AN ABSTRACT OF THE THESIS OF

Jaisimha K. Durgam for the degree of Master of Science in Electrical and Computer Engineering presented on July 26, 1988.

Title : Dynamically Configurable Systolic Arrays.

Abstract approved : Redacted for Privacy

Sayfe Kiaei

Digital signal and image processing and other real time applications involve simple but large amounts of computations. These problems have an enormous amount of inherent parallelism and demand high speed computation. Conventional computers do not possess these characteristics and this had led to the development of new architectural concepts. Among the new architectures, the systolic and wavefront arrays processors have gained a lot of attention because of their high processing bandwidth. The systolic arrays combine massive pipelining with parallelism. There is one problem with this approach of designing application specific chips wherein each of these chips is capable of processing a single algorithm. A number of real time applications require more than one algorithm to be executed for a complete solution. .

One solution to the above problem is to develop reconfigurable array structures. One notable proposal is the

Confiurable Highly Parallel Computer (CHiP) which is capable of reconfiguring the array to suit different interconnection schemes. In this thesis, the CHiP has been adapted to suit a family of systolic architectures. The dynamically Configurable Systolic Array proposed is designed to accommodate the linear configuration to solve convolution and polynomial multiplication, a square configuration to solve full matrix multiplication and a hexagonal array for band matrix multiplication.

The array is a 2-dimensional array arranged in a square grid and functions as an attached processor to a host. Each processor in the array is connected only to it's immediate neighbours and all external communication is only through the edge processors.

The actual interconnection patterns are implemented by a set of tristate drivers that are part of the communication links between neighbouring processors. The drivers are controlled by the controller and patterns are determined by the control signals generated. The arithmetic unit is a simple multiplier along with an adder capable of executing the inner product computation common to many signal and image processing applications.

The array has been built on a Genesil Silicon Compiler using 2-micron CMOS technology. The three array configurations have been successfully simulated and tested. The algorithms have been executed in times that closely

match the theoretical times. More importantly, the feasibility of building a single chip to implement a number of algorithms has been demonstrated and paves the way for further research in this area.

DYNAMICALLY CONFIGURABLE SYSTOLIC ARRAYS

by

Jaisimha.K.Durgam

A Thesis

submitted to

Oregon State University

in partial fulfillment of

the requirements for the

degree of

Master of Science

Completed July 26, 1988

Commencement June 1989

APPROVED :

Redacted for Privacy

Assistant Professor in charge of major

Redacted for Privacy

Head of the Department of

Electrical & Computer Engineering

Redacted for Privacy

Dean of the Graduate School

Date of presentation: July 26, 1988

Jaisimha.K.Durgam

TABLE OF CONTENTS

DYNAMICALLY RECONFIGURABLE SYSTOLIC ARRAYS

I. INTRODUCTION

Conventional computer architectures such as the Von-Neumann machine suffer from two problems, (i) the processing element is separated from the other parts by long communication paths such as busses and (ii) they are sequential machines wherein the system sequentially fetches instructions and executes them. As a result, there is a definite loss in the efficiency of the system. The sequential nature of these machines prevent the use of any inherent concurrence within the algorithm being executed.

Long communication paths substantially slow down the flow of information between the various components of the system and as a result, reduce the overall system performance [1]. In addition, long wires consume a great deal of space or area on a chip and thus require substantial power to drive them. Clearly, both these characteristics are undesirable for VLSI implementation. The presence of long communication paths and the inability of traditional architectures to utilize the inherent concurrence in any process make them unsuitable for a variety of high speed applications.

The demand for high speed processing capabilities for real time signal and image processing applications has led

to new design in computer architecture. In general, there are two approaches for achieving the desired bandwidth required by many real time applications. One approach is to design a DSP (Digital Signal Processing) microprocessor and program it to implement the specific algorithms [1,3]. Some examples are the Motorola DSP56000 and the TMS 320 series systems. Yet another approach is to design algorithmically specialized processors wherein each processor is capable of implementing a set of specific algorithms. The algorithmically specialized processors are designed to exploit locality in the algorithm and also use pipelining to achieve high processor utilization. Care is taken to avoid long communication paths and make use of the inherent parallelism.

Among the algorithmically specialized processors, systolic arrays [2] and wavefront array processors [2] have gained a lot of importance. Other specialized processors include the tree processors, butterfly networks etc. A systolic system combines massively parallel computations with a high degree of data pipelining to attain a high throughput. A systolic system typically consists of a set of interconnected cells, each of which is capable of performing some simple operation. Information in a systolic array flows in a pipelined fashion, and communication with the outside world occurs only at the "boundary cells". Each cell in the array communicates only with it's nearest neighbours thus

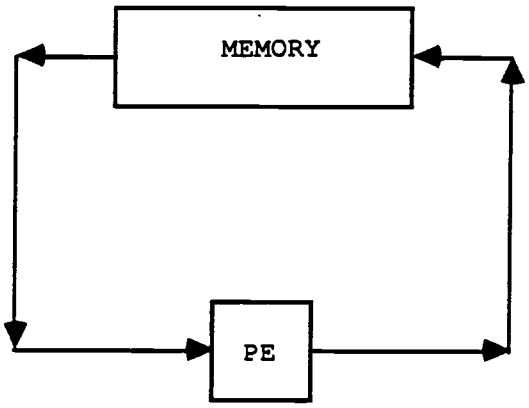
eliminating the need for long communication paths. Moreover, systolic arrays lend themselves to easy VLSI implementation due to their highly modular structure and regular interconnection patterns [2].

The basic principle of a systolic architecture is shown in Figure 1 [from 2]. By replacing the single processing element with an array of processing elements or cells, a higher computation bandwidth is achieved without increasing the memory bandwidth. As seen from Figure 1, the essence of the systolic approach is that once a data item is brought into the array, it is used effectively in each cell it passes. Thus massive pipelining is possible and the processor utilization is greatly enhanced.

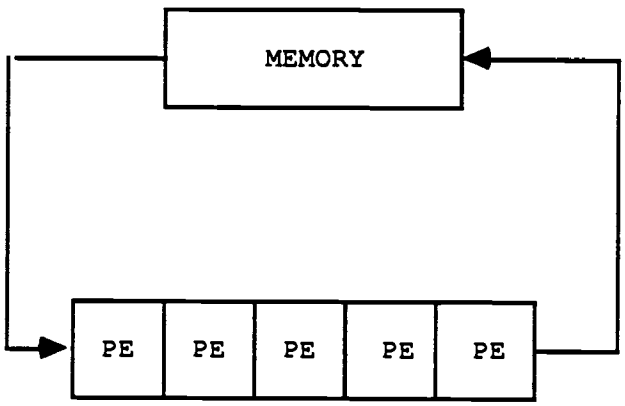
The systolic approach is suitable for a wide class of compute bound computations where multiple operations are performed on each item in a repetitive manner. Most signal and image processing problems (convolution, LU decomposition, etc.)lend themselves to this kind of approach and as a result, systolic architectures have proved to be a very strong proposition to provide solutions for such problems.

A variety of systolic configurations have been proposed for computationally intensive problems [2]. Some examples are the linear arrays to solve problems like convolution, square arrays for full matrix multiplication and hexagonal arrays for band matrix multiplication and LU decomposition.

INSTEAD OF :



WE HAVE :



THE SYSTOLIC ARRAY.

FIG 1. BASIC PRINCIPLE OF A SYSTOLIC SYSTEM.

The main drawback with designing algorithmically specific architectures, and in particular, systolic arrays, is that each array implements only one specific algorithm. Furthermore, the majority of the computationally intensive applications require more than one algorithm to be processed. In such cases the above approach would be unfeasible and make the solution uneconomical.

An obvious alternative solution is to have a reconfigurable system wherein the processor array is configured to suit the algorithm. Many ideas have been proposed for these reconfigurable systems. One notable system is the CHiP (Configurable Highly Parallel Computer) [3]. Other ideas include the WARP computer [6] and the Programmable Systolic Chip (PSC) [7].

The CHiP computer is a reconfigurable array structure. It essentially consists of three components : a set of homogeneous processors, a switch lattice and a controller. The switch lattice is the most important component of the CHiP as far as the reconfiguration scheme is concerned. The settings of the switches in the lattice are predefined and based on the control inputs, any available setting can be achieved. The settings are contained in local memories within the switches. A configuration setting enables the switch to establish a direct, static connection between two or more of its incident data paths.

A global controller is responsible for loading the switch memory. CHiP processing begins with the controller broadcasting a command to all switches to invoke a particular configuration setting. Once the desired configuration is achieved, computations begin. More details of the CHiP computer can be found in [3].

The WARP project consists of a array of processors interconnected to form a one dimensional linear array. Algorithms that require two dimensional and other array structures are mapped on to this array. This reduces the speed of execution of the algorithm but also reduces the number of input and output paths required to communicate with the external world.

In this thesis, the CHiP computer has been adapted to suit systolic systems. Systolic architectures have a limited number of configurations to implement a wide range of applications. It has extensive hardware to support different arithmetic and logic functions. Moreover, the CHiP is capable of implementing different configurations to suit a wide range of applications including some systolic array applications. As a result, the CHiP is a more general purpose processor than specific systolic architectures.

An attempt is made here to design a reconfigurable systolic system using the ideas of interconnection patterns and reconfiguration control from the CHiP. The array is designed to work as an attached processor to a host computer

and the various interconnection schemes are controlled by a microprogrammed control unit. The array has been designed specifically to provide three configurations namely

(i). Linear array : to solve problems like convolution and polynomial multiplication,

(ii). Square array : for Full matrix multiplication, etc. and

(iii). Hexagonal array : for Band matrix multiplication, LU decomposition, etc.

The above three configurations are typical to a number of systolic algorithms and have therefore been chosen. The proposed array differs from the CHiP in implementation details and is an attempt to designing reconfigurable systolic systems to enhance the usage of algorithmically specific computers.

2. ARCHITECTURE OVERVIEW

Any architecture that is capable of implementing different interconnection patterns must have an overall structure that incorporates these patterns. Different patterns may be hardwired within the overall structure of the architecture. The architecture must then be able to configure itself into any of the hardwired patterns depending on the requirement of the algorithm it executes. Thus it is very important that the layout of the array must be such that it is capable of implementing these different interconnection patterns.

With these considerations taken into account, the proposed dynamically reconfigurable systolic array is laid out as a 2-dimensional structure [Fig 2]. In general, the number of processors can be "k^2" where "k" is a power of 2. The "k^2" processors are arranged in a square grid with "k" processors along each edge of the square. The array is capable of configuring itself into either a linear array, a square array or a hexagonal array.

Each processor in the array communicates only with it's immediate neighbours thus reducing any long communication paths. Input and output functions are performed only by the processors on the boundaries of the array [Figure 2]. Within the array itself, there is scope for bi-directional data transfer in the horizontal direction while unidirectional

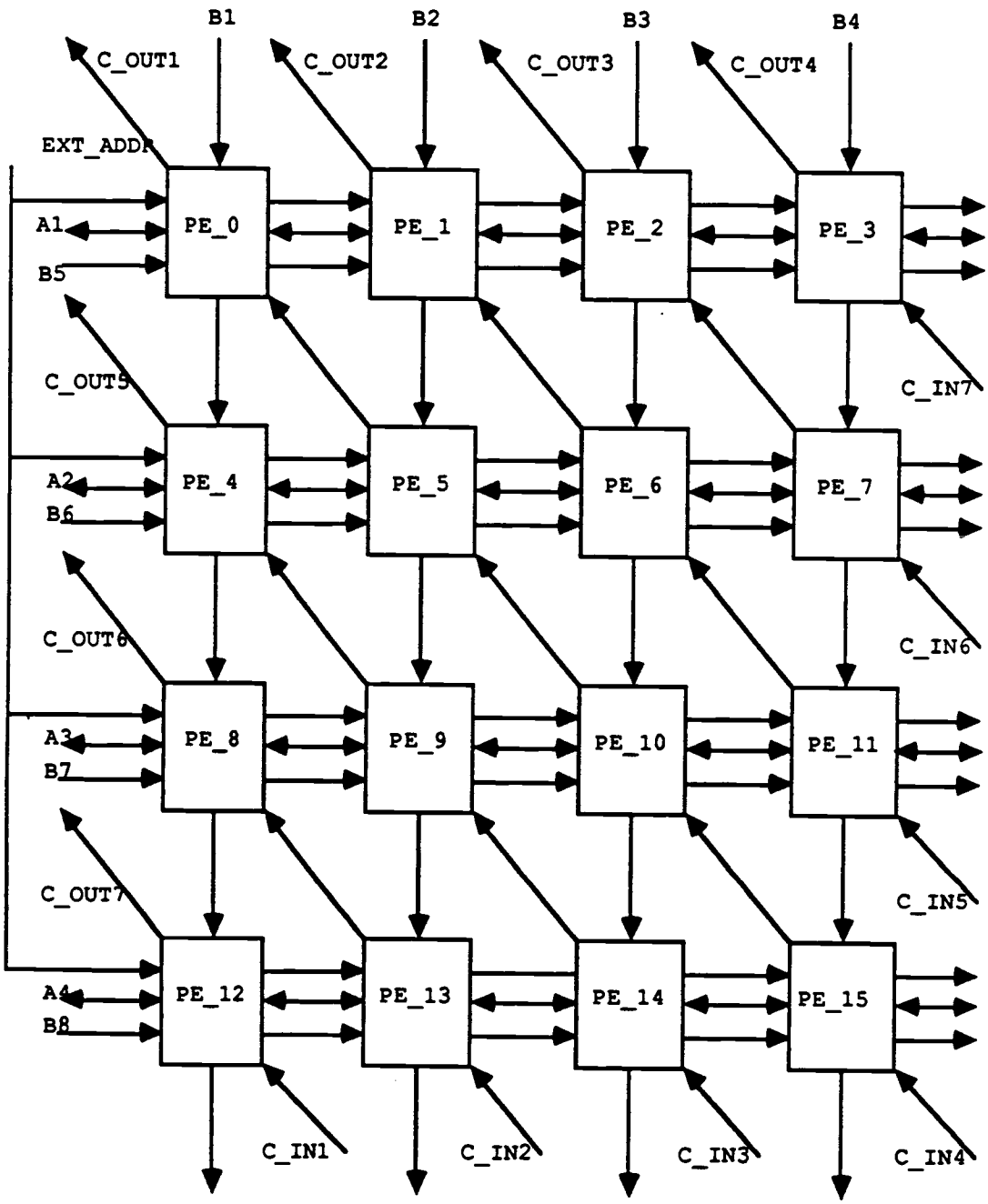


FIG 2. LAYOUT OT THE DYNAMICALLY CONFIGURABLE SYSTOLIC ARRAY

data flow occurs along the vertical and diagonal paths. The array is designed to work as an attached processor to a host computer and any action in the array is initiated by this external host. Each processor within the array is constructed from three components [Figure 3] :

(i). The Inner Product Element (IPE) which forms the computational or arithmetic unit of the processor,

(ii). The Interconnect Switch (IS), a set of drivers that form the interconnects between the processors in the array and

(iii). A Microprogrammed Control Unit (MCU).

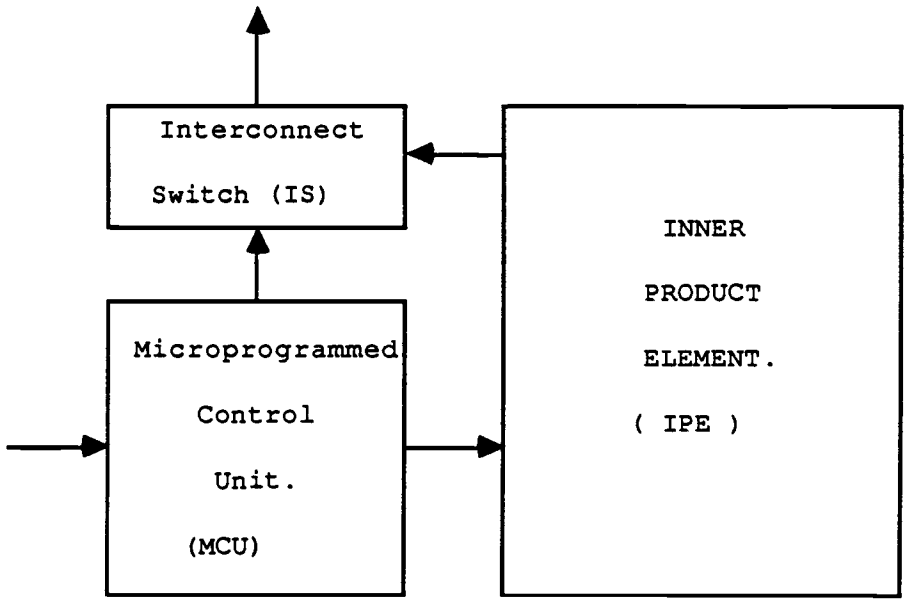


FIG 3. BLOCK DIAGRAM OF A TYPICAL PROCESSOR.

2.1 The Inner Product Element :

A number of systolic algorithms involve very simple computations. In fact, quite a few of them involve the inner product computation given by

C_Out = C_In + (A_In * B_In).

The variables A_In and B_In represent the inputs to the multiplier and enter the processing element either from it's neighbours or from an external source depending on the algorithm. Figure 4 shows a typical IPE of a processor in the array. The 'A' input can be from the neighbouring east/west processing element and the 'B' input is from the east/north neighbour. C_In represents the intermediate result from any of the neighbouring IPE's or the accumulated value generated within the processor. The edge processors receive their data from the external host.

2.2 The Interconnect Switch (IS) :

Each processor in the array has a set of tristate drivers that enable the implementation of various interconnection schemes [Fig 4]. The Interconnect Switch is the pathway for all data transfers between processors. In conjunction with the definition of a systolic array, each processor in the array communicates only with it's immediate neighbours [Fig 2]. Depending on the algorithm being executed, the appropriate drivers of the IS are enabled thus ensuring correct data flow. Once the IS is set for a particular configuration, it acts only as a propagation

channel and as a result, no extra delays are introduced in the system by the IS. The IS is controlled by signals generated from the local controller and the configuration is determined by the controller.

