43589

National Library of Canada
Canadian Theses Division
Ottawa, Canada.
K1A 0N4

Bibliothèque nationale du Canada
Division des thèses canadiennes

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

- Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

ROBERT G. WILLONER

Date of Birth — Date de naissance

MAY 15, 1950

Country of Birth — Lieu de naissance

HUNGARY

Permanent Address — Résidence fi

Rob Willoner,
1906 Mountain Highway,
North Vancouver,
British Columbia V7J 2M8.

Title of Thesis — Titre de la thèse

ON THE DESIGN OF A PARALLEL ARITHMETIC UNIT

University — Université

UNIV. OF ALBERTA

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

PH.D.

Year this degree conferred — Année d'obtention de ce grade

1979

Name of Supervisor — Nom du directeur de thèse

DR. I. N. CHEN

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date

September 13, 1979

Signature

Rob Willoner

THE UNIVERSITY OF ALBERTA

ON THE DESIGN OF A PARALLEL ARITHMETIC UNIT

by

© ROBERT G. WILLONER

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON ALBERTA CANADA

FALL 1979

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "ON THE DESIGN OF A PARALLEL ARITHMETIC UNIT", submitted by Robert Willoner in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

I Ngo Chen

I-Ngo Chen (supervisor)

John Tartar

John Tartar

Stanley Cabay

Douglas Bates

Kishor Trivedi (external)

Date August 31, 1979

ABSTRACT

In this study, the feasibility of performing the four basic arithmetic operations "on-line" and in linear time is investigated. The operands and the results are assumed to be fixed point and in binary notation. An operation is said to perform in linear time if its execution time is bounded from above (and below) by some constant multiple of the length of the longest of the operands. The on-line property is satisfied if, in order to generate the ith bit of the result, it is necessary and sufficient to have the operands available to the ith bit only. The operands are assumed to flow through the arithmetic unit in a bit-by-bit, least-significant-bit-first fashion, and the results are produced in the same manner. The advantages of this mode of computation stem from the fact that a sequence of operations can be performed in an overlapped fashion, resulting in a significant speeding up over traditional sequential algorithms.

The proposed multiplier consists of a set of modules, one for each bit of the product. A basic module of the multiplier is only slightly more complex than a full adder; instead of three inputs and two outputs, it has five inputs and three outputs, and is designed to execute in the same time as the adder. A divider is designed which accepts its inputs and produces a remainder and quotient in the same

bit-sequential right-to-left fashion, but with an initial time delay proportional to the word length. The divider, which also operates in linear time, consists of three separate circuits executing concurrently, each of approximately the same complexity as the circuits for the other three operations.

Suitable applications for this proposed mode of arithmetic operation are considered. The evaluation of expressions and the use of linear on-line arithmetic in associative memories are prominent among these. A new, superior method of performing modular exponentiation using these concepts is developed. Using this, a high speed hardware device for encoding and decoding messages in a "public-key cryptosystem" is designed.

ACKNOWLEDGEMENTS

I wish to express sincere gratitude to my supervisor, Dr. I-Ngo Chen, for his guidance and assistance at every stage in the development of this thesis. I thank the members of my committee, Drs. John Tartar, Stan Cabay, Doug Bates and Kishor Trivedi, who have been invaluable in their thorough assessment of this dissertation. All the people in the Computing Science department have contributed to making my four year stay here a most enjoyable experience.

Special thanks must go to my fiancee, Annie, for providing the motivation necessary for the completion of this work. Above all, I am grateful to my father, Guido, to whom this thesis is dedicated, for instilling in me his belief in the value of education.

TABLE OF CONTENTS

LIST OF FIGURES

figure page

LIST OF ALGORITHMS

algorithm page

chapter one

INTRODUCTION

Parallel processors have been categorized in a number of different ways[49]. Perhaps the categorization most commonly accepted is one based on the <u>stream</u> concept introduced by Flynn[19]. A stream is a sequence of instructions or data as executed or operated on by a processor. Using this concept, parallel computers are classified by the magnitude (either in space or in time) of interactions of their instructions and data streams. In the simplest case, an SISD (single instruction stream, single data stream) machine, only one instruction can be operating on one set of data at any given time. SIMD (single instruction stream, multiple data stream) computers have only one stream of instructions in execution at any time, but each instruction may affect many different data items. These parallel computers are further categorized in [19] as follows:

- parallel in space, as are structured array computers
  e.g. ILLIAC IV, OMEN-60, SIMDA;
- unstructured linear array computers, or <u>ensembles</u>
  e.g. PEPE, Goodyear STARAN S;

- parallel primarily in time, as are pipelined computers
  e.g. Amdahl 470 V/7, Cray-1, TI ASC;
- associative processors whose memories are addressed by contents rather than by addresses
  e.g. PEPE, Goodyear STARAN S.

MIMD (multiple instruction stream, multiple data stream) machines have more than one stream of instructions, in fact, as many streams as there are data streams. These parallel computers are essentially interconnected sequential computers, usually called multiprocessors. The UNIVAC 1108 is an example of such a machine. Flynn's categorization is summarized in Figure 1.1, along with examples.

The arithmetic unit is the heart for computation on a digital computer, and as such, has been the focus of research in the hardware area from the time the field was first defined. The standard ripple-carry adder[31] is the simplest to build, and it performs in O(n) time, where n is the number of bits of the addends. This adder is both linear and on-line. A "ripple-borrow" subtracter performs in an analogous manner. Theoretically, each of these can be speeded up to O(log n) by the use of "carry-lookahead"[31], but at the cost of enormously increasing the complexity of the circuitry. Information-theoretic arguments can be used

DATA

INSTRUCTIONS	Single Data Stream (SD)	Multiple Data Stream (MD)
Single Instruction Stream (SI)	Unit Processor e.g. IBM 360	Array or Associative Processor e.g. ILLIAC IV
Multiple Instruction Stream (MI)	Pipelined Processor e.g. Amdahl 470 V/6	Multiprocessor e.g. UNIVAC 1108

Figure 1.1. Flynn's categorization of parallel processors.

to show that this is asymptotically optimal. The simplest algorithm for performing multiplication without the use of parallelism requires $O(n^2)$ time. Knuth[30] has shown that this can be reduced to $O(n\uparrow(1+\epsilon))$[+] for any arbitrarily small ϵ, but this result is not practical as the algorithms become increasingly complex as $\epsilon \longrightarrow 0$, causing the constant of proportionality to become extremely large. Schonhage and Strassen[44] have shown a serial algorithm that is a considerable improvement: $O(n \log n \log \log n)$. This is the fastest known algorithm but it is not yet known whether or not it is optimal. Knuth shows a division algorithm of the same order of time complexity, based on the Schonhage-Strassen multiplication algorithm.

Atrubin[5] was the first to show that by allowing parallelism, it is possible to perform multiplication in time $O(n)$. He also showed a linear on-line algorithm, considerably more complex than the one to be demonstrated, for evaluating the three-operand expression $A*B + C$. This algorithm will be discussed in some detail in Chapter 3. Atrubin's work is of interest because it introduced the concept of linear on-line arithmetic, which is the focus of interest in this thesis. An operation is said to perform in _linear time_ if its execution time is bounded from above (and below) by some constant multiple of the length of the longest of the operands. The _on-line_ property is satisfied

[+] The symbol "↑" is used to denote exponentiation if, in order to generate the ith bit of the result, it is necessary and sufficient to have the operands available to the ith bit only. The operands and the results are assumed to be fixed point and in binary notation. They are further assumed to flow through the arithmetic unit in a bit-by-bit, least-significant-bit-first fashion, and the results are produced in the same manner.

A number of studies have been made of algorithms which execute on-line, but which perform in a left-to-right, most-significant-bit-first fashion. The most recent result, with references to others, is given by Trivedi and Ercegovac[51]. The nature of their algorithms necessitates the use of a redundant digit set, at least for the bits of the result. This implies the need for post-processing of the results. The algorithms presented here use a redundant notation internally, but produce all results in customary binary notation. Trivedi and Ercegovac present algorithms for multiplication and division which have an inherent fixed time delay independent of the length of the operands.

The advantages of arithmetic operations being performed on-line and in linear time stem from the fact that a sequence of operations can be performed in an overlapped fashion, resulting in a significant speeding up over traditional sequential algorithms. This technique is known as pipelining. Pipelining is a commonly used hardware design technique for utilizing parallelism to increase the computation rate of a computer. The effectiveness of this technique is highly dependent on the structures of the algorithm and of its input data. For certain types of computations, pipelining can result in a significant increase in performance, whereas for others, a noticeable gain may be impossible to achieve.

In general, pipelining can be used effectively to realize, an algorithm whenever that algorithm can be divided into a fixed number of steps that are to be executed in sequence. A pipelined realization of such an algorithm consists of several hardware stages separated by registers. There is one stage for each step of the algorithm and they are interconnected in the same order that the steps are executed. This is illustrated in Figure 1.2. Many computers today use this technique for instruction execution. However, the arithmetic computation phase of an arithmetic instruction is currently always a single stage in the pipeline. The development of on-line techniques in this thesis and in other work allow the potential of breaking up individual arithmetic operations into yet another level of stages to be pipelined.

A number of recent technological developments make the results of this thesis particularly applicable. The bit-sequential nature of the arithmetic discussed herein is

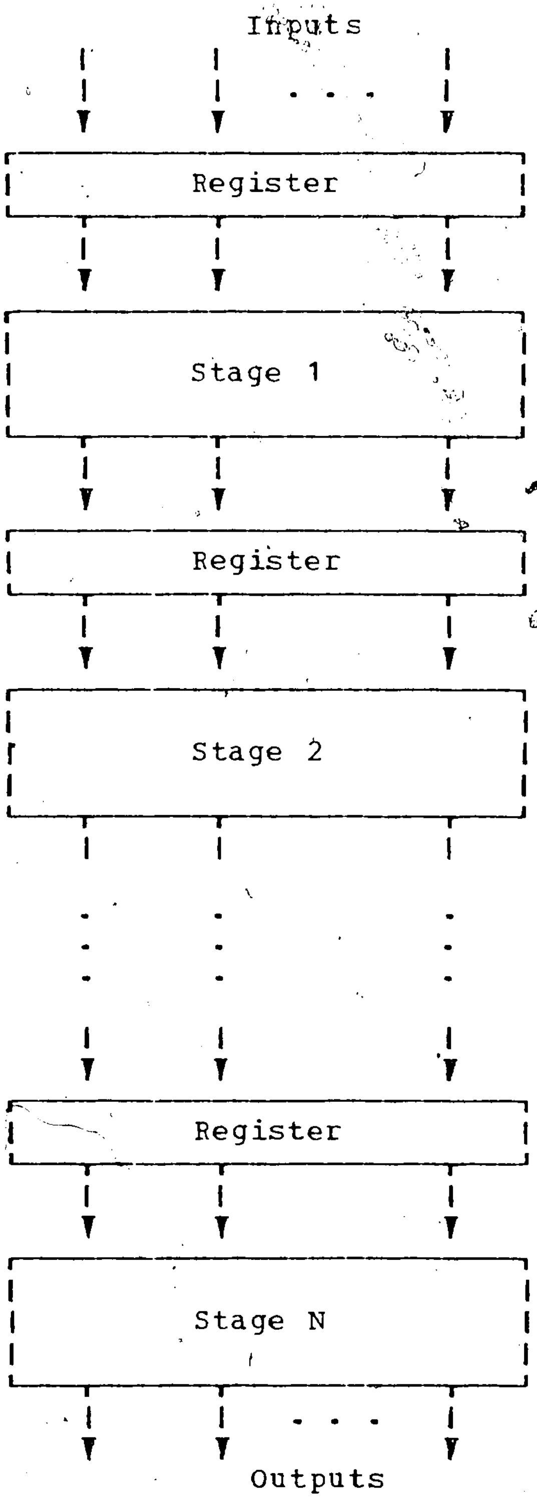


Figure 1.2. A linear pipeline illustrated.

entirely compatible with the nature of bubble memories and CCD's (charge-coupled devices), which are essentially continuous shift registers. The modular construction of each of the devices to be described allows for representation in the bit-slice architecture which is currently popular with many microprocessor manufacturers. Furthermore, the small number of inputs and outputs on each of the designs to be described permit LSI implementation wherein the limited number of pins per chip is a constant constraint.

The thesis begins with a discussion of redundant notation in computer arithmetic, in Chapter 2. The ripple-carry adder and ripple-borrow subtracter are introduced in preparation for the development and explanation of the multiplier. The next chapter discusses the linear on-line multiplier in detail. A logical realization of the central part of this multiplier is given. It is seen to be simple in construction and, most important, compatible in timing with the adder and subtracter. Chapter 3 ends with a section concerning multiple-precision multiplication in an SIMD processor. An algorithm is described and proved to be optimal. The significance of this result lies in the corollaries which follow from it. Lower bounds can be tightened on a number of multiple-precision algorithms, such as determinant computation, matrix multiplication, and the solution of a system of linear equations.

Chapter 4 describes the problems imposed by division. It is shown that strictly on-line division is not possible. The introduction of a delay between inputs and outputs allows the construction of a suitable divider. This proposed divider makes use of the on-line multiplier and adder/subtracter in an iterative convergent reciprocal algorithm.

A number of applications for these elementary results are considered. In Chapter 5, a strong correspondence between on-line arithmetic and bit-slice operations in an associative processor is demonstrated. This leads to a number of interesting applications in an associative processor, such as efficient algorithms for many vector and matrix operations. Another major application of on-line arithmetic appears in Chapter 6. This concerns the problem of modular exponentiation, which is useful in certain proposed implementations of public-key cryptosystems. The standard cubic algorithm for this problem is improved to one of quadratic complexity.

chapter two

LINEAR ON-LINE ADDITION AND SUBTRACTION

A brief discussion of adders and subtracters is presented. Ripple-carry and carry-lookahead adders are described and compared in time and design complexity. The notion of redundant arithmetic is defined and discussed. Ripple carry adders are discussed in view of the arithmetic unit to be proposed.

2.1 The Concept of Redundant Notation

The adder and subtracter presented here are capable of producing a result in a redundant notation[26] in unit time (independent of the length of the operands). In the case of the adder, the redundant notation is an extension of standard binary, by allowing the digit 2 in a position. For illustration of the subtracter, it is most convenient to allow the digit -1, denoted $\underline{1}$. Since the result of each of these operations can be performed in unit time, the result in non-redundant form can be produced after n time steps, by effectively performing n ripple-carries.

For the purposes of the multiplier and divider which are to be discussed in the next two chapters, it will also be necessary to have a means of producing a sum in redundant form when one of the two addends is in redundant form, in unit time. An analogous result will be needed for subtraction. These can all be done, and the corresponding tables appear in Figure 2.1. A few examples which should illustrate the algorithms are given in Figure 2.2. It is easy to realize the required functions as Boolean logic circuits.

2.2 A Bit-Sequential View of the Ripple-Carry Adder

The design of the arithmetic unit to be described in this thesis was motivated by observation of the performance of a ripple-carry adder[31]. Let us examine the operation of this simple adder. Assume, throughout this thesis, that all computations are done in fixed point integers and that all numbers are expressed in binary notation. The two addends which are input to this adder must initially be lined up, "right justified", so that the least significant bits of the inputs are in the same position. The adder can then be thought of as a device which moves over the two addends from right to left, putting out a bit of the sum as it passes over each pair of bits of the operands.

A point to be noted is that the adder will have to have some kind of temporary store (possibly denoted as a change of state) which is equivalent to the carry bit.

In this thesis, it will be more useful to think of this same adder as it is depicted in Figure 2.3. The addends "move through" the adder, one bit at a time with the least

ADDITION

	without carry[1]	with carry[2]
0 + 0	0	1
0 + 1	1	2
1 + 0	1	2
1 + 1	0	1
2 + 0	0	1
2 + 1	1	2

SUBTRACTION

	without borrow[3]	with borrow[4]
0 - 0	0	$\underline{1}$
0 - 1	1	0
1 - 0	1	0
1 - 1	0	$\underline{1}$
$\underline{1}$ - 0	1	0
$\underline{1}$ - 1	0	$\underline{1}$

In ADDITION,

[1]"without carry" means $\overset{0}{0}$, $\overset{0}{1}$, or $\overset{1}{0}$ in column to right

[2]"with carry" means $\overset{1}{1}$, $\overset{2}{0}$, or $\overset{2}{1}$ in column to right

In SUBTRACTION,

[3]"without borrow" means $\overset{0}{0}$, $\overset{1}{0}$, or $\overset{1}{1}$ in column to right

[4]"with borrow" means $\overset{0}{1}$, $\overset{\underline{1}}{0}$, or $\overset{\underline{1}}{1}$ in column to right

Figure 2.1. Addition and subtraction tables.

$$\begin{array}{rcccccccccl}
 & & 0 & 2 & 1 & 2 & 1 & 1 & 0 & = & 102 \\
+ & & 0 & 1 & 0 & 0 & 1 & 1 & 1 & = & \underline{39} \\
\hline
 & & 1 & 1 & 2 & 1 & 1 & 0 & 1 & & \\
 & & 1 & 2 & 0 & 1 & 1 & 0 & 1 & & \\
 & & 2 & 0 & 0 & 1 & 1 & 0 & 1 & & \\
 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & = & 141
\end{array}$$

$$\begin{array}{rccccccccl}
 & 1 & 0 & 0 & 1 & \underline{1} & 1 & \underline{1} & = & 69 \\
- & 0 & 1 & 0 & 0 & 1 & 1 & 0 & = & \underline{38} \\
\hline
 & 0 & 1 & 0 & 0 & 0 & \underline{1} & 1 & & \\
 & 0 & 1 & 0 & 0 & \underline{1} & 1 & 1 & & \\
 & 0 & 1 & 0 & \underline{1} & 1 & 1 & 1 & & \\
 & 0 & 1 & \underline{1} & 1 & 1 & 1 & 1 & & \\
 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & = & 31
\end{array}$$

$$\begin{array}{rccccccccl}
 & 0 & 0 & 2 & 2 & 1 & 1 & 2 & = & 56 \\
+ & 0 & 1 & 1 & 1 & 0 & 0 & 0 & = & \underline{56} \\
\hline
 & 0 & 2 & 2 & 1 & 1 & 2 & 0 & & \\
 & 1 & 1 & 0 & 1 & 2 & 0 & 0 & & \\
 & 1 & 1 & 0 & 2 & 0 & 0 & 0 & & \\
 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & = & 112
\end{array}$$

$$\begin{array}{rccccccccl}
 & 1 & 1 & 0 & \underline{1} & \underline{1} & 0 & \underline{1} & = & 83 \\
- & 1 & 0 & 0 & 1 & 0 & 1 & 1 & = & \underline{75} \\
\hline
 & 0 & 1 & \underline{1} & \underline{1} & 0 & 0 & 0 & & \\
 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & = & 8
\end{array}$$

Figure 2.2. Examples of addition and subtraction.

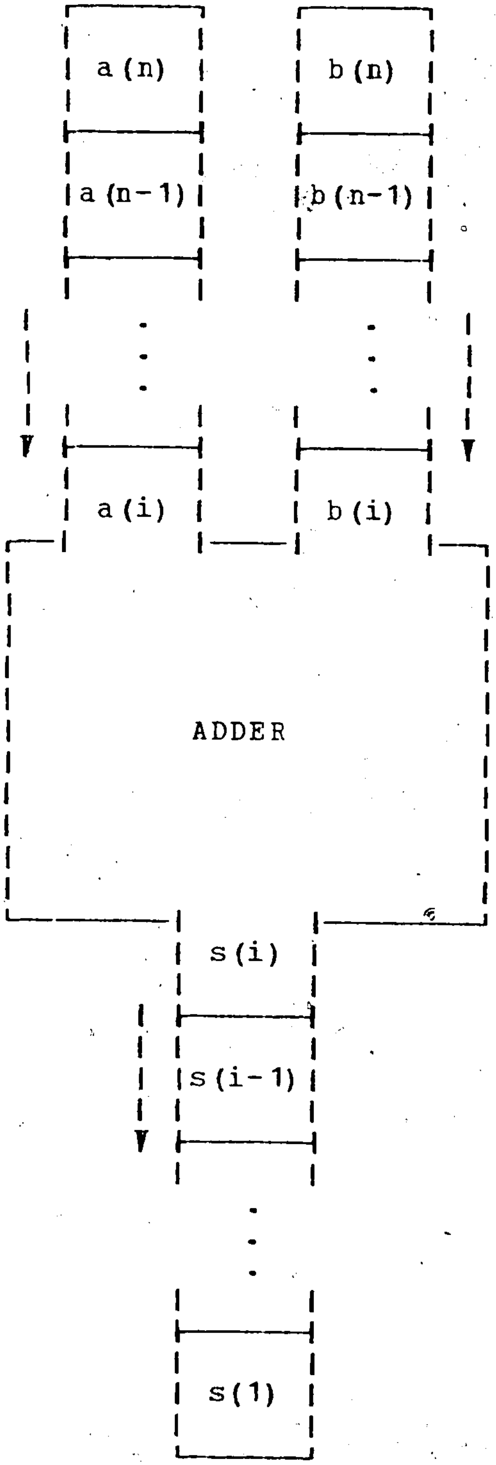


Figure 2.3. A seqrential adder, viewed in the context of this thesis.

significant bit first. The sum emerges in the same fashion—one bit at a time, least significant bit first. It begins to emerge exactly one time step after the first bits of the addends are received. Hence, it is not necessary to know the full addends before addition can begin; it is only desirable to supply the input bits at a rate fast enough to keep the adder continuously busy.

The construction of such an adder is trivial. Subtraction has all the simple properties of addition, and the construction of a sequential subtracter along the same lines is equally easy. The design of an adder which accepts three or more operands and still operates in bit-sequential input/output mode would proceed in a similar manner. In fact, an arithmetic circuit could be designed which would accept any fixed number of operands and could perform a fixed sequence of additions/subtractions in bit-sequential input/output mode. Such circuits are not practical, however, because of the fixed number of operands and the fixed sequence of operations required.

A different approach to the problem of evaluating an expression consisting of only additions and subtractions is suggested if a number of parallel adders and subtracters of the above type are available. The expression

$$((A + B) - C) + D$$

can be evaluated by cascading two adders and one subtracter.

