithm II we see from figure 4 that a_l meets a_k twice (in fact: at least twice, but it is easy to show that they meet no more). So in this case not at *every* occasion a product bit should be computed. Note that in any cell two product bits may be computed. Call them the left and right product bit. Then the left product bit of cell B at time $t+1$ is the same as the right product bit of cell A at time t. This holds for all cells except at the ends of the array. So it is sufficient to compute in all cells either the left product bit or the right product bit. If all left product bits are computed an exception occurs at the right end of the array, where the last right product is not taken over by a next cell. If all right products are computed

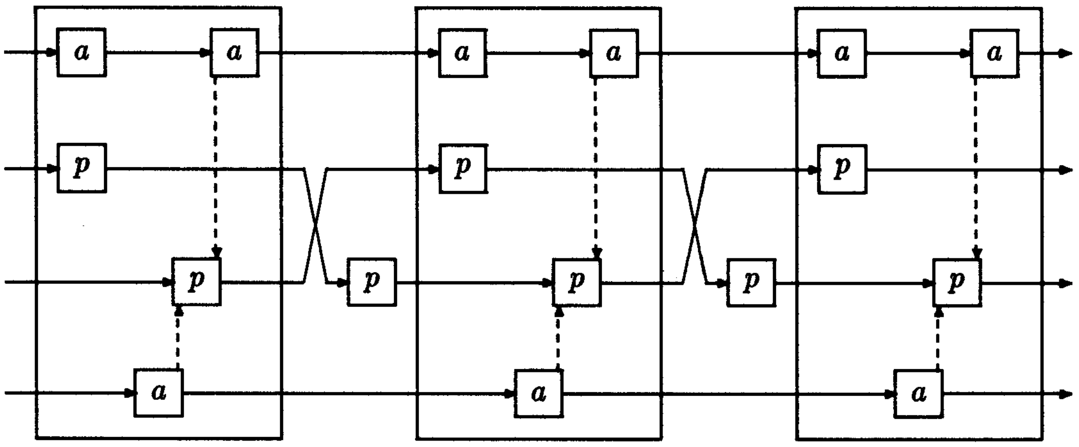


Figure 5: Symbolic representation of algorithm II

the exception occurs at the left end. Since the cell at the left end is already special, we prefer the latter case.

This leads to the situation of figure 5. Note that figure 3 may be viewed as the corresponding figure for algorithm I. The choice which cell contains a certain latch is arbitrary if the contents of that latch is not involved in the computation. So in fact it does not really make a difference whether the left products or the right products are computed, the only difference being the placement of cell boundaries.

4.2 Assembling the product bits

Now we are ready to concern the problem of how to assemble product bits. In each internal cell a product bit is received that depends on a part of the operand bits. The cell adds the product of just one other pair of operand bits to this partial result and sends that result bit to the next cell. Let a_i and a_j be the operand bits used in this cell; call the incoming product bit p'_{i+j}, and the outgoing product bit p''_{i+j}. Then naively we find

$$p''_{i+j} = p'_{i+j} + a_i \cdot a_j$$

But as indicated in section 2.2 this might result in a carry, which has to be added to the next product bit. Consequently, we have to include a carry (c_{i+j}) from the previous product bit as well, giving:

$$c_{i+j+1} : p''_{i+j} = p'_{i+j} + a_j \cdot a_i + c_{i+j}$$

where $c : p$ denotes $2c + p$ for bits c and p.

The carry c_{i+j+1} has to be added to p'_{i+j+1}, possibly giving a carry c_{i+j+2}. This carry in turn has to be added to p'_{i+j+2}. That can always be done in the same cell at a later time (one or two clock periods later, depending on which skeleton is used), when

$$c_{i+j+3} : p''_{i+j+2} = p'_{i+j+2} + a_{i+1} \cdot a_{j+1} + c_{i+j+2}$$

is computed. Therefore we may assume that all carries are kept locally. Note, however, that many cells contain their own instances of c_{i+j} in the course of a computation. Now, at any time three bits have to be added together in a cell, so the result can always be represented by one carry-bit and one result bit. In this way no further complications arise, so no additional carry bits are needed in a cell.