2.3 The Microprogrammed Control Unit (MCU) :

The heart of each processor in the array is the MCU. Each processor has a local microprogrammed controller. This MCU locally controls the switch settings of the IS and provides the necessary control signals for the IPE. Figure 5 shows the architecture of the MCU. The MCU essentially consists of a ROM (Read Only Memory) which is divided into segments of four words each. Each such segment contains the control words required by the particular algorithm. The Control Address Register (CAR) acts as a pointer to the ROM locations. Based on the function being executed by the processor, the contents of the address register vary. The contents of the CAR are fetched from another register, the Address Merge Register (AMR). This register merges the external control inputs with the next state address generated by the control ROM and outputs a five bit number which is the address of a location in the Control_Rom.

The modular nature of the MCU makes it flexible to any changes by simply changing the contents of the Control_Rom or by adding extra memory space to the Control_Rom and sufficient addressing hardware. It was mentioned that the Control_Rom is divided into segments. Each of these segments

is responsible for implementing a particular algorithm. Moreover, each of these segments is capable of being addressed by the host computer. In order to enable the execution of new applications and configurations, more segments can be added to the Control_Rom. Thus by simple adding additional memory to the Control_Rom of the MCU, the number of applications of the array can be increased thereby making it very flexible for enhancement.

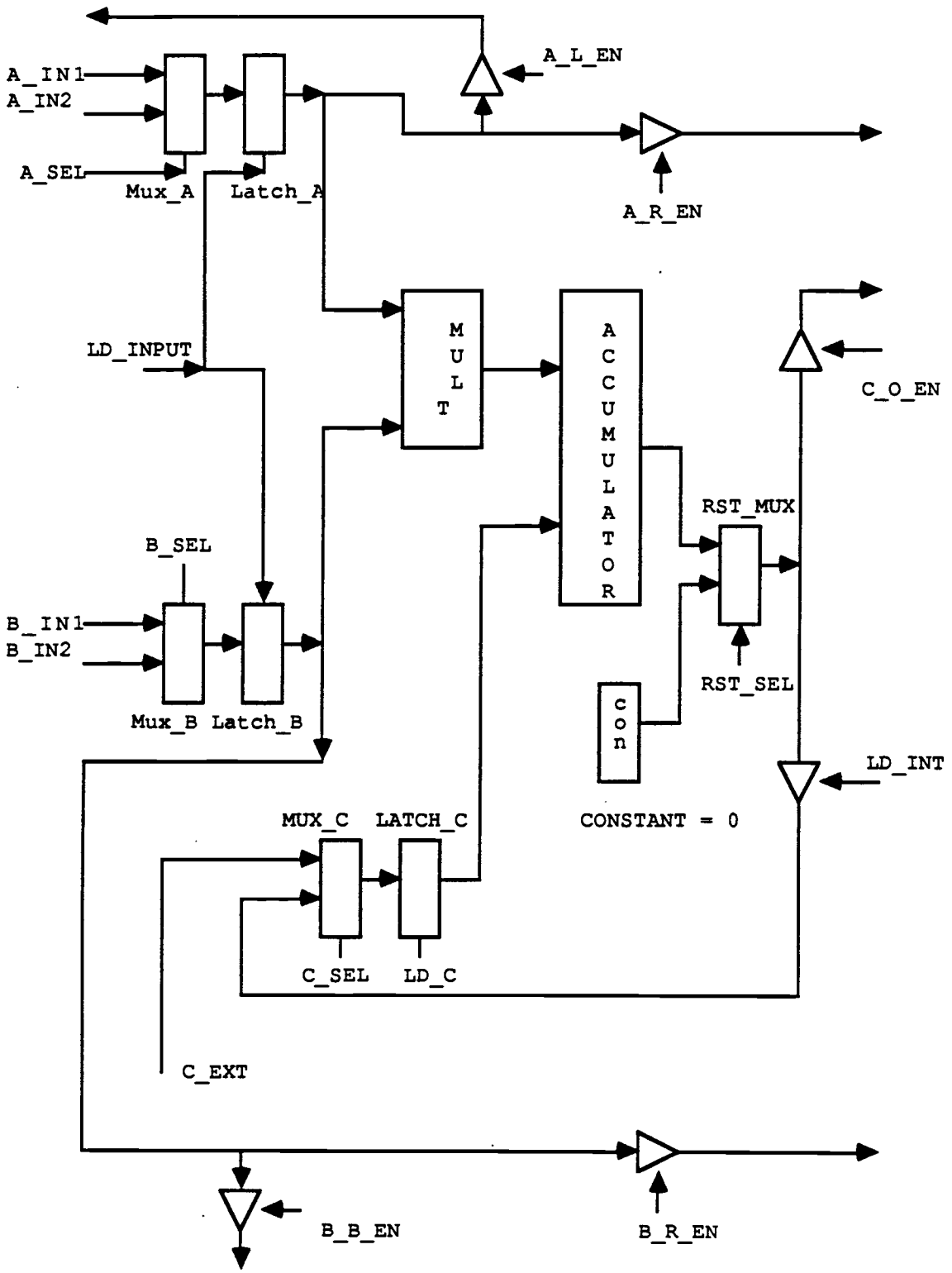


FIG 4. BLOCK DIAGRAM OF THE IPE AND INTERCONNECT SWITCH.

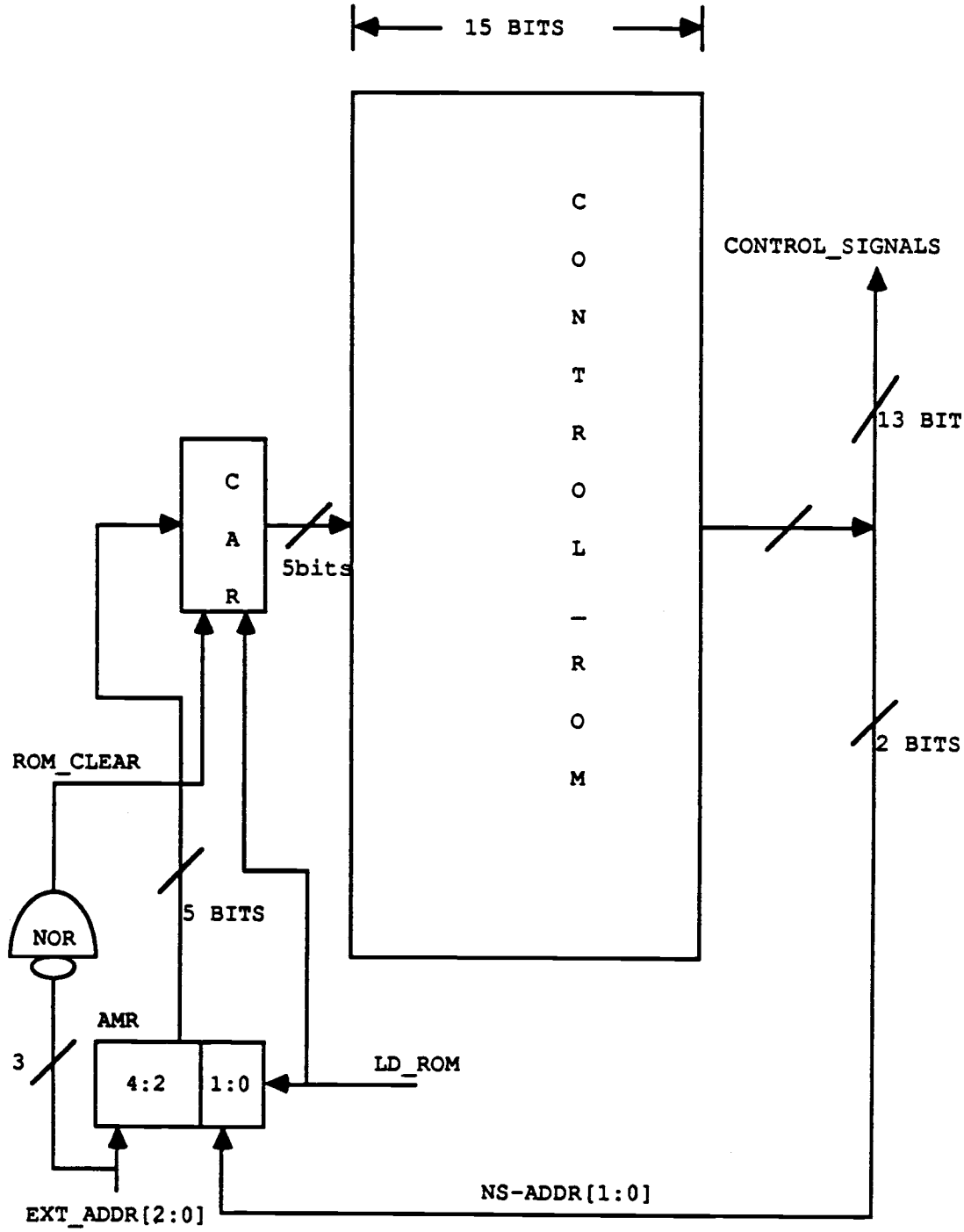


FIG 5. MICROPROGRAMMED CONTROL UNIT (MCU) DETAILS

3. MICRO ARCHITECTURE OF THE RECONFIGURABLE ARRAY

In this chapter, the details of the architecture of each processing element in the array are discussed. The entire array has been constructed using building blocks from the library of the Genesil Silicon Compiler. The microarchitecture of the three components of each processing element namely the IPE, the IS and the MCU are explained.

3.1 The Components Of The IPE :

Each IPE can be subdivided into the following blocks:

3.1.1 The Input Selectors :

The input selectors in each PE determine which inputs are fed to it and also determine the source of these inputs.

The Input selectors consist of a MULTIPLEXER and a LATCH. The multiplexer has two inputs and selects one of these and feeds it to the Latch. There are two such selector switches in the input path of the multiplier. Mux_A and Latch_A are associated with the A input of the multiplier and Mux_B and Latch_B are for the B input. [See Figure 4].

The input to the Mux_A are A_IN1 and A_IN2 respectively. A_IN1 is the input from the neighbouring processor to the East and A_IN2 is the input to the processor from the neighbour in the West. For convolution,the algorithm requires that the A input come from the West and therefore A_IN2 is selected and for the other two algorithms, A_IN1 is selected.

Similarly, Mux_B has two inputs B_IN1 and B_IN2. B_IN1 comes from the neighbour in the North and B_IN2 from the neighbour in the East. For the convolution, B_IN1 is selected and for the other two algorithms, B_IN2 is selected.

The various signals associated with the input selectors are as follows :

Input Selector A :

Inputs : A_IN1 and A_IN2.

Select signal for multiplexer Mux_A : A_SEL.

Output : A_OUT.

Input Selector B :

Inputs : B_IN1 and B_IN2.

Select signal for multiplexer Mux_B : B_SEL.

Output : B_OUT.

The outputs of the two multiplexers are fed into the latches LATCH_A and LATCH_B before they are input to the multiplier. These latches are gated to control the input to the multiplier. Since at all times the two inputs to the multiplier need to be applied at the same time, a single control signal LD_INPUT is used to gate these two latches. The outputs of these latches called OP_A and OP_B respectively are the inputs to the multiplier and the interconnect switches. The interconnect switches will be discussed later.

3.1.2.The Computational Element :

The computational element consists of a Multiplier and an Accumulator [Figure 4]. The multiplier is a 4-bit parallel multiplier. The multiplier output is a 8-bit output and is available as two four bit outputs, the MULT_MSB[3:0] which is the most significant 4 bits of the product and the LS_OUT[3:0] which form the least significant 4 bits of the product. The multiplier therefore implements the computation A*B in the inner step computation.

The accumulator is a 8-bit parallel datapath and implements the summation part of the inner step product[Figure 4]. It consists of an ADDER and some other support hardware such as latches and multiplexers, details of which are given below. The two inputs to the adder are (i) the product from the multiplier and (ii) another input C which can be either the partial result of the inner step computation within the processor or from the neighbouring processor in the south-east.

Accumulator Hardware :

Multiplexer C (Mux_C) :

Inputs : C_IN1 (partial result from within the processor) and C_EXT (partial result from the processor in the south-east).

Select signal for the multiplexer : C_SEL.

Latch_C : used to latch the output of the multiplexer. Control signal for this gated latch is LD_C.

Adder

:Inputs : (i). Output of the multiplier and (ii) the output of Latch_C.

Output :

An 8 bit sum called RESULT.

The partial result RESULT is fed into the Mux_C by means of a tristate driver which requires a control signal LD_INT. The output of the adder also drives an output port so that the value of C can be sent to the processor in the SW direction or to the external world in case of an edge processor [Figure 4].

3.2 IS Details

The interconnect switches are the main components of the reconfiguration scheme[See figures in appendix]. As already mentioned, they are a set of bus drivers. Each input A and B, after coming out of the latches of the Input Selectors, drive two such drivers each. The output of the latches in the Input selectors namely OP_A and OP_B are inputs to the drivers in the interconnect switch. Two drivers, the A_RIGHT and the A_LEFT are associated with the A operand and two drivers, B_RIGHT and B_BOTTOM comprise the interconnect switch of the B operand.

The A_RIGHT output is connected to the A_IN1 input of the processor on the right and the A_LEFT output is connected to the A_IN2 input of the neighbouring processor

on the Left. Depending on the algorithm, the appropriate drivers are enabled and inputs selected.

For the convolution algorithm, the A_LEFT is enabled and for the other two algorithms, the A_RIGHT driver is enabled [See Appendix for algorithm details]. Each of these drivers has a enable signal [See figure in Appendix].

Similarly, the B_RIGHT is connected to the B_IN1 input of the processor on the RIGHT and the B_BOTTOM is connected to the B_IN2 input of the processor in the South direction. Again the convolution algorithm requires that the B_RIGHT be enabled and the other two require that B_BOTTOM be enabled.

The structure of the IS and is thus very simple and provides an effective way of controlling the data flow between processing elements in the array.

3.3 MCU Internals

The MCU is implemented using a ROM and some other access circuitry. There is an input latch that latches the address to the ROM [Fig 5].

The Control ROM's structure is as follows : It has 5 address bits which means that it can address upto 32 locations. Even though the requirements are not 32 words, limitations of the Compiler force this usage. The ROM has an internal address latch to which we feed the contents of the external latch that we mentioned before. This internal latch can be cleared using a CLEAR signal. There is also a load

signal to these two latches and they have been assigned the same signal namely, LD_ROM.

3.3.1 ROM Organization

The ROM has 14 fields. The number of bits in each word of the ROM are 15. There are 2 bits in the word that define the least significant 2 bits of the Next state Address. The other three bits of the address are from the external address in the case of an edge processor or from the left processor otherwise.

Then there are 3 control signals for the Input Selectors. 4 signals for the Interconnect Switch, 5 control signals (including one signal RST_SEL for initially clearing the contents of the IPE) for the IPE and finally, the LD_ROM signal for the latching of the ROM address latches.

The 32 ROM locations have been divided into blocks of 4 words each [See ROM coding file]. The various blocks are as follows (i) Locations 0-3 : These are for initialization. Initialization is always done before implementing any algorithm and clears all internal registers and also provides the LD_ROM signal. This enables the processor to get the 3 MSB's of the NS address from the external world. (ii) Locations 4-7 : These are for output. They are called upon to output the results of the computations at the end of the computations.(iii) Locations 8-11 : These implement the Convolution algorithm (Linear array with bi-directional data flow). They provide suitable control signals to all the

components of the PE. (iv) Locations 12-15 : They implement the full matrix multiplication algorithm (Square array). (v) Locations 16-20 : They are for the band matrix multiplication (Hexagonal Configuration).

Since the most significant 3 bits of the ROM address come from external source, it is easy to see that we can jump to any appropriate block depending on the algorithm. Thus this provides for easy implementation of all algorithms. The modular nature of the MCU makes it flexible for adding additional blocks of micro instructions for other configurations. Right now, the MCU has been programmed for three algorithms (i) Convolution, (ii) Full matrix multiplication and (iii) Band matrix multiplication.

The system has currently been designed to implement the above three algorithms but it is easy to see that it can be expanded for other applications by increasing the ROM space and adding extra blocks of control instructions in this space.

4. SYSTEM OPERATION AND ANALYSIS OF PERFORMANCE

The array processor is designed to work as an attached processor to a host. As a result, it requires external control inputs to initiate any action. This control input is in the form of a 3-bit address EXT_ADDR[2:0] that is broadcast to all the edge processors on the east, PE_0, PE_4, PE_8 and PE_12 [Fig 2]. This 3-bit address forms the most significant three bits of the CONTROL_ROM address. This EXT_ADDR input is also connected to the CLEAR input of the CONTROL_ROM Address Register(CAR). This is done for initializing the array prior to any algorithm execution.

4.1 Initialization.

Initialization is done in the following steps :

Step 1 : The value of the 3-bit EXT_ADDR is set to zero. This enables the CLEAR signal. As a result, the CAR (CONTROL_ROM Address Register) contents are cleared and point to the zeroeth location of the CONTROL_ROM [Figure 6, Step 1]. The control word here contains the control signals required to initialize the system including the LD_ROM signal which allows loading both the CAR and the Address Merge Register (AMR) [Fig 5].

Step 2 : The three most significant bits of the address of the CONTROL_ROM from the above processors are propagated to the neighbouring processor to the right [Figure6, Step 2]. These address bits are taken from the AMR output. It is

therefore required that the AMRs of these processors contain the correct address to be propagated. However, in order to load the AMRs, the LD_ROM signals are required [Fig 5]. But these signals are generated by the controller itself during the initialization process. Therefore, the EXT_ADDR input must retain it's value (zero) for atleast two time cycles so that the AMRs of PE_0, PE_4, PE_8 and PE_12 are loaded with this value of EXT_ADDR. At this point, the address is ready to be propagated to the neighbouring processor (PE_1, PE_5, PE_9 and PE_13) and the PE's 0, 4, 8 and 12 are initialized.