This concept is referred to as <u>chaining</u> in the language of the Cray-1[39], and <u>forwarding</u> in the language of the IBM 360/91[3]. Assume the rightmost bits of each of the four operands is available initially, and that successive bits can be supplied one per time step. The first adder can start working on A + B immediately. One time step later, the subtracter can start working on (A + B) - C, and one time step after that, the second adder can start working on the entire expression. Hence, three time steps after the start of the computation, the first bit of the answer is computed. In general, the first bit of the answer is completed after ø steps, where ø is the number of operations in the expression. The entire answer is complete after n + ø steps, where n is the number of bits of the answer. For arbitrary expressions, define n to be the word length of the machine.

The virtues of an arithmetic unit which operates in the above parallel manner are obvious. Newly presented in this thesis are circuits for performing multiplication and division in bit-sequential input/output, right-to-left fashion, thus completing the arithmetic unit capable of operating in this mode.

Multiplication by the traditional algorithm is inherently done in right-to-left fashion, but the complete multiplier and multiplicand are needed before the computation can begin. The circuit to be described uses a modified version of this algorithm to precisely fulfill the properties required. One time step after the first bits of the inputs are available, the least significant bit of the product emerges. After that, successive bits of the product emerge, one bit per time step.

Let us now examine the familiar ripple-carry adder in a format compatible with the descriptions of the other three parallel circuits. Let [a(n)-----a(1)] represent the number A in bit form. The other addend, B, and the sum, S, are represented in an analogous way. Internally, the parallel adder consists of a set of n+1 modules, where n is the maximum length of the addends. It is to be noted that only one module is operational at any given time, and that the description consisting of several modules is useful only for comparison with the multiplier to be introduced in the next chapter. Each module has four inputs and two outputs; see Figure 2.4 and Figure 2.5. The four inputs consist of two inputs, A and B, from the addends, one carry, C, from the previous module at the previous time step, and a sum bit, S, which is cycled back from the current module at the previous time step. The two outputs from each module consist of a carry bit, C, and a sum bit, S, which is cycled back to the same module. Although there are four inputs, it is never possible to have four ones, which would require two carries, since either A and B are both 0 and S is 0 or 1, or S is 0 and A and B are 0 or 1. Hence, one carry bit, C is

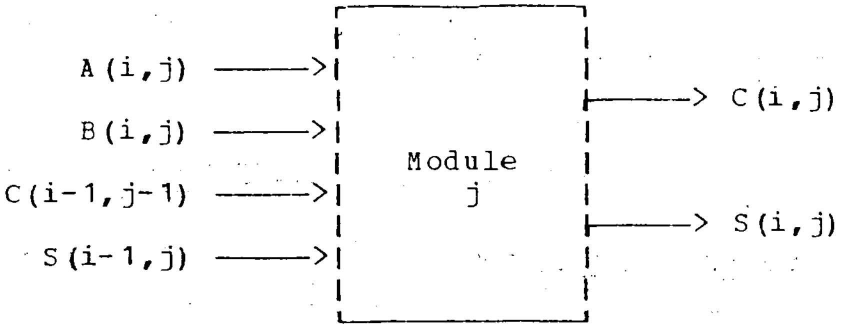


Figure 2.4. One of the modules of the parallel adder.

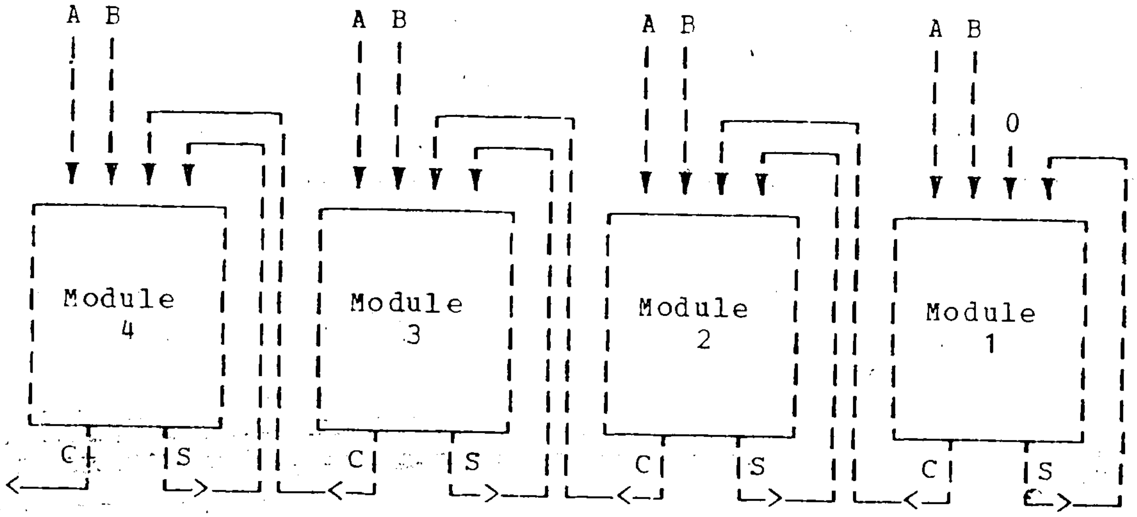


Figure 2.5. Linkage of the adder modules.

sufficient. The four input bits generated for the jth module during the ith iteration of the addition algorithm are as follows:

$$A(i,j) = \begin{cases} a(i) & \text{if } i = j-1 \\ 0 & \text{otherwise} \end{cases}$$

$$B(i,j) = \begin{cases} b(i) & \text{if } i = j-1 \\ 0 & \text{otherwise} \end{cases}$$

$$C(i,j) = \begin{cases} 0 & \text{if } j = 1 \\ S23[\,A(i-1,j),\ B(i-1,j),\ C(i-1,j-1),\ S(i-1,j)\,] & \text{otherwise} \end{cases}$$

$$S(i,j) = \begin{cases} 0 & \text{if } j = 1 \\ S13[\,A(i-1,j),\ B(i-1,j),\ C(i-1,j-1),\ S(i-1,j)\,] & \text{otherwise} \end{cases}$$

In the above, the functions S13 and S23 represent the symmetric function[31]. For example, S13[v, w, x, y] is 1 if exactly 1 or 3 of the arguments v, w, x, and y are 1; and 0 otherwise. A(i,j) and B(i,j) are the ith bits of the addends. These bits are available at the ith time step. The functioning of the n+1 modules of the parallel adder is presented as Algorithm 2.1.

This algorithm is written serially, but most of it is executed in parallel by the parallel adder. Lines 1-7 are initializations and can be considered to be done at the

Algorithm 2.1

Linear on-line addition

```
begin

     comment given a(1), a(2), . . ., a(n), and b(1),
       b(2), . . ., b(n), the bits of the two numbers to
       be added (least significant bit first), this
       algorithm computes s(1), s(2), . . ., s(n),
       s(n+1), the bits of the sum (again, least
       significant bit first);

     for i <— 1 until n+1 do
       begin
            A(i) <— 0;
            B(i) <— 0;
            C(i) <— 0;
            S(i) <— 0
       end;

     for i <— 1 until n do
       begin

            for j <— 1 until n do
              begin
                   A(j) <— 0;
                   B(j) <— 0
              end;

            A(i) <— a(i);
            B(i) <— b(i);

            for j <— n step -1 until 1 do

              begin
                   comment Define C(0) = 0;
                   t(1) <— S23[A(j), B(j), C(j-1),
                                           S(j)];
                   t(2) <— S13[A(j), B(j), C(j-1),
                                           S(j)];
                   C(j) <— t(1);
                   S(j) <— t(2)
              end

       end

     for i <— 1 until n+1 do s(i) <— S(i)

end;
```

start of the calculation. The block in lines 10-14 and in lines 15-16 is executed in one parallel operation. This is followed by execution of the block in lines 17-23, also in one parallel operation. Note that there is no need for the temporary variables t(1), and t(2), if the algorithm is executed in parallel rather than serially. The last step, line 25, is a retrieval of the answer. This can be done while computation of the previous steps is under way, with judicious timing such that the result bit of the sum is not retrieved until the circuit has executed a sufficient number of times (bit s(i) of the sum is ready for retrieval after i+1 time steps).

The parallel subtracter is constructed in an analogous manner, and will not be presented. The important property of the traditional addition algorithm which allowed for the previous description also holds for the subtraction algorithm; in subtraction, the carry bit is replaced by a "borrow" bit. For the purposes of this presentation, the possibility of the minuend being larger than the subtrahend causes no more concern than the possibility of an overflow in addition; both leave a carry (borrow) bit after the leftmost position.

chapter three

THE MULTIPLIER

Previous proposals for fast multipliers are discussed[26, 54], along with a summary of the known theoretical limitations of such designs. Then, a new parallel multiplier with a very simple configuration is suggested. This multiplier operates in time O(n), where n is the maximum of the lengths of the multiplier and multiplicand, both of which are fixed point, expressed in binary notation. It is a logical circuit consisting of 2n modules, each being only slightly more complex than a full adder; instead of three inputs and two outputs, each module has five inputs and three outputs. A logical circuit realization is given for the modules. But perhaps the most significant aspect of this design is the property that the input is required only bit-sequentially and the output is generated bit-sequentially, both at the rate of one bit per time step, least significant bit first. The advantages of such bit-sequential input and output arithmetic units are described.+

+A significant part of this chapter is to appear as a full paper[12] in the October 1979 issue of IEEE Transactions on Computers

3.1 Previous Proposals for Multipliers

The design of fast, efficient multipliers is currently a very active area[54]. Many scientific and engineering applications such as inversion of matrices, solution of differential equations, computation of eigenvalues, modular exponentiation, etc. require large numbers of multiplications. Especially in the light of LSI technology[48], it is important for computer hardware designers to be aware of current research concerning multipliers which are as efficient as theoretically possible.

The most primitive method of multiplication for binary numbers is a direct adaptation of the traditional "school boy method". This algorithm proceeds as follows. The accumulator (which is to eventually contain the product) is initialized to 0. Then, each bit of the multiplier is examined, with the rightmost (least significant) bit first. As each bit is scanned, the multiplicand is shifted left one position. Whenever a 1 bit is found in the multiplier, the multiplicand is added to the accumulator. When all the 1 bits in the multiplier have been exhausted, the accumulator holds the required product. This elementary algorithm is not used in computers with hardware multipliers because of its relative inefficiency. However, in machines where multiplication must be performed through software routines, variations of this method are often used.

A simple analysis of the time required to perform a multiplication by the above algorithm should suffice to convince the reader why it is not in common use. Let n be the maximum of the lengths (numbers of bits) of the multiplier and the multiplicand. Assuming that the shifts of the multiplicand and examinations of the bits of the multiplier can be done in negligible time (a realistic assumption), the problem reduces to that of a number of additions. The number of additions required is one less than the number of 1 bits in the multiplier. In the worst case, this means n-1 additions. The lengths of the addends in each case is not more than 2n. A standard ripple-carry adder can perform such an addition in O(n) time steps. Thus, the multiplication requires $(n-1)O(n) = O(n^2)$ steps. Theoretically, this can be speeded up by use of a carry-lookahead adder[54, 9], but at the cost of enormously increasing the complexity of the circuitry. The carry-lookahead adder can add two binary numbers of length n in time O(log n). Hence, incorporating this in the above multiplication algorithm yields a multiplier which performs in time O(n log n). In practice, the n stages of an adder are divided into groups such that each group contains a full carry-lookahead adder, while a carry ripple is maintained between the groups[9]. This gives a multiplier whose time complexity lies between $O(n \log n)$ and $O(n^2)$.

The concept of a parallel multiplier was first introduced by Dadda[13]. Parallel multipliers are discussed by Wallace[53], by Karatsuba and Ofman[28] and more recently by Stenzel et al.[46]. An excellent overview of hardware multipliers is given by Waser[54]. Such parallel multipliers generally operate in time proportional to the maximum of the lengths of the multiplier and multiplicands.

Winograd[58] derived an absolute lower bound on the computation time of a parallel multiplier. He showed if each element of the logical circuit of the multiplier has a fan-in lower than r, then the circuit has a gate delay of at least $\log(n-2)/\log r$;+ hence, $O(\log n)$ is an asymptotic lower time bound for multiplication, as it is for addition.

The type of multiplier that is of interest to us in this study involves bit-sequential input and output. The operation of such a device is illustrated in Figure 3.1, which depicts a "snapshot" of the multiplier just before the start of the ith step of its procedure. Traditional ripple-carry adders and subtracters have this bit-sequential property. An entire arithmetic unit consisting only of circuits of this type would have an advantage over traditional units, much as the concept of pipelining can be used to perform different segments of hardware instructions

+All logarithms in this thesis are taken to base 2

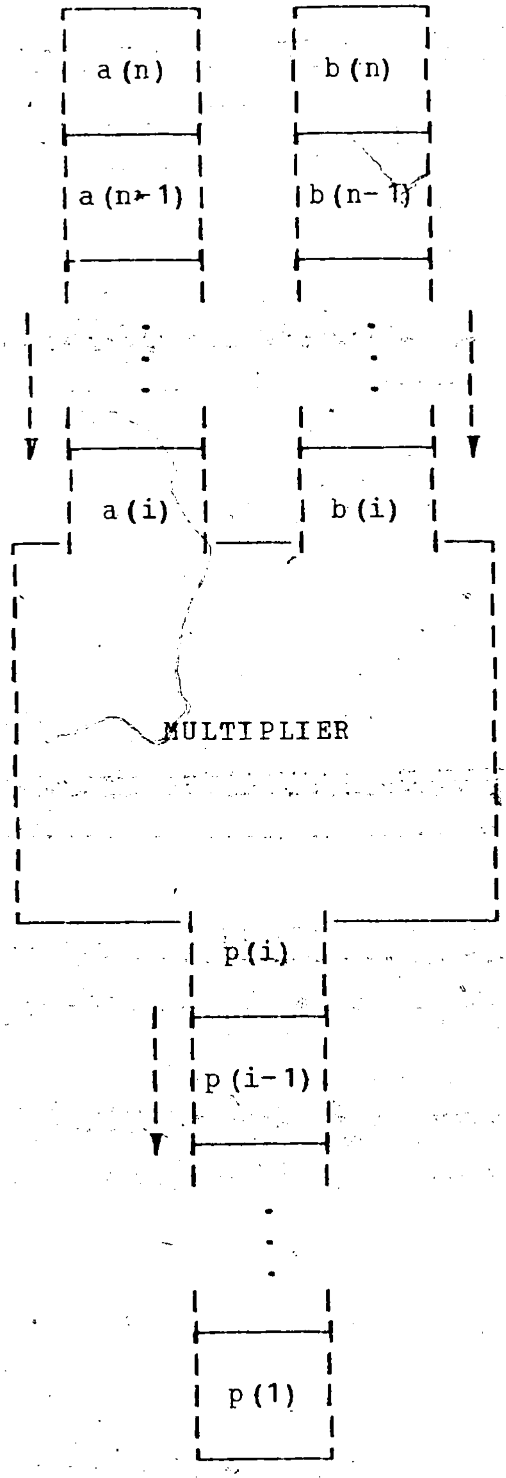


Figure 3.1. The multiplier immediately before initiating step i in the computation of the product of A and B.

independently and simultaneously. Arithmetic operations during the computation of an expression could be overlapped in time. For example, in the computation of A*B+C, it would not be necessary to complete the computation of the product of a and b before the addition can be initiated. As soon as the first bit of A*B is computed, the addition can be started. In fact, if the length of C were less than the length of A*B, the expression would be computed in the same length of time as the simple product A*B. Other authors have discussed the problem of extending a multiplier to compute an expression such as A*B+C. The first example of this was given by Atrubin[5]; this is discussed below. Advanced Micro Devices[1] manufactures an MSI device, the AMD 25S05, which computes this expression using an iterative array. The multiplier described herein can also be easily extended to compute this expression, but that is not the gist of this design. The point is that with the bit-sequential nature of the algorithm, any expression could be effectively speeded up (assuming a suitable control device is available).

The generalization to more complicated arithmetic expressions is evident. The success of such a system depends on a number of factors:

(i) each circuit must operate in bit-sequential input/output mode;

(ii) the time required for outputting (and inputting) each bit must be constant, and the same as the time required for a full adder (presumably, this cannot be reduced for more advanced operations);

(iii) there must be such a circuit for each of the four arithmetic operations (+, -, *, and /).

In this chapter, a multiplier which meets these requirements is described. A standard ripple-carry adder and subtracter will suffice to satisfy these. A simple counterexample shows that knowledge of the least significant bits of the divisor and the dividend do not determine the least significant bit of either the quotient or the remainder of a division operation, so a divider meeting these strict constraints cannot be designed, but slight relaxations of these restrictions could possibly lead to a divider which would make the system complete. Further discussion of this is given in the next chapter.

A further desirable property of any multiplier is that its components consist of one basic circuit type, much as a ripple-carry adder consists of a sequence of full adders; this leads to economical large-scale integration. Atrubin[5] refers to such bit-sequential circuits as operating in "real-time", and he discusses one such multiplier. His multiplier consists of a linear iterative array of automata, each of which can be in a finite set of states, at each step of the computation. Each automaton is too complex to operate in time corresponding to a full adder, so that property (ii) above is not met. To the author's knowledge, this multiplier has never been implemented. Atrubin's idea was subsequently simplified by Knuth[30]. The following is the basic idea behind their construction. Suppose the product of A and B is desired, and that the bits of these operands are made available sequentially, least significant bit first. Let the bits of A and B be a(1), a(2), . . ., a(n), and b(1), b(2), . . .,b(n) respectively, least significant bit first. Initially, the first automaton recognizes a(1) and b(1), so it is able to output a(1)b(1) at time 1. Then, it sees a(2) and b(2), so it can output {a(1)b(2) + a(2)b(1) + c} mod 2, where c is the carry left over from the previous step, at time 2. Next it sees a(3) and b(3), and outputs {a(1)b(3) + a(2)b(2) + a(3)b(1) + c} mod 2; furthermore, its state records the values of a(3) and b(3) so that the second automaton will be able to sense these values at time 3, and will be able to compute a(3)b(3) for the benefit of the first automaton at time 4. Thus, the first automaton arranges to start the second one multiplying {a(3), b(3)}, {a(4), b(4)}, . . ., and the second automaton will ultimately give the third one the job of multiplying {a(5), b(5)}, {a(6), b(6)}, . . ., etc. Approximately n/2 automata are required to multiply two n-bit numbers. All bits of the product are output by the first automaton, one per time step, least significant bit first.

In contrast, the multiplier described herein has one module (automaton) for each bit of the product, and the result bits are output by their respective modules. There are 11 inputs to each of the Knuth-Atrubin automata, leading to a possible 2^{11} states per automaton; by comparison, the present modules have only 5 inputs.

Trivedi and Ercegovac[51] refer to such bit-sequential circuits as "on-line" circuits. They approach the problem with the inputs and outputs being involved with the most significant bit first. Using a redundant number system and allowing a certain constant time delay (independent of the length of the operands), they design an on-line multiplier and divider. It is not possible to perform a conversion from the redundant system to a non-redundant binary system in bit-sequential mode (most significant bit first). However, their design can accept operands in redundant notation, so the conversion from redundant notation to binary notation can be insignificant in cost when a large number of operations are involved.

3.2 A Linear On-Line Design

In this section, we shall describe a multiplier which takes two n-bit numbers A (the multiplier) and B (the multiplicand) and outputs their product P. It inputs one bit of A and one bit of B at each time step (least significant bit first) and outputs one bit of P at each time step after the first (again, least significant bit first). Just before the start of step i, the multiplier may be visualized as in Figure 3.1. After 2n steps, the product is complete; after n steps, A and B are consumed.

Let [a(n)——a(1)] represent the number A in bit form, with [a(1)] being its degenerate form. Also, let [a(k)——a(1)], for $k \leq n$, represent the first k bits of A. The number B is represented in an analogous manner. We will denote the full product p by [p(2n)——p(1)] and the partial product of [a(k)——a(1)] and [b(k)——b(1)] by [p(2k)——p(1)]. Then, the traditional method of multiplication may be derived as follows:

$$[p(2n)\text{——}p(1)] = [a(n)\text{——}a(1)]\,[b(n)\text{——}b(1)]$$

$$= \{\sum_{i=1}^{n} a(i)\bullet 2\uparrow(i-1)\}\ \{\sum_{i=1}^{n} b(i)\bullet 2\uparrow(i-1)\}$$

$$= \sum_{i=1}^{n} \{a(i) \sum_{j=1}^{n} b(j) \bullet 2\uparrow(j-1)\} \bullet 2\uparrow(i-1)$$

$$= \sum_{i=1}^{n} a(i)[\, b(n)\text{———}b(1)\,] \bullet 2\uparrow(i-1)$$

By contrast, the multiplier to be constructed in this thesis is derived in the following way:

$$\begin{aligned}
[\, p(2k)\text{———}p(1)\,] &= [\, a(k)\text{———}a(1)\,]\,[\, b(k)\text{———}b(1)\,] \\
&= \{a(k) \bullet 2\uparrow(k-1) \\
&\qquad + [\, a(k-1)\text{———}a(1)\,]\}\,[\, b(k)\text{———}b(1)\,] \\
&= a(k)[\, b(k)\text{———}b(1)\,] \bullet 2\uparrow(k-1) \\
&\qquad + [\, a(k-1)\text{———}a(1)\,]\,\{b(k) \bullet 2\uparrow(k-1) \\
&\qquad\qquad + [\, b(k-1)\text{———}b(1)\,]\} \\
&= [\, p(2k-2)\text{———}p(1)\,] \\
&\qquad + a(k)[\, b(k)\text{———}b(1)\,] \bullet 2\uparrow(k-1) \\
&\qquad + b(k)[\, a(k-1)\text{———}a(1)\,] \bullet 2\uparrow(k-1), \\
&\qquad\qquad\qquad \text{for } k > 1
\end{aligned}$$

$$[\, p(1)\,] = a(1)b(1)$$

Two main points are to be noted about this recursive formula:

(i) To compute [p(2k)——p(1)] from [p(2k-2)——p(1)] requires at most two additions (a(i) and b(i) are either 0 or 1) of numbers of length n. Using parallelism, we shall show how a sufficient portion of these additions can be carried out in one time step to output the least significant bit of the sum, and to generate enough carry information to allow processing of the next pair of input addends. Thus we shall be able to compute [p(2n)——p(1)] in 2n time units.

(ii) In the computation of the sequence [p(1)], [p(2) p(1)], ... [p(2i)——p(1)], a(i) and b(i) are involved only in the last step. Hence, at the ith step in the computation of [p(2i)——p(1)], only the least significant i bits of A and B are required.