The derived computation is shown in figure 6a. In algorithm II this computation is performed in every cell. In algorithm I two pairs of operand bits per cell are multiplied in the same clock period, so there a more complex computation is performed. This computation is shown in figure 6b, but it has been reflected to be able to use it directly in later figures.

As argumented in section 2.2, the special cell that takes care of copying the operand stream should compute product bits with half the usual weight. This means that the partial product bit computed by this cell should be assembled in the result stream at a position *preceding* the position where another cell would have placed it. But then suddenly four bits have to be added instead of three: an incoming partial product, a normal product bit, the carry from the lower-ordered bits and the half-weighted product bit. So this special cell should keep two carry bits and perform a more complex addition, unless we can get rid of one of the bits to be added.

Now notice that the partial product bits that depend on *no* operand bits so far, have to enter the array as zero's. Moreover, they have to enter the array at that end from which the results travel through the array. If the special cell is at that end of the array, we may reduce the complexity of the special cell by discarding the incoming partial product.

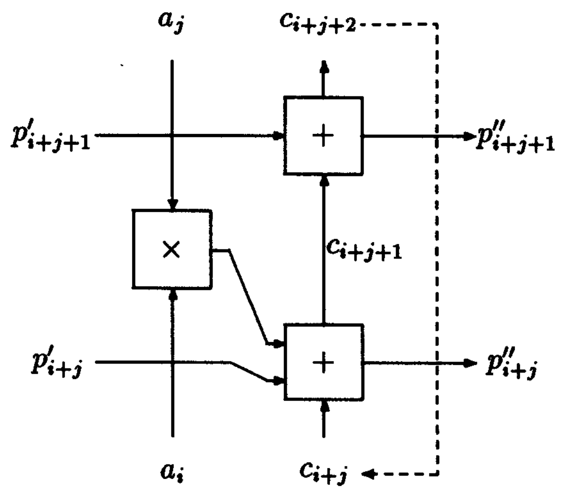
Indeed this situation occurs in algorithm Ia as well as algorithm II, but not in algorithm Ib. So here we have the additional argument, referred to in section 3.1, in favour of a slower copied stream. Henceforth we will not go into further details of algorithm Ib.

To be able to add the half-weighted result at the right place we have to compute the preceding result bit in the special cell as well. The resulting computations for the special cell are given in figure 7. Since the upper adder now has only one input its sum output is identical to its (single) input, while the carry output is always zero. So this adder may as well be removed, while fixing c_{2i-1} (i.e. the delayed carry output) at 0. Then in figure 7a the other adder becomes superfluous too. Furthermore $a_i \cdot a_i = a_i$ for single-bit a_i, so the corresponding multipliers may be removed.

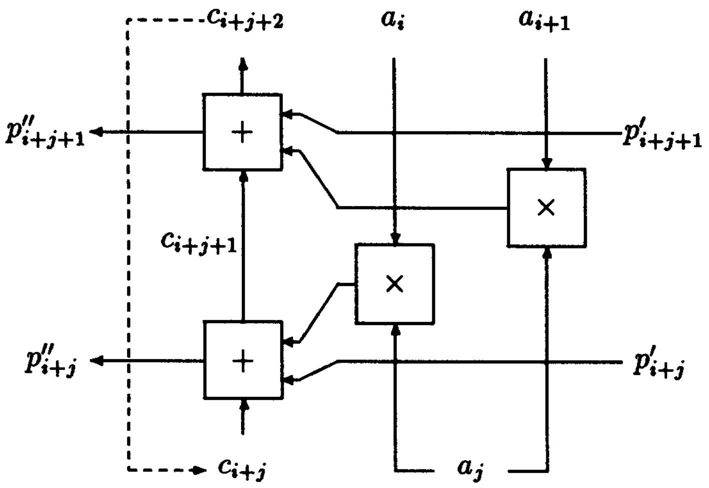
5 The algorithms

In the preceding sections we derived all parts for two bit-serial systolic squaring algorithms. In the following subsections we put together all parts to form the final algorithms.