Step 3 : PE's 1, 5, 9 and 13 also behave in the same fashion as the PE's 0, 4, 8 and 12 and as a result, require that the external control inputs to them retain their value for two time cycles. This in turn implies that the EXT_ADDR input to the array retain it's value for this extended amount of time. As the AMR's of these PE's are loaded, the CAR's of PE's 2, 6, 10 and 14 are cleared [Figure 6, Step 3].

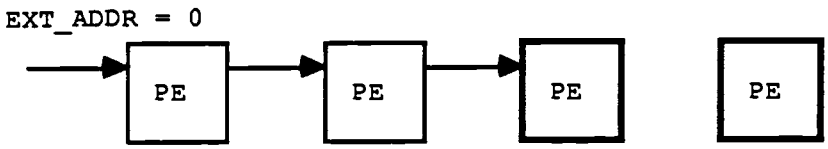
Step 4 : In this step, the AMR's of PE's 2, 6, 10 and 14 are loaded with the value of EXT_ADDR and the CAR's of PE's 3, 7, 11 and 15 are cleared.

Step 5 : This is the final step. Here, the AMR's of PE's 3, 7, 11 and 15 are loaded with EXT_ADDR and the array initialization process is complete. Thus it is clear that the EXT_ADDR input to the array should retain it's value of

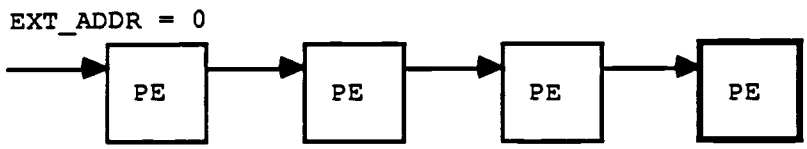
zero to allow the entire array to be initialized. Exact times will be calculated later.

Once initialization is complete, the array processor is ready to execute any of the three algorithms that it is capable of executing. The execution of these algorithms is again initiated by the external host.

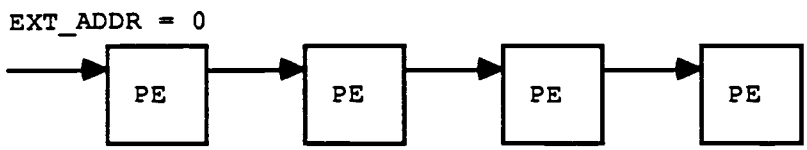
Step 1. CAR's of PE's 0, 4, 8 and 12 cleared.



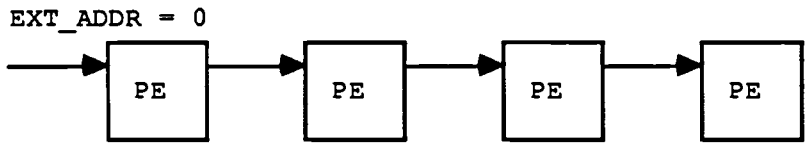
Step 2. AMR's of above PE's loaded with EXT_ADDR. CAR's of PE's 1, 5, 9 and 13 cleared.



Step 3. AMR's of PE's 1, 5, 9 and 13 also loaded with EXT_ADDR. CAR's of PE's 2, 6, 10 and 14 are now cleared.



Step 4. AMR's of PE's 2, 6, 10 and 14 are loaded with EXT_ADDR. Also the CAR's of PE's 3, 7, 11 and 15 are cleared.



Step 5. AMR's of PE's 3, 7, 11 and 15 are loaded with EXT_ADDR. Array initialization is complete.

FIG 6. SNAPSHOTS OF INITIALIZATION PROCESS.

4.2 Algorithm Execution.

Algorithm execution begins with the external host generating the correct value of EXT_ADDR corresponding to the algorithm. Before computations can begin on actual data values, the entire array has to be reconfigured to suit the algorithm. This means that all the processors in the array must reach a particular controller state that defines the reconfiguration scheme. This is achieved as follows :

Step 1 : The EXT_ADDR value for the particular algorithm is broadcast to the four edge processors PE_0, PE_4, PE_8 and PE_12 and then propagated along each row of the array. This is similar to the initialization process. The number of time cycles required for the entire array to configure itself for any application is fewer than the system initialization. This can be attributed to the fact that the MCU generates the LD_ROM signal during every system state. As a result, both the AMR and CAR are loaded from the instant the EXT_ADDR value is available unlike the system initialization process where at the start, the control signal for the Rom address registers is not enabled and hence extra time cycles are needed to allow the various address registers to be loaded.

Step 2 : Before the actual computations begin, all the datapaths and registers have to be initialized. To allow this clearing of the datapaths, each block of four control

words has been further subdivided into two sub-groups of two words each [See microcode for controller in Appendix]. The first subgroup comprises of the control words whose least significant two bits of the address (NS_ADDR) have value zero and one. The second sub-group consists of words with NS_ADDR values equal to two and three.

Each time the system requires a new algorithm to be executed after system initialization, the MCU is designed so that control is passed on to the second sub-group of control words in a block. Computations are performed on zero values and these values have to be fed to the array from the host along with the EXT_ADDR inputs at appropriate intervals of time. Details of the timing will be discussed later. One important fact to be noted here is that unlike system initialization, the entire array need not be cleared before the computation can begins. Once a processor is cleared, the computation begin. This is achieved by passing control to the first subgroup after the processor has gone through the second subgroup [See Appendix for subgroups in ROM code file].

There are two control words in the first sub-group of each block. These are associated with the actual algorithm execution and generate all the necessary control signals. The first control word in each of the first sub-groups generates the signals required for computations within the processor. The second control word is for the Interconnect

switch. Here, suitable control signals required for achieving correct data flow as mandated by the algorithm are enabled.

Step 3 : After the MCU of each processor enters the first sub-group of control words in any block, it switches states within this subgroup. These computations continue until they are interrupted by a different value of EXT_ADDR. It should be mentioned that the EXT_ADDR input has to retain it's value during the entire execution period of the algorithm.

Step 4 : Output functions are performed by giving a setting the EXT_ADDR to one. In this block, the output driver of each processor is enabled by enabling the signal C_O_EN. The results of computations are then sent to the external host and the algorithm execution is complete.

5. SYSTEM TIMING AND PERFORMANCE EVALUATION

The array has been implemented on a Genesil Silicon Compiler using 2 - micron CMOS technology. The timing diagrams of the various components in each processing element [Figures in appendix] can be used to analyze the processor timing as well as the system timing of the entire array. In this chapter, a detailed analysis of the system timing for the three algorithms is done. Theoretical timing analysis is done first and then timing calculations based on actual component delays from Figures[] are done. Comparisons are made between theoretical execution times and the actual timing in the array.

5.1 Theoretical Timing Analysis.

5.1.1 Full Matrix Multiplication - A Square Array Application:

As already stated, the entire array needs to be initialized in order to initiate self controlled actions. As is evident from Figure 6, a minimum of five time cycles are required for this purpose.

In general, the theoretical time for initialization for an N*N array can be formulated as follows :
1. The total time required for the entire array is the same as that for a single row in the array. This is owing to the fact that the different rows are initialized simultaneously.

2. The time required to initialize the first PE in any row is 2 units [Figure6, Steps1&2].

3. Once this PE is initialized, for every time unit, one new PE is initialized [See Figure 6]. It is necessary to note that the last PE in any row does not need to have it's AMR loaded with the value of EXT_ADDR during this process. This is because it does not have to propagate this value of EXT_ADDR to any other processor.

The time required can therefore be calculated as :

Time for the first PE = 2 units,

Time required for the other (N - 1) processors is = (N - 1).

Therefore, the total number of time units needed for initialization (IT) is

IT = (N - 1) + 2 = (N + 1).

Once the array is initialized, the EXT_ADDR is set to value 3. This is the value of the three most significant bits of the Control_Rom address. It was also mentioned that along with the address, the various data with zero value also need to be input to clear the registers and datapath in the array [Refer Chapter 3].

The computation time for Full Matrix Multiplication consists of

(i). the time required for clearing all datapaths (Reset Time) and

(ii). the time required to compute the resultant matrix (Computation Time).

Reset Time : The array does not function unless all the processors reach a predefined controller state within the block of control words associated with the algorithm. This state can be one of the first two states in the block [See Appendix for Rom code file]. It may be recalled that these states are responsible for generating the control words needed for algorithm execution. After initialization, control is passed on to the third state in the control block. This state along with the next is responsible for resetting the processor prior to algorithm execution.

In order that the entire array reaches one of the predefined states, certain time allowance has to be made. The manner in which the processors reach these states is similar to the initialization process and as a result the reset time is identical and for an array of size N*N, the reset time (RT) is given by

RT = N + 1.

Computation Time : Computation time refers to the number of time units needed to finish execution of the algorithm on the actual data. Computation begins after the previous two steps namely the initialization and reset operations are complete. The data is input as shown in the snapshots in Figure 7.

The time required by an N*N array to execute the above algorithm is now calculated. In order to execute a matrix

multiplication involving two N*N matrices on this array, the time required can be calculated as follows :

Step 1 : PE_N1 and PE_1N receive their first valid data inputs (N-1) time cycles after execution begins [See Figure 7]. From then on, they receive new data during every cycle.

Step 2 : Using the same argument as in the previous case, PE_NN receives it's first valid data inputs (N-1) cycles after PE_N1 and PE_1N.

In order that this PE receives all the inputs including the last inputs (A_{nn} and B_{nn}), the time required is N units.

Therefore the total time required for the PE_NN to receive all the inputs is the sum of the times required by each of the three steps and is given by

(N-1) + (N-1) + N = (3N -2) time units.

Assuming that actual arithmetic computations on each of these input data require one time unit, the computation time is equal to the time required to input all data values and is given by

Computation Time_Theoretical (CT_theoretical)

= (3N - 2) units.

For the case when N = 4, the theoretical computation time = (3*4 - 2) = 10 time units.

Total Time for the entire algorithm, TT_fullmat is

TT_fullmat = IT + RT + CT = (N +1) + (N + 1) + (3N - 2) = 5N units.

5.1.2 Systolic Convolution : A Bi-directional Linear Array Application :

The convolution algorithm is the second algorithm that has been implemented on the array. Here, only one row of processing elements is used and it proposed that the other rows could be used to implement the same algorithm on different data. The entire array can be used to implement a single case of the algorithm but that would mean wrapping around the data externally. For now, only one row of the array has been used.

The solution to the convolution problem as given by Kung (See Appendix for Algorithm) requires that the two data streams flow in opposite directions through the array. The results are generated as the data passes through the linear array.

The procedure for executing the convolution algorithm is similar to the full matrix multiplication algorithm. The linear array has to be initialized in the same manner as in the case of the full matrix multiplication. The various times required for completion of execution are

1. Initialization Time (IT) : The time required to initialized the entire array is the same as that required to initialize a single row, the initialization time IT is same as that for Full Matrix Multiplication. Therefore

IT_theoretical = (N + 1) units.

2. Reset and Computation Times : For the algorithm, the A inputs are from the left i.e. from PE_3 and the B inputs are from the right i.e. from PE_0. Also, the computation does not begin until the first A input namely A3 reaches PE_0 [See snapshots in Figure 8].

In general, the time required for a linear array of N processors is calculated as follows :

Step 1 : Time needed for the first A input to reach PE_0 is N [See Figure 8].

Step 2 : When the A input reaches this PE, the first B input is also input and from then on, a new B input is read in once every two cycles. Since computation occurs simultaneously, the time required for execution is same as the time required to input all data values.

Step 3 : The number of B inputs is N and the number of time units required to input the last B value after the first B input is read is (2N - 1) [Figure 8].

Step 4 : Then another (N - 1) time units are required to propagate the last input across the array.

Therefore, Total Time required is

Computation Time_theoretical = N + (2N - 1) + (N - 1)

= (4N - 2) units.

5.1.3 Band Matrix Multiplication : A Hexagonal Array Application :

The procedure for executing the band matrix multiplication algorithm on a hexagonal array is similar to

the full matrix multiplication on a square array. Before execution can begin, the array needs to be initialized and local control by the processors themselves has to be activated.

1. Initialization Time (IT) : Since the above requirement is exactly similar to the previous two algorithms, the initialization time IT_theoretical is given by

IT_theoretical = (N - 1) + 2 where N is the number of processors in a row of the array.

2. Computation Time : The computation time for the band matrix multiplication is calculated as follows :

In general, for any multiplication involving two N*N band matrices, the time required for computation is the time required by the last element in the main diagonal to be read into the array plus any additional evaluation time. Since there are N elements in the main diagonal in any N*N band matrix [See Figure 9], the theoretical computation time for the band matrix multiplication is

Computation Time_theoretical = (3 * N) time units.

Total Time = Initialization Time + Computation Time

= (N + 1) + 3N = (4N - 1) time units.

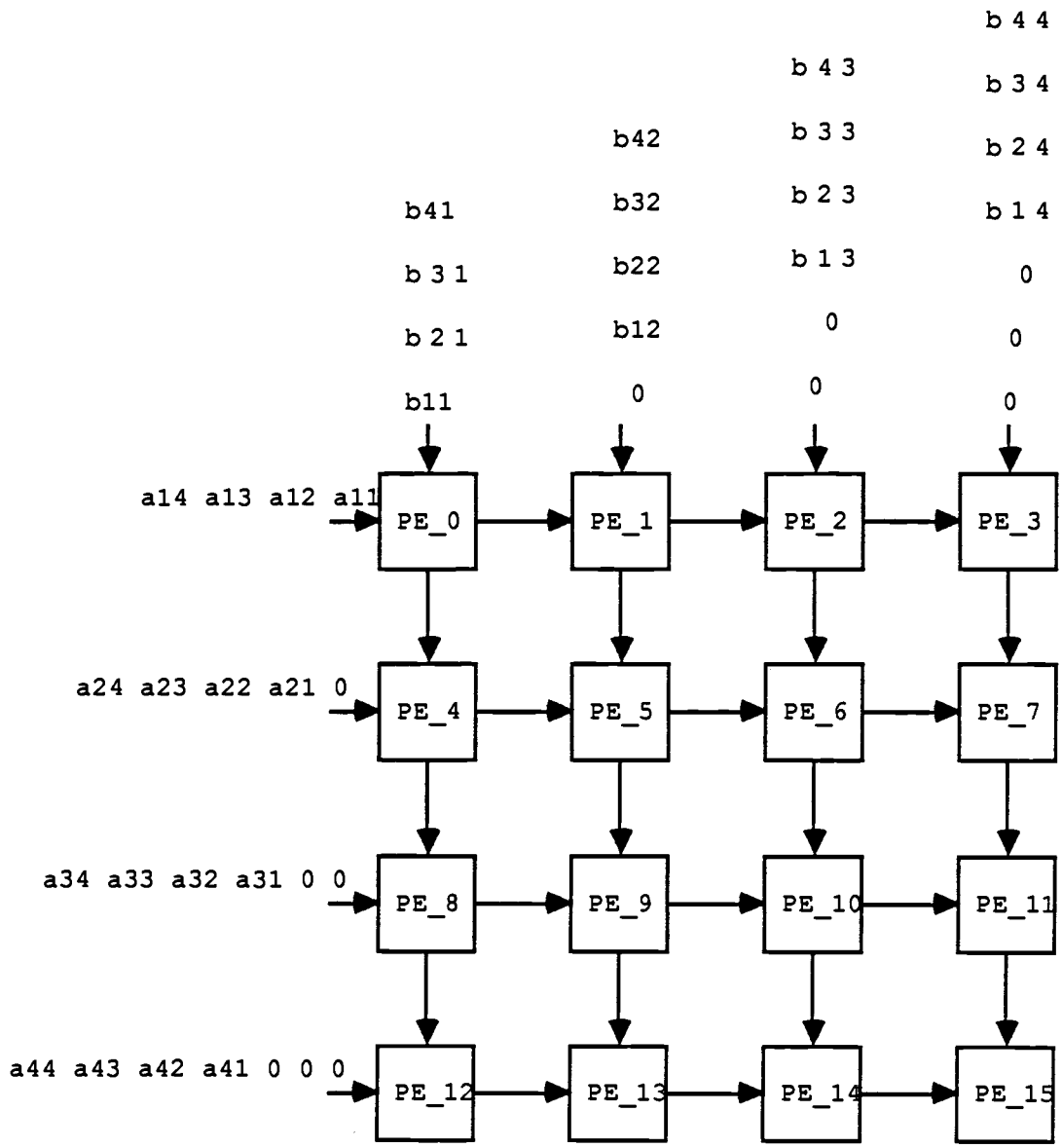


FIG 7. SNAPSHOTS OF FULL MATRIX MULTIPLICATION

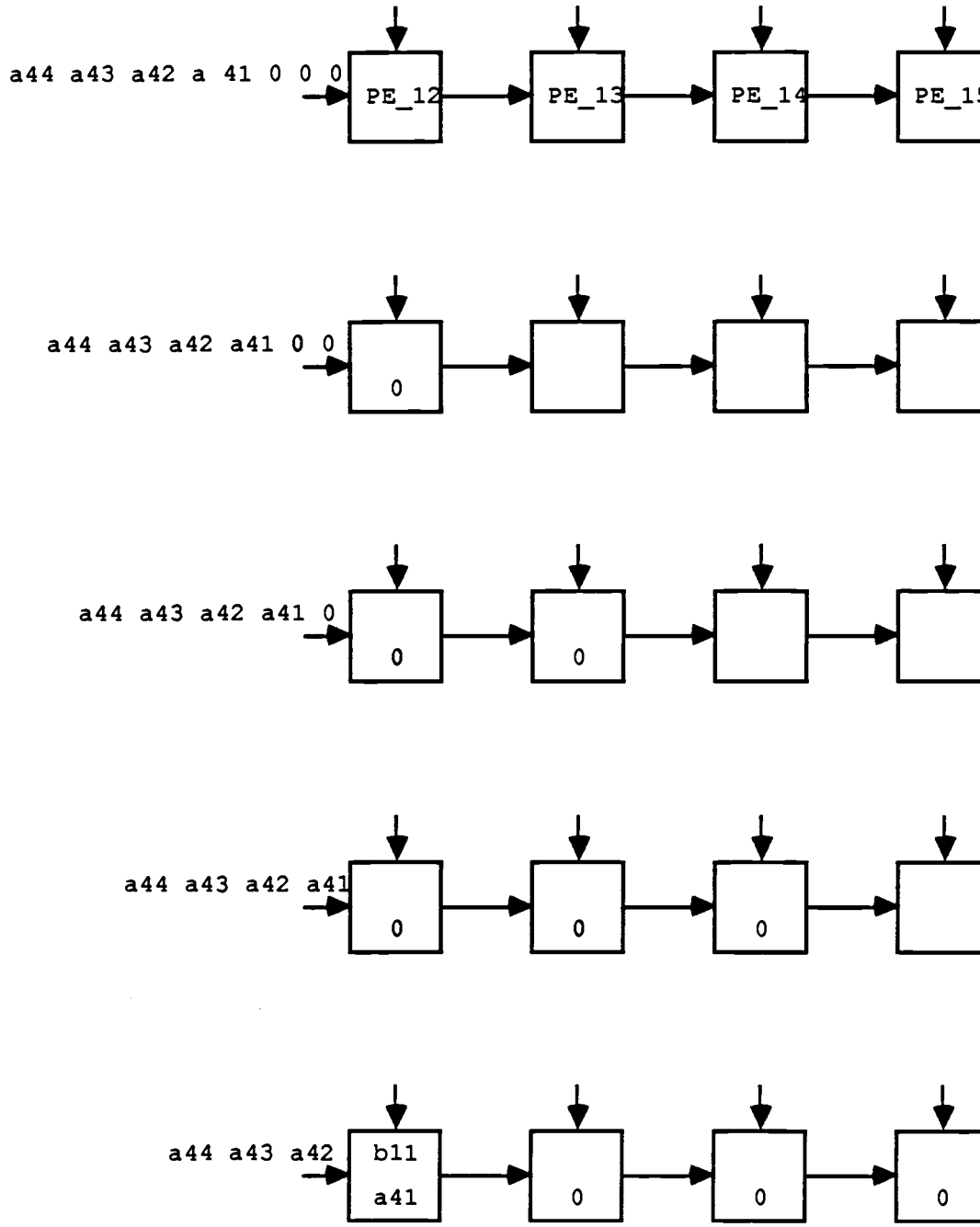


FIG 7. CONTINUED.