Using these facts, the multiplier is now constructed. The basic circuit (module) is an extension of the full adder. A full adder has three inputs (one bit from each addend and one carry bit from the previous summation) and two outputs—the sum bit and the carry bit. Our module has 5 inputs and 3 outputs. For a multiplier capable of multiplying numbers whose product does not exceed n bits, there are n modules. The 3 outputs of each module represent the sum of the 5 bit inputs. This is the same as a (5,3) counter of Dadda[13]; see Figure 3.2. One output bit is the sum bit, and since the sum can be as great as 5, two carry output bits are generated. The five input bits generated for the jth module during the ith iteration of the multiplication algorithm are as follows:

$$A(i,j) = \begin{cases} b(i)a(j-i+1) & \text{if } j < 2i-1 \\ 0 & \text{if } j \geq 2i-1 \end{cases}$$

$$B(i,j) = \begin{cases} a(i)b(j-i+1) & \text{if } j < 2i \\ 0 & \text{if } j \geq 2i \end{cases}$$

$$C1(i,j) = \begin{cases} 0 & \text{if } j = 1 \\ S23[A(i-1,j),\ B(i-1,j),\ C1(i-1,j-1), \\ \qquad C2(i-1,j-2),\ S(i-1,j)] & \text{otherwise} \end{cases}$$

$$C2(i,j) = \begin{cases} 0 & \text{if } j = 1 \text{ or } j = 2 \\ S45[A(i-1,j),\ B(i-1,j),\ C1(i-1,j-1), \\ \qquad C2(i-1,j-2),\ S(i-1,j)] & \text{otherwise} \end{cases}$$

$$S(i,j) = \begin{cases} 0 & \text{if } j = 1 \\ S135[A(i-1,j),\ B(i-1,j),\ C1(i-1,j-1), \\ \qquad C2(i-1,j-2),\ S(i-1,j)] & \text{otherwise} \end{cases}$$

All operations in the above are bit operations. The elementary symmetric functions, S23, S45, and S135, are denoted as usual[31]. For example, S45[v, w, x, y, z] is 1 if exactly 4 or 5 of the arguments v, w, x, y, and z are 1, and 0 otherwise.

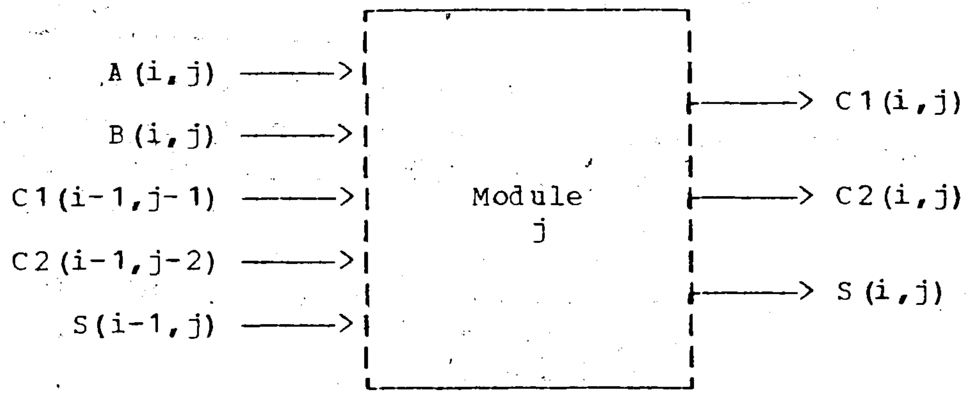


Figure 3.2. One of the modules of the parallel multiplier.

Figure 3.3 illustrates the linkage of the multiplier modules.

$A(i,j)$ and $B(i,j)$ are generated from the i least significant bits of the multiplier and multiplicand. These bits are available at the ith iteration via a simple data transfer operation. If $a(i) = 1$, then $B(i-1,i) = b(i)$, $B(i-1,i-1) = b(i-1)$, . . . $B(i-1,1) = b(1)$; if $a(i) = 0$, then $B(i-1,i) = B(i-1,i-1) = . . . = B(i-1,1) = 0$. If $b(i) = 1$, then $A(i-1,i-1) = a(i-1)$, $A(i-1,i-2) = a(i-2)$, . . . $A(i-1,1) = a(1)$; if $b(i) = 0$, then $A(i-1,i-1) = A(i-1,i-2) = . . . = A(i-1,1) = 0$. $C1(i,j)$ is the low order carry bit from the sum of the 5 inputs to module j at time i and $C2(i,j)$ is the high order carry bit from this sum. $C1(i-1,j)$ is the low order carry bit from module j-1 in the previous time step and $C2(i-1,j)$ is the high order carry bit from module j-2 in the previous time step.

The generation of the $A(i,j)$'s and $B(i,j)$'s can be elegantly illustrated by examining the order in which the multiplication "rhombus" is produced. In conventional multiplication, this rhombus of bits is generated row by row. In the on-line multiplier proposed here, this rhombus is expanded on two sides at each time step. The two different schemes are illustrated by example in Figure 3.4.

The entire algorithm for the functioning of the 2n modules throughout the duration of the multiplication is

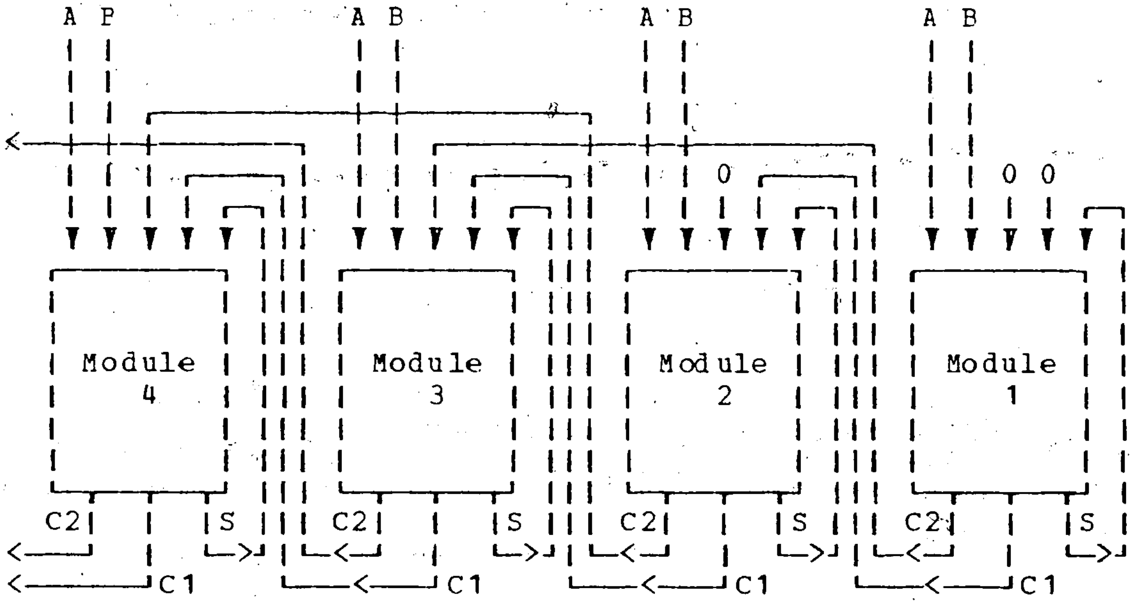


Figure 3.3. Linkage of the multiplier modules.

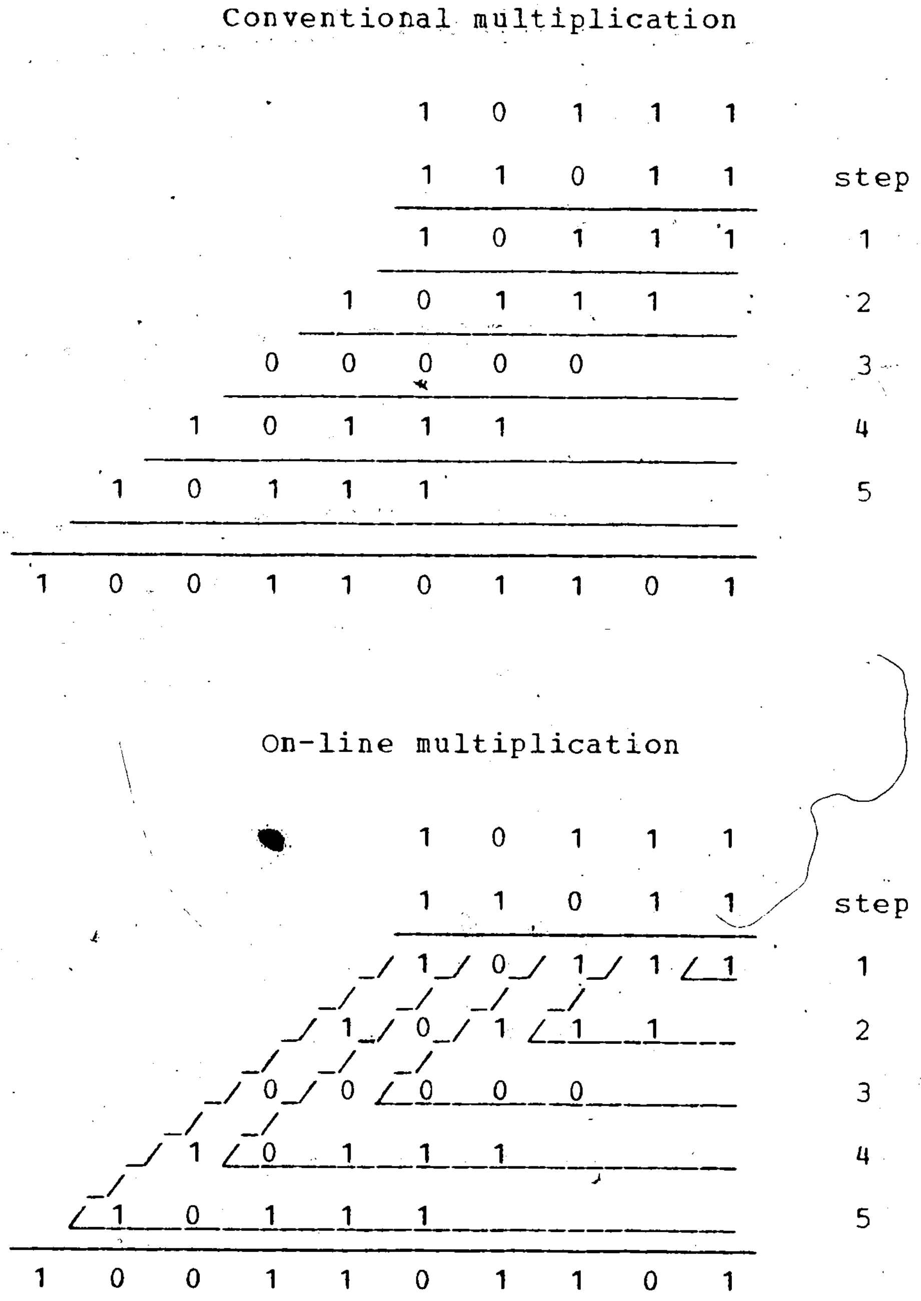


Figure 3.4. Generation of the elements of the multiplication rhombus.

presented as Algorithm 3.1. This algorithm is described in serial fashion, but it should be noted that most of it can be done in parallel. The initialization in lines 1-8 can be done in one step and may be ignored in timing considerations. Each cycle of the multiplier is started in line 10. Lines 11-15 reset the values of arrays A and B in each module. This can be done in one parallel operation. Lines 16-19 copy a part of the multiplier into array A (if b(i) = 1). Lines 20-24 copy a part of the multiplicand into array B (if a(i) = 1). These can both be done in one parallel step. Lines 26-33 compute the elementary symmetric functions of the 5 inputs and create the 3 outputs for each module. If done in parallel, the loop in line 25 is not needed, and furthermore, there is no need for the temporary variables t(1), t(2), and t(3). The last line, 35, extracts the output from each of the modules at the end of the multiplication.

Clearly, each module of this multiplier is very simple to construct since only the three elementary symmetric functions described above are involved; a logical realization is given in the next section as Figure 3.8. The parallel operation of the modules assures that the entire multiplier operates in time 2n, and each time step is no longer than that required by a full adder. Hence, the multiplier can multiply two numbers whose product is of length 2n in exactly the same time as it takes a

Algorithm 3.1

Linear on-line multiplication

```
begin
    comment Given a(1), a(2), . . ., a(n), and b(1), b(2),
      . . ., b(n), the bits of the two numbers to be
      multiplied (least significant bit first), this
      algorithm computes p(1), p(2), . . ., p(2n), the bits
      of the product (again, least significant bit first);

    for i <— 1 until 2n do
       begin
          A(i) <— 0;
          B(i) <— 0;
          C1(i) <— 0;
          C2(i) <— 0;
          S(i) <— 0
       end;
    for i <— 1 until 2n-1 do
       begin
          for j <— 1 until 2n do
             begin
                A(j) <— 0;
                B(j) <— 0
             end;
          if b(i) = 1 then
             begin
                for j <— 1 until i do A(i+j-1) <— a(j)
             end;
          if a(i) = 1 then
             begin
               if i > 1 then
                 for j <— 1 until i-1 do B(i+j-1) <— b(j)
             end;
```

Algorithm 3.1 (continued)

```
        for j <— 2n-1 step -1 until 1 do
           begin
            comment Define C1(0) = C2(0) =
                                  C1(-1) = C2(-1) = 0;

              t(1) <— S23[A(j), B(j), C1(j-1),
                                  C2(j-2), S(j) ];
              t(2) <— S45[A(j), B(j), C1(j-1),
                                  C2(j-2), S(j) ];
              t(3) <— S135[A(j), B(j), C1(j-1),
                                  C2(j-2), S(j) ];
              C1(j) <— t(1);
              C2(j) <— t(2);
              S(j) <— t(3)
           end

     end;

  for i <— 1 until 2n do p(i) <— S(i)

end.
```

ripple-carry adder consisting of 2n full adders to add two numbers whose sum is of length 2n.

An example of the generation of the A(i,j)'s and the B(i,j)'s is given in Figure 3.5. The full operation of this multiplier is illustrated by example in Figure 3.6. In this example, n = 7. Note that the multiplication is complete after 9 time steps. In general, 2n time steps may be required, but the correct product can usually be expected in much less time.

3.3 Logical Realization and Application

A parallel multiplier consisting of a series of identical modules has been proposed. Each module has 5 bit inputs and 3 bit outputs, the outputs being elementary symmetric functions of the inputs. The notation used for logical diagrams is shown in Figure 3.7, and a two-level logical realization of a multiplier module is given in Figure 3.8. The modules can execute in the same time as a traditional full adder. As opposed to the ripple-carry adder of Figure 2.5, where only one of the full adders is functioning at any given time, each of the modules of this multiplier is operating simultaneously. Hence, the modules may be thought of as operating in pipelined fashion, computing each bit of the product simultaneously. After each time step, a new bit of the product (least significant bit first) is generated.

```
time
 |
 |
 v
     b = 1                                          B(1,1)  =            1
0
     a = 1

     b = 0 1                                B(2,3)  B(2,2)  =          0 1
1
     a = 1 1                                        A(2,2)  =            0

     b = 0 0 1                      B(3,5)  B(3,4)  B(3,3)  =        0 0 0
2
     a = 0 1 1                              A(3,4)  A(3,3)  =          0 0

     b = 1 0 0 1            B(4,7)  B(4,6)  B(4,5)  B(4,4)  =      1 0 0 1
3
     a = 1 0 1 1                    A(4,6)  A(4,5)  A(4,4)  =        0 1 1

     b = 1 1 0 0 1       B(5,9)  B(5,8)  . . .  B(5,5)  =        1 1 0 0 1
4
     a = 1 1 0 1 1               A(5,8)  . . .  A(5,5)  =          1 0 1 1

     b - 1 1 1 0 0 1 B(6,11) B(6,10) . . .  B(6,6)  =      1 1 1 0 0 1
5
     a = 1 1 1 0 1 1         A(6,10) . . .  A(6,6)  =        1 1 0 1 1
```

Figure 3.5. Example of generation of A(i,j)'s and B(i,j)'s.

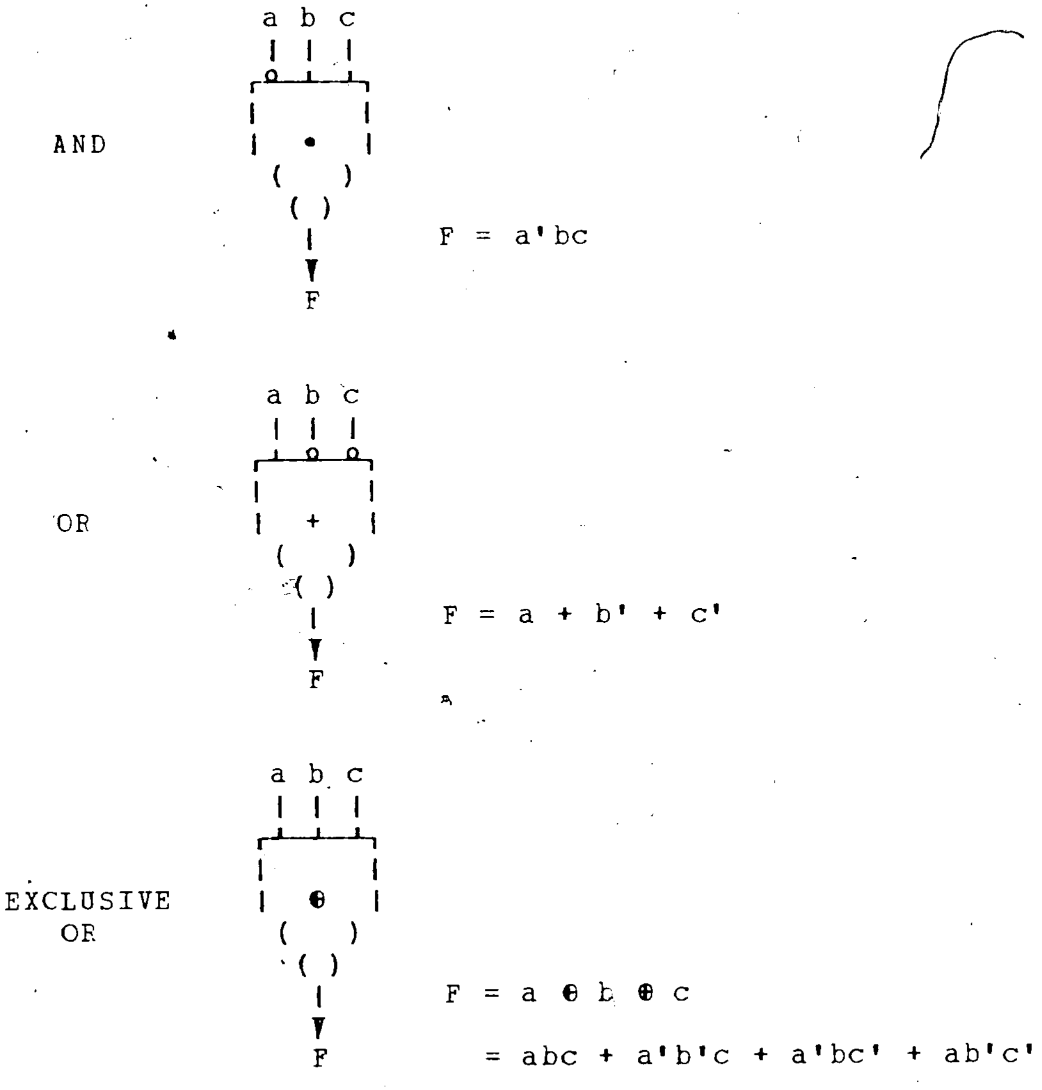
<——— j (module)

time	12	11	10	9	8	7	6	5	4	3	2	1	i
0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	01 00 0	1
1	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	01 00 0	00 00 1	2
2	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	00 00 1	00 00 1	3
3	00 00 0	00 00 0	00 00 0	00 00 0	00 00 0	01 00 0	00 00 0	10 00 0	11 00 0	00 00 0	00 00 1	00 00 1	4
4	00 00 0	00 00 0	00 00 0	01 00 0	11 00 0	00 00 1	10 00 0	11 01 1	00 00 0	00 00 0	00 00 1	00 00 1	5
5	00 00 0	01 00 0	11 00 0	11 01 1	00 00 0	10 10 1	11 00 1	00 00 0	00 00 0	00 00 0	00 00 1	00 00 1	6
6	00 00 0	00 11 1	00 00 0	00 00 0	00 01 0	00 01 1	00 00 1	00 00 0	00 00 0	00 00 0	00 00 1	00 00 1	7
7	00 01 0	00 00 1	00 00 0	00 00 0	00 01 1	00 00 0	00 00 1	00 00 0	00 00 0	00 00 0	00 00 1	00 00 1	8
8	00 00 1	00 00 1	00 00 0	00 00 1	00 00 0	00 00 0	00 00 1	00 00 0	00 00 0	00 00 0	00 00 1	00 00 1	9

```
B   =                 1 1 1 0 0 1
A   =                 1 1 1 0 1 1
          ---------------------------
A*B =     1 1 0 1 0 0 1 0 0 0 1 1
```

A	B
C2	C1
S	

Figure 3.6. Full multiplication example. The 5 inputs to each module are arranged as shown at the bottom right.



$F = a'bc$

$F = a + b' + c'$

$F = a \oplus b \oplus c$

$= abc + a'b'c + a'bc' + ab'c'$

Figure 3.7. Legend for logic diagrams, illustrated with examples.