To complete the algorithms we also have to specify the external behaviour of

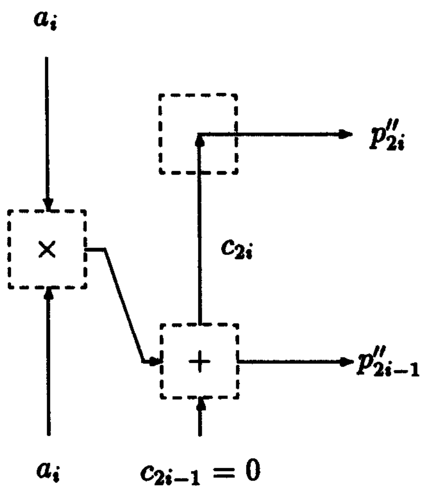


a. algorithm II

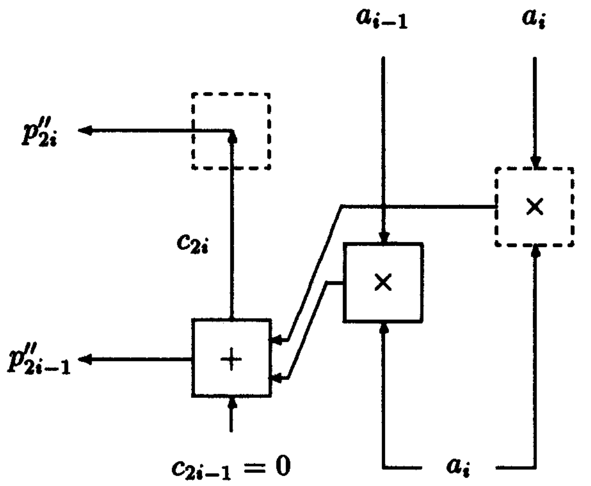


b. algorithm I

Figure 6: Computations performed by internal cells
See figure 10 for an explanation of the symbols used



a. algorithm II



b. algorithm I

Figure 7: Computations performed by the special cell
See figure 10 for an explanation of the symbols used

the systolic arrays. This consists of two parts: I/O-cells and timing of inputs and outputs. The function of I/O-cells is to supply inputs to and accept outputs from the rest of the array in the same format as used for internal communications. Thus there need not be any difference between a boundary cell and an internal cell. Furthermore the I/O-cells might perform parallel-to-serial and serial-to-parallel conversions and truncations on the result, if necessary. We will not consider I/O-cells in more detail, because their design is implementation dependent.

Then the only problem left is the timing of inputs and outputs. As we have seen in section 3.1 inputs and outputs of algorithm I are interleaved with dummy elements. If we do not make a distinction between proper and dummy elements, then the I/O-cells have to supply one input and to accept two outputs every clock cycle, for both algorithm I and algorithm II. Furthermore the input has to be padded with zero's, corresponding to a_i for $i < 0$ or $i \geq n$. This is necessary because every a will meet $n - 1$ other a's (at least). In section 6 we will find that the minimal number of zero's to be padded is $n-1$ for both algorithms. We will find other timing details too, such as the delay between input and output.

What was called algorithm Ib will not be considered any further, so we will write algorithm I instead of algorithm Ia from now on.