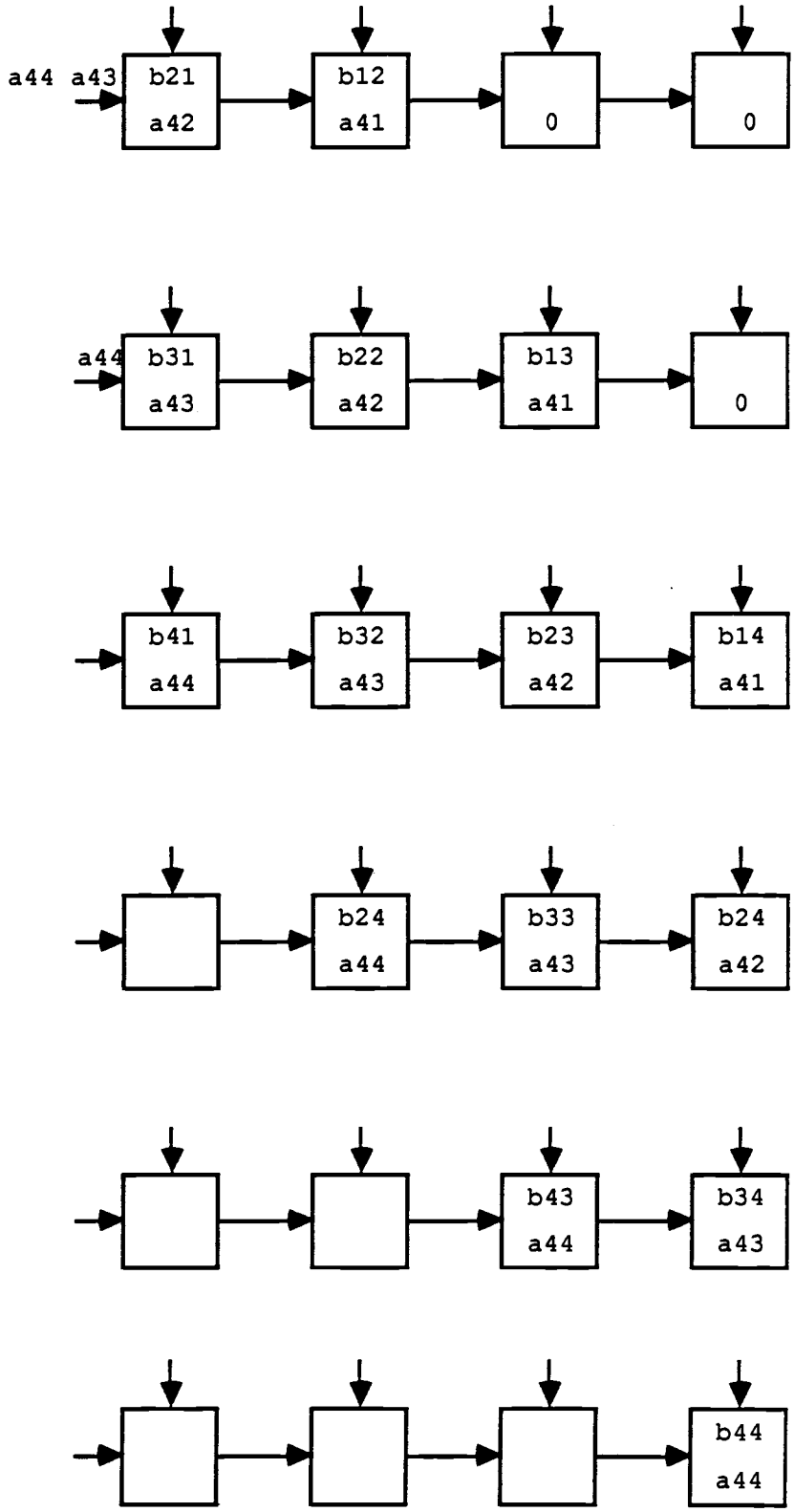


FIG 7. CONTINUED.

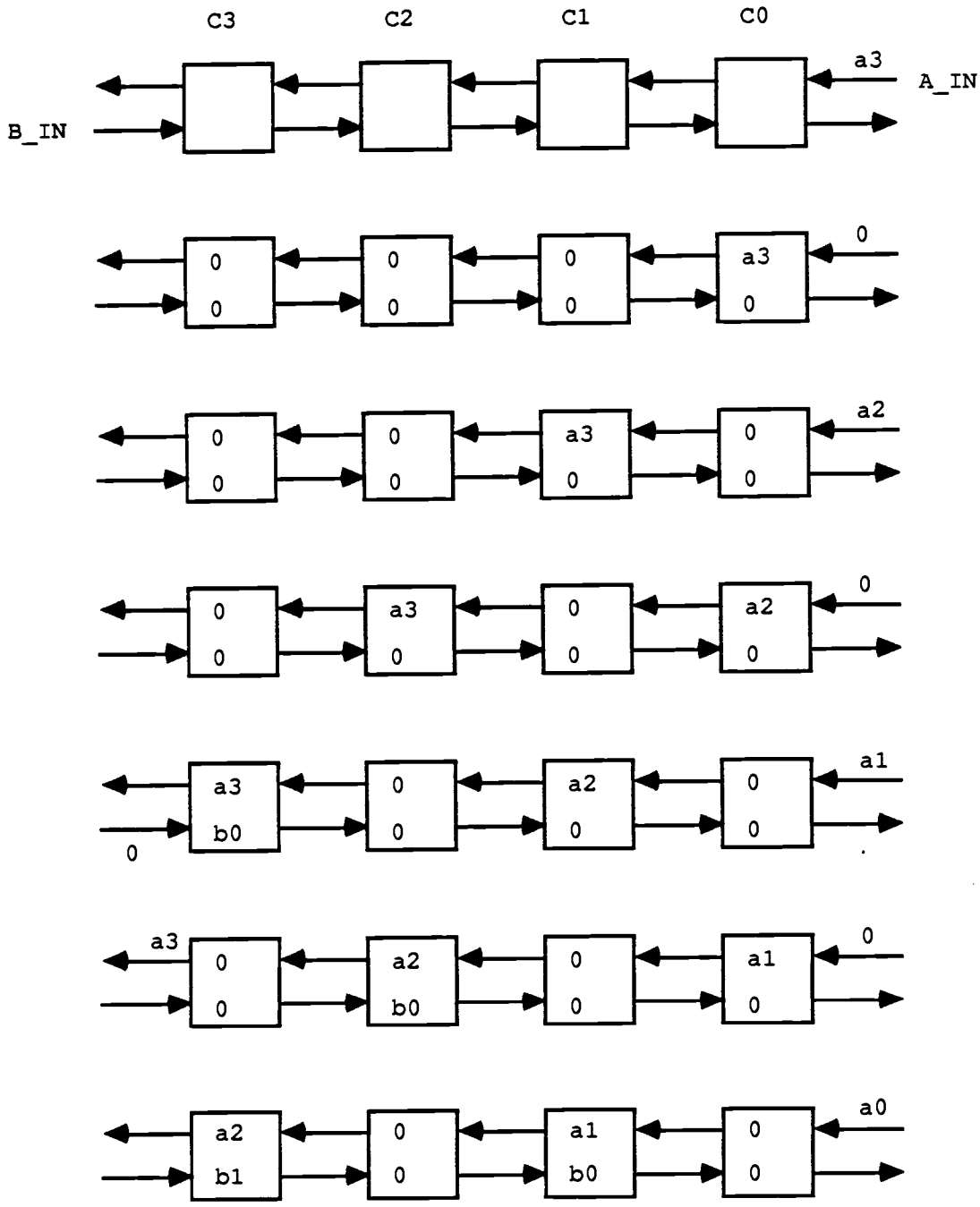


FIG 8. SNAPSHOTS OF CONVOLUTION ALGORITHM.

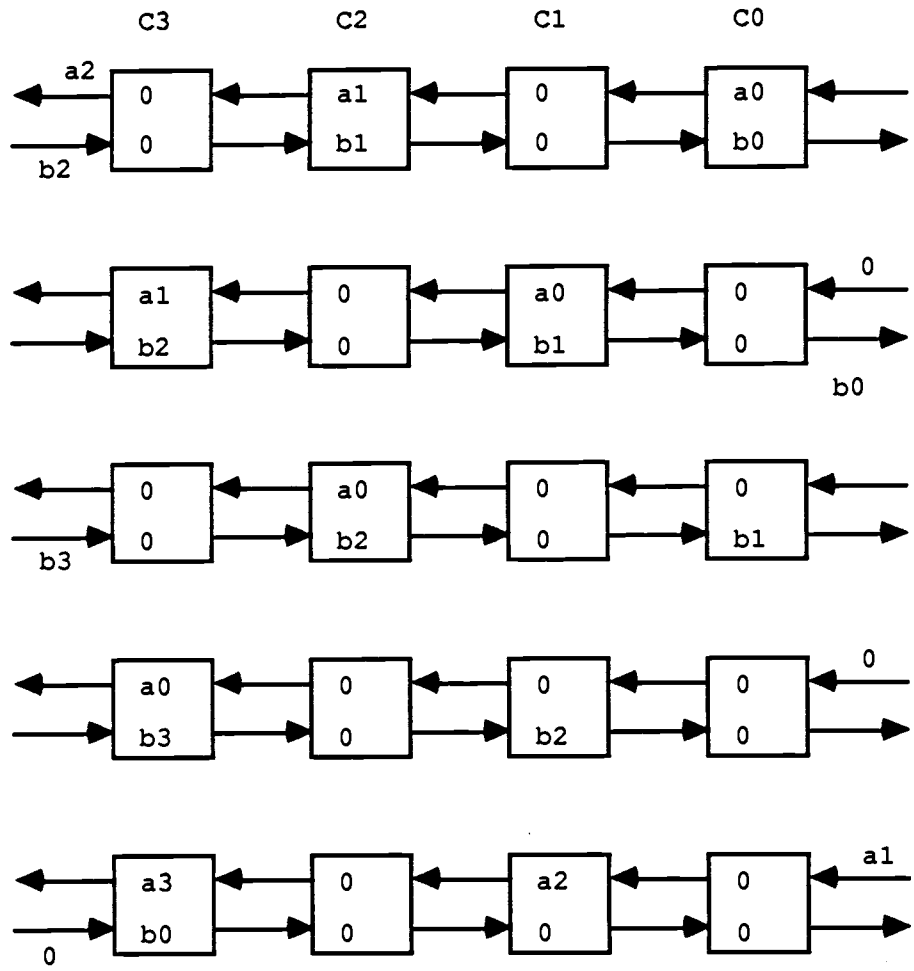


FIG 8. CONTINUED.

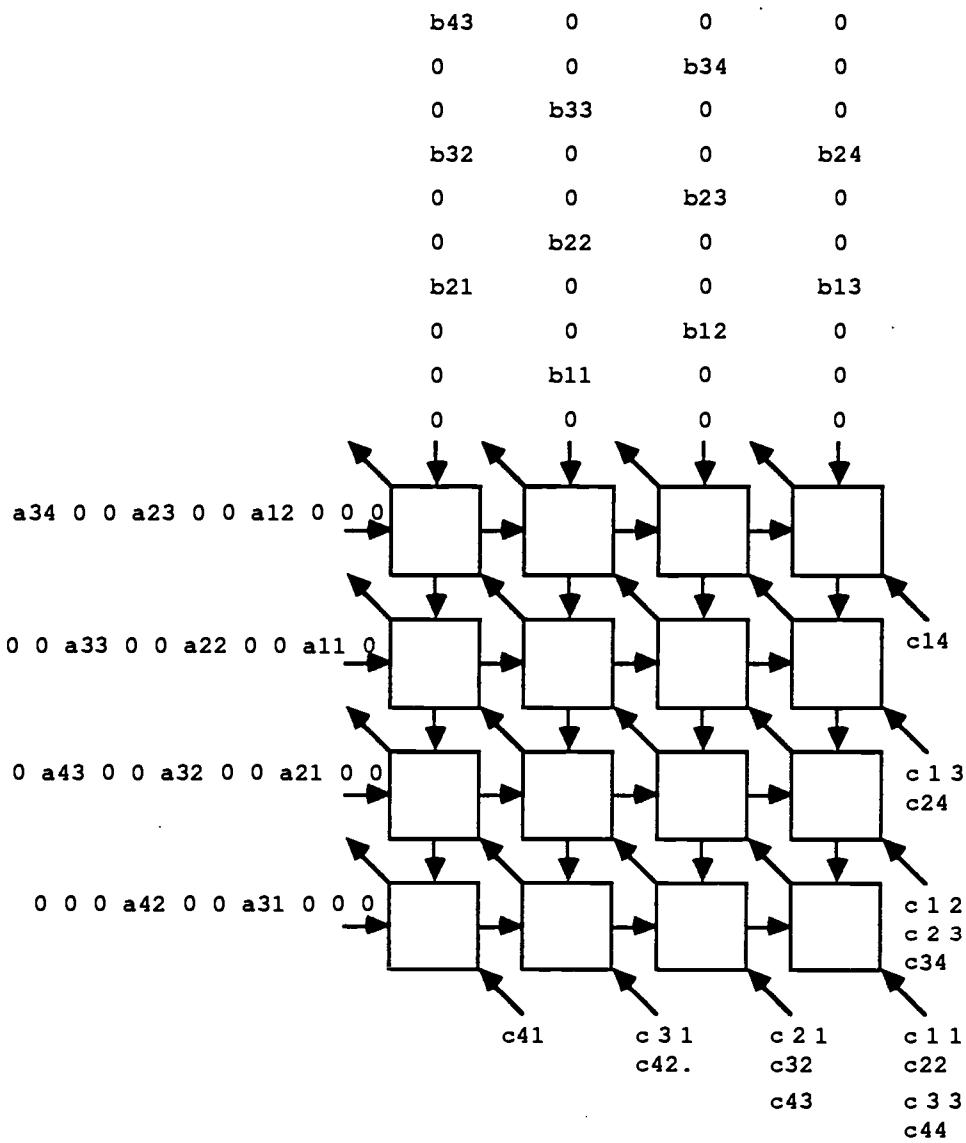


FIG 9. SNAPSHOTS OF BAND MATRIX MULTIPLICATION.

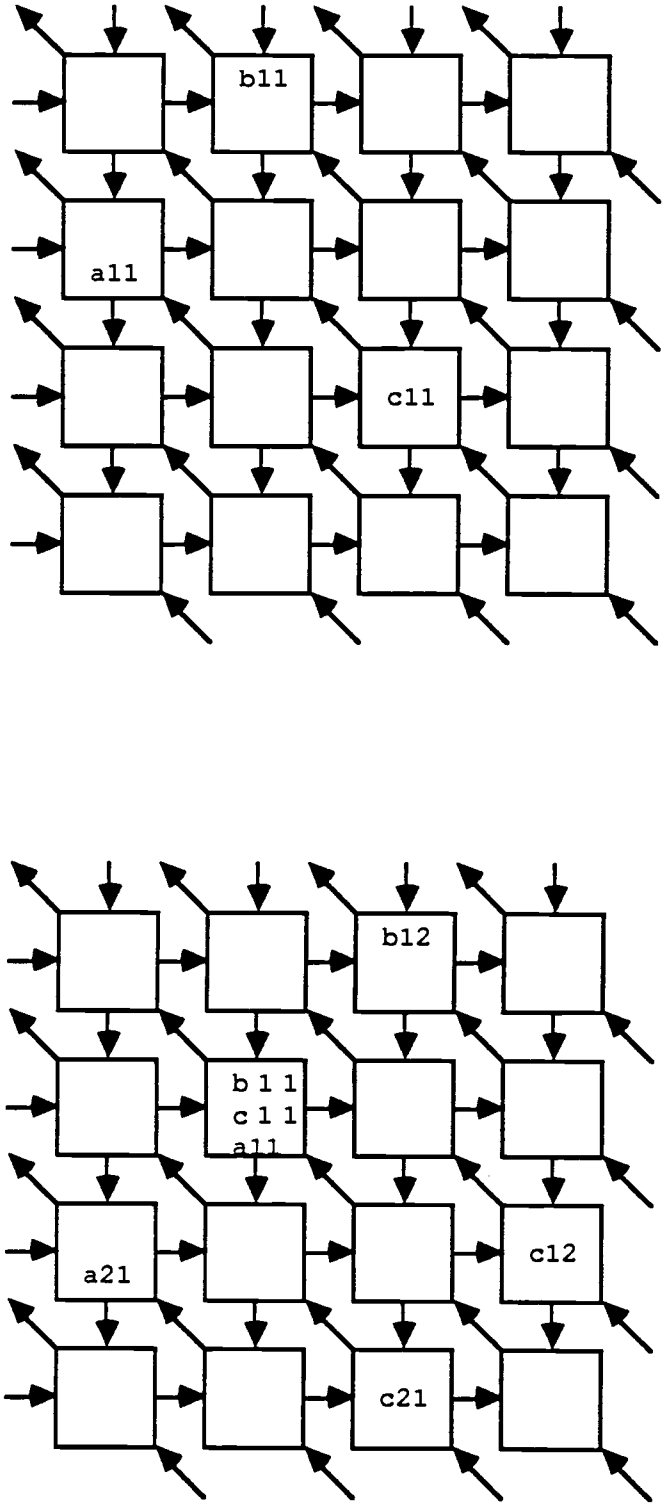


FIG 9. CONTINUED.