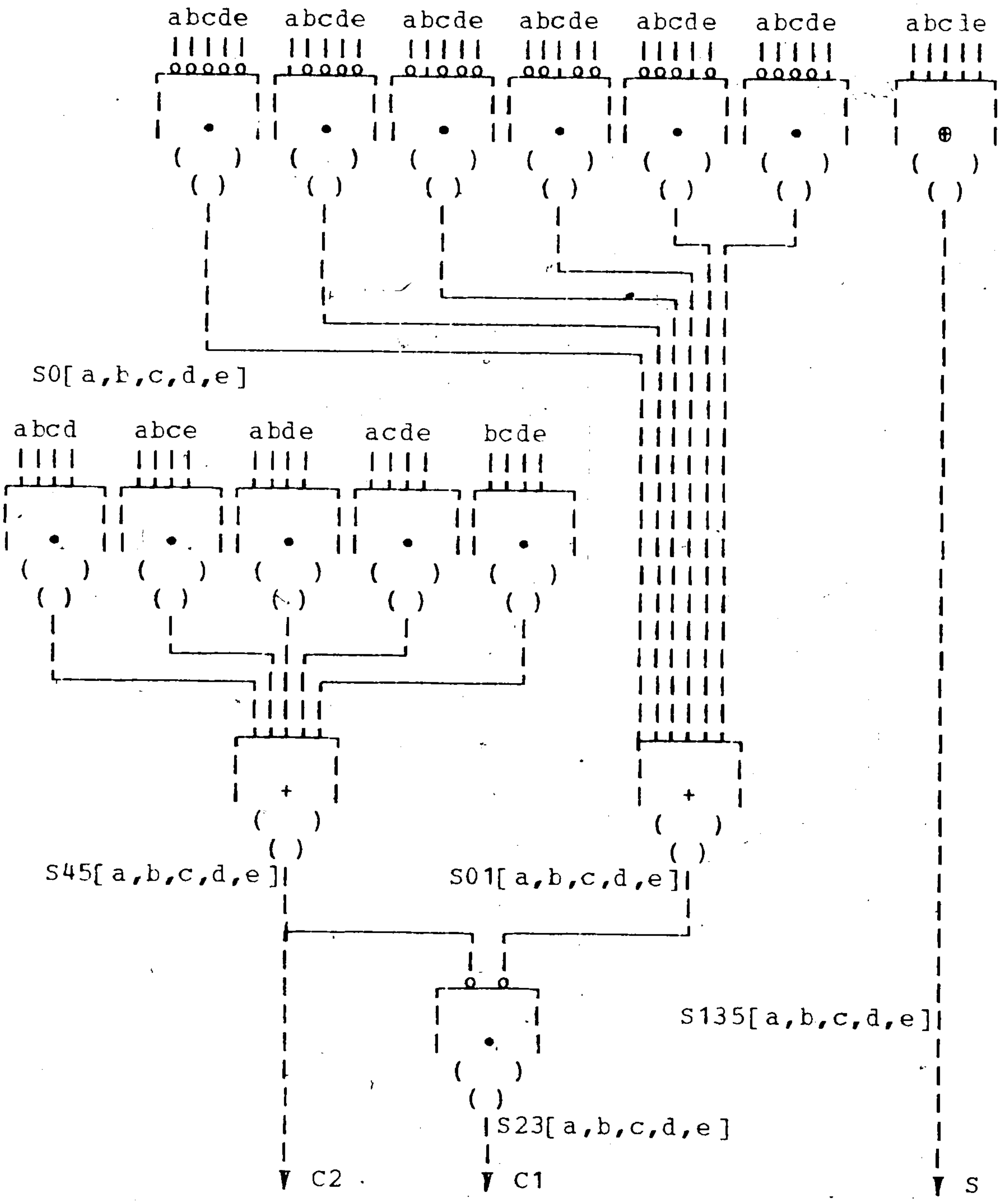


Figure 3.8. Logical realization of a multiplier module.

Ripple-carry adders and subtracters traditionally work in this manner. Adding a multiplier of the proposed type to an arithmetic unit allows pipelining of arithmetic operations at a higher level. In an expression involving several of these operations, it is not necessary to wait for one operation to complete before the next can be initiated, since the next operation can start as soon as the first bit of each of its operands is known. An example of this is given in Figure 3.9. Here, the evaluation of the expression (A*B - C*D + E) * F is performed using 3 multipliers of the type proposed here, 1 ripple-carry adder and 1 ripple-carry subtracter. It is assumed that all the operands are available at the start of the computation. If the length of the result is ø then the entire computation takes ø+3 time steps (note that after 4 time steps, the least significant bit of the result is available).

Negative numbers in uncomplemented form can be handled in the usual manner by the proposed multiplier, with the resultant sign being produced by an exclusive-or operation. Numbers in twos complement form must be complemented, but the complementing function can be evaluated bit-sequentially. Hence, signed numbers can be readily accomodated.

As shall be shown in Chapter 5, it is possible, by judicious arrangement of the addends, to compute the sum of

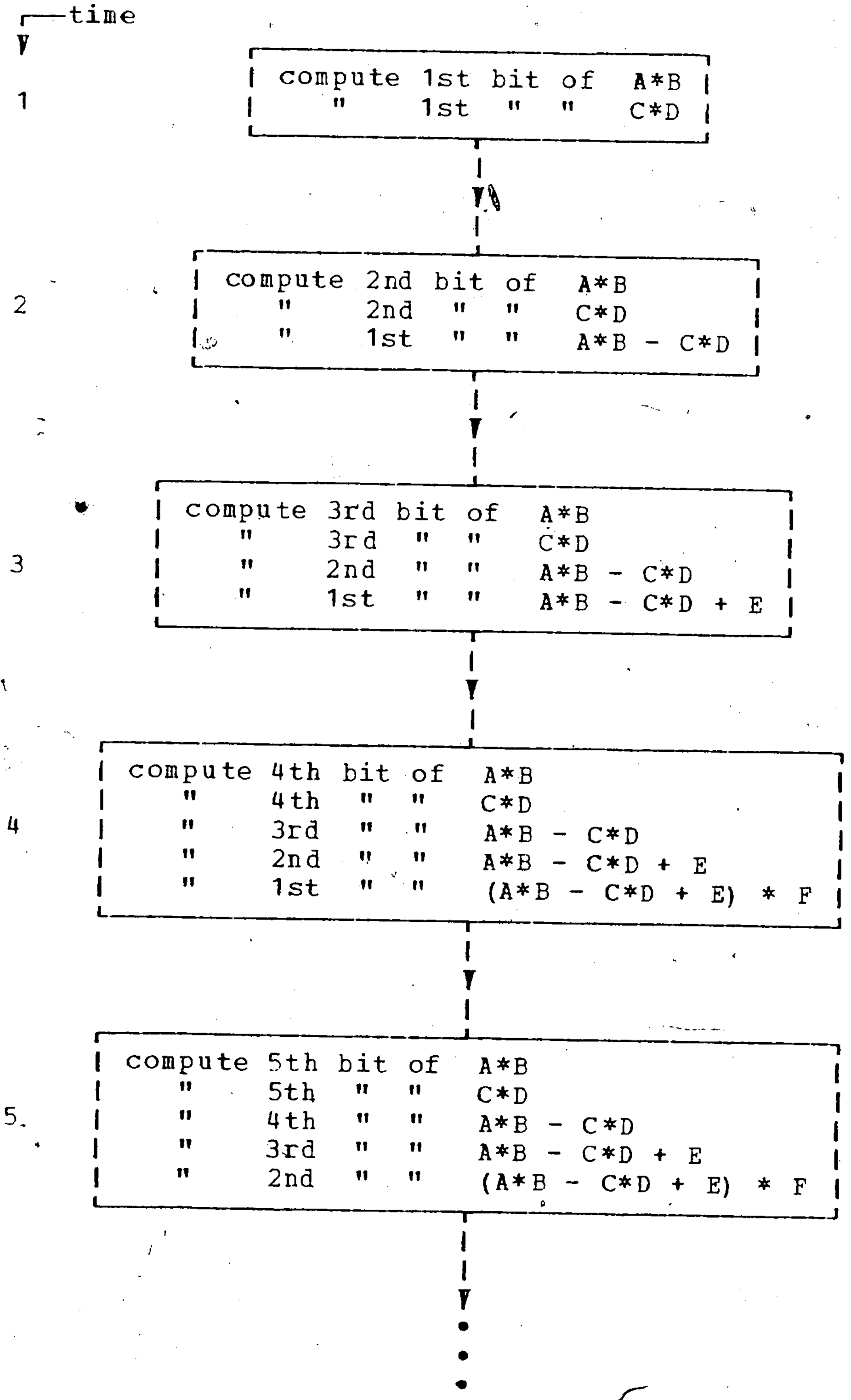


Figure 3.9. Pipelined computation of (A*B - C*D + E) * F.

m numbers, each of length n, in time n + 2log m in an associative processor with bit-slice operation capabilities[11, 7]. This can now be extended to a multiplication algorithm which leads to the computation of the product of m numbers of length n in time nm + log m = C(nm) in an associative processor. This represents a vast improvement over the traditional tree-product algorithm which requires $n^2(m^2-1)/3 = O(n^2m^2)$ time steps[11].

3.4 An Optimal Algorithm for Multiple-Precision Multiplication

This section discusses and solves a problem which originally arose in the research leading to an earlier thesis[32] in this department, but which has relevance to the subject of this chapter. The question posed is as follows. Given a set of numbers whose product is to be found, and an unlimited number of multipliers available at any given time, what is the optimal ordering of the required multiplications such that the total time taken is minimal. A multiplier is assumed to multiply two numbers of lengths n1 and n2 in time O(log(n1+n2)), which is the theoretically cptimal speed for a parallel multiplier, as shown by Winograd[58].

If the number of multiplicands, m, is a power of two, one's intuition judges that the optimal way to compute the product is by the traditional tree-product algorithm. With m operands, each of length n, first perform $m/2$ multiplications on pairs of the original operands. Each of these multiplications takes time approximately $\log 2n$. Since they can be done simultaneously using the assumed model, the total time so far is $\log 2n$. In the next step, $m/4$ multiplications on operands of length $2n$ are performed. The time required for this is $\log 4n$; total time so far is

$$\log 2n + \log 4n = \log (2n \bullet 4n).$$

For m operands, this procedure is repeated $\lceil \log m \rceil$[+] times, for a total time of

$$\begin{aligned}
&\log 2n + \log 4n + \log 8n + \ldots + \log mn \\
&= \log (2n \bullet 4n \bullet 8n \bullet \ldots \bullet mn) \\
&= \log [(2 \bullet 4 \bullet 8 \bullet \ldots \bullet m) \bullet n \uparrow \log m] \\
&= \log m \log n + \log (2 \bullet 4 \bullet 8 \bullet \ldots \bullet m) \\
&= \log m \log n + \log (2^1 \bullet 2^2 \bullet 2^3 \bullet \ldots \bullet 2 \uparrow \log m) \\
&= \log m \log n + \log [2 \uparrow (1 + 2 + 3 + \ldots + \log m)] \\
&= \log m \log n + (1 + 2 + 3 + \ldots + \log m) \\
&= \log m \log n + (1/2) (1 + \log m) \log m \\
&= (1/2) (1 + \log m) \log m, \quad \text{if } n = 1.
\end{aligned}$$

It will be shown below that in this case, intuition leads to the correct conclusion; this binary tree ordering does indeed prove to be optimal. Hence, the multiplication of m numbers can be computed in time $O(\log^2 m)$. This result has a number of major implications concerning the lower

[+] The symbol "$\lceil x \rceil$" is used to denote "the least integer greater than or equal to x"; "$\lfloor x \rfloor$" denotes "the greatest integer less than or equal to x" bounds of certain multiple-precision operations. For example, the previous lower bound for evaluation of determinants of matrices of size m was O(log m)[32]. Using the present result, we know that the determinant of a diagonal matrix requires $O(\log^2 m)$ steps, so the lower bound of the determinant problem can now be tightened to $O(\log^2 m)$.

From this, it follows that the lower bound for solving a system of m linear equations in m unknowns is $O(\log^2 m)$. This can be demonstrated by finding an example in which the numerator and denominator of a solution contain no common factors (so that at least m multiplications are required). It is also shown in [32] that the lower bound for multiplication of two mxm matrices is the same as that for solving a system of m linear equations in m unknowns. Hence, we have the same new lower bound for matrix multiplication in SIMD computers.

The multiplication ordering will be represented by a computation tree. To simplify the following proof, n will be assumed to be 1. Also, note that the time calculated above is a sum of logarithms, which is the same as the logarithm of the product. As a simplification, we will compute only the product, and call this the cost of the computation. The proof begins with a few definitions.

A binary tree is a tree for which each node has either 0 or 2 sons. This non-standard definition, which doesn't allow nodes with only one son, will be useful for our purposes. A computation tree, T, is a binary tree with a set of labeled nodes or vertices, denoted v[T], and two integer functions "l" and "d" on its nodes.

The length of a node n ∈ v[T], denoted l(n), is the number of leaves in the subtree rooted at n.

The depth of a node n ∈ v[T], denoted d(n), is an integer function satisfying the following:

$d(r) = 0$ where r is the root of T

$\forall n, m \in v[T], d(n) > d(m)$[+] where node n is a descendant of node m.

A computation tree, T, is said to be compact if for each pair of nodes n, m ∈ v[T] such that n is a son of m,

$$d(n) - d(m) = 1.$$

The height of a computation tree, T, denoted h[T], is defined to be the largest depth among the nodes in T, i.e.:

$$h[T] = \max_{\{n \in v[T]\}} \{ d(n) \}.$$

We note at this point that if T is compact, then the set

$$\{ i \mid n \in v[T] \text{ such that } d(n) = i \}$$

[+] The symbols "∀" and "∃" denote the universal ("for all") and existential ("there exixts") quantifiers, respectively is equivalent to the set $\{ i \mid 0 \le i \le h[T] \}$, since by the definition of a compact tree, the depth of the father of any node n, if the father exists (i.e. if the node n is not the root) is $d(n) - 1$.

The <u>factor</u> of a computation tree, T, at depth i, denoted $f(i, T)$, is defined for all non-negative integers i as follows:

$$f(i, T) = \begin{cases} 1, & \text{if there is no node } n \in v[T] \text{ for which } d(n) = i \\ \max\limits_{\{n \in v[T] \mid d(n) = i\}} \{ l(n) \}, & \text{otherwise.} \end{cases}$$

The <u>cost</u> of a computation tree, T, denoted $c[T]$, is an integer defined as follows:

$$c[T] = \prod_{i \ge 0} f(i, T).$$

Or, since $f(i, T) = 1$ for each $i > h[T]$, the product may be considered to be taken over all i, $0 \le i \le h[T]$.

Two computation trees, T and T', are said to be <u>similar</u> if there exists a one-to-one correspondence between $v[T]$ and $v[T']$ which preserves father-son relationships between the nodes. For every node $n \in v[T]$, the corresponding node in the similar tree T' will be denoted n'. Thus, if $m \in v[T]$ is the father of $n \in v[T]$, then $m' \in v[T']$ is the father of $n' \in v[T']$.

It follows immediately that for two similar computation trees T and T', for each $n \in v[T]$, $l(n) = l(n')$, since the number of leaves in the subtree of T' rooted at n' must be the same as the number of leaves in the subtree of T rooted at n.

Lemma 1: Let T be a compact computation tree, and T' be some other computation tree similar to T. Then,

$$c[T] \leq c[T'].$$

Proof: Since T is compact, for each i, $0 \leq i \leq h[T]$, there exists a node $n \in v[T]$ for which $d(n) = i$. Let $n(i)$ be one of the nodes of maximum length among all nodes of depth i in T, i.e.: for each depth i, $0 \leq i \leq h[T]$, and for each $n \in v[T]$ such that $d(n) = i$, $l(n) \leq l(n(i))$.

Then, $f(i, T) = l(n(i))$, and hence,

$$c[T] = \prod_{i=0}^{h[T]} l(n(i)).$$

The cost of T' is given by:

$$c[T'] = \prod_{i \geq 0} f(i, T').$$

Let us examine $f(i, T')$ for $i \geq 0$. For each i for which there exists some node $n \in v[T']$ such that $d(n) = i$, select one of the nodes of maximum length over all nodes in T' of depth i, and call it $m(i)$. Then,

$$c[T'] = \prod_{\substack{i=0 \\ \exists n \in v[T'] \ni d(n)=i^{+}}}^{h[T']} l(m(i)).$$

Our goal now is to prove that

$$\prod_{i=0}^{h[T]} l(n(i)) \leq \prod_{\substack{i=0 \\ \exists n \in v[T'] \ni d(n)=i}}^{h[T']} l(m(i)). \qquad (*)$$

Let us first show that the right side of (*) has at least as many factors as the left side. Select an arbitrary node $n \in v[T]$ which is of depth $h[T]$ in T. For the purposes of this argument, we will define a relabeling function L which labels node n and all its ancestors. Relabel n as $L[h[T]]$. Name the father of n as $L[h[T] - 1]$, the father of that as $L[h[T] - 2]$, and so on. Since T is compact, the depth of any node is 1 greater than the depth of its father. Thus, the root is relabeled $L[0]$, and in fact, for each i, $0 \leq i \leq h[T]$, $d(L[i]) = i$. Since father-son relationships are preserved between similar trees,

$$d(L[0]') < d(L[1]') < \ldots . < d(L[h[T]]')$$

and so, the integers $d(L[i]')$, $0 \leq i \leq h[T]$ are all distinct. Thus, the right side of (*) has at least as many factors as the left side.

[+] The symbol "∋" denotes "such that"

We will now show that there exists a one-to-one function, t,

$$t: [0,\ h[T]] \longrightarrow [0,\ h[T']]^{+}$$

such that

$$l(h(i)) \le l(m(t(i)))$$

for all $i \in [0,\ h[T]]$.

Note: It is implied that t is a function such that for all i, t(i) is one of those integers for which m(t(i)) exists, i.e. there exists at least one node $n \in v[T']$ for which $d(n) = t(i)$.

Once the existence of this function, t, has been shown, the inequality (*) will follow immediately.

The function, t, will be defined for the computation tree, T', by Algorithm 3.2. This algorithm determines N(i) for i = 0, 1, . . . h[T] to be some distinct set of nodes in T'. Afterwards, t(i) will be defined to be d(N(i)).

An example will now be given. It was designed to demonstrate all aspects of the algorithm. Figure 3.10 shows a compact computation tree, T. In this figure, solid nodes are those of maximum length at their respective depths. The lengths of the nodes are not given, and sufficient extra nodes are assumed such that the solid nodes are those of maximum length at their corresponding depths. These nodes

[+] The notation "[a, b]" is used to denote the integers between a and b inclusive

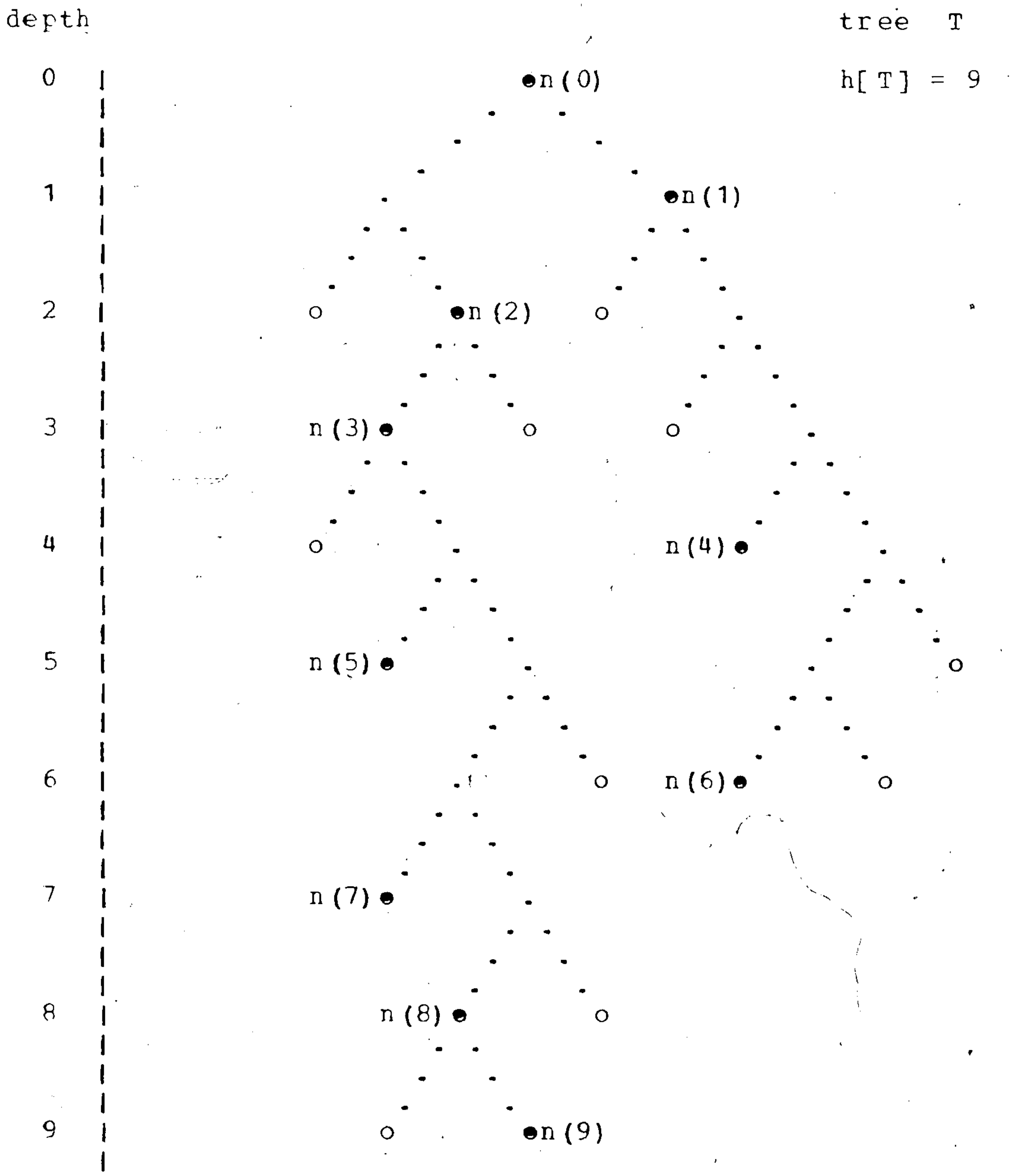


Figure 3.10. A compact computation tree, T.

are labeled "n(i)".

Figure 3.11 shows a computation tree, T', which is similar to T but is not compact. The nodes corresponding to the nodes n(i) ∈ v[T] are labeled "n(i)'" in v[T'] and are marked solid, as before. The squared nodes, labeled "N(i)" denote the values of N(i) after completion of the ith iteration of the loop in lines 2 through 10 of Algorithm 3.2. The evaluation of the function t by Algorithm 3.2 is summarized in Figure 3.12. This is done on this example tree, T', by giving the values of d(N(i)) before each execution of line 10.