5.1 Algorithm I

In algorithm I operands enter the array at one end, travel through the array to the other end, where they are reflected. Then they travel back to the end from which they originated, where they leave the array. The original stream of operands passes through one latch per cell, while the reflected stream passes through three latches per cell. To ensure that every operand bit meets every other operand bit dummy elements are interspersed in the operand streams. For reasons of regularity dummy elements are included in the result stream as well. The result stream moves one cell per time unit, but since the data rate is twice the data rate of the operand stream, there are in fact two separate result streams. One result stream consists of result bits with even index and the other consists of result bits with odd index. The results leave the array at the same end where operands enter the array.

In figures 2, 3, 6b and 7b all constituent parts of algorithm I are given. Figure 8 shows the result of putting all parts together.

5.2 Algorithm II

In algorithm II operands enter the array at one end, where they are copied to form two identical streams that travel to the other end at different speeds. One stream passes through one latch per cell, the other through two latches per cell. Now there is no need for dummy elements in the streams. Again the result stream is in fact split in two, each passing through three latches per two cells. In every cell one product bit is computed, because the other pair of operand bits available have already met in the previous cell. Results move in the same direction as the operands, so they leave the array at the end opposite to where the operands enter the array.

In figures 4, 5, 6a and 7a all constituent parts of algorithm II are given. Notice, however, that the partial product bits computed in the special cell are delayed by one position in the total result stream. Therefore, the additional latch between the cells should be left out between the special cell and its neighbour. One way to do this is by including the additional latch in the cell to its left. But then the last cell will contain a superfluous latch. Another way is to include the additional latch in the cell to its right, and adjust the special cell. We prefer the latter approach. Therefore, in the resulting array (see figure 9) the product outputs of the special cell are interchanged. Since the first latch of the second cell should be omitted, the usual output latch of the special cell is omitted instead.

6 Complexity of the algorithms

In section 4.1 we derived that the number of cells needed to square an n-bit number is $\lceil n/2 \rceil$ for algorithm I and n for algorithm II. But these cells are not identical, so we should also compare the complexity of the cells.

In algorithm II one multiplication and two additions are performed per cell, whereas in algorithm I two multiplications and two additions are performed per cell. The total number of latches per cell is 8 in both cases (including two latches for the carries). Taking the cost of one multiplication, one addition and 4 latches as unit of space, the space complexity of algorithm I therefore is n, while the space complexity of algorithm II is almost $2n$, assuming bit-wise operations. The difference is one multiplication.

Since multiplication is equivalent to a logical AND in the case of bit-wise operations, a multiplication may be neglected compared to addition or a latch. But when operations are performed on larger numbers of bits at a time the cost of multiplication dominates the other costs for already fairly small word-sizes. Then the space complexity approaches n for both algorithms.

Now we turn to the time complexity of the algorithms. We have to distinguish between delay, computation time and latency. *Delay* is the time from the first operand bit input to the first result bit output. *Computation* time is the time from the first operand bit input to the last result bit output. *Latency* (or inverse throughput) is the time from the first operand bit of one computation to the first operand bit of the next computation. We take a clock period as unit of time here.

The delay for algorithm I is the time a_0 needs to travel to the copying cell plus the time p_0 needs to travel back. Now, a_0 passes through one latch per cell on its way to the copying cell, so $n/2$ clock periods are needed to reach it; p_0 also meets one latch per cell, so the total delay for algorithm I is n. In algorithm II a_0 reaches the copying cell immediately, so the delay is the time p_0 needs to travel through the array in this case. Since p_0 passes through three latches in every pair of cells, the total delay is $\lfloor \frac{3}{2}n \rfloor$ in this case.

The computation time is equal to the time between a_{n-1} is input and p_{2n-1} is output plus the time it takes to input all operand bits. Now p_{2n-1} is output in the same clock cycle as p_{2n-2}, which depends on a_{n-1} exactly like p_0 depends on a_0, so this part of the computation time is equal to the delay. In algorithm II p_{2n-1} is output one clock cycle after p_{2n-2} for odd values of n, however. The time it takes to input all operand bits is n for algorithm II, but $2n-1$ for algorithm I, because dummy elements must be included. Thus we find a computation time of $3n-1$ for algorithm I and $\lceil \frac{5}{2}n \rceil$ for algorithm II.