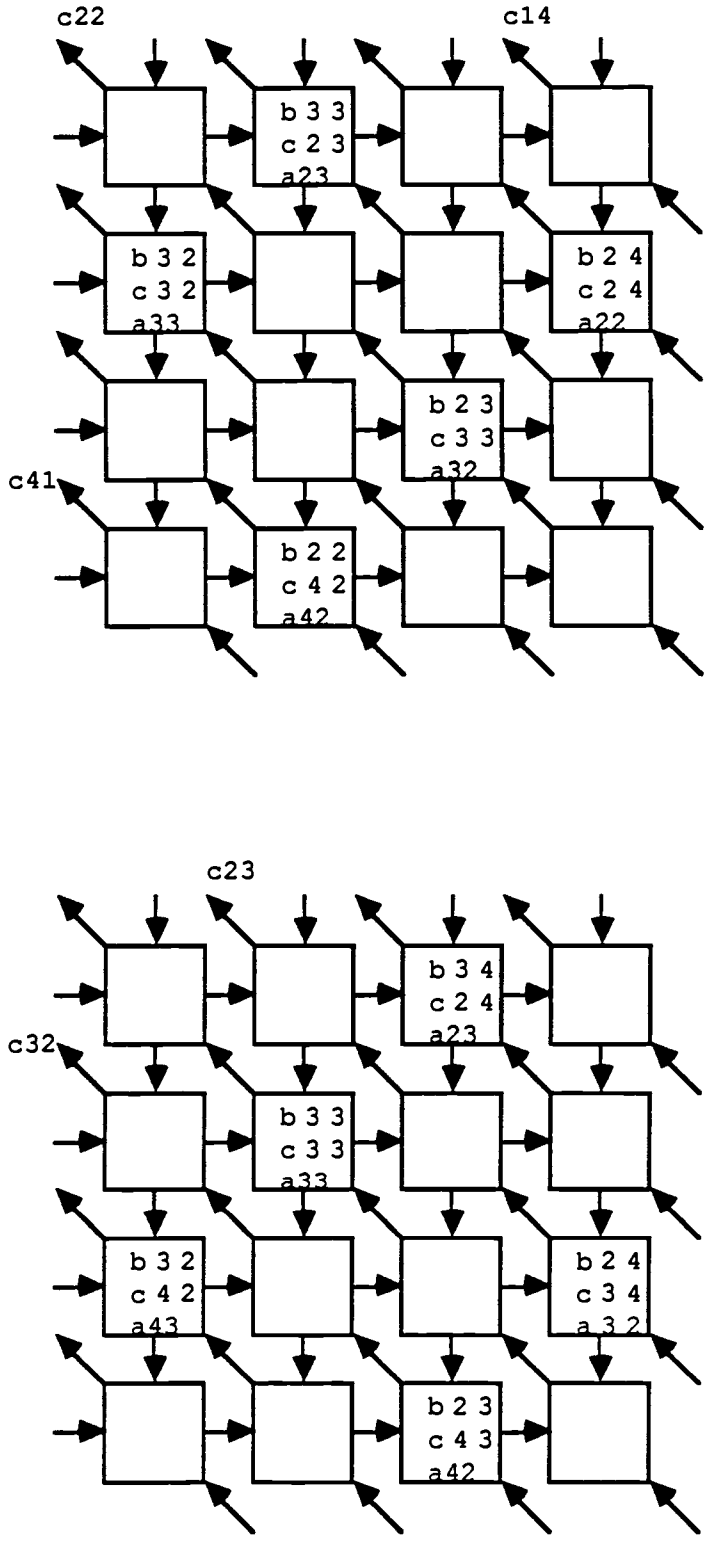
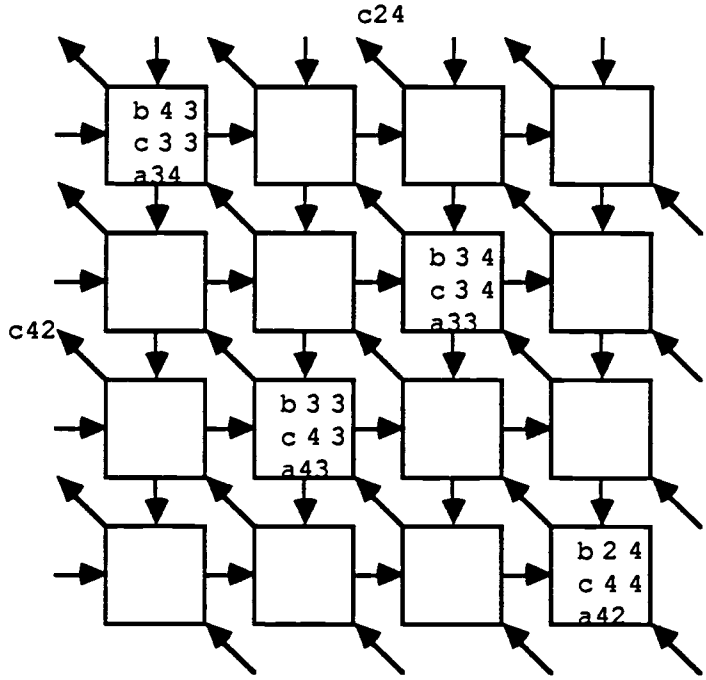


FIG 9. CONTINUED.



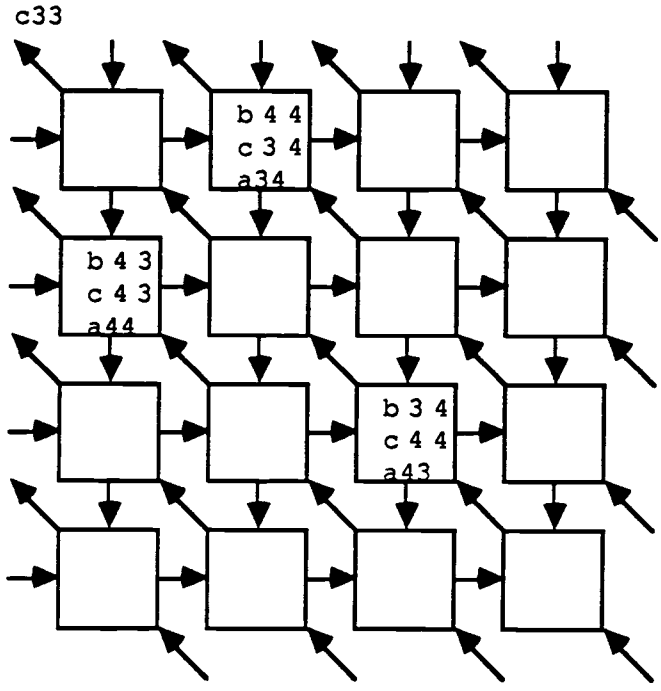
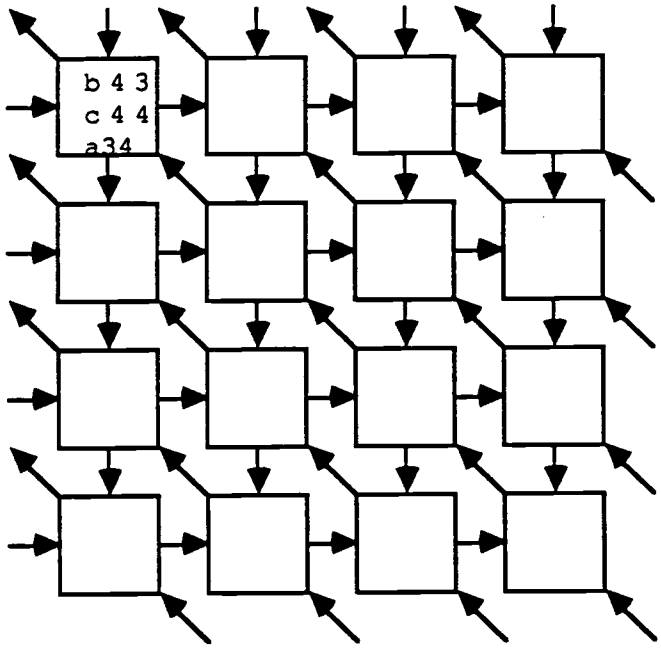


FIG 9. CONTINUED.



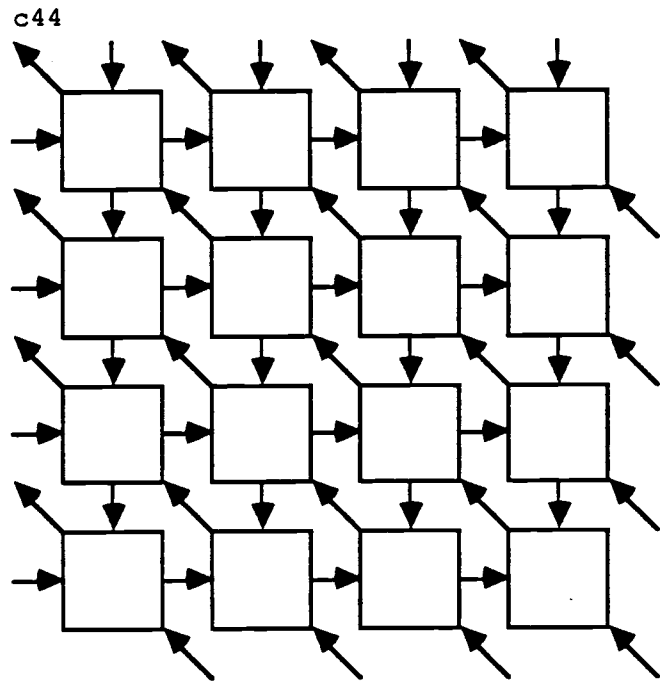


FIG 9. CONTINUED.

5.2 Actual Timing Calculations

5.2.1 Full Matrix Multiplication - A Square Array Application:

As already stated, the entire array needs to be initialized in order to initiate self controlled actions. As is evident from Figure 6, a minimum of five time cycles are required for this purpose.

The actual system timing is a little different. The compiler requires two time units to begin processing. Only after two time cycles does the system actually follow the initialization process as depicted in Figure 6.

Initialization Time : From the above explanation, the actual time required by the array

IT_actual = (N + 1) + 2 units.

Computation Time : The computation time for Full Matrix Multiplication consists of

1. the time required for clearing all datapaths (Reset Time) and
2. the time required to compute the resultant matrix (Computation Time).

Actual Reset Time : Timing constraints in the Silicon Compiler require two time units before actual processing can begin.

Therefore, actual reset time taken by the practical array is

RT_actual = RT_theoritical + 2 = (N + 1) + 2 units.

Computation Time : The data is input as shown in the snapshots in Figure 7.

There are two control words associated with the execution of the algorithm. These are the words at ROM locations 12 and 13 [See ROM microcode in Appendix]. ROM location 12 is responsible for generating control signals required for the IPE. ROM location 13 generates the signals required by the Interconnect Switch (IS). Control signals are generated one time cycle after the address is available in the AMR [See ROM timing diagram in Appendix]. As a result, the data for computation should be available one time unit after the AMR has address 12. This means that when the controller is in State 12, the interconnect switches have their controls available. When the controller is in State 13, the data is taken in by the processor and computations are performed. It is important to understand this operating condition since data has to be fed into the array in such a way that it is available at the correct time [See State Table in Appendix].

From the above discussion, it should be clear that computations take place every alternate time unit and data is read into a processing element in alternate cycles [See IPE timing diagram in Appendix]. The array is so designed that at any point in time, alternate processors are in the same state [See results of controller test in Appendix].

The time required by the array to complete the execution of the algorithm is equal to the time taken by the PE_NN to finish operating on the last data inputs and can be calculated in the following steps :

Step 1 : PE_NN receives it's first data inputs three time cycles after PE_1N and PE_N1 receive their inputs.

Step 2 : From Figure 7 it is clear that the last inputs into PE_1N and PE_N1 (b_{44} and a_{44})occurs seven time cycles after execution of the algorithm begins. Therefore, PE_NN receives these inputs ten cycles after execution begins. Therefore, theoretically, the algorithm requires ten time units for execution.

Now, it was already mentioned that data is input or read in by the array in alternate cycles. Therefore the number of time units required by the proposed array should be double the theoretical value i.e. twenty. This value concurs with the practical results.

This result can be extrapolated to calculate and predict the time required by an N*N array to execute the above algorithm. In order to execute a matrix multiplication involving two N*N matrices, the time required can be calculated as twice the theoritical time calculated in section 4.1.1 and is given by

Computation Time(Actual) = 2*(3N - 2) =

(6N - 4).

For the case when N = 4, the time is

(6*4) - 4 = 20 which is in agreement with the practical results.

The total time from start to finish is the sum of the initialization time IT, the reset time RT and the computation time CT.

The theoretical time (TT_fullmat) is

TT_fullmat = IT + RT + CT = (N + 1) + (N + 1) + (3N - 2)

and the actual time (AT_fullmat) is

AT_fullmat = (N + 3) + (N + 3) + (6N - 4).

5.2.2 Convolution : A Bi-directional Linear Array Application:

The initialization time IT is same as that for Full Matrix Multiplication as explained in section 4.1. Therefore IT_actual = (N + 3) time units.

Reset and Computation Times :

For the algorithm, the A inputs are from the left i.e. from PE_3 and the B inputs are from the right i.e. from PE_0. Also, the computation does not begin until the first A input namely A3 reaches PE_0 [See Figure 7].

The control block for this algorithm consists of the Rom locations 8, 9, 10 and 11. After initialization, control is passed to state 10 where the datapaths are cleared [See microcde file in Appendix]. Since the datapath clearing is the same as for the square array and because only one row needs to be cleared, the time required for resetting the row is 2 time units.

Once the row of processors is reset, computation can begin. A pictorial representation of the events at various time instants is given [Figure 8]. The snapshots represent the exact sequence of events in the row and therefore the number of time units required is 14. Since the algorithm also requires that data be input in alternate cycles, the theoretical time is also 14 cycles [See output of test vector file for this algorithm in Appendix].

In general, the time required for a linear array of N processors is the same as the theoretical value and can be calculated as follows.

Step 1 : The number of B inputs is N and the number of time units required after the first B input is read is (2N - 1).

Step 2 : Then another (N - 1) time units are required to propagate the last input across the array.

Therefore, the total time required is

Computation Time_actual = N + (2N - 1) + (N - 1)

= 4N - 2.

Computational Time_actual = Computation Time_theoretical.

5.2.3 Band Matrix Multiplication : A Hexagonal Array Application.

Initialization Time: The actual time for initialization, IT_actual is a function of the Silicon Compiler and as already mentioned, this time is given by IT_actual = IT_theoretical + 2 = (N + 3) units.

Computation Time : The actual computation time for the band matrix multiplication is calculated as follows :

From the snapshots in Figure 9, it is clear that the last element of the resultant matrix that is evaluated by the array is the last element in the main diagonal of the resultant matrix. This element in our case is C_{44}. The partial result of this element travels along the main diagonal of the array and the last multiply and add operation involved is the addition of the term (A_{34}*B_{43}) in the processor PE_0.

Therefore, the time required for algorithm execution is the time taken by the elements A_{34} and B_{43} to reach PE_0. Now, neglecting the first row and column of zeros that are input to the array [Fig 9], it is clear that a new element is input into any row or column once every three time cycles. Also, from the manner in which the data is input to the array, it is seen that the above two elements are input into the array nine time units after the first element is read in.

Actual Computation times : In the actual implementation of the algorithm, the controller cycles through three states described by the ROM locations 16, 17 and 19. The data however is read in only during one of these states. State 17 reads in the data, state 16 actuates the interconnect switch and state 19 sends the partial results to the processor in the northeast direction. Therefore the actual time required

by the array built on the compiler should be three times the theoretical value. Therefore

Computation Time_actual = (9 * N) time units. This is verified from the results obtained from the simulation.

From the results obtained on the Compiler, it is seen that the proposed array does not change the execution time in any manner except to accommodate the various delays in the actual architectural hardware.

5.3 Timing Analysis Based On Individual Component Delays

The maximum possible frequency at which the array will work can be determined by evaluating the actual delays in the components of the processors. The actual delays can be calculated by taking into account the setup times, hold times and propagation delays in these components [15].Within the array, all processors are identical. Also, different interconnection patterns do not change the micro-architecture of the processors. Therefore, except for the propagation delays due to interconnect lengths, the delays are the same for all configurations.

With reference to Figure 4, the various delays associated with each processor are

1. T_{sum} : Setup time for the multiplexers,
2. T_{pm} : Propagation delay in the multiplexers,
3. T_{sul} : Setup time for the latches in the input selector switch,
4. T_{hl} : Hold time for the latches,
5. T_{pl} : Propagation delay within the latches,
6. t_{pmult} : Propagation time for the multiplier and
7. t_{padd} : propagation time for the adder.