It remains to explain why the algorithm always finishes executing line 10 before N(i) becomes the root of T' (i.e. why the father of N(i) always exists). Since d(n(i)) = i in the compact tree T, node n(i), and hence node n(i)', has exactly i ancestors. At the beginning of the ith iteration of the loop in lines 2 through 10, N(i) = n(i)'. Of the i+1 numbers which are the depths of node N(i) and its ancestors, at most i can be equivalent to one of the numbers d(N(0)), d(N(1)), . . ., d(N(i-1)). Hence line 10 may be executed 0 times, or, at most, i times.

At the end of the ith iteration of the loop in lines 2-10, d(N(j)) ≠ d(N(i)), for each j ∈ [0, i-1], since the logical expression in lines 4-9 has the value "false". All subsequent iterations of the loop for greater values of i

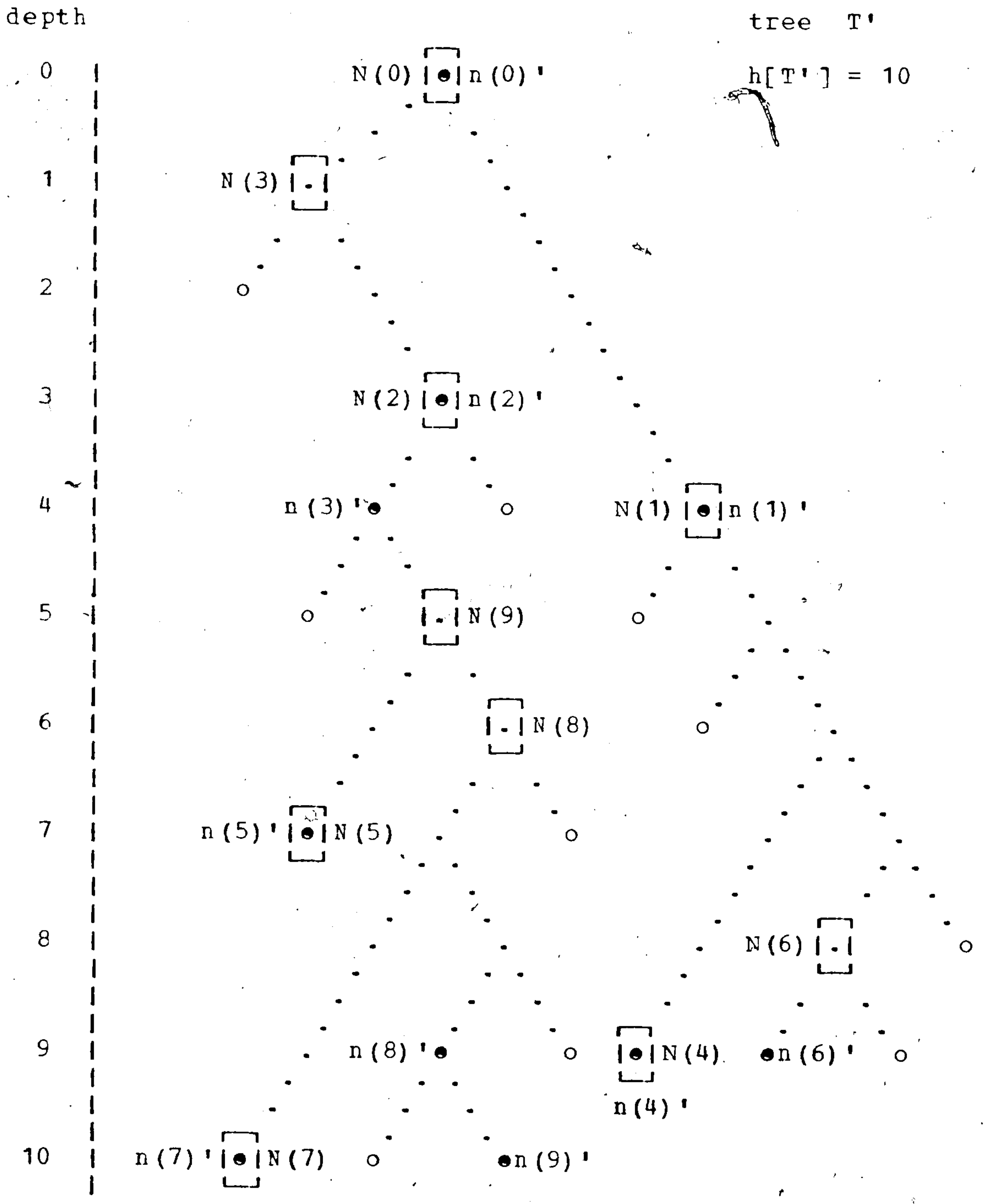


Figure 3.11. A computation tree, T', similar to T.

Algorithm 3.2

Evaluation of the intermediate function "t"

```
begin
      comment An algorithm which takes as input a set
        of nodes
             n(0)', n(1)', . . ., n(h(T))' ∈ v[T']
        whose corresponding nodes
               n(0), n(1), . . ., n(h(T)) ∈ v[T]
        are of maximum length at their respective
        depths in T, and returns a set of nodes
                  N(0), N(1), . . ., N(h[T]),
        which are all of different depths, and which
        can be used to determine the values of the
        function t. The nodes n(i)' are not all
        necessarily of different depths in T'.
        Notes:
            (a) N is an array of nodes in T' whose
            index ranges from 0 to h[T];
            (b) the father function, denoted "father"
            takes a node as argument and returns its
            father (immediate ancestor). It is
            forbidden to use a root as an argument to
            this function.
            (c) "check" is a logical variable
            (d) "v" denotes the logical "or" function;

  for i <— 0 until h[T] do N(i) <— n(i)';

  for i <— 1 until h[T] do
      while
          begin
               check <— false;
               for j <— 0 until i - 1  do
                   check <— check v [d(N(j)) = d(N(i))];
               check
          end;
          do N(i) <— father[N(i)]

end.
```

	number of executions of line 10						
	——>						t(i) =
i	0	1	2	3	4	5	d(N(i)) (final)
0	0	-	-	-	-	-	0
1	4	-	-	-	-	-	4
2	3	-	-	-	-	-	3
3	4	3	1	-	-	-	1
4	9	-	-	-	-	-	9
5	7	-	-	-	-	-	7
6	9	8	-	-	-	-	8
7	10	-	-	-	-	-	10
8	9	8	7	6	-	-	6
9	10	9	8	7	6	5	5

Hence, the function, t, is as follows:

$t(0) = 0$ $t(5) = 7$

$t(1) = 4$ $t(6) = 8$

$t(2) = 3$ $t(7) = 10$

$t(3) = 1$ $t(8) = 6$

$t(4) = 9$ $t(9) = 5$

Figure 3.12. A sample evaluation of the function "t".

cannot affect this result since they only alter the nodes $N(i+1)$, $N(i+2)$, . . ., $N(h[T])$. It therefore follows that after execution of the entire algorithm, $d(N(i)) \neq d(N(j))$ for each $i < j$ (and hence, for each $i \neq j$).

Now, for all $i \in [0, h[T]]$, $l(n(i)') \leq l(N(i))$ since either $N(i) = n(i)'$ (in which case equality holds) or $N(i)$ is an ancestor of $n(i)'$. Hence,

$$\begin{aligned}
c[T] &= \prod_{i=0}^{h[T]} l(n(i)) \\
&= \prod_{i=0}^{h[T]} l(n(i)') && \text{since } l(n(i))=l(n(i)'),\ \forall i \\
&\leq \prod_{i=0}^{h[T]} l(N(i)) && \text{since either } N(i)=n(i)' \text{ or} \\
& && N(i) \text{ is an ancestor of } n(i)' \\
&\leq \prod_{i=0}^{h[T]} l(m(t(i))) && \text{since } d(N(i)) = t(i) \\
&\leq \prod_{\substack{i=0 \\ \exists n\in v[T'] \ni d(n)=i}}^{h[T']} l(m(i)) && \text{since } \forall i\in[0,h[T]], \\
& && t(i)\in[0,h[T']] \text{ and} \\
& && l(m(i))\geq 1,\ \forall i\in[0,h[T']] \\
&= c[T']
\end{aligned}$$

Q. E. D.

Lemma 2: For all compact computation trees, T,

$$c[T] \geq \prod_{i\geq 0} \lceil p/2{\uparrow}i \rceil .$$

Proof: Consider f(i, T) for all i ≥ 0. There are two cases:

First, if $i < h[T]$, then let $n(i) \in v[T]$ be some node with $l(n(i)) = f(i, T)$ and $d(n(i)) = i$. Since $i < h[T]$, $l(n(i)) > 1$ so n has exactly 2 sons (of depth i+1). Since the sum of the lengths of its sons is l(n(i)), one of them must have length at least $\lceil l(n(i))/2 \rceil$. Hence,

$$\begin{aligned} f(i+1, T) &\geq \lceil l(n(i))/2 \rceil \\ &= \lceil f(i, T)/2 \rceil . \end{aligned}$$

In the second case, if $i \geq h[T]$ then $f(i, T) = 1$, so the above relaticn also holds.

Hence, for all $i \geq 0$, $f(i+1, T) \geq \lceil f(i, T)/2 \rceil$.

Now,

$$\begin{aligned} c[T] &= \prod_{i\geq 0} f(i, T) \\ &= f(0, T) \prod_{i\geq 1} f(i, T) \\ &= p \prod_{i\geq 1} f(i, T) \end{aligned}$$

$$= p \prod_{i \geq 0} f(i+1,\ T)$$

$$\geq p \prod_{i \geq 0} \lceil f(i,\ T)/2 \rceil$$

$$= p \lceil f(0,\ T)/2 \rceil \prod_{i \geq 1} \lceil f(i,\ T)/2 \rceil$$

$$= p \lceil p/2 \rceil \prod_{i \geq 0} \lceil f(i+1,\ T) \rceil$$

$$\geq p \lceil p/2 \rceil \prod_{i \geq 0} \lceil \lceil f(i,\ T)/2 \rceil /2 \rceil$$

$$= p \lceil p/2 \rceil \prod_{i \geq 0} \lceil f(i,\ T)/4 \rceil$$

$$\geq p \lceil p/2 \rceil \lceil p/4 \rceil \prod_{i \geq 0} \lceil f(i,\ T)/2^3 \rceil$$

$$\vdots$$

$$\geq \left\{ \prod_{i=0}^{k} \lceil p/2 \uparrow i \rceil \right\} \left\{ \prod_{i \geq 0} \lceil f(i,\ T)/2 \uparrow (i+1) \rceil \right\}$$

$$\geq \prod_{i\geq 0} \lceil p/2\uparrow i\rceil \qquad \text{since } \forall i\geq h[T],\ f(i,\ T) = 1$$

Q. E. D.

We now define a <u>balanced</u> computation tree as a compact computation tree for which any node having successors has a difference of at most one between the lengths of its two sons. The final theorem then follows:

<u>Theorem</u>: Let T be a computation tree with p leaves. If T is balanced, then no other computation tree with p leaves has lower cost than c[T].

<u>Proof</u>: If T is balanced, then it is compact. We shall now prove by induction that $f(i,\ T) = \lceil p/2\uparrow i\rceil$ for all $i \geq 0$.

Since T has p leaves, $f(0,\ T) = p$.

Assume $f(k,\ T) = \lceil p/2\uparrow k\rceil$.

If $k \geq h[T]$ then $\lceil p/2\uparrow k\rceil = 1$ since $2\uparrow[h(T)-1] < p \leq 2\uparrow h(T)$ for a balanced computation tree with p leaves. Hence, $f(k+1,\ T) = 1 = \lceil p/2\uparrow(k+1)\rceil$.

If $k < h[T]$ then let n be one of the nodes of T having depth k and length $l(n) = f(k,\ T)$. Node n must have some successors since $l(n) > 1$. The lengths of its two sons are $\lceil l(n)/2\rceil$ and $\lfloor l(n)/2\rfloor$. The maximum of the lengths of the two sons of node n is $\lceil l(n)/2\rceil$. In fact, since node n is of maximal length at depth k, $\lceil l(n)/2\rceil$ is the maximum of the lengths of all nodes having depth k+1. Hence,

$$\begin{aligned} f(k+1,\ T) &= \lceil f(k,\ T)/2\rceil \\ &= \lceil\lceil p/2\uparrow k\rceil/2\rceil \\ &= \lceil p/2\uparrow(k+1)\rceil . \end{aligned}$$

Therefore, $\forall i \geq 0,\ f(i,\ T) = \lceil p/2\uparrow i\rceil$. Then,

$$c[T] = \prod_{i\geq 0} \lceil p/2\uparrow i\rceil .$$

By lemma 2, this is minimal.

Q. E. D.

chapter four

THE DIVIDER

A number of current approaches to division are described: a restoring scheme, a non-restoring scheme, and a convergence scheme[26]. In comparison, a new linear on-line division algorithm is discussed in detail. Some of this is taken from our paper[56]. The difficulties with this divider are mentioned. Possible ways to overcome this are examined, with a good example being left for Chapter 6, where modular exponentiation is discussed.

4.1 Fixed-Point Dividers

In fixed-point division, two numbers, a divisor V and a dividend D, are given. The object is to compute a quotient Q and a remainder R such that

$$D = Q*V + R$$

where R is required to be of smaller absolute value than V, i.e. $0 \leq |R| < V$. The simplest division method is the traditional pencil-and-paper algorithm illustrated in Figure 4.1. Here, the quotient, Q = [q(n)——q(1)], is computed one bit at a time, most significant bit first. At the ith step of this method, (2↑-i)V, which represents the divisor shifted i bits to the right, is compared with the current partial remainder R(i). A full comparison of this nature takes time proportional to the length of the operands. The quotient bit q(n+1-i) is set to 0 or 1 according to whether

```
                                    quotient Q = [q(5) ——— q(1)]
                                    divisor  V = [v(4) ——— v(1)]
                            0 1 1 1 1
                      _________________
divisor V = 1 1 0 1 ) 1 1 0 0 1 1 0 1   dividend D = R(0)
                      0 0 0 0           q(5) V
                      _______________
                      1 1 0 0 1 1 0 1   R(1)
                        1 1 0 1         q(4) 2^-1 V
                      _______________
                        1 1 0 0 1 0 1   R(2)
                          1 1 0 1       q(3) 2^-2 V
                        _____________
                          1 1 0 0 0 1   R(3)
                            1 1 0 1     q(2) 2^-3 V
                          ___________
                            1 0 1 1 1   R(4)
                              1 1 0 1   q(1) 2^-4 V
                            _________
                              1 0 1 0   R(5) = R remainder
```

Figure 4.1. Example of standard division method.

(2↑-i)V is less than or greater than R(i).

The new partial remainder is computed as

$$R(i+1) \longleftarrow R(i) - q(n+1-i)\cdot(2\uparrow-i)V.$$

In hardware implementations, it is more convenient to shift the partial remainder to the left, changing the above relation to

$$R(i+1) \longleftarrow 2R(i) - q(n+1-i)V.$$

However, when this is done, the final partial remainder is the required remainder shifted to the left n positions. An example of this is shown in Figure 4.2.

The principal problem in fixed-point division is determination of the bits of the quotient Q. This can be done with a comparator circuit: if $V > 2R(i)$, then $q(n+1-i) = 0$; otherwise, $q(n+1-i) = 1$. For large values of n, a combinational comparator circuit is impractical, and the method usually used is to determine the quotient bit by subtracting V from 2R(i) and examining the sign of $2R(i) - V$. If $2R(i) - V < 0$ then $q(n+1-i) = 0$; otherwise, $q(n+1-i) = 1$.

In the case that the quotient bit is determined to be 1, this trial subtraction also yields the new partial remainder R(i+1). It is clear that the determination of the quotient bits and the computation of the partial remainders can be combined. Two major division algorithms are distinguished according to the way that they handle this

```
divisor V
 1 1 0 1                                                  quotient Q
              1 1 0 0 1 1 0 1    dividend D
              0 0 0 0            q(5) V                        0
            -----------------
              1 1 0 0 1 1 0 1    R(1)
            1 1 0 0 1 1 0 1 0    2R(1)
              1 1 0 1            q(4) V                      0 1
            -----------------
              1 1 0 0 1 0 1 0    R(2)
            1 1 0 0 1 0 1 0 0    2R(2)
              1 1 0 1            q(3) V                    0 1 1
            -----------------
              1 1 0 0 0 1 0 0    R(3)
            1 1 0 0 0 1 0 0 0    2R(3)
              1 1 0 1            q(2) V                  0 1 1 1
            -----------------
              1 0 1 1 1 0 0 0    R(4)
            1 0 1 1 1 0 0 0 0    2R(4)
              1 1 0 1            q(1) V                0 1 1 1 1
            -----------------
              1 0 1 0 0 0 0 0    R(5) = 2*R
```

Figure 4.2. The same example, modified for machine implementation.

combination.

The straightforward technique, described in Algorithm 4.1, always performs a subtraction of the form

$$R(i+1) \longleftarrow 2R(i) - V.$$

When the result of the subtraction is negative (the corresponding quotient bit is 0) an addition is performed:

$$R(i+1) \longleftarrow R(i) + V$$

to restore the partial remainder to its correct value. This technique is known as <u>restoring division</u>. If 0's and 1's occur with equal likelihood in the quotient then this algorithm requires n subtractions and an average of n/2 additions.

The computation time of this algorithm can be reduced by a technique called <u>nonrestoring division</u>. This is based on the observation that a restoration of the form

$$R(i) \longleftarrow R(i) + V$$

is always followed in the next step by a subtraction:

$$R(i) \longleftarrow 2R(i) - V.$$

These two operations can be combined into

$$R(i) \longleftarrow 2R(i) + V.$$

Thus, if $R(i) > 0$ then $q(n+1-i) = 1$ and $R(i+1)$ is computed by subtracting V; otherwise, $q(n+1-i) = 0$ and $R(i+1)$ is computed by adding V. This procedure is given as Algorithm 4.2.

Algorithm 4.1

Restoring division algorithm

```
begin
   comment  This algorithm finds the quotient, Q =
     [q(n)————q(1)], and the remainder, R =
     [r(n)————r(1)], after being given the dividend in D
     and the divisor in V;

   Q <— 0;
   R <— D;

   for i <— n step -1 until 1 do
     begin
        R <— 2*R;
        Q <— 2*Q;
        R <— R - V;

        if R < 0 then
          begin
             q(1) <— 0;
             R <— R + V
          end
          else q(1) <— 1
     end

 end.
```

Algorithm 4.2

Non-restoring division algorithm

```
begin
   comment  This  algorithm  finds  the  quotient,  Q  =
     [q(n)-----q(1)],   and   the   remainder,   R   =
     [r(n)-----r(1)], after being given the dividend in D
     and the divisor in V;

   Q <— 0;
   R <— D;

   Q <— 2*Q;
   R <— 2*R;
   R <— R - V;

   for i <— n step -1 until 1 do
     begin

        if R < 0 then
          begin
             q(1) <— 0;
             R <— R + V
          end
        else
          begin
             q(1) <— 1;
             R <— R - V
          end;

        Q <— 2*Q;
        R <— 2*R

   end;

   comment The final remainder may have to be corrected;
   if R < 0 then R <— R + V

end.
```

Nonrestoring division always requires exactly n additions/subtractions, as can be seen from the specification of the algorithm.

4.2 A New Proposal for a Divider

Another division technique often employed involves the use of iteration to approximate the required quotient. This is particularly applicable in floating-point division, but as we shall see, can also be adapted to our current problem.

The process of division can be split into two distinct phases: computation of the reciprocal of the divisor, followed by multiplication of the reciprocal by the dividend. [illegible] floating-point division, this is all that is required. In fixed-point division, however, this must be followed by a third phase which computes the remainder: the largest integer in the product of the reciprocal and the dividend, multiplied by the divisor, must be subtracted from the dividend, to obtain the desired remainder. This technique will be used to develop a divider which operates in linear time, using the concepts developed earlier in this thesis. The reciprocal algorithm to be described involves only additions/subtractions and multiplications, all of which can be performed on-line in linear time. Also, the computations described for the second and third phases of the division method involve only these three operations. As shall be seen however, certain truncations necessary in the division algorithm prevent it from being on-line.

It should not come as a surprise that the division algorithm is not on-line. Division poses a considerably more difficult problem than the other three operations, for division is inherently a left-to-right process. The traditional algorithm produces the most significant bit of the quotient first and the least significant bit last. Also, it requires knowledge of the full dividend and divisor before any computation can begin. Furthermore, this algorithm does not produce any bits of the remainder until after the last bit of the quotient is complete. The divider to be proposed accepts bits of the divisor and dividend beginning with the least significant bit, and performs some internal computation at each time step. After the complete dividend is in (assuming the dividend is longer than the divisor), it produces the first (least significant) bit of the quotient. The remaining bits of the quotient emerge after each time step. The first (least significant) bit of the remainder is ready one time step after the first bit of the quotient, and successive bits emerge in the same manner as before.

One can easily show by counterexample that it is not possible to produce a truly on-line divider. Given the rightmost bits of a dividend and a divisor, it is impossible to determine in general the rightmost bits of either the quotient or the remainder. In fact, it is not possible to do so until the entire divisor and dividend are known, for a change in the most significant bit of either will change the results unpredictably. However, it is possible to do some "preprocessing" before the entire inputs are known, so that the first bits of the outputs can be produced as quickly as possible after the complete divisor and dividend are known. This is the principle used in the subsequent development.

Let us now consider an iterative algorithm for computing the reciprocal of a number. Since the reciprocal of an n bit integer is a fraction, we shall define the reciprocal to be

$$\mathbf{rec}(x) = 2{\uparrow}(2n-1) \div x.$$

Observe that the reciprocal is normally an n bit integer, except when x is a power of two, in which case it is an n+1 bit integer. A recursive algorithm for computing reciprocals is given as Algorithm 4.3. An example illustrating its execution is given in Figure 4.3.