To find the latency we must ensure that the next computation starts at a time such that it is impossible that the previous computation is influenced by the new one. This means that no a_i from the next computation may meet an a_j from the previous one.

For algorithm I this means that a_0 of the second computation must not be entered before a_{n-1} of the first computation has left the array. So the latency in this case is the time a_{n-1} needs to travel all the way through the array plus the time it takes to input all operand bits. Now a_{n-1} passes through one latch per internal cell in one direction and through three in the other direction. In the special cell a_{n-1} passes through only three latches, so it takes $4 \cdot n/2 - 1 = 2n-1$ clock cycles for a_{n-1} to travel through the array. The total latency is therefore $4n-2$ for algorithm I.

For algorithm II a_0 of the second computation must not enter the last cell before a_{n-1} of the first one has left it. So in this case the latency is the time a_{n-1} needs to reach the right end of the array by the slow path, minus the time a_0 needs to reach it by the fast path, plus the time it takes to input all operand bits. These times are $2n-1$, n and n, respectively, giving a total latency of $2n-1$ for algorithm II.

The following table summarises these results for even values of n. The analysis of algorithm I depends on the number of cells, so the figures differ slightly for odd n. For algorithm II the figures for odd n may be found in the foregoing analysis.

	algorithm I	algorithm II
space	$\frac{1}{2}n$	n
delay	n	$\frac{3}{2}n$
computation time	$3n-1$	$\frac{5}{2}n$
latency	$4n-2$	$2n-1$

From the latency we may find the number of zero's to be padded between two successive computations. The first input of the latter computation may be supplied 'latency' clock cycles after the first input of the former one. In algorithm II an input is supplied every clock cycle, so in n out of $2n-1$ clock cycles a valid input may be supplied. Therefore there must be at least $n-1$ zero's between two successive computations. In algorithm I an input is supplied every other clock cycle, so the number of zero's is $(4n-2)/2 - n = n-1$ in this case too.

Since the latency determines the number of computations that may be performed in some given amount of time, this is the most important measure of time. Now, the latency of algorithm I is twice as large as the latency of algorithm II, while the situation for the space complexities is almost reversed. So the space-time product is the same in both cases.

However, in algorithm I only half of the resources is used at any time, so a saving of a factor two in space-time might be possible. There are two ways to achieve this. The first possibility is that the implementation compacts the cells by multiplexing resources or by using a two-phase clocking scheme. Recall that we converted a multi-phase clocking scheme to a situation with dummy elements in the streams in section 3.1.

The second possibility is to use the same hardware to perform two computations concurrently, one being the computation shown so far. Now all streams have proper elements and dummies alternating. Moreover, operations are always performed on either all proper elements or all dummies, so we might as well replace the dummies with elements from another operand, giving an second independent computation. Hence the latter computation lags just one clock cycle behind relative to the former, and the latter uses all resources not used by the former. By the foregoing we see that it is impossible that these two computations interfere.

7 Other solutions

On our way to find the algorithms presented in this report, we passed many sideways. Some of these apparently lead to less efficient solutions, but others are worth some investigation.

7.1 Stream copying

In section 2.1 we decided that the two operands as well as the result should be moving. Moreover in section 3 we made the choice that copying occurs in just one cell. If we make another choice we obtain different data rates for the operand streams. In fact a zero data rate for one stream may be obtained. Clearly, this is equivalent to the situation where one of the operands is stationary. Because the two operands are identical it is not necessary to load one of them in advance, however.

The space complexity of the cells in the resulting array is comparable to those of algorithm II, as far as data processing is concerned. But now some local control is needed too, because cells will have different operating states, depending on whether or not an operand bit has been stored already. Furthermore such an array has to be initialised before every computation. Consequently, the resulting array will be more complex than the arrays presented in the preceding sections.