The total delay within each processing element is the sum of all the above delays and is given by

$$T_{proc} = T_{sum} + T_{pm} + T_{sul} + T_{hl} + T_{pl} + T_{pmult} + T_{padd}.$$

The width of the global clock pulse must therefore be atleast as wide as T_{proc} if the processor should work without erroneous computations. The frequency of the clock is the inverse of the time period and is given by

Maximum frequency of the clock = $1/T_{proc}$.

The actual values of the various delays depend on the process and technology used to fabricate the array and the system speed can therefore be calculated for different technologies.

6. CONCLUSIONS

In this thesis a 2_dimensional systolic array that lends itself to dynamic reconfiguration has been proposed. The array is capable of implementing three different interconnection patterns. This enables the execution of problems that require different interconnection schemes on a single array of processors. The proposed dynamically configurable array does not reduce the efficiency of execution of the various algorithms and at the same time reduces cost and hardware. Systolic arrays are algorithmically specialized processors and the possibility of having a single reconfigurable array capable of executing a variety of algorithms makes the use of these arrays a better proposition for solving real time computationally intensive problems.

The implicit integration of the interconnecting switches and the other components of the processors in the array along with the capability of programming these switches by a controller preserves the locality of the algorithm. Moreover, the range of applications can be expanded by changing the control structure with no added costs in terms of hardware or extra delays in the system.

In terms of the actual area of each processor and the entire array itself, the systolic nature results in an area that is largely owing to the size of the processing elements. Each processor in th array has an area of 96*91

mils (A mil is one thousandth of an inch). The area of the entire array is 436*425 mils. Since the array was arranged as a 4*4 square grid, each dimension of the array is just over four times the corresponding dimension of each processor. The extra area is due to the interconnections between the various processing elements. It is therefore clear that the interconnection patterns do not increase the area of the array by a very large amount.

All these characteristics of the proposed array make it very suitable for a great many applications. The fact that the array has been implemented and functionally simulated make it viable for commercial applications.

There are however, a number of improvements that can be made. To begin with, the IPE operates only on 4 bit data. But it is common knowledge most applications operate on 8, 16 and 32 bit data. Therefore, the processing element can be redesigned to operate on these larger data values. Another area where improvements can be made is the I/O capability. Currently, the chip design has not been packaged and as a result, no pads have been put on the external I/O connectors. This was because of having many more I/O connectors than any standard commercially available package. The I/O design can be revamped to reduce this number and make it adaptable to the available packaging technology.

In terms of system timing, the behaviour of the array under clock frequencies greater than 10 MHz must be looked

into. This would give an idea about the maximum allowable clock frequency that the system can run on. This can be done by using the timing analysis based on the setup and hold times of the various components of the processor. Values for the setup and hold times can be substituted depending on the technology and the process used to fabricate the array.

The system can be improved and tested for all the above conditions but in this thesis the idea was to demonstrate the feasibility of a reconfigurable system and this objective was achieved.

BIBLIOGRAPHY

1. Carver Mead and Lynn Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, Mass., 1980.

2. H.T.Kung, " Why Systolic Architectures ? ", Computer, Vol 15, No 1, Jan 1982, pp.37-46.

3. Lawrence Snyder, " Introduction to the Configurable, Highly Parallel Computer ", Computer, Jan 1982, pp.47-56.

4. Jeffrey.D.Ullman, Computational Apects of VLSI, Computer Science Press, 1984.

5. S.Y.Kung, VLSI Array Processors, Prentice Hall, 1988.

6. M.Annaratone, E.Arnould, T.Gross, H.T.Kung and O.Menzilcioglu, K.Sarocky and J.A.Webb, " Warp Architecture and inplementation," in Proc. 13th Annu. Int. Symp. Comput. Architecture, IEEE/ACM, June, 1986, pp. 346-356.

7. A.L.Fisher, H.T.Kung and K.Sarocky, "Experience with the CMU programmable systolic chip," Microarchitecture VLSI Comput., pp.209-222, 1985.

8. P.J.Varman, I.V.Ramakrishnan and D.S.Fussell, "Fault tolerant matrix multiplication in VLSI," Technical report, University of Texas at Austin, 1981.

9. Guo-Jie Li and Benjamin Wah, "The design of optimal systolic algorithms," Technical report, Purdue University, 1983.

10. H.T.Kung, "On the implementation and use of systolic array processors," Proceedings of International Conference on Computer design : VLSI in Computers, pp. 370-373, IEEE, 1983.

11. Patrice Quinton, "The systematic design of systolic arrays," Technical Report 193, IRISA, April, 1983.

12. H.T.Kung and W.T.Lin, "An algebra for VLSI computation," In G.Birkhoff and A.L.Schoenstadt (editors), Elliptic Problems Solvers II, Academic Press, 1983.

13. M.J.Foster and H.T.Kung, "The design of special purpose VLSI chips," Computer Magazine 13(1), pp.26-40, January, 1980.

14. G.Alia, "VLSI systolic arrays for band matrix multiplication," Technical Report, Istituto di Elaborazione dell'Informazione, January, 1983.

15. G.Langdon, "Computer Design", Computeach Press Inc., 1982.

APPENDICES

FULL MATRIX MULTIPLICATION ON A SQUARE ARRAY.

The matrix product C = (C_{ij}) of A = (A_{ij}) and B($_{ij}$) can be computed by the following recurrences :

$c_{ij} = 0$

$c_{ij} = c_{ij} + a_{ik}b_{kj}$

$c_{ij} = c_{ij}$.

If A and B are n*n band matrices respectively, the recurrences can be evaluated by pipelining the a_{ij} and b_{ij} through an array of n*n processors. The square shaped interconnection network is shown in Figure[7] where the processors are square connected and data flows are indicated by arrows.

The elements in matrices A and B move through the network in two directions synchronously. Each c_{ij} is initialized to zero. The algorithm can be easily verified by studying the data flow as shown in the snapshots shown in Figure[7]. The algorithm can be summarized as follows :

If each processor contains three registers Reg_A, Reg_B and Reg_C which hold the entries in the A matrix, the B matrix and the C values (partial results), the each step of the algorithm consists of the following operations (for odd numbered steps, only the odd numbered processors are active and for even numbered steps, the even number processors are active) :

1. SHIFT.

Reg_A gets a new element in the row of matrix A.

Reg_B gets a new element in the column of matrix B.

Reg_C gets the contents of the accumulator from the previous step of Multiply and ADD.

2. MULTIPLY AND ADD.

Reg_C = Reg_C + Reg_A + Reg_B.

The test case that was run on the array was as follows :

$$A = \begin{matrix} 10 & 13 & 17 & 15 \\ 6 & 7 & 8 & 9 \\ 2 & 3 & 4 & 5 \\ 4 & 8 & 12 & 1 \end{matrix}$$

$$B = \begin{matrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{matrix}$$

The product A*B is given by he resultant matrix C and the elements of C can be calculated to be

$$C = \begin{matrix} 52 & 104 & 156 & 208 \\ 30 & 60 & 90 & 120 \\ 14 & 28 & 42 & 56 \end{matrix}$$

25 50 75 100

In terms of Hexadecimal numbers, C evaluates to

$$C = \begin{matrix} 34 & 68 & 9c & d0 \\ 1e & 3c & 5a & 78 \\ 0e & 1c & 2a & 38 \\ 19 & 32 & 4b & 64 \end{matrix}$$

The values obtained from the output of the array concur with these values thus verifying the validity of the array configuration.

SYSTOLIC CONVOLUTION.

Convolution is the problem where given two sequences of numbers a0, ..., an-1 and b0, ..., bn-1, a sequence of results c0, ..., c2n-1 are produced where

c_i = Sum of ($a_j b_{i-j}$) for j = 0 to n-1.

Therefore $c_0 = a_0 b_0$, $c_1 = a_0 b_1 + a_1 b_0$,

$c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0$ and so on.

The convolution operation is important in digital signal processing, and it is also close to integer multiplication, in the case where the a's and b's are each 0 or 1.

A linear array of processors is required to solve this problem. These processors may be in one row or they may be snaked around a chip.

The convolution algorithm has been implemented using the design R1 as given by H.T.Kung [3]. Here, the results stay in the processors while the two input streams move in opposite directions [Fig 8]. The results stay in the cells or processors to accumulate their terms. The A_i's and B_i's move systolically in opposite directions such that when any A_i meets a B_i at a cell, they are multiplied and the resulting product is accumulated to the result C staying at that cell. To ensure that each A_i is able to meet every B_i, consecutive A_i's on the A data stream are separated by two cycle times and so are the are the B_i's on the B stream.

A detailed pictorial representation is given in Figure[8].

This design makes efficient use of the available multiplier-accumulator hardware. Further details of the convolution algorithm can be obtained from [3].

BAND MATRIX MULTIPLICATION ON A HEX ARRAY

The matrix product C = (C_{ij}) of A = (A_{ij}) and B($_{ij}$) can be computed by the following recurrences :

$c_{ij} = 0$

$c_{ij} = c_{ij} + a_{ik}b_{kj}$

$c_{ij} = c_{ij}$.

If A and B are n*n band matrices of bandwidth w1 and w2 respectively, the recurrences can be evaluated by pipelining the a_{ij}, b_{ij} and c_{ij} through an array of w1w2 processors. The diamond shaped interconnection network is shown in Figure[9] where the processors are hex connected and data flows are indicated by arrows.

The elements in the bands of A, B and C move through the network in three directions synchronously. Each c_{ij} is initialized to zero as it enters the network through the bottom boundaries. Given the processor architecture, it can be easily seen that each c_{ij} is able to accumulate all its terms before it leaves the network through the upper boundaries.

The algorithm can be easily verified by studying the data flow as shown in the snapshots shown in Figure[]. The algorithm can be summarized as follows :

If each processor contains three registers Reg_A, Reg_B and Reg_C which hold the entries in the A matrix, the

B matrix and the C values (partial results), the each step of the algorithm consists of the following operations (for odd numbered steps, only the odd numbered processors are active and for even numbered steps, the even number processors are active) :

1. SHIFT.

   Reg_A gets a new element in the band of matrix A.

   Reg_B gets a new element in the band of matrix B.

   Reg_C gets the contents of the Reg_C from the neighbouring processor in the Southeast direction.

2. MULTIPLY AND ADD.

   Reg_C = Reg_C + Reg_A + Reg_B.

TEST VECTOR FILE FOR FULL MATRIX MULTIPLICATION : SQUARE ARRAY APPLICATION

```
$ define Sig Signal
$ define Pos Position
$ define Len Length
$ define In Input, Par = " to = 1"
$ define Out Output , Par = " to = 2 "
$ define Sh Shift
$ define Expr Expression
$ define null Default=0;

Fields {

addr    (In, Pos = 0, Sig = EXT_ADDR[2:0])          {}
a0      (In, Pos = 3, Sig = A0_IN1[3:0],Sh = 10 )   {null}
a4      (In, Pos = 7, Sig = A4_IN1[3:0],Sh = 10 )   {null}
a8      (In, Pos = 11,Sig = A8_IN1[3:0],Sh = 10 )   {null}
a12     (In, Pos = 15,Sig = A12_IN1[3:0],Sh =10 )   {null}
b0      (In, Pos = 19,Sig = B0_IN2[3:0],Sh = 10 )   {null}
b1      (In, Pos = 23,Sig = B1_IN2[3:0],Sh = 10 )   {null}
b2      (In, Pos = 27,Sig = B2_IN2[3:0],Sh =  0 )   {null}
b3      (In, Pos = 31,Sig = B3_IN2[3:0],Sh =  0 )   {null}

PHASE_A (Out,Pos = 0, Len = 1            )          {}
PHASE_B (Out,Pos = 1, Len = 1            )          {}
```

```
RES_0    (Out,Pos = 2, Len = 8          )          {}
RES_1    (Out,Pos = 10,Len = 8          )          {}
RES_2    (Out,Pos = 18,Len = 8          )          {}
RES_3    (Out,Pos = 26,Len = 8          )          {}
RES_4    (Out,Pos = 34,Len = 8          )          {}
RES_5    (Out,Pos = 42,Len = 8          )          {}
RES_6    (Out,Pos = 50,Len = 8          )          {}
RES_7    (Out,Pos = 58,Len = 8          )          {}
RES_8    (Out,Pos = 66,Len = 8          )          {}
RES_9    (Out,Pos = 74,Len = 8          )          {}
RES_10   (Out,Pos = 82,Len = 8          )          {}
RES_11   (Out,Pos = 90,Len = 8          )          {}
RES_12   (Out,Pos = 98,Len = 8          )          {}
RES_13   (Out,Pos = 106,Len = 8         )          {}
RES_14   (Out,Pos = 114,Len = 8         )          {}
RES_15   (Out,Pos = 122,Len = 8         )          {}

}

Templates {
initialize[] = addr\@0;
calc[][][][][][][][][][]                                              =
addr\@0,a0\@1,a4\@2,a8\@3,a12\@4,b0\@5,b1\@6,b2\@7,b3\@8;
}

Lineaction :: Expr( .=. +10);
```

```
Data {
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
/*  END  OF  ARRAY  INITIALIZATION.  DATAPATH  CLEARING  AND
CONTROL SETTING STARTS*/
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
/* CALCULATION AND ACTUAL DATA INPUTS BEGIN HERE */
calc[3][0][0][0][0][0][0][0][0];
calc[3][10][0][0][0][1][0][0][0];
calc[3][ 0][0][0][0][0][2][0][0];
calc[3][13][6][0][0][1][0][0][0];
calc[3][ 0][0][0][0][0][2][3][0];
calc[3][14][7][2][0][1][0][0][4];
calc[3][ 0][0][0][0][0][2][3][0];
```

```
calc[3][15][8][3][4][1][0][0][4];
calc[3][0 ][0][0][0][0][2][3][0];
calc[3][0 ][9][4][8][0][0][0][4];
calc[3][0][0][0][ 0][0][0][3][0];
calc[3][0][0][5][12][0][0][0][4];
calc[3][0][0][0][0 ][0][0][0][0];
calc[3][0][0][0][1 ][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
calc[3][0][0][0][0][0][0][0][0];
/*OUTPUT COMMAND */
initialize[1];
}
```

```
/* TEST VECTOR FILE FOR CONVOLUTION : A LINEAR ARRAY
                    APPLICATION */

$ define Sig Signal
$ define Pos Position
$ define Len Length
$ define In Input, Par = " to = 1"
$ define Out Output , Par = " to = 2 "
$ define Sh Shift
$ define Expr Expression
$ define null Default=0;
/* The various input and output fields are declared here */
/* In represents Input, Out stands for Output and Sig stands
/* for the signal name that the field represents.*/
Fields {
addr    (In, Pos = 0, Sig = EXT_ADDR[2:0])          {}
a0      (In, Pos = 3,Sig = A3_IN2[3:0],Sh = 10 )    {null}
b0      (In, Pos = 7,Sig = B0_IN1[3:0],Sh = 10 )    {null}
b1      (In, Pos = 11,Sig= B0_IN2[3:0],Sh = 10 )    {null}
b2      (In, Pos = 15,Sig= B1_IN2[3:0],Sh = 10 )    {null}
b3      (In, Pos = 19,Sig = B2_IN2[3:0],Sh = 10)    {null}
b4      (In, Pos = 23,Sig = B3_IN2[3:0],Sh = 10)    {null}

PHASE_A  (Out,Pos = 0, Len = 1        )             {}
PHASE_B  (Out,Pos = 1, Len = 1        )             {}
RES_0    (Out,Pos = 2, Len = 8        )             {}
```

```
RES_1     (Out,Pos = 10,Len = 8        )              {}
RES_2     (Out,Pos = 18,Len = 8        )              {}
RES_3     (Out,Pos = 26,Len = 8        )              {}

ROM0_ADDR(Out, Pos = 34,Len = 5        )              {}
ROM1_ADDR(Out, Pos = 39,Len = 5        )              {}
ROM2_ADDR(Out, Pos = 44,Len = 5        )              {}
ROM3_ADDR(Out, Pos = 49,Len = 5        )              {}

}

Templates { /* Defines the order of the inputs in the */
/* simulation file */
initialize[] = addr\@0;
calc[][][] = addr\@0,a0\@1,b0\@2;
output[]   = addr\@0;
}

Lineaction :: Expr( .=. +10);

Data { /* initialization begins here */
initialize[0];
initialize[0];
initialize[0];
initialize[0];
initialize[0];
```

```
initialize[0];
initialize[0];
initialize[0];
calc[2][0][0];

/* DATA INPUTS BEGIN HERE */

calc[2][15][0];
calc[2][ 0][0];
calc[2][10][0];
calc[2][ 0][8];
calc[2][5 ][0];
calc[2][0 ][6];
calc[2][1 ][0];
calc[2][0 ][4];
calc[2][0 ][0];
calc[2][0 ][2];
calc[2][0][0];
/* result output */
output[1];

}
```