It is shown in [2] that Algorithm 4.3 converges onto the reciprocal of an n bit number in log n iterations. Each iteration is a call to procedure **rec** and involves a computation of the form $A - B^2*C$. A device for computing the expression $A - B^2*C$ could be constructed in a similar manner as were the adder and multiplier in earlier sections. These could each be capable of execution in on-line fashion. In

Algorithm 4.3

Recursive function for computing reciprocals

```
function rec([d(k)———d(1)]);
   comment This function finds the reciprocal, R =
      [r(k)———r(1)], after being given the divisor, D =
      [d(k)———d(1)];

begin

   if k = 1 then [r(2),r(1)] <— [10]
   else

      begin

         [c(k/2+1)———c(1)] <— rec([d(k/2)———d(1)]);

         [t(k)———t(1)] <— [c(k/2+1)———c(1)] * 2↑(3k/2)
              - [c(k/2+1)———c(1)]$^2$ * [d(k)———d(1)];

         [r(k+1)———r(1)] <— [t(k+2)———t(2)]

      end;

   [r(k+1)———r(1)]

end.
```

Example: **rec**(10011001)

rec(1) = 10

rec(10) = 10 * 1000 - 10^2 * 10

= 1000

truncate 1: 100~~0~~ —> 100

rec(1001) = 100 * 10^6 - 100^2 * 1001

= 1110000

truncate 3: 1110~~000~~ —> 1110

rec(10011001) = 1110 * 10^{12} - 1110^2 * 10011001

= 1101010110111100

truncate 7: 11010101~~1011100~~ —> 11010101

Figure 4.3. Example of reciprocal evaluation.

fact, the entire reciprocal algorithm could be executed on-line were it not for the fact that each iteration requires truncations from the least significant bits, as illustrated in Figure 4.3. Each bit which is truncated from the right introduces a delay of one time step. The total delay introduced in this manner is

$$1 + 3 + 7 + \ldots + [2\uparrow(\log n) - 1]$$
$$= 2 + 4 + 8 + \ldots + n - \log n$$
$$= 2n - 2 - \log n \text{ steps.}$$

Having discussed the device for computing reciprocals, let us see what other hardware is needed to complete the division process. As mentioned above, a multiplier is needed to multiply the reciprocal by the divisor to produce the quotient. This has been thoroughly examined in the previous chapter. Next, for the last phase of division, a device which computes the expression A - B*C is required to compute the remainder. This can again be constructed using a similar design as was applied for the on-line adder and the on-line multiplier. The details will not be given, but it is conceptually clear that such a device accepting three inputs and producing one output could be built.

A summary of the parts needed for the divider is as follows:

Phase 1: devices for $A - B^2*C$ — log n of these
Phase 2: device for $A*B$ — 1 of these

Phase 3: device for A - B*C — 1 of these.

Now let us compute the total delay of the divider, i.e. the time from when the rightmost bits of the divisor and dividend are input to the time when the rightmost bit of the quotient is output (the rightmost bit of the remainder lags one step behind the quotient). Throughout the entire process, the inputs are expected to be flowing into the divider, one pair of bits per time step. As shown above, the delay introduced by phase 1 (computation of the reciprocal) is 2n - 2 - log n steps. The delay due to phase 2 (computation of the quotient) is only 2n steps since the product of an n bit reciprocal and a 2n bit dividend is a 3n bit number, of which only the most significant n bits comprise the quotient of interest. The delay corresponding to phase 3 (computation of the remainder) is one step. This is possible only because all three computations are being performed simultaneously. Hence, the total delay is:

4n -1 - log n steps.

chapter five

ASSOCIATIVE MEMORY APPLICATIONS

Chen's scheme[11] for adding columns of numbers using bit-slice arithmetic in associative memory is discussed. The problem of extending it to multiplication is introduced and solved. Other computations such as vector and matrix multiplications are studied in light of the preceding results. Some of these operations, such as computation of determinants, are less practical than others. The feasibility of designing the required data manipulator for associative memory operations is examined.

5.1 An Associative Processor Model

Let us consider a list of records in memory, each record containing a fixed number of fields, e.g.: a person's name (NAME), an identification number (ID), and an age (AGE). Most information storage and retrieval problems involve accessing certain fields within a set of records in answer to a question such as: "What are Taylor's ID number and age?" If a conventional random access memory is being used, it is necessary to specify exactly the physical address of the Taylor entry in the table. Although the programmer need not know it specifically, somehow a search through a series of addresses is required. At each address, a comparison must be made between the name residing there and 'Taylor'. Only after a match is found and the address is known can the required information be extracted. The instruction for obtaining this information contains the acquired address. This address has no logical relationship to Taylor; hence it can be viewed as an artificial construct which adds to programming complexity.

An alternative approach is to search the entire table using the NAME field as an address. In such a system, the request for the Taylor data would be in the form of an instruction which contains only the name 'Taylor' and a specification that it is the NAME field that is to be scanned. The conventional method of implementing this approach involves scanning all entries in the table sequentially and checking their NAME fields until a match is found. Sequential searching of this type is easily implemented with random access memories, but it is very slow. It can be speeded up if the table is sorted according to the field being searched and through a variety of other techniques. An associative memory eliminates this difficulty by simultaneously examining all entries in the table and selecting the one that matches the given address. It is clearly useful to be able to select other fields of the record to use as an address. In the current example, Taylor's ID number could be used as an address.

In general, an associative memory is one in which any stored item can be accessed directly by using the contents of the item in question, generally some specified subfield, as an address. Associative memories are also commonly called content addressable memories[20, 42, 43]. Figure 5.1 illustrates this property of an associative memory. The records are stored one per row. It is assumed that the width of the memory is sufficient to hold the desired number of fields and that there are enough rows to hold all records in the table. The record being searched for is specified via two registers: the comparator and the mask. The key to be searched for is placed in the comparator. The mask register masks out those bits of the comparator which are not significant to the search, thus allowing the selection of specific fields. The response to the search is given in the responder register, which has the length of the memory. Those rows in which there is a match show a 1 bit; the others don't change from their previous state. It is quite possible to have several matches.

Since all words in the memory are required to compare their contents with the comparator simultaneously, each must have its own match circuit. The match and response circuits make associative memories much more complex and expensive than conventional memories[59]. The advent of LSI techniques has made associative memories economically feasible. However, cost considerations still limit them to applications where a relatively small amount of information must be accessed very rapidly, e.g. memory address mapping.

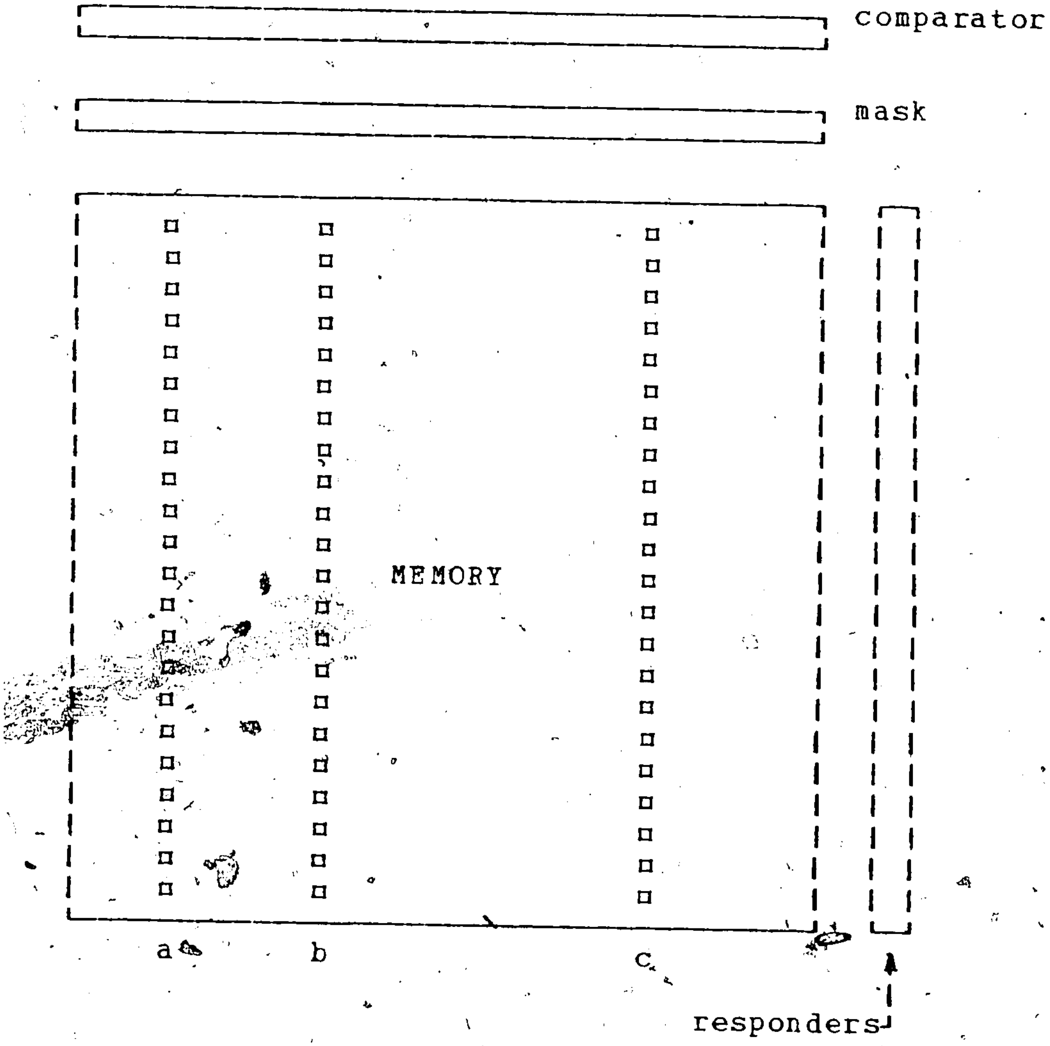


Figure 5.1. An associative processor with bit-slice capabilities.

When the match/mismatch logic in an associative memory is modified to include certain arithmetic capabilities at each memory word, the unit is often known as an <u>associative processor</u>. There are several variations of this in existence[59]. The unique feature which will be of primary importance in this thesis is the bit-slice operation capability. There are a number of machines of this type in existence, perhaps the most well known one being the STARAN, manufactured by Goodyear Aerospace Corporation[7].

Suppose it is necessary to obtain the sum of many pairs of numbers stored in the memory. In a sequential processor, each pair must be passed, one at a time, through a single adder; if many pairs must be added, the total computation time is proportional to the number of pairs. But in the associative processor, each word in the memory has its own simple adder, and all the adders can form sums simultaneously. Although each one is relatively slow and takes longer to obtain an individual sum than would a high-speed adder, the total elapsed time for obtaining all the sums is orders of magnitude less than with the single central adder. The total time is no longer dependent on the number of operands, assuming a sufficiently large memory.

Typically, an associative processor allows logical bit-slice operations on the contents of the memory. This "vertical processing" is also illustrated in Figure 5.1.

Here, a logical function (such as 'and' or 'or') of columns a and b is performed in one processor cycle and the result is placed in column c. More advanced operations such as addition of columns of words can be implemented from this capability either through hardware or software. In either case, the time taken is dependent only on the width of the words to be added, rather than on their widths as well as the number of operands, as would be the case with a conventional processor.

5.2 Addition of a Sequence of Numbers in an Associative Processor

The usefulness of an associative processor in adding many pairs of numbers has been illustrated. A more difficult problem in this environment is the computation of the sum of a set of numbers. Suppose the sum of m numbers A1, A2, . . ., Am is desired. We will denote the bit representations of the n-bit numbers as before:

$$A1 = [a1(n) \text{———} a1(1)]$$

$$A2 = [a2(n) \text{———} a2(1)]$$

$$\bullet$$

$$\bullet$$

$$\bullet$$

$$Am = [am(n) \text{———} am(1)].$$

The method which comes to mind immediately is to add pairs, then pairs of pairs, and so on. This simple algorithm is readily implemented in an associative processor and takes time O(n log m) bit steps.

Much more use of the vertical processing capability of the associative processor can be made by using a scheme discovered by Chen[11]. This algorithm makes use of the on-line property of ripple-carry addition. The general idea is the same as the method mentioned above. The significant gain in speed is obtained by overlapping the bit operations, as is allowed by the on-line property. The operands are divided into two halves. In one time step, the least significant bit column of the first half is added to the least significant bit column of the second half. This produces two columns: a sum and a carry column. This is illustrated by example in Figure 5.2. There, this first addition is indicated by boxes. Next, rather than storing the sum and proceeding to the next operation, the sum is split into two half columns and placed under the second least significant bits of the original operands. The carry column is stored somewhere in memory to be used in the next addition step. The processor now moves on to add the second least significant bit columns plus the carry column from the first addition. This time, it will also be adding the least significant bits of the second set of additions (the carry for these is zero, so it is assumed that the entire temporary carry column is initialized to zero at the start). This process repeats. The example in Figure 5.2 shows one addition to completion. For the purpose of clarity, the

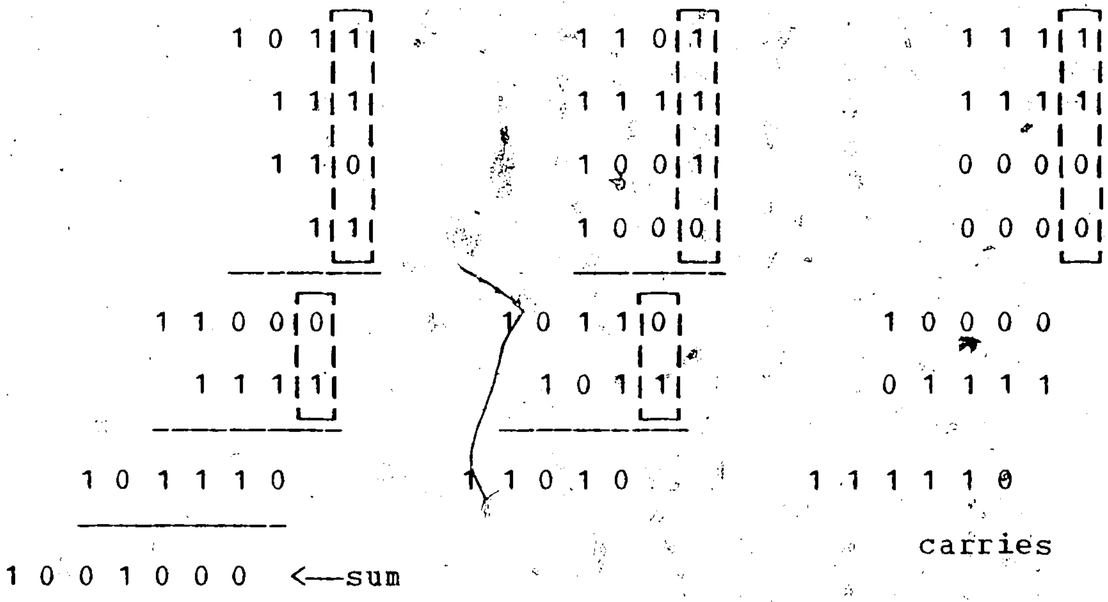


Figure 5.2. Example of bit-slice on-line addition in an associative memory.

carry column for each step is shown (moving right to left). In practice, only one column is necessary, with each successive carry overwriting the previous one. At the completion of the addition, the contents of the memory may be visualized as in Figure 5.3.

This algorithm will now be described more formally. The m operands are layed out in memory in two columns. Assuming m to be a power of 2, place the operands A1, A3, . . ., A(m-1) in one column, and A2, A4, . . ., $A\underline{m}$ in another column, as shown in Figure 5.4. Assume the least significant bit of the first set of numbers to be in column p and that of the second set to be in column q. Columns r and s will be used for temporary storage. It is, of course, necessary to have a sufficiently large memory available. As shall be seen, the number of columns necessary is at least 2n + 4log m and the required number of rows is m.

Assume the memory to be addressed sequentially with indices increasing from left to right and from top to bottom. As column operations are allowed, a special notation is in order for columns: the index for the given column is underlined. Hence, $\underline{r}$ denotes the rth column from the left. Logical operations are allowed on columns. Hence,

$$\underline{t} \longleftarrow \underline{r} \oplus \underline{s}$$

denotes the operation of taking the 'exclusive or' of columns $\underline{r}$ and $\underline{s}$ and putting the result in column $\underline{t}$.

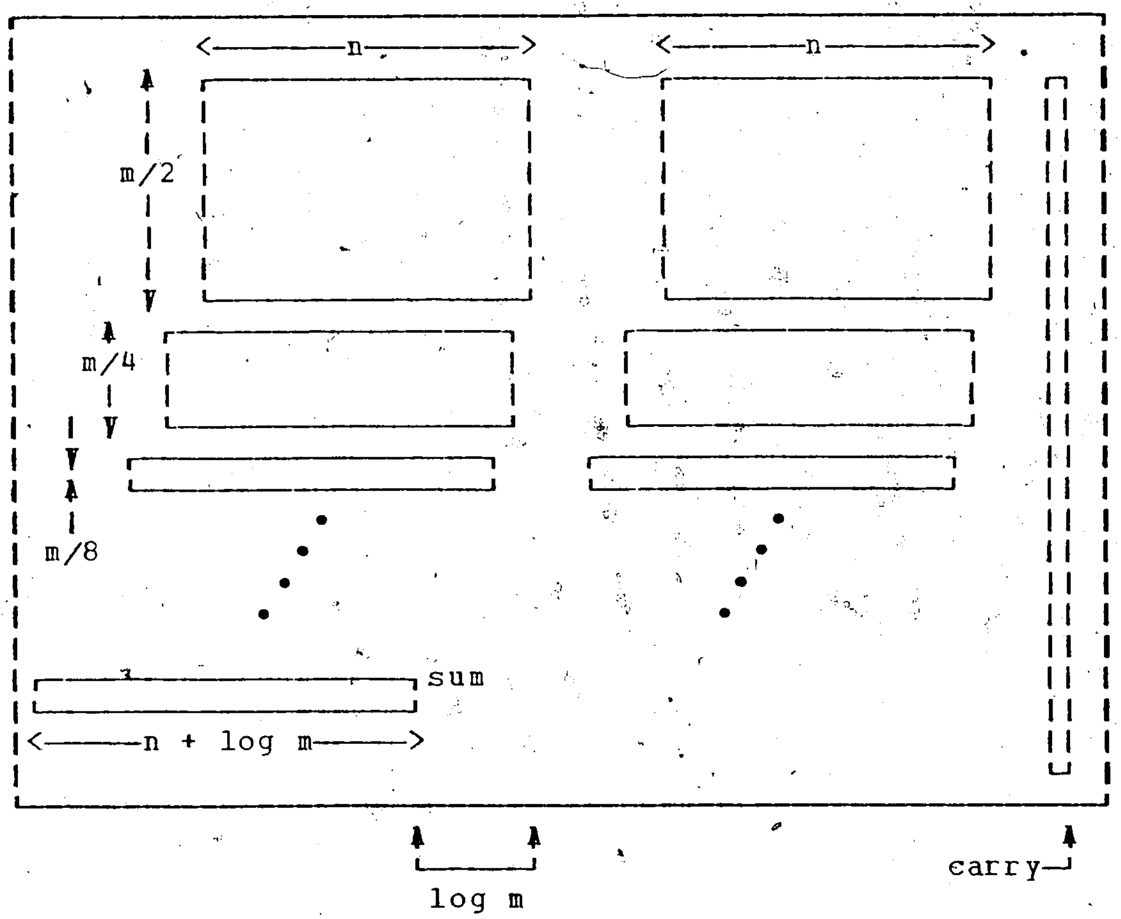


Figure 5.3. Addition of m numbers in an associative memory.

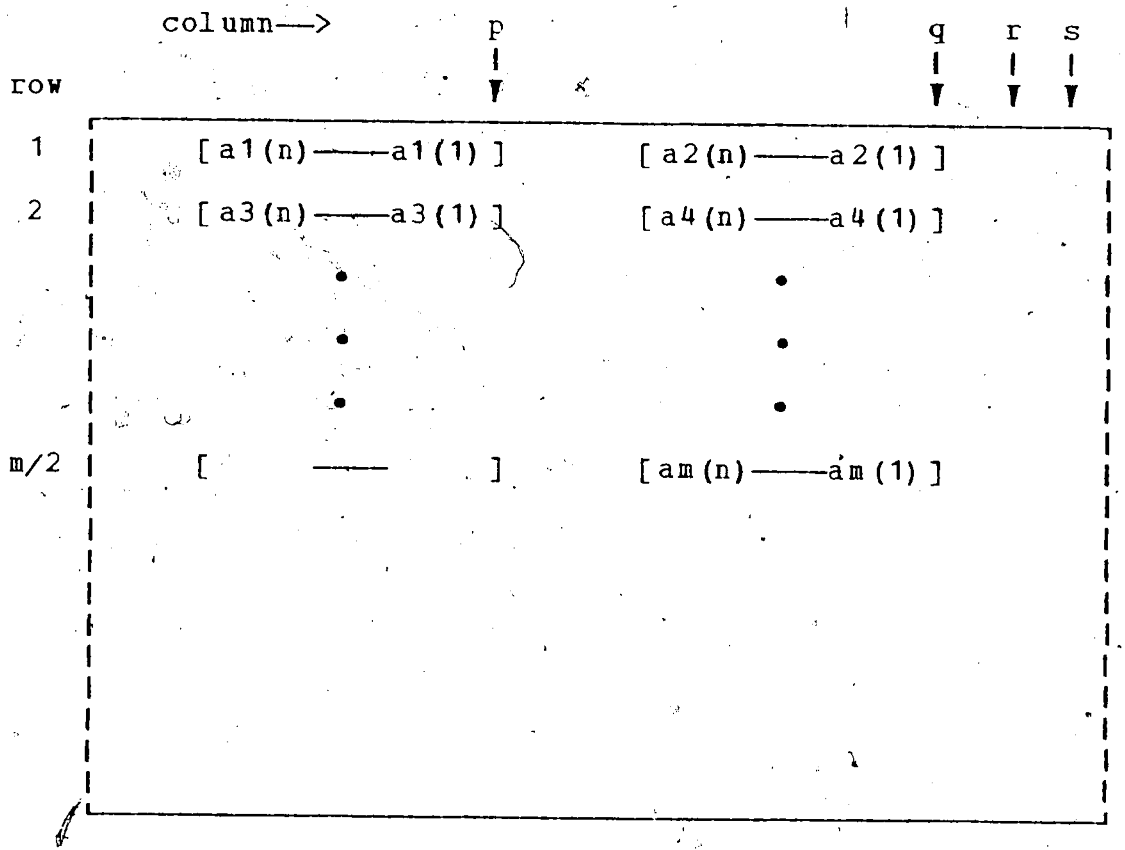


Figure 5.4. Initial layout of operands in memory prior to addition.