The same disadvantages also occur in other algorithms in which copying is not restricted to a special cell. Because of the higher data rates encountered, the complexity of the cells increases even further in these cases.

7.2 Stream speeds

In section 3 we found that the speeds of the operand streams should be different. From that point on we concentrated on those speeds for which a minimum number of delays per cell was needed. Of course any other combination of speeds will work, as long as the additional requirements on the speeds are met. For each of those possibilities the steps of section 4 should be made again. But with an increasing number of delays per cell, the cells will become more complex, while the details will become more tedious too.

8 Conclusions

A number of bit-serial systolic squaring algorithms were derived. They are all based on a linear array of cells. Two algorithms were worked out in detail. The techniques used here may be applied to the others as well.

The space and time efficiency of all algorithms found is linear in the length of the numbers to be squared. This compares favourably to the bit-parallel approach in [Hu86], where the space complexity is quadratic, while the time complexities are linear. However, it is not hard to modify the bit-parallel systolic algorithm to obtain a constant latency, without increasing the space complexity significantly. This gives the same time-space product as our algorithms. Furthermore, it is extremely simple to expand the systolic arrays presented in this report to square larger numbers. Usually, it is less simple to expand two-dimensional systolic arrays.

Unlike many other systolic algorithms, the algorithms presented in this report were derived, in an informal manner, by systematic reasoning. On the one hand, many systolic algorithms in literature just appear from nowhere. In fact, algorithm I, in a slightly different form, was found by the author in the same way. Only by making clear all decisions the other algorithms were found too. On the other hand, systolic algorithms may be obtained by formal manipulation of some initial problem specification (see e.g. [RF87]). It would be an interesting exercise to follow that approach on this problem.

References

[Hu86] C. Huijs. A fast squaring algorithm for VLSI. In *Proceedings NGI-SION 1986 symposium*, pages 233–243, Stichting Informatica Congressen, Paulus Potterstraat 40, NL-1071 DB Amsterdam, 1986.

[KL79] H.T. Kung and C.E. Leiserson. *Systolic arrays for VLSI.* Technical Report CMU-CS-79-103, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1979.

[Ku79] H.T. Kung. *Let's design Algorithms for VLSI Systems.* Technical Report CMU-CS-79-151, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1979.

[Ku82] H.T. Kung. Why systolic architectures? *IEEE Computer*, 37–46, January 1982.

[RF87] S.V. Rajopadhye and R.M. Fujimoto. Systolic array synthesis by static analysis of program dependencies. In A.J. Nijman J.W. de Bakker and P.C. Treleaven, editors, *PARLE, Parrallel Architectures and Languages Europe, Volume I: Parallel Architectures (LNCS 258)*, pages 295–310, Springer Verlag, Berlin etc., 1987.