/* TEST VECTOR FILE FOR BAND MATRIX MULTIPLICATION : A HEXAGONAL ARRAY APPLICATION */

```
$ define Sig Signal
$ define Pos Position
$ define Len Length
$ define In Input, Par = " to = 1"
$ define Out Output , Par = " to = 2 "
$ define Sh Shift
$ define Expr Expression
$ define null Default=0;

Fields {

addr (In, Pos = 0, Sig = EXT_ADDR[2:0])              {}
a0   (In,Pos = 3,Sig = A0_IN1[3:0],Sh = 0)           {null}
a4   (In, Pos = 7,Sig = A4_IN1[3:0],Sh =  0)         {null}
a8   (In, Pos = 11,Sig = A8_IN1[3:0],Sh = 0)         {null}
a12  (In,Pos = 15,Sig = A12_IN1[3:0],Sh = 0)         {null}
b0   (In, Pos = 19,Sig = B0_IN2[3:0],Sh =  0 )       {null}
b1   (In, Pos = 23,Sig = B1_IN2[3:0],Sh = 0)         {null}
b2   (In, Pos = 27,Sig = B2_IN2[3:0],Sh = 0)         {null}
b3   (In, Pos = 31,Sig = B3_IN2[3:0],Sh = 0)         {null}
c3   (In, Pos = 35,Sig = C3_EXT[7:0],Sh = 0)         {null}
c7   (In, Pos = 43,Sig = C7_EXT[7:0],Sh = 0)         {null}
c11  (In, Pos = 51,Sig = C11_EXT[7:0],Sh = 0)        {null}
```

```
c12  (In, Pos = 59,Sig = C12_EXT[7:0],Sh = 0)      {null}
c13  (In, Pos = 67,Sig = C13_EXT[7:0],Sh = 0)      {null}
c14  (In, Pos = 75,Sig = C14_EXT[7:0],Sh = 0)      {null}
c15  (In, Pos = 83,Sig = C15_EXT[7:0],Sh = 0)      {null}

PHASE_A  (Out,Pos = 0, Len = 1 )                    {}
PHASE_B  (Out,Pos = 1, Len = 1 )                    {}
RES_0    (Out,Pos = 2, Len = 8 )                    {}
RES_1    (Out,Pos = 10,Len = 8 )                    {}
RES_2    (Out,Pos = 18,Len = 8 )                    {}
RES_3    (Out,Pos = 26,Len = 8 )                    {}
RES_4    (Out,Pos = 34,Len = 8 )                    {}
RES_8    (Out,Pos = 42,Len = 8 )                    {}
RES_12   (Out,Pos = 50,Len = 8 )                    {}
}
Templates {
initialize[] = addr\@0;
calc[][][][][][][][]=addr\@0,a0\@1,a4\@2,a8\@3,a12\@4,b0\@
5,b1\@6,b2\@7,b3\@8;
}

Lineaction :: Expr( .=. +10);

Data {
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
```

```
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];
calc[0][0][0][0][0][0][0][0][0];

/* END OF ARRAY INITIALIZATION. ALGORITHM INITIALIZATION AND
EXECUTION BEGINS */

calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];

calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];
/*  DATA  INPUT  BEGINS  HERE.  GROUP  OF  THREE  TEMPLATES
REPRESENT ONE INPUT AND A SINGLE BEAT */
calc[4][0][1][0][0][0][1][0][0];
calc[4][0][1][0][0][0][1][0][0];
calc[4][0][1][0][0][0][1][0][0];

calc[4][0][0][1][0][0][0][2][0];
calc[4][0][0][1][0][0][0][2][0];
calc[4][0][0][1][0][0][0][2][0];
```

```
calc[4][2][0][0][1][1][0][0][3];
calc[4][2][0][0][1][1][0][0][3];
calc[4][2][0][0][1][1][0][0][3];

calc[4][0][2][0][0][0][2][0][0];
calc[4][0][2][0][0][0][2][0][0];
calc[4][0][2][0][0][0][2][0][0];

calc[4][0][0][2][0][0][0][3][0];
calc[4][0][0][2][0][0][0][3][0];
calc[4][0][0][2][0][0][0][3][0];

calc[4][3][0][0][2][2][0][0][4];
calc[4][3][0][0][2][2][0][0][4];
calc[4][3][0][0][2][2][0][0][4];

calc[4][0][3][0][0][0][3][0][0];
calc[4][0][3][0][0][0][3][0][0];
calc[4][0][3][0][0][0][3][0][0];

calc[4][0][0][3][0][0][0][4][0];
calc[4][0][0][3][0][0][0][4][0];
calc[4][0][0][3][0][0][0][4][0];

calc[4][4][0][0][0][3][0][0][0];
calc[4][4][0][0][0][3][0][0][0];
```

```
calc[4][4][0][0][0][3][0][0][0];

calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];

calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];



calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];

calc[4][0][0][0][0][0][0][0][0];
calc[4][0][0][0][0][0][0][0][0];

}
```

RESULTS OF THE BAND MATRIX MULTIPLICATION TEST CASE

```
)      C C C C C C C B B B B A A A A E  R R R R R R R PP
)      1 1 1 1 1 7 3 3 2 1 0 1 8 4 0 X  E E E E E E E HH
)      5 4 3 2 1 _ _ _ _ _ _ 2 _ _ _ T  S S S S S S S AA
)      _ _ _ _ _ E E I I I I _ I I I _  _ _ _ _ _ _ _ SS
)      E E E E E X X N N N N I N N N A  1 8 4 3 2 1 0 EE
)      X X X X X T T 2 2 2 2 N 1 1 1 D  2 [ [ [ [ [ [ __
)      T T T T T [ [ [ [ [ [ 1 [ [ [ D  [ 7 7 7 7 7 7 BA
)      [ [ [ [ [ 7 7 3 3 3 3 [ 3 3 3 R  7 : : : : : :
)      7 7 7 7 7 : : : : : : 3 : : : [  : 0 0 0 0 0 0
)      : : : : : 0 0 0 0 0 0 : 0 0 0 2  0 ] ] ] ] ] ]
)      0 0 0 0 0 ] ] ] ] ] ] 0 ] ] ] :  ]
)      ] ] ] ] ]             ]     0
)                                    ]
)      -- -- -- -- -- -- -- - - - - - -  -- -- -- -- -- -- -- --
)      xx xx xx xx xx xx xx x x x x x x x x  xx xx xx xx xx xx xx bb
rvshowdots 0
qckq
)   0: 00 00 00 00 00 00 00 0 0 0 0 I I I I 0 > II II II II II II II 01
)  10: 00 00 00 00 00 00 00 0 0 0 0 I I I I 0 > II II II II II II II 01
)  20: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 I 0 > 00 00 00 II II II 00 01
)  30: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 I 0 > 00 00 00 II II II 00 01
)  40: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 I 0 > 00 00 00 II II 00 00 01
)  50: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 I 0 > 00 00 00 II II 00 00 01
)  60: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 I 0 > 00 00 00 II 00 00 00 01
```

```
)  70: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 I 4 > 00 00 00 II 00 00 00 01
)  80: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 I 4 > 00 00 00 00 00 00 00 01
)  90: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 00 01
) 100: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 00 01
) 110: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 00 01
) 120: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 00 01
) 130: 00 00 00 00 00 00 00 0 0 1 0 0 0 1 0 4 > 00 00 00 00 00 00 00 01
) 140: 00 00 00 00 00 00 00 0 0 1 0 0 0 1 0 4 > 00 00 00 00 00 00 00 01
) 150: 00 00 00 00 00 00 00 0 0 1 0 0 0 1 0 4 > 00 00 00 00 00 00 00 01
) 160: 00 00 00 00 00 00 00 0 2 0 0 0 1 0 0 4 > 00 00 00 00 00 00 00 01
) 170: 00 00 00 00 00 00 00 0 2 0 0 0 1 0 0 4 > 00 00 00 00 00 00 00 01
) 180: 00 00 00 00 00 00 00 0 2 0 0 0 1 0 0 4 > 00 00 00 00 00 00 00 01
) 190: 00 00 00 00 00 00 00 3 0 0 1 1 0 0 2 4 > 00 00 00 00 00 00 00 01
) 200: 00 00 00 00 00 00 00 3 0 0 1 1 0 0 2 4 > 00 00 00 00 00 00 03 01
) 210: 00 00 00 00 00 00 00 3 0 0 1 1 0 0 2 4 > 00 00 00 00 00 00 03 01
) 220: 00 00 00 00 00 00 00 0 0 2 0 0 0 2 0 4 > 00 00 00 00 00 00 03 01
) 230: 00 00 00 00 00 00 00 0 0 2 0 0 0 2 0 4 > 00 00 03 00 00 06 00 01
) 240: 00 00 00 00 00 00 00 0 0 2 0 0 0 2 0 4 > 00 00 03 00 00 06 00 01
) 250: 00 00 00 00 00 00 00 0 3 0 0 0 2 0 0 4 > 00 00 03 00 00 06 00 01
) 260: 00 00 00 00 00 00 00 0 3 0 0 0 2 0 0 4 > 00 03 00 00 09 00 00 01
) 270: 00 00 00 00 00 00 00 0 3 0 0 0 2 0 0 4 > 00 03 00 00 09 00 00 01
) 280: 00 00 00 00 00 00 00 4 0 0 2 2 0 0 3 4 > 00 03 00 00 09 00 00 01
) 290: 00 00 00 00 00 00 00 4 0 0 2 2 0 0 3 4 > 02 00 00 08 00 00 0c 01
) 300: 00 00 00 00 00 00 00 4 0 0 2 2 0 0 3 4 > 02 00 00 08 00 00 0c 01
) 310: 00 00 00 00 00 00 00 0 0 3 0 0 0 3 0 4 > 02 00 00 08 00 00 0c 01
) 320: 00 00 00 00 00 00 00 0 0 3 0 0 0 3 0 4 > 00 00 0c 00 00 12 00 01
```

```
) 330: 00 00 00 00 00 00 00 0 0 3 0 0 0 3 0 4 > 00 00 0c 00 00 12 00 01
) 340: 00 00 00 00 00 00 00 0 4 0 0 0 3 0 0 4 > 00 00 0c 00 00 12 00 01
) 350: 00 00 00 00 00 00 00 0 4 0 0 0 3 0 0 4 > 00 0a 00 00 14 00 00 01
) 360: 00 00 00 00 00 00 00 0 4 0 0 0 3 0 0 4 > 00 0a 00 00 14 00 00 01
) 370: 00 00 00 00 00 00 00 0 0 0 3 0 0 0 4 4 > 00 0a 00 00 14 00 00 01
) 380: 00 00 00 00 00 00 00 0 0 0 3 0 0 0 4 4 > 00 00 00 00 00 00 1e 01
) 390: 00 00 00 00 00 00 00 0 0 0 3 0 0 0 4 4 > 00 00 00 00 00 00 1e 01
) 400: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 1e 01
) 410: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 0f 00 00 14 00 01
) 420: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 0f 00 00 14 00 01
) 430: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 0f 00 00 14 00 01
) 440: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 00 01
) 450: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 00 01
) 460: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 00 01
) 470: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 14 01
) 480: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 14 01
) 490: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 14 01
) vct: test vector file HEX_TEST end detected
) 500: 00 00 00 00 00 00 00 0 0 0 0 0 0 0 0 4 > 00 00 00 00 00 00 00 01
RUN_VECTORS
/* CONTROLLER TEST RESULTS */
control_test
) File HEX_TEST closed
) Running test vector Assembler.
) Created Ancillary file control_test.083.SMO & .SXR.
) trace running from control_test   Sat Jul 16 10:00:27 1988
```

```
)      E  R R R R R R R R R R R R R R R R PP
)      X  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 HH
)      T  M M M M M M M M M M M M M M M M AA
)      _  1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0 SS
)      A  5 4 3 2 1 0 _ _ _ _ _ _ _ _ _ _ EE
)      D  _ _ _ _ _ _ A A A A A A A A A A __
)      D  A A A A A A D D D D D D D D D D BA
)      R  D D D D D D D D D D D D D D D D
)      [  D D D D D D R R R R R R R R R
)      2  R R R R R R [ [ [ [ [ [ [ [ [ [
)      :  [ [ [ [ [ [ 4 4 4 4 4 4 4 4 4 4
)      0  4 4 4 4 4 4 : : : : : : : : : :
)      ]  : : : : : : 0 0 0 0 0 0 0 0 0 0
)         0 0 0 0 0 0 ] ] ] ] ] ] ] ] ] ]
)         ] ] ] ] ] ]
)      -  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
)      x  xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx bb
qckq
)    0: 0 > II II II II II II II II II II II II II II II II 01
)   10: 0 > II II II II II II II II II II II II II II II II 01
)   20: 0 > II II II 02 II II II 02 II II II 02 II II II 02 01
)   30: 0 > II II II 02 II II II 02 II II II 02 II II II 02 01
)   40: 0 > II II 02 02 II II 02 02 II II 02 02 II II 02 02 01
)   50: 0 > II II 02 02 II II 02 02 II II 02 02 II II 02 02 01
)   60: 0 > II 02 02 02 II 02 02 02 II 02 02 02 II 02 02 02 01
)   70: 2 > II 0a 0a 0a II 0a 0a 0a II 0a 0a 0a II 0a 0a 0a 01
```

```
)  80: 2 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 01
)  90: 2 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 0b 0b 0b 08 0b 08 01
) 100: 2 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 08 08 08 09 08 09 01
) 110: 2 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 09 09 09 08 09 08 01
) 120: 2 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 08 08 08 09 08 09 01
) 130: 2 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 09 09 09 08 09 08 01
) 140: 2 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 08 08 08 09 08 09 01
) 150: 0 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 01 01 01 00 01 00 01
) 160: 0 > 0a 0a 0a 0a 0a 0a 0a 0a 0a 0a 01 01 01 02 01 02 01
) 170: 0 > 0a 0a 0a 02 0a 0a 0a 02 0a 02 02 02 02 02 02 02 01
) 180: 0 > 0a 0a 0a 02 0a 0a 0a 02 0a 02 02 02 02 02 02 02 01
) 190: 0 > 0a 0a 02 02 0a 0a 02 02 02 02 02 02 02 02 02 02 01
) 200: 0 > 0a 0a 02 02 0a 0a 02 02 02 02 02 02 02 02 02 02 01
) 210: 0 > 0a 02 02 02 0a 02 02 02 02 02 02 02 02 02 02 02 01
) 220: 3 > 0a 0e 0e 0e 0a 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 01
) 230: 3 > 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 0e 01
) 240: 3 > 0f 0c 0f 0c 0f 0c 0f 0c 0f 0c 0f 0c 0f 0c 0f 0c 01
) 250: 3 > 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 01
) 260: 3 > 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 01
) 270: 3 > 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 01
) 280: 3 > 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 0d 0c 01
) 290: 0 > 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 01
) 300: 0 > 02 01 02 01 02 01 02 01 02 01 02 01 02 01 02 01 01
) 310: 0 > 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 01
) 320: 0 > 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 01
) 330: 0 > 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 01
```

```
) 340: 0 > 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 01

) 350: 0 > 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 02 01

) 360: 4 > 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 01

) 370: 4 > 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 01

) 380: 4 > 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 01

) 390: 4 > 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 01

) 400: 4 > 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 01

) 410: 4 > 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 01

) vct: test vector file control_test end detected

) 420: 4 > 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 01

RUN_VECTORS

/* RESULTS OF CONVOLUTION TEST */

test

) File control_test closed

) trace running from test   Sat Jul 16 10:02:19 1988

)      B B B B B A E   R  R  R  R  R  R  R  R  PP

)      3 2 1 0 0 3 X   0  0  0  0  E  E  E  E  HH

)      _ _ _ _ _ _ T   M  M  M  M  S  S  S  S  AA

)      I I I I I I _   3  2  1  0  _  _  _  _  SS

)      N N N N N N A   _  _  _  _  3  2  1  0  EE

)      2 2 2 2 1 2 D   A  A  A  A  [  [  [  [  __

)      [ [ [ [ [ [ D   D  D  D  D  7  7  7  7  BA

)      3 3 3 3 3 3 R   D  D  D  D  :  :  :  :

)      : : : : : : [   R  R  R  R  0  0  0  0

)      0 0 0 0 0 0 2   [  [  [  [  ]  ]  ]  ]

)      ] ] ] ] ] ] :   4  4  4  4
```

```
)                0   :  :  :  :
)                ]   0  0  0  0
)                    ]  ]  ]  ]
)      - - - - - - -  -- -- -- -- -- -- -- -- --
)      x x x x x x x  xx xx xx xx xx xx xx xx bb
qckq
)    0: 0 0 0 0 0 I 0 > II II II II II II II II 01
)   10: 0 0 0 0 0 I 0 > II II II II II II II II 01
)   20: 0 0 0 0 0 I 0 > II II II 02 II II II 00 01
)   30: 0 0 0 0 0 I 0 > II II II 02 II II II 00 01
)   40: 0 0 0 0 0 I 0 > II II 02 02 II II 00 00 01
)   50: 0 0 0 0 0 I 0 > II II 02 02 II II 00 00 01
)   60: 0 0 0 0 0 I 0 > II 02 02 02 II 00 00 00 01
)   70: 0 0 0 0 0 I 0 > II 02 02 02 II 00 00 00 01
)   80: 0 0 0 0 0 I 2 > 0a 0a 0a 0a 00 00 00 00 01
)   90: 0 0 0 0 0 I 2 > 0a 0a 0a 0a 00 00 00 00 01
)  100: 0 0 0 0 0 f 2 > 0b 08 0b 08 00 00 00 00 01
)  110: 0 0 0 0 0 0 2 > 08 09 08 09 00 00 00 00 01
)  120: 0 0 0 0 0 a 2 > 09 08 09 08 00 00 00 00 01
)  130: 0 0 0 0 8 0 2 > 08 09 08 09 00 00 00 00 01
)  140: 0 0 0 0 0 5 2 > 09 08 09 08 00 00 00 78 01
)  150: 0 0 0 0 6 0 2 > 08 09 08 09 00 00 50 78 01
```