Two functions, ODD and EVEN, are also required for splitting a column in half. ODD[$\underline{s}$] is a column having half the length of $\underline{s}$ which contains the odd elements of $\underline{s}$. Similarly, EVEN selects alternate elements starting with the second. A column with an index, e.g. $\underline{p}(k)$ denotes the column $\underline{p}$ from row k on. $\underline{0}$ denotes a column with all zero entries (this is not entirely consistent but will be found very useful).

With this notation established, the algorithm can now be formalized; see Algorithm 5.1. The feasibility of performing the data manipulating functions such as EVEN and ODD, and moving columns from one location in memory to another has been studied by Feng[16, 17]. As can be deduced immediately from the main **for** loop, this algorithm takes time O(n + 2log m) to add m numbers of n bits each. This compares favorably with the O(n log m) time taken by the algorithm mentioned at the beginning of this section. The best serial algorithm would need O(nm) time steps.

As a rigid means of comparison of the various algorithms to follow in this chapter, a uniform procedure for deducing the worst case times of execution is introduced. This is illustrated by the following derivation for the present algorithm. In the addition of m numbers of length n, the first bit of the sum is ready after $\lceil \log m \rceil$ steps. The maximum value of the sum is

Algorithm 5.1

Associative memory addition

```
begin
    comment Initialize: Assume the m operands to be added
      are layed out in memory as shown in Figure 5.4;

    $\underline{r} \leftarrow \underline{0}$;

    for i <— 0 until n -1 + 2log m do
    begin

        comment Add columns $\underline{p-i}$, $\underline{q-i}$, $\underline{r}$. Put the sum in
          column $\underline{s}$ and the carry in column $\underline{r}$;

        $\underline{s} \leftarrow \underline{p-i} \oplus \underline{q-i} \oplus \underline{r}$;
        $\underline{r} \leftarrow (\underline{p-i}) \bullet (\underline{q-i}) + (\underline{p-i}) \bullet \underline{r} + (\underline{q-i}) \bullet \underline{r}$;

        $\underline{p-i-1}(m/2+1) \leftarrow \mathrm{ODD}[\underline{s}]$;
        $\underline{q-i-1}(m/2+1) \leftarrow \mathrm{EVEN}[\underline{s}]$

    end

end.
```

$$m(2\uparrow n - 1).$$

Hence, the sum is complete after

$$\lfloor \log m(2\uparrow n - 1)\rfloor + 1 + \lceil \log m\rceil \text{ steps}$$
$$= \lfloor \log m + \log(2\uparrow n - 1)\rfloor + 1 + \lceil \log m\rceil$$
$$\pm 2\log m + n \text{ steps.}^{+}$$

The addition algorithm requires m rows of associative memory. The number of columns required is computed as follows:

$$2\{\lfloor \log m(2\uparrow n - 1)\rfloor + 1 + \lceil \log m\rceil\} - 1$$
$$\pm 2\log m + 2n + 2\log m$$
$$= 4\log m + 2n.$$

5.3 Multiplication of a Sequence of Numbers

The application of the above algorithm for addition of a set of numbers in associative memory translates readily to multiplication. This follows since the on-line property of the multiplication algorithm presented in Chapter 3 is the same as that for addition in Chapter 2. The fact that this would be possible if a linear on-line multiplication algorithm were available was recognized by Chen in [11]. A formal description of this procedure is given as Algorithm 5.2.

The similarities between this algorithm and Algorithm 3.1 are obvious. The key difference is that operations are

+The symbol "±" is used to denote "approximately equal to"

Algorithm 5.2

Associative memory multiplication

```
begin
  comment This algorithm computes the product of m
    numbers each of n bits in an associative memory.
    Assume the numbers are placed in memory starting in
    the first row, in two equal columns, the first ending
    at column p and the second at column q. The product
    will appear in row 2m of the memory;

  for i <— 0 until nm - 1 do
     begin
        comment Five column groups of work space will be
          needed: A, B, C1, C2, S. These correspond
          directly to their namesakes in Algorithm 3.1.
          They must all be initialized to 0;

        A+i <— 0;
        B+i <— 0;
        C1+i <— 0;
        C2+i <— 0;
        S+i <— 0
     end;

  for i <— 0 until nm - 1 + log m do
     begin

        begin
           for j <— 0 until i do A+i+j-1 <— p+j • q+i
        end;

        begin
           if i > 0 then
             for j <— 0 until i - 1 do
                         B+i+j-1 <— q+j • p+i
        end;
```

Algorithm 5.2 (continued)

```
        for j <— nm step -1 until 0 do
            begin
             comment Initialize C1 = C1-1 = C2 = C2-1 = 0;
               t+1 <— S23[A+j, B+j, C1+j-1,
                                C2+j-2, S+j];
               t+2 <— S45[A+j, B+j, C1+j-1,
                                C2+j-2, S+j];
               t+3 <— S135[A+j, B+j, C1+j-1,
                                C2+j-2, S+j];
               C1+j <— t+1;
               C2+j <— t+2;
               S+j <— t+3

               p-i-1(m/2+1)  <— ODD[S+j];
               q-i-1(m/2+1)  <— EVEN[S+j]
            end
      end
  end.
```

being done bit-slice and word parallel rather than on individual bits. This necessitates the elimination of comparisons such as are found in lines 16 and 20 of Algorithm 3.1. They are replaced by the 'and' operations of lines 12 and 16 in Algorithm 5.2. In both algorithms, the high degree of parallelism is evident, as was discussed in detail in the description of Algorithm 3.1. The data manipulations found in lines 26 and 27 are the same as those in Algorithm 5.1.

The time and space complexities of this algorithm are derived as follows, in a similar manner as was used in the previous section for Algorithm 5.1. In the multiplication of m numbers of length n, the first bit of the product is ready after

$$2\uparrow\lceil\log m\rceil - 1$$

steps. The maximum value of the product is

$$(2\uparrow n - 1)\uparrow m.$$

Hence, the product is complete after

$$\begin{aligned}
&\lfloor\log (2\uparrow n - 1)\uparrow m\rfloor + 1 + 2\uparrow\lceil\log m\rceil - 1 \text{ steps}\\
&= \lfloor m \log (2\uparrow n - 1)\rfloor + 1 + 2\uparrow\lceil\log m\rceil - 1\\
&\pm mn + m\\
&= m(n + 1) \text{ steps.}
\end{aligned}$$

Again, m rows of associative memory are required. The number of columns used is computed as follows:

$$\lfloor \log (2 \ast n - 1) \ast m \rfloor + 1 + 2 \ast \lceil \log m \rceil - 1 +$$
$$\lfloor \log (2 \ast n - 1) \ast (m/2) \rfloor + 1 + 2 \ast \lceil \log (m/2) \rceil - 1$$
$$\pm mn + m + nm/2 + m/2$$
$$= 3/2 \; m(n + 1).$$

Figure 5.5 demonstrates the multiplication algorithm by example. There, the four numbers 101, 110, 111, and 110 are multiplied to form a 12 bit product.

5.4 Vector and Matrix Operations

Certain other operations such as vector and matrix manipulations involving large numbers of additions, subtractions, and/or multiplications, can be effectively speeded up using bit-slice operations in an associative processor, dramatically demonstrating the power of this computer architecture.

Figure 5.6 shows how the dot product (also known as the scalar product) of two vectors can be readily computed in this environment. Assume the dot product of two vectors A = (A1, A2, . . ., Am) and B = (B1, B2, . . ., Bm) is to be found. If m is not even, make it even by letting the last elements Am = Bm = 0. Next, place the ~~two~~ vectors in memory as shown in the top half of Figure 5.6. The pairwise products of the elements (not the product of all the elements), P1, P2, . . ., Pm, can be readily found as discussed previously. Placing these products in another part

```
     11111               11111             11111
i—> 4321098765432 1     4321098765432 1     4321098765432 1

              101                 110    ┌──────────────┐
              111                 110    │              │
    ──────────────    ──────────────     │   carries    │
            11110              110001    │              │
    101101111100 <—product               └──────────────┘

  Carry columns after step i, for i = 1, 2, . . ., 13:

    i=1              0       i=8              11110
                     1                       110001
                                       101101111100

                    10                        11110
    i=2           1001       i=9             110001
                     0                 101101111100

                 11110                        11110
    i=3         110001       i=10            110001
                   100                 101101111100

                 11110                        11110
    i=4         110001       i=11            110001
                  1100                 101101111100

                 11110                        11110
    i=5         110001       i=12            110001
                 11100                 101101111100

                 11110                        11110
    i=6         110001       i=13            110001
            1111111100                 101101111100

                 11110
    i=7         110001
          101101111100
```

Figure 5.5. Example of bit-slice on-line multiplication in an associative memory.

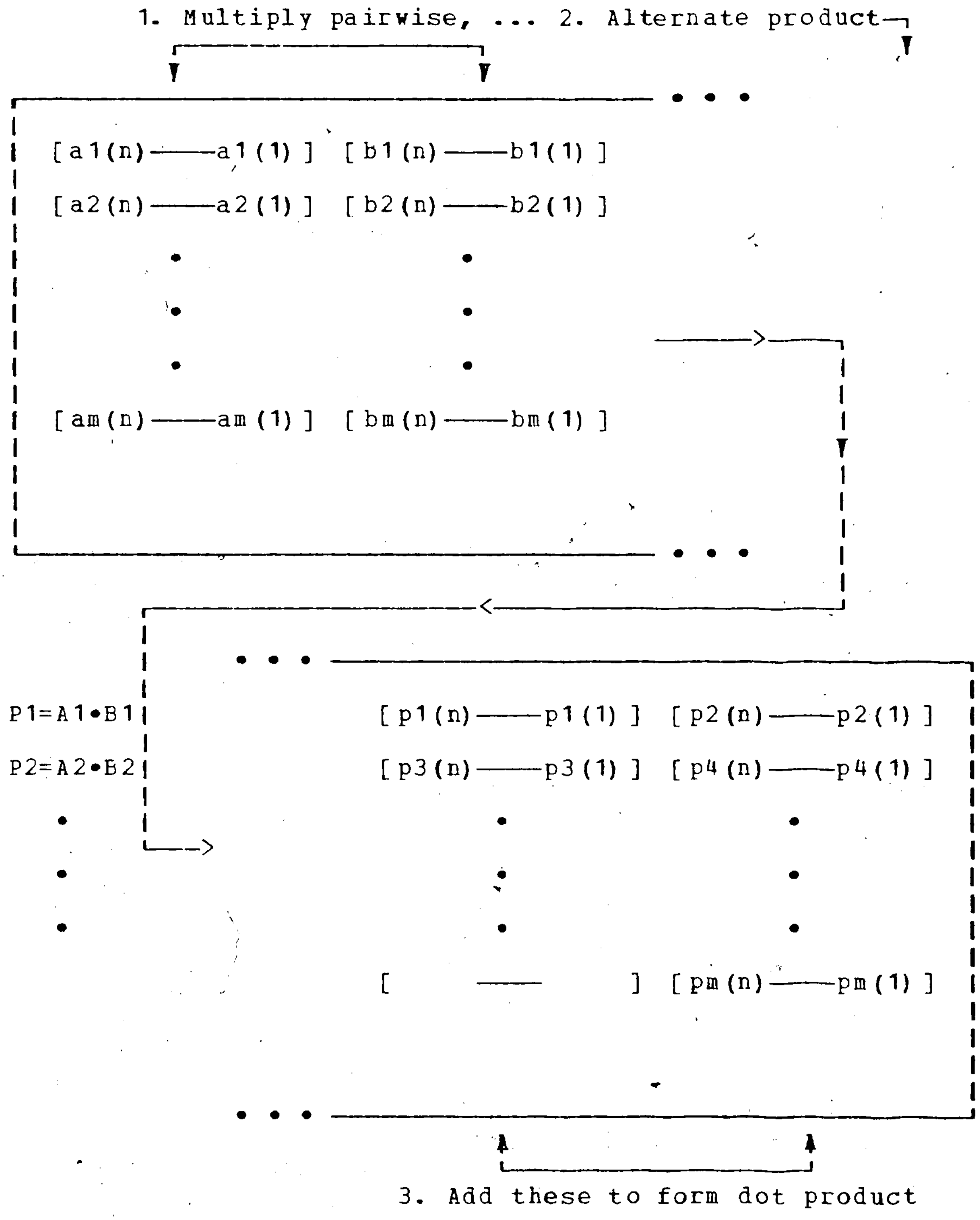


Figure 5.6. Computation of dot product.

of the memory in a column as shown in Figure 5.6, the total of these numbers can be computed by applying the addition algorithm, Algorithm 5.1.

A discussion of the time and space complexities of the dot product algorithm is of interest. In the first half, the pairwise multiplication, the first bit of Ai•Bi for each i is ready after 1 step. The first bit of the sum is ready after $\lceil \log m \rceil$ steps. Therefore, the first bit of the dot product is ready after

$$\lceil \log m \rceil + 1$$

steps. If each number has length n, the maximum value of each number is $2{\uparrow}n - 1$. Hence, the largest possible dot product is

$$m(2{\uparrow}n - 1){\uparrow}2.$$

Thus, the length of the largest possible dot product is

$$\lfloor \log m(2{\uparrow}n - 1){\uparrow}2 \rfloor + 1.$$

The complete dot product is ready after

$$\lfloor \log m(2{\uparrow}n - 1){\uparrow}2 \rfloor + 1 + \lceil \log m \rceil + 1$$
$$\pm 2\log m + 2n \text{ steps.}$$

As deduced from Figure 5.6, m rows of memory are required. For storing all the elements, 2n columns are needed. Adding m numbers, each of length 2n requires 4n + 4log m columns. Hence, the total number of columns required for computing the dot product is

$$6n + 4\log m.$$

The computation of the product of two m x m matrices is essentially the computation of m^2 dot products, each of vectors of length m. Hence, this operation can be easily performed in the same manner as the dot product, the chief difference being that m^3 rows of memory, rather than m, are required.

The high-speed computation of the determinant of a matrix is possible in associative memory if one allows virtually unlimited memory space. In a serial processor, the algorithm for computing determinants directly from the definition requires exponential time; in an associative processor, the exponential time complexity can be transformed into a linear time complexity and an exponential space complexity. Although the algorithm has no practical value, it is of interest for this reason.

The definition of the determinant of an m x m matrix includes the computation of a set of m! cofactors, each consisting of the product of m factors selected from the m^2 elements of the matrix according to one of the m! permutations. Those cofactors whose permutations are of like parity are then added to form two sums, and the sum of the even cofactors is subtracted from the sum of the odd cofactors to produce the determinant.

Assuming a large amount of associative memory is available, the elements to be multiplied to form the cofactors can be layed out one "under" the other in a group of m! sets of m rows each. The cofactors can then all be computed simultaneously. The even cofactors must now be summed up. At the same time, the sum of the odd ones can also be found. The m!/2 even cofactors are placed in two columns, each having length m!/4 and are then presented to the addition algorithm discussed previously. The m!/2 odd cofactors are treated similarly at the same time. As these two sums are being produced, their difference is computed to yield the final answer.

A computation of the time taken for this loosely described algorithm follows. The total time taken before the least significant bits of the cofactors are produced is

$$2\uparrow\lceil\log m\rceil - 1 \text{ steps.}$$

After this, addition of the cofactors can be initiated. As there are m!/2 even cofactors to be added, the first bit of the sum of the even cofactors is complete after a delay of

$$\lceil\log (m!/2)\rceil \text{ steps.}$$

The odd cofactors take the same time to be added. These two sums are then subtracted, producing the first bit of the final result after another delay of 1 step. The total delay until the first bit of the determinant is computed is

$$2\uparrow\lceil\log m\rceil - 1 + \lceil\log (m!/2)\rceil + 1 \text{ steps.}$$

The maximum value of the determinant is

$$m!\bullet(2\uparrow n - 1)\uparrow m.$$

Hence, the time taken for computing the determinant of an m x m matrix, each of whose elements is of length n, is:

$$2\uparrow\lceil \log m \rceil - 1 + \lceil \log (m!/2) \rceil + 1 +$$

$$\lfloor \log m! \bullet (2\uparrow n - 1)\uparrow m \rfloor + 1$$

$$= 2\uparrow\lceil \log m \rceil + \lceil \log (m!/2) \rceil +$$

$$\lfloor \log m! + m \log (2\uparrow n - 1) \rfloor + 1$$

$$\pm\ m + \lceil (1/2) \log m + m \log m - m \log e \rceil +$$

$$\lfloor (1/2) \log m + m \log m - m \log e +$$

$$m \log (2\uparrow n - 1) \rfloor +$$

$$\pm\ m + (m + 1/2) \log m - m \log e + (m + 1/2) \log m -$$

$$m \log e + mn$$

$$\pm\ (2m + 1) \log m + mn.$$

The full algorithm requires m•m! rows of memory. For the multiplication of m numbers, (3/2)m(n + 1) columns are required. Then, to add m!/2 numbers of length mn, twice, requires

$$2[4 \log (m!/2) + 2mn]$$

$$= 8 \log (1/2) + 8 \log (m!) + 4mn$$

$$\pm\ (8m + 4) \log m \text{ columns.}$$

A summary of the time and space requirements of the various algorithms discussed in this chapter is given in Figure 5.7.

[+] This makes use of Stirling's approximation for factorials: $x! \pm (2\pi x)\uparrow(1/2) \bullet x\uparrow x \bullet e\uparrow -x$

PROBLEM (all words of length n)	Circuits required (parallel bit-slice)	Bit steps required	Associative memory required	
			rows	columns
sum of m numbers	1 adder	n + 2log m	m	2n + $4\log m$
product of m numbers	1 multiplier	m(n+1)	m	3/2 m(n+1)
dot product of two vectors of length m	1 multiplier & 1 adder	2n + 2log m	m	6n + 4log m
product of two mxm matrices	1 multiplier & 1 adder	2n + 2log m	m^3	6n + 4log m
determinant of an mxm matrix	1 adder, 1 multiplier & 1 subtracter	(2m+1)log m +mn	m•m!	(8m+4) •log m

Figure 5.7. Summary of applications of linear on-line circuits in associative memory.

chapter six

APPLICATIONS IN CRYPTOGRAPHY

There has been considerable mention in the recent literature[21, 50, 14, 29, 36] about "public-key cryptosystems". A necessary component of such a public-key system implementation, as well as for certain other systems, is a hardware device for rapid modular exponentiation. A proposed method for this in time $O(n^2)$, using the linear on-line concepts developed in this thesis, is given.

6.1 What is a Public-Key Cryptosystem?

This chapter deals with a most interesting and useful application of the bit-sequential on-line arithmetic concepts developed in the preceding sections; namely to the encoding and decoding of messages in a "public-key cryptosystem".

Recently, a major advance has been made in the area of communications security—that of a practical way to implement public-key cryptosystems. Public-key cryptosystems make use of a method for encryption and decryption in which the encryption key is different from the decryption key. Not only are the keys different, but revealing one doesn't provide any useful help in determining the other. A particular value for the one key does, of course assign a value for the other, but for all practical purposes, the other is impossible to find without further information. To say that it is impossible, for all practical purposes, to find one key given the other, is to imply that it is not computationally feasible within reasonable time and space constraints. There is, of course, always the trial and error technique of trying all possibilities—a completely impractical method for sufficiently large keys.

As its name implies, a major implication of this scheme is that encryption keys can be made public; literally anyone can have access to them without threatening the security of encrypted communications. The decryption key is kept private; there is never any need for anyone to communicate his decryption key to anyone else. In the subsequent text, the encryption and decryption keys will often be referred to as the public and private keys, respectively. This scheme eliminates the need for a secret transferral of keys, as is the case with conventional encryption methods. In systems using conventional methods, before two parties can communicate, they must agree on a key to be used for both the encryption and decryption. This key must be kept secret as well, or someone who intercepts a message will be able to read and/or modify it. This agreement on a key is generally either very expensive and time-consuming or relatively insecure. In a public-key cryptosystem, it is necessary only for a central controller to distribute a private key to each user of the system. No previous communication between a set pair of users is necessary before they can begin to send encrypted messages to each other. In a system with n users, the number of keys to be distributed in a public-key cryptosystem is n, whereas in a conventional system, if each user is to be able to communicate with every other, n(n-1) keys must be distributed.

Hence, the problem of key distribution is largely eliminated in a public-key cryptosystem. There is still the need to properly distribute the public keys (an intruder could otherwise give a legitimate user an invalid key, unbeknownst to the user), but this problem is minimal and can be done inexpensively.

An equally important property of a public-key cryptosystem, which is not exhibited by a conventional one is the possibility of users "signing" messages in a way that is unforgeable but easily verifiable. This property is best explained by example. Bill can send Alice a "signed" message which she can later prove to a judge to have originated from him, even though the content and the encryption key may be made public knowledge. This is possible because the keys can be used in either order: encrypting with the private key and then decrypting with the public key produces the original message. Thus, to sign a message, Bill first encrypts the message with his private key, then encrypts the result with Alice's public key. Alice now has a doubly encrypted message which she first decrypts with her private decryption key, then decrypts again with Bill's public key to arrive at the English message. Since only Bill knows his private key, only he could have created the encoded message which produced English when his public key is applied to it. This concept can also be applied to messages emanating from the central controller. Any information (such as public keys of users) retrieved from the controller can be encoded by it using its private key. The controller itself has a public key which must be known by everyone. This is best publicized via some medium such as a newspaper, which is not easily altered by a would-be intruder into the system. In this manner, the controller's public key can be periodically changed as an extra measure of security. Thus, in a sense, the key distribution problem referred to previously has now been reduced to one of properly distributing one key: the public key of the controller.

6.2 An $O(n^2)$ Algorithm for Modular Exponentiation

This brilliant concept of public-key cryptosystems was first introduced by Diffie and Hellman of Stanford University[14]. In their article, the authors discussed certain functions called "trap-door one-way functions", which would enable the implementation of a public-key cryptosystem. They did not, however, suggest a specific trap-door function which would be suitable. Such a function was first published by Rivest, Shamir, and Adleman of MIT[40]. The suitability of their function is based on the computational complexity of factoring large numbers. This chapter contains a description of this function and a proposal for a rapid way of computing it using some of the concepts developed earlier in this thesis.