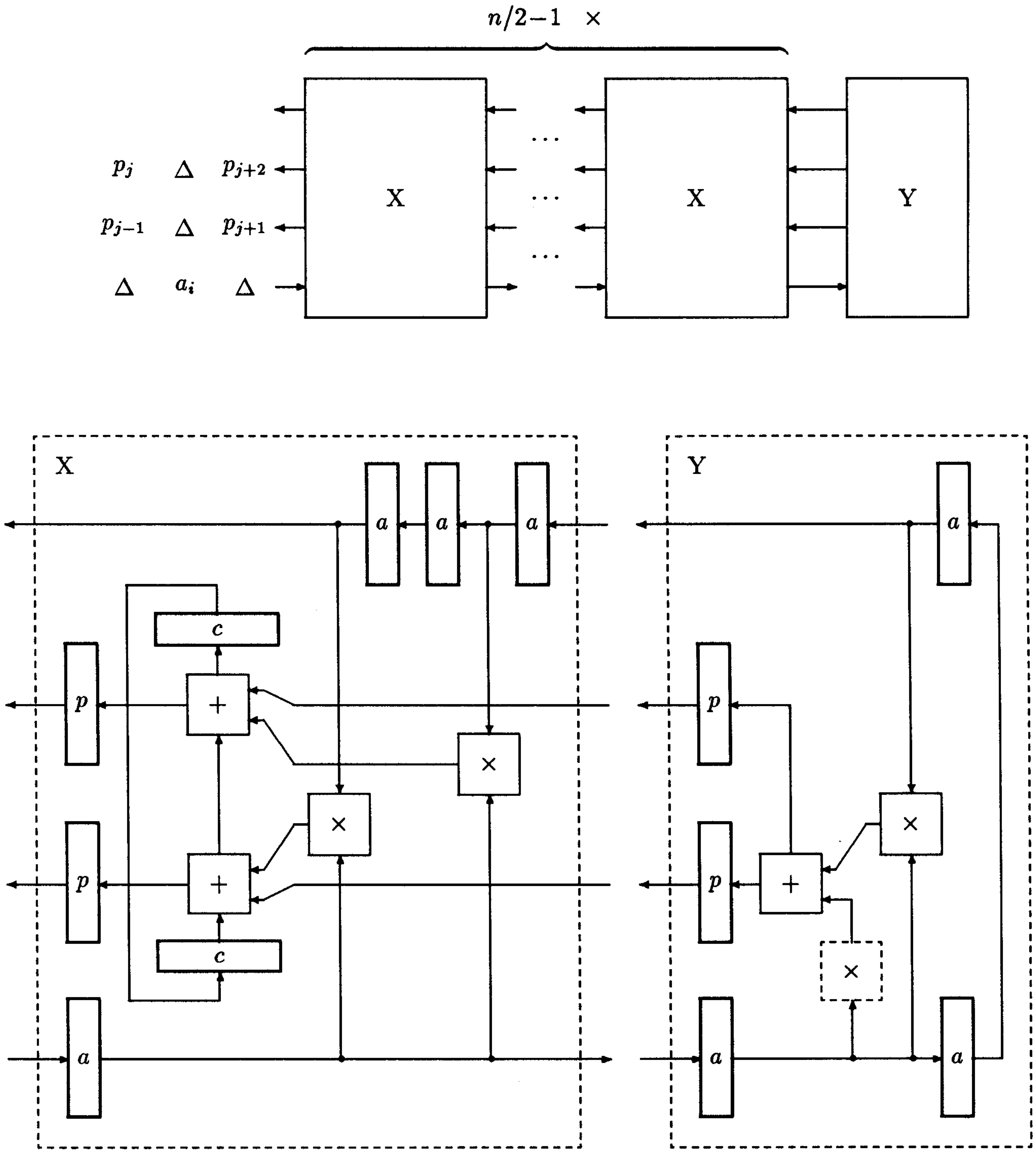


Figure 8: Algorithm I
See figure 10 for a key to the symbols

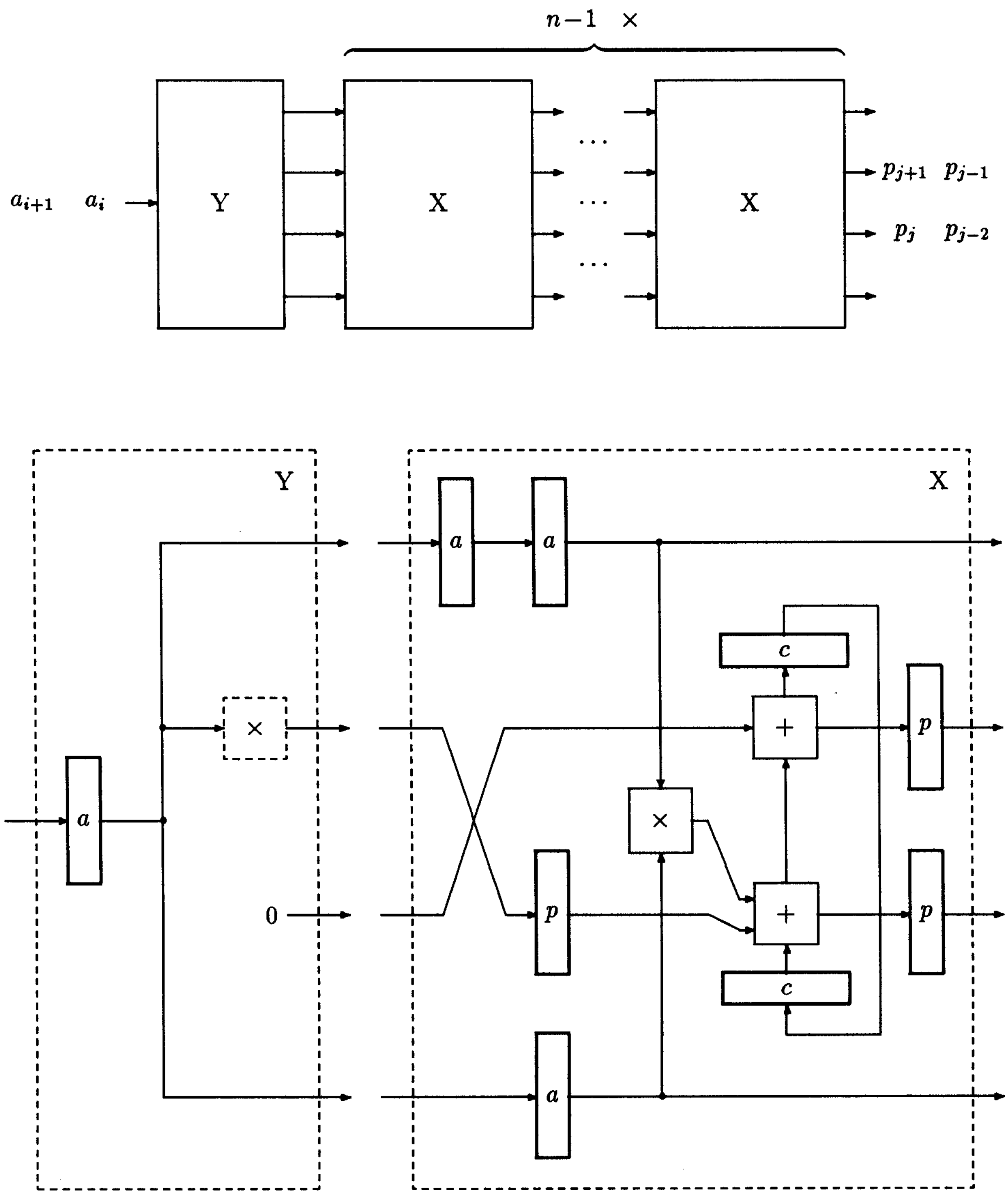
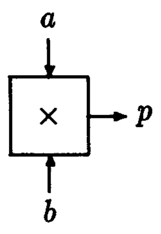
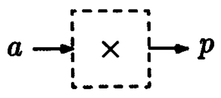


Figure 9: Algorithm II
See figure 10 for a key to the symbols

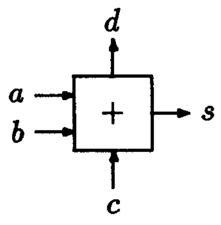


multiplier $p = a \cdot b$

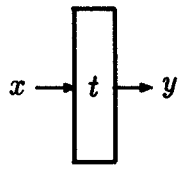


squarer $p = a \cdot a$

identity for bit-wise operations



full adder $d : s = a + b + c$



clocked latch $y = \text{delay}(x)$

t denotes the kind of information stored

$c_{i+j+2} \dashrightarrow c_{i+j}$ delayed feed-back (figures 6 and 7)

Figure 10: Key to figures 6, 7, 8 and 9