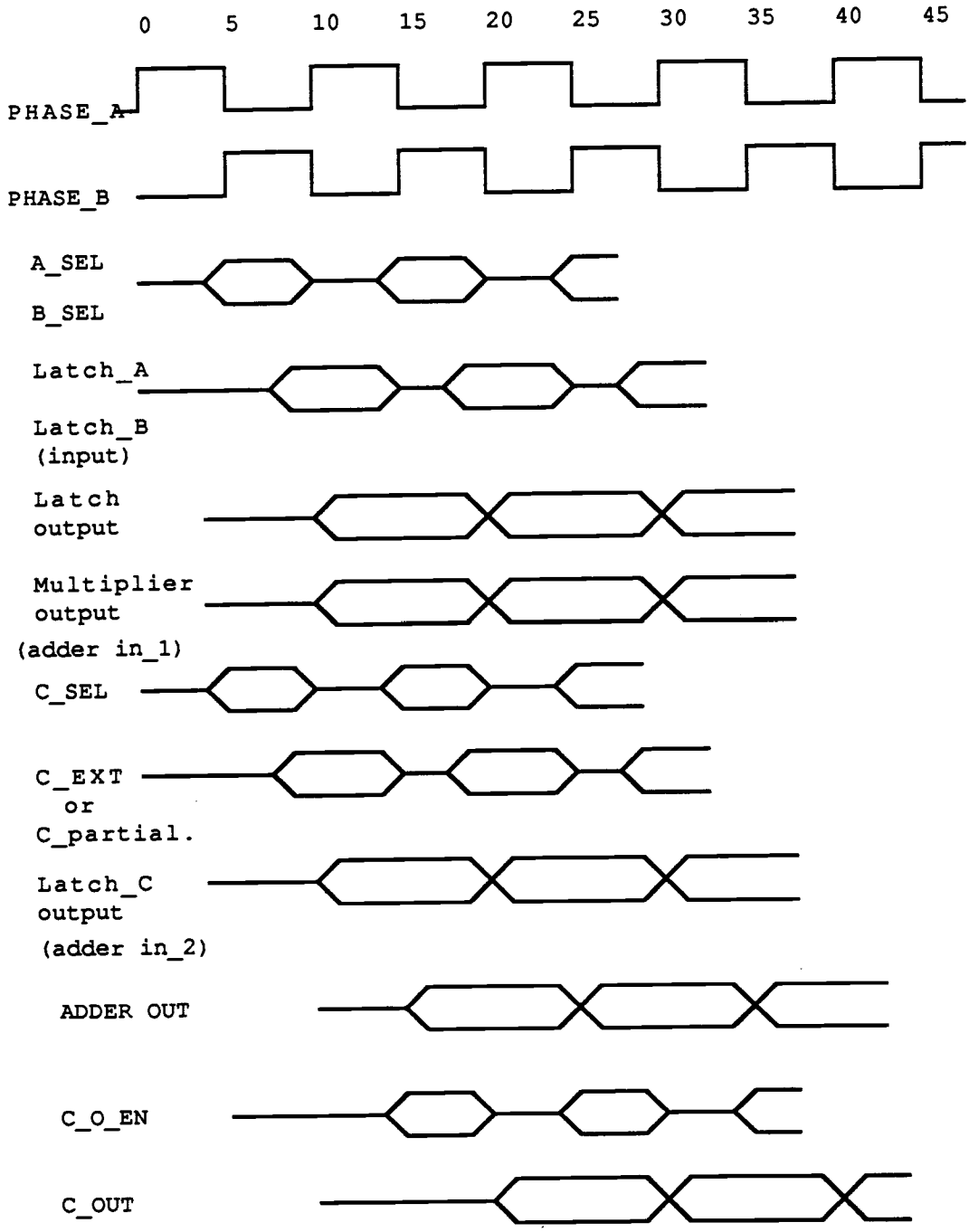


FIG 10. TIMING DIAGRAM OF IPE

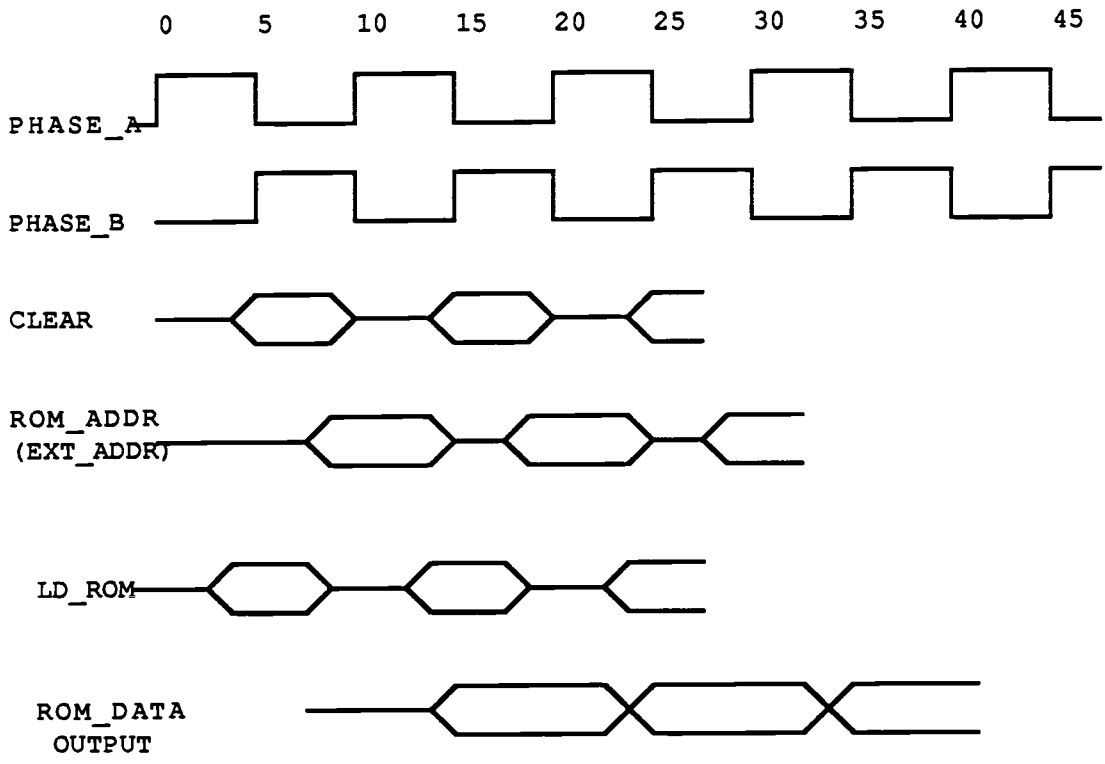
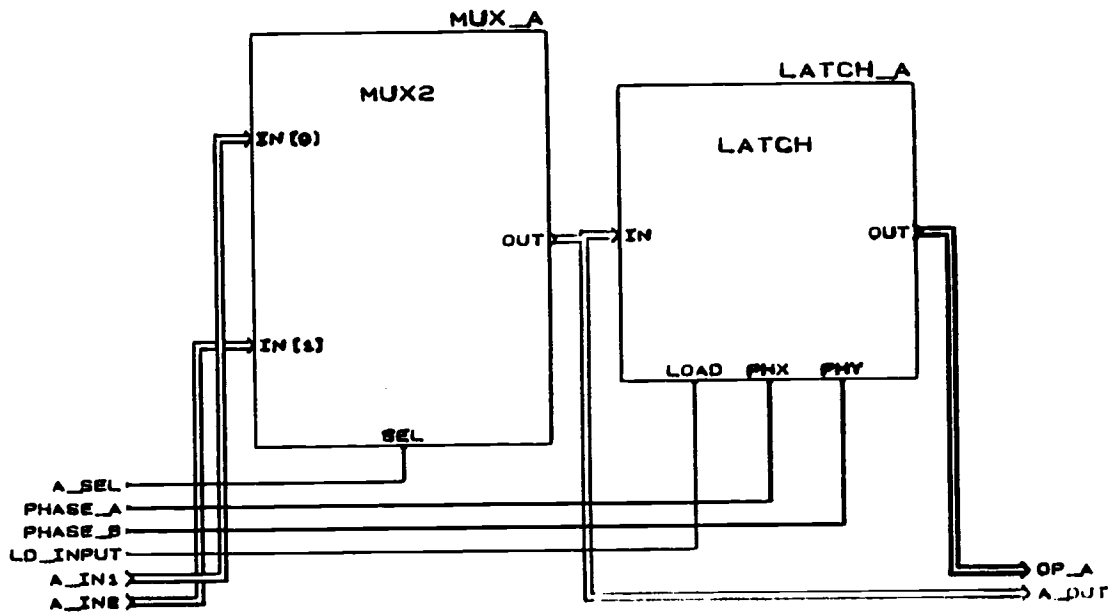
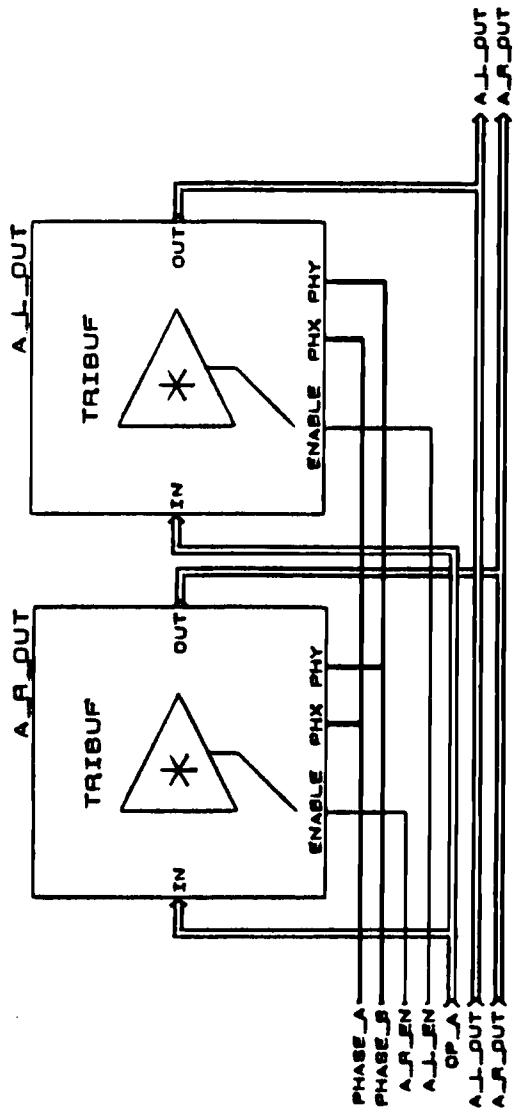


FIG 11. TIMING DIAGRAM FOR ROM

FIG 12. INPUT SELECTOR



Silicon Compiler Systems	Object. INPUT_SELECTOR	User. team5	Date. Jul 14 88 14.20



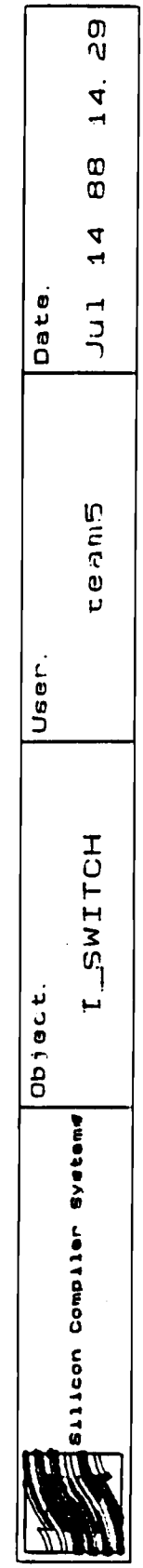


FIG 13. INTERCONNECT SWITCH

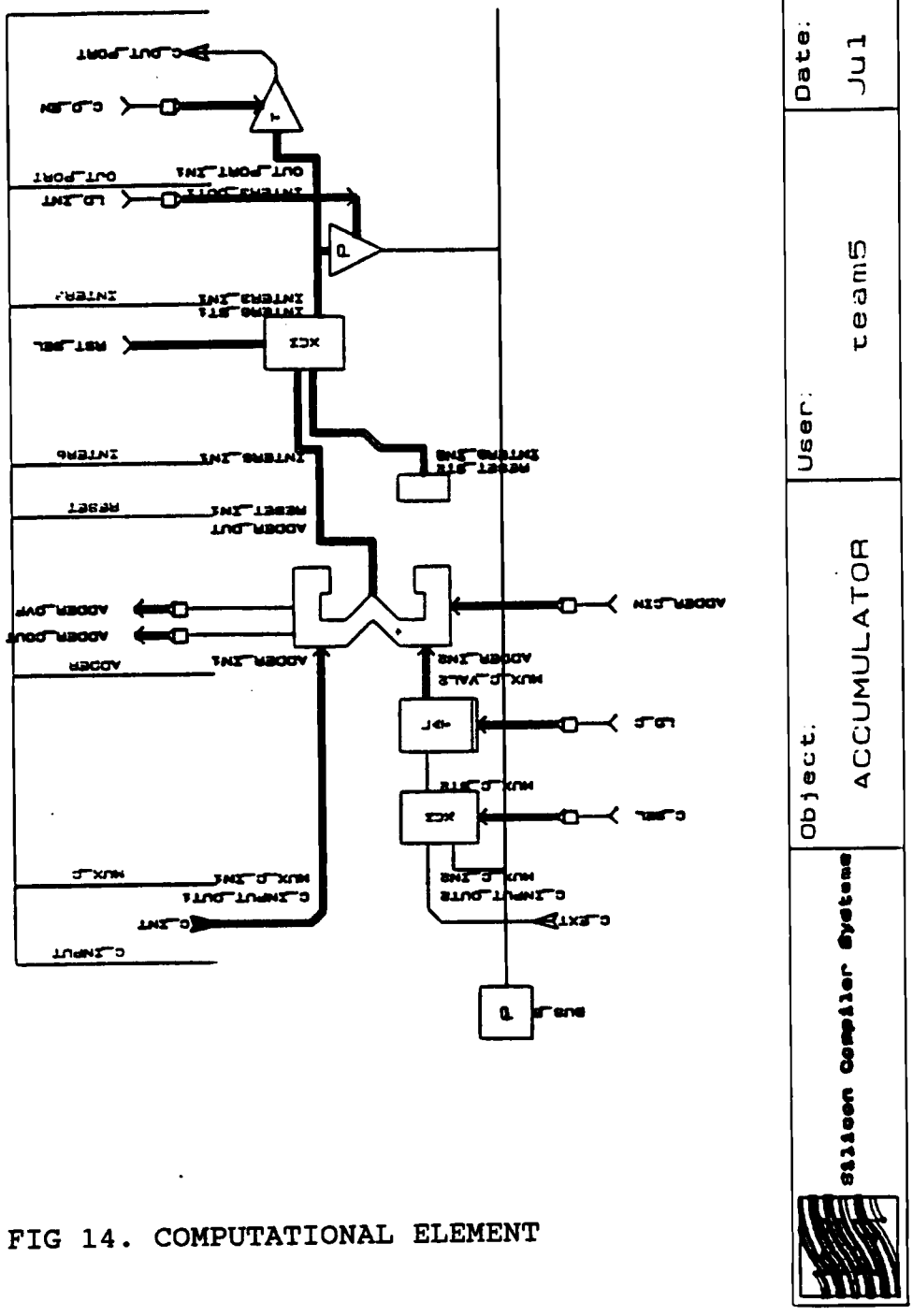


FIG 14. COMPUTATIONAL ELEMENT

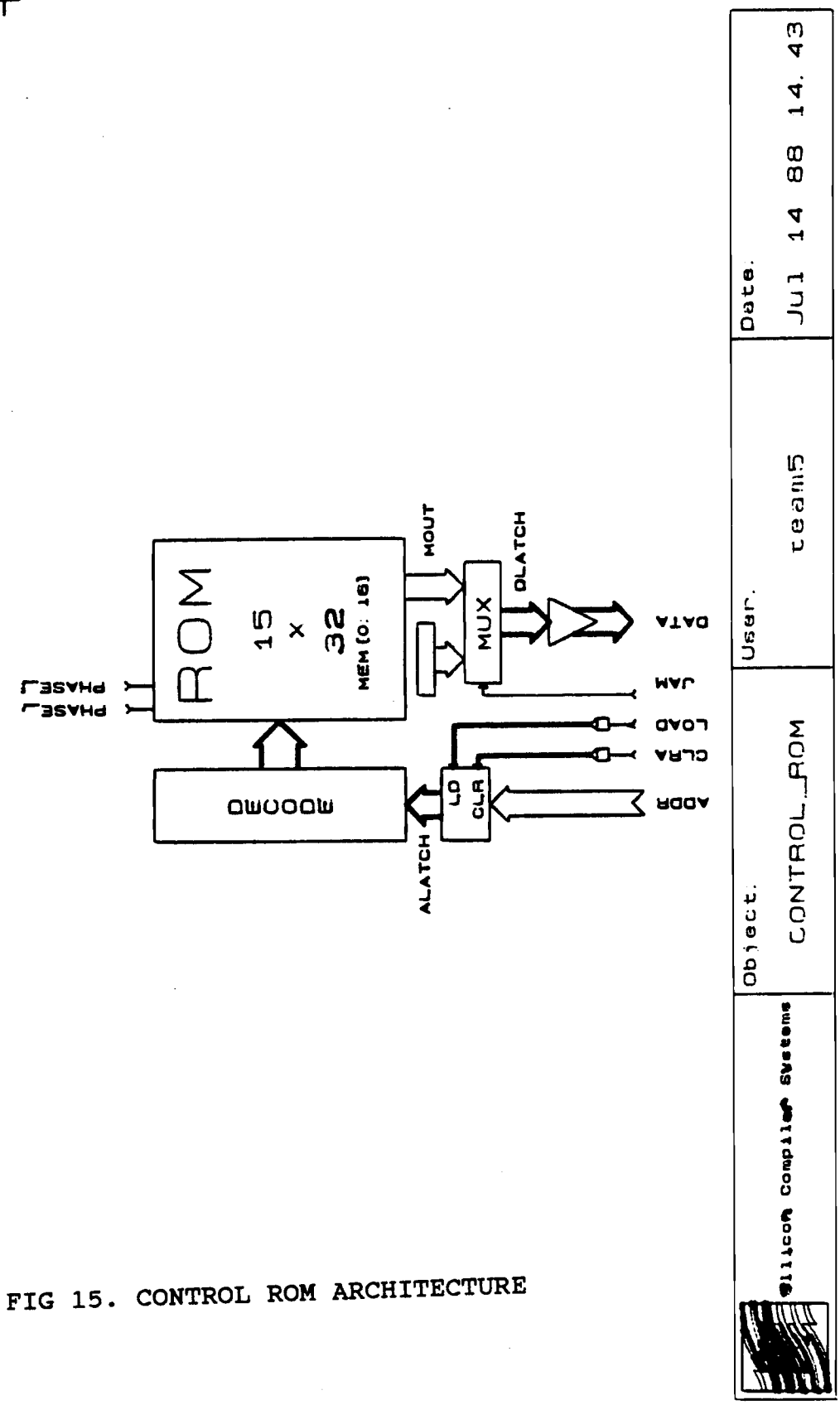


FIG 15. CONTROL ROM ARCHITECTURE