In the scheme described in [40], both the encryption and decryption keys are composed of a pair of positive integers: (e, t) and (d, t) respectively. As shown in that paper, the numbers e, d and t must be of considerable length to assure the security of the system: of the order of 100 to 200 bits each. To encrypt a message M, it must first be represented as an integer between 0 and t-1. A long message must be broken into a series of blocks and each block can then be represented as such an integer. Any standard representation such as the EBCDIC code for the characters in the message will do for this purpose. Next, the message is encrypted by raising it to the e-th power modulo t. The procedure for decryption is analogous. Stated formally, the two functions E and D are as follows:

Encryption: $Q = E(M) \equiv M\uparrow e \pmod t$ for a message M

Decryption: $M = D(Q) \equiv Q\uparrow d \pmod t$ for a ciphertext Q.

Suitable methods for choosing the encryption and decryption keys are described in [40]. The need for a quick way of evaluating this set of functions is obvious. The development of inexpensive LSI has made it possible to design hardware devices to perform the function of modular exponentiation on large numbers. This will be invaluable as the need for secure communication increases. Such applications as electronic mail, electronic funds transfer (EFT), and even rapid encoding and decoding of telephone conversations can be made secure with such hardware devices.

The problem of modular exponentiation, i.e. evaluating a↑b mod c for arbitrary a, b and c, has been studied in detail by Knuth in [30]. The procedure recommended in [40] is given as Algorithm 6.1. A study of this is made in [36]. There, it is rejected, along with all other known software algorithms on the basis of insufficient speed. These evaluations are based on the use of multiple precision algorithms for multiplication and division (in evaluation of the **rem** functions) taken from Knuth[30]. It is easily seen that the execution time of Algorithm 6.1 is $O(n^3)$, where n is the maximum of the lengths of the three operands.

A more suitable algorithm which leads easily to a hardware implementation is similarly based but also makes use of the linear on-line concepts developed in this thesis. It is presented as Algorithm 6.2. As opposed to the previous algorithm, this one does not efficiently perform modular exponentiation on three arbitrary numbers of length n. Rather, if t is constant, it rapidly computes M↑e mod t for

Algorithm 6.1

Modular exponentiation by repeated squaring
and multiplication

```
begin
   comment This algorithm evaluates the encryption
     function M↑e mod t where M, e and t are arbitrary n
     bit numbers, e.g. e = [e(n)———e(1)];

   comment Initialize;
   Q <— 1;

        comment Examine the bits of e from left to right.
          Square and multiply Q by M (mod t) according to
          the value of e(i);

        for i <— n step -1 until 1 do

        begin

             Q <— rem[Q*Q, t];
             if e(i) = 1 then Q <— rem[Q*M, t]

        end

 end.
```

Algorithm 6.2

Modular exponentiation algorithm suitable
for hardware implementation

```
begin
    comment Given M = [m(n)——m(1)], e = [e(n)——e(1)]
      and t := [t(n)——t(1)] (M is given on-line with the
      least significant bit first), this algorithm computes
      Q = [q(n)——q(1)], the ciphertext corresponding to M
      (Q is also produced on-line);

u <— n + $\lceil \log n \rceil$;
v <— 2u;

comment Compute the u residues r1, r2, . . ., ru of t;
for i <— 1 until u do
  begin
     [ri(n)——ri(1)] <— rem[p, t];
     p <— 2p
  end;

q(1) <— 1;
for i <— 2 until v do q(i) <— 0;

for k <— n step -1 until 1 do
  begin
    comment Square Q;
    for j <— 1 until u do

      begin
        for j <— 1 until u do
          begin
            A(j) <— 0;
            B(j) <— 0
          end;

      if q(i) = 1 then
        begin
          for j <— 1 until i do A(i+j-1) <— q(j);
          if i > 1 then
            for j <— 1 until i-1 do
                                      B(i+j-1) <— q(j)
        end;
```

Algorithm 6.2 (continued)

```
        for j <— 2n-1 step -1 until 1 do
          begin
            t(1) <— S23[ A(j), B(j), C1(j-1),
                                        C2(j-2), s(j) ];
            t(2) <— S45[ A(j), B(j), C1(j-1),
                                        C2(j-2), s(j) ];
            t(3) <— S135[ A(j), B(j), C1(j-1),
                                        C2(j-2), s(j) ];
            C1(j) <— t(1);
            C2(j) <— t(2);
            s(j) <— t(3)
          end;

        for j <— 1 until n do Q2(j) <— 0;
        if s(i) = 1 then
          for j <— 1 until n do Q2(j) <— ri(j);

        for j <— n-1 step -1 until 1 do
          begin
            t(1) <— S13[ Q1(j), Q2(j-1), q(j) ];
            t(2) <— S23[ Q1(j), Q2(j-1), q(j) ];
            Q1(j) <— t(1);
            q(j) <— t(2)
          end

    if e(k) = 1 then
    begin
      comment Multiply Q by M;
      for j <— 1 until u do

        begin
          for j <— 1 until u do
            begin
              A(j) <— 0;
              B(j) <— 0
            end;
```

Algorithm 6.2 (continued)

```
        if q(i) = 1 then
          begin
            for j <— 1 until i do A(i+j-1) <— m(j)
          end;

        if m(i) = 1 then
          begin
            if i > 1 then
              for j <— 1 until i-1 do
                                   B(i+j-1) <— q(j)
          end;

        for j <— 2n-1 step -1 until 1 do
          begin
            t(1) <— S23[A(j), B(j), C1(j-1),
                                       C2(j-2), s(j)];
            t(2) <— S45[A(j), B(j), C1(j-1),
                                       C2(j-2), s(j)];
            t(3) <— S135[A(j), B(j), C1(j-1),
                                       C2(j-2), s(j)];
            C1(j) <— t(1);
            C2(j) <— t(2);
            s(j) <— t(3)
          end;

        for j <— 1 until n do Q2(j) <— 0;
        if s(i) = 1 then
          for j <— 1 until n do Q2(j) <— ri(j);

        for j <— n-1 step -1 until 1 do
          begin
            t(1) <— S13[Q1(j), Q2(j-1), q(j)];
            t(2) <— S23[Q1(j), Q2(j-1), q(j)];
            Q1(j) <— t(1);
            q(j) <— t(2)
          end
      end
    end;

    [q(n)———q(1)] <— rem[Q, t]

end.
```

changing values of M (e may be constant or variable). This is, of course, exactly what a public-key cryptosystem requires.

An analysis of Algorithm 6.2 will show that it performs in time $O(n^2)$. The basic elements of Algorithm 6.1 will also be found in Algorithm 6.2. Lines 9-10 perform the initialization. Line 11 is the start of the main loop for repeated squaring and multiplication. Lines 12-43 correspond to the squaring of Q and lines 44-81 correspond to the multiplying of Q by M (if the appropriate bit in e is 1). The key idea in this algorithm is the saving of approximately n divisions which are otherwise required. The technique for this is as follows.

A table of residues modulo t of powers of 2 from 1 through 2↑v, where $v = 2(n + \lceil \log n \rceil)$, is constructed. Since t is constant for many values of M, it is irrelevant how this table is computed. In fact, it could be contained in a ROM. The two sets of multiplications are performed on-line as described in Chapter 3. As a given bit of a particular product $S = [s(2n)$——$s(1)]$ is produced by the multiplier, rather than letting the result Q increase in size exponentially, the corresponding residue is added to the accumulated value of Q. As discussed in Chapter 2, this addition can be done (assuming redundant notation) in one time step. In this way, Q never gets longer than $u = n +$

$\lceil \log n \rceil$ bits. In order to keep Q in redundant notation, the temporary storage locations Q1 and Q2 are needed. The bits of Q are q(1), . . ., q(u). The residue (if the corresponding bit of S is 1) is stored in Q2, as specified in lines 35-36 and 71-72. The reduction of Q in one time step to redundant notation is done in lines 37-43 and 73-79. This entire sequence is performed n times and there are n residues to be added at most twice. Hence, the worst time for the bulk of this algorithm is $2n^2 = O(n^2)$. There is also a final division to be performed in line 82. As this is done only once, the particular algorithm used is of no interest, as long as it takes no more than $O(n^2)$ time.

An example of this method of modular exponentiation is given in Figure 6.1. Although in this example all the values of Q contain only 0 or 1 bits, since the notation is redundant, another example might also contain some 2's.

$n = 4$

$M = 111$
$e = 1011$
$t = 1101$

Residue table

p	2↑p mod t
1	1
2	10
3	100
4	1000
5	11
6	110
7	1100
8	1011
9	1001
10	101
11	1010
12	111

Partial Powers

	Q
1	111
10	1010
100	10110
101	11000
1010	10001
e = 1011	11100

$M\uparrow e \equiv 10 \pmod{t}$

Figure 6.1. Example of modular exponentiation.

chapter seven

CONCLUSIONS

The research results reported in this thesis lead to the development of a fixed-point arithmetic processor in which the individual bit steps of the elementary operations are pipelined. There are numerous implementations of pipelining in existence[39] but these usually are at the instruction level, rather than at the lower bit operation level. As reported throughout this thesis, some research similar to the present work has been reported in the literature, but this has largely concerned arithmetic operations with the operands in a most-significant-bit-first fashion. New results involving the least significant bit first have been described in this dissertation.

The ripple carry adder was discussed in preparation for the development and explanation of the multiplier. The multiplier, which performs on-line, least significant bit first, and in linear time, is described. A logical realization of the central part of the multiplier is given. It is seen to be simple in construction and, more significantly, compatible in timing with the adder and subtracter.

A by-product of this research led to a result which is of interest in any SIMD processor. The problem raised here concerns the optimal ordering of fixed-point multiplications when the product of a set of numbers of constant length is desired. The logarithmic lower bound for one multiplication, derived by Winograd[58], is used as a model. The result arrived at is that a sequence of operations dictated by a balanced binary tree is optimal. The real significance of this result lies in the corollaries relating to tighter lower bounds on certain multiple-precision operations such as matrix manipulations.

The problems imposed by division have been summarized. One can prove by counterexample that an on-line divider operating right to left cannot be constructed. Hence, the divider must, by necessity, relax some of the strict constraints of linear on-line arithmetic. The design proposed herein involves a delay, proportional to the length of the operands, between the first bits of the inputs and the first bits of the outputs. It is therefore inconsistent with the other three operations. The proposed divider makes use of the on-line multiplier and adder/subtracter in an iterative convergent reciprocal algorithm.

There are certainly other possible designs which "come close" to satisfying the requirements of a linear on-line divider. Rather than allowing for a large delay between inputs and outputs, a different design may relax the linearity restriction, for example. Another interesting possibility arises if it may be assumed that one of the inputs (the divisor or the dividend) is available in its entirety while the other input is available on-line. Similar assumptions could also be made on the outputs. This area is ripe for future research.

All results reported in this thesis involve fixed-point operands and results. It appears most natural in this case to make use of the least significant bit first in the operations. However, if the operands were in floating-point notation, this would not be the most practical direction. Nevertheless, it is of interest to know how much right-to-left arithmetic can be usefully performed on floating-point operands. A significant amount of work remains to be done in this area. Perhaps the most difficult problem here is the development of the most practical floating-point notation for all operations.

A number of applications for the above elementary results were considered. A strong correspondence between bit-slice operations in an associative processor and on-line arithmetic was demonstrated. This led to a number of interesting applications in the associative processor. The foundation of these ideas appeared in a paper by Chen[11] in which it was shown that a clever arrangement of operands in associative memory combined with a data manipulator can lead to a very fast algorithm for computing the total sum of a set of numbers. This concept was extended to the computation of the product of a set of numbers. A further extension of these ideas produced very efficient algorithms for many vector and matrix operations.

A natural question arises as to what other applications are possible in an associative processor. With bit-slice operations, matrix inversion can likely be performed more rapidly than by the use of conventional algorithms. This problem would involve the divider developed herein. The reciprocal of an entry could be defined as in Chapter 4. Two other interesting possible applications are polynomial evaluation and Fourier transforms. Both of these involve a set of numbers to be stored in an associative memory, and a sequence of operations of the kinds studied in this thesis.

The second major application of on-line arithmetic which was studied extensively is the problem of modular exponentiation. This leads to a very useful result in public-key cryptography where the loose constraints on two of the three operands allows for a quadratic algorithm for modular exponentiation. This was possible only because of the particular sequence of alternating multiplications and divisions required by the standard algorithm. Normally, division imposes a constant delay on each application, and hence, algorithms which require a large number of divisions cannot efficiently make use of the divider developed in Chapter 4. However, the multiplications and divisions were cleverly combined to produce a truly practical algorithm. Cther problems for which the best algorithms involve large numbers of divisions can possibly be speeded up by similar techniques. This is yet another area in which further research would very likely be fruitful.

BIBLIOGRAPHY

[1] Advanced Micro Devices, Inc., *Low Power Schottky Data Book*, Sunnyvale, California, 1977 {28}[+]

[2] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974 {77}

[3] Anderson, S. F., J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit, *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 34-53, January 1967 {16}

[4] Atkins, D. E., "Higher Radix Division Using Estimates for the Divisor and Partial Remainders", *IEEE Transactions on Computers*, vol. C-17, no. 10, pp. 925-934, October 1968

[5] Atrubin, A. J., "A One-Dimensional Real-Time Iterative Multiplier", *IEEE Transactions on Computers*, vol. EC-14, no. 3, pp. 394-399, June 1965 {4, 28, 29}

[6] Avizienis, A., "Signed-Digit Number Representations for Fast Parallel Arithmetic", *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 389-400, September 1961

[7] Batcher, K. E., "STARAN/RADCAP Hardware Architecture", *Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing*, pp. 147-152, August 1973 {50, 86}

[8] Briley, B., "Some New Results on Average Worst Case Carry", *IEEE Transactions on Computers*, vol. C-22, no. 5, pp. 459-463, May 1973

[9] Brzozowski, J. A. and M. Yoeli, *Digital Networks*, Prentice-Hall, 1976 {25}

[10] Campeau, J. O., "Communication and Sequential Problems in the Parallel Processor", in *Parallel Processor Systems, Technologies and Applications*, edited by L. C. Hobbs, Spartan Books, 1970

[+] Numbers in braces indicate pages on which the sources are referenced

[11] Chen, I-N., "Performing Summation and Product in an Associative Processor", Proceedings of the 1977 International Conference on Parallel Processing, August 1977 {50, 82, 88, 95, 122}

[12] Chen, I-N. and R. Willoner, "An O(n) Parallel Multiplier with Bit-Sequential Input and Output", IEEE Transactions on Computers, to appear in vol. C-28, no. 10, October 1979 {23}

[13] Dadda, L., "Some Schemes for Parallel Multipliers", Alta Frequenza, vol. 19, pp. 349-356, March 1965 {26, 35}

[14] Diffie, W. and M. E. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, vol. IT-22, no. 6, pp. ~~644~~-654, November 1976 {107, 110}

[15] Feldman, J. D. and O. A. Reiman, "RADCAP: An Operational Parallel Processing Facility", Proceedings of the Sagamore Computer Conference on Parallel Processing, pp. 140-146, August 1973

[16] Feng, T. Y., "A Versatile Data Manipulator", Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing, p. 101, August 1973 {93}

[17] Feng, T. Y., "Data Manipulating Functions in Parallel Processors and Their Implementations", IEEE Transactions on Computers, vol. C-23, no. 3, pp. 309-318, March 1974 {93}

[18] Feng, T. Y. and C. Hsu, "Design and Evaluation of an Arithmetic Unit", Technical Report RADC-TR-73-86, March 1975

[19] Flynn, M. J., "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, vol. C-21, no. 9, pp. 948-960, September 1972 {1}

[20] Foster, C. C., Content Addressable Parallel Processors, Van Nostrand Reinhold, New York, 1976 {84}

[21] Gardner, M., "A New Kind of Cipher that Would Take Millions of Years to Break", Scientific American, vol. 236, no. 8, pp. 120-124, August 1978 {107}

[22] Garner, H. L., "A Survey of Some Recent Contributions to Computer Arithmetic", IEEE Transactions on Computers, vol. C-22, no. 6, pp. 552-555, June 1973

[23] Habibi, A. and P. J. Wintz, "Fast Multipliers", IEEE Transactions on Computers, vol. C-19, no. 2, pp. 153-157, February 1970

[24] Hayes, J. P., Computer Architecture and Organization, McGraw-Hill, New York, 1978

[25] Hellman, M. E., "The Mathematics of Public-Key Cryptography", Scientific American, vol. 241, no. 2, pp. 146-157, August 1979

[26] Hwang, K., Computer Arithmetic: Principles, Architecture, and Design, John Wiley & Sons, 1979 {10, 23, 68}

[27] Hwang, K., "Global and Modular Two's Complement Cellular Array Multipliers", IEEE Transactions on Computers, vol. C-28, no. 4, pp. 300-306, April 1979

[28] Karatsuba, A. and Y. Ofman, "Multiplication of Multiple-Digit Numbers with Computers", Doklady Akademii Nauk SSR, vol. 145, no. 2, pp. 293-294, February 1962 (in Russian) {26}

[29] Kline, C. S. and G. J. Popek, "Public Key vs. Conventional Key Encryption", AFIPS National Computer Conference Proceedings, New York, New York, vol. 48, pp. 831-837, June 4-7, 1979 {107}

[30] Knuth, D. E., The Art of Computer Programming: Seminumerical Algorithms, vol. 2, Addison-Wesley, 1971 {4, 30, 112}

[31] Kohavi, Z., Switching and Finite Automata Theory, McGraw-Hill, 1970 {2, 11, 20, 35}

[32] Lee, R., Optimal Parallel Computations for SIMD Computers, Ph.D. Thesis, Department of Computing Science, University of Alberta, Fall 1976 {50, 52}

[33] Martin, N. and S. Hufnagel, "Conditional-Sum Early Completion Adder Logic", IEEE Transactions on Computers, submitted

[34] Majithia, J. C. and R. Kitai, "An Iterative Array for Multiplication of Signed Binary Numbers", IEEE Transactions on Computers, vol. EC-13, no. 2, pp. 214-216, February 1971

[35] Merkle, R. C. and M. E. Hellman, "Hiding Information in Trapdoor Knapsacks", IEEE Transactions on Information Theory, vol. IT-24, no. 5, pp. 525-530, September 1978

[36] Michelman, E. H., "The Design and Operation of Public-Key Cryptosystems", AFIPS National Computer Conference Proceedings, New York, New York, vol. 48, pp. 305-311, June 4-7, 1979 {107, 112}

[37] Pezaris, S. D., "A 40-ns 17-Bit by 17-Bit Array Multiplier", IEEE Transactions on Computers, vol. EC-13, no. 4, pp. 442-447, April 1971

[38] Pohlig, S. C. and M. E. Hellman, "An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance", IEEE Transactions on Information Theory, vol. IT-24, no. 1, pp. 106-110, January 1978

[39] Ramamoorthy, C. V., and H. F. Li, "Pipeline Architecture", ACM Computing Surveys, vol. 9, no. 1, pp. 61-102, March 1977 {16, 120}

[40] Rivest, P. L., A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems", Communications of the ACM, vol. 21, no. 2, pp. 120-126, February 1978 {111, 112}

[41] Robertson, J. E., "A New Class of Digital Division Methods", IRE Transactions on Electronic Computers, pp. 218-222, September 1958

[42] Rudolph, J. A., L. C. Fulmer, and W. C. Meilander, "With Associative Memory, Speed Limit is no ", Barrier", Electronics, vol. 43, no. 14, pp. 96-101, June 22, 1970 {84}

[43] Rudolph, J. A., L. C. Fulmer, and W. C. Meilander, "The Coming of Age of the Associative Processor", Electronics, vol. 44, no. 4, pp. 91-96, February 15, 1971 {84}

[44] Schonhage, A. and V. Strassen, "Schnelle Multiplikation grosser Zahlen", Computing, vol. 7, pp. 281-292, 1971 {4}

[45] Spira, P. M., "Computation Times of Arithmetic and Boolean Functions in (d,r) Circuits", IEEE Transactions on Computers, vol. C-26, no. 10, pp. 948-957, October 1977

[46] Stenzel, W. J., W. J. Kubitz, and G. H. Garcia, "A Compact High-Speed Parallel Multiplicaton Scheme", *IEEE Transactions on Computers*, vol. C-26, no. 10, pp. 948-957, October 1977 {26}

[47] Swartzlander, E., "The Quasi-Serial Multiplier", *IEEE Transactions on Computers*, vol. C-22, no. 4, pp. 317-321, April 1973

[48] Taub, H. and D. L. Schilling, *Digital Integrated Electronics*, McGraw-Hill, 1977 {24}

[49] Thurber, K. J. and L. D. Wald, "Associative and Parallel Processors", *ACM Computing Surveys*, vol. 7, no. 4, pp. 215-255, December 1975 {1}

[50] TIME Magazine, "An Uncrackable Code?", p. 52, July 3, 1978 {107}

[51] Trivedi, K. S. and M. D. Ercegovac, "On-Line Algorithms for Division and Multiplication", *IEEE Transactions on Computers*, vol. C-26, no. 7, pp. 681-687, July 1977 {5, 31}

[52] Trivedi, K. S. and J. G. Rusnak, "Higher Radix On-Line Division", *Proceedings of the Fourth IEEE Symposium on Computer Arithmetic*, Santa Monica, California, pp. 164-174, October 25-27, 1978

[53] Wallace, C. S., "A Suggestion for a Fast Multiplier", *IEEE Transactions on Computers*, vol. EC-13, no. 2, pp. 14-17, February 1974 {26}

[54] Waser, S., "High-Speed Monolithic Multipliers for Real-Time Digital Signal Processing", *IEEE Computer*, vol. 11, no. 10, October 1978 {23, 24, 25, 26}

[55] Willoner, R., "A Highly Parallel Arithmetic Unit", *Proceedings of the 1979 ACM Computer Science Conference*, Dayton, Ohio, p. 21, February 20-22, 1979

[56] Willoner, R. and I-N. Chen, "A Parallel Arithmetic Unit", *Proceedings of the First European Conference on Parallel and Distributed Processing*, Toulouse, France, pp. 198-207, February 14-16, 1979 {68}

[57] Winograd, S., "On the Time Required to Perform Addition", *Journal of the Association for Computing Machinery*, vol. 12, no. 2, pp. 277-285, April 1965

[58] Winograd, S., "On the Time Required to Perform Multiplication", Journal of the Association for Computing Machinery, vol. 14, no. 4, pp. 793-802, October 1967 {26, 50, 121}

[59] Yau, S. S., and H. S. Fung, "Associative Processor Architecture——A Survey", ACM Computing Surveys, vol. 9, no. 1, pp. 3-27, March 1977 {84, 86}